(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
2 August 2012 (02.08.2012)

WIPO | PCT

(10) International Publication Number
WO 2012/101538 Al

(51) International Patent Classification:
G06F 12/10 (2006.01) *G06F 12/00* (2006.01)

(21) International Application Number:
PCT/IB20 12/050 154

(22) International Filing Date:
12 January 2012 (12.01 .2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11152253.8 26 January 20 11 (26.01.201 1) EP

(71) Applicant *(for all designated States except US):* **INTERNATIONAL BUSINESS MACHINES CORPORATION** [US/US]; New Orchard Road, Armonk, New York 10504 (US).

(71) Applicants *(for MG only):* **IBM UNITED KINGDOM LIMITED** [GB/GB]; PO Box 41, North Harbour, Portsmouth Hampshire P06 3AU (GB). **IBM Japan Limited** [JP/JP]; 19-21 Nihonbashi, Hakozaki-cho, Chuo-ku, Tokyo 103-85 10 (JP).

(72) Inventors; and

(75) Inventors/Applicants *(for US only):* **CAMPBELL, Neil, Anthony** [GB/GB]; IBM United Kingdom Limited, c/o Intellectual Property Law, Hursley, Park, Hursley, Winchester Hampshire S021 2JN (GB). **DANKEL, Gisle, Mikal, Nitter** [NO/GB]; IBM United Kingdom Limited, Maybrook House, 40 Blackfriars Street, Manchester Greater Manchester M3 2EG (GB).

(74) Agent: **STRETTON, Peter;** IBM United Kingdom Limited, Intellectual Property Law Hursley Park, Winchester Hampshire S021 2JN (GB).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available):* AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available):* ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK,

[Continued on next page]

(54) Title: EMULATION OF STRONGLY ORDERED MEMORY MODELS

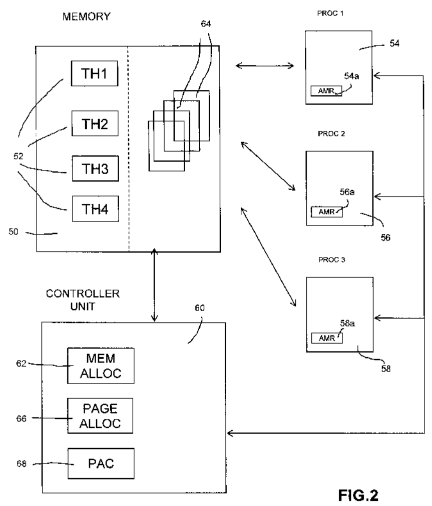

FIG.2

(57) Abstract: In a computer system comprising a memory (50), one or more processors (54, 56, 58), and a controller unit (60) coupled with the or each processor and the memory, virtual page class key protection is used to ensure that multiple threads (52) of a process may not concurrently access pages (64) of memory. A virtual page class is assigned to each thread, and no thread is ever permitted to access another thread's class, so all pages (64) of memory (50) are effectively partitioned into classes according to which thread is using them. When multiple threads need access to a page of memory, it is placed in a special shared class which all threads may access. All pages in the shared class are also marked with strong access ordering mode to ensure strong memory ordering semantics apply when needed. Pages not in the shared class do not need strong access ordering enabled as they are guaranteed to be accessible by only a single thread.

WO 2012/101538 A1

SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

EMULATION OF STRONGLY ORDERED MEMORY MODELS

Field of the Invention:

The present invention relates generally to the field of computers and computer systems. More particularly, the present invention relates to the protection of memory consistency in multiprocessor computing systems, and in processor devices for use in such systems.

Background Art:

Multiprocessor computer systems require reads and writes to memory to be communicated between processors in a consistent way. The semantics of this communication are defined by the architecture's memory consistency model, and different architectures make use of different models. Some devices, such as the IBM POWER series of processors, provide weaker memory ordering semantics than some other processors, such as those based on x86 and SPARC architectures. This leads to significant difficulties in correctly emulating such processors on a POWER or similar device whilst maintaining high performance.

It should be noted that the term "multiprocessor" as used herein encompasses dual- and multi-core processor devices, as well as multiple hardware thread and multiple CPU systems.

A number of existing solutions are available for providing sufficiently strong memory ordering on devices such as POWER processors:

- Explicit synchronisation: an emulator may make use of explicit synchronisation instructions (for example, sync, isync, lwsync on POWER) between all memory access instructions. This approach tends to exhibit poor performance though, as all memory operations are fully synchronised, rather than only those which may affect other processors.

- Force serial execution: an emulator or operating system may enforce correct memory ordering semantics by ensuring that only one software or program code portion (thread) of execution may proceed at once, and that between scheduling of each task a suitable barrier is inserted to allow all pending memory operations to complete. This effectively removes the benefit of having multiple processors, and so is not useful for highly parallel workloads.

- Programmer intervention: the application code may be modified by a programmer to insert appropriate synchronisation instructions or hints that allow the emulator to apply strong ordering only when the program requires it. This is generally difficult to achieve as even if source code is available, the programmer may not be able to manually identify all the areas requiring such ordering.

- Strong Access Ordering (SAO) mode: the IBM POWER7 processor allows pages to be marked as requiring strong memory ordering. Hardware in the form of a controller unit (described in more detail hereinafter) then ensures that any accesses by any thread to these pages occurs in a strongly ordered fashion, while access to other pages proceed as normal. There is a performance penalty for requiring this ordering though, and in a multi-threaded environment it has hitherto been assumed that all pages may be accessed by all threads, and as such all pages are marked for SAO and incur this cost.

- Shared memory detection: this technique is described in United Kingdom Patent GB-2444148B. In this approach, the emulator runs each separate thread of a process in its own separate virtual address space, as well as allocating a shared address space which all threads can access. Each page of virtual memory is mapped into only a single address space at a time; that is, either that belonging to one thread, or the shared address space. When an attempt is made by a thread to access a page of memory, if it is not in the thread's own address space the access will fault, and the page can either be moved to that address space, or moved into the shared address space. In the latter case, the instructions which accessed the page are also retranslated to insert appropriate memory ordering instructions. This ensures that any accesses to data which is shared between threads contains the appropriate synchronisation, while data required by only a single thread does not require said synchronisation and as such does not incur the associated performance penalty.

This last approach has the significant drawback that the decision as to whether to use strongly or weakly ordered accesses on a given piece of memory becomes a property of the code which accesses it, rather than the region of memory itself, as the code will typically need to select which address space to access the pages in. This means that a single piece of code which accesses both shared and thread local data will:

a) incur a fault and require retranslation each time it accesses a different piece of memory;
b) perform some address selection at runtime to determine the correct address space; or
c) make the most pessimistic assumption and move all data accessed into the shared address space, thus removing much of the benefit of the technique. A further drawback of this technique is the potential reliance on a very large virtual address space.

It is therefore an object of the present invention to provide a means for enabling emulation of memory consistency models in devices supporting differing memory consistency models and, in particular, the emulation of strongly ordered architectures on a less strongly ordered architecture.

Summary of the Invention:

In accordance with a first aspect of the present invention there is provided a computer system, comprising:
- a memory having a plurality of program code portions stored therein, including at least a first program code portion and a second program code portion;
- one or more processors arranged to execute the plurality of program code portions stored in the memory; and
- a controller unit arranged to control execution of the or each of the processors, wherein the controller unit comprises:
- a memory allocation unit arranged to divide a part of the memory into a plurality of discrete pages;
- a page allocation unit arranged to assign a virtual page class, either shared or unshared, to each page of the memory and to attach a respective first memory consistency model indicator to each shared page; and
- a page access control unit controlling access by program code portions to memory pages in dependence on the assigned virtual page class, wherein shared pages are accessible by all of

the program code portions, and unshared pages are accessible by only one of said first and second program code portions;

wherein the controller unit controls execution of each program code portion, including accessing for each a respective page of said memory, under a first memory consistency model if said page is marked with the first memory consistency model indicator, otherwise under a second memory consistency model.

The present invention makes use of a mechanism to ensure that multiple program code portions (threads) of a process may not concurrently access pages of memory. One example of such a mechanism is virtual page class key protection (described in more detail hereinafter) provided in the IBM POWER7 architecture. When it is determined that certain pages must be shared between threads, a first memory consistency model, such as strong access ordering (SAO) is used to enforce the memory ordering semantics on these pages. This provides higher performance than enabling SAO across all pages, but with a much reduced cost associated with detecting which pages to share, as compared with memory consistency emulation techniques such as those described in the above-mentioned United Kingdom patent GB-2444148B. Such a solution is of benefit when emulating an architecture with stronger memory ordering semantics on an architecture with weaker semantics, for example emulating x86 or SPARC processors on POWER processors. There are other possible uses described below.

Within a computer system according to the present invention, the page allocation unit may be arranged to periodically remove the assigned virtual page class and first memory consistency model indicator from all pages having the shared virtual page class. As will be described, over prolonged operation a large number of pages may migrate towards the shared status, affecting the efficiency of the operation, and periodically cutting the number of shared pages addresses this issue.

In operation, a sub-group may comprise two or more of the said plurality of program code portions (threads), and the page allocation unit may be arranged to assign the virtual page class following detection by the page access control unit of an attempt to access a page by any one of the threads of the sub-group. As will be described hereinafter, in such a system a mechanism must be provided to ensure that the memory ordering semantics are appropriately preserved - for example ensuring that only a single thread in a sub-group can operate at any one time.

Typically the computer system has finite capacity, with the page allocation unit being arranged to assign a fixed number n of virtual page classes, with one virtual page class being shared and the remaining n-1 being unshared; in such cases the page allocation unit may assign the shared virtual page class to all memory pages handled when the n-1 unshared virtual page classes have been allocated. In a further optimisation, the controller unit may be arranged to apply a prioritisation selection operation to select the n-1 threads or groups of threads to receive the unshared virtual page class when n or more threads or groups of threads are handled, for example the n-1 threads or groups of threads which most frequently access unshared pages, with the remaining threads being assigned to the generally less efficient shared virtual page class.

Also in accordance with the present invention there is provided a method to emulate memory consistency models in a computer system in which a plurality of program code portions access a memory during their respective execution, comprising the computer-implemented steps of:
- dividing said memory into a plurality of discrete pages;
- assigning a virtual page class, either shared or unshared, to each page of said memory, wherein shared pages are accessible by all of said plurality of program code portions, and unshared pages are accessible by a sub-group of said plurality of program code portions;
- marking each shared page with a first memory consistency model indicator; and
- executing each program code portion, including accessing a respective page of said memory, under a first memory consistency model if said page is marked with the first memory consistency model indicator, otherwise under a second memory consistency model.

In such a method the first memory consistency model applied to shared pages suitably has stronger memory ordering constraints than the second memory consistency model, thereby enabling emulation of strongly ordered memory models in a device conventionally supporting less strong ordering.

In the method of the present invention, each page may be initially neither shared nor unshared, and the step of assigning a virtual page class may comprise:
- detecting an attempt to access a page by a first program code portion;
- assigning the unshared virtual page class to that page; and
- allowing access to that page by the said first program code portion.

The step of assigning a virtual page class may then be extended to comprise:
- detecting an attempt to access an unshared page by a second program code portion, which second code portion is not part of the same sub-group of said plurality of program code portions as said first program code portion;
- changing from unshared to shared the virtual page class for that page; and
- allowing access to that page by the said second program code portion.

This method may further comprising the step of periodically removing the assigned virtual page class from all shared pages to address the possibility of long-term migration of large numbers of pages to the shared virtual page class as mentioned above.

The above-referenced sub-group of said plurality of program code portions may comprise just a single code portion or thread, or it may comprise two or more threads and the assigning of the unshared virtual page class may follow detection of an attempt to access a page by any one of those threads. When said sub-group comprises more than one thread, to maintain memory ordering semantics only a single one of those threads is permitted to execute at a time. On termination of a thread, any page associated thereto by the unshared virtual page class suitably has that virtual page class removed, thereby freeing the page class for other threads.

Although single pages have been referred to above, it will be recognised that two or more discrete pages of memory may be grouped, with the assignment of a virtual page class to one page of the group causing the assignment of the same virtual page class to all pages of the group.

The invention further provides a computer program stored on a computer readable medium and loadable into the internal memory of a digital computer, comprising software code portions, when said program is run on a computer, for performing the method according to the invention and as described above.

The summary of the present invention does not recite all the necessary features of the invention, and sub-combinations of those features may also encompass the invention.

Brief Description of the Drawings:

The present invention will now be described, by way of example only, with reference to preferred embodiments, as illustrated in the following figures, in which:

Figure 1 is block schematic diagram of the components of a multiprocessor computer system suitable to embody the invention;
Figure 2 represents functional components of a computer system according to the invention;
Figure 3 represents the application of virtual page classes to memory pages;
Figure 4 is a flowchart representing the determination of virtual page class to be applied;
Figure 5 is a flowchart detailing the process for periodically resetting virtual page class allocations;
Figure 6 represents the application of virtual page classes where plural threads are grouped; and
Figure 7 is a flowchart detailing a modification to the process of Figure 4.

Figure 1 schematically represents the components of a computer system 8 suitable to embody the present invention. A processor CPU 10 is coupled with random access memory RAM 12 and read only memory ROM 14 by an address and data bus 16. Also connected to CPU 10 via the address and data bus 16 is a further processor 42, which may be a further CPU sharing tasks with the first CPU 10, or may be a coprocessor device 42 supplementing the function of the CPU 10, handling processes such as floating point arithmetic, graphics processing, signal processing and encryption. Each of these internal hardware devices 10, 12, 14, 42 includes a respective interface (not shown) supporting connection to the bus 16. These interfaces are conventional in form and need not be described in further detail

Also connected to the CPU 10 via bus 16 are a number of external hardware device interface stages (generally denoted 18). A first interface stage 20 supports the connection of external input/output devices, such as a mouse 22 and/or keyboard 24. A second interface stage 26 supports the connection of external output devices such as a display screen 28 and/or audio output device 30, such as headphones or speakers. A third interface stage 32 supports the connection to external data storage devices in the form of computer readable media: such external storage may as shown be provided by a removable optical or magnetic disc 34 (accessed by a suitably configured disc reader 36). Alternatively or additionally the external storage may be in the form of a solid state memory device such as an extension drive or memory stick. The external storage may contain a computer program, containing program software code portions which, when run by the CPU 10, perform the method according to the present invention. A fourth interface stage 38 supports connection of the system to remote devices or systems via wired or wireless networks 40, for example over a local area network LAN or via the internet. A further computer system 44 is shown coupled via network 40 with the first computer system 8 described above.

The CPU 10 may be of many different types, from different manufacturers, and based on different instructions set architectures (ISAs). The feature relevant to the present invention is that the CPU 10 is required to emulate a memory consistency model that differs from its own, particularly but not exclusively to support a stronger degree of memory ordering than when handling programs specifically written for it. In the following examples, the CPU 10 is an IBM POWER type of processor emulating a device with stronger memory ordering (for example a device based on x86 or SPARC ISA) although the invention is not limited to such specific processor types.

Virtual page class key protection, implemented in IBM POWER7 processors, is a convenient means to allocate each page of storage to a separate storage class, although the present invention is not so limited. The current POWER7 implementation permits up to 32 classes. Each processor contains a register (the Authority Mask Register AMR) specifying read and write permissions for each class. When a processor attempts to access a page of storage, its permission to access the virtual page class of that storage is checked, and the access only succeeds if it has sufficient permissions. The class to which a page of storage belongs may be changed by updating its page table entry. The access permissions for a processor may be updated by modifying its AMR. In a multithreaded application, an AMR may be maintained for each thread which will be automatically applied by the operating system on thread dispatch. There are various alternatives to virtual page class key protection: the key feature of any alternative is that it provides a means to control access to different sets of pages by different threads. In one alternative, the operating system sets up the page tables differently for each thread, such that different threads are unable to access pages that are not shared or their own private (unshared) pages.

Figure 2 illustrates the functional components of the computer system 8 of Figure 1 configured to apply virtual page classes to enable stronger memory ordering. The system comprises a memory 50 which may be an area of RAM 12. The memory 50 holds a number of threads (TH1, TH2, TH3, TH4) 52 for execution. A number of processors 54, 56, 58 are coupled with the memory 50 and arranged to execute the threads. The processors 54, 56, 58 may comprise separate cores in CPU 10, the further processor 42, and/or a processor device hosted by the further computer system 44. Each of the processors has a respective AMR 54a, 56a, 58a.

The system further comprises a controller unit 60 coupled with the memory 50 and processors and arranged to control execution of the processors 54, 56, 58. The controller unit 60 may be a separate device or a functional subset of the CPU 10 ISA. The unit 60 includes a memory allocation unit 62 arranged to divide a part of the memory 50 into a plurality of discrete pages 64, a page allocation unit 66 arranged to assign a virtual page class (either shared or unshared) to each page of the memory and to attach a respective first memory consistency model indicator to each shared page, and a page access control unit PAC 68 controlling access by threads 52 to memory pages 64 during execution in dependence on the assigned virtual page class. Shared pages are accessible by all of the threads, and unshared pages are accessible by only one thread or group of threads, as discussed in more detail below.

The controller unit 60 controls execution of each thread, including accessing for each a respective page 64 of the memory 50, under a first memory consistency model giving stronger ordering if that page is marked with the first memory consistency model indicator, otherwise under a second memory consistency model providing less strong ordering.

The invention works by assigning a virtual page class to each thread 52. No thread is ever permitted to access another thread's class, so all pages 64 of memory are effectively partitioned into classes according to which thread is using them. When multiple threads need access to a page of memory, that page is placed in a special shared class which all threads may access; all pages in the shared class are also marked with a memory consistency model indicator (SAO mode indicator in this POWER7 example) to ensure strong memory ordering semantics apply when needed. Pages not in the shared class do not need SAO enabled, as they are guaranteed to be accessible by only a single thread.

A basic high-level algorithm for this method is now presented, with Figures 3A to 3D showing a sequence of thread accesses to pages or groups of pages and the resultant allocation and reallocation of classes. In the context of a POWER7 device, where 32 up to classes are supported, virtual page class 0 is reserved as the shared class and classes 1 to 31 are available for allocation as unshared.

In Figure 3A, a number of pages 70, 72, 74, 76, 78 and 80 are available for access by a first thread (TH1) 82. The pages are collected in three sub-groups, with the first sub-group containing pages 70, 72 and 74, the second containing page 76, and the third containing pages 78 and 80. To the left of each sub-group in the figure is shown the class type (CL=), the class number (CL#=), and the SAO set/not-set indicator that applies to all pages in the sub-group. The values shown in each of Figures 3A to 3D represent the values/settings for CL=, CL#=, and SAO immediately prior to access by the thread shown. Initially, all memory pages are marked as neither readable nor writable by any thread so in Figure 3A all sub-groups have CL= and CL#= unspecified and SAO unset.

The first thread 82 (TH1) of the process is allocated the class 1. Its AMR is set to allow access to class 1 and the shared class, class 0. In Figure 3A, the first thread accesses the single page sub-group 76: as can be seen in Figure 3B, the effect of this is to set CL=U (class = unshared) and CL#=1 (class 1 is assigned) for the page 76. As the class is unshared, strong ordering is not required and SAO is not set. The settings of CL=, CL#= and SAO are unchanged for the other sub-groups.

Each subsequently created thread is allocated the next available class, and it is permitted to access its own class, and the shared class. The second thread (TH2) 84 therefore has its AMR set to allow access to classes 2 and 0. In Figure 3B, the second thread accesses the page 72 from the three-page sub-group 70, 72, 74: as can be seen in Figure 3C, the effect of this is to set CL=U (class = unshared) and CL#=2 (class 2 is assigned) for each of the pages 70, 72 and 76. As above, since the class is unshared, strong ordering is not required and SAO is not set.

The third thread (TH3) 86 has its AMR set to allow access to classes 3 and 0. In Figure 3C, the third thread accesses the page 74 from the three-page sub-group 70, 72, 74: as can be seen in Figure 3D, the effect of this is to set CL=S (class = shared) and CL#=0 (shared class 0 is assigned) for each of the pages 70, 72 and 76. Now, since the class is shared, strong ordering is required and SAO is set.

As also shown in Figure 3D, the fourth thread (TH4) 88 has its AMR set to allow access to classes 4 and 0. Although the third thread 86 resulted in the allocation of the pages 70, 72, 74 to shared class 0, the thread 86 will retain its allocation to class 3. When a thread is terminated, all memory pages in that thread's class are marked as neither readable nor writable by any thread. This thread's class is now available for a future thread to use.

Whenever a page of memory is first accessed by any thread, the access will fault as the memory is neither readable nor writable. At this point, the correct read/write permissions are applied to the page, and it is assigned to the class of the thread which accessed it. Now that thread can access it, but no other thread can; as such, no memory synchronisation is required on that page. This process is illustrated in more detail by the flowchart of Figure 4 which commences at 100 with the division of an area of memory (50, Fig.2) into pages. At 102 an access by a thread is detected, and at 104 a check is made as to whether the page sought to be accessed has the shared class set. If so, the process moves to step 106 where the thread is permitted to access the page and, at 108, the thread executes "strongly" - that is to say under the stronger memory consistency specified by SAO. If not, at 110 a further check is made, this time as to whether the class of the page sought to be accessed has the threads own class (as specified by the AMR of the thread). If so, at 112 the thread is permitted to access the page and, at 114, the thread executes "weakly" - that is to say not under the stronger memory consistency specified by SAO.

If the test at 110 shows that the thread and page classes are not the same, the process moves to 116 where a further test determines whether the page already has an unshared class set. If so, the process moves to step 118, where the class for that page is set to shared, and then to 120 where the SAO marker for the page is set. After this, the process moves again to step 106 where the thread is permitted to access the page and, at 108, the thread executes strongly.

Should the test at step 116 show the page is not set to unshared class (with step 104 having already established that it is not shared), then at step 122 the class for that page is set to the unshared class specified by the AMR of the thread, following which the process again moves to access and strong execution at steps 106 and 108.

This is a relatively computationally inexpensive approach, requiring in the worst case only two faults per page - the first will move the page into a thread's (unshared) class, and the second will return it to the shared class. In most cases this cost will be more than offset by minimising the number of pages in the shared class.

In an optional step, if the test of 116 determines that the page has neither shared nor unshared class assigned (and the requesting thread has no class specified in its AMR), then at 124 a determination is made as to whether all of the unshared classes have been allocated. If not, the next available class number is assigned to the thread and, at 122, set for the page. If there are no free (unallocated) unshared classes, the process defaults to step 118 where the thread and class are set to shared, followed by access and strongly ordered execution at 106 and 108.

As an optimisation, in some cases it will be preferable to assign or reassign sub-groups comprising a number of pages - for example, when one page on a larger region such as a stack is accessed, the sub-group comprising the whole region can be moved at once, as with the sub-group of pages 70, 72, 74 in Figure 3. Assigning or reassigning regions rather than individual pages may reduce the overall number of faults taken.

After some time of execution, all pages in use by the application will belong to either the shared class, if they are used by multiple threads, or to individual threads' classes if they are only required by a single thread. This means that only a small proportion of the total number of pages must suffer the performance penalty of the stronger memory ordering semantics.

Despite the improvement in efficiency provided by the present invention as described above, the skilled reader will understand that there are two potential issues in the solution as described. Firstly, any pages which are used primarily by one thread, but are very occasionally accessed by one or more other threads will continue to pay the cost of the strong memory ordering. Secondly, the number of threads supported by this approach is limited by the number of virtual page classes provided by the processor. Further embodiments will now be described which address these issues.

To address the first of the above issues, at a regular interval all pages in the shared class are removed from that class, and moved back into a thread's class on the next access. Pages which continue to be shared will then move gradually back to the shared class as other threads require them. This periodic removal is illustrated by the flow chart of Figure 5. Following the start of the process at 140, a check is made at 142 to obtain operating system data from which at 150 a test determines whether predetermined criteria for triggering the purging of the shared class have been met. The criteria may take a number of forms, including the simple elapse of a fixed interval determined by reference to a timer 144. Alternatively, the criteria may be derived from a source of data 146 about the shared class, such as the total number of pages having the shared class or the percentage of total pages having the shared class. In a further alternative the criteria may be derived from a source of unshared class information 148 such as the number of unallocated classes available.

When the test step 150 determines that a purge is to occur the process moves to step 152 in which all pages with shared class have their class deleted, followed by step 154 in which the SAO indicators for those pages are unset. Following a suitable wait interval 156 the process reverts to obtaining data at the check step 142.

To address the second issue and allow more threads to be supported than the number of classes available, classes may instead be assigned to a group of threads, as will now be described with reference to Figure 6. Four threads (TH1A-C, TH2) 180, 182, 184, 186 are shown, each of which is potentially seeking to access memory page 190. Three of the threads (TH1A-C) 180, 182, 184 form a group 188, with a single class assigned for the group, and all threads in the group 188 are able to access memory pages of that class and the shared class, but not the unshared class belonging to any other page or group of pages. A page of memory is placed in a class when it is used by any thread in that group of threads; it is moved to the shared class when more than one thread group requires access.

In like manner to Figure 3, to the left of the page 190 in the figure is shown the class type (CL=), the class number (CL#=), and the SAO set/not-set indicator that applies to that page. The values shown in each of Figures 6A to 6D represent the values/settings for CL=, CL#=, and SAO immediately prior to access by the thread shown. Initially, all memory pages are marked as neither readable nor writable by any thread so in Figure 6A the page has CL= and CL#= unspecified and SAO unset.

The threads (TH1A-C) of the group 188 are allocated the class 1. The respective AMRs are set to allow access to class 1 and the shared class, class 0. In Figure 6A, a first thread (THIB) from the group 188 accesses the page 190: as can be seen in Figure 6B, the effect of this is to set CL=U (class = unshared) and CL#=1 (class 1 is assigned) for the page 190. As the class is unshared, strong ordering is not required and SAO is not set.

In Figure 6B, a second thread (TH1A) 180 from the group 188 accesses the page 190. As can be seen in Figure 6C, the effect of this is to leave CL=U (class = unshared) and CL#=1 (class 1 is assigned for all threads of the group) for the page 190. As above, since the class is unshared, strong ordering is not required and SAO is not set.

The fourth thread (TH2) 186 is not part of the group 188 and has its AMR set to allow access to classes 2 and 0. In Figure 6C, the fourth thread accesses the page 190: as can be seen in Figure 6D, the effect of this is to set CL=S (class = shared) and CL#=0 (shared class 0 is assigned) for the page 190. Now, since the class is shared, strong ordering is required and SAO is set.

Within each group, only one member thread is permitted to execute at once. This is similar to the serial execution approach described in the introductory portion of this document. In this case, however, rather than allowing only a single thread of execution, the maximum number of concurrently executing threads is equal to the number of virtual page classes available, minus one (for the shared class).

An alternative approach to the second issue is to assign each of the classes to single threads, but to place any additional threads in the shared class only, as shown at step 106 in Figure 4 where a determination that there are no unassigned classes leads to step 118 where the class is set directly to shared. Such threads tend to perform less well because all of their accesses would be to SAO pages.

A further alternative approach to the second issue is again to assign each of the classes to single threads and then to prioritise threads and place the most important or intensive threads in their own class, with the remaining (less important) threads placed in the shared class. This is illustrated in Figure 7 which shows a replacement for steps 106 and 108 in the flowchart of Figure 4. Following from step 104 (Fig. 4) which determines that a page desired to be accessed by a thread has no virtual page status set, step 200 tests whether all the available unshared classes have been allocated. If there are no free unshared classes, at step 202 the existing unshared classes and the newly received unshared class request are parsed or otherwise processed to determine a priority rating for each. The priority ratings are compared and, at step 204, if it is determined that the newly received unshared class request has no higher priority than any of the already allocated unshared classes, the process reverts to step 120 (Fig. 4) with the newly accessed page being set to shared status. If however the test at step 204 shows there to be one or more lower priority unshared classes allocated, then at 206 the pages or pages associated with the lowest priority of the allocated classes have their class reset to shared and the SAO indicator is set. This then frees an unshared class which, at step 208, is taken by the newly received page.

Some aspects of this invention are similar to the solution described in the above-referenced United Kingdom patent GB-2444148B (hereinafter ' 148) however there are several important distinctions. Firstly, the system of 148 detects the sharing of memory by the use of separate virtual address space regions, which limits that solution to the number of address spaces which can be supported within the addressable range available to processor. In the worst case, only one such address space region may be possible, in which case that solution would not be viable. The present invention does not require additional virtual address space, as it uses virtual page classes to detect memory sharing between threads.

The system of 148 is described in terms of binary translation or dynamic code optimisation, and it must modify the instruction stream in order for a thread to access the correct virtual address space. The present invention does not use separate virtual address spaces, and as such does not rely on code generation or modification. It is a significant benefit that the present invention may be used to efficiently strengthen the memory ordering model of a system by implementing the described features in the operating system. This could therefore be used, for example, to provide an inexpensive strong ordering mode for applications running on devices such as a POWER processor, perhaps to aid developers porting applications from a more strongly ordered architecture. In such applications it is suggested that instruction fetches are not protected by the existing virtual page class key protection mechanism, and so any changes to the instruction stream would need to be identified using other means.

In the present invention, the shared or unshared nature of a region of memory is associated with the pages of memory themselves, rather than being associated with the code that accesses them. This makes it more resilient in the case of utility code which accesses both shared and unshared regions. Whilst 148 does refer to the concept of having a mechanism similar to SAO which would remove the need to insert synchronisation instructions in code which accesses shared memory, any code which accesses both shared and unshared regions would however either need repeated retranslation, or potentially costly runtime selection to determine how to access the memory. The present invention does not have these drawbacks.

Whilst embodiments of the present invention have been described above, the technical scope of the invention is not limited to the scope of the above-described embodiments. It should be apparent to those skilled in the art that various changes or improvements can be made to the embodiments. It is apparent from the description of the appended claims that implementations including such changes or improvements are encompassed in the technical scope of the invention.

Registered Trademarks:

The following terms are registered trademarks and should be read as such wherever they occur in this document:

SPARC

IBM

POWER

POWER7

CLAIMS

1. A computer system, comprising:

- a memory having a plurality of program code portions stored therein, including at least a first program code portion and a second program code portion;

- one or more processors arranged to execute the plurality of program code portions stored in the memory; and

- a controller unit arranged to control execution of the or each of the processors, wherein the controller unit comprises:

- a memory allocation unit arranged to divide a part of the memory into a plurality of discrete pages;

- a page allocation unit arranged to assign a virtual page class, either shared or unshared, to each page of the memory and to attach a respective first memory consistency model indicator to each shared page; and

- a page access control unit controlling access by program code portions to memory pages in dependence on the assigned virtual page class, wherein shared pages are accessible by all of the program code portions, and unshared pages are accessible by only one of said first and second program code portions;

wherein the controller unit controls execution of each program code portion, including accessing for each a respective page of said memory, under a first memory consistency model if said page is marked with the first memory consistency model indicator, otherwise under a second memory consistency model.

2. A computer system as claimed in Claim 1, wherein the page allocation unit is arranged to periodically remove the assigned virtual page class and first memory consistency model indicator from all pages having the shared virtual page class.

3. A computer system as claimed in Claim 1 or Claim 2, wherein a sub-group comprises two or more of said plurality of program code portions, and the page allocation unit is arranged to assign the virtual page class following detection by said page access control unit of an attempt to access a page by any one of said two or more program code portions of the sub-group.

4. A computer system as claimed in any of Claims 1 to 3, wherein the page allocation unit is arranged to assign a fixed number n of virtual page classes, with one virtual page class being shared and the remaining n-1 being unshared, and to assign the shared virtual page class to all memory pages handled when the n-1 unshared virtual page classes have been allocated.

5. A computer system as claimed in Claim 4, wherein the controller unit is further arranged to apply a prioritisation selection operation to select the n-1 program code portions to receive the unshared virtual page class when n or more program code portions are handled.

6. A method to emulate memory consistency models in a multiprocessor computer system in which a plurality of program code portions access a memory during their respective execution, comprising the computer-implemented steps of:

- dividing said memory into a plurality of discrete pages;
- assigning a virtual page class, either shared or unshared, to each page of said memory, wherein shared pages are accessible by all of said plurality of program code portions, and unshared pages are accessible by a sub-group of said plurality of program code portions;
- marking each shared page with a first memory consistency model indicator; and
- executing each program code portion, including accessing a respective page of said memory, under a first memory consistency model if said page is marked with the first memory consistency model indicator, otherwise under a second memory consistency model.

7. A method as claimed in Claim 6, in which the first memory consistency model has stronger memory ordering constraints than the second memory consistency model.

8. A method as claimed in Claim 6 or Claim 7, wherein each page is initially neither shared nor unshared, and the step of assigning a virtual page class comprises:

- detecting an attempt to access a page by a first program code portion;
- assigning the unshared virtual page class to that page; and
- allowing access to that page by the said first program code portion.

9. A method as claimed in Claim 8, wherein the step of assigning a virtual page class further comprises:

- detecting an attempt to access an unshared page by a second program code portion, which second code portion is not part of the same sub-group of said plurality of program code portions as said first program code portion;

- changing from unshared to shared the virtual page class for that page; and

- allowing access to that page by the said second program code portion.

10. A method as claimed in Claim 8 or Claim 9, further comprising the step of periodically removing the assigned virtual page class from all shared pages.

11. A method as claimed in any of Claims 8 to 10, wherein said sub-group comprises two or more program code portions and the assigning of the unshared virtual page class follows detection of an attempt to access a page by any one of said two or more program code portions.

12. A method as claimed in any of Claims 6 to 11, wherein when said sub-group comprises more than one program code portion, and only a single one of those program code portions is permitted to execute at a time.

13. A method as claimed in any of Claims 6 to 12, wherein on termination of a program code portion, any page associated thereto by the unshared virtual page class has that virtual page class removed.

14. A method as claimed in of Claims 6 to 13, wherein two or more discrete pages of memory are grouped, and the assignment of a virtual page class to one page of the group causes the assignment of the same virtual page class to all pages of the group.

15. A computer program stored on a computer readable medium and loadable into the internal memory of a digital computer, comprising software code portions, when said program is run on a computer, for performing the method of any of claims 6 to 14.

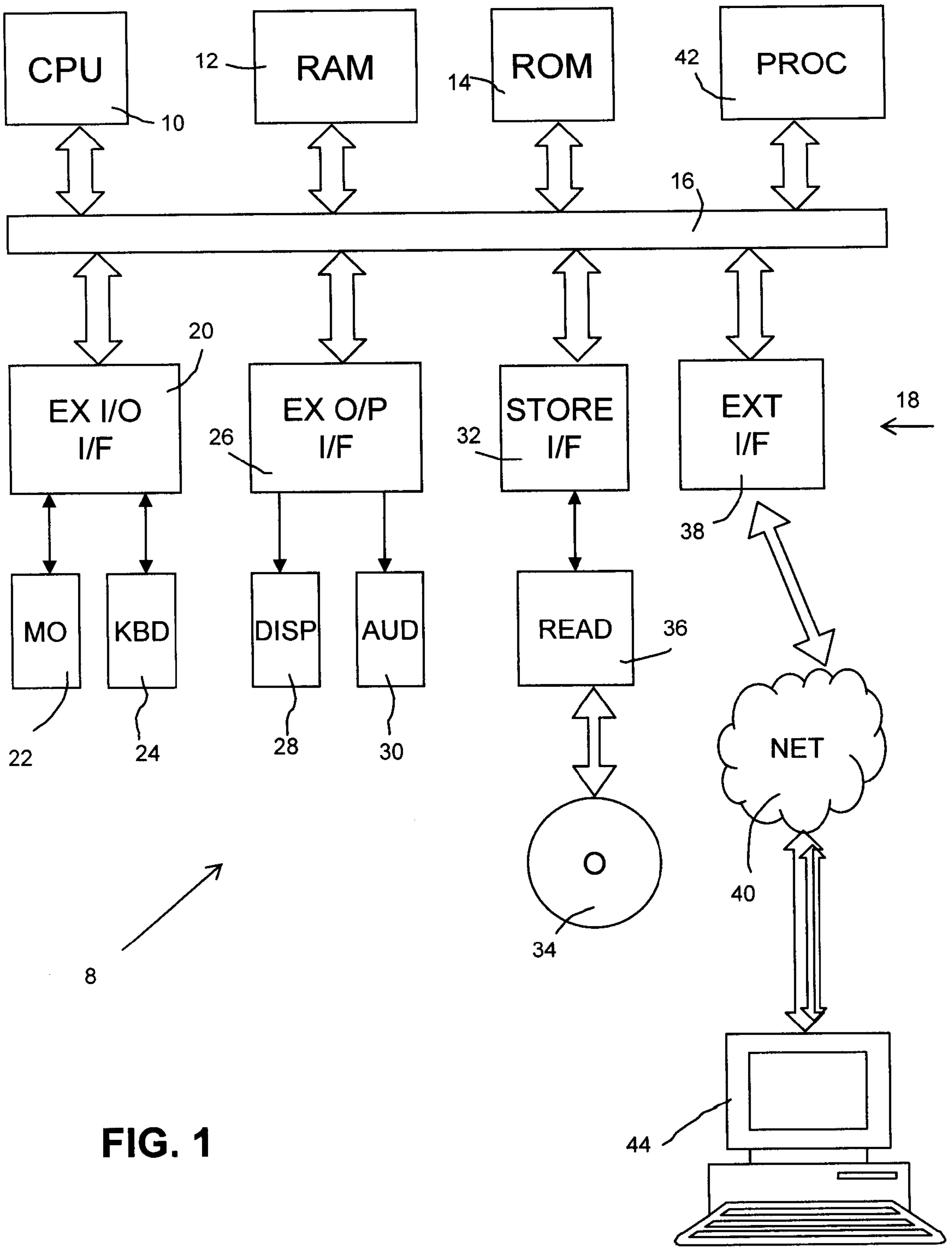
CPU
10
RAM
12
ROM
14
PROC
42
16
EX I/O I/F
20
EX O/P I/F
26
STORE I/F
32
EXT I/F
38
18
MO
22
KBD
24
DISP
28
AUD
30
READ
36
O
34
NET
40
44
8

FIG. 1

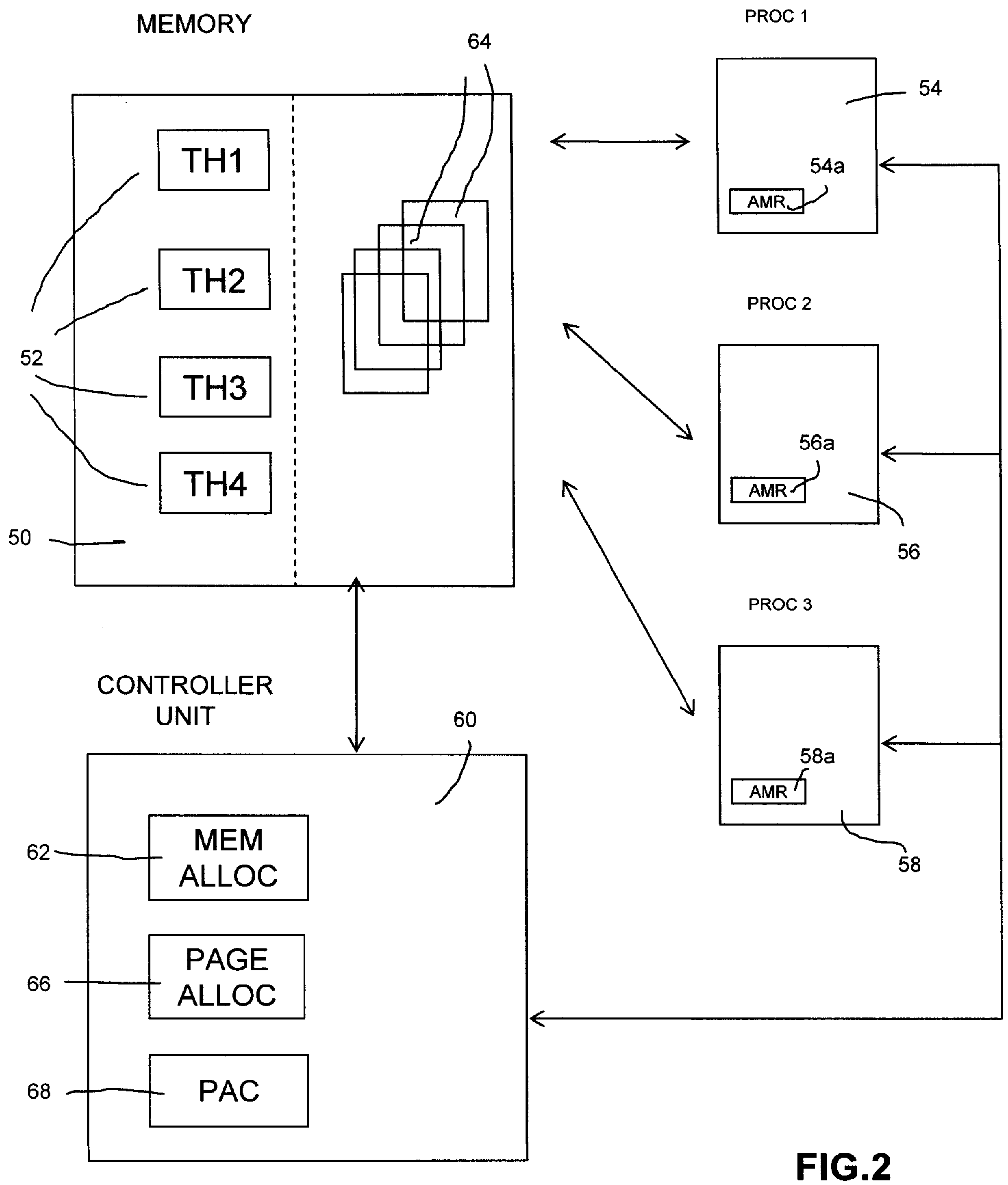
MEMORY
64
TH1
TH2
TH3
TH4
52
50
PROC 1
54
54a
AMR
PROC 2
56a
AMR
56
PROC 3
58a
AMR
58
CONTROLLER UNIT
60
62
MEM ALLOC
66
PAGE ALLOC
68
PAC

FIG.2

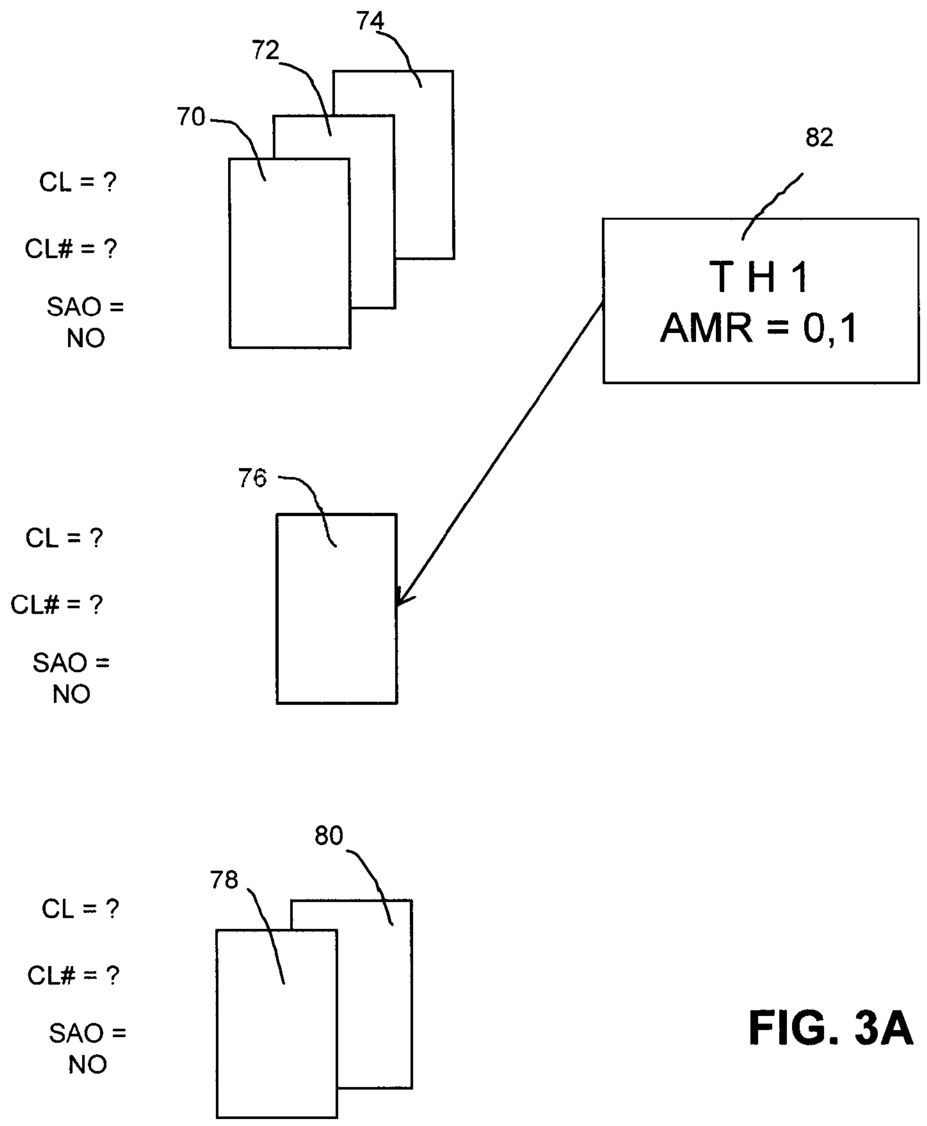

FIG. 3A

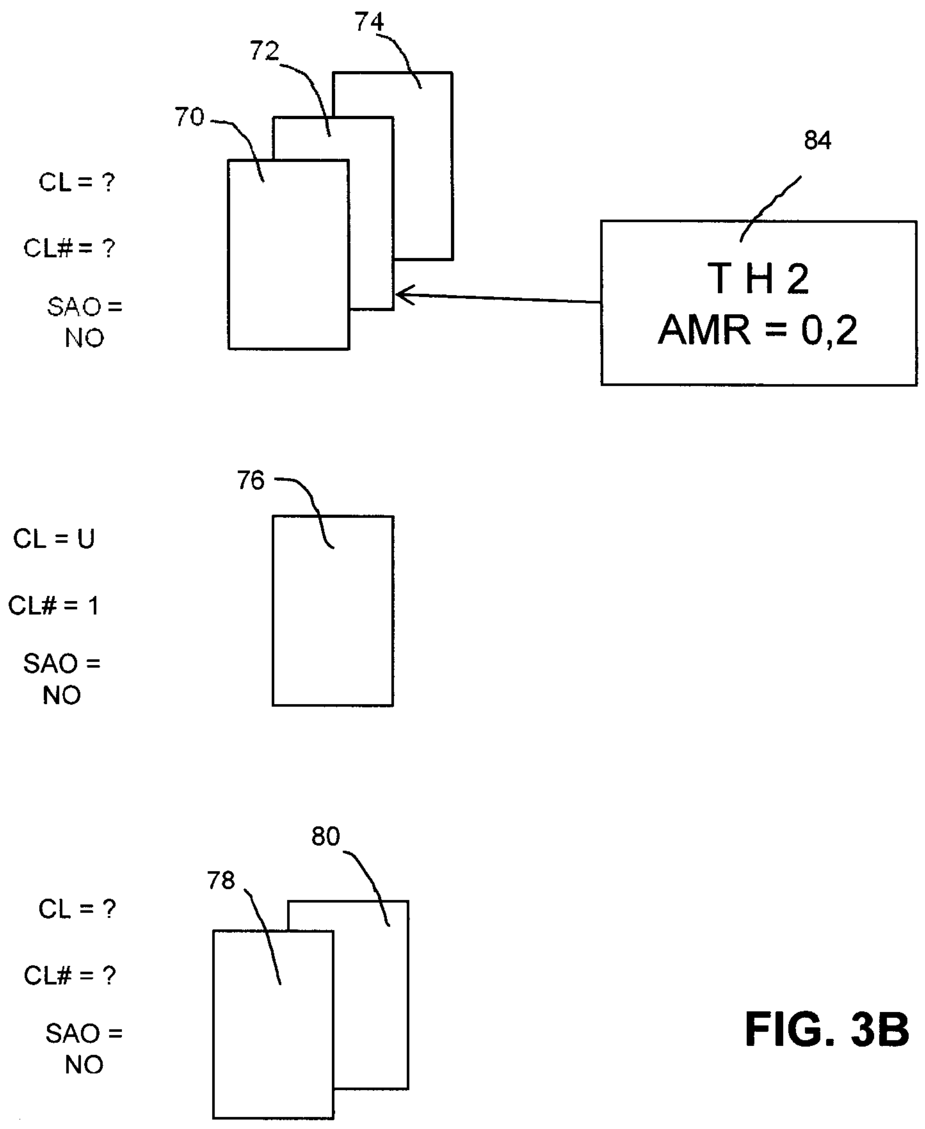
74
72
70
CL = ?
CL# = ?
SAO = NO
84
T H 2
AMR = 0,2
76
CL = U
CL# = 1
SAO = NO
80
78
CL = ?
CL# = ?
SAO = NO

FIG. 3B

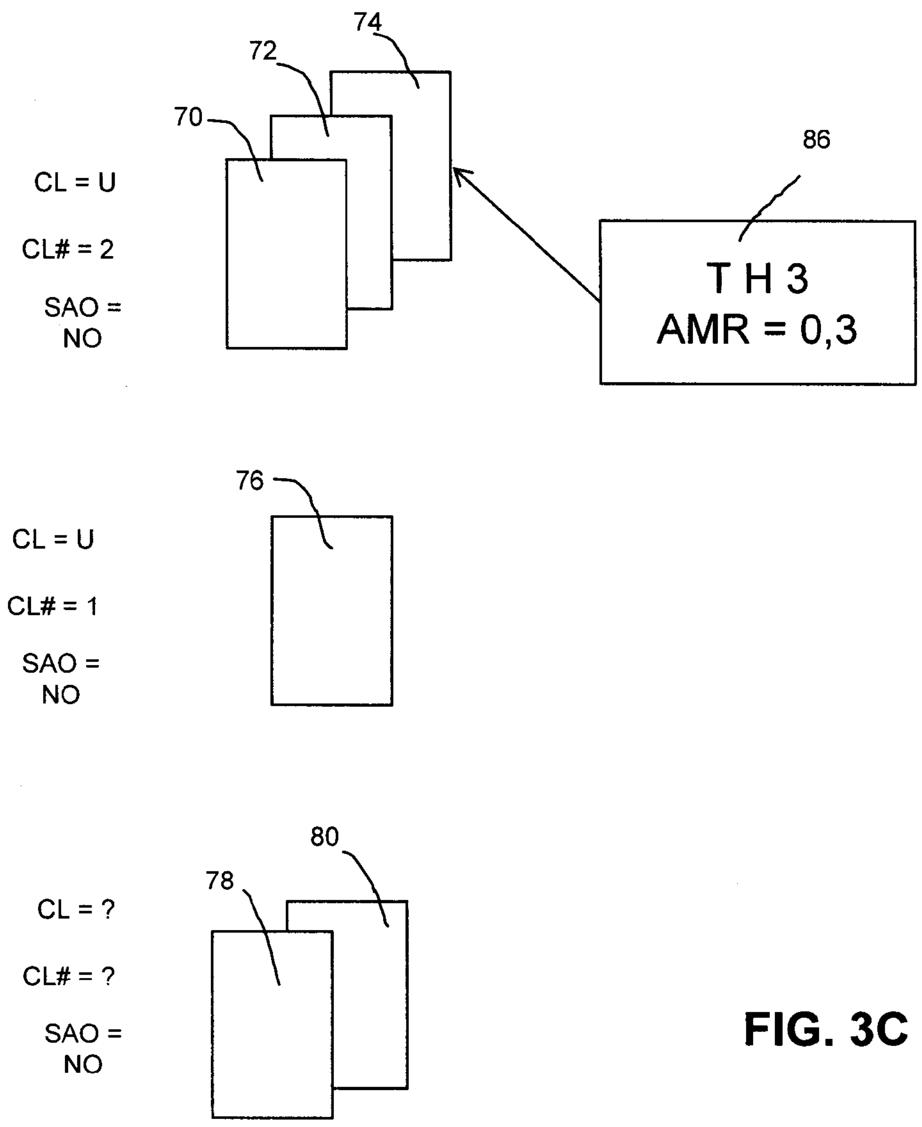
74
72
70
86
CL = U
CL# = 2
SAO = NO
T H 3
AMR = 0,3
76
CL = U
CL# = 1
SAO = NO
80
78
CL = ?
CL# = ?
SAO = NO

FIG. 3C

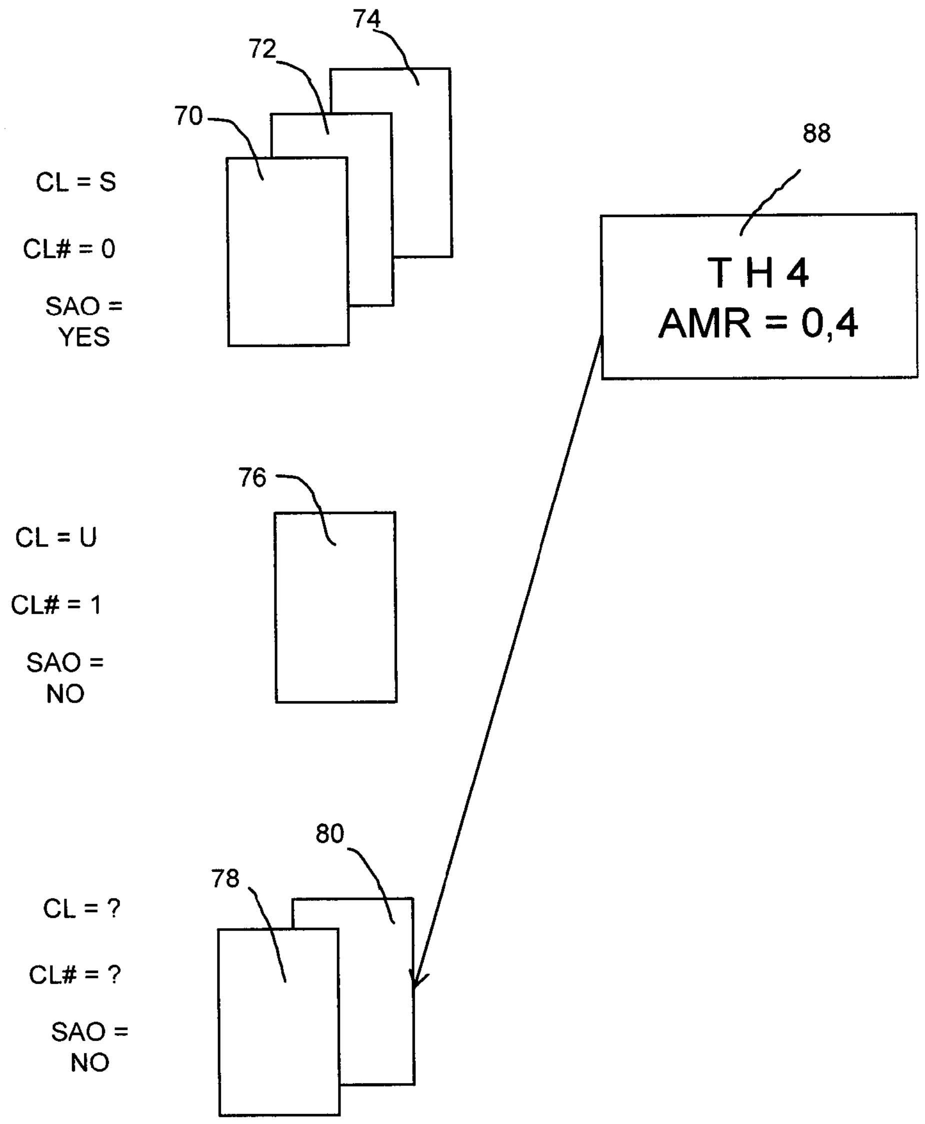
74
72
70
CL = S
CL# = 0
SAO = YES
88
T H 4
AMR = 0,4
76
CL = U
CL# = 1
SAO = NO
80
78
CL = ?
CL# = ?
SAO = NO

FIG. 3D

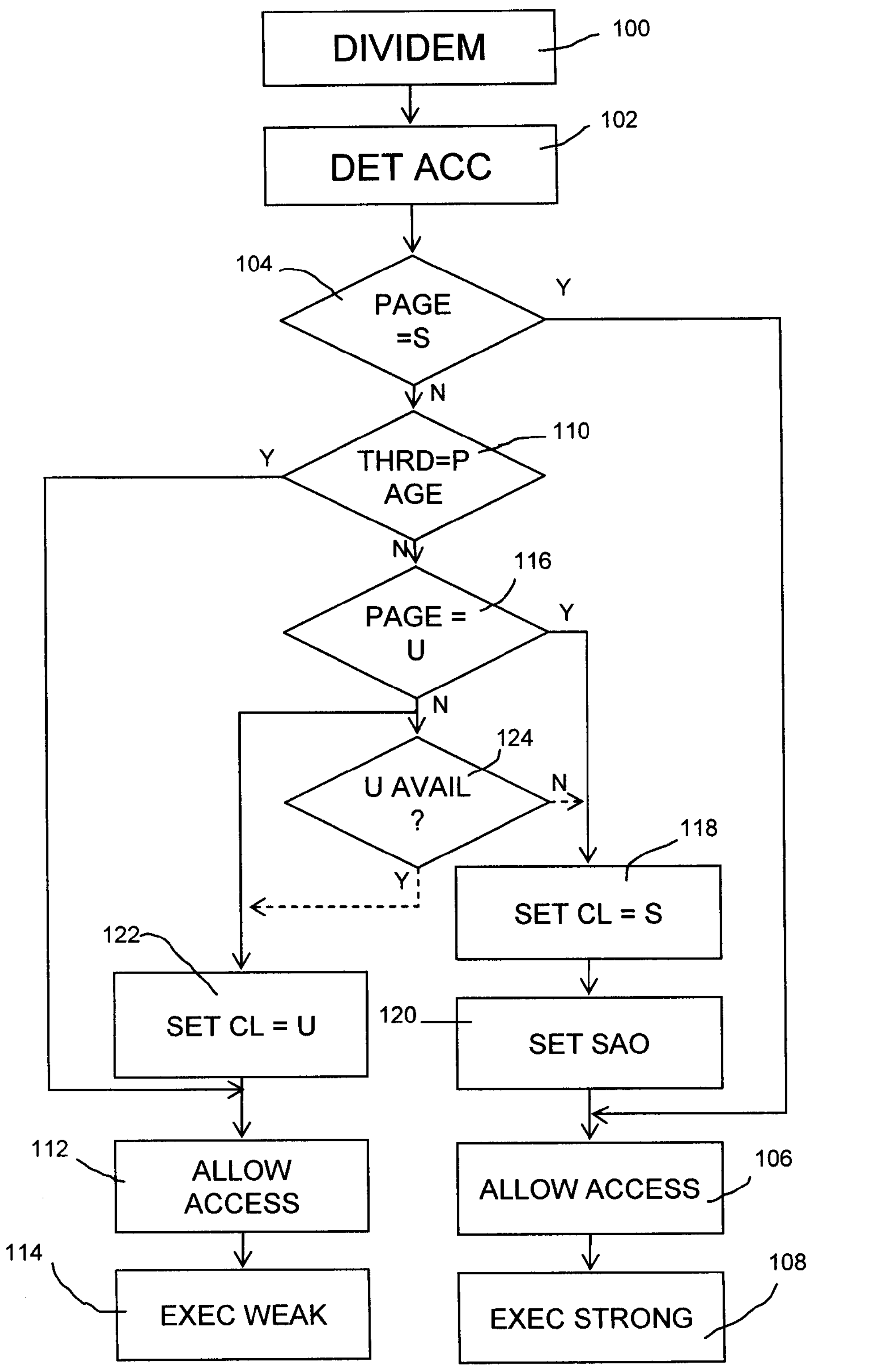
DIVIDEM
100
DET ACC
102
104
PAGE =S
Y
N
THRD=PAGE
110
Y
N
116
PAGE = U
Y
N
124
U AVAIL ?
N
Y
118
SET CL = S
122
SET CL = U
120
SET SAO
112
ALLOW ACCESS
ALLOW ACCESS
106
114
EXEC WEAK
EXEC STRONG
108

FIG.4

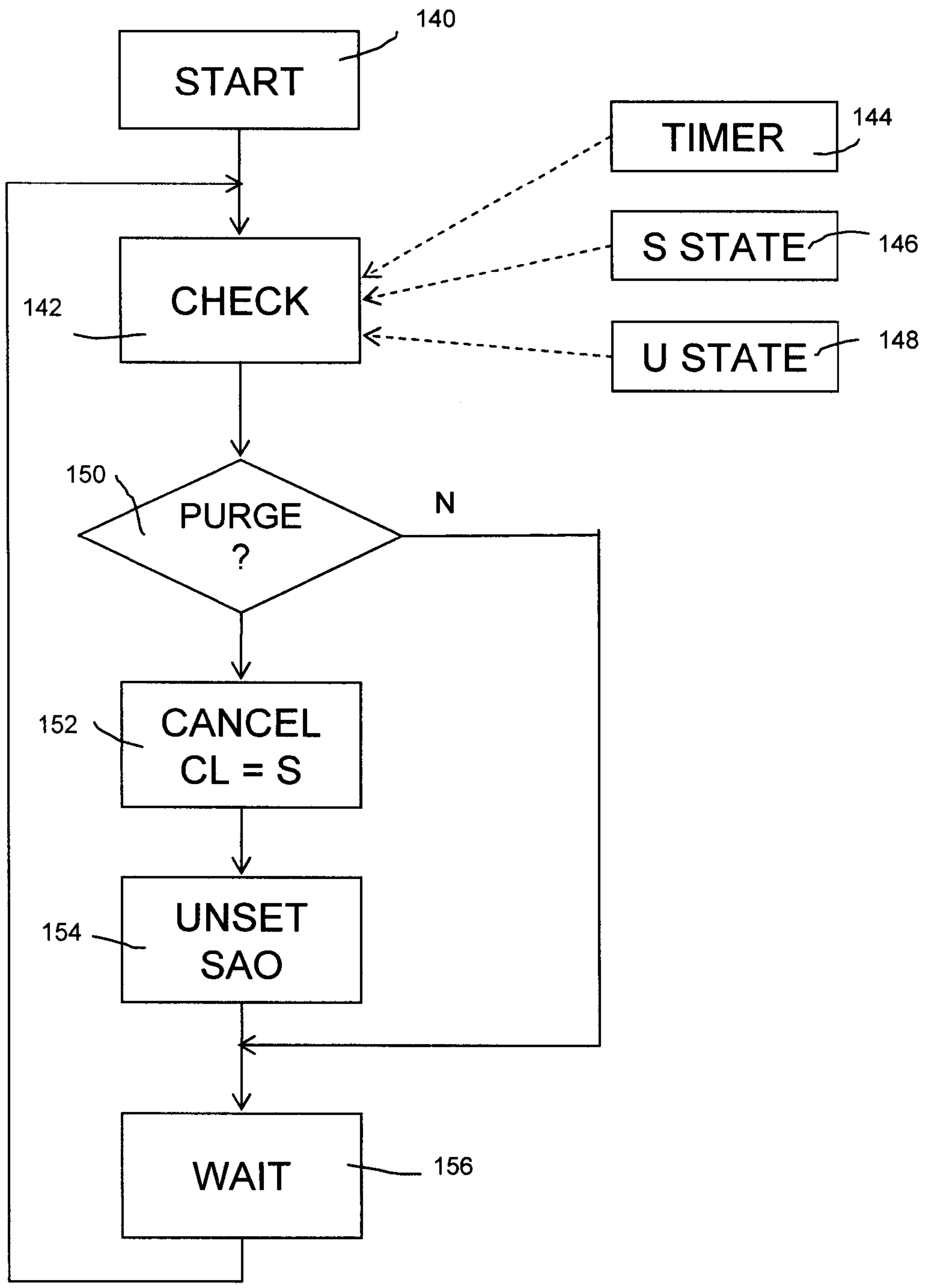
START
140
TIMER
144
S STATE
146
U STATE
148
CHECK
142
PURGE ?
150
N
CANCEL CL = S
152
UNSET SAO
154
WAIT
156

FIG. 5

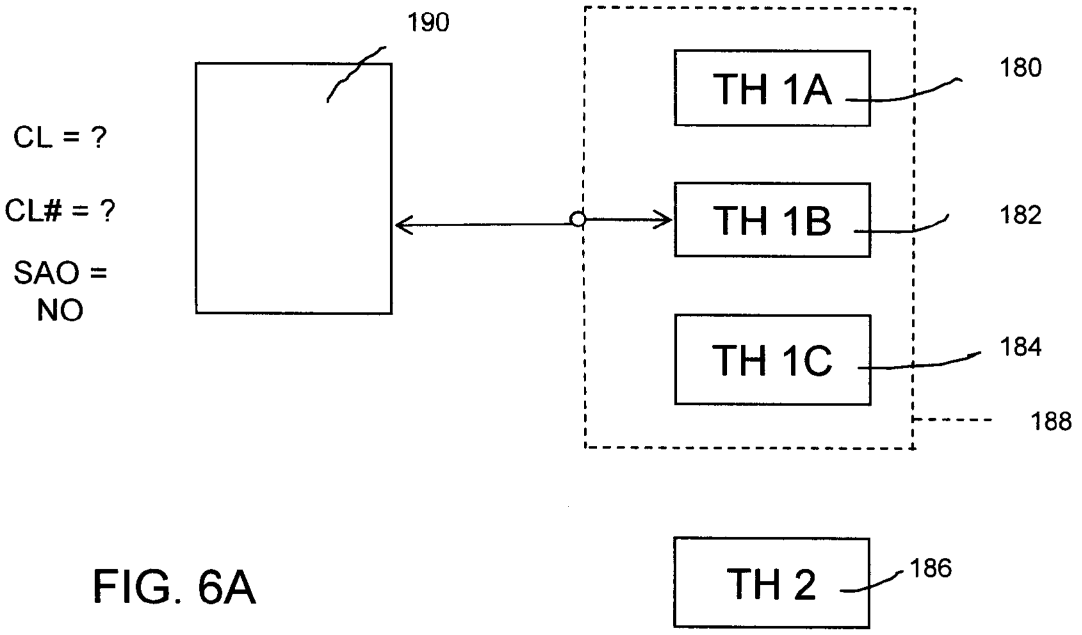
190
CL = ?
CL# = ?
SAO = NO
TH 1A
180
TH 1B
182
TH 1C
184
188
TH 2
186

FIG. 6A

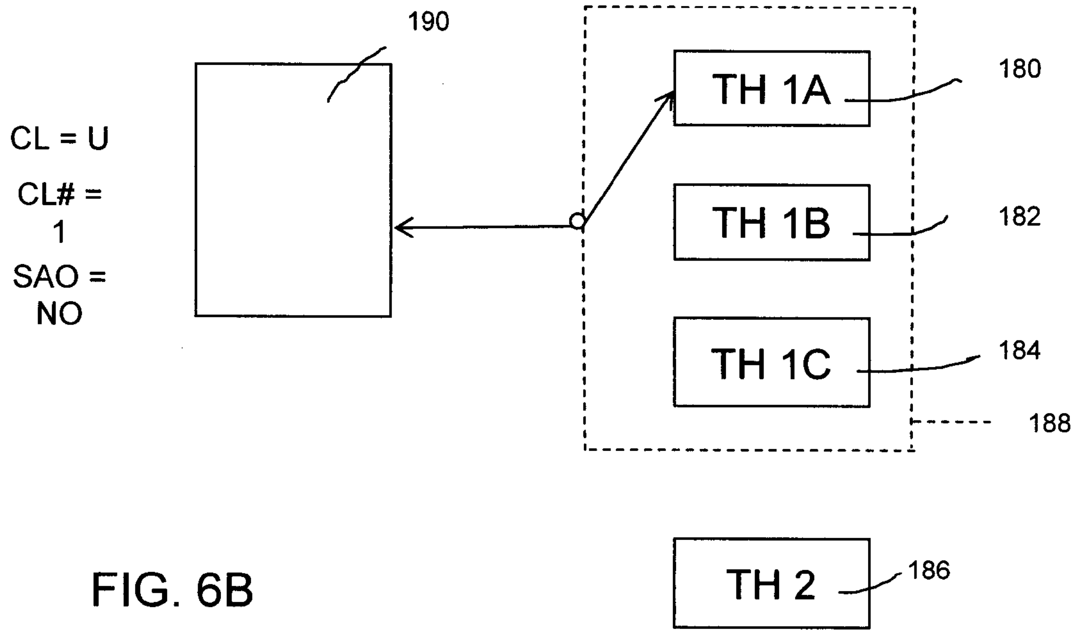
190
CL = U
CL# = 1
SAO = NO
TH 1A
180
TH 1B
182
TH 1C
184
188
TH 2
186

FIG. 6B

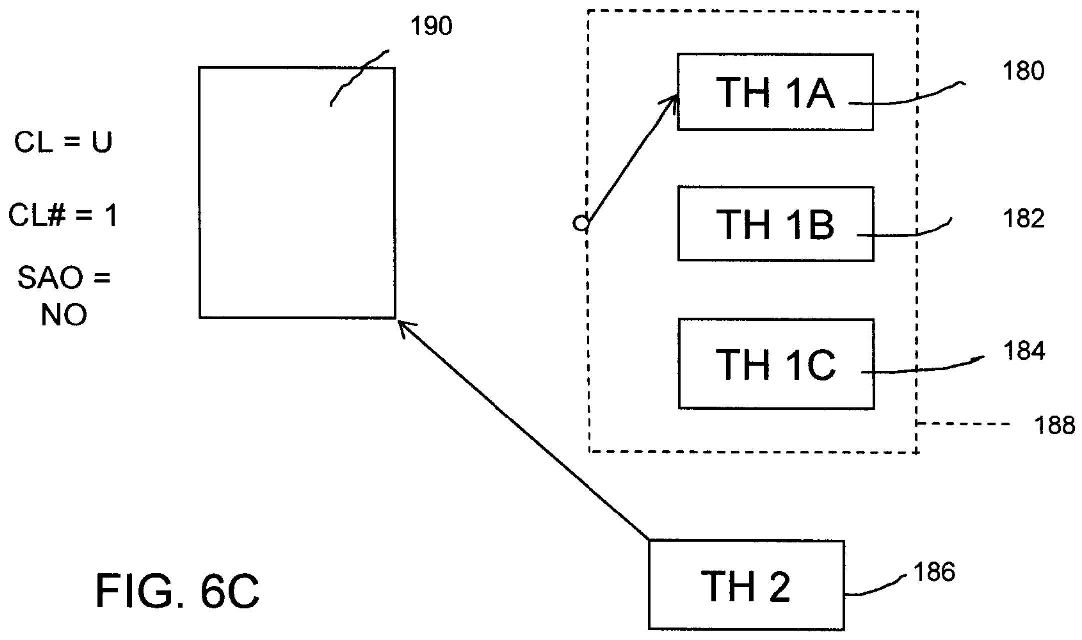
190
CL = U
CL# = 1
SAO = NO
TH 1A
180
TH 1B
182
TH 1C
184
188
TH 2
186

FIG. 6C

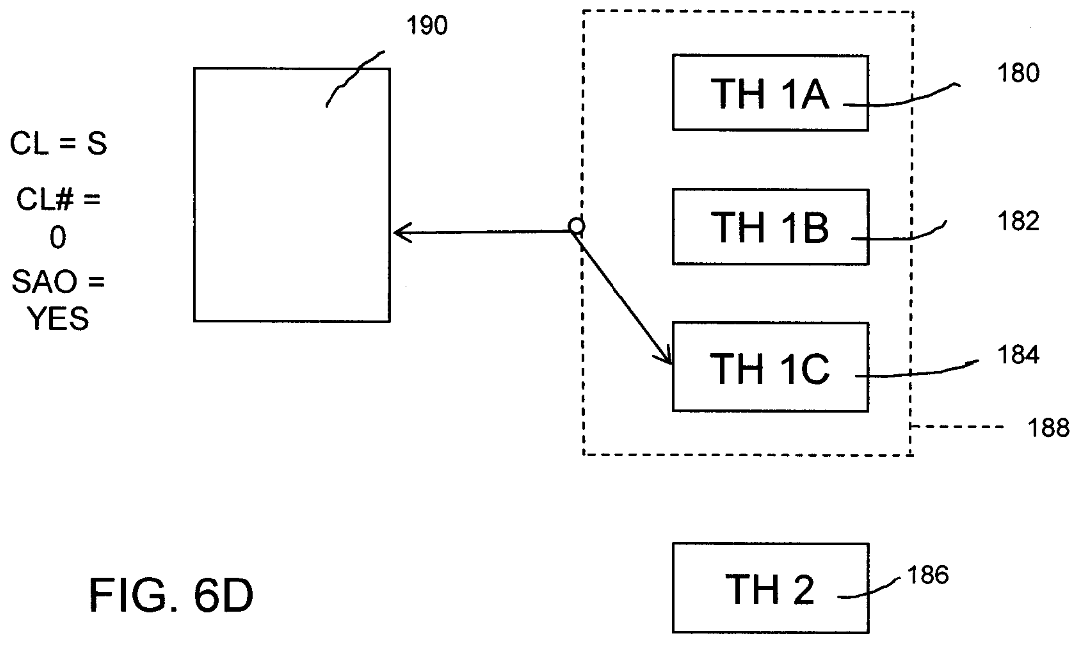
190
CL = S
CL# = 0
SAO = YES
TH 1A
180
TH 1B
182
TH 1C
184
188
TH 2
186

FIG. 6D

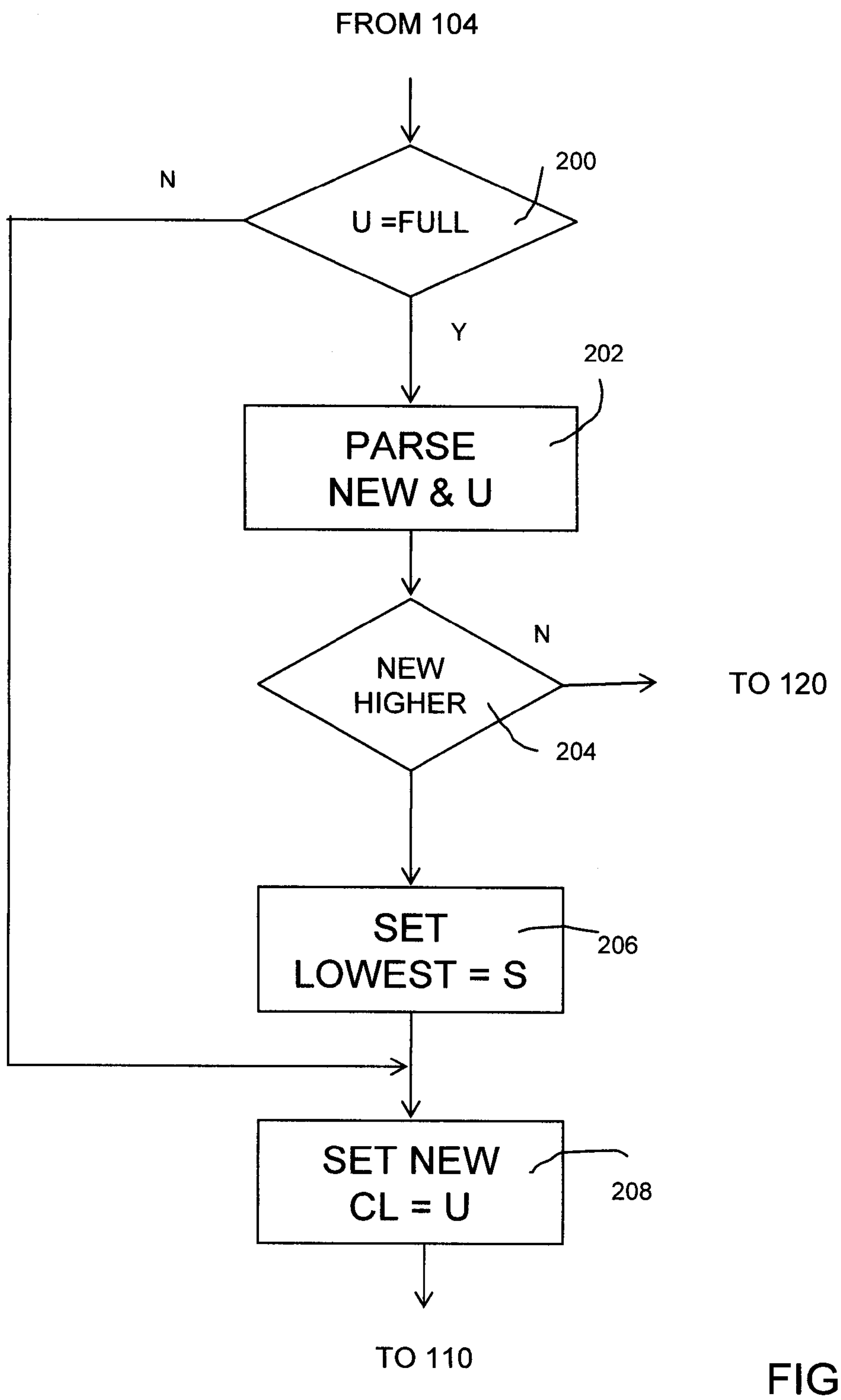
FROM 104
U =FULL
200
N
Y
PARSE NEW & U
202
NEW HIGHER
N
204
TO 120
SET LOWEST = S
206
SET NEW CL = U
208
TO 110

FIG. 7

INTERNATIONAL SEARCH REPORT

International application No.
PCT/IB2012/050154

A. CLASSIFICATION OF SUBJECT MATTER

Int.Cl. G06F12/10(2006.01)i, G06F12/00(2006.01)i

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)

Int.Cl. G06F12/10, G06F12/00

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

Published examined utility model applications of Japan 1922-1996
Published unexamined utility model applications of Japan 1971-2012
Registered utility model specifications of Japan 1996-2012
Published registered utility model applications of Japan 1994-2012

Electronic data base consulted during the international search (name of data base and, where practicable, 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.
X	GB 2444148 A (TRANSITIVE LIMITED) 2008.05.28, page 31 line 7 - page 32 line 36, Fig.lO-llB & JP 2010-510599 A & US 2008/0140971 Al & WO 2008/062225 Al & CN 101542441 A & KR 10-2009-0115118 A	1-15
A	US 2009/0037682 Al (INTERNATIONAL BUSINESS MACHINES CORPORATION) 2009.02.05, [0093]-[0110], FIG.23-26C & US 2009/0037907 Al & US 2009/0037908 Al & WO 2009/133072 Al	1-15

✓ Further documents are listed in the continuation of Box C. ☐ See patent family 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 application or patent 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

Date of the actual completion of the international search 09.04.2012	Date of mailing of the international search report 17.04.2012
Name and mailing address of the ISA/JP **Japan Patent Office** 3-4-3, Kasumigaseki, Chiyoda-ku, Tokyo 100-8915, Japan	Authorized officer 5U 9462 Yuji NAKANO Telephone No. +81-3-3581-1101 Ext. 3565

INTERNATIONAL SEARCH REPORT

International application No.
PCT/IB2012/050154

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2008/0168248 Al (INTERNATIONAL BUSINESS MACHINES CORPORATION) 2008.07.10, [0037]-[0039], FIG. 2 & US 2006/0036823 Al & US 2008/0263301 Al & WO 2006/015921 Al	1-15