PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 4 :		(11) International Publication Number:
G06F 15/16	A1	WO 89/ 00734
		(43) International Publication Date: 26 January 1989 (26.01.89)

(21) International Application Number: PCT/US88/02304

(22) International Filing Date: 8 July 1988 (08.07.88)

(31) Priority Application Number: 076,116

(32) Priority Date: 21 July 1987 (21.07.87)

(33) Priority Country: US

(71) Applicant: STELLAR COMPUTER INC. [US/US]; 75 Wells Avenue, Newton, MA 02159 (US).

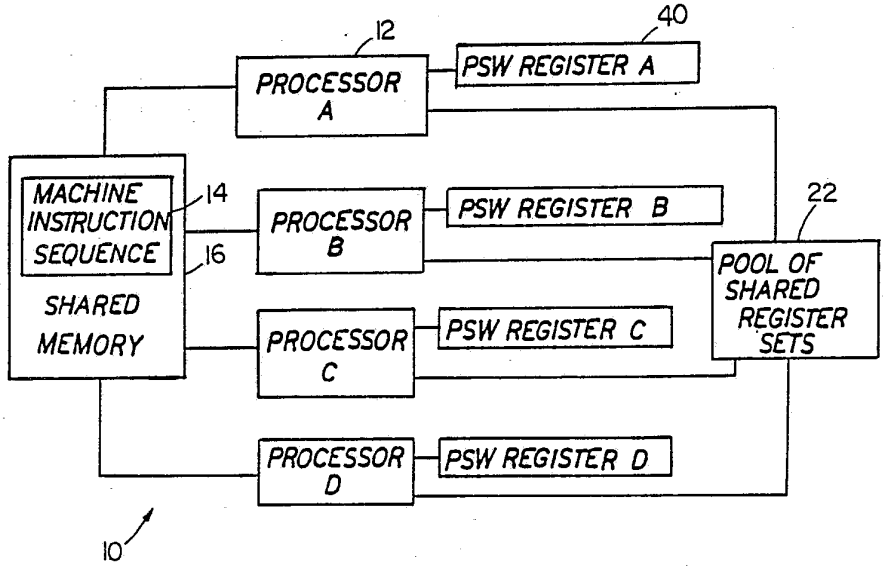
(72) Inventors: TEIXEIRA, Thomas, J. ; 89 Woodside Road, Harvard, MA 01451 (US). SMITH, Maxim, G. ; 31 Ridge Avenue, Natick, MA 01760 (US).

(74) Agent: FEIGENBAUM, David, L.; Fish & Richardson, One Financial Center, Suite 2500, Boston, MA 02111-2658 (US).

(81) Designated States: AT (European patent), AU, BB, BE (European patent), BG, BJ (OAPI patent), BR, CF (OAPI patent), CG (OAPI patent), CH (European patent), CM (OAPI patent), DE (European patent), DK, FI, FR (European patent), GA (OAPI patent), GB (European patent), HU, IT (European patent), JP, KP, KR, LK, LU (European patent), MC, MG, ML (OAPI patent), MR (OAPI patent), MW, NL (European patent), NO, RO, SD, SE (European patent), SN (OAPI patent), SU, TD (OAPI patent), TG (OAPI patent).

Published
With international search report.

(54) Title: DETECTING MULTIPLE PROCESSOR DEADLOCK



(57) Abstract

The simultaneous work of multiple processors (12) at different places in a machine instruction sequence (14) (of a kind which includes points at each of which processing may need to wait for occurrence of a predetermined event) is controlled by marking each point by a conditional branch which, when the predetermined event has not yet occurred, causes the processor (12) executing the branch to enter and repeat a loop including the conditional branch; and causes the processing to continue outside the loop only when the event has occurred; a processor (12) is determined to be waiting when it is executing such a conditional branch. In other aspects, there are a plurality of different types of events that may be operated on by the conditional instruction; and an idle (unassigned) waiting condition of a processor (12) is detected separately from a non-idle waiting condition.

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.

AT	Austria	**FR**	France	**ML**	Mali
AU	Australia	**GA**	Gabon	**MR**	Mauritania
BB	Barbados	**GB**	United Kingdom	**MW**	Malawi
BE	Belgium	**HU**	Hungary	**NL**	Netherlands
BG	Bulgaria	**IT**	Italy	**NO**	Norway
BJ	Benin	**JP**	Japan	**RO**	Romania
BR	Brazil	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SN**	Senegal
CH	Switzerland	**LI**	Liechtenstein	**SU**	Soviet Union
CM	Cameroon	**LK**	Sri Lanka	**TD**	Chad
DE	Germany, Federal Republic of	**LU**	Luxembourg	**TG**	Togo
DK	Denmark	**MC**	Monaco	**US**	United States of America
FI	Finland	**MG**	Madagascar		

Detecting Multiple Processor Deadlock

Background of the Invention

This invention relates to detecting deadlocks between multiple processors that are simultaneously executing one or more different parts of a machine instruction sequence.

One of the processors may reach a point in execution where it must wait until the occurrence of some condition, for example the completion of a calculation of a data value by another processor. It is known to cause the processor to wait by having the compiler insert at that point in the instruction sequence a special purpose "wait" instruction which has the effect of preventing the processor from proceeding until the needed data value is calculated. One way to indicate to the waiting processor when it may proceed is to have the other processor clear a bit in memory when it has completed the needed calculation; the waiting processor tests the bit and if it finds the bit set, waits until it becomes clear; when the bit becomes clear, the waiting process proceeds to the next portion of the instruction sequence.

A situation in which every processor is waiting for a condition to be satisfied and there is no other work available to be done within the instruction sequence is called a hard deadlock, and reflects an error in the algorithm underlying the machine instruction sequence. Such hard deadlocks cannot be resolved.

By contrast, a so-called soft deadlock, in which other available work remains to be done elsewhere in the instruction sequence, may be resolved by having one of the waiting processors proceed to that other work.

In one known scheme for detecting the existence of a soft deadlock, the computer includes special purpose logic hardware that can detect when a number of processors are simultaneously waiting and then can interrupt the operating system, which deals with the deadlock.

Summary of the Invention

A general feature of the invention provides a method for controlling the simultaneous work of multiple processors at different places in a machine instruction sequence of the kind which includes points at each of which processing may need to wait for the occurrence of a predetermined event; a conditional branch is included at each such point in the instruction sequence which, when the event has not yet occurred, causes the processor executing the branch to enter and repeat a loop including the conditional branch, and causes the processing to continue outside the loop only when the event has occurred; a processor is identified as waiting at one of the points when it is executing such a conditional branch.

Another general feature of the invention is that a plurality of different types of events (e.g., including the setting or clearing of a bit) may be operated on by the conditional instruction.

Preferred embodiments include the following features. The conditional branch has a single conditional branch instruction that branches to itself as long as the predetermined event has not occurred. The conditional branch instruction is of the kind that is also used as part of the machine instruction sequence for purposes other than to indicate that the processor is waiting. When a group of processors whose processing activities are currently interrelated are all identified

as waiting, at least one of the waiting processors is reassigned to another place in the machine instruction sequence. The group of processors may include fewer than all of the multiple processors, for example all processors working on a single process. The machine instruction sequence includes an operating system capable of interrupting the activities of each processor from time to time to reassign it to another place in the machine instruction sequence; processors that are waiting are interrupted before processors that are not waiting. The step of identifying when a group of processors are all waiting is performed by an individual processor, so that the identifying and reassigning may not require interruption of other processors.

An other general feature of the invention is a method for determining the status of one of a plurality of processors simultaneously working on a machine instruction sequence, which includes separately detecting (i) a first condition in which a processor has been assigned to work at a location within said machine instruction sequence but is waiting to proceed until the occurrence of a predetermined event, and (ii) a second condition in which an idle processor has not been assigned to work at any location within said machine instruction sequence.

Preferred embodiments include the following features. A value is stored corresponding to each processor indicating whether it is assigned to work at any location, and the first and second conditions are detected based on the value. The operating system interrupts a processor in the second condition before interrupting any processor in the first condition.

The invention provides a simple, versatile, effective method for indicating that a processor is

waiting, or detecting deadlocks among processors. No special new instruction needs to be added to the machine instruction set executable by the processors. Each processor determines immediately when the condition on which it is waiting has been satisfied. Only a single instruction, rather than several instructions, needs to be executed in order to maintain the processor in the waiting condition. The same instruction performs the functions of both causing the processor to wait, and providing the indication to the processor hardware that a waiting condition exists. Deadlocks can be determined on the basis of only some group of the processors being in a waiting condition, for example processors working on a single process. Idle processors can be identified and treated differently from non-idle waiting processors. A wide variety of conditions can be used to trigger the waiting condition.

Other advantages and features will become apparent from the following description of the following embodiment, and from the claims.

Description of the Preferred Embodiment

We first briefly describe the drawings.

Fig. 1 is a block diagram of a multiple processor computer.

Fig. 2 is a diagram of parallel regions and blocks in a machine instruction sequence.

Fig. 3 is a diagram of shared register sets.

Fig. 4 is a diagram of a portion of a program status word.

Fig. 5 is a conditional branch instruction.

Figs. 6, 7 are block diagrams of portions of the circuitry of one of the processors.

Fig. 8 is a block diagram of deadlock detection logic.

Fig. 9 is an example of a portion of a machine instruction sequence.

Structure and Operation

Referring to Fig. 1, in one example of a multiple processor computer 10, four processors 12 (labeled respectively A, B, C, D) are available to execute the machine instruction sequence 14 held in a shared memory 16.

Referring to Fig. 2, machine instruction sequence 14 may include one or more parallel regions 18 of machine instructions (representative parallel regions are labeled respectively W, X, Y, Z). A given parallel region X has 2 or more blocks 20 of machine instructions (representative blocks are labeled Q, R, S, T) which are independent in the sense that the same result is obtained whether one processor executes all blocks in the parallel region, or different processors execute different blocks (e.g., processor A executes block Q, C executes R, and D executes S and T).

Computer system 10 (Fig. 1) can execute from one to four processes simultaneously. (A process, as commonly defined, is a sequence of machine instructions together with information about the state of its execution). Each process may be capable of being split up for execution among more than one of the processors; the portion of a process executing on a given processor at a given time may be called a thread. Threads enter and leave parallel regions from time to time. A means (e.g., one of the processors) for executing the instructions in a thread is called a stream.

It may be necessary at various points within a thread to pause until another thread completes a calculation before proceeding. A thread is said to be waiting when it reaches such a point and the other

thread has not yet completed the calculation. A stream (processor) is said to be waiting if it is currently executing instructions of a thread which is waiting. A deadlock occurs when a set of streams (processors) are executing interrelated threads which are all waiting. The invention provides a technique (described later) for detecting such a deadlock.

Referring again to Fig. 1, in order to regulate the work of the four processors within the machine instruction sequence 14, computer system 10 includes (as a shared resource) a pool 22 of high speed register sets shared in common by the four processors.

Referring to Fig. 3, pool 22 includes as many register sets 24 as there are processors (in this case four, labeled respectively 0, 1, 2, 3). All of the register sets 24 have the same number (e.g, two in this example) of 32-bit registers 26. The two registers in a set are called concurrency registers 0 and 1, i.e., CR0 and CR1. Any of the four register sets can be dynamically assigned to a processor as it enters a parallel region 18, as explained below.

As a processor enters a parallel region it is assigned either a currently unused register set (if no processor is actively working in the region) or the register set already assigned to processors working in the region being entered. As a processor leaves a region the assignment of the register set to it is terminated. As long as at least one processor is actively working in a parallel region, the register set assigned to that processor is, of course, unavailable for reassignment. However, when no processor is currently active within a region, the register set previously associated with that region is free to be dynamically reassigned for use by other processors.

The assignment and reassignment of register sets to processors is accomplished dynamically by the processors themselves in the course of (and without interrupting) the normal execution of the machine instruction sequence. To accomplish this, the compiler inserts assignment instructions at appropriate locations in the machine instruction sequence.

There are several types of assignment instructions. One type of assignment instruction enables a processor to find an available currently unused register set and assign that set to itself, or to assign itself to the same register set currently in use by another processor so that those two processors can share the information in the register set. Another allocation instruction causes a processor to terminate the assignment of a register set to that processor. When all processors to which a register set had been assigned terminate the assignment, the register set becomes freed for reassignment. Because all of the register sets have the same configuration it is irrelevant which particular register set becomes assigned to the processors working in a region at a given time. The dynamic assignment and reassignment reduces overhead cost and permits a relatively small number of register sets to be used. The dynamic assignment of shared register sets and the use of parallel regions is described more fully in United States Patent Applications S.N. 034,084 and S.N. 034,166, filed April 2, 1987, and assigned to the same assignee as this application.

Referring again to Fig. 1, each processor has an associated non-shared PSW register 40 which holds a program status word (PSW). The PSW includes information pertinent to the thread currently executing on the

associated processor.

Referring to Fig. 4, the PSW 42 has both a protected (unaccessible to the user) portion 44 and an unprotected portion 46. The protected portion 44 includes: a one-bit parallel region valid (PRV) field 48 (which is set when the CRI and PID fields, described below, are valid); a two-bit concurrency register indicator (CRI) field 50 which identifies the register set (0, 1, 2, or 3) assigned to this processor (CRI is valid only if PRV is set); a two-bit process identifier (PID) field 52 (which identifies the process on which the processor is working); and a WAIT bit 53 (indicating whether the processor is currently waiting).

Each of the four (or fewer) processes which may be executing concurrently on computer system 10 is assigned a unique PID. The PSWs of all streams presently executing on behalf of a given process contain the PID of that process. A wide variety of possible combinations of processes and threads executing in parallel regions may occur.

The invention makes use of the contents of the PSWs for detecting deadlocks.

Two steps are required to detect a deadlock condition. The first is to determine when a given processor is waiting. The second is to determine whether the waiting condition of one or more processors indicates a deadlock.

At each place in the machine instruction sequence where processing should wait until some condition is met, the compiler inserts one of the available standard conditional branch instructions that are part of the set of machine instructions which are normally used to define the computational algorithm, and are executable by each processor 12. The inserted

conditional branch instruction is arranged in such a way that it both causes the processor that executes it to wait and enables detection of the waiting condition.

Referring to Fig. 5, suppose the conditional branch instruction 88 that is inserted in the machine instruction sequence is identified by an instruction number 90 (denoted xx) in the instruction sequence. Execution of the conditional branch instruction 88 by a processor involves, in a conventional manner, first testing whether a certain condition Y is met. But, unlike the typical use of a conditional branch as part of a computational algorithm, if the condition is met (89), the processor is directed back to and reexecutes the same conditional branch instruction xx. Thus the processor enters a tight (one instruction) loop where it (in effect) waits until condition Y is no longer met. When condition Y is not met, the processor falls out of the loop by proceeding (90) to the next instruction in the machine instruction sequence 14.

Referring to Fig. 6, each processor 12 has circuitry for detecting the waiting condition of that processor, including a detector 92 which receives signals representative of every instruction 94 to be executed by processor 12. Detector 92 identifies any conditional branch instruction that branches to itself in response to the existence of a given condition. When such a wait condition (conditional branch instruction) is identified, detector 92 signals a circuit 95 to set the WAIT bit 53 in the associated processor's PSW 40, indicating that the processor is waiting.

Once the PSW 40 WAIT bit has been set, it is cleared again as soon as detector 92 detects the execution of an instruction other than a conditional branch that branches to itself. This could occur, for

example, if the condition causing the waiting is satisfied, or if (after a deadlock is detected) processor 12 is reassigned to another place in the machine instruction sequence.

The detectors 92 for all processors 12 together assure that, at any given time, the WAIT bits 53 of the four PSWs 40 indicate the waiting status of all four processors. Deadlock determinations are based on the WAIT bits in the following way.

Referring to Fig. 7, the PID values and the WAIT bits of the four PSWs for processors A, B, C, and D are all delivered to logic 96 which analyses them to determine the existence of a deadlock; when a deadlock is detected logic 96 issues a deadlock signal 97 that calls the operating system 98 (stored in shared memory 16, Fig. 1) and also provides operating system 98 with information about the waiting processors. Operating system 98 then responds to the deadlock signal, for example by reassigning at least one of the processors to another place in the machine instruction sequence.

Logic 96 declares a deadlock in accordance with the following rule. If all processors having the same PID i.e., all processors working on a given process are waiting, deadlock is declared. For example, suppose three processors (E, F, G) are working respectively on executing blocks U, V, and W, respectively, of a given process P. Suppose that processors E and F have reached points in blocks U and V where they must wait for data values to be computed in block W by processor G, and those values have not yet been computed. Then, while processors E and F wait, no deadlock condition is detected, because another processor, G, which is also assigned to process P, is not waiting. Suppose that, due to other demands on computational resources, the

operating system removes processor G from working on process P and reassigns it to an unrelated process. Now all processors (E and F) assigned to process P are waiting; and logic 96 detects and signals a deadlock condition. In response to the deadlock condition, the operating system determines that either processor E or F would be better assigned to executing block W which is currently without a processor, than waiting in block E or F. Therefore, the operating system reassigns processor E, say, to executing block W. Then the deadlock condition no longer exists.

Referring to Fig. 8, logic 96 includes four inequality comparators 110, 112, 114, 116 each of which has a J input connected to receive the PID bits (e.g., PID_A) of one if the four PSWs (of processor A, B, C, or D). The other, K, input of each comparator receives (on a line 118) the PID bits (called PID_i) of the PSW of any processor which enters a waiting condition (i.e., is executing an instruction that branches to itself). Thus each time processor i (the current processor) executes an instruction that branches to itself, it applies its PID bits on line 110. The outputs of the comparators are fed respectively to OR gates 120, 122, 124, 126. The other inputs of the OR gates are connected to receive respectively the WAIT bits from the PSWs of the four processors. The outputs of the four OR gates are connected to an AND gate 128 whose output represents the existence or non-existence of a deadlock. Deadlock is indicated at the output of AND gate 128 if all either processors are waiting or if any processor that is not waiting has a different PID (i.e., is executing in a different process) from the current processor's PID appearing on line 118. The output of AND gate 128 is delivered to the current processor whose

PID is on line 118. The output of AND gate 128 does not specify whether a deadlock is hard or soft. That is determined by the operating system.

When a deadlock condition is signaled to the current processor (i), that processor stops executing the instruction that branches to itself and begins to execute operating system code whose purpose is to resolve the deadlock.

Operating system 98 handles a deadlock by interrupting only those processors involved in the deadlock, while allowing the non-waiting processors to continue execution. The operating system code determines which process P was involved in the deadlock condition by noticing which process was running at the time the deadlock occurred based on the PID bits of processor i. It then consults a scheduling table that indicates which processors are assigned to the different blocks of various processes to determine whether any blocks of process P where work remains to be done have no assigned processor.

If any such blocks of process P have no assigned processor, the operating system reassigns one or more processors to those blocks from the waiting blocks. If the deadlock was soft, then eventually (possibly after a few more deadlocks) a block of instructions where progress may be made will have a processor assigned to it, and that block may generate values that other blocks await, thus resolving the soft deadlock.

On the other hand, if all blocks of process P are awaiting other calculations, then repeated deadlocks will continue to occur, no matter what combination of processors are assigned to process P's blocks. The operating system keeps track of the number of deadlock

signals generated by each process per unit time. When the number exceeds a configurable parameter, the operating system declares that process P is in a hard deadlock, and terminates execution of process P.

Note that a deadlock is detected and handled whenever the processors working on a given process are deadlocked even though other processors working or other processes may not be deadlocked. Furthermore, deadlocks can be detected with respect to two different processes at the same time, that is a deadlock between processors A and B working on process P may be detected at the same time that a deadlock between processors C and D working on process Q is detected. Thus the deadlock detected is relatively fine-grained.

Referring again to Fig. 5, the condition on which instruction 88 branches to itself may be any one of a wide variety of conditions (not limited to the set or clear condition of a particular bit) including conditions set up by other instructions inserted in the machine instruction sequence by the compiler.

Referring to Fig. 9, in one simple example, if at a given point 100 in the machine instruction sequence, processing should wait until a data value M has been calculated, then a bit N in one of the concurrency registers 26 (Fig. 3) may be set by an instruction 102 placed by the compiler immediately after the instruction 104 where M is calculated. A conditonal branch instruction 106 causes bit N to be tested and, if not set, branches to itself; and otherwise proceeds.

Because of the versatility of typically available conditional branch instructions, a variety of synchronization functions can be implemented in a simple way by triggering waits on carefully constructed conditions (not limited to the set or clear condition of

a single bit).

For example, a readers/writers lock could be implemented by the following sets of instructions. Suppose that a word (LOCK) stored in memory includes low order bits (NUMBER) that represent the number of processors that are either reading or waiting to read at a given storage location (LOCATION) and a high order lock bit (BIT) that indicates whether a processor is writing at LOCATION. The goal is to assure that no processor reads until there are no other processors writing at LOCATION, and that no processor is writing at LOCATION if any other processor is reading there.

In order to lock LOCATION for reading (i.e., to determine when no other processors are writing at LOCATION and to prevent other processors from writing), the following instruction sequence could be used.

Instruction Sequence

```
Atomically fetch LOCK and
add ONE to NUMBER to increment
the number of processors
waiting to read at LOCATION

Test BIT to see if there
are any processors writing at
LOCATION and conditional branch
to self until BIT is clear
(no more writers), then proceed
```

In order to unlock a read lock, the following instruction sequence could be used.

Instruction Sequence

```
Atomically Fetch LOCK and
subtract ONE from NUMBER to
indicate one fewer readers.
```

To lock for writing:

Instruction Sequence

Load BIT in a register

Test LOCK for any writers and readers and conditional branch to self until LOCK is zero (no writers and no readers), then set BIT (atomic compare and store back in LOCK) and proceed.

To unlock a write Lock:

Instruction Sequence

Clear BIT

Thus the waiting condition of a processor waiting to write is indicated automatically by the execution of an already available conditional branch to self instruction that is part of the algorithm itself, based on a value (NUMBER) that is not simply a single bit

Note that while the same sequence of instructions could be used on other systems (provided the appropriate instructions -- comparable to fetch and add, fetch and subtract, compare and store, and clear -- were available), deadlock detection would not also be accomplished unless either special deadlock detection instructions were included or more complicated set up sequences were used.

In some cases it is not necessary to introduce additional new variables to trigger the waiting condition. For example, suppose a value 0 of a variable (BYTES) already stored in memory indicates an "empty' condition. Then a thread may determine when memory

becomes available by loading BYTES into a register (REG) and testing it.

Instruction Sequence

Load BYTES into REG.

Conditional branch to self
until BYTES is zero, then proceed.

The fact that a processor waits at such a conditional branch instruction for available memory is used as the basis for deadlock detection, without requiring additional instructions. Thus, the same instruction both accomplishes the testing needed to determine when memory is available and indicates when the processor is waiting.

Because the hardware of each processor is now capable of indicating when that processor is waiting (by setting the WAIT bit), it is possible for the operating system 98 to assign overhead tasks to processors that are known to be waiting, regardless of whether a deadlock exists.

Furthermore, the hardware is able to detect when a processor is idle (i.e., is both waiting and not currently assigned to any portion of the instruction sequence). In such a case, assigning the idle processor to an overhead task (e.g., an input/output task) is better than preempting a waiting (but not idle) processor to the task, because another processor may be waiting for results from the non-idle waiting processor. Of course preempting a waiting processor (idle or not) is preferable to preempting one which is not waiting at all.

The operating system is able to determine whether a waiting processor is idle or not by observing, for example, the PRV bit of the processor's PSW. If the bit is set, then the processor has a work assignment, hence it is not idle. If the processor has its PRV bit clear, then it has no work assignment, hence it is idling. The PRV bit is set by the processor executing a special instruction (one of the assignment instructions, called EPR, used to govern the execution of parallel regions) when its thread first enters a parallel region. When EPR is executed, the PRV field in the PSW of the processor whose thread is entering the parallel region is set to indicate that it is executing within a parallel region.

Another type of assignment instruction (XPR) is executed by a processor that exits a parallel region. During execution of XPR, the PRV is cleared indicating that the thread is no longer working in a parallel region.

When a processor reaches the end of its work in a parallel region and executes the XPR instruction, it may be precluded from proceeding until other processors in the parallel region have completed their work. A dependency bit may have been stored to indicate, for example, whether the blocks within a parallel region are dependent such that they all should be completed before subsequent instructions following that parallel region are begun. In that situation, the compiler includes a TEST DEPENDENCY BIT instruction (that branches to itself until that bit is cleared) in the sequence following the XPR instruction. If an exiting processor finds the dependency bit set, it is then both waiting (its WAIT bit is set) and idle (its PRV bit is cleared) indicating to the operating system that it is available to work on

a different thread. The last processor to exit the parallel region clears the dependency bit before executing the TEST instruction and then simply proceeds to the instructions that follow the parallel region. Other types of instructions may also be used to cause a processor to idle.

Claims

1. A method for controlling the simultaneous work of multiple processors at different places in a machine instruction sequence of the kind which includes points at each of which processing may need to wait for the occurrence of a predetermined event, comprising

including in said instruction sequence at each said point a conditional branch which, when said event has not yet occurred, causes the processor executing said branch to enter and repeat a loop including said conditional branch, and causes said processing to continue outside said loop only when said event has occurred, and

detecting that a processor is waiting at one of said points by determining that it is executing such a conditional branch.

2. A method for controlling the simultaneous work of multiple processors at different places in a machine instruction sequence of the kind which includes points at each of which processing may need to wait for the occurrence of a predetermined event, comprising

including in said instruction sequence at each said point a conditional instruction which, when said event has not yet occurred, causes the processor to wait and, when said event has occurred, causes said processor to stop waiting, and

detecting that a processor is waiting at one of said points by determining that it is executing such a conditional instruction, and wherein

there are a plurality of different types of said events on which said conditional instruction may operate.

3. The method of claim 1 or 2 wherein said conditional branch has a single conditional branch

instruction that branches to itself as long as said predetermined event has not yet occurred.

4. The method of claim 1 or 2 wherein said conditional branch is based on a conditional branch instruction of a kind that is also used as part of said machine instruction sequence for purposes other than to indicate that said processor is waiting.

5. The method of claim 1 or 2 further comprising

identifying when a group of said processors, whose processing activities are currently interrelated, are all waiting, and

reassigning at least one of said waiting processors to another place in said machine instruction sequence.

6. The method of claim 5 wherein said group of processors comprises fewer than all said multiple processors.

7. The method of claim 5 wherein all processors in said group are working on a single process.

8. The method of claim 1 or 2 wherein said machine instruction sequence includes an operating system capable of interrupting the activities of each said processor from time to time to reassign it to another place in said machine instruction sequence, and said method further comprises interrupting processors that are waiting before interrupting processors that are not waiting.

9. The method of claim 5 wherein said step of identifying when a group of said processors are all waiting is performed by an individual said processor, whereby said identifying and reassigning may not require interruption of other said processors.

10. The method of claim 2 wherein said events include the setting or clearing of a single bit.

11. A method for determining the status of one of a plurality of processors simultaneously working on a machine instruction sequence comprising

detecting separately

a first condition in which a processor has been assigned to work at a location within said machine instruction sequence but is waiting to proceed until the occurrences of a predetermined event, and

a second condition in which an idle processor has not been assigned to work at any location within said machine instruction sequence.

12. The method of claim 11 wherein

a value is stored corresponding to each processor indicating whether it is assigned to work at any said location, and

said first and second conditions are detected based on said value.

13. The method of claim 12 wherein said machine instruction sequence includes an operating system capable of interrupting the activities of each said processor from time to time to reassign it to another place in said machine instruction sequence, and said method further comprises interrupting a processor in said second condition before interrupting any processor in said first condition.

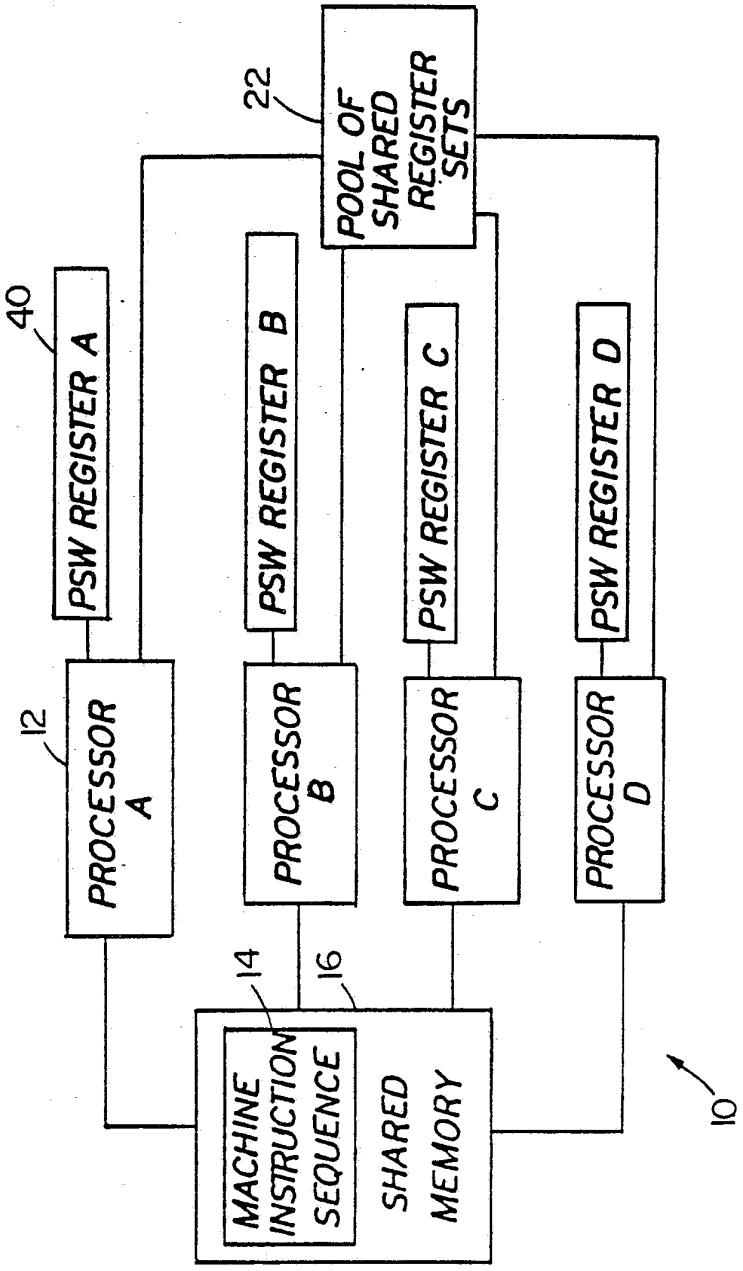

POOL OF SHARED REGISTER SETS
22
PSW REGISTER A
PSW REGISTER B
PSW REGISTER C
PSW REGISTER D
40
PROCESSOR A
PROCESSOR B
PROCESSOR C
PROCESSOR D
12
MACHINE INSTRUCTION SEQUENCE
SHARED MEMORY
14
16
10

FIG. 1

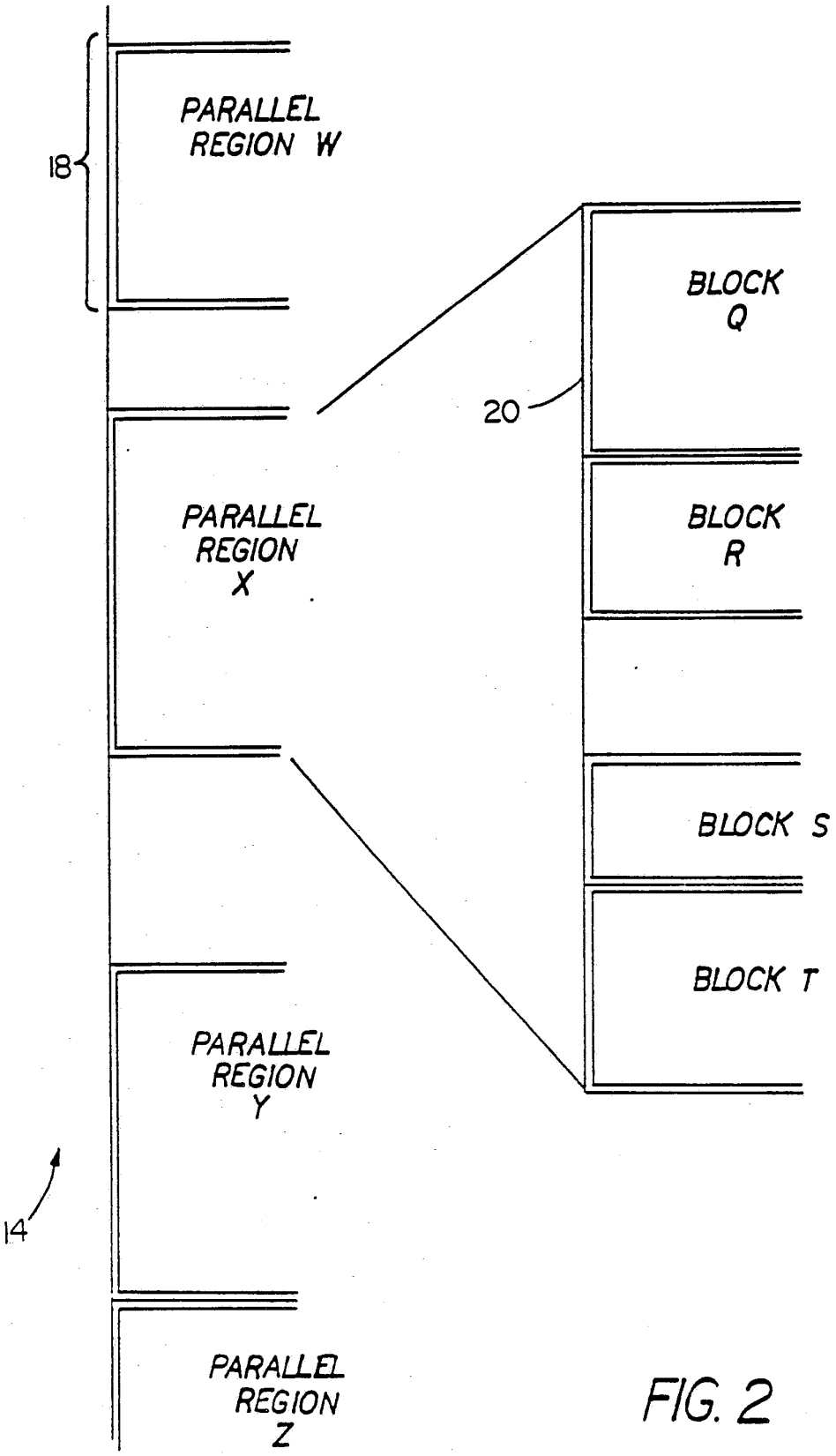

PARALLEL REGION W
18
PARALLEL REGION X
PARALLEL REGION Y
PARALLEL REGION Z
14
20
BLOCK Q
BLOCK R
BLOCK S
BLOCK T

FIG. 2

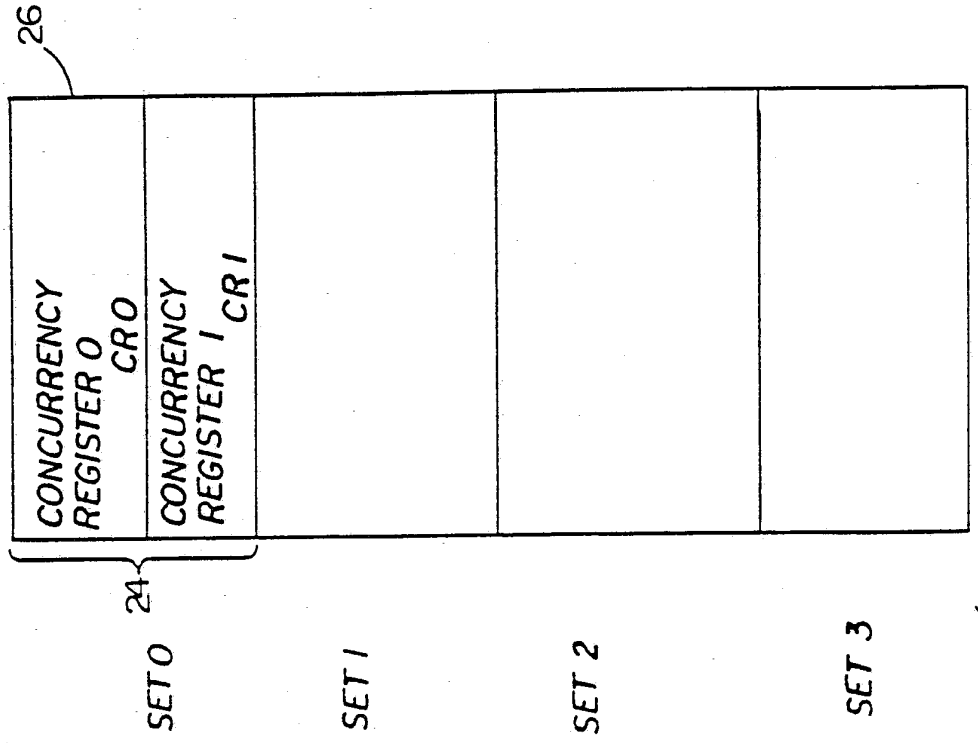


FIG. 3

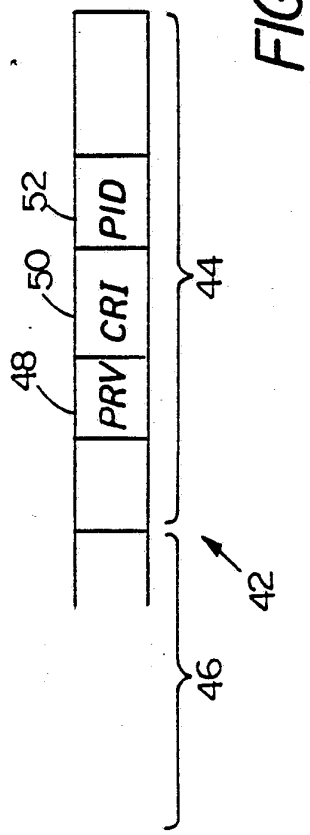


FIG. 4

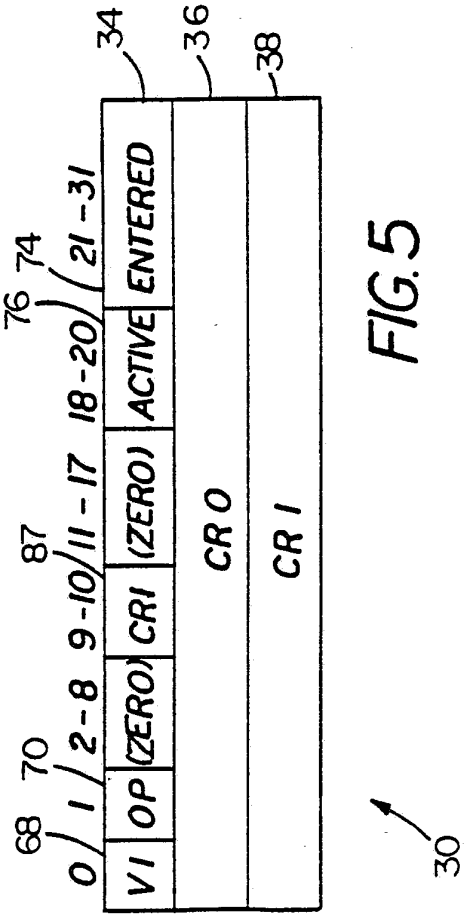


FIG. 5

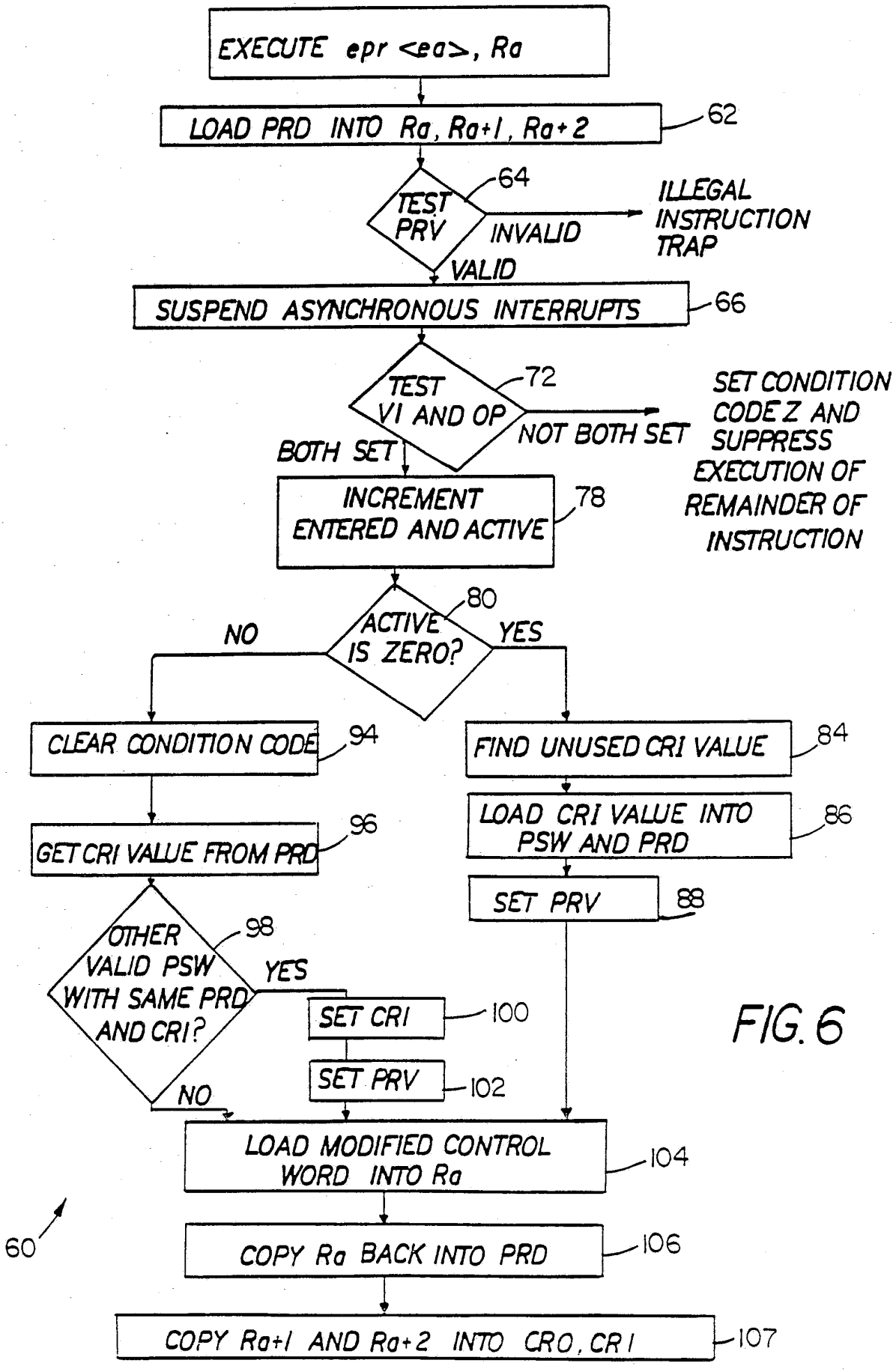

EXECUTE epr <ea>, Ra
LOAD PRD INTO Ra, Ra+1, Ra+2
62
TEST PRV
64
INVALID
ILLEGAL INSTRUCTION TRAP
VALID
SUSPEND ASYNCHRONOUS INTERRUPTS
66
TEST VI AND OP
72
NOT BOTH SET
SET CONDITION CODE Z AND SUPPRESS EXECUTION OF REMAINDER OF INSTRUCTION
BOTH SET
INCREMENT ENTERED AND ACTIVE
78
ACTIVE IS ZERO?
80
NO
YES
CLEAR CONDITION CODE
94
FIND UNUSED CRI VALUE
84
GET CRI VALUE FROM PRD
96
LOAD CRI VALUE INTO PSW AND PRD
86
SET PRV
88
OTHER VALID PSW WITH SAME PRD AND CRI?
98
YES
SET CRI
100
SET PRV
102
NO
LOAD MODIFIED CONTROL WORD INTO Ra
104
60
COPY Ra BACK INTO PRD
106
COPY Ra+1 AND Ra+2 INTO CR0, CR1
107

FIG. 6

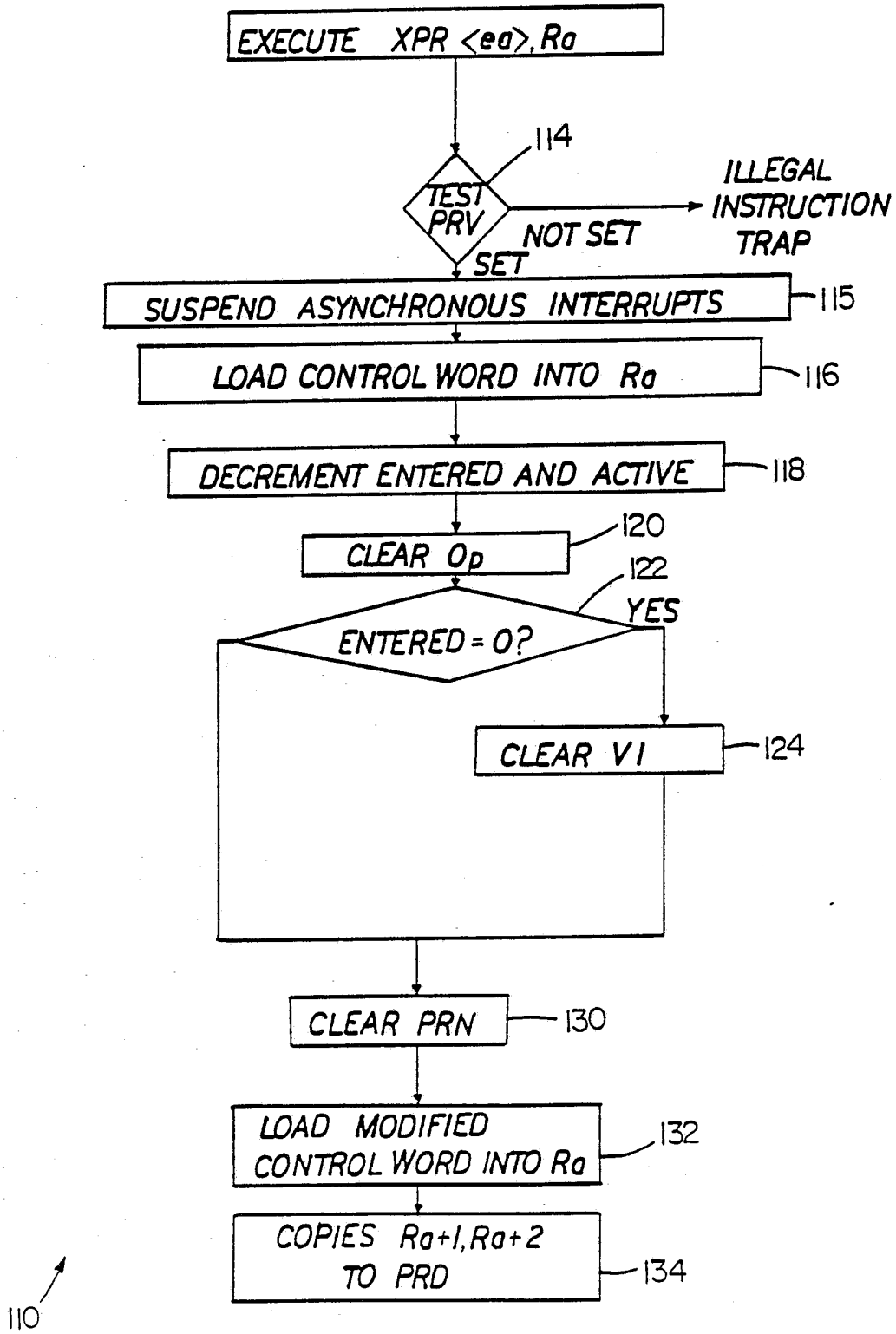

EXECUTE XPR <ea>, Ra
114
TEST PRV
NOT SET
ILLEGAL INSTRUCTION TRAP
SET
SUSPEND ASYNCHRONOUS INTERRUPTS
115
LOAD CONTROL WORD INTO Ra
116
DECREMENT ENTERED AND ACTIVE
118
CLEAR Op
120
122
YES
ENTERED = 0?
CLEAR VI
124
CLEAR PRN
130
LOAD MODIFIED CONTROL WORD INTO Ra
132
COPIES Ra+1, Ra+2 TO PRD
134
110

FIG. 7

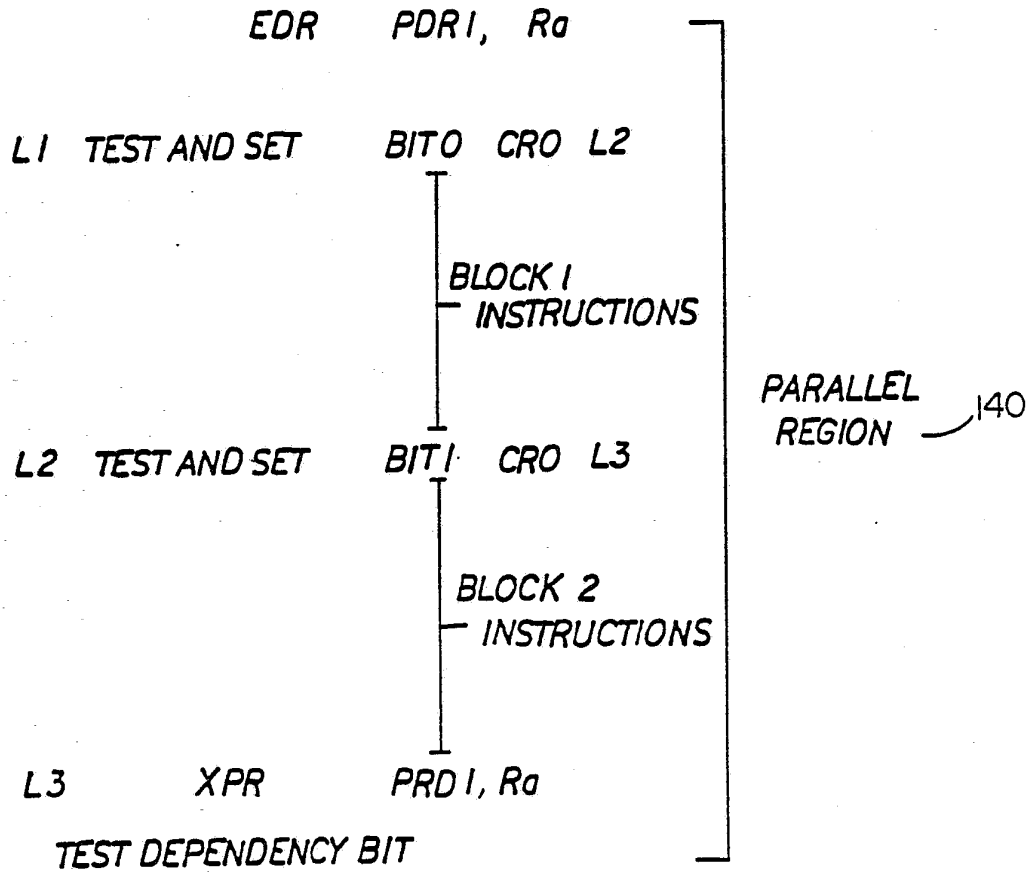

EOR PDR 1, Ra
L1 TEST AND SET BIT 0 CR0 L2
BLOCK 1 INSTRUCTIONS
L2 TEST AND SET BIT 1 CR0 L3
BLOCK 2 INSTRUCTIONS
L3 XPR PRD 1, Ra
TEST DEPENDENCY BIT
PARALLEL REGION
140

FIG. 8

INTERNATIONAL SEARCH REPORT

International Application No. PCT/US88/02304

I. CLASSIFICATION OF SUBJECT MATTER (if several classification symbols apply, indicate all) [6]

According to International Patent Classification (IPC) or to both National Classification and IPC

IPC (4): G06F 15/16

U.S. Cl. 364/200

II. FIELDS SEARCHED

Minimum Documentation Searched [7]

Classification System	Classification Symbols
U.S.	364/200, 900

Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched [8]

III. DOCUMENTS CONSIDERED TO BE RELEVANT [9]

Category *	Citation of Document, [11] with indication, where appropriate, of the relevant passages [12]	Relevant to Claim No. [13]
X	U.S., A, 3,810,119 (ZIEVE ET AL.) 07 MAY 1974, see entire document.	1-4
X	U.S., A, 4,318 182 (BACHMAN ET AL.) 02 MARCH 1982, see entire document.	1-13
X	U.S., A, 4,445,197 (LORIE ET AL.) 24 APRIL 1984, see entire document.	11-13
Y	U.S., A, 4,494,193 (BRAHM ET AL.) 15 JANUARY 1985, see entire document.	5-9
A	U.S., A, 3,518,413 (HOLTEY) 30 JUNE 1970, See column 1, lines 14-21 and column 2, lines 60-66.	1-2
A	U.S., A, 4,189,771 (ROEVER) 19 FEBRUARY 1980, see entire document.	1-13
A, P	U.S., A, 4,754,398 (PRIBNOW) 28 JUNE 1988, see entire document.	1-13

* Special categories of cited documents: [10]

"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

"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

IV. CERTIFICATION

Date of the Actual Completion of the International Search	Date of Mailing of this International Search Report
22 AUGUST 1988	13 OCT 1988
International Searching Authority	**Signature of Authorized Officer**
ISA/US	Debra Chun Debra Chun