CLICK ON A PATENT BELOW TO GO DIRECTLY TO THAT PATENT.

	PAT. NO.		Title
1	5,502,826	T	System and method for obtaining parallel existing instructions in a particular data processing configuration by compounding instructions
2	5,500,942	T	Method of indicating parallel execution compoundability of scalar instructions based on analysis of presumed instructions
3	5,448,746	T	System for comounding instructions in a byte stream prior to fetching and identifying the instructions for execution
4	5,446,850	T	Cross-cache-line compounding algorithm for scism processors
5	5,398,321	T	Microcode generation for a scalable compound instruction set machine
6	5,386,531	T	Computer system accelerator for multi-word cross-boundary storage access
7	5,355,460	T	In-memory preprocessor for compounding a sequence of instructions for parallel computer system execution
8	5,303,356	T	System for issuing instructions for parallel execution subsequent to branch into a group of member instructions with compoundability in dictation tag
9	5,295,249	T	Compounding preprocessor for cache for identifying multiple instructions which may be executed in parallel
10	5,287,467	T	Pipeline for removing and concurrently executing two or more branch instructions in synchronization with other instructions executing in the execution unit
11	5,214,763	T	Digital computer system capable of processing two or more instructions in parallel and having a coche and instruction compounding mechanism

12 5,197,135 T Memory management for scalable compound instruction set machines with in-memory compounding

13 5,051,940 T Data dependency collapsing hardware apparatus

United States Patent — ***5,502,826***

Vassiliadis , et al. — **March 26, 1996**

System and method for obtaining parallel existing instructions in a particular data processing configuration by compounding instructions

Abstract

Scalable compound instruction set machine and method which provides for processing a set of instructions or program to be executed by a computer to determine statically which instructions may be combined into compound instructions which are executed in parallel by a scalar machine. Such processing looks for classes of instructions that can be executed in parallel without data-dependent or hardware-dependent interlocks. Without regard to their original sequence the individual instructions are combined with one or more other individual instructions to form a compound instruction which eliminates interlocks. Control information is appended to identify information relevant to the execution of the compound instructions. The result is a stream of scalar instructions compounded or grouped together before instruction decode time so that they are already flagged and identified for selective simultaneous parallel execution by execution units. The compounding does not change the object code results and existing programs realize performance improvements while maintaining compatibility with previously implemented systems for which the original set of instructions was provided.

Inventors:	**Vassiliadis; Stamatis** (Vestal, NY); **Blaner; Bartholomew** (Newark Valley, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**186221**
Filed:	**January 25, 1994**

Current U.S. Class:	**712/213**; 712/23; 712/24
Intern'l Class:	G06F 009/44
Field of Search:	395/375,700,500,800

References Cited [Referenced By]

U.S. Patent Documents

3293616	Dec., 1966	Mullery et al.	395/375.
3343135	Sep., 1967	Frieman et al.	395/700.
3346851	Oct., 1967	Thornton et al.	364/704.
3611306	Oct., 1971	Reigel	395/65.
4295193	Oct., 1981	Pomerene.	
4439828	Mar., 1984	Martin	364/200.
4574348	Mar., 1986	Scallon.	
4594655	Jun., 1986	Hao et al.	
4780820	Oct., 1988	Sowa.	
4807115	Feb., 1989	Torug.	
4847755	Jul., 1989	Morrison et al.	
4858105	Aug., 1989	Kuriyama et al.	
4942525	Jul., 1990	Shiatani et al.	
5133077	Jul., 1992	Karne et al.	395/800.

Foreign Patent Documents

0052194	May., 1982	EP.
0449661	Oct., 1991	EP.
61-245239	Oct., 1986	JP.

Other References

Acosta, R. D., et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al., the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al., ed (McGraw-Hill, 1982, pp. 276-292.
Capozzi, A. J., et al., "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al., "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al., "SIMP (Single Instruction Stream/Multiple Instruction Pipelining); A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Syposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer,

Jul., 1989, pp. 21-35.
Smith, M. D., et al., "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al. ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al., "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, n. P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp.63-76.
Colwell, R. P., et al., "A VLIW Architecture for a Trace Scheduling Complier", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the CRISP Instruction Set Architecture", Proceedings of COMPCON, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al., "Compiling for the CRISP Microprocessor", Proceedings of COMPCON, Spring, 1987, pp. 96-100.
Hennessy, J., et al., "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communicatioins of the ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "the 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al., "Branch Folding in the CRISP Microprocessor".
"Reducing Branch Delay to Zero", Proceedings of COMPCON, Spring 1987, pp. 2-9.
Hwu, W. W., et al., "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al., "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computers, vol. 17, No. 1, Jan. 1984, pp. 6-22.
Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al., Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems , vol. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al. "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al., "A Study of Instructions Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture,

Jun., 1983, pp. 132-137.
Vassiliadis, S., et al., "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp.4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
R. J. Eberhard, IBM Technical Disclosure Bulletin, "Early Release of a Processor Following Address Translation Prior to Page Access Checking", vol. 33, No. 10A, Mar. 1991.
Higbee, "Overlapped Operation with Microprogramming", IEEE Transactions on Computers, vol. C-27, No. 3, pp. 270-275.
Wang, "Distributed Instruction Set Computer", Proceedings of the International Conference on Parallel Processing, Aug. 1988, vol. 1, pp. 426-429.

Primary Examiner: Kim; Ken S.
Attorney, Agent or Firm: Augspurger; Lynn L., Marhoefer; Laurence J.

Parent Case Text

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of application Ser. No. 08/013,982 filed Feb. 5, 1993, now abandoned, which is a continuation of application Ser No. 07/519,384, filed on May 4, 1990, now abandoned.

Claims

We claim:

1. A computer implemented method for processing a sequence of binary encoded scalar instructions, each instruction including an operation code, prior to fetching for execution in a data processing system, which places constraints on parallel instruction execution, including the steps of:

transferring said sequence of binary encoded scalar instructions from a data storage unit to an instruction compounding unit;

assigning each operation code in said sequence of instructions to one of a plurality categories based on a function performed by said data processing system in response to said operation code, the number of said categories being less than the number of operation codes in said sequence of instructions;

storing in said data processing system matrix data that encodes which instruction pairs in said sequence of instructions are compoundable based on the category assigned to the operation code of each adjacent instruction of said instruction pairs;

processing groups of instructions in said compounding unit to generate a compounding instruction for instruction pairs by comparing to said matrix data the category assigned in said assigning step respectively to each instruction of the instruction pairs in said group of instructions.

2. The method of claim 1 wherein said processing step is performed prior to the time the existing instructions are fetched for execution.

3. The method of claim 1 wherein said processing step retains the object code of the existing instructions in its original form for execution singly or for execution in parallel with another instruction when a fetched instruction has a compounding indicator that its associated instruction can be executed in parallel with an adjacent intruction of a fetched instruction stream.

4. The method of claim 1 wherein said processing step takes into account data dependent interlocks between instructions as well as the existence of related interlock collapsing functional units in the particular configuration of the data processing system.

5. The method of claim 1 wherein said processing step takes into account hardware dependent interlocks between instructions as well as the existence of related interlock collapsing functional units in the particular configuration of a data processing system.

6. The method of claim 1 wherein said processing step compares a first existing instruction with a second adjacent following instruction for possible compounding with each other, and then compares the second instruction with a third adjacent following intruction for possible compounding with each other, to identify multiple compound instructions identified by multiple compounding indicators, respectively.

7. The method of claim 6 further including the additional step of determining an optimum sequence of multiple compound instructions for a given portion of an instruction stream.

8. The computer implemented method as in claim 1 wherein said compounding instruction is associated with each instruction of the instruction stream which can be grouped as adjacent instructions which can be executed in parallel in a particular configuration of a data processing system has a value n which indicates the number of instructions following the instruction in the instruction stream which can be compounded with the instruction upon execution of the instruction when the original instruction is fetched with the associated compounding indicator for execution in a particular computer configuration.

Description

RELATED APPLICATIONS

The following related applications are commonly owned by the same assignee and are incorporated by reference herein: "Data Dependency Collapsing Hardware Apparatus" filed Apr. 4, 1990, Ser. No.

07/504,910, now U.S. Pat. No. 5,051,940, issued Sep. 24, 1991 and "General Purpose Compounding Technique For Instruction-Level Parallel Processors" filed May 4th, 1990, Ser. No. 07/519,382, now abandoned.

FIELD OF THE INVENTION

This invention relates generally to parallel processing by computer, and more particularly relates to processing an instruction stream to identify those instructions which can be issued and executed in parallel in a specific computer system configuration.

BACKGROUND OF THE INVENTION

The concept of parallel execution of instructions has helped to increase the performance of computer systems. Parallel execution is based on having separate functional units which can execute two or more of the same or different instructions simultaneously.

Another technique used to increase the performance of computer systems is pipelining. In general, pipelining is achieved by partitioning a function to be performed by a computer into independent subfunctions and allocating a separate piece of hardware, or stage, to perform each subfunction. Each stage is defined to occupy one basic machine cycle in time. Pipelining does provide a form of parallel processing since it is possible to execute multiple instructions concurrently. Ideally, one new instruction can be fed into the pipeline per cycle, with each instruction in the pipeline being in a different stage of execution. The operation is analogous to a manufacturing assembly line, with a number of instances of the manufactured product in varying stages of completion.

However, many times the benefits of parallel execution and/or pipelining are not achieved because of delays like those caused by data dependent interlocks and hardware dependent interlocks. An example of a data dependent interlock is a so-called write-read interlock where a first instruction must write its result before the second instruction can read and subsequently use it. An example of hardware dependent interlock is where a first instruction must use a particular hardware component and a second instruction must also use the same particular hardware component.

One of the techniques previously employed to avoid interlocks (sometimes called pipeline hazards) is called dynamic scheduling. Dynamic scheduling is based on the fact that with the inclusion of specialized hardware, it is possible to reorder instruction sequences after they have been issued into the pipeline for execution.

There have also been some attempts to improve performance through so-called static scheduling which is done before the instruction stream is fetched from storage for execution. Static scheduling is achieved by moving code and thereby reordering the instruction sequence before execution. This reordering produces an equivalent instruction stream that will more fully utilize the hardware through parallel processing. Such static scheduling is typically done at compile time. However, the reordered instructions remain in their original form and conventional parallel processing still requires some form of dynamic determination just prior to execution of the instructions in order to decide whether to execute the next two instructions serially or in parallel.

Such scheduling techniques can improve the overall performance of a pipelined computer, but cannot alone satisfy the ever present demands for increased performance. In that regard, many of the recent proposals for general purpose computing are related to the exploitation of parallelism at the instruction level beyond that attained by pipelining. For example, further instruction level parallelism has been achieved explicitly by issuing multiple instructions per cycle with so-called superscalar machines, rather than implicitly as with dynamic scheduling of single instructions or with vector machines. The name superscalar for machines that issue multiple instructions per cycle is to differentiate them from scalar machines that issue one instruction per cycle.

In a typical superscalar machine, the opcodes in a fetched instruction stream are decoded and analyzed dynamically by instruction issue logic in order to determine whether the instructions can be executed in parallel. The criteria for such last-minute dynamic scheduling are unique to each instruction set architecture, as well for the underlying implementation of that architecture in any given instruction processing unit. Its effectiveness is therefore limited by the complexity of the logic to determine which combinations of instructions can be executed in parallel, and the cycle time of the instruction processing unit is likely to be increased. The increased hardware and cycle time for such superscalar machines become even a bigger problem in architectures which have hundreds of different instructions.

There are other deficiencies with dynamic scheduling, static scheduling, or combinations thereof. For example, it is necessary to review each scalar instruction anew every time it is fetched for execution to determine its capability for parallel execution. There has been no way provided to identify and flag ahead of time those scalar instructions which have parallel execution capabilities.

Another deficiency with dynamic scheduling of the type implemented in super scalar machines is the manner in which scalar instructions are checked for possible parallel processing. Superscalar machines check scalar instructions based on their opcode descriptions, and no way is provided to take into account hardware utilization. Also, instructions are issued in FIFO fashion thereby eliminating the possibility of selective grouping to avoid or minimize the occurrence of interlocks.

There are some existing techniques which do seek to consider the hardware requirements for parallel instruction processing. One such system is a form of static scheduling called the Very Long Instruction Word machine in which a sophisticated compiler rearranges instructions so that hardware instruction scheduling is simplified. In this approach the compiler must be more complex than standard compilers so that a bigger window can be used for purposes of finding more parallelism in an instruction stream. But the resulting instructions may not necessarily be object code compatible with the pre-existing architecture, thereby solving one problem while creating additional new problems. Also, substantial additional problems arise due to frequent branching which limits its parallelism.

Therefore, none of these prior art approaches to parallel processing have been sufficiently comprehensive to minimize all possible interlocks, while at the same time avoiding major redesign of the architected instruction set and avoiding complex logic circuits for dynamic decoding of fetched instructions.

Accordingly, what is needed is an improvement in digital data processing which facilitates the execution of existing machine instructions in parallel in order to increase processor performance. Since the number of instructions executed per second is a product of the basic cycle time of the processor and the average number of cycles required per instruction completion, what is needed is a solution which takes both of these parameters under consideration. More specifically, a mechanism is needed that reduces the number of cycles required for the execution of an instruction for a given architecture. In addition, an improvement is needed which reduces the complexity of the hardware necessary to support parallel instruction execution, thus minimizing any possible increase in cycle time. Additionally, it would be highly desirable for the proposed improvement to provide compatibility of the implementation with an already defined system architecture while introducing parallelism at the instruction level of both new and existing machine code.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is an object of the present invention to provide a method for statically analyzing, at a time prior to instruction decode and execution, a sequence of existing instructions to generate compound instructions formed by adjacent grouping of existing instructions capable of parallel execution. A related object is to add relevant control information to the instruction stream including grouping information indicating where a compound instruction starts as well as indicating the number of existing instructions which are incorporated into each compound instruction.

Yet another object is to analyze a large window of an instruction byte stream prior to instruction fetch, with the window being adjustable to different positions in the instruction byte stream in order to achieve optimum selective grouping of individual adjacent instructions which form a compound instruction.

A further object is to provide an instruction compounding method with the aforementioned characteristics which is applicable to complex instruction architectures having variable length instructions and having data intermixed with instructions, and which is also applicable to RISC architectures wherein instructions are usually a constant length and wherein data is not mixed with instructions.

An additional object is to provide a method of pre-processing an instruction stream to create compound instructions, wherein the method can be implemented by software and/or hardware at various points in the computer system prior to instruction decoding and execution. A related object is to provide a method of pre-processing existing instructions which operates on a binary instruction stream as part of a post-compiler, or as part of an in-memory compounder, or as part of cache instruction compounding unit, and which can start compounding instructions at the beginning of a byte stream without knowing the boundaries of the instructions.

Thus, the invention contemplates a method of pre-processing an instruction stream to create compound instructions composed of scalar instructions which have still retained their original contents. Compound instructions are created without changing the object code of the scalar instructions which form the compound instruction, thereby allowing existing programs to realize a performance improvement on a compound instruction machine while maintaining compatibility with previously implemented scalar instruction machines.

More specifically, the invention provides a set of compounding rules based on an analysis of existing instructions to separate them into different classes. The analysis determines which instructions qualify, either with instructions in their own class or with instructions in other classes, for parallel execution in a particular hardware configuration. Such compounding rules are used as a standard for pre-processing an instruction stream in order to look for groups of two or more adjacent scalar instructions that can be executed in parallel. In some instances certain types of interlocked instructions can be compounded for parallel execution where the interlocks are collapsible in a particular hardware configuration. In other configurations where the interlocks are non-collapsible, the instructions having data dependent or hardware dependent interlocks are excluded from groups forming compound instructions.

Each compound instruction is identified by control information such as tags associated with the compound instruction, and the length of a compound instruction is scalable over a range beginning with a set of two scalar instructions up to whatever maximum number that can be executed in parallel by the specific hardware implementation. Since the compounding rules are based on an identification of classes of instructions rather than on individual instruction, complex matrices showing all possible combinations of specific individual instructions are no longer needed. While keeping their original sequence intact, individual instructions are selectively grouped and combined with one or more other adjacent scalar instructions to form a compound instruction which contains scalar instructions which still have object code compatibility with non-compound scalar instructions. Control information is appended to identify information relevant to the execution of the compound instructions.

These and other objects, features and advantages of the invention will be apparent to those skilled in the art in view of the following detailed description and accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a high level schematic diagram of the invention;

FIG. 2 is a timing diagram for a uniprocessor implementation showing the parallel execution of certain non-interlocked instructions which have been selectively grouped in a compound instruction stream;

FIG. 3 is a timing diagram for a multiprocessor implementation showing the parallel execution of scalar and compound instructions which are not interlocked;

FIG. 4 comprises FIGS. 4A and 4B which together illustrates an example of a possible selective categorization of a portion of the instructions executed by an existing scalar machine;

FIG. 5 shows a typical path taken by a program from source code to actual execution;

FIG. 6 is a flow diagram showing generation of a compound instruction set program from an assembly language program;

FIG. 7 is a flow diagram showing execution of a compound instruction set program;

FIG. 8 is an analytical chart for instruction stream texts with identifiable instruction reference points;

FIG. 9 is an analytical chart for an instruction stream text with variable length instructions without a reference point, showing their related sets of possible compound identifier bits;

FIG. 10 illustrates a logical implementation of an instruction compound facility for handling the instruction stream text of FIG. 9;

FIG. 11 is a flow diagram for compounding an instruction stream having reference tags to identify instruction boundary reference points;

FIG. 12 shows an exemplary compound instruction control field;

FIG. 13 is a flow chart for developing and using compounding rules applicable to a specific computer system hardware configuration and its particular architected instruction set;

FIG. 14 shows how different groupings of valid non-interlocked pairs of instructions form multiple compound instructions for sequential or branch target execution;

FIG. 15 shows how different groupings of valid non-interlocked triplets of instructions form multiple compound instructions for sequential or branch target execution;

FIG. 16 comprises FIGS. 16A and 16B which together is a flow chart for compounding an instruction stream like the one shown in FIG. 9 which includes variable length instructions without boundary reference points; and

FIG. 17 is a chart showing typical compoundable pairs of instruction categories for the portion of the System/370 instruction set shown in FIG. 4.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

The essence of the present invention is the pre-processing of a set of instructions or program to be executed by a computer, to determine statically which non-interlocked instructions may be combined into compound instructions, and the appending of control information to identify such compound instructions. Such determination is based on compounding rules which are developed for an instruction set of a particular architecture. Existing scalar instructions are categorized based on an analysis of their operands, hardware utilization and function, so that grouping of instructions by compounding to avoid non-collapsible interlocks is based on instruction category comparison rather than specific instruction comparison.

As shown in the various drawings and described in more detail hereinafter, this invention called a Scalable Compound Instruction Set Machine (SCISM) provides for a stream of scalar instructions to be compounded or grouped together before instruction decode time so that they are already flagged and identified for selective simultaneous parallel execution by appropriate instruction execution units. Since such compounding does not change the object code, existing programs can realize a performance improvement while maintaining compatibility with previously implemented systems.

As generally shown in FIG. 1, an instruction compounding unit 20 takes a stream of binary scalar instructions 21 (with or without data included therein) and selectively groups some of the adjacent scalar instructions to form encoded compound instructions. A resulting compounded instruction stream 22 therefore combines scalar instructions not capable of parallel execution and compound instructions formed by groups of scalar instructions which are capable of parallel execution. When a scalar instruction is presented to an instruction processing unit 24, it is routed to the appropriate functional unit for serial execution. When a compound instruction is presented to the instruction processing unit 24, its scalar components are each routed to their appropriate functional unit or interlock collapsing unit for simultaneous parallel execution. Typical functional units include but are not limited to an arithmetic and logic unit (ALU) 26, 28, a floating point arithmetic unit (FP) 30, and a store address generation unit (AU) 32. An exemplary data dependency collapsing unit is disclosed in co-pending application Ser. No. 07/504,910, entitled "Data Dependency Collapsing Hardware Apparatus" filed Apr. 4, 1990, now issued U.S. Pat. No. 5,051,940.

It is to be understood that the technique of the invention is intended to facilitate the parallel issue and execution of instructions in all computer architectures that process multiple instructions per cycle (although certain instructions may require more than one cycle to be executed)

As shown in FIG. 2, the invention can be implemented in a uniprocessor environment where each functional execution unit executes a scalar instruction (S) or alternatively a compounded scalar instruction (CS). As shown in the drawing, an instruction stream 33 containing a sequence of scalar and compounded scalar instructions has control tags (T) associated with each compound instruction. Thus, a first scalar instruction 34 could be executed singly by functional unit A in cycle 1; a triplet compound instruction 36 identified by tag T3 could have its three compounded scalar instructions executed in parallel by functional units A, C and D in cycle 2; another compound instruction 38 identified by tag T2 could have its pair of compounded scalar instructions executed in parallel by functional units A and B in cycle 3; a second scalar instruction 40 could be executed singly by functional unit C in cycle 4; a large group compound instruction 42 could have its four compounded scalar instructions executed in parallel by functional units A-D in cycle 5; and a third scalar instruction 44 could be executed singly by functional unit A in cycle 6.

It is important to realize that multiple compound instructions are capable of parallel execution in certain computer system configurations. For example, the invention could be potentially implemented in a multiprocessor environment as shown in FIG. 3 where a compound instruction is treated as a unit for parallel processing by one of the CPUs (central processing units). As shown in the drawing, the same instruction stream 33 could be processed in only two cycles as follows. In a first cycle, a CPU #1 executes the first scalar instruction 34; the functional units of a CPU #2 execute triplet compound instruction 36; and the functional units of a CPU #3 execute the two compounded scalar instructions in compound instruction 38. In a second cycle, the CPU #1 executes the second scalar instruction 40; the functional units of CPU #2 execute the four compounded scalar instructions in compound instruction 42; and a functional unit of CPU #3 executes the third scalar instruction 44.

One example of a computer architecture which can be adapted for handling compound instructions is an IBM System/370 instruction level architecture in which multiple scalar instructions can be issued for execution in each machine cycle. In this context a machine cycle refers to all the pipeline steps or stages required to execute a scalar instruction. A scalar instruction operates on operands representing single-valued parameters. When an instruction stream is compounded, adjacent scalar instructions are selectively grouped for the purpose of concurrent or parallel execution.

The instruction sets for various IBM System/370 architectures such as System/370, the System/370 extended architecture (370-XA), and the System/370 Enterprise Systems Architecture (370-ESA) are well known. In that regard, reference is given here to the Principles of Operation of the IBM System/370 (publication #GA22-7000-10 1987), and to the Principles of Operation, IBM Enterprise Systems Architecture/370 (publication #SA22-7200-0 1988).

In general, an instruction compounding facility will look for classes of instruction that may be executed in parallel, and ensure that no interlocks between members of a compound instruction exist that cannot be handled by the hardware. When compatible sequences of instructions are found, a compound instruction is created.

More specifically, the System/370 instruction set can be broken into categories of instructions that may be executed in parallel in a particular computer system configuration. Instructions with certain of these categories may be combined or compounded with instructions in the same category or with instructions in certain other categories to form a compound instruction. For example, a portion of the System/370 instruction set can be partitioned into the categories illustrated in FIG. 4. The rationale for this categorization is based on the functional requirements of the System/370 instructions and their hardware utilization in a typical computer system configuration. The rest of the System/370 instructions are not considered specifically for compounding in this exemplary embodiment. This does not preclude them from being compounded by the methods and technique of the present invention disclosed herein. It is noted that the hardware structures required for compound instruction execution can be readily controlled by horizontal microcode, allowing for exploitation of parallelism in the remaining instructions not considered for compounding and not included in the categories of FIG. 4, thereby increasing performance.

One of the most common sequences in System/370 programs is to execute an instruction of the TM or RX-format COMPARES (C, CH, CL, CLI, CLM), the result of which is used to control the execution of a BRANCH-on-condition type instruction (BC, BCR) which immediately follows. Performance can be improved by executing the COMPARE and the BRANCH instructions in parallel, and this has sometimes been done dynamically in high performance instruction processors. Some difficulty lies in quickly identifying all the various members of the COMPARE class of instructions and all the members of the BRANCH class of instructions in a typical architecture during the instruction decoding process. This is one reason why the superscalar machines usually consider only a small number of specific scalar instructions for possible parallel processing. In contrast, such limited dynamic scheduling based only on a last-minute comparison of two specific instructions is avoided by the invention, because the analysis of all the members of the classes are accomplished ahead of time in order to develop adequate compounding rules for creating a compound instruction which is sure to work.

The enormous problem that arises from dynamic scheduling of individual instructions after fetch is shown by realizing that two-way compounding of the fifty seven individual instructions in FIG. 4 produces a 57.times.57 matrix of more than three thousand possible combinations. This is in sharp contrast to the 10.times.10 matrix of FIG. 17 for the same number of instructions considered from the point of view of possible category combinations, as provided by the present invention.

Many classes of instructions may be executed in parallel, depending on how the hardware is designed. In addition to the COMPARE and BRANCH compoundable pairs described above, many other compoundable combinations capable of parallel execution are possible (See FIG. 17), such as LOADS (category 7) compounded with RR-format instructions (category 1), BRANCHS (categories 3-5) compounded with LOAD ADDRESS (category 8), and the like.

In some instances the sequence order will affect the parallel execution capabilities and therefore determine whether two adjacent instructions can be compounded. In that regard, the row headings 45 identify the category of the first instruction in a byte stream and the column headings 47 identify the category of the next instruction which follows the first instruction. For example, BRANCHES (categories 3-5) followed by certain SHIFTS (category 2) are always compoundable 49, while SHIFTS (category 2) followed by BRANCHES (categories 3-5) are only "sometimes" compoundable 51.

The "sometimes" status identified as "S" in the chart of FIG. 17 can often be changed to "always" identified as "A" in the chart by adding additional functional hardware units to the computer system configuration. For example, consider a configuration which supports two-way compounding and which has no add-shift collapsing unit, but instead has a conventional ALU and a separate shifter. In other words, there is no interlock collapsing hardware for handling interlocked ADD and SHIFT instructions. Consider the following instruction sequence:

AR R1,R2

SRL R3 by D2

It is clear that this pair of instructions is compoundable for parallel execution. But in some instances it would not be compoundable due to a non-collapsible interlock, as shown in the following instruction sequence:

AR R1,R2

SRL R1 by D2

So the chart shows that a category 1 instruction (AR) followed by a category 2 instruction (SRL) is sometimes compoundable 53. By including an ALU which collapses certain interlocks such as the add/shift interlock shown above, the S could become an A in the chart of FIG. 17. Accordingly, the compounding rules must be updated to reflect any changes which are made in the particular computer system configuration.

As an additional example, consider the instructions contained in category 1 compounded with instructions from that same category in the following instruction sequence:

AR R1,R2

SR R3,R4

This sequence is free of data hazard interlocks and produces the following results which comprise two independent System/370 instructions:

R1=R1+R2

R3=R3-R4

Executing such a sequence would require two independent and parallel two-to-one ALU's designed to the instruction level architecture. Thus, it will be understood that these two instructions can be grouped to form a compound instruction in a computer system configuration which has two such ALU's. This example of compounding scalar instructions can be generalized to all instruction sequence pairs that are free of data dependent interlocks and also of hardware dependent interlocks.

In any actual instruction processor, there will be an upper limit to the number of individual instructions that can comprise a compound instruction. This upper limit must be specifically incorporated into the hardware and/or software unit which is creating the compound instructions, so that compound instructions will not contain more individual instructions (e.g., pair group, triplet group, group of four) than the maximum capability of the underlying execution hardware. This upper limit is strictly a consequence of the hardware implementation in a particular computer system configuration--it does not restrict either the total number of instructions that may be considered as candidates for compounding or the length of the group window in a given code sequence that may be analyzed for compounding.

In general, the greater the length of a group window being analyzed for compounding, the greater the parallelism that can be achieved due to more advantageous compounding combinations. In this regard, consider the sequence of instructions in the following Table 1:

TABLE 1

```
X1                 ;any compoundable instruction
X2                 ;any compoundable instruction
LOAD     R1,(X)    ;load R1 from memory location X
ADD      R3,R1     ;R3 = R3 + R1
SUB      R1,R2     ;R1 = R1 - R2
COMP     R1,R3     ;compare R1 with R3
X3                 ;any compoundable instruction
X4                 ;any compoundable instruction
```

If the hardware imposed upper limit on compounding is two (at most, two instructions can be executed in parallel in the same cycle), then there are a number of ways to compound this sequence of instructions depending on the scope of the compounding software.

If the scope of compounding were equal to four, then the compounding software would consider together (X1, X2, LOAD, ADD) and then slide forward one instruction at a time to consider together (X2, LOAD, ADD, SUB) and (LOAD, ADD, SUB, COMP) and (ADD, SUB, COMP, X3) and (SUB, COMP, X3, X4), thereby producing the following optimum pairings as candidates for a compound instruction:

--X1! X2 LOAD! ADD SUB! COMP X3! X4 --!

This optimum pairing provided by the invention completely relieves the interlocks between the LOAD and ADD and between the SUB and COMP, and provides the additional possibilities of X1 being compounded with its preceding instruction and of X4 being compounded with its following instruction.

On the other hand, a superscalar machine which pairs instructions dynamically in its instruction issue logic on strictly a FIFO basis, would produce only the following pairings as candidates for parallel execution:

X1 X2! LOAD ADD! SUB COMP! X3 X4!

This inflexible pairing incurs the full penalty of certain interlocking instructions, and only partial benefits of parallel processing are achieved.

The self explanatory flow chart of FIG. 13 shows the various steps taken to determine which adjacent existing instructions in a byte stream are in categories or classes which qualify them for being grouped together to form a compound instruction for a particular computer system configuration.

Referring to FIG. 5, there are many possible locations in a computer system where compounding may occur, both in software and in hardware. Each has unique advantages and disadvantages. As shown in FIG. 5, there are various stages that a program typically takes from source code to actual execution. During the compilation phase, a source program is translated into machine code and stored on a disk 46. During the execution phase the program is read from the disk 46 and loaded into a main memory 48 of a particular computer system configuration 50 where the instructions are executed by appropriate instruction processing units 52, 54, 56. Compounding could take place anywhere along this path. In general as the compounder is located closer to an instruction processing unit or CPUs, the time constraints become more stringent. As the compounder is located further from the CPU, more instructions can be examined in a large sized instruction stream window to determine the best grouping for compounding for increasing execution performance. However such early compounding tends to have more of an impact on the rest of the system design in terms of additional development and cost requirements.

One of the important objects of the invention is to provide a technique for existing programs written in existing high level languages or existing assembly language programs to be processed by software means which can identify sequences of adjacent instructions capable of parallel execution by individual functional units.

The flow diagram of FIG. 6 shows the generation of a compound instruction set program from an assembly language program in accordance with a set of customized compounding rules 58 which reflect both the system and hardware architecture. The assembly language program is provided as an input to a software compounding facility 59 that produces the compound instruction program. Successive blocks of instructions having a predetermined length are analyzed by the software compounding facility 59. The length of each block 60, 62, 64 in the byte stream which contains the group of instructions considered together for compounding is dependent on the complexity of the compounding facility.

As shown in FIG. 6, this particular compounding facility is designed to consider two-way compounding for "m" number of fixed length instructions in each block. The primary first step is to consider if the first and second instructions constitute a compoundable pair, and then if the second and third constitute a compoundable pair, and then if the third and fourth constitute a compoundable pair, all the way to the end of the block.

Once the various possible compoundable pairs C1-C5 have been identified, an additional very desirable step is to determine the optimum choice of compound instructions formed by adjacent scalar instructions for parallel execution. In the example shown, the following different sequences of compounded instructions are possible (assuming no branching): I1, C2, I4, I5, C3, C5, I10; I1, C2, I4, I5, I6, C4, I9, I10; C1, I3, I4, I5, C3, C5, I10; C1, I3, I4, I5, I6, C4, I9, I10. Based on the particular hardware configuration, the compounding facility can select the preferred sequence of compounded instructions and use flags or identifier bits to identify the optimum sequence of compound instructions.

If there is no optimum sequence, all of the compoundable adjacent scalar instructions can be identified so that a branch to a target located amongst various compound instructions can exploit any of the compounded pairs which are encountered (See FIG. 14). Where multiple compounding units are available, multiple successive blocks in the instruction stream could be compounded at the same time.

The specific design of a software compounding facility will not be discussed here because the details are unique to a given instruction set architecture and underlying implementation. Although the design of such compounding programs is somewhat similar in concept to modern compilers which perform instruction scheduling and other optimizations based on a specific machine architecture, the criteria used to complete such compounding are unique to this invention, as best shown in the flow chart of FIG. 13. In both instances, given an input program and a description of the instruction set and also of the hardware architectures (i.e., the structural aspects of the implementation), an output program is produced. In the case of the modern compiler, the output is an optimized new sequence of existing instructions. In the case of the invention, the output is a series of compound instructions each formed by a group of adjacent scalar instructions capable of parallel execution, with the compound instructions being intermixed with non-compounded scalar instructions, and with the necessary control bits for execution of the compound instructions included as part of the output.

Of course, it is easier to pre-process an instruction stream for the purpose of creating compound instructions if known reference points already exist to indicate where instructions begin. As used herein, a reference point means some marking field or other indicator which provides information about the location of instruction boundaries. In many computer systems such a reference point is expressly known only by the compiler at compile time and only by the CPU when instructions are fetched. Such a reference point is unknown between compile time and instruction fetch unless a special reference tagging scheme is adopted.

When compounding is done after compile time, a compiler could indicate with reference tags (see FIG. 11) which bytes contain the first byte of an instruction and which contain data. This extra information results in a more efficient compounder since exact instruction locations are known. Of course, the compiler could identify instructions and differentiate between instructions and data in other ways in order to provide the compounder with specific information indicating instruction boundaries.

When such instruction boundary information is known, the generation of the appropriate compounding identifier bits proceeds in a straightforward manner based on the compounding rules developed for a particular architecture and system hardware configuration (See FIG. 8). When such instruction boundary information is not known, and the instructions are of variable length, a more complex problem is presented (See FIGS. 9 and 16). Incidentally, these figures are based on a preferred encoding scheme described in more detail in Table 2A below, wherein two-way compounding provides a tag bit of "1" if an instruction is compounded with the next instruction, and a tag bit of "0" if it is not compounded with the next instruction.

The control bits in a control field added by a compounder contain information relevant to the execution of compound instructions and may contain as little or as much information as is deemed effective for a particular implementation. An exemplary 8-bit control field is shown in FIG. 12. However, only the first control bit is required in the simplest embodiment to indicate the beginning of a compound instruction. The other control bits provide additional optional information relating to the execution of the instructions.

In an alternate encoding pattern for compounded instructions applicable to both two-way compounding as well as large group compounding, a first control bit is set to "1" to indicate that the corresponding instruction marks the beginning of a compound instruction. All other members of the compound instruction will have their first control bit set to "0". On occasion, it will not be possible to combine a given instruction with other instructions, so such a given instruction will appear to be a compound instruction of length one. That is, the first control bit will be set to "1", but the first control bit of the following instruction will also be set to "1". Under this alternate encoding scheme, the decoding hardware will be able to detect how many instructions comprise the compound instruction by monitoring all of the identifier bits for a series of scalar instructions, rather than merely monitoring the identifier bit for the beginning of a compound instruction as in the preferred encoding scheme shown below in Tables 2A-2C.

The flow diagram of FIG. 7 shows a typical implementation for executing a compound instruction set program which has been generated by a hardware preprocessor 66 or a software preprocessor 67. A byte stream having compound instructions flows into a compound instruction (CI) cache 68 that serves as a storage buffer providing fast access to compound instructions. CI issue logic 69 fetches compound instructions from the CI Cache and issues their individual compounded instructions to the appropriate functions units for parallel execution.

It is to be emphasized that compound instruction execution units (CI EU) 71 such as ALU's in a compound instruction computer system are capable of executing either scalar instructions one at a time by themselves or alternatively compounded scalar instructions in parallel with other compounded scalar instructions. Also, such parallel execution can be done in different types of execution units such as ALU's, floating point (FP) units 73, storage address-generation units (AU) 75 or in a plurality of the same type of units (FP1, FP2, etc) in accordance with the computer architecture and the specific computer system configuration. Thus, the hardware configurations which can implement the present invention are scalable up to virtually unlimited numbers of execution units in order to obtain maximum parallel processing performance. Combining several existing instructions into a single compound instruction allows one or more instruction processing units in a computer system to effectively decode and execute those compounded existing instructions in parallel without the delay that arises in conventional parallel processing computer systems.

In the simplest exemplary encoding schemes of this application, minimal compounding information is added to the instruction stream as one bit for every two bytes of text (instructions and data). In general, a tag containing control information can be added to each instruction in the compounded byte stream--that is, to each non-compounded scalar instruction as well as to each compounded scalar instruction included in a pair, triplet, or larger compounded group. As used herein, identifier bits refers to that part of the tag used specifically to identify and differentiate those compounded scalar instructions forming a compounded group from the remaining non-compounded scalar instructions. Such non-compounded scalar instructions remain in the compound instruction program and when fetched are executed singly.

In a system with all 4-byte instructions aligned on a four byte boundary, one tag is associated with each four bytes of text. Similarly, if instructions can be aligned arbitrarily, a tag is needed for every byte of text.

In the illustrated embodiment herein, all System/370 instructions are aligned on a halfword (two-byte) boundary with lengths of either two or four or six bytes, one tag with identifier bits is needed for every halfword. In a small grouping example for compounding pairs of adjacent instructions, an identifier bit "1" indicates that the instruction that begins in the byte under consideration is compounded with the following instruction, while a "0" indicates that the instruction that begins in the byte under consideration is not compounded. The identifier bit associated with halfwords that do not contain the first byte of an instruction is ignored. The identifier bit for the first byte of the second instruction in a compounded pair is also ignored. (However, in some branching situations, these identifier bits are not ignored.) As a result, this encoding procedure for identifier bits means that in the simplest case of two-way compounding, only one bit of information is needed by a CPU during execution to identify a compounded instruction.

Where more than two scalar instructions can be grouped together to form a compound instruction, additional identifier bits may be required to provide adequate control information. However, in order to reduce the number of bits required for minimal control information, there is still another alternative format for keeping track of the compounding information. For example, even with large group compounding, it is possible to achieve one bit per instruction with the following encoding: the value "1" means to compound with the next instruction, and the value "0" means to not compound with the next instruction. A compound instruction formed with a group of four individual instructions would have a sequence of compounding identifier bits (1,1,1,0). As with the execution of other compound instructions described herein, compounding identifier bits associated with halfwords which are not instructions and therefore do not have any opcodes are ignored at execution time.

Under the preferred encoding scheme described in detail below, the minimum number of identifier bits needed to provide the additional information of indicating the specific number of scalar instructions actually compounded is the logarithm to the base 2 (rounded up to the nearest whole number) of the maximum number of scalar instructions that can be grouped to form a compound instruction. For example, if the maximum is two, then one identifier bit is needed for each compound instruction. If the maximum is three or four, then two identifier bits are needed for each compound instruction. If the maximum is five, six, seven or eight, then three identifier bits are needed for each compound instruction. This encoding scheme is shown below in Tables 2A, 2B and 2C:

TABLE 2A

(maximum of two)

Identifier Bits	Encoded meanings	Total # Compounded
0	This instruction is not compounded with its following instruction	none
1	This instruction is compounded with its one following instruction	two

TABLE 2B

(maximum of four)

Identifier Bits	Encoded meaning	Total # Compounded
00	This instruction is not compounded with its following instruction	none
01	This instruction is compounded with its one following instruction	two
10	This instruction is compounded with its two following instructions	three
11	This instruction is compounded with its three following instructions	four

TABLE 2C

(maximum of eight)

Identifier Bits	Encoded meaning	Total # Compounded
000	This instruction is not compounded	

	with its following instruction	none
001	This instruction is compounded with its one following instruction	two
010	This instruction is compounded with its two following instructions	three
011	This instruction is compounded with its three following instructions	four
100	This instruction is compounded with its four following instructions	five
101	This instruction is compounded with its five following instructions	six
110	This instruction is compounded with its six following instructions	seven
111	This instruction is compounded with its seven following instructions	eight

It will therefore be understood that each halfword needs a tag, but under this preferred encoding scheme the CPU ignores all but the tag for the first instruction in the instruction stream being executed. In other words, a byte is examined to determine if it is a compound instruction by checking its identifier bits. If it is not the beginning of a compound instruction, its identifier bits are zero. If the byte is the beginning of a compound instruction containing two scalar instructions, the identifier bits are "1" for the first instruction and "0" for the second instruction. If the byte is the beginning of a compound instruction containing three scalar instructions, the identifier bits are "2" for the first instruction and "1" for the second instruction and "0" for the third instruction. In other words, the identifier bits for each half word identify whether or not this particular byte is the beginning of a compound instruction while at the same time indicating the number of instructions which make up the compounded group.

These exemplary methods of encoding compound instructions assume that if three instructions are compounded to form a triplet group, the second and third instructions are also compounded to form a pair group. In other words, if a branch to the second instruction in a triplet group occurs, the identifier bit "1" for the second instruction indicates that the second and third instruction will execute as a compounded pair in parallel, even though the first instruction in the triplet group was not executed.

Of course, the invention is not limited to this particular preferred encoding scheme. Various other encoding rules, such as the alternate encoding scheme previously described, are possible within the scope and teachings of the invention.

It will be apparent to those skilled in the art that the present invention requires an instruction stream to be compounded only once for a particular computer system configuration, and thereafter any fetch of compounded instructions will also cause a fetch of the identifier bits associated therewith. This avoids the need for the inefficient last-minute determination and selection of certain scalar instructions for parallel execution that repeatedly occurs every time the same or different instructions are fetched for execution in the so-called super scalar machine.

Despite all of the advantages of compounding a binary instruction stream, it becomes difficult to do so under certain computer architectures unless a technique is developed for determining instruction boundaries in a byte string. Such a determination is complicated when variable length instructions are allowed, and is further complicated when data and instructions can be intermixed, and when modifications are allowed to be made directly to the instruction stream. Of course, at execution time instruction boundaries must be known to allow proper execution. But since compounding is preferably done a sufficient time prior to instruction execution, a unique technique has been developed to compound instructions without knowledge of where instructions start and without knowledge of which bytes are data. This technique is described generally below and can be used for creating compound instructions formed from adjacent pairs of scalar instructions as well as for creating compound instructions formed from larger groups of scalar instructions. This technique is applicable to all instruction sets of the various conventional types of architectures, including the RISC (Reduced Instruction Set Computers) architectures in which instructions are usually a constant length and are not intermixed with data. Additional details of this compounding technique are disclosed in copending application Ser. No. 07/519,382 entitled "General Purpose Compounding Technique For Instruction-Level Parallel Processors" filed May 4th, 1990, now abandoned.

Generally speaking, the compounding technique provides for the compounding two or more scalar instructions from an instruction stream without knowing the starting point or length of each individual instruction. Typical instructions already include an opcode at a predetermined field location which identifies the instruction and its length. Those adjacent instructions which qualify for parallel execution in a particular computer system configuration are provided with appropriate tags to indicate they are candidates for compounding. In IBM System/370 architecture where instructions are either two, four or six bytes in length, the field positions for the opcode are presumed based on an estimated instruction length code. The value of each tag based on a presumed opcode is recorded, and the instruction length code in the presumed opcode is used to locate a complete sequence of possible instructions. Once an actual instruction boundary is found, the corresponding correct tag values are used to identify the commencement of a compound instruction, and other incorrectly generated tags are ignored.

This unique compounding technique is exemplified in the drawings of FIGS. 8-9 and 14-15 wherein the compounding rules are defined to provide that all instructions which are 2 bytes or 4 bytes long are compoundable with each other (i.e., a 2 byte instruction is capable of parallel execution in this particular computer configuration with another 2 byte or another 4 byte instruction). The exemplary compounding rules further provide that all instructions which are 6 bytes long are not compoundable at all (i.e., a 6 byte instruction is only capable of execution singly by itself in this particular computer configuration). Of course, the invention is not limited to these exemplary compounding rules, but is applicable to any set of compounding rules which define the criteria for parallel execution of existing instructions in a specific configuration for a given computer architecture.

The instruction set used in these exemplary compounding techniques of the invention is taken from the System/370 architecture. By examining the opcode for each instruction, the type and length of each instruction can be determined and the control tag containing identifier bits is then generated for that specific instruction, as described in more detail hereinafter. Of course, the present invention is not limited to any specific architecture or instruction set, and the aforementioned compounding rules are by way of example only.

The preferred encoding scheme for compound instructions in these illustrated embodiments has already been shown above in Table 2A-2C.

In a First case with fixed length instructions having no data intermixed and with a known reference point location for the opcode, the compounding can proceed in accordance with the applicable rules for that particular computer configuration. Since the field reserved for the opcode also contains the instruction length, a sequence of scalar instructions is readily determined, and each instruction in the sequence can be considered as possible candidates for parallel execution with a following instruction. A first encoded value in the control tag indicates the instruction is not compoundable with the next instruction, while a second encoded value in the control tag indicates the instruction is compoundable for parallel execution with the next instruction.

In a Second case with variable length instructions having no data intermixed, and with a known reference point location for the opcode and also for the instruction length code (which in System/370 is included as part of the opcode), the compounding can proceed in a routine manner. As shown in FIG. 8, the opcodes indicate an instruction sequence 70 as follows: the first instruction is 6 bytes long, the second and third are each 2 bytes long, the fourth is 4 bytes long, the fifth is 2 bytes long, the sixth is 6 bytes long, and the seventh and eighth are each 2 bytes long.

A C-vector 72 in FIG. 8 shows the values for the identifier bits (called compounding bits in the drawings) for this particular sequence 70 of instructions where a reference point indicating the beginning of the first instruction is known. Based on the values of such identifier bits, the second and third instructions form a compounded pair as indicated by the "1" in the identifier bit for the second instruction. The fourth and fifth instructions form another compounded pair as indicated by the "1" in the identifier bit for the fourth instruction. The seventh and eighth instructions also form a compounded pair as indicated by the "1" in the identifier bit for the seventh instruction.

The C-vector 72 of FIG. 8 is relatively easy to generate when there are no data bytes intermixed with the instruction bytes, and where the instructions are all of the same length with known boundaries.

Another situation is presented in a Third case where instructions are mixed with non-instructions, with a reference point still being provided to indicate the beginning of an instruction. The schematic diagram of FIG. 11 shows one way of indicating an instruction reference point, where every halfword has been flagged with a reference tag to indicate whether or not it contains the first byte of an instruction. This could occur with both fixed length and variable length instructions. By providing the reference point, it is unnecessary to evaluate the data portion of the byte stream for possible compounding. Accordingly, the compounding unit can skip over and ignore all of the non-instruction bytes.

A more complicated situation arises where a byte stream includes variable length instructions (without data), but it is not known where a first instruction begins. Since the maximum length instruction is six bytes, and since instructions are aligned on two byte boundaries, there are three possible starting points for the first instruction the the stream. Accordingly, the technique provides for considering all possible starting points for the first instruction in the text of a byte stream 79, as shown in FIG. 9.

Sequence 1 assumes that the first instruction starts with the first byte, and proceeds with compounding on that premise. In this exemplary embodiment, the length field is also determinative of the C-vector value for each possible instruction. Therefore a C-vector 74 for Sequence 1 only has a "1" value for the first instruction of a possible compounded pair formed by combinations of 2 byte and 4 byte instructions.

Sequence 2 assumes that the first instruction starts at the third byte (the beginning of the second halfword), and proceeds on that premise. The value in the length field for the third byte is 2 indicating the next instruction begins with the fifth byte. By proceeding through each possible instruction based on the length field value in the preceding instruction, the entire potential instructions of Sequence 2 are generated along with the possible identifier bits as shown in a C-vector 76.

Sequence 3 assumes that the first instruction starts at the fifth byte (the beginning of the third halfword), and proceeds on that premise. The value in the length field for the fifth byte is 4 indicating the next instruction begins with the ninth byte. By proceeding through each possible instruction based on the length field value in the preceding instruction, the entire potential instructions of Sequence 3 are generated along with the possible identifier bits as shown in a C-vector 78.

In some instances the three different Sequences of potential instructions will converge into one unique sequence. In FIG. 9 it is noted that the three Sequences converge on instruction boundaries at the end 80 of the eighth byte. Sequences 2 and 3, while converging on instruction boundaries at the end 82 of the fourth byte, are out-of-phase in compounding until the end of the sixteenth byte. In other words, the two sequences consider different pairs of instructions based on the same sequence of instructions. Since the seventeenth byte begins a non-compoundable instruction at 84, the out-of-phase convergence is ended.

When no valid convergence occurs, it is necessary to continue all three possible instruction sequences to the end of the window. However, where valid convergence occurs and is detected, the number of sequences collapses from three to two (one of the identical sequences becomes inoperative), and in some instances from two to one.

Thus, prior to convergence, tentative instruction boundaries are determined for each possible instruction sequence and identifier bits assigned for each such instruction indicating the location of the potential compound instructions. It is apparent from FIG. 9 that this technique generates three separate identifier bits for every two text bytes. In order to provide consistency with the pre-processing done in the aforementioned first, second and third cases, it is desirable to reduce the three possible sequences to a single sequence of identifier bits where only one bit is associated with each halfword. Since the only information needed is whether the current instruction is compounded with the following instruction, the three bits can be logically ORed to produce a single sequence in a CC-vector 86.

For purposes of parallel execution, the composite identifier bits of a composite CC-vector are equivalent to the separate C-vectors of the individual three Sequences 1-3. In other words, the composite identifier bits in the CC-vector allow any of the three possible sequences to execute properly in parallel for compound instructions or singly for non-compounded instructions. The composite identifier bits also work properly for branching. For example, if a branch to the beginning 88 of the ninth byte occurs, then the ninth byte must begin an instruction. Otherwise there is an error in the program. The identifier bit "1" associated with the ninth byte is used and correct parallel execution of such instruction with its next instruction proceeds.

The various steps in the compounding method shown in FIG. 9 as described above are illustrated in the self-explanatory flow chart of FIG. 16.

The best time for providing reference point information for instruction boundaries is at the time of compiling. Reference tags 101 could be added at compile time to identify the beginning of each instruction, as shown in FIG. 11. This enables the compounder to proceed with the simplified technique for the aforementioned First, Second and Third cases. Of course, the compiler could identify instruction boundaries and differentiate between instructions and data in other ways, in order to simplify the work of the compounding unit and avoid the complications of a technique like the one shown in FIG. 9.

FIG. 10 shows a flow diagram of a possible implementation of a compounder for handling instruction streams like the one in FIG. 9. A multiple number of compounder units 104, 106, 108 are shown, and for efficiency purposes this number could be as large as the number of halfwords that could be held in a text buffer. In this version, the three compounder units would begin their processing sequences at the first, third, and fifth bytes, respectively. Upon finishing with a possible instruction sequence, each compounder starts examining the next possible sequence offset by six bytes from its previous sequence. Each compounder produces compound identifier bits (C-vector values) for each halfword in the text. The three sequences from the three compounders are ORed 110 and the resulting composite identifier bits (CC-vector values) are stored in association with their corresponding textual bytes.

One beneficial advantage provided by the composite identifier bits in the CC-vector is the creation of multiple valid compounding bit sequences based on which instruction is addressed by a branch target. As best shown in FIGS. 14-15, differently formed compounded instructions are possible from the same byte stream.

FIG. 14 shows the possible combinations of compounded instructions when the computer configuration provides for parallel issuance and execution of no more than two instructions. Where an instruction stream 90 containing compounded instructions is processed in normal sequence, the Compound Instruction I will be issued for parallel execution based on decoding of the identifier bit for the first byte in a CC-vector 92. However, if a branch to the fifth byte occurs, the Compound Instruction II will be issued for parallel execution based on decoding of the identifier bit for the fifth byte.

Similarly, a normal sequential processing of another compounded byte stream 94 will result in Compound Instructions IV, VI and VIII being sequentially executed (the component instructions in each compound instruction being executed in parallel). In contrast, branching to the third byte in the compounded byte stream will result in Compound Instructions V and VII being sequentially executed, and the instruction beginning at the fifteenth byte (it forms the second part of Compound Instruction

VIII) will be issued and executed singly, all based in the identifier bits in the CC-vector 96.

Branching to the seventh byte will result in Compound Instructions VI and VIII being sequentially executed, and branching to the eleventh byte will result in Compound Instruction VIII being executed. In contrast, branching to the ninth byte in the compounded byte stream will result in Compound Instruction VII being executed (it is formed by the second part of Compound Instruction VI and the first part of Compound Instruction VIII).

Thus, the identifier bits "1" in the CC-vector 96 for Compound Instructions IV, VI and VIII are ignored when either of the Compound Instructions V or VII is being executed. Alternatively the identifier bits "1" in the CC-vector 96 for Compound Instructions V and VII are ignored when any of Compound Instructions IV, VI or VIII are executed. FIG. 15 shows the possible combinations of compounded instructions when the computer configuration provides for parallel issuance and execution of up to three instructions. Where an instruction stream 98 containing compounded instructions is processed in normal sequence, the Compound Instructions X (a triplet group) and XIII (a pair group) will be executed. In contrast, branching to the eleventh byte will result in Compound Instruction XI (a triplet group) being executed, and branching to the thirteenth byte will result in Compound Instruction XII (a different triplet group) being executed.

Thus, the identifier bits "2" in a CC-vector 99 for Compound Instructions XI and XII are ignored when Compound Instructions X and XIII are executed. On the other hand when Compound Instruction XI is executed, the identifier bits for the other three Compound Instructions X, XII, XII are ignored. Similarly when Compound Instruction XII is executed, the identifier bits for the other three Compound Instructions X, XI, XIII are ignored.

There are many possible designs for an instruction compounding unit depending on its location and the knowledge of the text contents. In the simplest situation, it would be desirable for a compiler to indicate with reference tags which bytes contain the first byte of an instruction and which contain data. This extra information results in a more efficient compounder since exact instruction locations are known. This means that compounding could always be handled as in the First, Second and Third case situations in order to generate the C-vector identifier bits for each compound instruction. A compiler could also add other information such as static branch prediction or even insert directives to the compounder.

Other ways could be used to differentiate data from instructions where the instruction stream to be compounded is stored in memory. For example, if the data portions are infrequent, a simple list of addresses containing data would require less space than reference tags. Such combinations of a compounder in hardware and software provide many options for efficiently producing compound instructions.

While exemplary preferred embodiments of the invention have been disclosed, it will be appreciated by those skilled in the art that various modifications and changes can be made without departing from the spirit and scope of the invention as defined by the following claims.

* * * * *

United States Patent	***5,500,942***
Eickemeyer , et al.	**March 19, 1996**

Method of indicating parallel execution compoundability of scalar instructions based on analysis of presumed instructions

Abstract

This is a method of compounding two or more instructions from an instruction stream without knowing the starting point or length of each individual instruction. All instructions include one OP Code at a predetermined field location which identifies the instruction and its length. Those instructions which qualify need to have appropriate tags to indicate they are candidates for compounding. In System 370 where instructions are either 2, 4 or 6 bytes in length, the field positions for the OP Code are presumed based on an estimated instruction length code. The value of each tag based on a presumed OP Code is recorded, and the instruction length code in the presumed OP Code is used to locate a complete sequence of possible instructions. Once an actual instruction boundary is found, the corresponding correct tag values are used to identify the commencement of a compound instruction, and other incorrectly generated tags are ignored.

Inventors:	**Eickemeyer; Richard J.** (Endicott, NY); **Vassiliadis; Stamatis** (Vestal, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**457765**
Filed:	**June 1, 1995**

Current U.S. Class:	**712/210**; 712/23; 712/24; 712/215
Intern'l Class:	G06F 009/38
Field of Search:	395/500,375,700,800

References Cited [Referenced By]

U.S. Patent Documents

4295193	Oct., 1981	Pomerene	364/200.
4439828	Mar., 1984	Martin	364/200.
4454578	Jun., 1984	Matsumoto et al.	364/200.
4502111	Feb., 1985	Riffe et al.	
4506325	Mar., 1985	Bennett et al.	364/200.
4847755	Jul., 1989	Morrison et al.	364/200.
5051885	Sep., 1991	Yates, Jr. et al.	364/200.
5051940	Sep., 1991	Vassiliadis et al.	364/736.
5202967	Apr., 1993	Matsuzaki et al.	395/375.
5203002	Apr., 1993	Wetzel	395/800.
5337415	Aug., 1994	Delano et al.	395/375.

Foreign Patent Documents

0354740	Feb., 1990	EP.
0363222	Apr., 1990	EP.
61-245239	Mar., 1987	JP.

Other References

Wang, Lingtao, "Distributed Instruction Set Computer", Proceedings of the International Conference on Parallel Processing, vol. 1, Aug. 1988, pp. 426-429.
Higbee, "Overlapped Operation with Microprogramming", IEEE Transactions on Computers, vol. C-27, No. 3, Mar. 1978, pp. 270-275.
Acosta, R. D., et al, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35, No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al, the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al, ed (McGraw-Hill, 1982, pp. 276-292.
Capozzi, A. J., et al, "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al, "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul. 1989, pp. 21-35.
Smith, M. D., et al, "Limits on Multiple Instruction Issue", Asplos III, 1989, pp. 290-302.

Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al, "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N. P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp. 63-76.
Colwell, R. P., et al, "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the CRISP Instruction Set Architecture", Proceedings of Compcon, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al, "Compiling for the CRISP Microprocessor", Proceedings of Compcon, Spring, 1987, pp. 96-100.
Hennessy, J., et al, "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, Jan. 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al, "Branch Folding in the Crisp Microprocessor: Reducing Branch Delay to Zero", Proceedings of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W. et al, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al, "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1. Jan. 1984, pp. 6-22.
Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James et al, Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al, "A Study of Instruction Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun., 1983, pp. 132-137.
Vassiliadis, S., et al, "Condition Code Predictory for Fixed-Arithmetic Units", International

Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
"Early release of a processor following address translation prior to page access checking" IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard pp. 371-374.

Primary Examiner: Kim; Ken S.
Attorney, Agent or Firm: Augspurger; Lynn L., Marhoefer; Laurence J., Aitken; Richard L.

Parent Case Text

This application is a continuation of application Ser. No. 08/184,891, filed Jan. 21, 1994, now abandoned, which is a continuation of application Ser. No. 08/015,272, filed Feb. 5, 1993, now abandoned, which is a continuation of application Ser. No. 07/519,382, filed May 4, 1990, now abandoned.

Claims

We claim:

1. A method of processing instructions in an instruction stream in a data processing system, said instruction stream having no known instruction boundary reference points, to identify adjacent scalar instructions which are capable of parallel execution in a particular computer configuration, including operating said data processing system to perform the following steps:

generating different sequences of presumed instructions starting at different possible instruction boundaries; at least one of said sequences being a sequence of real instructions;

encoding said presumed instructions with identifier tags indicating the capability of adjacent presumed instructions of being executed in parallel and;

controlling the execution of said real instructions to be serial or in parallel in accordance with said identifier tags.

2. The method of claim 1 wherein the instructions have a given number of different possible lengths, and wherein said identifying step includes creating a different sequence of presumed instructions for each of said given number of different possible lengths.

3. The method of claim 2 wherein said given number of possible lengths is two or more.

4. The method of claim 2 wherein said given number of possible lengths are aligned on the byte boundaries when the different sequence is created.

5. The method of claim 1 wherein the capability of being executed in parallel is determined in said encoding step by comparing groups of two or more adjacent presumed instructions which instructions are in adjacent relationship with each other.

6. The method of claim 1 further comprising the step of combining said identifier tags into a composite sequence of identifier tags, said composite sequence of identifier tags providing an indication of which scalar instructions of said instruction stream can be executed in parallel.

7. The method of claim 6 wherein said identifier tags are binary bits which are combined in said composite sequence by ORing the binary bits of identifier tags encoded with corresponding presumed instructions.

8. The method of claim 6 wherein said identifier tags have digital values identifying the number of succeeding instructions in a presumed sequence which can be executed in parallel, said identifier tags in said composite sequence each having the highest digital value for the corresponding presumed instruction encoded in said step of encoding.

9. The method of claim 7 further comprising controlling the parallel execution of instructions in said instruction stream in accordance with said composite sequence of identifier tags by responding to the identifier tags corresponding to the first instruction of each sequence of adjacent instructions which can be executed in parallel and ignoring all other identifier tags in said composite sequence.

10. A method to determine and to indicate possible parallel execution of scaler instructions in a stream of instructions in a data processing system prior to fetching said instructions for execution by a processing unit of said data processing system, comprising operating said data processing system to perform the following steps:

grouping a byte stream in a window of a predetermined length into byte sequences that each comprise presumed instructions within said window;

generating a first compounding bit vector that indicates whether or not each presumed instruction within said window and its next adjacent presumed instruction can be compounded in accordance with compounding instruction rules;

grouping the bytes in said byte stream window of predetermined length in an another sequence that comprises another presumed instruction sequence in said window;

generating a second compounding bit vector;

repeating the previous steps for new presumed instruction sequences in said window a sufficient number of times to ensure that the actual instruction sequence within said window has been compounded;

forming a composite compounding vector by combining each of said compounding vectors from said previous steps into a single vector;

controlling the fetching of instructions in said window to be executed serially or in parallel by said processor unit in accordance with said composite compounding vector.

11. A method of identifying as in claim 10 including operating the data processing system to perform the further steps of:

examining a presumed first instruction to determine a first instruction length field value for said presumed first instruction in said first possible instruction sequence;

using said first instruction length field value to locate at least a presumed second instruction; and

encoding said presumed first and said at least second instructions with a control identifier bit tag to indicate whether or not they are tagged as a potential instruction component of a compound instruction for parallel execution by a particular computer system configuration whereby in accordance with the hardware utilization requirements of the particular computer system at least two instructions are tagged for compounding and parallel execution in the particular computer system configuration.

12. The method of claim 11 further including the steps of

examining the presumed second instruction to determine a second instruction length field value for said presumed at least second instruction in said first possible instruction sequence;

using said second instruction length field value to locate a presumed at least third instruction; and

encoding said presumed second and said at least third instructions with a control identifier bit tag to indicate whether or not they are tagged as a potential instruction component of a compound instruction for parallel execution by a particular computer system configuration in accordance with the hardware utilization requirements of the particular computer system configuration.

13. The method of claim 12 wherein an encoding tag identifying the largest number of said as least three presumed instructions capable of parallel execution is maintained for use at instruction execution time in the particular computer system for which the preprocessing is employed.

14. The method of claim 12 wherein said encoding step includes encoding said presumed first and second and said at least third instructions with tags to indicate whether they may be compounded for parallel execution by a particular computer system configuration.

15. The method of claim 12 further including the steps of

keeping track of the byte positions of the byte stream of information associated with said presumed instructions in said first possible instruction sequence;

keeping track of the byte positions associated with said presumed instructions in said second possible instruction sequence; and

maintaining a separate identifier tag for each of the byte positions associated with said presumed instructions.

16. The method of claim 15 wherein an instruction is tagged for parallel execution whenever an instruction in either of said first or second possible instruction sequences in the byte stream of information is encoded for parallel execution.

17. The method of claim 11 further including the steps of

starting a second possible instruction sequence by selecting another presumed instruction different from said presumed first instruction from said byte stream of information;

examining said another presumed instruction to determine another instruction length field value for said another presumed instruction in said second possible instruction sequence;

using said another instruction length field value to locate a further presumed instruction; and

encoding said further and another presumed instructions with a control identifier bit tag to indicate whether or not they are tagged as a potential instruction component of a compound instruction for parallel execution by a particular computer system configuration in accordance with the hardware utilization requirements of the particular computer system configuration.

18. The method of claim 17 further including the steps of

comparing said presumed first and second instructions in said first possible instruction sequence with said further and another presumed instructions in said second possible instruction sequence to detect a valid convergence between instruction boundaries and, if a valid convergence occurs, to collapse the sequences, and if a valid convergence does not occur, to continue the instruction sequences until the end of a window of possible instruction, such that, prior to convergence, tentative instruction boundaries are determined for each possible instruction sequence and control identifier bits are assigned for each instruction identifying the location of potential compound instructions in the byte stream of information.

19. The method of claim 11 wherein the byte stream of information includes instructions having a fixed length.

20. The method of claim 11 wherein the byte stream of information includes instructions having a variable length.

21. The method of claim 11 wherein the byte stream of information includes non-instructions intermixed with instructions.

22. The method of claim 11 wherein there are not any instruction boundary reference points in the byte stream of information.

23. The method of claim 22 wherein the byte stream of information includes non-instructions intermixed with instructions.

24. The method of claim 23 wherein the byte stream of information includes instructions having a variable length.

Description

RELATED APPLICATIONS

The following related applications are commonly owned by the same assignee and are incorporated by reference herein: "Data Dependency Collapsing Hardware Apparatus" filed Apr. 4, 1990, Ser. No. 07/504,910, now issued U.S. Pat. No. 5,051,940, and "Scalable Compound Instruction Set Machine Architecture" Ser. No. 07/519,384, filed May 4, 1990, now abandoned.

FIELD OF THE INVENTION

This invention relates to parallel processing of instructions in a computer, and more particularly relates to processing a stream of binary information having instructions therein for the purpose of identifying those instructions which can be executed in parallel in a specific computer configuration.

BACKGROUND OF THE INVENTION

The concept of parallel execution of instructions has helped to increase the performance of computer systems. Parallel execution is based on having separate functional units which can execute two or more of the same or different instructions simultaneously.

Another technique used to increase the performance of computer systems is pipelining. Pipelining does provide a form of parallel processing since it is possible to execute multiple instructions concurrently.

However, many times the benefits of parallel execution and/or pipelining are not achieved because of delays like those caused by data dependent interlocks and hardware dependent interlocks. An example of a data dependent interlock is a so-called write-read interlock where a first instruction must write its result before the second instruction can read and subsequently use it. An example of hardware dependent interlock is where a first instruction must use a particular hardware component and a second instruction must also use the same particular hardware component.

One of the techniques previously employed to avoid interlocks (sometimes called pipeline hazards) is called dynamic scheduling. Dynamic scheduling means that shortly before execution, the opcodes in an instruction stream are decoded to determine whether the instructions can be executed in parallel. Computers which practice one type of such dynamic scheduling are sometimes called superscalar machines. The criteria for dynamic scheduling are unique to each instruction set architecture, as well for the underlying implementation of that architecture in any given instruction processing unit. The effectiveness of dynamic scheduling is therefore limited by the complexity of the architecture which leads to extensive logic to determine which combinations of instructions can be executed in parallel, and thus may increase the cycle time of the instruction processing unit. The increased hardware and cycle time for such dynamic scheduling become even a bigger problem in architectures which have hundreds of different instructions.

There have also been some attempts to improve performance through so-called static scheduling which is done before the instruction stream is fetched from storage for execution. Static scheduling is achieved by moving code and thereby reordering the instruction sequence before execution. This reordering produces an equivalent instruction stream that will more fully utilize the hardware through parallel processing. Such static scheduling is typically done at compile time. However, the reordered instructions remain in their original form and conventional parallel processing still requires some form of dynamic determination just prior to execution of the instructions in order to decide whether to execute the next two instructions serially or in parallel.

There are other deficiencies with dynamic scheduling, static scheduling, or combinations thereof. For example, it is necessary to review each scalar instruction anew every time it is fetched for execution to determine its capability for parallel execution. There has been no way provided to identify and flag ahead of time those scalar instructions which have parallel execution capabilities.

Another deficiency with dynamic scheduling of the type implemented in superscalar machines is the manner in which scalar instructions are checked for possible parallel processing. Super scalar machines check scalar instructions based on their opcode descriptions, and no way is provided to take into account hardware utilization. Also, instructions are issued in FIFO fashion thereby eliminating the possibility of selective grouping to avoid or minimize the occurrence of interlocks.

There are some existing techniques which do seek to consider the hardware requirements for parallel instruction processing. One such system is called the Very Long Instruction Word machine in which a sophisticated compiler rearranges instructions so that hardware instruction scheduling is simplified. In this approach the compiler must be more complex than standard compilers so that a bigger window can be used for purposes of finding more parallelism in an instruction stream. But the resulting instructions may not necessarily be object code compatible with the pre-existing architecture, thereby solving one problem while creating additional new problems. Also, substantial additional problems arise due to frequent branching which limits its parallelism.

A recent innovation which seeks to more fully exploit parallel execution of instructions is called Scalable Compound Instruction Set Machines (SCISM). A compound instruction is created by pre-processing an instruction stream in order to look for sets of two or more adjacent scalar instructions that can be executed in parallel. In some instances certain types of interlocked instructions can be compounded for parallel execution where the interlocks are collapsible in a particular hardware configuration. In other configurations where the interlocks are non-collapsible, the instructions having data dependent or hardware dependent interlocks are excluded from groups forming compound instructions. Each compound instruction is identified by control information such as tags associated with the compound instruction, and the length of a compound instruction is scalable over a range beginning with a set of two scalar instructions up to whatever maximum number of individual scalar instructions can be processed together by the specific hardware implementation.

When an instruction is fetched for execution, the instruction boundaries must be known in order to allow proper execution. However, where an instruction stream is pre-processed for purposes of creating compound instructions, the instruction boundaries are often not evident merely by examining a byte string. This is particularly true with architectures which allow variable length instructions. Further complications arise when the architecture allows data and instructions to be intermixed.

For example, in the IBM System 370 architecture, both of these difficulties make the pre-processing of an instruction stream to locate suitable scalar instruction groupings a very complex problem. First, the instructions have three possible lengths--two bytes or four bytes or six bytes. Even though the actual length of a particular instruction is indicated in the first two bits of the opcode of the instruction, the beginning of an instruction in a string of bytes cannot be readily identified by mere inspection. Second, instructions and data can be intermixed. Accordingly, the existence or non-existence of a reference point in an instruction byte stream is of critical importance for this invention. A reference point is defined as the knowledge of where instructions begin or where instruction boundaries are. Unless additional information has been added to the instruction stream, instruction boundaries are usually known only at compile time or at execution time when the instructions are fetched by a CPU.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is an object of the present invention to provide a technique for generating compound instructions from a binary instruction stream without knowing where instructions start and without knowing which bytes contain data instead of instructions.

Another object of the invention is to add control information to the instruction stream including grouping information indicating where a compound instruction starts as well as indicating the number of scalar instructions which are incorporated into the compound instruction.

A further object is to provide a technique which is applicable to complex instruction architectures having variable length instructions and data intermixed with instructions, and which is also applicable to RISC architectures wherein instructions are usually a constant length and wherein data is not mixed with instructions.

Still another object is to provide a method of pre-processing an instruction stream to create compound instructions composed of scalar instructions which have still retained their original contents. A related object is to create compound instructions without changing the object code of the scalar instructions which form the compound instruction, thereby allowing existing programs to realize a performance improvement on a compound instruction machine while maintaining compatibility with previously implemented scalar instruction machines.

An additional object is to provide a method of pre-processing an instruction stream to create compound instructions, wherein the method can be implemented by software and/or hardware at various points in the computer system prior to instruction execution. A related object is to provide a method of pre-processing of instructions which operates on a binary instruction stream as part of a post-compiler, or as part of an in-memory compounder, or as part of cache instruction compounding unit, and which can start compounding instructions at the beginning of a byte stream without knowing the boundaries of the instructions.

Thus, the invention seeks to achieve the aforementioned objectives by pre-processing a set of instructions (or a program) to determine statically which instructions may be combined into compound instructions. Such processing is done in a typical embodiment by software and/or hardware means which will look for classes of instructions that can be executed in parallel in a particular computer system configuration. The instruction classes and the compounding rules are implementation specific and will vary depending on the number and type of functional execution units. While keeping their original sequence and object code intact, individual instructions are selectively grouped and combined with one or more other adjacent scalar instructions to form a compound instruction byte stream having both compounded scalar instructions for parallel execution and non-compounded scalar instructions for execution singly. Control information is appended to identify information relevant to the execution of the compound instructions.

More specifically, this invention provides a technique of compounding two or more scalar instructions from an instruction stream without knowing the starting point or length of each individual instruction. All possible instruction sequences are considered by looking at a predetermined field location for a presumed instruction length. In an IBM System/370 system, the instruction length is part of the opcode. In other systems, the instruction length is part of the operands. In some instances of practicing the technique of the invention, a valid convergence occurs between two possible instruction sequences, thereby narrowing the possible choices for instruction boundaries. In other instances where no valid convergence occurs, the various possible instruction sequences are followed to the end of the byte stream. The actual instructions boundaries are not known until the instructions are fetched for execution. So all authentic instructions as well as all spurious instructions are encoded with identifier tag bits based on the particular compounding rules which apply to the hardware configuration. In IBM System/370 architecture instructions are either two, four or six bytes in length, based on the instruction length codes. The value of each identifier tag bit (based on a presumed opcode position) is recorded for each possible two, four or six byte instruction. Once an actual instruction boundary is found at execution, the corresponding correct tag values are used to identify the commencement of a compound instruction and/or the commencement of a non-compounded instruction, and other incorrectly generated tags are ignored.

These and other objects, features and advantages of the invention will be apparent to those skilled in the art in view of the following detailed description and accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a high level schematic diagram of the invention;

FIG. 2 is a timing diagram for a uniprocessor implementation showing the parallel execution of certain non-interlocked instructions which have been selectively grouped in a compound instruction stream;

FIG. 3 is a timing diagram for a multiprocessor implementation showing the parallel execution of scalar and compound instructions which are not interlocked;

FIG. 4 (FIG. 4 comprising FIGS. 4A and 4B) illustrates selective categorization of instructions executed by an existing scalar machine;

FIG. 5 shows a typical path taken by a program from source code to actual execution;

FIG. 6 is a flow diagram showing generation of a compound instruction set program from an assembly language program;

FIG. 7 is a flow diagram showing execution of a compound instruction set program;

FIGS. 8 is an analytical chart for instruction stream texts with identifiable instruction reference points;

FIG. 9 is an analytical chart for an instruction stream text with variable length instructions without a reference point, showing their related sets of possible compound identifier bits;

FIG. 10 is an analytical chart for a worst case instruction stream text having data intermixed with variable length instructions without a reference point, showing their related sets of possible compound identifier bits;

FIG. 11 illustrates a logical implementation of an instruction compound facility for handling the instruction stream texts of FIGS. 9 and 10;

FIG. 12 is an analytical chart for the worst case instruction text of FIG. 10, showing the sets of possible compound identifier bits for grouping up to four scalar instructions to form each compound instruction;

FIG. 13 is a flow diagram for compounding an instruction stream having tags to identify instruction boundary reference points;

FIG. 14 shows how different groupings of valid non-interlocked pairs of instructions form multiple compound instructions for sequential or branch target execution;

FIG. 15 shows how different groupings of valid non-interlocked triplets of instructions form multiple compound instructions for sequential or branch target execution; and

FIG. 16 (FIG. 16 comprising FIGS. 16A and 16B) is a flow chart for compounding an instruction stream like the one shown in FIG. 9.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

As shown in the various drawings to be described in more detail hereinafter, a recent innovation called a Scalable Compound Instruction Set Machine (SCISM) provides for a stream of scalar instructions to be compounded or grouped together before instruction decode time so that they are already flagged and identified for simultaneous parallel execution by appropriate instruction execution units. Since such compounding does not change the object code, existing programs can realize a performance improvement while maintaining compatibility with previously implemented systems.

As generally shown in FIG. 1, an instruction compounding unit 20 takes a stream of binary scalar instructions 21 (with or without data included therein) and selectively groups some of the adjacent scalar instructions to form encoded compound instructions. A resulting compounded instruction stream 22 therefore combines scalar instructions not capable of parallel execution and compound instructions formed by groups of scalar instructions which are capable of parallel execution. When a scalar instruction is presented to an instruction processing unit 24, it is routed to the appropriate functional unit for serial execution. When a compound instruction is presented to the instruction processing unit 24, its scalar components are each routed to their appropriate functional unit or interlock collapsing unit for simultaneous parallel execution. Typical functional units include but are not limited to an arithmetic and logic unit (ALU) 26, 28, a floating point arithmetic unit (FP) 30, and a store address generation unit (AU) 32. An exemplary data dependency collapsing unit is disclosed in co-pending application Serial No. 07/504,910, entitled "Data Dependency Collapsing Hardware Apparatus) filed Apr. 4, 1990. "Collapsing" was a word first used in connection with such an apparatus, and as such may be considered as being defined by the co-pending application U.S. Ser. No. 07/504,910.

It is to be understood that the technique of the invention is intended to facilitate the parallel issue and execution of instructions in all computer architectures that process multiple instructions per cycle (although certain instructions may require more than one cycle to be executed).

As shown in FIG. 2, the invention can be implemented in a uniprocessor environment where each functional execution unit executes a scalar instruction (S) or alternatively a compounded scalar instruction (CS). As shown in the drawing, an instruction stream 33 containing a sequence of scalar and compounded scalar instructions has control tags (T) associated with each compound instruction. Thus, a first scalar instruction 34 could be executed singly by functional unit A in cycle 1; a triplet compound instruction 36 identified by tag T3 could have its three compounded scalar instructions executed in parallel by functional units A, C and D in cycle 2; another compound instruction 38 identified by tag T2 could have its pair of compounded scalar instructions executed in parallel by functional units A and B in cycle 3; a second scalar instruction 40 could be executed singly by functional unit C in cycle 4; a large group compound instruction 42 could have its four compounded scalar instructions executed in parallel by functional units A-D in cycle 5; and a third scalar instruction 44 could be executed singly by functional A in cycle 6.

It is important to realize that multiple compound instructions are capable of parallel execution in certain computer system configurations. For example, the invention could be potentially implemented in a multiprocessor environment as shown in FIG. 3 where a compound instruction is treated as a unit for parallel processing by one of the CPUs (central processing units). As shown in the drawing, the same instruction stream 33 could be processed in only two cycles as follows. In a first cycle, a CPU #1 executes the first scalar instruction 34; the functional units of a CPU #2 execute triplet compound instruction 36; and the functional units of a CPU #3 execute the two compounded scalar instructions in compound instruction 38. In a second cycle, the CPU #1 executes the second scalar instruction 40; the functional units of CPU #2 execute the four compounded scalar instructions in compound instruction 42; and a functional unit of CPU #3 executes the third scalar instruction 44.

One example of a computer architecture which can be adapted for handling compound instructions is an IBM System/370 instruction level architecture in which multiple scalar instructions can be issued for execution in each machine cycle. In this context a machine cycle refers to all the pipeline steps or stages required to execute a scalar instruction. A scalar instruction operates on operands representing single-valued parameters. When an instruction stream is compounded, adjacent scalar instructions are selectively grouped for the purpose of concurrent or parallel execution.

The instruction sets for various IBM System/370 architectures such as System/370, the System/370 extended architecture (370-XA), and the System/370 Enterprise Systems Architecture (370-ESA) are well known. In that regard, reference is given here to the Principles of Operation of the IBM System/370 (publication #GA22-7000-10 1987), and to the Principles of Operation, IBM Enterprise Systems Architecture/370 (publication #SA22-7200-0 1988).

In general, an instruction compounding facility will look for classes of instructions that may be executed in parallel, and ensure that no interlocks between members of a compound instruction exist that cannot be handled by the hardware. When compatible sequences of instructions are found, a compound instruction is created.

More specifically, the System/370 instruction set can be broken into categories of instructions that may be executed in parallel in a particular computer system configuration. Instructions within certain of these categories may be combined or compounded with instructions in the same category or with instructions in certain other categories to form a compound instruction. For example, the System/370 instruction set can be partitioned into the categories illustrated in FIG. 4. The rationale for this categorization is based on the functional requirements of the System/370 instructions and their hardware utilization in a typical computer system configuration. The rest of the System/370 instructions are not considered specifically for compounding in this exemplary embodiment. This does not preclude them from being compounded by the technique of the present invention disclosed herein.

For example, consider the instructions contained in category 1 compounded with instructions from that same category in the following instruction sequence:

AR R1,R2

SR R3,R4

This sequence is free of data hazard interlocks and produces the following results which comprise two independent System/370 instructions:

R1=R1+R2

R3=R3-R4

Executing such a sequence would require two independent and parallel two-to-one ALU's designed to the instruction level architecture. Thus, it will be understood that these two instructions can be grouped to form a compound instructions in a computer system configuration which has two such ALU's. This example of compounding scalar instructions can be generalized to all instruction sequence pairs that are free of data dependent interlocks and also of hardware dependent interlocks.

In any actual instruction processor, there will be an upper limit to the number of individual instructions that can comprise a compound instruction. This upper limit must be specifically incorporated into the hardware and/or software unit which is creating the compound instructions, so that compound instructions will not contain more individual instructions (e.g., pair group, triplet group, group of four) that the maximum capability of the underlying execution hardware. This upper limit is strictly a consequence of the hardware implementation in a particular computer system configuration--it does not restrict either the total number of instructions that may be considered as candidates for compounding or the length of the group window in a given code sequence that may be analyzed for compounding. In general, the greater the length of a group window being analyzed for compounding, the greater the parallelism that can be achieved due to more advantageous compounding combinations.

Referring to FIG. 5, there are many possible locations in a computer system where compounding may occur, both in software and in hardware. Each has unique advantages and disadvantages. As shown in FIG. 5, there are various stages that a program typically takes from source code to actual execution. During the compilation phase, a source program is translated into machine code and stored on a disk 46. During the execution phase the program is read from the disk 46 and loaded into a main memory 48 of a particular computer system configuration 50 where the instructions are executed by appropriate instruction processing units 52, 54, 56. Compounding could take place anywhere along this path. In general as the compounder is located closer to an instruction processing unit or CPUs, the time constraints become more stringent. As the compounder is located further from the CPU, more instructions can be examined in a large sized instruction stream window to determine the best grouping for compounding for increasing execution performance. However such early compounding tends to have more of an impact on the rest of the system design in terms of additional development and cost requirements.

The flow diagram of FIG. 6 shows the generation of a compound instruction set program from an assembly language program in accordance with a set of customized compounding rules 58 which reflect both the system and hardware architecture. The assembly language program is provided as an input to a software compounding facility 59 that produces the compound instruction program. Successive blocks of instructions having a predetermined length are analyzed by the software compounding facility 59. The length of each block 60, 62, 64 in the byte stream which contains the group of instructions considered together for compounding is dependent on the complexity of the compounding facility.

As shown in FIG. 6, this particular compounding facility is designed to consider two-way compounding for "m" number of fixed length instructions in each block. The primary first step is to consider if the first and second instructions constitute a compoundable pair, and then if the second and third constitute a compoundable pair, and then if the third and fourth constitute a compoundable pair, all the way to the end of the block. Once the various possible compoundable pairs C1-C5 have been identified, the compounding facility can select the preferred sequence of compounded instructions and use flags or identifier bits to identify the optimum sequence of compound instructions.

If there is no optimum sequence, all of the compoundable adjacent scalar instructions can be identified so that a branch to a target located amongst various compound instructions can exploit any of the compounded pairs which are encountered (See FIG. 14). Where multiple compounding units are available, multiple successive blocks in the instruction stream could be compounded at the same time.

Of course it is easier to pre-process an instruction stream for the purpose of creating compound instructions if known reference points already exist to indicate where instructions begin. As used herein, a reference point means knowledge of which byte of text is the first byte in an instruction. This knowledge could be obtained by some marking field or other indicator which provides information about the location of instruction boundaries. In many computer systems such a reference point is expressly known only by the compiler at compile time and only by the CPU when instructions are fetched. Such a reference point is unknown between compile time and instruction fetch unless a special reference tagging scheme is adopted.

The flow diagram of FIG. 7 shows the execution of a compound instruction set program which has been generated by a hardware preprocessor 66 or a software proprocessor 67. A byte stream having compound instructions flows into a compound instruction (CI) cache 68 that serves as a storage buffer providing fast access to compound instructions. CI issue logic 69 fetches compound instructions from the CI Cache and issues their individual compounded instructions to the appropriate functional units for parallel execution.

It is to be emphasized that instruction execution units (CI EU) 71 such as ALU's in a compound instruction computer system are capable of executing either scalar instructions one at a time by themselves or alternatively compounded scalar instructions in parallel with other compounded scalar instructions. Also, such parallel execution can be done in different types of execution units such as ALU's, floating point (FP) units 73, storage address-generation units (AU) 75 or in a plurality of the same type of units (FP1, FP2, etc) in accordance with the computer architecture and the specific computer system configuration.

When compounding is done after compile time, a compiler could indicate with tags which bytes contain the first byte of an instruction and which contain data. This extra information results in a more efficient compounder since exact instruction locations are known. Of course, the compiler could differentiate between instructions and data in other ways in order to provide the compounder with specific information indicating instruction boundaries.

In the exemplary two-way compounding embodiment of this application, compounding information is added to the instruction stream as one bit for every two bytes of text (instructions and data). In general, a tag containing control information can be added to each instruction in the compounded byte stream--that is, to each non-compounded scalar instruction as well as to each compounded scalar instruction included in a pair, triplet, or larger compounded group. As used herein, identifier bits refers to that part of the tag used specifically to identify and differentiate those compounded scalar instructions forming a compounded group from the remaining non-compounded scalar instructions. Such non-compounded scalar instructions remain in the compound instruction program and when fetched are executed singly.

In a system with all 4-byte instructions aligned on a four byte boundary, one tag is associated with each four bytes of text. Similarly, if instructions can be aligned arbitrarily, a tag is needed for every byte of text.

The case of compounding at most two instructions provides the smallest grouping of scalar instructions to form a compound instruction, and uses the following preferred encoding procedure for the identifier bits. Since all System/370 instructions are aligned on a halfword (two-byte) boundary with lengths of either two or four or six bytes, one tag with identifier bits is needed for every halfword. In this small grouping example, an identifier bit "1" indicates that the instruction that begins in the byte under consideration is compounded with the following instruction, while a "0" indicates that the instruction that begins in the byte under consideration is not compounded. The identifier bit associated with halfwords that do not contain the first byte of an instruction is ignored. The identifier bit for the first byte of the second instruction in a compounded pair is also ignored. As a result, this encoding procedure for identifier bits means that in the simplest case only one bit of information is needed by a CPU during execution to identify a compounded instruction.

Where more than two scalar instructions can be grouped together to form a compound instruction, additional identifier bits may be required. The minimum number of identifier bits needed to indicate the specific number of scalar instructions actually compounded is the logarithm to the base 2 (rounded up to the nearest whole number) of the maximum number of scalar instructions that can be grouped to form a compound instruction. For example, if the maximum is two, then one identifier bit is needed for each compound instruction. If the maximum is three or four, then two identifier bits are needed for each compound instruction. If the maximum is five, six, seven or eight, then three identifier bits are needed for each compound instruction. This encoding scheme is shown below in Table 1:

TABLE 1

Identifier Bits	Encoded meaning	Total # Compounded
00	This instruction is not compounded with its following instruction	none
01	This instruction is compounded with its one following instruction	two
10	This instruction is compounded with its two following instructions	three
11	This instruction is compounded with its three following instructions	four

It will therefore be understood that each halfword needs a tag, but the CPU ignores all but the tag for the first instruction in the instruction stream being executed. In other words, a byte is examined to determine if it is a compound instruction by checking its identifier bits. If it is not the beginning of a compound instruction, its identifier bits are zero. If the byte is the beginning of a compound instruction containing two scalar instructions, the identifier bits are "1" for the first instruction and "0" for the second instruction. If the byte is the beginning of a compound instruction containing three scalar instructions, the identifier bits are "2" for the first instruction and "1" for the second instruction and "0" for the third instruction. In other words, the identifier bits for each half word identify whether or not this particular byte is the beginning of a compound instruction while at the same time indicating the number of instructions which make up the compounded group.

This method of encoding compound instructions assumes that if three instructions are compounded to form a triplet group, the second and third instructions are also compounded to form a pair group. In other words, if a branch to the second instruction in a triplet group occurs, the identifier bit "1" for the second instruction indicates that the second and third instruction will execute as a compounded pair in parallel, even though the first instruction in the triplet group was not executed.

It will be apparent to those skilled in the art that the present invention requires an instruction stream to be compounded only once for a particular computer system configuration, and thereafter any fetch of compounded instructions will also cause a fetch of the identifier bits associated therewith. This avoids the need for the inefficient last-minute determination and selection of certain scalar instructions for parallel execution that repeatedly occurs every time the same or different instructions are fetched for execution in the so-called super scalar machine.

Despite all of the advantages of compounding a binary instruction stream, it becomes difficult to do so under certain computer architectures unless a technique is developed for determining instruction boundaries in a byte string. Such a determination is complicated when variable length instructions are allowed, and is further complicated when data and instructions can be intermixed. Of course, at execution time instruction boundaries must be known to allow proper execution. But since compounding is preferably done a sufficient time prior to instruction execution, a technique is needed to compound instructions without knowledge of where instructions start and without knowledge of which bytes are data. This technique needs to be applicable to all of the accepted types of architectures, including the RISC (Reduced Instruction Set Computers) architectures in which instructions are usually a constant length and are not intermixed with data.

There are a number of variations of the technique of the present invention, depending on the information that is already available about the particular instruction stream being compounded. The various combinations of typical pertinent information are shown below in Table 2:

TABLE 2

Byte String Information			
Case	Instruction Length	Data Intermixed	Reference Point
A	fixed	no	yes
B	variable	no	yes
C	fixed or variable	yes	yes

D	fixed	no	no
E	variable	no	no
F	fixed	yes	no
G	variable	yes	no

It is to be noted that in some instances fixed and variable length instructions are identified as being different cases. This is done because the existence of variable length instructions creates more uncertainty where no reference point is known, thereby resulting in the creation of many more potential compounding bits. In other words, when generating the potential instruction sequences as provided by the technique of this invention, there are no compounding identifier tags for bytes in the middle of any fixed length instructions. Also, the total number of identifier tags required under the preferred encoding scheme is fewer (i.e., one identifier tag for every four bytes for instructions having a fixed length of four bytes). Nevertheless, the unique technique of this invention works equally well with either fixed or variable length instructions since once the start of an instruction is known (or presumed), the length can always be found in one way or another somewhere in the instructions. In the System/370 instructions, the length is encoded in the opcode, while in other systems the length is encoded in the operands.

In case A with fixed length instructions having no data intermixed and with a known reference point location for the opcode, the compounding can proceed in accordance with the applicable rules for that particular computer configuration. Since the length is fixed, a sequence of scalar instructions is readily determined, and each instruction in the sequence can be considered as possible candidates for parallel execution with a following instruction. A first encoded value in the control tag indicates the instruction is not compoundable with the next instruction, while a second encoded value in the control tag indicates the instruction is compoundable for parallel execution with the next instruction.

Similarly in case B with variable length instructions having no data intermixed, and with a known reference point for the instructions (and therefore also for the instruction length code, the compounding can proceed in a routine manner. As shown in FIG. 8, the opcodes indicate an instruction sequence 70 as follows: the first instruction is 6 bytes long, the second and third are each 2 bytes long, the fourth is 4 bytes long, the fifth is 2 bytes long, the sixth is 6 bytes long, and the seventh and eighth are each 2 bytes long.

For purposes of illustration, the technique for compounding herein is shown for creating compound instructions formed from adjacent pairs of scalar instructions (FIGS. 8-10) as well as for creating compound instructions formed from larger groups of scalar instructions (FIG. 12). The exemplary rules for the embodiments shown in the drawings are additionally defined to provide that all instructions which are 2 bytes or 4 bytes long are compoundable with each other (i.e., a 2 byte instruction is capable of parallel execution in this particular computer configuration with another 2 byte or another 4 byte instruction). The rules further provide that all instructions which are 6 bytes long are not compoundable at all (i.e., a 6 byte instruction is only capable of execution singly by itself in this particular computer configuration). Of course, the invention is not limited to these exemplary compound rules, but is applicable to any set of compounding rules which define the criteria for parallel execution of existing instructions in a specific configuration for a given computer architecture.

The instruction set used in these exemplary compounding techniques of the invention is taken from the System/370 architecture. By examining the opcode for each instruction, the type and length of each instruction can be determined and the control tag containing identifier bits is then generated for that specific instruction, as described in more detail hereinafter. Of course, the present invention is not limited to any specific architecture or instruction set, and the aforementioned compounding rules are by way of example only.

The preferred encoding for compound instructions in these illustrated embodiments is now described. If two adjacent instructions can be compounded, their identifier bits which are generated for storage are "1" for the first compounded instruction and "0" for the second compounded instruction. However, if the first and second instructions cannot be compounded, the identifier bit for the first instruction is "0" and the second and third instruction are then considered for compounding. Once an instruction byte stream has been pre-processed in accordance with this technique and identifier bits encoded for the various scalar instructions, more optimum results for achieving parallel execution may be obtained by using a bigger window for looking at larger groups, and then picking the best combination of adjacent pairs for compounding.

A C-vector 72 in FIG. 8 shows the values for the identifier bits (called compounding bits in the drawings) for this particular sequence 70 of instructions where a reference point indicating the beginning of the first instruction is known. Based on the values of such identifier bits, the second and third instructions form a compounded pair as indicated by the "1" in the identifier bit for the second instruction. The fourth and fifth instructions form another compounded pair as indicated by the "1" in the identifier bit for the fourth instruction. The seventh and eighth instructions also form a compounded pair as indicated by the "1" in the identifier bit for the seventh instruction.

The C-vector 72 of FIG. 8 is also relatively easy to generate in case B when there are no data bytes intermixed with the instruction bytes, and where the instructions are all of the same length with known boundaries.

A slightly more complex situation is presented in case C where instructions are mixed with non-instructions, with a reference point still being provided to indicate the beginning of an instruction. The schematic diagram of FIG. 13 shows one way of indicating an instruction reference point, where every halfword has been flagged with a tag to indicate whether or not it contains the first byte of an instruction. This could occur with both fixed length and variable length instructions. By providing the reference point, it is unnecessary to evaluate the data portion of the byte stream for possible compounding. Accordingly, the compounding unit can skip over and ignore all of the non-instruction bytes.

Case D does not present a difficult problem with fixed length instructions having no data intermixed, since the instructions and data are typically aligned on predetermined byte boundaries. So although the table shows that the reference point is not known, in fact it is readily determined based on the alignment requirements.

Case E is a more complicated situation where a byte stream includes variable length instructions (without data), but it is not known where a first instruction begins. Since the maximum length instruction is six bytes, and since instructions are aligned on two byte boundaries, there are three possible starting points for the first instruction the the stream. Accordingly, the invention provides for considering all possible starting points for the first instruction in the text of a byte stream 79, as shown in FIG. 9.

Sequence 1 assumes that the first instruction starts with the first byte, and proceeds with compounding on that premise. The value in the length field for the first byte is 6 indicating the next instruction begins with the seventh byte; the value in the length field for the seventh byte is 2 indicating the next instruction begins with the ninth byte; the value in the length field for the ninth byte is 2 indicating the next instruction begins with the eleventh byte; the value in the length field for the eleventh byte is 4 indicating the next instruction begins with the fifteenth byte; the value in the length field for the fifteenth byte is 2 indicating the next instruction begins with the seventeenth byte; the value in the length field for the seventeenth byte is 6 indicating the next instruction begins with the twenty third byte; the value in the length field for the twenty third byte is 2 indicating the next instruction begins with the twenty fifth byte; and the value in the length field for the twenty fifth byte is 2 indicating the next instruction (not shown) begins with the twenty seventh byte.

In this exemplary embodiment, the length field is also determinative of the C-vector value for each possible instruction. Therefore a C-vector 74 for Sequence 1 only has a "1" value for the first instruction of a possible compounded pair formed by combinations of 2 byte and 4 byte instructions.

Sequence 2 assumes that the first instruction starts at the third byte (the beginning of the second halfword), and proceeds on that premise. The value in the length field for the third byte is 2 indicating the next instruction begins with the fifth byte. By proceeding through each possible instruction based on the length field value in the preceding instruction, the entire potential instructions of Sequence 2 are generated along with the possible identifier bits as shown in a C-vector 76.

Sequence 3 assumes that the first instruction starts at the fifth byte (the beginning of the third halfword), and proceeds on that premise. The value in the length field for the fifth byte is 4 indicating the next instruction begins with the ninth byte. By proceeding through each possible instruction based on the length field value in the preceding instruction, the entire potential instructions of Sequence 3 are generated along with the possible identifier bits as shown in a C-vector 78.

In some instances the three different Sequences of potential instructions will converge into one unique sequence. The rate of convergence depends on the specific bits which are in the potential opcode field reserved for the instruction length. In some instruction byte streams there will be no convergence found during compounding of a particular window (for example, a sequence of instructions in which all the lengths happen to be four bytes). In other instances, convergence to the same instruction boundaries could occur with the compounding sequence of two different sequences out-of-phase. However, out-of-phase convergence is always corrected by the next non-compoundable instruction, if not earlier.

In FIG. 9 it is noted that the three Sequences converge on instruction boundaries at the end 80 of the eighth byte It is also noted that if additional sequences started at the end of the sixth, eighth and tenth bytes, they would also converge quickly. Sequences 2 and 3, while converging on instruction boundaries at the end 82 of the fourth byte, are out-of-phase in compounding until the end of the sixteenth byte. In other words, the two sequences consider different pairs of instructions based on the same sequence of instructions. Since the seventeenth byte begins a non-compoundable instruction at 84, the out-of-phase convergence is ended. In a situation where each window of instructions being reviewed contains more than two instructions, the various sequences might have converged sooner because the two instruction compounders might have chosen the same optimum pairings.

When no valid convergence occurs, it is necessary to continue all three possible instruction sequences to the end of the window. However, where valid convergence occurs and is detected, the number of sequences collapses from three to two (one of the identical sequences becomes inoperative), and in some instances from two to one. Where multiple sequences of instructions must be considered due to the unknown instruction boundaries, the rate of compounding will be slower than the compounding of FIG. 8 by a factor equal to the number of active sequences (assuming a single unit compounding facility). If convergence is fast, the rate of compounding exemplified in FIGS. 8 and 9 will be virtually equivalent.

Thus, prior to convergence, tentative instruction boundaries are determined for each possible instruction sequence and identifier bits assigned for each such instruction indicating the location of the potential compound instructions. It is apparent from FIG. 9 that this technique generates three separate identifier bits for every two text bytes. In order to provide consistency with the pre-processing done in cases A-D, it is desirable to reduce the three possible sequences to a single sequence of identifier bits where only one bit is associated with each halfword. Since the only information needed is whether the current instruction is compounded with the following instruction, the three bits can be logically ORed to produce a single sequence in a CC-vector 86.

The various steps in the compounding method shown in FIG. 9 as described above are illustrated in the flow chart of FIG. 16.

For purposes of parallel execution, the composite identifier bits of a composite CC-vector are equivalent to the separate C-vectors of the individual three Sequences 1-3. This can be shown by referring to the CC-vector 86 in FIG. 9. Proceeding with Sequence 1, if the first byte is considered for execution either because of conventional sequential processing or by branching, the instruction is fetched along with its associated identifier bits. Since the identifier bit is "0", the first instruction is executed serially as a single instruction. The identifier bits associated with the third and fifth bytes are ignored. The next instruction in Sequence 1 begins at the seventh byte, so such instruction is fetched by the CPU along with its identifier bit which is "1". Since this indicates the beginning of a compound instruction, the next instruction is also fetched (its identifier bit "1" in the CC-vector 86 is ignored, so the fact that its identifier bit in the C-vector 74 is different is of no consequence) for parallel execution with the instruction which begins at the seventh byte. So the CC-vector 86 works satisfactorily for Sequence 1 if it turns out to be an actual instruction sequence.

Proceeding with Sequence 2, if the third byte is considered for execution either because of conventional sequential processing or by branching, the instruction is fetched along with its associated identifier bits. Since the identifier bit is "1" and indicates the beginning of a compound instruction, the next instruction is also fetched (its identifier bit "1" in the CC-vector 86 is ignored, so the fact that its identifier bit in the C-vector 76 is different is of no consequence) for parallel execution with the instruction which begins at the third byte. So the CC-vector 86 also works satisfactorily for Sequence 2 if it turns out to be an actual instruction sequence.

Proceeding with Sequence 3, if the fifth byte is considered for execution either because of conventional sequential processing or by branching, the instruction is fetched along with its associated identifier bits. Since the identifier bit is "1" and indicates the beginning of a compound instruction, the next instruction is also fetched (its identifier bit "1" in the CC-vector 86 is ignored, so the fact that its identifier bit in the C-vector 78 is different is of no consequence) for parallel execution with the instruction which begins at the fifth byte. So the CC-vector also works satisfactorily for Sequence 3 if it turns out to be an actual instruction sequence.

Thus the composite identifier bits in the CC-vector allow any of the three possible sequences to execute properly in parallel for compound instructions or singly for non-compounded instructions. The composite identifier bits also work properly for branching. For example, if a branch to the beginning 88 of the ninth byte occurs, then the ninth byte must begin an instruction. Otherwise there is an error in the program. The identifier bit "1" associated with the ninth byte is used and correct parallel execution of such instruction with its next instruction proceeds.

One beneficial advantage provided by the composite identifier bits in the CC-vector is the creation of multiple valid compounding bit sequences based on which instruction is addressed by a branch target. As best shown in FIGS. 14-15, differently formed compounded instructions are possible from the same byte stream.

FIG. 14 shows the possible combinations of compounded instructions when the computer configuration provides for parallel issuance and execution of no more than two instructions. Where an instruction stream 90 containing compounded instructions is processed in normal sequence, the Compound Instruction I will be issued for parallel execution based on decoding of the identifier bit for the first byte in a CC-vector 92. However, if a branch to the fifth byte occurs, the Compound Instruction II will be issued for parallel execution based on decoding of the identifier bit for the fifth byte.

Similarly, a normal sequential processing of another compounded byte stream 94 will result in Compound Instructions IV, VI and VIII being sequentially executed (the component instructions in each compound instruction being executed in parallel). In contrast, branching to the third byte in the compounded byte stream will result in Compound Instructions V and VII being sequentially executed, and the instruction beginning at the fifteenth byte (it forms the second part of Compound Instruction VIII) will be issued and executed singly, all based in the identifier bits in the CC-vector 96.

Branching to the seventh byte will result in Compound Instructions VI and VIII being sequentially executed, and branching to the eleventh byte will result in Compound Instruction VIII being executed. In contrast, branching to the ninth byte in the compounded byte stream will result in Compound Instruction VII being executed (it is formed by the second part of Compound Instruction VI and the first part of Compound Instruction VIII).

Thus, the identifier bits "1" in the CC-vector 96 for Compound Instructions IV, VI and VIII are ignored when either of the Compound Instructions V or VII is being executed. Alternatively the identifier bits "1" in the CC-vector 96 for Compound Instructions V and VII are ignored when any of Compound Instructions IV, VI or VIII are executed.

FIG. 15 shows the possible combinations of compounded instructions when the computer configuration provides for parallel issuance and execution of up to three instructions. Where an instruction stream 98 containing compounded instructions is processed in normal sequence, the Compound Instructions X (a triplet group) and XIII (a pair group) will be executed. In contrast, branching to the eleventh byte will result in Compound Instruction XI (a triplet group) being executed, and branching to the thirteenth byte will result in Compound Instruction XII (a different triplet group) being executed.

Thus, the identifier bits "2" in a CC-vector 99 for Compound Instructions XI and XII are ignored when Compound Instructions X and XIII are executed. On the other hand when Compound Instruction XI is executed, the identifier bits for the other three Compound Instructions X, XII, XII are ignored. Similarly when Compound Instruction XII is executed, the identifier bits for the other three Compound Instructions X, XI, XIII are ignored.

Case G is the most complex case which deals with an instruction stream having data intermixed with variable length instructions, without any known reference point for the beginning of any instruction. This could occur when compounding a page in memory or in an instruction cache when a reference point is not known. The first embodiment (not shown) for dealing with Case G is identical to the one used for Case E, but there is an additional distinction because of the fact that data is intermixed with the instructions. If convergence occurs, a new sequence must always be started in place of each sequence eliminated by convergence. This is because convergence could occur in a byte containing data; consequently all three compounding sequences could converge to a spurious sequence of "instructions" which are in fact not instructions at all. This would eventually be corrected when a sequence of real instructions is encountered by one of the sequences. But in the meantime some compoundable instructions might not be detected. The resulting compounded instruction stream would still execute correctly, but fewer compounded instruction pairs would be tagged for parallel execution, and therefore CPU performance would decrease.

The preferred technique for dealing with Case G is shown in FIG. 10 for the same byte stream 79 as in FIG. 9. A new sequence of possible instructions is started at every halfword regardless of the values in the instruction length portion of the potential opcode field. As with the other cases, two adjacent potential instructions are examined and the appropriate identifier bits for various C-vectors 100 are determined. This is repeated starting two bytes (one halfword) later. As with the Case E, the various C-vector values for the same halfword are ORed together (See FIG. 11) to form the composite identifier bits of the related composite CC-vector 102. It is to be noted that in this particular embodiment where the compounder identifies a compound instruction by producing a "1" for the first byte only, and where in FIG. 10 each potential sequence is only two instructions in length, the output that results from examining each sequence using the preferred encoding scheme for two-way compounding is a single bit. Accordingly, to form the CC-vector 102 in this instance, all of the first identifier bits in in each sequence are concatenated, thereby producing the same CC-vector as would result in the general case of ORing the various C-vector values.

If a byte is selected for execution, it must in fact be an instruction if the program is correct, and the appropriate CC-vector identifier bit associated with that byte is checked to see if the byte is the beginning of a compound instruction. The tags associated with data will always be ignored during execution of actual instructions--both scalar instructions executed singly and compounded instructions executed in parallel.

If a branch instruction is compounded with data, the branch must be taken (assuming a correct program) and the second instruction in the pair which would have been executed in parallel, if the branch was not taken, is nullified. This capability must already be present in the execution unit if branches can be executed concurrently with a following instruction in a pipelined fashion.

It is important to note that the composite compounding sequences in the CC-vectors 88, 102 in FIGS. 9 and 10 are not the same, even though the text is identical. Since in FIG. 9 it is known that the text contains no data intermixed with the instructions, convergence results in a known reference point. The extra "1" values in the CC-vector 102 for FIG. 10 occur after the reference point is known in FIG. 9 and such extra "1"s do not correspond to halfwords that begin instructions because they account for the possibility of data being present in the text. However, if the text contains instructions only, as is assumed in the technique for Case E shown in FIG. 9, the different composite sequences in the two CC-vectors 88, 102 will nevertheless result in identical program execution in accordance with the advantages of the invention.

Case F involving fixed length instructions intermixed with data, and having no instruction reference point, is a simplified version of Case G. If the instructions are two bytes long aligned on halfword boundaries, then potential instruction sequences are started every halfword, and it is not necessary to use the instruction length to generate the potential sequences.

The worst case technique of FIG. 10 for dealing with Case G examines more possible instruction sequences than the techniques for Cases A-F. This may require more time and/or more compounding units to produce the necessary identifier bits in the tags, depending on the implementation.

There are many possible designs for an instruction compounding unit depending on its location and the knowledge of the text contents. In the simplest situation, it would be desirable for a compiler to indicate with tags which bytes contain the first byte of an instruction and which contain data. This extra information results in a more efficient compounder since exact instruction locations are known (see FIG. 13). This means that compounding could always be handled as Case C situations in order to generate the C-vector identifier bits for each compound instruction (See FIG. 8). A compiler could also add other information such as static branch prediction or even insert directives to the compounder.

Other ways could be used to differentiate data from instructions where the instruction stream to be compounded is stored in memory. For example, if the data portions are infrequent, a simple list of addresses containing data would require less space than tags. Such combinations of a compounder in hardware and software provide many options for efficiently producing compound instructions.

FIG. 11 shows a flow diagram of a possible implementation of a compounder for handling instruction streams in either the Case E, F or G category. A multiple number of compounder units 104, 106, 108 are shown, and for efficiency purposes this number could be as large as the number of halfwords that could be held in a text buffer. In this version, as applied to Case G, the three compounder units could begin their processing sequences at the first, third, and fifth bytes, respectively. Upon finishing with a possible instruction sequence, each compounder starts examining the next possible sequence offset by six bytes from its previous sequence. Each compounder produces compound identifier bits (C-vector values) for each halfword in the text. The three sequences from the three compounders are ORed 110 and the resulting composite identifier bits (CC-vector values) are stored in association with their corresponding textual bytes.

FIG. 12 shows how the worst case compounding technique for Case G is applied to large groups such as up to four instructions in each compound instruction. Considering again the same byte stream 79, each byte at the beginning of a halfword is examined as if it were the beginning of an instruction and its opcode evaluated to locate a potential sequence of three additional instructions. If it cannot be compounded, its identifier bit value is "0". If it can be compounded with the next potential instruction, the identifier bits are "1" for the first instruction in the pair and "0" for the second instruction in the pair. If it turns out that it can be compounded with the next two potential instructions, the compounding bits beginning with the first instruction are "2", "1" and "0", respectively. This method assumes that a branch to the middle of a large group compound instruction can execute the triplet or pair group which are a tail-end subset of the large group.

As with FIG. 10, the bytes beginning at each halfword must be examined to locate potential instruction boundaries. Each examined sequence produces a sequence of identifier bits called C-vectors 112. A composite sequence of identifier bits called CC-vector values 114 is formed by taking the maximum value of all the individual identifier bits associated with that halfword. When a large group compound instruction is issued and executed, the CPU ignores all compound bits associated with bytes other than the first byte of the group. In this method of encoding, the compound identifier bits in the CC-vector 114 indicate the beginning of a compound instruction as well as indicate the number of instructions constituting the compound instruction.

Depending on the actual compounding rules used, there may be some optimizations for this particular large group compounding technique. For example, the fifth sequence starting at the ninth byte 116 assumes instructions of lengths 2, 4, 2 and 6 bytes long. Since 6-byte instructions are never compoundable in this example, there is no benefit in attempting to compound starting at the other three potential instructions (eleventh, fifteenth and seventeenth bytes) since they have already been compounded as much as possible. In that regard, the identifier bits for potential instructions beginning at the eleventh and fifteenth bytes are already indicated in the C-vector 112 at 118, 120, respectively. On the presumption that the ninth byte begins an instruction sequence at 116, the thirteenth byte does not begin an instruction. However, this optimization just described still requires the thirteenth byte to be examined at 122 as the beginning of a possible instruction, since it has not been previously considered.

Of course, the large group compound method would continue with all of the halfwords in the text, even though the illustrated example of FIG. 12 stops with the fifteenth byte.

In order to reduce the number of bits to transfer, there may be alternative representations of the compounding information. For example, the compounding identifier bits could be translated into a different format once a true instruction boundary is determined. For example, it is possible to achieve one bit per instruction with the following encoding: the value "1" means to compound with the next instruction, and the value "0" means to not compound with the next instruction. A compound instruction formed with a group of four individual instructions would have a sequence of compounding identifier bits (1,1,1,0). As with the execution of other compound instructions previously described, compounding identifier bits associated with halfwords which are not instructions and therefore do not have any opcodes are ignored at execution time.

While exemplary preferred embodiments of the invention have been disclosed, it will be appreciated by those skilled in the art that various modifications and changes can be made without departing from the spirit and scope of the invention as defined by the following claims.

* * * * *

United States Patent — ***5,448,746***
Eickemeyer , et al. — **September 5, 1995**

System for comounding instructions in a byte stream prior to fetching and identifying the instructions for execution

Abstract

A system with an apparatus that can be used in the compounding of instructions for CISC architectures and architectures with other attributes, including RISC. The compounding is performed before instruction execution and it results in a compound instruction program that can be executed in a parallel fashion on appropriate instruction execution hardware. In particular, the proposed apparatus provides compounding capability for architectures that allow the intermingling of instructions and data, contain variable length instructions, and allow modifications of the instruction stream. The system provides for differing and partial reference point information. An embodiment of the proposed apparatus handles the worst-case situation when it is not known which text bytes are instructions and which are data. If some information is known, the system can be simplified. The apparatus as presented provides compounds capability for any number of instructions. The system is developed particularly for machines with a S/370 instruction set, for which a number of examples are given. A backward compounding apparatus is provided. Multiple compound units and logical ORing of sequences provides system support for more difficult organizations.

Inventors:	**Eickemeyer; Richard J.** (Binghamton, NY); **Vassiliadis; Stamatis** (Vestal, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**186218**
Filed:	**January 25, 1994**

Current U.S. Class:	**712/210**; 712/213; 712/216
Intern'l Class:	G06F 009/38
Field of Search:	395/800,775,575,425,375

References Cited [Referenced By]

U.S. Patent Documents

4295193	Oct., 1981	Pomerene	395/375.
4439828	Mar., 1984	Martin	395/375.
4594655	Jun., 1986	Hao et al.	395/775.
4780820	Oct., 1988	Sowa	395/800.
4847755	Jul., 1989	Morrison et al.	395/650.
4975837	Dec., 1990	Woodnard et al.	395/375.
5050068	Sep., 1991	Dollas et al.	395/375.
5086408	Feb., 1992	Sakator	395/600.
5121493	Jun., 1992	Ferguson	395/600.
5197137	Mar., 1993	Kumar et al.	395/375.
5201057	Apr., 1993	Uht	395/375.
5203002	Apr., 1993	Wetzel	395/375.

Other References

Acosta, R. D., et al., "an Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al, The IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al, ed (McGraw-Hill, 1982, pp. 276-292.
Capozzi, A. J., et al, "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al, "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al, "SIMP (single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M. D., et al, "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.

Jouppi, N. P., et al, "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N. P., "the Non-Uniform Distritubion of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp. 63-76.
Colwell, R. P., et al, "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the CRISP Instruction Set Architecture", Proceedings of Compcon, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al, "Compiling for the CRISP Microprocessor", Proceedings of Compcon, Spring, 1987, pp. 96-100.
Hennessy, J., et al, "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May. 1983, pp. 237-246.
Ditzel, D. R., et al, "Branch Folding in the Crisp Microprocessor: Reducing Branch Delay to Zero", Proceedings of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W., et al, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al, "Branch Prediction Strategies in Branch Target Buffer Design", IEEE computer, vol. 17, No. 1, Jan. 1984, pp. 6-22.
Riseman, E. M., "the Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al, Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al. "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al, "a Study of Instruction Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun., 1983, pp. 132-137.
Vassiliadis, S., et al, "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.

The Architecture of Pipelined Computers, by Peter M. Kogge hemisphere Publishing Corporation, 1981.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard.

Primary Examiner: bowler; Alyssa H.
Assistant Examiner: Shah; Alpesh M.
Attorney, Agent or Firm: Augspurger; Lynn L., Marhoefer; Laurence J.

Parent Case Text

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of application Ser. No. 07/677,066, filed Mar. 29, 1991, now abandoned, which claims priority and U.S. application continuation-in-part status continuing U.S. Ser. No. 07/519,384 filed May 4, 1990, now abandoned, and U.S. Ser. No. 07/543,458 filed Jun. 26, 1990, now U.S. Pat. No. 5,197,135, and U.S. Ser. No. 07/642,011 filed Jan. 15, 1991, now U.S. Pat. No. 5,295,249 and U.S. Ser. No. 07/519,382 filed May 4, 1990, now abandoned.

Claims

What we claim is:

1. A system for compounding instructions in a byte stream including instructions which cannot be compounded prior to fetching and identifying said instructions for execution, said instructions being compounded in accordance with compounding rules for classes of instructions that can be compounded for a particular processor architecture, said system comprising:

means for examining the byte stream in sets of a fixed length of bits assigned for an opcode as an instruction length code and proceeding sequentially through said byte stream from one instruction to the next,

means to identify a boundary of each instruction in said byte stream in response to said means for examining,

means for applying said compounding rules to instructions whose boundaries have been identified by said identification means; and

means responsive to said means for applying said compounding rules, for generating tag bit information for an instruction which can be compounded to indicate an initial instruction of a set of compoundable instructions;

said means for applying said compounding rules marking an instruction which cannot be compounded to indicate it is not an initial instruction of a set of compoundable instructions and examining a subsequent group of instructions which includes the next sequential instruction to an introduction marked as an instruction which cannot be compounded.

2. A system according to the preceding claim 1 in which a halfword in an instruction is marked to indicate that the halfword is not an initial compound instruction.

3. A system according to the preceding claim 1 in which instructions are examined for pairs.

4. A system according to the preceding claim 1 in which the examination for pairs looks at larger groups than adjacent instruction opcodes, and a best pair is picked for compounding according to said compounding rules.

5. A system according to the preceding claim 1 in which instructions to be examined are mixed with non-instructions or data and wherein said generating means marks every halfword either as containing the first byte of an instruction or as an instruction not containing the first byte of a compound instruction, and output means for continuing the process of compounding which skips over in the compounding process the instruction marked as not containing the first byte of a compound instruction.

6. A system according to claim 1 wherein in the event it is not known where in the instruction set stream being examined are bounded with a plurality of possible sequences of instructions, determination means are provided for each sequence possible boundaries to be determined, and logical means are provided for reducing plural sequences to a single sequence by computing the maximum of said plurality of possible sequences.

7. A system according to the preceding claim 6 wherein there are three possible sequences of instructions with potential boundaries, each of which would produce a different sequence of compounding bits, and wherein the three bit sequences are logically ORed by said logical OR means to produce a single sequence to determine whether the current instruction is compounded by the instruction processor.

8. A system according to the preceding claim 6 wherein said examination means examines a set of instructions and compounding bits are determined for a sequence by said determination means, and this examination and determination technique is repeated starting a number of bytes in sequence later, and said logical OR means is coupled for compounding bits from each sequence to be ORed to form a composite compound bit instruction.

9. A system according to claim 6 wherein in the event it is not known where in the base instruction set stream being examined are bounded with a plurality of possible sequences of instructions, determination means are provided for each sequence possible boundaries to be determined, and logical OR means are provided for reducing plural sequence to a single sequence by logically ORing said plurality of possible sequences.

10. A system according to the preceding claim 1 wherein there are several possible sequences of instructions with potential boundaries, each of which would produce a different sequence of compounding bits, and wherein there is provided a convergence means for compounding which experimentally tries the several possible sequences for convergence on a byte boundary, and when a

convergence is detected, one of the converging sequences is eliminated.

11. A system according to the preceding claim 10 wherein there is provided collapsing means for collapsing the number of sequences until a single sequence remains.

12. A system according to the preceding claim 10 wherein in the case when the byte stream of incoming information of the instruction sequence may have mixed instructions and data and wherein there is no knowledge of where any instructions begin, if a convergence occurs and a sequence is eliminated by convergence, said convergence means includes means for starting a new sequence in place of a sequence eliminated by convergence.

13. A system according to the preceding claim 12 wherein each sequence is started at every halfword.

14. A system according to the preceding claim 10 wherein in the case when the byte stream of incoming information of the instruction sequence being processed by the compounding facility may have mixed instructions and data and wherein there is no knowledge of where any instructions begin, if a convergence occurs and a sequence is eliminated by convergence, a new sequence is started in place of a sequence eliminated by convergence.

15. A system according to the preceding claim 14 wherein each sequence is started at every halfword.

16. A system according to the preceding claim 14 wherein after one group of a set of instructions are examined and compounding bits are determined, the examination means processes for examination again with a group repeated starling a number of bytes in sequence later, and compounding bits from each sequence are ORed to form a composite compound bit instruction.

17. A system according to the preceding claim 1 wherein the compound bits of an instruction indicate the number of instructions which make up a compound group instruction.

18. A system according to the preceding claim 1 in which there are a plurality of member units for a compound instruction and information is provided for the compound instruction applicable to a member instruction unit of the compound instruction indicating whether or not it is to be compounded with a subsequent instruction.

19. A system according to the preceding claim 1 wherein the system instruction processor central processor during execution ignores all but tag information of a first instruction member of a compound instruction.

20. A system according to the preceding claim 1 wherein during examination bytes not executed are marked as not examined in which case they will be reexamined could the code be later executed by the instruction processor.

21. A system according to the preceding claim 1 wherein during examinations bytes not executed are left in an examined state, indicating the correctness of location even though they may be ignored or not executed during execution.

22. A system according to the preceding claim 1 wherein during examination bytes are left in a partially examined sequence state for examination of other sequences later.

23. A system according to claim 1 wherein to reduce the number of bits to transfer, there can be alternative representations of the compound information, and wherein when a compound instruction is requested the compound bits can be be translated into a different format.

24. A system according to claim 23 wherein one bit per instruction with the encoding of a value which means compound with next instruction and no value 0 which means do not compound with next instruction.

25. A system for processing instructions by a target instruction processor in which certain component instructions of a base instruction sequence can be executed in parallel and certain component instructions in said base instruction sequence cannot be executed in parallel and for compounding the base instruction sequence for execution in parallel of at least some of the base instructions, comprising,

an instruction processor having a central processing unit, a cache, a memory,

a compounding facility for said instruction processor disposed between said memory and said cache for compounding instructions prior to fetching and identifying said instructions for execution,

said compounding facility including a plurality of compounding units which enable the system to process an incoming sequence of scalar instructions and to transform them into a compound instruction program in which at least some compound member units of a compound instruction are executable in parallel by said target instruction processor,

each of said compounding units including:

means for determining applicable architectural compounding rules in which classes of instructions which may be compounded for a particular architecture,

means for examining a base instruction sequence byte stream in sets of bits for an opcode and for proceeding sequentially through the base instruction byte stream from one instruction to the next,

means to identify a boundary of an instruction unit member of said compound instruction in said base instruction byte stream in response to said means for examining,

means for applying said compounding rules to instructions whose boundaries have been identified by said identification means; and

generating means responsive to said means for applying said compounding rules, for generating tag bit information for an instruction which tag bit information indicates an initial compound instruction of a set of compoundable instructions, and if an initial instruction of a set is not an initial compoundable instruction, marking that instruction so that an output means will not indicate to an executing instruction processor that it is an initial instruction, and as such instructions are marked as not initial instructions, said means for examining examines a subsequent group of instructions which includes a next sequential instruction in the group which had just been examined.

26. A system according to the preceding claim 25 wherein in said compounding facility there are means for a cache miss or a branch to the part of a line not yet compounded such that output means for providing a compound sequence of compound instructions provides a plurality of instructions to the central processing unit via said instruction program without compounding of the instructions.

27. A system according to the preceding claim 25 wherein should the compounding facility examine a byte stream and encounter data, the compounding facility output means will compound data as if the data consisted of instructions, and wherein an instruction processor which cannot execute data will accordingly execute a compound instruction program correctly.

28. A system according to the preceding claim 25 in which halfwords which exist in an instruction are initialized by said generating means to indicate that the halfword is not an initial compound instruction.

29. A system according to the preceding claim 25 in which instructions are examined for pairs.

30. A system according to the preceding claim 29 in which examination for pairs by said examination means looks at larger groups than adjacent instruction opcodes of a base instruction byte stream, and optimization means are provided such that a best pair is picked for compounding according to said compounding rules.

31. A system according to the preceding claim 25 in which instructions are mixed with non-instructions or data and instruction members of a compound instruction are marked every halfword either as containing the first byte of an instruction or as an instruction not containing the first byte of a compound instruction, and said output means causes skipping over in the compounding process the instruction marked as not containing the first bite of a compound instruction.

32. A system according to claim 25 wherein in the event it is not known where in the base instruction set stream being examined are bounded with a plurality of possible sequences of instructions, determination means are provided for each sequence possible boundaries to be determined, and logical OR means are provided for reducing plural sequence to a single sequence by logically ORing said plurality of possible sequences.

33. A system according to the preceding claim 32 wherein there are three possible sequences of instructions with potential boundaries, each of which would produce a different sequence of compounding bits, and wherein the three bit sequences are logically ORed by said logical OR means to produce a single sequence to determine whether the current instruction is compounded by the instruction processor.

34. A system according to the preceding claim 32 wherein said examination means examines a set of instructions and compounding bits are determined for a sequence by said determination means, and this examination and determination technique is repeated starting a number of bytes in sequence later, and said logical OR means is coupled for compounding bits from each sequence to be ORed to form a composite compound bit instruction.

35. A system according to the preceding claim 25 wherein there are several possible sequences of instructions with potential boundaries, each of which would produce a different sequence of compounding bits, and wherein there is provided a convergence means for compounding which experimentally tries the several possible sequences for convergence on a byte boundary, and when a convergence is detected, one of the converging sequences is eliminated.

36. A system according to the preceding claim 35 wherein there is provided collapsing means for collapsing the number of sequences until a single sequence remains.

37. A system according to the preceding claim 35 wherein in the case when the byte stream of incoming information of the base instruction sequence may have mixed instructions and data and wherein there is no knowledge of where any instructions begin, if a convergence occurs and a sequence is eliminated by convergence, said convergence means includes means for starting a new sequence in place of a sequence eliminated by convergence.

38. A system according to the preceding claim 37 wherein each sequence is started at every halfword.

39. A system according to the preceding claim 25 wherein a plurality of compounding units are provided and a set of instructions is examined and compounding bits are determined and a sequence is examined by said plurality of separate compounding units each of which has an output which is combined as a composite compound vector.

40. A system according to the preceding claim 39 wherein a logical OR unit is provided coupled to said plurality of compounding units and a set of instructions is examined and a composite compound vector created based upon the output of said logical OR unit and said compounding units by said output means.

41. A system according to the preceding claim 25 wherein two instruction member units of a compound instruction form a pair of instructions which can be executed in parallel by the instruction processor.

42. A system according to the preceding claim 25 in which instructions are identified during a process of fetching instructions from memory, wherein upon retrieval and before examination they are marked as instructions and not data.

43. A system according to the preceding claim 25 wherein the compound bits of an instruction indicate the number of instructions which make up a compound group instruction.

44. A system according to the preceding claim 25 in which there are a plurality of member units for a compound instruction and information is provided for the compound instruction applicable to a member instruction unit of the compound instruction indicating whether or not it is to be compounded with a subsequent instruction.

45. A system according to the preceding claim 25 wherein the system instruction processor central processor during execution ignores all but tag information of a first instruction member of a compound instruction.

46. A system according to the preceding claim 25 wherein during examination bytes not executed are marked as not examined in which case they will be reexamined could the code be later executed by the instruction processor.

47. A system according to the preceding claim 25 wherein during examinations bytes not executed are left in an examined state, indicating the correctness of location even though they may be ignored or not executed during execution.

48. A system according to the preceding claim 25 wherein during examination bytes are left in a partially examined sequence state for examination of other sequences later.

49. A system according to the preceding claim 25 where there is both forward compounding and backwards compounding.

50. A system according to the preceding claim 25 where there is provided with said compounding facility the control means to provide multiple modes of operation depending on the reference point information available.

Description

The present U.S. patent application is related to the following co-pending U.S. patent applications:

(1) Application Ser. No. 07/519,382 (IBM Docket EN9-90-020), filed May 4, 1990, now abandoned, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al; and

(2) Application Ser. No. 07/519,384 (IBM Docket EN9-90-019), filed May 4, 1990, now abandoned, entitled "General Purpose Compound Apparatus For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al; and

(3) Application Ser. No. 07/504,910 (IBM Docket EN9-90-014), filed Apr. 4, 1990, now U.S. Pat. No. 5,051,940, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al; and

(4) Application Ser. No. 07/522,219 (IBM Docket EN9-90-012), filed May 10, 1990, now U.S. Pat. No. 5,035,375, entitled "Compounding Preprocessor For Cache", the inventors being Bartholomew Blaner et al; and

(5) Application Ser. No. 07/543,464 (IBM Docket EN9-90-018), filed Jun. 26, 1990, now abandoned, entitled "An In-Memory Preprocessor for a Scalable Compound Instruction Set Machine Processor", the inventors being Richard Eickemeyer et al; and

(6) Application Ser. No. 07/543,458 (IBM Docket EN9-90-042), filed Jun. 26, 1990, now U.S. Pat. No. 5,197,135, entitled "Memory Management for Scalable Compound Instruction Set Machines With

In-Memory Compounding", the inventors being Richard J. Eickemeyer et al; and

(7) Application Ser. No. 07/619,868 (IBM Docket EN9-90-033), filed Nov. 28, 1990, now U.S. Pat. No. 5,301,341, entitled "Overflow Determination for Three-Operand ALUS in a Scalable Compound Instruction Set Machine", the inventors being Stamatis Vassiliadis et al; and

(8) Application Ser. No. 07/642,011 (IBM Docket EN9-90-049), filed Jan. 15, 1991, now U.S. Pat. No. 5,295,249, entitled "Compounding Preprocessor for Cache", the inventors being Stamatis Vassiliadis et al; and

(9) Application Ser. No. 07/677,685 (IBM Docket EN9-90-040), filed Mar. 29, 1991, now U.S. Pat. No. 5,303,356, entitled "System for Preparing Instructions for Instruction Processor and System With Mechanism for Branching in the Middle of a Compound Instruction", the inventors being S. Vassiliadis et al.

These co-pending applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y. The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

A review of these related cases will show that we have illustrated FIGS. 1-2 in U.S. Ser. No. 07/519,384 filed May 4, 1990, now abandoned, while FIG. 3 may also be found described in U.S. Ser. No. 07/543,458 filed Jun. 26, 1990, now U.S. Pat. No. 5,197,135 and FIG. 4-B has been generally described in U.S. Ser. No. 07/642,011 filed Jan. 15, 1991, now U.S. Pat. No. 5,295,249 while FIGS. 5-7 were also illustrated in U.S. Ser. No. 07/519,382 filed May 4, 1990, now abandoned.

FIELD OF THE INVENTION

This invention relates to digital computers and digital data processors, and particularly to digital computers and data processors capable of executing two or more instructions in parallel, and details the system and techniques for compounding instructions for a variety of instruction processors, including processors which use an architecture which processes instructions and data separately, processors which allow instructions and data to be mixed, those with and without reference indications identifying instructions and provides a backward compounding apparatus for compounding instructions which can be of a variable word length. In addition, compounding for pair and more extensive multiple instructions are provided.

BACKGROUND OF THE INVENTION

Traditional computers which receive a sequence of instructions and execute the sequence one instruction at a time are known. The instructions executed by these computers operate on single-valued objects, hence the name "scalar" for these computers.

The operational speed of traditional scalar computers has been pushed to its limits by advances in circuit technology, computer mechanisms, and computer architecture. However, with each new generation of competing machines, new acceleration mechanisms must be discovered for traditional scalar machines.

A recent mechanism for accelerating the computational speed of uni-processors is found in reduced instruction set architecture that employs a limited set of very simple instructions. Another acceleration mechanism is complex instruction set architecture which is based upon a minimal set of complex multi-operand instructions. Application of either of these approaches to an existing scalar computer would require a fundamental alteration of the instruction set and architecture of the machine. Such a far-reaching transformation is fraught with expense, downtime, and an initial reduction in the machine's reliability and availability.

In an effort to apply to scalar machines some of the benefits realized with instruction set reduction, so-called "superscalar" computers have been developed. These machines are essentially scalar machines whose performance is increased by adapting them to execute more than one instruction at a time from an instruction stream including a sequence of single scalar instructions. These machines typically decide at instruction execution time whether two or more instructions in a sequence of scalar instructions may be executed in parallel. The decision is based upon the operation codes (OP codes) of the instructions and on data dependencies which may exist between instructions. An OP code signifies the computational hardware required for an instruction. In general, it is not possible to concurrently execute two or more instructions which utilize the same hardware (a hardware dependency) or the same operand (a data dependency). These hardware and data dependencies prevent the parallel execution of some instruction combinations. In these cases, the affected instructions are executed serially. This, of course, reduces the performance of a super scalar machine.

Superscalar computers suffer from disadvantages which it is desirable to minimize. A concrete amount of time is consumed in deciding at instruction execution time which instructions can be executed in parallel. This time cannot be readily masked by overlapping with other machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Also, the parallel execution decision must be repeated each time the same instructions are to be executed.

In extending the useful lifetime of existing scalar computers, every means of accelerating execution is vital. However, acceleration by means of reduced instruction set architecture, complex instruction set architecture, or superscalar techniques is potentially too costly or too disadvantageous to consider for an existing scalar machine. It would be preferred to accelerate the speed of execution of such a computer by parallel, or concurrent, execution of instructions in an existing instruction set without requiring change of the instruction set, change of machine architecture, or extension of the time required for instruction execution.

SUMMARY OF THE INVENTION

Here and in the referenced applications we have detailed inventions which relate to compounding of instructions for instruction processors with the premise that these systems and techniques would allow enhancement and further development of existing processors, while enabling the further enhanced processors to execute more efficiently code which had been developed for their predecessors. However, while we will highlight the background of these developments as detailed herein, it should be recognized that in all of these developments described elsewhere by ourselves and others there is, without the benefit of the inventions described herein, an unfulfilled need to determine in the processing of a set of instructions or program to be executed by a computer how instructions may be combined into compound instructions for various possible architectures to which the inventions are applicable. Today there are many kinds of architectures, including RISC and S/370, and some have data mixed with instructions, some don't, some have variable length instructions, some don't. Some future machine may process only two parallel instructions while others can process longer instructions. These differences require solution to a number of problems which have been solved by the inventions addressed and detailed herein.

It is to this object that the inventions which are here stated are addressed as a system which enables original programs to be processed as parallel and single instructions as fits the original program functions which are implemented for execution by a machine capable of instruction parallel processing. We have provided a way for existing programs written in existing high level languages or existing assembly language program to be processed by a pre-processor which can identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions as instructions in parallel.

The instruction processor provides compounding decoding for a series of base instructions of a scalar machine, generates a series of compound instructions, provides for fetching of the compound instructions and for decoding the fetched compound instructions and necessary single instructions, and provides a compound instruction program which preserves intact the scalar execution of the base instructions of a scalar machine when the program with compound instructions is executed on the system. This system also provides a mechanism for branching in the middle of a compounding instruction. For branching in the middle of a compound instruction, compound instructions are examined for appended control bits. A nullification mechanism is provided for nullifying execution of an instruction unit of a compound instruction which would affect the correctness if there were branching in the middle of the compound instruction which would wrongly affect a recorded result dependent upon the interrelationship of the program.

The inventions detailed herein teach in what manner there shall be accomplished the appending of control information to the instructions in the program to be executed where the base architecture has different attributes. It is to this object that the inventions which are here stated are addressed. We have provided a way for existing programs written in existing high level languages or existing assembly language program to be processed by software which can identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions.

While the base apparatus for performing the compounding disclosed herein has been detailed in the aforesaid co-pending applications which relate to scalable compound instruction set machines, difficult problems exist in implementing compounding on systems. After detailing the base system which we prefer in the detailed description we will describe herein the advances interrelated the system hardware and compounding facility so as to make it applicable to a wide variety of systems, one of which is the IBM s/370 architecture which may act as an originating base instruction processor which then may be used together with successors having overlapping instruction sets within the architecture. Thus the newer instruction processor may have a particular set of compounding rules, as is contemplated in connection with the present system. The problems which we address for general compounding deal with the manner in which incoming instructions and data are processed as a continuing byte stream examined by the compounding facility which creates a compound instruction set program for running on an instruction processor. In order to solve some problems which may be encountered we have created a new system hardware here detailed for handling architectures with different attributes and those processing text with data, and having differing and partial reference point information with the apparatus which we will describe in detail herein.

We describe in this summary the kinds of interrelationships we discovered and dealt with in creating the new system which we describe in the detailed description which follows. Our work has resulted in solutions for systems which now use different architectures.

One such system may be a system where a byte string to be compounded is such that it is known that there are no data bytes in the instruction string, and when the location of a reference point, such as the opcode of the first instruction, is known. This kind of system is today typically a RISC architecture machine, but other specific cases may exist. The situation may result from the specific compiler which is used to compile an original program. In addition, the techniques which we describe to solve this program when used as a useful assumption in a shod section of text.

Our solution to this problem became a best case solution as the result of the techniques and system utilization also solves a case where both instructions and data are in the original text byte stream, but they are at known boundaries.

Accordingly we have provided a system with architectural compounding rules in which classes of instructions which may be compounded for a particular architecture are provided, and We apply applicable compounding rules to a byte stream of instructions of a base instruction sequence, and examine the base instruction sequence byte stream for opcodes and proceed through the stream while applying the compounding rules. During this compounding operation an instruction compounding unit of the compounding facility identifies each instruction after examining the instruction length code, and generates tag bit information for a compound instruction which indicates the initial compound instruction of a set of compounded instructions, and as the compounding facility continues compounding its output provides a compound sequence of compound instructions as a compound program. This is used by the instruction processor to allow execution of member units of a compound instruction in parallel, instructions which in prior machines would have been executed as scalar instructions, while handling the problems associated with mixed data and instructions, unknown boundaries by classical means, and other problems encountered by the more sophisticated architectures of our age.

In some situations, halfwords which exist in an instruction are initialized to indicate that the halfword is not an initial compound instruction. Best pairs are picked for compounding according to said compounding rules. As we have discovered there are situations which occur which preclude compounding, the system employs a technique for choosing instructions for compounding includes instructions which cannot be compounded and certain instructions which are examined are tagged to indicate that a specific instruction is not an initial instruction, and in the event that the initial instruction of the set being examined is not an appropriate as an initial compound instruction marking that instruction that it is not an initial instruction. The system proceeds to examine a subsequent group of instructions which includes the next sequential instruction in the group which had just been examined.

When instructions are mixed with non-instructions or data the system marks every halfword either as containing the first byte of an instruction or as an instruction not containing the first byte of a compound instruction, and skipping over in the compounding process the instruction marked as not containing the first bite of a compound instruction.

It will be recognized upon reading the detailed description what occurs in the system where there are compound instructions without a reference point, a case where compounding becomes more complex. In the event where in the base instruction set stream being examined are the boundaries of instructions members to be compounded, each with a plurality of possible sequences of instructions, then for each sequence possible boundaries are determined, and the the plural sequence are reduced to a single sequence of bits by our new hardware for logically ORing the plurality of possible sequences.

Typically in S/370 type systems there are three possible sequences of instructions with potential boundaries, each of which would produce a different sequence of compounding bits, and so the new system accommodates three bit sequences which are logically ORed to produce a single sequence to determine whether the current instruction is compounded by the instruction processor.

We have provided the system for compounding which experimentally tries the several possible sequences for convergence on a byte boundary, and when a convergence is detected, one of the converging sequences is eliminated until the number of sequences is collapsed until a single sequence remains.

A second higher level problem which has to be recognized is that in machines having a non-RISC architecture it is possible to has instructions and data intermixed in a byte stream, so that the compound instruction byte stream can be compounded correctly without additional information. This we consider to be be worst case, one that can occur if a byte stream of mixed instructions and data exists and there is no knowledge of where any instruction begins. The hardware which we provide accommodates this kind of system. As in the prior more general example, there is a convergence testing, and, if a convergence occurs and a sequence is eliminated by convergence, a new sequence is started in place of a sequence eliminated by convergence. Again here a sequence is started at every halfword. A set of instructions are examined and compounding bits are determined, and this examination process is repeated starting a number of bytes in sequence later, and compounding bits from each sequence are ORed to form a composite compound bit instruction.

In addition, more advanced instruction processor architectures such as those of the IBM S/370 type can process instructions where instructions have a variable length. Compounding instructions to form programs for more advanced machines of this architecture is difficult, and may be considered as exemplary of the highest order worst case because it is not apparent how the beginning of an instruction in a string of bytes can be identified even though it may be known that the string contains only instructions. Such a situation is characteristic of the current S/370 instruction processors, because such instruction processors have provision for identifying instructions in cache during the process of instruction fetch.

In our described preferred system, like the S/370 there is provided a compounding facility with a cache, and the system which we prefer has a cache and provides for a cache miss or a branch to a part of a line not yet compounded that a plurality of instructions are provided to the central processing unit via said instruction program without compounding of the instructions.

Such a S/370 system architecture compounding facility examines a byte stream and, upon encountering data, it will compound data as if the data consisted of instructions, but an S/370 architecture machine will have an instruction processor which cannot execute data and accordingly will execute a compound instruction program correctly. Instructions which are examined are tagged to indicate that a specific instruction is not an initial instruction, and in the event that the initial instruction of the set being examined is not an appropriate as an initial compound instruction marking that instruction that it is not an initial instruction, the detailed system examines a subsequent group of instructions which includes the next sequential instruction in the group which had just been examined. The compounding facility skips over in the compounding process the instruction marked as not containing the first byte of a compound instruction.

Another technique we have also discovered is how to compound instructions without a reference point, a case where compounding becomes more complex. S/370 type processors may have reference points, but programs contemplating variable length instructions are possible and so no reference point for an initial instruction may be present. In the event it is not known where in the base instruction set stream being examined is an instruction boundary and there are a plurality of possible sequences of instructions, each sequence is processed with compounding facility hardware for possible boundaries which are determined, and the the plural sequences obtained which are reduced to a single sequence of bits by logically ORing the plurality of possible sequences. Preferably for a S/370 type instruction processor the system will result in three bit sequences which are logically ORed to produce a single sequence to determine whether the current instruction is compounded by the instruction processor as detailed herein.

Using the base central processing unit for the compounding facility, during a process of fetching instructions from memory, wherein upon retrieval and before examination they are marked as instructions and not data. Furthermore, the compounding facility provides compound bits of an instruction which indicate the number of instructions which make up a compound group instruction. This enables us to handle multiple unit compound instructions for more advanced machines, and not just pairs. In each compound instruction the tag information provided for the compound instruction has bit information applicable to a member instruction unit of the compound instruction so as to indicating whether or not that particular member unit is to be compounded with a subsequent instruction.

There are also additional alternate embodiments which can be incorporated in the system. The system instruction processor central processor during execution may ignore all but tag information of a first instruction member of a compound instruction. During examination bytes not executed can be marked as not examined in which case they will be reexamined could the code be later executed by the instruction processor. During examinations bytes not executed can be left in an examined state, indicating the correctness of location even though they may be ignored or not executed during execution. During examination bytes can be left in a partially examined sequence state for examination of other sequences later.

These and other improvements are detailed in the following detailed description. For a better understanding of the inventions, together with advantages and features, reference be had to the co-pending applications for some detailed background. Further, specifically as to the improvements described herein reference should be made to the following description and the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

For a better understanding of the invention, together with its advantages and features, reference be had to the co-pending applications for some detailed background. Further, specifically as to the improvements described herein reference should be made to the following description and the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is high-level schematic diagram of a computing system which is capable of compounding instructions in a sequence of scalar instructions for concurrent execution.

FIG. 2 is a timing diagram for a uni-processor implementation showing the parallel execution of certain instructions which have been selectively grouped in a compound instruction stream.

FIG. 3 is a block diagram of a hierarchical memory organization in a scalable compound instruction set machine with in-memory processing shown as an alternative preferred embodiment for the operational environment.

FIG. 4-A is a high level schematic diagram of the process which will be described pictorially for providing means for processing existing programs to identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions, while

FIG. 4-B illustrates the preferred operational environment of the inventions and the inventions' location in the environment.

FIG. 5 shows a path taken by a program from original program code to actual execution.

FIG. 6 is a flow diagram showing generation of a compound instruction set program from an assembly language program; while

FIG. 7 is a flow diagram showing execution of a compound instruction set.

FIG. 8 illustrates a compound instruction execution engine.

FIG. 9 illustrates a compound instruction program.

FIG. 10 illustrates a situation where branch target may be at the beginning or middle of a compound instruction.

FIG. 11 illustrates an example of the best case for compounding.

FIG. 12 illustrates an example with a reference point.

FIG. 13 illustrates an example of compounding for the worst case in accordance with the preferred embodiment.

FIG. 14 illustrates the logical and hardware implementation of the worst case compounder.

FIG. 15 illustrates backward compounding.

FIG. 16 illustrates a compounding example of the worst case with four instructions per group.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring to FIG. 1 of the drawings, there is shown a representative embodiment of a portion of a digital computer system for a digital data processing system constructed in accordance with the present invention. This computer system is capable of executing 2 or more instructions in parallel. The system includes the capability of compounding instructions for parallel or concurrent execution. In this regard, "compounding" refers to the grouping of a plurality of instructions in a sequence of scalar instructions, wherein the size of the grouping is scalable from 1 to N. For example, the sequence of scalar instructions could be drawn from an existing set of scalar instructions such as that used by the IBM System/370 products.

In order to support the concurrent execution of a group of up to N instructions, the computer system includes a plurality of instruction execution units which operate in parallel in a concurrent manner; each on its own, is capable of processing one or more types of machine-level instructions.

As is generally shown in FIG. 1, an instruction compounding unit 20 takes a stream of binary scalar instructions 21 and selectively groups some of the adjacent scalar instructions to form encoded compound instructions. A resulting compounded instruction stream 22, therefore, provides scalar instructions to be executed singly or in compound instructions formed by groups of scalar instructions to be executed in parallel. When a scalar instruction is presented to an instruction processing unit 24, it is routed to the appropriated one of a plurality of execution units for serial execution. When a compound instruction is presented to the instruction processing unit 24, its scalar components are each routed to an appropriate execution unit for simultaneous parallel execution. Typical functional units include, but are not limited to, an arithmetic logic unit (ALU) 26, 28 a floating point arithmetic unit (FP) 30 and a storage address generation unit (AU) 32.

It is understood that compounding is intended to facilitate the parallel issue and execution of instructions in all computer architectures capable of processing multiple instructions per cycle.

Referring now to FIG. 2, compounding can be implemented in a uni-processor environment where each functional unit executes a scalar instruction (S) or, alternatively, a compound instruction (CS). As shown in the drawing, an instruction stream 33 containing a sequence of scalar and compounded scalar instructions has control tags (T) associated with each compound instruction. Thus, a first scalar instruction 34 could be executed singly by functional unit A in cycle 1; a triplet compound instruction 36 identified by tag T3 could have its 3 compounded scalar instructions executed in parallel by functional units A, C, and D in cycle 2; another compound instruction 38 identified by tag T2 could have its pair of compounded scalar instructions executed in parallel by functional units A and B in cycle 3; a second scalar instruction 40 could be executed singly by functional unit C in cycle 4; a large group compound instruction 42 could have its 4 compounded scalar instructions executed in parallel by functional units

A-D in cycle 5; and a third scalar instruction 44 could be executed singly by functional unit A in cycle 6.

One example of a computer architecture which can be adapted for handling compound instructions is an IBM System/370 instruction level architecture in which multiple scalar instructions can be issued for execution in each machine cycle. In this context, a machine cycle refers to a single pipeline stage required to execute a scalar instruction. When an instruction stream is compounded, adjacent scalar instructions are selectively grouped for the purpose of concurrent or parallel execution.

In general, an instruction compounding facility will look for classes of instructions that may be executed in parallel. When compatible sequences of instructions are found, a compound instruction is created.

Compounding techniques have be discussed in other applications. Reference is given to U.S. patent application Ser. No. 07/519,384 (IBM Docket EN9-90-020), filed May 4, 1990, and U.S. patent application Ser. No. 07/519,382, entitled GENERAL PURPOSE COMPOUNDING TECHNIQUE FOR INSTRUCTION-LEVEL PARALLEL PROCESSORS (IBM Docket EN9-90-019), filed May 4, 1990, for an understanding of compounding generally. An illustration of an instruction compounding unit for pairwise compounding is given in U.S. patent application Ser. No. 07/543,464 (IBM Docket EN9-90-018), filed Jun. 26, 1990.

Generally, it is useful to provide for compounding at a time prior to instruction issue so that the process can be done once for an instruction or instructions that may be executed many times. It has been proposed to locate the instruction compounding functions in the real memory of a computer system in order to implement compounding in hardware, after compile time, yet prior to instruction issue. Such compounding is considered to be a preferred alternative to other alternatives described herein and referred to as "in-memory compounding" which can be illustrated with reference to U.S. patent application Ser. No. 07/522,219 (IBM Docket EN9-90-012), filed May 10, 1990, and U.S. patent application Ser. No. 07/543,464 (IBM Docket EN9-90-018), filed Jun. 26, 1990, and FIG. 3 hereof. Memory management, as described herein by way of background, is also described in U.S. patent application Ser. No. 07/543,458 entitled MEMORY MANAGEMENT FOR SCALABLE COMPOUND INSTRUCTION SET MACHINES WITH IN-MEMORY COMPOUNDING (IBM Docket EN9-90-042), filed Jun. 26, 1990.

Generally, in-memory compounding is illustrated in FIG. 3. In FIG. 3, a hierarchical memory organization includes an I/O adaptor 40 which interfaces with auxiliary storage devices and with a computer real memory. The real memory of the organization includes a medium speed, relatively high capacity main memory 46 and a high-speed, relatively low capacity instruction cache 48. (The main memory and cache collectively are referred to herein as "real memory", "real storage", or, simply "memory".) A stream of instructions is brought in from auxiliary storage devices by way of the I/O adaptor 40, and stored in blocks called "pages" in the main memory 46. Sets of contiguous instructions called "lines" are moved from the main memory 46 to the instruction cache 48 where they are available for high-speed reference for processing by the instruction fetch and issue unit 50. Instructions which are fetched from the cache are issued, decoded at 52, and passed to the functional units 56, 58, . . . , 60 for execution.

During execution, when reference is made to an instruction which is in the program, the instruction's address is provided to a cache management unit 62 which uses the address to fetch one or more instructions, including addressed instruction, from the instruction cache 48 into the queue in the unit 50. If the addressed instruction is in the cache, a cache "hit" occurs. Otherwise, a cache "miss" occurs. A cache miss will cause the cache management unit 62 to send the line address of the requested instruction to a group of storage management functions 64. These functions can include, for example, real storage management functions which use the line address provided by the cache management unit 62 to determine whether the page containing the addressed line is in the main memory 46. If the page is in main memory, the real storage management will use the line address to transfer a line containing the missing instruction from the main memory 46 to the instruction cache 48. If the line containing the requested instruction is not in the main memory, the operating system will activate another storage management function, providing it with the identification of the page containing the needed line. Such a storage management function will send to the I/O adaptor 40 an address identifying the page containing the line. The I/O adaptor 40 will bring the page from auxiliary storage and provide it to the main memory 46. To make room for the fetched page, the storage management function selects a page in the main memory 46 to be replaced by the fetched page. In SCISM architecture, it is contemplated that the replaced page is returned to auxiliary storage through the I/O adaptor without compounding tag information. In this manner, those instructions most likely to be immediately required during execution of an instruction sequence are adjacent the functional units in the instruction cache 48. The hierarchical memory organization provides the capability for fast retrieval of instructions that are required but not in the cache.

In the context of SCISM architecture, in-memory instruction compounding can be provided by an instruction compounding unit 70 which is located functionally between the I/O adaptor 40 and the main memory 46 so that compounding of the scalar instruction stream can take place at the input to, or in, the main memory 46. In this location, instructions can be compounded during an ongoing page fetch.

Alternatively, the instruction compounding unit can occupy the position 72 between the main memory 46 and the instruction cache 48 and compound instructions are formed line-by-line as they are fetched into the instruction cache 48, as may be considered a preferred embodiment.

Execution of Compound Instructions with Branching

There is a need for executing compound instructions after processing, including execution of compound instruction in the presence of branching after instructions have been analyzed statically as any other instruction. We decode instructions in advance, and append control information to the instructions in the program to be executed.

Furthermore, while achieving this goal while providing the program compounding rule means for forming a compound instruction program, architectural information is supplied comparable to that supplied to modern compilers.

During the process of analysis all members of a branch class are analyzed statically as any other instruction so as to have compound execution of instructions which is guaranteed to work .during execution in a machine. Our preferred way of handling branching is discussed with reference to FIGS. 9 and 10 hereafter. To understand that environment, a discussion of our preferred ways of compounding is appropriate.

The particular technique for compounding is a matter of design choice. However, for purposes of illustration, one technique for creating compound instructions formed from adjacent scalar instructions can be stated, as illustrated in the aforementioned In Memory Preprocessor application U.S. Ser. No. 07/543,458. By way of example, instructions may occupy 6 bytes (3 half words), 4 bytes (2 half words), or 2 bytes (1 half word) of text. For this example, the rule for compounding a set of instructions which includes variable instruction lengths provides that all instructions which are 2 bytes or 4 bytes long are compoundable with each other. That is, a 2 byte instruction is capable of parallel execution in this particular example with another 2 byte or another 4 byte instruction and a 4 byte instruction is capable of parallel execution with another 2 byte or another 4 byte instruction. The rule further provides that all instructions which are 6 bytes long are not compoundable. Thus, a 6 byte instruction is only capable of execution singly by itself. Of course, compounding is not limited to these exemplary rules, but can embrace a plurality of rules which define the criteria for parallel execution of existing instructions in a specific configuration for a given computer architecture.

The instruction set used for this example is taken from the System/370 architecture. By examining the OP code for each instruction, the length of each instruction can be determined from an instruction length code (ILC) in the op code. The instruction;s type is further defined in other op code bits. Once the type and length of the instruction is determined, a compounding tag containing tag bits is then generated for that specific instruction to denote whether it is to be compounded with one or more other instructions for parallel execution, or to be executed singly by itself.

In the example (which is not limiting), if 2 adjacent instructions can be compounded, the tag bits, which are generated in memory, are "1" for the first compounded instruction and "zero" for the second compounded instruction. However, if the first and second instructions cannot be compounded, the tag bit for the first instruction is "zero" and the second and third instructions are then considered for compounding. Once an instruction byte stream has been processed in accordance with the chosen compounding technique and the compounding bits encoded for various scalar instructions, more optimum results for achieving parallel execution may be obtained by using a bigger window for looking at larger groups of instructions and then picking the best combination of N instructions for compounding.

However, taking the above by way of example of the problem solved here, it may be considered that we have generally a need to provide the system and process illustrated generally by FIG. 4. Existing programs written in existing high level languages, as described with reference to FIG. 5, or existing assembly language programs to be processed, as described with reference to FIG. 6, need to be processed, and we have provided system which has the capability to identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions.

Turning now to FIG. 4, is will be seen that there is illustrated pictorially the sequence in which a program is provided as an input to a compounding facility that produces a compound instruction program based on a set of rules which reflect both the system and hardware architecture. The preferred compounding facility is illustrated by U.S. Ser. No. 07/642,011 filed Jan. 15, 1991 (EN9-90-049). These rules are hereafter referred to as compounding rules. The program produced by the compounding facility can then be executed directly by a compound instruction execution engine generally illustrated by FIG. 8.

However, FIG. 5 shows a typical path taken by a program from higher level source code to actual execution, and may also be considered as one of the possible organizations suggested by FIG. 4. An alternative related to assembly level programs is discussed with respect to FIG. 6.

As will have been appreciated, referring to FIG. 5, there are many possible locations in a computer system where compounding may occur, both in software and in hardware. Each has unique advantages and disadvantages. As shown in FIG. 5, there are various stages that a program typically takes from source code to actual execution. During the compilation phase, a source program is translated into machine code and stored on a disk 46. During the execution phase the program is read from the disk 46 and loaded into a main memory 48 of a particular computer system configuration 50 where the instructions are executed by appropriate instruction processing units 52, 54, 56. Compounding could take place anywhere along this path. In general as the compounder is located closer to an instruction processing unit or CPUs, the time constraints become more stringent. As the compounder is located further from the CPU, more instructions can be examined in a large sized instruction stream window to determine the best grouping for compounding for increasing execution performance. However such early compounding tends to have more of an impact on the rest of the system design in terms of additional development and cost requirements.

The flow diagram of FIG. 6 shows the generation of a compound instruction set program from an assembly language program in accordance with a set of customized compounding rules 58 which reflect both the system and hardware architecture. The assembly language program is provided as an input to a software compounding facility 59 that produces the compound instruction program. Successive blocks of instructions having a predetermined length are analyzed by the software compounding facility 59. The length of each block 60, 62, 64 in the byte stream which contains the group of instructions considered together for compounding is dependent on the complexity of the compounding facility.

As shown in FIG. 6, this particular compounding facility is designed to consider two-way compounding for "m" number of fixed length instructions in each block. The primary first step is to consider if the first and second instructions constitute a compoundable pair, and then if the second and third constitute a compoundable pair, and then if the third and fourth constitute a compoundable pair, all the way to the end of the block. Once the various possible compoundable pairs C1-C5 have been identified, the compounding facility can select the preferred sequence of compounded instructions and use flags or identifier bits to identify the optimum sequence of compound instructions.

If there is no optimum sequence, all of the compoundable adjacent scalar instructions can be identified so that a branch to a target located amongst various compound instructions can exploit any of the compounded pairs which are encountered (See FIG. 14). Where multiple compounding units are available, multiple successive blocks in the instruction stream could be compounded at the same time.

Of course it is easier to pre-process an instruction stream for the purpose of creating compound instructions if known reference points already exist to indicate where instructions begin. As used herein, a reference point means knowledge of which byte of text is the first byte in an instruction. This knowledge could be obtained by some marking field or other indicator which provides information about the location of instruction boundaries. In many computer systems such a reference point is expressly known only by the compiler at compile time and only by the CPU when instructions are fetched. Such a reference point is unknown between compile time and instruction fetch unless a special reference tagging scheme is adopted.

The flow diagram of FIG. 7 shows the execution of a compound instruction set program which has been generated by a hardware preprocessor 66 or a software preprocessor 67. A byte stream having compound instructions flows into a compound instruction (CI) cache 68 that serves as a storage buffer providing fast access to compound instructions. CI issue logic 69 fetches compound instructions from the CI Cache and issues their individual compounded instructions to the appropriate functional units for parallel execution.

It is to be emphasized that instruction execution units (CI EU) 71 such as ALU's in a compound instruction computer system are capable of executing either scalar instructions one at a time by themselves or alternatively compounded scalar instructions in parallel with other compounded scalar instructions. Also, such parallel execution can be done in different types of execution units such as ALU's, floating point (FP) units 73, storage address-generation units (AU) 75 or in a plurality of the same type of units (FP1, FP2, etc) in accordance with the computer architecture and the specific computer system configuration.

When compounding is done after compile time, a compiler could indicate with tags which bytes contain the first byte of an instruction and which contain data. This extra information results in a more efficient compounder since exact instruction locations are known. Of course, the compiler would differentiate between instructions and data in other ways in order to provide the compounder with specific information indicating instruction boundaries.

In the exemplary two-way compounding embodiment of this application, compounding information is added to the instruction stream as one bit for every two bytes of text (instructions and data). In general, a tag containing control information can be added to each instruction in the compounded byte stream--that is, to each non-compounded scalar instruction as well as to each compounded scalar instruction included in a pair, triplet, or larger compounded group. As used herein, identifier bits refers to that part of the tag used specifically to identify and differentiate those compounded scalar instructions forming a compounded group from the remaining non-compounded scalar instructions remain in the compound instruction program and when fetched are executed singly.

In a system with all 4-byte instructions aligned on a four byte boundary, one tag is associated with each four bytes of text. Similarly, if instructions can be aligned arbitrarily, a tag is needed for every byte of text.

The case of compounding at most two instructions provides the smallest grouping of scalar instructions to form a compound instruction, and uses the following preferred encoding procedure for the identifier bits. Since all System/370 instructions are aligned on a halfword (two-byte) boundary with lengths of either two or four or six bytes, one tag with identifier bits is needed for every halfword. In this small grouping example, an identifier bit "1" indicates that the instruction that begins in the byte under consideration is compounded with the following instruction, while a "0" indicates that the instruction that begins in the byte under consideration is not compounded. The identifier bit associated with halfwords that do not contain the first byte of an instruction is ignored. The identifier bit for the first byte of the second instruction in a compounded pair is also ignored. As a result, this encoding procedure for identifier bits means that in the simplest case only one bit of information is needed by a CPU during execution to identify a compounded instruction.

Where more than two scalar instructions can be grouped together to form a compound instruction, additional identifier bits may be required. The minimum number of identifier bits needed to indicate the specific number of scalar instructions actually compounded is the logarithm to the base 2 (rounded up to the nearest whole number) of the maximum number of scalar instructions that can be grouped to form a compound instruction. For example, if the maximum is two, then one identifier bit is needed for each compound instruction. If the maximum is three or four, then two identifier bits are needed for each compound instruction. If the maximum is five, six, seven or eight, then three identifier bits are needed for each compound instruction. This encoding scheme is shown below in Table 1:

TABLE 1

Identifier Bits	Encoded meaning	Total # Compounded
00	This instruction is not compounded with its following instruction	none
01	This instruction is compounded with its one following instruction	two
10	This instruction is compounded with its two following instructions	three
11	This instruction is compounded with its three following instructions	four

It will therefore be understood that each halfword needs a tag, but the CPU ignores all but the tag for the first instruction in the instruction stream being executed. In other words, a byte is examined to determine if it is a compound instruction by checking its identifier bits. If it is not the beginning of a compound instruction, its identifier bits are zero. If the byte is the beginning of a compound instruction containing two scalar instructions, the identifier bits are "1" for the first instruction and "0" for the second instruction. If the byte is the beginning of a compound instruction containing three scalar instructions, the identifier bits are "2" for the first instruction and "1" for the second instruction and "0" for the third instruction. In other words, the identifier bits for each half word identify whether or not this particular byte is the beginning of a compound instruction while at the same time indicating the number of instructions which make up the compounded group.

This method of encoding compound instructions assumes that if three instructions are compounded to form a triplet group, the second and third instructions are also compounded to form a pair group. In other words, if a branch to the second instruction in a triplet group occurs, the identifier bit "1" for the second instruction indicates that the second and third instruction will execute as a compounded pair in parallel, even though the first instruction in the triplet group was not executed.

It will be apparent to those skilled in the art that the present invention requires an instruction stream to be compounded only once for a particular computer system configuration, and thereafter any fetch of compounded instructions will also cause a fetch of the identifier bits associated therewith. This avoids the need for the inefficient last-minute determination and selection of certain scalar instructions for parallel execution that repeatedly occurs every time the same or different instructions are fetched for execution in the so-called super scalar machine.

Despite all of the advantages of compounding an instruction stream, it becomes difficult to do so under certain computer architectures unless a technique is developed for determining instruction boundaries in a byte string. Such a determination is complicated when variable length instructions are allowed, and is further complicated when data and instructions can be intermixed. Of course, at execution time instruction boundaries must be known to allow proper execution. But since compounding is preferably done a sufficient time prior to instruction execution, a technique is needed to compound instructions without knowledge of where instructions start and without knowledge of which bytes are data. This technique needs to be applicable to all of the accepted types of architectures, including the RISC (Reduced Instruction Set Computers) architectures in which instructions are usually a constant length and are not intermixed with data.

There are a number of variations of the technique of the present invention, depending on the information that is already available about the particular instruction stream being compounded. The various combinations of typical pertinent information are shown below in Table 2:

TABLE 2

Case	Instruction Length	Data Intermixed	Reference Point
A	fixed	no	yes
B	variable	no	yes
C	fixed or variable	yes	yes
D	fixed	no	no
E	variable	no	no
F	fixed	yes	no
G	variable	yes	no

It is to be noted that in some instances fixed and variable length instructions are identified as being different cases. This is done because the existence of variable length instructions creates more uncertainty where no reference point is known, thereby resulting in the creation of many more potential compounding bits. In other words, when generating the potential instruction sequences as provided by the technique of this invention, there are no compounding identifier tags for bytes in the middle of any fixed length instructions. Also, the total number of identifier tags required under the preferred encoding scheme is fewer (i.e., one identifier tag for every four bytes for instructions having a fixed length of four bytes). Nevertheless, the unique technique of this invention works equally well with either fixed or variable length instructions since once the start of an instruction is known (or presumed), the length can always be found in one way or another somewhere in the instructions. In the System/370 instructions, the length is encoded in the opcode, while in other systems the length maybe encoded in the operands.

In case A with fixed length instructions having no data intermixed and with a known reference point location for the opcode, the compounding can proceed in accordance with the applicable rules for that particular computer configuration. Since the length is fixed, a sequence of scalar instructions is readily determined, and each instruction in the sequence can be considered as possible candidates for parallel execution with a following instruction. A first encoded value in the control tag indicates the instruction is not compoundable with the next instruction, while a second encoded value in the control tag indicates the instruction is compoundable for parallel execution with the next instruction.

Similarly in case B with variable length instructions having no data intermixed, and with a known reference point for the instructions (and therefore also for the instruction length code) the compounding can proceed in a routine manner. The opcodes indicate an instruction sequence for example as follows: the first instruction is 6 bytes long, the second and third are each 2 bytes long, the fourth is 4 bytes long, the fifth is 2 bytes long, the sixth is 6 bytes long, and the seventh and eighth are each 2 bytes long.

For convenience of illustration, as shown in FIG. 6 for example, the illustrated examples of opcodes illustrated may be either fixed or of variable length.

For purposes of illustration, the technique for compounding herein is shown for creating compound instructions formed from adjacent pairs of scalar instructions as well as for creating compound instructions formed from larger groups of scalar instructions. The exemplary rules for the embodiments shown in the drawings are additionally defined to provide that all instructions which are 2 bytes or 4 bytes long are compoundable with each other (i.e., a 2 byte instruction is capable of parallel execution in this particular computer configuration with another 2 byte or another 4 byte instruction). The rules further provide that all instructions which are 6 bytes long in the System/370 context are not compoundable at all (i.e., a 6 byte instruction is only capable of execution singly by itself in this particular computer configuration). Of course, the invention is not limited to these exemplary compound rules, but is applicable to any set of compounding rules which define the criteria for parallel execution of existing instructions in a specific configuration for a given computer architecture.

The instruction set used in these exemplary compounding techniques of the invention is taken from the System/370 architecture. By examining the opcode for each instruction, the type and length of each instruction can be determined and the control tag containing identifier bits is then generated for that specific instruction, as described in more detail hereinafter. Of course, the present invention is not limited to any specific architecture or instruction set, and the aforementioned compounding rules are by way of example only.

The preferred encoding for compound instructions in these illustrated embodiments is now described. If two adjacent instructions can be compounded, their identifier bits which are generated for storage are "1" for the first compounded instruction and "0" and the second and third instruction are then considered for compounding. Once an instruction byte stream has been pre-processed in accordance with this technique and identifier bits encoded for the various scalar instructions, more optimum results for achieving parallel execution may be obtained by using a bigger window for looking at larger groups, and then picking the best combination of adjacent pairs for compounding.

In connection with our invention, accomplishing the tasks generally described with reference to FIG. 4-A is achieved with the assistance of a compound instruction execution engine of the kind generally described in U.S. Ser. No. 07/519,382 filed May 4, 1990 in an alternative environment.

Generally the preferred operating environment may be represented by the operational environment shown in FIG. 4-B. While the compounding facility may be a software entity, the compounding facility may be implemented by an instruction compounding unit as described in detail in U.S. Ser. No. 07/642,011 filed Jan. 16, 1991. Referring to FIG. 4-B of the drawings, there is shown a representative embodiment of a portion of a digital computer system or digital data processing system constructed in accordance with the present invention with a cache management unit 144. This computer system is capable of processing two or more instructions in parallel. It includes a first storage mechanism for storing instructions and data to be processed in the form of a series of base instructions for a scalar machine. This storage mechanism is identified as higher-level storage 136. This storage (also "main memory") is a large capacity lower speed storage mechanism and may be, for example, a large capacity system storage unit or the lower portion of a comprehensive hierarchical storage system.

The computer system of FIG. 4-B also includes an instruction compounding facility or mechanism for receiving instructions from the higher level storage 136 and associating with these instructions compound information in the form of tags which indicate which of these instructions may be processed in parallel with one another. A suitable instruction compounding unit is represented by the instruction compounding unit 137. This instruction compounding unit 137 analyzes the incoming instructions for determining which ones may be processed in parallel. Furthermore, instruction compounding unit 137 produces for these analyzed instructions tab bits which indicate which instructions may be processed in parallel with one another and which ones may not be processed in parallel with one another but must be processed singly.

The FIG. 4-B system further includes a second storage mechanism coupled to the instruction compounding mechanism 137 for receiving and storing the analyzed instructions and their associated tag fields so that these stored compounded instructions may be fetched. This second or further storage mechanism is represented by compound instruction cache 138. The cache 138 is a smaller capacity, higher speed storage mechanism of the kind commonly used for improving the performance rate of a computer system by reducing the frequency of having to access the lower speed storage mechanism 136.

The FIG. 4 system further includes a plurality of functional instruction processing units which operate in parallel with one another. These functional instruction processing units 139, 140, 141, et cetera. These functional units 139-141 operate in parallel with one another in a concurrent manner and each, on its own, is capable of processing one or more types of machine-level instructions. Examples of functional units which may be used are: a general purpose arithmetic and logic unit (ALU), an address generation type ALU, a data dependency collapsing ALU (of the preferred type shown in co-pending U.S. Ser. No. 07/504,910 filed Apr. 4, 1990), a branch instruction processing unit, a data shifter unit, a floating point processing unit, and so forth. A given computer system may include two or more or some of the possible functional units. For example, a given computer system may include two or more general purpose ALUs. Also, no given computer system need include each and every one of these different types of functional units. The particular configuration will depend on the nature of the particular computer system being considered.

The computer system of FIG. 4-B also includes an instruction fetch and issue mechanism coupled to compound instruction cache 138 for supplying adjacent instructions stored therein to different ones of the functional instruction processing units 139-141 when the instruction tag bits indicate that they may be processed in parallel. This mechanism also provides single instructions to individual functional units when their tag bits indicate parallel execution is not possible and they must be processed singly. This mechanism is represented by instruction fetch and issue unit 142. Fetch and issue unit 142 fetches instructions form the cache 138 and examines the tag bits and instruction operation code (opcode) fields, performing a decode function, and based upon such examinations sends the instruction under consideration to the appropriate ones of the functional units 138-141.

In the context of SCISM architecture, in-cache instruction compounding is provided by the instruction compounding unit 137 so that compounding of each cache line can take place at the input to the compound instruction cache 138. Thus, as each cache line is fetched from the main memory 136 into the cache 138, the line is analyzed for compounding in the unit 137 and passed, with compounding information tag bits, for storage in the compound instruction cache 138.

Prior to caching, the line is compounded in the instruction compound unit 137 which generates a set of tag bits. These tag bits may be appended directly to the instructions with which they are associated. Or they may be provided in parallel with the instructions themselves. In any case, the bits are provided for storage together with their line of instructions in the cache 138. As, needed, the compounded instruction in the cache 138 is fetched together with its tag bit information by the instruction and issue unit 142. As instructions are received by the fetch and issue unit 142, their tag bits are examined to determine by decoding examination whether they may be processed in parallel and the opcode fields are examined to determine which of the available functional units is most appropriate for their processing. If the tag bits indicate that two or more of the instructions are suitable for processing in parallel, then they are sent to the appropriate ones in in the functional units in accordance with the codings of their opcode fields. Such instructions are then processed concurrently with one another by their respective functional units.

When an instruction is encountered that is not suitable for parallel processing, it is sent to the appropriate functional unit as determined by an opcode and it is thereupon processed alone and singly by itself in the selected functional unit.

In the most perfect case, where plural instructions are always being processed in parallel, the instruction execution rate of the computer system would be N times as great as for the case where instructions are executed one at a time, with N being the number of instructions in the groups which are being processed in parallel.

FIG. 8 illustrates a compound instruction execution engine in which it is possible to execute branching in the middle of a compound instruction. Compound instructions flow from storage into the CI cache (a storage buffer providing fast access to compound instructions). The CI issue logic fetches compound instructions from the CI cache and issues them to the appropriate functional unit. The set of functional units consists of traditional functional units, such as

.smallcircle. EU--execution unit, ALU

.smallcircle. AU--storage address-generation unit

.smallcircle. FP--floating-point arithmetic unit and others, but differ from the traditional in that each unit can execute a compound instruction, as well as a single scalar instruction. Thus, the name of each unit is prefixed with "CI", indicating that it can execute compound instructions; hence "CI EU", rather than "EU", etc. as illustrated by FIG. 8 in the drawings.

Combining several instructions into a single compound instruction allows the instruction processing unit in the computer to effectively decode and execute those instructions in parallel, thus improving performance. While the concept of parallel instruction decoding and execution is well known in the art and is discussed in the Introduction section, the normal technique employed is to dynamically decode the instruction stream at the time it enters the instruction decoding hardware to determine whether the instructions may be executed in parallel. This determination is unique to each instruction set architecture, as well as the underlying implementation of that architecture in any given instruction processor. Dynamic decoding (or dynamic scheduling, as it is described in the Introduction section) is often limited by the complexity of the architecture because it leads to complex logic to determine which combinations of instructions can be executed in parallel, and thus may increase the cycle time of the instruction processor. This invention teaches the instruction processor decoding provides compound instructions for a series of base instructions of a scalar machine generating a series of compound instructions with an instruction format text having appended control bits in the instruction format text enabling the execution of the compound instruction format text in said instruction processor which may be fetched and decodes for determining compound and single instructions which can be executed as compounded and single instructions by the arithmetic and logic units of the instruction processor and preserving intact the scalar execution of the base instructions of a scalar machine which were originally in storage. This assures that the results of executing the compound instruction are the same as would have been achieved had the instructions been executed individually, a necessary condition to the correct execution of existing programs, but capable of being generally executed in a faster manner due to the parallel nature of the compounded instruction stream.

The compounding facility may be a software entity. The design of the software will not be discussed here because the details are unique to a given instruction set architecture and underlying implementation, and because the design of such programs is somewhat similar in concept to modern compilers which perform instruction scheduling and other optimizations based on the specific machine architecture. That is, given an input program and a description of the system (instruction set) and hardware architectures (i.e., the structural aspects of the implementation), an output program is produced. In the case of the modern compiler, the output is an optimized sequence of instructions. In the case of the invention, the output is a series of compound instructions along with the necessary control bits, namely, a compound instruction program. As stated previously, the basis for forming the compound instruction program is the set of compounding rules, which supply architectural information comparable that supplied to modern compilers.

A compound instruction program is illustrated in FIG. 9. Here, the compounding facility has composed n code blocks, or compoundable units, into n compound instructions of varying lengths. Within each compound instruction is found the original instructions of varying lengths. Within each compound instruction is found the original instructions in the input program with the addition of a control field, T, appended to each instruction as illustrated by FIG. 9 in the drawings.

The bits T of FIG. 9 are defined to indicate a specific meaning. In one embodiment, the to bit will mark the beginning of an instruction, as illustrated below. In all cases a 0 shall mean not asserted. In all cases it is preferred to have the compound instruction fetched with a certainty. Accordingly, the number of bits chosen for fetching is the maximum of number of bits which may be contained and could be executed with an instruction of the maximum instruction length of the target machine which is executing the parallel compounded instructions. Thus the fetching system provides for maximum length fetching.

While one preferred embodiment will utilize a tag which indicates the start of a compound instruction, as illustrated by "T" in FIGS. 6 and 9. However, in the preferred embodiment the T shall be understood to represent the number of instructions which follow which are to be considered part of the compounded instruction. Thus, as 0 represents a single instruction, an 1 indicates that there is one more instruction in the compound instruction, a 2 two instructions and a 3 three more instructions are compounded with the present instruction.

Accordingly, within the apparatus each compound instruction has a tag "T" for tag information related to a compound instruction appended to the instruction and a value of the information represents that the instruction is not to be part of another compound instruction or single instruction and is to be considered a single instruction, while another value to the tag information indicates that the instruction unit appended is a part of a compound instruction. The value of the tag information relates to the length of the compounded instruction and the number of instruction units within the compounded instruction to be executed.

In the preferred embodiment of the inventions the fetched compound instruction has the tag information which indicates the number of following instructions as a count value, and wherein subsequent instructions of the same compounded instructions have sub-count information provided by a sub-count tag appended to a member unit of the compounded instruction. In the following description of a preferred embodiment which relates to the use of a tag value which indicates that the instruction is part of a compound instruction, the value tag of 1 is common to both embodiments, but it should be remembered that for longer instructions than pairs, the value will relate to the number of instruction units of the compounded instruction.

As stated previously, the control field contains information relevant to the execution of compound instructions and may contain as little or as much information as is deemed efficacious for a particular implementation. For example, a control field might be defined as an 8-bit field,

```
t.sub.0
      t.sub.1  t.sub.2
                     t.sub.3
                            t.sub.4
                                t.sub.5
                                       t.sub.6,
                                           t.sub.7
```

with the bits defined as follows

Bit	Function
t.sub.0	If 1, this instruction marks the beginning of a compound instruction.
t.sub.1	If 1, then execute two compound instructions in parallel
t.sub.2	If 1, then this compound instruction has more than one execution cycle.
t.sub.3	If 1, then suspend pipelining.
t.sub.4	If instruction is a branch, then if this bit is 1, the branch is predicted to be taken.
t.sub.5	If 1, then this instruction has a storage interlock from a previous compound instruction.
t.sub.6	If 1, then enable dynamic instruction issue.
t.sub.7	If 1, then this instruction uses an ALU.

The t.sub.0 bit merits further discussion, in that, of all the above bits, it alone is a necessity. The purpose of the bit is to identify the beginning instruction of a compound instruction. Defining a logic 1 to mean that two instructions are to be executed in parallel and 0 to execute a single instruction execution, the hardware will easily be able to detect how many instructions comprise the compound instruction.

In general, the compounding facility will look for classes of instructions that may be executed in parallel, and ensure that no interlocks between members of a compound instruction exist that cannot be handled by the hardware. When compatible sequences of instructions are found, a compound instruction is created. For example, the System/370 architecture might be partitioned into the following classes:

1. RR-format loads, logicals, arithmetics, compares

.smallcircle. LCR--Load Complement

.smallcircle. LPR--Load Positive

.smallcircle. LNR--Load Negative

.smallcircle. LR--Load Register

.smallcircle. LTR--Load and Test

.smallcircle. NR--AND

.smallcircle. OR--OR

.smallcircle. XR--Exclusive OR

.smallcircle. AR--Add

.smallcircle. SR--Subtract

.smallcircle. ALR--Add Logical

.smallcircle. SLR--Subtract Logical

.smallcircle. CLR--Compare Logical

.smallcircle. CR--Compare

2. RS-format shifts (no storage access)

.smallcircle. SRL--Shift Right Logical

.smallcircle. SLL--Shift Left Logical

.smallcircle. SRA--Shift Right Arithmetic

.smallcircle. SLA--Shift Left Arithmetic

.smallcircle. SRDL--Shift Right Logical

.smallcircle. SLDL--Shift Left Logical

.smallcircle. SRDA--Shift Right Arithmetic

.smallcircle. SLDA--Shift Left Arithmetic

3. Branches--on count and index

.smallcircle. BCT--Branch on Count (RX-format)

.smallcircle. BCTR--Branch on Count (RR-format)

.smallcircle. BXH--Branch on Index High (RS-format)

.smallcircle. BXLE--Branch on Index Low (RS-format)

4. Branches--on condition

.smallcircle. BC--Branch on Condition (RX-format)

.smallcircle. BCR--Branch on Condition (RR-format)

5. Branches--and link

.smallcircle. BAL--Branch and Link (RX-format)

.smallcircle. BALR--Branch and Link (RR-format)

.smallcircle. BAS--Branch and Save (RX-format)

.smallcircle. BASR--Branch and Save (RR-format)

6. Stores

.smallcircle. STCM--Store Characters Under Mask (0-4-byte store, RS-format)

.smallcircle. MVI--Move Immediate (1 byte, SI-format)

.smallcircle. ST--Store (4 bytes)

.smallcircle. STC--Store Character (1 byte)

.smallcircle. STH--Store Half (2 bytes)

7. Loads

.smallcircle. LH--Load Half (2 bytes)

.smallcircle. L--Load (4 bytes)

8. LA--Load Address

9. RX-format arithmetics, logicals, inserts, compares

.smallcircle. A--Add

.smallcircle. AH--Add Half

.smallcircle. AL--Add Logical

.smallcircle. N--AND

.smallcircle. O--OR

.smallcircle. S--Subtract

.smallcircle. SH--Subtract Half

.smallcircle. SL--Subtract Logical

.smallcircle. X--Exclusive OR

.smallcircle. IC--Insert Character

.smallcircle. ICM--Insert Characters Under Mask (0- to 4-byte fetch)

.smallcircle. C--Compare

.smallcircle. CH--Compare Half

.smallcircle. CL--Compare Logical

.smallcircle. CLI--Compare Logical Immediate

.smallcircle. CLM--Compare Logical Character Under Mask

10. TM--Test Under Mask

The rest of the System/370 instructions are not considered to be compounded for execution in this invention. This does not preclude them from being compounded on a future compound instruction execution engine. It should be noted that the hardware structures required for compound instruction execution can be readily controlled by horizontal microcode, allowing for exploitation of parallelism in the remaining instructions, and thereby increasing performance.

One of the most common sequences in programs is to execute an instruction of the TM or RX-format compare class, the result of which is used to control the execution of a BRANCH-on-condition type instruction which immediately follows. Performance can be improved by executing the COMPARE and the BRANCH instructions in parallel, and this is sometimes done dynamically in high performance instruction processors. Some difficulty lies in quickly identifying all the various members of the COMPARE class of instructions and all the members of the BRANCH class of instructions in a typical architecture during the instruction decoding process. This difficulty is avoided by the invention, because the analysis of all the members of the classes are accomplished ahead of time and a compound instruction which is guaranteed to work is created.

Many classes of instructions may be executed in parallel, depending on how the hardware is designed. In addition to the COMPARE and BRANCH compound instruction described above, other compound instructions can be envisioned, such as LOADS and RR-format instructions, BRANCH and LOAD ADDRESS, etc. A compound instruction may even include multiple instructions from the same class, for example, RR-format arithmetic, if the processor has the necessary execution units.

In any realizable instruction processor, there will be an upper limit to the number of instructions that can comprise a compound instruction. This upper limit, m, must be specified to the compounding facility which is creating the executable instructions by generating compound instructions, so that it can generate compound instructions no longer than the maximum capability of the underlying hardware. Note that m is strictly a consequence of the hardware implementation; it does not limit the scope of instructions that may be analyzed for compounding in a given code sequence by the software. In general, the broader the scope of the analysis, the greater the parallelism achieved will be, as more advantageous compoundings are recognized by the compounding facility. To illustrate, consider the sequence:

X1 ;any compoundable instruction

X2 ;any compoundable instruction

LOAD R1,(X);load R1 from memory location X

ADD R3,R1;R3=R3+R1

SUB R1,R2;R1=R1-R2

COMP R1,R3;compare R1 with R3

X3 ;any compoundable instruction

X4 ;any compoundable instruction.

If the hardware-imposed upper limit on compounding is m=2, then there are a number of ways to compound this sequence of instructions depending on the scope of the compounding facility. If the scope were equal to four, then the compounding software would produce the pairings <-X1> <X2 LOAD> <ADD SUB> <COMP X3> <X4->, relieving completely the hazards between the LOAD and the ADD and the SUB and the COMP. On the other hand, a superscalar machine with m=2, which pairs instructions in its instruction issue logic on strictly a first-in-first-out basis, would produce the pairings <X1 X2> <LOAD ADD> <SUB COMP> <X3 X4>, incurring the full penalty of the interlocking instructions. Unfortunately, the LOAD and ADD cannot be executed in parallel because ADD requires the result of the load. Similarly, the SUB and COMD cannot execute in parallel. Thus no performance advance is gained.

Typically there can be no guarantee that a branch into the middle of a compound instruction will not occur. This is easily handled by the hardware, as illustrated in FIG. 10. In FIG. 10, the compounding tag representation uses T=1 to mark the start of a Compounding Instruction which extends up to, but not including the next tag of value 1. If, when fetching instructions, it always fetches the maximum length compound instruction from storage, beginning at exactly the branch target address, and then executes all instructions with t.sub.0 =0 as a compound instruction, up to the point in the instruction text where an instruction is encountered having t.sub.0 =1, indicating the beginning of the next compound instruction. If the t.sub.0 bit of the branch target is 1, then it is the beginning of a compound instruction and may be executed directly. FIG. 10 illustrates this situation.

In this figure, for simplicity of exposition, the T field associated with each compound instruction has been reduced to only the to bit. Also, the maximum length of compound instructions is three. The second instruction of CI.sub.1 is a branch instruction, that, for the sake of this example, will have two possible target paths, a and b. The a path branches to the middle of CI.sub.j, while the b path branches to the beginning of CI.sub.j. If the branch were to follow path a, the hardware fetches the maximum length compound instruction, i.e., three instructions, then executes I.sub.2.sup.i and I.sub.3.sup.i as a compound instruction. The remainder of the fetch, namely I.sub.1.sup.k is recognized to be the beginning of a new compound instruction, and is held in reserve while the rest of CI.sub.k is fetched for subsequent execution.

If the branch instruction took the b path to the beginning of CI.sub.j, the hardware would again fetch the maximum length compound instruction, yielding, in this case, a complete compound instruction, namely I.sub.1.sup.i, I.sub.2.sup.i and I.sub.3.sup.i. Execution of that compound instruction proceeds directly as illustrated by FIG. 10 in the drawings.

Improvements For Architectures With Different Attributes And For S/370 Systems With Backward Compounding

The general method of compounding instructions is applicable to machines which have an instruction-level parallel processor, but these architectures can be of different types, as RISC, or preferably s/370 type base processors can be used, with their extensions in future generations with the system attributes we detail. Compounding is the process where adjacent instructions are grouped together for simultaneous execution as detailed in the foregoing description, and with variants as described in the foregoing referenced related applications. Compared to super-scalar machines of the kind described by N. P. Jouppi in "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," IEEE Transactions on Computers, vol. 38, no. 12, Dec. 1989, pp 1645-1658, a compound instruction implementation may result in higher speed because

1. the grouping of instructions for parallel execution is optimized for hardware utilization;

2. the text is preprocessed, allowing relief of some interlocks, e.g., memory interlocks; and

3. with multiple functional units, each of which can perform multiple sequential operations in a single cycle, performance improvement is obtained due to parallel instruction execution and register dependency collapsing as was described in connection with FIGS. 1-10 and in Scalable Compound Instruction Set Machine (SCISM) Architecture application, U.S. Ser. No. 07/519,384.

The process of compounding adds the grouping information to the instruction stream for presentation to the CPU for execution. The original contents of an instruction are not destroyed but they are to be outputted as a program which will execute on a target machine, either immediately, or in stored form, as a compound instruction program. The question which needs solving is how compounding can be achieved with various possible architectures with different attributes.

In architectures with variable length instructions it is not possible, in general, to determine instruction boundaries by examining a byte string. Further complications for compounding occur when the architecture allows data and instructions to be intermixed, and allows modifications to the instruction stream. At execution time, instruction boundaries must be known to allow proper execution. Since compounding can take place before instruction execution, a system is needed to compound instructions without knowledge of where instructions start and which bytes are data. This invention describes such an apparatus. In CISC architectures, such as S/370, as detailed in the Publication of IBM No. SA22-7200-0, entitled IBM Enterprise Systems Architecture/370 Principles of Operation, for example, these problems are present. In RISC architectures which were first developed by IBM and then continued in development at Berkeley and elsewhere with various commercial implementation, e.g. RISC 6000, a product of IBM, as described for example in 1985 by D. A. Patterson, "Reduced Instruction Set Computers,", Communications of the ACM, vol. 28, no. 1, Jan. 1985, pp 8-21, instructions are usually a constant length and data are not mixed with instruction; thus, the compounding is simpler. While we have given examples which are applicable to S/370 to demonstrate the proposed apparatus, the proposed apparatus is applicable to other architectures. In addition, our examples describe solutions for differing and partial references and we will also describe a backward compounding apparatus.

There are three difficulties with S/370 instruction set architecture that make the compounding process complex. First, writes to the instruction stream may make a previous compounding invalid. A simple solution is to invalidate the instruction (or instruction buffer, or instruction cache line, etc.). The compounding process could then be repeated on the invalidated text or the text could be executed without compounding. Since writes to the instruction stream usually cause performance degradation, due to maintaining data consistency, additional degradation due to execution without compounding may be acceptable. Some optimizations of this are possible with a cache, as has been detailed in U.S. Ser. No.

07/522,219 filed May 10, 1990, entitled "Compounding Preprocessor for Cache." A second S/370 difficulty is that instructions and data can be intermixed. This invention describes the problem and proposes a solution so that the byte string can be compounded correctly without any additional information. The third difficulty in S/370 is that instructions have variable length. There are only three possible lengths (2, 4, or 6 bytes) which are indicated in the first two bits of the opcode of an instruction. Because the lengths are not fixed, the beginning of an instruction in a string of bytes cannot be identified even if it it known that the string contains only instructions. The apparatus described here provides a solution to this problem also.

Hereinafter we detail general solutions to the above problems. For simplicity, the special case of compounding a pair of instructions is described in detail. Compounding more than two instructions is an extension of the techniques and algorithms used by the system and is also discussed. There are a number of possible locations in a computer system where compounding occurs, designated the Instruction Compounding Unit (ICU). A discussion of these issues is found in U.S. Ser. No. 07/543,464 entitled "An In-Memory Preprocessor for a Scalable Compound Instruction Set Machine Processor", for example. However the system techniques and algorithms described here are also applicable to other architectures. In fact, they become simpler when the architecture has a single length for all instructions or when instructions and data cannot be mixed, e.g., RISC.

Compound Instruction Representation

In the context here, compounding information is added to the instruction stream as one bit for every two bytes of text (instructions and data). In general, a tag containing control information can be added to each instruction as was detailed with respect to FIGS. 1-10 and as was described also in U.S. Ser. No. 07/519,384 referenced above, as an example. Here, "compounding bits", refer to that part of the tag used specifically to identify groups of compound instructions. The case of compounding at most two instructions uses the following procedure to indicate where compounding occurs. Since all instructions are aligned on a halfword (2-byte) boundary, and lengths are either 2, 4, or 6 bytes, one compounding tag is needed for every halfword. A one-bit tag is sufficient to indicate compound or not compound. Specifically, a 1 indicates that the instruction that begins in the byte under consideration is compounded with the following instruction. A 0 indicates no compounding. The compounding bit associated with halfwords that do not contain the first byte of an instruction is ignored. The compounding bit for the byte of the second instruction in a compound pair is also ignored. As a consequence of this, only one bit of information is needed by the CPU during execution to appropriately execute compounded instructions. When the system allows more than two instructions to be compounded, a different method of representing the compounding information is needed. This is discussed in the section describing extensions for more than two instructions.

How To Work With Instruction Pairs

There are a number of possible variations on the basic compounding depending on what information is available. The cases are distinguished by the known content of the text and whether instruction boundaries (reference point) are known. The following cases are considered.

1. The text contains instructions only and the reference point is known.

2. The text contains instructions and data at known boundaries.

3. The text contains instructions only but reference point is unknown.

4. The text contains instructions and data with unknown reference point.

5. The text contains instructions and data with partial reference point information.

Case 1 and Case 2

In the simplest case, a byte string is to be compounded when it is known that there are no data bytes in the instruction stream, and when the location of the opcode of the first instruction is known. This may be the case in a modern architecture, as the result of a specific compiler, or as a useful assumption in a shoer section of text. The instruction to which each byte belongs can be determined precisely in all instances by using the length bits in the opcode and proceeding sequentially through the byte string. Instructions can be compounded by moving from one instruction to the next while the compound rules are being checked.

Assuming two-way instruction compounding for machines which can execute instructions in pairs and in parallel, instructions can be examined in pairs with compounding rules applied. FIG. 11 is an example of the best-case compounding where there is a specific length known. A preferred embodiment will handle this case, as well as mixed and worst cases. For this case, the Instruction Compounding Unit (ICU) can identify each instruction by examining the instruction length code (ILC) contained in each instruction opcode. Tag bits are generated with a 1 indicating the first of a compound pair. Halfwords that do not contain the beginning of an instruction are initialized to 0 and are not modified. For simplicity, it is assumed in this example that two-byte and four-byte instructions are compoundable for the target architecture and six-byte instructions are not. This depends upon the target machine and applicable compounding rules which are not here described in detail, as specific rules used to determine which instructions can be compounded depends on the instruction set architecture and the CPU design. If two instructions can be compounded, the tags 1 and 0, respectively; the following two instructions are then considered. If however, the present two instructions cannot be compounded, the tag of the first instruction is 0; the second and following instruction are then considered for compounding, and the system advances sequentially in this event during examination of the byte stream of incoming data.

A slightly more complex situation is where instructions are mixed with non-instructions, but every halfword is identified as either containing the first byte of an instruction or not (perhaps with tags). Again every byte can be identified and compounding is straightforward. This identification bit could be produced by a compiler. Issues such have been detailed in U.S. Ser. No. 07/543,464 can be examined in this context. The difference here is that compounding non-instructions is not attempted. If extra time is required to skip over non-instruction bytes, the system will be slower by an amount proportional to the amount of non-instruction bytes in the byte stream being examined. Compounding provided by the output however is not incorrect, and the instruction processor will execute the code,, albeit more slowly.

Case 3: Compounding instructions without reference point

When it is known that a byte stream contains instruction bytes only, but it is not known where the first instruction starts, compounding becomes more complex. Since the maximum length instruction is 6, but aligned on a 2-byte boundary, there are three possible starting points for the first instruction in the stream. Since it is known that only instruction are present in this stream, a simple step algorithm for the system is to start at byte 0, as if it were the beginning of an instruction and proceed with compounding. Then start at byte 2 and compound the stream; finally, start at byte 4 and compound the stream.

FIG. 12 shows an example where three different sequences of instructions are considered. For each sequence, instruction boundaries are determined and compounding bits assigned as in the best-case description. The three sequences of potential instruction boundaries (FIG. 12b) each produce a different sequence of compounding bits. Given that this system which is preferred employs an algorithm which requires three bits for every two text bytes, it is desirable to reduce the three sequences to a single sequence of bits so that the number of bits is no greater than for the best case. Since the only information required is the maximum number of instructions that for a given byte compose a compound instruction and given that for two instructions compound instructions the maximum is equal to the logical OR, the three bit sequences can be logically ORed to produce a single sequence (FIG. 12c).

The composite compounding bits of FIG. 12c are equivalent to the three sequences of FIG. 12b. Consider the "cc-vector" in FIG. 12c. If the bytes beginning at byte 0 are Considered for execution either because of sequential execution or branching, the processor fetches the instruction and the tag bits. Since the compound bit is 0, the instruction is executed as a single instruction. The next instruction, beginning at byte 6 is then considered for execution. The CPU will fetch the instruction and tag. Since the compound bit is 1, a second instruction is also fetched. (Its compound bit is ignored.) The two execute simultaneously. Note that this sequence of instruction is the first sequence shown in FIG. 12b. Because it is not known where the actual instructions boundaries are, it is possible that byte 2 starts an instruction, rather than byte 0. When byte 2 is considered for execution the instruction and tag are fetched. The compound bit is 1, so another instruction is fetched and the two are executed as a pair. This execution corresponds to the second sequence in FIG. 12b. Likewise if the first instruction starts at byte 4, the third sequence is executed. The composite bits allow any of the three possible sequences to execute with compounding. If a branch to byte 8 occurs, byte 8 must begin an instruction. Otherwise there is an error in the program. The tag associated with byte 8 is used and sequential execution can proceed. This allows the possibility of multiple valid compounding bit sequences which are selected when addressed by a branch target. It may even be useful to add this capability to the best-case, and the most complete embodiment includes this algorithm.

It is possible that the three different sequences of potential instructions will converge into one unique sequence. The rate of convergence depends on the specific bits; there are cases where no convergence will occur. At first, the convergence to the same instruction boundaries could occur with compounding out of phase. This will be corrected at the first non-compoundable instruction or earlier. In FIG. 12 note that the three sequences converge on instruction boundaries at byte 8. Also note that if additional sequences were started at 6, 8, 10, etc. they would also converge quickly. Sequences 2 and 3, while converging on instruction boundaries at byte 4, are out of phase in compounding until byte 16. That is, the two sequences consider different pairs of instructions from the same sequence of instructions. Byte 16 starts a non-compoundable instruction. (If the compounding algorithm looked at more than two instructions at a time, they might have converged sooner by choosing the same optimal pairings.) Because of this convergence, the compounding algorithm for this case can have a front end that tries three different sequences. When convergence is detected, the number of sequences collapses to two then to one. From this point, the best-case algorithm can be used. The convergence detection will be slower than the best case described for known length architectures by a factor equal to the number of active sequences, if a single compounder is used. If convergence is fast, the asymptotic compounding rates will be equivalent. Given that the convergence rate is data-dependent statistics can be gathered about probabilities of various convergence rates. However, no upper limit can be placed on the number of bytes before convergence occurs, e.g., the length sequence (4,4,4, . . .).

Case 4: Worst-case compounding

The worst case that can occur is to have a byte stream of mixed instructions and data and to have no knowledge of where any instruction begins. This could occur when compounding a page in memory or in an instruction cache when the reference point is not known (see below). The method to handle this case begins like the instruction-only case but there is one important distinction. If convergence occurs, a new sequence must be started in place of each sequence eliminated by convergence. This is because convergence could occur in a byte containing data; consequently, all three sequences could converge to the wrong sequence of "instruction" boundaries. This could be corrected when a sequence of real instructions is encountered, in the same way that convergence occurs. (If text contains instructions only, eventually all compounding sequences converge to the same instruction boundaries, as in FIG. 12.) The resulting sequence would still execute correctly, but fewer compound pairs would be detected and CPU performance would decrease. See U.S. Ser. No. 07/519,382 filed May 4, 1990, entitled "General Purpose Compounding Technique For Instruction-Level Parallel Processors."

A solution is to start a sequence at every halfword. As in the best-case, two instructions are examined and compound bits are determined. This is repeated starting two byes later. The example in FIG. 13 assumed in the example of FIG. 12, the composite sequences in FIGS. 12 and 13 will result in identical program execution.

The worst-case is accommodated by a system which has provision for examination of more possible instruction sequences than the best-case or instruction-only cases. This implies more time and/or more compounders to produce the tags. Since no information is known about the use of the text, this is necessary to achieve maximum compounding. If less compounding is acceptable, modifications can be made to the above sequences.

Case 5: Mixed cases

In an actual implementation the ICU could be given any of the above cases, depending on where the ICU is located, the instruction set architecture, and the compiler. An interesting case is that of compounding instructions in the cache, as we prefer where compound bits could be retained in cache. Typical instruction cache hit ratios are high as recognized by others, see A. J. Smith "Cache Memories," Computing Surveys, vol. 14, no. 3, Sept. 1982, pp. 473, so it will be now appreciated that there will be much reuse of instructions that have already been compounded and will not have to be compounded again. This contrasts with compounding in an instruction buffer or stack where the compounding information is lost after the instruction is executed, even though even that may be employed in some cases. In S/370, instructions and data can be mixed, but the cache has some knowledge of which bytes are instructions, because the CPU can specify these bytes on an instruction fetch. In order to execute quickly, it has been proposed that on a cache miss or a branch to a part of a line not yet compounded, that a few (perhaps 1 or 2) instructions be sent to the CPU without attempting to compound the instructions according to U.S. Ser. No. 07/522,219 referenced above. The ICU compounds such instructions and saves the result. The ICU then begins compounding at the next instruction with knowledge of where that instruction begins. The ICU processes instructions at least as fast as CPU execution; consequently, from that point the CPU receives compound instructions (until the next cache miss or branch).

There are a few different situations that can occur under this scenario. If ICU and CPU process instructions at the same rate, ICU will remain slightly ahead of the CPU (until a branch to an uncompounded portion of a cache line or cache miss). Should ICU encounter data, it will compound the bytes as if they were instructions. Since they cannot be executed, the program will execute correctly. If the number of data bytes is small and followed by instructions, a few instructions could be compounded incorrectly until the ICU realizes that the CPU has branched around the data. However, the compounding information produced would still lead to correct execution. When the ICU determines that the CPU has branched, there are several possible actions that could be taken for those bytes which the ICU now knows were not executed. It is assumed that the ICU maintains information about which bytes have been examined for compounding and which have not.

1. The bytes not executed can be marked "not examined" in which case they will be re-examined should the code be executed later.

2. The bytes can be left in an "examined" state. The maximum compounding may not be achieved but the compound bits are correct, although they might be ignored at execution.

3. The bytes can be left in a "partially examined" state which is equivalent to one sequence of the worst-case compounding algorithm. Other sequences can be examined later.

The choice of the algorithm depends on hardware considerations and program characteristics.

Because the CPU execution rate could be slowed by data cache misses, microcoded instructions, or pipeline stalls, it is likely that the ICU could get well ahead of the CPU. The distance is dependent on characteristics of the code. If taken branches are infrequent, the distance could be large. Since the ICU is ahead of the CPU there is always a chance that the ICU will lose its reference point to instruction boundaries due to the presence of data. The further ahead it is, the greater this possibility. The situation can be handled as above, but the performance impact may be more significant as a greater number of bytes might be compounded using incorrect instruction boundaries. A solution is to slow the compounding rate. The worst-case compounding achieves this while at the same time examining all actual instructions for compounding. When ICU gets ahead of the CPU by more than a certain threshold, the ICU switches to worst-case solution program and multiple examination elements, until the CPU catches up. The value of the threshold is an implementation decision.

Should the CPU be executing from previously compounded instruction, the ICU is not needed. An implementation option is to use the ICU to compound other sections of code, in the event that they may be executed the future. The sections of code might be:

*Unexecuted parts of fetched cache lines, *Prefetched cache lines,

*Previously executed instructions at branch targets that were sent directly to the CPU without compounding, or

*Sequential bytes following a taken branch which may have been compounded but, since they were not executed, cannot be distinguished from data.

When compounding parts of lines that have yet to be executed, the worst-case system should be used, since the ICU has no knowledge of the use of the bytes it is examining.

Possible ICU Organizations

There are many possible designs for the ICU depending on its location and the knowledge of the text contents. In this section a logical description of one possible implementation is described. FIG. 14 shows three compounders used to implement the worst-case algorithm. The number is arbitrary, and could be as large as the number of halfwords in the text buffer. The compounders begin at bytes 0, 2, and 4, respectively. Upon finishing with a possible instruction sequence, the offset by 6 bytes from the previous sequence. Each compounder keeps an internal state of tags and produces compound bits for each halfword in the text. The three sequences are ORed and the resulting composite sequence is stored with the text.

Consider, again, the cache example. An implementation might have multiple compounders as illustrated by FIG. 14, where element 160, 161, and 162 comprise separate compounders for sequences identified by the instruction facility examination. It will be noted that the output of these compounders is also coupled to logical OR element 164. One would compound using the best-case length count, staying ahead of the CPU. Another would use the worst-case system (which can be in combination with the best case) and compound other parts of the same cache line. For example, the second compounder could start at the end of the line and compound the line backwards.

Backward Compounding Apparatus

FIG. 15 illustrates backwards compounding. Beginning at the last halfword, the instruction length code of three previous halfwords are checked for consistent instruction boundaries. If the two could be sequential instruction, the compounding bits are produced according to the usual rules. The system process is provided with the sequential apparatus which repeats itself moving backward one halfword. Compounding bits are again ORed. In FIG. 15 the potential instruction at byte 24 is checked. Only byte 22 is a possible predecessor instruction. When byte 22 is checked, bytes 20, 18, and 16 are all possible predecessor instructions due to the length 2, 4, and 6, respectively. When checking byte 20 no possible instructions are found. How

To Compound More Than Two Instructions

The minimum amount of information needed to indicate the number of instructions compounded is the log of the maximum that can be compounded. If the maximum is two, one bit is needed for each compound instruction. If the maximum is four, two bits are needed because three, two, one, or zero instructions might be compounded with a given instruction. As described previously, each halfword needs a tag, but the CPU ignores all but the tag for the first instruction.

Consider again the worst-case algorithm, but with four-way compounding; refer to FIG. 16. A byte is examined as if it were an instruction. If it cannot be compounded, its compounding bits are 0. If it can be compounded with one other instruction, the compounding bits are 1, for the first instruction, and 0 for the second. Likewise if three are compounded, the compounding bits are 2, 1, and 0, respectively. This method assumes that if instructions A, B, and C can be compounded as a triplet, B and C can be compounded as a pair. This appears to be a valid assumption. Hence, should a branch to B occur, B and C will execute as a compound pair.

As before, the bytes beginning at each halfword must be examined for potential instruction boundaries. If more information is available, the instruction-only or best-case algorithm could be used. Each examined sequence produces a sequence of compounding bits. The composite sequence is formed by taking the maximum value of the individual compounding bits produced by the sequences. When a compound group is executed, the CPU ignores all compound bits associated with bytes other than the first byte of the group. The compound bits indicate the number of instructions which make up the compound group.

Depending on the actual compounding rules used there may be some optimizations for this algorithm. For example, in FIG. 16b, the fifth sequence (starting at byte 8) assumes instructions of lengths 2, 4, 2, and 6. Since six-byte instructions are never compoundable in this example, there is no benefit in attempting to compound starling at the other three instructions (bytes 10, 12, 14), since they have already been compounded as much as possible. Due to the brevity of this example, no potential sequences starting after byte 14 are examined.

To reduce the number of bits to transfer, there may be alternative representations of the compound information. When a compound instruction is requested from the cache, the compound bits could be translated into a different format. For example, One bit per instruction with the encoding: "1" means compound with next instruction and 0 means do not compound with next instruction. A group of four compounded instructions would have the compounding bits (1,1,1,0). In the preferred embodiment, the group of four compounded instructions would have compounding tags for instruction member units represented as having a value of 3,2,1,0. Compounding bits associated with halfwords not containing opcodes are ignored.

While a general purpose apparatus for compounding groups of instructions has been presented, we prefer to provide the system with an apparatus that operates in the worst-case situation of no knowledge of the use of the bytes under examination. When some knowledge of the text content is available, the apparatus can be simplified, as in the more simplified architectures we discussed. An extension to the basic length count can be used when the maximum number of instructions in a compound group is greater than two.

Clearly, the inventions we have described by way of example and in illustration of the best mode for practicing the inventions provide a basis for much potential growth in processor performance. Accordingly, it will be understood that those skilled in the art after reviewing our presently and alternative embodiments, that various modifications and improvements may be made, now and in the future, as those skilled in the art will appreciate upon the understanding of our disclosed inventions. Such modifications and future improvements are to be understood to be intended to be within the scope of the appended claims which are to be construed to protect the rights of the inventors who first conceived of these inventions.

* * * * *

United States Patent	***5,446,850***
Jeremiah , et al.	**August 29, 1995**

Cross-cache-line compounding algorithm for scism processors

Abstract

A system for compounding instructions across cache line boundaries transfers an instruction line from a relatively slow memory to a instruction compounding unit if there is a miss for an instruction in that line in the instruction cache. At the same time the numerically preceding instruction in cache is transferred to the instruction compounding unit and instructions from the two lines are compounded. If a numerically preceding cache line has been compounded with a cache line that has been deleted and then replaced, compounding tags for the numerically preceding cache line are deleted.

Inventors:	**Jeremiah; Thomas L.** (Endwell, NY); **Blaner; Bartholomew** (Newark Valley, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**281321**
Filed:	**July 27, 1994**

Current U.S. Class:	**712/215**; 711/118; 711/125
Intern'l Class:	G06F 012/08
Field of Search:	395/375,800,250

References Cited [Referenced By]

U.S. Patent Documents

5197135	Mar., 1993	Eickemeyer	395/375.
5214763	May., 1993	Blaner	395/375.

Primary Examiner: Eng; David Y.
Attorney, Agent or Firm: Augspurger; Lynn L., Marhoefer; Laurence J.

Parent Case Text

RELATED APPLICATIONS UNDER 35 U.S.C. 120

This is a Continuation of application No. 07/875,507, filed Apr. 29, 1992 now abandoned. This application is entitled to claim and claims priority from the following applications of which it is a continuation-in-part: application Ser. No. 07/642,011, filed Jan. 15, 1991, entitled "Compounding Preprocessor for Cache", the inventors being Bartholomew Blaner et al. now U.S. Pat. No. 5,295,249; and application Ser. No. 07/677,685, filed Mar. 29, 1991, entitled "System for Preparing Instructions for Instruction Processor and System with Mechanism for Branching in the Middle of a Compound Instruction", the inventors being S. Vassiliadis et al. now U.S. Pat. No. 5,303,356.

FIELD OF THE INVENTION

These invention relate to high-speed computers and computer systems and particularly to computer systems which preprocess instructions for the purpose of appending control information which is placed in a cache along with the instructions and subsequently used to control execution of the computer system.

CROSS REFERENCE TO APPLICATIONS

The present application related to the following patent applications:

(1) application Ser. No. 07/519,384, filed May 4, 1990, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al now abandoned;

(2) application Ser. No. 07/519,382, filed May 4, 1990, entitled "General Purpose Compound Apparatus For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al now abandoned;

(3) application Ser. No. 07/522,219, filed May 10, 1990, entitled "Compounding Preprocessor For Cache", the inventors being Bartholomew Blaner et al now U.S. Pat. No. 5,035,378; and

(4) application Ser. No. 07/543,464, filed Jun. 26, 1990, entitled "An In-Memory Preprocessor for a Scalable Compound Instruction Set Machine Processor, the inventors being Richard Eickemeyer et al. now abandoned

(5) application Ser. No. 07/543,458, filed Jun. 26, 1990, entitled "Memory Management for Scalable Compound Instruction Set Machines with

(In countries which publish after eighteen months from the priority date corresponding applications related to the above five U.S. applications have been filed.) now U.S. Pat. No. 5,197,135

(6) application Ser. No. 07/619,868, filed Nov. 28, 1990, entitled "Overflow Determination for Three-Operand ALUS in a Scalable Compound Instruction Set Machine", the inventors being Stamatis Vassiliadis et al now U.S. Pat. No. 5,301,341; and

(7) application Ser. No. 07/642,011, filed Jan. 15, 1991, entitled "Compounding Preprocessor for Cache", the inventors being Bartholomew Blaner et al., an application from which priority is claimed now U.S. Pat. No. 5,295,249; and

(8) application Ser. No. 07/677,066, filed Mar. 29, 1991, entitled "System for Compounding Instructions for an Instruction Processor with Different Attributes with Apparatus for Handling Test and Data with Differing Reference Point Information and Backward Compounding Apparatus for Compound Instructions", the inventors being Eickemeyer et al now abandoned; and

(9) application Ser. No. 07/677,685, filed Mar. 29, 1991, entitled "System for Preparing Instructions for Instruction Processor and System with Mechanism for Branching in the Middle of a Compound Instruction", the inventors being S. Vassiliadis et al. now U.S. Pat. No. 5,305,356, an application from which priority is claimed.

These applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y.

The descriptions set forth in these applications are hereby incorporated into the present application by this reference. These descriptions have been published in related applications filed in other countries as of approximately 18 months after the U.S. filing date.

Claims

What is claimed is:

1. A data processing system in which instructions are transferred in blocks called instruction lines from a relatively low speed memory to a relatively high speed cache memory and from which cache memory instruction lines are fetched for execution and are deleted a line at a time, said system including an instruction compounding unit in which instructions are processed in order to generate tag information that indicates instructions that can be executed in parallel, said data processing system comprising in combination:

means to address a first instruction line in said cache memory in order to transfer said first instruction line from said cache memory to an instruction fetch unit;

means to generate a miss signal if said first instruction line is not resident in said high speed cache memory;

means responsive to said miss signal for transferring said first instruction line from said relatively low speed memory to said instruction compounding unit;

means responsive to said miss signal for determining an address of a second instruction line that is next in succession to said first instruction line for transfer to said instruction fetch unit;

means for transferring said second instruction line to said instruction compounding unit if said second instruction line resides in said cache memory;

said instruction compounding unit processing instructions from said first instruction line and said second instruction line in order to generate tag information indicating an instruction in said first instruction line that can be executed in parallel with an instruction in said second instruction line

means for determining if an instruction line in said cache memory has been deleted and replaced prior to its execution;

means to determine if said deleted and replaced instruction line contains instructions that have been compounded with another instruction line in said cache memory; and

means to delete tag information that indicates an instruction in said another instruction line can be compounded with an instruction in said instruction line that has been deleted and replaced.

2. A data processing system as in claim 1 wherein said second instruction line is transferred to said instruction compounding unit during an interval when said first instruction line is transferred from said relatively low speed memory to said instruction compounding unit.

3. A processing system as in claim 1, wherein said means for determining an address includes means to decrement the address of a missed instruction.

4. A processing system as in claim 1, wherein said means for determining an address includes means to decrement the address of a missed instruction.

5. A processing system as in claim 1, wherein said means for determining an address includes means to decrement the address of a missed instruction.

Description

BACKGROUND OF THE INVENTION

U.S. Pat. No. 5,051,940, issued Sep. 25, 1991, to S. Vassiliadis et al., entitled: "Data Dependency Collapsing Hardware Apparatus," is one of several prior developments in the art related to a SCISM processor, a high speed computer which is enabled by compounding and compounding apparatus, to provide parallel performance of systems which can process instructions and data for programs which could be handled by older architectures, but which can also be handled by newer architectures which employ the Scalable Compound Set Machine Architecture which was introduced in the description of

U.S. Pat. No. 5,051,940 and in the above referenced applications.

In high speed computers, it is desirable; to reduce the time required to complete, or execute, each instruction in order to improve performance. This is typically done by clocking the processor at the maximum rate that can be sustained by the underlying circuitry, or by reducing the average number of clock cycles needed to complete instruction execution through some form of parallel operation. One such form of parallelism well known in the art is pipelining, wherein instruction execution is subdivided into a-number of specifically defined steps related to various areas of logic, or pipeline stages, in the processor. As one instruction completes its activity in a given pipeline stage, it is sent to the next stage, and a subsequent instruction can then make use of the stage vacated by the instruction ahead of it. Thus, several instructions are typically being executed simultaneously in such a computer system, but each instruction is dispatched for the execution process one at a time. More recently, in order to further improve performance, computer designs have been developed wherein multiple instructions may be simultaneously dispatched for execution, provided such instructions do not conflict with each other while being executed. Sufficient hardware must be provided so that the instructions which simultaneously occupy a given stage in the pipeline can execute without interfering with each other. Typically, the instructions are processed through the pipeline together and are completed simultaneously, or at least in conceptual order. This mode of execution has been given the name superscalar execution.

One of the difficulties which typically must be addressed in superscalar processor design is making the decision whether multiple instructions may in fact be simultaneously executed. In most cases, the superscalar designs will not be able to simultaneously execute any and all possible combinations of instructions due to interdependencies between some instructions, and perhaps some limitations of the underlying hardware. Therefore, as instructions reach the point where execution is to begin, a decision must be made whether to permit parallel execution, or default to single instruction execution mode. The decision is usually made at the time instructions enter the pipeline, by logic circuits which decode the instructions to detect whether conflicts actually exist. Depending on the particular instruction set architecture, the decoding process may be relatively complicated and require a large number of logic stages. This can reduce performance either by increasing the cycle time of the processor, or by requiring an additional pipeline stage to perform the aforementioned decoding process, either of which will reduce performance.

SCISM application Ser. No. 07/519,382 provides a solution for the problem of delay caused by the need to analyze instructions for superscalar execution through the expedient of preprocessing the instruction stream and making a determination of groups of instructions suitable for superscalar execution. These groups of instructions are called compound instructions, and are composed of the original instructions and an associated tag which indicates whether parallel execution is permitted. SCISM application Ser. No. 07/522,291 proposes an Instruction Compounding Unit, or ICU as a means of performing the instruction compounding analysis required by Scalable Compound Instruction Set Machines (SCISM). Instructions are analyzed by the ICU as they are fetched from memory and placed in a cache. The ICU forms the tag, which is logically stored along with the instructions in the cache. Certain problems arise, however, when the ICU concept is applied to S/370 and related architectures. In particular, portions of cache lines that have not or cannot be analyzed for compounding may result.

U.S. Pat. No. 5,051,940 has provided a solution for this problem to a large extent using what is termed the worst-case compounding algorithm. With this algorithm, the contents of a cache line, be it instructions, data, or instructions mixed with data, may be analyzed for compounding in its entirety without regard to any instruction boundaries within the cache line. Still, the problem of compounding across cache line boundaries, or cross-line compounding, remains. An instruction can only be compounded with a subsequent instruction if the subsequent instruction is available for analysis at the time the compounding process occurs. Instructions situated near the end of a cache line may not be considered for compounding unless the next sequentially addressable cache line is also present, and therefore typically are ineligible for parallel execution, thereby decreasing processor performance.

The degree to which performance is compromised depends on a number of circumstances, such as cache line size and the frequency of execution of particular sequences of instructions. Larger cache line sizes reduce the percentage of instructions which reside adjacent to cache line boundaries, but there is usually an optimum upper bound on cache line size that if exceeded, will decrease performance due to excessive storage accesses for unneeded data. Frequency of instruction execution is typically not correlated with cache line boundaries, and it is perfectly possible for a performance-critical loop in the instruction stream to sit astride a cache line boundary. This effect can contribute to unpredictable and unsatisfactory performance.

In application Ser. No. 07/522,291, the inventors suggest cache line pre-fetching as a means of facilitating cross-line compounding. However, cache line prefetching creates other problems, two of which are set out here.

1. Room must be made in the cache for the prefetched line, possibly causing a soon-to-be-needed line to be removed from the cache in favor of the prefetched line, which may in fact, never be used, resulting in decreased processor performance.

2. Depending on the processor busing structure, prefetching may require occupation of the processor data bus while the line is being prefetched. Consequently, the processor's execution units may be blocked from using the bus while the fetch is in progress. Any such blockage results in decreased performance.

It is desirable to provide a means for allowing compounding of instructions across cache line boundaries without the requirement to prefetch cache lines.

SUMMARY OF THE INVENTION

The improvements which we have made achieve an enhancement in cross-cache-line compounding that makes use of existing cache lines, i.e., does not require that new cache lines be prefetched, and therefore displaces no lines from the cache; and can be accomplished while a missed cache line is being fetched from the next level in the storage hierarchy, thus requiring no additional processor time beyond that required to process the cache miss and analyze the line for compounding. Thus, the operation of the algorithm can be substantially overlapped with normal cache miss and ICU operation, causes no undesirable side effects in the cache, and, therefore, exhibits an improvement over prior techniques.

These improvements are accomplished by providing means for accessing a cache and its associated directory for a second cache line, following a cache miss for a first cache line, during the period the cache is waiting for the first cache line to be returned from the next level in the memory hierarchy. The address used to access the cache and directory during this interval is that of the immediately preceding cache line.

An address decrementer is provided, along with multiplexer means to allow the decremented address to be used to access the cache and directory. Further, means are provided to save the result of the cache and directory access for the second cache line until such time as they are required in the instruction compounding process. The directory access will indicate whether the second cache line resides in the cache, and this indication will control whether or not cross-cache-line compounding is enabled. The portion of the second cache line closest to the end of the line is saved in a buffer and used by the ICU to create compound instructions which may cross the boundary between said second cache line and said first cache line.

Certain computer architectures which permit instruction stream modification, either by the processor itself or other processors in a multiprocessor configuration require a further step in the cross-line compounding process to avoid erroneous results. In such an environment, it is possible for a cache line that has previously been compounded with the numerically preceding cache line to be deleted from the cache and then later refetched in modified form, in which case, the compounding information contained in the numerically preceding cache line could be incorrect. Deletion of a cache line is most often caused by the need to make room for new cache lines, but it can also be caused by various cache coherency protocols commonly employed in multi-processors. A buffer provides a means for saving the address tag of the line being deleted from the cache during the access for said first cache line, for decrementing it appropriately, and then for comparing it to all address tags read out from the directory during the access for said second cache line. A determination is made whether the line; being deleted sequentially follows any of the lines identified during the second directory access. If the sequentially previous line is present, its compounding bits that were derived based on instructions in the line being replaced are reset. If the sequentially previous line is not present, no action is necessary. Furthermore, if some form of instruction buffer exists between the cache and the point at which compound instructions are dispatched into the pipeline, it is possible that the buffer may contain some portion of the cache line whose compounding bits must be reset. In this case, it will be necessary to delete the erroneous compounding information from said instruction buffer, or, alternatively as a simplification, to delete all compounding information from the buffer whenever any line is replaced in the cache.

These and other improvements are set forth in the following detailed description. For a better understanding of the invention with advantages and features, reference may be had to the description and to the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows schematically an overview of the preferred embodiment and particularly shows a cache operatively coupled to a memory system through an Instruction Compounding Unit. The output of the cache is connected to the instruction fetching means of the processor and provides compound instructions to the processor.

FIG. 2 shows a representative Instruction Compounding Unit.

FIG. 3 shows a representative cache line inpage operation with instruction compounding, starting with the next-to-last quadword in the cache line.

FIG. 4 shows a representative cache line inpage operation with instruction compounding, starting with the last quadword in the cache line.

FIG. 5 shows a representative cache line inpage operation with instruction compounding, starting with the first quadword in the cache line.

FIG. 6 shows a memory address apportioned into directory tag, cache line index, and byte index fields, using big-endian notation.

FIG. 7 shows a cache with associated address and data registers, and cache directory. The address decrementer means and multiplexer means in the preferred embodiment are also shown.

FIG. 8 shows a representative instruction buffer with associated latches which specify from where in a cache each buffer entry was fetched

FIG. 9 shows a representative Instruction Compounding Unit with additional register and multiplexer means necessary to support cross-line compounding operations.

FIG. 10 shows a sequence chart for a cache inpage operation starting with the next-to-last quadword in a cache line and additional steps necessary to perform cross-line compounding.

Our detailed description explains the preferred embodiments of our invention, together with advantages and features, by way of example with reference to the following drawings.

DETAILED DESCRIPTION OF THE INVENTIONS

Before considering our preferred embodiments in detail, it may be worthwhile to illustrated by way of example, the operation of a representative Instruction Compounding Unit (ICU) with reference to FIG. 1 which shows schematically the digital processing systems instruction compounding mechanism.

In FIG. 1, the ICU 12 is situated between MEMORY 10 and CACHE 14, where it processes instructions to form tag entries for TAG array 16.

An ICU is illustrated in FIG. 2. L2.sub.-- STG.sub.-- BUS is the 16-byte (quadword, QW) data bus connecting level-2 cache storage (L2) to the ICU and instruction cache. A QW from L2.sub.-- STG.sub.-- BUS is latched in L2REG. The low-order doubleword (DW) of L2REG is pipelined to L2BUL. An instruction cache line is assumed to be 128 bytes; thus, 8 L2.sub.-- STG.sub.-- BUS transfers are required to transfer the entire line. QWs in the line are numbered from 0 to 7, from the low-order address to the high. The line is assumed to be rotated so that the QW containing the instruction needed by the instruction execution unit is received first. For example, if the instruction required by the instruction execution unit is in QW6 of a given line, the line is received in the order QW6, QW7, QW0, OW1, QW2, QW3, QW4, and QW5.

The Compounding boxes (CBOX) perform the actual compounding analysis. The boxes are designed and organized to perform a worst-case compounding algorithm as described in application Ser. No. 07/519,384 (IBM Docket EN9-90-019) wherein each halfword (HW) in the line is assumed to start an instruction. Each CBOX produces a compounding bit (C bit) for the presumed instruction present on its I1 input. Eight HWs are processed per cycle, yielding C bits C0-C7. C0-C3 are latched at the end of the cycle in C0.sub.-- LTH through C3.sub.-- LTH. The complete set of C bits for OWn are given by C0.sub.-- LTH .parallel. C1.sub.-- LTH .parallel. C2.sub.-- LTH .parallel. C3.sub.-- LTH .parallel. C4 .parallel. C5 .parallel. C6 .parallel. C7 and are valid when QWnL.sup.1 is in L2BUL and OWn+1 (n modulo 8) is in L2REG. The sequential nature of the C bit production will become more apparent from the timing diagrams discussed below.

The suffix "H" refers to bits 0-63 (bytes 0-3); " L" refers to bits 64-127 (bytes 4-7). The symbol .parallel. represents bit concatenation.

The ICU is designed to correctly perform compounding for an arbitrarily rotated line, where "correct" means

1. up to the last three C bits for a line may be forced to zero (truncated), i.e., the C bits for the last 3 HWs of QW7, since compounding across cache lines is accomplished by other means described herein, and

2. if the line has been rotated, i.e., a QW other than QW0 is received first, then compounding analysis is performed for instructions lying on the boundary between the last and first QWs received.

In the above example, QW6H is saved in the S register so that when QW5 is received, instructions in QW5L may be compounded with those in QW6H.

A controlling finite state machine (ICU FSM) is required to properly sequence the compounding of an eight-QW cache line. The following signals are input from an instruction cache miss handling finite state machine:

DVALID If asserted, the QW on L2.sub.-- STG.sub.-- BUS is a valid datum.

FIRSTQW Asserted when the first QW of the bus transfer is on the L2.sub.-- STG.sub.-- BUS.

LASTQW Asserted when the last QW of the bus transfer is on the L2.sub.-- STG.sub.-- BUS.

EOL (End of Line) Asserted when QW7 is on the L2.sub.-- STG.sub.-- BUS.

In response to these inputs, the ICU FSM produces the following output signals:

LD.sub.-- L2 If asserted, load L2REG

LD.sub.-- S If asserted, load S; otherwise, hold S

LD.sub.-- L2BUL If asserted, load L2BUL and Cx.sub.-- LTH; otherwise, hold L2BUL and Cx.sub.-- LTH.

GT.sub.-- S.sub.-- L2H Gate S to L2REGH

TRUNCATE If asserted, zero the C bits for HWs 5, 6, and 7 of QW7.

CVALID If asserted, the contents of C0-3.sub.-- LTH .parallel. C4-C7 are valid C bits for the QW whose low DW is in L2BUL.

FIGS. 3, 4, and 5 illustrate the operation of the ICU for three representative rotations of the incoming line with contiguous transfer of all eight QWs. The QW notation is as before, with one addition: CQWn refers to the C bits for OWn.

Referring to FIG. 3, the ICU operation is as follows. Assume that an instruction cache miss has occurred and that QW6 is the required QW. In cycle 0, QW6 is on L2.sub.-- STG.sub.-- BUS, and DVALID is asserted. This causes FIRSTQW to be asserted, which, in turn, causes LD.sub.-- L2 to be asserted and also triggers the one-cycle assertion of LD.sub.-- S in cycle 1. LD.sub.-- L2 remains asserted until the cycle after GT.sub.-- S.sub.-- L2H has been asserted (cycle 8). In cycle 1, QW7 is on L2.sub.-- STG.sub.-- BUS. Since this is the last (right-most) QW of the line, EOL is asserted. The assertion of EOL triggers the assertion of TRUNCATE in cycle 3. Since LD.sub.-- S is asserted, QW6H will be latched in S at the end of cycle 1, where it will be held until cycle 9. LD.sub.-- L2BUL is also asserted. Thus, the low DW of L2REG will be latched in L2BUL at the end of the cycle. Also, C bits for QW6H (CQW6H) are produced by CBOXes 0-3 and are latched in their respective latches.

In cycle 2, QW7 is latched in L2REG. Now valid C bits can be produced for QW6L, which is now in L2BUL. These C bits are present on the outputs of CBOXes 4-7. Thus, C0-3.sub.-- LTH.parallel.C4-7 constitute the C bits for QW6, the low DW of which is now in L2BUL.

In cycle 3, valid C bits are available for QW7, now in L2BUL, but C5, C6, and C7 must be truncated. Thus, TRUNCATE is asserted in this cycle, which forces these bits to zero.

Production of C bits continues routinely until cycle 7 when LASTQW is asserted. In cycle 8, GT.sub.-- S.sub.-- L2H is asserted, causing QW6H (the high DW of the first QW received) to be gated into L2REGH at the end of the cycle. Simultaneously, QWSL is gated into L2BUL, as usual. Then, in cycle 9, valid C bits are produced for QW5. The operation is complete.

FIG. 4 illustrates the timing of ICU operations when QW7 is received first. The diagram is instructive in that it demonstrates ICU behavior in the event that FIRSTQW and EOL are asserted simultaneously. FIG. 5 illustrates ICU timing when the line is not rotated, i.e., QW0 is received first.

THE PREFERRED EMBODIMENT

Turning now to our invention in greater detail, it will be seen from FIG. 1 that instructions are fetched from memory 10 and brought into the cache 14 through Instruction Compounding unit 12 and subsequently made available to the processor for execution via Instruction Fetch Unit 20. Instructions are brought into the cache in blocks, or lines, and placed into the cache 14 according to mechanisms that are well known in the art, and consequently not reiterated here. Lines are mapped into the cache based on a certain portion of the memory address, the cache line index 220 in FIG. 6. Lines whose addresses differ in the cache line index portion of the address are said to reside in different congruence classes. Multiple lines may exist in the cache-in the same congruence class if the cache is so designed, and lines in the same congruence class are said to reside in different associativity classes. The particular design shown in FIG. 7 is that of a two-way associative cache. However, the invention is equally applicable to caches with greater associativity. As the instructions are brought in from memory 10, they pass through the Instruction Compounding Unit 12, or ICU, where individual instructions are analyzed and where possible, are aggregated into compound instructions according to one of the methods described in the references. The ICU produces a series of tag bits which are stored in tag array 16 which is accessed in parallel with the cache array, and correspond to instructions accessed from the cache array itself. Thus the cache and tag arrays, when accessed together, provide compound instructions for subsequent execution. Cache control logic 18 receives address and control information from instruction fetch unit 20 and provides address and control signals to cache 14, tag array 16, and memory 10.

Referring now to FIG. 7, the cache directory 118 is accessed using the cache line index 220 portion of the I-FETCH ADDRESS 110 to determine whether a particular cache line is present in the cache. LRU array 120, cache array 114, and compounding tag array 116 are also accessed at the same time using cache line index 220. The presence of a cache line is indicated by a match of a directory tag 210 portion of the address with one of the directory tags accessed from directory 118. The comparison is done by compare logic 124 and 126. If the line is present in cache array 114, the contents of the cache line, or a subportion thereof, are selected by mux 128 under control of the compare circuits 124 and 126. Selection of the correct compounding tag bits is accomplished by mux 130 using the same controls. Fetched instructions are passed to the processor and processing continues. If the particular line is absent, the directory compare circuitry will so indicate, causing a fetch to memory. Usually, this fetch will take several cycles, during which time the processor is stalled, and the cache merely waits for the requested data. In the present invention, the address of the line that will be replaced (if any) is saved in T0REG 136 or T1REG 138 for further use. Selection of the line to be replaced is made in this example by a Least-Recently-Used, or LRU indicator bit that is fetched from LRU array 120 and saved in LRU REG 134, in combination with a bit for each cache line (not shown) which is typically used to indicate whether a valid cache line has been previously fetched. The output of the LRU REG 134 and cache validity bits control selection of either T0REG 136 or T1REG 138 in MUX 144. The output of MUX 144 is the address tag of the line being replaced, assuming a valid line is in fact being replaced. The address tag is decremented by one in the low order position by decrementer 146.

Meanwhile, the address used to access the cache is decremented by one in the low order position of the CACHE LINE INDEX portion of the address 220 using decrementer 142. The BYTE INDEX portion of the address 230 is forced to all 1's to cause the subsequent access to fetch the last doubleword in the cache line. The decremented address is saved in A REG 140. Assuming a cache miss occurs, the cache is subsequently accessed using the decremented address in A REG 140 which is gated through MUX 122. The purpose of the second access is to ascertain whether the line preceding the one being simultaneously fetched from memory is present and if it is, to fetch and retain the last doubleword in the line. Determination of the presence of the previous line is made on the second cache access via COMPARE 124 and COMPARE 126 which operate in the same manner as they did on the original cache access, but this time with the decremented address from A REG 140 through MUX 122. Assuming a successful compare occurs, the last doubleword of the cache line accessed from CACHE ARRAY 114 and is selected in MUX 128 by the output of the COMPARE 124 or COMPARE 126. The selected doubleword is saved in DATA REG 132. The output of DATA REG 132 is sent back to ICU 12 for use at the appropriate step in the compounding process.

Also during the second access, the address tags read from CACHE DIRECTORY 118 are compared with the output of DECREMENTER 146 by COMPARE 148 and COMPARE 150. A successful compare at this point indicates the line being replaced in the cache by the new line may have been compounded across the line boundary, and that its sequentially previous line is still in the cache, necessitating the resetting of the compounding bits associated with the last doubleword of said previous line. TAG RESET CONTROL 152 uses the outputs of COMPARE 148 and COMPARE 150 to generate the necessary control signals to TAG ARRAY 116 to reset the appropriate compounding tags. One such control signal is an input of AND function 154, which consists of four AND gates in the preferred embodiment. Said input causes the tag data at the input of TAG ARRAY 116 to be in the reset state. Another control generated by TAG RESET CONTROL 152 enables the write control input of TAG ARRAY 116. The write control input is activated only if a compare occurs in COMPARE 148 or COMPARE 150, and the corresponding associativity class in TAG ARRAY 116 is enabled for resetting the compounding bits.

We have just described the process of accessing a cache for instructions which causes a cache miss, subsequently resulting in a second cache access to determine whether the cache line immediately preceding the one to be fetched from memory is present, and the process by which the last doubleword of the preceding line is saved for compounding with the beginning of the new line being fetched. We have also described the process by which compounding bits are reset if necessary for the cache line being replaced with the new line. We will now describe the process for resetting compounding bits for a cache line which immediately precedes a line which is invalidated.

In certain processor architectures, cache lines may be subject to removal from the cache for reasons other than making room for a new line. The removal process is frequently referred to as invalidation. An invalidation request is received by the cache control logic, along with the identity of the cache line to be removed. In FIG. 7, the identity of the line is provided on INVALIDATION ADDR bus 156. The address provided may consist of a complete address, or may only identify the congruence class and associativity slot in the directory. In the latter case, the normal procedure is to merely reset the validity bit identified by the congruence class and associativity slot directly. If a complete address is specified, the directory must first be accessed to determine whether the cache line is in fact present, and if present, in which associativity slot it resides. The validity bit associated with the line to be invalidated is then reset on a subsequent access.

In the preferred embodiment, a complete address is specified on INVALIDATION ADDR BUS 156, and CACHE DIRECTORY 118 is accessed in normal fashion. Directory tags are compared to the DIRECTORY TAG portion 210 of the address. DECREMENTER 142 decrements the address by one in the low order portion of the CACHE LINE INDEX 220 portion of the address, saving the result in A REG 140. It is not necessary to alter the BYTE INDEX 230 portion of the address. COMPARE 124 and COMPARE 126 identify whether the line to be invalidated is present. Successful compare signals are saved in SLOT REG 133 for use subsequently in selecting T0REG or T1REG via MUX 144.

Assuming the line is present, the directory is again accessed using INVALIDATION ADDR 156, this time resetting the validity bit for the appropriate line in the directory. On the following cycle, A REG 140 is gated through MUX 122 to access the directory. The address tags in CACHE DIRECTORY 118 are sent to COMPARE 148 and COMPARE 150 for comparison with the output of DECREMENTER 146. A match in either COMPARE circuit indicates the line immediately preceding the one just invalidated is also present in CACHE 118. TAG RESET CONTROL 152 generates the necessary control lines to TAG ARRAY 116 exactly as previously described for the case where the line was replaced by a new cache line.

If an instruction buffer exists in the particular computer employing the present invention, and if the instruction set architecture allows instruction stream modification, means must be provided to ensure that compounding bits previously fetched into the instruction buffer remain valid at all times. It is possible that the latter portion of a first cache line could have been fetched from the cache, along with its compounding bits, into the instruction buffer just prior to the time when a second, sequential line is removed from the cache. Since the second line may be subject to modification before being refetched, and the first line is already in the instruction buffer, its compounding bits which depend on the state of the second line may be incorrect.

In the preferred embodiment, the default value for the compounding tag bits is zero, indicating parallel instruction dispatch is not enabled. It is merely necessary to reset the aforementioned compounding bits in the instruction buffer whenever a line is deleted from the cache, either by replacement or invalidation. It is not necessary to remove the instructions themselves from the instruction buffer, since they are still valid and can be executed sequentially with minor performance degradation. Various strategies may be employed to determine which compounding bits in the instruction buffer to rest. It is desirable to avoid unnecessarily resetting compounding bits for performance reasons. At one extreme, the finite state machine which controls the cache may simply cause all compounding bits in the instruction buffer to be reset whenever a line is deleted from cache. This is the simplest approach, but results in the most performance degradation due to resetting compounding bits unnecessarily. At the other extreme, it is possible to identify within the instruction buffer exactly which address each instruction was fetched from, and only reset the compounding bits that depend on the line being deleted. Between the two extremes, various strategies may be employed to help identify entries in the instruction buffer whose compounding tag bits are subject to being reset.

FIG. 8 shows a representative instruction buffer having eight buffer locations with compounding tags 300 and three status fields LASTDW 310, CCL 312, and SLOT 314. In the preferred embodiment, each instruction location in the instruction buffer is provided with a latch, called LASTDW, which indicates whether it was fetched from the portion of any cache line, i.e., the last doubleword of the line, whose tag bits can be reset due to line deletion. The BYTE INDEX 230 portion of I-FETCH ADDRESS 110 is decoded for a binary value of `1111xxx` by decoder 320 to identify the last doubleword in the line. The decoder output is used to set the latches in the instruction buffer whenever an instruction fetch occurs and one or more buffer locations are being set. Upon performing a line deletion, the cache control finite state machine will signal the BUFFER TAG RESET CONTROL 350 to reset all compounding TAGs 300 whose LASTDW latch is ON. In this way, only the tags for instructions fetched from the last doubleword of the cache line are reset, significantly reducing the instances of unnecessarily resetting compounding tags.

If further performance improvement is desired, each location in the instruction buffer can be further identified by associativity slot or congruence class (or both) upon fetching, and that information stored in the instruction buffer along with the compounding tag. In FIG. 8, the congruence class is obtained from the CACHE LINE INDEX 220 portion of the I-FETCH address 110 and stored in the CCL 312 entry in the instruction buffer. The associativity slot is obtained from directory output COMPAREs 124 and 126 and stored in the SLOT 314 entry in the instruction buffer. Upon line deletion, SLOT0 through SLOT7 are compared with the associativity slot from SLOTREG 133 using COMPARE 330. CCL0 through CCL7 are compared with the decremented congruence class provided by DECREMENTER 146 in COMPARE 340. If a match occurs, the corresponding compounding tag bits are reset in the instruction buffer by BUFFER TAG RESET CONTROL 350. It is not necessary to save all congruence class address bits in the CCL entry in the instruction buffer. In order to reduce the number of bits that are saved, any number of congruence class bits can be saved, and matched with the corresponding bits from DECREMENTER 146. As the number of bits are reduced, the incidence of unnecessary tag bit reset operations increases, however.

We now describe the process of creating the compounding bits associated with the last doubleword of a first cache line which immediately precedes a second line being inpaged.

Referring now to FIG. 9, the instructions from the first cache line necessary to create the compounding bits of this invention reside in DATA REG 132 after having been fetched from CACHE ARRAY 114 as previously described. As quadwords of instruction text are being inpaged to CACHE ARRAY 114, they pass through ICU 12, and in particular through L2REG 410. The line may be arbitrarily rotated, i.e., QW0 may not be the first quadword to be received from MEMORY 10. Whenever QW0 arrives at the ICU and is latched in L2REG 410, it is latched in S0 420 on the following cycle and retained until after the compounding tag bits for the second line have been calculated. 0n a subsequent cycle, the contents of S0 420 are gated back into the high half of L2REG 410. Simultaneously, the contents of DATA REG 132 are gated into L2BUL 430. A final compounding operation is then performed in CBOX5 440, CBOX6 450, and CBOX7 460, providing the lattermost compounding bits for the first cache line. Said compounding bits are subsequently stored in TAG ARRAY 116 using the address from A REG 140.

FIG. 10 shows a representative timing sequence for a cross-line compounding operation starting with QW6. This operation may be compared to the sequence shown in FIG. 3 which shows a compounding operation also starling with QW6, but without cross-line compounding.

Referring to FIG. 10, the ICU operation is as follows. Assume that an instruction cache miss has occurred and that QW6 is the required QW. In cycle 0, QW6 is on L2.sub.-- STG.sub.-- BUS, and DVALID is asserted. This causes FIRSTQW to be asserted, which, in turn, causes LD.sub.-- L2 to be asserted and also triggers the one-cycle assertion of LD.sub.-- S in cycle 1. LD.sub.-- L2 remains asserted until the cycle after GT.sub.-- S.sub.-- L2H has been asserted (cycle 8). In cycle 1, QW7 is on L2.sub.-- STG.sub.-- BUS. Since this is the last (right-most) QW of the line, EOL is asserted. The assertion of EOL will trigger the assertion of TRUNCATE and LD.sub.-- S0 in cycle 3. Since LD.sub.-- S is asserted, QW6H will be latched in S at the end of cycle 1, where it will be held until cycle 9. LD.sub.-- L2BUL is also asserted. Thus, the low DW of L2REG will be latched in L2BUL at the end of the cycle. Also, C bits for QW6H (,CQW6H) are produced by CBOXes 0-3 and are latched in their respective latches.

In cycle 2, QW7 is latched in L2REG. Now valid C bits can be produced for QW6L, which is now in L2BUL. These C bits are present on the outputs of CBOXes 4-7. Thus, C0-3.sub.-- LTH.parallel.C4-7 constitute the C bits for QW6, the low DW of which is now in L2BUL.

In cycle 3, valid C bits are available for QW7, now in L2BUL, but C5, C6, and C7 must be truncated.

Thus, TRUNCATE is asserted in this cycle, which forces these bits to zero. The assertion of LD.sub.-- S0 causes QW0H to be saved in S0 until it is needed for cross-line compounding.

Production of C bits continues routinely until cycle 7 when LASTQW is asserted. In cycle 8, GT.sub.-- S.sub.-- L2H is asserted, causing QW6H (the high DW of the first QW received) to be gated into L2REGH at the end of the cycle. Simultaneously, QW5L is gated into L2BUL, as usual. Then, in cycle 9, valid C bits are produced for QW5.

The assertion of GT.sub.-- S.sub.-- L2H in cycle 8 also triggers the assertion of GT.sub.-- S0.sub.-- L2H and GT.sub.-- DR.sub.-- L2BUL in cycle 9. The high order portion of QW0 is loaded back into the high order part of L2REG, and the latter portion of the previous cache line saved earlier in DATA REG 132 is loaded into L2BUL, as indicated by DR in cycle 10 in FIG. 10. In cycle 10, the compounding bits C5, C6, and C7 for the previous cache line (CDR in FIG. 10) are created and passed to the TAG ARRAY 116. Only C5, C6, and C7 are valid during cycle 10. The remaining tag bits in TAG ARRAY 116 are not altered by the cross-line compounding operation.

While we have described our preferred embodiments of our invention, it will be understood that those skilled in the art, both now and in the future, may make various improvements and enhancements which fall within the scope of the claims which follow. These claims should be construed to maintain the proper protection for the invention first disclosed.

* * * * *

United States Patent	***5,398,321***
Jeremiah	**March 14, 1995**

Microcode generation for a scalable compound instruction set machine

Abstract

An apparatus for generating microcode in a scalable compound instruction set machine operates in response to compounding information indicating that two or more adjacent instructions are to be executed in parallel. Separate and independent microcode is held in control store for each possible instruction in a group. Microcode sequences for each instruction of a group of instructions to be executed in parallel are merged in response to the compounding information into a single microinstruction sequence.

Inventors:	**Jeremiah; Thomas L.** (Endwell, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**184401**
Filed:	**January 21, 1994**

Current U.S. Class:	**712/216**; 712/245
Intern'l Class:	G06F 009/22; G06F 009/38
Field of Search:	395/375

References Cited [Referenced By]

U.S. Patent Documents

4295193	Oct., 1981	Pomerene	395/375.
4376976	Mar., 1983	Lahti et al.	395/375.
4439828	Mar., 1984	Martin	364/200.

4594655	Jun., 1986	Hao et al.	395/775.
4825363	Apr., 1989	Baumann et al.	395/375.
4858105	Aug., 1989	Kuriyama et al.	395/375.
4942525	Jul., 1990	Shintani et al.	395/375.
4967343	Oct., 1990	Ngai et al.	395/800.
5005118	Apr., 1991	Lenoski	395/375.
5051940	Sep., 1991	Vassiliadis et al.	364/736.
5117490	May., 1992	Duxbury et al.	395/375.
5129067	Jul., 1992	Johnson	395/375.
5140545	Aug., 1992	Vassiliadis et al.	364/765.
5155819	Oct., 1992	Watkins et al.	395/375.
5163139	Nov., 1992	Haigh et al.	395/375.
5229321	Feb., 1993	Iizuka	395/375.
5241636	Aug., 1993	Kohn	395/375.
5287467	Feb., 1994	Blaner et al.	395/375.
5295249	Mar., 1994	Blaner et al.	395/375.
5299319	Mar., 1994	Vassiliadis et al.	395/375.
5301341	Apr., 1994	Vassiliadis et al.	395/800.
5303356	Apr., 1994	Vassiliadis	395/375.

Foreign Patent Documents

0045634	Feb., 1982	EP.
0184158	Jun., 1986	EP.
0397414	Nov., 1990	EP.
2293932	Dec., 1990	JP.

Other References

Acosta, R. D., et al, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al, the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles And Examples (Siewiorek, et al, ed (McGraw-Hill, 1982, pp.276-292.
Capozzi, A. J., et al, "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al, "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al., "SIMP (Single Instruction Stream/Multiple Instruction Pipelining)": A Novel High-Speed Single Processor Architecture, Proceedings of the Sixteenth Annual Symposium On Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M. D., et al, "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Artihmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al, "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N. P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions On Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp. 63-76.
Colwell, R. P., et al, "A VLIW Architecture for a Trace Scheduling Complier", IEEE Transaction On Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the CRISP Instruction Set Architecture", Proceedings Of Compcon, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al, "Compiling for the CRISP Microprossesor", Proceedings Of Compcon, Spring, 1987, pp. 96-100.
Hennessy, J., et al, "MIPS: A VSI Processor Architecture", Proceedings Of The CMU Conference On VLSI Systems And Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications Of The ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal Of Research And Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings Of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W., et al, "Checkpoint Repair for High-Performance Out-of-Order Execution

Machines", IEEE Transactions On Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al, "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1. Jan. 1984, pp. 6-22.
Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions On Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings Of The Eight Annual Symposium On Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al, Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions On Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al, "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal Of Parallel And Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memorie", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al, "A Study of Instruction Cache Oraganizations and Replacement Policies", IEEE Proceedings Of The Tenth Annual International Symposium On Computer Architecture, Jun., 1983, pp. 132-137.
Vassiliadis, S., et al, "Condition Code Predictory for Fixed-Arithmetic Units", International Journal Of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture Of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991) by R. J. Eberhard.
"Self Aligning End of Instruction and Next Instruction Load", Research Disclosure, Jul. 1989, No. 303, p. 519.

Primary Examiner: Lall; Parshotam S.
Assistant Examiner: Philipp; Timothy
Attorney, Agent or Firm: Augspurger; Lynn L., Meador; Terrance A.

Parent Case Text

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation of application Ser. No. 07/653,006, filed Feb. 8, 1991, now abandoned.

Claims

I claim:

1. In a computer having a plurality of execution units in which object level instructions are executed singly in a scalar manner and in a group in a parallel, concurrent manner, and in which parallel execution of a group of object level instructions is indicated by compounding control information which is generated prior to fetching said object level instructions for execution, an apparatus for generating microcode that implements and controls execution of said object level instructions, said apparatus comprising:

compound processing means for generating said compound control information for said group of object level instructions prior to fetching said group of object level instructions for execution;

means for storing said group of object level instructions, including means for storing a first object level instruction, means for storing a second object level instruction, and means for storing said compounding control information indicating parallel execution of said first object level instruction and said second object level instruction;

means for storing a plurality of first microcode instruction sequences that includes a sequence of microcode instructions for executing each object level instruction that is executed by said computer singly in a scaler manner;

means for storing a second plurality of microcode instruction sequences that includes a sequence of partial microcode instructions for each object level instruction that is executed as a second instruction in a group of object level instructions;

means to map said first object level instruction to said means for storing a first microcode instruction sequence to fetch a first microcode instruction;

means to map said second instruction to said means for storing a second microcode instruction sequence to fetch a second microcode instruction;

means responsive to said first object level instruction, said second object level instruction and said compounding control information to merge said first microcode instruction and said second microcode instruction;

means coupling said means to merge said means for storing said first microcode instruction sequence and said means for storing a second microcode instruction sequence;

said means to merge including;

means to determine an execution resource dependency between said first microcode instruction and said second microcode instruction;

means responsive to said means to determine an execution resource dependency to substitute predetermined fields of said second microcode instruction into certain predetermined fields of said first microcode instruction to form a merged microcode instruction; and

means for parallel execution of said merged microcode instruction connected to said means to merge.

2. The apparatus of claim 1, wherein

said merging means includes multiplexing means connected to said first and second microinstruction storages for selecting microinstruction fields from said first and second sequences of microinstructions in response to said compounding information.

3. The apparatus of claim 1, wherein

said first sequence of microinstructions includes a microinstruction with first fields for controlling instruction processing by one of said execution units of said first instruction and second fields for controlling instruction processing by one of said execution units of said second instruction:

said means for pipelined execution including register means for receiving said microinstruction; and

said merging means including multiplexing means connected to said first and second microinstruction storages and to said register means for entering field information from either said first sequence microinstructions or said second sequence of microinstructions into a second field of said microinstruction in response to said compounding information.

4. The apparatus of claim 1, wherein

said number of microinstructions in said first sequence of microinstructions and said number of microinstructions in said sequence of microinstructions are unequal, said merging means including means for synchronizing said completion of said first and second sequences of microinstructions in response to said compounding information by adding at least one no-operation microinstruction to said first sequence of microinstructions.

5. The apparatus of claim 1, wherein

said number of microinstructions in said first sequence of microinstructions in said number of microinstructions in said sequence of microinstructions are unequal, said merging means including means for synchronizing said completion of said second sequences of microinstructions in response to said first and second instructions by delaying said second sequence of microinstructions with respect to said first sequence of microinstructions.

6. A computer as in claim 1 further including hardware field setting means for generating a hardware microinstruction field indicating a particular operation of an execution unit, and multiplexing means connected to said hardware field setting means and to said merging means for selectively substituting said hardware microinstruction field for a field in said microinstruction in said first microinstruction sequence.

7. In a computer having a plurality of execution units in which object level instructions are executed singly in a scaler manner and in a group in a parallel, concurrent manner, and in which parallel execution of a group of object level instructions is indicated by compounding control information which is generated prior to fetching said object level instructions for execution, an apparatus for generating

microcode that implements and controls execution of said object level instructions, said apparatus comprising:

means for storing said group of object level instructions, including means for storing a first object level instruction, means for storing a second object level instruction, and means for storing said compounding control information, which indicates parallel execution of said first object level instruction and said second object level instruction;

means for storing a plurality of first microcode instruction sequences that includes a sequence of microcode instructions for executing each object level instruction that is executed by said computer singly in a scaler manner;

means for storing a second plurality of microcode instruction sequences that includes a sequence of microcode instructions for each object level instruction that is executed as a second instruction in a group of object level instructions;

means to map said first object level instruction to said means for storing a first microcode instruction sequence to fetch a first microcode instruction;

means to map said second instruction to said means for storing a second microcode instruction sequence to fetch a second microcode instruction;

means responsive to said first object level instruction, said second object level instruction and said compounding control information to merge said first microcode instruction and said second microcode instruction;

means coupling said means to merge to said means for storing said first microcode instruction sequence and to said means for storing a second microcode instruction sequence;

said means to merge including;

means to determine an execution resource dependency between said first microcode instruction and said second microcode instruction;

means responsive to said means to determine an execution resource dependency to substitute predetermined fields of said second microcode instruction into certain predetermined fields of said first microcode instruction to form a merged microcode instruction; and

means for parallel execution of said merged microcode instruction connected to said means to merge.

8. A computer as in claim 7 further including hardware field setting means for generating a hardware microinstruction field indicating a particular operation of an execution unit, and multiplexing means connected to said hardware field setting means and to said merging means for selectively substituting said hardware microinstruction field for a field in said microinstruction in said first microinstruction sequence.

9. In a computer having a plurality of execution units in which object level instructions can be executed singly in a scalar manner and in parallel in a concurrent manner and in which parallel execution of a group of object level instructions is indicated by compounding control information which specifies whether a group of object level instructions can be concurrently executed, an apparatus for generating microcode for said group of object level instructions, said apparatus comprising:

a first microinstruction storage for providing a first sequence of microinstructions for executing a first object level instruction of said group of instructions;

a second microinstruction storage for providing a second sequence of microinstructions for executing a second object level instruction of said group of instructions;

merging means connected to said first microinstruction storage and said second microinstruction storage;

said merging means forming a merged microinstruction in response to said compounding information from said first sequence of microinstructions and said second sequence of microinstructions by substituting a field from a microinstruction in said second sequence of microinstructions for a field in a microinstruction in said first sequence of microinstructions;

hardware field setting means for generating a hardware microinstruction field indicating a particular operation of an execution unit;

multiplexing means connected to said hardware field setting means and to said merging means for selectively substituting said hardware microinstruction field for a field in said microinstruction in said first microinstruction sequence; and

means connected to said merging means for pipelined execution of said merged composite sequence of microinstructions.

Description

CROSS REFERENCE TO RELATED APPLICATIONS

The present U.S. patent application is related to the following co-pending U.S. patent applications:

(1) Application Ser. No. 07/519,382 (IBM Docket EN9-90-020), filed May 4, 1990, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al;

(2) Application Ser. No. 07/519,384 (IBM Docket EN9-90-019), filed May 4, 1990, entitled "General Purpose Compound Apparatus for Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al;

(3) Application Ser. No. 07/504,910 (IBM Docket EN9-90-014), now U.S. Pat. No. 5,051,940 filed Apr.

4, 1990, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al;

(4) Application Ser. No. 07/522,291, now U.S. Pat. No. 5,214,763, filed May 10, 1990, entitled "Compounding Preprocessor for Cache", the inventors being Bartholmew Blaner et al;

(5) Application Ser. No. 07/543,464, filed Jun. 26, 1990, entitled "An In-Memory Preprocessor for a Scalable Compound Instruction set Machine Processor", the inventors being Richard J. Eickemeyer et al;

(6) Application Ser. No. 07/543,458, now U.S. Pat. No. 5,197,135, filed Jun. 26, 1990, entitled "Memory Management for Scalable Compound Instruction Set Machines with In-Memory Compounding", the inventors being Richard J. Eickemeyer et al;

(7) Application Ser. No. 07/619,868, filed Nov. 28, 1990, now U.S. Pat. No. 5,301,341, entitled "Overflow Determination for Three-Operand ALUS in a Scalable Compound Instruction Set Machine", the inventors being Stamatis Vassiliadis et al; and

(8) Application Ser. No. 07/642,011, filed Jan. 15, 1991, now U.S. Pat. No. 5,295,249, entitled "Compounding Preprocessor for Cache", the inventors being B. Blaner et al. (A continuation-in-part of U.S. patent application Ser. No. 07/522,291, now U.S. Pat. No. 5,197,135 filed May 10, 1990).

These co-pending applications and the present application are owned by one and the same assignee, namely, INTERNATIONAL BUSINESS MACHINES CORPORATION of Armonk, N.Y.

The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

BACKGROUND OF THE INVENTION

This invention relates to digital computers and digital data processors and particularly to digital computers and data processors capable of processing two or more instructions in parallel.

The performance of traditional computers which execute instructions singly in a sequential manner has improved significantly in the past largely due to improvements in circuit technology. Machines which execute instructions one at a time are sometimes referred to as "scalar" computers or processors. As circuit technology is pushed to its limits, computer designers have had to investigate other means to obtain significant performance improvements.

Recently, so-called "superscalar" computers have been proposed which attempt to increase performance by selectively executing more than one instruction at a time from a single instruction stream. Superscalar machines typically decide at instruction execution time if a given number of instructions may be executed in parallel. Such a decision is based on the operation codes (OP codes) of the instructions and on data dependencies which may exist between adjacent instructions. The OP codes determine the particular hardware components each of the instructions will utilize and, in general, it is not possible for two or more instructions to utilize the same hardware component at the same time nor to execute one of the instructions if it depends on the results of another of the instructions (a "data dependency" or "data interlock"). These hardware and data dependencies prevent the parallel execution of some instruction combinations. In these cases, instructions are instead executed by themselves in a non-parallel manner. This, of course, reduces the performance of a superscalar machine.

Superscalar computers provide some improvement in performance but also have disadvantages which it would be desirable to minimize. For example, deciding at instruction execution time which instructions can be executed in parallel takes a significant amount of time which cannot be very readily masked by overlapping the decision with other normal machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Another disadvantage is that the decision making must be repeated if-the same instructions are to be executed a second or further time.

The cross-referenced applications all concern a digital computer or data processor called a scalable compound instruction set machine (SCISM) in which the performance of the parallel execution decision is made prior to execution time. In SCISM architecture, the decision to execute in parallel is made early in the overall instruction handling process. For example, the decision can be made ahead of the instruction buffer in those machines which have instruction buffers or instruction stacks or ahead of the instruction cache in those machines which flow the instructions through a cache unit.

Because the decision to execute in parallel is made prior to a point where instructions are stored, the results of the decision making can be preserved with the instructions and reused in the event that the same instructions are used a second or further time.

Preferably, the recording of the parallel execution decision making is in the form of tags which accompany the individual instructions in an instruction stream. These tags tell whether the instructions can be executed in parallel or whether they need to be executed one at a time. This instruction tagging process is sometimes referred to herein as "compounding". It serves, in effect, to combine at least two individual instructions into a single compound instruction for parallel processing purposes.

The exemplary embodiment of a SCISM is underpinned by the architecture and instructions of the System/370 product family available from the IBM Corporation, Armonk, N.Y., the assignee of this application. Preferably, the SCISM compounds instructions while they are in object form. As is known, System/370 architecture typically employs microcoded instructions to implement and control the execution of object-level instructions. Consequently, all System/370 instructions which are executed, either singly or in parallel in a SCISM, are controlled by one or more microinstructions. Microinstruction execution of object instructions is a widely used concept for which many implementations are known. The challenge in executing scalar instructions in parallel is to provide microinstruction sequences which reflect such parallelism.

One approach known to the inventor is implemented in a machine which executes up to two instructions in parallel. This approach provides a unique microcoded routine for all possible pairs of instructions, as well as routines for each instruction individually. While conceptually simple, this approach requires significant additional microcode storage to support the parallel instruction routines. Each pair of instructions which can be executed in parallel becomes, in effect, a new instruction with its own microcode. The storage and management overhead for such an approach can become substantial, adding many unique microcode routines to a standard set of microinstructions. Furthermore, the number of combinations proliferates geometrically with the number of instructions which are executed in parallel.

Consequently, there is a need in computers or processors which can execute two or more instructions simultaneously to provide machine-level instructions for all possible combinations without adding substantially to the overhead required for storing and retrieving the microinstructions.

SUMMARY OF THE INVENTION

It is an object of this invention to provide for the generation of microcode in a scalable compound instruction set machine which groups two or more instructions for parallel execution prior to execution time.

A related object is to devise an efficient mechanism for generating microinstructions in response to pairs of machine-level instructions which have been marked for concurrent execution.

The foregoing objects are satisfied in an apparatus based on the inventor's critical observation that when compounded instructions are encountered during instruction fetch and issue, a microinstruction sequence can be fetched for each of the compounded instructions and merged to produce a single microinstruction sequence able to control the simultaneous execution of the compounded instructions. Using this approach, only a single microinstruction routine need be coded and stored for each individual machine-level instruction, regardless of whether the instruction is compounded with another instruction, or not. A second form of coding is provided for instructions that are executed as the second half of a compound instruction.

The apparatus of the invention is found in a computer in which instructions can be executed singly and in parallel and in which parallel execution of a group of instructions is indicated by compounding information generated prior to the execution of the instruction. However, the invention is not limited to the case where compounding is performed prior to instruction issue. It is only necessary that the instruction issuing mechanism indicate to the microinstruction merging mechanism that the instructions being issued are legitimate candidates for parallel execution. In this context the invention is an apparatus for generating microcode for the group of instructions and includes a first microinstruction storage for providing a first sequence of microinstructions for executing a first instruction of the group of instructions and a second microinstruction storage for providing a second sequence of microinstructions for executing a second instruction of a group of instructions. A merging unit is coupled to the first and second microinstruction storages and combines the first and second sequences of microinstructions into a composite sequence of microinstructions in response to the compounding information, the length of the composite sequence of microinstructions equalling the longer of the first and the second sequence of microinstructions. Last, a series of registers is connected to the merging unit provide for pipelined execution of the composite sequence of microinstructions.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a scalable compound instruction set machine which includes an apparatus for generating microcode for compounded instruction pairs.

FIG. 2 is a partial schematic diagram showing how compounding information is used to issue compounded instructions.

FIG. 3 is a block diagram showing the principal elements of the apparatus of the invention.

FIG. 4 is a schematic diagram showing details of a compound instruction register.

FIG. 5 is a schematic diagram showing in greater detail how two microinstruction sequences are merged in a merge unit in the apparatus illustrated in FIG. 3.

FIGS. 6A and 6B illustrate a detail of the merge unit for selecting microinstruction field information from one microinstruction two microinstruction sequences.

FIG. 7 illustrates a detail of the merge unit which selects microinstruction field information from a hardware source.

FIG. 8 illustrates how two merging microinstruction sequences of unequal length are simultaneously terminated in response to a first set of conditions.

FIG. 9 is a schematic diagram illustrating a detail of the merge unit which performs the function illustrated in FIG. 8.

FIG. 10 illustrates how two merged microinstruction sequences of unequal length are simultaneously terminated in response to a second set of conditions.

FIG. 11 is a schematic diagram illustrating a detail of the merge unit which accomplishes the procedure of FIG. 10.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The invention does not reside in the use of microcode to execute machine instructions. However, a computer employing a microcode generator for machine instruction execution does form the environment within which the invention is practiced. Such a computer system can include a machine which executes the instructions of the IBM System/370 instruction set, for example. This set of instructions is explained at length in the work by C. J. Kacmar, entitled "IBM 370 ASSEMBLY LANGUAGE WITH ASSIST", Prentice-Hall, 1988.

When used herein, the term "machine instruction" means an instruction which is in object form. In OPERATING SYSTEMS, Second Edition, C. H. Deitel has defined microprogramming as "a layer of programming below a computer's machine language." Expanding Deitel's definition, a microprogram is a collection of microinstructions. A "microinstruction" (or, a "microword") is an instruction "that controls data and sequencing in a processor at a more fundamental level than a machine instruction . . . ", IBM DICTIONARY OF COMPUTING, Eighth Edition, 1987. A group of microinstructions is commonly referred to as "microcode".

As is known, a computer microprogram is typically retained in a portion of computer memory which is not accessible to a programmer. Instead the machine-level instructions of a compiled program are individually mapped to microinstructions in the section of storage referred to as "control store".

FIG. 1 illustrates a SCISM architecture in which microprogramming is used to implement and control the execution of machine-level instructions, singly or in pairs. In particular, a machine-level instruction stream is provided to a compounding preprocessor 10. The instruction stream is a sequence of individual instructions which have typically been complied from a source program. The stream is provided to the CPU of a computer for execution. Conventionally, the machine-level instructions are staged into the CPU through a cache 12. Before entry into the cache, the instruction stream is examined by the compounding preprocessor 10 to determine whether at least two adjacent instructions can be concurrently executed. The compounding preprocessor 10 is described in detail in the eighth referenced co-pending U.S. patent application, entitled "COMPOUNDING PREPROCESSOR FOR CACHE", inventors: B Blaner, et al, Ser. No. 07/642,011, filed on Jan. 15, 1991, now U.S. Pat. No. 5,295,249. Also described in this application is the structure of a compounding instruction cache which functions as the cache 12 of FIG. 1.

The operation of the compounding preprocessor 10 results in the generation of compounding information indicating whether adjacent instructions of the compiled instruction stream which have been cached at 12 can be concurrently executed. Thus, for each instruction, the compounding preprocessor 10 generates compounding information indicating whether that instruction and an adjacent instruction can be executed in parallel.

Following processing by the compounding preprocessor 10, the analyzed instructions and the compounding information are stored in the compound instruction cache 12. Except for the provision of extra space to store the compounding information, it is assumed that the cache 12 is operated conventionally. In particular, entries in the cache 12 are typically groups of adjacent instructions ("lines") which are entered into the cache so that they can be quickly obtained as required by an executing program.

In providing the compounding information together with instructions in the cache 12, the SCISM architecture takes fuller advantage of parallelism than a computer which makes the parallel execution decision when instructions are brought out of the cache for immediate execution ("issued"). Relatedly, an instruction in the cache 12 may be used more than once in, for example, a loop or branch. So long as the instruction is in the cache, it does not have to be re-analyzed if obtained again for execution because the compounding information for the instruction which is stored with it in the cache can be reused.

The compounding preprocessor 10 is assumed to be of the type described in the co-pending patent application which generates, for each instruction, a one-bit tag. These tags are used to identify which pairs of instructions may be processed in parallel. Instructions and their tags are supplied to and stored into the compound instruction cache 12. An instruction fetch and issue unit 14 fetches the instructions and their tags from the compound instruction cache 12, as needed, and arranges for their execution by the appropriate one or ones of a plurality of execution units 34, 36. The fetch and issue unit 14 examines the tag and OP codes of fetched instructions. If a tag indicates that two successive instructions are to be processed in parallel, then the fetch and issue unit 14 enters both into a compound instruction register (CIR) 19. The compound instruction register 19 includes a left compound instruction register (CIRL) 20 and a right compound instruction register (CIRR) 21. A bit-wide field is provided in the CIRL 20 for storage of the compounding tag for the first instruction of a compounded pair of instructions. The first instruction is stored in the remainder of the CIRL 20, while the second instruction is stored in the CIRR

21. In the discussion which follows, the first, or left, instruction is assumed to precede the second, or right, instruction in the instruction sequence, and its compounding tag indicates whether or not it is to be executed with the instruction following it. Thus, the first instruction of a compounded pair is stored in the CIRL 20, while the second instruction is stored in the CIRR 21. In the preferred embodiment, the compounding tag is a single bit, hereinafter, a "C-bit", which, if set to a value of "one" indicates that the CIRR 21 contains an instruction which is to be executed simultaneously with the instruction contained in CIRL 20. If set to set "zero" the compounding tag indicates that the contents of CIRR 21 are to be ignored during execution of the instruction contained in CIRL 20.

For example, in FIG. 2 a line of eight instructions 40 is fetched from the cache 12, together with an array of C-bits 42 which is termed a C-vector. The C-vector 42 contains at least eight bits, one for each of the eight instructions in the line 40. Each numbered bit of the C-vector 42 constitutes the C-bit for its respective instruction; for example, C-bit 1 is the compounding tag for instruction 1 (INSTR 1). The instruction fetch and issue unit 14 inspects the instructions in the line 40 in sequence and simultaneously inspects their C-bits. According to the preferred method of compounding, if two instructions are to be executed in parallel, the C-bit for the first instruction is set to one, while the bit for the succeeding instruction is ignored. This is only by way of example and is not intended to limit the mapping of compounding bits to instructions, nor the number of adjacent instructions which can be grouped for compounding. Assuming that the first two instructions of line 40 have been marked for parallel execution, the C-bit for the first instruction has a value of one. The logic of the fetch and issue unit 14 loads the first instruction and its associated C-bit into the CIRL 20 and the C field 22. A gate 44 in the logic of the fetch and issue unit 14 is enabled by decoding the length of the first instruction, which permits the second instruction (INSTR 2) of the line 40 to be loaded into the CIRR 21. Of course, if the value of C-bit 1 is zero, the instruction loaded into CIRR will be subsequently ignored. When execution of the instruction or instructions in the CIR 19 has completed, the fetch and issue unit advances to the next instruction to be executed and operates as described.

The precise structure of the instruction fetch and issue unit 14 is not the subject of this invention, and it is sufficient to say that this unit contains logic which is capable of determining on the basis of its compounding tags whether the next instruction is to be executed singly or in parallel with the instruction which immediately follows it. It is further asserted that this logic implements its decision by appropriately examining the C-bit in the compound instruction register 19 as described above.

At the point where they are entered into the instruction fetch and issue unit 14, instructions are in object code form. When an instruction is entered into the compound instruction register 19, it is decoded by a microcode generator 23 which generates microcode according to the invention. The microcode generator 23 decodes an instruction in the compound instruction register by mapping it to a sequence of microinstructions which are contained in a control store (CS). The sequence may contain one or more microinstructions. The contents of the CIRL 20 are mapped to a first sequence of microinstructions by way of control logic 24 and a main control store (MCS) 25. Simultaneously, the contents, if any, of the CIRR 21 are mapped to a second sequence of microinstructions by control logic 26 and a secondary control store (SCS) 27. If the contents of the C-bit field of the CIRL 20 indicate that an instruction is contained in the CIRR 21, the first sequence of microinstructions output by the MCS 25 is merged in a merge unit 29 with the second sequence of microinstructions output by the SCS 27. If the C-bit indicates that no instruction is contained in the CIRR 21, the merge unit 29 ignores the output of the SCS 27 and passes on the first sequence of microinstructions for execution.

Provision of a microinstruction sequence out of either the MCS 25 or SCS 27 is essentially conventional. In this regard, the MCS 25, for example, stores microinstructions at addressable locations. Conventionally, the address for the first microinstruction of a microinstruction sequence is obtained from a predetermined field of the instruction in the CIRL 20. For example, in the IBM System/370 instruction set, the OP code (the first byte) of an instruction forms the basis for the MCS address location of a microinstruction sequence which has been coded to execute the machine instruction. If the microinstruction sequence is longer than a single microinstruction, each microinstruction of the sequence, save the last, contains the address of the next microinstruction in the sequence in a next address field (NXA). This field is fed back to the control unit 24 to generate the MCS address of the next microinstruction sequence. The first microinstruction of a sequence is found by appending zeros (or another predetermined value) to the OP code of the CIRL instruction; the microinstruction sequence is followed by the NXA field contents of succeeding microinstructions.

Typically, microinstructions in the MCS 25 will include a field to indicate when the end of the sequence is reached. Thus, the only microinstruction in a single-microinstruction sequence and the last microinstruction of a multi-microinstruction sequence will contain an indication in this field (the EOP or end-of-operation field) signalling the end of the operation. In SCISM architecture, the EOP field is bit-wide. When the bit in this field is set, the end of the operation is signified; when not set, the operation continues. Typically, detecting of a set EOP bit results in a generation of an end-of-operation (ENDOP) signal provided to the fetch and issue unit 14 signalling it to issue another instruction for execution, which also initiates the OP breakout process.

Returning to the description of FIG. 1, microinstruction sequences are passed by the merge unit 29 to an execution pipeline 32 which controls two or more execution units 34, 36, and a bank of registers 38 which can include general purpose and storage address registers.

The merge unit 29 passes to the execution pipeline 32 only the microinstruction fields which are appropriate for controlling the operations of the execution units 34 and 36 and the transfer of data between those units and the registers 38. Relatedly, the microinstructions are passed to the execution pipeline 32 shorn of microprogramming addressing and branching fields and the EOP field. The remaining fields are to control operations necessary to execute the instruction or instructions. As detailed below, certain fields are dedicated to controlling the operations of the first execution unit 34, while other fields are allocated for controlling the operations of the second execution unit 36. If a potentially-compoundable instruction is being executed singly (perhaps due to the lack of a subsequent instruction suitable for compounding), only the execution unit 34 is operated. Non-compoundable instructions are free to use both execution units 34 and 36. Parallel execution implies concurrent operation of the execution units 34 and 36 in response to the execution pipeline 32.

The microcode generator 23 is shown in more detail in FIG. 3. As will be discussed below, the fundamental microinstruction format is inherent in the microinstructions stored in the MCS 25. The format is indicated by 43, representing the currently-addressed MCS microinstruction. In addition to the NXA and EOP fields, discussed above, microinstruction 43 includes fields containing branching information (BR) and control fields (CTL) containing information necessary to control execution units and registers in executing an instruction. The control unit for the MCS 25 includes control logic 24a, a multiplexing unit 24b, and a control store address register (CSAR) 24c. The contents of the CSAR 24c provide the address input to the MCS 25, in response to which a microinstruction at the address is read out of the MCS 25. When available at the output of the MCS, the microinstruction has the format indicated by 43. Preferably, the MCS 25 includes pageable sections which can be fed from an auxialliary storage area in processor main memory that is reserved for processor use as necessary. The control logic 24a performs the essential function of providing the next instruction address to the MCS 25; a secondary function is to generate a page address and to control the entry of paged data into the MCS 25. The control logic 24a selects the next address component by control of the multiplexer 24b. Selection is implemented by the control logic 24a in response to the current address in the CSAR, the BR and NXA fields of the addressed microinstructions, and to conditions indicated by the state of the EOP field of the current microinstruction and branch conditions indicated in certain fields of a microinstruction in the address generation stage of the execution pipeline 32.

In operation, when an instruction has finished executing, the EOP bit will have forced the control logic 24a to multiplex the starting address of the next instruction into the CSAR 24c, leading to the readout of a first microinstruction. Thereafter, the branching, addressing, and EOP fields of the first microinstruction and any following microinstructions will produce the particular sequence of microinstructions designed to execute the instruction in the CIRL 20. If the sequence is a single instruction sequence, EOP bit of the first microinstruction will be set. Simultaneously upon the control portion of the instruction being placed in the execution pipeline 32, the control logic 24a will initialize the CSAR 24c to await the next instruction in the CIRL 20. If more than one instruction is contained in the sequence, the EOP field of the last instruction will initialize the CSAR 24c when the final instruction is put into the execution pipeline 32.

The main control store 25 contains microcode for instructions which can be compounded. In addition, this store contains microcode for instructions which are never compounded, interrupt handlers and miscellaneous microcode. Preferably, the address used in the MCS 25 is 16 bits, giving total addressable range of 64k words. All microcode residing from address zero to address 4095 is fixed; that is, once loaded during initialization, it remains in the MCS 25. Microcode at assigned address 4096 and above is demand-paged from an auxiliary storage in the CPU main memory (not illustrated) reserved for processor use.

The secondary control store 27 has an address space of 256 words. This address space has no relation to the MCS space. The SCS addresses 0 through 255 contain the first microword of every compoundable System/370 instruction. Addresses in the SCS 27 corresponding to noncompoundable instructions are used to contain microcode for second and succeeding, cycles of multi-cycle compoundable instructions. As FIG. 3 illustrates, the primary components of the SCS control unit are a multiplexing unit 26b and a control store address register (CSAR) 26c. The multiplexer is controlled by the merge unit 29 as described below. The format of microinstructions in the SCS 27 is indicated by reference numeral 45. In this regard, conditional branching in SCS microcode is not allowed, however a next address field (NXA) similar to that used in the MCS 25 is a convenient means for multi-instruction sequences. The smaller address space of the SCS 25 requires only provision of the OP code from the CIRR 21, and thereafter address data from the NXA field, if necessary. The set EOP bit in the currently-addressed microinstruction will initialize the contents of the CSAR 26c to prepare for receipt of the OP code of the next instruction in the CIRR 21.

MICROCODE GENERATOR OVERVIEW

The fundamental sequence of operations performed by the microcode generator of FIG. 3 includes five pipeline stages which permit the initiation of instructions into the pipeline each cycle. The first stage, IF, is instruction fetch. This stage is signified to the fetch and issue unit by the ENDOP signal from the microcode generator 23. Relatedly, in this stage, an instruction is fetched from the compound instruction cache or an instruction buffer. At the end of the IF cycle, one instruction or a pair of adjacent instructions are ready to be loaded into CIR 19 for decoding to begin instruction execution.

The second cycle or pipeline stage is called instruction decode (ID). This cycle is controlled by logic decoding of the CIRL 20 and, if appropriate, the CIRR 21. In this respect, logic decoding includes providing an instruction OP code to the appropriate control store and latching the OP code into the appropriate CSAR. Access of the first microinstruction needed to control subsequent stages of the pipeline is called "OP breakout". OP breakout consists of accessing control store using the OP code of the instruction to generate the microinstruction address. Storage addressing operands are fetched from a copy of the general purpose register array in this cycle if required.

The address generation (AGEN) cycle is used to calculate an effective address for operands needed from storage. Up to three address operands may be added. Operands to be used in the next cycle in an execution unit are accessed from the general registers in this cycle as well.

The execution (EX) cycle is used to perform operations in one or more execution units. This cycle is also used as a cache access cycle for instructions requiring storage operands. In most IBM System/370 processors, certain instructions, such as RX format adds, subtracts, and so on, require an EX cycle to fetch a second operand from storage followed by another EX cycle to compute the result.

The last cycle, termed the putaway (PA) cycle is used to store results from the EX cycle in the general purpose registers. Storage of data in the cache for STORE-type instructions may also occur in this cycle, provided they are not delayed by a subsequent fetch operation.

Inherent in the microcode generator of FIG. 3 is a microcode pipeline which mimics the pipeline for IBM System/370 instructions, except that there is no explicit ID cycle. Instead, the AGEN cycle immediately follows the fetch cycle of a microword. This is possible because the portions of the microinstruction used in the AGEN cycle are horizontal and require minimal decoding. The execution pipeline 32 includes the last three stages, each stage represented by one of three registers 32a, 32b and 32c. Following the sequence given above, the register 32a represents the AGEN cycle and includes an address instruction register (AIR). The microinstruction flows into this register from the merge unit 29 and is held in the register for one cycle of a pipeline clock to control AGEN cycle operations. In this regard, relevant control fields in the microinstruction which are necessary to control address generation are accessed. At the next pipeline clock cycle the microinstruction is transferred to the EX instruction register 32b where fields controlling execution unit operations are accessed. Last, the microinstruction is shifted at the next pipeline clock cycle to the putaway instruction register (PIR) 32c to control operations necessary to store results generated in the execution cycle.

Shift controls for the sequence of registers 32a, 32b, and 32c are conventional, as is pipeline clocking.

The pipeline supports overlapped execution of machine instructions.

FIG. 4 is a schematic diagram showing control of the CIRL portions by the EOP fields of the MCS 25 and the SCS 27. These fields are decoded respectively to MCS ENDOP and SCS ENDOP signals. As described below, these signals are precursors of the ENDOP signal produced by the merge unit 29 to synchronize instruction fetch and issue in the unit 14. While either signal is low, indicating that a microinstruction sequence has not completed, the output of the AND gate 70 will be low. This disables the instruction gates 71, 72, and 73 preventing entry of the left instruction, its associated C-bit, and the right instruction into the CIRL 20 and CIRR 21. While the output of the AND gate is low, the inverter 75 keeps the gates 77, 79, and 80 active, which recirculates the contents of the CIR 19.

When the ENDOP signal is active, both the MCS and SCS ENDOP signals are active. In response, the fetch and issue unit 14 sends either a single instruction with its C-bit into CIRL 20, or the left instruction and its C-bit of a compounded pair of instructions into the CIRL 20 and the right instruction CIRR 21. The instructions and the accompanying C-bit are gated by the gates 71, 72, and 73. While being shifted into the compound instruction register, the OP codes are simultaneously registered in the CSARs. Entry into the compound instruction register drives the decode stage of the microcode generator. The OP code from the CIRL 20 is latched in the CSAR 24c and used for the initial OP breakout address operation. In a compounded pair of instructions, the OP code from the CIRR 21 is gated to CSAR 26c and used to access the SCS 27.

MICROINSTRUCTION MERGING

The MCS 25 contains complete microcode for all instructions, while the SCS 27 contains only the pieces of microinstruction needed for instructions which are compoundable as right-hand or second instructions, in a compounded pair. Merger, if necessary, is accomplished in the merge unit 29 as shown in FIG. 5. The task of the merge unit 29 is to generate a composite microinstruction by merging fields from the SCS 27 with fields from the MCS 25 and to generate other fields by decoding the contents of the CIRL 20 and CIRR 21. FIG. 5 shows the relevant fields of a merged mircoinstruction registered in the AIR 32a, which immediately follows the merge unit 29. In the SCISM embodiment, the complete microinstruction includes at least 34 fields. At least fields 1-5, 7-9, 12, and 34 are unchanged from their form in the MCS 25. At least fields 6, 10, 11, and 19 can be changed from their condition in the MCS 25 by the contents of corresponding fields in the SCS 27. In addition, field 19 can be altered from hardware (HW) 84 in the merge unit 29. The source of data for fields 6, 10, 11, 13, and 19 is determined by the state of multiplexers 90, 91, 92, and 94, respectively. The invention provides for specific actions by these multiplexers which generate a merged microinstruction.

The first action which is possible to form a merged microword is executed by substituting a field value from the SCS 27 for the value of a field in a microinstruction from the MCS 25. Direct substitution is warranted in control fields where it is known ahead of time that the hardware resource is dedicated to the execution of one of the instructions, and never needed by the other. In other words, it is not a shared execution unit. For example, the instruction in the CIRL 20 could have the form Add Register 5, 3, while the instruction in the CIRR 21 could have the form Add Register 1, 2. Assume that the first Add instruction uses execution unit 34 in FIG. 1, while the second uses execution unit 36. No forwarding of operands to the second execution unit is necessary, so the fields used to control operand fetching, and those used to control the second execution unit need not be modified from the state provided by SCS. The microinstruction in the MCS 25 for the left instruction has fields which control the execution unit 34 and the fetching of operands for that execution unit are coded to execute the instruction in normal fashion. The fields to control the execution unit 36 do exist in the microinstruction in the MCS 25, but are not needed to control execution of the first Add, since it executes in the execution unit 34 only. The microinstruction in the SCS 27 for the second Add instruction is always coded to use the execution unit 36, since the SCISM architecture requires execution of an Add Register as a second instruction to occur in this execution unit. Fields needed to control execution in the execution unit 34 do not exist in the SCS 27.

Execution of the second instruction of a compound instruction pair can never alter the execution of the first, so the microcode fields needed to execute the first instruction are always gated unchanged from the MCS 25 into the AIR 32a. In the case where, execution of the first instruction does not interfere with operand fetch or execution unit operation for the second instruction, the value for control field F in the SCS 27 which controls the second execution unit is directly substituted for the value of the F field coming from the MCS 25. Appropriate merge unit logic for field substitution in this manner is illustrated in FIG. 6A. Substitution of a field value from the SCS 27 by the merge unit 29 is implemented in an AND gate 85 controlled by the C-bit field of the CIRL. When the C-bit is set the AND gate 85 is enabled to provide the control value for the field F, and inverter 86 disables the AND gate 87, blocking the output of MCS 25. The OR gate 88 controls the entry into the F field location in the AIR 32a, receiving as inputs the output of the AND gate 85 and the AND gate 87.

FIG. 6B permits either the MCS or the SCS to conveniently control a shared hardware resource, but with the limitation that both cannot control it simultaneously. The merge unit 29 has no a priori way to determine whether the left or right instruction will use the shared execution resource controlled by field F, rather this information is implicit in the setting of the C-bit. The operation of the circuit of FIG. 6B is similar to that of the non-shared resource example just described, with the exception that the default value of control field F' will be zero. The C-bit then merely enables AND gate 85, the output of which is ORed directly with control field F' currently addressed in MCS 25.

Assume now that the previous case is modified such that the result of the first instruction is required (logically) as an input to the second instruction. In order to absolutely limit the amount of memory required for the control stores, the SCS 27 contains only microwords necessary to support independent execution of instructions by the interlock-reducing unit. Therefore, in the face of a data interlock certain fields in the microinstruction which control operation of the interlock-collapsing execution unit need to be generated by logic rather than by using the SCS 27. For example, assume that the left instruction is SR 1,4 (subtract the contents of a register R4 from the contents of a register R1 and place the results in register R1). Assume that the second instruction is AR 3,1. Since the AR instruction follows the SR instruction, non-parallel execution of the instructions will find the SR instruction altering results in register R1 at the time the AR instruction executes. However execution of these instructions in parallel requires a control field for the interlock-collapsing execution unit which in effect tells the execution unit that it must combine three operands (R3+R1-R4) and place the contents in register R3.

FIG. 7 illustrates how the merge unit 29 detects an interlock dependency and appropriately conditions a microinstruction field F controlling the second execution unit to indicate the interlock dependency case.

The structure and operation of FIG. 7 represents field value substitution in the merging unit 29 by means of hardware which recognizes an interlock condition existing between two instructions of a compounded pair. It assumes IBM System/370-type instructions. Thus for the SR and AR instructions discussed above, a data dependency can be discovered by inspection of the first two bytes of each instruction. This first byte of each instruction includes the instruction's OP code, while the second byte identifies the registers containing the instruction operands. The field substitution hardware in the merge unit 29 therefore must identify particular pairs of instructions and determine whether the two instructions are operating on a common operand. Particularly, if the instructions are compounded based upon rules which first categorize the instructions and then permit compounding between predefined categories, the merge unit 29 must be able to identify the categories and test for operand register equivalence.

These conditions are tested in FIG. 7. The operands of the left and right instructions are decoded, respectively, in operand decoders 91 and 92. If the decoder 91 looks only for category one (CAT I) instructions, it will activate its output when the left instruction is in that category. Assume that the right decoder 92 activates a respective signal when the right instruction is in one of a group of instructions categories which can be compounded with instructions in category one. The output of the left decoder 91 is fed then to each of plurality of AND gates 97, 98, with each AND gate also receiving a respective category validation signal from the right decoder 92. An operand testing circuit 94 receives the register identification fields (termed RA and RB) from each operand to test for data interlock between the instructions. The output of the detector 94 is activated only if the first register of the first instruction is identical with either register of the second instruction, said identity meaning that the result of the first instruction is logically required as an input for the second instruction. This output is also fed to the AND gates 97 and 98. An OR gate 99 collects the outputs of the AND gates 97 and 98, and the output of the OR gate 99 is fed to AND gate 90 which is enabled if the C-bit is a one. The output of AND gate 90 is fed to a GATE circuit 100. The GATE circuit 100 receives two inputs, one an input from a multiplexer 107 which conforms essentially to the substitution circuit illustrated in FIG. 6, and a second from a hardware field set circuit 105. The hardware field set circuit 105 operates to set the field F to a value appropriate for 3-operand operation of the execution unit 36 when an interlock situation occurs.

Thus, assume that the two instructions discussed above have been loaded into the CIRL 20 and CIRR 21, that both instructions are category one instructions, and that category one instructions are compoundable. The OP code of the right instruction is passed to the decoder 92 and the register field contents of the instruction is passed to the detector 94. At the same time, the OP code of the left instruction is provided to the decoder 91 and the operand register field contents to the detector 94. Since the left instruction is utilizing register R1 to store updated results and the right instruction is using the contents of register R1, interlock exists and the field value for the circuit 105 must be entered into F field in AIR 32a. This is accomplished by activation of the CAT I outputs from the decoders 91 and 92 and activation of the output from the detector 94. This activates the AND gate 97 whose output is fed through OR gate 99 to the input of AND gate 90. The output of AND gate 90 is fed to the control input of the gate 100. Gate 100 is designed to select the output of the circuit 105 when the output of the OR gate 99 is active. Therefore, the field F will be set to a value determined by the hardware circuit 105.

MERGING MICROINSTRUCTION SEQUENCES WITH UNEQUAL LENGTHS

Not all paired instructions require the same number of cycles to execute. Consequently a mechanism must be provided in the merge unit to accommodate the merging of microinstruction sequences of unequal length. Preferably, the present embodiment of the compounding preprocessor limits all paired or compounded instructions to one-, or two-cycle microinstruction sequences. In this regard, the instructions require one or two EX cycles. Depending on the pairing, the one-cycle microinstruction sequence may need to be executed during the first or second EX cycle of the two-cycle sequence. When the left instruction requires a single cycle, it is permitted to execute immediately. However, if the right instruction requires a single cycle, its microinstruction is delayed. In both cases, the unequal lengths are accommodated by synchronizing the ends of the sequences to the same pipeline clock period.

When the left instruction requires a single execution cycle, the first microinstruction for each operation is fetched in the instruction decode cycle and fed into the pipeline beginning at the AIR register 32a. In the last microinstruction of every microcode sequence, the EOP bit is set to signal the decoding hardware to begin decoding the next instruction. In the single-cycle instruction, this field activated. The next address field of each potentially-compoundable single cycle microinstruction sequence in the MCS 25 points to a no operation (NOP) microinstruction which has its EOP bit set. The NXA value for the NOP microinstruction points to the NOP microinstruction. The merge unit 29 fetches this NOP microinstruction from the MCS 25 and merges it with the second microinstruction fetched from the SCS 27 for the right-hand instruction in the pair during the second cycle. The NOP microinstructions may be merged with other microinstructions without interference. That is, it makes no use of data flow functions, so its control fields are coded with no-OPs or default codes. The last microinstruction of the sequence for the right instruction will also have its EOP bit set, and when EOP is detected for both sequences, the compound instruction is completed, and the pipeline advances to the next machine instruction.

The situation just described is illustrated in FIG. 8 where the outputs of the MCS 25 and SCS 27 are shown during two succeeding periods of the pipeline clock. These periods are labelled t and t+1. The left-hand instruction is the single-cycle instruction and the microinstruction pointed to by the left OP code (OPL) is coded with the pointer to the NOP instruction and its NXA field, with its EOP bit set, and with appropriate values in its control fields for execution of the left instruction. This microinstruction is output from the MCS at pipeline clock period t simultaneously with the first microinstruction of a multicycle sequence output from the SCS for the right instruction. The first SCS microinstruction is at the address location indicated by the OP code of the right instruction (OPR). The NXA field of the first SCS microinstruction has a pointer NXT to the address of the next microinstruction in the sequence, its EOP bit is set to zero, and its control fields are appropriately coded for execution unit 36. In pipeline clock period t+1, the NOP instruction is output from the MCS, while the instruction at address NXT is output by the SCS. Now the EOP bits for both sequences are set, permitting generation of the ENDOP bit by the merge unit, and fetching the next machine instruction.

FIG. 9 illustrates the logic required to implement the ENDOP generation during the sequence-equalizing process illustrated in FIG. 8. In FIG. 9, the merge unit 29 includes an AND gate which receives the value in the EOP field of the MCS 25. The gate 124 also receives the OR gate 122. The OR gate 122 receives the EOP of the SCS 25 and also the C-bit value, inverted at 120. Now, assume that a left instruction requiring a single execution cycle is compounded with a right instruction requiring more than one execution cycle. The microinstruction sequence for the left instruction requires only a single microinstruction, while the sequence for the right instruction requires more than one microinstruction. During pipeline cycle period t, the first microinstruction from the MCS activates the EOP which is fed to the AND gate 124. However, the C-bit is inverted at 120 and EOP bit for the SCS 25 has not yet been set. Therefore, the output of the AND gate 124 is low, disabling the ENDOP signal. At pipeline cycle period t+1, the EOP output by the SCS 25 is activated, the output of the OR gate 122 rises, thereby activating the ENDOP signal output by the AND gate 124. Activation of the ENDOP signal synchronizes fetching of the next machine instruction to completion of all execution cycles required for the current compounded pair.

The significance of the inverter 120 is that when no valid instruction is in the CIRR 21, the SCS EOP is forced on, enabling the AND gate 124 to generate the ENDOP signal when the microinstruction sequence for the left instruction is completed, signified by activation of the EOP from the MCS 25.

The technique and mechanism for merging NOP microinstruction from the MCS with a microinstruction sequence from the SCS is not limited to the example shown. Generally, the appending of NOP microinstruction to a microinstruction sequence output by the MCS can be used whenever the MCS sequence is shorter than the SCS sequence. It is merely required that the NXA field of the NOP microinstruction point to itself.

An alternative implementation to FIG. 9 is to latch the EOP output by the MCS sequencer and hold it in a latch until EOP is also detected for the SCS sequence. During this period, the MCS-controlled fields of merged microinstructions can be zeroed or set to NOP default values.

FIG. 10 illustrates the condition where, in response to the issue of a pair of compounded instructions, the microinstruction sequence generated for the right instruction is shorter than that for the left. In this case, decoding of the right OP code (OPR) is delayed until after the first execution cycle of the left instruction. One reason for such delay would be if both instructions required access to a common resource. For example, if the left instruction were an RX-format ADD, two execution cycles would be required. The first would fetch an operand from storage, while a second would add the operands. Assume that the right instruction is an RX-format LOAD instruction. This is a single execution cycle instruction in which the second operand is fetched from memory and placed in a register. Delay of the single microinstruction sequence for the right instruction is required in order to avoid resource conflict with the first cycle of the RX ADD instruction. When a single microinstruction sequence for the right instruction is delayed to align with the last of a two-microinstruction instruction sequence for the left instruction, the procedure is illustrated in FIG. 10. The first microinstruction of the first sequence is obtained from the MCS 25 by decoding the left operand (OPL). The microinstruction at this address, available at pipeline clock period t, is entered into the AIR 32a. In the following period of the pipeline clock (t+1), the microinstruction at address NXT is available from the MCS. The single microinstruction of the sequence of the right instruction is at the SCS address obtained by decoding the OP code of the right instruction (OPR). Since the right and left instructions are entered into the CIR 19 simultaneously, the OP code for the right instruction is available at pipeline clock period t. In the example, the merge unit detects the occurrence of the potential for resource conflict and retains the OP code for the right instruction until pipeline period t+1. In the meantime, during clock period t, the SCS continues to output the microinstruction at the addressed location. Merging of the microinstruction from SCS is prevented during pipeline period t. This effectively delays merging of the one-cycle microinstruction from the SCS until pipeline period t+1.

FIG. 11 illustrates an implementation of the procedure of FIG. 10. Assume that the RX-format ADD instruction in CIRL 20 decodes to "5A" (hexidecimal) while the RX-format LOAD instruction decodes to "58". In this instance, the outputs of the decoders 120 and 121, respectively, are activated, activating the output of the AND gate 122. The AND gate 122 is fed to a counter 123 which counts the number of pipeline clock periods by which the OPR address is to be delayed. In this case, only a one period delay is required, implying that the counter 123 can be a latch which is set by activation of the AND gate 122 during pipeline clock period t and reset at the beginning of period t+1. The OR gate 124 collectes the outputs of other circuits which detect the occurrence of resource conflict between pairs of instructions and count down the number of clock periods necessary to expiate the conflict. The OR gate 124 activates its output in response to the resource conflict detection circuits which feed it. This output will remain active for a number of pipeline cycle clock periods counted by the respective counter which activates the OR gate. This output is labeled HOLD SCSAR. The HOLD SCSAR signal will remain active for as long as is required to resolve resource conflict. For so long as this signal is activated, a HOLD CLOCK circuit will prevent the provision of the pipeline clock to the SCSAR 26c. When the HOLD SCSAR signal is inactive, the clock is provided to the SCSAR and its contents are changed. This is represented by the LSSD latch pair L1/L2. This configuration assumes that the pipeline clock is a multi-phase signal in which the first phase is fed to the first latch section L1 of the CSAR 26c, while a second phase is fed to the second latch section L2. The HOLD CLOCK circuit 125 blocks provision of the clock phase which operates the L2 section of the SCSAR. Thus, when the OP code is available at the input of the CIR 21, it is latched into the L1 section of the SCSAR during the same period of the pipeline clock. Thereafter, it is held in the L2 section of the SCSAR if the hold clock signal is active. Following deactivation of the HOLD SCSAR signal, the L2 section of the SCSAR 26c latches the OP code of the next right instruction and the SCSAR 26c provides the address OPR to the SCS 27.

While the invention is particularly described with reference to a preferred embodiment, it is to be appreciated that the method focuses on the generation of microcode in a scalable compound instruction machine. Machine-level instruction sets, other than those executed by the IBM System/370, are contemplated. The scope of the invention also includes compounded instruction groups including more than two instructions.

* * * * *

Images
View Cart Add to Cart
Hit List Previous Next Top
Home Quick Advanced Pat Num Help

United States Patent *5,386,531*
Blaner , et al. **January 31, 1995**

Computer system accelerator for multi-word cross-boundary storage access

Abstract

An instruction processing unit (IPU) and a storage array, a storage-to-instruction-processing-unit interface, including a hardware accelerator for cross-boundary storage access with a cross-boundary buffer for providing residual read and write data in support of high speed block concurrent accessing of multi-word operands of a computer system. A cross-boundary buffer (CBB) is used, coupled to a write rotating shifter, a write merger (WMERGE) and a write merge controller (WMCTL) which is coupled for an input to said control register (CREG) for sequencing data transmitted on the data bus for merger with data contained in the cross-boundary buffer (CBB) by the write merger before it is latched in a data bus out register, and for simultaneously also latching the data in the cross-boundary buffer (CBB), and for writing data from the data bus out register into the storage array in the next clock cycle of the instruction processor at the doubleword address addressed. The cross-boundary buffer (CCB) is also coupled to a read rotating shifter (RROTATE), a read merger (RMERGE) and a read merge controller which responds to control instruction sequencing. The storage-to-instruction-processing-unit interface operates on multiple words, with residues from a second and subsequent accesses allowing continuation of the accessing process beyond two memory words. The hardware can repeat a second microword until an operand of arbitrary length is transferred. The interface permits efficient data transfer to be interrupted and resumed at a desired point, for efficient execution of Load Multiple and Store Multiple operations.

Inventors:	**Blaner; Bartholomew** (Newark Valley, NY); **Eberhard; Raymond J.** (Endicott, NY); **Jeremiah; Thomas L.** (Endwell, NY); **Mack; Michael J.** (Endicott, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**700732**
Filed:	**May 15, 1991**

Current U.S. Class: **711/201**

Intern'l Class:	G06F 012/04; G06F 012/06
Field of Search:	395/375,250,400,425

References Cited [Referenced By]

U.S. Patent Documents

3602896	Aug., 1971	Zeheb.	
3916388	Oct., 1975	Shimp et al.	395/250.
4189772	Feb., 1980	Liptay.	
4435792	Mar., 1984	Bechtolsheim	365/230.
4449185	May., 1984	Oberman et al.	
4502115	Feb., 1985	Eguchi.	
4520439	May., 1985	Liepa.	
4814976	Mar., 1989	Hansen et al.	
4823302	Apr., 1989	Christopher	395/425.
4868553	Sep., 1989	Kawamata	340/731.
4888687	Dec., 1989	Allison et al.	
5168561	Dec., 1992	Vo	395/425.

Other References

IBM TDB, vol. 29, No. 12, May 1987 L. J. LaBalbo et al.
IBM TDB, vol. 32, No. 2, Jul. 1989, G. F. Grohoski et al.
IBM TDB, vol. 20, No. 9, Feb. 1978, C. D. Holtz et al.
IBM TDB, vol. 25 No. 7A, Dec. 1982, A. Y. Ngai et al.
Technical Newsletter No. SN22-5279, Apr. 17, 1989, S370-01.

Primary Examiner: Lall; Parshotam S.
Assistant Examiner: Mohamed; Ayni
Attorney, Agent or Firm: Augspurger; Lynn L.

Claims

What is claimed is:

1. A data processing system having an instruction processing unit (IPU) and a storage array organized on

word boundaries or multi-word boundaries where a word is some number of consecutive bytes representing the basic unit of computation for said instruction processing unit, comprising:

said instruction processing unit and said storage array; and hardware accelerator means for cross-boundary storage access to said storage array including a cross boundary buffer means for providing residual read and write data to said instruction processing unit in support of high speed, block concurrent accessing of multi-word operands of said system and for operating on multiple words, with residues from a second and subsequent accesses enabling continuation of the accessing process by said hardware accelerator means beyond two memory words to span more than a word boundary and to allow high speed block-concurrent accesses to and from said storage array for load multiple and store multiple instructions.

2. A data processing system according to claim 1 wherein said hardware accelerator means includes means for providing off boundary alignment handling which is detected and performed implicitly by said hardware accelerator means.

3. A data processing system according to claim 2 wherein said off boundary storage alignment handling is an automatic function of said hardware accelerator means without the use of explicitly coded instructions to perform the automatic function.

4. A data processing system according to claim 1 wherein said hardware accelerator means includes means for automatically processing multiword storage operands during block concurrent accesses to said storage array.

5. A data processing system according to claim 1 wherein said hardware accelerator means includes means for automatically processing data which may straddle a memory boundary in said accesses to said storage array.

6. A data processing system according to claim 1 wherein said hardware accelerator means includes means for fetching multiple bytes of a word and for combining them with other bytes on subsequent accesses to said storage array.

7. A data processing system according to claim 1 wherein said hardware accelerator means includes means for reading multiple bytes from memory in a single block-concurrent access to said storage array and means for storing block-concurrent multiple bytes of a word.

8. A data processing system according to claim 7 wherein a storage cache directory is provided for the memory and wherein handling of cross-boundary reads and writes is not limited to those within a cache line or a memory page.

9. A data processing system according to claim 8 wherein said hardware accelerator means includes means for repeating the use of said cross boundary buffer means for handling operands of an arbitrary length,

10. A data processing system according to claim 1 wherein said storage array is provided with a general register, an address register and a control register for respectively receiving signals over an address bus

and a control bus from said instruction processing unit, and wherein said hardware accelerator means includes as pad of said cross boundary buffer means a cross boundary buffer, and further includes a merge unit, a controller for said merge unit, microcode for said controller for said merge unit for providing control for said hardware accelerator means during a storage access,and a rotating means for rotating a word unit or multi-word unit being transferred to or from said general register during a storage access.

11. A data processing system according to claim 1 wherein said storage array is provided with a general register, an address register and a control register for respectively receiving signals over an address bus and a control bus from said instruction processing unit, and wherein said hardware accelerator means includes as pad of said cross boundary buffer means a cross boundary buffer, and further includes both a read and a write merge unit, a controller for each of said merge units, microcode for said controller for said merge unit for providing control for said hardware accelerator means during a storage access, and a rotating means for rotating a word unit being transferred to or from said general register during a storage access.

12. A data processing system according to claim 1 wherein said storage array is provided with a general register, an address register and a control register for respectively receiving signals over an address bus and a control bus from said instruction processing unit, and wherein said hardware accelerator means includes as pad of said cross boundary buffer means a cross boundary buffer, and further includes both a read and a write merge unit, a controller for each of said merge units, microcode for said controller for said merge unit for providing control for said hardware accelerator means during a storage access, and a read and a write rotating means for rotating a word unit being transferred to or from said general register during a storage access.

13. A data processing system according to claim 1 wherein said storage array is provided with a general register, an address register and a control register for respectively receiving signals over an address bus and a control bus from said instruction processing unit, and wherein said hardware accelerator means includes as pad of said cross boundary buffer means a cross boundary buffer, and further includes both a read and a write merge unit a controller for each of said merge units, microcode for said controller for said merge unit for providing control for said hardware accelerator means during a storage access, and a microcode selection means responsive to the length of said multi-word operands for automatically selecting an appropriate microcode sequence.

14. A data processing system according to claim 1 wherein said storage array is provided with a general register, an address register and a control register for respectively receiving signals over an address bus and a control bus from said instruction processing unit, and wherein said hardware accelerator means includes as pad of said cross boundary buffer means a cross boundary buffer, and further includes both a read and a write merge unit, a controller for each of said merge units, microcode for said controller for said merge unit for providing control for said hardware accelerator means during a storage access, and a microcode sequencing means responsive to the length of said multi-word operands for automatically disabling a microcode sequence termination control specified in a microword until said multi-word operand has been completely accessed.

15. The data processing system according to claim 1 including rotating means for rotating an output of said storage array and operatively coupled to a data bus such that the first byte of data is positioned at a

starting byte address.

16. The data processing system according to claim 15 wherein a bus register is provided, and means are provided for merging data in said bus register with data in a cross boundary buffer of said cross boundary buffer means during a storage access.

17. A data processing system having an instruction processing unit and a storage array organized on multi-word boundaries where a word is some number of consecutive bytes representing the basic unit of computation for a processor, comprising,

an instruction processing unit (IPU); and

a storage array (STORAGE),

an interface between said storage array (STORAGE) and said instruction processing unit (IPU);

said interface including an address bus (ABUS) which supplies a storage address from the instruction processing unit (IPU) IPU to said storage array (STORAGE), a control bus (CBUS) for providing a command which indicates the kind of storage address and whether the address is a read address, a write address, the length of the access in bytes, and a word byte boundary address (ABUS 29:31); said interface also including a data bus (DBUS), and a data bus out register (DBUSOUT) for said instruction processing unit (IPU);

said instruction processing unit having a control store array (CS) which contains microwords which direct operations of the instruction processing unit (IPU) and storage array, a general register array (GR), an address register (AREG), a control register (CREG) and a read register (RREG), an instruction sequencing means (MSEQ) for fetching microwords from the control store array into a microinstruction register (MIR), an address generation adder (AGEN) having an output which includes a multi-word address (latched AREG 0:28) for addressing the storage array and said word byte boundary address (ABUS 29:31);

a cross boundary buffer means including a cross boundary buffer (CBB) coupled to a rotating shifter, a merging means (WMERGE) and to a merge controller (WMCTL) which is coupled for an input to said control register (CREG); said instruction sequencing means (MSEQ) controlling data transmitted on said data bus for merge with data contained in said cross boundary buffer (CBB) of said cross boundary buffer means by said merger means before it is latched in said data bus out register (DBUSOUT) for said instruction processing unit (IPU) and for simultaneously also latching the data in said cross boundary buffer (CBB), and for writing data from said data bus out register (DBUSOUT) for said instruction processing unit (IPU) into the storage array (STORAGE) in the next clock cycle of said instruction processing unit at the multi-word address addressed by said multi-word address.

18. A data processing system according to claim 17 wherein said cross boundary buffer (CCB) means is coupled to a read rotating shifter (RROTATE), and there is provided a read merger means (RMERGE) and a read merger controller which responds to control signals in the control register (CREG), and wherein the said instruction sequencing means (MSEQ) on a read access generates a storage address

which together with a command is latched in said address register (AREG) and said control register (CREG) respectively, and wherein in a subsequent cycle information in said address register (AREG) is used to read a multi-word from said storage array, said multi-word being latched in a provided read cross boundary buffer (CBB) and at the same time being passed through a provided read merger means (RMERGE) where it may be merged with data already in said read control boundary buffer, said merger means being under the control of said read merger controller (RMCTL).

19. A data processing system according to claim 17 wherein said cross boundary buffer means (CBB) is coupled to a write rotating shifter, write merging means (WMERGE) and a write merger controller (WMCTL) which is coupled for an input to said control register (CREG), said instruction sequencing means (MSEQ) controlling data transmitted on the data bus for merger with data contained in a write cross .boundary buffer (CBB) of said cross boundary buffer means by said write merger means before it is latched in said data bus out register (DBUSOUT), and for simultaneously also latching the data in said write cross boundary buffer (CBB), and for writing data from said data bus out register (DBUSOUT) into the storage array (STORAGE) in the next clock cycle of said instruction processing unit at a multi-word address addressed by said multi-word address.

20. A data processing system according to claim 19 wherein said cross boundary 1Suffer (CCB) means has a read cross boundary buffer (RCCB) which is coupled to a read rotating shifter (RROTATE), and to a read merger means (RMERGE) and to a read merge controller which responds to control signals in the control register (CREG), and wherein the said instruction sequencing means (MSEQ) on a read access generates a storage address which together with a command is latched in said address register (AREG) and said control register (CREG) respectively, and wherein in a subsequent cycle information in said address register (AREG) is used to read a multi-word from said storage array, said multi-word being latched in said read cross boundary buffer (RCCB) and at the same time being passed through said read merger means (RMERGE) where it may be merged with data already in said read cross boundary buffer, said merger means being under the control of said read merger controller (RMCTL).

21. The data processing apparatus according to claim 1 wherein the instruction processing unit includes a microword from a control store to complete a transfer, said transfer being invoked on a microcode branch on a carry out of an adder.

22. The data processing apparatus according to claim 1 wherein the instruction processing unit includes a merge unit, and a merge control microword allows data transfer to be interrupted by other microwords and resumed at an arbitrary desired point.

23. The data processing apparatus according to claim 1 wherein the instruction processing unit includes a control store array which contains microwords which direct the operations of the instruction processing unit and storage, and a microsequencer, said microsequencer and a cross boundary buffer means providing residual accessing of multi-word operands of said system for executing for multiple words load multiple and store multiple instructions.

Description

FIELD OF THE INVENTION

This invention relates to computer systems and particularly to an accelerator system employing hardware with microcode for increasing the performance of multi-word cross-boundary storage accesses.

BACKGROUND OF THE INVENTION

Processor storage arrays, ranging from smaller high-speed caches to large, comparatively low-speed random access memories (RAM), are commonly organized on an n-word basis where a word is some number of consecutive bytes representing the basic unit of computation for a processor, and n is a positive integer. A storage access, by definition, references one n-word. This organization allows for efficient busing structures between the processor and storage and simplifies addressing the storage arrays, which may be composed of many individually-addressable storage arrays. For example, in the S/370 architecture, as illustrated by the IBM Corporation publication entitled "The ESA/370 CPU Architecture", published 1989, SA22-7200, the 8-bit byte is the smallest unit of addressable storage, and 4 consecutive bytes constitute a word--the basic unit of computation. Present day S/370 processors have storage arrays organized on 2-word (doubleword or DW) and 4-word (quadword or QW) boundaries.

One example of a possible S/370 processor development dealing with operands which are not aligned on n-word boundaries is the operand fetch logic illustrated by U.S. Pat. No. 4,189,772 to Liptay issued Feb. 19, 1980 entitled "Operand Alignment Controls for VFL Instructions".

Another patent in the general area is U.S. Pat. No. 3,602,896, issued Aug. 31, 1971 to D. Zaheb entitled "Random Access Memory with Flexible Data Boundaries" which discloses a random access memory where an accessed data word may overlap one memory word boundary into an adjacent memory word. The initial byte location is provided along with a number of bytes (up to one word length).The partitioning of the cache required by the disclosure imposes unacceptable circuit delay in the cache access critical path.

Yet another patent in the general area is U.S. Pat. No. 4,435,792 issued Mar. 6, 1984 to Bechtolsheim entitled "Raster Memory Manipulation Apparatus" wherein a computer can access memory over word boundaries. A shifter and offset data (i.e. length of access and boundary) are used to align the data, but again the partitioning necessitates placing an incrementer, multiplexer, and decoder in the main memory address path which imposes unacceptable delays.

These kinds of additions in the main memory address path are imposed in U.S. Pat. No. 4,520,439 issued May 28, 1985 to Liepa about "Variable Field Partial Write Data Merge" which discloses accessing memory and crossing over word boundaries by providing a starting address, read/write information, start location and access length. Words across word boundaries are merged with not needed bits being masked using a write data interface. This bit masking approach is unrelated to our work.

U.S. Pat. No. 4,814,976 issued Mar. 21, 1989 disclosed a "RISC Computer with Unaligned Reference Handling and Method for the Same" wherein is shown accessing across boundaries of a cache memory using a shift/merge unit which requires explicit coding in order to handle off-boundary accesses, while U.S. Pat. No. 4,814,553 issued Sep. 19, 1989 to Kawamata related to a "Raster Operation Device" which shows a way of crossing word boundaries based on shift width, bit width of data of a raster screen display. No provision here was made for data fetching, retaining a residue of the second word accessed in the read-modify-write operation, and other features required for cross storage boundary accesses of a computer memory.

Other art in the general field but thought unrelated to our own developments includes U.S. Pat. No. 4,449,185, issued May 15, 1984 to Oberman et al, which related to the "Implementation of Instructions for a Branch which can Cross One Page Boundary"; U.S. Pat. No. 4,502,115 issued Feb. 26, 1985 to Eguchi which related to a "Data Processing Unit of a Microprogram Control System for Variable Length Data"; U.S. Pat. No. 4,888,687 issued Dec. 19, 1989 to Allison et al relating to a "Memory Control System" which is not directed to high speed accesses which can handle block concurrent stores and accesses.

Within IBM, as shown by the IBM Technical Disclosure Bulletin, Vol. 25 No. 7A, December 1982, A. Y. Ngai and C. H. Ngai proposed "Boundary Crossing with a Cache Line". The Ngais' publication included a byte shifter for data alignment, pp. 3540. This technical disclosure facilitates cross-boundary fetching by partitioning the cache memory into two segments, A and B, which are basically even and odd addressed arrays. This partitioning necessitates placing an incrementer and multiplexer on the segment A cache address and multiplexers on the outputs of the cache arrays. As we have said, such partitioning runs counter to our developments since additional circuit delay is added to the cache critical path.

"Mark Bit Generator" was a topic covered in another TDB, Vol. 20. No. 9, of February 1978, by C. D. Holtz and K. J. Parchinski; while also in the data storage general field, the TDB included the item "Storage Byte Mark Decode With Boundary Recognition" by L. J. LaBalbo, W. L. Mostowy and A. J. Ruane Vol. 29 No. 12, May 1987, p. 5264 and the item by G. F. Grohoski and C. R. Moore entitled "Cache Organization to Maximize Fetch Bandwidth" in Vol. 32 No. 2 in July 1989, p. 62.

Other internal IBM developments which dealt with cross-boundary buffers, in addition to the Ngai publication, could be cited as a product called "RACETRACK 11" which was proposed and as illustrated by U.S. Ser. No. 07/291,510, filed Dec. 29, 1988, now abandoned, entitled "Hardware Implementation of Complex Data Transfer Instructions", p. 24. This machine prototype was provided for the LM (Load Multiple) instruction a register for storing the entire doubleword called a cross-boundary buffer (20-66 in that application) which effected a save of the data destined for a general purpose register (GPR). A mask could be set with the data saved in this cross-boundary buffer and later used for merging with fetched data. For the LM instruction, the cross-boundary buffer was controlled by a combination of "mini-instructions" and a baroque hardware control mechanism to handle various circumstances of GPR loading and storage boundary alignment. Alternatively, the cross-boundary buffer could be controlled by microcode for microcoded execution of instructions with multi-word storage operands. Looping controls were provided to execute microwords repeatedly until the storage operand was consumed; however, if the length of the storage operand in doublewords was not an integral of the number of storage read microwords in the loop, machine cycles were wasted issuing nullified read microwords. Also provided for the STM (Store Multiple) instruction was a register for storing the entire doubleword called a save register (pp. 28-29) which effected a save of the data destined for storage. A mask could be set with the data saved in this save register and later used for merging with data fetched from a GPR and destined for storage. Controls for STM were provided by means analogous to those for LM. The save register could not be controlled by storage write microwords and was therefore limited in use to the STM instruction. A corresponding European Patent Application has been published as of the date of filing of the present application, claiming U.S. Ser. No. 07/291,510, filed Dec. 29, 1988 as a priority document.

Generally in a data processing system where the processor can access a memory that is organized on multi-word boundaries, the storage address is sent to memory along with the kind of access (read or write, and length of access). A doubleword memory organization is used in many systems.

In S/370 and similar architectures, the disparity between the smallest unit of addressable storage (a byte) and the basic unit of computation (the 4-byte word) on which the storage organization is based gives rise to the cross-boundary storage access phenomenon. A cross-boundary storage access requires two n-words to be accessed to complete the storage reference, and therefore takes twice the amount of time to process as a non-cross-boundary or on-boundary access. These problems give rise to other possibilities, some examples of which are contained in the detailed description of our inventions, to provide a further background to the developments which we have achieved.

SUMMARY OF OUR INVENTIONS

In accordance with our inventions, we have provided for a data processing system a hardware accelerator for cross-boundary storage access. Generally, the system will have a processing unit for a storage array organized on multi-word boundaries where a word is some number of consecutive bytes representing the basic unit of computation for a processor.

The hardware which we have provided buffers residual read or write data in support of high speed, block-concurrent accessing of multi-word storage operands of the system. With our improvements, cross-boundary alignment and handling is detected and performed implicitly by hardware, and no special instructions need to be explicitly coded to handle cross-boundary storage accesses.

In using our preferred embodiment, the cache directory of the memory (a main memory and auxiliary memory or high speed cache) is not partitioned, and thus the handling of cross-boundary reads and writes is not limited to those occurring within a cache line or within a memory page. The hardware accelerator of our preferred embodiment has the ability to handle operands of arbitrary length through repeated use of the cross-boundary buffer. We support block-concurrent accesses for operands spanning multiple memory words with cross-boundary fetch and store logic. Block-concurrent memory accesses are defined such that all bytes within a contiguous field (halfword, word, or doubleword, for example) appear to have been accessed in a single reference by a first processor to the memory. Thus, no other processor may modify bytes within the block during the course of fetching by the first processor, nor may another processor observe intermediate results during the course of storing data by the first processor.

Multi-word storage operands are processed by the hardware we have provided, and the processing is an automatic function of the hardware. This automatic handling includes the automatic handling of data which may straddle a memory boundary. There is no need to explicitly code loop-controlling instructions by an instruction which decrements the storage operand length, or one that says to branch if length not equal to zero, or other such explicitly coded instructions. Furthermore, no machine cycles are wasted in processing the multi-word storage operand, irrespective of its length. In our preferred embodiment, we provide for saving fetches of multiple bytes and combining them with other bytes on subsequent accesses. We read multiple bytes from memory in a single block-concurrent access, and we are able to block concurrently store multiple bytes.

In order to illustrate the data processing system having these features we have provided a detailed description of our preferred embodiments together with examples of how different instructions are handled. In order to illustrate our inventions, our preferred detailed embodiment has an instruction processing unit and a storage array with doubleword organization, with one word being equal to four bytes. The hardware accelerator employs for transfer between storage and the instruction processing unit a storage-to-instruction-processing-unit interface. This interface includes an address bus which supplies the doubleword address from the instruction processing unit to storage (ABUS(0:28)); a control bus which indicates the kind of storage address, including whether the address is a read address or a write address, and the length of the access in bytes; and a byte address (ABUS(29:31)). This interface includes a data bus and data-in (DIREG) and data-out (DOREG) bus registers. The instruction processing unit has a control store array which contains microwords which direct operations of the instruction processing unit and the storage array, a general register array, an address register, a control register, and a read register. There is also an instruction microsequencer for fetching microwords from the control store array into a microinstruction register, an address generation adder with the output of the address generation adder including an address for addressing a doubleword in the storage array and the byte address for addressing the byte within the doubleword.

There can be a single or plural accelerator cross-boundary buffers. We prefer a single cross-boundary buffer. This enables us to share the use of a cross-boundary buffer for reads and for writes. When used for writing, the cross-boundary buffer (CBB) is coupled to a write rotating shifter. In addition, for the hardware accelerator we provide a write merger (WMERGE) and a write merge controller (WMCTL) which is coupled for an input to said control register (CREG). Instruction sequencing controls data transmitted on the data bus for merger with data contained in the cross-boundary buffer by a write merger before it is latched in a data bus out register, and for simultaneously also latching the data in the cross-boundary buffer, and for writing data from the data bus out register into the storage array in the next clock cycle of the instruction processor at a location addressed by said doubleword address.

When used for reading, the cross-boundary buffer of the accelerator is coupled to a read rotating shifter, and there is a read merger and a read merge controller which responds to control signals in the interface control register. Here the instruction sequencing on a read access generates a storage address together with a command which are latched in said address register and control register respectively. Then, in a subsequent cycle, the address register information is used to read a doubleword from the storage array. The doubleword operand is latched in the cross-boundary buffer and at the same time is passed through the read merger means where it may be merged with data already in the cross-boundary buffer, said merger means being under the control of said read merge controller.

These and other improvements are set forth in the following detailed description. For a better understanding of the inventions, together with advantages and features, reference may be had to the co-pending applications for other developments we have made in the field. However, specifically as to the improvements, advantages and features described herein, reference will be made in the description which follows to the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS.

FIG. 1 shows our preferred IPU and storage interface.

FIGS. 2A and 2B combined which show more detailed level of the elements in FIG. 1.

FIG. 3 illustrates our combinatorial logic related to the write operation which begins with SAVE active and MERGE inactive, and the register connections for low order AREG register address bits.

FIG. 4 illustrates our combinatorial logic related to the write operation which begins with SAVE active and MERGE inactive, and the FIGURE continues from left to right the logic illustrated in FIG. 3.

FIG. 5 illustrates the CBB and our combinatorial logic related to the write operation which begins with SAVE active and MERGE inactive, and the FIGURE continues the logic illustrated in FIG. 4.

FIG. 6 illustrates our combinatorial logic related to a read operation which begins with SAVE active and MERGE inactive with the register connections for low order AREG register address bits.

FIG. 7 illustrates our combinatorial logic related to a read operation which begins with SAVE active and MERGE inactive with the next stage following that illustrated by FIG. 6.

FIG. 8 illustrates our combinatorial logic related to a read operation which begins with SAVE active and MERGE inactive with the next stage following that illustrated by FIG. 7.

FIG. 9 illustrates our combinatorial logic related to a read operation which begins with SAVE active and MERGE inactive with the next stage following that illustrated by FIG. 8.

FIG. 10 illustrates our combinatorial logic related to a read operation which begins with SAVE active and MERGE inactive with the next stage following that illustrated by FIG. 9.

FIG. 11 illustrates our combinatorial logic related to a read operation which begins with SAVE active and MERGE inactive with the next stage following that illustrated by FIG. 10.

FIG. 12 shows the detailed logic for the writing of data to the storage array under control of the STORAGE READ/WRITE CTLS unit illustrated in FIG. 2.

FIG. 13 shows a timing diagram of the CBB operation for a Load Multiple instruction.

FIG. 14 shows another timing diagram for a Load Multiple instruction where the MSEQ detects that a cross-boundary read is occurring.

FIG. 15 shows another timing diagram for a Load Multiple instruction where data for the last read is supplied entirely from the CBB.

FIG. 16 shows a timing diagram for an on-boundary Load Multiple instruction.

FIG. 17 shows a timing diagram of the CBB operation for a Store Multiple instruction of 8 bytes or less where the MSEQ detects that a cross-boundary write is occurring.

FIG. 18 shows a timing diagram of the CBB operation for a Store Multiple instruction of more than 8 bytes where the MSEQ detects that a cross-boundary write is occurring.

FIG. 19 shows a timing diagram of the CBB operation for a Store Multiple instruction of more than 8 bytes where the MSEQ detects that a cross-boundary write is occurring and data for the last store comes entirely from the CBB.

FIG. 20 shows a timing diagram for an on-boundary Store Multiple instruction of more than 8 bytes.

Our detailed description follows as pads explaining our preferred embodiments of our inventions provided by way of example.

DETAILED DESCRIPTION OF THE INVENTIONS

Before considering our preferred embodiments it may be worthwhile to illustrate, by way of example, some possibilities which we have further considered and discarded. As we have said before, in S/370 and similar architectures, the disparity between the smallest unit of addressable storage (a byte) and the basic unit of computation (the 4-byte word) on which the storage organization is based gives rise to the cross-boundary storage access phenomenon. Furthermore, a cross-boundary storage access requires two n-words to be accessed to complete the storage reference, and as a consequence two times the period needed for processing as a non-cross-boundary or on-boundary access.

For example, consider the following S/370 Load instruction, with data arranged in storage as shown in Example 1A. ##STR1##

In this example, storage is organized on doubleword boundaries. Each lower-case letter represents a byte of data. The Load instruction will load general register (GR) 1 with a word of data from address 0. Thus, GR 1 may be loaded with a single access to DW 0.

Now consider the same operation, only this time the storage access begins at address 6, as illustrated by Example 1B.

EXAMPLE 1B ##STR2##

Two storage accesses are required to complete the operation: the first, a load of bytes ab from DW 0, and the second, a load of bytes cd from DW 1. The operation takes twice as long to complete as the previous example, even though the same number of bytes were loaded as by the instruction in the previous example.

The problem is multiplied when single instructions are permitted to access many words of storage, as, for example, the SS-format instructions in S/370, some of which may have storage operands up to 64 words in length. Assuming a doubleword storage organization requiring one machine cycle to access one doubleword, such an instruction would take 32 cycles to completely access a single storage operand if all accesses were on-boundary. If the accesses were all cross-boundary, however, 64 cycles would be required to complete the operation. A sufficiently-large frequency of cross-boundary accesses will markedly increase the cycles-per-instruction (CPI) of the processor, and therefore decrease the performance of the machine.

From the foregoing, there is considerable motivation to improve the cross-boundary storage access time. Quantitatively, for a doubleword storage organization requiring one machine cycle to access one doubleword, that time in cycles, t.sub.cb, is ##EQU1## where L is the length of the storage access in bytes and the quantity in brackets is rounded to the next highest integer.

One obvious method of decreasing t.sub.cb would be to prohibit cross-boundary storage accesses. This solution is not within the realm of possibilities for well-established architectures such as S/370 which must maintain compatibility with earlier versions of the architecture.

Another method is to partition the storage array into odd and even n-words, as proposed by the IBM Technical Disclosure Bulletin, Vol. 25 No. 7A, December 1982, A. Y. Ngai and C. H. Ngai proposed "Boundary Crossing with a Cache Line", discussed supra. This solution introduces extra delay in the storage array address path (an address incrementer), which may be unacceptable when the storage array is a high-speed cache. Often, the path encompassing the cache address, cache access and data transfer back to the instruction processing unit (IPU) constitutes the longest or critical path in the machine and thus constrains the machine cycle time, the second factor in the machine performance equation. Also, each array requires its own output bus. This doubles the array output wiring complexity, which may cause both wiring and circuit delay problems.

The subject of our inventions which are illustrated by our preferred embodiments is a hardware accelerator for cross-boundary storage accesses that improves t.sub.cb to ##EQU2## which, for sufficiently large L., approaches half of Eqn. (1). (The quantity in brackets is rounded to the next highest integer.) This improvement is achieved without either requiring storage accesses to be on-boundary or inserting delay in the processor critical path.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

This invention will be described in the context of the S/370 instruction-set architecture. Further, assume the storage array to be organized on doubleword boundaries and that one doubleword may be accessed in one machine cycle. Also, assume the IPU to be a pipelined, microcoded processor, with pipeline stages defined as follows:

Stage	Description
IF	microword fetch
AG	storage address generation
EX	storage array access
PA	storage data put away

Microwords are issued and executed in an overlapped manner. Thus, in terms of pipeline stages, an execution sequence of consecutive microwords would appear as

```
IF    AG         EX     PA
      IF         AG     EX       PA
                 IF     AG       EX   PA
```

and so forth.

It is to be understood that these assumptions by no means limit the applicability of the invention to either S/370 architecture or the foregoing storage and IPU organizations. These are merely set forth for expository purposes.

The IPU and storage interface is shown in FIG. 1. ABUS(0:31) (the address bus) supplies the address from the IPU to storage for a storage access. CBUS(0:6) (the command bus) indicates the kind of storage access, i.e., read from storage or write to storage, the length of the access in bytes, and two other signals pertaining to the invention which will be discussed below. To avoid confusion with the S/370 Load and Store instructions, the terms "read" and "write" will be used for "load from storage" and "store to storage," respectively. DBUSOUT(0:63) (the out-going data bus from the IPU) supplies up to eight bytes of data for write accesses, and DBUSIN(0:63) (the in-coming data bus to the IPU) supplies up to eight bytes for read accesses.

FIG. 2 takes the structures shown in FIG. 1 to the next level of detail, In the IPU, a control store array (CS) contains microwords which direct the operations of the IPU and storage. Microwords are fetched from the CS by the microsequencer (MSEQ) into the microinstruction register (MIR). Fields from the MIR control the activity of the CBUS, the address generation adder (AGEN), the write rotator (WROTATE) and the GR array. Another field, ENDOP, terminates execution of a microcode sequence and initiates decoding of the next instruction.

The AGEN adder performs the usual S/370 address generation, i.e., base GR +index GR+displacement, with the additional capability of incrementing the AGEN output by a specified amount for use in the following cycle. The AGEN output is the ABUS(0:31), which is further partitioned into a doubleword address ABUS(0:28) which is eventually used to address the storage array, and a byte address ABUS(29:31), whose use will be described further on.

The CBUS consists of the following signals:

Signal	Function
LEN(0:2)	zero-origin length (a value of B'000' means one, a value of B'111' means eight) of the storage access in bytes
READ	if asserted, data is to be read from the storage array to the IPU
WRITE	if asserted, data is to be written from the IPU to the storage array
SAVE	(see below)
MERGE	(see below)

The WRITE, SAVE, and MERGE signals are used to control the cross-boundary storage access acceleration mechanism and will be further described below.

Assume for simplicity that the GR array contains 16 registers numbered 0-15 and is organized into odd and even halves. An even-numbered GR may be written from DIREG(0:31) and read into RREG(0:31). An odd-numbered GR may be written from DIREG(32:63) and read into RREG(32:63).

The sequence of events that takes place for write and read accesses of our preferred embodiments illustrated by FIGS. 2 (et seq.) will now be described.

On a write access, the storage address is generated and transmitted to the storage hardware on the ABUS together with the CBUS command in the AG cycle. These are latched by the storage hardware in the AREG and CREG, respectively. Concurrently, up to a doubleword of data may be read from the GR array into the RREG. In the EX cycle, the output of RREG is rotated by WROTATE and transmitted to storage on the data bus DBUSOUT(0:63). WROTATE rotates the data so that the first byte of the data is positioned at the starting byte address of the write. For example, suppose GR1 contained the data abcd and it was desired to write one byte, the byte d, to storage at address 0. First, GR1 would be read into RREG(32:63) in the AG cycle. The contents of RREG(0:31) is unknown; let it be represented by xxxx. Since the byte address ABUS(29:31)=B'000', WROTATE will rotate the output of the RREG such that byte d is at address B'000'. Thus, the output of WROTATE will be dxxxxabc. This inherent capability in the rotator will be further exploited in the cross-boundary accelerator. Note that since the CBUS will indicate a write of length one, the bytes xxxxabc that accompany byte d on DBUSOUT are ignored.

The data transmitted on DBUSOUT may be merged with data in the cross-boundary buffer (CBB) by the write merger (WMERGE) before it is latched in the DBUSOUT register (DOREG). Simultaneously, the data is gated through MUX (since the WRITE signal is asserted) and is latched in the CBB. In the next cycle (PA), the data is written from DOREG into the storage array at the doubleword addressed by EA(0:28), the latched AREG(0:28). WMERGE and the CBB are under control of the write-merge controller (WMCTL) which takes inputs from the CREG. The CBB, WMERGE, and WMCTL hardware, with the rotation capabilities of WROTATE, together with control provided by a microcoded algorithm, constitute the cross-boundary acceleration mechanism when the storage access is a write.

On a read access, in the AG cycle the storage address is generated and transmitted to the storage hardware on the ABUS together with the CBUS command, which are latched in the .AREG and CREG, respectively. In the EX cycle, the AREG(0:28) is used to read a doubleword from the storage array. This doubleword is gated through MUX (since the WRITE signal is not asserted) and latched in the CBB and at the same time is passed through the read merger (RMERGE), where it may be merged with data already in the CBB. RMERGE is under control of the read-merge controller (RMCTL), which responds to control signals in the CREG. The doubleword is then rotated by the read rotator (RROTATE) and transmitted on DBUSIN to the IPU where it is latched in DIREG. The data may then be written to the desired GR(s) during the PA cycle.

The function of RROTATE is entirely analogous to WROTATE. For example, suppose it was desired to read one byte from storage location 0 to the rightmost byte of GR0. Assume doubleword 0 in storage contained abcdefgh. Then, the rotator would produce the doubleword fghabcde based on the byte address EA(29:31). IPU hardware must ensure that only byte a is written to GR0, i.e., other bytes in the data transfer are ignored.

The CBB, RMERGE, and RMCTL hardware, with the rotation capabilities of RROTATE, together with control provided by a microcoded algorithm, constitute the cross-boundary acceleration mechanism when the storage access is a read.

The W2(0:7) register in the IPU is an 8-bit register that initially contains the total zero-origin length of the storage operand to be processed. The actual length sent on the CBUS, i.e., LEN(0:2), is produced by LENCTL and is a derivative of both W2 and the byte address ABUS(29:31) and is controllable by microcode. In particular, assume microcode may specify the following logical lengths (actual length refers to the zero-origin length transmitted on the CBUS):

Logical Length	Actual length
DBDY	LEN= ABUS(29:31)
LW2	If W2(0:4)=0 then LEN=W2(5:7) else LEN=7

The DBDY logical length provides a means to access storage from an arbitrary byte address up to the next doubleword boundary. For example, if ABUS(29:31)=B'001' and LEN=DBDY in the microword, the the actual length is B'001' or B'110', i.e., zero-origin 6, meaning a 7-byte access.

The LW2 logical length allows a storage operand to be accessed in 8-byte quantities up until the last access where the remaining bytes (from 1 to 8 bytes) are accessed.

The usages of these lengths will be illustrated in the examples set forth hereinafter; see Examples 2 and 3.

The logic-level detail of the WMCTL, WMERGE, and CBB for a write access will now be described.

Referring to FIG. 2, at the beginning of a storage write operation the write address is latched in the AREG, and the SAVE, MERGE, WRITE and LEN controls are latched in the CREG. The combinatorial logic contained in WMCTL uses these signals to control the transfer of data through the WMERGE unit to the input register of the storage array (DOREG).

Referring to FIGS. 3, 4, and 5, a write operation begins with WRITE and SAVE active and MERGE inactive. The low order AREG address bits, AREG(29:31) are combined with the LEN bits, LEN(0:2) of the CREG to produce the signal STORE.sub.-- EXCESS.sub.-- BYTE.sub.-- X (0.ltoreq.X.ltoreq.6). The aforementioned combination of SAVE and MERGE is decoded and is used to gate data from the DBUSOUT bus to the DOREG register, the input register of the storage array. Since WRITE is asseded, bytes 0 through 6 of the DBUSOUT bus are gated through MUX and stored in the CBB each cycle.

Note that for all cross-boundary write accesses only DBUSOUT(0:55) needs to be latched in CBB since at least one byte from the data currently on DBUSOUT(0:63) is always written to the storage array in the cycle immediately following the transfer. Thus, 8 latches may be saved by defining CBB to be 7 bytes (56 bits, numbered 0:55) wide.

During subsequent write cycles when WRITE and both SAVE and MERGE are active, selected bytes are transferred from the CBB to the DOREG register. The latched active STORE.sub.-- EXCESS.sub.-- BYTE.sub.-- X, SAVE and MERGE signals are combined to produce W.sub.-- CBB.sub.-- BYTE.sub.-- X.sub.-- SELECT, a signal used to select bytes from the CBB to be stored to the DOREG register. The latched inactive STORE.sub.-- EXCESS.sub.-- BYTE.sub.-- X, SAVE and MERGE signals are combined to produce DBUSOUT.sub.-- BYTE.sub.-- X.sub.-- SELECT, a signal used to select bytes from DBUSOUT to be latched in DOREG.

The actual writing of the data to the storage array is under control of the STORAGE READ/WRITE CTLS unit, shown in FIG. 2. The detailed logic for this function is shown in FIG. 12. Here, the LEN bits from the CREG are inverted and latched to obtain the field length of the write and the starting write address is latched in the WAREG register from the AREG. During the following cycle these controls are used to write the data from the DOREG to the storage array.

The logic-level detail of the RMCTL, RMERGE, and CBB on a read access will now be described.

Referring to FIG. 2, at the beginning of a storage read operation, the read address is latched in the AREG and the SAVE, MERGE, READ and LEN controls are latched in the CREG. The combinatorial logic contained in RMCTL uses these signals to control the transfer of data through the RMERGE unit to the input of RROTATE.

Referring to FIGS. 5, 6, 7, 8, 9, 10 and 11, a read operation begins with SAVE and READ active and WRITE and MERGE inactive. The low order AREG address bits, AREG(29:31) are combined with the LEN bits, LEN(0:2) of the CREG to produce the signal LOAD.sub.-- EXCESS.sub.-- BYTE.sub.-- X (1.ltoreq.X.ltoreq.7). The aforementioned combination of SAVE and MERGE is decoded and is used to gate data from the output of the storage array through RMERGE to the input of RROTATE. Since the

WRITE signal is not asserted, bytes 1 through 7 of the storage array output are gated through MUX and are stored in the CBB each cycle. The storage array which is supplied a starting doubleword address from the AREG reads 8 bytes of data each cycle.

Note that for all cross-boundary read accesses only 7 bytes of the data read from the storage array, STORAGE.sub.-- ARRAY.sub.-- OUTPUT(8:63), are latched in the CBB since at least one byte of data is read from the storage array and transferred to the IPU when the requested number of bytes exceeds the number of valid bytes stored in CBB. Therefore, the CBB need only be 7 bytes (56 bits, numbered 8:63) wide for a read access, thereby allowing a single 7-byte CBB to be shared for read and write accesses. During subsequent read cycles when both SAVE and MERGE are active and WRITE is inactive (READ is active), selected bytes are transferred from the CBB to the RROTATE shifter through RMERGE. The latched active LOAD.sub.-- EXCESS.sub.-- BYTE.sub.-- X, SAVE and MERGE signals are combined to produce R.sub.-- CBB.sub.-- BYTE.sub.-- X.sub.-- SELECT, a signal used to select bytes from the CBB to transfer to the RROTATE shifter. The latched inactive LOAD.sub.-- EXCESS.sub.-- BYTE.sub.-- X, SAVE and MERGE signals are combined to produce STORAGE.sub.-- BYTE.sub.-- X.sub.-- SELECT, a signal used to steer bytes from the output of the storage array to RROTATE. Data passed through RROTATE is aligned according to AREG(29:31) as described previously, and is then transmitted to DIREG on DBUSIN.

EXAMPLES OF OPERATION

Examples illustrating the operation of the IPU and storage systems described will now be considered. The examples employ particular/370 instructions to illustrate the functions previously described and are in no way intended to be limiting.

In the examples, an Instruction Decode and Setup (ID) cycle has been added to each instruction to facilitate making necessary preparations for execution of the instruction, e.g, initializing the W2 register, selecting and fetching the proper microcode algorithm, etc. IF cycles are not shown. Also note that the CBB is defined for bits 0:63. Previously, it was shown that the CBB need only be 7 bytes wide, which is indeed the case. However, defining the eighth byte allows the explanation to be given in terms of doublewords, which is more readily understood.

CROSS-BOUNDARY READS

Consider a Load Multiple instruction with data organized in storage as shown in the following Example 2A.

EXAMPLE 2A ##STR3##

Assume that separate microcode algorithms are defined for LM with length.ltoreq.8 bytes and LM>8 bytes. Which algorithm to choose is determined in the ID cycle by logic which examines the LM instruction text to determine the length of the LM storage operand. The former case requires a single microword; the latter case requires two, with iterations on the second word until all GRs are loaded. In both cases, W2 is initialized to the total number of bytes to be loaded (zero origin).

The timing diagram in FIG. 13 illustrates the CBB operation for the instruction. Microword 1 for LM specifies ENDOP=1, READ=1, SAVE=1, MERGE=0 and LEN=LW2. The value shown for LEN in the figure is that to which the logical length LW2 resolves and is the value transmitted on the CBUS. MSEQ detects that this coding together with ABUS(29:31).noteq.0 implies a cross-boundary read. As a result, MSEQ automatically repeats microword 1, increments ABUS by 8 and forces MERGE to 1. This is the first of two cases where hardware overrides the merge control specified in the microword: a read with SAVE=1 is specified and the length of the storage operand is such that a cross-boundary access is required.

The doubleword at address 0 is fetched from the storage array in cycle 3 and is saved in the CBB at the end of the cycle. The excess bytes, i.e., the bytes not returned to the IPU, namely bytes abcd, are so marked in the CBB. The second read request, issued in cycle 3, causes storage to access the doubleword at 8. Since MERGE was asserted for this request, the excess bytes in the CBB are merged with required bytes from the second doubleword and the rotated result is forwarded on the DBUSIN to the IPU in cycle 4 and latched in DIREG from where GR0 and GR1 may then be written in cycle 5. ENDOP causes termination of the microcode sequence.

Consider a second example of LM, Example 2B, as follows:

EXAMPLE 2B ##STR4## The timing is illustrated in FIG. 14. Microword 1 for LM is coded ENDOP=0, READ=1, SAVE=1, MERGE=0 and LEN=LW2. A mode control bit in MSEQ is also set by microword 1 to disable subsequent ENDOP signals until the operand is fully accessed. Once again, MSEQ detects that a cross-boundary read is occurring and thus automatically reissues microword 1, incrementing ABUS by 8 and forcing MERGE to 1. In response to the first read, RMCTL saves the doubleword at 0 in the CBB and flags the excess bytes. On the read of the doubleword at 8, the excess bytes from the CBB are merged with sufficient bytes from the doubleword at 8 to satisfy the length requested. The result is rotated and forwarded on DBUSIN to the IPU. The doubleword at 8 is saved in the CBB, with the excess bytes flagged.

A second microword is required to read the remaining bytes. Microword 2 for LM is coded ENDOP=1, READ=1, SAVE=1, MERGE=1 and LEN=LW2. Since MERGE is explicitly asserted by the microword, MSEQ does not reissue the microword; the doubleword currently fetched from storage is instead merged with the excess bytes in the CBB. This is illustrated in cycle 5, where byte p from doubleword X'10' is merged with excess bytes ijklmno from the CBB.

The storage operand length could be such that the last read requires no storage array access, i.e., all bytes required are already in the CBB. Consider a LM 0,2,1(0), operating on the same data as before. The timing is illustrated in FIG. 15. The sequencing is similar to that for LM 0,3,1(0), except note that in cycle 4, a 4-byte read is requested. Although the ABUS points to the doubleword at X'10', RMCTL recognizes that the read length requested is less than or equal to the number of excess bytes in the CBB. Therefore, no storage array access is required; the required bytes are simply unloaded from the CBB, rotated and delivered to IPU on DBUSIN.

Notice in cycle 5 that the data for GR2 is on DBUSIN(0:31), and unwanted data is on DBUSIN(32:63). If it is assumed that simple incrementers are used to address the GR pairs to be loaded from DIREG each cycle, then GR3 would be scheduled to be loaded from DIREG(32:63) in cycle 6. This can not be permitted in this case since the LM instruction only loads GRs 0, 1, and 2.

Hardware is provided to prevent this, and consists of simply determining if the LM instruction loads an odd number of GRs, and, if so, blocking the load of the odd GR during the PA cycle of the last LM (this is indicated when W2 is decremented through zero).

Once the operand is fully accessed, as determined by the contents of W2, ENDOP is enabled, causing termination of the microcode sequence. The mode control bit set in microword 1 to disable ENDOP is reset automatically.

Finally, consider an on-boundary LM as shown by Example 2C.

EXAMPLE 2C ##STR5##

The timing is illustrated in FIG. 16. Note that although the CBB is loaded in both accesses, each time the number of bytes requested (LEN) could be completely fulfilled with the storage array access alone, i.e., no bytes are flagged as excess in the CBB. When this condition is true, RMCTL ignores the merge imperative.

The described mechanism is general in that it can be utilized in all instructions which read from storage and may cross a doubleword boundary on the access. In particular,/370 SS-format instructions may utilize the mechanism to read operand 2 in a manner which is both high performance (avoids all but one cross-boundary stall) and satisfies architectural block concurrency requirements. Accesses to operand 1 (which is first read from then written to) are coded without SAVE or MERGE specified so as to avoid corrupting the operand 2 data in the CBB with operand 1 data. Typically, operand 1 accesses can be doubleword aligned anyway, after an initial access using LEN=DBDY.

CROSS-BOUNDARY WRITES

Consider a Store Multiple instruction with data to be written in storage as shown in Example 3A.

EXAMPLE 3A ##STR6##

Assume that separate microcode algorithms are defined for STM with length .ltoreq.8 bytes and STM>8 bytes. Which algorithm to choose is determined in the ID cycle by logic which examines the STM instruction text to determine the length of the STM storage operand. The former case requires a single microword; the latter case requires two, with iterations on the second word until all GRs are stored. In both cases, W2 is initialized to the total number of bytes to be stored (zero origin).

The timing diagram in FIG. 17 illustrates the CBB operation for the instruction. Microword 1 for this STM is coded ENDOP=1, WRITE=1, SAVE=1, MERGE=0 and LEN=LW2. The length resolves to 3(4 bytes) and ABUS(28:31)=6, implying a cross-boundary write. Detecting this, MSEQ automatically repeats word 1, incrementing ABUS by 8 and forcing MERGE to 1. This is the second of two cases where hardware overrides the merge control specified in the microword: a write with SAVE is specified and the length of the storage operand is such that a cross-boundary access is required.

The contents of GR0 are rotated and transferred to L1 on the DBUSOUT in cycle 3. Bytes ab are stored in the DOREG from where they may be written to the storage array. Simultaneously, the entire doubleword is stored in the CBB with excess bytes cd flagged. Since microword 1 is repeated in cycle 3, the storage command is reissued, ABUS is incremented by 8, and the rotated contents of GR0 are on the DBUSOUT again in cycle 4. WMCTL may then select the remaining bytes to be stored from either DBUSOUT or CBB and form the next storage array entry.

This mechanism may be applied to all storage write accesses of length.ltoreq.8 bytes.

A STM of more than 8 bytes uses the CBB differently. Consider the following STM instruction with data to be stored as shown in Example 3B.

EXAMPLE 3B ##STR7##

The timing for the instruction is illustrated in FIG. 18. Microword 1 is coded ENDOP=0, WRITE=1, SAVE=1, MERGE=0 and LEN=DBDY and is therefore an on-boundary access. Microword 2 is coded ENDOP=1, WRITE=1, SAVE=1, MERGE=1, and.sub.-- LEN=LW2. Word 2 loops on itself until all necessary GRs are stored. For both words, ABUS is incremented by 8.

In cycle 2, the 4-byte write up to the doubleword boundary is issued. The rotated GR0 and GR1 are transferred to storage on DBUSOUT in cycle 3. The entire transfer is saved in the CBB, with bytes efgh marked as excess. Bytes abcd form the first storage array entry. Also in cycle 3, microword 2 has issued an 8-byte write with SAVE=1 and MERGE=1, and the rotated GRs 2 and 3 are then transferred to storage on DBUSOUT. Again, WMCTL places the doubleword in the CBB, marking bytes mnop as excess. Since MERGE=1, WMCTL forms the next storage array entry by merging excess bytes from the previous transfer with bytes ijkl from the current DBUSOUT transfer.

Since W2 indicates that 4 bytes remain to be written, word 2 is issued a second time in cycle 4. When storage receives the write command, it will detect that the length of the write is less than or equal to the number of excess bytes flagged in the CBB and will therefore form the final storage array entry entirely from the CBB excess bytes. Thus, the contents of DBUSOUT in cycle 5 is disregarded. ENDOP is handled in the same manner as in Example 2B.

Consider another STM instruction with data to be stored as shown in Example 3C.

EXAMPLE 3C ##STR8##

The timing for the instruction is illustrated in FIG. 19. The main difference between this and the previous STM occurs in cycles 4 and 5. In cycle 4, an 8-byte write with SAVE=1 and MERGE=1 with GR4 rotated and transferred on DBUSOUT in cycle 5. WMCTL will determine that since there are 4 excess bytes in the CBB, but a write of 8 bytes has been requested, it must look to DBUSOUT for the remaining 4 bytes, i.e., qrst. These are merged with the excess bytes (mnop) from the CBB,and a final storage array entry is formed.

Finally, consider an on-boundary STM as shown in Example 3D.

EXAMPLE 3D ##STR9##

The timing for the instruction is illustrated in FIG. 20. In cycle 2, an 8-byte write command with SAVE=1 and MERGE=0 is issued, and in cycle 3, the doubleword is transferred to storage. Since the length of the write completely fills a doubleword, no bytes are flagged as excess when the data is written into the CBB in cycle 4. Concurrently, the data is loaded into DOREG. forming the storage array entry. The final 4 bytes of data are transferred in cycle 4. Since no excess bytes are flagged in the CBB, no merge is performed, even though the command in cycle 3 indicates MERGE=1. A final storage array entry is formed from the last data transfer.

The mechanism described may be applied to a variety of/370 instructions requiring multiple contiguous writes, e.g., STM, Store Access Multiple (STAM), Branch and Stack (BAKR), etc.

ALTERNATIVES TO PREFERRED EMBODIMENTS

The partitioning of the GR array into even and odd halves was done for expository purposes only. The invention may be readily applied to an implementation using a single, non-partitioned storage array with its general registers, as that implementation, with additional circuitry, can be created by those skilled in the art without difficulty (the circuity not being deemed germane to our invention). Thus, an arbitrary GR organization may be supported.

While we have described our preferred embodiments of our inventions it will be understood that those skilled in the art, both now and in the future, upon the understanding of these discussions will make various improvements and enhancements thereto which fall within the scope of the claims which follow. These claims should be construed to maintain the proper protection for the inventions first disclosed.

* * * * *

United States Patent *5,355,460*
Eickemeyer , et al. **October 11, 1994**

In-memory preprocessor for compounding a sequence of instructions for parallel computer system execution

Abstract

A digital computer system capable of processing two or more computer instructions in parallel and having a main memory unit for storing information blocks including the computer instructions includes an instruction compounding unit for analyzing the instructions and adding to each instruction a tag field which indicates whether or not that instruction may be processed in parallel with another neighboring instruction. Tagged instructions are stored in the main memory. The computer system further includes a plurality of functional instruction processing units which operate in parallel with one another. The instructions supplied to the functional units are obtained from the memory by way of a cache storage unit. At instruction issue time, the tag fields of the instructions are examined and those tagged for parallel processing are sent to different ones of the functional units in accordance with the codings of their operation code fields.

Inventors:	**Eickemeyer; Richard J.** (Endicott, NY); **Vassiliadis; Stamatis** (Vestal, NY); **Blaner; Bartholomew** (Newark Valley, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**098240**
Filed:	**July 29, 1993**

Current U.S. Class:	**712/215**; 712/216
Intern'l Class:	G06F 009/40
Field of Search:	395/DIG. 1,DIG. 2,200,250,275,375,400,600,425,550,658,700,800

References Cited [Referenced By]

U.S. Patent Documents

3401376	Sep., 1968	Barnes et al.	364/DIG.
4025771	May., 1977	Lynch, Jr. et al.	364/736.
4295193	Oct., 1981	Pomerene	364/DIG.
4439828	Mar., 1984	Martin	364/200.
4594655	Jun., 1986	Hao et al.	364/DIG.
4847755	Jul., 1989	Morrison et al.	395/650.
5021945	Jun., 1991	Morrison et al.	395/375.

Other References

Acosta, R. D., et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al., the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al.,) ed (McGraw-Hill, 1982,) pp. 276-292.
Capozzi, A. J., et al., "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al., "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al., "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M. D., et al., "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al. ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al., "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N. P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp. 63-76.

Colwell, R. P., et al., "A VLIW Architecture for a Trace Scheduling Complier", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the Crisp Instruction Set Architecture", Proceedings of Compcon, Spring, 1987, pp. 86-89.
Bandoyopadhyay, S., et al., "Compiling for the CRISP Microprocessor", Proceedings of Compcon, Spring, 1987, pp. 96-100.
Hennessy, J., et al., "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al., "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W., et al., "Checkpoint Repair for High-Performance Out-of -Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al., "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1, Jan. 1984, pp. 6-22.
Riesman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecuture, May 1981, pp. 135-148.
Archibold, James, et al., Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al., "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E. et al., "A Study of Instruction Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun. 1983, pp. 132-137.
Vassiliadis, S., et al., "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard.

Primary Examiner: Harrell; Robert B.
Attorney, Agent or Firm: Augspurger; Lynn L., Meador; Terrance A.

Parent Case Text

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation of application Ser. No. 07/543,464, filed Jun. 26, 1990, now abandoned.

The present United States patent application is related to the following co-pending United States patent applications:

(1) application Ser. No.: U.S. Ser. No. 07/519,384 filed May 4, 1990, and now abandoned in favor of a continuation application U.S. Ser. No. 08/013,982 filed Feb. 5, 1993 entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al;

(2) application Ser. No.: U.S. Ser. No. 07/519,382 filed May 4, 1990, and now abandoned in favor of a continuation application U.S. Ser. No. 08/015,272 filed Feb. 5, 1993 entitled "General Purpose Compound Apparatus For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al;

(3) U.S. Pat. No. 5,051,940, granted Sep. 24, 1991 entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al; and

(4) application Ser. No.: U.S. Ser. No. 07/522,291 now U.S. Pat. No. 5,214,763 granted May 25, 1993 and continued as U.S. Ser No. 08/001,479 filed Jan. 7, 1993, entitled "Compounding Preprocessor For Cache", the inventors being Bartholomew Blaner et al.

These co-pending applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y.

The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

Claims

We claim:

1. In a digital computer system including a means for executing a plurality of instructions having a form said instructions have when presented for scalar execution in parallel, a combination, comprising:

an in-memory preprocessor for a sequence of instructions which may be executed by a plurality of functional units of the digital computer system, each functional unit being capable of processing one or more types of machine-level instructions, said in-memory preprocessor providing for compounding of instructions for execution in parallel by one or more functional units of the digital computer system in parallel prior to issue and execution of a sequence of instructions to be executed by the functional units, including in combination,

(a) an input/output interface for providing a group of instructions to be processed,

(b) an instruction compounding mechanism for receiving the group of instructions to be processed from said input/output interface and for producing compounding tag information for an instruction which denotes a grouping of instructions for parallel execution for each one instruction of the group of instructions, the compounding tag information indicating whether an instruction associated with a produced compounding tag information may be compounded and executed in parallel in said digital computer system by a value of said compounding tag information for each one instruction of the group of instructions; and

(c) a main storage connected to the input/output interface and to the instruction compounding mechanism for storing the group of instructions with the compounding tag information.

2. The combination of claim 1, wherein compounding tag information is comprised of a plurality of tag fields, each tag field being associated with a respective instruction analyzed by the instruction compounding mechanism.

3. The combination of claim 2, wherein the instruction compounding mechanism includes:

a plural-instruction instruction register for receiving a plurality of successive instructions;

a plurality of rule-based instruction analyzer mechanisms, each rule-based instruction analyzer mechanism analyzing a particular pair of side-by-side instructions in the instruction register and producing a compoundability signal which indicates whether or not the two instructions in said pair may be processed in parallel; and

a tag generating mechanism responsive to the compoundability signals for generating the individual tag fields for the different instructions in the instruction register.

4. The combination of claim 3, wherein the computer system has a particular instruction processing configuration, and each instruction analyzer mechanism includes logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution in the particular instruction processing configuration used for the computer system, the logic circuitry producing the compoundability signal for that analyzer mechanism.

5. The combination of claim 1, wherein the digital computer system includes a plurality of said functional units which operate in parallel with one another, the compounding tag information being used in issuing instructions to different ones of the functional units concurrently when a first of the instructions issuing concurrently has compounding tag information associated with the first of the instructions which specifies that the first of the instructions may be executed in parallel with a next instruction and the next instruction has compounding tag information associated with the next instruction which indicates that the next instruction may be executed concurrently with the first of the instructions issuing concurrently.

6. The combination of claim 1 wherein the compounding tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a subsequent instruction of the sequence of instructions, or, specifies, if the one instruction may not be executed in parallel with a preceding instruction of the sequence of instructions, that the one instruction must be executed as a single instruction.

7. The combination of claim 1 wherein the compounding tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a next one instruction of the sequence of instructions, the number of subsequent instructions with which the one instruction can be executed, or which specifies, if the one instruction may not be executed in parallel with a preceding instruction of the sequence of instructions, that the one instruction must be executed as a single instruction.

8. In a digital computer system capable of processing two or more instructions having a form said instructions have when presented for scalar execution in parallel, a combination, comprising:

an in-memory preprocessor for a sequence of instructions which may be executed by a plurality of functional units of the digital computer system, each functional unit being capable of processing one or more types of machine-level instructions, said in-memory preprocessor providing for compounding of instructions for execution in parallel by one or more functional units of the digital computer system in parallel prior to issue and execution of a sequence of instructions to be executed by the plurality of functional units, including in combination,

(a) means for receiving a group of instructions in a sequence of instructions to be processed;

(b) an instruction compounding mechanism connected to said means for receiving the group of instructions and associating with the group of instructions compounding tag fields which denote a grouping of instructions for parallel execution and indicate which instructions of the group of instructions taken from a sequence of existing machine instructions as a sequence of instructions may be compounded and executed in parallel in said digital computer system; and

(c) a storage mechanism coupled to the instruction compounding mechanism for receiving and storing the group of instructions and the associated compounding tag fields.

9. The combination of claim 8, further including instruction execution means connected to the storage mechanism for:

fetching a plurality of instructions; and

in response to the compounding tag fields, executing a plurality of instructions in parallel.

10. The combination of claim 9, wherein the instruction execution means operates asynchronously with the instruction compounding mechanism.

11. The combination of claim 8, further including:

a plurality of functional instruction processing units which operate in parallel with one another; and

an instruction issue mechanism coupled to the storage mechanism for supplying instructions stored therein to different ones of the functional instruction processing units when their compounding tag fields indicate that they may be processed in parallel.

12. The combination of claim 11, wherein the storage mechanism includes a main memory for storing a block of information including said instructions and a cache storage mechanism connected to the main memory and to the instruction issue mechanism.

13. The combination of claim 8, wherein the storage mechanism includes a main memory having a word size sufficient to append compounding tag fields to the instructions of the group of instructions.

14. The combination of claim 8, wherein the storage mechanism includes a main memory for storage of the group of instructions and a tag memory for storage of the compounding tags fields.

15. The combination of claim 8, wherein the storage mechanism includes a main memory, the main memory including a tag table for the compounding tag fields and a separate section for storage of a plurality of groups of instructions.

16. The combination of claim 8, wherein the storage mechanism includes a main memory, the main memory including a page section having a first section for storage of the group of instructions and a second section for storage of the compounding tag fields.

17. The combination of claim 8, wherein the group of instructions is a page of instructions.

18. The combination of claim 8 wherein the compounding tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a subsequent instruction of the sequence of instructions, or, specifies, if the one instruction may not be executed in parallel with a preceding instruction of the sequence of instructions, that the one instruction must be executed as a single instruction.

19. The combination of claim 8 wherein the compounding tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a next one instruction of the sequence of instructions, the number of subsequent instructions with which the one instruction can be executed, or which specifies, if the one instruction may not be executed in parallel with a preceding instruction of the sequence of instructions, that the one instruction must be executed as a single instruction.

20. In a digital computer system capable of concurrently executing a plurality of instructions having a form said instructions have when presented for scalar execution, a combination including:

an in-memory preprocessor for a sequence of instructions which may be executed by a plurality of functional units of the digital computer system, each functional unit being capable of processing one or more types of machine-level instructions, said in-memory preprocessor providing for compounding of instructions for execution in parallel by one or more functional units of the digital computer system in parallel prior to issue and execution of a sequence of instructions to be executed by the plurality of functional units, including in combination,

(a) a memory mechanism for storing a predetermined group of instructions to be executed;

(b) an instruction compounding unit connected to the memory mechanism for generating compounding tag information which denotes a grouping of instructions for parallel execution and identifies instructions in the predetermined group of instructions taken from a sequence of existing machine instructions as a sequence of instructions that may be compounded and executed in parallel in said digital computer system which are to be concurrently executed;

(c) compounding tag information storage means in the memory mechanism and connected to the instruction compounding unit for storing the compounding tag information; and

(d) instruction execution means connected to the memory mechanism for executing instructions singly or concurrently in response to the compounding tag information.

21. The combination of claim 20, wherein the instruction compounding unit is asynchronous with the instruction execution means.

22. The combination of claim 20, wherein the compounding tag information storage means is a tag memory.

23. The combination of claim 20, wherein the compounding tag information storage means is a tag table.

24. The combination of claim 20, wherein the compounding tag information storage means includes memory mechanism space appended to the predetermined group of instructions.

25. The combination of claim 20 wherein the compounding tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a subsequent instruction of the sequence of instructions, or, specifies, if the one instruction may not be executed in parallel with a preceding instruction of the sequence of instructions, that the one instruction must be executed as a single instruction.

26. The combination of claim 20 wherein the compounding tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a next one instruction of the sequence of instructions, the number of subsequent instructions with which the one instruction can be executed, or which specifies, if the one instruction may not be executed in parallel with a preceding instruction of the sequence of instructions, that the one instruction must be executed as a single instruction.

Description

TECHNICAL FIELD

This invention relates to digital computers and digital data processors, and particularly to digital computers and data processors capable of executing two or more instructions in parallel.

BACKGROUND OF THE INVENTION

Traditional computers which receive a sequence of instructions and execute the sequence one instruction at a time are known. The instructions executed by these computers operate on single-valued objects, hence the name "scalar".

The operational speed of traditional scalar computers has been pushed to its limits by advances in circuit technology, computer mechanisms, and computer architecture. However, with each new generation of competing machines, new acceleration mechanisms must be discovered for traditional scalar machines.

A recent mechanism for accelerating the computational speed of uni-processors is found in reduced instruction set architecture that employs a limited set of very simple instructions. Another acceleration mechanism is complex instruction set architecture which is based upon a minimal set of complex multi-operand instructions. Application of either of these approaches to an existing scalar computer would require a fundamental alteration of the instruction set and architecture of the machine. Such a far-reaching transformation is fraught with expense, downtime, and an initial reduction in the machine's reliability and availability.

In an effort to apply to scalar machines some of the benefits realized with instruction set reduction, so-called "superscalar" computers have been developed. These machines are essentially scalar machines whose performance is increased by adapting them to execute more than one instruction at a time from an instruction stream including a sequence of single scalar instructions. These machines typically decide at instruction execution time whether two or more instructions in a sequence of scalar instructions may be executed in parallel. The decision is based upon the operation codes (OP codes) of the instructions and on data dependencies which may exist between instructions. An OP code signifies the computational hardware required for an instruction. In general, it is not possible to concurrently execute two or more instructions which utilize the same hardware (a hardware dependency) or the same operand (a data dependency). These hardware and data dependencies prevent the parallel execution of some instruction combinations. In these cases, the affected instructions are executed serially. This, of course, reduces the performance of a super scalar machine.

Superscalar computers suffer from disadvantages which it is desirable to minimize. A concrete amount of time is consumed in deciding at instruction execution time which instructions can be executed in parallel. This time cannot be readily masked by overlapping with other machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Also, the parallel execution decision must be repeated each time the same instructions are to be executed.

In extending the useful lifetime of existing scalar computers, every means of accelerating execution is vital. However, acceleration by means of reduced instruction set architecture, complex instruction set architecture, or superscalar techniques is potentially too costly or too disadvantageous to consider for an existing scalar machine. It would be preferred to accelerate the speed of execution of such a computer by parallel, or concurrent, execution of instructions in an existing instruction set without requiring change of the instruction set, change of machine architecture, or extension of the time required for instruction execution.

SUMMARY OF THE INVENTION

In co-pending patent application Ser. No. U.S. Ser. No. 07/519,384 a scalable compound instruction set machine (SCISM) architecture is proposed in which instruction level parallelism is achieved by statically analyzing a sequence of scalar instruction at a time prior to instruction execution to generate compound instructions formed by adjacent grouping of existing instructions in the sequence which are capable of parallel execution. Relevant control information in the form of tags is added to the instruction stream to indicate where a compound instruction starts, as well as to indicate the number of existing instructions which are incorporated into a compound instruction. Relatedly, when used herein, the term "compounding" refers to the grouping of instructions contained in a sequence of instructions, the grouping being for the purpose of concurrent or parallel execution of the grouped instructions. At minimum, compounding is satisfied by "pairing" of two instructions for simultaneous execution. Preferably, compounded instructions are unaltered from the forms they have when presented for scalar execution. As explained below, compounded instructions are accompanied by compounding tag information, that is, bits appended to the grouped instructions which denote the grouping of the instructions for parallel execution.

In a digital computer system which includes a means for executing a plurality of instructions in parallel, a particularly advantageous embodiment of the invention is based upon a memory architecture which provides for compounding of instruction prior to their issue and execution. Such a memory is a component of a hierarchical memory structure which provides instructions to the CPU (central processing unit) of a computer. Typically, such a structure includes a high-speed cache storage containing frequently accessed instructions, a lower speed main memory or primary storage connected to the cache, and a low-speed, high-capacity auxiliary storage. Typically, the cache and main storage contain instructions which can be directly referenced for execution. Access to instructions in the auxiliary storage is had through an input/output (I/O) adaptor connected between the main memory and the auxiliary storage.

In a scalar computer having a hierarchical storage organization, the invention resides in a combination including an input/output interface for providing, from secondary storage, a sequence of instructions for execution, an instruction compounding mechanism which produces compounding tag information in response to the instruction sequence, the compound tag information indicating instructions of the sequence which may be executed in parallel, and a main storage connected to the input/output interface and to the instruction compounding mechanism for storing the sequence of instructions with the compound tag information.

As is known, main memory provides residence for data and instructions which are immediately accessible to a CPU for reference for execution. The use of the main memory in a well-designed hierarchical storage system in and of itself serves to improve the overall performance of a scalar computer. In the invention, the storing of the compounding tag information in the main memory enables the information to be used over and over so long as the instructions remain in the main memory. Furthermore, instructions in main memory, once passed to a cache, frequently remain in a cache long enough to be used more than once.

For a better understanding of the invention, together with its advantages and features, reference is made to the following description and the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a representative embodiment of a portion of a digital computer system constructed in accordance with present invention,

FIGS. 2A, 2B, 2C and 2D illustrate alternative implementations for storage of compounding tag information in main memory.

FIG. 3 shows in greater detail the data flow structure between an IO adaptor and the main memory in the computer system of FIG. 1.

FIG. 4 is a timing diagram for instruction transfer in the data flow structure of FIG. 3.

FIG. 5a shows a length of an instruction stream having compounding tags or tag fields associated with the instructions;

FIG. 5b shows a length of an instruction stream having instruction boundary fields associated with the instructions;

FIG. 6 shows in greater detail the internal construction of a representative embodiment of an instruction compounding unit which can be used in the computer system of FIG. 1;

FIG. 7 shows in greater detail a representative internal construction for each of the compound analyzer units of FIG. 3;

FIG. 8 shows an example of logic circuitry that may be used to implement the compound analyzer and tag generator portions of FIG. 6 which produce the compounding tags for the first three instructions in the instruction stream;

FIG. 9 is a table used in explaining the operation of the FIG. 8 example;

FIG. 10 shows a representative embodiment of a portion of a digital computer system and is used to explain how the compounded instructions may be processed in parallel by multiple functional instructions processing units;

FIG. 11 shows an example of a particular sequence of instructions which may be processed by the computer system of FIG. 10; and

FIG. 12 is a table used in explaining the processing of the FIG. 11 instruction sequence by the computer system of FIG. 10.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring to FIG. 1 of the drawings, there is shown a representative embodiment of a portion of a digital computer system or digital data processing system constructed in accordance with this invention. The illustrated computer system is capable of executing two or more instructions in parallel. It includes a hierarchically-arranged storage system in which auxiliary or secondary storage devices are connected via an I/O bus to a computer. The computer interfaces with the I/O bus through an adaptor which is also connected to a memory bus. The main memory and a high-speed cache are connected to the memory bus.

This hierarchy typically permits the computational components of the computer system to directly access or refer to the contents of the main memory and, the cache, while the adaptor provides access to the auxiliary storage. Instructions and data which must be accessed or referenced to support current computer operations are kept in the memory. When no longer required, these are returned to the auxiliary memory by way of the adaptor, while new instructions and data are entered into the main memory. The cache supports high-speed access by the CPU and is used to store instructions and data which are currently being used or are highly likely to be used next by the CPU. Hierarchical storage structures are explained in detail in Chapter 7 of Deitel's OPERATING SYSTEMS, second edition, 1990.

Referring now to FIG. 1 of the drawing, a representative embodiment of a portion of a digital computer system having a hierarchically-arranged memory structure is shown in accordance with the present invention. This computer system is capable of processing two or more instructions in parallel. It includes a first storage mechanism for storing instructions and data to be processed. The storage mechanism is identified as a main memory 10. The main memory 10 is connected to a memory bus 9 having an address and command bus 9a and a text bus 9b. The main memory 10 exchanges instructions and data over the memory bus with an IO adaptor 8. The IO adaptor 8 is connected to the memory bus 9 and to an IO bus 7. It is asserted that one or more auxiliary storage devices (not shown) are coupled to the IO bus 7. The adaptor 8 transfers data over the IO bus 7 by storing program information to and obtaining program information from the auxiliary storage devices. The adaptor 8 also exchanges program data on the memory bus 9 with the main memory 10 by providing instructions and data to and receiving instructions and data from the main memory over the bus 9. The adaptor 8 buffers instructions and data between the buses 7 and 9, which may have differing speeds and formats. Last, the adaptor 8 also includes checking and error functions. An IO adaptor corresponding to the component indicated by reference number 8 is found in, for example, the channelized IO subsystem of the model 3090 computer system, available from IBM Corporation, the assignee of this patent application.

The main memory 10 is a relatively large capacity, medium-speed storage mechanism which is connected by way of the memory bus 9 to a lower capacity, higher-speed cache. This cache is identified as a compound instruction cache 12.

The computer system in FIG. 1 also includes an instruction compounding mechanism 11 for receiving instructions from the adaptor 8 and associating with those instructions compound tag information in the form of tag fields indicating which of these instructions may be processed in parallel. This instruction compounding mechanism is represented by instruction compounding unit 11. The compounding unit 11 analyzes the incoming instructions for determining which ones may be processed in parallel. Furthermore, the instruction compounding unit 11 produces for those analyzed instructions compounding tag information in the form of tag fields which indicate which instructions may be processed in parallel with one another and which ones may not be processed in parallel with one another.

In FIG. 1, instructions are provided to the computing system from an auxiliary storage device by way of the adaptor 8, the instruction compounding unit 11, and the main memory 10. The main memory 10 receives and stores the analyzed instructions and their associated tag fields. The main memory 10 then provides the analyzed instructions and their associated tag fields to the compound instruction cache 12. The cache 12 has a smaller capacity and higher speed than the main memory 10 and is of the kind commonly used for improving performance rate of a computer system by reducing the frequency of the access to the main memory 10.

The computer system of FIG. 1 also includes a plurality of functional instruction processing units. These functional instruction processing units are represented by functional units 13, 14, 15 and so on. These functional units 13-15 operate in parallel with one another in a concurrent manner and each, on its own, is capable of processing one or more type of machine-level instructions. Examples of functional units which may be used include a general purpose arithmetic and logic unit (ALU) and address generation type ALU, a data dependency collapsing ALU of the type taught in U.S. Pat. No. 5,051,940, granted Sep. 24, 1991, a branch instruction processing unit, a data shifting unit, a floating-point processing unit, and so on. A given computer system may include two or more of some of these types of functional units. For example, a given computer system may include two or more general purpose ALU's. Also, a given computer system may include each and every one of these different types of functional units. The particular configuration of functional units will depend on the nature of the particular computer system being considered.

The computer system of FIG. 1 also includes an instruction fetch and issue mechanism coupled to the compound instruction cache 12 for supplying adjacent instructions stored therein to different ones of the functional instruction processing units 13-15 when the instruction tag fields indicate that they may be processed in parallel. This mechanism is represented by instruction fetch and issue unit 16. Instruction fetch and issue unit 16 fetches instructions from cache 12, examines their tag fields and operation code (OP code) fields and, based upon such examinations, sends the instructions to the appropriate ones of the functional units 13-15. If a desired instruction is resident in the compound instruction cache 12, the appropriate address is sent to the cache 12 to fetch therefrom the desired instruction. This is sometimes referred to as a "cache hit". If the requested instruction does not reside in the cache 12, then it must be fetched from the main memory 10 and brought in to the cache 12. This is sometimes referred to as a "cache miss". When a miss occurs, the address of the requested instruction is sent to the main memory 10. In response thereto, the main memory 10 commences the transfer out or read out of a line of instructions which includes the requested instruction, together with the tag fields of the instructions in the line.

A cache miss causes reference to be made to the main memory 10 to determine whether the requested instruction is contained in the memory 10. In this regard, instructions are commonly stored in the main memory in blocks called "pages" and the memory management facility (not shown) of the computing system is able to determine from the requested instruction whether the page which contains it is in the main memory. If the page is in the main memory, the line containing the instruction is transferred out or read out of the main memory 10 into the cache 12. However, if the page containing the requested instruction is not in main memory 10, a "page fault" occurs requiring the missing page to be "fetched" from auxiliary storage and placed into the main memory 10. When a page is fetched, the identification of the missing page is sent to the adaptor 8, which retrieves it and then provides it, over the memory bus 9, for storage into the main memory 10.

In the invention, pages which are fetched for storage in the main memory 10 are transferred to the input of the instruction compounding unit 11, which unit proceeds to analyze these incoming instructions and generate the appropriate tag field for each instruction. The tags and instructions are thereafter applied to the main memory 10 and stored therein for subsequent placement, if needed, in the compound instruction cache 12.

Although the instruction compounding unit 11 is illustrated in FIG. 1 as being connected between the adaptor 8 and the main memory 10, it is contemplated that the unit may be a separate drop on the memory bus 9 or connected at the input to the main memory 10.

Storage of compounded instructions in the main memory 10 can be implemented in a number of ways, some of which are illustrated in FIGS. 2A-2D. The examples in FIGS. 2A-2D assume an 8-byte wide text bus 9b plus extra lines for the tag information. In general, it is assumed that the basic memory transfer between the main memory 10 and the compound instruction cache 12 involves a 64-byte cache line, with one tag bit for each two bytes of instruction text. One cache line is shown in each of the examples of FIGS. 2A-2D. In general, the number of tag bits is determined by the maximum number of instructions to be compounded and the information available to the instruction compounding unit 11. These considerations are covered in co-pending application Ser. Nos. 07/519,382 filed May 4, 1990, and now abandoned in favor of a continuation application U.S. Ser. No 08/015,272 filed Feb. 5, 1993; and U.S. Pat. No. 5,051,940, granted Sep. 24, 1991.

The simplest tag storage implementation from a control point of view is illustrated in FIG. 2A. If it is assumed that compounding is limited to two instructions, a minimum of a one-bit tag for each two bytes of instruction text is required. Thus, for the line stored in the memory of FIG. 2A, every 64 bits (that is every eight bytes) requires four bits of compounding tag information. As illustrated in FIG. 2A storage of this information involves extension of the word size from 64 to 68 bits. Other optional tag bits would increase the size of the extended words.

A second approach, more compatible with available memory technology is illustrated in FIG. 2B. In FIG. 2B, separate text and tag memories are provided by storage of instructions and associated compounding tag information. In FIG. 2B the tag memory operates in parallel with the text memory. Implicit in the memory structure of FIG. 2B is the requirement for an extra set of tag lines forming a tag bus on the memory bus 9 to provide parallel operation of the text and tag memories. This has several advantages over the extended word approach in FIG. 2A. First, the tag memory may cover only part of the words in main memory. The operating system uses certain parts of memory only for data pages (as opposed to instruction pages), tags are not necessary over these parts. Distinction between data and instruction pages can be a hardware decision, or one made in software and implemented by commands to the tag memory which indicate that certain pages contain data only and therefore do not require the memory page address to be mapped into the tag memory address for these pages. The second advantage is that the tag memory can be removed at will to produce a lower cost system. This broadens the performance range possible in a family of computers. If more tag bits are needed, as would be required for more than two-way compounding, a new tag memory will be substituted for the tag memory in FIG. 2B without requiring a change in the main memory design. Further, each memory can be provided with its own error correction.

With regard to FIGS. 2A-2D, it is asserted that once generated by the compounding unit, the compounding tags accompany the instruction stream in the memory, whether woven into the stream, appended to sections of it, or maintained in parallel with it.

Other approaches to implementing tag storage are illustrated in FIG. 2C and 2D. In FIG. 2C, a first section of the main memory contains tag tables, and a second storage of instruction text pages. In this example, operating system support is required to reserve the tag table portion of the memory and pair memory pages with tag pages. In FIG. 2D, portions of each page are reserved for tags. This requires a capability in the compiler for page construction. For example, with 64 byte cache lines, a compiler would use 60 bytes for instructions and 4 bytes for tags. In FIG. 2D tags are paired with instruction bytes in the instruction cache when requested by the CPU.

An implication of the computer system in FIG. 1 is that the instruction compounding unit 11 can form a part of the bus adaptor 8. Thus when any page is brought in from the IO system, it is subjected to the compounding process implicit in the unit 11 and moved on the memory bus 9 to the main memory 10. From hereon, the discussion assumes a page structure according to FIG. 2A, implying that the text bus 9b is 68 bits wide and that the main memory is configured and controlled to store pages such as that illustrated in FIG. 2A. Of course, the compounding instruction cache 12 is configured and controlled to receive lines including extended words as illustrated in FIG. 2A.

Upon a page fault, a page is loaded into a page buffer in the adaptor 8 and provided to the instruction compounding unit 11 as described below. In FIG. 3, two page buffers 18a and 18b send a sequence of pages to the instruction compounding unit 11 which undertakes compounding operations by adding compounding tag information to page instructions. Pages processed by the compounding unit 11 are fed to the main memory 10 through compounded page buffers 19a and 19b. As FIG. 4 shows, the compounding unit adds time to that required to fetch a text segment from auxiliary storage and enter it into the main memory 10. However, the time added is small relative to the total time required, and is asynchronous to the CPU.

In FIG. 4, each segment i is transferred from an auxiliary storage device such as a disk drive to one of the page buffers 18a or 18b. Each of the time segments b.sub.i indicate the time required to transfer text segment i from a page buffer to the main memory 10. Thus, text segment i is transferred in time a.sub.i into one of the page buffers 18a or 18b, following which text segment i+l is transferred to the other of the buffers. Without compounding, text segment i is transferred in time b.sub.i from the page buffer, where it is currently stored, into the main memory. As FIG. 4 shows, this time is substantially shorter than the time required to fetch a page to one of the buffers 18a or 18b. With the practice of this invention, the time required for the operation of the compounding unit 11 to be performed on a text segment in one of the page buffers plus the time spend in a compounded buffer 19a or 19b is represented by compound time c.sub.i. Now, in FIG. 4, the time b.sub.i is that required to transfer text segment i from a page buffer to the compounding unit 11. Next, the compounding time c.sub.i is incurred while text segment i is subjected to the process of the compounding unit 11. As FIG. 4 shows, the sum of the time b.sub.i and c.sub.i is less than the time a.sub.i. Recall that the superscaler machine must decide at instruction execution time whether instructions may be executed in parallel. This decision is a discrete step in instruction execution, thereby adding substantially to execution time in the superscaler machine. Contrastingly, as FIG. 4 illustrates, compounding in the computer system of FIG. 1 does not significantly extend the time required to perform computer operations. Thus, the instruction compounding unit 11 offers greater performance than a compounder located at the instruction execution unit.

FIGS. 3 and 4 illustrate two principal advantages of compounding in main memory. First, the compounding can be made part of the asynchronous page fault process without extending the time to complete that process. Second, compounding of large blocks of instruction text, such as pages, provides a larger scope of consideration for compounding, which can result in more optimized compounding. A consequence of this is that an in-memory instruction compounding unit, such as that illustrated in FIG. 1, will provide performance advantages, as the CPU will always execute instructions that have been compounded and the compounding can be better optimized than when performed synchronously on a smaller section of instruction text.

The operation of the instruction compounding unit will now be explained beginning with reference to FIG. 5a. FIG. 5a shows a portion of a stream of compounded or tagged instructions as they might appear at the output of the instruction compounding unit 11 of FIG. 1. As is seen, each instruction (Instr.) has a tag field added to it by the instruction compounding unit 11. The tagged instructions, like those shown in FIG. 5a are stored into the main memory in the page block for the page containing the instructions. As needed, tagged instructions in the main memory 10 are transferred to the cache 12 when a "miss" occurs. Thereafter, the tagged instructions in the cache 12 are fetched by the instruction fetch and issue unit 16. As the tagged instructions are received by the fetch and issue unit 16, their tagged fields are examined to determine if they may be processed in parallel and their operation code (OP CODE) fields are examined to determine which of the available functional units is most appropriate for their processing. If the tag fields indicate that two or more of the instructions are suitable for processing in parallel, then they are sent to the appropriate ones of the functional units in accordance with the codings of their 0P CODE fields. Such instructions are then processed concurrently with one another by their respective functional units.

When an instruction is encountered that is not suitable for parallel processing, then it is sent to the appropriate functional unit as determined by its OP CODE and it is thereupon processed alone and by itself by the selected functional unit.

In the most perfect case, where plural instructions are always being processed in parallel, the instruction execution rate of the computer system would be N times as great as for the case where instructions are executed one at a time, with N being the number of instructions in the groups which are being processed in parallel.

The tagged instruction stream of FIG. 5a is easier to preprocess by an instruction compounding unit if known reference points exist to indicate where instructions begin. Such a reference point will provide precise knowledge of where an instruction boundary occurs. In many computer systems, instruction boundaries are expressly known only by a compiler at compile time and only by a CPU when instructions are fetched. A boundary reference point is unknown between compile time and instruction fetch unless a special boundary reference scheme is adopted. Such a scheme is illustrated in FIG. 5b by instruction boundary bits B. As FIG. 5b illustrates, the boundary bits may be placed in the instruction stream by the compiler at compile time to provide a reference for instruction alignment just prior to compounding.

Generally, the patent applications entitled "Scalable Compound Instruction Set Machine Architecture" and "general Purpose compound Apparatus for Instruction-Level Parallel Processors" deal with the considerations of compounding with text streams in which instruction boundaries are indefinite. Of course, where instruction boundaries are determinable from the text stream, as where the stream includes only instructions and all instructions are the same length, boundary definition is unnecessary.

DESCRIPTION OF FIG. 6 INSTRUCTION COMPOUNDING UNIT

FIG. 6 shows in greater detail the internal construction of a representative embodiment of an instruction compounding unit in accordance with the present invention. This instruction compounding unit 20 is suitable for use as the instruction compounding unit 11 of FIG. 1. The instruction compounding unit 20 of FIG. 6 is designed for the case where a maximum of two instructions at a time may be processed in parallel. However, this is not meant to limit the invention only to pairwise compounding. In this example, a 1-bit tag field is used. A tag bit value of "1" (one) means that the instruction is a "first" instruction. A tag bit value of "0" (zero) means that the instruction is a "second" instruction and may be executed in parallel with the preceding first instruction. An instruction having a tag bit value of 1 may be executed either by itself or at the same time and in parallel with the next instruction, depending upon the tag bit value for such next instruction.

Each pairing of an instruction having a tag bit value of one with a succeeding instruction having a tag bit value of zero forms a compound instruction for parallel execution purposes, that is, the instructions in such a pair may be processed in parallel with one another. When the tag bits for two succeeding instructions each have a value of one, the first of these instructions is executed by itself in a nonparallel manner. In the worst possible case, all of the instructions in the sequence would have a tag bit value of one. In this worst case, all of the instructions would be executed one at a time in a nonparallel manner.

At the input to the instruction compounding unit 20, an instruction alignment unit receives from the I/O adaptor the instruction stream which is to be compounded. The instruction stream may include boundary bits B, as illustrated in FIG. 5b. In this case, instruction alignment is simply a matter of detecting boundary bits and decoding instruction OP codes. As is known, in the IBM System/370 instruction set, OP codes include bits which give instruction length in bytes or half words. Therefore, once a boundary bit B has been identified for an instruction, the next instruction can be unambiguously identified by counting the number of bytes or half words from the boundary bit. Instruction alignment is not a feature of this invention, it being understood that instruction boundaries are identified by any known method, including the use of boundary bits.

The instruction compounding unit 20 of FIG. 6 includes a plural-instruction instruction register 21 for receiving a plurality of successive instructions from the page buffers 18a and 18b of the adapters. Instruction compounding unit 20 also includes a plurality of rule-based instruction analyzer mechanisms. Each such instruction analyzer mechanism analyzes a different pair of side-by-side instructions in the instruction register 21 and produces a compoundability signal which indicates whether or not the two instructions in its pair may be processed in parallel. In FIG. 6, there are shown a plurality of compound analyzer units 22-25. Each of these compound analyzer units 22-25 includes two of the instruction analyzer mechanisms just mentioned. Thus, each of these analyzers units 22-25 produces two of the compoundability signals. For example, the first compound analyzer unit 22 produces a first compoundability signal M01 which indicates whether or not Instructions 0 and 1 may be processed in parallel. Compound analyzer unit 22 also produces a second compoundability signal M12 which indicates whether or not Instructions 1 and 2 may be processed in parallel.

In a similar manner, the second compound analyzer unit 23 produces a first compoundability signal M23 which indicates whether or not Instructions 2 and 3 may be processed in parallel and a second compoundability signal M34 which indicates whether Instructions 3 and 4 may be processed in parallel. The third compound analyzer 24 produces a first compoundability signal M45 which indicates whether or not Instructions 4 and 5 may be processed in parallel and a second compoundability signal M56 which indicates whether or not Instructions 5 and 6 may be processed in parallel. The fourth compound analyzer 25 produces a first compoundability signal M67 which indicates whether or not Instructions 6 and 7 may be processed in parallel and a second compoundability signal M78 which indicates whether Instructions 7 and 8 may be processed in parallel.

The instruction compounding unit 20 further includes a tag generating mechanism 26 responsive to the compoundability signals appearing at the outputs of the analyzer units 22-25 for generating the individual tag fields for the different instructions in the instruction register 21. These tag fields T0, T1, T2 etc. are supplied to a tagged instruction register 27, as are the instructions themselves, the latter being obtained from the input instruction register 21. In this manner, there is provided in the compounding unit output register 27 a tag field T0 for Instruction 0, a tag field T1 for Instruction 1, etc.

In the present embodiment, each tag field T0, T1, T2, etc. is comprised of a single binary bit. A tag bit value of "one" indicates that the immediately following instruction to which it is attached is a "first" instruction. A tag bit value of "zero" indicates that the immediately following instruction is a "second" instruction. An instruction having a tag bit value of one followed by an instruction having a tag bit value of zero indicates that those two instructions may be executed in parallel with one another. The tagged instructions in the compounding unit output register 27 are supplied to the input of the main memory 10 of FIG. 1 via one or the other of the compounding buffers 19a or 19b of FIG. 3. The compounded instructions are stored into the main memory 10.

Referring now to FIG. 7, there is shown in greater detail the internal construction used for the compound analyzer unit 22 of FIG. 6. The other compound analyzer units 23-25 are of a similar construction. As shown in FIG. 7, the compound analyzer 22 includes instruction compatibility logic 30 for examining the op code of Instruction 0 and the op code of Instruction 1 and determining whether these two op codes are compatible for purposes of execution in parallel. Logic 30 is constructed in accordance with predetermined rules to select which pairs of op codes are compatible for execution in parallel. More particularly, logic 30 includes logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution in the particular hardware configuration used for the computer system being considered. If the op codes for Instruction 0 and 1 are compatible, then logic 30 produces at its output a binary one level signal. If they are not compatible, logic 30 produces a binary zero value on its output line.

Compound analyzer 22 further includes a second instruction compatibility logic 31 for examining the op codes of Instructions 1 and 2 and determining whether they are compatible for parallel execution. Logic 31 is constructed in the same manner as logic 30 in accordance with the same predetermined rules used for logic 30 to select which pairs of op codes are compatible for execution in parallel for the case of

Instructions 1 and 2. Thus, logic 31 includes logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution, these rules being the same as those used in logic 30. If the op codes for Instructions 1 and 2 are compatible, then logic 31 produces a binary one level output. Otherwise, it produces a binary zero level output.

Compound analyzer 22 further includes first register dependency logic 32 for detecting conflicts in the usage of the general purpose registers designated by the R1 and R2 fields of Instructions 0 and 1. These general purpose registers will be discussed in greater detail hereinafter. Among other things, dependency logic 32 may be constructed to detect the occurrence of a data dependency condition wherein a second instruction (Instruction 1) needs to use the results obtained by the performance of the proceeding instruction (Instruction 0). In this case, either the second instruction can be executed by the dependency collapsing hardware, thus executing in parallel with the first instruction, or the execution of the second instruction must await completion of the execution of the preceding instruction and, hence, cannot be executed in parallel with the preceding instruction. (It is noted that a technique for circumventing some data dependencies of this type will be discussed hereinafter.) If there are no register dependencies which prevent execution of Instructions 0 and 1 in parallel, then the output line of logic 32 is given a binary value of one. If there is a dependency, then it is given a binary value of zero.

Compound analyzer 22 further includes second register dependency logic 33 for detecting conflicts in the usage of the general purpose registers designated by the R1 and R2 fields of Instructions 1 and 2. This logic 33 is one of the same construction as the previously discussed logic 32 and produces a binary one level output if there are no register dependencies or the register dependencies can be executed by the data dependency collapsing hardware, and a binary zero level output otherwise.

The output lines from the instruction compatibility logic 30 and the register dependency logic 32 are connected to the two inputs of an AND circuit 34. The output line of AND 34 has a binary one value if the two op codes being considered are compatible and if there are no register dependencies. This binary one value on the AND 34 output line indicates that the two instructions being considered are compatible, that is, are executable in parallel. If, on the other hand, the AND 34 output line has a binary value of zero, then the two instructions are not compoundable. Thus, there is produced on the AND 34 output line a first compoundability signal M01 which indicates whether or not Instructions 0 and 1 may be processed in parallel. This M01 signal is supplied to the tag generator 26.

The output lines from the second compatibility logic 31 and the second dependency logic 33 are connected to the two inputs of AND circuit 35. AND 35 produces on its output line a second compoundability signal M12 which has a binary value of one if the two op codes being considered (op codes for Instructions 1 and 2) are compatible and if there are no register dependencies for Instructions 1 and 2 or register dependencies that can be executed by the data dependency collapsing hardware. Otherwise, the AND 35 output line has a binary value of zero. The output line from AND 35 runs to a second input of the tag generator 26.

The other compound analyzers 23-25 shown in FIG. 6 are of the same internal construction as shown in FIG. 7 for the first compound analyzer.

Referring now to FIG. 8, there is shown an example of the logic circuitry that can be used to implement the compound analyzer 22 and the portion of the tag generator 26 which is used to generate the first three tags, Tag 0 and Tag 1 and Tag 2. For the example of FIG. 5, it is assumed that there are two categories of instructions which are designated as category A and category B. The rules for compounding these categories of instructions are assumed to be as follows:

(1) A can always compound with A

(2) A can never compound with B

(3) B can never compound with B

(4) B can always compound with A

(5) Rule (4) has preference over Rule (1).

Note that these rules are sensitive to the order of occurrence of the instructions.

It is further assumed that these rules are such that when they are observed, there will be no problems with register dependencies because the rules implicitly indicate that in case there is any interlock, such an interlock is always executable by the data dependency collapsing hardware. In other words, it is assumed for the FIG. 8 example, that the register dependency logics 32 and 33 of FIG. 7 are not needed. In such case, AND circuits 34 and 35 are also not needed and the output of logic 30 becomes the M01 signal and the output of logic 31 becomes the M12 signal.

For these assumptions, FIG. 8 shows the internal logic circuitry that may be used for the instruction compatibility logic 30 and the instruction compatibility logic 31 of FIG. 7. With reference to FIG. 8, the instruction compatibility logic 30 includes decoders 40 and 41, AND circuits 42 and 43 and OR circuit 44. The second instruction compatibility logic 31 includes decoders 41 and 45, AND circuits 46 and 47 and OR circuit 48. The middle decoder 41 is shared by both logics 30 and 31.

The first logic 30 examines the op codes OP0 and OP1 of Instructions 0 and 1 to determine their compatibility for parallel execution purposes. This is done in accordance with Rules (1)-(4) set forth above. Decoder 40 looks at the op code of the first instruction and if it is a category A op code, the A output line of decoder 40 is set to the one level. If OP0 is a category B op code, then the B output line of decoder 40 is set to a one level. If Op0 belongs to neither category A nor category B, then both outputs of decoder 40 are at the binary zero level. The second decoder 41 does a similar kind of decoding for the second op code OP1.

AND circuit 42 implements Rule (1) above. If OP0 is a category A op code and OP1 is also a category A op code, then AND 42 produces a one level output. Otherwise, the output of AND 42 is a binary zero level. AND 43 implements Rule (4) above. If the first op code is a category B op code and the second op code is a category A op code, then AND 43 produces a one level output. Otherwise, it produces a zero level output. If either AND 42 or AND 43 produces a one level output, this drives the output of OR circuit 44 to one level, in which case, the compoundability signal M01 has a value of one. This one value indicates that the first and second instructions (Instructions 0 and 1) are compatible for parallel execution purposes.

If any other combination of op code categories is detected by decoders 40 and 41, then the outputs of AND 42 and 43 remain at the zero level and compoundability signal M01 has the noncompoundability-indicating value of zero. Thus, the occurrence of the combinations indicated by Rules (2) and (3) above do not satisfy AND's 42 and 43 and MO1 remains at the zero level. If there are further categories of op codes in addition to categories A and B, their occurrences in the instruction stream do not activate the outputs of decoders 40 and 42. Hence, they likewise result in an M01 compoundability signal value of zero.

The second instruction compatibility logic 31 performs a similar type of op code analysis for the second and third instructions (Instructions 1 and 2). If the second op code OP1 is a category A op code and the third op code OP2 is a category A op code, then, per Rule (1), AND 46 produces a one level output and the second compoundability signal M12 is driven to the compoundability-indicating binary one level. If, on the other hand OP1 is a category B op code and OP2 is a category A op code, then, per Rule (4), AND 47 is activated to produce a binary one level for the second compoundability signal M12. For any op code combination other than those set forth in Rules (1) and (4), the M12 signal has a value of zero.

The M01 and M12 compoundability signals are supplied to the tag generator 26. FIG. 8 shows the logic circuitry that can be used in tag generator 26 to respond to the M01 and M12 compoundability signals to produce the desired tag bit values for Tags 0, 1 and 2. A tag bit value of one indicates that the associated instruction is "first" instruction for parallel execution purposes. A tag bit value of zero indicates that the associated instruction is a "second" instruction for parallel execution purposes. The only instruction in the pair has a tag bit value of zero. Any instruction having a tag bit value of one which is followed by another instruction having a tag bit value of one is executed by itself in a singular manner and not in parallel with the following instruction.

For the case of the first row in FIG. 9, all three tag bits have a value of one. This means that each of Instructions 0 and 1 will be executed in a singular, nonparallel manner. For the second row of FIG. 2, Instructions 0 and 1 will be executed in parallel since Tag 0 has the required one value and Tag 1 has the required zero value. For the third row in FIG. 9, Instruction 0 will be executed in a singular manner, while Instructions 1 and 2 will be executed in parallel with one another. For the fourth row, Instructions 0 and 1 will be executed in parallel with one another.

For those cases where Tag 2 has a binary value of one, the status of its associated Instruction 2 is dependent on the binary value for Tag 3. If Tag 3 has a binary value of zero, then Instructions 2 and 3 can be executed in parallel. If, on the other hand, Tag 3 has a binary value of one, then Instruction 2 will be executed in a singular, nonparallel manner. It is noted that the logic implemented for the tag generator 26 does not permit the occurrence of two successive tag bits having binary values of zero.

An examination of FIG. 9 reveals the logic needed to be implemented by the portion of tag generator 26 shown in FIG. 8. As indicated in FIG. 9, Tag 0 will always have a binary value of one. This is accomplished by providing a constant binary value of one to tag generator output line 50 which constitutes the Tag 0 output line. An examination of FIG. 9 further reveals that the bit value for Tag 1 is always the opposite of the bit value of the M01 compoundability signal. This result is accomplished by connecting output line 51 for Tag 1 to the output of NOT circuit 52, the input of which is connected to the M01 signal line.

The binary level on Tag 2 output line 53 is determined by an OR circuit 54 and a NOT circuit 55. One input of OR 54 is connected to the M01 line. If M01 has a value of one, then Tag 2 has a value of one. This takes care of the Tag 2 values in the second and fourth rows of FIG. 9. The other input of OR 54 is connected by way of NOT 44 to the M12 signal line. If M12 has a binary value of zero, this value is inverted by NOT 55 to supply a binary one value to the second input of OR 54. This causes the Tag 2 output line 53 to have a binary one value. This takes care of the Tag 2 value for row one of FIG. 9. Note that for the row 3 case, Tag 2 must have a value of zero. This will occur because, for this case, M01 will have a value of zero and M12 will have a value of one which is inverted by NOT 55 to produce a zero at the second input of OR 54.

Implicit in the logic of FIG. 9 is a prioritization rule for the row four case where each of M01 and M12 has a binary value of one. this row four case can be produced by an instruction category sequence of BAA. This could be implemented by a tag sequence of 101 as shown in FIG. 9 or, alternatively, by a tag sequence of 110. In the present embodiment, Rule (5) is followed and the 101 sequence shown in FIG. 9 is chosen. In other words, the BA pairing is given preference over the AA pairing.

The 1,1 pattern for M01 and M12 can also be produced an an op code sequence of AAA. In this case, the 101 tag sequence of FIG. 9 is again selected. This is better because it provides a one value for Tag 2 and, hence, potentially enables Instruction 2 to be compounded with Instruction 3 if Instruction 2 is compatible with Instruction 3.

DESCRIPTION OF THE FIG. 10 EMBODIMENT

Referring to FIG. 10, there is shown a detailed example of how a computer system can be constructed for using the compounding tags of the present invention to provide parallel processing of machine-level computer instructions. The instruction compounding unit 20 used in FIG. 10 is assumed to be of the type described in FIG. 6 and, as such, it adds to each instruction a one-bit tag field. These tag fields are used to identify which pairs of instructions may be processed in the parallel. Pages containing these tagged instructions are supplied to and stored into the main memory 10. As the tagged instructions are needed, they are read or transferred into the cache 12. Fetch/Issue control unit 60 fetches the tagged instructions from cache 12, as needed, and arranges for their processing by the appropriate one or ones of a plurality of functional instruction processing units 61, 62, 63 and 64. Fetch/Issue unit 60 examines the tag fields and op code fields of the fetched instructions. If the tag fields indicate that two successive instructions may be processed in parallel, then fetch/issue unit 60 assigns them to the appropriate ones of the functional units 61-64 as determined by their op codes and they are processed in parallel by the selected functional units. If the tag fields indicate that a particular instruction is to be processed in a singular, nonparallel manner, then fetch/issue unit 60 assigns it to a particular functional unit as determined by its op code and it is processed or executed by itself.

The first functional unit 61 is a branch instruction processing unit for processing branch type instructions. The second functional unit 62 is a three input address generation arithmetic and logic unit (ALU) which is used to calculate the storage address for instructions which transfer operands to or from storage. The third functional unit 63 is a general purpose arithmetic and logic unit (ALU) which is used for performing mathematical and logical type operations. The fourth functional unit 64 in the present example is a data dependency collapsing ALU of the kind described in the above-referenced U.S. Pat. No. 5,051,940, granted Sep. 24, 1991. This dependency collapsing ALU 64 is a three-input ALU capable of performing two arithmetical/logical operations in a single machine cycle.

The computer system embodiment of FIG. 10 also includes a set of general purpose registers 65 for use in executing some of the machine-level instructions. Typically, these general purpose registers 65 are used for temporarily storing data operands and address operands or are used as counters or for other data processing purposes. In a typical computer system, sixteen (16) such general purpose registers are provided. In the present embodiment, general purpose registers 65 are assumed to be one of the multiport type wherein two or more registers may be accessed at the same time.

The computer system of FIG. 10 further includes a high-speed data cache storage mechanism 66 for storing data operands obtained from the higher-level storage unit 10. Data in the cache 66 may also be transferred back to the main memory 10. Data cache 66 may be of a known type and its operation relative to the main memory 10 may be conducted in a known manner.

FIG. 11 shows an example of a compounded or tagged instruction sequence which may be processed by the computer system of FIG. 10. The FIG. 11 example is composed of the following instructions in the following sequence: Load, Add, Compare, Branch on Condition and Store. These are identified as instructions I1-I5, respectively. The tag bits for these instructions are 1,1,0,1 and 0, respectively. Because of the organization of the machine shown in FIG. 10, the Load instruction is processed in a singular manner by itself. The Add and Compare instructions are treated as a compound instruction and are processed in parallel with one another. The Branch and Store instructions are also treated as a compound instruction and are also processed in parallel with one another.

The table of FIG. 12 gives further information on each of these FIG. 11 instructions. The R/M column in FIG. 12 indicates the content of a first field in each instruction which is typically use to identify the particular one of general purpose registers 65 which contains the first operand. An exception is the case of the Branch on Condition instruction, wherein the R/M field contains a condition code mask. The R/X column in FIG. 12 indicates the content of a second field in each instruction, which field is typically used to identify a second one of the general purpose registers 65. Such register may contain the second operand or may contain an address index value (X). The B column in FIG. 12 indicates the content of a third possible field in each instruction, which field may identify a particular one of the general purpose registers 65 which contains a base address value. A zero in the B column indicates the absence of a B field or the absence of a corresponding address component in the B field. The D field of FIG. 12 indicates the content of a further field in each instruction which, when used for address generation purposes, includes an address displacement value. A zero in the D column may also indicate the absence of a corresponding field in the particular instruction being considered or, alternatively, an address displacement value of zero.

Considering now the processing of the Load instructions of FIG. 11, the fetch/issue control unit 60 determines from the tag bits for this Load instruction and the following Add instruction that the Load instruction is to be processed in a singular manner by itself. The action to be performed by this Load instruction is to fetch an operand from storage, in this case the data cache 66, and to place such operand into the R2 general purpose register. The storage address from which this operand is to be fetched is determined by adding together the index value in register X, the base value in register B and the displacement value D. The fetch/issue control unit 60 assigns this address generation operation to the address generation ALU 62. In this case, ALU 62 adds together the address index value in register X (a value of zero in the present example), the base address value contained in general purpose register R7 and the displacement address value (a value of zero in the present example) contained in the instruction itself. The resulting calculated storage address appearing at the output of ALU 62 is supplied to the address input of data cache 66 to access the desired operand. This accessed operand is loaded into the R2 general purpose register in register set 65.

Considering now the processing of the Add and Compare instructions, these instructions are fetched by the fetch/issue control unit 60. The control unit 60 examines the compounding tags for these two instructions and notes that they may be executed in parallel. As seen from FIG. 12, the Compare instruction has an apparent data dependency on the Add instruction since the Add must be completed before R3 can be compared. This dependency, however, can be handled by the data dependency collapsing ALU 64. Consequently, these two instructions can be processed in parallel in the FIG. 10 configuration. In particular, the control unit 60 assigns the processing of the Add instruction to ALU 63 and assigns the processing of the Compare instruction to the dependency collapsing ALU 64.

ALU 63 adds the contents of the R2 general purpose register to the contents of the R3 general purpose register and places the result of the addition back into the R3 general purpose register. At the same time, the dependency collapsing ALU 64 performs the following mathematical operation:

R3+R2-R4

The condition code for the result of this operation is sent to a condition code register located in branch unit 61. The data dependency is collapsed because ALU 64, in effect, calculates the sum of R3+R2 and then compares this sum with R4 to determine the condition code. In this manner, ALU 64 does not have to wait on the results from the ALU 63 which is performing the Add instruction. In this particular case, the numerical results of calculated by the ALU 64 and appearing at the output of ALU 64 is not supplied back to the general purpose registers 65. In this case, ALU 64 merely sets the condition code.

Considering now the processing of the Branch instruction and the Store instruction shown in FIG. 11, these instructions are fetched from the compound instruction cache 12 by the fetch/issue control unit 60. Control unit 60 determines from the tag bits for these instructions that they may be processed in parallel with one another. It further determines from the op codes of the two instructions that the Branch instruction should be processed by the branch unit 61 and the Store instruction should be processed by the address generation ALU 62. In accordance with this determination, the mask field M and the displacement field D of the Branch instruction are supplied to the branch unit 61. Likewise, the address index value in register X and the address base value in register B for this Branch instruction are obtained from the general purpose registers 65 and supplied to the branch unit 61. In the present example, the X value is zero and the base value is obtained from the R7 general purpose register. The displacement value D has a hexadecimal value of twenty, while the mask field M has a mask position value of eight.

The branch unit 61 commences to calculate the potential branch address (0+R7+20) and at the same time compares the condition code obtained from the previous Compare instruction with the condition code mask M. If the condition code value is the same as the mask code value, the necessary branch condition is met and the branch address calculated by the branch unit 61 is thereupon loaded into an instruction counter in control unit 60. This instruction counter controls the fetching of the instructions from the compound instruction cache 12. If, on the other hand, the condition is not met (that is, the condition code set by the previous instruction does not have a value of eight), then no branch is taken and no branch address is supplied to the instruction counter in control unit 60.

At the same time that the branch unit 61 is busy carrying out its processing actions for the Branch instruction, the address generation ALU 62 is busy doing the address calculation (0+R7+0) for the Store instruction. The address calculated by ALU 62 is supplied to the data cache 66. If no branch is taken by the branch unit 61, then the Store instruction operates to store the operand in the R3 general purpose register into the data cache 66 at the address calculated by ALU 62. If, on the other hand, the branch condition is met and the branch is taken, then the contents of the R3 general purpose register is not stored into the data cache 66.

The foregoing instruction sequence of FIG. 11 is intended as an example only. The computer system embodiment of FIG. 10 is equally capable of processing various and sundry other instruction sequences. The example of FIG. 11, however, clearly shows the utility of the compound instructions tags in determining which pairs of instructions may be processed in parallel with one another.

Each pairing of an instruction having a tag bit value of one with a succeeding instruction having a tag bit value of zero forms a compound instruction for parallel execution purposes, that is, the instructions in such a pair may be processed in parallel with one another. When the tag bits for two succeeding instructions each have a value of one, the first of these instructions is executed by itself in a nonparallel manner. In the worst possible case, all of the instructions in the sequence would have a tag bit value of one. In this worst case, all of the instructions would be executed one at a time in a nonparallel manner.

The hardware example discussed above in connection with the preferred embodiment of this invention compounds over a small scope. In this regard, each pair of adjacent instructions is analyzed to determine whether the pair can be executed in parallel. In fact, memory compounding offers the possibility of examining many compoundings over more than two instructions and choosing the best grouping available.

The examples given above also use a compounding technique that assumes knowledge of where instructions start. In the general case, instruction boundaries can be identified by the compiler, as discussed above, or by instruction decoding prior to execution. Reference is given to the co-pending applications entitled "Scalable Compounding Instruction Set Machine Architecture" and "General Purpose Compound Apparatus for Instruction-Level Parallel Processors".

Last, the instruction compounding unit has been illustrated particularly as being positioned between the I/O adaptor and the memory bus. This example is not meant to exclude any other locations in memory where the instruction compounding unit can operate. For example, it can be absorbed into the I/O adaptor, it can operate as a separate unit on the memory bus 9 (at which location it could compound either in the main memory 10 or in the compound instruction cache 12), or it can comprise a unit attached only to the main memory through a private memory port not accessible over the memory bus 9. The compounder can also function between the main memory and instruction cache, as taught in the co-pending application entitled "Compounding Preprocessor for Cache".

While we have described a preferred embodiment of our invention, it should be understood that modifications and adaptations thereof will occur to persons skilled in the art. Therefore, the protection afforded our invention should only be limited in accordance with the scope of the following claims.

* * * * *

United States Patent ***5,303,356***
Vassiliadis , et al. **April 12, 1994**

System for issuing instructions for parallel execution subsequent to branch into a group of member instructions with compoundability in dictation tag

Abstract

An instruction processor system for decoding compound instructions created from a series of base instructions of a scalar machine, the processor generating a series of compound instructions with an instruction format text having appended control bits in the instruction format text enabling the execution of the compound instruction format text in said instruction processor with a compounding facility which fetches and decodes compound instructions which can be executed as compounded and single instructions by the arithmetic and logic units of the instruction processor while preserving intact the scalar execution of the base instructions of a scalar machine which were originally in storage. The system nullifies any execution of a member instruction unit of a compound instruction upon occurrence of possible conditions, such as branch, which would affect the correctness of recording results of execution of the member instruction unit portion based upon the interrelationship of member units of the compound instruction with other instructions. The resultant series of compounded instructions generally executes in a faster manner than the original format which is preserved due to the parallel nature of the compounded instruction stream which is executed.

Inventors: **Vassiliadis; Stamatis** (Vestal, NY); **Blaner; Bartholomew** (Newark Valley, NY); **Jeremiah; Thomas L.** (Endwell, NY)
Assignee: **International Business Machines Corporation** (Armonk, NY)

Appl. No.: **677685**
Filed: **March 29, 1991**

Current U.S. Class:	**712/238**; 712/23; 712/24; 712/214
Intern'l Class:	G06F 009/38
Field of Search:	395/375,800

References Cited [Referenced By]

U.S. Patent Documents

4295193	Oct., 1981	Pomerene.	
4439828	Mar., 1984	Martin.	
4574348	Mar., 1986	Scallon.	
4586127	Apr., 1986	Horvath.	
4594655	Jun., 1986	Hao et al.	
4755966	Jul., 1988	Lee et al.	
4760520	Jul., 1988	Shintani et al.	
4780820	Oct., 1988	Sowa.	
4807115	Feb., 1989	Torng.	
4847755	Jul., 1989	Morrison et al.	
4942525	Jul., 1990	Shintani et al.	
5075844	Dec., 1991	Jardine et al.	395/375.
5203002	Apr., 1993	Wetzel	395/800.
5237666	Aug., 1992	Suzuki et al.	395/375.

Other References

Acosta, R. D., et al, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al., the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al., ed (McGraw-Hill, 1982, pp. 276-292)).
Capozzi, A. J., et al., "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al., "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al., "SIMP (Single Instruction Stream/Multiple Instruction Pipelining); A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M. D., et al., "Limits on Multiple Instruction Issue", Asplos III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units",

Computer Structures, Principles, and Examples (Siewiorek, et al. ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al., "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, n.P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun. 1988, pp. 63-76.
Colwell, R. P., et al., "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The Vliw Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the Crisp Instruction Set Architecture", Proceedings of Compcon, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al., "Compiling for the Crisp Microprocessor", Proceedings of Compcon, Spring, 1987, pp. 96-100.
Hennessy, J., et al., "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al., "Branch Folding in the Crisp Microprocessor: Reducing Branch Delay to Zero", Proceedings of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W., et al., "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al., "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1. Jan. 1984, pp. 6-22.
Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al., Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al. "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al., "A Study of Instruction Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun. 1983, pp. 132-137.
Vassiliadis, S., et al., "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.

Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard "Early Release of a Processor Following Address Translation Prior to Page Access Checking".

Primary Examiner: Harrell; Robert B.
Assistant Examiner: Kim; Ken S.
Attorney, Agent or Firm: Augspurger; Lynn L.

Parent Case Text

CROSS REFERENCE TO RELATED APPLICATIONS

This application claims priority and U.S.A. continuation-in-part status continuing U.S. Ser. No. 07/519,384, filed May 4, 1990, now abandoned, U.S. Ser. No. 07/543,458, filed Jun. 26, 1990, now issued U.S. Pat. No. 5,197,135, and U.S. Ser. No. 07/642,011, filed Jan. 15, 1991, and U.S. Ser. No. 07/519,382, filed May 4, 1990, now abandoned.

Claims

What is claimed is:

1. A system mechanism for processing instructions including compoundable instructions which are comprised of a group of member units up to a maximum of member units which can be executed after issue in parallel as a compound instruction in a machine having a plurality of functional units for processing instructions concurrently with branching into the middle of a compound instruction set of member units which may be executed together in parallel as an executable compound instruction, comprising,

means for executing a branch instruction in one of the functional units; and

means for fetching for decoding an instruction string text including one or more compound instruction having one or more member units which can be addressed by a branch target address for potential execution;

decoding means for decoding a compound instruction contained in fetched instruction string text, said compound instructions comprising a string sequence of instructions to be executed by the machine and including of member units of a compound instruction that have an instruction format including an opcode and control bits associated with the member unit's instruction, said control bits indicating whether the associated member unit can be executed in parallel with a subsequent member unit as an executable compound instruction; and wherein said decoding means examines the opcode and appended control bits in an instruction format of said compound instruction member units which are relevant to the execution of a compound instruction, and determines if a member unit of the text begins a compound instruction or are a subsequent portion of an executable compound instruction; and

issuing means, responsive at least in part to the determination result of the decoding means, for issuing the member units of a compound instruction which includes a branch target addressed member unit to the functional units from a target addressed member unit at any sequential place in said string sequence to the end of the compound instruction containing the target addressed member unit for executing all of the member units of the executable compound instruction beginning with the target addressed member unit to a beginning member unit of a subsequent executable compound instruction so issued concurrently and maintaining the remainder of the text including the beginning member unit of a subsequent executable compound instruction for potential execution after the execution of the compound instruction has been initiated.

2. A system mechanism according to claim 1, wherein

the issue means issues instructions to the functional units after said decoding means has made a determination whether a fetched member unit of the instruction string text begins a an executable compound instruction sequence; and further comprising,

execution means within the functional units for executing said member units of an executable compound instruction sequence which includes a branch member unit; and

nullifying means for nullifying any execution of a member unit of a compound instruction upon occurrence of possible conditions which would affect the correctness of results of execution of the said member unit of the compound instruction following a branch member unit or additional subsequent instructions containing a second branch member unit that is followed by text that cannot be allowed to update a machine state of the system mechanism should the second branch member unit be taken by issue to the functional units.

3. A system mechanism according to claim 1, wherein nullifying means are provided for nullifying the result of a member unit, and when said compound instruction includes a member unit that is a branch instruction which is able to be executed in parallel with another member unit of the compound instruction, then, if a branch is taken, said nullifying means nullifies the result of the member unit(s) following the branch taken so that the effect of the subsequent instruction is as if said nullified member unit(s) were not executed.

4. A system mechanism according to claim 1, where fetching means have been provided to fetch a complete compound instruction and wherein said decoding means causes execution of component member units of a compound instruction to start at an arbitrary point in the compound instruction as determined by branch execution and executes to the end of the compound instruction.

5. A system mechanism according to claim 1, wherein said decoding means detects the end of a

compound instruction and causes the issuing means to maintain the remainder of the instruction text for later execution.

6. A system mechanism according to claim 1 wherein the compounding control bits for each one member unit instruction of the string of instructions, specifies, if the one instruction member unit may be executed in parallel with a subsequent instruction of the sequence of instructions as a compound instruction, or, specifies, if the instruction may not be executed in parallel with a preceding instruction member unit of the string of instructions, that the instruction must be executed as a single instruction.

7. A system mechanism according to claim 1 wherein the compounding control bits for each one member unit instruction of the string of instructions, specifies, if the one instruction member unit may be executed in parallel with a next instruction of the sequence of instructions as a compound instruction, or, the number of subsequent instructions with which the instruction can be executed, or which specifies, if the instruction may not be executed in parallel with a preceding instruction of the sequence of instruction, that the instruction must be executed as a single instruction.

8. A system mechanism according to claim 1 wherein the compounding control bits for each one member unit instruction of the string of instructions, specifies, if the one instruction may be executed in parallel with a subsequent instruction of the sequence of instructions, or, specifies, if the instruction may not be executed in parallel with a preceding instruction of the sequence of instruction, that the instruction must be executed as a single instruction.

9. A system mechanism according to claim 2, wherein the compound instruction is a multimember compound instruction and one of said member units of the compound instruction is a branch followed by at least a second member unit, and wherein during execution if a branch is taken the execution results of the second member unit following the branch are nullified.

10. A system mechanism according to claim 2, wherein if a second member unit of a compound instruction after a branch appears to be an instruction, and such said second member unit could begin to be executed as if it were an instruction, the nullifying means nullifies the execution of the second member unit of a compound instruction after a branch.

11. A system mechanism according to claim 2 wherein the compounding control bits for each one member unit instruction of the string of instructions, specifies, if the one instruction member unit may be executed in parallel with a subsequent instruction of the sequence of instructions as a compound instruction, or, specifies, if the instruction may not be executed in parallel with a preceding instruction member unit of the string of instructions, that the instruction must be executed as a single instruction.

12. A system mechanism according to claim 2 wherein the compounding control bits for each one member unit instruction of the string of instructions, specifies, if the one instruction member unit may be executed in parallel with a next instruction of the sequence of instructions as a compound instruction, or, the number of subsequent instructions with which the instruction can be executed, or which specifies, if the instruction may not be executed in parallel with a preceding instruction of the sequence of instruction, that the instruction must be executed as a single instruction.

13. A system mechanism according to claim 2 wherein the compounding control bits for each one

member unit instruction of the string specifies, if the one instruction may be executed in parallel with a next one instruction of the sequence of instructions, the number of subsequent instructions with which the instruction can be executed, or which specifies, if the instruction may not be executed in parallel with a preceding instruction of the sequence of instruction, that the instruction must be executed as a single instruction.

Description

The present U.S. patent application is related to the following co-pending U.S. patent applications:

(1) Application Ser. No. 07/519,384, filed May 4, 1990, now abandoned, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al; and

(2) Application Ser. No. 07/519,382, filed May 4, 1990, now abandoned, entitled "General Purpose Compounding Technique For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al; and

(3) Application Ser. No. 07/504,910, filed Apr. 4, 1990, now issued U.S. Pat. No. 5,051,940, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al; and

(4) Application Ser. No. 07/522,291, filed May 10, 1990, now issued U.S. Pat. No. 5,214,763 entitled "Compounding Preprocessor For Cache", the inventors being Bartholomew Blaner et al; and

(5) Application Ser. No. 07/543,464, filed Jun. 26, 1990, now abandoned, entitled "An In-Memory Preprocessor for a Scalable Compound Instruction Set Machine Processor", the inventors being Richard Eickemeyer et al; and

(6) Application Ser. No. 07/543,458, filed Jun. 26, 1990, now issued U.S. Pat. No. 5,197,135, entitled "Memory Management for Scalable Compound Instruction Set Machines With In-Memory Compounding", the inventors being Richard J. Eickemeyer et al; and

(7) Application Ser. No. 07/619,868, filed Nov. 28, 1990, entitled "Overflow Determination for Three-Operand ALUS in a Scalable Compound Instruction Set Machine", the inventors being Stamatis Vassiliadis et al; and

(8) Application Ser. No. 07/642,011, filed Jan. 15, 1991, entitled "Compounding Preprocessor for Cache", the inventors being Bartholomew Blaner et al.

These co-pending applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y.

The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

A review of these related cases will show that we have illustrated FIGS. 1-2 in U.S. Ser. No. 07/519,384 filed May 4, 1990, while FIG. 3 may also be found described in U.S. Ser. No. 07/543,458 filed Jun. 26, 1990, and FIG. 4-B has been generally described in U.S. Ser. No. 07/642,011 filed Jan. 15, 1991, while FIGS. 5-7 were also illustrated in U.S. Ser. No. 07/591,382 filed May 4, 1990.

FIELD OF THE INVENTION

This invention relates to digital computers and digital data processors, and particularly to digital computers and data processors capable of executing two or more instructions in parallel, and provides a mechanism for branching in the middle of a compound instruction. Particularly also, the system thus provided enables a set of instructions which may be combined into compound instructions to be statically determined, and allows the appending of control information to the instructions in the program to be executed with branching in the middle of a compound instruction enabled.

BACKGROUND OF THE INVENTION

Traditional computers which receive a sequence of instructions and execute the sequence one instruction at a time are known, and are referred to as "scalar" computers. With each new generation of computing machines, new acceleration mechanism must be discovered for traditional scalar machines. A recent mechanism for accelerating computational speed is found in reduced instruction set architecture (RISC) that employs a limited set of very simple instructions executing at a high rate of speed. Alternatively, another acceleration mechanism may be to add more complex instructions to the architecture to provide more computation function per instruction and thus reduce the number of instructions in a program. Application of either of these approaches to an existing scalar computer would require a fundamental alteration of the instruction set and architecture of the machine. Such a far-reaching transformation is fraught with expense, down-time, and an initial reduction in the machine's reliability and availability.

Recently, "superscalar" computers have been developed which further accelerate computation speed. These machines are essentially scalar machines whose performance is increased by adapting them to execute more than one instruction at a time from an instruction stream including a sequence of single scalar instructions. These machines typically decide at instruction execution time whether two or more instructions in a sequence of scalar instructions may be executed in parallel. The decision is based upon the operation codes (OP codes) of the instructions and on data dependencies which may exist between instructions. An OP code signifies the computational hardware required for an instruction. In general, it is not possible to concurrently execute two or more instructions which utilize the same hardware (a hardware dependency) or the same operand (a data dependency). These hardware and data dependencies prevent the parallel execution of some instruction combinations. In these cases, the affected instructions are executed serially. This, of course, reduces the performance of a superscalar machine.

Superscalar computers suffer from disadvantages which it is desirable to minimize. A concrete amount of time is consumed in deciding at instruction execution time which instructions can be executed in parallel. This time cannot be readily masked by overlapping with other machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Also, the parallel execution decision must be repeated each time the same instructions are to be executed.

In extending the useful lifetime of existing scalar computers, every means of accelerating execution is vital. However, acceleration by means of reduced instruction set architecture, complex instruction set architecture, or traditional superscalar techniques is potentially too costly or too disadvantageous to consider for an existing scalar machine.

It would be preferred to accelerate the speed of execution of such a computer by parallel, or concurrent, execution of instructions in an existing instruction set without requiring change of the instruction set, change of machine architecture, or extension of the time required for instruction execution.

SUMMARY OF THE INVENTION

It is to this object that the inventions which are here stated are addressed as a system which enables original programs to be processed as parallel and single instructions as fits the original program functions which are implemented for execution by a machine capable of instruction-level parallel processing. We have provided a way for existing programs written in existing high level languages or existing assembly language programs to be processed by a preprocessor which can identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions.

The instruction processor system provides compounding decoding for a series of base instructions of a scalar machine, generates a series of compound instructions, provides for fetching of the compound instructions and for decoding the fetched compound instructions and necessary single instructions, and provides a compound instruction program which preserves intact the scalar execution of the base instructions of a scalar machine when the program with compound instructions is executed on the system. This system also provides a mechanism for branching in the middle of a compounding instruction. For branching in the middle of a compound instruction, compound instructions are examined for appended control bits and a nullification mechanism responsive to the control bits is provided for nullifying the execution of member units of the branched-to or subsequent compound instructions if their execution would cause incorrect program behavior.

The instruction processor system of the inventions described herein provides decoding of compound instructions created from a series of base instructions of a scalar machine, the processor generating a series of compound instructions with instruction text having appended control bits enabling the execution of the compound instruction in said instruction processor with a compounding facility which is provided in the system. The system fetches and decodes compound instructions which can be executed as compounded and single instructions by the functional instruction units of the instruction processor while preserving intact the scalar execution of the base instructions of a scalar machine which were originally in storage. The resultant series of compounded instructions generally executes in a faster manner than the original format due to the parallel nature of the compounded instruction stream which is executed.

For a better understanding of the invention, together with its advantages and features, reference be had to the co-pending applications for some detailed background. Further, specifically as to the improvements described herein reference should be made to the following description and the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a high-level schematic diagram of a computing system which is capable of compounding instructions in a sequence of scalar instructions for concurrent execution.

FIG. 2 is a timing diagram for a uni-processor implementation showing the parallel execution of certain instructions which have been selectively grouped in a compound instruction stream.

FIG. 3 is a block diagram of a hierarchical memory organization in a scalable compound instruction set machine with in-memory processing shown as an alternative preferred embodiment for the operational environment.

FIG. 4-A is a high level schematic diagram of the process which will be described pictorially for providing means for processing existing programs to identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions, while

FIG. 4-B illustrates the preferred operational environment of the inventions and the inventions' location in the environment.

FIG. 5 shows a path taken by a program from original program code to actual execution.

FIG. 6 is a flow diagram showing generation of a compound instruction set program from an assembly language program; while

FIG. 7 is a flow diagram showing execution of a compound instruction set. FIG. 8 illustrates a situation where a branch target may be at the beginning or middle of a compound instruction.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Background

In co-pending patent application Ser. No. 07/519,382, a scalable compound instruction set machine (SCISM) architecture method is proposed in which instruction level parallelism is achieved by statically analyzing a sequence of scalar instruction at a time prior to instruction execution to generate compound instructions formed by grouping adjacent instructions in a sequence which are capable of parallel execution. That system is also disclosed here. Relevant control information in the form of compounding tags is added to the instruction stream to indicate where a compound instruction starts, as well as to indicate the number of instructions which are incorporated into a compound instruction. Relatedly, when used herein, the term "compounding" refers to the grouping of instructions contained in a sequence of instructions, the grouping being for the purpose of concurrent or parallel execution of the grouped instructions. At minimum, compounding is satisfied by "pairing" of two instructions for simultaneous execution. Preferably, compounded instructions are unaltered from the forms they have when presented for scalar execution.

In a digital computer system which includes a means for executing a plurality of instructions in parallel, a particularly advantageous embodiment of the invention of this application is based upon a memory architecture which provides for compounding of instructions prior to their issue and execution. This memory structure provides instructions to the CPU (central processing unit) of a computer. Typically, a hierarchical memory structure includes a high-speed cache storage containing currently accessed instructions, a medium speed main memory connected to the cache, and a low-speed, high-capacity auxiliary storage. Typically, the cache and main storage (referred to collectively as "real storage") contain instructions which can be directly referenced for execution. Access to instructions in the auxiliary storage is had through an input/output (I/O) adaptor connected between the main memory and the auxiliary storage.

In co-pending patent application Ser. No. 07/543,464, an in-memory preprocessor for a SCISM architecture is proposed in which an instruction compounding mechanism in real storage produces compounding tag information for a sequence of scalar instructions, the compounding tag information indicating instructions of the sequence which may be executed in parallel. The instruction compounding unit produces the compounding tag information as a page of instructions is being prefetched and stored in the main memory. Although the particular embodiment of that patent application teaches compounding of up to two instructions, it is contemplated that up to N instructions may be compounded for concurrent execution in a scalar computer.

In patent application Ser. No. 07/522,219, a compounding preprocessor for a cache in a hierarchical memory is disclosed in which the instruction compounding unit is located between the main memory and cache of a hierarchical storage organization. The instruction compounding unit produces compounding tag information for the instructions in a line of instructions being fetched into the cache from the main memory.

In these applications, instructions in a cache have accompanying compounding tags to which a plurality of parallel execution units respond by executing one or more of the instructions in a group of up to N instructions in parallel. In these related systems, including U.S. Ser. No. 07/543,458 filed Jul. 26, 1990, a MEMORY MANAGEMENT FOR SCALABLE COMPOUND INSTRUCTION SET MACHINES WITH IN-MEMORY COMPOUNDING as disclosed, by R. J. Eickemeyer et al, there is proposed in the context of a computer system capable of concurrently executing up to N instructions in a sequence of scalar instructions, the sequence including compounding tags which accompany the set of ordered scalar instructions and which are activated to indicate the instructions which are to be concurrently executed. There is a mechanism for managing the compounding tags of scalar instructions which are stored in the real storage of the computer system, and it includes a merging unit connected to the real memory for merging a modified instruction from the real memory with non-modified instructions in the real memory. A tag reduction unit connected to the merging unit and to the real memory deactivates the compounding tags of the modified instruction and N-1 instructions in the real memory with which the modified instruction could be compounded.

However, in all of these developments there is an unfulfilled need to determine in the processing of a set of instructions or program to be executed by a computer exactly which instructions may be combined into compound instructions, and in what manner there shall be accomplished the appending of control information to the instructions in the program to be executed. This need is addressed by the disclosure of U.S. Ser. No. 07/522,219, and U.S. Ser. No. 07/642,011.

Overview

Referring to FIG. 1 of the drawings, there is shown a representative embodiment of a portion of a digital computer system for a digital data processing system constructed in accordance with the present invention. This computer system is capable of executing two or more instructions in parallel. In order to support the concurrent execution of a group of two or more instructions, the computer system includes a plurality of functional units which operate in parallel in a concurrent manner; each on its own, is capable of processing one or more types of machine-level instructions. The system includes the capability of compounding instructions for parallel or concurrent execution. In this regard, "compounding" refers to the grouping of a plurality of instructions in a sequence of scalar instructions, wherein the size of the grouping is scalable from 1 to N. The sequence of scalar instructions could, for example, be drawn from an existing set of scalar instructions such as that used by the IBM System/370 products. However, it is understood that compounding is intended to facilitate the parallel issue and execution of instructions in all computer architectures that can be adapted to process multiple instructions per cycle.

When an instruction stream is compounded, adjacent scalar instructions are selectively grouped for the purpose of concurrent or parallel execution. In general, an instruction compounding facility will look for classes of instructions that may be executed in parallel. When compatible sequences of instructions are found, a compound instruction is created.

Compounding techniques have been discussed in other applications. Reference is given to patent application Ser. No. 07/519,384 (IBM Docket EN9-90-020), filed May 4, 1990, and U.S. patent application Ser. No. 07/519,382entitled GENERAL PURPOSE COMPOUNDING TECHNIQUE FOR INSTRUCTION-LEVEL PARALLEL PROCESSORS, filed May 4, 1990, for an understanding of compounding generally. An illustration of an instruction compounding unit for pairwise compounding is given in U.S. patent application Ser. No. 07/543,464, filed Jun. 26, 1990.

Instruction Compounding and Execution

As is generally shown in FIG. 1, an instruction compounding unit 20 takes a stream of binary scalar instructions 21 and selectively groups some of the adjacent scalar instructions to form encoded compound instructions. A resulting compounded instruction stream 22, therefore, provides scalar instructions to be executed singly or in compound instructions formed by groups of scalar instructions to be executed in parallel. When a scalar instruction is presented to an instruction processing unit 24, it is routed to the appropriate one of a plurality of execution units for serial execution. When a compound instruction is presented to the instruction processing unit 24, its scalar components are each routed to an appropriate execution unit for simultaneous parallel execution. Typical functional units include, but are not limited to, an arithmetic logic unit (ALU) 26, 28 a floating point arithmetic unit (FP) 30 and a storage address generation unit (AU) 32.

Referring now to FIG. 2, compounding can be implemented in a uniprocessor environment where each functional unit executes a scalar instruction (S) or, alternatively, a compound instruction (CS). As shown in the drawing, an instruction stream 33 containing a sequence of scalar and compounded scalar instructions has control bits or tags (T) associated with each compound instruction. Thus, a first scalar instruction 34 could be executed singly by functional unit A in cycle 1; a triplet compound instruction 36 identified by tag T3 could have its 3 compounded scalar instructions executed in parallel by functional units A, C, and D in cycle 2; another compound instruction 38 identified by tag T2 could have its pair of compounded scalar instructions executed in parallel by functional units A and B in cycle 3; a second scalar instruction 40 could be executed singly by functional unit C in cycle 4; a large group compound instruction 42 could have its 4 compounded scalar instructions executed in parallel by functional units A-D in cycle 5; and a third scalar instruction 44 could be executed singly by functional unit A in cycle 6.

One example of a computer architecture which can be adapted for handling compound instructions is an IBM System/370 instruction level architecture in which multiple scalar instructions can be issued for execution in each machine cycle, in accordance with FIG. 2. In this context, a machine cycle refers to a single pipeline stage required to execute a scalar instruction.

Generally, it is useful to provide for compounding at a time prior to instruction issue so that the process can be done once for an instruction or instructions that may be executed many times. It has been proposed to locate the instruction compounding functions in the real memory of a computer system in order to implement compounding in hardware, after compile time, yet prior to instruction issue. Such compounding is considered to be a preferred alternative to other alternatives described herein and referred to as "in-memory compounding" which can be illustrated with reference to U.S. patent application Ser. No. 07/522,219, filed May 10, 1990, and U.S. patent application Ser. No. 07/543,464, filed Jun. 26, 1990, and FIG. 3 hereof. Memory management, as described herein by way of background, is also described in U.S. patent application Ser. No. 07/543,458 entitled MEMORY MANAGEMENT FOR SCALABLE COMPOUND INSTRUCTION SET MACHINES WITH IN-MEMORY COMPOUNDING, filed Jun. 26, 1990.

Generally, in-memory compounding is illustrated in FIG. 3. In FIG. 3, a hierarchical memory organization includes an I/O adaptor 40 which interfaces with auxiliary storage devices and with a computer real memory. The real memory of the organization includes a medium speed, relatively high capacity main memory 46 and a high-speed, relatively low capacity instruction cache 48. (The main memory and cache collectively are referred to herein as "real memory", "real storage", or, simply "memory".) A stream of instructions is brought in from auxiliary storage devices by way of the I/O adaptor 40, and stored in blocks called "pages" in the main memory 46. Sets of contiguous instructions called "lines" are moved from the main memory 46 to the instruction cache 48 where they are available for high-speed reference for processing by the instruction fetch and issue unit 50. Instructions which are fetched from the cache are issued, decoded at 52, and passed to the functional units 56, 58, . . . , 60 for execution.

During execution when reference is made to an instruction which is in the program, the instruction's address is provided to a cache management unit 62 which uses the address to fetch one or more instructions, including the addressed instruction, from the instruction cache 48 into the queue in the unit 50. If the addressed instruction is in the cache, a cache "hit" occurs. Otherwise, a cache "miss" occurs. A cache miss will cause the cache management unit 62 to send the line address of the requested instruction to a group of storage management functions 64. These functions can include, for example, real storage management functions which use the line address provided by the cache management unit 62 to determine whether the page containing the addressed line is in the main memory 46. If the page is in main memory, the real storage management will use the line address to transfer a line containing the missing instruction from the main memory 46 to the instruction cache 48. If the line containing the requested instruction is not in the main memory, the operating system will activate another storage management function, providing it with the identification of the page containing the needed line. Such a storage management function will send to the I/O adaptor 40 an address identifying the page containing the line. The I/O adaptor 40 will bring the page from auxiliary storage and provide it to the main memory 46. To make room for the fetched page, the storage management function selects a page in the main memory 46 to be replaced by the fetched page. In SCISM architecture, it is contemplated that the replaced page is returned to auxiliary storage through the I/O adaptor without compounding tag information. In this manner, those instructions most likely to be immediately required during execution of an instruction sequence are adjacent to the functional units in the instruction cache 48. The hierarchical memory organization provides the capability for fast retrieval of instructions that are required but not in the cache.

In the context of SCISM architecture, in-memory instruction compounding can be provided by an instruction compounding unit 70 which is located functionally between the I/O adaptor 40 and the main memory 46 so that compounding of the scalar instruction stream can take place at the input to, or in, the main memory 46. In this location, instructions can be compounded during an ongoing page fetch.

Alternatively, the instruction compounding unit can occupy the position 72 between the main memory 46 and the instruction cache 48 and compound instructions are formed line-by-line as they are fetched into the instruction cache 48, as may be considered a preferred embodiment.

Example of Compounding

The particular technique for compounding is a matter of design choice. However, for purposes of illustration, one technique for creating compound instructions formed from adjacent scalar instructions can be stated, as illustrated in the aforementioned In Memory Preprocessor application U.S. Ser. No. 07/543,458. By way of example, instructions may occupy 6 bytes (3 half words), 4 bytes (2 half words), or 2 bytes (1 half word) of text. For this example, the rule for compounding a set of instructions which includes variable instruction lengths provides that all instructions which are 2 bytes or 4 bytes long are compoundable with each other. That is, a 2 byte instruction is capable of parallel execution in this particular example with another 2 byte or another 4 byte instruction and a 4 byte instruction is capable of parallel execution with another 2 byte or another 4 byte instruction. The rule further provides that all instructions which are 6 bytes long are not compoundable. Thus, a 6 byte instruction is only capable of execution singly by itself. Of course, compounding is not limited to these exemplary rules, but can embrace a plurality of rules which define the criteria for parallel execution of existing instructions in a specific configuration for a given computer architecture.

The instruction set used for this example is taken from the System/370 architecture. By examining the OP code for each instruction, the length of each instruction can be determined from an instruction length code (ILC) in the op code. The instruction's type is further defined in other op code bits. Once the type and length of the instruction is determined, a compounding tag containing tag bits is then generated for that specific instruction to denote whether it is to be compounded with one or more other instructions for parallel execution, or to be executed singly by itself.

In the tag formal of this example (which is not limiting), if 2 adjacent instructions can be compounded, the tag bits, which are generated in memory, are "1" for the first compounded instruction and "zero" for the second compounded instruction. However, if the first and second instructions cannot be compounded, in this first tag format the tag bit for the first instruction is "zero" and the second and third instructions are then considered for compounding. Once an instruction byte stream has been processed in accordance with the chosen compounding technique and the compounding bits encoded for various scalar instructions, more optimum results for achieving parallel execution may be obtained by using a bigger window for looking at larger groups of instructions and then picking the best combination of N instructions for compounding.

Compounding Programs

However, taking the above by way of example of the problem solved here, it may be considered that we have generally a need to provide the system and process illustrated generally by FIG. 4-A. Existing programs written in existing high level languages, as described with reference to FIG. 5, or existing assembly language programs to be processed, as described with reference to FIG. 6, need to be processed, and we have provided a system which has the capability to identify sequences of instructions which can be executed as a single compound instruction in a computer designed to execute compound instructions.

Turning now to FIG. 4-A, it will be seen that there is illustrated pictorially the sequence in which a program is provided as an input to a compounding facility that produces a compound instruction program based on a set of rules which reflect both the system and hardware architecture. The preferred compounding facility is illustrated by U.S. Ser. No. 07/642,011 filed Jan. 15, 1991. These rules are hereafter referred to as compounding rules. The program produced by the compounding facility can then be executed directly by a compound instruction execution engine generally illustrated by FIG. 7.

However, FIG. 5 shows a typical path taken by a program from higher level source code to actual execution, and may also be considered as one of the possible organizations suggested by FIG. 4-A. An alternative related to assembly level programs is discussed with respect to FIG. 6.

As will have been appreciated, referring to FIG. 5, there are many possible locations in a computer system where compounding may occur, both in software and in hardware. Each has unique advantages and disadvantages. As shown in FIG. 5, there are various stages that a program typically takes from source code to actual execution. During the compilation phase, a source program is translated into machine code and stored on a disk 46. During the execution phase the program is read from the disk 46 and loaded into a main memory 48 of a particular computer system configuration 50 where the instructions are executed by appropriate instruction processing units 52, 54, 56. Compounding could take place anywhere along this path. In general as the compounding facility is located closer to an instruction processing unit (IPU), the time constraints become more stringent. As the compounding facility is located further from the IPU, more instructions can be examined in a large sized instruction stream window to determine the best grouping for compounding for increasing execution performance.

Compounding an Assembly-language Program

The flow diagram of FIG. 6 shows the generation of a compound instruction set program from an assembly language program in accordance with a set of customized compounding rules 58 which reflect both the system and hardware architecture. The assembly language program is provided as an input to a software compounding facility 59 which parses the program in m1-, m2-, . . . m.sub.n -instruction blocks (m=1,2, . . .) and produces the compound instruction program. Successive blocks of instructions are analyzed by the compounding facility 59. Blocks are numbered from 1 to n. Member instructions of block n are indicated by 1.sub.m.sup.n, where m ranges from 1 up to the number of instructions in the block. The number of instructions in each block 60, 62, 64 in the byte stream which contains the group of instructions considered together for compounding depends on the design of the compounding facility and may vary from block to block.

As shown in FIG. 6, this particular compounding facility is designed to consider two-way compounding for "m" number of instructions in each block. The primary first step is to consider if the first and second instructions constitute a compoundable pair, and then if the second and third constitute a compoundable pair, and then if the third and fourth constitute a compoundable pair, all the way to the end of the block. Once the various possible compoundable pairs C1-C5 have been identified, the compounding facility can select the preferred of compounded instructions and use flags or identifier bits to identify the optimum grouping of compound instructions.

If there is no optimum grouping, all of the compoundable adjacent scalar instructions can be identified so that a branch to a target located amongst various compound instructions can exploit any of the compounded pairs which are encountered. Where multiple compounding units are available, multiple successive blocks in the instruction stream could be compounded at the same time.

Executing a Compounded Program

The flow diagram of FIG. 7 shows the execution of a compound instruction set program which has been generated by a hardware preprocessor 66 or a software preprocessor 67. A byte stream having compound instructions flows into a compound instruction (CI) cache 68 that serves as a storage buffer providing fast access to compound instructions. CI issue logic 69 fetches compound instructions from the CI Cache and issues their individual compounded instructions to the appropriate functional units for parallel execution.

It is to be emphasized that instruction execution units (CI EU) 71 such as ALU's in a compound instruction computer system are capable of executing either scalar instructions one at a time by themselves or alternatively compounded scalar instructions in parallel with other compounded scalar instructions. Also, such parallel execution can be done in different types of execution units such as ALU's, floating point (FP) units 73, storage address-generation units (AU) 75 or in a plurality of the same type of units (FP1, FP2, etc) in accordance with the computer architecture and the specific computer system configuration.

Control Information In Compounding

In the first tag format discussed in the aforementioned example, compounding information is added to the instruction stream as one bit for every two bytes of text (instructions and data).

In general, a tag containing control information can be added to each instruction in the compounded byte stream--that is, to each non-compounded scalar instruction as well as to each compounded scalar instruction included in a pair, triplet, or larger compounded group. As used herein, identifier bits refers to that part of the tag used specifically to identify and differentiate those compounded scalar instructions forming a compounded group from the remaining non-compounded scalar instructions which remain in the compound instruction program and that when fetched are executed singly.

Where more than two scalar instructions can be grouped together to form a compound instruction, additional identifier bits may be advantageous. The minimum number of identifier bits needed to indicate the specific number of scalar instructions actually compounded is the logarithm to the base 2 (rounded up to the nearest whole number) of the maximum number of scalar instructions that can be grouped to form a compound instruction. For example, if the maximum is two, then one identifier bit is needed for each compound instruction. If the maximum is three or four, then two identifier bits are needed for each compound instruction. If the maximum is five, six, seven or eight, then three identifier bits are needed for each compound instruction.

A second tag format, implementing this encoding scheme, is shown below in Table 1:

TABLE 1

Identifier Bits	Encoded meaning	Total # Compounded
00	This instructions not compounded with its following instruction	none
01	This instuction is compounded with its one following instruction	two
10	This instruction is compounded with its two following instructions	three
11	This instruction is compounded with its three following instructions	four

Assuming instruction alignment is such that each halfword of text needs a tag, it can be appreciated that the IPU ignores all but the tag for the first instruction when an instruction stream is fetched for execution. In other words, a half word of fetched text is examined to determine if it begins a compound instruction by checking its identifier bits. In accordance with Table 1, if it is not the beginning of a compound instruction, its identifier bits are zero. If the half word is the beginning of a compound instruction containing two scalar instructions, the identifier bits are "1" for the first instruction and "0" for the second instruction. If the half word is the beginning of a compound instruction containing three scalar instructions, the identifier bits are "2" for the first instruction and "1" for the second instruction and "0" for the third instruction. In other words, the identifier bits for each half word identify whether or not this particular half word is the beginning of a compound instruction while at the same time indicating the number of instructions which make up the compounded group.

This method of encoding compound instructions assumes that if three instructions are compounded to form a triplet group, the second and third instructions are also compounded to form a pair group. In other words, if a branch to the second instruction in a triplet group occurs, the identifier bit "1" for the second instruction indicates that the second and third instruction will execute as a compounded pair in parallel, even though the first instruction in the triplet group was not executed.

It will be apparent to those skilled in the art that the present invention requires an instruction stream to be compounded only once for a particular computer system configuration, and thereafter any fetch of compounded instructions will also cause a fetch of the identifier bits associated therewith. This avoids the need for the inefficient last-minute determination and selection of certain scalar instructions for parallel execution that repeatedly occurs every time the same or different instructions are fetched for execution in the so-called super scalar machine.

Compounding Without Reference Points Between Instructions

It is straightforward to create compound instructions from an instruction stream if reference points are known that indicate where instructions begin. As used herein, a reference point means knowledge of which byte of text, which may contain instructions and/or data, is the first byte of an instruction. This knowledge could be obtained by some marking field or other indicator which provides information about the location of instruction boundaries. In many computer systems such a reference point is expressly known only by the compiler at compile time and only by the CPU when instructions are fetched. When compounding is done after compile time, a compiler could indicate with tags which bytes contain the first byte of an instruction and which contain data. This extra information results in a more efficient compounding facility since exact instruction locations are known. Of course, the compiler would differentiate between instructions and data in other ways in order to provide the compounding facility with specific information indicating instruction boundaries.

In a system with all 4-byte instructions aligned on a four byte boundary, one tag is associated with each four bytes of text. Similarly, if instructions can be aligned arbitrarily, a tag is needed for every byte of text. However, certain computer architectures allow instructions to be of variable length, and may further allow data and instructions to be intermixed, thus complicating the compounding process. Of course, at execution time, instruction boundaries must be known to allow proper execution. But since compounding is preferably done a sufficient time prior to instruction execution, a technique is needed to compound instructions without knowledge of where instructions start and without knowledge of which bytes are data. This technique needs to be applicable to all of the accepted types of architectures, including the RISC (Reduced Instruction Set Computers) architectures in which instructions are usually a constant length and are not intermixed with data. In connection with our invention, accomplishment of these tasks, which are generally described with reference to FIG. 4-A, is achieved with the assistance of a compound instruction execution engine of the kind generally described in U.S. Pat. No. 07/519,382 filed May 4, 1990 in an alternative environment.

Organization of Preferred System With Compounding Facility

Generally the preferred operating environment may be represented by the operational environment shown in FIG. 4-B. While the compounding facility may be a software entity, the preferred embodiment of the compounding facility may be implemented by an instruction compounding unit as described in detail in U.S. Pat. No. 07/642,011 filed Jan. 16, 1991. Referring to FIG. 4-B of the drawings, there is shown a representative embodiment of a portion of a digital computer system or digital data processing system constructed in accordance with the present invention with a cache management unit 144. This computer system is capable of processing two or more instructions in parallel. It includes a first storage mechanism for storing instructions and data to be processed in the form of a series of base instructions for a scalar machine. This storage mechanism is identified as higher-level storage 136. This storage (also "main memory") is a large capacity lower speed storage mechanism and may be, for example, a large capacity system storage unit or the lower portion of a comprehensive hierarchical storage system.

The computer system of FIG. 4-B also includes an instruction compounding facility or mechanism for receiving instructions from the higher level storage 136 and associating with these instructions compound information in the form of tags which indicate which of these instructions may be processed in parallel with one another. A suitable instruction compounding unit is represented by the instruction compounding unit 137. This instruction compounding unit 137 analyzes the incoming instructions for determining which ones may be processed in parallel. Furthermore, instruction compounding unit 137 produces for these analyzed instructions tag bits which indicate which instructions may be processed in parallel with one another and which ones may not be processed in parallel with one another but must be processed singly.

The FIG. 4-B system further includes a second storage mechanism coupled to the instruction compounding unit 137 for receiving and storing the analyzed instructions and their associated tag fields so that these stored compounded instructions may be fetched. This second or further storage mechanism is represented by compound instruction cache 138. The cache 138 is a smaller capacity, higher speed storage mechanism of the kind commonly used for improving the performance rate of a computer system by reducing the frequency of having to access the lower speed storage mechanism 136.

The FIG. 4-B system further includes a plurality of functional instruction processing units 139, 140, 141, et cetera. These functional units 139-141 operate in parallel with one another in a concurrent manner and each, on its own, is capable of processing one or more types of machine-level instructions. Examples of functional units which may be used are: a general purpose arithmetic and logic unit (ALU), an address generation type ALU, a data dependency collapsing ALU (of the preferred type shown in co-pending U.S. Ser. No. 07/504,910 filed Apr. 4, 1990), a branch instruction processing unit, a data shifter unit, a floating point processing unit, and so forth. A given computer system may include two or more or some of the possible functional units. For example, a given computer system may include two or more general purpose ALUs. Also, no given computer system need include each and every one of these different types of functional units. The particular configuration will depend on the nature of the particular computer system being considered.

The computer system of FIG. 4-B also includes an instruction fetch and issue mechanism coupled to compound instruction cache 138 for supplying adjacent instructions stored therein to different ones of the functional instruction processing units 139-141 when the instruction tag bits indicate that they may be processed in parallel. This mechanism also provides single instructions to individual functional units when their tag bits indicate parallel execution is not possible and they must be processed singly. This mechanism is represented by instruction fetch and issue unit 142. Fetch and issue unit 142 fetches instructions form the cache 138 and examines the tag bits and instruction operation code (opcode) fields, performing a decode function, and based upon such examinations and other such pipeline control signals as may be relevant, sends the instruction under consideration to the appropriate ones of the functional units 138-141.

In the context of SCISM architecture, in-cache instruction compounding is provided by the instruction compounding unit 137 so that compounding of each cache line can take place at the input to the compound instruction cache 138. Thus, as each cache line is fetched from the main memory 136 into the cache 138, the line is analyzed for compounding in the unit 137 and passed, with compounding information tag bits, for storage in the compound instruction cache 138.

Prior to caching, the line is compounded in the instruction compounding unit 137 which generates a set of tag bits. These tag bits may be appended directly to the instructions with which they are associated. Or they may be provided in parallel with the instructions themselves. In any case, the bits are provided for storage together with their line of instructions in the cache 138. As needed, the compounded instruction in the cache 138 is fetched together with its tag bit information by the instruction and issue unit 142. Instruction fetch and issue unit 142 and compound instruction cache 138 are designed such that a maximum length compound instruction (as defined by the system) may be fetched from compound instruction cache 138 and issued to the functional units 139, 140, 141, and so on. As instructions are received by the fetch and issue unit 142, their tag bits are examined to determine by decoding examination whether they may be processed in parallel and the opcode fields are examined to determine which of the available functional units is most appropriate for their processing. If the tag bits indicate that two or more of the instructions are suitable for processing in parallel, then they are sent to the appropriate ones in the functional units in accordance with the codings of their opcode fields. Such instructions are then processed concurrently with one another by their respective functional units.

When an instruction is encountered that is not suitable for parallel processing, it is sent to the appropriate functional unit as determined by its opcode and it is thereupon processed alone and singly by itself in the selected functional unit.

In the ideal case, where plural instructions are always being processed in parallel, the instruction execution rate of the computer system would be N times as great as for the case where instructions are executed one at a time, with N being the number of instructions in the groups which are being processed in parallel.

Further Control Information (Tag) in Compounding

As stated previously, the control field or tag contains information delimiting compound instruction boundaries, but may contain as much additional information as is deemed efficacious for a particular implementation.

For example, in a third tag format, a control field (tag) might be defined as an 8-bit field,

t.sub.0 t.sub.1 t.sub.2 t.sub.3 t.sub.4 t.sub.5 t.sub.6 t.sub.7
with the bits defined as follows

Bit	Function
t.sub.0	If 1, this instruction marks the beginning of a compound instruction.
t.sub.1	If 1, then execute two compound instructions in parallel
t.sub.2	If 1, then this compound instruction has more than one execution cycle.
t.sub.3	If 1, then suspend pipelining.
t.sub.4	If instruction is a branch, then if this bit is 1, the branch is predicted to be taken.
t.sub.5	If 1, then this instruction has a storage interlock from a previous compound instruction.
t.sub.6	If 1, then enable dynamic instruction issue.
t.sub.7	If 1, then this instruction uses an ALU.

Classifying Instructions for Compounding

In general, the compounding facility will look for classes of instructions that may be executed in parallel, and ensure that no interlocks between members of a compound instruction exist that cannot be handled by the hardware. When compatible sequences of instructions are found, a compound instruction is created. For example, the System/370 architecture might be partitioned into the following classes:

1. RR-format loads, logicals, arithmetics, compares

LCR--Load Complement

LPR--Load Positive

LNR--Load Negative

LR--Load Register

LTR--Load and Test

NR--AND

OR--OR

XR--Exclusive OR

AR--Add

SR--Subtract

ALR--Add Logical

SLR--Subtract Logical

CLR--Compare Logical

CR--Compare

2. RS-format shifts (no storage access)

SRL--Shift Right Logical

SLL--Shift Left Logical

SRA--Shift Right Arithmetic

SLA--Shift Left Arithmetic

SRDL--Shift Right Logical

SLDL--Shift Left Logical

SRDA--Shift Right Arithmetic

SLDA--Shift Left Arithmetic

3. Branches--on count and index

BCT--Branch on Count (RX-format)

BCTR--Branch on Count (RR-format)

BXH--Branch on Index High (RS-format)

BXLE--Branch on Index Low (RS-format)

4. Branches--on condition

BC--Branch on Condition (RX-format)

BCR--Branch on Condition (RR-format)

5. Branches--and link

BAL--Branch and Link (RX-format)

BALR--Branch and Link (RR-format)

BAS--Branch and Save (RX-format)

BASR--Branch and Save (RR-format)

6. Stores

STCM--Store Characters Under Mask (0-4-byte store, RS-format)

MVI--Move Immediate (1 byte, SI-format)

ST--Store (4 bytes)

STC--Store Character (1 byte)

STH--Store Half (2 bytes)

7. Loads

LH--Load Half (2 bytes)

L--Load (4 bytes)

8. LA--Load Address

9. RX-format arithmetics, logicals, inserts, compares

A--Add

AH--Add Half

AL--Add Logical

N--AND

O--OR

S--Subtract

SH--Subtract Half

SL--Subtract Logical

X--Exclusive OR

IC--Insert Character

ICM--Insert Characters Under Mask (0-to 4-byte fetch)

C--Compare

CH--Compare Half

CL--Compare Logical

CLI--Compare Logical Immediate

CLM--Compare Logical Character Under Mask

10. TM--Test Under Mask

The rest of the System/370 instructions are not considered to be compounded for execution in this invention. This does not preclude them from being compounded on a future compound instruction execution engine.

One of the most common sequences in programs is to execute an instruction of the TM or RX-format compare class, the result of which is used to control the execution of a BRANCH-on-condition type instruction which immediately follows. Performance can be improved by executing the COMPARE and the BRANCH instructions in parallel, and this is sometimes done dynamically in high performance instruction processors. Some difficulty lies in quickly identifying all the various members of the COMPARE class of instructions and all the members of the BRANCH class of instructions in a typical architecture during the instruction decoding process. This difficulty is avoided by the invention, because the analysis of all the members of the classes are accomplished ahead of time and a compound instruction which is guaranteed to work is created.

Many classes of instructions may be executed in parallel, depending on how the hardware is designed. In addition to the COMPARE and BRANCH compound instruction described above, other compound instructions can be envisioned, such as LOADS and RR-format instructions, BRANCH and LOAD ADDRESS, etc. A compound instruction may even include multiple instructions from the same class, for example, RR-format arithmetic, if the processor has the necessary execution units.

Optimizing the Compounding of Instructions

In any realizable instruction processor, there will be an upper limit to the number of instructions that can comprise a compound instruction. This upper limit, m, must be specified to the compounding facility which is creating the executable instructions by generating compound instructions, so that it can generate compound instructions no longer than the maximum capability of the underlying hardware. Note that m is strictly a consequence of the hardware implementation; it does not limit the scope of instructions that may be analyzed for compounding in a given code sequence by the software. In general, the broader the scope of the analysis, the greater the parallelism achieved will be, as more advantageous compoundings are recognized by the compounding facility. To illustrate, consider the sequence:

X1; any compoundable instruction

X2; any compoundable instruction

LOAD R1, (X); load R1 from memory location X

ADD R3, R1; R3=R3+R1

SUB R1, R2; R1=R1-R2

COMP R1, R3; compare R1 with R3

X3; any compoundable instruction

X4; any compoundable instruction.

If the hardware-imposed upper limit on compounding is m=2, then there are a number of ways to compound this sequence of instructions depending on the scope of the compounding facility. If the scope were equal to four, then the compounding facility would produce the pairings <-X1> <X2 LOAD> <ADD SUB> <COMP X3> <X4->, relieving completely the hazards between the LOAD and the ADD and the SUB and the COMP. On the other hand, a superscalar machine with m=2, which pairs instructions in its instruction issue logic on strictly a first-in-first-out basis, would produce the pairings<X1 X2> <LOAD ADD> <SUB COMP> <X3 X4>, incurring the full penalty of the interlocking instructions. Unfortunately, the LOAD and ADD cannot be executed in parallel because ADD requires the result of the load. Similarly, the SUB and COMP cannot execute in parallel. Thus no performance improvement is gained.

Branching and Compound Instructions

The systems described provide a method of generating a compound instruction program and further suggest a variety of methods for maintaining this association of compounding information (tags) with instruction text in a relatively permanent manner. But since there can be no guarantee that a branch into the middle of a compound instruction will not occur, what is not thus far apparent is how this compounding information can be maintained, and further, and more importantly, how correct program behavior can be maintained, in the presence of branches to arbitrary locations. This problem is in fact solved by instruction fetch and issue unit 142 of FIG. 4-B, as will be explained. First, assume a fourth definition of the compounding tag, T, such that T=1 if the associated instruction either begins a compound instruction or is an instruction to be executed singly. Further, T=0 for member instructions of a compound instruction other than the first. Then, assuming that instruction fetch and issue unit 142 can fetch a maximum-length compound instruction (as stated above), when fetching instructions in response to a taken branch, it always fetches a block of text equal to the maximum length compound instruction beginning at exactly the branch target address, and then executes all instructions fetched with T=0 as a compound instruction, up to the point in the fetched text where an instruction is encountered having T=1, indicating the beginning of the next compound instruction. If the T bit of the branch target is 1, then it is the beginning of a compound instruction and may be executed directly. FIG. 8 illustrates this situation. The notation used in FIG. 8 is I.sub.m.sup.n where n is the compound instruction number, and m is the number of the instruction within compound instruction n. The allowable range for m is from 1 to the maximum length of a compound instruction, which, in this example, is assumed to be three. Instruction 2 of compound instruction 1 is a branch instruction, that, in this example, will have two possible target paths, "a" and "b". The "a" path branches to the middle (I.sub.2) of compound instruction j, while the "b" path branches to the first instruction of compound instruction j. If the branch were to follow path "a", the hardware fetches the maximum length compound instruction, that is, three instructions, then executes I.sub.2.sup.j and I.sub.3.sup.j as a compound instruction. The remainder of the fetch, namely I.sub.1.sup.k is recognized to be the beginning of either a new compound or single instruction (from its having T=1) and is held in reserve while the next block of text is is fetched for subsequent execution. If the branch instruction took the "b" path to the first instruction of compound instruction j, the hardware would again fetch a block of text equal in length to the maximum length compound instruction, yielding, in this case, a complete compound instruction, namely I.sub.1.sup.j I.sub.2.sup.j and I.sub.3.sup.j. Execution of that compound instruction proceeds directly.

This technique is equally applicable to other tag formats, such as the second tag format discussed above.

Note in FIG. 8 that branch instruction B.sub.2.sup.1 is compounded with the instructions following it, but since the branch is taken, the prefetched text following the branch, whether it is comprised of member units of the compound instruction containing the branch or subsequent single or compound instructions or in fact data (in systems that allowdata and instructions to intermix), cannot be allowed to update the machine state if correct program behavior is to be maintained. Further, fetching the target of a branch instruction may result in the fetching of a compound instruction containing a second branch that too can be followed by text that can not be allowed to update the machine state should the second branch be taken. This branch-to-branch sequence may continue ad infinitum. In both cases, the solution for maintaining correct program behavior is to prevent instructions that should not have been executed from updating the machine state. That is, the execution results of prefetched text sequentially following the taken branch must be nullified. Algorithmically, this may be stated as follows:

1. Begin the execution of the compound instruction, including any branch instruction within the compound instruction.

2. If the compound instruction does in fact contain a branch instruction, do not permit any instructions following the branch, be they in the same compound instruction as the branch, or in subsequent compound or single instructions, to update the machine state.

3. If the branch is taken, nullify the execution results of any instructions sequentially following the taken branch. Do not, however, nullify the execution of the branch target.

While we have described our inventions in their preferred embodiment and alternative embodiments, various modifications may be made, both now and in the future, as those skilled in the art will appreciate upon the understanding of our disclosed inventions. Such modifications and future improvements are to be understood to be intended to be within the scope of the appended claims which are to be construed to protect the rights of the inventors who first conceived of these inventions.

* * * * *

Images
View Cart Add to Cart
Hit List Previous Next Top
Home Quick Advanced Pat Num Help

United States Patent	***5,295,249***
Blaner , et al.	**March 15, 1994**

Compounding preprocessor for cache for identifying multiple instructions which may be executed in parallel

Abstract

A digital computer system capable of processing two or more computer instructions in parallel and having a cache storage unit for temporarily storing machine-level computer instructions in their journey from a higher-level storage unit of the computer system to the functional units which process the instructions. The computer system includes an instruction compounding unit located intermediate to the higher-level storage unit and the cache storage unit for analyzing the instructions and generating for to each instruction a compounding information which indicates whether or not that instruction may be processed in parallel with one or more neighboring instructions in the instruction stream. These tagged instructions are then stored in the cache unit with the compounding information. The computer system further includes a plurality of functional instruction processing units which operate in parallel with one another. The instructions supplied to these functional units are obtained from the cache storage unit. At instruction issue time, the compounding information for the instructions is examined and those instructions indicated for parallel processing are sent to different ones of the functional units in accordance with the codings of their operation code fields.

Inventors:	**Blaner; Bartholomew** (Newark Valley, NY); **Vassiliadis; Stamatis** (Vestal, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**642011**
Filed:	**January 15, 1991**

Current U.S. Class:	**712/213**; 712/24; 712/214
Intern'l Class:	G06F 009/38; G06F 013/00; G06F 015/16
Field of Search:	395/375,425,800,650

References Cited [Referenced By]

U.S. Patent Documents

4439828	Mar., 1984	Martin	364/200.
4476525	Oct., 1984	Ishii	395/375.
4847755	Jul., 1989	Morrison et al.	395/650.
4928226	May., 1990	Kamada et al.	395/375.
5027270	Jun., 1991	Riordan et al.	395/375.
5075844	Dec., 1991	Jardine et al.	395/375.

Other References

Acosta, R. D., et al, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al, the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al, ed (McGraw-Hill, 1982, pp. 276-292.
Capozzi, A. J., et al, "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al, "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K, et al, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining); A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M. D., et al, "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al, "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N. P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp. 63-76.
Colwell, R. P., et al, "A VLIW Architecture for a Trace Scheduling Compiler", IEEE

Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A. D., "Introduction to the CRISP Instruction Set Architecture", Proceedings of Compcon, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al, "Compiling for the CRISP Microprocessor", Proceedings of Compcon, Spring, 1987, pp. 96-100.
Hennessy, J., et al, "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W., et al, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al, "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1, Jan. 1984, pp. 6-22.
Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al, Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions On Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance"0 Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al, "A Study of Instruction Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun., 1983, pp. 132-137.
Vassiliadis, S., et al, "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard.

Primary Examiner: Bowler; Alyssa H.
Assistant Examiner: Shah; Alpesh M.
Attorney, Agent or Firm: Augspurger; Lynn L.

Parent Case Text

This application Is a continuation-in-part of U.S. patent application Ser. No. 07/519,384, filed May 4, 1990, now abandoned, and U.S. patent application Ser. No. 07/522,291, filed May 10, 1990, now U.S. Pat. No. 5,214,763, issued May 25, 1993.

Claims

What is claimed is:

1. In a digital computer system operably disposed to process two or more instructions in parallel, the combination comprising:

an instruction processing unit;

a main memory for storing object code instructions;

an instruction compounding unit coupled to said main memory for pre-processing a sequence of instructions selected from said main memory as a scalar instruction sequence;

a cache memory for storing said scalar instruction sequence of said object code instructions;

a fetch and issue mechanism for fetching and decoding needed object code instructions and for issuing them for execution by said instruction processing unit of the digital computer system during execution of a program by said digital computer system;

said instruction processing unit including a plurality of execution units which operate in parallel and in a concurrent manner for supporting parallel execution of a group of up to N instructions from a sequence of instructions to be executed by the digital computer system;

said instruction compounding unit having means for compounding of instructions for execution by said plurality of instruction units for grouping a plurality of instructions in said scalar instruction sequence wherein the size of the grouping is scalable from 1 to N and whereby the compounding of instructions leaves the object code of compounded instructions unaltered and takes a stream of instructions and selectively categorizes adjacent scalar instructions which would otherwise be executed singly for parallel execution and generates a compounded instruction stream as a sequence of tagged instructions and tag information associated with each of the instructions which indicates whether the scalar instruction can be executed singly or executed in parallel by said plurality of execution units as a group of object code instructions when the group of instructions is presented to said instruction processing unit after a fetch and issue of the group of instructions to said instruction processing unit by said fetch and issue mechanism;

said cache memory being a high speed cache memory coupled to said main memory by said instruction compounding unit for storing into the high speed cache memory said scalar instruction sequence as a compounded stream of categorized and tagged instructions and the associated tag information for the tagged instructions after pre-processing by the instruction compounding unit;

said fetch and issue unit mechanism being coupled to the cache memory for fetching from the cache memory said scalar instruction sequence as a sequence of adjacent tagged instructions stored therein as needed for program execution along with their associated tag information and for issuing the instructions to the appropriate instruction execution units when an examination of the tag information associated with an instruction fetched from the cache indicates that the tagged instruction may be processed concurrently, and also for issuing instructions to individual functional units when their associated tag information indicates that parallel execution is not possible, the fetch and issue mechanism examining the tag information and decoding instruction operation code fields of an instruction, and based upon such examination and decoding determining which of available functional units is most appropriate for their processing, and sending the instructions to appropriate ones of the functional units for execution.

2. The digital computer system of claim 1 wherein the digital computer system for processing instructions includes an arithmetic logic unit, a floating point unit, general purpose registers, and a data-dependency collapsing arithmentic unit.

3. The digital computer system of claim 1 wherein the system includes an arithmetic logic unit, a floating point unit, an address generation unit, a branch execution unit and a data-dependency collapsing arithmentic unit. The compounding instruction unit includes means for generating tag information for the categorizes of instructions which categories are determined by rules for instruction tags which define in which category an instruction associated with tag information belongs, the tag information being generated by circuits which analyze the instructions to be execution, and wherein the system includes a branch execution unit, and the fetch and issue unit includes means for determining from the tag information associated with one of the instructions that the instruction is a branch which should be processed by a branch execution unit and while a branch is being processed the address generation unit can calculate an address calculation for a store if a branch is taken.

4. The digital computer system of claim 1 wherein the compounding instruction unit includes means for generating tag information for the categorized instructions tags which indicates in which category an instruction associated with tag information belongs, the tag information being generated by analyzing an instruction to be categorized under a set of compounding rules which reflect the categorizes of those instructions which may be processed by the digital computer system and by generating values for the tag information being associated with the instruction after a determination is made by said means for compounding whether the instruction falls in a category which may be executed concurrently in parallel or which must be executed singly, said tag information having a value which indicates whether the associated instruction may be compounded with the next instruction of the group of instructions or must be executed singly such that the tag can identify a compounded instruction to the instruction processing unit after issue and during execution of the instruction.

5. The digital computer system according to claim 4 wherein said fetch and issue mechanism includes means for examining the tag information associated with adjacent instructions of a sequence of instructions to be executed, and if the tag information associated with the adjacent instructions indicates

that the adjacent instructions are suitable for processing in parallel for sending the suitable instructions to the appropriate ones of the functional unit in accordance with the codings of their operational codes for concurrent processing by their respective functional units, and for sending to a selected functional unit for processing alone and by itself in the selected functional unit when in the sequence of instructions examined by the fetch and issue mechanism an instruction or instruction sequence is encountered which is not suitable for concurrent execution.

6. The digital computer system according to claim 5 wherein the compounding instruction unit comparing operands of an instructions with a succeeding instruction of a group of instructions to determine in accordance with a set of rules for the particular digital computer system whether each of the instructions is in a category which can be compounded with the other instruction.

7. The digital computer system according to claim 6 wherein the rules for categorizing instructions for a particular digital computer system are programmable, with programmable changes providing flexibility as more functional units are added or subtracted, more or fewer types of compoundings are desires, or as the computing environment changes, with rule decisions made upon machine configuration.

8. The digital computer system according to claim 7 wherein the compounding of instructions which leaves the object code of compounded instructions unaltered maintains object code compatibility with previously implemented computer systems.

9. The digital computer system according to claim 1 wherein between the cache memory and the instruction fetch and issue mechanism the word size of the digital computer system is increased of that in main memory to accommodate the number of bits of the tags associated with the instructions.

10. The digital computer system according to clam 1 wherein the cache memory includes an instruction cache and a separate tag cache operating in parallel with the instruction cache, whereby groups of instructions and the tag information associated with each instruction is provided to the fetch and issue unit in parallel streams of associated information.

11. The digital computer system according to claim 1 wherein when a first instruction which is categorized as an instruction which may be compounded with a succeeding instruction and has associated tag information having a value which indicates that it is an instruction which may be compounded is followed in the sequence of instructions which are needed for program execution by a second instruction which has tag information also having a value that indicates that it is an instruction which may be compounded, then the first instruction and said second instruction upon examination by the fetch and issue mechanism are issued as instructions which are to be executed singly, but if the second instruction has tag information with a second value which indicates that the instruction has tag information with a second value which indicates that the instruction may be compounded with the first instruction, then the first and second instructions are issued for concurrent execution by the fetch and issue mechanism concurrently to appropriate functional units.

12. The digital computer system according to claim 1 wherein tag information associated with an instruction is generated by the compounding instruction unit for every half word in a cache line.

13. The digital computer system according to claim 1 wherein the instruction compounding unit means

for compounding instruction includes circuits which perform compounding analysis according to a worst case scenario when an instruction stream has variable length instructions intermixed with data and no reference to indicate where a first instruction of a cache line is.

14. The digital computer system according to claim 1 wherein the instruction compounding unit means for compounding instruction includes circuits for comparing instruction operational codes and operand and addressing registers for two instructions for determining whether any interlocks exists between the two instructions in the form of data dependency or address hazards.

15. The digital computer system according to claim 1 wherein an instruction sequence having in sequence instructions AR 2,3 and SR 4, 2 have instruction tag information generated which permits the fetch and issue unit to issue these instructions to two execution units, one of which execution unit calculating R2=R2+R3 for the AR instruction and a second interlock collapsing execution unit performing R4=R4-(R2+R3) as a 3-1 compounding operation.

16. The digital computer system according to clam 1 wherein the execution units include an execution unit for performing 3-1 compounding operations.

17. The digital computer system according to claim 1 wherein in-cache instruction compounding is provided by the instruction compounding unit for compounding of each cache line at the input to the compound instruction cache, such that as each cache line is fetched from the main memory into the cache the line is analyzed for compounding and passed with the compounding information for storage in the cache memory.

18. The digital computer system according to claim 1 wherein

auxiliary storage is provided for bringing a stream of instruction into main memory from auxiliary storages for storing in blocks in main memory, and means are provided for moving lines of continuous instructions from the main memory to said cache memory for compounded instructions where they are available for high speed reference for processing by the fetch and issue unit, the fetch and issue unit fetching instructions from the cache and decoding them, and then and dispatches the decoded instructions to the function unit for execution.

19. The digital computer system according to claim 1 wherein the number of bits used to indicate for the tag information the specific number of scalar instructions actually compounding is a whole number rounded up from a logarithm to the base two of the maximum number of scalar instructions that can be grouped to form a compound instruction during execution in the digital computer system.

20. The digital computer system according to claim 1 wherein tag bits of the tag information associated with a instruction has a different value determined after categorization of the instruction, one value indicating that the instruction may be compounded with a following instruction of the sequence, while another value indicates that the instruction under consideration is not compounded with the following instruction.

21. The digital computer system according to claim 1 wherein each functional unit being capable of

processing one or more types of machine-level instructions, and said function units include an arithmetic logic unit for executing an instruction in response to two operands, a floating point unit, a storage address generation unit and a data dependency collapsing logic unit.

22. The digital computer system according to claim 1 wherein tag information is comprised of a plurality of tag fields, each tag field being associated with a particular instruction analyzed by the instruction compounding mechanism.

23. The digital computer system according to claim 1 wherein a instruction compounding mechanism includes: an instruction register;

a plurality of rule-based instruction analyzer mechanisms, each rule-based instruction analyzer mechanism analyzing a particular pair of side-by-side instructions in the instruction register and producing a compoundability signal which indicates whether or not the two instructions in said pair may be processed in parallel; and

a tag generating mechanism responsive to the compoundability signals for generating the individual tag information fields for the different instructions in the instruction register.

24. The digital computer system according to claim 1 wherein the instruction compounding unit is asynchronous with the instruction execution.

25. The combination of claim 1 wherein associated tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a subsequent instruction of the sequence of instructions, or, specifies, if the instruction may not be executed in parallel with a preceding instruction of the sequence of instruction, such that the instruction must be executed as a single instruction.

26. The combination of claim 1 wherein associated tag information is produced for each one instruction of the group of instructions and specifies, if the one instruction may be executed in parallel with a next one instruction of the sequence of instructions, the number of subsequent instructions with which the instruction can be executed, or which specifies, if the instruction may not be executed in parallel with a preceding instruction of the sequence of instruction, such that the instruction must be executed as a single instruction.

Description

CROSS REFERENCE TO RELATED APPLICATIONS

The present U.S. patent application is related to the following co-pending U.S. patent application:

(1) U.S. application Ser. No. 07/519,382, filed May 4, 1990, now abandoned, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al;

(2) U.S. application Ser. No. 07/519,384, filed May 4, 1990, now abandoned, entitled "General Purpose Compound Apparatus For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al;

(3) U.S. application Ser. No. 07/504,910, filed Apr. 4, 1990, now U.S. Pat. No. 5,051,940 issued Sep. 24, 1991, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al;

(4) U.S. application Ser. No. 07/522, 291, now U.S. Pat. No. 5,214,763 issued May 25, 1993, entitled "Compounding Preprocessor for Cache", the inventors being Bartholomew Blaner et al;

(5) U.S. application Ser. No. 07/543,464, filed Jun. 26, 199, now abandoned, entitled "An In-Memory Preprocessor for a Scalable Compound Instruction Set Machine Processor", the inventors being Richard J. Eickemeyer et al;

(6) U.S. application Ser. No. 07/543,458, filed Jun 26, 1990, now U.S. Pat. No. 5,197,135 issued Mar. 23, 1993, entitled "Memory Management for Scalable Compound Instruction Set Machines with In-Memory Compounding", the inventors being Richard J. Eickemeyer et al; and

(7) U.S. application Ser. No. 07/619,868, filed Nov. 28, 1990, entitled "Overflow Determination for Three-Operand ALUS in a Scalable Compound Instruction Set Machine", the inventors being Stamatis Vassiliadis et al.

These co-pending applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y.

The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

TECHNICAL FIELD

This invention relates to digital computers and digital data processors and particularly to digital computers and data processors capable of processing two or more instructions in parallel.

BACKGROUND OF THE INVENTION

The performance of traditional computers which execute instructions one at a time in a sequential manner has improved significantly in the past largely due to improvements in circuit technology. Such on-at-a-time instruction execution computers are sometimes referred to as "scalar" computers or processors. As the circuit technology is pushed to its limits, computer designers have had to investigate other means to obtain significant performance improvements.

Recently, so-called "super scalar" computers have been proposed which attempt to increase performance by executing more than one instruction at a time from a single instruction stream. Such proposed super scalar machines typically decide at instruction execution time if a given number of instructions may be executed in parallel. Such decision is based on the operation codes (op codes) of the instructions and on data dependencies which may exist between adjacent instructions. The op codes determine the particular hardware components each of the instructions will utilize and, in general, it is not possible for two or more instructions to utilize the same hardware component at the same time nor to execute an instruction that depends on the results of a previous instruction (a data dependency). These hardware and data dependencies prevent the execution of some instruction combinations in parallel. In this case, instructions are instead executed by themselves in a non-parallel manner. This, of course, reduces the performance of a super scalar machine.

Proposed super scalar computers provide some improvement in performance but also have disadvantages which it would be desirable to minimize. For one thing, deciding at instruction execution time which instructions can be executed in parallel takes significant amount of time which cannot be very readily masked by overlapping it with other normal machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Another disadvantage is that the decision making must be repeated all over again each time the same instructions are to be executed a second or further time.

SUMMARY OF INVENTION

As discussed in co-pending U.S application Ser. No. 07/519,382, one of the attributes of a Scalable Compound Instruction Set Machine (SCISM) is performance of the parallel execution decision prior to execution time. In SCISM architecture, the decision to execute in parallel is made at an earlier point in the overall instruction handling process. For example, the decision can be made ahead of the instruction buffer in those machines which have instruction buffers or instruction stacks. For another example, the decision can be made ahead of the instruction cache in those machines which flow the instructions through a cache unit.

Another attribute of a SCISM machine is to record the results of the parallel execution decision making so that such results are available in the event that those same instructions are used a second or further time.

In one embodiment of the present invention, the recording of the parallel execution decision making is accomplished by generating information in the form of tags which accompany the individual instructions in an instruction stream. These tags tell whether the instructions can be executed in parallel or whether they need to be executed one at a time. This instruction tagging process is sometimes referred to herein as "compounding". It serves, in effect, to combine at least two individual instructions into a single compound instruction for parallel processing purposes.

In a particularly advantageous embodiment of the present invention, the computer is one which includes a cache storage mechanism for temporarily storing machine instructions in their journey from a higher-level storage unit of the computer to the instruction execution units of the computer. The compounding process is performed intermediate to the higher-level storage unit and the cache storage mechanism so that there is stored in the cache storage mechanism both instructions and compounding information. As is known, the use of a well-designed cache storage mechanism, in and of itself, serves to improve the overall performance of a computer. Further, the storing of the compounding information into the cache storage mechanism enables the information to be used over and over again so long as the instructions in question remain in the cache storage mechanism. As is known, instructions frequently remain in a cache long enough to be used more than once.

For a better understanding of the present invention, together with other and further advantages and features thereof, reference is made to the following description taken in connection with the accompanying drawings, the scope of the invention being pointed out in the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring to the drawings:

FIG. 1 illustrates the location of the invention in a stream of scalar instructions.

FIGS. 2A and 2B illustrate categorization of instructions in an exemplary instruction set.

FIG. 3 illustrates how an instruction stream is analyzed according to a set of rules establishing which instructions of which categories can be executed in parallel with instructions of other categories.

FIG. 4 illustrates the operational environment of the invention and the invention's location in the environment.

FIG. 5 illustrates the formats of instructions which are analyzed for parallel execution according to the invention.

FIG. 6A and 6B form a block diagram illustrating a compounding unit according to the invention which analyzes instructions for parallel execution according to a set of rules and generates information indicating the outcome of the analysis.

FIG. 7 is a partial block diagram illustrating how the instruction compounding unit of FIG. 6 analyzes two instructions.

FIGS. 8A, 8B, and 8C are timing diagrams which illustrate operation of the invention according to various conditions.

FIG. 9A and 9B form a logic diagram illustrating in greater detail a rule-based analysis component of the instruction compounding unit of FIG. 6.

FIG. 10 is a block diagram of an industrial application of the invention.

FIG. 11 is a representation of a block of instructions analyzed by the instruction compounding unit of FIG. 6 together with an information vector indicating the results of the analysis.

FIGS. 12A and 12B are schematic diagrams illustrating cache storage of instruction blocks and accompanying compounding information.

FIG. 13 illustrates a fragment of an instruction stream analyzed according to the invention with an accompanying information vector containing the results of the analysis.

FIG. 14 is a chart which shows how the instructions of FIG. 13 are executed in response to the accompanying analysis information.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Instruction Compounding

Referring to FIG. 1 of the drawings, there is shown a representative embodiment of a portion of a digital computer system or a digital data processing system constructed in accordance with the present invention. The illustrated computer system is capable of executing two or more instructions in parallel. The system includes the capability of compounding instructions for parallel execution. In this regard, the compounding process refers to the grouping of a plurality of instructions in a scalar instruction sequence for parallel execution, wherein the size of the grouping is scalable from 1 to N. Preferably, the sequence of scalar instructions are drawn from an existing set of scalar instructions such as that used in the IBM System/370 products. The compounding process described herein leaves the object code of compounded instructions unaltered, thereby maintaining compatability with previously-implemented computer systems.

In order to support the parallel execution of a group of up to N instructions, the computer system includes a plurality of instruction execution units which operate in parallel and in a concurrent manner.

As is generally shown in FIG. 1, an instruction compounding unit 20 takes a stream of binary scalar instructions 21 and selectively groups some of the adjacent scalar instructions (which would otherwise be executed singly) for parallel execution. A resulting compounded instruction stream 22 therefore provides scalar instructions to be executed singly or in compound instructions formed by groups of scalar instructions to be executed in parallel. When a scalar instruction is presented to an instruction processing unit 24, it is routed to the appropriate one of a plurality of execution units for serial execution. When compounded instructions are presented to the instruction processing unit 24, each of the scalar components is routed to an appropriate execution unit for simultaneous parallel execution with the others. Typical execution units include, but are not limited to, an arithmetic logic unit (ALU) 26 for executing an instruction in response to two operands, a floating point arithmetic unit (FP) 30, a storage address generation unit (AU) 32, and a data-dependency collapsing ALU 28. An exemplary data-dependency collapsing unit is disclosed in co-pending U.S. application Ser. No. 07/504,910.

The compounding procedure upon which this invention depends can be implemented in a uniprocessor environment having a plurality of execution units where each execution unit executes a scalar instruction or alternatively a compounded scalar instruction. Further, compounded instructions can be executed in parallel in certain other computer system configurations. For example, compounding can be exploited in a multi-processor environment where a compound instruction is treated as a single unit for execution by one of a plurality of CPU's (central processing units).

Preferably, a computer architecture which can be adapted for handling compounded instructions is an IBM System/370 instruction-level architecture in which multiple scalar instructions can be issued for execution in each machine cycle. In this context, in the System/370 pipelined computer architecture, a machine cycle encompasses all of the pipeline steps or stages required to execute a scalar instruction.

The instruction sets for various IBM System/370 architectures such as the System/370, the System/370 extended architecture (370-XA), and the System/370 Enterprise Systems Architecture (370-ESA) are well known. Respecting these architectures, reference is given here to the Principles of Operation of the IBM System/370 (Publication #GA22-7000-10, 1987), and to the Principles of Operation, of the IBM Enterprise Systems Architecture/370 (Publication #SA22-7200-0, 1988). Also helpful is the publication entitled IBM 370 Assembly Language with ASSIST: Structured Concepts in Advanced Topics, by C. J. Kacmar, Prentice Hall, 1988.

In general, an instruction compounding facility will look for classes of instructions that may be executed in parallel, and will ensure that no interlocks between members of a compound instruction exist that cannot be handled by the hardware. When compatible sequences of instructions are found, the instructions are compounded.

Relatedly, an interlock occurs in parallel execution when concurrently-executing instructions require access to the same execution resource and no hardware means is provided for affording the concurrent access. If access is required to obtain operand data from the resource, a data-dependency interlock exists if the data must be written by one instruction before being either read or written by the other instruction. An address generation interlock exists if data being produced by execution of one of the instructions is required by a simultaneously-executing instruction for address calculation.

In order to identify instructions of a known instruction set which are compatible with other instructions for simultaneous execution, the set from which the instructions are drawn can be broken into categories of instructions that may be executed in parallel in a computer system configuration which executes all instructions of the instruction set. Instructions within certain of these categories may be compounded with instructions in the same category or with instructions in certain other categories. For example, the System/370 instruction set can be partitioned into the categories illustrated in FIG. 2. The rationale for this categorization is based on the functional requirements of the System/370 instructions and their hardware utilization in a typical System/370 computer system configuration. Other instructions of the System/370 instruction set are not considered specifically for compounding in this exemplary embodiment. This does not preclude them from being compounded by the technique of the present invention.

For example, consider the instructions contained in category 1 compounded with instructions from that same category in the following instruction sequence:

AR R1, R2

SR R3, R4

This sequence is free of data dependence interlocks and produces the following results which comprise two independent System/370 instructions:

R1=R1+R2

R3=R3-R4

Executing such a sequence would require two independent and parallel two-to-one ALU's designed to the instruction level architecture. Thus, it will be understood that these two instructions can be grouped to form a compound instruction in a computer system configuration which has two such ALU's. This example of compounding scalar instructions can be generalized to all instruction sequence pairs that are free of data dependence interlocks, hardware dependence interlocks, and address generation interlocks.

The flow diagram in FIG. 3 shows the generation of a compound instruction set program from an object code program in accordance with a set of customized compounding rules which reflect the categories of FIG. 2 together with both the system and hardware architecture of a System/370 complex. Successive blocks of object code instructions are provided as a byte stream which is input to a compounding facility that produces compounded instructions. Successive blocks of instructions in the byte stream having predetermined lengths are analyzed by the compounding facility 37. The length of each block 33, 34, 35 in the byte stream which contains the group of instructions considered together for compounding is dependent on the complexity of the compounding facility.

The particular compounding facility illustrated in FIG. 3 is designed to consider two-way compounding for "m" instructions in each block. The compounding facility 25 employs a two-instruction-wide window to consider every pair of instructions in each block.

In this exemplary two-way compounding scheme, compounding information is added to the instruction stream as one bit for every two bytes of text. In general, a tag containing control information can be produced for each instruction in the compounded byte stream--that is for each non-compounded scalar instruction as well as for each compounded scalar instruction included in a pair, triplet, or larger compounded group. This general approach is employed in the example of this invention. Relatedly, the tags specifically identify and differentiate those compounded scalar instructions forming a compound group from the remaining non-compounded scalar instructions remain in the block. The non-compounded scalar instructions remain in the block, and when fetched are executed alone.

The case of compounding at most two instructions provides the smallest grouping of scalar instructions to form a compound instruction, and uses the following preferred encoding procedure for the compounding information. Since all System/370 instructions are aligned on a half word (two-byte) boundary with lengths of either two, four, or six bytes, only one bit of compounding information need be provided for every half word. Hereinafter, the bits which contain the compounding information are called "tag bits" or "C bits". In this example, the tag bit value "one" indicates that the instruction that begins in the byte under consideration is compounded with the following instruction, while a tag bit value of "zero" indicates that the instruction that begins in the byte under consideration is not compounded with the following instruction. The tag bits associated with half words not containing the first byte of an instruction are ignored. When a compounded pair is fetched for execution, the tag bit for the first byte of the second instruction of a compounded pair is also ignored. As a result, this encoding procedure requires only one bit of information to identify a compounded instruction to a CPU during execution of the instruction.

As will be appreciated, when more than two scalar instructions can be grouped together to form a compound instruction, additional tag bits may be required. The minimum number of tag bits needed to indicate the specific number of scalar instructions actually compounded is the logarithm to the base two (rounded up to the nearest whole number) of the maximum number of scalar instructions that can be grouped to form a compound instruction. For example, if the maximum is two, then one tag bit is needed for each compound instruction. If the maximum is three or four, then two tag bits are needed for each compound instruction, and so on.

It will be apparent to those skilled in the art that the present invention requires an instruction stream to be compounded only once for a particular computer system configuration, and thereafter any fetch of compounded instructions will also cause a fetch of the tag bits associated therewith. This avoids the need for the inefficient last-minute determination in selection of certain scalar instructions for parallel execution that repeatedly occurs every time the same or different instructions are fetched for execution in the so-called super scalar machine.

Despite the advantage of compounding an object code instruction stream, it becomes a difficult procedure to implement under certain computer architectures unless a technique is developed for determining instruction boundaries in a byte stream. Such a determination is complicated when variable length instructions are allowed, and is further complicated when data and instructions can be intermixed in the same byte stream. Of course, at execution time instruction boundaries must be known to allow proper execution. But since compounding is preferably done a sufficient time prior to instruction execution, a technique is needed to compound instructions without knowledge of where instructions start and without knowledge of which bytes are data. In the example of this invention, the worst case is assumed, that is that instruction lengths are variable, that data is intermixed with instructions in the byte stream being compounded, and no reference points are available in the byte stream to identify instructions. As will be appreciated, for compounding, the absence of a reference point to identify the beginning of an instruction creates uncertainty in that many more tag bits will be generated by the compounding unit than might otherwise be necessary. Nevertheless, the unique technique of this invention works equally well with either fixed or variable length instructions. Once the start of an instruction is known (or presumed), the length can always be found in one way or another somewhere in the instructions. In the System/370 instructions, the length is encoded in the first two bits of the op code. In other systems, the length may be encoded in the operands or implicit if all instructions are the same length.

Operational Environment

Referring to FIG. 4 of the drawings, there is shown a representative embodiment of a portion of a digital computer system or digital data processing system constructed in accordance with the present invention. This computer system is capable of processing two or more instructions in parallel. It includes a first storage mechanism for storing instructions and data to be processed. This storage mechanism is identified as higher-level storage 36. This storage 36 (also "main memory") is a larger-capacity, lower-speed storage mechanism and may be, for example, a large-capacity system storage unit or the lower portion of a comprehensive hierarchical storage system or the like.

The computer system of FIG. 4 also includes an instruction compounding mechanism for receiving instructions from the higher-level storage 36 and associating with these instructions compounding information in the form of tags which indicate which of these instructions may be processed in parallel with one another. This instruction compounding mechanism is represented by instruction compounding unit 37. This instruction compounding unit 37 analyzes the incoming instructions for determining which ones may be processed in parallel. Furthermore, instruction compounding unit 37 produces for these analyzed instructions tag bits which indicate which instructions may be processed in parallel with one another and which ones may not be processed in parallel with one another.

The FIG. 4 system further includes a second storage mechanism coupled to the instruction compounding mechanism 37 for receiving and storing the analyzed instructions and their associated tag fields. This second or further storage mechanism is represented by compound instruction cache 38. The cache 38 is a smaller-capacity, higher-speed storage mechanism of the kind commonly used for improving the performance rate of a computer system by reducing the frequency of having to access the lower-speed storage mechanism 36.

The FIG. 4 system further includes a plurality of functional instruction processing units which operate in parallel with one another. These functional instruction processing units are represented by functional units 39, 40, 41, et cetera. These functional units 39-41 operate in parallel with one another in a concurrent manner and each, on its own, is capable of processing one or more types of machine-level instructions. Examples of functional units which may be used are: a general purpose arithmetic and logic unit (ALU), an address generation type ALU, a data dependency collapsing ALU (per co-pending U.S. application Ser. No. 071/504,910, a branch instruction processing unit, a data shifter unit, a floating-point processing unit, .and so forth. A given computer system may include two or more of some of these types of functional units. For example, a given computer system may include two or more general purpose ALU's. Also, no given computer system need include each and every one of these different types of functional units. The particular configuration of functional units will depend on the nature of the particular computer system being considered.

The computer system of FIG. 4 also includes an instruction fetch and issue mechanism coupled to compound instruction cache 38 for supplying adjacent instructions stored therein to different ones of the functional instruction processing units 39-41 when the instruction tag bits indicate that they may be processed in parallel. This mechanism also provides single instructions to individual functional units when their tag bits indicate parallel execution is not possible. This mechanism is represented by instruction fetch and issue unit 42. Fetch and issue unit 42 fetches instructions from cache 38, examines the tag bits and instruction operation code (op code) fields and, based upon such examinations, sends the instructions to the appropriate ones of the functional units 38-41.

A stream of instructions is brought in from auxiliary storage devices by known means, and stored in blocks called "pages" in the main memory 36. Sets of continuous instructions called "lines" are moved from the main memory 36 to the compound instruction cache 38 where they are available for high-speed reference for processing by the instruction fetch and issue unit 42. Instructions which are fetched from a cache are issued, decoded at 42, and dispatched to the functional units 39-41 for execution.

During execution, when reference is made to an instruction which is in the program, the instruction's address is provided to a cache management unit 44 which uses the address to fetch one or more instructions, including the addressed instruction, from the instruction cache 38 into a queue in the unit 42. If the addressed instruction is in the cache, a cache "hit" occurs. Otherwise, a cache "miss" occurs. A cache miss will cause the cache management unit 44 to send the line address of the requested instruction to a group of storage management functions illustrated collectively as a memory management unit 45.

These functions use the line address provided by the cache management unit 44 to send a line of instructions ("cache line") to the compound instruction cache 38.

In the context of SCISM architecture, in-cache instruction compounding is provided by the instruction compounding unit 37 so that compounding of each cache line can take place at the input to the compound instruction cache 38. Thus, as each cache line is fetched from the main memory 36 into the cache 38, the line is analyzed for compounding in the unit 37 and passed, with compounding information, for storage in the compound instruction cache 38.

Prior to caching, a line is compounded in the instruction compounding unit 37 which generates a set of tag bits. These tag bits may be appended directly to the instructions with which they are associated, or may be provided in parallel with the instructions. In any case, the bits are provided for storage together with their line of instructions in the cache 38. As needed, the compounded instructions in the cache 38 are fetched together with their tag bits by the instruction fetch and issue unit 42. As the instructions are received by the fetch and issue unit 42, their tag bits are examined to determine if they may be processed in parallel and their operation code (op code) fields are examined to determine which of the available functional units is most appropriate for their processing. If the tag bits indicate that two or more of the instructions are suitable for processing in parallel, then they are sent to the appropriate ones in the functional units in accordance with the codings of their op code fields. Such instructions are then processed concurrently with one another by their respective functional units.

When an instruction is encountered that is not suitable for parallel processing, it is sent to the appropriate functional unit as determined by an op code and it is thereupon processed alone and by itself in the selected functional unit.

In the most perfect case, where plural instructions are always being processed in parallel, the instruction execution rate of the computer system would be N times as great as for the case where instructions are executed one at a time, with N being the number of instructions in the groups which are being processed in parallel.

Instruction Formats

In FIG. 5, there is illustrated a quadword 50 which forms a portion of a cache line, the remainder of which is not illustrated. The quadword 50 includes four words, denoted as WORD0-WORD3. Each word includes a pair of half words, each half word including two bytes of data. Each byte includes 16 bits. Bit positions are numbered in ascending order for the quadword from bit 0 through bit 127.

Assume that the first half word in WORD0 includes a conventional two-byte instruction such as would be found in the instruction set for the System/370. The half word instruction 52 includes 16 bits of which the first eight, bits 0-7, form the op code. In the op code, bits 0 and 1 provide the length field code. In System/370 instructions, a code value of 0 indicates that the instruction is one half word long, the codes 01 and 10 denote a double half word (four byte) instruction, and the code 11 denotes that the instruction includes three half words (six bytes). The two byte instruction format includes a designation of a first operand in bit positions 8-11 and the second operand in bit positions 12-15. These operand fields identify registers of a set of general purpose registers where the operands for the instruction are stored.

Reference numeral 54 in FIG. 5 indicates the format for a double half word (four byte) instruction. In the double half word instruction, the first eight bits (byte 0) contain an op code with a length field code of 01 or 10. The first four bits of the second byte of the double word (byte 1) identify the first operand for the instruction in the form of a register (R) in the general purpose registers. The second four bits of byte 1 in the double half word instruction identify an address index register (RX) in the general purpose registers, while the first four bits of byte 2 identify a base address register (RB). As is known, the RX and RB registers are used for operand address calculation.

Instruction Compounding Unit

For the purpose of understanding the description of the instruction compounding unit which follows, instructions are provided in a cache line comprising a block of eight quad words, designated QW0-QW7. The instruction compounding unit, shown in greater detail in FIG. 6A and 6B (hereinafter "FIG. 6"), is suitable for use as the instruction compounding unit 37 of FIG. 1 to compound a cache line. The instruction compounding unit of FIG. 6 is designed for the general case in which instructions may be two, four, or six bytes in length, data may be interspersed in the cache line, and no reference point is provided to indicate where the first instruction begins. The instruction compounding unit of FIG. 6 simultaneously compounds a maximum of eight instructions, two instructions at a time, for parallel execution. In this case, a one-bit compounding signal is generated, with a compounding bit being generated for each half word of the line. Consequently, sixty four compounding bits (C bits) will be generated for each cache line.

To understand the operation of the instruction compounding unit of FIG. 6, consider the compounding rules which it implements. If d is a dependency function over two instructions, i.sub.j and i.sub.k, where j and k represent an instruction category number, i.sub.j will be referred to as the first or left instruction, while i.sub.k will be referred to as the second or right instruction. The dependency function d maps the dependencies between the two instructions being compounded into a set [A, E, .phi.] where A is an address generation dependency, E is an execution unit (data) dependency, and .phi. represents no dependencies, that is, independent execution.

Consider a compounding function C over two instructions being compounded. Given a value for d for these two instructions, together with a hardware requirement for each instruction, C is a binary function defined simply as C=1 meaning that the instructions can be compounded, or C=0, meaning that the instructions cannot be compounded.

Consider, for example, the following code sequence:

```
(1) AR     2,3
(2) SR     4,2
(3) AR     2,3
(4) SR     4,5
(5) SRL    6,1(0)
```

```
(6) AR     6,5
(7) AR     2,6
(8) L      1,0(0,2)
```

Instructions (1) and (2) may be compounded using two execution units (EU2 and EU3) to calculate R2=R2+R3 and R4=R4-(R2+R3). In this regard, EU2 is an execution unit which collapses the interlock between the instructions by performing a 3-to-1 compound operation. Such an execution unit is taught in co-pending U.S. patent application Ser. No. 07/504,910. Over instructions (1) and (2), C=1 and d=E.

Instructions (3) and (4) may be compounded using EU2 and EU3 to calculate R2=R2+R3 and R4=R4-R5. No dependency exists between the instructions, therefore C=1 and d=.phi..

For instructions (5) and (6), d=E, but C 32 0 because the interlock cannot be collapsed as the execution unit hardware of instruction (6) is defined. Instructions (7) and (8) demonstrate an address generation dependency: according to the compounding rules implemented by the instruction compounding unit of FIG. 6, C=0 because d=A.

The following symbology is used for considering two potentially compoundable instructions:

op1 r1,r2 ;first or left instruction

op2 r3,r4,(r5) ;second or right instruction

In this symbology, the designation op refers to the op code found in the first byte of each instruction, while the designations r1, r2 are registers in the register fields of the first instruction and r3, r4, (and possibly r5) are the registers in the fields of the second (and possibly third) byte of a second instruction.

Now, considering the symbology described above, if r4 is used as an addressing operand, as for example in the BCTR and BCR instructions of the System/370 instruction set, r1=r4 is considered an address generation dependency. The designations op1 and op2 are generic in that they may refer to an instruction of any format. The r fields are generally applied to two or four-byte instructions of well-known formats.

Compounding Rules

The rules for compounding category 1 instructions in an exemplary instruction set such as the System/370 instruction set are given below. These rules are implemented in the compounding unit of FIG. 6 and permit the compounding of fixed-point with fixed-point instructions and fixed-point with floating-point instructions. The categories are those designated in FIG. 2.

Category 1 Rules:

1. Categories 1 and 1

C=1

Exceptions

C=0 for the following:

1. op1=any, op2=any, and r1=r3=r4

2. op1={AR, SR, ALR, SLR}, op2={LPR, LNR}, and r1=r4

2. Categories 1 and 2

C=1 if d=.phi.; C=0 otherwise

3. Categories 1 and 3

1. If op 2={BCT,BCTR}, then C=1 if d={E,.phi.}; C=0 otherwise

2. If op2={BXH,BXLE}, then C=1 if d=.phi.

4. Categories 1 and 4

C=0

5. Categories 1 and 5

C=0

Exceptions

1. If op1=any and op2={BASR} then c=1 if d={E,.phi.}.phi.; C=0 otherwise

2. If op1=any and op2={BAS} then C=0 if d={A}C=1 otherwise

6. Categories 1 and 6

C=1 if d=.phi.; C=0 otherwise

7. Categories 1 and 7

C=0 if d=A; C=1 otherwise

8. Categories 1 and 8

C=0 if d=A; C=1 otherwise

9. Categories 1 and 9

C=0 if d=A; C=1 otherwise

10. Categories 1 and 10

C=0 if d=A; C=1 otherwise

11. Categories 1 and 11

C=1

12. Categories 1 and 12

C=1

13. Categories 1 and 13

C=1

14. Categories 1 and 14

C=0 if d=A; C=1 otherwise

15. Categories 1 and 15

C=0 if d=A; C=1 otherwise

16. Categories 1 and 16

C=0 if d=A; C=1 otherwise

17. Categories 1 and 17

C=0 if d=A; C=1 otherwise

The rules given above are complete for compounding an instruction pair in which the first instruction of the pair is a category 1 instruction. An exhaustive set of rules would include all categories, and would be constructed according to the compatibility and interlock analysis discussed above.

An instruction compounding unit such as that illustrated in FIG. 6 would implement a complete set of compounding rules for the general case. For purposes of illustration, the operation of the instruction compounding unit of FIG. 6 is presented in the context of two-instruction compounding employing the exemplary rules for category 1 compounding given above.

Detailed Description of the Instruction Compounding Unit

The instruction compounding unit of FIG. 6 includes a sixteen-byte bus 60 corresponding to the storage bus in FIG. 4, which transfers a cache line, quadword-by-quadword, from the main memory 36 to the instruction cache 38. Each quadword on the bus 60 is latched in a staging unit 61. When latching the current quadword on the bus 60, the staging unit 61 also retains the two most significant words of the prior quadword and the two most significant words of the first quadword. Compounding analysis, including instruction categorization, data dependency determination, and address generation dependency determination, is performed in a rules base unit 62 which applies the compounding rules given above. The rules base unit 62 generates a C bit for each of eight half words of a quadword currently in the staging unit 61. A compounding tag register 64 includes 16 individual four-bit registers for storage of the 64 C bits produced for the eight quadwords in a cache line being transferred to the compound instruction cache. The latched C bits can be obtained in parallel from the compounding register 64 to form a compounding bit vector, a C vector, for the cache line being transferred. An instruction compounding unit finite state machine (ICU FSM) 66 generates control signals for synchronization of the operations of the instruction compounding unit of FIG. 6.

In FIG. 6, the staging unit includes four registers 75, 76, 77, and 78. Each of the registers is capable of storing one half of a quadword, that is a double word of 64 bits. A multiplexer 74 fills the register 76 either from the bus 60, or from the output of the register 78. The registers 76 and 77 are designated as the L2LO and L2HI registers, respectively, while the registers 75 and 78 are denoted as S1 and S2 registers. Preferably, quadwords are forwarded to the cache from the registers 76 and 77.

Each quadword on the bus 60 is loaded into the registers 76 and 77, with the double word in bits 0-63 entered into the L2LO register and bits 64-127 into the L2HI register. Following loading of the first quadword into the registers 76 and 77, the last double word of the previous quadword is loaded from L2HI register into the S1 register. When the second quadword of a line is loaded, the double word in bit positions 0-63 of the first quadword is loaded into the S2 register 78 from the L2LO register 76. This double word is retained in the S2 register until the last quadword of the eight-quadword line has been loaded into the L2LO and L2HI registers 76 and 77. At the next available cycle for quadword transfer following loading of the last quadword of the current line, the double word is transferred out of the S2 register 78 through the multiplexer 74 into the L2LO register 76.

Refer now to FIG. 7 for an understanding of how compounding proceeds according to the instruction compounding unit of FIG. 6. Bits 64:127 of quadword i are held in the bit positions 0:63 of the S1 register 75. Relatedly, these positions are occupied by two half words 80 and 81, forming word 82, and half words 84 and 85, forming word 86. Bit positions 0:63 of quadword i+1 are in the corresponding bit positions of L2LO register 76. Bit positions 0:31 are occupied by half words 87 and 88, forming full word 89.

Recall that the worst case compounding process requires that a C bit be generated for each half word in an instruction byte stream. Therefore, the instruction compounding unit of FIG. 6 will generate a compounding bit for each of the half words of the quadword which is shown, in part, in FIG. 7. In generating C bits it is assumed that each half word is potentially either a two-or four-byte instruction. (Six-byte instructions are not compounded in this example, although it is contemplated by the inventors that instructions of any size can be compounded). A compounding box (CBOX) 62a of the rules base unit

62 (FIG. 6) generates a C bit for the half word 80 occupying bit positions 0:15 in the S1 register 75. The C bit for this half word is generated by the application, in CBOX 62a, of the compounding rules given above. Thus, the CBOX 62a must first determine whether the half word 80 contains the entirety of a two-byte instruction or the first half of a four-byte instruction. The CBOX 62a must also compare the operand of the instruction beginning in the half word 80 with the succeeding instruction to determine whether each instruction is in a category which can be compounded with the other instruction; it must also determine whether any interlocks exist between the two instructions in the form of data dependency or address generation hazards. Thus the CBOX must compare instruction op codes and operand and addressing registers of the two instructions.

The CBOX 62a assumes that an instruction begins in the half word 80. Recalling the instruction formats illustrated above in FIG. 5, it will be appreciated that the first 12 bits of the half word 80 will provide the instruction op code, the length code field of the instruction, and r1. If the length field code of instruction in the half word 80 decodes to a two-byte instruction, the CBOX 62a assumes that the next instruction begins with the half word 81. In order to determine whether an instruction beginning in half word 81 can be compounded with an instruction in half word 80, the CBOX 62a must have access to the 20 bits beginning in half word 81 and extending to the first 4 bits in the half word 84. These 20 bits are required in case the instruction beginning in the half word 81 is 4 bytes long, in which case, the first byte includes the instruction op code, the second byte, designations of r3 and r4, and the following half byte, the designation of (possibly) register r5.

Continuing with the assumption that the instruction in the half word 80 is a two-byte instruction, the CBOX 62a receives bits 0:11 of the half word 80 at input I1 and bits 16:35 beginning with the half word 81 at input I21, giving it enough information to determine instruction size, op code compatibility, and any interlocks.

Assuming that the length field code in bits 0:1 of the half word 80 indicate that the instruction is four bytes long, the CBOX 62a must have access to the 20 bits beginning with the half word 84, since the half word 81 is included in the instruction beginning in the half word 80. These 20 bits are obtained from register positions 32:51 of the S1 register, which embrace all of the half word 84 in the first four bits of the half word 85. The 20 bits for the second instruction following a four byte instruction are applied at I22 of the CBOX.

Attention is drawn to the fact that determination of the compoundability of an instruction beginning in a half word 81 with the following instructions requires access to the 20 bits beginning in the half word 84, and to the 20 bits beginning in the half word 85. However, as explained above, the 20 bits beginning in the half word 85 include the first 4 bits of the half word 87 in the register 76. Therefore the input to the CBOX which determines the compounding bit value for the half word 81 receives at its I22 input the 20 bits comprising bits 48:63 in the S1 register 75 and bits 0:3 in the half word 87 stored in bits 0:15 of the L2LO register 76.

Returning to the instruction compounding unit of FIG. 6, eight CBOX circuits 81-87 are shown. The CBOX circuits perform the actual compounding analysis according to the worst case scenario in which an instruction stream has variable length instructions intermixed with data and no reference to indicate where the first instruction of the cache line is. Since, in the System/370 example, all instructions are aligned on half word boundaries, a starting point for instructions is presumed, that reference point corresponding with bit position 0 of the first quad- word received in a cache line.

Each of CBOXs 80-87 generates a C bit for a respective one of the eight half words contained in the S1 and L2LO registers 75 and 76. Each box receives, at its Il input, the first 12 bits of a respective one of the half words, and at its I21 and I22 inputs, the first 20 bits beginning with the first and second half words following that which provides the Il input. Thus, for example, the CBOX 80 corresponds to the CBOX 62a of FIG. 7 in that it receives the first 12 bits of the first half word in the S1 register at its Il input, the 20 bits beginning with the second half word in the S1 register at its I21 input, and the 20 bits beginning with the third half word in the S1 register at its I22 input. In response, the CBOX 80 generates a C bit for the first half word of the S1 register.

The CBOX 81 generates a C bit for the second half word of the S1 register. It is noted that input I22 of CBOX 81 receives the 20 bits beginning with the last half word of the S1 register (bits 48:63) and continuing to the first four bits of the first half word in the L2LO register 76. Similarly, C bits are generated for the third and fourth half words in the S1 register by CBOXs 82 and 83, while the CBOXs 84-87 generate C bits for the first, second, third, and fourth half words in the L2LO register 76.

In the register 64, there are illustrated in FIG. 6, 16 separate 4-input, 4-output D registers 100-115. Each of the even-numbered registers receives an input from each of CBOXs 84-87, while each of the odd-numbered registers receives an input from each of the CBOXs 80-83. In FIG. 6, C bits from the CBOXs 81, 82, and 83, are provided through truncation elements 90, 91, and 92 respectively. For so long as a TRUNCATE signal output by the FSM 66 is low, the C bits input into circuit elements 90, 91 and 92 are forwarded through those elements to the odd-numbered latches of the register 64.

The instruction compounding unit in FIG. 6 is designed to correctly perform compounding for an arbitrarily rotated cache line, observing the following conditions:

1. No compounding occurs across cache lines. That is, the last instruction in QW7 of a cache line is not compounded with the first instruction in QW0 of a following cache line;

2. Up to the last three C bits for a line, that is the C bits for the last three half words of QW7, are truncated by being forced to 0, in view of condition (1); and

3. If the cache line has been rotated such that a quadword other than QW0 is received first, then compounding analysis is performed for instructions lying on the boundary between last and first quadwords received.

In order to compound between the last and first quadwords of a rotated cache line, the S2 register 78 receives the first four half words from the first quadword loaded from the bus 60 and retains them until the last quadword has been received, at which time, contents of the S2 register 78 are gated through the multiplexer 74 into the L2L0 register 76.

The controlling finite state machine 66 is of conventional design and responds to the following input signals:

FIRSTQW, which is asserted when the first quadword of the cache line is placed on the bus 60;

LASTQW, which is asserted when the last quadword of the cache line is on the bus 60;

EOL (End of Line), which is asserted when QW7 is on the bus 60; and

NUMFQW, which is the number (0 to 7) of the first quad word transferred on the bus 60, and which is valid when FIRSTQW=1.

These signals are produced by the cache management unit 44 (FIG. 4) in the course of a protocol which controls transfer of cache lines from the high level storage 36 to the compound instruction cache 38 in response to a cache miss.

The finite state machine 66 which controls the instruction compounding unit in FIG. 6 produces the following signals:

LD.sub.-- L2, signifying load the L2LO and L2HI registers;

LD.sub.-- S1, which signifies loading the S1 register;

LD.sub.-- S2, which signifies loading the S2 register;

GT.sub.-- S2.sub.-- L2LO, signifying gating of the contents of the S2 to the L2LO register;

LD.sub.-- CVR (0:15), signifying loading of the C-vector register 64. Each bit of this signal loads a corresponding four-signal register, that is, if LD.sub.-- CVR (0)=1, the register 100 is loaded; if LD.sub.-- CVR (1)=1, the register 101 is loaded. Preferably, in the design illustrated in FIG. 6, two LD.sub.-- CVR lines may be asserted simultaneously; and

TRUNCATE, which is activated in order to zero the C bits for instructions in the 6th, 7th and 8th half words in QW7.

Timing of the Instruction Compounding Unit

FIGS. 8A-8C show the timing of the instruction compounding unit in FIG. 6 for three representative rotations of incoming cache lines. The unit operates in a cycle of ten periods. In these figures, an eight quadword line is transferred, one quadword at a time in eight successive cycles on the bus 60. The current quadword on the line is designated as QWN, where N=0, 1, . . . , 7. As the quad words are registered, they are designated as QWNL or QWNH, where "L" signifies bits 0:63 of quadword QWN, while "H" signifies bits 64:127 of QWN.

With reference now to FIG. 8A, the compounding of the non-rotated cache line will be explained. In FIG. 8A, the eight quadwords of the cache line are sequentially sent on the bus 60 for storage in the cache. The presence of the first quadword of the transfer is signified by the signal FIRST.sub.-- QW, which is asserted during cycle period 0 when QW0 is on the bus 60, and which falls slightly past the beginning of period 1, when QW1 is on the bus. While the signal FIRST.sub.-- QW is valid, the FSM 66 gates in the

NUMFQW signal. The NUMFQW signal initializes a nine state cycling counter to a state representing the number of the first quadword on the bus. In FIG. 8A, NUMFQW has a (decimal) value of 0, indicating that quadword QW0 is on the bus. In response to the signals FIRST.sub.-- QW and NUMFQW, the FSM 66 activates the LD.sub.-- L2 signal which loads the quadwords from the bus 60 in their arrival sequence into the L2LO and L2HI registers 76 and 77.

Late in the second cycle period, the FSM 66 raises the LD.sub.-- S1 signal, which loads the S1 register 75 with the HI double word in the L2LO register 76 in the third cycle period. Thereafter, in each remaining cycle period the S1 register 75 receives the half word loaded in the L2HI register 77 during the previous cycle period until the LD.sub.-- S1 signal falls. In the second cycle period, FSM 66 also pulses the LD.sub.-- S2 signal, loading into the S2 register the lower double word of the first quadword received on the L2 bus 60. When the last quadword is being placed on the L2 bus 60, the LASTQW signal input to the FSM 66 is activated. In response, in the ninth cycle period of the compounding process, the FSM generates the GT.sub.-- S2.sub.-- L2LO signal, gating the contents of the S2 register in the L2LO register 78 in the tenth cycle period. The last quadword of the line, that is QW7, is signified to the FSM 66 by the EOL signal. This signal is latched by the FSM 66 for one period, represented by the EOLLTH signal which is internal to the FSM. In the cycle period following the EOLLTH signal, the FSM 66 activates the TRUNCATE signal and deactivates the LD.sub.-- L2 and LD.sub.-- S1 signals.

Therefore, for an unrotated cache line, quadwords are placed on the bus 60 in each of a sequence of eight cycle periods. In all, a ten cycle period defines the sequence for latching quadwords of the cache line and generating C bits for every half word in the line. Initially, in cycle period 0, QW0 is placed on L2 bus 60. In cycle period one, QW0 is latched in the staging unit 61, with its lower double word QWOL in the L2LO register 77 and its upper double word QWOH in the L2HI register 76. In the cycle period 2, the double word QWOL is latched in the S2 register 78, where it is held until cycle period 8. At the same time, the next quadword, QW1 is latched into the registers 76 and 77, while the contents of the L2HI register 76 are transferred into the S1 register 75. The sequence of entering the quadword into the registers 76 and 77 and transferring the high double word of the previous word into register 75 is repeated for cycle periods 3-8. In the last cycle period, the contents of register 78 are transferred back into the register 76, while the high double word of the previous cycle is transferred into register 75.

C bits are generated by the compounding unit 62 and latched into the CVR register 64 in cycle periods 1-9. In cycle period 1, C bits are generated only for the four half words in the register 76, while in cycle periods 2-8, C bits are generated and latched for the half words in the registers 75 and 76. In cycle period 9, C bits are generated only for the S1 register 75. Activation of the TRUNCATE signal forces the C bits for the last three half words in QW7H to 0.

Latching of the C bits generated in the sequence described above can be understood with reference to the LDCVR and NUMFQW signals in FIG. 8A. The NUMFQW signal is a three-bit signal which is valid while the FIRST-QW signal is active. The decimal value represented by the digits of the signal correspond to the number of the first quadword being transferred. For the unrotated line in FIG. 8A, the value is 0 (decimal). The FSM 66 uses the value of NUMFQW to initialize a state sequence having nine states. During the first and ninth states of the sequence, only one LDCVR signal is generated; during the other seven states, two LDCVR signals are generated. In FIG. 8A, LDCVR signals are given as a hexadecimal representation of the 16-bit LDCVR signal. Each hexadecimal digit represents four consecutive bits of the LDCVR signal. The first hexadecimal digit represents LDCVR bits 0-3, the second, bits 4-7, the third, bits 8-11, and the fourth, bits 12-15. Each bit of the LDCVR signal loads the correspondingly-numbered 4-bit CVR register. Thus, for example, LDCVR0, when active, loads the 4-bit CVR register 100, while LDCVR11, when active, loads the 4-bit CVR register 111. In cycle period 1 of FIG. 8A, the hexadecimal representation of the LDCVR signal is 8000. This means that the first hexadecimal digit has the value "1000". Thus, the load signal for the 4-bit register 100 is active, meaning that the C bits for the half words in the L2LO register 76 are being latched into the CVR register. In cycle period two, the first digit of the hexadecimal number is "6" while all the other digits are "0". Decoding the first digit gives the binary number "0110". Relatedly, the load signals for the 4-bit registers 101 and 102 are active. The 4-bit register 101 receives the C bits generated by the CBOXs 80-83 for the half word in the S1 register 75, which is QWOH in cycle period two. Similarly, the 4-bit register 102 is loaded with the C bits generated in CBOXs 84-87 for QW1L. The sequence of FIG. 8A proceeds through cycle periods 3-8 with the C bits generated by compounding across the quadword in the S1 and L2LO registers 75 and 76 being captured in the appropriate pair of 4-bit CVR registers. In cycle period 9, the last hexadecimal digit of the LDCVR signal has a value of "1" corresponding to the binary value of "0001", which loads the 4-bit CVR register 115 with the final four C bits for the cache line.

FIG. 8B illustrates the quadword loading and C bit generation cycle in the case where a cache line has been rotated to place the last quadword, QW7, first on the bus 60. In this case, the EOL signal is concurrent with the FIRST.sub.-- QW signal. Consequently, the EOLLTH signal is generated internally to the FSM 66, delaying the EOL signal for one cycle period and resulting in the generation of the TRUNCATE signal during cycle period two. The TRUNCATE signal prevents the compounding of the last three half words in QW7H with any instruction. As described below, such compounding is prevented by forcing the C bits for the last three half words of QW7H to 0. However, the lower double word in QW7, that is QW7L, is retained in the S2 register 78 until cycle period nine when it is entered into the L2LO register 76 for compounding with the instructions in QW6H. The initial value of NUMFQW synchronizes the generation of the LDCVR signals with the order of the rotated cache line.

FIG. 8C illustrates the ten-period cycle for compounding a rotated cache line in which the first quad word is neither QW0 nor QW7.

FIG. 9A and 9B (hereinafter "FIG. 9") shows a partial design for a CBOX. The design is partial in that only compounding rules for category 1 instructions are shown. Such compounding is instructive since category 1 is the worst-case category and places an upper bound on the design complexity of a CBOX. The skilled artisan will be able to derive corresponding logic which implements compounding rules for categories 2-12.

The inputs to the CBOX are I1 (0:11), the first twelve bits of the first half word in a pair of instructions. Following this, this half word will be referred to as "instruction 1". As discussed above in connection with FIG. 7, these bits contain the op code and r1 fields of the half word being considered for compounding. Because instruction 1 can be either a two-or four-byte instruction, two choices are possible for the second instruction (I2): if instruction 1 is a single half word, (bits 0:1="00"), then instruction 2 comes from the next half word following instruction 1. This corresponds to input I21 (0:19). As discussed above, instruction 2 may be a four-byte instruction, in which case the first 20 bits of the instruction text are required for compounding analysis. If bits 0:1 of I1="01", "10", or "11" instruction 2 comes from input I22 (0:19). These are the first twenty bits in the second half word following instruction 1.

Once the instruction length of instruction 1 is determined, instruction 1 and instruction 2 are decoded by decode blocks (DEC) as required. In this regard, the decode blocks simply decode the instruction op codes, producing an active output only if the op code corresponds with a predetermined category op code pattern employed by the decode block. At the same time, the first operand of instruction 1 is compared with the potential operand and address register fields of instruction 2 to determine whether any data or address generation interlocks exist. Dependency indications are combined with the op code decoding in a manner which implements the compounding rules given above. The signal generated by the logic of FIG. 9 (termed "category 1" logic) is a signal CMP.sub.-- C1, which is asserted if instruction 1 is in category 1 and compoundable with instruction 2. This signal is combined with signals CMP.sub.-- C2 through CMP.sub.-- C17, which correspond to instruction 1 being in a category from 2 through 17. The final result is the C bit output which is asserted if instruction 1 compounds with instruction 2.

Returning now to FIG. 9 and referring to instruction 1 as "I1" and instruction 2 as "I2", the first 12 bits of I1 are received at input A5. Bits 0:1 of I1 are fed to the input of OR gate 200 whose output is activated if either of these bits is set. Either bit being set signifies that I1 embraces more than two bytes. An inactive output of the OR gate 200 signifies that I1 is a two-byte instruction. The output of the OR gate 200 controls a multiplexer 201. If the output of the OR gate 200 is inactive, input A3 is output by the multiplexer 201. The input at A3 is the I21 input which constitutes bits 0:19 from the half word immediately following I1. Otherwise, if the output of the OR gate 200 is activated, the input at A4 is selected by the multiplexer 201. As illustrated, the input at A4 is I22, constituting the first 20 bits (0:19) of the second half word after I1. The op code portion (bits 0:7) of I1 is decoded in three decoders 210a, 210b, and 210c. All of these decoders decode category 1 instructions. Further, decoder 210b decodes either an AR or an ALR instruction, while decoder 210c decodes an SR or an SLR instruction.

The op code of the half word selected by the multiplexer 201 is fed to a bank of decode blocks 212a and 212b. If the op code satisfies the decoding condition of one of the blocks, the decoding block will activate. The decoding block conditions are listed in Table I. For example, if I2 has an op code which is decoded as a branch on count, the decoder denoted as I =BCTR will activate its output.

TABLE I

I=Cl	Instruction is Category 1
I=AXR	Instruction is AR, or ALR
I=SXR	Instruction is SR, or SLR
I=LXR	Instruction is LPR or LNR
I=C2	Instruction is Category 2
I=BCT	Instruction is BCT
I=BCTR	Instruction is BCTR
I=BAXR	Instruction is BASR
I=BAX	Instruction is BAS
I=C6	Instruction is Category 6
I=C7	Instruction is Category 7
I=C8	Instruction is Category 8

```
I=C9          Instruction is Category 9
I=C10         Instruction is Category 10
I=C11         Instruction is Category 11
I=C12         Instruction is Category 12
I=C13         Instruction is Category 13
I=C14         Instruction is Category 14
I=C15         Instruction is Category 15
I=C16         Instruction is Category 16
I=C17         Instruction is Category 17
```

Comparisons of register fields for I1 and I2 are performed in comparison (CMP) blocks 214-217. These comparisons are for the purpose of identifying dependencies which may constitute interlocks. Each of these blocks compares register r1 identified in bits 8:11 of I1 with the contents of the register field locations of I2. If the compared values are unequal, the output of a CMP block is active; if equal, the output is inactivated. In this regard, bits 8:11 of I2 correspond with register r3, bits 12:15 with register r4, and bits 16:19 with register r5. The comparison block 217 is provided to compare register r1 with only the first three bits of the r4 register field of I2. This comparison is used to detect execution dependencies between I1 and a BXH or BXLE instruction where bits 12:15 identify an even register but the instruction makes provision for comparison with an adjacent register with an odd number. In this case equivalence of bits 8:10 of I1 and bits 12:14 of I2 will signify equivalence of the register r1 with either of the odd or even registers designated in the r4 field of I2. This, of course, indicates an execution interlock.

In FIG. 9, the remaining logic up to and including the OR gate 251 is provided for combining the register field comparisons with op code indications to determine whether I1 and I2 are instructions which can be compounded. If compoundable, the output of the OR gate 251 is asserted, which will result in activation of the C bit for the half word identified as I1.

With reference to the compounding rules given above, the remainder of the logic in FIG. 9 will be explained. In the first rule, the category 1 instruction is compoundable with another category 1 instruction, with two exceptions. The first exception is when r1 is equal to both r3 and r4. This condition is tested in the OR gate 220, connected to comparison blocks 214 and 215. The output of the OR gate 220 is fed, together with the output of the decoder 210a and the decoder in the decoder bank 212 which decodes I=CI to the AND gate 221. If the condition exception is not met, the output of the AND gate 221 will be asserted, indicating that the first exception to the compounding of two category one instructions does not apply. The second exception is listed above and occurs when the op code of I1 identifies an AR, an SR, an ALR, or an SLR instruction, the op code of I2 identifies an LPR or an LNR instruction, and rl=r4. The I1 op codes for this instruction are tested in the OR gate 222, while the AND gate 223 tests the concurrence of the I1 and I2 op code exceptions. Thus, if the output of the AND gate 223 is asserted, the op codes for I1 and I2 indicate instructions in the respective exception classes. The output of the AND gate 223 is combined in the AND gate 224 with the output of the comparator block 215. If r1=r4, the output of this block will be inactive, which will keep the output of the AND gate 224 from activating. If the comparator block 215 is active, indicating inequality of the registers, the output of the AND gate 224 will activate, indicating that the conditions of the exception have not been met. The outputs of the AND gates 221 and 224 are forwarded through the OR gate to the OR gate 251.

The AND gate 227 tests for compounding according to rule 2 of the compounding rules. Thus the gate is activated if the op code of I1 is in category 1, the op code of I2 is in category 2, and r1 does not equal r3.

The OR gate 233 applies rule 3 with its two exceptions. In this regard, if the op code of I2 decodes to BCTR, address generation dependency must be cleared. For the BCTR instruction, such dependency occurs if r1=r4. This exception is evaluated by the AND gate 231. The AND gate 230 checks for address generation dependency when I1 is a category 1 instruction and I2 is a BCT instruction. When I2 is a BCT instruction, address generation dependency occurs if r1=r4 or r5. To detect this dependency, the AND gate 230 receives inputs from comparison blocks 215 and 216. Occurrence of the last exception of the compounding rule 3 is detected by the AND gate 229. This exception arises when I2 is a BXH or BXLE instruction, in which case address generation dependency occurs if r1=r5, or execution dependency occurs if r1=r3, or r1=r4, or if r1 equals the odd or even register in the r4 field. Thus, if I1 is a category 1 instruction, I2 is a category 3 instruction, and none of the exceptions to rule 3 occur, the output of the OR gate 233 is activated.

Category 1 and 4 instructions are not compounded. If I1 is a category 1 instruction and I2 is a category 4, the output of OR gate 251 will remain inactive.

Rule 5 is implemented by the OR gate 239, with the two exceptions to rule 5 being tested, respectively in the AND gates 236 and 237.

Rules 6, 7, 8, and 9 are implemented, respectively, by AND gates 241, 242, 245, and 246.

The exceptions to rules 10 and 14-17 are tested in AND gates 247, 248, 249, 250, and 252. Rules 11-13 have no exceptions. The OR gate receives the outputs of the AND gates 247-250 and 252, and the outputs from the decoders for categories 11-13. The output of the OR gate 253 is combined with the output of the decoder 210a in AND gate 254 to test for compounding according to rules 10-17. The output of the AND gate 254 is fed to the OR gate 251.

The OR gate 251 collects the results of testing I1 and I2 according to the category 1 rules. The output of the OR gate 251 is combined with outputs of groups of CBOX logic which apply appropriate rules categorization for the cases where I1 is in any one of categories 2-17. The output of all category rule logic is collected in the OR gate 254 whose output at B1 provides the C bit for the half word identified as I1.

Truncation

Referring now to FIGS. 6, 8A and 8B, the truncation of compounding for the last three half words in QW7 will be explained. In FIG. 6, truncation components 90, 91, and 92 receive the C bits produced for the last three half words in register S1 by CBOXs 81, 82, and 83. Each of these elements is an AND gate circuit which receives a non-inverted C bit and the inverse sense of the TRUNCATE signal. When the TRUNCATE signal is inactive, the C outputs of the CBOXs are passed through the AND gates 90, 91, and 92, respectively. Activation of the TRUNCATE signal (refer to FIGS. 8A-8C) occurs when the last double word of QW7, including bits 64-127, is in the S1 register 75. At this point, the CBOX 83 attempts to determine compounding of the last half word in the S1 register with the first or second half word in the L2LO register 76. However, activation of the TRUNCATE signal, inverted at the input to the AND gate, inactivates the output of that gate and forcing the C bit for the half word at the I1 input of the CBOX 83 to zero. The next-to-last and last half words of QW7 are truncated in the same manner as the first by AND gates 91 and 90, respectively.

A Scalable Compound Instruction Set Machine Architecture

Referring to FIG. 10, there is shown a detailed example of how a computer system can be constructed for using the compounding tags of the present invention to provide parallel processing of object code computer instructions. The instruction compounding unit 420 used in FIG. 10 is assumed to be of the type described in FIG. 6 and, as such, it generates for each instruction a one-bit tag. These tags are used to identify which pairs of instructions that may be processed in parallel. These instructions and their tags are supplied to and stored into the compound instruction cache 412. The fetch and issue unit 460 fetches the instructions and their tags from cache 412, as needed, and arranges for their processing by the appropriate one or ones of a plurality of functional instruction processing units 461, 462, 463 and 464. Fetch and issue unit 460 examines the tags and op code fields of the fetched instructions. If the tags indicate that two successive instructions may be processed in parallel, then fetch and issue unit 460 assigns them to the appropriate ones of the functional units 461-464 as determined by their op codes and they are processed in parallel by the selected functional units. If the tags indicate that a particular instruction is to be processed in a singular, nonparallel manner, then fetch and issue unit 460 assigns it to a particular functional unit as determined by its op code and it is processed or executed by itself.

The first functional unit 461 is a branch instruction processing unit for processing branch type instructions. The second functional unit 462 is a three input address generation arithmetic and logic unit (ALU) which is used to calculate the storage address for instructions which transfer operands to or from storage. The third functional unit 463 is a general purpose arithmetic and logic unit (ALU) which is used for performing mathematical and logical type operations. The fourth functional unit 464 in the present example is a data dependency collapsing ALU of the kind described in the above-referenced co-pending U.S. application Ser. No. 07/504,910. This dependency collapsing ALU 464 is a three-input ALU capable of performing two arithmetical/logical operations in a single machine cycle.

The computer system embodiment of FIG. 10 also includes a set of general purpose registers 465 for use in executing some of the machine-level instructions. Typically, these general purpose registers 465 are used for temporarily storing data operands and address operands or are used as counters or for other data processing purposes. In a typical computer system, sixteen (16) such general purpose registers are provided. In the present embodiment, general purpose registers 465 are assumed to be of the multiport type wherein two or more registers may be accessed at the same time.

The computer system of FIG. 10 further includes a high-speed data cache storage mechanism 466 for storing data operands obtained from a higher-level storage unit (not shown). Data in the cache 466 may also be transferred back to the higher-level storage unit. A cache management unit receives instruction addresses from the control unit 460 and either moves the addressed instruction and its tag to the unit, or detects a miss and begins the process of moving a cache line into the cache.

The particular mode in which the tags accompany compounded instructions for storage in the cache 466 is a matter of design choice. In many of the cross-referenced applications, the tags are inserted into the compounded instruction stream, with each tag bit appended to the half word for which it was generated. For purposes of illustration, a technique for providing tag bits for storage and use with a cache line is illustrated in FIG. 11. As FIG. 11 shows, instructions may occupy six, four, or two bytes. For the example of this invention, the compounding rules apply only to instructions of two or four bytes' length. Instructions which are six bytes in length are not compounded. However, tags are generated for every half word in a cache line. As FIG. 11 illustrates, the tag bits are preferably assembled into a C-vector which is separate from the compounded cache line. In FIG. 11, a portion of a cache line including quadwords QWI and QWI+1 is indicated by 390, while the accompanying tags are shown in the form of a C-vector 372. It will be obvious to those reasonably skilled in the art that the C-vector can be formed by parallel extraction of C bits registered in the CVR64 of FIG. 6. With the compounding bits vectored as illustrated in FIG. 11, there are a number of ways to implement their storage in cache. FIGS. 12A and 12B illustrate two such ways. FIGS. 12A and 12B both assume a quadword-wide bus, which comports with the bus 60 in FIG. 6, plus extra lines between the instruction compounding unit and the compound instruction cache for tags. Further, in keeping with the example explained above, the cache line is assumed to be eight quadwords in length, with the instruction compounding unit generating one compounding bit for every two bytes of text in a cache line. Thus, 64 compounding bits are generated for each compound cache line. These bits must be accommodated in a cache architecture which associates the compounding bits with their respective half words.

The simplest implementation for caching compounding bits with an associated cache line would see an increase in the internal word size of the processor between the cache and the instruction fetch and issue unit, as illustrated in FIG. 12A. This implies that the compounding bits are appended to quadwords, or inserted into the instruction stream at each half word. In FIG. 12A, a cache line organized into eight storage locations is illustrated. Without compounding, each location is eight bytes wide. With eight locations, a 16 byte cache line is stored. With one compounding tag per half word, and two-way compounding, a minimum of one extra bit of storage for every half word of instruction text is required. Thus, eight compounding bit locations are required for every sixteen bytes. The implication is that the cache word size must be expanded from 128 to 136 bits. FIG. 12A illustrates a cache structure for two-way compounding and a quadword-wide cache bus. The cache bus and internal word size are expanded to 136 bits. The drawback to this scheme is that a new memory design is required, implying, for example, error correction for larger words.

A second approach is illustrated in FIG. 12B and utilizes a tag cache that is separate from, but operated in parallel with, the instruction cache. This structure implies that tags are separate from the instruction text. However, as with FIG. 12A, the requirement that the tags accompany their respective instructions necessitates expansion of the bus between the cache and the instruction fetch and issue unit. In this case, the internal cache word size is unchanged; however, the size of the bus between the cache and the instruction fetch and issue unit must increase to accommodate parallel operation of the tag cache. The design of FIG. 12B may be hardwired. Alternatively, a separate tag cache management unit would be provided.

Example of SCISM Operation

FIG. 13 shows an example of a compounded instruction sequence 500 which may be processed by the computer system of FIG. 10. The FIG. 13 example is composed of the following instructions in the following sequence: Load, Add, Compare, Branch on Condition and Store. These are identified as instructions I1-I5, respectively. The tags for these instructions are 1,1,0,1 and 0, respectively. These tags are arrayed in a C-vector 502 which accompanies the instructions 500. Because of the organization of the machine shown in FIG. 10, the Load instruction is processed in a singular manner by itself. The Add and Compare instructions are treated as a compound instruction and are processed in parallel with one another. The Branch and Store instructions are also treated as a compound instruction and are also processed in parallel with one another. When these instructions are provided to the instruction fetch/issue unit, they are accompanied by the C-vector 502.

The table of FIG. 14 summarizes information for each one of the FIG. 13 instructions. The R/M column in FIG. 14 indicates the contents of the first field in each instruction. As discussed above, this field is typically used to identify a particular one of the general purpose registers which contains the first operand. An exception is a case of the Branch on Condition Instruction, wherein the R/M field contains a condition code mask. The R/X column of FIG. 14 indicates the contents of the field in two-byte instructions which identifies the second operand register and which, in four-byte instructions, identifies the register containing the address index value. The B column in FIG. 14 indicates the contents of the register field in a four-byte instruction identifying the base register. As is conventional with System/370 instructions, a zero in the B column indicates the absence of a B field or the absence of a corresponding address component in the B field. The D field of FIG. 14 indicates the content of a further field in each instruction which, when used for address generation purposes, includes an address displacement value. A zero in the D column may also indicate the absence of a corresponding field in the particular instruction being considered or, alternatively, an address displacement value of zero.

Considering now the processing of the Load instruction of FIG. 13, the fetch/issue control unit 460 determines from the tags for this Load instruction that the Load instruction is to be processed in a singular manner by itself. The action to be performed by this Load instruction is to fetch an operand from storage, in this case the data cache 466, and to place such operand into the R2 general purpose register. The storage address from which this operand is to be fetched is determined by adding together the index value in register X, the base value in register B and the displacement value D. The fetch/issue control unit 460 assigns this address generation operation to the address generation ALU 462. In this case, ALU 462 adds together the address index value in register X (a value of zero in the present example), the base address value contained in general purpose register R7 and the displacement address value (a value of zero in the present example contained in the instruction itself. The resulting calculated storage address appearing at the output of ALU 462 is supplied to the address input of data cache 466 to access the desired operand. This accessed operand is loaded into the R2 general purpose register in register set 465.

Considering now the processing of the Add and Compare instructions, these instructions and their tags are fetched by the fetch/issue control unit 460. The control unit 460 examines the tags for these two instructions and notes that they may be executed in parallel. As seen from FIG. 14, the Compare instruction has an apparent data dependency on the Add instruction since the Add must be completed before R3 can be compared. This dependency, however, can be handled by the data dependency collapsing ALU 464. Consequently, these two instructions can be processed in parallel in the FIG. 11 configuration. In particular, the control unit 460 assigns the processing of the Add instruction to ALU 463 and assigns the processing of the Compare instruction to the dependency collapsing ALU 464.

ALU 463 adds the contents of the R2 general purpose register to the contents of the R3 general purpose register and places the result of the addition back into the R3 general purpose register. At the same time, the dependency collapsing ALU 464 performs the following mathematical operation:

R3+R2-R4

The condition code for the result of this operation is sent to a condition code register located in branch unit 461. The data dependency is collapsed because ALU 464, in effect, calculates the sum of R3+R2 and then compares this sum with R4 to determine the condition code. In this manner, ALU 464 does not have to wait on the results from the ALU 463 which is performing the Add instruction. In this particular case, the numerical results calculated by the ALU 464 and appearing at the output of ALU 464 is not supplied back to the general purpose registers 465. In this case, ALU 464 merely sets the condition code.

Considering now the processing of the Branch instruction and the Store instruction shown in FIG. 13, these instructions and their tags are fetched from the compound instruction cache 412 by the fetch/issue control unit 460. Control unit 460 determines from the tags for these instructions that they may be processed in parallel with one another. It further determines from the op codes of the two instructions that the Branch instruction should be processed by the branch unit 461 and the Store instruction should be processed by the address generation ALU 462. In accordance with this determination, the mask field M and the displacement field D of the Branch instruction are supplied to the branch unit 461. Likewise, the address index value in register X and the address base value in register B for this Branch instruction are obtained from the general purpose registers 465 and supplied to the branch unit 461. In the present example, the X value is zero and the base value is obtained from the R7 general purpose register. The displacement value D has a hexadecimal value of twenty, while the mask field M has a mask position value of eight.

The branch unit 461 commences to calculate the potential branch address (0+R7+20) and at the same time compares the condition code obtained from the previous Compare instruction with the condition code mask M. If the condition code value is the same as the mask code value, the necessary branch condition is met and the branch address calculated by the branch unit 461 is thereupon loaded into an instruction counter in control unit 460. This instruction counter controls the fetching of the instructions from the compound instruction cache 412. If, on the other hand, the condition is not met (that is, the condition code set by the previous instruction does not have a value of eight), then no branch is taken and no branch address is supplied to the instruction counter in control unit 460.

At the same time that the branch unit 461 is busy carrying out its processing actions for the Branch instruction, the address generation ALU 462 is busy doing the address calculation (0+R7+0) for the Store instruction. The address calculated by ALU 462 is supplied to the data cache 466. If no branch is taken by the branch unit 461, then the Store instruction operates to store the operand in the R3 general purpose register into the data cache 466 at the address calculated by ALU 462. If, on the other hand, the branch condition is met and the branch is taken, then the contents of the R3 general purpose register is not stored into the data cache 466.

The foregoing instruction sequence of FIG. 13 is intended as an example only. The computer system embodiment of FIG. 12 is equally capable of processing various other instruction sequences. The example of FIG. 13, however, clearly shows the utility of the compound instruction information in determining which pairs of instructions may be processed in parallel with one another.

Considerations of Industrial Application

The discussion above provides a hardware implementation for compounding instructions for parallel execution. It is asserted that this solution does not compromise the cycle time of the machine in which it is embodied. As the example of FIGS. 12-14 shows, it can support and even simplify the control of a large number of functional units. As FIG. 6-11 show, the instruction compounding unit, the cache configuration and the instruction processing architecture which result are all feasible for implementation.

The compound instruction cache architecture gives rise to a number of distinct advantages in the industrial application of the invention. First, it eliminates the need for a software compounding facility, which permits the invention to be applied to existing instructions without modifying their object code forms and which can accommodate future codes, thereby obviating modification to compilers or assemblers. Next, the overhead required for storage of the compounding information is limited to the compound instruction cache. No overhead is imposed on any storage means standing above the cache in the memory hierarchy: not in the semiconductor memory (main memory), in the direct access device storage, or anywhere else. Further, the only time a performance penalty will occur for non-sequential operations is when the target instruction required for the operation is not in the cache. In the case of branches, the likelihood of that occurring is directly related to the miss ratio of the cache. It is entirely conceivable for a compound instruction cache of sufficient size to contain entire program loops of compound instructions, making the branch penalties negligible. Another advantage of this architecture is the ability of self-modifying code to be handled simply by trapping writes to the compound instruction stream, invalidating the cache line written to, requesting the updated line from the upper levels of the memory hierarchy, and recompounding the line. Last, even though the proposed architecture changes neither the amount nor the duration of the analysis that must be performed to attain a particular level of compounding (and, thus, parallelism), the analysis is performed only when a cache miss occurs and is thus infrequent by definition: no designer would purposefully build an instruction cache with a high miss ratio into a high-performance computer. The compounding analysis will increase cache miss service time by some amount proportional to the degree of analysis performed.

The first design consideration in developing an industrial application of this invention can be appreciated with reference to FIG. 6. The staging unit 61 effectively permits compounding over an entire quadword, which is precisely the unit of transfer between the main memory and the compound instruction cache. In matching the size of the unit of transfer into the cache, the compounding process can consider all available pairs of instructions as they are presented to the cache for storage therein. This reduces the time penalty for two-way compounding. In the general case, the size of the staging unit is a function of the number of instructions that constitute a single compound instruction and the scope of the analysis for compounding. In some cases it may turn out that increasing the size of the staging unit beyond a certain value may have diminishing returns.

The complexity of the instruction compounding unit will vary with the goals which compounding is intended to achieve. In this regard, the instruction compounding unit of FIG. 6 implements compounding rules for seventeen categories of instructions in a scheme which compounds at the maximum only two instructions. More complex compounding over, for example, three or more instructions can be accomplished by a compounding unit whose compounding section extrapolates the basic design of the CBOX illustrated in FIG. 9. Such a design may result in a more complex tag which would include control information, compounding information, steering bits, and other information of the type typically associated with horizontal microcode. The creation of compounding information and the semantics imputed to the tag are limited only by size constraints of the design and the time penalty ascribed to cache miss servicing. Relatedly, the tag can be as minimal or maximal as time and space allow. For example, consider the very frequent System/370 instruction pair Test Under Mask (TM) followed by Branch on Condition (BC). Given the high frequency of the instruction pair, compounding it alone for parallel execution can improve processor performance. Should a designer choose to compound only this pair, then the rules base for the compounding unit contains only one rule, and the CBOX and compounding unit become trivial. At the other extreme, the rules base may contain rules for subset, but still a substantial part of, a complete instruction-set architecture. It may additionally contain further information pertaining to the physical properties of the functional units, facilitating the embedding of control information in the tags. The rules base, though implementable in hardwired, random logic, may be implemented in some form of fast-access programmable storage, thereby allowing for flexibility as more functional units are added or subtracted, more or fewer types of compoundings are desired, or even as the computing environment changes. Relatedly, certain compoundings may be more advantageous in a commercial environment than in an engineering-scientific environment, or vice versa. This implies that the rules base can be programmable, with rules decisions being made at machine configuration time. Therefore, the inventors contemplate that, instead of being hardwired, the CBOX functions of the instruction compounding unit could be implemented in a fast-access, multi-ported memory which is programmable with a desired set of rules at the time a machine is manufactured.

Proposals have been made for decreasing the cache miss ratio by prefetching cache lines without waiting for a cache miss. If the cache management unit were designed to prefetch the next-sequential line of instructions, it would be possible to hide much of the time required by the instruction compounding unit for compounding. The fraction of all line compoundings that are hidden will be determined in this case by the program instruction-fetching behavior, as well as the organization for the compound instruction cache.

Certain specific design decisions have been incorporated into the discussion above for the purpose of presenting examples. Thus, this invention may be practiced by incorporation of C bits directly into the instruction stream at each half word boundary. Further, compound instructions could simply issue directly from the cache, rather than employing an instruction fetch/issue control unit with a buffer or stack. Also, when a cache miss and subsequent line fetch occur, it may be beneficial from a performance standpoint to pass the instruction addressed for execution directly to the functional units for execution at the scalar rate, rather than stall the functional units while the line is analyzed for compounding.

Therefore, while we have described what are considered to be preferred embodiments of this invention, it will be obvious to those skilled in the art in view of all of the considerations discussed above that various changes and modifications may be made to the invention without departing from its spirit. Therefore the invention and this description are intended to cover all changes and modifications as fall within the spirit and scope of the appended claims.

*** * * * ***

Images

View Cart | Add to Cart

Hit List | Previous | Next | Top

Home | Quick | Advanced | Pat Num | Help

United States Patent — ***5,287,467***
Blaner , et al. — **February 15, 1994**

Pipeline for removing and concurrently executing two or more branch instructions in synchronization with other instructions executing in the execution unit

Abstract

The parallelism of a multi-pipelined digital computer is enhanced by detection of branch instructions from the execution pipelines and concurrent processing of up to two of the detected instructions in parallel with the operations of the execution pipelines. Certain branch instructions, when detected, are removed altogether from the pipeline, but still processed. The processing is synchronized with the execution pipeline to, first, predict an outcome for detected branch instructions, second, test the conditions for branch instructions at their proper place in the execution sequence to determine whether the predicted outcome was correct, and third, fetch a corrected target instruction if the prediction proves wrong.

Inventors: **Blaner; Bartholomew** (Newark Valley, NY); **Jeremiah; Thomas L.** (Endwell, NY); **Vassiliadis; Stamatis** (Vestal, NY); **Williams; Phillip G.** (Vestal, NY)
Assignee: **International Business Machines Corporation** (Armonk, NY)

Appl. No.: **687309**
Filed: **April 18, 1991**

Current U.S. Class:	**712/235**; 712/237; 712/240
Intern'l Class:	G06F 009/38
Field of Search:	395/375,800

References Cited [Referenced By]

U.S. Patent Documents

3764988	Oct., 1973	Onishi	395/375.
4766566	Aug., 1988	Chuang	395/375.
4827402	May., 1989	Wada	395/375.
4833599	May., 1989	Colwell et al.	395/650.

4853840	Aug., 1989	Shibuya	395/375.
5075844	Dec., 1991	Jardine et al.	395/375.
5081574	Jan., 1992	Larsen et al.	395/375.
5101344	Mar., 1992	Bonet et al.	395/375.
5142634	Aug., 1992	Fite et al.	395/375.
5185868	Feb., 1993	Tran	395/375.
5197136	Mar., 1993	Kimura et al.	395/375.

Primary Examiner: Lee; Thomas C.
Assistant Examiner: Harrity; Paul
Attorney, Agent or Firm: Augspurger; Lynn L.

Claims

We claim:

1. In a digital computer which includes an instruction source for providing instructions to be executed, an execution pipeline for executing a stream of instructions, and a condition determination means connected to the execution pipeline for providing condition signals indicating results of executed instructions, a combination for branch instruction processing, the combination comprising:

instruction issue means coupled to the instruction source and to the execution pipeline for providing a stream of instructions to be executed; and

branch instruction processing mean s for searching all instruction text of a current instruction sequence of said instruction stream to detect all branch instructions in said instruction sequence and determine the text and location of each of said detected branch instructions, for removing ones of said detected branch instructions not requiring execution in the execution pipeline from the instruction sequence before said instruction sequence is loaded into the execution pipeline, for decoding and predicting whether or not a branch will be taken by using the branch instruction location as an address for the branch in a branch history cache, for initiating a necessary fetch or prefetch according to said prediction, and for concurrently executing a plurality of said removed branch instructions substantially in parallel with other ones of said instructions of said instruction sequence executing in the execution pipeline.

2. The combination of claim 1, wherein the digital computer further includes compounding means for generating compounding information for each of said instructions, the compounding information indicating whether the instruction is to be executed in parallel with another one of said instructions, the compounding means conditioning the compounding information for each of said branch instructions to indicate a predicted instruction sequence resulting from execution of the branch instruction, the combination wherein

prediction means in the branch instruction processing means and coupled to the instruction source for fetching to the instruction issue means a target sequence of instructions in responses to compounding information indicating whether execution of a branch instruction will result in the sequence of instructions branching to a target sequence before logically previous ones of said instructions are dispatched.

3. The combination of claim 2, further including:

test means in the branch instruction processing means for producing a refetch signal in response to condition signals indicating that the result predicted by the compounding information is incorrect; and

refetching means in the branch instruction processing means and coupled to the test means for fetching to the instruction issue means a corrected sequence of instructions in response to the refetch signal.

4. The combination of claim 3, further including means responsive to the refetch signal for changing the compounding information to indicate another result.

5. The combination of claim 1, wherein the instruction source provides instructions in response to an instruction identification signal, the branch instruction processing means including:

means connected to the instruction issue means for generating a prediction signal for each said branch instruction in the stream of instructions, the prediction signal indicating a predicted outcome for execution of the branch instructions;

means connected to the prediction means and to the instruction issue means for enqueueing said branch instructions and prediction signals;

means responsive to the prediction signal and to the stream of instructions for generating an identification signal identifying instructions to be executed as a result of a predicted outcome; and

test means connected to the enqueueing means and to the prediction means for executing one of said enqueued branch instructions and for changing the identification signal if results produced by executing the branch instruction indicate that the outcome indicated by the prediction signal enqueued with the branch instruction is incorrect.

6. The combination of claim 5, wherein the test means includes means for dequeueing concurrently more than one of said branch instructions and for changing the identification signal if a predicted outcome for at least one of said executed branch instructions is incorrect.

7. The combination of claim 6, further including synchronization means connected to the execution pipeline and to the enqueueing means for providing queued instructions to the test means in synchronization with execution of the stream of instructions.

8. The combination of claim 6, wherein the branch instruction processing means includes a branch target buffer.

9. A digital computer including an instruction source responsive to fetch address signals for providing instructions identified by the fetch address signals, an execution pipeline for executing a stream of instructions, a condition means connected to the execution pipeline for providing condition signals indicating instruction execution results, and instruction sequencing means connected to the instruction source and to the execution pipeline for providing a sequence of fetch address signals to the instruction source and for coupling to the execution pipeline a sequence of instructions of said instruction stream provided by the instruction source in response to the sequence of fetch address signals, a method for executing branch instruction in parallel with the execution of instruction by the execution pipeline, the method including the steps of:

(a) detecting all of said branch instructions in the sequence of instructions;

(b) removing ones of said detected branch instructions not requiring execution in the execution pipeline;

(c) generating a prediction signal indicating a predicted outcome for each of said removed branch instructions;

(d) in response to the prediction signal for one of said branch instructions, conditioning the fetch address signals to a predicted sequence corresponding to a predicted outcome indicated by the condition of the prediction signal;

(e) in synchronization with execution of other said instructions in the execution pipeline, concurrently executing two or more of said removed branch instructions in the instruction sequencing means to determine actual outcomes in response to condition signals;

(f) comparing the actual outcome for each of the two or more executed branch instructions with the predicted outcome indicated by the prediction signal for each of said branch instructions; and

(g) if the predicted outcome for one of the two or more branch instructions is incorrect, conditioning the address signals to a refetch sequence corresponding to a correct outcome for the branch instruction; otherwise

(h) continuing the sequence of fetch addresses and performing steps (a)-(g).

10. The method of claim 9, further including, after step (d), for each branch instruction;

(d1) generating an instruction address for the branch instruction;

(d2) generating a branch target address for the branch instruction; and

(d3) enqueueing the branch instruction, the branch instruction address, the branch target address, and the

prediction signal for the branch instruction; and, wherein step (d) includes, prior to executing, dequeueing the two or more branch instructions.

11. The method of claim 9 further including, before step (e) generating a token for each said branch instruction in the sequence of instructions and propagating the generated tokens in the execution pipeline; and

wherein step (e) is executed in responses to the tokens propagating to a predetermined pipeline stage.

12. The method of claim 11, wherein first tokens are generated for the branch instructions which only test the condition signals and wherein step (e) is executed in response to the first tokens propagating to an address generation stage of the execution pipeline.

13. The method of claim 12 wherein second tokens are generated for the branch instructions which require pipelined execution to change the condition signals and wherein step (e) is executed in response to the second tokens propagating to an execution stage of the execution pipeline, the execution stage following the address generation stage.

14. The method of claim 12, further including, after generation of the first tokens, removing from the sequence of instructions the branch instructions which only test the condition signals.

15. In a digital computer including an instruction source responsive to fetch address signals for providing instruction identified by the fetch address signals, an execution pipeline for executing a stream of instructions, a condition means connected to the execution pipeline for providing condition signals indicating instruction execution results, an instruction sequencing means connected to the instruction source and to the execution pipeline for providing a sequence of fetch address signals to the instruction source and for coupling to the execution pipeline a sequence of instructions provided by the instruction source in response to the sequence of fetch address signals, a method for

executing branch instructions in parallel with the execution of instructions by the execution pipeline, the method including the steps of:

(a) detecting all of said branch instructions in the instruction sequence by decoding each of said branch instructions and issuing a prediction signal for each of said decoded branch instructions to indicate a predicted outcome for the decoded branch instruction;

(b) removing from the instruction sequence ones of said detected branch instructions which only test conditions created by execution of previous ones of said instructions, thereby not requiring execution in the execution pipeline;

(c) in response to the prediction signal for each of said removed branch instructions, conditioning the fetch address signals to a predicted sequence corresponding to the predicted outcome indicated by the prediction signal;

(d) in synchronization with execution of other ones of said instructions of said instruction sequence in the execution pipeline, concurrently executing at least two of said removed branch instructions by testing condition signals according to the branch instruction to determine an actual outcome for the branch instruction;

(e) comparing the actual outcome for the branch instruction with the predicted outcome indicated by the prediction signal for the branch instruction; and

(f) if the predicted outcome is incorrect, conditioning the address signal to a refetch sequence corresponding to a correct outcome for the branch instruction; otherwise,

(g) continuing the sequence of fetch addresses and performing steps (a)-(f).

16. The method of claim 15, further including, after step (c):

(c1) generating an instruction address for the branch instruction;

(c2) generating a branch target address for the branch instruction; and

(c3) enqueueing the branch instruction, the branch instruction address, the branch target address, and the

prediction signal for the branch instruction; and, wherein step (d) includes, prior to executing, dequeueing the branch instruction enqueued in step c3.

17. The method of claim 15, further including, before step (d) generating a token for each of said branch instruction in the sequence of instructions and propagating the generated tokens in the execution pipeline; and

wherein step (d) is executed in response to the tokens propagating to a predetermined pipeline stage in the execution pipeline.

18. The method of claim 17, wherein first tokens are generated for the branch instructions which only test conditions created by execution of previous instructions and wherein step (d) is executed in response to the first tokens propagating to an address generation stage of the execution pipeline.

19. The method of claim 18, wherein second tokens are generated for the branch instructions which require pipelined execution in the execution pipeline to change the condition signals and wherein step (d) is executed in responses to the second tokens propagating to an execution stage of the execution pipeline, the execution stage following the address generation stage.

Description

CROSS-REFERENCE TO RELATED APPLICATIONS

The present United States Patent Application is related to the following co-pending United States Patent Applications:

(1) Application Ser. No. 07/519,382 filed May 4, 1990 now abandoned, continued as 08/013,982, filed Feb. 5, 1993 still pending, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al;

(2) Application Ser. No. 07/519,384 filed May 4, 1990 now abandoned, continued as 08/013,982, filed Feb. 5, 1993, still pending entitled "General Purpose Compound Apparatus for Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al;

(3) Application Ser. No. 07/504,910 filed Apr. 4, 1990, now U.S. Pat. No. 5,051,940, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al;

(4) Application Ser. No. 07/522,291, filed May 10, 1990 now U.S. Pat. No. 5,214,763, entitled "Compounding Preprocessor for Cache", the inventors being Bartholmew Blaner et al;

(5) Application Ser. No. 07/543,464, filed June 26, 1990 still pending, entitled "An In-Memory Preprocessor for a Scalable Compound Instruction Set Machine Processor", the inventors being Richard J. Eickemeyer et al;

(6) Application Ser. No. 07/543,458, filed Jun. 26, 1990, now U.S. Pat. No. 5,197,135, entitled "Memory Management for Scalable Compound Instruction Set Machines with In-Memory compounding", the inventors being Richard J. Eickemeyer et al;

(7) Application Ser. No. 07/619,868, filed Nov. 28, 1990, still pending, entitled "Overflow Determination for Three-Operand ALUS in a Scalable Compound Instruction Set Machine", the inventors being Stamatis Vassiliadis et al;

(8) Application Ser. No. 07/642,011, filed Jan. 15, 1991, still pending, entitled "Compounding Preprocessor for Cache", the inventors being B. Blaner et al. (A continuation-in-part of U.S. Patent Application Ser. No. 07/522,291, filed May 10, 1990).

(9) Application Ser. No. 07/653,006, filed Feb. 8, 1991, still pending, entitled "Microcode Generation for a Scalable Compound Instruction Set Machine", the inventor being Thomas L. Jeremiah.

These co-pending applications and the present application are owned by one and the same assignee, namely, INTERNATIONAL BUSINESS MACHINES CORPORATION of Armonk, New York.

The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

BACKGROUND OF THE INVENTION

This invention concerns the operation of digital computers, and is particularly directed to the processing of branching instructions in a digital computer employing pipelined instruction processing.

Branch instructions can reduce the speed and efficiency of pipelined instruction processing. This deleterious effect has an even greater impact on the performance of processors with multiple pipelines processing a single instruction stream. Such processors include those referred to as "scalable compound instruction-set machines" (SCISM). A machine with SCISM architecture is taught in detail in the cross-referenced patent applications.

Branch prediction schemes have been proposed to reduce the performance penalty extracted by branch instruction execution. Two such schemes are of interest. The first involves dynamic prediction of branch outcomes by tagging branch instructions in an instruction cache with predictive information regarding their outcomes. See, for example, the article by J.E. Smith entitled "A Study of Branch Prediction Strategies", in the March 1981 PROCEEDINGS of the Eighth Annual Symposium on Computer Architecture. SCISM architecture which provides spare information capacity in an instruction stream is particularly adapted for this scheme. In this regard, a bit, called a "compounding" or "C"-bit, is provided in every half word of a SCISM instruction stream. Whenever one of these bits follows an instruction which can be executed in parallel with ("compounded with") the following instruction, the C-bit in the first halfword of the first instruction is set to indicate this capability. If the instruction is longer than one-half word, the compounding provides unused bits for the extra halfwords. The C-bit or bits which are not used in the compounding scheme are available for alternate uses. For branch instructions, one such use is the prediction of the outcome of a branch instruction.

A second branch prediction strategy involves the use of a branch target buffer (BTB) which contains a history of the outcome of executed branch instructions. When a branch instruction is first executed, its outcome is stored in the BTB. When the branch instruction is executed a second time, its predicted outcome is the outcome stored in the BTB. Such a mechanism is described in detail in the article by J.K.F. LEE ET AL, entitled "Branch Prediction Strategies in Branch Target Buffer Design" in the January 1984 issue of IEEE COMPUTER.

While both dynamic prediction and branch target buffer mechanisms do speed up a pipeline which executes an instruction stream including branch instructions, the techniques involved have not been adapted for application in a SCISM architecture. Further, the advantages of parallelism enjoyed in multiple-pipeline architectures have not yet been realized in processing branch instructions.

SUMMARY OF THE INVENTION

The invention is based upon the inventors' critical observation that branch instructions which do not require execution unit operations can be extracted from an executing instruction stream and held for branch condition testing in synchronism with execution of the execution stream.

A significant objective of this invention, is, therefore, to increase the parallelism of digital computers with multiple execution pipelines.

This objective, and other significant objectives and advantages, are achieved in a digital computing machine which includes an instruction source, an execution pipeline for executing a stream of instructions, and a condition code determination mechanism connected to the instruction pipeline for providing condition signals indicating results of executed instructions. In this context, the invention is a combination for processing branch instruction conditions, and the combination includes:

an instruction issue mechanism coupled to the instruction source and to the execution pipeline for providing a sequence of instructions for pipelined execution; and

a branch condition processor for:

removing branch instructions from the sequence of instructions; and

in response to the condition indication signals, executing removed branch instructions substantially in parallel with execution of instructions by the execution pipeline.

BRIEF DESCRIPTION OF THE DRAWINGS

The achievement of this invention's objectives can be understood with reference to the detailed description of the preferred embodiments of the invention and to the drawings, in which:

FIG. 1 is a block diagram with a multiple pipeline, scalable compound instruction set machine (SCISM);

FIG. 2 is a schematic drawing of compounded instruction formats showing two arrangements for providing compounding information with compounded instructions;

FIG. 3 is a chart showing the statistics of branch instructions in a model instruction trace;

FIG. 4 is a general block diagram of an instruction fetch and issue unit and branch processing unit in the SCISM architecture of FIG. 1;

FIG. 5 is a more detailed block diagram of a first embodiment of a branch processing unit;

FIG. 6 is a table showing three prediction categories for a set of branch instructions;

FIG. 7 shows in block diagram form the branch address mechanism of a second embodiment according to the invention;

FIG. 8 illustrates a branch queue for a second embodiment;

FIG. 9 illustrates a two-phase unload pointer for the second embodiment;

FIG. 10 illustrates a branch test unit;

FIG. 11 is a digital logic schematic of the branch test unit;

FIG. 12 illustrates how branch instructions are entered into extracted from a branch queue; and

FIG. 13 is a timing diagram illustrating the operation of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIG. 1 illustrates a SCISM architecture in which microprogramming is used to implement and control the execution of machine-level instructions singly or in pairs. In particular, a machine-level instruction stream is provided to a compounding preprocessor 10. The instruction stream is a sequence of individual instructions which have typically been compiled from a source program. The stream is provided to the CPU of a SCISM computer for execution. Conventionally, the machine-level instructions are staged into the CPU through a cache (compound instruction cache) 12. Before entry into the cache, the instruction stream is examined by the compounding preprocessor 10 to determine whether adjacent instructions can be concurrently executed. The compounding preprocessor 10 is described in detail in the eighth referenced co-pending U.S. Patent Application, entitled "COMPOUNDING PREPROCESSOR FOR CACHE" also described in the referenced application is the structure of a compounding instruction cache which functions as a cache 12 of FIG. 1.

The operation of the compounding preprocessor 10 results in the generation of compounding information indicating whether adjacent instructions of the compiled instruction stream which have been cached at 12 can be concurrently executed. Thus, for each instruction, the compounding preprocessor 10 generates compounding information indicating whether that instruction and an adjacent instruction can be executed in parallel.

Following processing by the compounding preprocessor 10, the analyzed instructions and the compounding information are stored in the compounding instruction cache 12. Except for the provision of extra space to store the compounding information, it is asserted that the cache 12 is operated conventionally. In particular, entries in the cache 12 are typically groups of adjacent instructions ("lines") which are entered into the cache so that they can be quickly obtained as required by an executing program.

In providing the compounding information together with instructions in the cache 12, the SCISM architecture takes fuller advantage of parallelism than a computer which makes the parallel execution decision when instructions are brought out of the cache for immediate execution ("issued"). Relatedly, an instruction in the cache 12 may be used more than once in, for example, a loop or branch. So long as the instruction is in the cache, it does not have to be reanalyzed if obtained again for execution because the compounding information for the instruction which is stored with it in the cache can be reused.

The compounding preprocessor 10 is assumed to be of the type described in the referenced patent application and generates, for each instruction, at least one C-bit. These C-bits are used to identify which pairs of instructions may be processed in parallel. Instructions and their C-bits are supplied and stored into the compound instruction cache 12. An instruction fetch and issue unit 14 fetches the instructions and their C-bits from the compound instruction cache 12, as needed, and arranges for their execution by the appropriate one or ones of a plurality of execution units 34, 36.

The information fetched from the cache 12 includes instruction text and associated compounding information. Each fetch is in the form of a quadword (that is, a four-word line) as indicated by reference numeral 13a and includes eight halfwords numbered 0, 1, ... 7. Associated with each quadword of instruction text is a 1.times.8 array of C-bits comprising the compounding information produced by the compounding preprocessor 10. The bits are referred to as a "C-vector".

FIG. 2 is a more detailed illustration of the results of the operation of the compounding preprocessor 10. If it is assumed that the structure of FIG. 1 is included in a digital computer of the System/370 type available from the assignee, then it will be realized that the instruction stream includes instructions which can vary in length from one to three halfwords. For operational efficiency, the preprocessor 10 generates a C-bit for every halfword stored in the cache. The C-bit for the first halfword of an instruction is conditioned to indicate whether the following instruction is to be executed in parallel with the instruction. A set C-bit associated with the first halfword of an instruction indicates that the instruction has been "compounded" with the following instruction and that the two instructions are to be executed in parallel. Otherwise, the instructions are executed singly.

Instructions and associated compounded information are fetched from the cache 12 by an instruction fetch and issue unit 14 which includes a branch processing unit 15. Control information for instruction fetching is fed in the form of a signal FETCH which is conditioned to indicate that a quadword of instruction text is to be obtained from the cache 12 and provided to the instruction fetch and issue unit 14. When the signal is conditioned to indicate a fetch, it is accompanied by a fetch address of the next instruction to be processed by the instruction fetch and issue unit 14 for execution. In this regard, the fetch address could be the address of the next instruction in the portion of the instruction sequence following an instruction in the fetch and issue unit 14. The fetch address could also be an address in the instruction sequence which is pointed to by an executed branch instruction. In any event, the FETCH and fetch address are provided to a cache management mechanism 16 which maps the fetch address to a cache address if the instruction is in cache or which provides the address to a real memory management mechanism together with a MISS indication if the instruction is not in the cache.

In the description which follows, it is assumed that the SCISM architecture supports the simultaneous execution of pairs of instructions in the sequence of machine-level instructions provided by way of the preprocessor 10 and cache 12 through the instruction fetch and issue unit 14. Relatedly, a compound instruction register including left and right portions receives issued instructions and positions them for single or parallel execution. The instruction which is next to be executed is entered by the instruction fetch and issue unit 14 into the left-hand compound instruction register (CIRL) 20 which includes register portion 22 for storage of the C-bit which indicates compounding for the instruction. If the C-bit is conditioned to indicate that the instruction in the CIRL 20 is to be executed with the immediately following instruction, that instruction is placed by the fetch and issue unit into the right-hand compound instruction register (CIRR) 21. On the other hand, if the C-bit indicates that the instruction in the CIRL 20 is not compounded with the following instruction, and the contents of the CIRR 21 are disregarded.

A microcode generator 24 translates the contents of the compound instruction registers into microcode which is staged through an execution pipeline consisting of an address generation instruction register (AIR) 25, an execution instruction register (EIR) 26, and a put-away instruction register (PIR) 27. Instruction fetch and issue unit 14, compound instruction registers 20 and 21, microcode generator 24, and lower pipeline stages 25-27 form an instruction execution pipeline which is synchronized by a pipeline clock provided by a pipeline clock generator 28.

The sequence of operations used in the SCISM architecture to execute machine-level instructions is partitioned into five pipelined stage, with the intent of feeding subsequent instructions into the pipeline each cycle of the pipeline clock. The first stage, IF, is instruction fetch which occurs when an instruction is fetched from the cache 12 or from an instruction buffer residing in the fetch and issue unit 14. At the end of the IF cycle, the instruction is ready to be loaded into the compound instruction registers where it will be decoded to start instruction execution. If the C-bit of the instruction indicates that the next following instruction is to be executed in parallel with the current instruction, both instructions are available at the end of the IF cycle, with the first instruction and the C-bit being available for loading into the CIRL 20 and the following instruction into the CIRR 21. The instruction decode (ID) cycle is controlled by logic decoding of CIRL and CIRR 20 and 21, respectively. The generation of the first microword needed to control subsequent stages of the pipeline occurs in this cycle and consists of accessing a control store included in the microcode generator 24 using instruction op codes as microword addresses. The address generation (AG) cycle is used to calculate an effective address for operands needed from storage. The execution (EX) cycle is used to do operations in one or more execution units. The put-away (PA) cycle is used to store results from the EX cycle into the general purpose registers (GPR) 37.

As the ninth-referenced patent application teaches, the microcode generator 24 consists of a main control store (MCS) and a secondary control store (SCS) to form a merged microinstruction. During the ID cycle, the contents of the CIRL 20 are used to address the MCS. If the C-bit in register field 22 indicates that the CIRR 21 contains an instruction which is compounded with that contained in the CIRL 20, the op code of the instruction in the CIRR is gated by the C-bit to address the SCS. The microinstruction output by the microcode generator 24 contains all the fields necessary for controlling the execution of two instructions in parallel. Certain of the fields of the microinstruction are reserved for execution of the first instruction, while certain other fields are reserved for execution of the second instruction. If only a single instruction is presented to the microcode generator 24, that instruction is resident in the CIRL 20 and results in generation of a microinstruction sequence in which the fields for execution of a second instruction are set to default values, while the fields for the first instruction are appropriately set to execute it. If instructions are contained in the CIRL 20 and CIRR 21, the output of the SCS is merged with the output of the MCS by entering into the second fields the information output from the SCS.

The microinstruction sequences output by the microcode generator 24 are fed to the last three pipeline stages 25-27.

In the pipeline, conventional horizontal microcode execution is undertaken by generation of controls at each stage of the pipeline which operate either the execution unit 34 or the execution units 34 and 36 for conventional instruction execution with operands and results transferred between the execution units and the GPR 37. In addition, pipelined sequence controls including an END OP bit signifying the end of a microinstruction sequence are fed back to the instruction fetch and issue unit in order to maintain the sequence of instructions fed to the compound instruction registers.

In the operation of the invention, two additional control signals (also "tokens"), BC and BR, are fed back to the branch processor 15 from the execution pipeline 25-27 for control of operations which are explained below.

The results of instruction execution are provided from the execution unit 34 if a single instruction is executed, or from the execution units 34 and 36 if a pair of instructions are executed to a branch condition generator 39. The branch condition generator operates conventionally to generate a condition code (CC) and other branch conditions resulting from execution of branch instructions. These condition signals are conducted from the branch condition generator 39 on the pathway 41 back to the branch processor 15 for use which is described below.

BRANCH INSTRUCTION EFFECTS

The technique of pipelined instruction processing is widely known and employed in computer design, offering a manifold performance acceleration if conditions conducive to pipelining can be met and sustained. This benefit is compounded in the SCISM architecture illustrated in FIG. 1 by provision of a pipeline structure which operates a pair of pipelines in the event that two instructions are to be executed in parallel. Unfortunately, the characteristics of program behavior are such that the conditions for pipelining frequently go unmet, thus reducing the speedup actually attained when pipelining is employed.

A primary inhibiting characteristic is a comparatively high frequency of branch instructions found in programs. Up to 25% or more of all instructions executed in a given program may comprise branch instructions. Branch instructions disrupt pipeline operations by introducing unused cycles into a pipeline when a decision must be made, based on a prior or coincident execution result, to either fetch a new (target) instruction stream or continue executing the current instruction stream in sequence. Delay in making the decision and fetching the appropriate stream introduces more unused cycles and further diminishes the performance achieved by pipelining.

The performance-diminishing effect of branch instructions is amplified in the SCISM architecture exemplified in FIG. 1. Execution of multiple instructions in parallel from a single instruction stream makes the executing machine particularly susceptible to the adverse effects of branches, because not only will branch instructions cause multiple pipelines to stall, but since the SCISM architecture consumes instructions between branches at a higher rate, at any instant in time, the likelihood of a branch instruction entering the pipelines is greater than otherwise.

More particularly, the presence of a branch instruction in a pipeline raises the uncertainty of the address of the instruction following the branch instruction. Ideally, instructions are executed in a continuous sequence which enables the address of the next instruction to be determined simply by adding the length of the current instruction to its address. This is termed "sequential addressing". The address of the next instruction following the branch instruction may be called the "sequential address". Alternatively, execution of a branch instruction may result in movement of programming execution to a nonsequential instruction address. This address is termed "branch target address". Whether a sequential address or a branch target address follows the address of the branch instruction depends upon the outcome of a test which is implicit in the branch instruction. If the test conditions are met, the instruction stream branches to the branch target instruction; if the branch conditions are not met, the next address following the branch instruction is the sequential address.

In pipelined instruction execution, the uncertainty inherent in a branch test can be met by preventing the issuance of any instructions until the branch test is completed, upon which the correct address can be determined and the corresponding instruction fetched. However, this introduces a delay in the pipeline following every branch instruction. Alternatively, the outcome of a branch can be predicted and the predicted instruction can be fetched and placed in the pipeline immediately following the branch instruction.

If the predicted outcome of a branch instruction is that the branch is not taken, instruction fetching can proceed in the normal fashion simply by incrementing the address of the branch instruction by its length and issuing the next following instruction. Provision of an instruction buffer in an instruction fetch and issue unit utilizes the updated address to validate the sequential contents of the buffer which includes the instruction sequence following the branch instruction in the stream. Alternatively, if the branch target address is predicted as the outcome, the buffer must be cleared and a sequence of instructions beginning at the predicted branch target address must be "prefetched" from a cache and placed into the instruction buffer of the fetch and issue unit.

If, after execution of the branch instruction, the outcome of the branch test indicates that the incorrect prediction has been made, the contents of the instruction buffer must be cleared and a "refetch" of the correct instruction sequence must be brought into the buffer.

The bit-prediction and branch target buffer mechanisms mentioned in the Background hereinabove are the two principal mechanisms for dynamically predicting the outcome of a branch instruction.

In the first mechanism, any cached branch instruction is accompanied in a cache by one or more bits which are used to predict the outcome of the instruction. In this regard, the information is conditioned to indicate the last outcome of the branch instruction.

In the second mechanism, a table ("branch target buffer") is maintained apart from the instruction cache in which the history of recently executed branch instructions is maintained. Preferably, the addresses of the branch instructions in the cache are used to address a branch target buffer which has, for each cached branch instruction, a record of its last outcome and the branch target address. For branches which have not been executed but which are cached, a predetermined outcome, based on a statistical profile of the branch instruction in the same or similar programs, is made for the instruction when it is first brought from the cache for instruction issue. These mechanisms and strategies for implementing them are considered in detail in the articles by Smith and Lee et al referenced above.

In SCISM architecture where instructions may include one or more unused compounding bits, the unused bits may be used advantageously for predicting the outcome of the branch instructions. Thus, for any branch instruction which includes two or more halfwords, the extra bit or bits produced by the compounding preprocessor are stored with the instruction and cache and can be used for predicting the outcome of the instruction. Preferably, the rules implemented by the compounding preprocessor for determining whether instructions are to be compounded would not compound single halfword branch instructions, thereby freeing the single C-bit for service in branch outcome prediction.

The table of FIG. 3 lists branch instructions for the extended system architecture (ESA) version of the System/370 digital computer product available from the assignee. These instructions, their formats, their functions, and their tests can all be understood with reference to, for example, the book by C.J. Kacmar entitled IBM 370 ASSEMBLY LANGUAGE WITH ASSIST, Prentis Hall, 1988, especially chapter 6. The table of FIG. 3 also gives, for a representative instruction trace, the frequencies of the System/370 branch instructions relative to all instructions, their frequencies relative to branch instructions only, their frequencies of taken and not taken branches, and their lengths in half words. For example, the BC instruction constitutes 17.23% of the instructions of the representative trace and constitutes 68.6% of all branching instructions in the trace.

BRANCH INSTRUCTION ISSUE PROCESSING

The invention concerns the processing of branch instructions in the central processing unit (CPU) of a computer which includes multiple parallel pipelines for executing instructions either singly or in parallel. The inventors have made the critical observation that parallelism of such a CPU can be enhanced by removing from its pipeline branch instructions which require no execution unit operation, but which only test conditions created by execution of previous instructions. Condition testing for these instructions is accomplished in correct synchronism with the execution of other instructions by advancing indication signals in the execution pipeline of the CPU. All other branch instructions are entered into the pipeline, where their locations are tracked with similar indication signals. Provision of a multiple pipeline raises the possibility that the conditions of more than one branch instruction can be tested in the same pipeline cycle, thereby extending inherent parallelism of multiple pipeline architecture which underpins parallel instruction execution.

The invention is practiced in the branch processing unit 15 of the instruction fetch and issue unit which is illustrated in an environmental context in FIG. 1 and which is presented in greater detail in FIG. 4, together with portions of the execution pipeline. All storage elements of these units, including registers, pointers, and state machines change their outputs in response to the pipeline clock.

In FIG. 4, the instruction fetch and issue unit 14 provides the pipelined execution circuitry with a stream of instructions for processing. Interruption of the stream is reduced by predicting the outcome of branch instructions and prefetching the predicted stream into an instruction buffer (IB). The prefetched instructions are fed to the execution pipeline for execution until the true outcome of the branch instruction can be determined. At this point, if the branch direction was predicted incorrectly, execution of the prefetched stream can be cancelled. In the case of a misprediction or incorrect branch target address, the correct instruction will be fetched into the decode stage of the execution pipeline and the following instructions will be fetched into the IB.

The instruction fetch and issue unit, indicated by reference numeral 14 in FIG. 4, consists of an alignment unit (ALIGN) 50, an instruction buffer 52, an instruction buffer output aligner 55, and a multiplexer/formatter unit 60 provided for multiplexing and formatting instructions for placement in the compound instruction registers.

Instruction fetch and issue unit 14 issues instruction fetches to the cache management unit 16, providing a FETCH signal and a fetch address. Each fetch address addresses four contiguous words (a "quadword") and their associated C-bits. The quadword and associated C bits are provided to the alignment unit 50 where the text is rotated to align the 1st halfword fetched to the left boundary. This aligner output is sent directly to the CIR formatter 60 for possible destination to the CIR registers 20 and 21. The same text is then rotated to the right based on the current IB load pointer 53 value. This completes the text alignment functions and allows the first halfword from the fetch to be gated into the first available IB register 52 location, and the remaining halfwords to be entered into the remaining available locations. As is known, in the IBM System/370 instruction set, the instruction format provides, in the first two bits of the instruction OP code, the length of the instruction in half words. Thus, as the first instruction is aligned with a reference point, its length code gives the location of the beginning of the following instruction.

The registers of the instruction buffer 52 are not illustrated in FIG. 4, it being assumed that the instruction buffer is of conventional design and is loaded under control of a load pointer 53. The load pointer is a cycling counter which is incremented each time an instruction is loaded into a register of the IB to point to the next available register.

Unloading of the IB registers 52 is by way of the instruction buffer output aligner 55 under control of an unloader 57. The output aligner 55 rotates IB instruction text being pointed to by the output pointer 57 leftward to a boundary. Beginning from the boundary, five half words of instruction text along with corresponding C-bits are sent to the CIR multiplexer/formatter 60 for formatting, the final destination being the compound instruction registers 20 and 21.

The CIR multiplexer/formatter 60 receives the five half words of the instruction text with the purpose of producing up to two instructions for registration in the compound instruction registers 20 and 21. In addition to the instruction text, the multiplexer/formatter also receives a C-bit for each half word. The formatting portion of the unit then inspects the length of the first instruction, along with the controlling C-bit. If the controlling C-bit for the instruction indicates that it is compounded with the next following instruction, that instruction is loaded into the CIRR 21. Preferably, the "controlling" C-bit for an instruction is the C-bit associated with the first halfword of an instruction.

The CIR multiplexer/formatter 60 can be sourced from either the alignment unit 50 or the output aligner 55, depending upon whether the instruction buffer is empty or not. If empty, the first five halfwords of the fetched quadword are forwarded to the multiplexer/formatter directly from the alignment unit 50 while the remaining half words are fed to the IB. This avoids delay in filling the pipeline during an initial fetch of instructions or during a refetch which clears the pipeline.

Preferably, the IB 52 is one and one-half quadwords in length and contains prefetched instruction text. The text may not be from sequential storage locations as the branch processor may prefetch target streams of instructions for branches predicted to be taken. These branch target streams will be loaded into the IB at their logical execution points, overwriting the sequential instruction stream that may have been there. Two pointers are associated with the IB 52, the load pointer 53, and the unload pointer 57.

The load pointer 53 is a circular counter whose value is incremented each time new text is loaded into the IB 52. For branch target loading of the IB 52, the load pointer value is set to point at the halfword of instruction text following the branch instruction that initiated the target fetch. The count of the load pointer 53 is given by the signal IBLPTR.

The unload pointer 57 always points to an instruction boundary in the IB 55 and will be advanced by an amount equal to the length of instruction text loaded into the compound instruction registers. Advancing of the value of the pointer will be accomplished by calculating the length of text being loaded into the compound instruction registers and adding it to the current value. The current value of the unload pointer is given by the signal IBULPTR.

In operation, as successive halfword portions of an instruction sequence are fed to the multiplexer/formatter 60, they are inspected and loaded into the compound instruction registers with two exceptions. When the multiplexer/formatter encounters either a BC or BCR instruction, it increments to the boundary of the next instruction, without validating the BC or BCR instruction for entry into the compound instruction registers. This is accomplished by a pair of decoders 64 and 65. The decoder 64 receives the op code of the instruction which is on the current left-hand boundary of the multiplexor/formatter, while the decoder 65 receives the op code of the instruction immediately following it. When either decoder detects either a BC or BCR instruction, it activates an output. The outputs are fed back to the multiplexer/formatter unit and prevent the loading into the compound instruction registers of the decoded branch instruction. All other branch instructions are loaded into the compound instruction registers. In this manner, the BC and BCR instructions are removed from the instruction pipeline, without introducing unused cycles. However, the presence of a BC or BCR instruction in the instruction stream is signified by the output of either of two latches 66 and 67. The latch 66 receives the output of the decoder 64, while the latch 67 receives the output of the decoder 65. The latches 66 and 67 are clocked by the pipeline clock and remain set for only one period of that signal. The same pipeline clock cycle enters instruction text into the compound instruction registers; therefore, the contents of the latches 66 and 67 correspond with the instruction decode (ID) stage of the pipeline.

Branch instructions which are entered into the compound instruction registers 20 and 21 are decoded at 70 and 72 during the instruction decode cycle. In the AGEN stage of the pipeline, there are provided four latches, 73, 74, 75, and 76 which respectively receive outputs from the latch 66, the decoder 70, the decoder 72, and the latch 67. Last, a pair of latches 79 and 80 are provided in the execution stage of the pipeline to receive the contents of the latches 74 and 75 in the address generation stage.

The latch sequence 66, 73 advances through the ID and AGEN stages of the pipeline a token indicating the presence of a BC or BCR instruction in the "left-hand" side of the pipeline, corresponding to that portion of the pipeline controlled through the CIRL 20. The token, termed a "BCL" token, is advanced through these two pipeline stages in synchronism with the operation of the pipeline. Similarly, the latches 67 and 76 advance a "BCR" token on the right-hand side of the pipeline for BC and BCR instructions which, but for being removed, would have been fed into the pipeline through the CIRR 21. Generally, BCR and BCL tokens are referred to as "BC" tokens; they signify decoding of BC or BCR instructions.

A token is advanced through the AGEN and EX stages of the pipeline by latches 74 and 79 for all branch instructions which are entered into the pipeline through the CIRL 20. Such tokens are called BRL tokens. Similarly, BRR tokens are carried through the AGEN and EX stages of the right-hand side of the pipeline for branch instructions entering the pipeline through the CIRR 21. Generally, BRR and BRL tokens are referred to as "BR" tokens; they signify decoding of branch instructions other than BC and BCR instructions.

The operations of the execution unit 34 and 36 are controlled conventionally by the pipeline to perform calculations and logical operations required by instructions in the pipeline. The results of instruction execution are fed to a condition code unit 83 which sets the condition code (CC) according to the outcome of instruction execution. A branch condition (BC) unit 85 provides branch condition signals in response to the outputs of the execution units 34 and 36.

Pipeline operations are synchronized with instruction issue by an AND gate 88. The gate 88 receives and END OP bit at the output of the MCS which is combined with the output of a pipeline condition sensor 87. The sensor 87 monitors the AGEN and EX stages of the execution pipeline for conditions requiring interruption of pipeline operations. For so long as no inhibiting conditions exist, the gate 88 activates a signal, NEXT, whenever the END OP bit of the currently accessed microinstruction is set, indicating the end of the microinstruction sequence for the current contents of the CIR's. When NEXT is activated, the multiplexer formatter is enabled to enter the next instruction or instruction pair into the compound instruction registers and the decoders 64, 65, 70, and 72 are enabled.

BRANCH CONDITION PROCESSING

In this description, branch condition processing consists of detecting a branch instruction in the current instruction stream, predicting whether or not the branch is taken, prefetching the instruction text for a branch which is predicted as being taken, performing the branch test inherent in the branch instruction, and issuing a corrected instruction "refetch", if necessary.

The first step is done by searching all instruction text before it is loaded into the first stage of the execution pipeline. This detection is done at the input to the IB 52 for the first five half words of a prefetched or refetched quadword, and the output of the IB 52 for all other instruction texts. As each branch instruction is detected, its text and location in the instruction sequence are provided to the branch processing unit 15. Based on the decoding of the branch instruction, the branch processing unit predicts whether or not the branch will be taken and initiates the necessary fetch or prefetch according to the prediction. The branch instruction location information is used to generate the address of the branch instruction in the cache. The branch target address (BTA) is generated. Next, the branch instruction text, its address, and a prefetch (PF) indicator are entered into a branch queue, which is operated as a FIFO queue. The PF indicator is conditioned to indicate whether or not a prefetch at the branch target address was made. In this regard, the PF indicator is a token which represents whether the branch was predicted as taken (PF= 1) or not (PF=0). If taken, of course, a prefetch has occurred; if not, sequential fetching has continued.

Branch instructions are dequeued from the BQ in response to BC and BR tokens in the execution pipeline. Up to two instructions can be dequeued simultaneously. Dequeued instructions are tested in a branch test unit which utilizes the CC and branch conditions generated in response to execution unit operation to determine whether a branch is to be taken or not. The outcome is compared with the PF indicators of the dequeued instructions to determine whether a refetch must be made or not. If a refetch is required, the correct refetch address is generated and made available to the cache 12. It is asserted that the execution pipeline is constructed to be cleared in the event of a refetch; this is well within the ambit of the prior art.

The branch processing unit includes a decode and boundary detection unit 92 which receives instruction text and instruction location information either from the align unit 50 or the IB 52 of the instruction fetch and issue unit 14. The source is selected through a multiplexer (MUX) 90 in response to a sequential (S) or prefetch (PF) signal. The PF signal is active when a prefetch is being conducted, in which case the contents of the IB 52 will be overwritten with the prefetched instructions. This results in the provision of five halfwords of instruction text and 0 associated C-bits from the alignment unit 50, following which the PF signal will deactivate and the S signal will select from the IB 52 the five halfwords currently being selected by the align unit 55 for multiplexing and formatting into the compound instruction registers. The output of the alignment unit 50 is searched during a prefetch cycle since cache interface text will be provided beginning on an instruction boundary. This is a requirement for any detection, as instruction boundaries must be known in order to perform instruction decodes. During sequential fetching, the instruction text is obtained from the IB 52 where branch instructions may be detected in a manner similar to that employed by the align unit 55. When a branch instruction is detected, it is gated into a branch instruction register (BIR) 98 by a control signal (BR DEC). In searching for branch instructions, the decode/boundary detection circuitry 92 inspects every instruction which is streamed into the instruction processing unit. Recall that each instruction contains an indication of its own length. For each instruction that it inspects, the decode/boundary detection unit 92 generates an update amount corresponding to the length of the next instruction to be analyzed. This amount is registered at 93. Detection of a branch instruction by the unit 92 is indicated by a signal (BR DEC) which gates the instruction into the branch instruction register 98. The address of the branch instruction in the register 98 is provided by a branch instruction address generation (BIAG) unit 99 which is updated continuously from the last prefetch or refetch address by the stream of update amounts registered at 93.

A prediction decode circuit 96 responds to the decoding of a branch instruction by issuing a signal (PREDICT) which is conditioned to indicate the predicted outcome of the branch instruction as taken, in which case a branch target address (BTA) must be provided for prefetching, or not taken, in which case the next sequential address must be provided. The PREDICT signal is fed to an address prediction circuit 94 which generates the predicted address. The predicted address is fed through a multiplexer (MUX) 95 to an instruction fetch address register 97 which is available to the cache management mechanism for instruction fetching. Whether the branch outcome is predicted as taken or not taken, the address prediction circuit 94 also generates the branch target address (BTA) which is available, together with the branch instruction address (BIA) and PF token when the branch instruction text is available from the BIR 98.

Branch instructions which have been identified in the instruction stream and decoded by the branch processing unit 15 are queued in a FIFO branch queue (BQ) 102. Each time a branch instruction is registered at 98, the instruction text, together with the BTA BIA and PF token is entered into the branch queue 102.

Branch queue entries are dequeued in response to the BC and BR signals propagated in the execution pipeline. As FIG. 4 illustrates, the BQ 102 has two outputs 102a and 102b which permit up to two branch instructions to be dequeued concurrently.

The dequeued instructions are fed to a branch test unit 103 which receives the CC and branch condition signals from the execution units. The branch test unit uses these conditions to determine the proper outcome for a dequeued instruction, compares the determined outcome with the predicted outcome, and (if necessary) generates a REFETCH signal indicating that the branch outcome was incorrectly predicted.

If the REFETCH signal is active during any pipeline clock period, the branch queue 102 will provide branch instruction address and branch target address information stored with the incorrectly-predicted branch instruction to a refetch address circuit 105. The refetch address circuit 105 uses the address information stored with incorrectly predicted branch instructions to generate correct instruction fetch addresses. When the REFETCH signal is active, the multiplexer 95 selects the output of the refetch address unit 105, resulting in registration of the corrected address in the instruction fetch address register 97.

An instruction fetch sequencer 106 in the branch processing unit receives the outputs of the load and unload pointers 53 and 57 in the instruction issue unit, the PREDICT signal from the prediction decode unit 96, and the REFETCH signal from the branch test unit. The sequencer 106 responds to the condition of the PREDICT signal to set either the FETCH or PREFETCH signals. The FETCH signal, of course, indicates to the instruction fetch and issue unit that a sequential fetch is progress, resulting in sequential incrementation of all pointers and aligners in the unit. If the PREFETCH signal is active, pointers and aligners in the instruction fetch and issue unit are initiated to the prefetch address available in the instruction fetch address register 97. Whenever the REFETCH signal is active, indicating an incorrect prediction of a branch instruction, the sequencer 106 produces a signal (INVALID) not shown in FIG. 4 invalidating the current content of registers, aligners, and pointers in the instruction fetch and issue unit and the contents of the instruction pipeline.

During sequential fetching, the sequencer 106 will maintain the FETCH signal active for so long as the IB 52 has room to accept instruction text. However, if the load pointer 53 advances to a value equal to the value in the unload pointer 57, the IB 52 is considered full and the FETCH signal is deactivated.

An OR circuit 107 funnels the REFETCH, FETCH, and PREFETCH signals to the cache as a cache fetch (CFETCH) signal. When the CFETCH signal is activated, the contents of the instruction fetch address register 97 are validated and the cache management initiates a fetch from the cache at the address in register 97.

BRANCH PROCESSING UNIT FIRST EMBODIMENT

FIG. 5 illustrates a first embodiment of the branch processing unit 15 in which branch prediction is by way of C-bits which are generated by the compounding preprocessor 10 (FIG. 1). In this embodiment, branch instructions, together with at least their controlling C-bits, are stored in the branch instruction register 98. The branch target address is unconditionally generated, and the C-bits for the decoded branch instructions are used to determine whether the branch target address will be sent to the cache. If a branch is predicted to be taken, the branch target address is sent to the cache and a prefetch is begun. The prefetched instruction text is divided on instruction boundaries as described above and placed in the instruction buffer following the branch instruction just decoded. The address of the instruction sequentially following the branch instruction is saved in the BQ in case the branch was wrongly predicted.

If the branch is predicted not to be taken, then the branch target address is unconditionally generated, but is only saved in the BQ. The branch processing unit sends the address of the instruction text following the last valid instruction text in the IB to initiate a sequential fetch in case a location becomes available in the IB on the next cycle. In other words, the branch processing unit attempts to fill the instruction buffer with sequential instructions whenever it fails to encounter a predicted taken branch. In this manner, the branch processing unit 15 creates a single instruction stream to be processed by the execution unit and by itself, based on a predicted branch path.

In understanding this embodiment and the following embodiment, it is asserted that the instruction buffer 52 is loaded and unloaded under the control of pointers. Thus, while it may contain more than one branch instruction at one time, it is unloaded serially so that only a single instruction is presented at any time to the BIR 98 and branch processing unit for processing and enqueueing. Thus, in FIG. 5, the branch instruction currently residing in the BIR 98 is fed to a decoder 204 which inspects the OP code of the instruction to determine its format. For example, if the instruction is an RR type, it contains two register designations for operands, one of which is the branch target address. On the other hand, if the register is a RX type, it contains an address given by D(XB). The decoded instruction type controls a 3-to-1 adder 202 which receives inputs from a copy 200 of the GPR which is addressed by the register fields of the instruction. If the instruction is an RR type, one operand output from the GPR copy 200 is added to an implied zero; if the instructions is an RX type, the GPR copy operands denoted in the X and B fields of the instruction are added to the displacement in the D field of the instruction. The adder 20 is similarly controlled by the output of the decoder 204 to correctly add the relevant fields of an RS type instruction to produce the branch target address.

The branch processing unit also includes a prefetch address register (PFAR) 206 and a branch instruction address register (BIAR) 209. When an instruction sequence is initially loaded into the instruction fetch and issue unit, or when a prefetch or refetch operation is undertaken, the contents of the PFAR 206 and BIAR 209 are initialized to the address of the fetched instruction. Thereafter, every time a quadword is sequentially fetched, the contents of the PFAR 206 are updated at 207 by the length, in half words, of the fetch. Thus, the PFAR always points to the beginning address of the quadword currently being fetched from the cache. The BIAR 209 is updated at 210 by the contents of the register UPD 93, which is the length between branch instructions. Therefore, the BIAR 209 always points to the address of the currently-unloaded branch instruction.

The branch queue is indicated by 102a in the first embodiment. In FIG. 5, the branch queue 102a consists of four registers 220-223 which are loaded and unloaded under control of load and unload pointers 234 and 232, respectively. Each time a branch instruction is decoded at 204, its OP code is placed in the OP field of the register indicated by the load control (LD CNTRL) signal output by the load pointer 234. While the OP code of a branch instruction is being loaded into the indicated register of the branch queue 102a, its instruction address and its branch target address are loaded, respectively, into the fields BIA and TGT of the register. The BIA field value is obtained from the output of the incrementer 210, while the TGT field value is obtained from the output of the adder 202. Also loaded with the OP code of the currently-unloaded branch instruction is the C-bit used to predict the branch outcome. Following loading of the register, the load pointer is incremented to the next register.

Refer now to the branch queue register 220 which shows the four fields C, OP, BIA, and TGT for, respectively, the predictive C-bit, the OP code, the instruction address, and the branch target address of a queued branch instruction. The output of the C and OP fields of the register are fed to a multiplexer (OPMUX) 228, as are the corresponding fields of the other queue registers 221, 222, and 223. In addition, the contents of the C field are inverted at 222, the inverted signal being used to operate a two-to-one multiplexer 224. Assuming that the C-bit is set to predict a taken branch, and reset to predict a branch not taken, inversion of the value will cause the multiplexer 224 to select the address for the non-predicted outcome. This outcome is available through a refetch multiplexer (RFAMUX) 226. Each of the remaining registers 221, 222, and 223 is similarly coupled to the RFAMUX 226 to provide the non-predicted sequence of the branch instruction in the respective register.

When a BCL or BCR token arrives at the AGEN stage of the execution pipeline, the related branch token is available from the latch 73 or 76, respectively. Similarly, the BRL and BRR tokens are available at the execution stage of the pipeline from the latches 79 and 80, respectively. The BCL, BCR, BRL and BRR tokens are used to increment the unload pointer 232. The unload pointer 232 sets the OPMUX 228 and RFAMUX 226 to select either the next one or next two of the branch queue registers for provision to the branch test circuit 103a.

The branch test circuit 103a conducts a branch test indicated by the OP code of a branch instruction dequeued from 102a. The test consists of performing the architected branch test and comparing the outcome with the predicted outcome indicated by the setting of the C-bit for the branch instruction. If the actual outcome agrees with the predicted outcome, the entry is removed from the branch queue 102a and the instruction is completed. If the prediction does not match the actual branch decision, the branch test unit 103a activates the REFETCH signal. When the REFETCH signal is set, the selected instruction address in the currently-unloaded branch queue register is provided through RFAMUX 226. This is termed the REFETCH address. If the REFETCH address is a sequential address, the BIA field 20 value of the register indicated by the unload pointer 232 will be output by the RFAMUX 226. In this case, the branch instruction length code (ILC) in the branch OP code is provided, together with the BIA to the incrementer 235, which adds the two values, thereby providing as the REFETCH address the address of the instruction following the tested branch instruction in the instruction stream. Activation of the REFETCH signal sets the instruction address multiplexer (IAMUX) 215 to select the output of the incrementer 235, providing this value to the instruction fetch register (reference numeral 97 in FIG. 4) as well as to the PFAR 206 and BIAR 209. Otherwise, if the branch target address is selected, the branch ILC value is set to zero and the incrementer 235 merely passes on the branch target address as the REFETCH address. When the REFETCH signal is inactive, IAMUX 215 is conditioned by the C-bit for the branch instruction in BIR 98 being decoded. If the branch is predicted not taken, the contents of the PFAR 206, incremented by the length of the previous fetch are provided to the IFAR 97, the PFAR 206, and the BIAR 209. Otherwise, the output of the adder 202 is selected by the IAMUX 215.

It should be noted that activation of the REFETCH signal indicates misprediction of a branch outcome. Therefore, this signal can be used to condition the cached version of the C-bit for the mispredicted branch instruction. The instruction's address is available in the BIA field of the currently-unloaded branch queue register.

The branch test conducted by the branch test unit 103a for BC and BCR instructions merely consists of testing the condition code (CC) against the mask field in the instruction. Thus, the OP field of the branch queue registers must be wide enough to store both the OP code and mask field for these instructions. Further, no execution cycle is required for these instructions, and they appear to execute in zero time, at least as seen by the execution pipeline. All other branch instructions require an execution cycle to modify a GPR value. The EX instruction reads a general purpose register and modifies the fetched instruction text, conceptually requiring an execution cycle. The branch processing unit 15 performs the branch test for this instruction (which is decoded as a BR instruction generating a BRL or BRR token) following the execution cycle for that instruction and from that point on, operates as described above for BC and BCR instructions. Preferably, the BC and BCR instructions are not compounded with other instructions because they require no execution cycle, and would prevent compounding which would otherwise be possible. As explained above, the instruction fetch and issue unit decodes these instructions, but skips over them as they are encountered in IB, and loads the next non-BC/BCR instruction. Other branch instructions may be compounded (subject to certain constraints), and also enter the execution pipeline. Thus, it is possible for the branch unit to be required to do two branch tests in a single cycle.

Consider the case where a BCT instruction follows an AR instruction, and is compounded with it. Assume that a BC instruction follows the BCT instruction and tests the condition code resulting from execution of the AR instruction. In this case, the AR instruction would be detected at the output of the CIRL 20 when the BCT instruction is in the CIRR 21. When entry into these registers is made, the BC would be detected and decoded at 64 and latched at 66 in the period of the pipeline clock following latching of the AR and BCT at 20 and 21, respectively. When the BCT is latched at 21, it is detected at 70 and a BR token is entered into BRR 75. On the following cycle, the BC token is latched at 73, and the BRR token is latched at 80. Both tokens are, therefore, available concurrently in the pipeline for unloading the branch queue. This results in both branch tests being performed simultaneously, with the results being provided in the correct order. That is, if the BCT is wrongly predicted, the instruction stream is refetched using the correct entry for the BCT instruction in the branch queue. If the BCT prediction was correct, while the BC prediction was wrong, then the correct address derived from the BC entry in the branch queue would be used to refetch the instruction stream. If both branches are correctly predicted, both entries are effectively removed from the branch queue in the same cycle.

BRANCH PROCESSING UNIT SECOND EMBODIMENT

Refer now to FIGS. 6-9 for an understanding of the second embodiment of the branch processing unit 15. In the second embodiment, a branch target buffer (BTB) is used to record the history of branch outcome for branch instructions recently executed.

In FIG. 6 twelve branch instructions of the System/370 type are listed. The second embodiment of the branch processing unit processes the instructions listed in FIG. 6 by detecting the branch instruction in the BIR 98. Next, the branch direction is predicted, a branch target is prefetched (if predicted taken), and instruction text and related address and prediction information is entered into the branch queue. A branch instruction is held in the queue until a test of the branch prediction and target address (if predicted taken) can be performed in proper sequence. Corrective action is then taken if the branch direction or target address was mispredicted. These corrective actions include: invalidating the current contents of the instruction buffer, branch queue, and execution pipeline, fetching the true target instruction stream, and making or modifying a BTB entry if appropriate.

The current contents of BIR 98 are decoded by branch decode logic. In the decode logic, branch instructions are categorized according to FIG. 6. In FIG. 6, the branch instructions are divided into three categories, each with a different prediction method. Instructions from the first category cause no prefetch, but are entered into the branch queue and are unloaded at branch test time for consistency with the handling of other branch instructions. The instructions from the second category are known to be taken and will cause a prefetch to be issued to the cache. For these instructions, the branch target address may be generated in parallel with a BTB access. If a BTB entry is not found for this instruction, the branch target address is generated and will be used as the fetch address for the branch target prefetch. If a valid BTB entry is found for this branch instruction, it will be used instead. In either case, both BIAR 308 and PFAR 305 values will be set to the prefetch address to indicate the new instruction stream. Instructions from the third category are either unconditionally taken branches or conditionally taken branches. In either case, only a valid BTB entry will initiate a branch target prefetch from a storage location indicated in the BTB. If no valid BTB entry exists, no prefetch or target address generation occurs at this point. For instructions from all three categories, a branch queue entry is made following the branch decode.

FIG. 7 illustrates in block diagram form a prediction mechanism which operates in response to the branch decode operation just described. A branch instruction in the BIR 98 is decoded at 300 according to the conditions listed for the three categories in FIG. 6. If the branch instruction is in the second category, its prefetch address is generated and the decoder 300 produces a prefetch register (PFR) signal signifying the category. If the branch instruction is in the third category, the decoder 300 produces a prefetch branch target buffer (PFB) signal signifying the third category. Either of these signals requires a prefetch and the OR circuit 302 combines them and activates a PREFETCH signal if either is active.

The address prediction mechanism of FIG. 7 also includes prefetch address generation circuitry 305 and 306 which operates in the same manner as the PFAR circuitry of FIG. 5 to generate an address for the next quadword in sequence in the cache 12. A branch instruction address generator including elements 308 and 309 operates in the manner described for the BIAR circuitry of FIG. 5 to generate the address of the current branch instruction in the BIR 98. A copy of the general purpose registers (GPR) 310 is addressed by relevant fields of the branch instruction in the BIR 98. As those skilled in the art will realize, if the branch instruction is of the type which generates a branch target address by register reference, this address will be output by 310 during the decode operation.

A BTB 312 contains a plurality of entries at addressable locations. Each entry consists of the most significant portion of a branch instruction address (BIAh), a validity bit, and a branch target address (BTA). An entry is made into the BTB when a REFETCH is issued as the result of a failed branch test that causes a target instruction stream to be fetched. In general, the first time a taken branch is encountered it will not cause a target prefetch, as no entry will reside in the BTB. Instead, the branch test will detect the need for target fetch, will request the fetch, and make a BTB entry. In this way, the taken branch will cause the target prefetch to be sourced from the BTB the next time it is executed. Conversely, an entry in the BTB will be invalidated as the result of a failed branch test that causes the next sequential instruction to be fetched. This would occur when a BTB entry indicating the branch was taken was proven incorrect during testing of the branch instruction. Entries in the BTB 312 are addressed by the least significant bit portion of the branch address BIA1. Thus, assume that a branch instruction of category two or three is decoded at 300. In both cases, the output of the OR circuit 302 will activate PREFETCH, enabling the BTB 312 to be read. No refetch is underway so the multiplexer 313 is conditioned to select the current branch instruction address output by the incrementer 309. BIA1 is fed through the multiplexer 313 to the address port of the BTB 312, while BIAh is fed to a comparator 314. If an entry exists in the BTB 312 at the current address, the most significant BIA portion of the entry will be fed to the comparator 314. If the most significant portions are equal, the output of the comparator 314 will be activated, indicating that an entry in the BTB 312 exists for the decoded branch instruction, and implying that the BTA field of the entry may be used as the target address.

An IAMUX (instruction address multiplexer) 317 is controlled by the PREFETCH and REFETCH signals. If the REFETCH signal is inactive, the multiplexer 317 selects its input according to the state of the PREFETCH signal. If the PREFETCH signal is active, indicating that the decoder 300 has detected a branch instruction classed in one or two of the target prefetch categories, the output of the BD AGEN 310 is selected. If the PREFETCH signal is inactive, the implication is that the decoded branch signal is in the first prediction category; therefore, the output of the PFAR INC 306 circuit is selected.

The prefetch address for the next fetch is fed to the instruction fetch address register (IFAR) 97 from a prefetch multiplexer (PFMUX) 318. The PFMUX 318 selects the instruction fetch address from either the output of the IAMUX 317 or the BTA field of the currently-addressed BTB entry. The PFMUX 318 is controlled by the REFETCH signal and the output of the comparator 314. So long as the REFETCH signal is inactive, the output of the comparator 314 will select between the output of the IAMUX 317 and the BTA field of the currently-addressed BTB entry.

Under the condition that no refetch is occurring, the next fetch address to be entered into the IFAR 97 is selected through the IAMUX 317 and PFMUX 318. If the decoder 300 detects an instruction in the second or third category of FIG. 6, the PREFETCH signal is active, selecting the output of the GPR 310. If the decoded branch instruction is in the second category of FIG. 6 and no history of previous execution of this instruction has been entered into the BTB 312, the output of the comparator 314 will be inactive resulting in the entry of the GPR contents into the IFAR 97. If the PREFETCH is active, but a current BTB entry exists for the decoded branch instruction, the contents of the BTA field of the currently-addressed BTB entry will be entered into the IFAR 97. If the current branch instruction is a category one instruction and REFETCH is inactive, the PREFETCH signal will also be inactive, and no entry for the instruction will be found in the BTB. Therefore, the sequential address will be registered at 97 via 306, 317, 318.

The PFAR 305 and BIAR 308 are both initialized from the PFMUX 318 in response to the output of an OR circuit 321. The OR circuit 321 funnels the REFETCH and PREFETCH signals to the load control inputs of the PFAR and BIAR. If either of these signals is active, register contents are forced to the value output by the PFMUX.

Loading of the branch queue (BQ) can be understood with reference to FIGS. 7 and 8. Assume that a branch instruction has just been lodged in the BIR 98 and decoded at 300 and that a prefetch instruction address has been entered into the IFAR 97. At the same time that the IFAR 97 is loaded, the BQ receives the instruction text from the BIR 98, the branch instruction address from the output of the incrementer 309, the branch target address from the output of the multiplexer 320, and the PREFETCH signal from the OR gate 302. The multiplexer 320 receives inputs either from the currently-addressed GPR in 310 or the BTA field of the currently-addressed entry in the BTB 312. Assuming a BTB entry for the decoded instruction, the output of the comparator 314 is activated, causing the multiplexer 320 to select the BTA value. Assuming no entry exists, the output of BD AGEN 310 is provided through the multiplexer 320 to the BTA field BQ. The PFETCH signal which is equivalent to prefetch, is carried in the branch queue to record whether the branch was predicted as taken or not taken. If the branch instruction is decoded to an unconditional sequential fetch (category 1 of FIG. 6), the PREFETCH signal will be inactive.

In FIG. 8, the branch queue for the second embodiment is illustrated. The branch queue consists of register arrays 330, 331 332, and 333. Each register array includes a respective portion for branch instruction text, branch instruction address, branch target address, and the PFETCH prediction signal. Register arrays are loaded under control of the branch queue load pointer (BQ LD PTR). The load pointer is conventional, incrementing to point to the next register array after loading of the current one and pausing when the next available register array is pointed to by the branch queue unload pointer. The contents of the branch queue array are received from the prediction circuitry of FIG. 7. Branch instruction text is received from the BIR 98, the branch instruction address from the incrementer 309, the branch target address from the multiplexer 320, and the PREFETCH bit from the OR gate 302.

The branch queue is unloaded through four multiplexers; each is a four-to-two multiplexer which can provide one or two outputs from four possible inputs. The first multiplexer (BQ BI MUX) 340 selects branch instruction text to forward for branch testing; the multiplexer 342 (BQ NSEQMUX) selects branch instruction address information to forward for branch testing; the multiplexer 343 (BQ BTA MUX) selects branch target address information for branch testing; and the multiplexer 344 BQ PFETCH MUX selects PREFETCH bits for branch testing. The multiplexers 340, 342, 343, and 344 are operated in lock step so that each provides the same input-to-output configuration as the others. In this manner, either the contents of one register array are provided for branch testing, or the contents of two adjacent register arrays are provided.

Loading and unloading of the four branch queue register arrays is under the control of five pointers. The loading pointer is BQ.sub.-- LD.sub.-- PTR. Two unload pointers are provided for address generation unloading (BQ.sub.-- AG.sub.-- ULPTR1&2) and two unload pointers are provided for branch test unloading (BQ.sub.-- BT.sub.-- ULPTR1&2). Address generation and branch test activity require two pointers each because there may be two branch instructions being processed in any given cycle. The load pointer is incremented as branch queue entries are enqueued. The unload pointers are incremented as branch queue entries are dequeued.

The branch queue load pointer, reference numeral 350 in FIG. 8, indicates the next branch entry to load. The load pointer is conventional in all respects and can be implemented according the the needs of a particular application. Generally, the pointer will be initialized to zero with a refetch or prefetch and will increment with the loading of each branch queue entry.

As each branch decode operation commences, it will only be allowed to end if a branch queue entry is available to be enqueued; otherwise, the operation will suspend until the queue space becomes available.

In parallel with the loading activity, the branch queue is also being unloaded for branch testing, resulting in the dequeueing of entries.

Branch queue unload pointers are generated at 352 in FIG. 8. The unload pointers are incremented in response to the BCR, BCL, BRR, and BRL tokens which advance in the execution pipeline. In both the first and second embodiments, the compounding rules prevent the parallel execution of two BR type instructions, permit parallel execution of BR with BC instructions, and BC/BC parallel execution. Thus, at any period of the pipeline clock, a single BR token may be active, a single BC token may be active, a BR and a BC token may be active, and two BC tokens may be active. The unload pointers are advanced as follows. If one token --BCR, BCL, BRR, or BRL --is active, the unload pointers are advanced by one count and the oldest entry in the branch queue is unloaded through the first port of each of the multiplexers 340, 342, 343, and 344. If two branch tokens are active in the same pipeline clock period, the branch unload pointers are advanced by two counts, the oldest branch queue entry is unloaded through the first ports of the multiplexers and the next oldest branch queue entry through the second ports of the multiplexers.

The pointer pairs are offset by one period of the pipeline clock as suggested by the unload pointer representation 352 in FIG. 9. In this regard, when one token is active during the current pipeline clock period, BQ.sub.-- AGN.sub.--

ULPTR1 is advanced to point to the next branch queue register array; in the immediately following pipeline clock period, BQ.sub.-- BT.sub.-- ULPTR1 is advanced to point to the same BQ register array.

Relatedly, when it is stated that a BQ unload pointer "points", the meaning is that it configures a multiplexer to connect the "pointed to" register portion to the multiplexer output controlled by the pointer.

Reference to FIG. 9 will provide an understanding of why a two-phase unload pointer is required in the second embodiment. In the second embodiment, the instruction text for the branch instruction next to be dequeued is gated by the respective BQ.sub.-- AGN.sub.-- ULPTR in order to permit calculation of the branch target address using the latest GPR copy. Thus, if the GPR contents have been updated since the last branch target address calculation for this instruction, the branch target address stored for this instruction in the BTB is invalid and a valid address can be provided with no further delay to the pipeline. Thus, in FIG. 9, branch target address generation circuit 354 includes a copy of the GPRs and a three-to-one adder for calculation of the branch target address according to well-known methods based upon System/370 branch instruction formats. The branch target instruction calculated at 354 is registered at 355. If two branch instructions are dequeued concurrently, a second branch target instruction calculation circuit 356 generates the branch target address for the second oldest branch queue occupant, with the generated address being registered at 357.

The structure of the multiplexer 340 and its control by the AGEN and BT unload pointers is illustrated in FIG. 9. The multiplexer 340 includes a first gate 359 controlled by the BQ.sub.-- BT.sub.-- ULPTR 1 and a second gate 360 associated with the gate 359 and controlled by the BQ.sub.-- AGEN.sub.-- ULPTR 1. Assume a single branch token is active during the current cycle of pipeline clock. That token causes the BQ.sub.-- AGEN.sub.-- ULPTR 1 to advance to a value unlocking the gate 360. This causes the instruction text to be gated from the BQ 102 through the AGEN1 port to the branch target address generation circuit 354. The address is calculated and registered at 355. Assume that the calculated branch target address is registered on the rising edge of the pipeline clock; the same edge will cause the BQ.sub.-- BT.sub.-- ULPTR 1 to increment to the same value as BQ.sub.-- AGEN.sub.-- ULTPR1, thereby activating the gate 359 and providing the instruction text for the oldest resident of the branch queue through the BTI port to the branch test circuit 103. If two branch tokens are active during the same cycle of the pipeline clock, the just-described two-step procedure is completed for the next-oldest resident of the branch queue through the AGEN2 and BT2 ports, via gates 361 and 362.

When the output of the address register 355 is first available, the BQ.sub.-- BT.sub.-- ULPTR 1 causes the multiplexers 340, 342, 343, and 344 to unload, respectively, the branch instruction text, the branch instruction address, the branch target address, and PFETCH bit of the branch queue register array which the pointer indicates. This results in the provision of the BTA stored in the branch queue to the comparator 363 which compares the value of the BTA in the branch queue with the value just calculated. The output of the comparator is activated if the values are unequal, indicating that the just-calculated BTA should be used if the branch is taken. The comparator 364 similarly compares the branch target address register at 357 with the branch target address of the next-oldest resident of the branch queue.

When the oldest branch instruction in the branch queue is dequeued, its address is fed through the multiplexer 342 to the adder 345. The adder 345 also receives the instruction length code from the instruction text of the oldest branch instruction and generates the next sequential address for use in the event that the branch is not taken. The adder 346 similarly calculates the next sequential address for the next oldest instruction in the branch queue if dequeued with the oldest resident.

BRANCH TEST

FIGS. 10 and 11 and the following discussion concern the implementation of branch testing in the context of the second preferred embodiment. However, those skilled in the art will realize that the concepts can be applied equally to the first embodiment, with only minor adjustments.

In FIG. 10, the branch test unit 103 includes a first branch test mechanism (BT1) 103a for testing the branch conditions of the oldest resident of the branch. If two register arrays are unloaded from the branch simultaneously, a second branch test mechanism (BT2) 103b tests the branch conditions for the second-oldest of the dequeued instructions. In each case, branch testing requires provision of the condition code (CC) and branch test results from the execution unit, as well as branch instruction text, the PFETCH bit and the result of the branch target address comparison. Each of the branch test mechanisms provides two signals. The first signal (BT FAIL) indicates whether the predicted branch outcome matches the actual outcome, and the second (TAKEN) indicates whether, based upon execution unit condition, the branch is taken or not. Thus, for example, if the BT.sub.-- FAIL.sub.-- 1 signal is active, the predicted outcome for the branch was incorrect and the signal TAKEN.sub.-- 1 indicates whether the actual outcome of the branch test is the branch that is taken or not taken. Preferably, if the branch is taken, the TAKEN signal is active, otherwise, if the branch is not taken, the signal is inactive. The branch test mechanism 103b for the next-oldest of two dequeued branch instructions provides signals BT.sub.-- FAIL.sub.-- 2 and TAKEN.sub.-- 2 which are conditioned in the same manner as the signals output by the branch test mechanism 103a.

A "first-to-fail" circuit 368 receives the outputs of the branch test unit 103 and inspects the BT FAIL signals. If either signal is active, the circuit 368 activates the REFETCH signal. In addition, the circuit 368 decides which of two possible branch instructions was the first to be predicted incorrectly. In this regard, if only a single branch instruction is being dequeued and the circuit 368 detects a BT FAIL, that instruction is considered to be the "first to fail". If two branch instructions have been dequeued and BT.sub.-- FAIL.sub.-- 1 is active, the oldest branch instruction of the dequeued pair is considered to have failed; otherwise, if BT.sub.-- FAIL.sub.-- 1 is inactive but BT.sub.-- FAIL.sub.-- 2 is active, the second instruction is considered the first to fail.

The first to fail circuit 368 conditions the BT.sub.-- IA.sub.-- SEL signal according to which currently-tested branch instruction has been deemed the first to fail. If BT.sub.-- FAIL.sub.-- 1 is active, the BT.sub.-- IA.sub.-- SEL signal is conditioned to select the contents of the branch target address register 355, if TAKEN.sub.-- 1 is active, or the output of the adder 345, if TAKEN.sub.-- 1 is inactive. If two branch instructions are being tested and only BT.sub.-- FAIL.sub.-- 2 is active, the BT.sub.-- IA.sub.-- SEL signal complex is conditioned to select the contents of register 357, if TAKEN.sub.-- 2 is active, or the output of the register 346, if TAKEN.sub.-- 2 is inactive. In this manner, the branch test instruction address multiplexer (BT.sub.-- IA.sub.-- MUX) 362 is conditioned by the BT.sub.-- IA.sub.-- SEL signal complex to select for the first branch instruction to fail either the branch target address if the branch was predicted not taken or the sequential address if the branch was predicted as taken.

Consider now the effect of activating the REFETCH signal on the prediction mechanism of FIG. 7. When the REFETCH signal is activated, the IAMUX 317 is conditioned to select the output of the multiplexer 362. The active REFETCH signal causes the PFMUX 318 to select the output of the IAMUX 317, resulting in entry of a refetch address selected by the BT IA multiplexer 362 as described above. This address is registered at 97. In addition, the OR gate 321 responds to the refetch activation by forcing the contents of the PFAR 305 and BIAR 308 to the refetch address value at the output of the PFMUX 318.

FIG. 11 is a digital logic schematic illustrating the structure of a branch test mechanism which generates the BT FAIL and taken signals required of the mechanisms indicated by reference numerals 103a and 103b in FIG. 10. The branch test mechanism includes an instruction decoder 400 which resolves the OP code of the branch instruction text into one of five classes of branch instructions. The operation of the decoder assumes that the branch instructions are drawn from the IBM System/370 instruction set. In this case, the BC, BCT, BXH, and BXLE outputs of the decoder are mnemonics for instructions in that set. The decoder assumes that all other branch instructions are unconditionally taken. If a BC instruction is decoded, the mask portion of the instruction text is provided to an AND/OR (AO) circuit 402 which compares the mask with the condition code (CC). Assuming that the mask and condition code match, the output of the circuit 402 is activated and, if the instruction is BC, the output of the AND gate 403 is activated, indicating that the branch is to be taken. If the instruction is decoded as BCT, the inverse of the R=0 branch condition generated by the execution unit is enabled by the AND gate 404. The AND gate 405 enables BXH instruction to test the R>0 branch condition from the execution unit, while the AND gate 406 tests the BXLE instruction conditions against the inverse of the R>0 branch condition. All unconditionally taken branches result in activation of the UNCOND TAKEN signal by the decoder 400. If the output of any of the AND gates 403-406 is active, or if an unconditionally taken branch is detected, the output of the OR gate 409 activates. Activation of the output of this gate indicates that the indicated branch must be taken for the branch instruction being tested. This output is provided as the TAKEN signal of the branch test mechanism. The exclusive OR gate 410 compares the output of the OR gate with the PFETCH signal for the tested branch instruction. If the TAKEN and PFETCH signals are non-equivalent, the output of the XOR gate 410 activates indicating that the incorrect outcome was predicted for the branch and that a refetch is required. The AND gate 412 tests whether, in the event a branch target address was prefetched (indicated by activation of the PFETCH signal) a subsequent change in conditions since the prefetch has changed the branch target address. In this case, the output of the relevant comparator will indicate non-equivalence of the branch target addresses in the output of the AND gate 412 will be activated. The OR gate funnels the outputs of the exclusive OR and AND gates 410 and 412, producing a BT.sub.-- FAIL signal if either is active. This output is provided on signal line 415 as the BT.sub.-- FAIL for the mechanism, while the TAKEN signal on signal line 416 is provided as the TAKEN signal for the mechanism.

BTB UPDATING

If a refetch is required to obtain a branch target, an entry must be made to the BTB 312 for the branch instruction causing the refetch. In this regard, an entry for the instruction must be made including branch target address being fetched and the most significant portion of the branch instructions address. The entry must be stored at an address equivalent to the least significant portion of the branch instruction address. When a refetch is underway, the address of the branch instruction causing the operation is available through either the first or the second output of the multiplexer 342 in FIG. 8. The instruction text for the first-to-fail branch instruction is selected by a multiplexer 427 in FIG. 7, which is enabled by the refetch signal and which selects in response to a select signal output by a control circuit 427. The circuit 427 combines the BT.sub.-- FAIL.sub.-- 1 & 2 and TAKEN 1&2 signals to select from the multiplexer 342 the branch instruction which is judged first to fail as described above. In this regard, if the BT.sub.-- FAIL.sub.-- 1 and TAKEN.sub.-- 1 signals are concurrent, the oldest of the two dequeued branch instructions has caused the refetch and its address is selected from the first output of multiplexer 342. If BT.sub.-- FAIL.sub.-- 1 is off, and if BT.sub.-- FAIL.sub.-- 2 and TAKEN.sub.-- 2 are concurrent, the instruction address from the second output of multiplexer 342 is selected. The instruction address selected by the multiplexer 425 is provided to the BTB 312, with the most significant portion of the address being provided to the port for the BIAh value and the least significant portion being provided through the multiplexer 313 to the address port of the BTB 312. The multiplexer 313 will select this output for so long as the REFETCH signal is active. The BTA field of the addressed entry is provided from the contents of the BIAR and placed into the BTA field of the currently-addressed BT location during a branch target refetch. The BTB 312 is enabled to write by activation of the REFETCH signal for take branches.

POINTER OPERATION

FIG. 13 is a timing diagram illustrating, during 11 consecutive periods of the pipeline clock, how the invention tests branch conditions for one, or two, queued branch instructions. In preparation for explaining FIG. 13, reference is first given to FIGS. 11 and 12. FIG. 11 illustrates a branch test mechanism which can be used in both the first and second portions of the branch test unit. However, in order to operate the units correctly, indication must be provided as to the number of branch tests being conducted. Assuming that the explanation given hereinabove with reference to FIG. 11 fully lays out the common elements, reference is given to a decode circuit 419 in FIG. 11 which is present only in the second branch test mechanism. The input to the decode circuit 419 is a signal LAMT. This is a two-bit signal which is set either to a value `01` indicating that one branch instruction is being dequeued, or to `10`, indicating that two branch instructions are being dequeued. If one branch instruction is being dequeued, operation of the second portion of the branch test unit is suppressed. On the other hand, if two branch instructions are dequeued, the second portion of the branch test unit must be enabled. The decode circuit 419 enables the second portion by activating its output when LAMT=10. This enables the gates 403-406. An additional AND gate 420 is provided in the second portion to gate the UNCOND TAKEN output of the instruction in response to the output of the decode unit 419.

FIG. 12 is a block diagram illustrating in greater detail how branch instructions are entered into and extracted from the branch queue. The decoding of a branch instruction in BIR 98 is indicated by an output of the decoder 300. The AND gate 499 combines the decoder output with the inverse of the REFETCH signal. If no refetch procedure is underway and a branch instruction is detected in the BIR 98, the output of the AND gate activates a signal (BR.sub.-- DEC.sub.-- VALID) to indicate that the branch decode is valid. This signal is provided to the branch queue load pointer circuit 350. This signal is also provided, together with the instruction text in the BIR 98, to a select (SEL) circuit 500. The branch queue load pointer circuit 350 is configured as a two-bit register which holds a binary value (BQ.sub.-- LPTR) indicating the next BQ entry to load. Initialized to zero, the value of this pointer will increment with the loading of each BQ entry. The pointer value will wrap through zero and will only be limited by the availability of a free BQ register array. The availability of a BQ register array is detected by the absence of a corresponding BQ validity bit. As FIG. 12 illustrates, each register array has, in addition to fields illustrated in FIG. 8, a validity bit field V. As each entry is enqueued, the bit in this field is set; as the entry is dequeued, this bit is reset. Thus, the branch queue load pointer circuit 350 must also receive the condition of the validity bit fields 330a, 331a, 332a, and 333a from the register arrays. When a branch instruction is decoded, the load pointer is incremented from the last-filled register array to the next. The register array which is to receive the contents of the BIR is indicated by the branch queue load pointer (BQ.sub.-- LPTR), a two-bit signal encoded to indicate which of the register arrays is to receive the instruction text. This selection is done conventionally in the select circuit 500. The V field of the selected register array is conditioned by the output of the AND gate 499 via the select circuit 500. Thus, as instruction text is entered into the next branch queue register array, the validity bit is set.

The validity bit fields 330a, 331a, 332a, and 333a are controlled by respective select circuits 530, 531, 532, and 533. Each of these circuits is a two-to-one multiplexer whose state is conditioned by a control circuit 536. The default condition for each of these circuits is to the corresponding V-bit output from the SELECT circuit 500. However, if the REFETCH signal is activated, or if the contents of the register array are being unloaded for branch testing as indicated by one of the two branch queue unload pointers BQ.sub.-- BT.sub.-- ULPT.sub.-- 1 or 2, the selector input is connected to a hard wired "zero" for one cycle of the pipeline clock. This invalidates the branch queue entries whenever a refetch operation begins. This also synchronizes the dequeueing the branch queue entries for testing with the enqueueing of decoded branch instructions in the BIR 98.

The branch queue load pointer circuit 350 is initialized to zero whenever a refetch or prefetch operation occurs. The logic for incrementing the BQ.sub.-- LPTR is given in Table I in conventional Boolean form. As an instruction is decoded from the BIR 98, an enqueueing is performed, making a complete entry as described above in the description of the branch queue. The branch queue load circuit 350 will load available BQ entries in sequence until the queue is full. It should be noted that in parallel with this activity, the queue is also being unloaded for branch testing, resulting in the dequeueing of entries. Thereby, positions in the queue are released (as indicated by the reset V-bits) to be loaded again. Four branch queue unload pointer signals are generated by the branch queue unload pointer 352. Each pointer is a two-bit signal whose setting "points" to a correspondingly-numbered branch queue register array. Two of the pointers, BQ.sub.-- AG.sub.-- ULPTR 1&2, are for unloading instruction text from the BQ through the AGEN 1 and AGEN 2 ports of the BI multiplexer 340 in FIG. 9. Two other unload pointers, BQ.sub.-- BT.sub.-- ULPTR 1&2 are for unloading instruction text through the BT1 and BT2 ports of the BI multiplexer 340. A signal output by the unload pointer 352, LAMT, indicate the number of branch queue entries being dequeued for address generation and branch testing, respectively. This signal is a two-bit value set to either `01` to indicate that one instruction is being dequeued or to `10` to indicate the dequeueing of two instructions. The signal LAMT represents the latching of a signal AMT 20 generated internally in the unload pointer circuit 352. That signal is latched to delay it by one period of the pipeline clock.

The unload pointer circuit 352 increments the unload pointer signals in response to the branch tokens in the execution pipeline. The BC tokens are provided for pointer incrementation from the address generation stage of the pipeline and are denoted as BCL.sub.-- AG and BCR.sub.-- AG. All other branch tokens are provided in the execution stage of the pipeline and are denoted as BRL.sub.-- EX and BRR.sub.-- EX. The unload pointer circuit 352 also receives the REFETCH signals and the PIPELINE.sub.-- CLOCK signal.

The BQ.sub.-- BT.sub.-- ULPTR 1 pointers indicate the oldest resident of the branch queue; the BQ.sub.-- BT.sub.-- ULPTR 2 signals always indicate the next-oldest branch queue resident. Each BQ.sub.-- AG.sub.-- ULPTR is updated one pipeline clock period in advance L of the corresponding BQ.sub.-- BT.sub.-- ULPTR in order to access instruction text so that the most current branch target address can be calculated.

The BQ unload pointers BQ.sub.-- AG.sub.-- ULPTR 1 and BQ.sub.-- BT.sub.-- ULPTR.sub.-- 1 are configured as two-bit registers. The pointers BQ.sub.-- AG.sub.-- ULPTR.sub.-- 2 and BQ.sub.-- BT.sub.-- ULPTR.sub.-- 2 are decoded from the corresponding register pointers. All pointers hold binary values indicating which branch queue entries are to be gated to the appropriate AGEN and BT outputs of the BI multiplexer 340. The AG unload pointers constantly provide branch instruction text to the branch target address generation functions. The results of the two branch target address calculations are latched into the corresponding AGEN AR registers 355 and 357 at the end of a pipeline clock. In the following pipeline clock period, the BQ.sub.-- AG.sub.-- ULPTR values are incremented by an amount equal to the number of branch instructions dequeued.

The two BT unload pointers are one entry behind the AG unload pointers. The BT pointers use the results generated by updating the AGEN pointers to perform branch direction testing. Like the AGEN pointers, the BT pointers are advanced with each completed branch test under control of the branch tokens.

Internal logic for a branch queue unload pointer finite state machine is given in Tables II and III in well-known Boolean form. The Boolean value AG.sub.-- INCR.sub.-- AMT (0:1), when latched by PIPELINE.sub.-- CLOCK becomes LAMT.

As FIG. 12 shows, the AGEN unload pointers and the AMT signal are fed to the BI multiplexer to control the AGEN outputs. Thus, during the current pipeline clock period, the register array pointed to by the AGEN unload pointer 1 is provided through the AGEN 1 output of the multiplexer 340. If AMT=10, instruction text in the register pointed to by the AGEN unload pointer 2 is fed through the AGEN 2 output of the multiplexer 340. The BT outputs of the multiplexer 340 are similarly controlled by the BT unload pointers 1 and 2 and the value of LAMT signal.

The control logic 536 responds to the updating of the BT unload pointers and to the LAMT signal by resetting the V bit and the register array pointed to by the BT unload pointer 1; if LAMT=10, the V bit in the register array pointed to by the BT unload pointer 2 is also reset.

OPERATION OF THE BRANCH PROCESSING UNIT

FIG. 13 is a timing diagram illustrating, for 11 consecutive periods of the pipeline clock, how the branch processing unit 14 operates. In clock period 1, assume a fetch occurs bringing three consecutive branch instructions to the instruction fetch and issue unit. The first branch instruction Br a is fed immediately to the branch processing unit and latched into the BIR 98. Simultaneously, the CIR multiplexer feeds the instruction to the CIRL 20. In the following period, cycle period 2, the branch instruction is decoded by the decoding circuit 70 in the execution pipeline and by the decoding circuit 300 in the branch processing unit. Branch decoding is indicated by "BD". Assuming that the branch instruction generates a single-cycle microinstruction sequence, the END.sub.-- OP bit will be available at the output of the MCS during this cycle, resulting in the entry of a BRL token into the AGEN latch

74 in cycle period 3. (This is indicated by BR.sub.-- AG.) Following the first branch instruction, the second branch instruction, Br b, is registered into the CIRL and BIR in cycle period 2, and is decoded at 70 and 300 in cycle period 3. In cycle period 3, the third branch instruction BC instr is registered at the CIRL 20 and BIR 98 and is decoded at 70 and 300 in cycle period 4.

In cycle period 4, the BRL token for the branch instruction Br a has advanced to the execution stage of the pipeline as indicated by a in the BR.sub.-- EX line of FIG. 13. This activates the DEQUE.sub.-- AG.sub.-- ULPTR signal. In addition, the AG.sub.-- INCR.sub.-- AMT signal is set to `01` since there is only one BR token in the execution stage of the execution pipeline and there are no BC tokens in the address generation stage. In cycle period 5, the AGEN unload pointers are both incremented by one, while the DEQUE.sub.-- BT.sub.-- ULPT signal is activated. Since LAMT is set to `01`, only the contents of the first branch queue register array are provided to the branch test circuit through BTl of the multiplexer 340.

In period 5 of the pipeline clock, the BR token for branch instruction b has reached the execution stage of the execution pipeline while the BC token for the BC instruction has reached the address generation stage of the pipeline. At this point, the DEQUE.sub.-- AG.sub.-- ULPTR signal remains active, while the AG.sub.-- INCR.sub.-- AMT signal is set to `10`. This results in the dequeueing of branch instruction b and the BC instruction from branch queue through the AGEN 1 and AGEN 2 outputs of the BI multiplexer 340 during clock period 4. In clock period 5, the AGEN unload pointers both incremented by the digital value (10) while the DEQUE.sub.-- BT.sub.-- ULPTR signal remains set. As a result, the branch instruction information in queue positions `00` and `01` are dequeued for branch testing in cycle period 6. In cycle period 7, the BT unload pointers are each updated by the digital value `10`.

It should be noted that the END.sub.-- OP activation in FIG. 13 determines the length in pipeline clock periods which a branch instruction stays in the queue. If a branch instruction is decoded with a two-microinstruction sequence, the END.sub.-- OP bit will occur one period after the branch instruction is decoded; consequently, the token for the instruction will be delayed for one clock period, thereby stretching the time on the queue two clock periods.

It is observed that FIG. 13 further shows the fetching of two branch instructions, Br d and Br e, as a result of decoding the BC instruction. The fetch occurs in clock period 5, branch instruction d is decoded in clock period 6, waits on the queue during clock period 7, is dequeued for address generation in clock period 8, and dequeued for branch testing in clock period 9. Branch instruction e exhibits this sequence, delayed by one clock period from branch instruction d.

While we have described our inventions in their preferred embodiment and alternative embodiments, various modifications may be made, both now and in the future, as those skilled in the art will appreciate upon the understanding of our disclosed inventions. Such modifications and future improvements are to be understood to be intended to be within the scope of the appended claims which are to be construed to protect the rights of the inventors who first conceived of these inventions.

TABLE I

```
/*Detect current Branch Test*/
/*Latch up previous cycle agen*/
DEQUE.sub.-- BT.sub.-- ULPTR.sub.-- L1=DEQUE.sub.-- AG.sub.-- ULPTR;
/*data port of BT ULPTR register, master portion.*/
/*This is effectively equal to AG point from previous
cycle.*/
BQ.sub.-- BT.sub.-- ULPTR.sub.-- O.sub.-- L1(0:1)=
                BQ.sub.-- AG.sub.-- ULPTR.sub.-- O.sub.-- L2(0:1);
/*BT ULPTR.sub.-- 1 always equals incremented BT.sub.-- ULPTR.sub.--
O.*/
BT.sub.-- ULPTR .sub.-- INC(0:1)=
                '00' when (BQ.sub.-- BT.sub.-- ULPTR.sub.-- L2(0:1)='11')
                or
                '01' when (BQ.sub.-- BT.sub.-- ULPTR.sub.-- L2(0:1)='00')
                or
                '10' when (BQ.sub.-- BT.sub.-- ULPTR.sub.-- L2(0:1)='01')
                or
                '11' when (BQ.sub.-- BT.sub.-- ULPTR.sub.-- L2(0:1)='10');
                .
BQ.sub.-- BT.sub.-- ULPTR.sub.-- 1(0:1)=
                BT.sub.-- ULPTR.sub.-- INC(0:1);
```

TABLE II

```
/*Detect current AGen
DEQUE.sub.-- AG.sub.-- ULPTR = BR.sub.-- EX.sub.-- L2(0) or BR.sub.--
EX.sub.-- L2(1) or BC.sub.-- AG.sub.-- L2(0)
  or BC.sub.-- AG.sub.-- L2(1);
/*new values for pointer.*/
AG.sub.-- ULPTR.sub.-- INC.sub.-- 1(0:1)=
               '00' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='11') or
               '01' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='00') or
               '10' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='01') or
               '11' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='10');
AG.sub.-- ULPTR.sub.-- INC.sub.-- 2(0:1)=
               '00' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='10') or
               '01' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='11') or
               '10' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='00') or
               '11' when
                     (BQ.sub.-- AG.sub.-- ULPTR.sub.-- L2(0:1)='01');
/*compute number of AGens currently being done.*/
/*This defines increment amount.*/
AG.sub.-- AMT(0:3) = BR.sub.-- EX.sub.-- L2(0:1) .vertline..vertline.
BC.sub.-- AG.sub.-- L2(0:1);
AG.sub.-- INCR.sub.-- AMT(0:1)=
               '01' when (
                     (AG.sub.-- AMT(0:3)='1000') or
                     (AG.sub.-- AMT(0:3)='0100') or
                     (AG.sub.-- AMT(0:3)='0010') or
                     (AG.sub.-- AMT(0:3)='0001'))
               '10' when (
                     (AG.sub.-- AMT(0:3)='0011') or
                     (AG.sub.-- AMT(0:3)='0101') or
                     (AG.sub.-- AMT(0:3)='0110') or
                     (AG.sub.-- AMT(0:3)='1001') or
                     (AG.sub.-- AMT(0:3)='1010') or
                     (AG.sub.-- AMT(0:3)='1100'));
AG.sub.-- ULPTR.sub.-- INC(0:1)=
               (AG.sub.-- ULPTR.sub.-- INC.sub.-- 1(0:1) and
                  (AG.sub.-- INCR.sub.-- AMT(0:1)='01')) or
               (AG.sub.-- ULPTR.sub.-- INC.sub.-- 2(0:1) and
                 (AG.sub.-- INCR.sub.-- AMT(0:1)='10'));
/*data port of AG ULPTR register, master portion.*/
BQ.sub.-- AG.sub.-- ULPTR.sub.-- O.sub.-- L1(0:1)=
               (AG.sub.-- ULPTR.sub.-- INC(0:1) and
                 DEQUE.sub.-- AG.sub.-- ULPTR) or
               (BQ.sub.-- AG.sub.-- ULPTR.sub.-- O.sub.-- L2(0:1) and
               not
                 (DEQUE.sub.-- AG.sub.-- ULPTR));
/*AG ULPTR.sub.-- 1 always equals incremented AG.sub.-- ULPTR.sub.--
O.*/
BQ.sub.-- AG.sub.-- ULPTR.sub.-- 1(0:1)=
               AG.sub.-- ULPTR.sub.-- INC.sub.-- 1(0:1);
```

TABLE III

```
/*increment for new load pointer value.*/
LPTR.sub.-- INC(0:1)=
             '00' when (BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='11') or
             '01' when (BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='00') or
             '10' when (BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='01') or
             '11' when (BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='10');
/*detection of valid selected entry.*/
SEL.sub.-- BQ.sub.-- VAL=
             ((BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='00') and BQ.sub.--
             VAL.sub.-- L2(0))
or
             ((BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='01') and BQ.sub.--
             VAL.sub.-- L2(1))
or
             ((BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='10') and BQ.sub.--
             VAL.sub.-- L2(2))
or
             ((BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1)='11') and BQ.sub.--
             VAL.sub.-- L2(3));
/*gating condition to allow advancement of pointer.*/
```

```
/*also controls latching of new data into corresponding BQ
register.*/
ENQUE.sub.-- BQ=BR.sub.-- DEC.sub.-- VALID and not (SEL.sub.-- BQ.sub.--
VAL);
/*data input to master portion of load pointer register.*/
BQ.sub.-- LD.sub.-- PTR.sub.-- L1(0:1)=
             (LPTR.sub.-- INC(0:1) and ENQUE.sub.-- BQ) or
             (BQ.sub.-- LD.sub.-- PTR.sub.-- L2(0:1) and not ENQUE.sub.--
             BQ));
```

---

* * * * *

United States Patent	***5,214,763***
Blaner , et al.	**May 25, 1993**

Digital computer system capable of processing two or more instructions in parallel and having a coche and instruction compounding mechanism

Abstract

A digital computer system capable of processing two or more computer instructions in parallel and having a cache storage unit for temporarily storing machine-level computer instructions in their journey from a higher-level storage unit of the computer system to the functional units which process the instructions. The computer system includes an instruction compounding unit located intermediate to the higher-level storage unit and the cache storage unit for analyzing the instructions and adding to each instruction a tag field which indicates whether or not that instruction may be processed in parallel with one or more neighboring instructions in the instruction stream. These tagged instructions are then stored in the cache unit. The computer system further includes a plurality of functional instruction processing units which operate in parallel with one another. The instructions supplied to these functional units are obtained from the cache storage unit. At instruction issue time, the tag fields of the instructions are examined and those tagged for parallel processing are sent to different ones of the functional units in accordance with the codings of their operation code fields.

Inventors:	**Blaner; Bartholomew** (Newark Valley, NY); **Vassiliadis; Stamatis** (Vestal, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**522291**
Filed:	**May 10, 1990**

Current U.S. Class:	**712/212**; 712/23; 712/24; 712/213; 712/215
Intern'l Class:	G06F 015/16
Field of Search:	395/375,800

References Cited [Referenced By]

U.S. Patent Documents

3293616	Dec., 1966	Mullery et al.	364/200.
3470540	Sep., 1969	Levy	364/200.
3611306	Feb., 1969	Reigel	364/200.
3781814	Dec., 1973	Deerfield	364/200.
4197589	Apr., 1980	Cornish et al.	364/900.
4295193	Oct., 1981	Pomerene	364/200.
4439828	Mar., 1984	Martin	364/200.
4594655	Jun., 1986	Hao et al.	364/200.
4626989	Dec., 1986	Torti	364/200.
4722050	Jan., 1988	Lee et al.	364/200.
4766566	Aug., 1988	Chuang	364/900.
4829422	May., 1989	Morton et al.	364/200.
4847755	Jul., 1989	Morrison et al.	364/200.
4942525	Jul., 1990	Shintani et al.	364/200.
5050068	Sep., 1991	Dollas et al.	364/200.

Other References

Acosta, R.D., et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35 No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al., the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", Computer Structures: Principles and Examples (Siewiorek, et al., ed) McGraw-Hill, 1982, pp. 276-292.
Capozzi, A. J., et al., "Non-Sequential High-Performance Processing" IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al., "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K., et al., "SIMP (Single Instruction Stream/Multiple Instruction Pipelining); A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J.E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M.D., et al., "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units",

Computer Structures, Principles, and Examples (Siewiorek, et al. ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S. "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, mar. 1988, pp. 70-84.
Jouppi, N. P., et al., "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N.P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D.E., "Intel's 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun., 1988, pp. 63-76.
Colwell, R.P., et al., "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.
Berenbaum, A.D., "Introduction to the CRISP Instruction Set Architecture", Proceedings of COMPCON, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al., "Compiling for the CRISP Microprocessor", Proceedings of COMPCON, Spring, 1987, pp. 96-100.
Hennessy, J., et al., "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E.A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D.R., et al., "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of COMPCON, Spring 1987, pp. 2-9.
Hwu, W.W., et al., "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J.K.F., et al., "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1, Jan. 1984, pp. 6-22.
Riseman, E.M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J.E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al., Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J.L., et al. "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A.J., "Cache Memories", Computing Surveys, vol. 14, No. 3 Sep., 1982, pp. 473-530.
Smith, J.E., et al., "A Study of Instruction Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun., 1983, pp. 132-137.
Vassiliadis, S., et al., "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.

Tucker, S.G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication No. SA22-7200-0, Principles of Operation, and IBM Pub. #SA22-7125-3
IBM Enterprise Systems Architecture/370, 1988 and Tech Newsletter SN22-5342.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981 pp. 220-278.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard Cache Mgmt., pp. 371-374.
Motorola's MC88100 User's Manual, pp. 1-1-1-13, publication prior to Apr. 27, 1990.

Primary Examiner: Lall; Parshotam S.
Assistant Examiner: Kim; Ken S.
Attorney, Agent or Firm: Augspurger; Lynn L.

Claims

What is claimed is:

1. In a digital computer system capable of processing two or more instructions in parallel, the combination comprising:

a larger-capacity, lower-speed storage mechanism for storing instructions to be processed;

a smaller-capacity, higher-speed cache storage mechanism for storing instructions with associated tag information;

an instruction fetch and issue unit;

and an instruction compounding mechanism coupled to receive an input of instructions from the lower-speed storage mechanism and to provide an output of instructions with associated tag information to the smaller-capacity storage mechanism and located between the lower-speed storage mechanism and the higher-speed storage mechanism, for analyzing these received instructions and producing the associated tag information for supplying these instructions and associated tag information to the higher-speed cache storage mechanism for storage therein prior to instruction fetch and issue by said instruction fetch and issue unit, the associated tag information comprising one or more bits indicating whether the instruction with which the tag is associated may be executed in parallel and which indicates which instructions may be processed in parallel with one another and having of a plurality of tag fields, a different one of which is associated with each instruction analyzed by the instruction compounding mechanism by an examination of the opcode; and wherein the instruction compounding mechanism includes a plural-instruction instruction register for receiving a plurality of successive instructions from the lower-speed storage mechanism and means for producing a compoundability signal including a plurality of rule-based instruction analyzer mechanism, each of which analyzes the opcode of a different pair of side-by-side instructions in the instruction register, said means for producing a compoundability signal indicating whether or not the two instructions in its pair may be processed in parallel, and each instruction analyzer mechanism including logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution in the particular instruction processing configuration used for the computer system, such logic circuitry providing said means for producing the compoundability signal for the analyzer mechanism,

and a tag generating mechanism responsive to the compoundability signals for generating the individual tag fields for the analyzed instructions in the instruction register for providing said tag fields as associated tags for the instructions analyzed and providing them as output to said higher-speed cache storage mechanism for storage and for access of the instructions by said instruction fetch and issue unit;

a plurality of functional instruction processing units of different functional types which operate in parallel with one another and wherein the tag information generated by the instruction compounding mechanism is used by said instruction fetch and issue unit in causing the issuance of two or more instructions in parallel from the higher-speed storage mechanism to appropriate functional units which can perform the required function for the issuing instructions.

2. The combination according to claim 1 wherein instruction compounding analysis is performed when an instruction cache miss occurs.

3. The combination according to claim 1 wherein as needed, the tagged instructions in the cache storage mechanism are fetched by the instruction fetch and issue unit, and as the tagged instructions are received by fetch and issue unit, their tag fields are examined to determine if they may be processed in parallel and their opcode fields are examined to determine which of the available functional units is most appropriate for their processing, and if the tag fields indicate that two or more of the instructions are suitable for processing in parallel, then they are sent to the appropriate ones of the functional units in accordance with the codings of their opcode fields and such instructions are then processed concurrently with one another by their respective functional units; but when an instruction is encountered that is not suitable for parallel processing, then it is sent to the appropriate functional unit as determined by its opcode and it is thereupon processed alone and by itself by the selected functional unit.

4. The combination according to claim 1 wherein the instruction compounding unit for a case where a maximum of two instructions at a time may be processed in parallel in a particular computer configuration has, a one-bit tag field, and a tag bit value of "one" means that the instruction is a "first" instruction, a tag bit value of "zero" means that the instruction is "second" instruction, and may be executed in parallel with the proceeding first instruction, and an instruction having a tag bit value of one may be executed either by itself or at the same time and in parallel with the next instruction, depending on the tag bit value for such next instruction, each pairing of an instruction having a tag bit value of one with a succeeding instruction having a tag bit value of zero forming a compound instruction for parallel execution purposes, such that the instructions in such a pair may be processed in parallel with one another, and when the tag bits for two succeeding instructions each have a value of one, the first of these instructions is executed by itself in a nonparallel manner, and in a worst case, all of the instructions in the sequence would have a tag bit value of one and all of the instructions would be executed one at a time in a nonparallel manner.

5. The combination according to claim 1 wherein each instruction analyzer mechanism produces two of

the compoundability signals, one of which indicates whether or not Instructions 0 and 1 may be processed in parallel, and another second compoundability signal indicates whether or not Instructions 1 and 2 may be processed in parallel;

and, in a similar way, a second compound analyzer mechanism produces a first compoundability signal which indicates whether or not Instructions 2 and 3 may be processed in parallel and a second compoundability signal which indicates whether Instructions 3 and 4 may be processed in parallel, and a third compound analyzer produces a first compoundability signal which indicates whether or not Instructions 4 and 5 may be processed in parallel and a second compoundability signal which indicates whether or not Instructions 5 and 6 may be processed in parallel, and a fourth compound analyzer mechanism produces a first compoundability signal which indicates whether or not Instructions 6 and 7 maybe processed in parallel and a second compoundability signal which indicates whether Instructions 7 and 8 may be processed in parallel.

6. The combination according to claim 1 wherein the tag generating mechanism is responsive to the compoundability signals appearing at the outputs of the analyzer units for generating the individual tag fields for the different instructions in the instruction register, and these tag fields are supplied to a tagged instruction register, as are the instructions themselves, the latter being obtained from an input instruction register, whereby the compounding unit output register, a tag field for each instruction, wherein each tag field is comprised of a single binary bit, and a tag bit value of "one" indicates that the immediately following instruction to which it is attached is a "first" instruction, and a tag bit value of "zero" indicates that the immediately following instruction is a "second" instruction, and an instruction having a tag bit value of one followed by an instruction having a tag bit value of zero indicates that those two instructions may be executed in parallel with one another, the tagged instructions in the compounding unit output register being supplied to the input of the compound instruction cache storage mechanism and stored into such cache storage mechanism.

7. The combination according to claim 1 wherein each compound analyzer mechanism includes instruction compatibility logic for examining the opcode of instruction pairs and determining whether these two opcodes are compatible for purposes of execution in parallel, and wherein the compabatiabilty logic includes logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution in the particular hardware configuration used for the computer system being considered, and if the opcodes for the pair are compatible, then logic produces at its output a binary one level signal, and if they are not compatible, logic 30 produces a binary zero value on its output line, and wherein the compound analyzer further includes a second instruction compatibility logic for examining the opcodes of another pair of instructions and determining whether they are compatible for parallel execution by examination of their opcodes.

8. The combination according to claim 1 wherein the compound analyzer mechanism includes first register dependency logic for detecting conflicts in the usage of the general purpose registers designated by R1 and R2 fields of the instruction pair which are general purpose registered, such that the dependency logic may to detect the occurrence of a data depending condition wherein a second instruction (Instruction 2) needs to use the results obtained by the performance of the proceeding instruction (Instruction 0), in which case, either the second instruction can be executed by the dependency collapsing hardware, thus executing in parallel with the first instruction, or in the execution of the second instruction must await completion of the execution of the preceding instruction, and hence,

cannot be executed in parallel with the preceding instruction, and wherein if there are no register dependencies which prevent execution of Instructions 0 and 1 in parallel, then the output line of dependency logic is given a binary value of one, and if there is a dependency, then it is given a binary value of zero.

9. The combination according to claim 8 wherein output lines from the instruction compatibility logic and the register dependency logic are connected to the two inputs of an AND circuit, wherein the output line of the AND circuit has a binary one value if the two opcodes being considered are compatible and if there are no register dependencies; however, this binary one value on the AND circuit output line indicates that the two instructions being considered are compoundable, that is, are executable in parallel, but if, on the other hand, the AND circuit output line has a binary value of zero, then the two instructions are not compoundable and thus, there is produced on the AND circuit output line a first compoundability signal which indicates whether or not Instructions 0 and 1 may be processed in parallel, which signal is supplied to the tag generator.

10. The combination according to claim 1 wherein the logic compoundability internal circuitry for the instruction includes a first compatibility logic circuit which includes a plurality of decoders and AND circuits and an OR circuit, and a second instruction compoundability logic circuit which includes a plurality of decoders and AND circuits and an OR circuit with a middle decoder shared by both first and second logic circuitry.

Description

CROSS REFERENCE TO RELATED APPLICATIONS

The present United States patent application is related to the following copending United States patent applications:

(1) Application Ser. No. 07/519,384 filed May 4 1990, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al;

(2) Application Ser. No. 07/519,382 filed May 4, 1990, entitled "General Purpose Compound Apparatus For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al;

(3) Application Serial No. 07/504,910, now U.S. Pat. No. 5,051,940, filed Apr. 4, 1990, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al.

These copending applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y.

The descriptions set forth in these copending applications are hereby incorporated into the present application by this reference thereto.

TECHNICAL FIELD

This invention relates to digital computers and digital data processors and particularly to digital computers and data processors capable of processing two or more instructions in parallel.

BACKGROUND OF THE INVENTION

The performance of traditional computers which execute instructions one at a time in a sequential manner has improved significantly in the past largely due to improvements in circuit technology. Such one-at-a-time instruction execution computers are sometimes referred to as "scalar" computers or processors. As the circuit technology is pushed to its limits, computer designers have had to investigate other means to obtain significant performance improvements.

Recently, so called "super scalar" computers have been proposed which attempt to increase performance by executing more than one instruction at a time from a single instruction stream. Such proposed super scalar machines typically decide at instruction execution time if a given number of instructions may be executed in parallel. Such decision is based on the operation codes (op codes) of the instructions and on data dependencies which may exist between adjacent instructions. The op codes determine the particular hardware components each of the instructions will utilize and, in general, it is not possible for two or more instructions to utilize the same hardware component at the same time nor to execute an instruction that depends on the results of a previous instruction (data dependency). These hardware and data dependencies prevent the execution of some instruction combinations in parallel. In this case, the affected instructions are instead executed by themselves in a non-parallel manner. This, of course, reduces the performance of a super scalar machine.

Proposed super scalar computers provide some improvement in performance but also have disadvantages which it would be desirable to minimize. For one thing, deciding at instruction execution time which instructions can be executed in parallel takes a small, but noticeable, amount of time which cannot be very readily masked by overlapping it with other normal machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Another disadvantage is that the decision making must be repeated all over again each time the same instructions are to be executed a second or further time.

SUMMARY OF INVENTION

As discussed in copending application Ser. No. 07/519,384, one of the attributes of a Scalable Compound Instruction Set Machine (SCISM) is: Don't do the parallel execution decision making at execution time. Do it at an earlier point in the overall instruction handling process. For example, do it ahead of the instruction buffer in those machines which have instruction buffers or instruction stacks. For another example, do it ahead of the instruction cache in those machines which flow the instructions through a cache unit.

Another attribute of a SCISM machine is to record the results of the parallel execution decision making so that such results are available in the event that those same instructions are used a second or further time.

In one embodiment of the present invention, the recording of the parallel execution decision making is accomplished by generating tags which are added to or inserted into the individual instructions in an instruction stream. These tags tell whether the instructions can be executed in parallel or whether they need to be executed one at a time. This instruction tagging process is sometimes referred to herein as "compounding". It serves, in effect, to combine two or more individual instructions into a single compound instruction for parallel processing purposes.

In a particularly advantageous embodiment of the present invention, the computer is one which includes a cache storage mechanism for temporarily storing machine instructions in their journey from a higher-level storage unit of the computer to the instruction execution units of the computer and the compounding or instructing tagging process is performed intermediate to the higher-level storage unit and the cache storage mechanism so that there is stored in the cache storage mechanism both instructions and compounding tags. As is known, the use of a well-designed cache storage mechanism, in and of itself, serves to improve the overall performance of a computer. And the storing of the compounding tags into the cache storage mechanism enables the tags to be used over and over again so long as the instructions in question remain in the cache storage mechanism. As is known, instructions frequently remain in a cache long enough to be used more than once.

For a better understanding of the present invention, together with other and further advantages and features thereof, reference is made to the following description taken in connection with the accompanying drawings, the scope of the invention being pointed out in the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring to the drawings:

FIG. 1 shows a representative embodiment of a portion of a digital computer system constructed in accordance with the present invention;

FIG. 2 shows a length of an instruction stream having compounding tags or tag fields associated with the instructions;

FIG. 3 shows in greater detail the internal construction of a representative embodiment of an instruction compounding unit which can be used in the computer system of FIG. 1;

FIG. 4 shows in greater detail a representative internal construction for each of the compound analyzer units of FIG. 3;

FIG. 5 shows an example of logic circuitry that may be used to implement the compound analyzer and tag generator portions of FIG. 3 which produce the compounding tags for the first three instructions in the instruction stream;

FIG. 6 is a table used in explaining the operation of the FIG. 5 example;

FIG. 7 shows a representative embodiment of a portion of a digital computer system and is used to explain how the compounded instructions may be processed in parallel by multiple functional instructions processing units;

FIG. 8 shows an example of a particular sequence of instructions which may be processed by the computer system of FIG. 7; and

FIG. 9 is a table used in explaining the processing of the FIG. 8 instruction sequence by the computer system of FIG. 7.

DESCRIPTION OF THE FIG. 1 EMBODIMENT

Referring to FIG. 1 of the drawings, there is shown a representative embodiment of a portion of a digital computer system or digital data processing system constructed in accordance with the present invention. This computer system is capable of processing two or more instructions in parallel. It includes a first storage mechanism for storing instructions and data to be processed. This storage mechanism is identified as higher-level storage 10. This storage 10 is a larger-capacity, lower-speed storage mechanism and may be, for example, a large-capacity system storage unit or the lower portion of a comprehensive hierarchical storage system or the like.

The computer system of FIG. 1 also includes an instruction compounding mechanism for receiving instructions from the higher-level storage 10 and associating with these instructions tag fields which indicate which of these instructions may be processed in parallel with one another. This instruction compounding mechanism is represented by instruction compounding unit 11. This instruction compounding unit 11 analyzes the incoming instructions for determining which ones may be processed in parallel. Furthermore, instruction compounding unit 11 produces for these analyzed instructions tag information or tag fields which indicate which instructions may be processed in parallel with one another and which ones may not be processed in parallel with one another.

The FIG. 1 system further includes a second storage mechanism coupled to the instruction compounding mechanism 11 for receiving and storing the analyzed instructions and their associated tag fields. This second or further storage mechanism is represented by compound instruction cache 12. Cache 12 is a smaller-capacity, higher-speed storage mechanism of the kind commonly used for improving the performance rate of a computer system by reducing the frequency of having to access the lower-speed storage mechanism 10.

The FIG. 1 system further includes a plurality of functional instruction processing units which operate in parallel with one another. These functional instruction processing units are represented by functional units 13, 14, 15, et cetera. These functional units 13-15 operate in parallel with one another in a concurrent manner and each, on its own, is capable of processing one or more types of machine-level instructions. Examples of functional units which may be used are: a general purpose arithmetic and logic unit (ALU), an address generation type ALU, a data dependency colapsing ALU (per copending application Ser. No. 504,910, now U.S. Pat. No. 5,051,940, a branch instruction processing unit, a data shifter unit, a floating-point processing unit, and so forth. A given computer system may include two or more of some of these types of functional units. For example, a given computer system may include two or more general purpose ALU's. Also, no given computer system need include each and every one of these different types of functional units. The particular configuration of functional units will depend on the nature of the particular computer system being considered.

The computer system of FIG. 1 also includes an instruction fetch and issue mechanism coupled to the compound instruction cache 12 for supplying adjacent instructions stored therein to different ones of the functional instruction processing units 13-15 when the instruction tag fields indicate that they may be processed in parallel. This mechanism is represented by instruction fetch and issue unit 16. Fetch and issue unit 16 fetches instructions from cache 12, examines their tag fields and operation code (op code) fields and, based upon such examinations, sends the instructions to the appropriate ones of the functional units 13-15.

If a desired instruction is resident in the compound instruction cache 12, the appropriate address is sent to the cache 12 to fetch therefrom the desired instruction. This is sometimes referred to as a "cache hit". If the requested instruction does not reside in cache 12, then it must be fetched from the higher-level storage 10 and brought into cache 12. This is sometimes referred to as a "cache miss". When a miss occurs, the address of the requested instruction is sent to the higher level storage 10. In response thereto, storage 10 commences the transfer out or read out of a line of instructions which includes the requested instruction. These instructions are transferred to the input of the instruction compounding unit 11, which unit proceeds to analyze these incoming instructions and generate the appropriate tag field for each instruction. The tagged instructions are thereafter supplied to the compound instruction cache 12 and stored therein for subsequent use, if needed, by the functional units 13, 14 and 15.

The instruction analysis performed in the instruction compounding unit 11 does require a certain relatively small amount of time. However, the instruction compounding analysis is performed only when an instruction cache miss occurs and is thus relatively infrequent.

FIG. 2 shows a portion of a stream of compounded or tagged instructions as they might appear at the output of the instruction compounding unit 11 of FIG. 1. As is seen, each instruction (Instr.) has a tag field added to it by the instruction compounding unit 11. The tagged instructions, like those shown in FIG. 2, are stored into the compound instruction cache 12. As needed, the tagged instructions in cache 12 are fetched by the instruction fetch and issue unit 16. As the tagged instructions are received by fetch and issue unit 16, their tag fields are examined to determine if they may be processed in parallel and their operation code (op code) fields are examined to determine which of the available functional units is most appropriate for their processing. If the tag fields indicate that two or more of the instructions are suitable for processing in parallel, then they are sent to the appropriate ones of the functional units in accordance with the codings of their op code fields. Such instructions are then processed concurrently with one another by their respective functional units.

When an instruction is encountered that is not suitable for parallel processing, then it is sent to the appropriate functional unit as determined by its op code and it is thereupon processed alone and by itself by the selected functional unit.

In the most perfect case, where plural instructions are always being processed in parallel, the instruction execution rate of the computer system would be N times as great as for the case where instructions are executed one at a time, with N being the number of instructions in the groups which are being processed in parallel.

DESCRIPTION OF FIG. 3 INSTRUCTION COMPOUNDING UNIT

FIG. 3 shows in greater detail the internal construction of a representative embodiment of an instruction compounding unit constructed in accordance with the present invention. This instruction compounding unit 20 is suitable for use as the instruction compounding unit 11 of FIG. 1. The instruction compounding unit 20 of FIG. 3 is designed for the case where a maximum of two instructions at a time may be processed in parallel. In this case, a one-bit tag field is used. A tag bit value of "one" means that the instruction is a "first" instruction. A tag bit value of "zero" means that the instruction is "second" instruction and may be executed in parallel with the proceeding first instruction. An instruction having a tag bit value of one may be executed either by itself or at the same time and in parallel with the next instruction, depending on the tag bit value for such next instruction.

Each pairing of an instruction having a tag bit value of one with a succeeding instruction having a tag bit value of zero forms a compound instruction for parallel execution purposes, that is, the instructions in such a pair may be processed in parallel with one another. When the tag bits for two succeeding instructions each have a value of one, the first of these instructions is executed by itself in a nonparallel manner. In the worst possible case, all of the instructions in the sequence would have a tag bit value of one. In this worst case, all of the instructions would be executed one at a time in a nonparallel manner.

The instruction compounding unit 20 of FIG. 3 includes a plural-instruction instruction register 21 for receiving a plurality of successive instructions from the higher-level storage unit 10. Instruction compounding unit 20 also includes a plurality of rule-based instruction analyzer mechanisms. Each such instruction analyzer mechanism analyzes a different pair of side-by-side instructions in the instruction register 21 and produces a compoundability signal which indicates whether or not the two instructions in its pair may be processed in parallel. In FIG. 3, there are shown a plurality of compound analyzer units 22-25. Each of these compound analyzer units 22-25 includes two of the instruction analyzer mechanisms just mentioned. Thus, each of these analyzers units 22-25 produces two of the compoundability signals. For example, the first compound analyzer unit 22 produces a first compoundability signal M01 which indicates whether or not Instructions 0 and 1 may be processed in parallel. Compound analyzer unit 22 also produces a second compoundability signal M12 which indicates whether or not Instructions 1 and 2 may be processed in parallel.

In a similar manner, the second compound analyzer unit 23 produces a first compoundability signal M23 which indicates whether or not Instructions 2 and 3 may be processed in parallel and a second compoundability signal M34 which indicates whether Instructions 3 and 4 may be processed in parallel. The third compound analyzer 24 produces a first compoundability signal M45 which indicates whether or not Instructions 4 and 5 may be processed in parallel and a second compoundability signal M56 which indicates whether or not Instructions 5 and 6 may be processed in parallel. The fourth compound analyzer 25 produces a first compoundability signal M67 which indicates whether or not Instructions 6 and 7 maybe processed in parallel and a second compoundability signal M78 which indicates whether Instructions 7 and 8 may be processed in parallel.

The instruction compounding unit 20 further includes a tag generating mechanism 26 responsive to the compoundability signals appearing at the outputs of the analyzer units 22-25 for generating the individual tag fields for the different instructions in the instruction register 21. These tag fields T0, T1, T2, etc. are supplied to a tagged instruction register 27, as are the instructions themselves, the latter being obtained from the input instruction register 21. In this manner, there is provided in the compounding unit output register 27 a tag field T0 for Instruction 0, a tag field T1 for Instruction 1, etc.

In the present embodiment, each tag field T0, T1, T2, etc. is comprised of a single binary bit. A tag bit value of "one" indicates that the immediately following instruction to which it is attached is a "first" instruction. A tag bit value of "zero" indicates that the immediately following instruction is a "second" instruction. An instruction having a tag bit value of one followed by an instruction having a tag bit value of zero indicates that those two instructions may be executed in parallel with one another. The tagged instructions in the compounding unit output register 27 are supplied to the input of the compound instruction cache 12 of FIG. 1 and are stored into such compound instruction cache 12.

It should be noted that the amount of register hardware shown in FIG. 3 can be reduced by storing the compound instructions directly to the compound instruction cache.

Referring now to FIG. 4, there is shown in greater detail the internal construction used for the compound analyzer unit 22 of FIG. 3. The other compound analyzer units 23-25 are of a similar construction. As shown in FIG. 4, the compound analyzer 22 includes instruction compatibility logic 30 for examining the op code of Instruction 0 and the op code of Instruction 1 and determining whether these two op codes are compatible for purposes of execution in parallel. Logic 30 is constructed in accordance with predetermined rules to select which pairs of op codes are compatible for execution in parallel. More particularly, logic 30 includes logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution in the particular hardware configuration used for the computer system being considered. If the op codes for Instructions 0 and 1 are compatible, then logic 30 produces at its output a binary one level signal. If they are not compatible, logic 30 produces a binary zero value on its output line.

Compound analyzer 22 further includes a second instruction compatibility logic 31 for examining the op codes of Instructions 1 and 2 and determining whether they are compatible for parallel execution. Logic 31 is constructed in the same manner as logic 30 in accordance with the same predetermined rules used for logic 30 to select which pairs of op codes are compatible for execution in parallel for the case of Instructions 1 and 2. Thus, logic 31 includes logic circuitry for implementing rules which define which types of instructions are compatible for parallel execution, these rules being the same as those used in logic 30. If the op codes for Instructions 1 and 2 are compatible, then logic 31 produces a binary one level output. Otherwise, it produces a binary zero level output.

Compound analyzer 22 further includes first register dependency logic 32 for detecting conflicts in the usage of the general purpose registers designated by the R1 and R2 fields of Instructions 0 and 1. These general purpose registers will be discussed in greater detail hereinafter. Among other things, dependency logic 32 may be constructed to detect the occurrence of a data dependency condition wherein a second instruction (Instruction 1) needs to use the results obtained by the performance of the proceeding instruction (Instruction 0). In this case, either the second instruction can be executed by the dependency collapsing hardware, thus executing in parallel with the first instruction, or the execution of the second instruction must await completion of the execution of the preceeding instruction and, hence, cannot be executed in parallel with the preceeding instruction. (It is noted that a technique for circumventing some data dependencies of this type will be discussed hereinafter.) If there are no register dependencies which prevent execution of Instructions 0 and 1 in parallel, then the output line of logic 32 is given a binary value of one. If there is a dependency, then it is given a binary value of zero.

Compound analyzer 22 further includes second register dependency logic 33 for detecting conflicts in the usage of the general purpose registers designated by the R1 and R2 fields of Instructions 1 and 2. This logic 33 is of the same construction as the previously discussed logic 32 and produces a binary one level output if there are no register dependencies or the register dependencies can be executed by the data dependency collapsing hardware, and a binary zero level output otherwise.

The output lines from the instruction compatibility logic 30 and the register dependency logic 32 are connected to the two inputs of an AND circuit 34. The output line of AND 34 has a binary one value if the two op codes being considered are compatible and if there are no register dependencies. This binary one value on the AND 34 output line indicates that the two instructions being considered are compoundable, that is, are executable in parallel. If, on the other hand, the AND 34 output line has a binary value of zero, then the two instructions are not compoundable. Thus, there is produced on the AND 34 output line a first compoundability signal M01 which indicates whether or not Instructions 0 and 1 may be processed in parallel. This M01 signal is supplied to the tag generator 26.

The output lines from the second compatibility logic 31 and the second dependency logic 33 are connected to the two inputs of AND circuit 35. AND 35 produces on its output line a second compoundability signal M12 which has a binary value of one if the two op codes being considered (op codes for Instructions 1 and 2) are compatible and if there are no register dependencies for Instructions 1 and 2 or register dependencies that can be executed by the data dependency collapsing hardware. Otherwise, the AND 35 output line has a binary value of zero. The output line from AND 35 runs to a second input of the tag generator 26.

The other compound analyzers 23-25 shown in FIG. 3 are of the same internal construction as shown in FIG. 4 for the first compound analyzer 22.

Referring now to FIG. 5, there is shown an example of the logic circuitry that can be used to implement the compound analyzer 22 and the portion of the tag generator 26 which is used to generate the first three tags, Tag 0, Tag 1 and Tag 2. For the example of FIG. 5, it is assumed that there are two categories of instructions which are designated as category A and category B. The rules for compounding these categories of instructions are assumed to be as follows:

(1) A can always compound with A

(2) A can never compound with B

(3) B can never compound with B

(4) B can always compound with A

(5) Rule (4) has preference over Rule (1).

Note that these rules are sensitive to the order of occurrence of the instructions.

It is further assumed that these rules are such that when they are observed, there will be no problems with register dependencies because the rules implicitly indicate that in case there is any interlock, such an interlock is always executable by the data dependency collapsing hardware. In other words, it is assumed for the FIG. 5 example, that the register dependency logics 32 and 33 of FIG. 4 are not needed. In such case, AND circuits 34 and 35 are also not needed and the output of logic 30 becomes the M01 signal and the output of logic 31 becomes the M12 signal.

For these assumptions, FIG. 5 shows the internal logic circuitry that may be used for the instruction compatibility logic 30 and the instruction compatibility logic 31 of FIG. 4. With reference to FIG. 5, the instruction compatibility logic 30 includes decoders 40 and 41, AND circuits 42 and 43 and OR circuit 44. The second instruction compatibility logic 31 includes decoders 41 and 45, AND circuits 46 and 47 and OR circuit 48. The middle decoder 41 is shared by both logics 30 and 31.

The first logic 30 examines the op codes OP0 and OP1 of Instructions 0 and 1 to determine their compatibility for parallel execution purposes. This is done in accordance with Rules (1)-(4) set forth above. Decoder 40 looks at the op code of the first instruction and if it is a category A op code, the A output line of decoder 40 is set to the one level. If OP0 is a category B op code, then the B output line of decoder 40 is set to a one level. If OP0 belongs to neither category A nor category B, then both outputs of decoder 40 are at the binary zero level. The second decoder 41 does a similar kind of decoding for the second op code OP1.

circuit 42 implements Rule (1) above. If OP0 is a category A op code and OP1 is also a category A p code, then AND 42 produces a one level output. Otherwise, the output of AND 42 is a binary zero level. AND 43 implements Rule (4) above. If the first op code is a category B op code and the second op code is a category A op code, then AND 43 produces a one level output. Otherwise, it produces a zero level output. If either AND 42 or AND 43 produces a one level output, this drives the output of OR circuit 44 to the one level, in which case, the compoundability signal M01 has a value of one. This one value indicates that the first and second instructions (Instructions 0 and 1) are compatible for parallel execution purposes.

If any other combination of op code categories is detected by decoders 40 and 41, then the outputs of AND 42 and AND 43 remain at the zero level and compoundability signal M01 has the noncompoundability-indicating value of zero. Thus, the occurrence of the combinations indicated by Rules (2) and (3) above do not satisfy AND's 42 and 43 and M01 remains at the zero level. If there are further catorgories of op codes in addition to catorgories A and B, their occurrences in the instruction stream do not activate the outputs of decoders 40 and 41. Hence, they likewise result in an M01 compoundability signal value of zero.

The second instruction compatibility logic 31 performs a similar type of op code analysis for the second and third instructions (Instructions 1 and 2). If the second op code OP1 is a category A op code and the third op code OP2 is a category A opcode, then, per Rule (1), AND 46 produces a one level output and the second compoundability signal M12 is driven to the compoundability-indicating binary one level. If, on the other hand OP1 is a category B opcode and OP2 is a category A opcode, then, per Rule (4), AND 47 is activated to produce a binary one level for the second compoundability signal M12. For any op code combinations other than those set forth in Rules (1) and (4), the M12 signal has a value of zero.

The M01 and M12 compoundability signals are supplied to the tag generator 26. FIG. 5 shows the logic circuitry that can be used in tag generator 26 to respond to the M01 and M12 compoundability signals to produce the desired tag bit values for Tags 0,1 and 2. The table of FIG. 6 shows the logic which is implemented by the tag generator 26 for Tags 0,1 and 2. A tag bit value of one indicates that the associated instruction is a "first" instruction for parallel execution purposes. A tag bit value of zero indicates that the associated instruction is a "second" instruction for parallel execution purposes. The only instruction pairs which are compounded and executed in parallel are those for which the first instruction in the pair has a tag bit value of one and the second instruction in the pair has a tag bit value of zero. Any instruction having a tag bit value of one which is followed by another instruction having a tag bit value of one is executed by itself in a singular manner and not in parallel with the following instruction.

For the case of the first row in FIG. 6, all three tag bits have a value of one. This means that each of Instructions 0 and 1 will be executed in a singular, nonparallel manner. For the second row of FIG. 6, Instructions 0 and 1 will be executed in parallel since Tag 0 has the required one value and Tag 1 has the required zero value. For the third row in FIG. 6, Instruction 0 will be executed in a singular manner, while Instructions 1 and 2 will be executed in parallel with one another. For the fourth row, Instructions 0 and 1 will be executed in parallel with one another.

For those cases where Tag 2 has a binary value of one, the status of its associated Instruction 2 is dependent on the binary value for Tag 3. If Tag 3 has a binary value of zero, then Instructions 2 and 3 can be executed in parallel. If, on the other hand, Tag 3 has a binary value of one, then Instruction 2 will be executed in a singular, nonparallel manner. It is noted that the logic implemented for the tag generator 26 does not permit the occurrence of two successive tag bits having binary values of zero.

An examination of FIG. 6 reveals the logic needed to be implemented by the portion of tag generator 26 shown in FIG. 5. As indicated in FIG. 6, Tag 0 will always have a binary value of one. This is accomplished by providing a constant binary value of one to tag generator output line 50 which constitutes the Tag 0 output line. An examination of FIG. 6 further reveals that the bit value for Tag 1 is always the opposite of the bit value of the M01 compoundability signal. This result is accomplished by connecting output line 51 for Tag 1 to the output of NOT circuit 52, the input of which is connected to the M01 signal line.

The binary level on Tag 2 output line 53 is determined by an OR circuit 54 and a NOT circuit 55. One input of OR 54 is connected to the M01 line. If M01 has a value of one, then Tag 2 has a value of one. This takes care of the Tag 2 values in the second and fourth rows of FIG. 6. The other input of OR 54 is connected by way of NOT 55 to the M12 signal line. If M12 has a binary value of zero, this value is inverted by NOT 55 to supply a binary one value to the second input of OR 54. This causes the Tag 2 output line 53 to have a binary one value. This takes care of the Tag 2 value for row one of FIG. 6. Note that for the row 3 case, Tag 2 must have a value of zero. This will occur because, for this case, M01 will have a value of zero and M12 will have a value of one which is inverted by NOT 55 to produce a zero at the second input of OR 54.

Implicit in the logic of FIG. 6 is a prioritization rule for the row four case where each of M01 and M12 has a binary value of one. This row four case can be produced by an instruction category sequence of BAA. This could be implemented by a tag sequence of 101 as shown in FIG. 6 or, alternatively, by a tag sequence of 110. In the present embodiment, Rule (5) is followed and the 101 sequence shown in FIG. 6 is chosen. In other words, the BA pairing is given preference over the AA pairing.

The 1,1 pattern for M01 and M12 can also be produced by an op code sequence of AAA. In this case, the 101 tag sequence of FIG. 6 is again selected. This is better because it provides a one value for Tag 2 and, hence, potentially enables Instruction 2 to be compounded with Instruction 3 if Instruction 2 is compatible with Instruction 3.

DESCRIPTION OF THE FIG. 7 EMBODIMENT

Referring to FIG. 7, there is shown a detailed example of how a computer system can be constructed for using the compounding tags of the present invention to provide parallel processing of machine-level computer instructions. The instruction compounding unit 20 used in FIG. 7 is assumed to be of the type described in FIG. 3 and, as such, it adds to each instruction a one-bit tag field. These tag fields are used to identify which pairs of instructions may be processed in the parallel. These tagged instructions are supplied to and stored into the compound instruction cache 12. Fetch/Issue control unit 60 fetches the tagged instructions from cache 12, as needed, and arranges for their processing by the appropriate one or ones of a plurality of functional instruction processing units 61, 62, 63 and 64. Fetch/Issue unit 60 examines the tag fields and op code fields of the fetched instructions. If the tag fields indicate that two successive instructions may be processed in parallel, then fetch/issue unit 60 assigns them to the appropriate ones of the functional units 61-64 as determined by their op codes and they are processed in parallel by the selected functional units. If the tag fields indicate that a particular instruction is to be processed in a singular, nonparallel manner, then fetch/issue unit 60 assigns it to a particular functional unit as determined by its op code and it is processed or executed by itself.

The first functional unit 61 is a branch instruction processing unit for processing branch type instructions. The second functional unit 62 is a three input address generation arithmetic and logic unit (ALU) which is used to calculate the storage address for instructions which transfer operands to or from storage. The third functional unit 63 is a general purpose arithmetic and logic unit (ALU) which is used for performing mathematical and logical type operations. The fourth functional unit 64 in the present example is a data dependency collapsing ALU of the kind described in the above-referenced copending application Ser. No.504,910, now U.S. Pat. No. 5,051,940. This dependency collapsing ALU 64 is a three-input ALU capable of performing two arithmetical/logical operations in a single machine cycle.

The computer system embodiment of FIG. 7 also includes a set of general purpose registers 65 for use in executing some of the machine-level instructions. Typically, these general purpose registers 65 are used for temporarily storing data operands and address operands or are used as counters or for other data processing purposes. In a typical computer system, sixteen (16) such general purpose registers are provided. In the present embodiment, general purpose registers 65 are assumed to be of the multiport type wherein two or more registers may be accessed at the same time.

The computer system of FIG. 7 further includes a high-speed data cache storage mechanism 66 for storing data operands obtained from the higher-level storage unit 10. Data in the cache 66 may also be transferred back to the higher-level storage unit 10. Data cache 66 may be of a known type and its operation relative to the higher-level storage 10 may be conducted in a known manner.

FIG. 8 shows an example of a compounded or tagged instruction sequence which may be processed by the computer system of FIG. 7. The FIG. 8 example is composed of the following instructions in the following sequence: Load, Add, Compare, Branch on Condition and Store. These are identified as instructions I1-I5, respectively. The tag bits for these instructions are 1,1,0,1 and 0, respectively. Because of the organization of the machine shown in FIG. 7, the Load instruction is processed in a singular manner by itself. The Add and Compare instructions are treated as a compound instruction and are processed in parallel with one another. The Branch and Store instructions are also treated as a compound instruction and are also processed in parallel with one another.

The table of FIG. 9 gives further information on each of these FIG. 8 instructions. The R/M column in FIG. 9 indicates the content of a first field in each instruction which is typically used to identify the particular one of general purpose registers 65 which contains the first operand. An exception is the case of the Branch on Condition instruction, wherein the R/M field contains a condition code mask. The R/X column in FIG. 9 indicates the content of a second field in each instruction, which field is typically used to identify a second one of the general purpose registers 65. Such register may contain the second operand or may contain an address index value (X). The B column in FIG. 9 indicates the content of a third possible field in each instruction, which field may identify a particular one of the general purpose registers 65 which contains a base address value. A zero in the B column indicates the absence of a B field or the absence of a corresponding address component in the B field. The D field of FIG. 9 indicates the content of a further field in each instruction which, when used for address generation purposes, includes an address displacement value. A zero in the D column may also indicate the absence of a corresponding field in the particular instruction being considered or, alternatively, an address displacement value of zero.

Considering now the processing of the Load instruction of FIG. 8, the fetch/issue control unit 60 determines from the tag bits for this Load instruction and the following Add instruction that the Load instruction is to be processed in a singular manner by itself. The action to be performed by this Load instruction is to fetch an operand from storage, in this case the data cache 66, and to place such operand into the R2 general purpose register. The storage address from which this operand is to be fetched is determined by adding together the index value in register X, the base value in register B and the displacement value D. The fetch/issue control unit 60 assigns this address generation operation to the address generation ALU 62. In this case, ALU 62 adds together the address index value in register X (a value of zero in the present example), the base address value contained in general purpose register R7 and the displacement address value (a value of zero in the present example) contained in the instruction itself. The resulting calculated storage address appearing at the output of ALU 62 is supplied to the address input of data cache 66 to access the desired operand. This accessed operand is loaded into the R2 general purpose register in register set 65.

Considering now the processing of the Add and Compare instructions, these instructions are fetched by the fetch/issue control unit 60. The control unit 60 examines the compounding tags for these two instructions and notes that they may be executed in parallel. As seen from FIG. 9, the Compare instruction has an apparent data dependency on the Add instruction since the Add must be completed before R3 can be compared. This dependency, however, can be handled by the data dependency collapsing ALU 64. Consequently, these two instructions can be processed in parallel in the FIG. 7 configuration. In particular, the control unit 60 assigns the processing of the Add instruction to ALU 63 and assigns the processing of the Compare instruction to the dependency collapsing ALU 64.

ALU 63 adds the contents of the R2 general purpose register to the contents of the R3 general purpose register and places the result of the addition back into the R3 general purpose register. At the same time, the dependency collapsing ALU 64 performs the following mathematical operation:

R3+R2-R4

The condition code for the result of this operation is sent to a condition code register located in branch unit 61. The data dependency is collapsed because ALU 64, in effect, calculates the sum of R3+R2 and then compares this sum with R4 to determine the condition code. In this manner, ALU 64 does not have to wait on the results from the ALU 63 which is performing the Add instruction. In this particular case, the numerical results calculated by the ALU 64 and appearing at the output of ALU 64 is not supplied back to the general purpose registers 65. In this case, ALU 64 merely sets the condition code.

Considering now the processing of the Branch instruction and the Store instruction shown in FIG. 8, these instructions are fetched from the compound instruction cache 12 by the fetch/issue control unit 60. Control unit 60 determines from the tag bits for these instructions that they may be processed in parallel with one another. It further determines from the op codes of the two instructions that the Branch instruction should be processed by the branch unit 61 and the Store instruction should be processed by the address generation ALU 62. In accordance with this determination, the mask field M and the displacement field D of the Branch instruction are supplied to the branch unit 61. Likewise, the address index value in register X and the address base value in register B for this Branch instruction are obtained from the general purpose registers 65 and supplied to the branch unit 61. In the present example, the X value is zero and the base value is obtained from the R7 general purpose register. The displacement value D has a hexadecimal value of twenty, while the mask field M has a mask position value of eight.

The branch unit 61 commences to calculate the potential branch address (0+R7+20) and at the same time compares the condition code obtained from the previous Compare instruction with the condition code mask M. If the condition code value is the same as the mask code value, the necessary branch condition is met and the branch address calculated by the branch unit 61 is thereupon loaded into an instruction counter in control unit 60. This instruction counter controls the fetching of the instructions from the compound instruction cache 12. If, on the other hand, the condition is not met (that is, the condition code set by the previous instruction does not have a value of eight), then no branch is taken and no branch address is supplied to the instruction counter in control unit 60.

At the same time that the branch unit 61 is busy carrying out its processing actions for the Branch instruction, the address generation ALU 62 is busy doing the address calculation (0+R7+0) for the Store instruction. The address calculated by ALU 62 is supplied to the data cache 66. If no branch is taken by the branch unit 61, then the Store instruction operates to store the operand in the R3 general purpose register into the data cache 66 at the address calculated by ALU 62. If, on the other hand, the branch condition is met and the branch is taken, then the contents of the R3 general purpose register is not stored into the data cache 66.

The foregoing instruction sequence of FIG. 8 is intended as an example only. The computer system embodiment of FIG. 7 is equally capable of processing various and sundry other instruction sequences.

The example of FIG. 8, however, clearly shows the utility of the compound instruction tags in determining which pairs of instructions may be processed in parallel with one another.

While there have been described what are at present considered to be preferred embodiments of this invention, it will be obvious to those skilled in the art that various changes and modifications may be made therein without departing from the invention, and it is, therefore, intended to cover all such changes and modifications as fall within the true spirit and scope of the invention.

* * * * *

United States Patent ***5,197,135***
Eickemeyer , et al. **March 23, 1993**

Memory management for scalable compound instruction set machines with in-memory compounding

Abstract

A digital computer system is described which is capable of processing 2 or more computer instructions in parallel and which has the capability of generating compounding tag information for those instructions, the compounding tag information being associated with instructions for the purpose of indicating groups of instructions which are to be concurrently executed. A compounding tag has a value which indicates the size of the group of instructions which are to be concurrently executed. The computer system includes a hierarchially-arranged memory which provides instructions to a CPU for execution. The instructions are compounded in the memory, and provision is made in the memory for storage of their compounding tags. In the event of modification of an instruction in memory, the invention provides for reduction of the value of the compounding tags for the modified instruction and instructions which are capable of being compounded with the modified instruction or for generation of new tag values for the modified instruction and instructions which are adjacent it in memory.

Inventors:	**Eickemeyer; Richard J.** (Endicott, NY); **Vassiliadis; Stamatis** (Vestal, NY); **Blaner; Bartholomew** (Newark Valley, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**543458**
Filed:	**June 26, 1990**

Current U.S. Class:	**712/217**; 712/23; 712/24; 712/215
Intern'l Class:	G06F 009/38
Field of Search:	395/375,800,775,650

References Cited [Referenced By]

U.S. Patent Documents

4439828	Mar., 1984	Martin	364/200.
4466057	Aug., 1984	Houseman et al.	395/375.
4722049	Jan., 1988	Lahti	395/375.
5086408	Feb., 1992	Sakata	395/600.

Other References

Acosta, R. D., et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, Fall, C-35, No. 9, Sep. 1986, pp. 815-828.
Anderson, V. W., et al, the IBM System/360 Model 91: "Machine Philosophy and Instruction Handling", computer structures: Principles and Examples (Siewiorek, et al, ed (McGraw-Hill, 1982, pp. 276-292.
Capozzi, A. J., et al, "Non-Sequential High-Performance Processing", IBM Technical Disclosure Bulletin, vol. 27, No. 5, Oct. 1984, pp. 2842-2844.
Chan, S., et al, "Building Parallelism into the Instruction Pipeline", High Performance Systems, Dec., 1989, pp. 53-60.
Murakami, K, et al, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining); A Novel High-Speed Single Processor Architecture", Proceedings of the Sixteenth Annual Symposium on Computer Architecture, 1989, pp. 78-85.
Smith, J. E., "Dynamic Instructions Scheduling and the Astronautics ZS-1", IEEE Computer, Jul., 1989, pp. 21-35.
Smith, M. D., et al, "Limits on Multiple Instruction Issue", ASPLOS III, 1989, pp. 290-302.
Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", Computer Structures, Principles, and Examples (Siewiorek, et al ed), McGraw-Hill, 1982, pp. 293-302.
Wulf, P. S., "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Jouppi, N. P., et al, "Available Instruction-Level Parallelism for Superscalar Pipelined Machines", ASPLOS III, 1989, pp. 272-282.
Jouppi, N. P., "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance", IEEE Transactions on Computers, vol. 38, No. 12, Dec., 1989, pp. 1645-1658.
Ryan, D. E., "Entails 80960: An Architecture Optimized for Embedded Control", IEEE Microcomputers, vol. 8, No. 3, Jun. 1988, pp. 63-76.
Coldwell, R. P., et al, "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, vol. 37, No. 8, Aug., 1988, pp. 967-979.
Fisher, J. A., "The VLIW Machine: A Multi-Processor for Compiling Scientific Code", IEEE Computer, Jul., 1984, pp. 45-53.

Berenbaum, A. D., "Introduction to the CRISP Instruction Set Architecture", Proceedings of Compcon, Spring, 1987, pp. 86-89.
Bandyopadhyay, S., et al, "Compiling for the CRISP Microprocessor", Proceedings of Compcon, Spring, 1987, pp. 96-100.
Hennessey, J., et al, "MIPS: A VSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.
Patterson, E. A., "Reduced Instruction Set Computers", Communications of the ACM, vol. 28, No. 1, (Jan., 1985, pp. 8-21.
Radin, G., "The 801 Mini-Computer", IBM Journal of Research and Development, vol. 27, No. 3, May, 1983, pp. 237-246.
Ditzel, D. R., et al, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of Compcon, Spring 1987, pp. 2-9.
Hwu, W. W., et al, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", IEEE Transactions on Computers, vol. C36, No. 12, Dec., 1987, pp. 1496-1594.
Lee, J. K. F., et al, "Branch Prediction Strategies in Branch Target Buffer Design", IEEE Computer, vol. 17, No. 1, Jan. 1984, pp. 6-22.
Riseman, E. M., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Dec., 1972, pp. 1405-1411.
Smith, J. E., "A Study of Branch Prediction Strategies", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, May 1981, pp. 135-148.
Archibold, James, et al, Cache Coherence Protocols: "Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol. 4, No. 4, Nov. 1986, pp. 273-398.
Baer, J. L., et al "Multi-Level Cache Hierarchies: Organizations, Protocols, and Performance" Journal of Parallel and Distributed Computing vol. 6, 1989, pp. 451-476.
Smith, A. J., "Cache Memories", Computing Surveys, vol. 14, No 3 Sep., 1982, pp. 473-530.
Smith, J. E., et al, "A Study of Instructions Cache Organizations and Replacement Policies", IEEE Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun., 1983, pp. 132-137.
Vassiliadis, S., et al, "Condition Code Predictory for Fixed-Arithmetic Units", International Journal of Electronics, vol. 66, No. 6, 1989, pp. 887-890.
Tucker, S. G., "The IBM 3090 System: An Overview", IBM Systems Journal, vol. 25, No. 1, 1986, pp. 4-19.
IBM Publication NO. SA22-7200-0, Principles of Operation, IBM Enterprise Systems Architecture/370, 1988.
The Architecture of Pipelined Computers, by Peter M. Kogge Hemisphere Publishing Corporation, 1981.
IBM Technical Disclosure Bulletin (vol. 33 No. 10A, Mar. 1991), by R. J. Eberhard.

Primary Examiner: Kriess; Kevin A.
Attorney, Agent or Firm: Baker, Maxham, Jester & Meador

Claims

We claim:

1. In a computer system capable of concurrently executing up to N instructions in a sequence of scalar instructions, the sequence including compounding tags associated with the scalar instructions, the compounding tags having values conditioned to indicate how many instructions are to be concurrently executed, a mechanism for managing compounding tag values of scalar instructions which are stored in a real memory of the computer system, the mechanism comprising:

a merging means connected to the real memory for merging a modified instruction from the real memory with non-modified instructions in the real memory; and

a tag reduction unit connected to the merging means and to the real memory for reducing the values of the compounding tags of the modified instruction and up to N-1 of the instructions in the real memory with which the modified instruction could be compounded.

2. The mechanism of claim 1, wherein the tag reduction unit reduces the values of the compounding tags of the modified instruction to zero.

3. The mechanism of claim 2, wherein the tag reduction unit reduces the values of the compounding tags of all of the instructions in the real memory with which the modified instruction could be compounded to zero.

4. The mechanism of claim 1 wherein compounding tags are provided for every two bytes in the sequence of scalar instructions.

5. The mechanism of claim 1, wherein L is the length in bytes of the longest scalar instruction capable of being compounded with another scalar instruction, A is the number of bytes covered by a compounding tag, and wherein the tag reduction unit reduces the values of the compounding tags of up to (NL-A) bytes preceding the modified instruction in the sequence of scalar instructions.

6. The mechanism of claim 1, wherein the compounding tags are generated by an instruction compounding unit in the real memory of the computer system.

7. The mechanism of claim 6, wherein the real memory of the computer system includes a high-capacity, medium speed main memory and a small-capacity, high speed cache connected to the main memory, to the merging means and to the tag reduction unit.

8. The mechanism of claim 7, wherein the instruction compounding unit is connected to the main memory and to the cache for generating compounding tags for instructions in the cache.

9. The mechanism of claim 7, wherein the instruction compounding unit is connected to the main memory for generating compounding tags for instructions in the main memory.

10. In a computer system which receives a sequence of scalar instructions and includes compounding means for grouping instructions for concurrent execution by generating compounding tags for groups of

up to N instructions, the compounding tags having values indicating the number of instructions in the groups, a method for managing the compounding tags of instructions stored in the real memory of the computer system, the method including the steps of:

during operation of the computer system, modifying an instruction stored in the real memory;

merging the instruction with instructions in the real memory;

reducing the value of the compounding tag for the real instruction to zero; and

reducing the values of the compounding tags of up to N-1 instructions in the real memory with which the instruction can be compounded.

11. The method of claim 10, wherein the step of reducing the values of the compounding tags of N-1 instructions includes reducing the value to zero.

12. The method of claim 10, wherein L is the length in bytes of the longest scalar instruction of the sequence which can be compounded with another scalar instruction of the sequence and A is the number of bytes associated with a compounding tag, the step of reducing the values of the compounding tags of N-1 instructions including reducing the values of the compounding tags of up to (NL-A) bytes in the sequence preceding the instruction.

13. The method of claim 12, wherein the values of the compounding tags of the (NL-A) bytes are reduced to zero.

14. In a computer system capable of concurrently executing up to N compounded instructions in a sequence of scalar instructions, the sequence including compounding tags which accompany the scalar instructions, the compounding tags having values conditioned to indicate how many instructions are compounded for concurrent execution, a mechanism for managing compounding tag values of scalar instructions which are stored in real memory of the computer system, the mechanism comprising:

a merging means connected to the real memory for merging a modified instruction with non-modified instructions in the real memory; and

a recompounding mechanism connected to the merging means and to the real memory for generating compounding tags for the modified instruction and at least N-1 non-modified instructions adjacent the modified instruction in real memory.

15. The mechanism of claim 14, wherein L is the length in bytes of the longest instruction which can be compounded with another instruction and A is the number of bytes to which a compounding tag applies, the recompounding mechanism being for generating compounding tags for the modified instruction and at least (NL-A) bytes adjacent the modified instruction.

Description

CROSS REFERENCE TO RELATED APPLICATIONS

The present United States patent application is related to the following co-pending United States patent applications:

(1) application Ser. No.: 07/519,384 filed May 4, 1990, entitled "Scalable Compound Instruction Set Machine Architecture", the inventors being Stamatis Vassiliadis et al;

(2) application Ser. No.: 07/519,382 filed May 4, 1990, entitled "General Purpose Compound Apparatus For Instruction-Level Parallel Processors", the inventors being Richard J. Eickemeyer et al;

(3) application Ser. No.: 07/504,910 filed Apr. 4, 1990, entitled "Data Dependency Collapsing Hardware Apparatus", the inventors being Stamatis Vassiliadis et al, now U.S. Pat. No. 5,051,940;

(4) application Ser. No.: 07/522,291 filed May 10, 1990, entitled "Compounding Preprocessor For Cache", the inventors being Bartholomew Blaner et al; and

(5) application Ser. No.: 07/543,464 filed Jun. 26, 1990, entitled "An In-Memory Preprocessor for a Scalable compound Instruction Set Machine Processor, the inventors being Richard Eickemeyer et al.

These co-pending applications and the present application are owned by one and the same assignee, namely, International Business Machines Corporation of Armonk, N.Y.

The descriptions set forth in these co-pending applications are hereby incorporated into the present application by this reference thereto.

BACKGROUND OF THE INVENTION

This invention relates to digital computers and digital data processors, and particularly to digital computers and data processors capable of executing two or more instructions in parallel.

Traditional computers which receive a sequence of instructions and execute the sequence one instruction at a time are known. The instructions executed by these computers operate on single-valued objects, hence the name "scalar" for these computers.

The operational speed of traditional scalar computers has been pushed to its limits by advances in circuit technology, computer mechanisms, and computer architecture. However, with each new generation of competing machines, new acceleration mechanisms must be discovered for traditional scalar machines.

A recent mechanism for accelerating the computational speed of uni-processors is found in reduced instruction set architecture that employs a limited set of very simple instructions. Another acceleration mechanism is complex instruction set architecture which is based upon a minimal set of complex multi-operand instructions. Application of either of these approaches to an existing scalar computer would require a fundamental alteration of the instruction set and architecture of the machine. Such a far-reaching transformation is fraught with expense, downtime, and an initial reduction in the machine's reliability and availability.

In an effort to apply to scalar machines some of the benefits realized with instruction set reduction, so-called "superscalar" computers have been developed. These machines are essentially scalar machines whose performance is increased by adapting them to execute more than one instruction at a time from an instruction stream including a sequence of single scalar instructions. These machines typically decide at instruction execution time whether two or more instructions in a sequence of scalar instructions may be executed in parallel. The decision is based upon the operation codes (OP codes) of the instructions and on data dependencies which may exist between instructions. An OP code signifies the computational hardware required for an instruction. In general, it is not possible to concurrently execute two or more instructions which utilize the same hardware (a hardware dependency) or the same operand (a data dependency). These hardware and data dependencies prevent the parallel execution of some instruction combinations. In these cases, the affected instructions are executed serially. This, of course, reduces the performance of a super scalar machine.

Superscalar computers suffer from disadvantages which it is desirable to minimize. A concrete amount of time is consumed in deciding at instruction execution time which instructions can be executed in parallel. This time cannot be readily masked by overlapping with other machine operations. This disadvantage becomes more pronounced as the complexity of the instruction set architecture increases. Also, the parallel execution decision must be repeated each time the same instructions are to be executed.

In extending the useful lifetime of existing scalar computers, every means of accelerating execution is vital. However, acceleration by means of reduced instruction set architecture, complex instruction set architecture, or superscalar techniques is potentially too costly or too disadvantageous to consider for an existing scalar machine. It would be preferred to accelerate the speed of execution of such a computer by parallel, or concurrent, execution of instructions in an existing instruction set without requiring change of the instruction set, change of machine architecture, or extension of the time required for instruction execution.

SUMMARY OF THE INVENTION

In co-pending patent application Ser. No. 07/519,384 a scalable compound instruction set machine (SCISM) architecture is proposed in which instruction level parallelism is achieved by statically analyzing a sequence of scalar instruction at a time prior to instruction execution to generate compound instructions formed by adjacent grouping of existing instructions in the sequence which are capable of parallel execution. Relevant control information in the form of compound tags is added to the instruction stream to indicate where a compound instruction starts, as well as to indicate the number of existing instructions which are incorporated into a compound instruction. Relatedly, when used herein, the term "compounding" refers to the grouping of instructions contained in a sequence of instructions, the grouping being for the purpose of concurrent or parallel execution of the grouped instructions. At minimum, compounding is satisfied by "pairing" of two instructions for simultaneous execution. Preferably, compounded instructions are unaltered from the forms they have when presented for scalar execution. As explained below, compounded instructions are signified by compounding tag information, that is, bits accompanying the grouped instructions which denote the grouping of the instructions for parallel execution.

In a digital computer system which includes a means for executing a plurality of instructions in parallel, a particularly advantageous embodiment of the invention of this application is based upon a memory architecture which provides for compounding of instructions prior to their issue and execution. This memory structure provides instructions to the CPU (central processing unit) of a computer. Typically, a hierarchical memory structure includes a high-speed cache storage containing currently accessed instructions, a medium speed main memory connected to the cache, and a low-speed, high-capacity auxiliary storage. Typically, the cache and main storage (referred to collectively as "real storage") contain instructions which can be directly referenced for execution. Access to instructions in the auxiliary storage is had through an input/output (I/O) adaptor connected between the main memory and the auxiliary storage.

In co-pending patent application Ser. No. 07/543,464 an in-memory preprocessor for a SCISM architecture is proposed in which an instruction compounding mechanism in real storage produces compounding tag information for a sequence of scalar instructions, the compounding tag information indicating instructions of the sequence which may be executed in parallel. The instruction compounding unit produces the compounding tag information as a page of instructions is being prefetched and stored in the main memory. Although the particular embodiment of that patent application teaches compounding of up to two instructions, it is contemplated that up to N instructions may be compounded for concurrent execution in a scalar computer.

In patent application Ser. No. 07/522,291, a compounding preprocessor for a cache in a hierarchical memory is disclosed in which the instruction compounding unit is located between the main memory and cache of a hierarchical storage organization. The instruction compounding unit produces compounding tag information for the instructions in a line of instructions being fetched into the cache from the main memory.

In both of these applications, instructions in the cache have accompanying compounding tags to which a plurality of parallel execution units respond by executing one or more of the instructions in a group of up to N instructions in parallel.

In either of these cases, if the program issues a WRITE into the instruction text, any compound tags associated with the modified text must be invalidated and that section of the code must be run without compounding for so long as it resides in the memory.

Typically, when text in a cache is modified by a WRITE, the modified text is placed back into the cache by merging it with a line of text obtained from the cache. In this regard, merging means that the modified text is placed into the line of text containing its prior unmodified form at the location in the line where the prior form occurs.

The invention can be understood in the context of a computer system capable of concurrently executing up to N instructions in a sequence of scalar instructions, the sequence including compounding tags which accompany the scalar instructions and which are activated to indicate the instructions to be concurrently executed. The invention is a mechanism for managing the compounding tags of scalar instructions which are stored in the real storage of the computer system, and it includes a merging unit connected to the real memory for merging a modified instruction from the real memory with non-modified instructions in the real memory. A tag reduction unit connected to the merging unit and to the real memory deactivates the compounding tags of the modified instruction and N-1 instructions in the real memory with which the modified instruction could be compounded.

The invention is also expressed as a method practiced in a computer system, the computer system receiving a sequence of scalar instructions and including a compounder which groups instructions for concurrent execution by generating compounding tags for groups of instructions. The compounding tags are activated by the compounding unit for groups of up to N instructions to indicate that the instructions are to be concurrently executed. The method manages the compounding tags of instructions stored in the real memory of the computer system and includes the following steps:

during execution of an instruction stored in the real memory, modifying the instructions;

merging the instruction with instructions in the real memory;

reducing the compounding tags of the instructions; reducing the compounding tags of N-1 instructions in the real memory with which the instruction can be compounded.

Alternatively, the practice of this invention may include management of the compounding tags of scalar instructions which are stored in the real storage of a computer system by recompounding the tags of the modified instruction, the N-1 instructions preceding the modified instruction in real memory, and the N-1 instructions succeeding modified instruction in the real memory. In this case, compounding tag management consists simply of recompounding the modified instruction with the instructions which surround it in real memory.

For a better understanding of the invention, together with its advantages and features, reference is made to the following description and the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is high-level schematic diagram of a computing system which is capable of compounding instructions in a sequence of scalar instructions for concurrent execution.

FIG. 2 is a timing diagram for a uni-processor implementation showing the parallel execution of certain instructions which have been selectively grouped in a compound instruction stream.

FIG. 3 is a block diagram of a hierarchical memory organization in a scalable compound instruction set machine with in-memory processing.

FIG. 4 is an analytical chart for instruction stream text which has been compounded by association of compounding tags.

FIGS. 5A and 5B illustrate alternative implementations of compounding tags in memory.

FIGS. 6A-6J are flow diagrams which illustrate how compounding tags are managed in the hierarchical memory organization of FIG. 3.

FIG. 7 is a block diagram illustrating a mechanism for compounding tag management in the memory of FIG. 3.

FIG. 8 is a more detailed block diagram of tag reduction hardware in the mechanism of FIG. 7 in the case where up to 3 instructions can be compounded.

FIG. 9 is a table illustrating the operation of a tag reduction unit in FIG. 8.

FIG. 10 is a more detailed block diagram of tag reduction hardware for the compounding of up to 2 instructions.

FIG. 11 is a table illustrating operation of a tag reduction unit in FIG. 10.

FIG. 12 illustrates the operation of a tag reduction algorithm according to this invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring to FIG. 1 of the drawings, there is shown a representative embodiment of a portion of a digital computer system for a digital data processing system constructed in accordance with the present invention. This computer system is capable of executing 2 or more instructions in parallel. The system includes the capability of compounding instructions for parallel or concurrent execution. In this regard, "compounding" refers to the grouping of a plurality of instructions in a sequence of scalar instructions, wherein the size of the grouping is scalable from 1 to N. For example, the sequence of scalar instructions could be drawn from an existing set of scalar instructions such as that used by the IBM System/370 products.

In order to support the concurrent execution of a group of up to N instructions, the computer system includes a plurality of instruction execution units which operate in parallel in a concurrent manner; each on its own, is capable of processing one or more types of machine-level instructions.

As is generally shown in FIG. 1, an instruction compounding unit 20 takes a stream of binary scalar instructions 21 and selectively groups some of the adjacent scalar instructions to form encoded compound instructions. A resulting compounded instruction stream 22, therefore, provides scalar instructions to be executed singly or in compound instructions formed by groups of scalar instructions to be executed in parallel. When a scalar instruction is presented to an instruction processing unit 24, it is routed to the appropriated one of a plurality of execution units for serial execution. When a compound instruction is presented to the instruction processing unit 24, its scalar components are each routed to an appropriate execution unit for simultaneous parallel execution. Typical functional units include, but are not limited to, an arithmetic logic unit (ALU) 26, 28 a floating point arithmetic unit (FP) 30 and a storage address generation unit (AU) 32.

It is understood that compounding is intended to facilitate the parallel issue and execution of instructions in all computer architecture capable of processing multiple instructions per cycle.

Referring now to FIG. 2, compounding can be implemented in a uni-processor environment where each functional unit executes a scalar instruction (S) or, alternatively, a compound instruction (CS). As shown in the drawing, an instruction stream 33 containing a sequence of scalar and compounded scalar instructions has control tags (T) associated with each compound instruction. Thus, a first scalar instruction 34 could be executed singly by functional unit A in cycle 1; a triplet compound instruction 36 identified by tag T3 could have its 3 compounded scalar instructions executed in parallel by functional units A, C, and D in cycle 2; another compound instruction 38 identified by tag T2 could have its pair of compounded scalar instructions executed in parallel by functional units A and B in cycle 3; a second scalar instruction 40 could be executed singly by functional unit C in cycle 4; a large group compound instruction 42 could have its 4 compounded scalar instructions executed in parallel by functional units A-D in cycle 5; and a third scalar instruction 44 could be executed singly by functional unit A in cycle 6.

One example of a computer architecture which can be adapted for handling compound instructions is an IBM System/370 instruction level architecture in which multiple scalar instructions can be issued for execution in each machine cycle. In this context, a machine cycle refers to a single pipeline stage required to execute a scalar instruction. When an instruction stream is compounded, adjacent scalar instructions are selectively grouped for the purpose of concurrent or parallel execution.

In general, an instruction compounding facility will look for classes of instructions that may be executed in parallel. When compatible sequences of instructions are found, a compound instruction is created.

Compounding, per se, is not the subject of this patent application. Reference is given to patent application Ser. No. 07/519,384 filed May 4, 1990, and U.S. patent application Ser. No. 07/519,382 filed May 4, 1990, for an understanding of compounding generally. An instantiation of an instruction compounding unit for pairwise compounding is given in U.S. patent application Ser. No. 07/543,464 filed Jun. 26, 1990.

Generally, it is useful to provide for compounding at a time prior to instruction issue so that the process can be done once for an instruction or instructions that may be executed many times. It has been proposed to locate the instruction compounding functionality in the real memory of a computer system in order to implement compounding in hardware, after compile time, yet prior to instruction issue. Such compounding is referred to as "in-memory compounding" and can be understood with reference to U.S. patent application Ser. No. 07/522,291 filed May 10, 1990, and U.S. patent application Ser. No. 07/543,464 filed Jun. 26, 1990.

Generally, in-memory compounding is illustrated in FIG. 3. In FIG. 3, a hierarchical memory organization includes an I/O adaptor 40 which interfaces with auxiliary storage devices and with a computer real memory. The real memory of the organization includes a medium speed, relatively high capacity main memory 46 and a high-speed, relatively low capacity instruction cache 48. (The main memory and cache collectively are referred to herein as "real memory", "real storage", or, simply "memory".) A stream of instructions is brought in from auxiliary storage devices by way of the I/O adaptor 40, and stored in blocks called "pages" in the main memory 46. Sets of contiguous instructions called "lines" are moved from the main memory 46 to the instruction cache 48 where they are available for high-speed reference for processing by the instruction fetch and issue unit 50. Instructions which are fetched from the cache are issued, decoded at 52, and passed to the functional units 56, 58, . . . , 60 for execution.

During execution, when reference is made to an instruction which is in the program, the instruction's address is provided to a cache management unit 62 which uses the address to fetch one or more instructions, including addressed instruction, from the instruction cache 48 into the queue in the unit 50. If the addressed instruction is in the cache, a cache "hit" occurs. Otherwise, a cache "miss" occurs. A cache miss will cause the cache management unit 62 to send the line address of the requested instruction to a group of storage management functions 64. These functions can include, for example, real storage management functions which use the line address provided by the cache management unit 62 to determine whether the page containing the addressed line is in the main memory 46. If the page is in main memory, the real storage management will use the line address to transfer a line containing the missing instruction from the main memory 46 to the instruction cache 48. If the line containing the requested instruction is not in the main memory, the operating system will activate another storage management function, providing it with the identification of the page containing the needed line. Such a storage management function will send to the I/O adaptor 40 an address identifying the page containing the line. The I/O adaptor 40 will bring the page from auxiliary storage and provide it to the main memory 46. To make room for the fetched page, the storage management function selects a page in the main memory 46 to be replaced by the fetched page. In SCISM architecture, it is contemplated that the replaced page is returned to auxiliary storage through the I/O adaptor without compounding tag information. In this manner, those instructions most likely to be immediately required during execution of an instruction sequence are adjacent the functional units in the instruction cache 48. The hierarchical memory organization provides the capability for fast retrieval of instructions that are required but not in the cache.

In the context of SCISM architecture, in-memory instruction compounding can be provided by an instruction compounding unit 70 which is located functionally between the I/O adapter 40 and the main memory 46 so that compounding of the scalar instruction stream can take place at the input to, or in, the main memory 46. In this location, instructions can be compounded during an ongoing page fetch. Alternatively, the instruction compounding unit can occupy the position 72 between the main memory 46 and the instruction cache 48 and compound instructions are formed line-by-line as they are fetched into the instruction cache 48.

The particular technique for compounding is a matter of design choice. However, for purposes of illustration, a technique for creating compound instructions formed from adjacent scalar instructions is illustrated in FIG. 4. As FIG. 4 shows, instructions may occupy 6 bytes (3 half words), 4 bytes (2 half words), or 2 bytes (1 half word) of text. For this example, the rule for compounding a set of instructions which includes variable instruction lengths provides that all instructions which are 2 bytes or 4 bytes long are compoundable with each other. That is, a 2 byte instruction is capable of parallel execution in this particular example with another 2 byte or another 4 byte instruction and a 4 byte instruction is capable of parallel execution with another 2 byte or another 4 byte instruction. The rule further provides that all instructions which are 6 bytes long are not compoundable. Thus, a 6 byte instruction is only capable of execution singly by itself. Of course, compounding is not limited to these exemplary rules, but can embrace a plurality of rules which define the criteria for parallel execution of existing instructions in a specific configuration for a given computer architecture.

The instruction set used for this example is taken from the System/370 architecture. By examining the OP code for each instruction, the length of each instruction can be determined from an instruction length code (ILC) in the op code. The instruction's type is further defined in other op code bits. Once the type and length of the instruction is determined, a compounding tag containing tag bits is then generated for that specific instruction to denote whether it is to be compounded with one or more other instructions for parallel execution, or to be executed singly by itself.

In the example (which is not limiting), if 2 adjacent instructions can be compounded, the tag bits, which are generated in memory, are "1" for the first compounded instruction and "zero" for the second compounded instruction. However, if the first and second instructions cannot be compounded, the tag bit for the first instruction is "zero" and the second and third instructions are then considered for compounding. Once an instruction byte stream has been processed in accordance with the chosen compounding technique and the compounding bits encoded for various scalar instructions, more optimum results for achieving parallel execution may be obtained by using a bigger window for looking at larger groups of instructions and then picking the best combination of N instructions for compounding.

With the compounding example of FIG. 4, a C-vector 72 is generated which shows the values for compounding tag bits for the particular sequence 70 of instructions where pairwise compounding is employed. Based on the values of the compounding bits, the second and third instructions at text sequence byte positions 6-9 form a compounded pair as indicated by the "1" in the identifier bit for the second instruction. The fourth and fifth instruction at text byte locations 10-13 form another compounded pair as indicated by "1" in the identifier bit for the fourth instruction. The seventh and eighth instructions at byte locations 22 and 24 also form a compounded pair as indicated by the "1" in the identifier bit for the seventh instruction.

The compounding tags of the C-vector 72 are generated by the instruction compounding unit. In generating the tags, the compounding unit provides a compounding tag bit for each half word of the instruction text sequence. For compounding in general, the instruction compounding unit can generate a tag containing control information which is to accompany each instruction in the compounded text sequence; that is, tag bits are generated for each non-compounded scalar instruction as well as for each compounded scalar instruction in a group of N compounded instructions.

The example of compounding N instructions where N=2 provides the smallest grouping of scalar instructions to form a compound instruction, and uses the following preferred encoding procedure. It is asserted that all instructions are aligned on a half word (2 byte) boundary with lengths of either 2, 4, or 6 bytes, in which case, a tag is needed for every half word. In this small grouping example, a 1 bit tag indicates that an instruction beginning in the byte under consideration is compounded with the following instruction, while a "zero" indicates that the instruction beginning in the byte under consideration is not compounded. The tag bit associated with half words that do not contain the first byte of an instruction is ignored. The tag bit for the first byte of second instruction in the compounded pair is also ignored. As a result, this encoding procedure for tag bits means that in the simplest case, only 1 bit of information is needed for an instruction to identify a compounded instruction.

In the general case, where up to N scalar instructions can be grouped together to form a compound instruction, additional tag bits are required. The minimum number of bits needed to indicate the specific number of scalar instructions actually compounded is the logarithm to the base 2 (rounded up to the nearest whole number) of the maximum number N of scalar instructions that can be grouped to form a compounded instruction. Thus, in the example, N=2 and 1 tag bit is needed for each compound instruction. If N=3 or 4, 2 tag bits are needed for each compounded instruction.

There are a number of ways to implement the storage of compounding tags in memory as illustrated in FIGS. 5a and 5b. Both examples assume an 8 byte memory bus plus extra lines for tags, a 64 byte cache line as the basic memory transfer, and 1 tag bit for every 2 bytes of text in memory. One cache line is shown in each example. In keeping with the foregoing discussion, the number of compounding tag bits is determined by the maximum number N of instructions to be compounded and the information available to the compounder.

The simplest tag storage implementation is to increase the internal word size of the processor as illustrated in FIG. 5A. This implies that the tag bits are appended to instructions or inserted into the instruction stream at each half word. In FIG. 5A, a cache line organized into 8 storage locations is illustrated. Without compounding, each location is 8 bytes (64 bits) wide. With 8 locations, a 64 byte cache line is stored. With 1 compounding tag per half word, and 2-way compounding (N=2), a minimum of 1 compounding bit for every half word of instruction text is required. Thus, 4 compounding tag bits (T) are required for every 8 bytes. The implication is that the memory word size must be expanded from 64 to 68 bits. For 3 or 4 way compounding, a 2 bit compounding tag for each 2 bytes of instruction would require that the memory word size be extended to 72 bits. FIG. 5A illustrates a memory structure for the first case, that is, N=2 and an 8 byte memory bus. The memory bus and internal word size are expanded to 68 bits. The drawback to this scheme is that a new memory design is required implying, for example, error correction for larger words. A second approach is illustrated in FIG. 5B and utilizes a tag memory that is separate from, but operated in parallel with, the text memory. This structure implies that tags are separate from the instruction text. However, as with FIG. 5A, the tags accompanying their respective instructions, the parallel operation of FIG. 5B implies different memory bus lines. In this case, the internal memory word size is unchanged; however, the bus size might increase to accommodate parallel operation of the tag memory. This approach has several advantages over the wider word approach. First, the tag memory may cover only part of the words in main memory. If the operating system uses certain parts of the memory only for data pages, tags are not necessary over these parts. The design of FIG. 5B may be hardwired. Alternatively, a separate tag memory controller will be provided, commands to which will indicate that certain pages contain data only, in which case, the controller will not map the memory page address into a tag memory address for these pages. The second advantage is that the tag memory could be removed to produce a lower cost system. This broadens the performance range possible in a family of computers. If N-way compounding is employed and N is greater than 2, the growth in tag bits can be accommodated by substitution of new tag memory without requiring a change in the main memory design. In the design of FIG. 5B, each memory can have its own error correction. Relatedly, error correction is not necessary in tag memory, only error detection is needed. A detected tag bit error can always be "corrected" to all zeros, which will lose compounding, but maintain correct execution.

Other approaches to implementing tags are possible but are not illustrated. These other approaches are discussed at length in co-pending U.S. patent application Ser. No. 07/543,464, filed Jun. 26, 1990.

THE INVENTION

The result of instruction execution may require modification of an instruction in the executing program by alteration of one of its data fields. When the instruction is thus modified, all superseded copies of the instruction which are in memory must be replaced by the modified instruction. If the instruction is compounded with one or more other instructions, modification of the instruction may alter the compounding conditions. The conditions may be changed to the point where the instruction is no longer compoundable, is compoundable with fewer instructions than previously, or is compoundable only with different instructions. Alternatively, instruction modification may provide an opportunity for recompounding the instruction with its surrounding instructions.

In order to retain the benefits of compounding, without sacrificing data consistency when an instruction in a compounded stream is altered, the inventors have chosen two solutions. The first solution is to reduce compounding tag values in order to delete the altered instruction from a compounded instruction. This requires adjustment of the values of the compounding tags which precede the instruction in the instruction stream. The second solution is simply to recompound the instructions which surround the modified instruction. This may result in the generation of new compounding tags for these instructions.

Referring again to FIG. 3, a compounding tag management unit 73 according to the invention is illustrated in a feedback path around the instruction cache 48. In this location, compounding tag management by either tag reduction or recompounding can be activated in response to alteration of an instruction in the cache. Although only a single cache is illustrated in FIG. 3, it is contemplated that the invention may be practiced in a computer with plural CPUs and plural caches. For maintaining data coherence among all of the components of the real memory, the inventors assert that a compounding tag management unit 74 may also be located in a feedback path around the main memory 46 to accommodate compounding tag management of instructions in the real memory.

Compounding tag reduction is now explained from the standpoint of compounded instructions in the instruction cache 48. This explanation is not intended to be limiting, it being understood that cache coherence may require compounding tag reduction for instructions in the main memory 46 and for other caches which the system may include. Referring now to FIGS. 3 and 4, the C-vector 72 accompanies the text sequence 70 when the text sequence is stored in the instruction cache. Further, that each compounding tag bit in the C-vector 72 is associated with a respective 2 bytes of the text sequence 70. The cache management unit 62 monitors the address and length of fields that are overwritten. Now, assume that the text of an instruction in byte locations 14 and 15 is modified with a WRITE. Clearly, the tag for this instruction must be reduced to zero, as the compounding status of the instruction is no longer known. If it is assumed that the selected compounding procedure operates only on compounding tag bits associated with the first half word of an instruction, no tags after the changed bytes need to be altered. Thus, there is no need to change the C-vector bits associated with the text sequence in byte locations 16-23. Thus, disregarding for the moment the prohibition against compounding with 6 byte instructions, if the instruction at byte 14 were compounded with the instruction beginning at byte 16, the compounding tag bit indicating this would be associated with byte 14. This bit would have been invalidated with the WRITE. The tag for byte 16, which indicates compounding starting for the instruction at byte 16 and following instructions, is not affected by a change to bytes before byte 16. However, the compounding tags of the potential instructions that precede the changed instruction must be considered to accommodate the possibility that they may be in a compound instruction that includes the altered instruction. The maximum possible number of compounding tags to alter depends upon N, the maximum number of instructions that could be compounded, the maximum length L of compoundable instructions, and A the number of bytes covered by a compounding tag. In simple terms, the tags of the N-1 instructions preceding the first modified instruction that could be part of the same compound instruction must be considered for possible alteration. For example, if the largest compound instruction is made up of 2 four-byte instructions, then tags for the 6 bytes preceding the first modified half word must be analyzed. This is shown in FIG. 4 by the arrow labeled "reduced tags". Thus, with a WRITE to the instruction at byte 14, the compounding tag bits back to and including that for byte 8 of the text sequence must be considered for reduction.

In this invention, the "alteration" of compounding tag information can mean "invalidation" of compounding tags for up to the N-1 instructions which precede an altered instruction. In this regard, "invalidation" would mean that all of the compounding tags within this window could be reduced to zero if all of these instructions are compounded. This would not impede instruction execution as all of the instructions in this window would simply be executed sequentially. Alternatively, tag alteration according to the invention may involve decrementation of the compounding tag values for the (N-1) instructions preceding the modified bytes if this group includes instructions that were compounded with the modified instruction. The invention also contemplates neither invalidation nor decrementation of the compounding tag of bytes preceding modified bytes if those tags indicate no compounding with the modified bytes.

Refer now to FIGS. 6A-6J for an understanding of the procedure of the invention. In the procedure, the alteration of compounding tags to reflect a WRITE to a compounded instruction is termed "tag reduction". This definition accounts for the fact that a zero tag is still valid. Further, when the degree of compounding is greater than 2, the value of the tag may be reduced to a value that is not zero. Relatedly, the first three algorithms illustrated in FIG. 6E are referred to as tag reduction algorithms.

The algorithms of FIG. 6A-6J are shown for the general case in which instructions and data are intermixed, with no knowledge of where each instruction starts, and where compounding tags are generated for each half word of text. Certain optimizations can be made for specific cases. For example, if the computer system has no WRITES into lines containing instructions, no tags need ever be reduced. Furthermore, if there are separate instruction and data caches, a data cache has no use for tags. In another example, if instruction boundaries are known, tags for bytes that do not begin instructions need never be checked for reduction. Also, when checking instruction text bytes preceding modified bytes for tag reduction, checking can stop at the first tag bit, beginning an instruction, that is already zero.

In FIGS. 6A-6J, it is assumed that compounding tags are interpreted as follows: the value of a tag is the number of following instructions that are compounded with the current instruction. When an instruction is fetched by a CPU, compounding tags associated with a first byte of an instruction, that is, the beginning or boundary of the instruction, are used; others are ignored. This is not to imply that the algorithms cannot be used for alternative tag representations. Rather, the example is selected for illustration only. The algorithms further assume no compounding across the boundary of a cache line. Transline compounding complicates tag reduction. One potential solution would be to detect a modification close enough to the beginning of a line to cause tags in the previous line to be reduced. The cache can invalidate all or part of a previous line and set an appropriate flag signaling to other caches and the main memory the occurrence of reduction. Alternatively, the cache can request the line in a cache coherency protocol and reduce the appropriate tags, in which case the text is considered to be modified.

In FIG. 6A, conventional cache management operations are shown. As the routine practitioner will realize, FIG. 6A illustrates typical memory management techniques as modified before compounding in the hierarchical memory organization illustrated in FIG. 3. Thus, the virtual address 74 of text being referenced is translated into a real address by a routine address translation look-up 75 which checks the page number of the address against a directory of pages resident in the main memory. If the page is resident in the main memory, there is no page fault and the negative exit is taken from decision 76, the real address 77 is assembled and cache operations are commenced. Assuming that the text whose real address is illustrated by 77 is altered or modified by a WRITE, one of two consistency routines is invoked to update the copy of the modified text in the main memory. Thus, when a WRITE occurs, one of the two branches will be taken out of the cache-type decision 78. The left-hand exit assumes a WRITE-through cache routine wherein the copy of the text in the main memory is updated concurrently with updating the text in the cache. Alternatively, a WRITE-back routine in step 80 permits the text to be modified while it is in the cache and then ensures that the main memory copy is updated by writing the cache line back to the main memory at some later time. The branches out of step 78 rejoin at step 81 where a cache coherency routine is invoked to ensure consistency between all caches and main memory.

Assuming that the page containing the requested text is not in main memory (and, therefore, not in the cache), the positive exit is taken out of the page fault step 76, routine page fault handling is invoked in step 83, the new page fetched from auxiliary storage is subjected to compounding in step 84, and the page and compounding tags are stored in main memory in step 85. After this, the procedure executes as described above.

A WRITE-through cache routine for step 79 of FIG. 6A is illustrated in FIG. 6B. This routine follows the usual WRITE-through policy in which text modification causes the copy of the text in the main memory and a cache to be updated concurrently. In this regard, if instruction execution requires reference to cached text, the first determination is whether the reference is a WRITE. If not, the negative exit is taken from decision 90. If the referenced text is in the cache, the negative exit is taken from step 91, and decision 92 is encountered. If the addressed text is not an instruction, it is provided to the CPU and the routine is exited. If an instruction fetch is required, the instruction is obtained and the instruction and its associated compounding tags are provided to the CPU for execution. If the addressed text is not in the cache, the positive exit is taken from step 91, and replacement and miss handling routines 94 and 95 are invoked to replace a line in the cache with the line containing the addressed instruction. Such routines are known in the art; the examples shown in FIGS. 6F and 6G have been modified to include compounding tags. These steps require that a line and accompanying tags to be replaced in the cache first be returned to the memory and then that the line and accompanying tags containing the addressed text be entered in the cache in the place of the line which was returned to memory. Processing then proceeds through decision 92 as described above.

In FIG. 6B, assuming that the addressed text is being written to or modified, the positive exit is taken from decision 90 and the cache is inspected in step 94 to determine whether the text is in the cache. If not in the cache, a miss occurs, the positive exit is taken and a partial line write routine is invoked in step 95 to modify the text in the main memory.

On the other hand, partial line write routines are illustrated in FIGS. 6H and 6J. In FIG. 6H, the modified text is merged in main memory with the line containing its predecessor. A tag reduction routine generates new tags as needed and the modified text and tags are stored. The compounding tag reduction routine is executed at location 74 in FIG. 3; it zeros the compounding tags for the modified text and processes the tags for the (NL-A) bytes preceding the modified text. The alternative partial line write routine of FIG. 6-J assumes that the source of the transferred line performs the tag reduction. In this case, the tag reduction could be executed at location 73 in FIG. 3 and the new tags generated thereby would be transferred to the main memory together with the modified text.

Assuming now that a WRITE is generated and the written-to text is in the cache, a hit occurs and the negative exit is taken from decision 94. In this case, the addressed text is modified in the cache in step 97 while the compounding tags for the instruction are reduced, according to the invention, by a tag reduction routine in step 98. Then, the copy of the instruction is modified in the main memory by the partial line WRITE routine of step 95.

FIG. 6 illustrates a WRITE-back cache routine for ensuring data consistency between main memory and a cache. Upon a WRITE-hit, only the cache copy of the text is altered. Later, when the line is returned to the main memory, the update occurs. In write back, the management policy may require that every line written to the cache be written back to the memory. Alternatively, upon a WRITE-hit, the affected cache line can be marked as "dirty". Later, only "dirty" lines are written back to the main memory. In FIG. 6C, in the event of a reference miss, the positive exit is taken from decision 100 and the line containing the referenced text is swapped from the memory into the cache for a line in the cache by invocation of the replacement and miss handling routines 94 and 95 for which, see FIGS. 6F and 6G. If the reference is not a WRITE reference, the negative exit is taken from step 101 and processing proceeds as described above for step 92 in FIG. 6B. If the reference is a WRITE, the positive exit is taken from decision 101, the modified text is written into the cache and a tag reduction routine according to the invention is invoked to reduce the tags of the modified instruction in step 103.

In FIG. 6D, a cache coherency routine including provision for tag reduction is illustrated. The cache coherency routine is essentially one which is used to ensure that data in a cache is coherent with data which is transferred into the main memory from an auxiliary storage device, or with data which is updated in another cache. The cache coherency routine of FIG. 6D is applicable to any cache in a computer with multiple processors, multiple I/O, and a real memory with multiple caches. Standard coherency algorithms are available for application as represented in step 110. Such algorithms can be understood, for example, with reference to COMPUTER ARCHITECTURE AND DESIGN, A. J. van de Goor, 1989, at pages 492-507.

Three such algorithms are represented in FIG. 6D by the decisions 111, 112, and 113. These algorithms are shown as decision blocks simply for convenience. The algorithms represented by the decision blocks 112 and 113 assume that coherency problems are avoided in real memory by requiring all WRITES to be made to main memory, In the algorithm of decision 112, a WRITE to the main memory results in transfer of an entire line containing the written text to the affected caches. In this algorithm, it is assumed that the write to the memory results in a tag reduction routine executed at the main memory so that the entire line including the modified text and the accompanying tags are provided to the cache. Alternately, if a partial line is transferred as in decision 113, one of the partial line write routines illustrated in FIGS. 6H and 6J is invoked.

Decision 111 represents an algorithm in which an invalidate signal is relied upon to maintain cache coherency. In this case, for so long as the relevant invalidate signal is not activated, whole or partial lines of text may be received by the cache from the main memory. However, when an invalidate signal relating to specified text in the cache is activated, the standard algorithm results in invalidation of the specified text. In the invention, tag reduction, represented by step 115, is activated to alter the compounding tags for (N-1) instructions which may precede the invalidated text in the affected line.

FIG. 6E illustrates four tag management algorithms according to the invention. Any of these routines can be invoked at the appropriate points in the previously-described procedures. The first algorithm, algorithm 1, reduces tags by zeroing all tags in a modified cache line. Algorithm 4 employs the opposite extreme, which is to recompound bytes of the modified instruction and at least (NL-A) bytes which precede and follow the modified byte. In algorithm 2, the compounding tag bits for the modified line are reduced by first zeroing the tag bits for the modified text. Next, the tag bits for the (NL-A) previous bytes are also zeroed and the algorithm is exited.

In algorithm 3 of FIG. 6E, the compounding tag bits for modified text in the instruction cache will be reduced to zero in step 120. Next, tags for the (NL-A) bytes of the cache line preceding the modified text are examined, tag by tag, to determine whether any further compounding tag reduction is required. In this regard, once the address of the modified text is known, its line and line location can be determined to establish a starting point for inspecting the portion of the line which precedes it. Thus, in step 122, a loop is entered in which compounding tag bits for the A bytes preceding the line location reference are inspected. If the value of the tag for the first half word is equal to zero, the negative exit is taken from decision 123 and step 122 is re-entered by way of the negative exit from decision 124. Returning to decision 123, if the value of the tag for the next A bytes being analyzed is greater than zero, the end of the compound instruction beginning at this byte is located and a determination is made whether any other bytes in this compound instruction have been modified. If not, the negative exit is taken from decision 125 back to decision 124. Otherwise, the tag for the A bytes currently under consideration is reduced in value by 1 and the procedure returns to decision 123. Thus, the positive exit from decision 123 will adjust the size of the compound instruction in the portion of a cache line preceding a modified instruction only if bytes in the compound instruction are modified. If the modified instruction was included in the compound instruction preceding it, the tag for the compound instruction would be reduced by 1, eventually eliminating the modified instruction from the compound instruction with the result that the second positive exit from decision 123 will exit the algorithm via the negative exit from step 125 and the positive exit from step 124.

Algorithm 3 of FIG. 6E can be understood with reference to FIG. 12 which shows a C-vector for text in an instruction cache where 5 instructions, Instr 1-Instr 5, are compounded. Assume further that A=2, L=4, and N=5. Initially, instruction 1, a 2-byte instruction, has a compounding tag equal to 4 indicating that it is compounded with the 4 following instructions, that is instructions 2-5. Instruction 2, a 4-byte instruction has a compounding tag value of 3 for its first 2 bytes, indicating that it is compounded with the following 3 instructions (Instructions 3-5), and a compounding tag value of 0 for its second 2 bytes. Instruction 3, a 4-byte instruction, has compounding tag values of 2 and 0. Instruction 4, a 4-byte instruction, has compounding tag values of 1 and 0, and instruction 5, a 2-byte instruction has a compounding tag value of 0, indicating that it is not compounded with any following instructions. Assuming instruction 3 is modified, algorithm 3 in FIG. 6E operates on instruction 3 and the (NL-A) bytes preceding it, ignoring all bytes which follow it, including instructions 4 and 5. Relatedly, the compounding tag information for instructions 4 and 5 will be unaltered, while the 2 tags for instruction 3 will be reduced to 0 according to step 120 of algorithm 3. Next, algorithm 3 examines compounding tag information for 2 bytes preceding instruction 3, finds a value of 0, takes the negative exit from step 123 and the negative exit from step 124. Now, the compounding tag information for the first 2 bytes of instruction 2 are examined in step 122, the value is determined to be greater than 0 and the positive exit is taken from step 123. After this exit, the value of 3 for the compounding tag indicates that instruction 5 ends the compound instruction beginning at the first 2 bytes of instruction 2. In decision 125, instruction 3, the modified instruction, is in this compound instruction, so the positive exit is taken, the value of 3 is reduced to 2, and decision 123 is re-entered. This loop continues until the value of the compounding tag for the first 2 bytes of instruction 2 is reduced to 0, in which case, instruction 3 will no longer be included in the compound instruction, and decision 124 will be entered via the negative exit from step 125. Since (NL-A) bytes have not been completed, algorithm 3 returns to step 122 by the negative exit from decision 124. Now, the A bytes preceding instruction 2 are examined, which comprise instruction 1 of the compound instruction. The value of this compounding tag is greater than 0, and the loop, including the positive exits from steps 123 and 125, is traversed until the value of the compounding tag for instruction 1 is reduced to 1, which leaves instruction 1 compounded with instruction 2. Now, the remainder of the (NL-A) bytes preceding instruction 3 will continue to be analyzed by algorithm 3. However, none of the compounding tags for these bytes will be altered because any compound instructions in which they are included will not include the modified text of instruction 3. Thus, algorithm 3 will loop through the negative exit of 125 and 124 until all (NL-A) bytes preceding instruction 3 have been analyzed.

FIG. 7 illustrates a compounding tag management unit which is intended to be used in connection with a compound instruction cache configured as illustrated in FIG. 5B and including a text array portion 200 and a compounding tag array 201. The compounding tag management unit of FIG. 7 responds to the modification of an addressed instruction by operations which effectively implement the third algorithm of FIG. 6E. The first algorithm of FIG. 6E is the trivial case of reducing the tag bit values for an entire line to zero. The second algorithm is actually a special case of the third algorithm. The fourth algorithm is recompounding and can be an extension of the third algorithm.

FIG. 7 illustrates compounding tag management for the special case of tag reduction in the instruction cache, which applies for WRITE-BACK and WRITE-THROUGH caches. In a WRITE-THROUGH cache, the unit of FIG. 7 can be applied similarly for the main memory.

Given the address of referenced cache text, and given that the text is being written to (in effect a "WRITE hit"), the text is modified by provision of the address of the line containing the text on signal line 202 to the text array 200. The address includes the byte address information identifying the location of the modified information in the addressed cache line. This byte address information is provided on signal line 203. The new text produced by writing or otherwise modifying the addressed text is provided at 204 to a conventional merge unit 205. The merge unit 205 receives the entire line which is addressed and inserts the new text into the addressed line at the byte address in the line indicated by the information on signal line 203. A new text line results which is returned to the cache text array on signal line 208.

The line address and byte address information which define the cache line and line location of the modified text also identify the related compounding tag information for the text. Thus, provision of the line address on signal line 202 to the compounding tag array will extract an array of compounding tags for the addressed line of text, and the byte address information will indicate which of the addressed array of tags is for the modified text. Assuming that compounding tag information for the addressed line is in the format of the C-vector in FIG. 4, the vector for the addressed line will be brought to a select circuit

212. Similarly, the entire new text line will be brought to a select circuit 210. The text line select circuit 210 uses the location of the modified text in the line to select the (NL-A) bytes in the line which precede the changed text. This is registered at 214. Similarly, the selector 212 selects the compounding tags from the C-vector for the (NL-A) bytes registered at 214. These compounding tags are registered at 216. In response to the text bytes registered at 214 and the tags registered at 216, tag reduction hardware 218 generates new tags representing the reduction of tags according to the third algorithm of FIG. 6E. These new tags are provided on signal line 220 to a merge circuit 221. The merge circuit 221 initially receives the C-vector for the addressed cache line and the byte address information identifying line location of the modified text. This byte address information precisely identifies the location of the compounding tag in the addressed C-vector for the addressed cache line. This tag is replaced by a hard-wired zero provided to the merge circuit on signal line 222. Once the merge circuit 221 knows the location in the C-vector of the compounding tag for the modified text, it inserts the new tags on signal line 220 into the C-vector for the (NL-A) bytes which precede the modified text. The new tag line which results is returned to the tag array 201 to be stored at the line address location.

FIGS. 8 and 9 illustrate tag reduction hardware for a compounding example in which 2 and 4 byte instructions are compounded, the compounding scope embraces up to 3 instructions, and a 2 bit compounding tag is provided for each text half word. In this case, the maximum size of a compounded instruction would be 3.times.4=12 bytes. Since the tag for the first half word of the modified text has already been reduced, 2 of the 12 bytes are accounted for and only the 10 bytes preceding the first half word in the modified text need be inspected. These 10 bytes determine the size of the register 214. Similarly, provision of a 2 bit tag (to encode up to 3-way compounding) means that the tag register 216 must accommodate 5 compounding tags of 2 bits each, or 10 bits in total.

The tag reduction hardware 218 includes a plurality of tag reduction units (TRU) 218a to 218e. Refer now to FIG. 8 and FIG. 9, each of the TRUs 218a to 218e is associated with a respective half word in the register 214. As the TRU 9 in FIG. 9 (which is representative of all of the TRUs in FIG. 8) illustrates there are 4 inputs to the unit, in response to which the unit produces 2 outputs. Unit inputs are instruction length code (ILC) for the instruction in which the unit's half word is included, the compounding tag for that half word, a tag input denoted as tag.sub.1 and a tag input denoted at tag.sub.2. The inputs tag.sub.1 and tag.sub.2 represent tags of the next 2 half words adjacent the half word with which the unit is associated. The TRU 9 produces 2 outputs tag.sub.n and tag'. The output tag' is the reduced tag for the unit half word; the output tag.sub.n is fed backward to the units which represent half words in the set of (NL-A) half words whose compounding tags are being reduced.

Since the tagging representation selected in the example of FIGS. 8 and 9 uses 2 bits per tag to indicate the number of instructions compounded with this instruction, the tag values are zero, 1, and 2. The TRU implementation sets the compounding tags so that no compound instruction extends into the text bytes that were modified. The result is that the compounding tag values for the (NL-A) half words which immediately precede the modified text decrease or go to zero depending upon the next compounding tag value and the number of bytes between a given byte and the modified bytes. The tag value tag.sub.n is used internally and can assume any one of the values -1, 0, 1, or 2. For any instruction that was all or partially modified, this tag is given value -1. The other output, tag', is the new compounding tag value for the half word after reduction according to the third algorithm of FIG. 6E.

Refer now to FIG. 8 for an explanation of the operation of the tag reduction hardware. In FIG. 8, the

TRU 218e is associated with a half word which immediately precedes the first half word of the modified text. Its 2 inputs tag.sub.1 and tag.sub.2 are hardwired to a value of -1, which indicates that the text immediately succeeding this half word have been modified. In the TRU 218d, the tag.sub.2 input is hardwired to a value of -1 to account for modification of the addressed text.

As an example, assume that the modified text is the third scalar instruction of a compounded set of 3 scalar instructions. Assume further that the instruction immediately preceding it is a 2 byte instruction preceded by a 4 byte instruction. Assume that the 4 byte instruction, the first instruction of a compounded set, is itself preceded by a non-compounded 4 byte instruction. In this case, the contents of the register 214 will indicate that the last half word of the register at byte positions 8-9 is occupied by a 2 byte instruction, byte positions 4-7 by a 4 byte instructions and byte positions 0-3 by another 4 byte instruction. The compounding tags for the half words will be as shown in the register 216. Since the last half word is the 2 byte instruction with which the modified instruction is compounded, it will have a single compounding tag of 2 bits whose value will be set to 1 (decimal) to indicate that this instruction is compounded with its one following instruction. The next compounding tag value will be for the second half word of the 4 byte instruction, this half word being at byte positions 6-7 of the register 214. In the compounding scheme, the compounding bits for this half word are set to zero, since the CPU will ignore them. The compounding tag for the half word at byte positions 4-5 of register 214 will be set to the value of 2 (decimal) because this is the first half word of a compounded instruction and because it has been compounded with its 2 following instructions, that is, the instruction in byte positions 8-9 of the register 214 and the instruction including the first byte of modified text. The instruction at byte position 0-3 being non-compounded, the compounding tag values for its 2 half words are respectively zero and zero. The compounding tag values for this example are shown in sequence in the register 216. The tag.sub.1 and tag.sub.2 inputs to the TRU 218e are both "-1". In this case, the outputs for this unit are given in the first line of the table in FIG. 9, that is, tag.sub.n and tag' are both equal to "zero". Now the inputs to TRU 218d, representing the second half word of the 4 byte instruction at positions 4-7 are as follows: tag.sub.2 =-1, tag.sub.1 =0, tag=0, and ILC=4. This unit outputs tag.sub.n and tag' values according to line 3 of the table in FIG. 9. Resultantly, the TRU input values at the unit 218c, which represents the first half word in the 4 byte instruction beginning the compounded 3 instruction set, are as follows: tag.sub.1 and tag.sub.2 =0, tag=2, and ILC=4. This produces an output given by the sixth line of the table in FIG. 9. Since the 4 byte instruction at byte locations 0-3 is non-compounded, its tag value is 0 and the outputs of the TRUs 218a and 218b are given by the third line of FIG. 9.

The reduced tag values for the example just given are illustrated at 225. Now, the value of the compounding tag for the 2 byte instruction at byte locations 8-9 is 0, indicating that this instruction is not compounded with the following instruction, the value for the second half word of the 4 byte instruction at locations 4-7 is 0, and the value of the compounding tag for the first half word of the 4 byte instruction at locations 4-7 is 1 indicating that this instruction is compounded with its 1 following instruction, namely, the 2 byte instruction at locations 8-9.

FIG. 10 illustrates tag reduction hardware for the case where N=2, L=4, and A=2. As the Figure shows, the auxiliary tag tag.sub.n can still assume a value of -1, which indicates that the value of the compounding tag for its associated instruction has been reduced.

FIG. 11 illustrates algorithm 4 of FIG. 6B, that is, recompounding in response to text modification. The elements in FIG. 11 which are numbered identically with corresponding elements in FIG. 7 are the equivalents of those elements. In this regard, when new text is produced by a WRITE, the new text is merged at 205 with the line containing its predecessor from the cache 200. The new text line at 206 is fed back to the cache 200 to replace its predecessor, while the select circuit 210 selects a portion of the line centered on the modified bytes and extending (NL-A) bytes on either side of the modified text. Instructions in the selected text are compounded by the instruction compounding unit 50, which generates new tags for this portion of text. The new tags are inserted in the C-vector for the addressed align at 251, and the new tag line is placed in compounding tag array 201.

In the foregoing discussion of FIGS. 6A through 12, the tag reduction and tag recompounding management solutions are both described with reference to a text stream which includes multi-byte instructions of variable length which are aligned on 2 byte boundaries. Further, the description assumes that a compounding tag is generated for every 2 bytes of text, whether or not the 2 bytes contain an instruction op code. This is in no way meant to limit the application of the tag management policies and mechanisms of this invention. Generally, in a computer system which compounds up to N instructions, tag reduction must consider compounding tag modification for the largest possible compound instruction. Therefore, in general, tag reduction looks to the N-1 possible instructions which precede modified text in memory. Similarly, when tag recompounding is selected as the tag management policy, it is best to consider for recompounding as many instructions as possible on either side of the modified text in order to maximize the opportunity for optimum compounding. In this application, recompounding considers the N-1 instructions preceding and following the modified text. Other implementations may extend this number.

While we have described several preferred embodiments of our invention, it should be understood that modifications and adaptations thereof will occur to persons skilled in the art. Therefore, the protection afforded our invention should only be limited in accordance with the scope of the following claims.

* * * * *

United States Patent	***5,051,940***
Vassiliadis , et al.	**September 24, 1991**

Data dependency collapsing hardware apparatus

Abstract

A multi-function ALU (arithmetic/logic unit) for use in digital data processing facilitates the execution of instructions in parallel, thereby enhancing processor performance. The proposed apparatus reduces the instruction execution latency that results from data dependency hazards in a pipelined machine. This latency reduction is accomplished by collapsing the interlocks due to these hazards. The proposed apparatus achieves performance improvement while maintaining compatibility with previous implementations designed using an identical architecture.

Inventors:	**Vassiliadis; Stamatis** (Vestal, NY); **Phillips; James E.** (Binghamton, NY); **Blaner; Bartholomew** (Newark Valley, NY)
Assignee:	**International Business Machines Corporation** (Armonk, NY)
Appl. No.:	**504910**
Filed:	**April 4, 1990**

Current U.S. Class:	**708/524**
Intern'l Class:	G06F 007/38
Field of Search:	364/736,768,787,716,200 MS File,900 MS File

References Cited [Referenced By]

U.S. Patent Documents

4754412	Jun., 1988	Deering	364/736.
4766416	Aug., 1988	Noujaim	364/736.

4775952	Oct., 1988	Danielsson et al.	364/736.
4819155	Apr., 1989	Wulf et al.	364/200.
4852040	Jul., 1989	Oota	364/736.

Other References

Hwang, Kai & Faye A. Briggs, "Computer Architecture and Parallel Processing", 1984, pp. 325-328.
Wulf, Wm. A. "The WM Computer Architecture", Computer Architecture News, vol. 16, No. 1, Mar. 1988, pp. 70-84.
Hwang, Kai, Computer Arithmetic, 1979, pp. 88-100.
Jouppi, Norman P. & David W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", Third Intn'l. Conference on Architectural Support for Prog. Languages & Operating Sys., 1989, pp. 272-282.

Primary Examiner: Harkcom; Gary V.
Assistant Examiner: Nguyen; Long T.
Attorney, Agent or Firm: Baker, Maxham, Jester & Meador

Claims

What is claimed is:

1. In a computer architected for serial execution of a sequence of single scalar instructions in a succession of execution cycles, an apparatus for supporting parallel execution of a plurality of scalar instructions in a single execution cycle, the apparatus comprising:

an instruction means for receiving a plurality of scalar instructions, a first of the scalar instructions producing a result used as an operand by a second of the scalar instructions;

an operand means for substantially simultaneously providing a plurality of operands, at least two of said operands being used by the first and second scalar instructions;

a control means connected to the instruction means for generating control signals to indicate operations which execute the plurality of scalar instructions; and

an execution means connected to the operand means and to the control means and responsive to the control signals and to a plurality of operands including the two operands for producing, in a single execution cycle, a single result corresponding to the performance of said operations on said plurality of operands.

2. The apparatus of claim 1, wherein the execution means includes an adder which produces a single adder result in response to three operands.

3. The apparatus of claim 2, wherein the adder includes a carry save adder which produces two outputs in response to the three operands and a carry look ahead adder, connected to the carry save adder, which produces one output in response to the two outputs of the carry save adder.

4. The apparatus of claim 2, wherein the execution means further includes logical means connected to the operand means and to the adder for performing a logic function on the operands to produce a logic result, the adder producing said single adder result in response to the logic result and one of the operands.

5. The apparatus of claim 2, wherein the execution means further includes logic means connected to the operand means and to the adder for performing a logic function on a first and second operand to produce a logic result, the execution means producing the single result in response to the logic result and the single adder result.

6. The apparatus of claim 1 wherein the first scalar instruction is a logical instruction and the second scalar instruction is an arithmetic instruction and the execution means includes logical means for combining first and second operands to produce a logical result required by said logical instruction and arithmetic means for combining the logical result with a third operand to produce said single result, said single result being required by the arithmetic instruction.

7. The apparatus of claim 1 wherein the first scalar instruction is an arithmetic instruction and the second scalar instruction is a logical instruction and the execution means includes arithmetic means for combining first and second operands to produce an arithmetic result required by said arithmetic instruction and logical means for combining the arithmetic result with a third operand to produce said single result, said single result being required by the logical instruction.

8. The apparatus of claim 1, wherein the first scalar instruction is an arithmetic instruction and the second scalar instruction is an arithmetic instruction and the execution means includes arithmetic means for combining the three operands to produce a single arithmetic result, said single arithmetic result being provided as said single result.

9. The apparatus of claim 1 wherein the first scalar instruction is a logical instruction and the second scalar instruction is a logical instruction and the execution means includes logical means for combining first and second operands to produce a first logical result, said first logical result required by said first logical instruction, and second logical means for combining the first logical result with a third operand to produce a second logical result, said second logical result being required by the second scalar instruction and said second logical result being provided as said single result.

10. A multifunction ALU (arithmetic logic unit) for combining three operands to produce a single result in response to a pair of instructions, including:

a first set of logical elements for logically combining two operands to produce a first logical result;

an adder for arithmetically combining three operands to produce a single arithmetic result;

a circuit for inputting to the adder either all of said operands, two of said operands and a zero, one of said operands, a zero, and said first logical result, or two zeros and said first logical result;

a second set of logical elements for logically combining one of said operands with said single arithmetic result to produce a second logical result; and

a circuit for providing as an output either said arithmetic result or said second logical result.

11. The multifunction ALU of claim 10, wherein said adder includes:

a carry-save adder for producing two outputs in response to three operands; and

a carry look ahead adder connected to said carry save adder for producing one output in response to said two outputs.

12. In a computer architected for serial execution of a sequence of scalar instructions in a succession of execution periods, an interlock-collapsing apparatus for supporting simultaneous parallel execution of a plurality of scalar instructions, the apparatus comprising:

an instruction register means for receiving a plurality of scalar instructions for simultaneous execution, a first instruction of the plurality of scalar instructions producing a result used as an operand by a second instruction of the plurality of scalar instructions;

an operand means for substantially simultaneously providing a plurality of operands used in executing the plurality of scalar instructions;

a control means connected to the instruction register means for generating control signals to indicate operations which execute the plurality of scalar instructions; and

an interlock-collapsing execution means connected to the operand means and to the control means an responsive to the control signals and to the plurality of operands for producing a single result corresponding to the simultaneous execution of first and second instructions in a single execution period.

13. The apparatus of claim 12, wherein the interlock-collapsing execution means includes an adder which produces a single adder result in response to three operands.

14. The apparatus of claim 13, wherein the adder includes a carry save adder which produces two outputs in response to the three operands and a carry lookahead adder connected to the carry save adder which produces one output in response to the two outputs of the carry save adder.

15. The apparatus of claim 13, wherein the interlock-collapsing execution means further includes logic means connected to the operand means and to the adder for performing a logic function on the operands to produce a logic result, the adder producing the single adder result in response to the logic result and one of the operands.

16. The apparatus of claim 13, wherein the interlock-collapsing execution means further includes logic means connected to the operand means and to the adder for performing a logic function on a first operand and a second operand to produce a logic result, the interlock-collapsing execution means producing the single result in response to the logic result and the single adder result.

17. The apparatus of claim 12, wherein the first instruction is a logical instruction and the second instruction is an arithmetic instruction and the interlock-collapsing execution means includes logical means for combining first and second operands to produce a logic result required by the logical instruction and arithmetic means for combining the logic result with a third operand to produce the single result, the single result representing execution of the arithmetic instruction.

18. The apparatus of claim 12, wherein the first instruction is an arithmetic instruction and the second instruction is a logic instruction and the interlock-collapsing execution means includes arithmetic means for combining first and second operands to produce an arithmetic result required by said arithmetic instruction and logic means for combining the arithmetic result with a third operand to produce the single result, the single result representing execution of the logical instruction.

19. The apparatus of claim 12, wherein the first instruction is an arithmetic instruction and the second instruction is an arithmetic instruction and the interlock-collapsing execution means includes arithmetic means for combining three operands to produce a single arithmetic result, the three operands including two operands used in the execution of the first and second instructions.

20. The apparatus of claim 12, wherein the first instruction is a first logic instruction and the second instruction is a second logic instruction and the interlock-collapsing execution means includes logic means for combining first and second operands to produce a first logic result, the first logic result being required by the first logic instruction, and second logic means for combining the first logic result with a third operand to produce a second logic result, the second logic result representing execution of the second logic instruction and the second logic result being provided as the single result.

21. In a computer architected for serial execution of a sequence of scalar instructions in a succession of execution cycles, an execution apparatus for, in a single execution cycle, producing a result representing simultaneous execution of a first scalar instruction and a second scalar instruction in which the second scalar instruction requires a result produced by execution of the first scalar instruction, the execution apparatus comprising:

an instruction register means for receiving the first and second scalar instructions;

an operand means for substantially simultaneously providing a plurality of operands, at least two of the plurality of operands being used in executing the first and second scalar instructions;

a control means connected to the instruction register means for generating control signals which indicate execution of the first scalar instruction and the second scalar instruction;

a first execution means connected to the operand means and to the control means and responsive to the control signals and to the two operands for producing, in an execution cycle, a result corresponding to the execution of the first instruction; and

a second execution means connected to the operand means and to the control means and responsive to the control signals and to a plurality of operands including the two operands for producing, in said execution cycle, a single result corresponding to the execution of the first and second instructions.

22. The apparatus of claim 21, wherein the first execution means includes an adder which produces a single adder result in response to two operands.

23. The apparatus of claim 21, wherein the second execution means includes an adder which produces a single adder result in response to three operands.

24. The apparatus of claim 23, wherein the adder includes a carry save adder which produces two outputs in response to the three operands and a carry lookahead adder connected to the carry save adder, which produces one output in response to the two outputs of the carry save adder.

Description

BACKGROUND OF THE INVENTION

This invention relates to the execution of scalar instructions in a scalar machine. More particularly, the invention concerns the parallel execution of scalar instructions when one of the instructions uses as an operand a result produced by a concurrently-executed instruction.

Pipelining is a standard technique used by computer designers to improve the performance of computer systems. In pipelining an instruction is partitioned into several steps or stages for which unique hardware is allocated to implement the function assigned to that stage. The rate of instruction flow through the pipeline depends on the rate at which new instructions enter the pipe, rather than the pipeline's length. In an idealized pipeline structure where a maximum of one instruction is fed into the pipeline per cycle, the pipeline throughput, a measure of the number of instructions executed per unit time, is dependent only on the cycle time. If the cycle time of an n-stage pipeline implementation is assumed to be m/n, where m is the cycle time of the corresponding implementation not utilizing pipelining techniques, then the maximum potential improvement offered by pipelining is n.

Although the foregoing indicates that pipelining offers the potential of an n-times improvement in computer system performance, several practical limitations cause the actual performance gain to be less than that for the ideal case. These limitations result from the existence of pipeline hazards. A hazard in a pipeline is defined to be any aspect of the pipeline structure that prevents instructions from passing through the structure at the maximum rate. Pipeline hazards can be caused by data dependencies, structural (hardware resource) conflicts, control dependencies and other factors.

Data dependency hazards are often called write-read hazards or write-read interlocks because the first instruction must write its result before the second instruction can read and subsequently use the result. To allow this write before the read, execution of the read must be blocked until the write has occurred. This blockage introduces a cycle of inactivity, often termed a "bubble" or "stall", into the execution of the blocked instruction. The bubble adds one cycle to the overall execution time of the stalled instruction and thus decreases the throughput of the pipeline. If implemented in hardware, the detection and resolution of structural and data dependency hazards may not only result in performance losses due to the under-utilization of hardware but may also become the critical path of the machine. This hardware would then constrain the achievable cycle time of the machine. Hazards, therefore, can adversely affect two factors which contribute to the throughput of the pipeline: the number of instructions executed per cycle; and the cycle time of the machine.

The existence of hazards indicates that the scheduling or ordering of instructions as they enter a pipeline structure is of great importance in attempting to achieve effective use of the pipeline hardware. Effective use of the hardware, in turn, translates into performance gains. In essence, pipeline scheduling is an attempt to utilize the pipeline to its maximum potential by attempting to avoid hazards. Scheduling can be achieved statically, dynamically or with a combination of both techniques Static scheduling is achieved by reordering the instruction sequence before execution to an equivalent instruction stream that will more fully utilize the hardware than the former. An example of static scheduling is provided in Table I and Table II, in which the interlock between the two Load instructions has been avoided.

TABLE I

```
X1                  ;any instruction
X2                  ;any instruction
ADD    R4,R2        ;R4 = R4 + R2
LOAD   R1,(Y)       ;load R1 from memory location Y
LOAD   R1,(X[R1])
                    ;load R1 from memory location X
                    function of R
ADD    R3,R1        ;R3 = R3 + R1
LCMP   R1,R4        ;load the 2's complement of (R4) to R1
SUB    R1,R2        ;R1 = R1 - R2
COMP   R1,R3        ;compare R1 with R3
X3                  ;any compoundable instruction
X4                  ;any compoundable instruction
```

TABLE II

```
X1                  ;any instruction
X2                  ;any instruction
LOAD   R1,(Y)       ;load R1 from memory location Y
ADD    R4,R2        ;R4 = R4 + R2
LOAD   R1,(X[R1])
```

```
                     ;load R1 from memory location X
                     function of R
ADD    R3,R1         ;R3 = R3 + R1
LCMP   R1,R4         ;load the 2's complement of (R4) to R1
SUB    R1,R2         ;R1 = R1 - R2
COMP   Rl,R3         ;compare R1 with R3
X3                   ;any compoundable instruction
X4                   ;any compoundable instruction
```

While scheduling techniques may relieve some hazards resulting in performance improvements, not all hazards can be relieved. For data dependencies that cannot be relieved by scheduling, solutions have been proposed. These proposals execute multiple operations in parallel. According to one proposal, an instruction stream is analyzed based on hardware utilization and grouped into a compound instruction for issue as a single unit. This approach differs from a "superscalar machine" in which a number of instructions are grouped strictly on a first-in-first-out basis for simultaneous issue. Assuming the hardware is designed to support the simultaneous issue of two instructions, a compound instruction machine would pair the instruction sequence of Table II as follows: (-X1) (X2 LOAD) (ADD LOAD) (ADD LCMP) (SUB COMP) (X3,X4), thereby avoiding the data dependency between the second LOAD instruction and the second ADD instruction. A comparable superscalar machine, however, would issue the following instruction pairs: (X1,X2) (LOAD,ADD) (LOAD,ADD) (LCMP SUB) (COMP X3) (X4-) incurring the penalty of the LOAD-ADD data dependency.

A second solution for the relief of data dependency interlocks has been proposed in Computer Architecture News, March, 1988, by the article entitled "The WM Computer Architecture," by W. A. Wulf. The WM Computer Architecture proposes:

1. architecting an instruction set that imbeds more than one operation into a single instruction;

2. allowing register interlocks within an architected instruction; and

3. concatenating two ALU's as shown in FIG. 1 to collapse interlocks within a single instruction.

Obviously, in Wulf's proposal, new instructions must be architected for all instruction sequence pairs whose interlocks are to be collapsed. This results in either a prohibitive number of opcodes being defined for the new instruction set, or a limit, bounded by the number mf opcodes available, being placed upon the number of operation sequences whose interlocks can be collapsed. In addition, this scheme may not be object code compatible with earlier implementations of an architecture. Other drawbacks for this scheme include the requirement of two ALUs whose concatenation can result in the execution of a multiple operation instruction requiring close to twice the execution time of a single instruction. Such an increase in execution time would reflect into an increase in the cycle time of the machine and unnecessarily penalize all instruction executions.

In the case where an existing machine has been architected to sequentially issue and execute a given set of instructions, it would be beneficial to employ parallelism in instruction issuing and execution. Parallel issue and execution would increase the throughput of the machine. Further, the benefits of such parallelism should be maximized by minimization of instruction execution latency resulting from data dependency hazards in the instruction pipeline. Thus, the adaptation to parallelism should provide for the reduction of such latency by collapsing interlocks due to these hazards. However, these benefits should be enjoyed without having to pay the costs resulting from architectural changes to the existing machine, creating a new instruction set to provide all possible instruction pairs and their combinations possessing interlocks, and adding a great deal of hardware. Further, the adaptation should present a modest or no impact on the cycle time of the machine.

SUMMARY OF THE INVENTION

The invention achieves these objectives in providing a computer architected for serial execution of a sequence of scalar operations with an apparatus for simultaneously executing a plurality of scalar instructions in a single machine cycle. The apparatus is one which collapses data dependency between simultaneously-executed instructions, which means that a pair of instructions can be executed even when one of the pair requires as an operand the result produced by execution of the other of the pair of instructions.

In this invention, the apparatus for collapsing data dependency while simultaneously executing a plurality of scalar instructions includes a provision for receiving a plurality of scalar instructions to be concurrently executed and information as to an order of execution of those instructions, a second of the scalar instructions using as an operand the result produced by execution of a first of the scalar instructions. The apparatus further has provision for receiving three operands which are used by the first and second scalar instructions and has a control component connected to the provision for receiving the instructions which generates control signals that indicate operations which execute the plurality of scalar instructions and which indicate the order of their execution. A multi-function ALU is connected to the operands and to the control provisions and responds to the control signals and the operands by producing, in parallel with execution of the first instruction, a single result corresponding to execution of the second instruction.

Viewed from another aspect, the invention is an apparatus which supports simultaneous execution of a plurality of scalar instructions where a result produced by a first of the simultaneously executing instructions is used as an operand in a second of the simultaneously executing instructions. The apparatus executes the second instruction in parallel with execution of the first instruction by provision of a data dependency-collapsing ALU which has provision for receiving three operands which are used by the first and second instruction to provide the result of the second instruction concurrently with the result of the first instruction.

It is therefore a primary object of this invention to provide an apparatus which facilitates the execution of instructions in parallel to increase existing computer performance.

A significant advantage of the apparatus is the reduction of instruction execution latency that results from data dependency hazards existing in the executed instructions.

An objective in this apparatus is to collapse the interlocks due to data dependency hazards existing between instructions which are executed in parallel.

These objectives and advantages are achieved, with a concomitant improvement in performance and instruction execution by an apparatus which is compatible with the scalar computer designed for sequential execution of the instructions.

The achievement of these and other objectives and advantages will be appreciated when the following detailed description is read with reference to the below-described drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a prior art architecture for execution of an instruction which pairs operations.

FIG. 2 is a set of timing sequences which illustrate pipelined execution of scalar instructions.

FIG. 3 is an illustration of an adder which accepts up to three operands and produces a single result.

FIGS. 4A and 4B illustrate categorization of instructions executed by an existing scalar machine.

FIG. 5 illustrates functions produced by interlocking cases where logical and add-type instructions in category 1 of FIG. 4A are combined.

FIGS. 6A and 6B specify the operations required to be performed on operands by an ALU according to the invention to support instructions contained in compoundable categories in FIGS. 4A and 4B.

FIGS. 7A and 7B summarize the routing of operands to an ALU defined in FIGS. 6A and 6B.

FIG. 8 is a block diagram showing how the invention is used to effect parallel execution of two interlocking instructions.

FIG. 9 is a multi-function ALU defined by FIGS. 6A, 6B, 7A, and 7B.

FIG. 10 illustrates functions requiring implementation to collapse interlocks inherent in hazards encountered in address generation.

FIG. 11 is a logic diagram illustrating a multi-function ALU according to FIG. 10.

FIG. 12 lays out the functions supported by an ALU to collapse interlocks in compounded branching instructions.

FIG. 13 is a logic diagram illustrating an ALU according to FIG. 12.

FIG. 14 illustrates an adder configuration required to collapse interlocks for instructions involving nine operands.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In the discussion which follows, the term "machine cycle" refers to the pipeline steps necessary to execute an instruction. A machine cycle includes individual intervals which correspond to pipeline stages. A "scalar instruction" is an instruction which is executed using scalar operands. Scalar operands are operands representing single-valued quantities. The term "compounding" refers to the grouping of instructions contained in a sequence of instructions, the grouping being for the purpose of concurrent or parallel execution of the grouped instructions. At minimum, compounding is represented by "pairing" of two instructions for simultaneous execution. In the invention being described, compounded instructions are unaltered from the forms they have when presented for scalar execution. As explained below, compounded instructions are accompanied by "tags", that is, bits appended to the grouped instructions which denote the grouping of the instructions for parallel execution. Thus, the bits indicate the beginning and end of a compound instruction.

In the sections to follow an improved hardware solution to relieve execution unit interlocks that cannot be relieved using prior art techniques will be described. The goal is to minimize the hardware required to relieve these interlocks and to incur only a modest or no penalty to the cycle time from the added hardware. No architectural changes are required to implement this solution; therefore, object code compatibility is maintained for an existing architecture.

The presumed existing architecture is exemplified by a sequential scalar machine such as the System/370 available from the International Business Machines Corporation, the assignee of this application. In this regard, such a system can include the System/370, the System/370 extended architecture (370-XA), and the System/370 enterprise systems architecture (370-ESA). Reference is given here to the Principles of Operation of the IBM System/370, publication number GA22-7000-10, 1987, and to the Principles of Operation, IBM Enterprise Systems Architecture/370, publication number SA22-7200-0, 1988.

The instruction set for these existing System/370 scalar architectures is well known. These instructions are scalar instructions in that they are executed by operations performed on scalar operands. References given hereinbelow to particular instructions in the set of instructions executed by the above-described machines are presented in the usual assembly-level form.

Assume the following sequence of instructions is to be executed by a superscalar machine capable of executing four instructions per cycle:

TABLE III

(1) LOAD	R1 , X	load the content of X to R1
(2) ADD	R1 , R2	add R1 to R2 and put the result in R1
(3) SUB	R1 , R3	subtract R3 from R1 and put the result in
(4) STORE	R1 , Y	store the result in memory location Y

Despite the capability of multiple instruction execution per cycle, the superscalar machine will execute the above sequence serially because of instruction interlocks. It has been suggested based on analysis of program traces, that interlocks occur approximately one third of the time. Thus much of the superscalar machine's resources will be wasted, causing the superscalar's performance to degrade. The superscalar machine performance of interlocked scalar instructions is illustrated by the timing sequence indicated by reference numeral 8 in FIG. 2. In this figure, the pipeline structure for the instructions of

Table III is assumed to be as follows:

(1) LOAD:ID AG CA PA

(2) and (3) ADD and SUBTRACT:ID EX PA

where ID is instruction decode and register access, AG is operand address generation, CA represents cache access, EX represents execute, and PA (put away) represents writing the result into a register. To simplify exposition all examples provided in this description assume, unless explicitly stated, that bypassing is not implemented. In the superscalar machine, the execution of the instruction stream is serialized due to instruction interlocks reducing the performance of the superscalar to that of a scalar machine.

In FIG. 2, instructions (2) and (3) require no address generation (AG). However, this stage of the pipeline must be accounted for. Hence the unlabeled intervals 7 and 9. This convention holds also for the other three sequences in FIG. 2.

The above example demonstrates that instruction interlocks can constrain the parallelism that is available at the instruction level for exploitation by a superscalar machine. Performance can be gained with pipelining and bypassing of the results of one interlocked instruction to the other; nevertheless, the execution of interlocked instructions must be serialized.

COMPOUND INSTRUCTIONS

If loss of execution cycles due to interlocks is to be avoided, the interlocked instructions must be executed in "parallel" and viewed as a unique instruction. This leads to the concept of a compounded interlocked instruction, a set of scalar instructions that are to be treated as a single unique instruction despite the occurrence of interlocks. A desirable characteristic of the hardware executing a compounded instruction is that its execution requires no more cycles than required by one of the compounded instructions. As a consequence of instruction compounding and its desired characteristics, a compound instruction set machine must view scalar instructions by hardware utilization rather than opcode description.

EXECUTION OF INTERLOCKED INSTRUCTIONS

The concepts of compounded interlocked instructions can be clarified using the ADD and SUB instructions in Table III. These two instructions can be viewed as a unique instruction type because they utilize the same hardware. Consequently they are combined and executed as one instruction. To exploit parallelism their execution requires the execution of:

R1=R1+R2-R3

in one cycle rather than the execution of the sequence:

R1=R1+R2

R1=R1-R3

which requires more than one cycle to execute. The interlock can be eliminated because the add and subtract utilize identical hardware. Moreover, by employing an ALU which utilizes a carry save adder, CSA, and a carry look-ahead adder, CLA, as shown in FIG. 3, the combined instruction R1+R2-R3 can be executed in one cycle provided that the ALU has been designed to execute a three-to-one addition/subtraction function.

As should be evident, the combined form (R1+R2-R3) corresponds to rewriting the two operands of the second instruction in terms of three operands, thereby implying the requirement of an adder which can execute the second instruction in response to three operands.

In FIG. 3, the carry save adder (CSA) is indicated by reference numeral 10. The CSA 10 is conventional in all respects and receives three operands to produce two results, a sum (S) on output 12 and a carry (C) on output 14. For the example given above, the inputs to the CSA 10 are the operands contained in the three registers R1, R2, and R3 (complemented). The outputs of the CSA 10 are staged at 16 and 17 for the provision of a leading "1" or "0" (a "hot" 1 or 0) on the carry value by way of input 20. The value on input 20 is set conventionally according to the function to be performed by the CSA 10.

The sum and carry (with appended 1 or 0) outputs of the CSA 10 are provided as the two inputs to the carry look-ahead adder (CLA) 22. The CLA 22 also conventionally receives a "hot" 1 or 0 on input 24 according to the desired operation and produces a result on 26. In FIG. 3 the result produced by CLA 22 is the combination of the contents of the three registers R1, R2 and R3 (complemented).

Carry save and carry look-ahead adders are conventional components whose structures and functions are well known. Hwang in his COMPUTER ARITHMETIC: Principles, Architecture and Design, 1979, describes carry look-ahead adders at length on pages 88-93 and carry save adders on pages 97-100.

Despite the three-to-one addition requiring an extra stage, the CSA in FIG. 3, in the critical path of the ALU, such a stage should not compromise the cycle time of the machine since the length of other paths usually exceed that of the ALU. These critical paths are usually found in paths possessing an array access, address generation which requires a three-to-one ALU and a chip crossing; therefore, the extra stage delay is not prohibitive and the proposed scheme will result in performance improvements when compared to scalar or superscalar machines. The performance improvement is shown in FIG. 2 by the set of pipelined plots indicated by reference numeral 26. These plots show the execution of the instruction sequence under consideration by a compound instruction set machine which includes an ALU with an adder configured as illustrated in FIG. 3.

As shown by the timing sequences 8 and 26 of FIG. 2, execution of the sequence by the compound instruction set machine requires eight cycles or two cycles per instruction, CPI, as compared to the 11 cycles or 2.75 CPI achievable by the scalar and superscalar machines. If bypassing is assumed to be supported in all of the machines, plot sets 28 and 30 of FIG. 2 describe the execution achievable with the scalar/superscalar machines and the compound instruction set machine respectively. From these sets, the superscalar machine requires eight cycles or two CPI to execute the example code while the compound instruction set machine requires six cycles or 1.5 CPI. The advantage of the compounded machine over both superscalar and scalar machines should be noted along with the lack of advantage of the superscalar machine over the scalar for the assumed instruction sequence.

Compounding of instructions with their simultaneous execution by hardware is not limited to arithmetic operations. For example, most logical operations can be compounded in a manner analogous to that for arithmetic operations. Also, most logical operations can be compounded with arithmetic operations. Compounding of some instructions, however, could result in stretching the cycle time because unacceptable delays must be incurred to perform the compounded function. For example, an ADD-SHIFT compound instruction may stretch the cycle time prohibitively which would compromise the overall performance gain. The frequency of interlocks between these instructions, however, is low given the low frequency of occurrence of shift instructions; therefore, they can be executed serially without substantial performance loss.

As described previously, data hazard interlocks occur when a register or memory location is written and then read by a subsequent instruction. The proposed apparatus of this invention collapses these interlocks by deriving new functions that arise from combining the execution of instructions whose operands present data hazards while retaining the execution of functions inherent in the instruction set. Though some instruction and operand combinations would not be expected to occur in a functioning program, all combinations are considered. In general all the functions derived from the above analysis as well as the functions arising from a scalar implementation of the instruction set would be implemented. In practice, however, certain functions arise whose implementation is not well suited to the scheme proposed for this apparatus. The following presentation elucidates these concepts by discussing how new functions arise from combining the execution of two instructions. Examples of instruction sequences that are well handled according to the invention are presented along with some sequences that are not handled well. A logical diagram of the preferred embodiment of the invention is shown.

The apparatus of the invention is proposed to facilitate the parallel issue and execution of instructions. An example of issuing instructions in parallel is found in the superscalar machine of the prior art; the invention of this application facilitates the parallel execution of issued instructions which include interlocks. The use of the data dependency collapsing hardware of this invention, however, is not limited to any particular issue and execution architecture but has general applicability to schemes that issue multiple instructions per cycle.

To provide a hardware platform for the present discussion a System/370 instruction level architecture is assumed in which up to two instructions can be issued per cycle. The use of these assumptions, however, neither constrains these concepts to a System/370 architecture nor to two-way parallelism. The discussion is broken into sections cover ALU operations, memory address generation, and branch determination.

In general, the System/370 instruction set can be broken into categories of instructions that may be executed in parallel. Instructions within these categories may be combined or compounded to form a compound instruction. The below-described apparatus of the invention supports the execution of compounded instructions in parallel and ensures that interlocks existing between members of a compound instruction will be accommodated while the instructions are simultaneously executed. For example, the System/370 architecture can be partitioned into the categories illustrated in FIGS. 4A and 4B.

Rationale for this categorization was based on the functional requirements of the System/370 instructions and their hardware utilization. The rest of the System/370 instructions are not considered to be compounded for execution in this discussion. This does not preclude them from being compounded on a future compound instruction execution engine and possibly use the conclusions of interlock "avoidance" as presented by the present paper.

Consider the instructions contained in category 1 compounded with instructions from that same category as exemplified in the following instruction sequence:

AR R1,R2

SR R3,R4

This sequence, which is free of data hazard interlocks, produces the results:

R1=R1+R2

R3=R3+R4

which comprise two independent instructions specified by the 370 instruction level architecture. Executing such a sequence would require two independent and parallel two-to-one ALU's designed to the instruction level architecture. These results can be generalized to all instructions sequence pairs that are free of data hazard interlocks in which both instructions specify an ALU operation. Two ALU's are sufficient to execute instructions issued in pairs since each instruction specifies at most one ALU operation.

Many instruction sequences, however, are not free of data hazard interlocks. These data hazard interlocks lead to pipeline bubbles which degrade the performance in a typical pipeline design. A solution for increasing processor performance is to eliminate these bubbles from the pipeline by provision of a single ALU that can accommodate data hazard interlocks. To eliminate these interlocks, the ALU must execute new functions arising from instruction pairing and operand conflicts. The functions that arise depend on the ALU operations specified, the sequence of these operations, and operand "conflicts" between the operations (the meaning of the term operand conflicts will become apparent in the following discussion). All instruction sequences that can be produced by pairing instructions that are contained within the compoundable list given earlier in this section and will specify an ALU operation must be analyzed for all possible operand conflicts.

INTERLOCK-COLLAPSING ALU

The general framework for collapsing interlocks according to the invention has been presented above. The following presents a more concrete example of the analyses to be performed in determining the requirements of an interlock collapsing ALU. Assume the existence of a three-to-one adder as described above in reference to FIG. 3. Let OP1 and OP2 represent respectively the first and second of two operations to be executed. For instance, for the following sequence of instructions,

NR R1,R2

AR R3,R4

OP1 corresponds to the operation NR while OP2 corresponds to the operation AR (see below for a description of these operations). Let AI0, AI1, and AI2 represent the inputs corresponding to (R1), (R2), and (R3) respectively of the three-to-one adder in FIG. 3. Consider the analysis of compounding the set of instructions (NR, OR, XR, AR, ALR, SLR, SR), a subset of category 1 as defined in FIGS. 4A and 4B. The operations of this set of instructions are specified by:

NR	Bitwise Logical AND represented by .LAMBDA.
OR	Bitwise Logical OR represented by V
XR	Bitwise Exclusive OR represented by .sym.
AR	32 bit signed addition represented by +
ALR	32 bit unsigned addition represented by +
SR	32 bit signed subtraction represented by -
SLR	32 bit unsigned subtraction represented by -

This instruction set can be divided into two sets for further consideration. The first set would include the logical instructions NR, OR and XR, and the second set would include the arithmetic instructions, AR, ALR, SR, and SLR. The grouping of the arithmetics can be justified as follows. The AR and ALR can both be viewed as an implicit 33 bit 2's complement addition by using sign extension for AR and 0 extension for ALR and providing a hot `0` to the adder. Though the setting of condition code and overflow are unique for each instruction, the operation performed by the adder, a binary addition, is common to both instructions. Similarly, SR and SLR can be viewed as an implicit 33 bit 2's complement addition by using sign extension for SR and 0 extension for SLR, inverting the subtrahend, and providing a hot `1` to the adder. The inversion of the subtrahend is considered to be external to the adder. Because the four arithmetic operations essentially perform the same operation, a binary add, they will be referred to as ADD-type instructions while the logical operations will be referred to as LOGICAL-type instructions.

As a result of the reduction of the above instruction set to two operations, the following sequences of operations must be considered to analyze the compounding of this instruction set:

LOGICAL followed by ADD

ADD followed by LOGICAL

LOGICAL followed by LOGICAL

ADD followed by ADD.

For each of these sequences, all combinations of registers must be considered. The combinations are all four register specifications are different plus the number of ways out of four possible register specifications that: 1) two are the same; 2) three are the same; and 3) four are the same. The number of combinations, therefore, can be expressed as: ##EQU1## where .sub.n C.sub.r represents n combined r at a time. But,

.sub.n C.sub.r =n!/((n-r)!r!)

from which formulas the number of combinations can be found to be 12. These 12 register combinations are:

1.R1.noteq.R2.noteq.R3.noteq.R4

2.R1=R2.noteq.R3.noteq.R4

3.R2=R3.noteq.R1.noteq.R4

4.R2=R4.noteq.R1.noteq.R3

5.R3=R4.noteq.R1.noteq.R2

6.R2=R3=R4.noteq.R1

7.R1=R3.noteq.R2.noteq.R4

8.R1=R4.noteq.R2.noteq.R3

9.R1=R2=R3.noteq.R4

10.R1=R2=R4.noteq.R3

11.R1=R3=R4.noteq.R2

12.R1=R2=R3=R4

Of these combinations, only seven through twelve give rise to data dependency interlocks. The functions produced by the above interlocking cases for the LOGICAL-ADD sequences listed earlier are given in FIG. 5. In this Figure, the LOGICAL-type operations are designated by an .phi. and the ADD-type operations are denoted by .zeta..

While FIG. 5 specifies the operations that must be performed on the operands of ADD-type and LOGICAL-type instructions to collapse the interlocks, FIGS. 6A and 6B specify the ALU operations required to be performed on the ALU inputs AI0, AI1, and AI2 to support all 370 instructions that are contained in the compoundable categories of FIGS. 4A and 4B. In FIGS. 6A and 6B a unary - indicates 2's complement and /x/ indicates the absolute value of x. This Figure was derived using an analysis identical to that given above; however, all possible category compoundings were considered. For the operations of FIG. 5 to be executed by the ALU, the execution unit controls must route the desired register contents to the appropriate inputs of the ALU. FIGS. 7A and 7B summarize the routing of the operands that needs to occur for the ALU defined as in FIGS. 6A and 6B to perform the operations of FIG. 5. Along with these routings, the LOGICAL and ADD-type instructions have been given to facilitate the mapping of these results to FIGS. 6A and 6B. Routing for some ADD-ADD compoundings were not included since these operations require a four input ALU (see "Idiosyncrasies") and are so noted.

While the description thus far has focused the consideration of compound instruction analysis on four specifically-enumerated registers, R1, R2, R3, R4, it should be evident that the practice of the invention is not limited to any four specific registers. Rather, selection of these designations is merely an aid to analysis and understanding. In fact, it should be evident that the analysis can be generalized, as implied by the equations given above.

A logical block diagram illustrating an apparatus for implementing the multifunctional ALU described essentially in FIGS. 5, 6A, 6B, 7A, and 7B is illustrated in FIG. 8. In FIG. 8, a register 50 receives a compound instruction including instructions 52 and 54. The compounded instructions have appended tags 56 and 58. The instructions and their tags are provided to decode and control logic 60 which decodes the instructions and the information contained in their tags to provide register select signals on output 62 and function select signals on output 66. The register select signals on output 62 configure a cross-connect element 64 which is connected to general purpose registers 63 to provide the contents of up to three registers to the three operand inputs AI0, AI1, and AI2 of a data dependency collapsing ALU 65. The ALU 65 is a multi-function ALU whose functionality is selected by function select signals provided on the output 66 of the decode and control logic 60. With operands provided from the registers connected through the cross-connect 64, the ALU 65 will perform the functions indicated by the function select signals produce a result on output 67.

Operating in parallel with the above-described ALU apparatus is a second ALU apparatus including decode and control logic 70 which decodes the first instruction in the instruction field 52 to provide register select signals to a conventional cross connect 872 which is also connected to the general purpose registers 63. The logic 870 also provides function select signals on output 874 to a conventional two-operand ALU 875. This ALU apparatus is provided for execution of the instruction in instruction field 52, while the second instruction in instruction field 54 is executed by the ALU 65. As described below, the ALU 65 can execute the second instruction whether or not one of its operands depends upon result data produced by execution of the first instruction. Both ALUs therefore operate in parallel to provide concurrent execution of two instructions, whether or not compounded.

Returning to the compounded instructions 52 and 54 and the register 50, the existence of a compounder is presumed. It is asserted that the compounder pairs or compounds the instructions from an instruction stream including a sequence of scalar instructions input to a scalar computing machine in which the compounder resides. The compounder groups instructions according to the discussion above. For example, category 1 instructions (FIG. 5) are grouped in logical/add, add/logical, logical/logical, and add/add pairs in accordance with Table 5. To each instruction of a compound set there is added a tag containing control information. The tag includes compounding bits which refer to the part of a tag used specifically to identify groups of compound instructions. Preferably, in the case of compounding two instructions, the following procedure is used to indicate where compounding occurs. In the System/370 machines, all instructions are aligned on a half word boundary and their lengths are either 2, 4 or 6 bytes. In this case, a compounding tag is needed for every half word. A one-bit tag is sufficient to indicate whether an instruction is or is not compounded. Preferably, a "1" indicates that the instruction that begins in a byte under consideration is compounded with the following instruction. A "0" indicates no compounding. The compounding bit associated with half words that do not contain the first byte of an instruction is ignored. The compounding bit for the first byte of the second instruction in the compound pair is also ignored. Consequently, only one bit of information is needed to identify and appropriately execute compounded instructions. Thus, the tag bits 56 and 58 are sufficient to inform the decode and control logic 60 that the instructions in register fields 52 and 54 are to be compounded, that is executed in parallel. The decode and control logic 60 then inspects the instructions 52 and 54 to determine what their execution sequence is, what interlock conditions, if any obtain, and what functions are required. This determination is illustrated for category 1 instructions in FIG. 5. The decode and control logic also determines the functions required to collapse any data hazard interlock as per FIGS. 6A and 6B. These determinations are consolidated in FIGS. 7A and 7B. In FIGS. 7A and 7B, assuming that the decode and control logic 60 has, from the tag bits, determined that instructions in fields 52 and 54 are to be compounded, the logic 60 sends out a function select signal on output 66 indicating the desired operation according to the left-most column of FIG. 7A. The OP codes of the instructions are explicitly decoded to provide, in the function select output, the specific operations in the columns headed OP1 and OP2 of FIGS. 7A and 7B. The register select signals on output 62 route the registers in FIG. 8 by way of the cross-connect 64 as required in the AI0, AI1, and AI2 columns of FIGS. 7A and 7B. Thus, for example, assume that the first instruction in field 52 is ADD R1, R2, and that the second instruction is ADD R1, R4. The eighteenth line in FIG. 7A shows the ALU operations which the decode and control circuit indicates by OP1=+ and OP2=+, while register R2 is routed to input AI0, register R4 to input AI1, and register R1 to input AI2.

Refer now to FIG. 9 for an understanding of the structure and operation of the data dependency collapsing ALU 65. In FIG. 9, a three-operand, single-result adder 70, corresponding to the adder of FIG. 3 is shown. The adder 70 obtains inputs through circuits connected between the adder inputs and the ALU inputs AI0, AI1, and AI2. From the input AI2, an operand is routed through three logic functional elements 71, 72 and 73 corresponding to logical AND, logical OR, and logical EXCLUSIVE-OR, respectively. This operand is combined in these logical elements with one of the other operands and routed to AI0 or AI1 according to the setting of the multiplexer 80. The multiplexer 75 selects either the unaltered operand connected to AI2 or the output of one of the logical elements 71, 72, or 73. The input selected by the multiplexer 75 is provided to an inverter 77, and the multiplexer 78 connects to one input of the adder 70 either the output of the inverter 77 or the uninverted output of the multiplexer 75. The second input to the adder 70 is obtained from ALU input AI1 by way of a multiplexer 82 which selects either "0" or the operand connected to ALU input AI1. The output of the multiplexer is inverted through inverter 84 and the multiplexer 85 selects either the noninverted or the inverted output of the multiplexer 82 as a second operand input to the adder 70. The third input to the adder 70 is obtained from input AI0 which is inverted through inverter 87. The multiplexer 88 selects either "0", the operand input to AI0, or its inverse provided as a third input to the adder 70. The ALU output is obtained through the multiplexer 95 which selects the output of the adder 70 or the output of one of the logical elements 90, 92 or 93. The logical elements 90, 92, and 93 combine the output of the adder by means of the indicated logical operation with the operand input to AI1.

It should be evident that the function select signal consists essentially of the multiplexer select signals A B C D E F G and the "hot" 1/0 selections input to the adder 70. It will be evident that the multiplexer select signals range from a single bit for signals A, B, E, and F to two-bit signals for C, D, and G.

The states of the complex control signal (A B C D E F G 1/0 1/0) are easily derived from FIG. 7A and 7B. For example, following the ADD R1, R2 ADD R1, R4 example given above, the OP1 signal would set multiplexer signal C to select the signal present on AI2, while the F signal would select the noninverted output of the multiplexer 75, thereby providing the operand in R1 to the right-most input of the adder 70. Similarly, the multiplexer signals B and E would be set to provide the operand available at AI1 in uninverted form to the middle input of the adder 70, while the multiplexer signal D would be set to provide the operand at AI0 to the left-most input of the adder 70, without inversion. Last, the two "I/O" inputs are set appropriately for the two add operations. With these inputs, the output of the adder 70 is simply the sum of the three operands, which corresponds to the desired output of the ALU. Therefore, the control signal G would be set so that the multiplexer 95 would output the result produced by the adder 70, which would be the sum of the operands in registers R1, R2, and R3.

When instruction compounding a logical/add sequence, the logical function would be selected by the multiplexer 75 and provided through the multiplexer 78 to the adder 70, while the operand to be added to the logical operation would be guided through one of the multiplexers 85 or 88 to one of the other inputs of the adder 70, with a 0 being provided to the third input. In this case, the multiplexer 95 would be set to select the output of the adder 70 as the result.

Last, in an add/logical compound sequence, the two operands to be first added will be guided to two of the inputs of the adder 70, while the 0 will be provided to the third input. The output of the adder is instantaneously combined with the non-selected operand in logical elements 90, 92 and 93. The control signal G will be set to select the output of the element whose operation corresponds to the second instruction of the compound set.

More generally, FIG. 9 presents a logical representation of the data dependency collapsing ALU 65. In deriving this dataflow, the decision was made to not support interlocks in which the result of the first instruction is used as both operands of the second instruction. More discussion of this can be found in the "Idiosyncrasies" section. That this representation implements the other operations required by LOGICAL-ADD compoundings can be seen by comparing the dataflow with the function column of FIG. 5. In this column, a LOGICAL-type operation upon two operands is followed by an ADD-type operation between the LOGICAL result and a third operand. This is performed by routing the operands to be logically combined to AI0 and AI2 of FIG. 9 and through the appropriate one of logical blocks 71, 72, or 73, routing this result to the adder 70, and routing the third operand through AI1 to the adder. Inversions and provision of hot 1's or 0's are provided as part of the function select signal as required by the arithmetic operation specified. In other cases, an ADD-type operation between two operands is followed by a LOGICAL-type operation between the result of the ADD-type and a third operand. This is performed by routing the operands for the ADD-type operation to AI0 and AI2, routing these inputs to the adder, routing the output of the adder to the post-adder logical blocks 90, 92 and 93, and routing the third operand through AI3 to these post-adder logical blocks. LOGICAL-type followed by LOGICAL-type operations are performed by routing the two operands for the first LOGICAL-type to AI0 and AI2 which are routed to the pre-adder logical blocks, routing the results from the pre-adder logical blocks through the ALU without modification by addition to zero to the post-adder logical block, and routing the third operand through AI3 to the post-adder logical block. For an ADD-type operation followed by an ADD-type operation, the three operands are routed to the inputs of the adder, and the output of the adder is presented to the output of the ALU.

The operation of the ALU 65 to execute the second instruction in instruction field 54 when there is no data dependency between the first and second instructions is straightforward. In this case, only two operands are provided to the ALU. Therefore, if the second instruction is an add instruction, the two operands will be provided to the adder 70, together with a zero in the place of the third operand, with the output of the adder being selected through the multiplexer 95 as the output of the ALU. If the second instruction is a logical instruction, the logical operation can be performed by routing the two operands to the logical elements 71, 72, and 73, selecting the appropriate output, and then flowing the result through the adder 70 by providing zeros to the other two adder inputs. In this case, the output of the adder would be equal to the logical result and would be selected by the multiplexer 95 as the output of the ALU. Alternatively, one operand can be flowed through the adder by addition of two zeros, which will result in the adder 70 providing this operand as an output. This operand is combined with the other operand in the logical elements 90, 92, and 93, with the appropriate logical element output being selected by the multiplexer 95 as the output of the ALU.

When instructions are compounded as illustrated in FIG. 8, whether or not dependency exists, the instruction in instruction field 52 of register 50 will be conventionally executed by decoding of the instruction 870, 874, selection of its operands by 70, 871, 872, and performance of the selected operation on the selected operands in the ALU 875. Since the ALU 875 is provided for execution of a single instruction, two operands are provided from the selected register through the inputs AI0 and AI1, with the indicated result being provided on the output 877.

Thus, with the configuration illustrated in FIG. 8, the data dependency collapsing ALU 65, in combination with the conventional ALU 875 supports the concurrent (or, parallel) execution of two instructions, even when a data dependency exists between the instructions.

AHAZ - COLLAPSING ALU

Address generation can also be affected by data hazards which will be referred to as address hazards, AHAZ. The following sequence represents a compounded sequence of System/370 instructions that is free of address hazards:

AR R1,R2

S R3,D(R4,R5)

where D represents a three nibble displacement. No AHAZ exists since R4 and R5 which are used in the address calculation were not altered by the preceding instruction. Address hazards do exist in the following sequences:

AR R1,R2

S R3,D(R1,R5)

AR R1,R2

S R3,D(R4,R1)

The above sequences demonstrate the compounding of an RR instruction (category 1 in FIG. 5) with RX instructions (category 9) presenting AHAZ. Other combinations include RR instructions compounded with RS and SI instructions.

For an interlock collapsing ALU, new operations arising from collapsing AHAZ interlocks must be derived by analyzing all combinations of instruction sequences and address operand conflicts. Analysis indicates that common interlocks, such as the ones contained in the above instruction sequences, can be collapsed with a four-to-one ALU.

The functions that would have to be supported by an ALU to collapse all AHAZ interlocks for a System/370 instruction level architecture are listed in FIG. 10. For those cases where four inputs are not specified, an implicit zero is to be provided. The logical diagram of an AHAZ interlock collapsing ALU defined by FIG. 10 is given in FIG. 11. A large subset, but not all, of the functions specified in FIG. 10 are supported by the illustrated ALU. This subset consists of the functions given in rows one to 21 of FIG. 10. The decision as to which functions to include is an implementation decision whose discussion is deferred to the "Idiosyncrasies" section.

As FIG. 11 shows, the illustrated ALU includes an adder 100 in which two three-input, two-output carry save adders 101 and 102 are cascaded with a two-input, single-output carry look ahead adder 103 in such a manner that the adder 100 is effectively a four-operand, single-result adder necessary for operation of the ALU in FIG. 11.

In generating FIG. 10, the complexity of the ALU structure was simplified at the expense of the control logic. This is best explained by example. Consider the two following System/370 instruction sequences:

NR R1,R2 (4)

S R3,D(R1,R5)

and

NR R1,R2 (5)

S R3,D(R4,R1).

Let the general notation for this sequence be

NR r1,r2

S r3,D(R4,R5).

For the first sequence, the address of the operand is:

OA=D+(R1.andgate.R2)+5

while that for the second sequence is:

OA=D+R4+(R1.andgate.R2)

To simplify the execution controls at the expense of ALU complexity, the following two operations would need to be executed by the ALU:

OA=AG10+(AG11.andgate.AG12)+AG13

OA=AG10+AG12+(AG11.andgate.AG13)

in which D is fed to AGI0, r2 is fed to AGI1, r4 is fed to AGI2 and r5 is supplied to AGI3. The ALU could be simplified however if the controls detect which of r4 and r5 possess a hazard with r1 and dynamically route this register to AGI2. The other register would be fed to AGI3. For this assumption, the ALU must only support the operation:

OA=AG10+(AG11.andgate.AG12)+AG13

Trade-offs such as these are made in favor of reducing the complexity of the address generation ALU as well as the execution and branch determination ALU's.

The ALU of FIG. 11 can be substituted for the ALU 65 in FIG. 8. In this case, the decode and control logic 60 would appropriately reflect the functions of FIG. 10.

BRANCH HAZARD-COLLAPSING ALU

Similar analyses to those for the interlock collapsing ALU's for execution and address generation must be performed to derive the affects of compounding on a branch determination ALU which is given by FIGS. 12 and 13. The branch determination ALU covers functions required by instructions comparing register values. This includes the branch instructions BXLE, BXH, BCT, and BCTR, in which a register value is incremented by the contents of a second register (BXLE and BXH) or is decremented by one (BCT and BCTR) before being compared with a register value (BXLE and BXH) or 0 (BCT and BCTR) to determine the result of the branch. Conditional branches are not executed by this ALU.

The ALU illustrated in FIG. 13 include a multi-stage adder 110 in which two carry save adders 111 and 112 are cascaded, with the two outputs of the carry save adder 112 providing the two inputs for the carry look ahead adder 113. This combination effectively provides the four-input, single result adder provided for the ALU of FIG. 13.

As an example of the data hazards that can occur, consider the following instruction sequence:

AL R1,D(R2,R3)

BCT R1,D(R2,R3)

Let [x] denote the contents of memory location x. The results following execution are:

R1=R1+[D+R2+R3]-1

Branch if (R1+[D+R2+R3])-1=0

This comparison could be done by performing the operation:

R1+[D+R2+R3]1-0.

The results of analyses for the branch determination ALU are provided in FIGS. 12 and 13 without further discussion. The functions supported by the dataflow include those specified by rows one to 25 of FIG. 12.

The ALU of FIG. 13 can be substituted for the ALU 65 in FIG. 8. In this case, the decode and control logic 60 would appropriately reflect the functions of FIG. 12.

IDIOSYNCRASIES

Some of the functions that arise from operand conflicts are more complicated than others. For example, the instruction sequence:

AR R1,R2

AR R1,R1

requires a four-to-one ALU, along with its attendant complexity, to collapse the data interlock because its execution results in:

R1=(R1+R2)+(R1+R2).

Other sequences result in operations that require additional delay to be incorporated into the ALU in order to collapse the interlock. A sequence which illustrates increased delay is:

SR R1,R2

LPR R1,R1

which results in the operation

R1=/R1-R2/.

This operation does not lend itself to parallel execution because the results of the subtraction are needed to set up the execution of the absolute value.

Rather than collapse all interlocks in the ALU, an instruction issuing logic or a preprocessor can be provided which is designed to detect instruction sequences that lead to these more complicated functions. Preprocessor detection avoids adding delay to the issue logic which is often a near-critical path. When such a sequence is detected, the issuing logic or preprocessor would revert to issuing the sequence in scalar mode, avoiding the need to collapse the interlock. The decision as to which instruction sequences should or should not have their interlocks collapsed is an implementation decision which is dependent upon factors beyond the scope of this invention. Nevertheless, the trade-off between ALU implementation complexity and issuing logic complexity should be noted.

Hazards present in address generation also give rise to implementation trade-offs. For example, most of the address generation interlocks can be collapsed using a four-to-one ALU as discussed previously. The following sequence

AR R1,R2

S R3,D(R1,R1);

however, does not fit in this category. For this case, a five-to-one ALU is required to collapse the AHAZ interlock because the resulting operation is:

OA=D+(R1+R2)+(R1+R2)

where OA is the resulting operand address. As before, inclusion of this function in the ALU is an implementation decision which depends on the frequency of the occurrence of such an interlock. Similar results also apply to the branch determination ALU.

GENERALIZATION OF THE ADDER

Analyses similar to those presented can be performed to derive interlock collapsing hardware for the most general case of n interlocks. For this discussion, refer to FIG. 14. Assuming simple data interlocks such as:

AR R1,R2

AR R3,R1

in which the altered register from the first instruction is used as only one of the operands of the second instruction, a(n+1) by one ALU would be required to collapse the interlock. To collapse three interlocks, for example, using the above assumption would require a four-to-one ALU This would also require an extra CSA stage in the ALU.

The increase in the number of CSA stages required in the ALU, however, is not linear. An ALU designed to handle nine operands as a single execution unit would take four CSA stages and one CLA stage. This can be seen from FIG. 14 in which each vertical line represents an adder input and each horizontal line indicates an adder. Carry-save adders are represented by the horizontal lines 200-206, while the carry look-ahead adder is represented by line 209. Each CSA adder produces two outputs from three inputs. The reduction in input streams continues from stage to stage until the last CSA reduces the streams to two. The next adder is a CLA which produces one final output from two inputs. Assuming only arithmetic operations, a one stage CLA adder, and a four stage CSA adder, the execution of nine operands as a single unit using the proposed apparatus could be accomplished, to a first order approximation, in an equivalent time as the solution proposed by Wulf in the reference cited above.

Data hazard interlocks degrade the performance obtained from pipelined machines by introducing stalls into the pipeline. Some of these interlocks can be relieved by code movement and instruction scheduling. Another proposal to reduce the degradation in performance is to define instructions that handle data interlocks. This proposal suffers from limitations on the number of interlocks that can be handled in a reasonable instruction size. In addition, this solution is not available for 370 architecture compatible machines.

In this invention, an alternative solution for relieving instruction interlocks has been presented. This invention offers the advantages of requiring no architectural changes, not requiring all possible instruction pairs and their interlocks to be architected into an instruction set, presents only modest or no impacts to the cycle time of the machine, requires less hardware than is required in the prior art solution of FIG. 1, and is compatible with System/370-architected machines.

While the invention has been particularly shown and described with reference to the preferred embodiment thereof, it will be understood by those skilled in the art that many changes in form and details may be made therein without departing from the spirit and scope of the invention.

* * * * *