

TRANSLATION OF INPUT/OUTPUT ADDRESSES TO MEMORY ADDRESSES

BACKGROUND

5 This invention relates, in general, to translating addresses in a computing environment, and in particular, to translating an input/output address to a memory address usable in accessing system memory of the environment.

10 System memory is accessible by read and write requests. These requests may come from various components of a computing environment, including central processing units, as well as adapters. Each request includes an address that is to be used to access system memory. This address, however, typically does not have a one-to-one correspondence with a physical location in system memory. Therefore, address translation is performed.

15 Address translation is used to translate an address that is provided in one form not directly usable in accessing system memory to another form that is directly usable in accessing a physical location in system memory. For instance, a virtual address included in a request provided by a central processing unit is translated to a real or absolute address in system
20 memory. As a further example, a Peripheral Component Interconnect (PCI) address provided in a request from an adapter may be translated to an absolute address in system memory.

US Publication No. 2008/0168208 A1, published July 10, 2008, Gregg, "I/O Adapter LPAR Isolation In A Hypertransport Environment With Assigned Memory Space Indexing A TVT
25 Via Unit Ids" describes a data processing system and method of isolating a plurality of I/O adapters in the system. The data processing system also comprises a set of processors, a host bridge, and a system bus connecting the set of processors and the host bridge. Each of the I/O adapters has a respective ID and send commands to the host bridge which include one or more of the IDs of the I/O adapters. In the preferred embodiment, these IDs are
30 HyperTransport defined Unit IDs, and the commands issued by the I/O Adapters include a Unit ID field containing one or more of the Unit IDs of I/O Adapters. The Unit IDs each are used to index a TVT to identify unique and independent system memory spaces

US Publication No. 2009/0182966 A1, published July 16, 2009, Greiner et al., "Dynamic Address Translation with Frame Management" discloses a frame management function defined for a machine architecture of a computer system. In one embodiment, a frame management instruction is obtained which identifies a first and second general register. The first general register contains a frame management field having a key field with access-protection bits and a block-size indication. If the block-size indication indicates a large block then an operand address of a large block of data is obtained from the second general register. The large block of data has a plurality of small blocks each of which is associated with a corresponding storage key having a plurality of storage key access-protection bits. If the block size indication indicates a large block, the storage key access-protection bits of each corresponding storage key of each small block within the large block is set with the access-protection bits of the key field.

BRIEF SUMMARY

The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a computer program product for translating addresses in a computing environment. The computer program product includes a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method. The method includes, for instance, obtaining an address from an adapter to be translated, the address comprising a plurality of bits, the plurality of bits comprising a first portion of bits and a second portion of bits; receiving an address range value that indicates a range of permitted addresses; validating the address using at least the first portion of bits and the received address range; and converting the address to a memory address usable in accessing memory of the computing environment, the converting ignoring the first portion of bits and using the second portion of bits to obtain address information from one or more levels of address translation tables to perform the converting.

Methods and systems relating to one or more aspects of the present invention are also described and claimed herein.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

One or more aspects of the present invention are particularly pointed out and distinctly claimed as examples in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

FIG. 1 depicts one embodiment of a computing environment to incorporate and use one or more aspects of the present invention;

FIG. 2 depicts one embodiment of further details of the system memory and I/O hub of FIG. 1, in accordance with an aspect of the present invention;

FIG. 3A depicts one embodiment of an overview of the logic to register a DMA (Direct Memory Access) address space for an adapter, in accordance with an aspect of the present invention;

FIG. 3B depicts one embodiment of various details of registering the DMA address space for the adapter, in accordance with an aspect of the present invention;

FIG. 4 depicts one embodiment of the logic to process a DMA operation, in accordance with an aspect of the present invention;

FIG. 5A depicts one example of the levels of translation employed when an entire address is used to index into address translation tables to translate the address and to access the page;

FIG. 5B depicts one example of levels of translation employed when a part of the address is ignored when indexing into the address translation tables, in accordance with an aspect of the present invention;

FIG. 5C depicts examples of various CPU DAT compatible formats usable in accordance with one or more aspects of the present invention;

FIG. 5D depicts examples of various I/O extended address translation formats usable in accordance with one or more aspects of the present invention;

FIG. 6A depicts one embodiment of a Modify PCI Function Controls instruction used in accordance with an aspect of the present invention;

FIG. 6B depicts one embodiment of a field used by the Modify PCI Function Controls instruction of FIG. 6A, in accordance with an aspect of the present invention;

5 FIG. 6C depicts one embodiment of another field used by the Modify PCI Function Controls instruction of FIG. 6A, in accordance with an aspect of the present invention;

FIG. 6D depicts one embodiment of the contents of a function information block (FIB) used in accordance with an aspect of the present invention;

10 FIG. 7 depicts one embodiment of an overview of the logic of the Modify PCI Function Controls instruction, in accordance with an aspect of the present invention;

FIG. 8 depicts one embodiment of the logic associated with a register I/O address translation parameters operation that may be specified by the Modify PCI Function Controls instruction, in accordance with an aspect of the present invention;

15 FIG. 9 depicts one embodiment of the logic associated with an unregister I/O address translation parameters operation that may be specified by the Modify PCI Function Controls instruction, in accordance with an aspect of the present invention;

FIG. 10 depicts one embodiment of a computer program product incorporating one or more aspects of the present invention;

20 FIG. 11 depicts one embodiment of a host computer system to incorporate and use one or more aspects of the present invention;

FIG. 12 depicts a further example of a computer system to incorporate and use one or more aspects of the present invention;

FIG. 13 depicts another example of a computer system comprising a computer network to incorporate and use one or more aspects of the present invention;

25 FIG. 14 depicts one embodiment of various elements of a computer system to incorporate and use one or more aspects of the present invention;

FIG. 15A depicts one embodiment of the execution unit of the computer system of FIG. 14 to incorporate and use one or more aspects of the present invention;

30 FIG. 15B depicts one embodiment of the branch unit of the computer system of FIG. 14 to incorporate and use one or more aspects of the present invention;

FIG. 15C depicts one embodiment of the load/store unit of the computer system of FIG. 14 to incorporate and use one or more aspects of the present invention; and

FIG. 16 depicts one embodiment of an emulated host computer system to incorporate and use one or more aspects of the present invention.

DETAILED DESCRIPTION

5

10

In accordance with an aspect of the present invention, a capability is provided for translating addresses in a computing environment. In one example, the addresses being translated are addresses provided by an adapter (referred to herein as input/output (I/O) addresses), which are to be translated to addresses usable in accessing system memory. To perform the address translation, a number of levels of address translation is used, and the number of levels is based on, for instance, a size of the memory address space assigned to the adapter, a size of one or more address translation tables used in the translation, and/or a size of the page (or other unit of memory) to be accessed.

15

20

The address being translated includes a plurality of bits, and in one embodiment, only a portion of those bits are used to index into the address translation tables to obtain the translated address. The other bits are ignored for translation. For instance, the address includes high order bits and low order bits (based on the size of the assigned address space). In this example, the low order bits are used to index into the address translation tables, including the page table, and to index into the page itself. The high order bits (regardless of value; i.e., zero or non-zero) are ignored for translation and are not used to index into the address translation tables. This reduces the number of levels of address translation tables used for translating. (The indication of low order and high order bits is independent of how the bits are numbered.)

25

30

In one example, the translation is performed while providing protection suitable for an enterprise class server, such as a System z[®] server. As an example, a full address (e.g., the entire 64-bit address) is used in accessing memory; however, only a portion of the address is used for translation. Another portion of the address, which is ignored for translation, is used for validation (e.g., a range check), along with at least a portion of the address used for translation. By using only a portion of the address for translation, address translation look-ups are minimized. This is accomplished by performing a number of levels of translation

based on, for instance, the size of the DMA address space registered for access by the adapter, rather than the size of the address itself.

One embodiment of a computing environment to incorporate and use one or more aspects of the present invention is described with reference to FIG. 1. In one example, a computing environment 100 is a System z[®] server offered by International Business Machines Corporation. System z[®] is based on the z/Architecture[®] offered by International Business Machines Corporation. Details regarding the z/Architecture[®] are described in an IBM[®] publication entitled, "z/Architecture Principles of Operation," IBM Publication No. SA22-7832-07, February 2009. IBM[®], System z[®] and z/Architecture[®] are registered trademarks of International Business Machines Corporation, Armonk, New York. Other names used herein may be registered trademarks, trademarks or product names of International Business Machines Corporation or other companies.

In one example, computing environment 100 includes one or more central processing units (CPUs) 102 coupled to a system memory 104 (a.k.a., main memory) via a memory controller 106. To access system memory 104, a central processing unit 102 issues a read or write request that includes an address used to access system memory. The address included in the request is typically not directly usable to access system memory, and therefore, it is translated to an address that is directly usable in accessing system memory. The address is translated via a translation mechanism (XLATE) 108. For example, the address is translated from a virtual address to a real or absolute address using, for instance, dynamic address translation (DAT).

The request, including the translated address, is received by memory controller 106. In one example, memory controller 106 is comprised of hardware and is used to arbitrate for access to the system memory and to maintain the memory's consistency. This arbitration is performed for requests received from CPUs 102, as well as for requests received from one or more adapters 110. Like the central processing units, the adapters issue requests to system memory 104 to gain access to the system memory.

In one example, adapter 110 is a Peripheral Component Interconnect (PCI) or PCI Express (PCIe) adapter that includes one or more PCI functions. A PCI function issues a request that requires access to system memory. The request is routed to an input/output hub 112 (e.g., a PCI hub) via one or more switches (e.g., PCIe switches) 114. In one example, the
5 input/output hub is comprised of hardware, including one or more state machines.

As used herein, the term adapter includes any type of adapter (e.g., storage adapter, network adapter, processing adapter, PCI adapter, cryptographic adapter, other type of input/output adapters, etc.). In one embodiment, an adapter includes one adapter function. However, in
10 other embodiments, an adapter may include a plurality of adapter functions. One or more aspects of the present invention are applicable whether an adapter includes one adapter function or a plurality of adapter functions. Moreover, in the examples presented herein, adapter is used interchangeably with adapter function (e.g., PCI function) unless otherwise noted.

The input/output hub includes, for instance, a root complex 116 that receives the request from a switch. The request includes an input/output address to be translated, and thus, the root complex provides the address to an address translation and protection unit 118. This unit is, for instance, a hardware unit that translates the I/O address to an address directly
20 usable to access system memory 104, as described in further detail below.

The request initiated from the adapter, including the translated address, is provided to memory controller 106 via, for instance, an I/O-to-memory bus 120. The memory controller performs its arbitration and forwards the request with the translated address to the system
25 memory at the appropriate time.

Further details regarding the system memory and the input/output hub are described with reference to FIG. 2. In this embodiment, the memory controller is not shown. However, the I/O hub may be coupled to the system memory directly or via a memory controller. In one
30 example, system memory 104 includes one or more address spaces 200. An address space is a particular portion of system memory that has been assigned to a particular component of the computing environment, such as a particular adapter. In one example, the address space

is accessible by direct memory access (DMA) initiated by the adapter, and therefore, the address space is referred to in the examples herein as a DMA address space. However, in other examples, direct memory access is not used to access the address space.

5 Further, in one example, system memory 104 includes address translation tables 202 used to translate an address from one that is not directly usable to access system memory to one that is directly usable. In one embodiment, there are one or more address translation tables assigned to a DMA address space, and those one or more address translation tables are configured based on, for instance, the size of the address space to which it is assigned, the
10 size of the address translation tables themselves, and/or the size of the page (or other unit of memory).

In one example, there is a hierarchy of address translation tables. For instance, as shown in FIG. 2, there is a first-level table 202a (e.g., a segment table) pointed to by an IOAT pointer
15 218 (described below), and a second, lower level table 202b (e.g., a page table) pointed to by an entry 206a of the first-level table. One or more bits of a received address 204 are used to index into table 202a to locate a particular entry 206a, which indicates a particular lower level table 202b. Then, one or more other bits of address 204 are used to locate a particular entry 206b in that table. In this example, that entry provides the address used to locate the
20 correct page and additional bits in address 204 are used to locate a particular location 208 in the page to perform a data transfer. That is, the address in entry 206b and selected bits of received PCI address 204 are used to provide the address directly usable to access system memory. For instance, the directly usable address is formed from a concatenation of high order bits of the address in entry 206b (e.g., bits 63:12, in a 4k page example) and selected
25 low order bits from the received PCI address (e.g., bits 11:0 for a 4k page).

In one example, it is an operating system that assigns a DMA address space to a particular adapter. This assignment is performed via a registration process, which causes an initialization (via, e.g., trusted software) of a device table entry 210 for that adapter. The
30 device table entry is located in a device table 211 located in I/O hub 112. For example, device table 211 is located within the address translation and protection unit of the I/O hub.

In one example, device table entry 210 includes a number of fields, such as the following:

Format 212: This field includes a plurality of bits to indicate various information, including, for instance, the address translation format (including level) of an upper level table of the address translation tables (e.g., in the example above, the first-level table);

Page Size 213: This field indicates a size of a page (or other unit of memory) to be accessed;

PCI base address 214 and PCI limit 216: These values provide a range used to define a DMA address space and verify a received address (e.g., PCI address) is valid; and

IOAT (Input/Output Address Translation) pointer 218: This field includes a pointer to the highest level of address translation table used for the DMA address space.

In other embodiments, the DTE may include more, less or different information.

In one embodiment, the device table entry to be used in a particular translation is located using a requestor identifier (RID) located in a request issued by a PCI function 220 associated with an adapter (and/or by a portion of the address). The requestor id (e.g., a 16-bit value specifying, for instance, a bus number, device number and function number) is included in the request, as well as the I/O address (e.g., a 64-bit PCIe address) to be used to access system memory. The request, including the RID and I/O address, are provided to, e.g., a contents addressable memory (CAM) 230 via, e.g., a switch 114, which is used to provide an index value. For instance, the CAM includes multiple entries, with each entry corresponding to an index into the device table. Each CAM entry includes the value of a RID. If, for instance, the received RID matches the value contained in an entry in the CAM, the corresponding device table index is used to locate the device table entry. That is, the output of the CAM is used to index into device table 211 to locate device table entry 210. If there is no match, the received packet is discarded with no access to system memory being performed. (In other embodiments, a CAM or other look-up is not needed and the RID is used as the index.)

Subsequently, fields within the device table entry are used to ensure the validity of the address and the configuration of the address translation tables. For example, the inbound

address in the request is checked by the hardware of the I/O hub (e.g., the address translation and protection unit) to ensure that it is within the bounds defined by PCI base address 214 and PCI limit 216 stored in the device table entry located using the RID of the request that provided the address. This ensures that the address is within the range previously registered and for which address translation tables are validly configured.

In accordance with an aspect of the present invention, in one embodiment, to translate an I/O address (i.e., an address provided by an adapter or another component of an I/O subsystem) to a system memory address (i.e., an address directly usable to access system memory), initially, a registration process is performed. This registration process registers a particular address space with a specific requestor, such as a specific adapter or adapter function. One example of an overview of this registration process is described with reference to FIG. 3A.

Initially, an operating system executing within one of the central processing units coupled to system memory determines a size and location of the address space that the adapter is to access, STEP 300. In one example, the size of the address space is determined by the PCI base address and PCI limit set by the operating system. The operating system determines the base and limit using one or more criteria. For instance, if the operating system wishes to have PCI addresses map directly to CPU virtual addresses, then the base and limit are set as such. In a further example, if additional isolation between adapters and/or operating system images is desired, then the addresses being used are selected to provide non-overlapping and disjoint address spaces. The location is also specified by the operating system, and is based, for instance, on the characteristics of the adapter.

Thereafter, one or more address translation tables are created to cover that DMA address space, STEP 302. As examples, the tables can be compatible with CPU address translation tables or a unique format can be provided that is supported by the input/output hub. In one example, the creation includes building the tables and placing the appropriate addresses within the table entries. As an example, one of the translation tables is a 4k page table having 512 64-bit entries, and each entry includes a 4k page address compatible with the assigned address space.

Thereafter, the DMA address space is registered for the adapter, STEP 304, as described in further detail with reference to FIG. 3B. In this example, it is assumed there is one PCI function per adapter, and therefore, one requestor ID per adapter. This logic is performed, for instance, by a central processing unit coupled to the system memory, responsive to an operating system request.

Initially, in one embodiment, an available device table entry is selected that is to correspond to the requestor ID of the adapter, STEP 310. That is, the requestor ID will be used to locate a device table entry.

Additionally, the PCI base address and the PCI limit are stored in the device table entry, STEP 312. Further, the format of the highest level address translation table is also stored in the device table entry (e.g., format field), STEP 314, as well as the input/output address translation (IOAT) pointer used to point to the highest level address translation table, STEP 316. This completes the registration process.

Responsive to performing registration, a DMA address space and corresponding address translation tables are ready for use, as well as a device table entry. Details regarding processing a request issued by a requestor, such as an adapter, to access system memory are described with reference to FIG. 4. The processing described below is performed by the I/O hub. In one example, it is the address translation and protection unit that performs the logic. In one embodiment, initially, a DMA request is received at the input/output hub, STEP 400. For instance, a PCI function issues a request that is forwarded to the PCI hub via, for instance, a PCI switch. Using the requestor ID in the request, the appropriate device table entry is located, STEP 402. Thereafter, a determination is made as to whether the device table entry is valid, INQUIRY 404. In one example, validity is determined by checking a validity bit in the entry itself. This bit is set, for instance, in response to execution of an enable function request by the operating system. If enabled, the bit is set to, e.g., one (i.e., valid); otherwise, it remains at zero (i.e., invalid). In a further example, the bit may be set when the registration process is complete.

If the device table entry is invalid, an error is presented, STEP 405. Otherwise, a further determination is made as to whether the PCI address provided in the request is less than the PCI base address stored in the device table entry, INQUIRY 406. If it is, then the address is outside a valid range and an error is provided, STEP 407. However, if the PCI address is greater than or equal to the base address, then another determination is made as to whether the PCI address is greater than the PCI limit value in the device table entry, INQUIRY 408. If the PCI address is greater than the limit, then once again, an error is presented since the address is outside the valid range, STEP 409. However, if the address is within a valid range, then processing continues.

In one example, the format provided in the device table entry is used to determine the PCI address bits in the address to be used for address translation, STEP 410. For instance, if the format indicates the upper level table is a first-level table with 4k pages, then bits 29:21 of the address are used to index into the first-level table; bits 20:12 are used to index into the page table; and bits 11:0 are used to index into the 4k page. The bits used depend on how many bits are needed to index into the given size page or table. For instance, for a 4k page with byte level addressing, 12 bits are used to address 4096 bytes; and for a 4k page table with 512 entries, 8 bytes each, 9 bits are used to address 512 entries, etc.

Next, the PCI hub fetches the appropriate address translation table entry, STEP 412. For instance, initially, the highest level translation table is located using the IOAT pointer of the device table entry. Then, bits of the address (those after the high order bits used for validity and not translation; e.g., bits 29:21 in the above example) are used to locate the particular entry within that table.

A determination is then made based, for instance, on the format provided in the device table entry, as to whether the located address translation entry has a correct format, INQUIRY 414. For instance, the format in the device table entry is compared with a format indicated in the address translation entry. If equal, then the format in the device table entry is valid. If not, an error is provided, STEP 415; otherwise, processing continues with a determination as to whether this is the last table to be processed, INQUIRY 416. That is, a determination is made as to whether there are other address translation tables needed to obtain the real or

absolute address or whether the lowest level table entry has been located. This determination is made based on the provided format and size of the tables already processed. If it is not the last table, then processing continues with STEP 412. Otherwise, the I/O hub continues processing to enable a fetch or store of the data at the translated address, STEP 418. In one example, the I/O hub forwards the translated address to the memory controller, which uses the address to fetch or store data at the DMA location designated by the translated address.

As described above, in accordance with an aspect of the present invention, the number of levels of translation, and therefore, the number of fetches required to perform translation are reduced. This is accomplished by, for instance, ignoring the high order bits of an address during translation and only using the low order bits to traverse the translation tables, which are based, for instance, on the size of the DMA address space assigned to the adapter. The use of a partial address versus the full address is further shown in the following examples.

Referring initially to FIG. 5A, an example is depicted in which the entire address is used in address translation/memory access. With this prior technique, six levels of translation tables are needed, including the page table. The beginning of the highest level table (e.g., the 5th-level table in this example) is pointed to by an IOAT pointer, and then bits of the PCI address are used to locate an entry in the table. Each translation table entry points to the start of a lower level translation table or to a page (e.g., an entry in the 5th-level table points to the start of a 4th-level table, etc.)

In this example, the DMA address space (DMAAS) is 6M in size, and each table is 4k bytes having a maximum of 512 8-byte entries (except the 5th level table which has 128 entries based on the size of the address). The address is, for instance, 64 bits: FFFF C000 0009 C600. The beginning of the 5th-level table is pointed to by the IOAT pointer and bits 63:57 of the PCI address are used to index into the 5th-level table to locate the beginning of the 4th-level table; bits 56:48 of the PCI address are used to index into the 4th-level table to locate the beginning of the 3rd-level table; bits 47:39 are used to index into the 3rd-level table to locate the beginning of the 2nd-level table; bits 38:30 are used to index into the 2nd-level table to locate the beginning of the 1st-level table; bits 29:21 are used to index into the 1st-level table to locate the beginning of the page table; bits 20:12 are used to index the page

table to locate the beginning of the page; and bits 11:0 are used to locate the entry in the 4k page. Thus, in this example, all of the address bits are used for translation/access.

This is in contrast to the example in FIG. 5B, in which the address space is the same size (e.g., 6M) and the address is the same, but the translation technique ignores some of the address bits during translation. In this example, bits 63:30 of the address are ignored for translation. The IOAT pointer points to the beginning of the 1st-level table and bits 29:21 of the PCI address are used to index into the 1st-level table to locate the beginning of the page table; bits 20:12 are used to index into the appropriate page table to locate the beginning of the page; and bits 11:0 are used to index into the 4k page.

As shown, 1st-level table 500 includes three entries 502, each providing an address to one of the three page tables 504. The number of page tables needed, and therefore, the number of other level tables, depends, for instance, on the size of the DMA address space, the size of the translation tables, and/or the size of the pages. In this example, the DMA address space is 6M, and each page table is 4k, having up to 512 entries. Therefore, each page table can map up to 2M of memory ($4k \times 512$ entries). Thus, three page tables are needed for the 6M address space. The 1st-level table is able to hold the three entries, one for each page table, and thus, no further levels of address translation tables are needed, in this example.

In a further embodiment, there may be different formats of address translation tables used for address translation, and there may be variations within the formats. Thus, there are variations in the bits used to index into a particular table or page. Some of these examples are described with reference to FIGs. 5C and 5D.

For instance, one format is a CPU DAT compatible format in which the translation tables are compatible with translation tables used for CPU DAT translations. There may be various CPU DAT compatible formats, examples of which are described with reference to FIG. 5C. As shown, one CPU DAT compatible format is a 4k page CPU DAT compatible format 550, and another is a 1M page CPU DAT compatible format 552, as examples. The number of bits shown are the number of address bits used to index into that page or table (or otherwise locate an entry in that page or table). For instance, 12 bits 554 of a PCI address are used as a

byte offset into a 4k page 556; 8 bits 558 are used as an index into a page table 560; 11 bits 562 are used as an index into a segment table 564, etc. Located under the designated address translation table is the maximum size of the address space supported by that address translation table. For instance, page table 560 supports a 1M DMA address space; segment table 564 supports a 2G DMA address space, etc. In this figure, as well as in FIG. 5D, K =
5 kilobytes, M = megabytes, G = gigabytes, T = terabytes, P = petabytes, and E = exabytes.

As depicted, as the size of the page increases, the number of levels of translation tables decreases. For instance, for 4k page 556, a page table is needed, but it is not needed for the
10 1M page. Other examples and variations are possible.

Another address translation format is an I/O extended address translation format in which extended address translation tables are used. Various examples of I/O extended address translation formats are depicted in FIG. 5D. For example, the following formats are shown:
15 a 4k address translation table with 4k pages 570; 1M address translation tables with 4k pages 572; and 1M address translation tables with 1M pages 574. As with the CPU DAT compatible formats, the number of bits listed are those bits used to locate an entry in the particular table. For instance, at reference number 576, the 12 bits are an offset into the 4k page. Similarly, at reference number 578, the 9 bits are used to index into an I/O page table.
20 This I/O page table allows for a DMA address space that has a size of 2M. Many other examples exist.

In one particular implementation, to perform the registration of a DMA address space to the adapter, an instruction referred to as a Modify PCI Function Controls (MPFC) instruction, is
25 used. For example, the operating system determines which address translation format it wishes to use, builds the address translation tables for that format, and then issues the MPFC instruction with that format included as an operand of the instruction. In one example, the format and other operands of the instruction are included in a function information block (described below), which is an operand of the instruction. The function information block is
30 then used to update the DTE and, in one embodiment, optionally, a function table entry (FTE) that includes operational parameters of the adapter.

One embodiment of the details related to this instruction, and particularly the registration process, are described with reference to FIGs. 6A-9. Referring to FIG. 6A, a Modify PCI Function Controls instruction 600 includes, for instance, an op code 602 indicating the Modify PCI Function Controls instruction; a first field 604 specifying a location at which various information is included regarding the adapter function for which the operational parameters are being established; and a second field 606 specifying a location from which a PCI function information block (FIB) is fetched. The contents of the locations designated by Fields 1 and 2 are further described below.

In one embodiment, Field 1 designates a general register that includes various information. As shown in FIG. 6B, the contents of the register include, for instance, a function handle 610 that identifies the handle of the adapter function on behalf of which the modify instruction is being performed; an address space 612 designating an address space in system memory associated with the adapter function designated by the function handle; an operation control 614 which specifies the operation to be performed for the adapter function; and status 616 which provides status regarding the instruction when the instruction completes with a predefined code.

In one embodiment, the function handle includes, for instance, an enable indicator indicating whether the handle is enabled, a function number that identifies an adapter function (this is a static identifier and may be used to index into a function table); and an instance number specifying the particular instance of this function handle. There is one function handle for each adapter function, and it is used to locate a function table entry (FTE) within the function table. Each function table entry includes operational parameters and/or other information associated with its adapter function. As one example, a function table entry includes:

Instance Number: This field indicates a particular instance of the adapter function handle associated with the function table entry;

Device Table Entry (DTE) Index 1...n: There may be one or more device table indices, and each index is an index into a device table to locate a device table entry (DTE). There are one or more device table entries per adapter function, and each entry includes

information associated with its adapter function, including information used to process requests of the adapter function (e.g., DMA requests, MSI requests) and information relating to requests associated with the adapter function (e.g., PCI instructions). Each device table entry is associated with one address space within system memory assigned to the adapter function. An adapter function may have one or more address spaces within system memory assigned to the adapter function.

Busy Indicator: This field indicates whether the adapter function is busy;

Permanent Error State Indicator: This field indicates whether the adapter function is in a permanent error state;

Recovery Initiated Indicator: This field indicates whether recovery has been initiated for the adapter function;

Permission Indicator: This field indicates whether the operating system trying to control the adapter function has authority to do so;

Enable Indicator: This field indicates whether the adapter function is enabled (e.g., 1=enabled, 0=disabled);

Requestor Identifier (RID): This is an identifier of the adapter function, and includes, for instance, a bus number, a device number and a function number.

In one example, this field is used for accesses of a configuration space of the adapter function. (Memory of an adapter may be defined as address spaces, including, for instance, a configuration space, an I/O space, and/or one or more memory spaces.) In one example, the configuration space may be accessed by specifying the configuration space in an instruction issued by the operating system (or other configuration) to the adapter function. Specified in the instruction is an offset into the configuration space and a function handle used to locate the appropriate function table entry that includes the RID. The firmware receives the instruction and determines it is for a configuration space. Therefore, it uses the RID to generate a request to the I/O hub, and the I/O hub creates a request to access the adapter. The location of the adapter function is based on the RID, and the offset specifies an offset into the configuration space of the adapter function.

As used herein, firmware includes, e.g., the microcode, millicode and/or macrocode of the processor. It includes, for instance, the hardware-level instructions and/or data structures used in implementation of higher-level machine code. In one embodiment, it includes, for instance, proprietary code that is typically delivered as microcode that includes trusted
5 software or microcode specific to the underlying hardware and controls operating system access to the system hardware.

Base Address Register (BAR) (1 to n): This field includes a plurality of unsigned integers, designated as $\text{BAR}_0 - \text{BAR}_n$, which are associated with the originally specified adapter
10 function, and whose values are also stored in the base address registers associated with the adapter function. Each BAR specifies the starting address of a memory space or I/O space within the adapter function, and also indicates the type of address space, that is whether it is a 64 or 32 bit memory space, or a 32 bit I/O space, as examples;

In one example, it is used for accesses to memory space and/or I/O space of the adapter
15 function. For instance, an offset provided in an instruction to access the adapter function is added to the value in the base address register associated with the address space designated in the instruction to obtain the address to be used to access the adapter function. The address space identifier provided in the instruction identifies the address space within the adapter
20 function to be accessed and the corresponding BAR to be used;

Size 1...n: This field includes a plurality of unsigned integers, designated as $\text{SIZE}_0 - \text{SIZE}_n$. The value of a Size field, when non-zero, represents the size of each address space with each entry corresponding to a previously described BAR.

Further details regarding BAR and Size are described below.

1. When a BAR is not implemented for an adapter function, the BAR field and its corresponding size field are both stored as zeros.
2. When a BAR field represents either an I/O address space or a 32-bit memory
30 address space, the corresponding size field is non-zero and represents the size of the address space.
3. When a BAR field represents a 64-bit memory address space,

- a. The BAR_n field represents the least significant address bits.
- b. The next consecutive BAR_{n+1} field represents the most significant address bits.
- c. The corresponding SIZE_n field is non-zero and represents the size of the address space.
- d. The corresponding SIZE_{n+1} field is not meaningful and is stored as zero.

Internal Routing Information: This information is used to perform particular routing to the adapter. It includes, for instance, node, processor chip, and hub addressing information, as examples.

Status Indication: This provides an indication of, for instance, whether load/store operations are blocked or the adapter is in the error state, as well as other indications.

In one example, the busy indicator, permanent error state indicator, and recovery initiated indicator are set based on monitoring performed by the firmware. Further, the permission indicator is set, for instance, based on policy; and the BAR information is based on configuration information discovered during a bus walk by the processor (e.g., firmware of the processor). Other fields may be set based on configuration, initialization, and/or events. In other embodiments, the function table entry may include more, less or different information. The information included may depend on the operations supported by or enabled for the adapter function.

Referring to FIG. 6C, in one example, Field 2 designates a logical address 620 of a PCI function information block (FIB), which includes information regarding an associated adapter function. The function information block is used to update a device table entry and/or function table entry (or other location) associated with the adapter function. The information is stored in the FIB during initialization and/or configuration of the adapter, and/or responsive to particular events.

Further details regarding a function information block (FIB) are described with reference to FIG. 6D. In one embodiment, a function information block 650 includes the following fields:

5 Format 651: This field specifies the format of the FIB.

Interception Control 652: This field is used to indicate whether guest execution of specific instructions by a pageable mode guest results in instruction interception;

10 Error Indication 654: This field includes the error state indication for direct memory access and adapter interruptions. When the bit is set (e.g., 1), one or more errors have been detected while performing direct memory access or adapter interruption for the adapter function;

15 Load/Store Blocked 656: This field indicates whether load/store operations are blocked;

PCI Function Valid 658: This field includes an enablement control for the adapter function. When the bit is set (e.g., 1), the adapter function is considered to be enabled for I/O operations;

20 Address Space Registered 660: This field includes a direct memory access enablement control for an adapter function. When the field is set (e.g., 1) direct memory access is enabled;

25 Page Size 661: This field indicates the size of the page or other unit of memory to be accessed by a DMA memory access;

30 PCI Base Address (PBA) 662: This field is a base address for an address space in system memory assigned to the adapter function. It represents the lowest virtual address that an adapter function is allowed to use for direct memory access to the specified DMA address space;

PCI Address Limit (PAL) 664: This field represents the highest virtual address that an adapter function is allowed to access within the specified DMA address space;

5 Input/Output Address Translation Pointer (IOAT) 666: The input/output address translation pointer designates the first of any translation tables used by a PCI virtual address translation, or it may directly designate the absolute address of a frame of storage that is the result of translation;

10 Interruption Subclass (ISC) 668: This field includes the interruption subclass used to present adapter interruptions for the adapter function;

Number of Interruptions (NOI) 670: This field designates the number of distinct interruption codes accepted for an adapter function. This field also defines the size, in bits, of the adapter interruption bit vector designated by an adapter interruption bit vector address and adapter interruption bit vector offset fields;

15 Adapter Interruption Bit Vector Address (AIBV) 672: This field specifies an address of the adapter interruption bit vector for the adapter function. This vector is used in interrupt processing;

Adapter Interruption Bit Vector Offset 674: This field specifies the offset of the first adapter interruption bit vector bit for the adapter function;

20 Adapter Interruption Summary Bit Address (AISB) 676: This field provides an address designating the adapter interruption summary bit, which is optionally used in interrupt processing;

Adapter Interruption Summary Bit Offset 678: This field provides the offset into the adapter interruption summary bit vector;

25 Function Measurement Block (FMB) Address 680: This field provides an address of a function measurement block used to collect measurements regarding the adapter function;

Function Measurement Block Key 682: This field includes an access key to access the function measurement block;

30 Summary Bit Notification Control 684: This field indicates whether there is a summary bit vector being used;

Instruction Authorization Token 686: This field is used to determine whether a pageable storage mode guest is authorized to execute PCI instructions without host intervention.

5 In one example, in the z/Architecture[®], a pageable guest is interpretively executed via the Start Interpretive Execution (SIE) instruction, at level 2 of interpretation. For instance, the logical partition (LPAR) hypervisor executes the SIE instruction to begin the logical partition in physical, fixed memory. If z/VM[®] is the operating system in that logical partition, it issues the SIE instruction to execute its guests (virtual) machines in its V=V (virtual) storage. Therefore, the LPAR hypervisor uses level-1 SIE, and the z/VM[®] hypervisor uses level-2 SIE; and

10

Address Translation Format 687: This field indicates a selected format for address translation of the highest level translation table to be used in translation (e.g., an indication of highest level table (e.g. segment table, region 3rd, etc.)).

15

The function information block designated in the Modify PCI Function Controls instruction is used to modify a selected device table entry, a function table entry and/or other firmware controls associated with the adapter function designated in the instruction. By modifying the device table entry, function table entry and/or other firmware controls, certain services are provided for the adapter. These services include, for instance, adapter interruptions; address translations; reset error state; reset load/store blocked; set function measurement parameters; and set interception control.

20

25 One embodiment of the logic associated with the Modify PCI Function Controls instruction is described with reference to FIG. 7. In one example, the instruction is issued by an operating system (or other configuration) and executed by the processor (e.g., firmware) executing the operating system. In the examples herein, the instruction and adapter functions are PCI based. However, in other examples, a different adapter architecture and corresponding instructions may be used.

30

In one example, the operating system provides the following operands to the instruction (e.g., in one or more registers designated by the instruction): the PCI function handle; the DMA address space identifier; an operation control; and an address of the function information block.

5

Referring to FIG. 7, initially, a determination is made as to whether the facility allowing for a Modify PCI Function Controls instruction is installed, INQUIRY 700. This determination is made by, for instance, checking an indicator stored in, for instance, a control block. If the facility is not installed, an exception condition is provided, STEP 702. Otherwise, a determination is made as to whether the instruction was issued by a pageable storage mode guest (or other guest), INQUIRY 704. If yes, the host operating system will emulate the operation for that guest, STEP 706.

10

15

Otherwise, a determination is made as to whether one or more of the operands are aligned, INQUIRY 708. For instance, a determination is made as to whether the address of the function information block is on a double word boundary. In one example, this is optional. If the operands are not aligned, then an exception condition is provided, STEP 710.

Otherwise, a determination is made as to whether the function information block is accessible, INQUIRY 712. If not, then an exception condition is provided, STEP 714.

20

Otherwise, a determination is made as to whether the handle provided in the operands of the Modify PCI Function Controls instruction is enabled, INQUIRY 716. In one example, this determination is made by checking an enable indicator in the handle. If the handle is not enabled, then an exception condition is provided, STEP 718.

25

If the handle is enabled, then the handle is used to locate a function table entry, STEP 720. That is, at least a portion of the handle is used as an index into the function table to locate the function table entry corresponding to the adapter function for which operational parameters are to be established.

30

A determination is made as to whether the function table entry was found, INQUIRY 722. If not, then an exception condition is provided, STEP 724. Otherwise, if the configuration issuing the instruction is a guest, , INQUIRY 726, then an exception condition (e.g.,

interception to host) is provided, STEP 728. This inquiry may be ignored if the configuration is not a guest or other authorizations may be checked, if designated.

5 A determination is then made as to whether the function is enabled, INQUIRY 730. In one example, this determination is made by checking an enable indicator in the function table entry. If it is not enabled, then an exception condition is provided, STEP 732.

10 If the function is enabled, then a determination is made as to whether recovery is active, INQUIRY 734. If recovery is active as determined by a recovery indicator in the function table entry, then an exception condition is provided, STEP 736. However, if recovery is not active, then a further determination is made as to whether the function is busy, INQUIRY 738. This determination is made by checking the busy indicator in the function table entry. If the function is busy, then a busy condition is provided, STEP 740. With the busy condition, the instruction can be retried, instead of dropped.

15 If the function is not busy, then a further determination is made as to whether the function information block format is valid, INQUIRY 742. For instance, the format field of the FIB is checked to determine if this format is supported by the system. If it is invalid, then an exception condition is provided, STEP 744. If the function information block format is
20 valid, then a further determination is made as to whether the operation control specified in the operands of the instruction is valid, INQUIRY 746. That is, is the operation control one of the specified operation controls for this instruction. If it is invalid, then an exception condition is provided, STEP 748. However, if the operation control is valid, then processing continues with the specific operation control being specified.

25 One operation control that may be specified is a register I/O address translation parameters operation used in controlling address translations for an adapter. With this operation, the PCI function parameters relevant to I/O address translation are set in the DTE, FTE and/or other location from the appropriate parameters of the FIB, which is an operand to the
30 instruction. These parameters include, for instance, the PCI base address; the PCI address limit (a.k.a., PCI limit or limit); the address translation format; the page size; and the I/O address translation pointer, which are operands to this operation. There are also implied

operands, including a starting DMA address (SDMA) and an ending DMA address (EDMA), which are stored in a location accessible to the processor executing the instruction.

One embodiment of the logic to establish the operational parameters for I/O address translation is described with reference to FIG. 8. Initially, a determination is made as to whether the PCI base address in the FIB is greater than the PCI limit in the FIB, INQUIRY 800. If the comparison of the base address and the limit indicate that the base address is greater than the limit, then an exception condition is recognized, STEP 802. However, if the base address is less than or equal to the limit, then a further determination is made as to whether the address translation format and the page size are valid, INQUIRY 804. If they are invalid, then an exception condition is provided, STEP 806. However, if they are valid, then a further determination is made as to whether the size of the address space (based on the base address and limit) exceeds the translation capacity, INQUIRY 808. In one example, the size of the address space is compared to the maximum address translation capacity possible based on the format of the upper level table. For example, if the upper level table is a DAT compatible segment table, the maximum translation capacity is 2 Gbytes.

If the size of the address space exceeds the translation capacity, then an exception condition is provided, STEP 810. Otherwise, a further determination is made as to whether the base address is less than the starting DMA address, INQUIRY 812. If so, then an exception condition is provided, STEP 814. Otherwise, another determination is made as to whether the address limit is greater than the ending DMA address, INQUIRY 816. If so, then an exception condition is provided, STEP 818. In one example, the starting DMA address and ending DMA address are based on a system-wide policy.

Thereafter, a determination is made as to whether sufficient resources, if any are needed, are available to perform an I/O address translation, INQUIRY 820. If not, then an exception condition is provided, STEP 822. Otherwise, a further determination is made as to whether the I/O address translation parameters have already been registered in the FTE and DTE, INQUIRY 824. This is determined by checking the values of the parameters in the FTE/DTE. For instance, if the values in the FTE/DTE are zero or another defined value,

then registration has not been performed. To locate the FTE, the handle provided in the instruction is used, and to locate the DTE, a device index in the FTE is used.

If the adapter function has already been registered for address translation, then an exception
5 condition is provided, STEP 826. If not, then a determination is made as to whether the
DMA address space that is specified is valid (i.e., is it an address space for which a DTE has
been enabled), INQUIRY 828. If not, then an exception condition is provided, STEP 830. If
all the checks are successful, then the translation parameters are placed in the device table
entry and optionally, in the corresponding function table entry (or other designated location),
10 STEP 832. For instance, the PCI function parameters relevant to I/O address translation are
copied from the function information block and placed in the DTE/FTE. These parameters
include, for instance, the PCI base address, the PCI address limit, the translation format, the
page size, and the I/O address translation pointer. This operation enables DMA accesses to
the specified DMA address space. It enables I/O address translation for the adapter function.

15 Another operation control that may be specified by the Modify PCI Function Controls
instruction is an unregister I/O address translation parameters operation, an example of
which is described with reference to FIG. 9. With this operation, the function parameters
relevant to I/O address translation are reset to zeros. This operation disables DMA accesses
20 to the specified DMA address space and causes a purge of I/O translation lookaside buffer
entries for that DMA address space. It disables address translation.

Referring to FIG. 9, in one embodiment, a determination is made as to whether the I/O
address translation parameters are not registered, INQUIRY 900. In one example, this
25 determination is made by checking the values of the appropriate parameters in the FTE or
DTE. If those fields are zero or some specified value, they are not registered. Therefore, an
exception condition is provided, STEP 902. If they are registered, then a determination is
made as to whether the DMA address space is valid, INQUIRY 904. If it is invalid, then an
exception condition is provided, STEP 906. If the DMA address space is valid, then the
30 translation parameters in the device table entry and optionally, in the corresponding function
table entry are cleared, STEP 908.

Described in detail above is an efficient mechanism for translating an input/output address provided by an adapter to a system memory address. In one example, the full PCI address (e.g., the entire 64-bit address) is used in accessing memory; however, address translation look-ups are minimized by only using a portion of the address for translation. While use of the full address provides added protection, the use of, for instance, only the lower order bits for translation allows more efficient translation by means of less levels of translation look-up. This allows for operating system flexibility in using addresses that can co-exist with or be the same as operating system virtual addresses. Further, it allows CPU address translation tables to be shared by adapters, while allowing the I/O to reduce the number of tables that need to be scanned. Further, additional protection is provided in allowing different adapters and/or operating systems to use disjoint address space ranges.

In the embodiments described herein, the adapters are PCI adapters. PCI, as used herein, refers to any adapters implemented according to a PCI-based specification as defined by the Peripheral Component Interconnect Special Interest Group (PCI-SIG), including but not limited to, PCI or PCIe. In one particular example, the Peripheral Component Interconnect Express (PCIe) is a component level interconnect standard that defines a bi-directional communication protocol for transactions between I/O adapters and host systems. PCIe communications are encapsulated in packets according to the PCIe standard for transmission on a PCIe bus. Transactions originating at I/O adapters and ending at host systems are referred to as upbound transactions. Transactions originating at host systems and terminating at I/O adapters are referred to as downbound transactions. The PCIe topology is based on point-to-point unidirectional links that are paired (e.g., one upbound link, one downbound link) to form the PCIe bus. The PCIe standard is maintained and published by the PCI-SIG.

As will be appreciated by one skilled in the art, aspects of the present invention may be embodied as a system, method or computer program product. Accordingly, aspects of the present invention may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module" or "system". Furthermore, aspects of the present invention

may take the form of a computer program product embodied in one or more computer readable medium(s) having computer readable program code embodied thereon.

Any combination of one or more computer readable medium(s) may be utilized. The computer readable medium may be a computer readable storage medium. A computer readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, electromagnetic, infrared or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the computer readable storage medium include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), an optical fiber, a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain or store a program for use by or in connection with an instruction execution system, apparatus, or device.

Referring now to FIG. 10, in one example, a computer program product 1000 includes, for instance, one or more computer readable storage media 1002 to store computer readable program code means or logic 1004 thereon to provide and facilitate one or more aspects of the present invention.

Program code embodied on a computer readable medium may be transmitted using an appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

Computer program code for carrying out operations for aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language, such as Java, Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language, assembler or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the

user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

These computer program instructions may also be stored in a computer readable medium that can direct a computer, other programmable data processing apparatus, or other devices to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

The flowchart and block diagrams in the figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods and computer program products

according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function(s). It should also be noted that, in some alternative implementations, the functions
5 noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be
10 implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

In addition to the above, one or more aspects of the present invention may be provided, offered, deployed, managed, serviced, etc. by a service provider who offers management of
15 customer environments. For instance, the service provider can create, maintain, support, etc. computer code and/or a computer infrastructure that performs one or more aspects of the present invention for one or more customers. In return, the service provider may receive payment from the customer under a subscription and/or fee agreement, as examples. Additionally or alternatively, the service provider may receive payment from the sale of
20 advertising content to one or more third parties.

In one aspect of the present invention, an application may be deployed for performing one or more aspects of the present invention. As one example, the deploying of an application comprises providing computer infrastructure operable to perform one or more aspects of the
25 present invention.

As a further aspect of the present invention, a computing infrastructure may be deployed comprising integrating computer readable code into a computing system, in which the code in combination with the computing system is capable of performing one or more aspects of
30 the present invention.

As yet a further aspect of the present invention, a process for integrating computing infrastructure comprising integrating computer readable code into a computer system may be provided. The computer system comprises a computer readable medium, in which the computer medium comprises one or more aspects of the present invention. The code in combination with the computer system is capable of performing one or more aspects of the present invention.

Although various embodiments are described above, these are only examples. For example, computing environments of other architectures can incorporate and use one or more aspects of the present invention. As examples, servers other than System z[®] servers, such as Power Systems servers or other servers offered by International Business Machines Corporation, or servers of other companies can include, use and/or benefit from one or more aspects of the present invention. Further, although in the example herein, the adapters and PCI hub are considered a part of the server, in other embodiments, they do not have to necessarily be considered a part of the server, but can simply be considered as being coupled to system memory and/or other components of a computing environment. The computing environment need not be a server. Further, although translation tables are described, any data structure can be used and the term table is to include all such data structures. Yet further, although the adapters are PCI based, one or more aspects of the present invention are usable with other adapters or other I/O components. Adapter and PCI adapter are just examples. Moreover, other size address spaces, address tables and/or pages may be used without departing from the present invention. Further, the DTE may include more, less or different information. Yet further, other types of addresses may be translated using one or more aspects of the present invention. Many other variations are possible.

Further, other types of computing environments can benefit from one or more aspects of the present invention. As an example, a data processing system suitable for storing and/or executing program code is usable that includes at least two processors coupled directly or indirectly to memory elements through a system bus. The memory elements include, for instance, local memory employed during actual execution of the program code, bulk storage, and cache memory which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

Input/Output or I/O devices (including, but not limited to, keyboards, displays, pointing devices, DASD, tape, CDs, DVDs, thumb drives and other memory media, etc.) can be coupled to the system either directly or through intervening I/O controllers. Network adapters may also be coupled to the system to enable the data processing system to become
5 coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the available types of network adapters.

Referring to FIG. 11, representative components of a Host Computer system 5000 to
10 implement one or more aspects of the present invention are portrayed. The representative host computer 5000 comprises one or more CPUs 5001 in communication with computer memory (i.e., central storage) 5002, as well as I/O interfaces to storage media devices 5011 and networks 5010 for communicating with other computers or SANs and the like. The CPU 5001 is compliant with an architecture having an architected instruction set and
15 architected functionality. The CPU 5001 may have dynamic address translation (DAT) 5003 for transforming program addresses (virtual addresses) into real addresses of memory. A DAT typically includes a translation lookaside buffer (TLB) 5007 for caching translations so that later accesses to the block of computer memory 5002 do not require the delay of address translation. Typically, a cache 5009 is employed between computer memory 5002 and the
20 processor 5001. The cache 5009 may be hierarchical having a large cache available to more than one CPU and smaller, faster (lower level) caches between the large cache and each CPU. In some implementations, the lower level caches are split to provide separate low level caches for instruction fetching and data accesses. In one embodiment, an instruction is fetched from memory 5002 by an instruction fetch unit 5004 via a cache 5009. The
25 instruction is decoded in an instruction decode unit 5006 and dispatched (with other instructions in some embodiments) to instruction execution unit or units 5008. Typically several execution units 5008 are employed, for example an arithmetic execution unit, a floating point execution unit and a branch instruction execution unit. The instruction is executed by the execution unit, accessing operands from instruction specified registers or
30 memory as needed. If an operand is to be accessed (loaded or stored) from memory 5002, a load/store unit 5005 typically handles the access under control of the instruction being

executed. Instructions may be executed in hardware circuits or in internal microcode (firmware) or by a combination of both.

As noted, a computer system includes information in local (or main) storage, as well as
5 addressing, protection, and reference and change recording. Some aspects of addressing include the format of addresses, the concept of address spaces, the various types of addresses, and the manner in which one type of address is translated to another type of address. Some of main storage includes permanently assigned storage locations. Main storage provides the system with directly addressable fast-access storage of data. Both data
10 and programs are to be loaded into main storage (from input devices) before they can be processed.

Main storage may include one or more smaller, faster-access buffer storages, sometimes called caches. A cache is typically physically associated with a CPU or an I/O processor.
15 The effects, except on performance, of the physical construction and use of distinct storage media are generally not observable by the program.

Separate caches may be maintained for instructions and for data operands. Information within a cache is maintained in contiguous bytes on an integral boundary called a cache
20 block or cache line (or line, for short). A model may provide an EXTRACT CACHE ATTRIBUTE instruction which returns the size of a cache line in bytes. A model may also provide PREFETCH DATA and PREFETCH DATA RELATIVE LONG instructions which effects the prefetching of storage into the data or instruction cache or the releasing of data from the cache.

Storage is viewed as a long horizontal string of bits. For most operations, accesses to storage proceed in a left-to-right sequence. The string of bits is subdivided into units of eight bits. An eight-bit unit is called a byte, which is the basic building block of all information
25 formats. Each byte location in storage is identified by a unique nonnegative integer, which is the address of that byte location or, simply, the byte address. Adjacent byte locations have consecutive addresses, starting with 0 on the left and proceeding in a left-to-right sequence. Addresses are unsigned binary integers and are 24, 31, or 64 bits.
30

Information is transmitted between storage and a CPU or a channel subsystem one byte, or a group of bytes, at a time. Unless otherwise specified, in, for instance, the z/Architecture[®], a group of bytes in storage is addressed by the leftmost byte of the group. The number of

5 When used in a CPU operation, a group of bytes is called a field. Within each group of bytes, in, for instance, the z/Architecture[®], bits are numbered in a left-to-right sequence. In the z/Architecture[®], the leftmost bits are sometimes referred to as the “high-order” bits and the rightmost bits as the “low-order” bits. Bit numbers are not storage addresses, however.

10 Only bytes can be addressed. To operate on individual bits of a byte in storage, the entire byte is accessed. The bits in a byte are numbered 0 through 7, from left to right (in, e.g., the z/Architecture[®]). The bits in an address may be numbered 8-31 or 40-63 for 24-bit addresses, or 1-31 or 33-63 for 31-bit addresses; they are numbered 0-63 for 64-bit

addresses. Within any other fixed-length format of multiple bytes, the bits making up the format are consecutively numbered starting from 0. For purposes of error detection, and in preferably for correction, one or more check bits may be transmitted with each byte or with a

15 group of bytes. Such check bits are generated automatically by the machine and cannot be directly controlled by the program. Storage capacities are expressed in number of bytes.

When the length of a storage-operand field is implied by the operation code of an instruction, the field is said to have a fixed length, which can be one, two, four, eight, or

20 sixteen bytes. Larger fields may be implied for some instructions. When the length of a storage-operand field is not implied but is stated explicitly, the field is said to have a variable length. Variable-length operands can vary in length by increments of one byte (or with some instructions, in multiples of two bytes or other multiples). When information is placed in storage, the contents of only those byte locations are replaced that are included in the

25 designated field, even though the width of the physical path to storage may be greater than the length of the field being stored.

Certain units of information are to be on an integral boundary in storage. A boundary is called integral for a unit of information when its storage address is a multiple of the length of

30 the unit in bytes. Special names are given to fields of 2, 4, 8, and 16 bytes on an integral boundary. A halfword is a group of two consecutive bytes on a two-byte boundary and is the basic building block of instructions. A word is a group of four consecutive bytes on a

four-byte boundary. A doubleword is a group of eight consecutive bytes on an eight-byte boundary. A quadword is a group of 16 consecutive bytes on a 16-byte boundary. When storage addresses designate halfwords, words, doublewords, and quadwords, the binary representation of the address contains one, two, three, or four rightmost zero bits, respectively. Instructions are to be on two-byte integral boundaries. The storage operands of most instructions do not have boundary-alignment requirements.

On devices that implement separate caches for instructions and data operands, a significant delay may be experienced if the program stores into a cache line from which instructions are subsequently fetched, regardless of whether the store alters the instructions that are subsequently fetched.

In one embodiment, the invention may be practiced by software (sometimes referred to licensed internal code, firmware, micro-code, milli-code, pico-code and the like, any of which would be consistent with the present invention). Referring to FIG. 11, software program code which embodies the present invention is typically accessed by processor 5001 of the host system 5000 from long-term storage media devices 5011, such as a CD-ROM drive, tape drive or hard drive. The software program code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users from computer memory 5002 or storage of one computer system over a network 5010 to other computer systems for use by users of such other systems.

The software program code includes an operating system which controls the function and interaction of the various computer components and one or more application programs. Program code is normally paged from storage media device 5011 to the relatively higher-speed computer storage 5002 where it is available for processing by processor 5001. The techniques and methods for embodying software program code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a "computer program product".

The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

FIG. 12 illustrates a representative workstation or server hardware system in which the present invention may be practiced. The system 5020 of FIG. 12 comprises a representative base computer system 5021, such as a personal computer, a workstation or a server, including optional peripheral devices. The base computer system 5021 includes one or more processors 5026 and a bus employed to connect and enable communication between the processor(s) 5026 and the other components of the system 5021 in accordance with known techniques. The bus connects the processor 5026 to memory 5025 and long-term storage 5027 which can include a hard drive (including any of magnetic media, CD, DVD and Flash Memory for example) or a tape drive for example. The system 5021 might also include a user interface adapter, which connects the microprocessor 5026 via the bus to one or more interface devices, such as a keyboard 5024, a mouse 5023, a printer/scanner 5030 and/or other interface devices, which can be any user interface device, such as a touch sensitive screen, digitized entry pad, etc. The bus also connects a display device 5022, such as an LCD screen or monitor, to the microprocessor 5026 via a display adapter.

The system 5021 may communicate with other computers or networks of computers by way of a network adapter capable of communicating 5028 with a network 5029. Example network adapters are communications channels, token ring, Ethernet or modems. Alternatively, the system 5021 may communicate using a wireless interface, such as a CDPD (cellular digital packet data) card. The system 5021 may be associated with such other computers in a Local Area Network (LAN) or a Wide Area Network (WAN), or the system 5021 can be a client in a client/server arrangement with another computer, etc. All of these configurations, as well as the appropriate communications hardware and software, are known in the art.

FIG. 13 illustrates a data processing network 5040 in which the present invention may be practiced. The data processing network 5040 may include a plurality of individual networks, such as a wireless network and a wired network, each of which may include a plurality of individual workstations 5041, 5042, 5043, 5044. Additionally, as those skilled in the art will

appreciate, one or more LANs may be included, where a LAN may comprise a plurality of intelligent workstations coupled to a host processor.

Still referring to FIG. 13, the networks may also include mainframe computers or servers, such as a gateway computer (client server 5046) or application server (remote server 5048 which may access a data repository and may also be accessed directly from a workstation 5045). A gateway computer 5046 serves as a point of entry into each individual network. A gateway is needed when connecting one networking protocol to another. The gateway 5046 may be preferably coupled to another network (the Internet 5047 for example) by means of a communications link. The gateway 5046 may also be directly coupled to one or more workstations 5041, 5042, 5043, 5044 using a communications link. The gateway computer may be implemented utilizing an IBM eServer™ System z® server available from International Business Machines Corporation.

Referring concurrently to FIG. 12 and FIG. 13, software programming code which may embody the present invention may be accessed by the processor 5026 of the system 5020 from long-term storage media 5027, such as a CD-ROM drive or hard drive. The software programming code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users 5050, 5051 from the memory or storage of one computer system over a network to other computer systems for use by users of such other systems.

Alternatively, the programming code may be embodied in the memory 5025, and accessed by the processor 5026 using the processor bus. Such programming code includes an operating system which controls the function and interaction of the various computer components and one or more application programs 5032. Program code is normally paged from storage media 5027 to high-speed memory 5025 where it is available for processing by the processor 5026. The techniques and methods for embodying software programming code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash

memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a “computer program product”. The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

5

The cache that is most readily available to the processor (normally faster and smaller than other caches of the processor) is the lowest (L1 or level one) cache and main store (main memory) is the highest level cache (L3 if there are 3 levels). The lowest level cache is often divided into an instruction cache (I-Cache) holding machine instructions to be executed and a data cache (D-Cache) holding data operands.

10

Referring to FIG. 14, an exemplary processor embodiment is depicted for processor 5026. Typically one or more levels of cache 5053 are employed to buffer memory blocks in order to improve processor performance. The cache 5053 is a high speed buffer holding cache lines of memory data that are likely to be used. Typical cache lines are 64, 128 or 256 bytes of memory data. Separate caches are often employed for caching instructions than for caching data. Cache coherence (synchronization of copies of lines in memory and the caches) is often provided by various “snoop” algorithms well known in the art. Main memory storage 5025 of a processor system is often referred to as a cache. In a processor system having 4 levels of cache 5053, main storage 5025 is sometimes referred to as the level 5 (L5) cache since it is typically faster and only holds a portion of the non-volatile storage (DASD, tape etc) that is available to a computer system. Main storage 5025 “caches” pages of data paged in and out of the main storage 5025 by the operating system.

15

20

25

30

A program counter (instruction counter) 5061 keeps track of the address of the current instruction to be executed. A program counter in a z/Architecture[®] processor is 64 bits and can be truncated to 31 or 24 bits to support prior addressing limits. A program counter is typically embodied in a PSW (program status word) of a computer such that it persists during context switching. Thus, a program in progress, having a program counter value, may be interrupted by, for example, the operating system (context switch from the program environment to the operating system environment). The PSW of the program maintains the program counter value while the program is not active, and the program counter (in the

PSW) of the operating system is used while the operating system is executing. Typically, the program counter is incremented by an amount equal to the number of bytes of the current instruction. RISC (Reduced Instruction Set Computing) instructions are typically fixed length while CISC (Complex Instruction Set Computing) instructions are typically variable length. Instructions of the IBM z/Architecture[®] are CISC instructions having a length of 2, 4 or 6 bytes. The Program counter 5061 is modified by either a context switch operation or a branch taken operation of a branch instruction for example. In a context switch operation, the current program counter value is saved in the program status word along with other state information about the program being executed (such as condition codes), and a new program counter value is loaded pointing to an instruction of a new program module to be executed. A branch taken operation is performed in order to permit the program to make decisions or loop within the program by loading the result of the branch instruction into the program counter 5061.

Typically an instruction fetch unit 5055 is employed to fetch instructions on behalf of the processor 5026. The fetch unit either fetches “next sequential instructions”, target instructions of branch taken instructions, or first instructions of a program following a context switch. Modern Instruction fetch units often employ prefetch techniques to speculatively prefetch instructions based on the likelihood that the prefetched instructions might be used. For example, a fetch unit may fetch 16 bytes of instruction that includes the next sequential instruction and additional bytes of further sequential instructions.

The fetched instructions are then executed by the processor 5026. In an embodiment, the fetched instruction(s) are passed to a dispatch unit 5056 of the fetch unit. The dispatch unit decodes the instruction(s) and forwards information about the decoded instruction(s) to appropriate units 5057, 5058, 5060. An execution unit 5057 will typically receive information about decoded arithmetic instructions from the instruction fetch unit 5055 and will perform arithmetic operations on operands according to the opcode of the instruction. Operands are provided to the execution unit 5057 preferably either from memory 5025, architected registers 5059 or from an immediate field of the instruction being executed. Results of the execution, when stored, are stored either in memory 5025, registers 5059 or in other machine hardware (such as control registers, PSW registers and the like).

A processor 5026 typically has one or more units 5057, 5058, 5060 for executing the function of the instruction. Referring to FIG. 15A, an execution unit 5057 may communicate with architected general registers 5059, a decode/dispatch unit 5056, a load store unit 5060, and other 5065 processor units by way of interfacing logic 5071. An
5 execution unit 5057 may employ several register circuits 5067, 5068, 5069 to hold information that the arithmetic logic unit (ALU) 5066 will operate on. The ALU performs arithmetic operations such as add, subtract, multiply and divide as well as logical function such as and, or and exclusive-or (XOR), rotate and shift. Preferably the ALU supports specialized operations that are design dependent. Other circuits may provide other
10 architected facilities 5072 including condition codes and recovery support logic for example. Typically the result of an ALU operation is held in an output register circuit 5070 which can forward the result to a variety of other processing functions. There are many arrangements of processor units, the present description is only intended to provide a representative understanding of one embodiment.

15 An ADD instruction for example would be executed in an execution unit 5057 having arithmetic and logical functionality while a floating point instruction for example would be executed in a floating point execution having specialized floating point capability. Preferably, an execution unit operates on operands identified by an instruction by performing
20 an opcode defined function on the operands. For example, an ADD instruction may be executed by an execution unit 5057 on operands found in two registers 5059 identified by register fields of the instruction.

25 The execution unit 5057 performs the arithmetic addition on two operands and stores the result in a third operand where the third operand may be a third register or one of the two source registers. The execution unit preferably utilizes an Arithmetic Logic Unit (ALU) 5066 that is capable of performing a variety of logical functions such as Shift, Rotate, And, Or and XOR as well as a variety of algebraic functions including any of add, subtract, multiply, divide. Some ALUs 5066 are designed for scalar operations and some for floating
30 point. Data may be Big Endian (where the least significant byte is at the highest byte address) or Little Endian (where the least significant byte is at the lowest byte address) depending on architecture. The IBM z/Architecture[®] is Big Endian. Signed fields may be

sign and magnitude, 1's complement or 2's complement depending on architecture. A 2's complement number is advantageous in that the ALU does not need to design a subtract capability since either a negative value or a positive value in 2's complement requires only an addition within the ALU. Numbers are commonly described in shorthand, where a 12 bit
5 field defines an address of a 4,096 byte block and is commonly described as a 4 Kbyte (Kilo-byte) block, for example.

Referring to FIG. 15B, branch instruction information for executing a branch instruction is typically sent to a branch unit 5058 which often employs a branch prediction algorithm such
10 as a branch history table 5082 to predict the outcome of the branch before other conditional operations are complete. The target of the current branch instruction will be fetched and speculatively executed before the conditional operations are complete. When the conditional operations are completed the speculatively executed branch instructions are either completed or discarded based on the conditions of the conditional operation and the speculated
15 outcome. A typical branch instruction may test condition codes and branch to a target address if the condition codes meet the branch requirement of the branch instruction, a target address may be calculated based on several numbers including ones found in register fields or an immediate field of the instruction for example. The branch unit 5058 may employ an ALU 5074 having a plurality of input register circuits 5075, 5076, 5077 and an output
20 register circuit 5080. The branch unit 5058 may communicate with general registers 5059, decode dispatch unit 5056 or other circuits 5073, for example.

The execution of a group of instructions can be interrupted for a variety of reasons including a context switch initiated by an operating system, a program exception or error causing a
25 context switch, an I/O interruption signal causing a context switch or multi-threading activity of a plurality of programs (in a multi-threaded environment), for example. Preferably a context switch action saves state information about a currently executing program and then loads state information about another program being invoked. State information may be saved in hardware registers or in memory for example. State information preferably
30 comprises a program counter value pointing to a next instruction to be executed, condition codes, memory translation information and architected register content. A context switch activity can be exercised by hardware circuits, application programs, operating system

programs or firmware code (microcode, pico-code or licensed internal code (LIC)) alone or in combination.

A processor accesses operands according to instruction defined methods. The instruction may provide an immediate operand using the value of a portion of the instruction, may provide one or more register fields explicitly pointing to either general purpose registers or special purpose registers (floating point registers for example). The instruction may utilize implied registers identified by an opcode field as operands. The instruction may utilize memory locations for operands. A memory location of an operand may be provided by a register, an immediate field, or a combination of registers and immediate field as exemplified by the z/Architecture[®] long displacement facility wherein the instruction defines a base register, an index register and an immediate field (displacement field) that are added together to provide the address of the operand in memory for example. Location herein typically implies a location in main memory (main storage) unless otherwise indicated.

Referring to FIG. 15C, a processor accesses storage using a load/store unit 5060. The load/store unit 5060 may perform a load operation by obtaining the address of the target operand in memory 5053 and loading the operand in a register 5059 or another memory 5053 location, or may perform a store operation by obtaining the address of the target operand in memory 5053 and storing data obtained from a register 5059 or another memory 5053 location in the target operand location in memory 5053. The load/store unit 5060 may be speculative and may access memory in a sequence that is out-of-order relative to instruction sequence, however the load/store unit 5060 is to maintain the appearance to programs that instructions were executed in order. A load/store unit 5060 may communicate with general registers 5059, decode/dispatch unit 5056, cache/memory interface 5053 or other elements 5083 and comprises various register circuits, ALUs 5085 and control logic 5090 to calculate storage addresses and to provide pipeline sequencing to keep operations in-order. Some operations may be out of order but the load/store unit provides functionality to make the out of order operations to appear to the program as having been performed in order, as is well known in the art.

Preferably addresses that an application program “sees” are often referred to as virtual addresses. Virtual addresses are sometimes referred to as “logical addresses” and “effective addresses”. These virtual addresses are virtual in that they are redirected to physical memory location by one of a variety of dynamic address translation (DAT) technologies including, but not limited to, simply prefixing a virtual address with an offset value, translating the virtual address via one or more translation tables, the translation tables preferably comprising at least a segment table and a page table alone or in combination, preferably, the segment table having an entry pointing to the page table. In the z/Architecture[®], a hierarchy of translation is provided including a region first table, a region second table, a region third table, a segment table and an optional page table. The performance of the address translation is often improved by utilizing a translation lookaside buffer (TLB) which comprises entries mapping a virtual address to an associated physical memory location. The entries are created when the DAT translates a virtual address using the translation tables. Subsequent use of the virtual address can then utilize the entry of the fast TLB rather than the slow sequential translation table accesses. TLB content may be managed by a variety of replacement algorithms including LRU (Least Recently used).

In the case where the processor is a processor of a multi-processor system, each processor has responsibility to keep shared resources, such as I/O, caches, TLBs and memory, interlocked for coherency. Typically, “snoop” technologies will be utilized in maintaining cache coherency. In a snoop environment, each cache line may be marked as being in any one of a shared state, an exclusive state, a changed state, an invalid state and the like in order to facilitate sharing.

I/O units 5054 (FIG. 14) provide the processor with means for attaching to peripheral devices including tape, disc, printers, displays, and networks for example. I/O units are often presented to the computer program by software drivers. In mainframes, such as the System z[®] from IBM[®], channel adapters and open system adapters are I/O units of the mainframe that provide the communications between the operating system and peripheral devices.

Further, other types of computing environments can benefit from one or more aspects of the present invention. As an example, an environment may include an emulator (e.g., software

or other emulation mechanisms), in which a particular architecture (including, for instance, instruction execution, architected functions, such as address translation, and architected registers) or a subset thereof is emulated (e.g., on a native computer system having a processor and memory). In such an environment, one or more emulation functions of the emulator can implement one or more aspects of the present invention, even though a computer executing the emulator may have a different architecture than the capabilities being emulated. As one example, in emulation mode, the specific instruction or operation being emulated is decoded, and an appropriate emulation function is built to implement the individual instruction or operation.

In an emulation environment, a host computer includes, for instance, a memory to store instructions and data; an instruction fetch unit to fetch instructions from memory and to optionally, provide local buffering for the fetched instruction; an instruction decode unit to receive the fetched instructions and to determine the type of instructions that have been fetched; and an instruction execution unit to execute the instructions. Execution may include loading data into a register from memory; storing data back to memory from a register; or performing some type of arithmetic or logical operation, as determined by the decode unit. In one example, each unit is implemented in software. For instance, the operations being performed by the units are implemented as one or more subroutines within emulator software.

More particularly, in a mainframe, architected machine instructions are used by programmers, usually today “C” programmers, often by way of a compiler application. These instructions stored in the storage medium may be executed natively in a z/Architecture[®] IBM[®] Server, or alternatively in machines executing other architectures. They can be emulated in the existing and in future IBM[®] mainframe servers and on other machines of IBM[®] (e.g., Power Systems servers and System x[®] Servers). They can be executed in machines running Linux on a wide variety of machines using hardware manufactured by IBM[®], Intel[®], AMD[™], and others. Besides execution on that hardware under a z/Architecture[®], Linux can be used as well as machines which use emulation by TurboHercules (www.turbohercules.com/), Hercules (www.hercules-390.org/) or FSI (Fundamental Software, Inc) (www.funsoft.com/), where generally execution is in an

emulation mode. In emulation mode, emulation software is executed by a native processor to emulate the architecture of an emulated processor.

5 The native processor typically executes emulation software comprising either firmware or a native operating system to perform emulation of the emulated processor. The emulation software is responsible for fetching and executing instructions of the emulated processor architecture. The emulation software maintains an emulated program counter to keep track of instruction boundaries. The emulation software may fetch one or more emulated machine instructions at a time and convert the one or more emulated machine instructions to a
10 corresponding group of native machine instructions for execution by the native processor. These converted instructions may be cached such that a faster conversion can be accomplished. Notwithstanding, the emulation software is to maintain the architecture rules of the emulated processor architecture so as to assure operating systems and applications written for the emulated processor operate correctly. Furthermore, the emulation software is
15 to provide resources identified by the emulated processor architecture including, but not limited to, control registers, general purpose registers, floating point registers, dynamic address translation function including segment tables and page tables for example, interrupt mechanisms, context switch mechanisms, Time of Day (TOD) clocks and architected interfaces to I/O subsystems such that an operating system or an application program
20 designed to run on the emulated processor, can be run on the native processor having the emulation software.

A specific instruction being emulated is decoded, and a subroutine is called to perform the function of the individual instruction. An emulation software function emulating a function
25 of an emulated processor is implemented, for example, in a "C" subroutine or driver, or some other method of providing a driver for the specific hardware as will be within the skill of those in the art after understanding the description of the preferred embodiment. Various software and hardware emulation patents including, but not limited to U.S. Letters Patent No. 5,551,013, entitled "Multiprocessor for Hardware Emulation", by Beausoleil et al.; and
30 U.S. Letters Patent No. 6,009,261, entitled "Preprocessing of Stored Target Routines for Emulating Incompatible Instructions on a Target Processor", by Scalzi et al; and U.S. Letters Patent No. 5,574,873, entitled "Decoding Guest Instruction to Directly Access Emulation

Routines that Emulate the Guest Instructions”, by Davidian et al; and U.S. Letters Patent No. 6,308,255, entitled “Symmetrical Multiprocessing Bus and Chipset Used for Coprocessor Support Allowing Non-Native Code to Run in a System”, by Gorishek et al; and U.S. Letters Patent No. 6,463,582, entitled “Dynamic Optimizing Object Code Translator for
5 Architecture Emulation and Dynamic Optimizing Object Code Translation Method”, by Lethin et al; and U.S. Letters Patent No. 5,790,825, entitled “Method for Emulating Guest Instructions on a Host Computer Through Dynamic Recompilation of Host Instructions”, by Eric Traut; and many others, illustrate a variety of known ways to achieve emulation of an instruction format architected for a different machine for a target machine available to those
10 skilled in the art.

In FIG. 16, an example of an emulated host computer system 5092 is provided that emulates a host computer system 5000' of a host architecture. In the emulated host computer system 5092, the host processor (CPU) 5091 is an emulated host processor (or virtual host
15 processor) and comprises an emulation processor 5093 having a different native instruction set architecture than that of the processor 5091 of the host computer 5000'. The emulated host computer system 5092 has memory 5094 accessible to the emulation processor 5093. In the example embodiment, the memory 5094 is partitioned into a host computer memory 5096 portion and an emulation routines 5097 portion. The host computer memory 5096 is
20 available to programs of the emulated host computer 5092 according to host computer architecture. The emulation processor 5093 executes native instructions of an architected instruction set of an architecture other than that of the emulated processor 5091, the native instructions obtained from emulation routines memory 5097, and may access a host instruction for execution from a program in host computer memory 5096 by employing one
25 or more instruction(s) obtained in a sequence & access/decode routine which may decode the host instruction(s) accessed to determine a native instruction execution routine for emulating the function of the host instruction accessed. Other facilities that are defined for the host computer system 5000' architecture may be emulated by architected facilities routines, including such facilities as general purpose registers, control registers, dynamic address
30 translation and I/O subsystem support and processor cache, for example. The emulation routines may also take advantage of functions available in the emulation processor 5093 (such as general registers and dynamic translation of virtual addresses) to improve

performance of the emulation routines. Special hardware and off-load engines may also be provided to assist the processor 5093 in emulating the function of the host computer 5000'.

5 The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the invention. As used herein, the singular forms “a”, “an” and “the” are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms “comprises” and/or “comprising”, when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or
10 addition of one or more other features, integers, steps, operations, elements, components and/or groups thereof.

The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below, if any, are intended to include any structure, material,
15 or act for performing the function in combination with other claimed elements as specifically claimed. The description of the present invention has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope of the invention. The embodiment was
20 chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

CLAIMS

1. A method of translating addresses in a computing environment, said method comprising:

5 obtaining an address from an adapter to be translated, the address comprising a plurality of bits, the plurality of bits comprising a first portion of bits and a second portion of bits;

receiving an address range value that indicates a range of permitted addresses;

10 validating the address using at least the first portion of bits and the received address range; and

converting the address to a memory address usable in accessing memory of the computing environment, the converting ignoring the first portion of bits and using the second portion of bits to obtain address information from one or more levels of address translation tables to perform the converting.

15 2. The method of claim 1, wherein the first portion of bits comprises high order bits of the address and the second portion of bits comprises low order bits of the address, the low order bits determined based on a size of an assigned address space that includes the memory address.

20 3. The method of claim 1, wherein a number of levels of address translation tables is based on at least one of a size of an assigned address space that includes the memory address, a size of one or more address translation tables to be used in the converting, and a size of the unit of memory accessed by the memory address.

25 4. The method of claim 1, wherein the converting comprises selecting an address translation table to be used to convert the address, the selecting using a pointer in a device table entry used in the converting.

30 5. The method of claim 4, wherein the method further comprises locating the device table entry, the locating using at least one of a requestor identifier of the adapter issuing a request that includes the address to be translated or a portion of the address.

6. The method of claim 4, wherein the method further comprises determining a format of the selected address translation table, the determining using a format field of the device table entry.

5 7. The method of claim 4, wherein the address range is defined by at least one of a base field and a limit field of the device table entry to validate the address.

8. The method of claim 7, wherein the validating the address comprises:

10 permitting execution responsive to determining that the first portion of bits are within a range of corresponding bits of address range, wherein the corresponding bits of the address range are not all 0's; and

no permitting execution responsive to determining that the first portion of bits are not within a range of corresponding bits of the address range, wherein the corresponding bits of the address range are not all 0's.

15 9. The method of claim 1, wherein the adapter comprises a function, the function comprising a Peripheral Component Interconnect function.

20 10. A computer system for translating addresses in a computing environment, the computer system comprising:

a memory;

a processor in communications with the memory;

25 an address obtainer for obtaining an address from an adapter to be translated, the address comprising a plurality of bits, the plurality of bits comprising a first portion of bits and a second portion of bits;

a receiver for receiving an address range value that indicates a range of permitted addresses;

a validator for validating the address using at least the first portion of bits and the received address range; and

30 a converter for converting the address to a memory address usable in accessing memory of the computing environment, the converting ignoring the first portion of bits and

using the second portion of bits to obtain address information from one or more levels of address translation tables to perform the converting.

11. A computer program comprising computer program code stored on a computer readable medium to, when loaded into a computer system and executed thereon, cause said computer system to perform all the steps of a method according to any of claims 1 to 9.

12. A computer system for translating addresses in a computing environment, the computer system comprising:

a memory; and

a processor in communications with the memory, wherein the computer system is configured to perform a method, said method comprising:

obtaining an address from an adapter to be translated, the address comprising a plurality of bits, the plurality of bits comprising a first portion of bits and a second portion of bits;

receiving an address range value that indicates a range of permitted addresses;

validating the address using at least the first portion of bits and the received address range; and

converting the address to a memory address usable in accessing memory of the computing environment, the converting ignoring the first portion of bits and using the second portion of bits to obtain address information from one or more levels of address translation tables to perform the converting.

13. The computer system of claim 12, wherein the first portion of bits comprises high order bits of the address and the second portion of bits comprises low order bits of the address, the low order bits determined based on a size of an assigned address space that includes the memory address.

14. The computer system of claim 12, wherein a number of levels of address translation tables is based on at least one of a size of an assigned address space that includes the memory address, a size of one or more address translation tables to be used in the converting, and a size of the unit of memory accessed by the memory address.

15. The computer system of claim 12, wherein the converting comprises selecting an address translation table to be used to convert the address, the selecting using a pointer in a device table entry used in the converting.

5 16. The computer system of claim 15, wherein the method further comprises locating the device table entry, the locating using at least one of a requestor identifier of the adapter issuing a request that includes the address to be translated or a portion of the address.

10 17. The computer system of claim 15, wherein the method further comprises determining a format of the selected address translation table, the determining using a format field of the device table entry.

18. The computer system of claim 15, wherein the address range is defined by at least one of a base field and a limit field of the device table entry to validate the address.

15 19. The computer system of claim 18, wherein the validating the address comprises:
permitting execution responsive to determining that the first portion of bits are within a range of corresponding bits of address range, wherein the corresponding bits of the address range are not all 0's; and

20 no permitting execution responsive to determining that the first portion of bits are not within a range of corresponding bits of the address range, wherein the corresponding bits of the address range are not all 0's.

25 20. The computer system of claim 12, wherein the adapter comprises a function, the function comprising a Peripheral Component Interconnect function.

21. A computer program product for translating addresses in a computing environment, the computer program product comprising:

30 a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method according to any of claims 1 to 9.