PA-RISC 2.0 Architecture

This document is an excerpt from PA-RISC 2.0 ARCHITECTURE, by Gerry Kane, published by Prentice Hall PTR, isbn 0-13-182734-0. Print copies of the book can be ordered at www.hp.com/hpbooks.

This document contains:
  Chapter 3: Addressing and Access Control
3 Addressing and Access Control

Data storage is organized as a storage hierarchy based on speed of access: user-accessible registers are at the highest level followed by the memory system which consists of high-speed buffers that hold recently referenced instructions and/or data, and main memory. The high-speed buffers, called instruction and/or data caches, reduce the effective access time to main memory.

The I/O system is memory-mapped with I/O modules mapped into physical pages that, although not part of the main memory, are addressed in the same way. With virtual pages mapped into physical pages and I/O registers represented by words in a page, communication between a processor and an I/O module can be performed with load and store instructions to virtual addresses. The privilege level and access rights of such a page provide versatile protection. Non-privileged code may therefore be given direct access to some I/O modules without compromising system security.

PA-RISC processors use byte addressing to fetch instructions and data from main memory or the I/O registers. The byte addresses may be either virtual addresses or absolute addresses. Virtual addresses are translated to absolute addresses and undergo protection and access rights checking. Memory accesses using virtual addresses are called virtual accesses. When absolute addresses are used directly, no protection or access rights checks are performed. Memory accesses using absolute addresses are called absolute accesses.

The instructions that reference memory are load (memory-to-register), store (register-to-memory), and semaphore instructions. Additionally, several system control and cache-related instructions generate addresses that use the address translation, protection, and access rights checking mechanisms. Computation instructions do not reference memory, but perform data transformations by using values obtained from general registers and returning results to these registers.

Physical and Absolute Addressing

Objects in the main memory and I/O system reside in a 64-bit physical address space and can be accessed using byte addresses which may be either virtual addresses or absolute addresses. The physical address space and absolute accesses are described in the paragraphs that follow. Virtual accesses are described later in this chapter.

Physical Address Space

The Physical Address Space is 64 bits in size as shown in Figure 3-1 and has three components:

- Memory Address Space - Addresses 0 through 0xFFFFFFFF FFFFFFFF can reference 15 Exabytes of memory. This space represents 15/16ths of the Physical Address Space.
- PDC Address Space - Addresses 0xF0000000 00000000 through 0xF0FFFFFF FFFFFFFF reference Processor Dependent Code (PDC) and it associated resources. This space represents 1/256th of the Physical Address Space.
- I/O Address Space - Addresses 0xF1000000 00000000 through 0xFFFFFFFFFF FFFFFFFF can reference nearly 1 Exabyte of I/O registers. The I/O and PDC Address Spaces together represent 1/16th of the Physical Address Space.
Although software views the Physical Address Space as being 64 bits in size, implementations are only required to support physical address spaces between 32 and 64 bits in size. If less than 64 bits of physical address space are supported, the following rules must be observed:

- The Memory, PDC, and I/O Address Spaces must each occupy the same fraction of the implemented physical address space as they do in the 64-bit physical address space, as shown in Figure 3-2.

- In an n-bit physical address space implementation, implementations must ignore the most significant 64-n bits of a 64-bit physical address for references to the Memory and I/O Address Spaces. For references to the PDC Address Space, implementations may transform a 64-bit physical address into an n-bit physical address in a processor-specific fashion provided the eight most significant bits of both addresses are identical.
Absolute Accesses

Accesses made with LOAD WORD ABSOLUTE and STORE WORD ABSOLUTE instructions, or when virtual address translation is disabled (the PSW D-bit is 0 for data accesses or the PSW C-bit is 0 for instruction accesses) are called absolute accesses.

Absolute Accesses when PSW W-bit is 1

When the PSW W-bit is 1 (see “Processor Status Word (PSW)” on page 2-7 for the definition of the PSW W-bit), an absolute address is a 62-bit unsigned integer whose value is the address of the lowest-addressed byte of the operand it designates (see Figure 3-3).

<table>
<thead>
<tr>
<th>ne</th>
<th>Absolute Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>62</td>
</tr>
</tbody>
</table>

Figure 3-3. 62-bit Absolute Pointer

Figure 3-2. n-bit Physical Address Space Implementation
Refer to “Absolute Accesses when PSW W-bit is 1” on page H-10 for details on address formation for these accesses.

Absolute Accesses when PSW W-bit is 0

When the PSW W-bit is 0, an absolute address is a 32-bit unsigned integer whose value is the address of the lowest-addressed byte of the operand it designates (see Figure 3-4).

<table>
<thead>
<tr>
<th>non-existent</th>
<th>Absolute Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 3-4. 32-bit Absolute Pointer

Refer to “Absolute Accesses when PSW W-bit is 0” on page H-11 for details on address formation for these accesses.

Memory Addressable Units and Alignment

Memory is always referenced with byte addresses, starting with address 0 and extending through the largest defined non-I/O address (0xFFFFFFFF FFFFFFFF). Addressable units are bytes, halfwords (2 bytes), words (4 bytes), and doublewords (8 bytes). A comparison of the addressable units is shown in Figure 3-5 with the relative byte numbers indicated inside the blocks.

All addressable units must be stored on their naturally aligned boundaries. A byte may appear at any address, halfwords must begin at even addresses, words must begin at addresses that are multiples of 4, and doublewords must begin at addresses that are multiples of 8. If an unaligned virtual address is used, an interruption occurs.

Bits within larger units are always numbered from 0 starting with the most significant bit.

I/O address space is referenced in doublewords, words, halfwords, and bytes. I/O registers are accessed using the normal load and store instructions.
Virtual Addressing

Virtual memory is organized into linear spaces. These spaces can range in size from $2^{32}$ bytes each to $2^{64}$ bytes each. The object within the space is specified by a 32-bit to 64-bit offset. The space identifier is combined with the offset to form a complete global virtual address (GVA). The offset and space portions are aligned as shown, and bits 34..63 of the space are ORed together with bits 2..31 of the offset to form the GVA. The lower 32 bits of the GVA come directly from the offset, and the upper 34 bits come directly from the space.

A bit in the Processor Status Word (PSW[W]) provides compatibility with older programs. When PSW[W] is 0, offsets are truncated to 32 bits (the upper 32 bits of the offset are forced to 0). The GVA is then formed in the same way, by ORing the offset with the space. Since the offset is truncated, though, this is simply equivalent to concatenating the space with the lower 32 bits of the offset.

Implementations also provide an implementation-dependent, software-writable default width bit. The default width bit controls whether the PSW W-bit is set to 0 or 1 on interruptions, and also whether the EIRR is treated as a 32-bit or a 64-bit register. (See “Processor Status Word (PSW)” on page 2-7.)

Translation from virtual to absolute addresses is accomplished by translation lookaside buffers (TLBs), which are described in Chapter 3, “Addressing and Access Control”. Fields in the TLB entry for a particular page permit control of access to the page for reading, writing or execution. Such access may be restricted to a single process, or a set of processes, or may be permitted to all processes.

To a user application, the virtual address space appears to be flatly addressable and 64 bits in size. User applications are concerned only with the 64-bit address offset. Full support for 32-bit applications with 32-bit pointers is also provided.

To Operating System software, the address space can be thought of as consisting of a set of address spaces, each with its own space identifier, and where each address space can be between 32 and 62 bits in size, depending on the needs of the individual application. For example, an implementation with 32-bit space identifiers would allow for 4 billion 32-bit spaces, or 1 million 44-bit spaces, or 4 62-bit spaces, or any combinations of these. Space identifiers can range up to 64 bits in size, allowing for a 96-bit virtual address. The virtual address model in PA-RISC provides a powerful means for efficiently managing a large address space.
For memory management purposes, the address space is logically subdivided into pages, each of which can range in size from 4 Kbytes to 64 Mbytes in length. The byte offset into the page is specified by the least significant 12 to 26 bits of the virtual address, depending on the page size. Figure 3-7 illustrates the structure of spaces, pages, and offsets.

![Figure 3-7. Structure of the Virtual Address Space](image)

**Pointers and Address Specification**

For virtual accesses, addresses can be specified two different ways. With explicit pointers, an instruction computes an address offset and explicitly specifies a space identifier. This provides efficient access to the entire global virtual address space. With implicit pointers, an instruction computes an address offset, and this offset calculation implicitly specifies a space ID. This provides the appearance of a single 64-bit (or 32-bit) flat address space. The offset and space ID are combined to form the full virtual address.

Eight Space Registers hold space identifiers used in forming virtual addresses. Additionally, the Instruction Address Space Queue holds the space ID for the current instruction address, and two control registers are used to hold space ID information after interruptions.

**Data Addresses**

Data addresses are computed for regular memory reference instructions (load, store, and semaphore instructions) and for system instructions used in managing the address space (access probe, and data cache and data TLB control instructions). A 64-bit offset is calculated by adding a 64-bit base register.
plus a 64-bit index register or a sign-extended immediate displacement. One of the Space Registers is selected, either implicitly, by the top two bits of the base register, or explicitly, by a field in the instruction. The space ID is then logically ORed with the offset of form the virtual address.

The space identifier is selected from Space Registers 1 through 7 as follows, based on the presence and value of the 2-bit $s$ field in the memory reference instruction.

- If the instruction does not have an $s$ field, or if the value of the $s$ field is zero, space ID selection is implicit. The top two bits of the base register are used to select one of Space Registers 4 through 7. This permits the addressing of four distinct spaces selected by program data, and is called implicit pointer addressing, since a regular 64-bit value specifies the offset and space ID for a full virtual address.

- If the instruction does have an $s$ field, and the value of the $s$ field is non-zero, the $s$-field explicitly selects one of Space Registers 1, 2 or 3. Figure 3-8 illustrates space identifier selection.

### Instruction Addresses

Instruction addresses for instruction fetch are computed from the IA queues and as a result of branch target calculations. Instruction addresses are also computed for system instructions used in managing the address space (instruction cache and instruction TLB control instructions).

The current instruction address (IA) consists of a space identifier and a 64-bit byte offset. The byte offset is a word-aligned address and contains, in its least significant two bit positions, the current privilege level. This privilege level controls both instruction and data references. The current instruction address is maintained in the front elements of the Instruction Address Queues (IA queues).

In forming instruction addresses, the space ID can either remain unchanged from the last address (as with in-line instruction fetching and with intraspace branches), or the space ID can be selected from one of the Space Registers. The selection of the Space Register is either implicit (one of Space Registers 4 through 7 selected by the top two bits of the base register), explicit with a 2-bit $s$ field, like data addressing (one of SRs 1, 2 or 3 selected by the instruction), or explicit with a 3-bit $s$ field (one of SRs 0 through 7 selected by the instruction). See Figure 3-8.

As with data addresses, the space ID is logically ORed with the offset to form the virtual address.

Executing or branching beyond the end of the current space is undefined.
32-bit Addresses

Programs which use 32-bit data and 32-bit pointers are fully supported. A bit in the Processor Status Word (PSW W-bit) is used to control address formation. For 32-bit programs, address offset calculations are done just as for 64-bit programs, but then the offset is truncated to a 32-bit value. Space identifier selection for implicit pointers is done with the upper two bits of the lower 32-bits, just as in PA-RISC 1.1.

For 32-bit PA-RISC 1.1 programs that directly manipulate space identifiers to gain access to an address space larger than 32 bits, Space Registers appear to be 32 bits in size, and the virtual address is formed just as in PA-RISC 1.1. (The space identifier is concatenated with the lower 32-bits of the address offset to form the virtual address. This happens automatically as a result of the offset being truncated to a 32-bit value.)

In brief, the addressing model of PA-RISC 2.0 is fully compatible with 32-bit programs written for PA-RISC 1.1.

Absolute Addresses

For absolute accesses, the space identifiers are unused, and the absolute address is calculated from the offset alone. If the most significant 4 bits of the offset are 1, the address accesses the I/O address space and the absolute address is simply equal to the offset. (See “Absolute Accesses” on page 3-3.) If not all of the most significant 4 bits of the offset are 1, the address accesses the memory address space, and the

---

Figure 3-8. Space Identifier Selection

---

<table>
<thead>
<tr>
<th>Space Registers</th>
<th>s-field</th>
<th>Space ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR[7]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implicit pointer addressing

Explicit pointer addressing with 2-bit s-field

Explicit pointer addressing with 3-bit s-field
absolute address is formed by taking the offset and forcing the most significant 2 bits to 0.

This way of forming absolute address for memory allows software more flexibility in address space layout. Note that this has no impact on machine implementing fewer than 62 bits of physical address.

**Address Resolution and the TLB**

Virtual addresses are translated to absolute addresses using a hardware structure called the Translation Lookaside Buffer (TLB). A TLB accepts a Virtual Page Number and returns the corresponding Physical Page Number. The TLB is organized as two parts. The instruction TLB (ITLB) is only used for instruction references, while the data TLB (DTLB) is only used for data references. A system may implement a combined TLB which is used for both instruction and data references.

A TLB is typically not large enough to hold all the current translations. Translations for all pages in memory are stored in a memory structure called the Page Table. Multiple page sizes are supported, from 4 Kbytes to 64 Mbytes. This allows large contiguous regions to be mapped with a single TLB entry. This increases the virtual address range of the TLB, thereby minimizing the virtual address translation overhead.

Given a virtual address, the selected TLB is searched for an entry matching the Virtual Page Number. If the entry exists, the 38 to 52-bit Physical Page Number (contained in the TLB entry) is concatenated with the original 12 to 26-bit page offset (depending on the page size in the matching entry) to form a 64-bit absolute address. If no such entry exists, the TLB is updated by either software TLB miss handling or hardware TLB miss handling.

In systems with software TLB miss handling, a TLB miss fault interruption routine performs the translation, explicitly inserts the translation and protection fields into the appropriate TLB, and restarts the interrupted instruction. To insure the completion of instructions, the TLBs must be organized to simultaneously hold all necessary translations.

In implementations that provide hardware for TLB miss handling, the hardware attempts to find the virtual to physical page translation in the Page Table. If the hardware is successful, it inserts the translation and protection fields into the appropriate instruction or data TLB. No interruption occurs in this case. If hardware is not successful, due to a search of the Page Table that was not exhaustive or due to the appropriate translation not existing in the Page Table, an interruption occurs so that the software can complete the process.

The translation lookaside buffer performs other functions in addition to the basic address translation. The other functions include access control, program debugging support and operating system support for virtual memory. Figure 3-9 summarizes the information maintained for each TLB entry.
The following describes the function of each of the 1-bit fields.

O  Ordered. When 0, data memory references using this translation (except those explicitly marked as ordered or strongly ordered) may be weakly ordered. When 1 and the PSW[O] bit is 1, data memory references using this translation are ordered. See “Ordering of References” on page G-1.

U  Uncacheable. When 0, data references to a page from memory address space may be moved into the cache. When 1, data references to a page from I/O address space or memory address space must not be moved into the cache. The U-bit must be set to 1 for pages which map to the I/O address space, and is commonly set to 1 for pages in the memory address space where I/O module written data and processor written data must co-exist within the same cache line. Referencing a page which maps to the I/O address space and for which the U-bit is 0 is an undefined operation. Implementation of the U-bit is optional. See “Data Cache Move-In” on page F-8 for additional details.

T  Page Reference Trap. When 1, data references using this translation cause a page reference trap. The T-bit is most commonly used for program debugging.

D  Dirty. When 0, store and semaphore instructions cause a TLB dirty bit trap. When 1, no trap occurs. The D-bit may be used by the operating system to determine which pages have been modified.

B  Break. When 1, instructions that could modify data using this translation (store and semaphore instructions, and the PURGE DATA CACHE instruction) cause a data memory break trap, if enabled. The B-bit is most commonly used for program debugging.

P  Prediction method for branching. When 0, branch prediction is performed based on static prediction hints encoded in the instructions. When 1, branch prediction is performed based on dynamic prediction hardware, on implementations so equipped. This bit functions solely as a performance hint. Implementation of the P-bit is optional.

Since the ITLB is not used for data operands, the O, U, T, D, and B bits are only implemented in the

Virtual Page Number
52 to 84 bits

Physical Page Number
52 bits

Page Size
4 bits

Access Rights
7 bits

Access ID
15 to 31 bits

O U T D B P Single-Bit Flags

Figure 3-9. TLB Fields
DTLB or a combined TLB. Similarly, since the P-bit controls branch prediction, it is only implemented in the ITLB or combined TLB.

The TLB is managed by a mixture of hardware and software mechanisms. Translations are brought into the TLB by either hardware or software when a TLB miss occurs. In systems which provide hardware for TLB miss handling, the Page Table holds the information needed for the TLB. For systems with software TLB miss handling, and for explicit insertion of a translation by systems with hardware TLB miss handling, TLB management instructions provide the TLB with this information. The INSERT INSTRUCTION TLB TRANSLATION instruction places the complete translation and access control information into the ITLB. A similar instruction (INSERT DATA TLB TRANSLATION) places the complete translation and access control information and also initializes the system software and debugging support bit fields in the DTLB.

TLB miss traps do not occur on nullified instructions.

Page Size

The TLBs support a range of page sizes, in multiples of four, from 4 Kbytes to 64 Mbytes. Each page is aligned to an address which is an integer multiple of its size. The page size is inserted into the TLB with each translation, and is encoded as shown in Table 3-1. TLB purge instructions can also specify a page size, allowing a large contiguous address range to be purged in a single instruction.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Page size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4 KB</td>
</tr>
<tr>
<td>1</td>
<td>16 KB</td>
</tr>
<tr>
<td>2</td>
<td>64 KB</td>
</tr>
<tr>
<td>3</td>
<td>256 KB</td>
</tr>
<tr>
<td>4</td>
<td>1 MB</td>
</tr>
<tr>
<td>5</td>
<td>4 MB</td>
</tr>
<tr>
<td>6</td>
<td>16 MB</td>
</tr>
<tr>
<td>7</td>
<td>64 MB</td>
</tr>
<tr>
<td>8-15</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Access Control

User processes can be provided with a secure and protected environment via a part of the architecture’s address translation mechanism. Processor resources, including the PSW, Control Registers, and TLB entries, contain information used to determine the allowed use of a page. Access control is available only when address translation is enabled, and is done on a per-page basis.

An access is validated if the check of the access rights and the protection identifiers both succeed. If the access is validated, the instruction reference or data reference is completed. If the access is not validated, the instruction is terminated with a protection trap. Instruction access violations are reported with instruction memory protection traps. Data read and write access violations are reported with data memory access rights or data memory protection ID traps. Probe instructions are special; they save the
result of the access validation in a General Register and do not cause a protection trap. An access rights check is based on the type of access and the current privilege level. The protection identifier check compares the Protection ID Registers with a page-based access identifier in the TLB. State bits within the PSW determine when these checks are enabled.

**Process Attributes**

The type of access, privilege level, the current values in the Protection ID Registers, and the state of the PSW completely describes the access to the TLB. These resources are managed for each process by the operating system and collectively termed the process attributes. The following defines each of the process attributes.

**Privilege Level (PL)**

Every instruction is fetched and executed at one of four privilege levels (numbered 0, 1, 2, 3) with 0 being the most privileged. The privilege level is kept in the least significant two bits of the current instruction’s address (the front element of IAOQ). For all accesses, except the probe instructions, the privilege check uses the privilege level of the current instruction. The probe instructions explicitly specify the privilege level to be used in the access rights check.

**Access type**

The access type is either read, write, or execute. Load, semaphore, and read probe instructions make read accesses to their operands. Store, semaphore and write probe instructions and cache purge operations make write accesses to their operands. Note that semaphore instructions make both read and write accesses to their operands. An execute access occurs when an instruction is fetched for execution.

**Protection IDs**

The four Control Registers CR 8, CR 9, CR 12, and CR 13 contain the protection identifiers associated with the current process (Figure 3-10). These registers are used to allow several different protection groups to be accessed. The least significant bit of each protection ID is the write-disable (WD) bit. When 0, write accesses that match that protection ID are allowed. The remaining 15 to 31 bits hold the protection ID. Figure 3-10 depicts the maximum width of the protection identifier.

![Figure 3-10. Protection ID](image)

**PSW access attributes**

The PSW protection validation (P-bit), code address translation (C-bit), and data address translation (D-bit) bits further qualify the process attributes. When address translation is enabled and the P-bit is 1, the protection ID check is performed. When 0, the protection ID check is always considered successful. An execute access uses the
C-bit to determine if address translation and access rights check are enabled. When 1, address translation is performed and execute access rights checks are made. When 0, no address translation is performed and the access is always allowed. Read and write accesses use the D-bit in an equivalent manner. For probe instructions, address translation is performed, and access rights checks are made independent of the state of the PSW D-bit.

Access ID and Access Rights

For each entry in the TLB, the access ID and the access rights fields determine if an access is allowed. The access ID is a 15- to 31-bit field in the TLB that is used with the protection IDs in the protection ID check. The length of the access ID is implementation dependent but must match the length of the protection ID (excluding the WD bit).

The access rights field (Figure 3-11) is a 7-bit field that encodes the allowed access types and the needed privilege levels. In some cases a minimum privilege is specified, while other access types may be specified with an upper and a lower bound. The three sub-fields type, PL1 (privilege level 1), and PL2 (privilege level 2) combine to form the access rights field. The type sub-field defines the type of access that can be made to this page. Any of read-only, read/write, read/execute, read/write/execute, or execute-only is allowed. The PL1 sub-field qualifies read and execute accesses. The PL2 sub-field qualifies write and execute accesses.

<table>
<thead>
<tr>
<th>Type</th>
<th>PL1</th>
<th>PL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 3-11. Access Rights Field

The access rights check compares the current privilege level with the appropriate sub-field of the TLB access rights field and checks if the type of access is allowed. For a read access, the current privilege level must be at least as privileged as PL1 and the type field must allow read access. The read probe instructions explicitly specify the privilege level.

For a write access, the current privilege level must be at least as privileged as PL2 and the type field must allow write access. The write probe instructions explicitly specify the privilege level.

For an execute access, the current privilege level must be at least as privileged as PL1 and no more privileged than PL2. PL1 and PL2 are a lower and an upper bound, respectively, for execute access. The type field must also allow execute access.

For the PURGE DATA CACHE instruction, if implemented as a purge operation, the access rights check has a special case. The access rights check is done normally, except that if the access rights matches this binary pattern: “111 0X 1X” (where each X stands for either a 1 or a 0), then access is allowed. This facilitates cache management. See “Cache Flushing” on page F-10. If PURGE DATA CACHE is implemented as a flush operation, then no access rights check is performed.

The type field is also used by the BRANCH instruction with the .GATE (for gateway) completer to specify the new privilege level. When the type value is 4 or greater and the encoded new privilege level is of greater privilege, then promotion occurs at the target of the branch.

Table 3-2 defines the type encodings and the necessary conditions of the PL1 and PL2 fields with the
current privilege level (PL). This table uses the actual binary encoding when doing the privilege level comparison.

The protection identifier check compares the eight Protection ID Registers with the TLB entry’s access ID. This check is validated if one or more of the protection IDs compare equal with the access ID. In case of a write access, the write disable bit of at least one of the matching protection IDs must be zero for the check to be validated. An access ID of zero is special and specifies a public page. A public page always satisfies a protection ID check for any type of access and only an access rights check is performed. If no match occurs and a public page is not being referenced, then the access is not allowed.

The PSW P-bit determines whether the protection ID check is performed. When 0, no protection check occurs and only the access rights check is performed. Figure 3-12 on page 3-15 illustrates the access rights and protection ID checks and the processor resources that participate.

Table 3-2. Access Rights Interpretation

<table>
<thead>
<tr>
<th>Type value (in binary)</th>
<th>Allowed access types and B,GATE promotion</th>
<th>Privilege check</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Read-only: data page</td>
<td>read: PL ≤ PL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: Not allowed</td>
</tr>
<tr>
<td>001</td>
<td>Read/Write: dynamic data page</td>
<td>read: PL ≤ PL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: PL ≤ PL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: Not allowed</td>
</tr>
<tr>
<td>010</td>
<td>Read/Execute: normal code page</td>
<td>read: PL ≤ PL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: PL2 ≤ PL ≤ PL1</td>
</tr>
<tr>
<td>011</td>
<td>Read/Write/Execute: dynamic code page</td>
<td>read: PL ≤ PL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: PL ≤ PL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: PL2 ≤ PL ≤ PL1</td>
</tr>
<tr>
<td>100</td>
<td>Execute: promote to privilege level 0*</td>
<td>read: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: PL2 ≤ PL ≤ PL1</td>
</tr>
<tr>
<td>101</td>
<td>Execute: promote to privilege level 1*</td>
<td>read: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: PL2 ≤ PL ≤ PL1</td>
</tr>
<tr>
<td>110</td>
<td>Execute: promote to privilege level 2*</td>
<td>read: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: PL2 ≤ PL ≤ PL1</td>
</tr>
<tr>
<td>111</td>
<td>Execute: remain at privilege level 3*</td>
<td>read: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write: Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute: PL2 ≤ PL ≤ PL1</td>
</tr>
</tbody>
</table>

*Change of privilege level only occurs if the indicated new value is of higher privilege than the current privilege level; otherwise the target of the BRANCH instruction with the ,GATE completer executes at the same privilege as the BRANCH itself.
Page Table Structure

Address translations are stored in memory in a structure called the Page Table. The exact form of these tables is a software convention, but many aspects of the page tables are common and are described in this section.

The most common use of the Page Table is to translate a virtual address to a physical address after a TLB miss. The virtual address space is quite large, and a traditional approach of a multi-level forward
mapped table, where each level is directly indexed by a portion of the virtual address, requires too many memory accesses and hence is an inefficient way to provide virtual to physical translations.

A better approach is to index the Page Table using the result of a hash function applied to the virtual address. The purpose of the hash function is to translate virtual addresses to a smaller, more uniform name space. The particular function used is implementation dependent. Collisions created by multiple addresses hashing to the same entry can be resolved using a sequentially searched linked list or some other structure.

The number of entries in the Page Table is typically a power of two. One possible format of a table entry is shown in Figure 3-13.

![Page Table Entry](image)

The fields are:
- **V** is the valid bit. If V = 1, this entry represents a valid translation.
- **Tag** is a unique key used to identify the virtual address that this entry translates.
- **R** is the reference bit. If R = 1, the page has been accessed (read, write, or execute) by a processor since the bit was last cleared to 0.
- **Physical Page Number** is the physical page number corresponding to the virtual address, provided this entry is valid and the virtual address matches the tag.
- **Size** is the page size, encoded as in Table 3-1 on page 3-11.
- **Next Page Table Entry** is an index/pointer to perhaps another structure containing overflow page table entries.
- **0** is a reserved bit field.
- **s** is a bit field reserved for operating system use.

The O, U, T, D, B, P, Access Rights, and Access ID fields correspond to those for TLB entries (see “Address Resolution and the TLB” on page 3-9).

**Caches**

Caches are high-speed intermediate storage buffers which contain recently accessed instructions and data. The caches are visible to software due to the fact that:

- The architecture supports virtually-indexed caches
• Hardware does not maintain coherence between the instruction cache and the data cache
• In some systems, hardware does not maintain coherence between I/O and processor caches.

For these reasons, the caches are managed by software in certain circumstances.

System software can control which portions of memory may be brought into cache. Additionally, software can explicitly remove items from the cache. As a result, software can control which portions of memory may be present in the cache. In some situations, such as self-modifying code, the use of non-equivalent address aliasing, and coordination with non-coherent I/O, software uses this control of the caches to effect coherence.

Items in the cache may be removed by hardware at any time. Software may therefore not rely on particular items remaining in the cache.

A consistent software view of cache operation requires that implementations never write a clean cache line back to memory. (A cache line can be 16, 32, or 64 bytes in length.) Clean means “not stored into” as opposed to “not changed”. Dirty means “stored into”. A cache line which was stored into in such a way that it was unchanged is considered to be dirty.

To insure memory system coherence, and to minimize cache flushing, instructions and data in memory may be brought into the caches only under certain circumstances. This operation of bringing information from memory into a cache is referred to as move-in. In general, when address translation is enabled, any data or instructions for which there is a valid translation in the TLB may be moved in. When translation is disabled, generally only data or instructions referenced by executed instructions may be moved in. Software may use reference bits and other mechanisms controlled by interruptions to determine when lines are potentially in the instruction cache, data cache, or both. See “Cache Move-in Restrictions” on page F-7.

The U (uncacheable) bit in the data TLB entry also affects caching. A page from the memory address space which has its U-bit set to 0 is called a cacheable page. Pages from the I/O address space and pages which have their U-bit set to 1 are called uncacheable pages. It is possible for data cache lines from an uncacheable page to exist in a data cache. This case may be caused by changing a cacheable page to uncacheable after references to this page were moved into the data cache. Changing the state of the U-bit for a page has no effect on the data cache lines from that page which already exist in the cache.