PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : **G06F 9/38** | **A1**

(11) International Publication Number: **WO 97/13198**

(43) International Publication Date: 10 April 1997 (10.04.97)

(21) International Application Number: PCT/US96/15420

(22) International Filing Date: 3 October 1996 (03.10.96)

(30) Priority Data:

60/005,069	6 October 1995 (06.10.95)	US
60/005,021	10 October 1995 (10.10.95)	US
592,150	26 January 1996 (26.01.96)	US

(71) Applicant: ADVANCED MICRO DEVICES, INC. [US/US]; One AMD Place, Sunnyvale, CA 94088-3453 (US).

(72) Inventors: BEN-MEIR, Amos; 10447 Merriman Road, Cupertino, CA 95014 (US). FAVOR, John, G.; 5246 Leesa Ann Court, San Jose, CA 95124 (US).

(74) Agents: O'BRIEN, David, W. et al.; Skjerven, Morrill, MacPherson, Franklin & Friel, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).

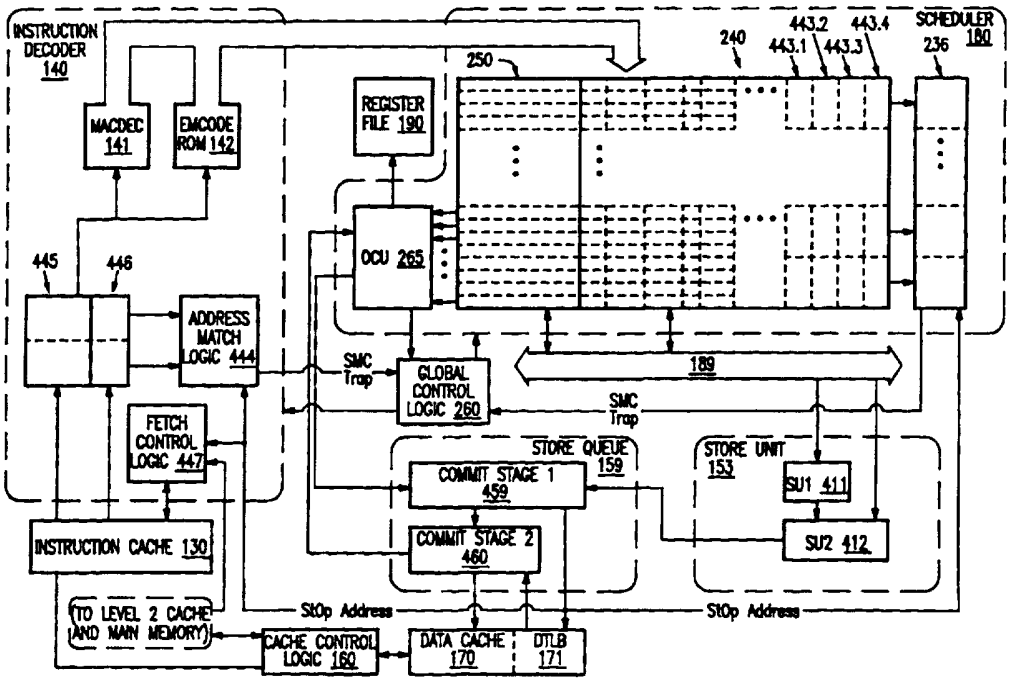
(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
With amended claims.

(54) Title: SELF-MODIFYING CODE HANDLING SYSTEM

(57) Abstract

A processor (100) which includes tags indicating memory addresses for instructions advancing through pipeline stages of the processor and which includes an instruction decoder (140) having a store target address buffer allows a self-modifying code handling system to detect store operations writing into the instruction stream and trigger a self-modifying code fault. In one embodiment of a self-modifying code handling system, a store pipe (153, 159) is coupled to a data cache (170) to commit results of a store operation to a memory subsystem (122). The store pipe supplies as store operation target address indication on commitment of a store operation result. A scheduler (180) includes ordered Op entries for Ops decoded from instructions and includes corresponding first address tags covering memory addresses for the instructions. First comparison logic (236) is coupled to the store pipe and to the first address tags to trigger self-modifying code fault handling means in response to a match between the store operation target address and one of the first address tags. An instruction decoder (140) is coupled between the instruction cache (130) and the scheduler (180). The instruction decoder includes instruction buffer entries and second address tags associated with the instruction buffer entries. Second comparison logic (444) is coupled to the store pipe and to the second address tags to trigger the self-modifying code fault handling means in response to a match between the store operation target address and one of the second address tags.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	**GB**	United Kingdom	**MW**	Malawi
AT	Austria	**GE**	Georgia	**MX**	Mexico
AU	Australia	**GN**	Guinea	**NE**	Niger
BB	Barbados	**GR**	Greece	**NL**	Netherlands
BE	Belgium	**HU**	Hungary	**NO**	Norway
BF	Burkina Faso	**IE**	Ireland	**NZ**	New Zealand
BG	Bulgaria	**IT**	Italy	**PL**	Poland
BJ	Benin	**JP**	Japan	**PT**	Portugal
BR	Brazil	**KE**	Kenya	**RO**	Romania
BY	Belarus	**KG**	Kyrgystan	**RU**	Russian Federation
CA	Canada	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SG**	Singapore
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovakia
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LR**	Liberia	**SZ**	Swaziland
CS	Czechoslovakia	**LT**	Lithuania	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**LV**	Latvia	**TJ**	Tajikistan
DK	Denmark	**MC**	Monaco	**TT**	Trinidad and Tobago
EE	Estonia	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**UG**	Uganda
FI	Finland	**ML**	Mali	**US**	United States of America
FR	France	**MN**	Mongolia	**UZ**	Uzbekistan
GA	Gabon	**MR**	Mauritania	**VN**	Viet Nam

SELF-MODIFYING CODE HANDLING SYSTEM

TECHNICAL FIELD

The invention relates to processors, and in particular to a system for handling self-modifying code in a pipelined processor.

BACKGROUND ART

Computer programs are typically designed, coded, and compiled with a simplifying assumption that the resulting object code will be executed in sequential order. However, despite this assumption, modern processor design techniques seek to exploit opportunities for concurrent execution of machine instructions, i.e., instruction parallelism. To maximize computational throughput, pipelining techniques can be used to map instruction parallelism to multiple stages of a single functional unit or execution path. In contrast, superscalar techniques, which include out-of-order instruction issue, out-of-order instruction completion, and speculative execution of instructions, map instruction parallelism to multiple functional units or execution paths. Modern processor designs often exploit both pipelining and superscalar techniques.

Out-of-order instruction issue involves the issuance of instructions to execution units with little regard for the actual order of instructions in executing code. A superscalar processor which exploits out-of-order issue need only be constrained by dependencies between the output (results) of a given instruction and the inputs (operands) of subsequent instructions in formulating its instruction dispatch sequence. Out-of-order completion, on the other hand, is a technique which allows a given instruction to complete (e.g. store its result) prior to the completion of an instruction which precedes it in the program sequence. Finally, speculative execution involves the execution of an instruction sequence based on predicted outcomes (e.g., of a branch) and allows a processor to execute instructions without waiting for branch conditions to actually be evaluated. Assuming that branches are predicted correctly more often than not, and assuming that a reasonable efficient method of undoing the results of an incorrect prediction is available, the instruction parallelism (i.e., the number of instructions available for parallel execution) will typically be increased by speculative execution (*see* Johnson, Superscalar Processor Design, Prentice-Hall, Inc., New Jersey, 1991, pp. 63-77 for an analysis).

Superscalar techniques largely concern processor organization independent of instruction set and other architectural features. Thus, one of the attractions of superscalar techniques is the possibility of developing a processor that is code compatible with an existing processor architecture, for example the x86 processor architecture. Many superscalar techniques apply equally well to either RISC or CISC architectures. However, because of the regularity of many of the RISC architectures, superscalar techniques have initially been applied to RISC processor designs. In particular, the three operand load/store architecture, fixed instruction lengths, limited addressing modes, and fixed-width registers associated with a RISC architecture and instruction set facilitate the single cycle decoding of multiple instructions necessary to consistently supply multiple execution units with work.

One approach to developing a superscalar processor that is code compatible with an x86 architecture has been to dynamically translate x86 instructions into RISC instructions, or operations, which may then be executed by

a RISC core or execution engine. Techniques for designing such a superscalar RISC processor are described in Johnson, Superscalar Processor Design.

Executing instructions out of sequential order, i.e., issuing and completing instructions out of sequential order, can increase a superscalar processor's performance by allowing the superscalar processor to keep multiple execution units operating in parallel and thereby improve throughput. Accordingly, a scheduler for a superscalar processor can improve overall performance by determining which instructions can be executed out-of-order and providing, or dispatching, those instructions to appropriate execution units. A scheduler for a superscalar processor must also handle interrupts and traps. Many processor architectures, including the x86 processor architecture, require that an architectural state be known just before or after an instruction generates an error, interrupt, or trap. This presents a difficulty when instructions are executed out of sequential order. Therefore, the scheduler must be able to undo instructions and reconstruct the system's state as if instructions executed in sequential order.

Self-modifying code represents a further complication. In the case of certain architectures, including those conforming to the x86 processor architecture, one part of an executing program may modify other parts of the same program. The modified instruction sequence parts may then executed.

For certain CISC architectures which allow programs to modify itself, including the x86 processor architecture, this type of questionable programming practice has become established within a relevant portion of the existing software base. As a result, to maintain compatibility, new processor implementations often must not only implement the direct semantics of the architecture's instructions set, but also maintain expected secondary semantic behavior. In the case of high performance pipelined, superscalar implementations, this can become a significant, and potentially difficult, requirement to satisfy.

To the extent that instructions are fetched from the memory subsystem after a store into the instruction stream has completed, there is no problem. However, if unmodified representations of an instruction exist within the various pipeline stages or functional units of a pipelined superscalar processor, consistency problems exist. The maintenance of consistency must encompass not only conventional data/instruction cache consistency, but also consistency with respect to memory store instructions modifying other instructions which are executed shortly thereafter.

The consistency problem is similar to that encountered with more conventional data/instruction cache structures used in high-performance processors where memory writes must be appropriately reflected in the state and/or contents of any affected cache entry. However, the scope of the self-modifying code problem is more severe. In extreme "store-into-instruction-stream" cases, a modifying instruction may be immediately followed by a branch and then a modified target instruction. Particularly for highly pipelined, high-performance processor designs, guaranteeing an execution path identical to that of an architectural standard processor (such as the x86 processor) can prove difficult and expensive in terms of additional hardware circuitry and design complexity.

Pipelining, particularly the deep pipelining that is common in high-performance implementations of CISC architectures, results in large instruction processing latencies and high degrees of overlap between the processing of successive instructions. On the other hand, the execution of a memory write generally takes place late in such

pipelines. Consequently, actions such as fetching instructions from memory or cache and speculatively dispatching instructions to execution pipelines can easily occur before the completion of a memory write which precedes the fetched or dispatched instruction in the execution sequence.

DISCLOSURE OF THE INVENTION

A processor which includes tags indicating memory addresses for instructions advancing through pipeline stages of the processor and which includes an instruction decoder having a store target address buffer allows self-modifying code support logic to detect store operations writing into the instruction stream and trigger a self-modifying code fault.

In one embodiment of the present invention, a self-modifying code handling system for a computer having operation entries for representing operations in stages from instruction fetch to result commitment and having a store pipe for committing store operands to target addresses in memory, includes first tag stores, first comparison logic, and control logic. The first tag stores are respectively associated with a first group of the operation entries and represent first addresses in memory of instructions corresponding to the associated operation entries. The first comparison logic is coupled to the first tag stores and to the store pipe. The first comparison logic supplies a self-modifying code indication in response to a match between the target address for a store operation committed by the store pipe and any of the first addresses represented in the first tag stores. The control logic is coupled to the first comparison logic and to the operation entries. The control logic flushes uncommitted ones of the operation entries in response to the self-modifying code indication.

In another embodiment of the present invention, an apparatus includes a memory subsystem, instruction and data caches coupled to the memory subsystem, execution units, a scheduler, first and second comparison logic, and an instruction decoder. One of the execution units includes a store pipe coupled to the data cache to commit results of a StOp to the memory subsystem. The store pipe supplies a StOp target address indication on commitment of a StOp result. The scheduler includes ordered Op entries for Ops decoded from instructions and includes corresponding first address tags covering memory addresses for the instructions. The first comparison logic is coupled to the store pipe and to the first address tags to trigger self-modifying code fault handling means in response to a match between the StOp target address and one of the first address tags. The instruction decoder is coupled between the instruction cache and the scheduler. The instruction decoder includes instruction buffer entries and second address tags associated with the instruction buffer entries. The second comparison logic is coupled to the store pipe and to the second address tags to trigger the self-modifying code fault handling means in response to a match between the StOp target address and one of the second address tags.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

FIGURE 1 is a block diagram of a superscalar computer processor providing out-of-order execution control in accordance with an embodiment of the present invention.

FIGURE 2 is a block diagram of a scheduler constructed in accordance with an embodiment of the present invention.

FIGURE 3 is a pipeline staging diagram illustrating architectural stages in the execution of instructions in accordance with an embodiment of the present invention.

FIGURE 4 is a block diagram of out-of-order load and store execution control components in accordance with an embodiment of the present invention.

FIGURE 5 is a block diagram of a computer system incorporating a processor that provides out-of-order load/store execution control in accordance with an embodiment of the present invention.

Use of the same reference symbols in different figures indicates similar or identical items.

MODE(S) FOR CARRYING OUT THE INVENTION

FIGURE 1 depicts a superscalar processor embodiment of the present invention. Superscalar processor **100** includes an execution engine **150** which implements a reduced instruction set computing (RISC) architecture, an instruction decoder **140**, caches, and a system interface **120** providing access to an address space represented in memory subsystem **122** and to devices on local busses (not shown).

Superscalar processor **100** includes a cache which, in the embodiment described herein, is organized as separate data and instruction portions. Data cache **170** and instruction cache **130** are coupled (through cache control logic **160** and via system interface **120**) to the address space represented in memory subsystem **122** which includes main memory and optionally includes additional levels of cache, illustratively an L2 cache. Access to an L2 level cache, i.e., to L2 cache control logic and an L2 data portion (not shown), may be provided via system interface **120**. Alternatively, L2 cache control logic may be interposed between cache control logic **160** (for L1) and system interface **120**.

Cache system designs are well known in the art. In particular, suitable designs implementing split, "Harvard Architecture" instruction and data caches (such as **170** and **130**) and multi-level cache hierarchies are well known in the cache arts. In most respects, the cache subsystem of superscalar processor **100** (i.e., data cache **170**, instruction cache **130**, cache control logic **160**, and an optional L2 cache) is of any such suitable design. However, for reasons apart from its caching performance, instruction cache **130** is integrated with pre-decode logic (not shown). Such integrated pre-decode logic identifies x86 instruction boundaries in the fetched instruction stream and facilitates the rapid decoding of instructions by instruction decoder **140**.

Referring again to FIGURE 1, instruction sequences are loaded from the memory subsystem into instruction cache **130** for anticipated execution by execution engine **150**. In accordance with the embodiment of processor **100** shown in FIGURE 1, instructions in instruction cache **130** are CISC instructions selected from a complex instruction set such as the x86 instruction set implemented by processors conforming to the x86 processor architecture.

Instruction decoder **140** converts CISC instructions received from instruction cache **130** to operations for execution engine **150**. In the embodiment of FIGURE 1, these operations are RISC-like operations (hereafter OPs) and a single x86 instruction from instruction cache **130** decodes into one or more OPs for execution engine **150**. Individual OPs fall into one of several type groups including register operations (RegOps), load-store operations (LdStOps), load immediate value operations (LIMMOps), special operations (SpecOps), and floating point operations (FpOps). Alternative embodiments may decode different instruction sets and supply different operation types for execution.

Instruction decoder **140** includes two instruction translation portions, a hardware translation portion MacDec **141** and a ROM-based translation portion **142**, together with branch prediction logic **143**. Most common x86 instructions are translated into short sequences of 1 to 4 OPs using multiple parallel hardware decoders included in hardware translation portion **141**. Hardware translation portion **141** decodes these common x86 instructions received from instruction cache **130** into short sequences of OPs which are then supplied to scheduler **180**. Less common x86 instructions and those x86 instructions which translate into OP sequences longer than 4 OPs are translated by a ROM-based translation portion **142** which fetches (from ROM) a translated sequence of OPs corresponding to the particular x86 instruction to be translated. Translated OP sequences from either source, whether generated by hardware decoders or fetched from ROM, are supplied to scheduler **180** for execution by execution engine **150**.

Referring again to FIGURE 1, execution engine **150** includes a scheduler **180**, a register file **190**, and multiple execution units which receive and execute OPs dispatched by scheduler **180**. In the embodiment of FIGURE 1, execution engine **150** includes seven execution units: load unit **152**, store unit **153**, register units **154** and **155**, floating point unit **156**, multimedia unit **157**, and a branch unit **158**, although alternative embodiments may add to or subtract from the set of execution units. In an exemplary embodiment, floating point unit **156** and multimedia unit **157** are omitted. Execution engine **150** also includes a store queue **159** interposed between store unit **153** and data cache **170**.

Scheduler **180** is organized as an ordered array of storage entries and logic blocks coupled thereto, which together provide support for out-of-order dispatch of Ops to execution units and for forwarding of Op results to one or more execution units. The ordered array of storage entries and logic blocks also implements a reorder buffer and provides for renaming of the architectural registers defined in register file **190** and speculative execution recovery. Instruction decoder **140** supplies scheduler **180** with new Ops decoded from the instruction stream. In turn, scheduler **180** stores and maintains (in a storage entry) data associated with each new Op received. In this way scheduler **180** tracks the status of each Op and its associated data as the Op is issued to, and executed by, an execution unit. After a given Op is fully executed and data dependencies have been resolved, it is retired and the corresponding scheduler entry is released.

Scheduler **180** is coupled to execution units (i.e., load unit **152**, store unit **153**, register units **154** and **155**, floating point unit **156**, multimedia unit **157**, and branch unit **158**) via a group of busses and control lines collectively shown as a bus **189**. Scheduler **180** supplies Ops, register operands, and control signals to the execution units and receives result values and status indications back from the execution units, illustratively via bus **189**. Of

course, all busses and control lines need not be fully connected and bus **189** is merely illustrative of the bi-directional coupling of scheduler **180** with the execution units.

Load unit **152** and store unit **153** execute LdStOps (i.e., LdOps and StOps), respectively loading data from and storing data to addressable memory. Depending on the caching state of a particular memory address, a LdStOp may complete at the L1 data cache **170**, at an L2 cache (not shown), or at main memory (also not shown). Store queue **159** temporarily stores data from store unit **153** so that store unit **153** and load unit **152** can operate in parallel without conflicting accesses to data cache **170**. Register units **154** and **155** execute RegOps which operate on a data associated with the architectural registers of register file **190**.

Scheduler Overview

FIGURE 2 depicts an exemplary embodiment of scheduler **180** having 24 entries (shown as rows) wherein each entry is associated with a pending Op. Each entry includes a series of fields, collectively shown as scheduling reservoir **240**, for representing static and dynamic data associated with a pending Op. In addition, scheduler **180** provides a series of specialized logic blocks, collectively shown as control logic **230**, coupled to the entries of scheduling reservoir **240** to receive data associated with pending Ops. The specialized logic blocks (shown as columns **231**, **232**, **233**, **235**, and **236**) of control logic **230** supply signals which control the sequencing of Op execution and the supply of operands to and distribution of results from the execution units. Control logic **230** includes issue selection logic **231**, operand selection logic **232**, load-store ordering logic **234**, status flag handling logic **235**, and self-modifying code support logic **536**.

Issue selection logic **231** controls the selection of Ops from scheduling reservoir **240** for issue to available execution units during each cycle. Operand selection logic **232** identifies an appropriate source for operand data required by Ops which have been issued to execution units. Depending on data dependencies and sequencing of Ops within the execution engine **150**, the appropriate source may be register file **190**, a destination value field associated with another pending Op entry (destination value fields for scheduler entries are shown collectively as **250**), or the result of a complete Op which is supplied on one of the result buses (shown collectively as result busses **272**). Control signals supplied by issue selection logic **231** and operand selection logic **232** allow scheduler **180** to issue Ops from scheduling reservoir **240** to available execution units and to select the appropriate operand source for each Op issued.

Although scheduler **180** issues Ops out-of-order and execution units (e.g., load unit **152**, store unit **153**, register unit X **154**, register unit Y **155**, and branch unit **158**) execute Ops out-of-order, certain Op pairs must be completed in-order with respect to each other. For example, LdOps and StOps which read from and write to the same physical memory location must access memory in-order. Load-store ordering logic **234** maintains such execution ordering between LdOps and StOps.

Self-modifying code support logic **236**, which is described in greater detail below, triggers a self-modifying code fault in response to indications from store queue **159** and physical address tag fields **243**. Store queue **159** provides several bits of the target linear and physical addresses for StOps that store queue **159** is preparing to commit. Self-modifying code support logic **236** compares these address bits to instruction address (or

addresses, if the instructions were from different pages) stored as physical address tag fields **243** for each Op quad. If any quad matches, there may be a write to an instruction which has already been fetched or is now present (decoded) as an operation. Accordingly, self-modifying code support logic **236** signals global control logic **260** to flush scheduler **180** and the fetch/decode process is restarted from the instruction following the last committed instruction (i.e., the instruction following the instruction that modified the instruction stream). Scheduler **180** treats detection of self-modifying code as a trap or fault (i.e., it factors into "trap pending").

Scheduler **180** includes a destination value field associated with each scheduler entry. Collectively these destination value fields are shown as **250**. In conjunction with operand selection logic **232**, destination value fields **250** implement a reorder buffer and implicit register renaming. Operand values associated with architectural registers of register file **190** are represented in destination value fields **250** and are typically supplied to execution units as register operand values via operand busses **271**. However, operand values may instead be supplied from register file **190** if none of the destination value fields **250** represent a more recent register state (i.e., an as yet uncommitted register state). Results of completed Ops are supplied via result busses **272** to the destination value field of the scheduler entry associated with the completed Op. In addition, these results may also be supplied to execution units as operands for pending Ops. Results are forwarded via result busses **272**.

The fields of a scheduling reservoir entry (illustratively, scheduling reservoir entry **240.1**) contain information regarding an operation (Op) which is awaiting execution, which is in the process of being executed, or which is completed. Most of the fields of a scheduling reservoir entry are initialized when instruction decoder **130** loads a new Op into scheduling reservoir **240**. However, other fields are later loaded or updated. For example, a state field (shown for each entry as field **242**) is updated as the corresponding Op advances through stages of an execution pipeline. Storage fields that retain a value from the time an Op is loaded into scheduling reservoir **240** until retired from scheduler **180** are referred to as "static fields." Fields which can be updated with new values are referred to as "dynamic fields." Static field data and initial data values of dynamic fields are supplied by instruction decoder **140**.

A 3-bit field, `Type[2:0]`, of each scheduling reservoir entry (shown in FIGURE 2 as type field **241**) specifies the Op type associated with the scheduling reservoir entry. Op type is particularly important for issue selection purposes (e.g., LdOps should issue to a load unit such as **150**); however, load/store ordering control also makes use of type field **241**. The following signals are decoded from type field **241**:

```
000=      A Special operation not actually executed.
010=LU    A LdOp executed by load unit 152.
10x=SU    A StOp executed by store unit 153.
101=ST    A StOp which references memory or    at least generates a faultable
          address (i.e. not an LEA operation).
11x=RU    A RegOp executed by register unit X 154 or possibly register unit Y
          155.
110=RUX   A RegOp executable ONLY by register unit X 154.
111=RUY   A RegOp executable by register unit X 154 or register unit Y 155.
```

A 4-bit field, `State[3:0]`, of each scheduling reservoir entry (shown in FIGURE 2 as type state **242**) indicates the current execution state of an Op (`S3`, `S2`, `S1`, and `S0` are alternate signal names for `State[3:0]`.) Five possible states of type field **242** are encoded by a shifting field of ones as follows:

```
0000   Unissued
0001   Stage 0
0011   Stage 1
0111   Stage 2
1111   Completed
```

Intermediate states correspond to the current execution stage for an Op corresponding to the entry in which the type field appears. The bits are updated (effectively by left shifting) as the Op is successfully issued or advances out of a stage. `State[3:0]` is also set to 1111 during abort cycles.

Scheduler Op Quad Organization

Scheduler **180** includes 24 entries in scheduling reservoir **240** and destination value fields **250** which are managed as a FIFO. Data corresponding to new Ops are loaded in at the "top," shift toward the "bottom" as execution of progresses, and are retired from the bottom of scheduling reservoir **240**. To simplify control, scheduler **180** manages scheduling reservoir **240** and destination value fields **250** on an Op quad basis. Ops are loaded into, shifted through, and retired from scheduling reservoir **240** in groups of four. In this way, scheduler granularity matches the decode bandwidth of both the emcode ROM **142** and MacDec **141** of instruction decoder **140**. Scheduler **180** therefore manages 24 Op entries as six Op quad entries in a six-deep, four-wide FIFO.

Consequently, scheduling reservoir **240** can be viewed as a six-entry shift register containing Op quads. Each Op quad contains four Op entries, plus additional fields associated with the Op quad as a whole. These Op quad fields, e.g., physical address tag fields **243**, are supplied by instruction decoder **140**.

Physical address tag fields **243** include `Smc1stAddr`, `Smc1stPg`, `Smc2ndAddr`, and `Smc2ndPg` fields. Together with an Op quad valid field, `OpQV`, these physical address tag fields **243** provide descriptive information to self-modifying code support logic **236**, which is organized with Op quad granularity. Illustratively, physical address tag fields **243.1** and self-modifying code support logic **236.1** correspond to Op quad 0 of scheduler **180**. `Smc1stAddr` and `Smc1stPg` represent portions of a first physical memory address for CISC instructions from which an Op (or Ops) of the associated Op quad were decoded. In the exemplary embodiment, physical address tag fields **243** `Smc1stPg` and `Smc1stAddr` encode bits 19:12 and 11:5 (respectively) of the physical memory address for the CISC instruction associated with the first Op of the Op quad. Because the CISC instruction precursors of the Ops of an Op quad may cross cache line boundaries a second physical memory address may be necessary to fully tag an Op quad with the addresses of its associated CISC instructions. In such a case, `Smc2ndAddr` and `Smc2ndPg` represent portions of a second physical memory address for CISC instructions from which an Op (or Ops) of the associated Op quad were decoded. In the exemplary embodiment, physical address tag fields **243** `Smc2ndPg` and `Smc2ndAddr` encode bits 19:12 and 11:5 (respectively) of the physical memory address for the cross-cache-line CISC instructions associated with a subsequent Op (or Ops) of the Op quad. Instruction decoder **140** supplies

physical address tag fields **243** `Smc1stAddr` and `Smc1stPg` (and `Smc2ndAddr` and `Smc2ndPg` if there are CISC instructions from more than one physical memory page represented in the Op quad) to scheduling reservoir **240**.

Operation (Op) Timing and Execution Stages

Each entry of scheduling reservoir **240** includes fields describing outstanding Ops. These fields store static information originally derived from the Ops fetched or decoded by instruction decoder **140** and also dynamic state information resulting from Op execution or characterizing the execution pipeline status of a given Op.

From a processor control perspective, scheduler **180** is an instruction sequence-ordered array of Op state information (scheduling reservoir **240**) with associated control logic **230** generating control signals to issuing Ops from the array to respective execution units, to control Op execution through sequences of pipeline stages, and to eventually retiring Ops from the scheduler. As shown in FIGURE 2, control logic **230** includes five specialized blocks of control logic (issue selection logic **231**, operand selection logic **232**, load-store ordering logic **234**, status flag handling logic **235**, and self-modifying code support logic **236**), each having portions (illustratively portion **234.3** of load-store ordering logic **234**) receiving information from corresponding entries of scheduling reservoir **240**. Control logic blocks supply control signals to the execution units. For example, load-store ordering logic **234** supplies control signals to load unit **152** and store unit **153** via control lines represented collectively as **273**.

The particular control signals supplied by control logic blocks of scheduling reservoir **240** depend on the state of fields in Op entries. In particular, the `State[3:0]` field indicates the progress of execution of associated operations. From a logical perspective, all state sequencing within the scheduler is single cycle in nature. State transition decisions are made each cycle based on the machine state during the cycle. The structure of scheduler **180** reflects the pipelined nature of Op execution. Scheduler **180** (and correspondingly each entry) can be divided into many distinct, rather independent logic portions, each of which is directly associated with a specific processing stage of a given type of operation or execution pipeline.

Pipeline staging of execution engine **150** is now described with reference to FIGURE 3. Once an Op is loaded into execution engine **150**, the Op goes through a three or four stage pipeline, and correspondingly transitions between four or five states represented by the field `State[3:0]` within the scheduler entry associated with the Op. Instruction fetch and decode are performed before execution engine **150**, therefore the first scheduler-related pipeline stage is the issue stage. FIGURE 3 shows pipeline staging for RegOps and LdStOps.

Scheduler **180** exerts primary control over execution pipelines during the issue and operand fetch stages, **330** and **340**. Processing within issue stage **330** and within operand fetch stage **340** can be broken down into two phases per stage, wherein each phase nominally occupying a half clock cycle. Issue stage **330** includes an issue selection phase and a broadcast phase, while operand fetch stage **340** includes an operand selection phase and operand forwarding phase.

Issue Stage

During the issue selection phase **330.1** of issue stage **330**, scheduler **180** selects the next Ops to enter the pipelines associated with load unit **152**, store unit **153**, register unit X **154**, and register unit Y **155** (four Op

selections occur at once). During the broadcast phase **330.2** of issue stage **330**, information about each of the register operands for each selected Op is broadcast to all scheduler entries and to external logic (including register file **190** and the execution units). In this way, the broadcast phase **330.2** sets up to locate operand values which may reside in one of the destination value fields **250** of scheduler **180** or in register file **190**, or which may correspond to results to be produced on result busses **272** one of the execution units (e.g., load unit **152**, store unit **153**, or register units **154** and **155**).

Operand Fetch Stage

During the operand selection phase **340.1** of operand fetch stage **340**, scheduler **180** locates up to eight operand values (4 Ops * 2 operands/Op) and determines the status of each operand value, i.e., whether a valid value is in fact available from the designated source. Based on this information, scheduler **180** determines which of the Ops in operand fetch stage 0 (stage **340**) will advance into their respective execution pipes, i.e., into stage 1 (stage **350**), following the operand forward phase. Advancement decisions are made independently for each Op and only operand dependencies need constrain the order with which operations are actually executed. Absent such data dependencies, Ops which issue to different execution units are generally processed through their respective pipelines in arbitrary order with respect to those Ops assigned to other execution units. One exception to this general rule involves the respective ordering of loads and stores (i.e., of LdOps and StOps) and is in greater detail discussed below.

LdStOp Execution Stages

The first two scheduler-related stages, the "operand issue" stage **330** and the "operand fetch" stage **340** are common to RegOps and LdStOps. Subsequent stages are the execution stages. RegOps include a single execution stage **350** because all RegOps execute in a single cycle. Furthermore, once a RegOp enters the execution stage, it always successfully completes and exits stage **350** at the end of that clock cycle. LdStOps, on the other hand, have two execution stages **352** and **360**, during which address calculation, segment and page translation (and protection checking), and data cache accessing (in the case of LdOps) all take place. Unlike RegOps, LdStOps can be held up for arbitrary periods of time in either stage **360** or **370**. Most hold ups appear in the second stage **370**. Most commonly, hold ups in stage **370** result from data cache **170** misses, data TLB **171** misses, and page faults. Hold ups in stage **360** result from misaligned memory references and from stage **370** being occupied and blocked by an LdStOp not advancing to completion.

During the operand forward phase **340.2** of operand fetch stage **340**, scheduler **180** transfers operand values from the designated sources via operand busses and/or result busses shown collectively in FIGURE 2 as busses **271** and **272** to execution units such as load unit **152**, store unit **153**, register unit X **154**, and register unit Y **155**. The exemplary embodiment includes nine operand busses **271**, eight of which provide operand values for operations in stage 0. Also in the exemplary embodiment, operand transfers occur regardless of whether values are valid, thereby simplifying control logic. If an operand value is invalid, it is ignored by the respective execution unit because the scheduler **180** does not advance the associated operation to stage 1. Immediate values for RegOps are handled as part of the register operand forwarding mechanism described above. In such cases, the immediate value

is forwarded directly from the particular one of the destination value fields **250** of the scheduler **180** entries associated with the Op.

Displacement values are also transferred during operand forward phase **340.2**, via displacement busses **189.4**, to load unit **152** and store unit **153** (independent values to each unit). These displacements are 32-bit values and always come from the entries of scheduler **180**. The selection of the source entry occurs during operand selection phase **340.1**. When a LdOp or a StOp enters stage 1, load unit **152** and store unit **153** latch associated displacement and operand values.

Scheduler **180** implements the four-phase control mechanism (as described above) for providing the address operands and displacement; however, StOps require a store data operand in addition to address operands and displacement values. Scheduler **180** performs a four-phase process for obtaining the store data for a StOp. The StOp data obtaining process is similar to that described above; however the store data is obtained during execution stage 2 (**370**). The process for providing the store data is synchronized with stages 1 and 2 of the StOp and includes a selection phase **390.1** identifying the StOp in execution stage 1, a broadcast phase **390.2** transmitting information describing the source of a data operand, a data operand selection phase **390.3**, and an data operand forwarding phase **390.4**. Store data is fetched in parallel with StOp execution; and the actual data value is obtained and provided to store queue **159** upon completion of StOp processing. If a valid store data value is not available, the StOp is held up in stage 2.

Op Completion Stage

In the exemplary embodiment, RegOps and LdOps complete by storing results to one of the destination value fields **250** of scheduler **180**. Each of the destination value fields **250** is associated with an Op entry and is used as a temporary store (a reorder buffer) for values which may eventually be committed to register file **190** by OCU **265**. For StOps, the corresponding temporary store before commitment to memory is store queue **159**. Store queue **159** buffers memory writes associated with a StOp in a first commit stage until OCU **265** releases the memory write to a second commit stage.

Op Commitment and Retirement

Register, flag, and memory state changes associated with completed Ops are committed (or made permanent) by OCU (Operation Commit Unit) **265**. OCU **265** then retires the corresponding Op entry from scheduler **180**. Several types state changes can result from the execution of an Op. The principal types of state changes are abortable and include: general register changes; status flag changes; and memory writes. General register changes result from all RegOps, LdOps, LIMM Ops, LDKxx operations, and STUPD StOps. Status flag changes result from ".cc" RegOps, and memory writes result STxxx StOps. Scheduler **180** and store queue **159** support abortable state changes through the general technique of temporarily storing register and status results in the destination value fields **250** and scheduling reservoir **240** of scheduler **180** and by storing memory write data in store queue **159**. Temporary (or speculative) register values, status values, and memory write values are held until the associated Ops are committed and retired by OCU **265**. Scheduler **180** speculatively supplies register values, status values, and memory write values residing in scheduling reservoir **240** and store queue **159** to dependent Ops

as necessary. However, permanent state changes to register file **190** and to the memory address space (distributed among data cache **170**, instruction cache **130**, an L2 cache, and main memory) are made during Op commitment.

During each cycle, OCU **265** examines each of the Op entries within the bottom Op quad entry and tries to commit the results of as many of these operations as possible. The state changes associated with the four Ops of an Op quad may be committed in one cycle or over many cycles. If all the Ops of an Op quad have been committed or are being successfully committed, the Op quad is retired from scheduler **180** at the end of the current cycle. Otherwise, as many state changes as possible are committed and the process is repeated during successive cycles until all state changes have been committed.

Commitments of register results, status results, and memory writes are performed independently. For Ops which have multiple results (e.g., a RegOp with both register and status results, or a STUPD operation with both a register result and a memory write), the various results are not necessarily committed simultaneously. Instead, commitment of one type of state change is independent of the other. The overall commitment of an Op occurs when the last result is committed. In general, results of an Op are not committed until:

1. the Op execution state (`State[3:0]`) of the Op entry indicates the Op is `Completed`;
2. the `State[3:0]` of any preceding faultable operations, namely any preceding LdStOps, is `Completed`, which implies that the operations are fault-free; and
3. the `State[3:0]` of any preceding BRCOND operation is `Completed`, which implies that the BRCOND was correctly predicted.

For StOps which generate a memory write, an additional constraint is that only one write can be committed per cycle from store queue **159** into data cache **170**. OCU **265** can commit up to four register and four status results and one memory write per cycle and typically commits and retires an Op quad from scheduler **180** every cycle. An Op quad can remain unretired at the bottom of scheduler **180** for more than one cycle only if the Op quad contains multiple memory write StOps or if one of the operations in the Op quad is sufficiently delayed in its execution that the associated `State[3:0]` field **242** is not yet marked `Completed`.

OCU **265** manages and controls the commitment of memory write data values associated with StOps to the memory address space, i.e., to locations in the L1 cache (data cache **170** and instruction cache **130**), an L2 cache, and main memory. Memory write commitment involves an associated store queue **159** entry and at most one memory write is committed by OCU **265** per cycle. OCU **265** scans scheduling reservoir **240** field values for Op entries in the bottom two Op quad to identify StOps with memory writes to commit.

When a StOp completes execution in store unit **153**, the associated target memory address and store data is entered in store queue **159**. Later, when the memory write for a StOp is actually committed, this entry is read and retired from store queue **159**. Since StOps are executed in order and committed in order, store queue **159** is managed as a simple FIFO. As a result, the matching of store queue **159** entries with associated StOps in scheduler **180** is straightforward.

During each cycle, OCU **265**'s memory write commit logic searches the bottom two Op quad entries of scheduler **180** for the next/oldest uncommitted memory-writing StOp (i.e. for the next StOp and associated store

queue **159** entry to try and commit). Since scheduler **180** and store queue **159** are both managed as FIFOs, the Op entry selected by OCU **265** must be associated with the bottom/oldest entry of store queue **159**.

The StOp (memory write) commitment process is implemented as a two-stage commit pipeline. During the first commit stage, no control decisions are made. Instead, OCU **265** triggers a data cache tag lookup for the store queue **159** entry associated with the next/oldest uncommitted memory-writing StOp in scheduler **180**. The accessed tag data is simply latched for examination during the second commit stage. Data cache **170** tag lookup is performed "blindly," i.e., without consideration for whether the associated StOp is presently committable. In the exemplary embodiment, OCU **265** selects an Op entry from scheduler **180** and store queue **159** concurrently presents the memory write address for the associated store queue **159** entry to data cache **170** (i.e., initiates a tag lookup).

A write commit is able to advance into commit stage 2 when that stage is either empty or is successfully completing the commitment of a write. When a memory write from store queue **159** does enter commit stage 2, the associated StOp can be retired from scheduler **180**. OCU **265** determines whether the selected StOp is committable, i.e., whether:

1. the Op execution state (`State[3:0]`) of the Op entry indicates the selected StOp is `Completed`;
2. the `State[3:0]` of any preceding faultable operations is `Completed`; and
3. the `State[3:0]` of any preceding BRCOND operation is `Completed`.

If the selected StOp is committable and the write commit has been able to advance into the second write commit stage, OCU **265** considers the StOp to be committed. In the next cycle, OCU **265** searches for and moves on to the next memory-writing StOp and the remainder of the commit process proceeds asynchronous to OCU **265** and scheduler **180**.

The write commit pipeline of store queue **159** is one write wide and therefore supports the commitment of only one memory-writing StOp per cycle. For Op quads which containing no more than one memory-writing StOp, this allows the possible commitment and retirement of one Op quad per cycle. However, for Op quads containing two, three, or four such StOps, a corresponding minimum number of cycles is required to commit the each StOp entry of the Op quad. As a result such an Op quad remains at the bottom of scheduler **180** for at least the corresponding number of cycles.

This throughput mismatch is partially mitigated by OCU **265** support for committing memory writes associated with StOps in the next oldest Op quad (Op quad 4). Since memory writes are committed in order, this allows OCU **265** to get a "head start" on multiple write Op quads when the bottom Op quad is held up but otherwise empty of uncommitted memory writes, or when it simply does not contain any StOps. This helps to better match OCU **265**'s one write per cycle commitment rate to the average number of memory writes per Op quad which is less than one per Op quad.

A special situation arises when a StOp's memory reference crosses an alignment boundary (currently 8 bytes) and is split by store unit **153** into two memory writes having two associated entries in store queue **159**. In

such situations, OCU **265** take two cycles to retire the two entries of store queue **159** and does not officially commit the StOp until the second cycle. If the StOp faults, it is aborted without retirement of either entry of store queue store queue **159**.

The following pseudo-RTL description summarizes the functionality of the OCU **265**'s write commit logic. OP0 is the oldest Op and OP3 is the youngest Op in the bottom/last Op quad of scheduler **180**. Similarly, OP4-OP7 are the corresponding Ops in the second to last Op quad of scheduler **180** and OP8-OP11 are the corresponding Ops in the third to last Op quad of scheduler **180**. The operation of OCU **265** is based on a set of mask bits (CmtMask[7:0]) which represent the OCU **265**'s progress in committing memory-writing StOps from the last two Op quad.

In operation, the first N-bits (starting from bit 0) of CmtMask[7:0] are clear indicating that OCU **265** has committed any StOps up to the Nth such Op position, which contains the next StOp to be committed. All Ops corresponding to the remaining, set mask bits of CmtMask[7:0] have yet to be examined for committable StOps. OCU **265** also maintains a set of bits (UncmtStOp[7:0]) indicating which Op positions contain uncommitted memory-writing StOps.

During each cycle, OCU **265** selects the next uncommitted StOp and generates a new set of mask bits based on the position of the selected StOp. The unmasked Ops are examined to determine whether the selected StOp is presently committable or an abort cycle needs to be initiated. In the former case, if the selected StOp is committable and if stage 2 of the commit pipe is able to accept a new write commit at the end of the cycle, OCU **265** commits the StOp and updates the UncmtStOp bits. OCU **265** also shifts the bits of UncmtStOp to match any shifting of the last two Op quads.

```
StCmtSel[3:0] = priority_encode(
      (OPQ5:OpQV UncmtStOp[0]), ... ,
      (OPQ5:OpQV UncmtStOp[3]),
      (OPQ4:OpQV UncmtStOp[4]), ... ,
      (OPQ4:OpQV UncmtStOp[7]) )
      //StCmtSel=0000 if OP0 selected (highest priority)
      //StCmtSel=0111 if OP7 selected (lowest priority)
      //StCmtSel=1111 if no Op selected

CmtMask[7:0] = {(StCmtSel[2:0] < 'b111),...,
      (StCmtSel[2:0] < 'b000)}
//this generates a field of zeroes from bit 0 up to and
//including the bit pointed at by StCmtSel[2:0], and a
//field of ones past this up to bit 7
//note: most of these terms can be simplified

CmtCiaCda =
      (~CmtMask[7] OP7:Type[2]) +
      (~CmtMask[6] CmtMask[7] OP6:Type[2]) +
      (~CmtMask[5] CmtMask[6] OP5:Type[2]) +
      (~CmtMask[4] CmtMask[5] OP4:Type[2]) +
      (~CmtMask[3] CmtMask[4] OP3:Type[2]) +
      (~CmtMask[2] CmtMask[3] OP2:Type[2]) +
      (~CmtMask[1] CmtMask[2] OP1:Type[2]) +
```

```
        (~CmtMask[0] CmtMask[1] OP0:Type[2])
    StCmtInh = CmtInh + StCmtSel[2] & (OPQ4:LimViol +
        SmcHit ~CmtCiaCda + "trap pending")
    StCmtV = ~StCmtSel[3] ~StCmtInh (CmtMask[7] + OP7:S3) &
        (CmtMask[6] + OP6:S3 + OP6:RU) &
        (CmtMask[5] + OP5:S3 + OP5:RU) &
        (CmtMask[4] + OP4:S3 + OP4:RU) &
        (CmtMask[3] + OP3:S3 + OP3:RU) &
        (CmtMask[2] + OP2:S3 + OP2:RU) &
        (CmtMask[1] + OP1:S3 + OP1:RU)
    Q5StCmtV = ~StCmtSel[2] ~CmtInh (CmtMask[3] + OP3:S3) &
        (CmtMask[2] + OP2:S3 + OP2:RU) &
        (CmtMask[1] + OP1:S3 + OP1:RU) &
        (CmtMask[0] + OP0:S3 + OP0:RU)
    StAdv = ~STQ_FirstAddr ~DC_HoldSC1 CHP_AdvSC2 +
        CmtCiaCda
    StRetire = StCmtV StAdv
    Q5StRetire = StAdv Q5StCmtV
    NewUncmtStOp[7:0] = { (CmtMask[7] OP7:Type=ST), ... ,
        (CmtMask[0] OP0:Type=ST) }
    AllStCmt = StCmtSel[2] + Q5StRetire ~NewUncmtStOp[3]
        &...& ~NewUncmtStOp[0]
    //indicates when all memory-writing StOps have been
    //committed or are being successfully committed in the
    //bottom scheduler Op quad entry

    //update UncmtStOp bits:
    NextUncmtStOp[7:0] =
        (StRetire) ? NewUncmtStOp[7:0] : UncmtStOp[7:0]
    NextUncmtStOp[11:8] = { OP11:Type=ST, OP10:Type=ST,
        OP9:Type=ST, OP8:Type=ST }
    @clk:
        UncmtStOp[7:4] = (LdEntry4) ? NextUncmtStOp[11:8] :
            NextUncmtStOp[7:4]
        UncmtStOp[3:0] = (LdEntry5) ? NextUncmtStOp[7:4] :
            NextUncmtStOp[3:0]
    SC_HoldSC1 = ~StQCmtV + CmtCiaCda
    StAbort = ~StCmtSel[2] SUViol &
        ((StCmtSel[1:0] == 00) ~OP0:S3 +
        (StCmtSel[1:0] == 01) ~OP1:S3 OP0:S3 +
        (StCmtSel[1:0] == 10) ~OP2:S3 OP1:S3 OP0:S3 +
        (StCmtSel[1:0] == 11) ~OP3:S3 OP2:S3 OP1:S3 OP0:S3)
```

Self-Modifying Code Handling Logic

Memory writes are committed to the address space (i.e., to data cache **170**, to instruction cache **130**, to an L2 cache, and/or to main memory) in phase 2 **382.2** of LdStOp commitment stage **382**. Since load-store ordering logic **234** enforces execution ordering between LdOps and StOps which access the same memory address, a younger load is guaranteed to return the just-committed memory write data. However, if the memory write committed in phase 2 **382.2** of LdStOp commitment stage **382** stores into the instruction stream, younger Ops (and their precursor x86 instructions) in various pipeline stages (i.e., x86 instruction fetch stage **310**, x86 instruction decode stage **320**, issue stage **330**, operand fetch stage **340**, execution stages **351**, **352**, and **360**) may be based on stale instruction bytes. Even Ops which have completed and are awaiting commitment by OCU **265** may be based

on stale instruction bytes. Self-modifying code handling components of scheduler **180** and instruction decoder **140** trap stores into the instruction stream to flush stale data as described below.

Referring to FIGURE 4, StOps are committed to the address space by stage 2 **460** of store queue **159**. The corresponding Op quad is retired from scheduler **180** by OCU **265** if each of entries of the Op quad has been completed (or is in the process of being committed). Stage 1 **459** of store queue **159** provides portions of the linear and physical address (i.e., the StOp address) for memory write data which the store queue **159** is preparing to commit in stage 2 **460**. In particular, stage 1 **459** of store queue **159** provides bits 11-5 of the linear address `STQ_LinAddr(11,5)` and bits 19-12 of the physical address `STQ_PhysAddr(19,12)`. Self-modifying code support logic **236** of scheduler **180** receives the StOp address and compares it against respective physical address tags `Smc1stAddr`, `Smc1stPg`, `Smc2ndAddr`, and `Smc2ndPg` stored in Op quad fields **443.1**, **443.2**, **443.3**, and **443.4** of scheduling reservoir **240**. Based on this comparison, self-modifying code support logic **236** determines whether the StOp being committed by store queue **159** writes to an address covered by any Op quad in scheduler **180**. If so, self-modifying code support logic **236** triggers a Self-Modifying Code (SMC) trap. Global control logic **260** flushes scheduler **180** and the fetch/decode process is restarted from the instruction following the last committed instruction (i.e., the instruction following the instruction that modified the instruction stream).

As previously described, instruction decoder **140** supplies the contents of Op quad fields **443.1**, **443.2**, **443.3**, and **443.4** (collectively shown in FIGURE 2 as physical address tag fields **243**) as Ops are issued to scheduler **180**. The physical address tags `Smc1stAddr`, `Smc1stPg`, `Smc2ndAddr`, and `Smc2ndPg` stored in Op quad fields **443.1**, **443.2**, **443.3**, and **443.4** represent bits 19-5 of the first and second physical memory addresses for x86 instructions from which Ops of the corresponding Op quad were decoded. Two physical memory addresses are required when the Ops of the corresponding Op quad were decoded from an x86 instruction (or instructions) which cross a cache line boundary. These following pseudo-RTL further describes the design and operation of self-modifying code support logic **236**:

```
for (i=0; i < 5; ++i) {
      unit Match1st =
         (STQ_LinAddr(11,5) == OpQi:Smc1stAddr) &&
         (STQ_PhysAddr(19,12) == OpQi:Smc1stPg);
      unit Match2nd =
         (STQ_LinAddr(11,5) == OpQi:Smc2ndAddr) &&
         (STQ_PhysAddr(19,12) == OpQi:Smc2ndPg);
      MatchSMC[i] = (Match1st || Match2nd) && OpQi:OpQV;
}
SmcHit =
      "STQ store is not a special memory access" &&
      ("self-modifying code detected by DEC
      (fetch/decode) unit" || MatchSMC[0] ||
      MatchSMC[1] || MatchSMC[2] ||
      MatchSMC[3] || MatchSMC[4]);
```

Instruction decoder **140** also traps self-modifying code using physical address tags. In particular, address match logic **444** and fetch control logic **447** of instruction decoder **140** receive portions of the linear and physical address (i.e., the StOp address) for the memory write which the store queue **159** is preparing to commit in stage 2

460. As before, stage 1 459 of store queue 159 provides bits 11-5 of the linear address `STQ_LinAddr(11,5)` and bits 19-12 of the physical address `STQ_PhysAddr(19,12)`. Address match logic 444 compares the StOp address against address tags 446 respectively associated with entries in instruction buffer 445. If a match is found, address match logic 444 triggers an SMC trap. Global control logic 260 flushes instruction decoder 140 and the fetch/decode process is restarted from the last committed instruction.

In the exemplary embodiment, an SMC trap is handled as follows. After all Ops associated with the triggering StOp are committed (i.e., the set of Ops decoded from the same x86 instruction as the triggering StOp or the entire Op quad in which the triggering StOp is a member, whichever is larger), Ops associated with subsequent x86 instructions are aborted. In the exemplary embodiment, the following emcode implements an SMC trap:

```
        =>
DfhSMC: RDSR4     _,_,_        //start two-step process to read SR4
        RDSR4     t7,_,_       //get faultPC
        LDK       t9,0x0030    //clear SSTF andDTF to reset any
        WRSR1     _,t9,_       //pending debug traps and,
        NoEretRetire           //especially, to also clear SMCTF
        =
DfhSMC1: RDSRO    t8,_,_       //get (fresh) copy of STCV bit
        EAND.cc   _,t8,0x20
        Brcc      DfhSMC1,EZFO,pt,SeqEret
                               //if STCV bit is (still) set,
                               //then keep waiting, else go jump to
        <=                     //faultPC (i.e. next mI)

SeqEret: WRIP     _,t7,_       //note,em. env. is invalid here
        ERET
        <=
```

The SMC trap emcode obtains the Extended Instruction Pointer (EIP) of the above aborted instruction. The SMC trap emcode then waits until the triggering StOp is acknowledged by the memory subsystem. In an alternative embodiment which includes an L2 cache interposed between the data cache 170 and main memory, the triggering StOp could instead be acknowledged by the L2 cache. In either case, such an acknowledgment means that a snoop to instruction cache 130 has already been issued. After the SMC trap emcode has synchronized with the memory write associated with the triggering StOp, it then jumps back (doing a WrIP) to fetch the next x86 instruction in the instruction stream. At this point it is guaranteed that the next bytes fetched from main memory (or alternatively from the L2 cache) will be up to date.

Even a StOp that does not trigger an SMC trap creates a window of time after the associated memory write is committed, but before a snoop is issued to instruction cache 130, during which any new instruction bytes fetched by instruction decoder 140 are potentially stale. To overcome this, fetch control logic 447 of instruction decoder 140 stores a copy of the physical address (i.e., the StOp address) associated with the committed memory write. Whenever instruction decoder 140 fetches new instruction bytes from instruction cache 130, fetch control logic 447 checks the current fetch address against its stored copy of StOp address for the last-committed memory write. If the current fetch address matches with the stored copy of the StOp address, then fetch control logic 447 nullifies the fetch. Fetch control logic 447 of instruction decoder 140 continues to reissue the same fetch address until the

committed StOp is acknowledged by the memory subsystem. When fetch control logic **447** receives an acknowledgment from the memory subsystem, it clears its StOp address store. In an alternative embodiment which includes an L2 cache interposed between the data cache **170** and main memory, the acknowledgment could be supplied instead by the L2 cache.

In the exemplary embodiment, the memory subsystem issues a snoop to instruction cache **130** before or (at the latest) concurrent with its StOp acknowledgment. While the instruction cache **130** is processing a snoop, it inhibits the processing of fetches from instruction decoder **140**. By inhibiting fetches during snoop processing, instruction cache **130** closes a second short window during which instruction fetches could potentially return stale bytes.

Each Op quad of scheduler **180** may contain bytes of decoded x86 instructions spanning two lines of instruction cache **130**. Similarly, an entry in instruction buffer **445** may span two lines of instruction cache **130**. In the exemplary embodiment, a line in the instruction decoder **140** is 32 bytes. This means that the physical address tags associated with each Op quad entry of scheduler **180** and with each entry of instruction buffer **445** need to encode addresses for both possible 32-byte cache lines. In one embodiment of address tags **446** and physical address tag fields **243**, a pair of complete physical addresses tags (bits 31:5) is stored for each Op quad entry of scheduler **180** and for each entry of instruction buffer **445**. However, to reduce hardware, while at the same time avoiding a high frequency of false matches, the exemplary embodiments of address tags **446** and physical address tag fields **243** store partial physical addresses, each containing bits 19:5 of the physical memory address of the associated x86 instruction (or instructions).

The exemplary embodiment supports single-cycle throughput of writes to memory. The data cache **170** is a write-back cache. When a memory write commitment associated with a StOp hits in data cache **170** and the line is found to be Owned or Dirty, then the write can be processed at a rate of 1 per-cycle. This situation presents some difficulty with respect to handling self-modifying code if an Owned/Dirty line is allowed to reside in both the data cache **170** and instruction cache **130**. In one embodiment, instruction cache **130** would have to be snooped immediately with the StOp being committed, which would add complexity since contention issues arise in access instruction cache **130** tag RAM. In addition, a dedicated address bus (not shown) would have to be sent from the data cache **170** to the instruction cache **130**. In order to minimize this complexity, while still maintaining mutual exclusion between the instruction cache **130** and the data cache **170**, cache control logic **160** prevents a cache line from residing in both caches at the same time in the exemplary embodiment. The estimated performance impact of this restriction is negligible.

One constraint imposed by this scheme is that a StOp cannot write to the instruction stream if the modified bytes are decoded into the same Op quad entry as the writing StOp and the StOp is older with respect to the modified bytes. However, a processor conforming to the x86 processor architecture must transfer control specification before starting to execute from the modified instruction stream. *See* Intel Pentium Processor, Software Reference Manual. In the exemplary embodiment, this requirement (if followed) eliminates the possibility that a StOp which stores into the instruction stream and the bytes that it writes will ever be in the same Op quad of scheduler **180**.

While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Many variations, modifications, additions, and improvements of the embodiments described are possible. For example, the organization of Op entries in scheduler **180** as Op quads is merely illustrative. Alternative embodiments may incorporate other structures and/or methods for representing the nature and state of operations in a computer having multiple and/or pipelined execution units. Furthermore, alternative embodiments may incorporate different hierarchies of memories and caches, for example L1 and L2 caches. In such alternative embodiments, store acknowledgments may be provided by an L2 cache.

Alternative embodiments may provide for a different distribution of structures and functionality, including structures for tag representation and comparison, among the scheduler **180**, the store unit **153**, the store queue **159**, and the instruction decoder **140**. Additionally, structures and functionality presented as hardware in the exemplary embodiment may be implemented as software, firmware, or microcode in alternative embodiments. A wide variety of computer system configurations are envisioned, each embodying self-modifying code handling in accordance with the present invention. For example, such a computer system (e.g., computer system **1000**) includes a processor **100** providing self-modifying code handling in accordance with the present invention, a memory subsystem (e.g., RAM **1020**), a display adapter **1010**, disk controller/adapter **1030**, various input/output interfaces and adapters (e.g., parallel interface **1009**, serial interface **1008**, LAN adapter **1007**, etc.), and corresponding external devices (e.g., display device **1001**, printer **1002**, modem **1003**, keyboard **1006**, and data storage). Data storage includes such devices as hard disk **1032**, floppy disk **1031**, a tape unit, a CD-ROM, a jukebox, a redundant array of inexpensive disks (RAID), a flash memory, etc. These and other variations, modifications, additions, and improvements may fall within the scope of the invention as defined in the claims which follow.

WE CLAIM:

1. In a computer having operation entries for representing operations in stages from instruction fetch to result commitment and having a store pipe for committing store operands to target addresses in memory, a self-modifying code handling system comprising:

a plurality of first tag stores respectively associated with a first group of the operation entries, the first tag stores representing first addresses in memory of instructions corresponding to the associated operation entries;

first comparison logic coupled to the first tag stores and to the store pipe, the first comparison logic supplying a self-modifying code indication in response to a match between the target address for a store operation being committed by the store pipe and any of the first addresses represented in the first tag stores; and

control logic coupled to the first comparison logic and to the operation entries, the control logic flushing uncommitted ones of the operation entries in response to the self-modifying code indication.

2. A self-modifying code handling system, as recited in claim 1:

wherein the first group of operation entries comprises a plurality of Op entries organized in Op groups represented in a scheduler, and

wherein the first tag stores each include a pair of tag fields covering memory addresses for a group of instructions from which the Op entries of the associated Op group decode, tag field pairs covering memory addresses on either side of a cache line boundary when the group of instructions crosses the cache line boundary.

3. A self-modifying code handling system, as recited in claim 2, wherein the first addresses represented in the pair of tag field are partial addresses, and wherein the first comparison logic supplies the self-modifying code indication in response to a match between any of the partial addresses represented in the tag fields and a corresponding portion of the target address for the store operation being committed by the store pipe.

4. A self-modifying code handling system, as recited in claim 1, wherein the first group and a second group of the operation entries are respectively associated with a scheduler and with an instruction decoder, the self-modifying code handling system further comprising:

a plurality of second tag stores respectively associated with ones of the second group of operation entries, the second tag stores representing second addresses in memory of instructions corresponding to the associated operation entries;

second comparison logic coupled to the second tag stores, to the store pipe, and to the control logic, the second comparison logic supplying the self-modifying code indication in response to a match between the target address for the store operation being committed by the store pipe and any of the addresses represented in the second tag stores;

wherein the control logic flushes the second group of operation entries and uncommitted ones of the first group of operation entries in response to the self-modifying code indication.

5. A self-modifying code handling system, as recited in claim 4:
wherein the first group of operation entries comprises a plurality of Op entries organized in Op groups represented in a scheduler,
wherein the first tag stores each include a pair of tag fields covering memory addresses for a group of instructions from which the Op entries of the associated Op group decode, tag field pairs covering memory addresses on either side of a cache line boundary when the group of instructions crosses the cache line boundary;
wherein the second group of operation entries comprises a plurality of instruction entries organized as an instruction buffer in the instruction decoder, each instruction buffer entry corresponding to a cache line; and
wherein the second addresses cover the cache line.

6. A self-modifying code handling system, as recited in claim 5,
wherein the first and second addresses are partial addresses,
wherein the first comparison logic supplies the self-modifying code indication in response to a match between any of the partial addresses represented in the tag fields and a corresponding portion of the target address, and
wherein the second comparison logic supplies the self-modifying code indication in response to a match between any of the partial addresses represented in the second tag stores and a corresponding portion of the target address.

7. A self-modifying code handling system, as recited in claim 2, further comprising:
an address store coupled to the store pipe to receive the target address for successive store operations, the address store being cleared in response to a store acknowledgment from a memory subsystem; and
fetch control logic coupled to the address store, the fetch control logic nullifying an instruction fetch from a current fetch address in response to a match between the current fetch address and the target address stored in the address store.

8. A self-modifying code handling system, as recited in claim 7, further comprising:
an instruction cache coupled between the instruction decoder and the memory subsystem, wherein the instruction cache inhibits processing of fetches from the instruction decoder while processing a snoop from the memory subsystem; and
a data cache coupled between the store pipe and the memory subsystem; and
instruction/data cache control logic for preventing a cache line from simultaneously residing in both the instruction cache and the data cache.

9. An apparatus comprising:
a memory subsystem;
instruction and data caches coupled to the memory subsystem;
a plurality of execution units including a store pipe coupled to the data cache to commit results of a (Store Op) to the memory subsystem, the store pipe supplying a StOp target address indication on commitment of a StOp result;
a scheduler including an ordered plurality of Op entries for Ops decoded from instructions and a corresponding plurality of first address tags covering memory addresses for the instructions;
first comparison logic coupled to the store pipe and to the first address tags, the first comparison logic coupled to trigger self-modifying code fault handling means in response to a match between the StOp target address and a one of the first address tags;
an instruction decoder coupled between the instruction cache and the scheduler, the instruction decoder including a plurality of instruction buffer entries and second address tags associated with the instruction buffer entries; and
second comparison logic coupled to the store pipe and to the second address tags, the second comparison logic coupled to trigger the self-modifying code fault handling means in response to a match between the StOp target address and a one of the second address tags.

10. An apparatus, as recited in claim 9, wherein the self-modifying code fault handling means comprises:
control logic coupled to the first and second comparison logic and to the scheduler and instruction decoder, the control logic flushing uncommitted ones of the Op from the Op entries and flushing instructions from the instruction buffer in response to a self-modifying code fault indication from either first or second comparison logic.

11. An apparatus, as recited in claim 10, wherein the self-modifying code fault handling means further comprises a self-modifying code fault handler for performing the following steps:
committing those Ops associated with the same instruction as the triggering StOp;
obtaining an instruction pointer for the triggering StOp;
waiting until the triggering StOp is acknowledged by the memory subsystem; and
jumping back in the instruction stream to an instruction immediately following the instruction associated with the triggering StOp.

12. An apparatus, as recited in claim 10, wherein the self-modifying code fault handling means further comprises:
an address store coupled to the store pipe to receive the target address for successive StOps, the address store being cleared in response to a StOp acknowledgment from the memory subsystem; and

fetch control logic coupled to the address store, the fetch control logic nullifying an instruction fetch from a fetch address by the instruction decoder in response to a match between the fetch address and the target address stored in the address store.

13. In a computer system including a plurality of execution units, a scheduler, an instruction decoder, a memory subsystem and instruction and data caches each coupled to the memory subsystem, a self-modifying code handling system characterized in that:

the plurality of execution units include a store pipe coupled to the data cache to commit results of a (Store Op) to the memory subsystem, the store pipe supplying a StOp target address indication on commitment of a StOp result;

the scheduler includes an ordered plurality of Op entries for Ops decoded from instructions and a corresponding plurality of first address tags covering memory addresses for the instructions;

first comparison logic is coupled to the store pipe and to the first address tags, the first comparison logic coupled to trigger self-modifying code fault handling means in response to a match between the StOp target address and a one of the first address tags;

the instruction decoder is coupled between the instruction cache and the scheduler, the instruction decoder including a plurality of instruction buffer entries and second address tags associated with the instruction buffer entries; and

second comparison logic is coupled to the store pipe and to the second address tags, the second comparison logic coupled to trigger the self-modifying code fault handling means in response to a match between the StOp target address and a one of the second address tags.

AMENDED CLAIMS

[received by the International Bureau on 26 February 1997 (26.02.97);
original claims 1-7,9,10,12 and 13 amended;
remaining claims unchanged (4 pages)]

1. In a computer having operation entries for representing operations in stages from instruction fetch to result commitment and having a store pipe for committing store operands to target addresses in memory, a self-modifying code handling system comprising:

a plurality of first tag stores respectively associated with a first group of the operation entries, the first tag stores for representing first addresses in memory of instructions corresponding to the associated operation entries;

first comparison logic coupled to the first tag stores and to the store pipe to supply a self-modifying code indication in response to a match between the target address for a store operation being committed by the store pipe and any of the first addresses represented in the first tag stores; and

control logic coupled to the first comparison logic and to the operation entries to flush uncommitted ones of the operation entries in response to the self-modifying code indication.

2. A self-modifying code handling system, as recited in claim 1:

wherein the first group of operation entries comprises a plurality of Op entries organized in Op groups represented in a scheduler, and

wherein the first tag stores each include a pair of tag fields covering memory addresses for a group of instructions from which the Op entries of the associated Op group decode, tag field pairs for covering memory addresses on either side of a cache line boundary when the group of instructions crosses the cache line boundary.

3. A self-modifying code handling system, as recited in claim 2, wherein the first addresses represented in the pair of tag field are partial addresses, and wherein the first comparison logic supplies the self-modifying code indication in response to a match between any of the partial addresses represented in the tag fields and a corresponding portion of the target address for the store operation being committed by the store pipe.

4. A self-modifying code handling system, as recited in claim 1, wherein the first group and a second group of the operation entries are respectively associated with a scheduler and with an instruction decoder, the self-modifying code handling system further comprising:

a plurality of second tag stores respectively associated with ones of the second group of operation entries, the second tag stores for representing second addresses in memory of instructions corresponding to the associated operation entries;

second comparison logic coupled to the second tag stores, to the store pipe, and to the control logic, the second comparison logic for supplying the self-modifying code indication in response to a match between the target address for the store operation being committed by the store pipe and any of the addresses represented in the second tag stores;

wherein the control logic flushes the second group of operation entries and uncommitted ones of the first group of operation entries in response to the self-modifying code indication.

5. A self-modifying code handling system, as recited in claim 4:
wherein the first group of operation entries comprises a plurality of Op entries organized in Op groups represented in a scheduler,
wherein the first tag stores each include a pair of tag fields for covering memory addresses for a group of instructions from which the Op entries of the associated Op group decode, tag field pairs covering memory addresses on either side of a cache line boundary when the group of instructions crosses the cache line boundary;
wherein the second group of operation entries comprises a plurality of instruction entries organized as an instruction buffer in the instruction decoder, each instruction buffer entry corresponding to a cache line; and
wherein the second addresses cover the cache line.

6. A self-modifying code handling system, as recited in claim 5,
wherein the first and second addresses are partial addresses,
wherein the first comparison logic supplies the self-modifying code indication in response to a match between any of the partial addresses represented in the tag fields and a corresponding portion of the target address, and
wherein the second comparison logic supplies the self-modifying code indication in response to a match between any of the partial addresses represented in the second tag stores and a corresponding portion of the target address.

7. A self-modifying code handling system, as recited in claim 2, further comprising:
an address store coupled to the store pipe to receive the target address for successive store operations, the address store cleared in response to a store acknowledgment from a memory subsystem; and
fetch control logic coupled to the address store to nullify an instruction fetch from a current fetch address in response to a match between the current fetch address and the target address stored in the address store.

8. A self-modifying code handling system, as recited in claim 7, further comprising:
an instruction cache coupled between the instruction decoder and the memory subsystem, wherein the instruction cache inhibits processing of fetches from the instruction decoder while processing a snoop from the memory subsystem: and
a data cache coupled between the store pipe and the memory subsystem; and
instruction/data cache control logic for preventing a cache line from simultaneously residing in both the instruction cache and the data cache.

9. An apparatus comprising:

a memory subsystem;

instruction and data caches coupled to the memory subsystem;

a plurality of execution units including a store pipe coupled to the data cache to commit a result of a store operation (StOp) to the memory subsystem, wherein the store pipe supplies a StOp target address indication on commitment of the StOp result;

a scheduler including an ordered plurality of Op entries for Ops decoded from instructions and a corresponding plurality of first address tags for covering memory addresses for the instructions;

first comparison logic coupled to the store pipe and to the first address tags to trigger self-modifying code fault handling means in response to a match between the StOp target address and a one of the first address tags;

an instruction decoder coupled between the instruction cache and the scheduler, the instruction decoder including a plurality of instruction buffer entries and second address tags associated with the instruction buffer entries; and

second comparison logic coupled to the store pipe and to the second address tags to trigger the self-modifying code fault handling means in response to a match between the StOp target address and a one of the second address tags.

10. An apparatus, as recited in claim 9, wherein the self-modifying code fault handling means comprises:

control logic coupled to the first and second comparison logic and to the scheduler and instruction decoder, the control logic for flushing uncommitted ones of the Op from the Op entries and flushing instructions from the instruction buffer in response to a self-modifying code fault indication from either first or second comparison logic.

11. An apparatus, as recited in claim 10, wherein the self-modifying code fault handling means further comprises a self-modifying code fault handler for performing the following steps:

committing those Ops associated with the same instruction as the triggering StOp;

obtaining an instruction pointer for the triggering StOp;

waiting until the triggering StOp is acknowledged by the memory subsystem; and

jumping back in the instruction stream to an instruction immediately following the instruction associated with the triggering StOp.

12. An apparatus, as recited in claim 10, wherein the self-modifying code fault handling means further comprises:

an address store coupled to the store pipe to receive the target address for successive StOps, the address store being cleared in response to a StOp acknowledgment from the memory subsystem; and

fetch control logic coupled to the address store, the fetch control logic for nullifying an instruction fetch from a fetch address by the instruction decoder in response to a match between the fetch address and the target address stored in the address store.

13. In a computer system including a plurality of execution units, a scheduler, an instruction decoder, a memory subsystem and instruction and data caches each coupled to the memory subsystem, a self-modifying code handling system characterized in that:

the plurality of execution units include a store pipe coupled to the data cache to commit a result of a store operation (StOp) to the memory subsystem, wherein the store pipe supplies a StOp target address indication on commitment of the StOp result;

the scheduler includes an ordered plurality of Op entries for Ops decoded from instructions and a corresponding plurality of first address tags covering memory addresses for the instructions;

first comparison logic is coupled to the store pipe and to the first address tags, the first comparison logic coupled to trigger self-modifying code fault handling means in response to a match between the StOp target address and a one of the first address tags;

the instruction decoder is coupled between the instruction cache and the scheduler, the instruction decoder including a plurality of instruction buffer entries and second address tags associated with the instruction buffer entries; and

second comparison logic is coupled to the store pipe and to the second address tags, the second comparison logic coupled to trigger the self-modifying code fault handling means in response to a match between the StOp target address and a one of the second address tags.

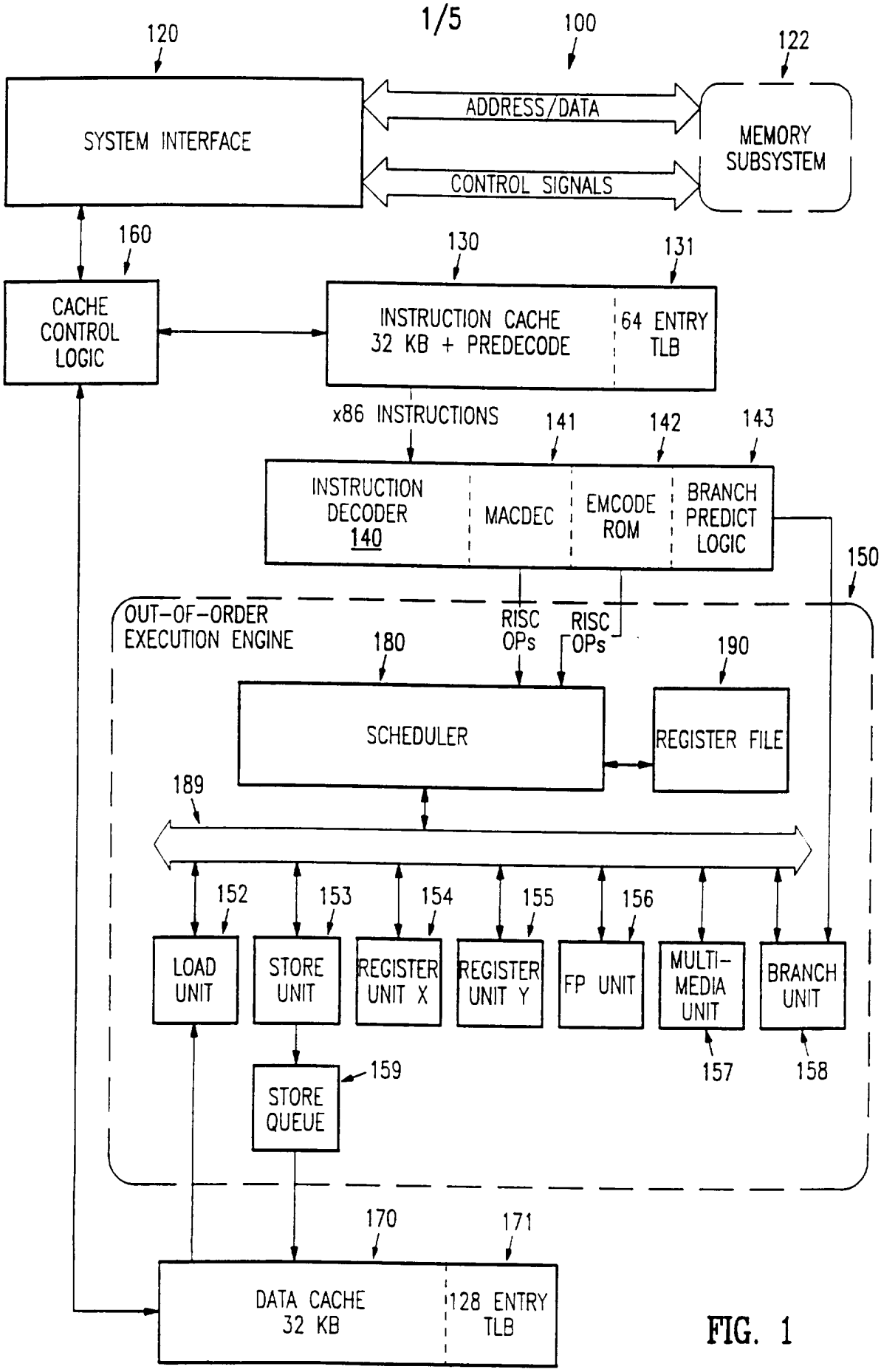

120
100
122
SYSTEM INTERFACE
ADDRESS/DATA
CONTROL SIGNALS
MEMORY SUBSYSTEM
160
130
131
CACHE CONTROL LOGIC
INSTRUCTION CACHE 32 KB + PREDECODE
64 ENTRY TLB
x86 INSTRUCTIONS
141
142
143
INSTRUCTION DECODER 140
MACDEC
EMCODE ROM
BRANCH PREDICT LOGIC
150
OUT-OF-ORDER EXECUTION ENGINE
RISC OPs
RISC OPs
180
190
SCHEDULER
REGISTER FILE
189
152
153
154
155
156
LOAD UNIT
STORE UNIT
REGISTER UNIT X
REGISTER UNIT Y
FP UNIT
MULTI-MEDIA UNIT
BRANCH UNIT
157
158
159
STORE QUEUE
170
171
DATA CACHE 32 KB
128 ENTRY TLB

FIG. 1

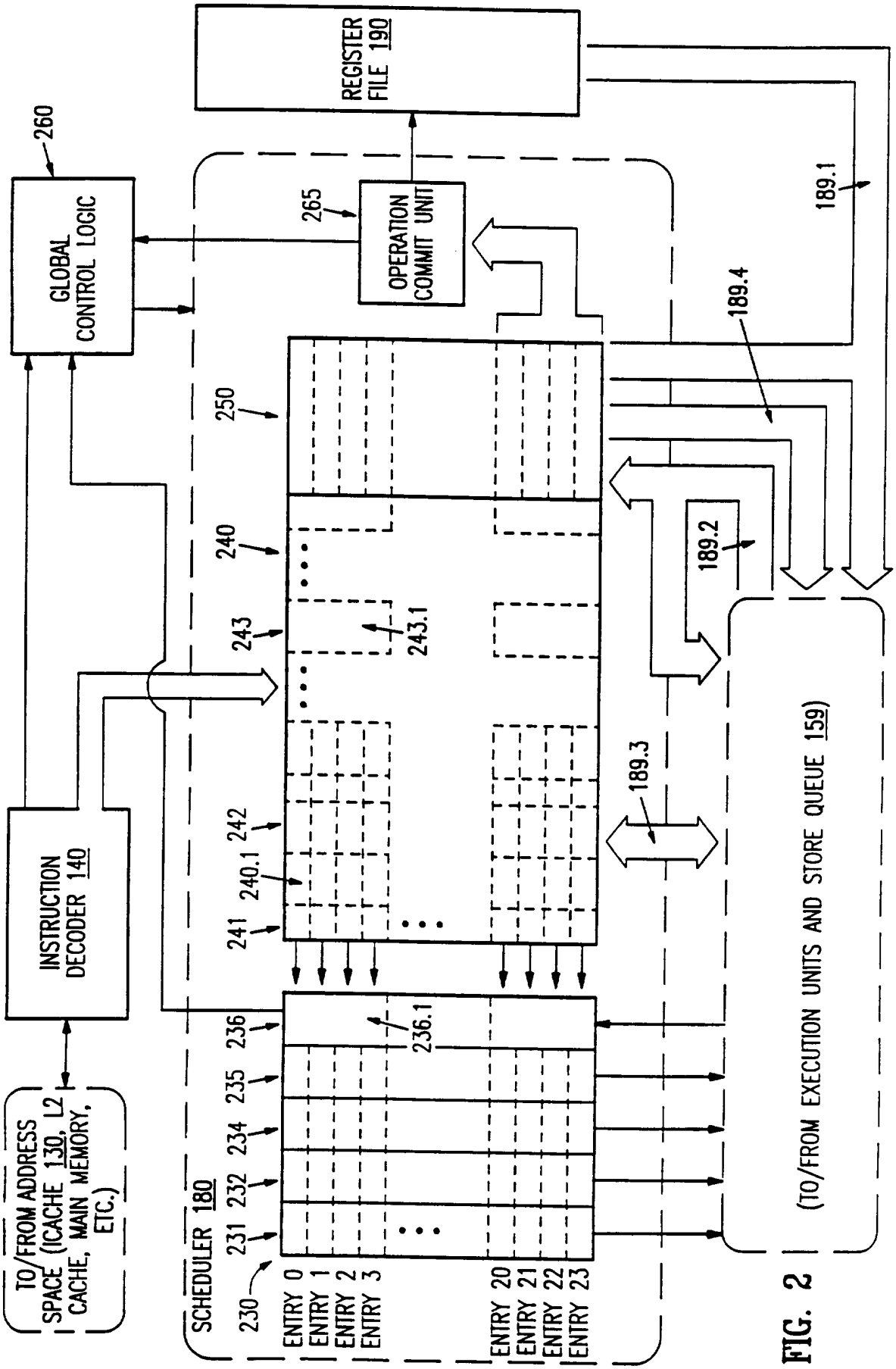

REGISTER FILE 190
260
GLOBAL CONTROL LOGIC
265
OPERATION COMMIT UNIT
189.1
189.4
250
240
243
243.1
189.2
189.3
242
240.1
241
INSTRUCTION DECODER 140
TO/FROM ADDRESS SPACE (ICACHE 130, L2 CACHE, MAIN MEMORY, ETC.)
SCHEDULER 180
230
231
232
234
235
236
236.1
ENTRY 0
ENTRY 1
ENTRY 2
ENTRY 3
ENTRY 20
ENTRY 21
ENTRY 22
ENTRY 23
(TO/FROM EXECUTION UNITS AND STORE QUEUE 159)

FIG. 2

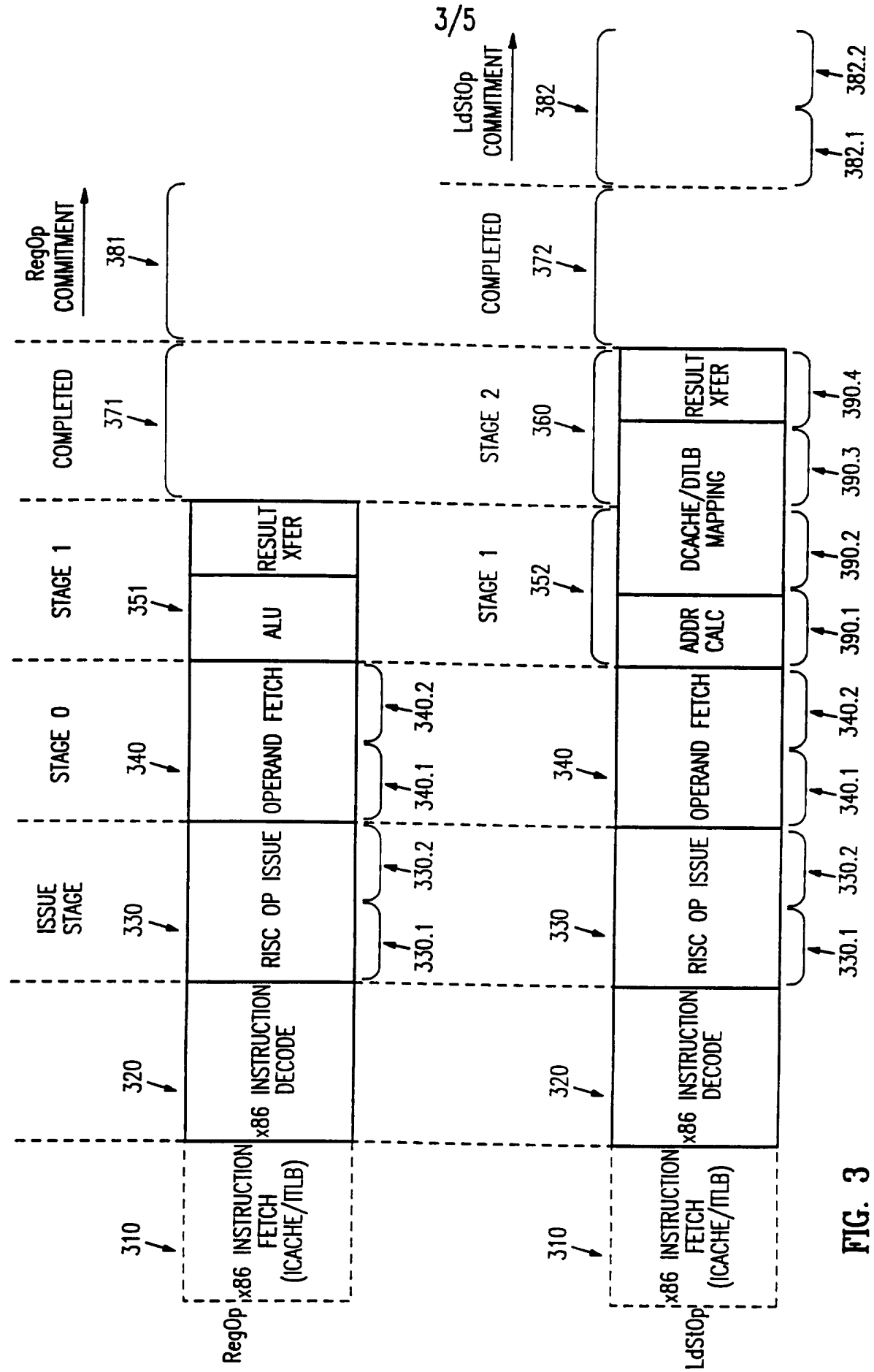

RegOp
LdStOp
310
x86 INSTRUCTION FETCH (ICACHE/ITLB)
320
x86 INSTRUCTION DECODE
ISSUE STAGE
330
RISC OP ISSUE
330.1
330.2
STAGE 0
340
OPERAND FETCH
340.1
340.2
STAGE 1
351
ALU
RESULT XFER
COMPLETED
371
RegOp COMMITMENT
381
STAGE 1
352
ADDR CALC
STAGE 2
360
DCACHE/DTLB MAPPING
390.1
390.2
390.3
390.4
COMPLETED
372
LdStOp COMMITMENT
382
382.1
382.2

FIG. 3

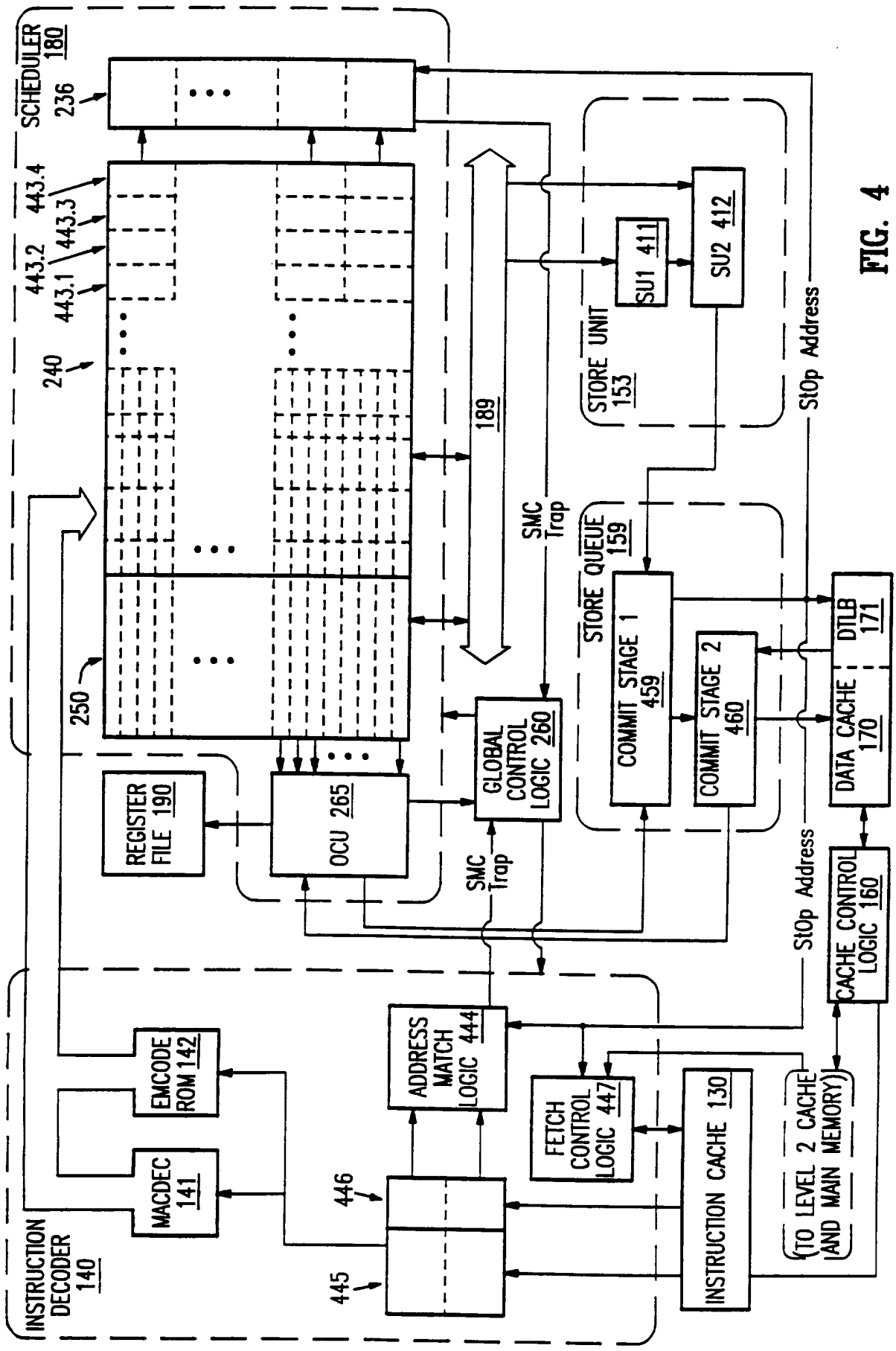

SCHEDULER 180
236
443.4
443.3
443.2
443.1
240
250
189
SU1 411
SU2 412
STORE UNIT 153
StOp Address
SMC Trap
STORE QUEUE 159
COMMIT STAGE 1 459
COMMIT STAGE 2 460
DTLB 171
DATA CACHE 170
CACHE CONTROL LOGIC 160
GLOBAL CONTROL LOGIC 260
REGISTER FILE 190
OCU 265
SMC Trap
StOp Address
ADDRESS MATCH LOGIC 444
EMCODE ROM 142
MACDEC 141
INSTRUCTION DECODER 140
446
445
FETCH CONTROL LOGIC 447
INSTRUCTION CACHE 130
TO LEVEL 2 CACHE AND MAIN MEMORY

FIG. 4

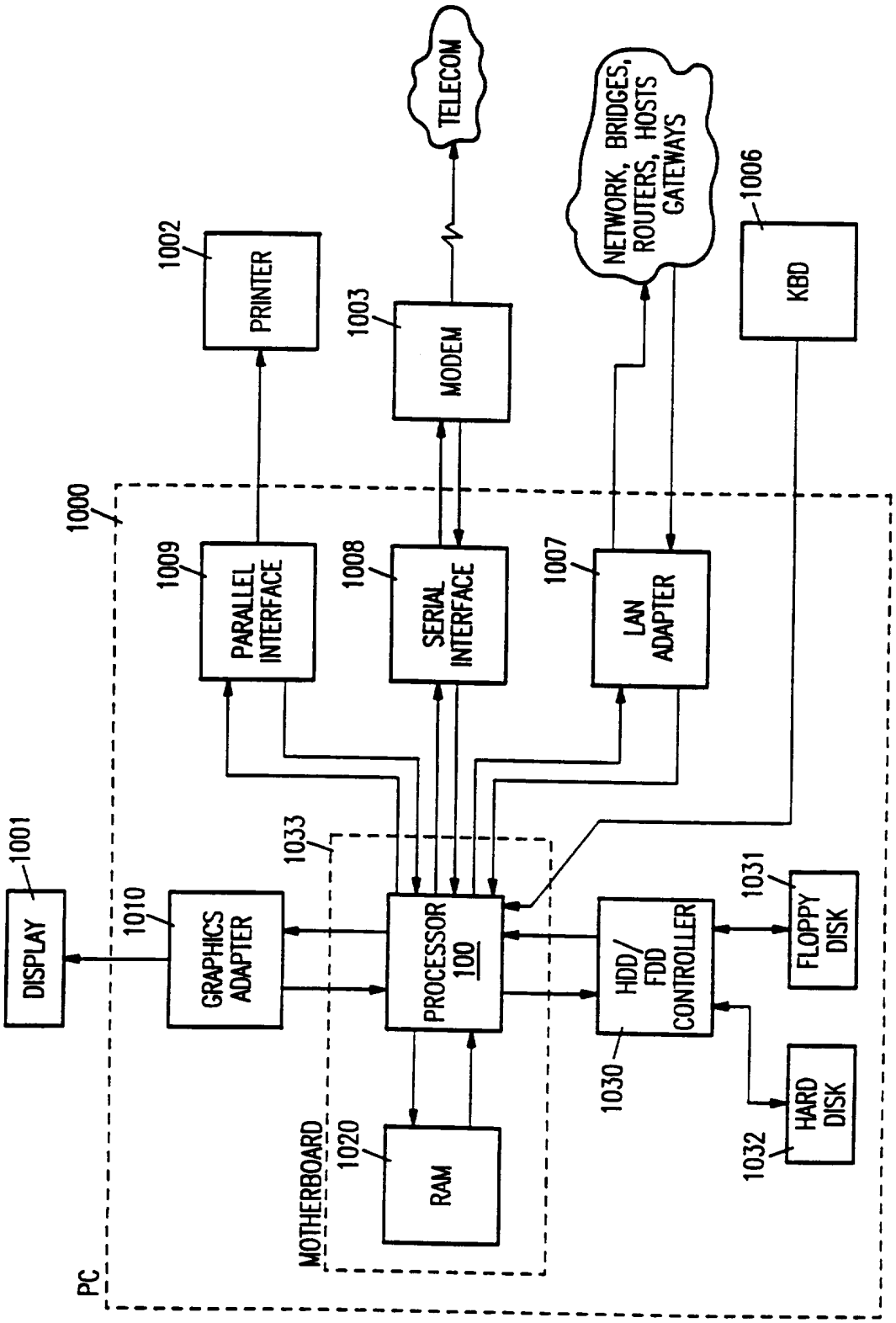

TELECOM
NETWORK, BRIDGES, ROUTERS, HOSTS GATEWAYS
1002
PRINTER
1003
MODEM
1006
KBD
1000
1009
PARALLEL INTERFACE
1008
SERIAL INTERFACE
1007
LAN ADAPTER
1001
DISPLAY
1010
GRAPHICS ADAPTER
1033
PROCESSOR 100
HDD/ FDD CONTROLLER
1031
FLOPPY DISK
1030
1032
HARD DISK
MOTHERBOARD
1020
RAM
PC

FIG. 5

INTERNATIONAL SEARCH REPORT

In ional Application No
PCT/US 96/15420

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/38

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EP,A,0 159 712 (HONEYWELL INF SYSTEMS) 30 October 1985 ---	1,4,7,8, 12
A	US,A,5 434 987 (ABRAMSON JEFFREY M ET AL) 18 July 1995 see summary; column 7, line 16 - column 10, line 8 ---	1,3,5,6
A	IBM TECHNICAL DISCLOSURE BULLETIN, vol. 10, no. 2, July 1967, pages 125-126, XP000615113 ANDERSON ET AL.: "Instruction prefetching interlock" see the whole document -----	1,3,4,7, 9,13

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search	Date of mailing of the international search report
20 December 1996	08. 01. 97
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax (+31-70) 340-3016	Authorized officer Klocke, L

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

In ational Application No
PCT/US 96/15420

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP-A-0159712	30-10-85	AU-B- 572647	12-05-88
		AU-A- 4173385	31-10-85
		CA-A- 1243411	18-10-88
US-A-5434987	18-07-95	NONE	