Chapter 7
Memory Management

This chapter describes the memory management unit (MMU) specifications provided by the PowerPC operating environment architecture (OEA) for PowerPC processors. The primary functions of the MMU in a PowerPC processor are to translate logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses, I/O accesses (most I/O accesses are assumed to be memory-mapped), and direct-store interface accesses. In addition, the MMU provides various levels of access protection on a segment, block, or page basis. Note that there are many aspects of memory management that are implementation-dependent. This chapter describes the conceptual model of a PowerPC MMU; however, PowerPC processors may differ in the specific hardware used to implement the MMU model of the OEA, depending on the many design trade-offs inherent in each implementation.

Two general types of accesses generated by PowerPC processors require address translation—instruction accesses, and data accesses to memory generated by load and store instructions. In addition, the addresses specified by cache instructions and the external control instructions also require translation. Generally, the address translation mechanism is defined in terms of segment descriptors and page tables used by PowerPC processors to locate the effective to physical address mapping for instruction and data accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the virtual address to a physical address.

The definition of the segment and page table data structures provides significant flexibility for the implementation of performance enhancement features in a wide range of processors. Therefore, the performance enhancements used to store the segment or page table information on-chip vary from implementation to implementation.

Translation lookaside buffers (TLBs) are commonly implemented in PowerPC processors to keep recently-used page address translations on-chip. Although their exact characteristics are not specified in the OEA, the general concepts that are pertinent to the system software are described.

The segment information, used to generate the interim virtual addresses, is stored as segment descriptors. These descriptors may reside in on-chip segment registers (32-bit implementations) or as segment table entries (STEs) in memory (64-bit implementations). In much the same way that TLBs cache recently-used page address translations, 64-bit
processors may contain segment lookaside buffers (SLBs) on-chip that cache recently-used segment table entries. Although the exact characteristics of SLBs are not specified, there is general information pertinent to those implementations that provide SLBs.

**Temporary 64-Bit Bridge**

The OEA defines an additional, optional bridge to the 64-bit architecture that may make it easier for 32-bit operating systems to migrate to 64-bit processors. The 64-bit bridge retains certain aspects of the 32-bit architecture that otherwise are not supported, and in some cases not permitted, by the 64-bit version of the architecture. In processors that implement this bridge, segment descriptors are implemented by using 16 SLB entries to emulate segment registers, which, like those defined for the 32-bit architecture, divide the 32-bit memory space (4 Gbytes) into sixteen 256-Mbyte segments. These segment descriptors however use the format of the segment table entries as defined in the 64-bit architecture and are maintained in SLBs rather than in architecture-defined segment registers.

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs).

The MMU, together with the exception processing mechanism, provides the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. Exception processing is described in Chapter 6, “Exceptions.” Section 2.3.1, “Machine State Register (MSR),” describes the MSR, which controls some of the critical functionality of the MMU. (Note that the architecture specification refers to exceptions as interrupts.)

### 7.1 MMU Features

The memory management specification of the PowerPC OEA includes models for both 64- and 32-bit implementations. The MMU of a 64-bit PowerPC processor provides \(2^{64}\) bytes of effective address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. PowerPC processors also have a block address translation (BAT) mechanism for mapping large blocks of memory. Block sizes range from 128 Kbyte to 256 Mbyte and are software-selectable. In addition, the MMU of 64-bit PowerPC processors uses an interim virtual address (80 bits) and hashed page tables in the generation of physical addresses that are \(\leq 64\) bits in length.

The MMU of a 32-bit PowerPC processor is similar except that it provides 4 Gbytes of effective address space, a 52-bit interim virtual address and physical addresses that are \(\leq 32\) bits in length. Table 7-1 summarizes the features of PowerPC MMUs for 64-bit implementations and highlights the differences for 32-bit implementations.
Chapter 7. Memory Management

Note that this chapter describes address translation mechanisms from the perspective of the programming model. As such, it describes the structure of the page and segment tables, the MMU conditions that cause exceptions, the instructions provided for programming the
MMU, and the MMU registers. The hardware implementation details of a particular MMU
(including whether the hardware automatically performs a page table search in memory)
are not contained in the architectural definition of PowerPC processors and are invisible to
the PowerPC programming model; therefore, they are not described in this document. In
the case that some of the OEA model is implemented with some software assist mechanism,
this software should be contained in the area of memory reserved for implementation-
specific use and should not be visible to the operating system.

7.2 MMU Overview

The PowerPC MMU and exception models support demand-paged virtual memory. Virtual
memory management permits execution of programs larger than the size of physical
memory; the term demand paged implies that individual pages are loaded into physical
memory from backing storage only as they are accessed by an executing program.

The memory management model includes the concept of a virtual address that is not only
larger than that of the maximum physical memory allowed but a virtual address space that
is also larger than the effective address space. Effective addresses generated by 64-bit

---

Temporary 64-Bit Bridge

In addition to the features described above, the OEA provides optional features that
facilitate the migration of operating systems from 32-bit processor designs to 64-bit
processors. These features, which can be implemented in part or in whole, include the
following:

- Support for several 32-bit instructions that are otherwise defined as illegal in 64-bit
  processors. These include the following—\texttt{mtsr}, \texttt{mtsrin}, \texttt{mfsr}, \texttt{mfsrin}
- Additional instructions, \texttt{mtsr} and \texttt{mtsrin}, that allow software to associate
effective segments 0–15 with any of virtual segments 0–(2^{52} – 1) without otherwise
affecting the segment table. These instructions move 64 bits from a specified GPR
to a selected SLB entry.
- The \texttt{rfi} and \texttt{mtmsr} instructions, which are otherwise illegal in the 64-bit
architectures may optionally be implemented in 64-bit implementations.
- The bridge defines the following additional optional bits:
  - \texttt{ASR[V]} (bit 63) may be implemented to indicate whether \texttt{ASR[STABORG]}
    holds a valid physical base address for the segment table.
  - \texttt{MSR[ISF]} (bit 2) is defined as an optional bit that can be used to control the
    mode (64-bit or 32-bit) that is entered when an exception is taken. If the bit is
    implemented, it should have the properties described in Section 7.9.1, “ISF Bit
    of the Machine State Register.” Otherwise, it is treated as reserved, except that
    ISF is assumed to be set for exception processing.

To determine whether a processor implements any or all of the bridge features, consult the
user’s manual for that processor.
implementations are 64 bits wide; in the address translation process, the processor converts
an effective address to an 80-bit virtual address per the information in the selected
descriptor. Then the address is translated back to a 64-bit (or less) physical address. In
32-bit implementations, the process is similar except that the address ranges are smaller.

Note that in the cases that 64-bit (or 32-bit) implementations support a physical address
range that is smaller than 64 bits (or 32 bits), the higher-order bits of the effective address
may be ignored in the address translation process. The remainder of this chapter assumes
that implementations support the maximum physical address range.

The operating system manages the system’s physical memory resources. Consequently, the
operating system initializes the MMU registers (segment registers or address space register
(ASR), BAT registers, and SDR1 register) and sets up page tables (and segment tables for
64-bit implementations) in memory appropriately. The MMU then assists the operating
system by managing page status and optionally caching the recently-used address
translation information on-chip for quick access.

Effective address spaces are divided into 256-Mbyte regions called segments or into other
large regions called blocks (128 Kbyte–256 Mbyte). Segments that correspond to memory-
mapped areas can be further subdivided into 4-Kbyte pages. For each block or page, the
operating system creates an address descriptor (page table entry (PTE) or BAT array entry);
the MMU then uses these descriptors to generate the physical address, the protection
information, and other access control information each time an address within the block or
page is accessed. Address descriptors for pages reside in tables (as PTEs) in physical
memory; for faster accesses, the MMU often caches on-chip copies of recently-used PTEs
in an on-chip TLB. The MMU keeps the block information on-chip in the BAT array
(compromised of the BAT registers).

This section provides an overview of the high-level organization and operational concepts
of the MMU in PowerPC processors, and a summary of all MMU control registers. For
more information about the MSR, see Section 2.3.1, “Machine State Register (MSR).”
Section 7.4.3, “BAT Register Implementation of BAT Array,” describes the BAT registers,
Section 7.5.2.1, “Segment Descriptor Definitions,” describes the segment registers,
Section 7.6.1.1, “SDR1 Register Definitions,” describes the SDR1, and Section 7.7.1.1,
“Address Space Register (ASR),” describes the ASR.

7.2.1 Memory Addressing
A program references memory using the effective (logical) address computed by the
processor when it executes a load, store, branch, or cache instruction, and when it fetches
the next instruction. The effective address is translated to a physical address according to
the procedures described throughout this chapter. The memory subsystem uses the physical
address for the access.

In addition to the 64-and 32-bit memory management models defined by the OEA, the
PowerPC architecture also defines a 32-bit mode of operation for 64-bit implementations.
In this 32-bit mode (MSR[SF] = 0), the 64-bit effective address is first calculated as usual, and then the high-order 32 bits of the EA are treated as zero for the purposes of addressing memory. This occurs for both instruction and data accesses, and occurs independently from the setting of the MSR[IR] and MSR[DR] bits that enable instruction and data address translation, respectively. The truncation of the EA is the only way in which memory accesses are affected by the 32-bit mode of operation.

**Temporary 64-Bit Bridge**

Some 64-bit processors implement optional features that simplify the conversion of an operating system from the 32-bit to the 64-bit portion of the architecture. This architecturally defined bridge allows an operating system to use 16 on-chip SLB entries in the same manner that 32-bit implementations use the segment registers, which are otherwise not supported in the 64-bit architecture. These bridge features are available if the ASR[V] bit is implemented, and they are enabled when both ASR[V] and MSR[SF] are cleared.

For a complete discussion of effective address calculation, see Section 4.1.4.2, “Effective Address Calculation.”

**7.2.2 MMU Organization**

Figure 7-1 shows the conceptual organization of the MMU in a 64-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor and other hardware features (invisible to the system software) not depicted in the figure may be implemented. For example, the memory management function can be implemented with parallel MMUs that translate addresses for instruction and data accesses independently.

The instruction addresses shown in the figure are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Memory addresses are generated by load and store instructions (both for memory and the direct-store interface) and by cache instructions.

As shown in Figure 7-1, after an address is generated, the higher-order bits of the effective address, EA0–EA51 (or a smaller set of address bits, EA0–EAn, in the cases of blocks), are translated into physical address bits PA0–PA51. The lower-order address bits, A52–A63 are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 64-bit physical address to the memory subsystem.

In addition to the higher-order address bits, the MMU automatically keeps an indicator of whether each access was generated as an instruction or data access and a supervisor/user indicator that reflects the state of the PR bit of the MSR when the effective address was generated. In addition, for data accesses, there is an indicator of whether the access is for a load or a store operation. This information is then used by the MMU to appropriately direct
the address translation and to enforce the protection hierarchy programmed by the operating system. See Section 2.3.1, “Machine State Register (MSR),” for more information about the MSR.
Figure 7-1. MMU Conceptual Block Diagram—64-Bit Implementations
As shown in Figure 7-1, processors optionally implement on-chip translation lookaside buffers (TLBs) and optionally support the automatic search of the page tables for page table entries (PTEs).

In 64-bit implementations, the address space register (ASR) defines the physical address of the base of the segment table in memory. The segment table entries (STEs) contain the segment descriptors, which define the virtual address for the segment. Some 64-bit implementations may have dedicated hardware to search for STEs in memory, and copies of STEs may be cached on-chip in segment lookaside buffers (SLBs) for quicker access.

**Temporary 64-Bit Bridge**

Processes that implement the 64-bit bridge implement segment descriptors as a table of 16 segment table entries.

Figure 7-2 shows a conceptual block diagram of the MMU in a 32-bit implementation. This figure is similar to that for 64-bit implementations except that after an address is generated, the higher-order bits of the effective address, EA0–EA19 (or a smaller set of address bits, EA0–EAn, in the cases of blocks), are translated into physical address bits PA0–PA19. The lower-order address bits, A20–A31 are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 32-bit physical address to the memory subsystem.

Also, whereas 64-bit implementations use the ASR and a segment table to generate the 80-bit virtual address, 32-bit implementations use the 16 segment registers to generate the 52-bit virtual address.
Figure 7-2. MMU Conceptual Block Diagram—32-Bit Implementations
7.2.3 Address Translation Mechanisms

PowerPC processors support the following four types of address translation:

- Page address translation—translates the page frame address for a 4-Kbyte page size
- Block address translation—translates the block number for blocks that range in size from 128 Kbyte to 256 Mbyte
- Direct-store address translation—used to generate direct-store interface accesses on the external bus; not optimized for performance—present for compatibility only
- Real addressing mode address translation—when address translation is disabled, the physical address is identical to the effective address

Figure 7-3 shows the four address translation mechanisms provided by the MMU. The segment descriptors shown in the figure control both the page and direct-store segment address translation mechanisms. When an access uses the page or direct-store segment address translation, the appropriate segment descriptor is required. In 64-bit implementations, the segment descriptor is located via a search of the segment table in memory for the appropriate segment table entry (STE). In 32-bit implementations, one of the 16 on-chip segment registers (which contain segment descriptors) is selected by the highest-order effective address bits.

Temporary 64-Bit Bridge

Processors that implement the 64-bit bridge divide the 32-bit address space into sixteen 256-Mbyte segments defined by a table of 16 STEs maintained in 16 SLB entries.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to a direct-store segment. Note that the direct-store interface is present to allow certain older I/O devices to use this interface. When an access is determined to be to the direct-store interface space, the implementation invokes an elaborate hardware protocol for communication with these devices. The direct-store interface protocol is not optimized for performance, and therefore, its use is discouraged. The most efficient method for accessing I/O is by memory-mapping the I/O areas.

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 64-bit (or 32-bit) physical address used by the memory subsystem. In some cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in a TLB, the MMU searches the page table in memory (using the virtual address information and a hashing function) to locate the required physical address. Some implementations may have dedicated hardware to perform the page table search automatically, while others may define an exception handler routine that searches the page table with software.
Block address translation occurs in parallel with page and direct-store segment address translation and is similar to page address translation, except that there are fewer upper-order effective address bits to be translated into physical address bits (more lower-order address bits (at least 17) are untranslated to form the offset into a block). Also, instead of segment descriptors and a page table, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored. Note that a matching BAT array entry takes precedence over a translation provided by the segment descriptor in all cases (even if the segment is a direct-store segment).
Figure 7-3. Address Translation Types—64-Bit Implementations

**TEMPORARY 64-BIT BRIDGE**

Note that Figure 7-3 shows address sizes for a 64-bit processor operating in 64-bit mode. If the 64-bit bridge is enabled (ASR[V] is cleared), only the 32-bit address space is available and only 52 bits of the virtual address are used. However, the bridge supports cross-memory operations that permit an operating system to establish addressability to an address space, to copy data to it from another address space, and then to destroy the new addressability, without altering the segment table. For more information, see Section 7.9.5, “Segment Register Instructions Defined Exclusively for the 64-Bit Bridge.”

Direct-store address translation is used when the direct-store translation control bit (T bit) in the corresponding segment descriptor is set. In this case, the remaining information in the segment descriptor is interpreted as identifier information that is used with the remaining effective address bits to generate the protocol used in a direct-store interface.
access on the external interface; additionally, no TLB lookup or page table search is performed.

Real addressing mode address translation occurs when address translation is disabled; in this case, the physical address generated is identical to the effective address. Instruction and data address translation is enabled with the MSR[IR] and MSR[DR] bits, respectively. Thus, when the processor generates an access, and the corresponding address translation enable bit in MSR (MSR[IR] for instruction accesses and MSR[DR] for data accesses) is cleared, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. See Section 7.2.6.1, “Real Addressing Mode and Block Address Translation Selection,” for more information.

7.2.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMU provides access protection of supervisor areas from user access and can designate areas of memory as read-only as well as no-execute. Table 7-2 shows the eight protection options supported by the MMU for pages.

Table 7-2. Access Protection Options for Pages

<table>
<thead>
<tr>
<th>Option</th>
<th>User Read</th>
<th>User Write</th>
<th>Supervisor Read</th>
<th>Supervisor Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I-Fetch</td>
<td>Data</td>
<td>I-Fetch</td>
<td>Data</td>
</tr>
<tr>
<td>Supervisor-only</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Supervisor-only-no-execute</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Supervisor-write-only</td>
<td>√</td>
<td>√</td>
<td>—</td>
<td>√</td>
</tr>
<tr>
<td>Supervisor-write-only-no-execute</td>
<td>—</td>
<td>√</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Both user/supervisor</td>
<td>—</td>
<td>√</td>
<td>√</td>
<td>—</td>
</tr>
<tr>
<td>Both user/supervisor-no-execute</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Both read-only</td>
<td>√</td>
<td>—</td>
<td>√</td>
<td>—</td>
</tr>
<tr>
<td>Both read-only-no-execute</td>
<td>—</td>
<td>√</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

√ Access permitted
— Protection violation

The operating system programs whether or not instruction fetches are allowed from an area of memory with the no-execute option provided in the segment descriptor. Each of the remaining options is enforced based on a combination of information in the segment descriptor and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (corresponding to MSR[PR] = 0) to access the page. User accesses that map into a supervisor-only page cause an exception to be taken.
Note that independently of the protection mechanisms, care must be taken when writing to instruction areas as coherency must be maintained with on-chip copies of instructions that may have been prefetched into a queue or an instruction cache. Refer to Section 5.1.5.2, “Instruction Cache Instructions,” for more information on coherency within instruction areas.

As shown in the table, the supervisor-write-only option allows both user and supervisor accesses to read from the page, but only supervisor programs can write to that area. There is also an option that allows both supervisor and user programs read and write access (both user/supervisor option), and finally, there is an option to designate a page as read-only, both for user and supervisor programs (both read-only option).

For areas of memory that are translated by the block address translation mechanism, the protection options are similar, except that blocks are translated by separate mechanisms for instruction and data, blocks do not have a no-execute option, and blocks can be designated as enabled for user and supervisor accesses independently. Therefore, a block can be designated as supervisor-only, for example, but this block can be programmed such that all user accesses simply ignore the block translation, rather than take an exception in the case of a match. This allows a flexible way for supervisor and user programs to use overlapping effective address space areas that map to unique physical address areas (without exceptions occurring).

For direct-store segments, the MMU calculates a key bit based on the protection values programmed in the segment descriptor and the specific user/supervisor and read/write information for the particular access. However, this bit is merely passed on to the system interface to be transmitted in the context of the direct-store interface protocol. The MMU does not itself enforce any protection or cause any exception based on the state of the key bit for these accesses. The I/O controller device or other external hardware can optionally use this bit to enforce any protection required.

Finally, a facility defined in the VEA and OEA allows pages or blocks to be designated as guarded, preventing out-of-order accesses that may cause undesired side effects. For example, areas of the memory map that are used to control I/O devices can be marked as guarded so that accesses (for example, instruction prefetches) do not occur unless they are explicitly required by the program. Refer to Section 5.2.1.5.3, “Out-of-Order Accesses to Guarded Memory,” for a complete description of how accesses to guarded memory are restricted.

### 7.2.5 Page History Information

The MMU of PowerPC processors also defines referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the virtual page. This information can then be used by the operating system to determine which areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the
architecture specifies that the R and C bits are maintained by the processor and the processor updates these bits when required.

7.2.6 General Flow of MMU Address Translation
The following sections describe the general flow used by PowerPC processors to translate effective addresses to virtual and then physical addresses. Note that although there are references to the concept of an on-chip TLB and SLB, these entities may not be present in a particular hardware implementation for performance enhancement (and a particular implementation may have one or more TLBs and SLBs). Thus, they are shown here as optional and only the software ramifications of the existence of a TLB or SLB are discussed.

7.2.6.1 Real Addressing Mode and Block Address Translation Selection
When an instruction or data access is generated and the corresponding instruction or data translation is disabled (MSR[IR] = 0 or MSR[DR] = 0), real addressing mode translation is used (physical address equals effective address) and the access continues to the memory subsystem as described in Section 7.3, “Real Addressing Mode.”

Figure 7-4 shows the flow used by the MMU in determining whether to select real addressing mode or block address translation or to use the segment descriptor to select either direct-store or page address translation.
Figure 7-4. General Flow of Address Translation (Real Addressing Mode and Block)

Note that if the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access is determined to be protected (not allowed), an exception (ISI or DSI exception) is generated.

7.2.6.2 Page and Direct-Store Address Translation Selection

If address translation is enabled (real addressing mode translation not selected) and the effective address information does not match with a BAT array entry, then the segment descriptor must be located. Once the segment descriptor is located, the T bit in the segment descriptor selects whether the translation is to a page or to a direct-store segment as shown in Figure 7-5. In addition, Figure 7-5 also shows the way in which the no-execute protection is enforced: if the N bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted.
Figure 7-5. General Flow of Page and Direct-Store Address Translation

Notes:
* Not allowed for instruction accesses (causes ISI exception)
  __ Implementation-specific

---

PowerPC Microprocessor Family: The Programming Environments, Rev 0.1
The segment descriptor is contained in different constructs for 64- and 32-bit implementations as shown in Figure 7-6. For 64-bit implementations, the segment descriptor for each access is located in an STE that resides in a segment table in memory. The base address of this segment table is specified in the address space register (ASR) and the entries of the table are located by using a hashing function. Although it is not architecturally required, hardware implementations may have one or more on-chip SLBs that keep recently-used STEs for quick access.

For 32-bit implementations, the segment descriptor for an access is contained in one of 16 on-chip segment registers; effective address bits EA0–EA3 select one of the 16 segment registers.

**TEMPORARY 64-BIT BRIDGE**

Processors that implement the 64-bit bridge maintain segment descriptors on-chip by emulating segment tables in 16 SLB entries. As shown in Figure 7-6, this feature is enabled by clearing the optional ASR[V] bit. This indicates that any value in the STABORG is invalid and that segment table hashing is not implemented.
7.2.6.2.1 Selection of Page Address Translation

If the T bit in the corresponding segment descriptor is 0, page address translation is selected. The information in the segment descriptor is then used to generate the 80-bit (or 52-bit) virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). Once again, although the architecture does not require the existence of a TLB, one or more
TLBs may be implemented in the hardware to store copies of recently-used PTEs on-chip for increased performance.

If an access hits in the TLB, the page translation occurs and the physical address bits are forwarded to the memory subsystem. If the translation is not found in the TLB, the MMU requires a search of the page table. The hardware of some implementations may perform the table search automatically, while others may trap to an exception handler for the system software to perform the page table search. If the translation is found, a new TLB entry is created and the page translation is once again attempted. This time, the TLB is guaranteed to hit. Once the PTE is located, the access is qualified with the appropriate protection bits. If the access is determined to be protected (not allowed), an exception (ISI or DSI exception) is generated.

If the PTE is not found by the table search operation, an ISI or DSI exception is generated.

### 7.2.6.2.2 Selection of Direct-Store Address Translation

When the segment descriptor has the T bit set, the access is considered a direct-store access and the direct-store interface protocol of the external interface is used to perform the access. The selection of address translation type differs for instruction and data accesses only in that instruction accesses are not allowed from direct-store segments; attempting to fetch an instruction from a direct-store segment causes an ISI exception. See Section 7.8, “Direct-Store Segment Address Translation,” for more detailed information about the translation of addresses in direct-store segments.

### 7.2.7 MMU Exceptions Summary

In order to complete any memory access, the effective address must be translated to a physical address. A translation exception condition occurs if this translation fails for one of the following reasons:

- There is no valid entry in the page table for the page specified by the effective address (and segment descriptor) and there is no valid BAT translation.
- There is no valid segment descriptor and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

The translation exception conditions cause either the ISI or the DSI exception to be taken as shown in Table 7-3. The state saved by the processor for each of these exceptions contains information that identifies the address of the failing instruction. Refer to Chapter 6, “Exceptions,” for a more detailed description of exception processing, and the bit settings of SRR1 and DSISR when an exception occurs. Note that the bit settings shown for the SRR1 register are shown for 64-bit implementations. Since the SRR1 register is a 32-bit register in 32-bit implementations, the value 32 must be subtracted from the bit numbers shown for SRR1 in these cases.
In addition to the translation exceptions, there are other MMU-related conditions (some of them implementation-specific) that can cause an exception to occur. These conditions map to the exceptions as shown in Table 7-4. The only MMU exception conditions that occur when MSR[DR] = 0 are the conditions that cause the alignment exception for data accesses. For more detailed information about the conditions that cause the alignment exception (in particular for string/multiple instructions), see Section 6.4.6, “Alignment Exception (0x00600).” Refer to Chapter 6, “Exceptions,” for a complete description of the SRR1 and DSISR bit settings for these exceptions.

### Table 7-3. Translation Exception Conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page fault (no PTE found)</td>
<td>No matching PTE found in page tables (and no matching BAT array entry)</td>
<td>I access: ISI exception SRR1[33*] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D access: DSI exception DSISR[1] = 1</td>
</tr>
<tr>
<td>Block protection violation</td>
<td>Conditions described in Table 7-10 for block</td>
<td>I access: ISI exception SRR1[36*] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D access: DSI exception DSISR[4] = 1</td>
</tr>
<tr>
<td>Page protection violation</td>
<td>Conditions described in Table 7-20 for page</td>
<td>I access: ISI exception SRR1[36*] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D access: DSI exception DSISR[4] = 1</td>
</tr>
<tr>
<td>No-execute protection violation</td>
<td>Attempt to fetch instruction when SR[N] = 1 or STE[N] = 1</td>
<td>ISI exception SRR1[35*] = 1</td>
</tr>
<tr>
<td>Instruction fetch from direct-store segment</td>
<td>Attempt to fetch instruction when SR[T] = 1 or STE[T] = 1</td>
<td>ISI exception SRR1[35*] = 1</td>
</tr>
<tr>
<td>Instruction fetch from guarded memory</td>
<td>Attempt to fetch instruction when MSR[IR] = 1 and either:</td>
<td>ISI exception SRR1[35*] = 1</td>
</tr>
<tr>
<td></td>
<td>matching xBAT[G] = 1, or no matching BAT entry and PTE[G] = 1</td>
<td></td>
</tr>
</tbody>
</table>

* Bit settings shown for the SRR1 register are shown for 64-bit implementations. Since the SRR1 register is a 32-bit register in 32-bit implementations, the value 32 must be subtracted from the bit numbers shown for SRR1 in these cases.

### Table 7-4. Other MMU Exception Conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dcbz</strong> with W = 1 or I = 1 (may cause exception or operation may be performed to memory)</td>
<td><strong>dcbz</strong> instruction to write-through or cache-inhibited segment or block</td>
<td>Alignment exception (implementation-dependent)</td>
</tr>
<tr>
<td><strong>ldarx, stdcx, lwarx, or stwcx</strong> with W = 1 (may cause exception or execute correctly)</td>
<td>Reservation instruction to write-through segment or block</td>
<td>DSI exception (implementation-dependent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSISR[5] = 1</td>
</tr>
</tbody>
</table>
7.2.8 MMU Instructions and Register Summary

The MMU instructions and registers provide the operating system with the ability to set up the segment descriptors. Additionally, the operating system has the resources to set up the block address translation areas and the page tables in memory.

Note that because the implementation of TLBs and SLBs is optional, the instructions that refer to these structures are also optional. However, as these structures serve as caches of the page table (and segment table, in the case of an SLB), there must be a software protocol for maintaining coherency between these caches and the tables in memory whenever changes are made to the tables in memory. Therefore, the PowerPC OEA specifies that a processor implementing a TLB is guaranteed to have a means for doing the following:

- Invalidating an individual TLB entry
- Invalidating the entire TLB

Similarly, a processor that implements an SLB is guaranteed to have a means for doing the following:

- Invalidating an individual SLB entry (the architecture defines an optional \texttt{slbie} instruction for this purpose)
- Invalidating the entire SLB (the architecture defines an optional \texttt{slbia} instruction for this purpose)
When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.

A processor may implement one or more of the instructions listed in this section to support table invalidation. If an instruction is implemented that matches the semantics of an instruction listed here (and described in this document), the operation will be as described. Alternatively, an algorithm may be specified that performs one of the functions listed above (a loop invalidating individual TLB entries may be used to invalidate the entire TLB, for example), or instructions with different semantics may be implemented.

A processor may also perform additional functions (not described here) as well as those described in the implementation of some of these instructions. For example, an instruction whose semantics are to purge a TLB entry may be implemented so as to purge all TLB entries in a congruence class (that is, all TLB entries indexed by the specified EA which can include corresponding entries in data and instruction TLBs) or the entire TLB.

Note that if a processor does not implement an optional instruction it treats the instruction as a no-op or as an illegal instruction, depending on the implementation. Also, note that the segment register and TLB concepts described here are conceptual; that is, a processor may implement parallel sets of segment registers (and even TLBs) for instructions and data.

Because the MMU specification for PowerPC processors is so flexible, it is recommended that the software that uses these instructions and registers be “encapsulated” into subroutines to minimize the impact of migrating across the family of implementations.

Table 7-5 summarizes the PowerPC instructions that specifically control the MMU. For more detailed information about the instructions, refer to Chapter 8, “Instruction Set.”

### Table 7-5. Instruction Summary—Control MMU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtar SR,rS</td>
<td>Move to Segment Register SR[SR] ← rS</td>
</tr>
<tr>
<td></td>
<td>32-bit implementations and 64-bit bridge only</td>
</tr>
<tr>
<td>mtsrin rS,rB</td>
<td>Move to Segment Register Indirect SR[rB] ← rS</td>
</tr>
<tr>
<td></td>
<td>32-bit implementations and 64-bit bridge only</td>
</tr>
</tbody>
</table>
Table 7-5. Instruction Summary—Control MMU (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
</table>
| **TEMPORARY 64-Bit BRIDGE**
**mtsr** SR,rS | Move to Segment Register Double Word
SLB[SR] ← rS
64-bit bridge only |
| **mtsrin** rS,rB | Move to Segment Register Indirect Double Word
SLB[rB[32-35]] ← (rS)
64-bit bridge only |
| **mfsr** rD,SR | Move from Segment Register
rD ← SR[SR]
32-bit implementations and 64-bit bridge only |
| **mfsrin** rD,rB | Move from Segment Register Indirect
rD ← SR[rB[0–3]]
32-bit implementations and 64-bit bridge only |
| **tlbia** (optional) | Translation Lookaside Buffer Invalidate All
For all TLB entries, TLB[V] ← 0
Causes invalidation of TLB entries only for processor that executed the **tlbia** |
| **tlbie** rB (optional) | Translation Lookaside Buffer Invalidate Entry
If TLB hit (for effective address specified as rB), TLB[V] ← 0
Causes TLB invalidation of entry in all processors in system |
| **tlbsync** (optional) | Translation Lookaside Buffer Synchronize
Ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all processors |
| **slbia** (optional) | Segment Table Lookaside Buffer Invalidate All
For all SLB entries, SLB[V] ← 0
64-bit implementations only |
| **slbie** rB (optional) | Segment Table Lookaside Buffer Invalidate Entry
If SLB hit (for effective address specified as rB), SLB[V] ← 0
64-bit implementations only |

Table 7-6 summarizes the registers that the operating system uses to program the MMU. These registers are accessible to supervisor-level software only (supervisor level is referred to as privileged state in the architecture specification). These registers are described in detail in Chapter 2, “PowerPC Register Set.”
7.2.9 TLB Entry Invalidation

Optionally, PowerPC processors implement TLB structures that store on-chip copies of the PTEs that are resident in physical memory. These processors have the ability to invalidate resident TLB entries through the use of the tlbie and tlbia instructions. Additionally, these instructions may also enable a TLB invalidate signalling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. See Chapter 8, “Instruction Set,” for detailed information about the tlbie and tlbia instructions.

7.3 Real Addressing Mode

If address translation is disabled (MSR[IR] = 0 or MSR[DR] = 0) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as a real addressing mode address translation. If an implementation has a smaller physical address range than effective address range, the extra high-order bits of the effective address may be ignored in the generation of the physical address.

Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” describes the synchronization requirements for changes to MSR[IR] and MSR[DR].

The addresses for accesses that occur in real addressing mode bypass all memory protection checks as described in Section 7.4.4, “Block Memory Protection,” and Section 7.5.4, “Page Memory Protection” and do not cause the recording of referenced and changed information (described in Section 7.5.3, “Page History Recording”).

### Table 7-6. MMU Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment registers</td>
<td>The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC architecture. Figure 7-20 shows the format of a segment register. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the mtsr, mtsrin, mtsr, and mtsrin instructions.</td>
</tr>
<tr>
<td>BAT registers</td>
<td>There are 16 BAT registers, organized as four pairs of instruction BAT registers (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BAT registers (DBAT0U–DBAT3U and DBAT0L–DBAT3L). The BAT registers are defined as 32-bit registers in 32-bit implementations, and 64-bit registers in 64-bit implementations. These are special-purpose registers that are accessed by the mtspr and mfspr instructions.</td>
</tr>
<tr>
<td>SDR1 register</td>
<td>The SDR1 register specifies the base and size of the page tables in memory. SDR1 is defined as a 64-bit register for 64-bit implementations and as a 32-bit register for 32-bit implementations. This is a special-purpose register that is accessed by the mtspr and mfspr instructions.</td>
</tr>
<tr>
<td>Address space register</td>
<td>The 64-bit ASR specifies the physical address in memory of the segment table for 64-bit implementations. This is a special-purpose register that is accessed by the mtspr and mfspr instructions.</td>
</tr>
</tbody>
</table>
For data accesses that use real addressing mode, the memory access mode bits (WIMG) are assumed to be 0b0011. That is, the cache is write-back and memory does not need to be updated immediately (W = 0), caching is enabled (I = 0), data coherency is enforced with memory, I/O, and other processors (caches) (M = 1 so data is global), and the memory is guarded. For instruction accesses in real addressing mode, the memory access mode bits (WIMG) are assumed to be either 0b0001 or 0b0011. That is, caching is enabled (I = 0), and the memory is guarded. Additionally, coherency may or may not be enforced with memory, I/O, and other processors (caches) (M = 0 or 1 so data may or may not be considered global). For a complete description of the WIMG bits, refer to Section 5.2.1, “Memory/Cache Access Attributes.”

Note that the attempted execution of the eciwx or ecowx instructions while MSR[DR] = 0 causes boundedly-undefined results.

Whenever an exception occurs, the processor clears both the MSR[IR] and MSR[DR] bits. Therefore, at least at the beginning of all exception handlers (including reset), the processor operates in real addressing mode for instruction and data accesses. If address translation is required for the exception handler code, the software must explicitly enable address translation by accessing the MSR as described in Chapter 2, “PowerPC Register Set.”

Note that an attempt to access a physical address that is not physically present in the system may cause a machine check exception (or even a checkstop condition), depending on the response by the system for this case. Thus, care must be taken when generating addresses in real addressing mode. Note that this can also occur when translation is enabled and the ASR or SDR1 registers set up the translation such that nonexistent memory is accessed. See Section 6.4.2, “Machine Check Exception (0x00200),” for more information on machine check exceptions.

**TEMPORARY 64-BIT BRIDGE**

Note that if ASR[V] = 0, a reference to a nonexistent address in the STABORG field does not cause a machine check exception.

### 7.4 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

The following sections describe the implementation of block address translation in PowerPC processors, including the block protection mechanism, followed by a block translation summary with a detailed flow diagram.
7.4.1 BAT Array Organization

The block address translation mechanism in PowerPC processors is implemented as a software-controlled BAT array. The BAT array maintains the address translation information for eight blocks of memory. The BAT array in PowerPC processors is maintained by the system software and is implemented as a set of sixteen special-purpose registers (SPRs). Each block is defined by a pair of SPRs called upper and lower BAT registers that contain the effective and physical addresses for the block.

The BAT registers can be read from or written to by the `mfspr` and `mtspr` instructions; access to the BAT registers is privileged. Section 7.4.3, “BAT Register Implementation of BAT Array,” gives more information about the BAT registers. Note that the BAT array entries are completely ignored for TLB invalidate operations detected in hardware and in the execution of the `tlbie` or `tlbia` instruction.

Figure 7-7 shows the organization of the BAT array in a 64-bit implementation. Four pairs of BAT registers are provided for translating instruction addresses and four pairs of BAT registers are used for translating data addresses. These eight pairs of BAT registers comprise two four-entry fully-associative BAT arrays (each BAT array entry corresponds to a pair of BAT registers). The BAT array is fully-associative in that any address can reside in any BAT. In addition, the effective address field of all four corresponding entries (instruction or data) are simultaneously compared with the effective address of the access to check for a match.

The BAT array organization for 32-bit implementations is the same as that shown in Figure 7-7 except that the effective address field to be compared with the BEPI field (block effective page index) in the upper BAT register is EA0–EA14 instead of EA0–EA46.
Each pair of BAT registers defines the starting address of a block in the effective address space, the size of the block, and the start of the corresponding block in physical address space. If an effective address is within the range defined by a pair of BAT registers, its physical address is defined as the starting physical address of the block plus the lower-order effective address bits.

Blocks are restricted to a finite set of sizes, from 128 Kbytes \(2^{17}\) bytes to 256 Mbytes \(2^{28}\) bytes. The starting address of a block in both effective address space and physical address space is defined as a multiple of the block size.

It is an error for system software to program the BAT registers such that an effective address is translated by more than one valid IBAT pair or more than one valid DBAT pair. If this occurs, the results are undefined and may include a spurious violation of the memory protection mechanism, a machine check exception, or a checkstop condition.

The equation for determining whether a BAT entry is valid for a particular access is as follows:

\[
\text{BAT_entry_valid} = (Vs \& \neg\text{MSR[PR]}) \mid (Vp \& \text{MSR[PR]})
\]
If a BAT entry is not valid for a given access, it does not participate in address translation for that access. Two BAT entries may not map an overlapping effective address range and be valid at the same time.

Entries that have complementary settings of V[s] and V[p] may map overlapping effective address blocks. Complementary settings would be as follows:

BAT entry A: Vs = 1, Vp = 0
BAT entry B: Vs = 0, Vp = 1

7.4.2 Recognition of Addresses in BAT Arrays

The BAT arrays are accessed in parallel with segmented address translation to determine whether a particular effective address corresponds to a block defined by the BAT arrays. If an effective address is within a valid BAT area, the physical address for the memory access is determined as described in Section 7.4.5, “Block Physical Address Generation.”

Block address translation is enabled only when address translation is enabled (MSR[IR] = 1 and/or MSR[DR] = 1). Also, a matching BAT array entry always takes precedence over any segment descriptor translation, independent of the setting of the STE[T] (or SR[T]) bit, and the segment descriptor information is completely ignored.

Figure 7-8 shows the flow of the BAT array comparison used in block address translation for 64-bit implementations. When an instruction fetch operation is required, the effective address is compared with the four instruction BAT array entries; similarly, the effective addresses of data accesses are compared with the four data BAT array entries. The BAT arrays are fully-associative in that any of the four instruction or data BAT array entries can contain a matching entry (for an instruction or data access, respectively).

Note that Figure 7-8 assumes that the protection bits, BATL[PP], allow an access to occur. If not, an exception is generated, as described in Section 7.4.4, “Block Memory Protection.”
Two BAT array entry fields are compared to determine if there is a BAT array hit—a block effective page index (BEPI) field, which is compared with the high-order effective address bits, and one of two valid bits (Vs or Vp), which is evaluated relative to the value of MSR[PR]. Note that the figure assumes a block size of 128 Kbytes (all bits of BEPI are used in the comparison); the actual number of bits of the BEPI field that are used are masked by the BL field (block length) as described in Section 7.4.3, “BAT Register Implementation of
BAT Array.” Also, note that the flow for 32-bit implementations is the same as that shown in Figure 7-8 except that the effective address field to be compared with the BEPI field is EA0–EA14 instead of EA0–EA46.

Thus, the specific criteria for determining a BAT array hit are as follows:

- The upper-order 47 bits (or 15 bits for 32-bit implementations) of the effective address, subject to a mask, must match the BEPI field of the BAT array entry.
- The appropriate valid bit in the BAT array entry must set to one as follows:
  - MSR[PR] = 0 corresponds to supervisor mode; in this mode, Vs is checked.
  - MSR[PR] = 1 corresponds to user mode; in this mode, Vp is checked.

The matching entry is then subject to the protection checking described in Section 7.4.4, “Block Memory Protection,” before it is used as the source for the physical address. Note that if a user mode program performs an access with an effective address that matches the BEPI field of a BAT area defined as valid only for supervisor accesses (Vp = 0 and Vs = 1) for example, the BAT mechanism does not generate a protection violation and the BAT entry is simply ignored. Thus, a supervisor program can use the block address translation mechanism to share a portion of the effective address space with a user program (that uses page address translation for this area).

If a memory area is to be mapped by the BAT mechanism for both instruction and data accesses, the mapping must be set up in both an IBAT and DBAT entry; this is the case even on implementations that do not have separate instruction and data caches.

Note that a block can be defined to overlay part of a segment such that the block portion is nonpaged although the rest of the segment can be paged. This allows nonpaged areas to be specified within a segment. Thus, if an area of memory is translated by an instruction BAT entry and data accesses are not also required to that same area of memory, PTEs are not required for that area of memory. Similarly, if an area of memory is translated by a data BAT entry, and instruction accesses are not also required to that same area of memory, PTEs are not required for that area of memory.

### 7.4.3 BAT Register Implementation of BAT Array

Recall that the BAT array is comprised of four entries used for instruction accesses and four entries used for data accesses. Each BAT array entry consists of a pair of BAT registers—an upper and a lower BAT register for each entry. The BAT registers are accessed with the `mtspr` and `mfspr` instructions and are only accessible to supervisor-level programs. See Appendix F, “Simplified Mnemonics,” for a list of simplified mnemonics for use with the BAT registers. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)

Figure 7-9 shows the format of the upper BAT registers and Figure 7-10 shows the format of the lower BAT registers for 64-bit implementations.
Figure 7-9. Format of Upper BAT Registers—64-Bit Implementations

Figure 7-10. Format of Lower BAT Registers—64-Bit Implementations

The format and bit definitions of the upper and lower BAT registers for 32-bit implementations are similar to that of the 64-bit implementations, and are shown in Figure 7-11 and Figure 7-12, respectively.

Figure 7-11. Format of Upper BAT Registers—32-Bit Implementations

Figure 7-12. Format of Lower BAT Registers—32-Bit Implementations

The BAT registers contain the effective-to-physical address mappings for blocks of memory. This mapping information includes the effective address bits that are compared with the effective address of the access, the memory/cache access mode bits (WIMG), and the protection bits for the block. In addition, the size of the block and the starting address of the block are defined by the physical block number (BRPN) and block size mask (BL) fields.

Table 7-7 describes the bits in the upper and lower BAT registers for 64-bit implementations. Note that the W and G bits are defined for BAT registers that translate data accesses (DBAT registers); attempting to write to the W and G bits in IBAT registers...
causes boundedly-undefined results. The bit definitions for 32-bit implementations are the same except that the bit numbers from Figure 7-11 and Figure 7-12 should be substituted.

Table 7-7. BAT Registers—Field and Bit Descriptions for 64-Bit Implementations

<table>
<thead>
<tr>
<th>Upper/ Lower BAT Register</th>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64 Bit</td>
<td>32 Bit</td>
<td>Block effective page index. This field is compared with high-order bits of the effective address to determine if there is a hit in that BAT array entry.</td>
</tr>
<tr>
<td>Upper BAT Register</td>
<td>0–46</td>
<td>0–14</td>
<td>BEPI</td>
</tr>
<tr>
<td></td>
<td>46–50</td>
<td>15–18</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>51–61</td>
<td>19–29</td>
<td>BL</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>30</td>
<td>Vs</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>31</td>
<td>Vp</td>
</tr>
<tr>
<td>Lower BAT Register</td>
<td>0–46</td>
<td>0–14</td>
<td>BRPN</td>
</tr>
<tr>
<td></td>
<td>47–56</td>
<td>15–24</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>57–60</td>
<td>25–28</td>
<td>WIMG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>G</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 5.2.1, “Memory/Cache Access Attributes.”</td>
</tr>
<tr>
<td></td>
<td>61</td>
<td>29</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>62–63</td>
<td>30–31</td>
<td>PP</td>
</tr>
</tbody>
</table>

The BL field in the upper BAT register is a mask that encodes the size of the block. Table 7-8 defines the bit encodings for the BL field of the upper BAT register.
Only the values shown in Table 7-8 are valid for BL. An effective address is determined to be within a BAT area if the appropriate bits (determined by the BL field) of the effective address match the value in the BEPI field of the upper BAT register, and if the appropriate valid bit (Vs or Vp) is set. Note that for an access to occur, the protection bits (PP bits) in the lower BAT register must be set appropriately, as described in Section 7.4.4, “Block Memory Protection."

The number of zeros in the BL field determines the bits of the effective address that are used in the comparison with the BEPI field to determine if there is a hit in that BAT array entry. The rightmost bit of the BL field is aligned with bit 46 (or bit 14 for 32-bit implementations) of the effective address; bits of the effective address corresponding to ones in the BL field are then cleared to zero for the comparison. For 64-bit implementations operating in 32-bit mode, the highest-order 32 bits of the effective address (EA0–EA31) are treated as zeros.

The value loaded into the BL field determines both the size of the block and the alignment of the block in both effective address space and physical address space. The values loaded into the BEPI and BRPN fields must have at least as many low-order zeros as there are ones in BL. Otherwise, the results are undefined. Also, if the processor does not support 64 bits (or 32 bits, for 32-bit implementations) of physical address, software should write zeros to those unsupported bits in the BRPN field (as the implementation treats them as reserved). Otherwise, a machine check exception can occur.

### 7.4.4 Block Memory Protection

After an effective address is determined to be within a block defined by the BAT array, the access is validated by the memory protection mechanism. If this protection mechanism

**Table 7-8. Upper BAT Register Block Size Mask Encodings**

<table>
<thead>
<tr>
<th>Block Size</th>
<th>BL Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 Kbytes</td>
<td>000 0000 0000</td>
</tr>
<tr>
<td>256 Kbytes</td>
<td>000 0000 0001</td>
</tr>
<tr>
<td>512 Kbytes</td>
<td>000 0000 0011</td>
</tr>
<tr>
<td>1 Mbyte</td>
<td>000 0000 0111</td>
</tr>
<tr>
<td>2 Mbytes</td>
<td>000 0000 1111</td>
</tr>
<tr>
<td>4 Mbytes</td>
<td>000 0001 1111</td>
</tr>
<tr>
<td>8 Mbytes</td>
<td>000 0111 1111</td>
</tr>
<tr>
<td>16 Mbytes</td>
<td>000 1111 1111</td>
</tr>
<tr>
<td>32 Mbytes</td>
<td>001 1111 1111</td>
</tr>
<tr>
<td>64 Mbytes</td>
<td>011 1111 1111</td>
</tr>
<tr>
<td>128 Mbytes</td>
<td>111 1111 1111</td>
</tr>
<tr>
<td>256 Mbytes</td>
<td>111 1111 1111</td>
</tr>
</tbody>
</table>
prohibits the access, a block protection violation exception condition (DSI or ISI exception) is generated.

The memory protection mechanism allows selectively granting read access, granting read/write access, and prohibiting access to areas of memory based on a number of control criteria. The block protection mechanism provides protection at the granularity defined by the block size (128 Kbyte to 256 Mbyte).

As the memory protection mechanism used by the block and page address translation is different, refer to Section 7.5.4, “Page Memory Protection,” for specific information unique to page address translation.

For block address translation, the memory protection mechanism is controlled by the PP bits (which are located in the lower BAT register), which define the access options for the block. Table 7-9 shows the types of accesses that are allowed for the possible PP bit combinations.

<table>
<thead>
<tr>
<th>PP</th>
<th>Accesses Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No access</td>
</tr>
<tr>
<td>x1</td>
<td>Read only</td>
</tr>
<tr>
<td>10</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

Thus, any access attempted (read or write) when PP = 00 results in a protection violation exception condition. When PP = x1, an attempt to perform a write access causes a protection violation exception condition, and when PP = 10, all accesses are allowed. When the memory protection mechanism prohibits a reference, one of the following occurs, depending on the type of access that was attempted:

- For data accesses, a DSI exception is generated and bit 4 of DSISR is set.
- For instruction accesses, an ISI exception is generated and bit 36 of SRR1 (bit 4 in 32-bit implementations) is set.

See Chapter 6, “Exceptions,” for more information about these exceptions.

Table 7-10 shows a summary of the conditions that cause exceptions for supervisor and user read and write accesses within a BAT area. Each BAT array entry is programmed to be either used or ignored for supervisor and user accesses via the BAT array entry valid bits, and the PP bits enforce the read/write protection options. Note that the valid bits (Vs and Vp) are used as part of the match criteria for a BAT array entry and are not explicitly part of the protection mechanism.
Note that because access to the BAT registers is privileged, only supervisor programs can modify the protection and valid bits for the block.

Figure 7-13 expands on the actions taken by the processor in the case of a memory protection violation. Note that the dcbt and dcbtst instructions do not cause exceptions; in the case of a memory protection violation for the attempted execution of one of these instructions, the translation is aborted and the instruction executes as a no-op (no violation is reported). Refer to Chapter 6, “Exceptions,” for a complete description of the SRR1 and DSISR bits settings for the protection violation exceptions.

### Table 7-10. Access Protection Summary for BAT Array

<table>
<thead>
<tr>
<th>Vs</th>
<th>Vp</th>
<th>PP Field</th>
<th>Block Type</th>
<th>User Read</th>
<th>User Write</th>
<th>Supervisor Read</th>
<th>Supervisor Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>No BAT array match</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>User—no access</td>
<td>Exception</td>
<td>Exception</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x1</td>
<td>User-read-only</td>
<td>√</td>
<td>Exception</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>User read/write</td>
<td>√</td>
<td>√</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>Supervisor—no access</td>
<td>Not used</td>
<td>Not used</td>
<td>Exception</td>
<td>Exception</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x1</td>
<td>Supervisor-read-only</td>
<td>Not used</td>
<td>Not used</td>
<td>√</td>
<td>Exception</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>Supervisor read/write</td>
<td>Not used</td>
<td>Not used</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>Both—no access</td>
<td>Exception</td>
<td>Exception</td>
<td>Exception</td>
<td>Exception</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x1</td>
<td>Both-read-only</td>
<td>√</td>
<td>Exception</td>
<td>√</td>
<td>Exception</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>Both read/write</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

**Note:** "Not used" implies that the access is not translated by the BAT array and is translated by the page address translation mechanism described in Section 7.5, “Memory Segment Model,” instead.
7.4.5 Block Physical Address Generation

If the block protection mechanism validates the access, a physical address is formed as shown in Figure 7-14 for 64-bit implementations. Bits in the effective address corresponding to ones in the BL field, concatenated with the 17 lower-order bits of the effective address, form the offset within the block of memory defined by the BAT array entry. Bits in the effective address corresponding to zeros in the BL field are then logically ORed with the corresponding bits in the BRPN field to form the next higher-order bits of the physical address. Finally, the highest-order 36 bits of the BRPN field form bits 0–35 of the physical address (PA0–PA35).

**Figure 7-13. Memory Protection Violation Flow for Blocks**

Note: *Subtract 32 from bit number for bit setting in 32-bit implementations.*
The formation of physical addresses for 32-bit implementations is shown in Figure 7-15. In this case the highest-order four bits of the BRPN field form bits 0–3 of the physical address (PA0–PA3).

Access to the physical memory within the block is made according to the memory/cache access mode defined by the WIMG bits in the lower BAT register. These bits apply to the entire block rather than to an individual page as described in Section 5.2.1, “Memory/Cache Access Attributes.”
7.4.6 Block Address Translation Summary

Figure 7-16 is an expansion of the “BAT Array Hit” branch of Figure 7-4 and shows the translation of address bits for 64-bit implementations. Note that the figure does not show when many of the exceptions in Table 7-4 are detected or taken as this is implementation-specific.
7.5 Memory Segment Model

Memory in the PowerPC OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (80 or 52 bits).

A page address translation may be superseded by a matching block address translation as described in Section 7.4, “Block Address Translation.” If not, the page translation proceeds in the following two steps:

1. from effective address to the virtual address (which never exists as a specific entity but can be considered to be the concatenation of the virtual page number and the byte offset within a page), and
2. from virtual address to physical address.

The page address translation mechanism is described in the following sections, followed by a summary of page address translation with a detailed flow diagram.

7.5.1 Recognition of Addresses in Segments

The page address translation uses segment descriptors, which provide virtual address and protection information, and page table entries (PTEs), which provide the physical address and page protection information. The segment descriptors are programmed by the operating system to provide the virtual ID for a segment. In addition, the operating system
also creates the page table in memory that provides the virtual-to-physical address mappings (in the form of PTEs) for the pages in memory.

Segments in the OEA are defined as one of the following two types:

- Memory segment—An effective address in these segments represents a virtual address that is used to define the physical address of the page.
- Direct-store segment—References made to direct-store segments do not use the virtual paging mechanism of the processor. See Section 7.8, “Direct-Store Segment Address Translation,” for a complete description of the mapping of direct-store segments.

The T bit in the segment descriptor selects between memory segments and direct-store segments, as shown in Table 7-11.

### Table 7-11. Segment Descriptor Types

<table>
<thead>
<tr>
<th>Segment Descriptor T Bit</th>
<th>Segment Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Memory segment</td>
</tr>
<tr>
<td>1</td>
<td>Direct-store segment</td>
</tr>
</tbody>
</table>

#### 7.5.1.1 Selection of Memory Segments

All accesses generated by the processor can be mapped to a segment descriptor, however, if translation is disabled (MSR[IR] = 0 or MSR[DR] = 0 for an instruction or data access, respectively), real addressing mode translation is performed as described in Section 7.3, “Real Addressing Mode.” Otherwise, if T = 0 in the corresponding segment descriptor (and the address is not translated by the BAT mechanism), the access maps to memory space and page address translation is performed.

After a memory segment is selected, the processor creates the virtual address for the segment and searches for the PTE that dictates the physical page number to be used for the access. Note that I/O devices can be easily mapped into memory space and used as memory-mapped I/O.

#### 7.5.1.2 Selection of Direct-Store Segments

As described for memory segments, all accesses generated by the processor (with translation enabled) map to a segment descriptor. If T = 1 for the selected segment descriptor, the access maps to the direct-store interface space and the access proceeds as described in Section 7.8, “Direct-Store Segment Address Translation.” Because the direct-store interface is present only for compatibility with existing I/O devices that used this interface and because the direct-store interface protocol is not optimized for performance, its use is discouraged. The most efficient method for accessing I/O is by mapping the I/O areas to memory segments.
7.5.2 Page Address Translation Overview

The first step in page address translation for 64-bit implementations is the conversion of the 64-bit effective address of an access into the 80-bit virtual address. The virtual address is then used to locate the PTE in the page table in memory. The physical page number is then extracted from the PTE and used in the formation of the physical address of the access. Note that for increased performance, some processors may implement on-chip TLBs to store copies of recently-used PTEs.

Figure 7-17 shows an overview of the translation of an effective address to a physical address for 64-bit implementations as follows:

- Bits 0–35 of the effective address comprise the effective segment ID used to select a segment descriptor, from which the virtual segment ID (VSID) is extracted.
- Bits 36–51 of the effective address correspond to the page number within the segment; these are concatenated with the VSID from the segment descriptor to form the virtual page number (VPN). The VPN is used to search for the PTE in either an on-chip TLB or the page table. The PTE then provides the physical page number (RPN). Note that bits 36–40 form the abbreviated page index (API) which is used to compare with page table entries during hashing. This is described in detail in Section 7.6.1.7.1, “PTEG Address Mapping Example—64-Bit Implementation.”
- Bits 52–63 of the effective address are the byte offset within the page; these are concatenated with the RPN field of a PTE to form the physical address used to access memory.

**Temporary 64-Bit Bridge**

Because processors that implement the 64-bit bridge access only a 32-bit address space, only 16 STEs are required to define the entire 4-Gbyte address space. Page address translation for 64-bit processors using the 64-bit bridge uses a subset of the functionality described here for 64-bit implementations. For example, only bits 32–35 are used to select a segment descriptor, and as in the 32-bit portion of the architecture, only 16 on-chip segment registers are required. These segment descriptors are maintained in 16 SLB entries.

For details concerning the 64-bit bridge, see Section 7.9, “Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations.”
The translation of effective addresses to physical addresses for 32-bit implementations is shown in Figure 7-18, and is similar to that for 64-bit implementations, except that 32-bit implementations index into an array of 16 on-chip segment registers instead of segment tables in memory to locate the segment descriptor, and the address ranges are obviously different, as shown in Figure 7-18. Thus, the address translation is as follows:

- Bits 0–3 of the effective address comprise the segment register number used to select a segment descriptor, from which the virtual segment ID (VSID) is extracted.
- Bits 4–19 of the effective address correspond to the page number within the segment; these are concatenated with the VSID from the segment descriptor to form the virtual page number (VPN). The VPN is used to search for the PTE in either an on-chip TLB or the page table. The PTE then provides the physical page number (RPN).
- Bits 20–31 of the effective address are the byte offset within the page; these are concatenated with the RPN field of a PTE to form the physical address used to access memory.
7.5.2.1 Segment Descriptor Definitions

The format of the segment descriptors is different for 64-bit and 32-bit implementations. Additionally, the fields in the segment descriptors are interpreted differently depending on the value of the T bit within the descriptor. When T = 1, the segment descriptor defines a direct-store segment, and the format is as described in Section 7.8.1, “Segment Descriptors for Direct-Store Segments.”

**TEMPORARY 64-BIT BRIDGE**

For 64-bit processors using the 64-bit bridge, as is the case for 32-bit processors, only 16 segment descriptors are required, each defining 256-Mbyte segments (assuming T = 0). Although the 64-bit bridge implements 16 on-chip segment descriptors, it retains the same STE format used by 64-bit processors although values stored in the STEs reflect the smaller address space. The format for the segment descriptor used by 64-bit processors is described in Section 7.5.2.1.1, “STE Format—64-Bit Implementations.”

7.5.2.1.1 STE Format—64-Bit Implementations

In 64-bit implementations, the segment descriptors reside as segment table entries (STEs) in hashed segment tables in memory. These STEs are generated and placed in segment tables in memory by the operating system using the hashing algorithm described in
Section 7.7.1.2, “Segment Table Hashing Functions.” Each STE is a 128-bit entity (two double words) that maps one effective segment ID to one virtual segment ID. Information in the STE controls the segment table search process and provides input to the memory protection mechanism. Figure 7-19 shows the format of both double words that comprise a T = 0 segment descriptor (or STE) in a 64-bit implementation.

Table 7-12 lists the bit definitions for each double word in an STE.

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–35</td>
<td>ESID</td>
<td>Effective segment ID</td>
</tr>
<tr>
<td>36–55</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>V</td>
<td>Entry valid (V = 1) or invalid (V = 0)</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>T</td>
<td>T = 0 selects this format</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Ks</td>
<td>Supervisor-state protection key</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>Kp</td>
<td>User-state protection key</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>N</td>
<td>No-execute protection bit</td>
<td></td>
</tr>
<tr>
<td>61–63</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0–51</td>
<td>VSID</td>
<td>Virtual segment ID</td>
</tr>
<tr>
<td>52–63</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

The Ks and Kp bits partially define the access protection for the pages within the segment. The page protection provided in the PowerPC OEA is described in Section 7.5.4, “Page Memory Protection.” The virtual segment ID field is used as the high-order bits of the virtual page number (VPN) as shown in Figure 7-17.

The segment descriptors are programmed by the operating system and placed into segment tables in memory, although some processors may additionally have on-chip segment lookaside buffers (SLBs). These SLBs store copies of recently-used STEs that can be accessed quickly, providing increased overall performance. A complete description of the
structure of the segment tables is provided in Section 7.7, “Hashed Segment Tables—64-Bit Implementations.” The PowerPC OEA has defined specific instructions for controlling SLBs (if they are implemented). See Chapter 8, “Instruction Set,” for more detail on the encodings of these instructions.

**TEMPORARY 64-BIT BRIDGE**

Note that processors using the 64-bit bridge implement STEs as defined for 64-bit implementations as described in this section, however from a software perspective the function of these segment descriptors is indistinguishable from the segment registers as they are defined for 32-bit implementations. However, the values in the STEs reflect only a 32-bit address space. For example, the ESID field uses only four bits (ESID[32–35]), which, like the four highest-order bits in a 32-bit effective address, provide an index to one of the 16 segment descriptors.

### 7.5.2.1.2 Segment Descriptor Format—32-Bit Implementations

In 32-bit implementations, the segment descriptors are 32-bits long and reside in one of 16 on-chip segment registers. Figure 7-20 shows the format of a segment register used in page address translation (T = 0) in a 32-bit implementation.

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T = 0 selects this format</td>
</tr>
<tr>
<td>Ks</td>
<td>Supervisor-state protection key</td>
</tr>
<tr>
<td>Kp</td>
<td>User-state protection key</td>
</tr>
<tr>
<td>N</td>
<td>No-execute protection bit</td>
</tr>
<tr>
<td>8–31 VSID</td>
<td>Virtual segment ID</td>
</tr>
</tbody>
</table>

Table 7-13 provides the corresponding bit definitions of the segment register in 32-bit implementations.

---

Table 7-13. Segment Register Bit Definition for Page Address Translation—32-Bit Implementations
The Ks and Kp bits partially define the access protection for the pages within the segment. The page protection provided in the PowerPC OEA is described in Section 7.5.4, “Page Memory Protection.” The virtual segment ID field is used as the high-order bits of the virtual page number (VPN) as shown in Figure 7-18.

The segment registers are programmed with specific instructions that reference the segment registers. However, since the segment registers described here are merely a conceptual model, a processor may implement separate segment registers for instructions and for data, for example. In this case, it is the responsibility of the hardware to maintain the consistency between the multiple sets of segment registers.

The segment register instructions are summarized in Table 7-14. These instructions are privileged in that they are executable only while operating in supervisor mode. See Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for information about the synchronization requirements when modifying the segment registers. See Chapter 8, “Instruction Set,” for more detail on the encodings of these instructions.

### Table 7-14. Segment Register Instructions—32-Bit Implementations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mtsr</code> SR,rS</td>
<td>Move to Segment Register SR(SR)← rS</td>
</tr>
<tr>
<td><code>mtsrin</code> rS,rB</td>
<td>Move to Segment Register Indirect SR[rB[0–3]]← rS</td>
</tr>
<tr>
<td><code>mfsr</code> rD,SR</td>
<td>Move from Segment Register rD← SR[SR]</td>
</tr>
<tr>
<td><code>mfsrin</code> rD,rB</td>
<td>Move from Segment Register Indirect rD← SR[rB[0–3]]</td>
</tr>
</tbody>
</table>

Note: These instructions apply only to 32-bit implementations.

**TEMPORARY 64-BIT BRIDGE**

Note that segment registers and the instructions listed in Table 7-14 are intended for use in 32-bit implementations. In 64-bit implementations, these instructions are legal only in processors that support the 64-bit bridge architecture described in Section 7.9, “Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations.” However, if these features are not supported, attempting to execute these instructions on a 64-bit implementation causes an illegal instruction program exception.

### 7.5.2.2 Page Table Entry (PTE) Definitions

Page table entries (PTEs) are generated and placed in page table in memory by the operating system using the hashing algorithm described in Section 7.6.1.3, “Page Table Hashing Functions.” The PowerPC OEA defines similar PTE formats for both 64- and 32-bit implementations in that the same fields are defined. However, 64-bit implementations
define PTEs that are 128 bits in length while 32-bit implementations define PTEs that are 64 bits in length. Additionally, care must be taken when programming for both 64- and 32-bit implementations, as the bit placements of some fields are different. Some of the fields are defined as follows:

- The virtual segment ID field corresponds to the high-order bits of the virtual page number (VPN), and, along with the H, V, and API fields, it is used to locate the PTE (used as match criteria in comparing the PTE with the segment information).
- The R and C bits maintain history information for the page as described in Section 7.5.3, “Page History Recording.”
- The WIMG bits define the memory/cache control mode for accesses to the page.
- The PP bits define the remaining access protection constraints for the page. The page protection provided by PowerPC processors is described in Section 7.5.4, “Page Memory Protection.”

Conceptually, the page table in memory must be searched to translate the address of every reference. For performance reasons, however, some processors use on-chip TLBs to cache copies of recently-used PTEs so that the table search time is eliminated for most accesses. In this case, the TLB is searched for the address translation first. If a copy of the PTE is found, then no page table search is performed. As TLBs are noncoherent caches of PTEs, software that changes the page table in any way must perform the appropriate TLB invalidate operations to keep the on-chip TLBs coherent with respect to the page table in memory.

7.5.2.2.1 PTE Format for 64-Bit Implementations

In 64-bit implementations, each PTE is a 128-bit entity (two double words) that maps a virtual page number (VPN) to a physical page number (RPN). Information in the PTE is used in the page table search process (to determine a page table hit) and provides input to the memory protection mechanism. Figure 7-21 shows the format of the two double words that comprise a PTE for 64-bit implementations.

![Figure 7-21. Page Table Entry Format—64-Bit Implementations](image-url)
Table 7-15 lists the corresponding bit definitions for each double word in a PTE as defined above.

### Table 7-15. PTE Bit Definitions—64-Bit Implementations

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–51</td>
<td>VSID</td>
<td>Virtual segment ID—corresponds to the high-order bits of the virtual page number (VPN)</td>
</tr>
<tr>
<td></td>
<td>52–56</td>
<td>API</td>
<td>Abbreviated page index</td>
</tr>
<tr>
<td></td>
<td>57–61</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>H</td>
<td>Hash function identifier</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>V</td>
<td>Entry valid (V = 1) or invalid (V = 0)</td>
</tr>
<tr>
<td>1</td>
<td>0–51</td>
<td>RPN</td>
<td>Physical page number</td>
</tr>
<tr>
<td></td>
<td>52–54</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>R</td>
<td>Referenced bit</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>C</td>
<td>Changed bit</td>
</tr>
<tr>
<td></td>
<td>57–60</td>
<td>WIMG</td>
<td>Memory/cache access control bits</td>
</tr>
<tr>
<td></td>
<td>61</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>62–63</td>
<td>PP</td>
<td>Page protection bits</td>
</tr>
</tbody>
</table>

The PTE contains an abbreviated page index rather than the complete page index field because at least 11 of the low-order bits of the page index are used in the hash function to select a PTE group (PTEG) address (PTEG addresses define the location of a PTE). Therefore, these 11 lower-order bits are not repeated in the PTEs of that PTEG.

#### 7.5.2.2.2 PTE Format for 32-Bit Implementations

Figure 7-22 shows the format of the two words that comprise a PTE for 32-bit implementations.

![Figure 7-22. Page Table Entry Format—32-Bit Implementations](image-url)

---

7-50 PowerPC Microprocessor Family: The Programming Environments, Rev 0.1
Table 7-16 lists the corresponding bit definitions for each word in a PTE as defined above.

### Table 7-16. PTE Bit Definitions—32-Bit Implementations

<table>
<thead>
<tr>
<th>Word</th>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>V</td>
<td>Entry valid (V = 1) or invalid (V = 0)</td>
</tr>
<tr>
<td></td>
<td>1–24</td>
<td>VSID</td>
<td>Virtual segment ID</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>H</td>
<td>Hash function identifier</td>
</tr>
<tr>
<td></td>
<td>26–31</td>
<td>API</td>
<td>Abbreviated page index</td>
</tr>
<tr>
<td>1</td>
<td>0–19</td>
<td>RPN</td>
<td>Physical page number</td>
</tr>
<tr>
<td></td>
<td>20–22</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>R</td>
<td>Referenced bit</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>C</td>
<td>Changed bit</td>
</tr>
<tr>
<td></td>
<td>25–28</td>
<td>WIMG</td>
<td>Memory/cache control bits</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>30–31</td>
<td>PP</td>
<td>Page protection bits</td>
</tr>
</tbody>
</table>

In this case, the PTE contains an abbreviated page index rather than the complete page index field because at least ten of the low-order bits of the page index are used in the hash function to select a PTEG address (PTEG addresses define the location of a PTE). Therefore, these ten lower-order bits are not repeated in the PTEs of that PTEG.

### 7.5.3 Page History Recording

Referenced (R) and changed (C) bits reside in each PTE to keep history information about the page. The operating system then uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Referenced and changed recording is performed only for accesses made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store (T = 1) segments. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled (MSR[IR] = 1 or MSR[DR] = 1).
In general, the referenced and changed bits are updated to reflect the status of the page based on the access, as shown in Table 7-17.

**Table 7-17. Table Search Operations to Update History Bits**

<table>
<thead>
<tr>
<th>R and C bits</th>
<th>Processor Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Read: Table search operation to update R</td>
</tr>
<tr>
<td></td>
<td>Write: Table search operation to update R and C</td>
</tr>
<tr>
<td>01</td>
<td>Combination doesn’t occur</td>
</tr>
<tr>
<td>10</td>
<td>Read: No special action</td>
</tr>
<tr>
<td></td>
<td>Write: Table search operation to update C</td>
</tr>
<tr>
<td>11</td>
<td>No special action for read or write</td>
</tr>
</tbody>
</table>

In processors that implement a TLB, the processor may perform the R and C bit updates based on the copies of these bits resident in the TLB. For example, the processor may update the C bit based only on the status of the C bit in the TLB entry in the case of a TLB hit (the R bit may be assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose referenced and changed bits were cleared. See Section 7.6.3, “Page Table Updates,” for all of the constraints imposed on the software when updating the referenced and changed bits in the page tables.

The R bit for a page may be set by the execution of the `dcbt` or `dcbtst` instruction to that page. However, neither of these instructions cause the C bit to be set.

The update of the referenced and changed bits is performed by PowerPC processors as if address translation were disabled (real addressing mode address).

### 7.5.3.1 Referenced Bit

The referenced bit for each virtual page is located in the PTE. Every time a page is referenced (by an instruction fetch, or any other read or write access) the referenced bit is set in the page table. The referenced bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, some processors may assume the R bit in the TLB is always set. The processor never automatically clears the referenced bit.

The referenced bit is only a hint to the operating system about the activity of a page. At times, the referenced bit may be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this include the following:

- Fetching of instructions not subsequently executed
- Accesses generated by an `lswx` or `stswx` instruction with a zero length
- Accesses generated by a `stwcx.` or `stdcx.` instruction when no store is performed
- Accesses that cause exceptions and are not completed
7.5.3.2 Changed Bit

The changed bit for each virtual page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the changed bit in the matching TLB entry is checked. If it is already set, the processor does not change the C bit. If the TLB changed bit is 0, it is set and a table search operation is performed to set the C bit in the corresponding PTE in the page table.

Processors cause the changed bit (in both the PTE in the page tables and in the TLB if implemented) to be set only when a store operation is allowed by the page memory protection mechanism and the store is guaranteed to be in the execution path, unless an exception, other than those caused by one of the following occurs:

- System-caused interrupts (system reset, machine check, external, and decrementer interrupts)
- Floating-point enabled exception type program exceptions when the processor is in an imprecise mode
- Floating-point assist exceptions for instructions that cause no other kind of precise exception

Furthermore, the following conditions may cause the C bit to be set:

- The execution of an \texttt{stwcx.} or \texttt{stdcx.} instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of an \texttt{stswx} instruction is allowed by the memory protection mechanism but a store operation is not performed because the specified length is zero.
- The store operation is not performed because an exception occurs before the store is performed.

Note that although the execution of the \texttt{dcbt} and \texttt{dcbtst} instructions may cause the R bit to be set, they never cause the C bit to be set.

7.5.3.3 Scenarios for Referenced and Changed Bit Recording

This section provides a summary of the model (defined by the OEA) used by PowerPC processors that maintain the referenced and changed bits automatically in hardware, in the setting of the R and C bits. In some scenarios, the bits are guaranteed to be set by the processor, in some scenarios, the architecture allows that the bits may be set (not absolutely required), and in some scenarios, the bits are guaranteed to not be set. Note that when the hardware updates the R and C bits in memory, the accesses are performed as a physical memory access, as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required).

In implementations that do not maintain the R and C bits in hardware, software assistance is required. For these processors, the information in this section still applies, except that the
software performing the updates is constrained to the rules described (that is, must set bits shown as guaranteed to be set and must not set bits shown as guaranteed to not be set). Note that this software should be contained in the area of memory reserved for implementation-specific use and should be invisible to the operating system.

Table 7-18 defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an stwx instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions, by the eciwx instruction, and by the cache management instructions that are treated as a load with respect to address translation. Similarly, store operations include those operations generated by store instructions, by the ecowx instruction, and by the cache management instructions that are treated as a store with respect to address translation.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Scenario</th>
<th>Causes Setting of R Bit</th>
<th>Causes Setting of C Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No-execute protection violation</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Page protection violation</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>Out-of-order instruction fetch or load operation</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Out-of-order store operation contingent on: a branch, trap, sc or rfid</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>instruction, or a possible exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Out-of-order store operation contingent on an exception, other than a</td>
<td>Maybe¹</td>
<td>Maybe¹</td>
</tr>
<tr>
<td></td>
<td>trap or sc instruction, not occurring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Zero-length load (tswx)</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>Zero-length store (stswx)</td>
<td>Maybe¹</td>
<td>Maybe¹</td>
</tr>
<tr>
<td>8</td>
<td>Store conditional (stwcx, or stdcx) that does not store</td>
<td>Maybe¹</td>
<td>Maybe¹</td>
</tr>
<tr>
<td>9</td>
<td>In-order instruction fetch</td>
<td>Yes²</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>Load instruction or eciwx</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>Store instruction, ecowx, or dcbz instruction</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>icbi, dcbit, dcbtst, dcbst, or dcbl instruction</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>dcbi instruction</td>
<td>Maybe¹</td>
<td>Maybe¹</td>
</tr>
</tbody>
</table>

Notes:
¹ If C is set, R is guaranteed to also be set.
² This includes the case in which the instruction was fetched out of order and R was not set.
7.5.3.4 Synchronization of Referenced and Changed Bit Updates

Although the processor updates the referenced and changed bits in the page tables automatically, these updates are not guaranteed to be immediately visible to the program after executing the load, store, or instruction fetch operation that caused the update. If processor A executes a load or store that causes an R and/or C bit update, the following is guaranteed:

- If processor A subsequently executes a sync instruction, both the updates to the bits in the page table and the load or store operation are guaranteed to be performed with respect to all other processors and mechanisms before the sync instruction completes on processor A.
- Additionally, if processor B executes a tlbie instruction that
  — signals the invalidation to the hardware,
  — invalidates the TLB entry for the access in processor A, and
  — is detected by processor A after processor A has begun the access,
and processor B executes a tlbsync instruction after it executes the tlbie, both the updates to the bits and the original access are guaranteed to be performed with respect to all processors and mechanisms before the tlbsync instruction completes on processor A.

7.5.4 Page Memory Protection

In addition to the no-execute option that can be programmed at the segment descriptor level to prevent instructions from being fetched from a given segment (shown in Figure 7-5), there are a number of other memory protection options that can be programmed at the page level. The page memory protection mechanism allows selectively granting read access, granting read/write access, and prohibiting access to areas of memory based on a number of control criteria.

The memory protection used by the block and page address translation mechanisms is different in that the page address translation protection defines a key bit that, in conjunction with the PP bits, determines whether supervisor and user programs can access a page. For specific information about block address translation, refer to Section 7.4.4, “Block Memory Protection.”

For page address translation, the memory protection mechanism is controlled by the following:

- MSR[PR], which defines the mode of the access as follows:
  — MSR[PR] = 0 corresponds to supervisor mode
  — MSR[PR] = 1 corresponds to user mode
- Ks and Kp, the supervisor and user key bits, which define the key for the page
- The PP bits, which define the access options for the page
The key bits (Ks and Kp) and the PP bits are located as follows for page address translation:

- Ks and Kp are located in the segment descriptor.
- The PP bits are located in the PTE.

The key bits, the PP bits, and the MSR[PR] bit are used as follows:

- When an access is generated, one of the key bits is selected to be the key as follows:
  - For supervisor accesses (MSR[PR] = 0), the Ks bit is used and Kp is ignored
  - For user accesses (MSR[PR] = 1), the Kp bit is used and Ks is ignored
    That is, key = (Kp & MSR[PR]) | (Ks & $\neg$MSR[PR])
- The selected key is used with the PP bits to determine if instruction fetching, load access, or store access is allowed.

Table 7-19 shows the types of accesses that are allowed for the general case (all possible Ks, Kp, and PP bit combinations), assuming that the N bit in the segment descriptor is cleared (the no-execute option is not selected).

<table>
<thead>
<tr>
<th>Key1</th>
<th>PP2</th>
<th>Page Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Read/write</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>Read/write</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Read/write</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>Read only</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>No access</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Read only</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Read/write</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Read only</td>
</tr>
</tbody>
</table>

Notes:
1 Ks or Kp selected by state of MSR[PR]
2 PP protection option bits in PTE

Thus, the conditions that cause a protection violation (not including the no-execute protection option for instruction fetches) are depicted in Table 7-20 and as a flow diagram in Figure 7-25. Any access attempted (read or write) when the key = 1 and PP = 00, causes a protection violation exception condition. When key = 1 and PP = 01, an attempt to perform a write access causes a protection violation exception condition. When PP = 10, all accesses are allowed, and when PP = 11, write accesses always cause an exception. The processor takes either the ISI or the DSI exception (for an instruction or data access, respectively) when there is an attempt to violate the memory protection.
Any combination of the Ks, Kp, and PP bits is allowed. One example is if the Ks and Kp bits are programmed so that the value of the key bit for Table 7-19 directly matches the MSR[PR] bit for the access. In this case, the encoding of Ks = 0 and Kp = 1 is used for the PTE, and the PP bits then enforce the protection options shown in Table 7-21.

<table>
<thead>
<tr>
<th>Key</th>
<th>PP</th>
<th>Prohibited Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>Read/write</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Write</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
<td>None</td>
</tr>
<tr>
<td>x</td>
<td>11</td>
<td>Write</td>
</tr>
</tbody>
</table>

Table 7-20. Exception Conditions for Key and PP Combinations

Table 7-21. Access Protection Encoding of PP Bits for Ks = 0 and Kp = 1

<table>
<thead>
<tr>
<th>PP Field</th>
<th>Option</th>
<th>User Read (Key = 1)</th>
<th>User Write (Key = 1)</th>
<th>Supervisor Read (Key = 0)</th>
<th>Supervisor Write (Key = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Supervisor-only</td>
<td>Violation</td>
<td>Violation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>01</td>
<td>Supervisor-write-only</td>
<td>✓</td>
<td>Violation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td>Both user/supervisor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>11</td>
<td>Both read-only</td>
<td>✓</td>
<td>Violation</td>
<td>✓</td>
<td>Violation</td>
</tr>
</tbody>
</table>

However, if the setting Ks = 1 is used, supervisor accesses are treated as user reads and writes with respect to Table 7-21. Likewise, if the setting Kp = 0 is used, user accesses to the page are treated as supervisor accesses in relation to Table 7-21. Therefore, by modifying one of the key bits (in the segment descriptor), the way the processor interprets accesses (supervisor or user) in a particular segment can easily be changed. Note, however, that only supervisor programs are allowed to modify the key bits for the segment descriptor. For 64-bit implementations, although access to the ASR is privileged, the operating system must protect write accesses to the segment table as well. For 32-bit implementations, access to the segment registers is privileged.

When the memory protection mechanism prohibits a reference, the flow of events is similar to that for a memory protection violation occurring with the block protection mechanism. As shown in Figure 7-23, one of the following occurs depending on the type of access that was attempted:

- For data accesses, a DSI exception is generated and DSISR[4] is set. If the access is a store, DSISR[6] is also set.
• For instruction accesses,
  — an ISI exception is generated and SRR1[36] (SRR1[4] for 32-bit implementations) is set, or
  — an ISI exception is generated and SRR1[35] (SRR1[3] for 32-bit implementations) is set if the segment is designated as “no-execute.”

The only difference between the flow shown in Figure 7-23 and that of the block memory protection violation is the ISI exception that can be caused by an attempt to fetch an instruction from a segment that has been designated as “no-execute” (N bit set in the segment descriptor). See Chapter 6, “Exceptions,” for more information about these exceptions.

**Figure 7-23. Memory Protection Violation Flow for Pages**

If the page protection mechanism prohibits a store operation, the changed bit is not set (in either the TLB or in the page tables in memory); however, a prohibited store access may cause a PTE to be loaded into the TLB and consequently cause the referenced bit to be set in a PTE (both in the TLB and in the page table in memory).

*Note:* Subtract 32 from bit number for bit setting in 32-bit implementations.
7.5.5  Page Address Translation Summary

Figure 7-24 provides the detailed flow for the page address translation mechanism in 64-bit implementations. The figure includes the checking of the N-bit in the segment descriptor and then expands on the “TLB Hit” branch of Figure 7-5. The detailed flow for the “TLB Miss” branch of Figure 7-5 is described in Section 7.6.2, “Page Table Search Operation.” The checking of memory protection violation conditions for page address translation is shown in Figure 7-25. The “Invalidate TLB Entry” box shown in Figure 7-24 is marked as implementation-specific as this level of detail for TLBs (and the existence of TLBs) is not dictated by the architecture. Note that the figure does not show the detection of all exception conditions shown in Table 7-3 and Table 7-4; the flow for many of these exceptions is implementation-specific.

7.6  Hashed Page Tables

If a copy of the PTE corresponding to the VPN for an access is not resident in a TLB (corresponding to a miss in the TLB, provided a TLB is implemented), the processor must search for the PTE in the page tables set up by the operating system in main memory.

The algorithm specified by the architecture for accessing the page tables includes a hashing function on some of the virtual address bits. Thus, the addresses for PTEs are allocated more evenly within the page tables and the hit rate of the page tables is maximized. This algorithm must be synthesized by the operating system for it to correctly place the page table entries in main memory.

If page table search operations are performed automatically by the hardware, they are performed using physical addresses and as if the memory access attribute bit M = 1 (memory coherency enforced in hardware). If the software performs the page table search operations, the accesses must be performed in real addressing mode (MSR[DR] = 0); this additionally guarantees that M = 1.

This section describes the format of the page tables and the algorithm used to access them. In addition, the constraints imposed on the software in updating the page tables (and other MMU resources) are described.
Figure 7-24. Page Address Translation Flow for 64-Bit Implementations—TLB Hit

Note: Implementation Specific
7.6.1 Page Table Definition
The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of 2, and its starting address is a multiple of its size.

The page table contains a number of page table entry groups (PTEGs). For 64-bit implementations, a PTEG contains eight page table entries (PTEs) of 16 bytes each; therefore, each PTEG is 128 bytes long. For 32-bit implementations, a PTEG contains eight PTEs of eight bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations. Figure 7-26 shows two PTEG addresses (PTEGaddr1 and PTEGaddr2) where a given PTE may reside.
A given PTE can reside in one of two possible PTEGS—one is the primary PTEG and the other is the secondary PTEG. Additionally, a given PTE can reside in any of the PTE locations within an addressed PTEG. Thus, a given PTE may reside in one of 16 possible locations within the page table. If a given PTE is not in either the primary or secondary PTEG, a page table miss occurs, corresponding to a page fault condition.

A table search operation is defined as the search for a PTE within a primary and secondary PTEG. When a table search operation commences, a primary hashing function is performed on the virtual address. The output of the hashing function is then concatenated with bits programmed into the SDR1 register by the operating system to create the physical address of the primary PTEG. The PTEs in the PTEG are then checked, one by one, to see if there is a hit within the PTEG. If the PTE is not located, a secondary hashing function is performed, a new physical address is generated for the PTEG, and the PTE is searched for again, using the secondary PTEG address.

Note, however, that although a given PTE may reside in one of 16 possible locations, an address that is a primary PTEG address for some accesses also functions as a secondary PTEG address for a second set of accesses (as defined by the secondary hashing function). Therefore, these 16 possible locations are really shared by two different sets of effective addresses. Section 7.6.1.6, “Page Table Structure Examples,” illustrates how PTEs map into the 16 possible locations as primary and secondary PTEs.

Figure 7-26. Page Table Definitions
7.6.1.1 SDR1 Register Definitions
The SDR1 register contains the control information for the page table structure in that it defines the high-order bits for the physical base address of the page table and it defines the size of the table. The format of the SDR1 register differs for 64-bit and 32-bit implementations, as shown below.

7.6.1.1.1 SDR1 Register Definition for 64-Bit Implementations
The format of the SDR1 register for a 64-bit implementation is shown in Figure 7-27.

![Figure 7-27. SDR1 Register Format—64-Bit Implementations](image)

The bit settings for SDR1 are described in Table 7-22.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–45</td>
<td>HTABORG</td>
<td>Physical base address of page table</td>
</tr>
<tr>
<td>46–58</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>59–63</td>
<td>HTABSIZE</td>
<td>Encoded size of page table (used to generate mask)</td>
</tr>
</tbody>
</table>

The HTABORG field in SDR1 contains the high-order 46 bits of the 64-bit physical address of the page table. Therefore, the beginning of the page table lies on a $2^{18}$ byte (256 Kbyte) boundary at a minimum. If the processor does not support 64 bits of physical address, software should write zeros to those unsupported bits in the HTABORG field (as the implementation treats them as reserved). Otherwise, a machine check exception can occur.

A page table can be any size $2^n$ bytes where $18 \leq n \leq 46$. The HTABSIZE field in SDR1 contains an integer value that specifies how many bits from the output of the hashing function are used as the page table index. HTABSIZE is used to generate a mask of the form $0b00...011...1$ (a string of $n$ 0 bits (where $n$ is $28 – \text{HTABSIZE}$) followed by a string of 1 bits, the number of which is equal to the value of HTABSIZE). As the table size increases, more bits are used from the output of the hashing function to index into the table. The 1 bits in the mask determine how many additional bits (beyond the minimum of 11) from the hash are used in the index; the HTABORG field must have this same number of low-order bits equal to 0.
7.6.1.1.2 SDR1 Register Definition for 32-Bit Implementations

The format of SDR1 for 32-bit implementations is similar to that of 64-bit implementations except that the register size is 32 bits and the HTABMASK field is programmed explicitly into SDR1. Additionally, the address ranges correspond to a 32-bit physical address and the range of page table sizes is smaller. Figure 7-28 shows the format of the SDR1 register for 32-bit implementations; the bit settings are described in Table 7-23.

![Figure 7-28. SDR1 Register Format—32-Bit Implementations](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–15</td>
<td>HTABORG</td>
<td>Physical base address of page table</td>
</tr>
<tr>
<td>16–22</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>23–31</td>
<td>HTABMASK</td>
<td>Mask for page table address</td>
</tr>
</tbody>
</table>

The HTABORG field in SDR1 contains the high-order 16 bits of the 32-bit physical address of the page table. Therefore, the beginning of the page table lies on a $2^{16}$ byte (64 Kbyte) boundary at a minimum. As with 64-bit implementations, if the processor does not support 32 bits of physical address, software should write zeros to those unsupported bits in the HTABORG field (as the implementation treats them as reserved). Otherwise, a machine check exception can occur.

A page table can be any size $2^n$ bytes where $16 \leq n \leq 25$. The HTABMASK field in SDR1 contains a mask value that determines how many bits from the output of the hashing function are used as the page table index. This mask must be of the form $0b00...011...1$ (a string of 0 bits followed by a string of 1 bits). As the table size increases, more bits are used from the output of the hashing function to index into the table. The 1 bits in HTABMASK determine how many additional bits (beyond the minimum of 10) from the hash are used in the index; the HTABORG field must have the same number of lower-order bits equal to 0 as the HTABMASK field has lower-order bits equal to 1.

**Example:**

Suppose that the page table is 16,384 ($2^{14}$) 128-byte PTEGs, for a total size of $2^{21}$ bytes (2 Mbytes). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABMASK must be 3 and the value in HTABORG must have its low-order 3 bits (SDR1[31–33]) equal to 0. This means that the page table must begin on a $2 <3 + 11 + 7> = 2^{21} = 2$-Mbyte boundary.
7.6.1.2 Page Table Size

The number of entries in the page table directly affects performance because it influences the hit ratio in the page table and thus the rate of page fault exception conditions. If the table is too small, not all virtual pages that have physical page frames assigned may be mapped via the page table. This can happen if more than 16 entries map to the same primary/secondary pair of PTEGs; in this case, many hash collisions may occur.

7.6.1.2.1 Page Table Sizes for 64-Bit Implementations

In 64-bit implementations, the minimum allowable size for a page table is 256 Kbytes (2^{11} PTEGs of 128 bytes each). However, it is recommended that the total number of PTEGs in the page table be at least half the number of physical page frames to be mapped. While avoidance of hash collisions cannot be guaranteed for any size page table, making the page table larger than the recommended minimum size reduces the frequency of such collisions, by making the primary PTEGs more sparsely populated, and further reducing the need to use the secondary PTEGs.

Table 7-24 shows example sizes for total main memory. The recommended minimum page table sizes for these example memory sizes are then outlined, along with their corresponding HTABORG and HTABSIZE settings. Note that systems with less than 16 Mbytes of main memory may be designed with 64-bit implementations, but the minimum amount of memory that can be used for the page tables is 256 Kbytes in these cases.

Table 7-24. Minimum Recommended Page Table Sizes—64-Bit Implementations

<table>
<thead>
<tr>
<th>Total Main Memory</th>
<th>Memory for Page Tables</th>
<th>Number of Mapped Pages (PTEs)</th>
<th>Number of PTEGs</th>
<th>HTABORG (Maskable Bits 18-45)</th>
<th>HTABSIZE (28-Bit Mask)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 Mbytes (2^{24})</td>
<td>256 Kbytes (2^{18})</td>
<td>2^{14}</td>
<td>2^{11}</td>
<td>x . . . xxx</td>
<td>0 0000 (0 . . . 0000)</td>
</tr>
<tr>
<td>32 Mbytes (2^{25})</td>
<td>512 Kbytes (2^{19})</td>
<td>2^{15}</td>
<td>2^{12}</td>
<td>x . . . xxx0</td>
<td>0 0001 (0 . . . 0001)</td>
</tr>
<tr>
<td>64 Mbytes (2^{26})</td>
<td>1 Mbyte (2^{20})</td>
<td>2^{16}</td>
<td>2^{13}</td>
<td>x . . . xx0</td>
<td>0 0010 (0 . . . 0011)</td>
</tr>
<tr>
<td>128 Mbytes (2^{27})</td>
<td>2 Mbytes (2^{21})</td>
<td>2^{17}</td>
<td>2^{14}</td>
<td>x . . . x00</td>
<td>0 0011 (0 . . . 0111)</td>
</tr>
<tr>
<td>256 Mbytes (2^{28})</td>
<td>4 Mbytes (2^{22})</td>
<td>2^{18}</td>
<td>2^{15}</td>
<td>x . . . x000</td>
<td>0 0100 (0 . . . 1111)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As an example, if the physical memory size is $2^{31}$ bytes (2 Gbyte), there are $2^{31} - 2^{12}$ (4 Kbyte page size) = $2^{19}$ (512 Kbyte) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is $2^{18}$ PTEGs, or $2^{25}$ bytes (32 Mbytes) of memory for the page tables.

### 7.6.1.2.2 Page Table Sizes for 32-Bit Implementations

The recommended page table sizes in 32-bit implementations are similar to that of 64-bit implementations, except that the total number of pages mapped for a given page table size is larger, because the PTEs are only 8 bytes (instead of 16 bytes) in length. In a 32-bit implementation, the minimum size for a page table is 64 Kbytes ($2^{10}$ PTEGs of 64 bytes each). However, as with the 64-bit model, it is recommended that the total number of PTEGs in the page table be at least half the number of physical page frames to be mapped. While avoidance of hash collisions cannot be guaranteed for any size page table, making the page table larger than the recommended minimum size reduces the frequency of such collisions by making the primary PTEGs more sparsely populated, and further reducing the need to use the secondary PTEGs.

Table 7-25 shows some example sizes for total main memory in a 32-bit system. The recommended minimum page table size for these example memory sizes are then outlined, along with their corresponding HTABORG and HTABMASK settings in SDR1. Note that systems with less than 8 Mbytes of main memory may be designed with 32-bit processors, but the minimum amount of memory that can be used for the page tables in these cases is 64 Kbytes.

**Table 7-25. Minimum Recommended Page Table Sizes—32-Bit Implementations**

<table>
<thead>
<tr>
<th>Total Main Memory</th>
<th>Memory for Page Tables</th>
<th>Number of Mapped Pages (PTEs)</th>
<th>Number of PTEGs</th>
<th>HTABORG (Maskable Bits 7–15)</th>
<th>HTABMASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{31}$ Bytes</td>
<td>$2^{45}$ Bytes</td>
<td>$2^{41}$</td>
<td>$2^{38}$</td>
<td>x 0 . . . 0000</td>
<td>1 1011 (0 1 . . . 1111)</td>
</tr>
<tr>
<td>$2^{32}$ Bytes</td>
<td>$2^{46}$ Bytes</td>
<td>$2^{42}$</td>
<td>$2^{39}$</td>
<td>0 . . . 0000</td>
<td>1 1100 (1 . . . 1111)</td>
</tr>
</tbody>
</table>

As an example, if the physical memory size is $2^{31}$ bytes (2 Gbyte), there are $2^{31} - 2^{12}$ (4 Kbyte page size) = $2^{19}$ (512 Kbyte) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is $2^{18}$ PTEGs, or $2^{25}$ bytes (32 Mbytes) of memory for the page tables.

### 7.6.1.2.2 Page Table Sizes for 32-Bit Implementations

The recommended page table sizes in 32-bit implementations are similar to that of 64-bit implementations, except that the total number of pages mapped for a given page table size is larger, because the PTEs are only 8 bytes (instead of 16 bytes) in length. In a 32-bit implementation, the minimum size for a page table is 64 Kbytes ($2^{10}$ PTEGs of 64 bytes each). However, as with the 64-bit model, it is recommended that the total number of PTEGs in the page table be at least half the number of physical page frames to be mapped. While avoidance of hash collisions cannot be guaranteed for any size page table, making the page table larger than the recommended minimum size reduces the frequency of such collisions by making the primary PTEGs more sparsely populated, and further reducing the need to use the secondary PTEGs.

Table 7-25 shows some example sizes for total main memory in a 32-bit system. The recommended minimum page table size for these example memory sizes are then outlined, along with their corresponding HTABORG and HTABMASK settings in SDR1. Note that systems with less than 8 Mbytes of main memory may be designed with 32-bit processors, but the minimum amount of memory that can be used for the page tables in these cases is 64 Kbytes.

**Table 7-25. Minimum Recommended Page Table Sizes—32-Bit Implementations**

<table>
<thead>
<tr>
<th>Total Main Memory</th>
<th>Memory for Page Tables</th>
<th>Number of Mapped Pages (PTEs)</th>
<th>Number of PTEGs</th>
<th>HTABORG (Maskable Bits 7–15)</th>
<th>HTABMASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Mbytes ($2^{23}$)</td>
<td>64 Kbytes ($2^{16}$)</td>
<td>$2^{13}$</td>
<td>$2^{10}$</td>
<td>x xxxx xxxx</td>
<td>0 0000 0000</td>
</tr>
<tr>
<td>16 Mbytes ($2^{24}$)</td>
<td>128 Kbytes ($2^{17}$)</td>
<td>$2^{14}$</td>
<td>$2^{11}$</td>
<td>x xxxx xxx0</td>
<td>0 0000 0001</td>
</tr>
</tbody>
</table>
As an example, if the physical memory size is $2^{29}$ bytes (512 Mbyte), then there are $2^{29} - 2^{12}$ (4 Kbyte page size) = $2^{17}$ (128 Kbyte) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is $2^{16}$ PTEGs, or $2^{22}$ bytes (4 Mbytes) of memory for the page tables.

### 7.6.1.3 Page Table Hashing Functions

The MMU uses two different hashing functions, a primary and a secondary, in the creation of the physical addresses used in a page table search operation. These hashing functions distribute the PTEs within the page table, in that there are two possible PTEGs where a given PTE can reside. Additionally, there are eight possible PTE locations within a PTEG where a given PTE can reside. If a PTE is not found using the primary hashing function, the secondary hashing function is performed, and the secondary PTEG is searched. Note that these two functions must also be used by the operating system to set up the page tables in memory appropriately.

Typically, the hashing functions provide a high probability that a required PTE is resident in the page table, without requiring the definition of all possible PTEs in main memory. However, if a PTE is not found in the secondary PTEG, a page fault occurs and an exception is taken. Thus, the required PTE can then be placed into either the primary or secondary PTEG by the system software, and on the next TLB miss to this page (in those processors that implement a TLB), the PTE will be found in the page tables (and loaded into an on-chip TLB).

The address of a PTEG is derived from the HTABORG field of the SDR1 register, and the output of the corresponding hashing function (primary hashing function for primary PTEG and secondary hashing function for a secondary PTEG). The value in the HTABSIZE field of SDR1 (HTABMASK field for 32-bit implementations) determines how many of the
higher-order hash value bits are masked and how many are used in the generation of the physical address of the PTEG.

### 7.6.1.3.1 Page Table Hashing Functions—64-Bit Implementations

Figure 7-29 depicts the hashing functions defined by the PowerPC OEA for page tables. The inputs to the primary hashing function are the lower-order 39 bits of the VSID field of the STE (bits 13–51 of the 80-bit virtual address), and the page index field of the effective address (bits 52–67 of the virtual address) concatenated with 23 higher-order bits of zero. The XOR of these two values generates the output of the primary hashing function (hash value 1).

![Diagram of Hashing Functions for Page Tables—64-Bit Implementations](image)

When the secondary hashing function is required, the output of the primary hashing function is complemented with one’s complement arithmetic, to provide hash value 2.

### 7.6.1.3.2 Page Table Hashing Functions—32-Bit Implementations

Figure 7-30 depicts the hashing functions defined by the PowerPC OEA for 32-bit implementations. The inputs to the primary hashing function are the lower-order 19 bits of the VSID field of the selected segment register (bits 5–23 of the 52-bit virtual address), and
the page index field of the effective address (bits 24–39 of the virtual address) concatenated with three zero higher-order bits. The XOR of these two values generates the output of the primary hashing function (hash value 1).

As is the case for 64-bit implementations, when the secondary hashing function is required, the output of the primary hashing function is complemented with one’s complement arithmetic, to provide hash value 2.

**Figure 7-30. Hashing Functions for Page Tables—32-Bit Implementations**

### 7.6.1.4 Page Table Addresses

The following sections illustrate the generation of the addresses used for accessing the hashed page tables for both 64- and 32-bit implementations. As stated earlier, the operating system must synthesize the table search algorithm for setting up the tables.

Two of the elements that define the virtual address (the VSID field of the segment descriptor and the page index field of the effective address) are used as inputs into a hashing function. Depending on whether the primary or secondary PTEG is to be accessed, the processor uses either the primary or secondary hashing function as described in Section 7.6.1.3, “Page Table Hashing Functions.”
7.6.1.4.1 Page Table Address Generation for 64-Bit Implementations

The base address of the page table is defined by the high-order bits of SDR1[HTABORG]. Effectively, bits 18–45 of the PTEG address are derived from the masking of the higher-order bits of the hash value (as defined by SDR1[HTABSIZE]) concatenated with (implemented as an OR function) the high-order bits of SDR1[HTABORG] as defined by HTABSIZE. Bits 46–56 of the PTEG address are the 11 lower-order bits of the hash value, and bits 57–63 of the PTEG address are zero. In the process of searching for a PTE, the processor checks up to eight PTEs located in the primary PTEG and up to eight PTEs located in the secondary PTEG, if required, searching for a match. Figure 7-31 provides a graphical description of the generation of the PTEG addresses for 64-bit implementations.
Figure 7-31. Generation of Addresses for Page Tables—64-Bit Implementations
7.6.1.4.2 Page Table Address Generation for 32-Bit Implementations

For 32-bit implementations, the base address of the page table is defined by the high-order bits of SDR1[HTABORG].

Effectively, bits 7–15 of the PTEG address are derived from the masking of the higher-order bits of the hash value (as defined by SDR1[HTABSIZE]) concatenated with (implemented as an OR function) the high-order bits of SDR1[HTABORG] as defined by HTABSIZE. Bits 16-25 of the PTEG address are the 10 lower-order bits of the hash value, and bits 26–31 of the PTEG address are zero. In the process of searching for a PTE, the processor checks up to eight PTEs located in the primary PTEG and up to eight PTEs located in the secondary PTEG, if required, searching for a match. Figure 7-32 provides a graphical description of the generation of the PTEG addresses for 32-bit implementations.
Figure 7-32. Generation of Addresses for Page Tables—32-Bit Implementations
7.6.1.5 Page Table Structure Summary

In the process of searching for a PTE, the processor interprets the values read from memory as described in Section 7.5.2.2, “Page Table Entry (PTE) Definitions.” The VSID and the abbreviated page index (API) fields of the virtual address of the access are compared to those same fields of the PTEs in memory. In addition, the valid (V) bit and the hashing function (H) bit are also checked. For a hit to occur, the V bit of the PTE in memory must be set. If the fields match and the entry is valid, the PTE is considered a hit if the H bit is set as follows:

- If this is the primary PTEG, H = 0
- If this is the secondary PTEG, H = 1

The physical address of the PTE(s) to be checked is derived as shown in Figure 7-31 and Figure 7-32, and the generated address is the address of a group of eight PTEs (a PTEG). During a table search operation, the processor compares up to 16 PTEs: PTE0–PTE7 of the primary PTEG (defined by the primary hashing function) and PTE0–PTE7 of the secondary PTEG (defined by the secondary hashing function).

If the VSID and API fields do not match (or if V or H are not set appropriately) for any of these PTEs, a page fault occurs and an exception is taken. Thus, if a valid PTE is located in the page tables, the page is considered resident; if no matching (and valid) PTE is found for an access, the page in question is interpreted as nonresident (page fault) and the operating system must load the page into main memory and update the PTE accordingly.

The architecture does not specify the order in which the PTEs are checked. Note that for maximum performance however, PTEs should be allocated by the operating system first beginning with the PTE0 location within the primary PTEG, then PTE1, and so on. If more than eight PTEs are required within the address space that defines a PTEG address, the secondary PTEG can be used (again, allocation of PTE0 of the secondary PTEG first, and so on is recommended). Additionally, it may be desirable to place the PTEs that will require most frequent access at the beginning of a PTEG and reserve the PTEs in the secondary PTEG for the least frequently accessed PTEs.

The architecture also allows for multiple matching entries to be found within a table search operation. Multiple matching PTEs are allowed if they meet the match criteria described above, as well as have identical RPN, WIMG, and PP values, allowing for differences in the R and C bits. In this case, one of the matching PTEs is used and the R and C bits are updated according to this PTE. In the case that multiple PTEs are found that meet the match criteria but differ in the RPN, WIMG or PP fields, the translation is undefined and the resultant R and C bits in the matching entries are also undefined.

Note that multiple matching entries can also differ in the setting of the H bit, but the H bit must be set according to whether the PTE was located in the primary or secondary PTEG, as described above.
7.6.1.6 Page Table Structure Examples

The structure of the page tables is very similar for 64- and 32-bit implementations, except that the physical addresses of the PTEGs are 64 bits and 32 bits long for 64- and 32-bit implementations, respectively. Additionally, the size of a PTE for a 64-bit implementation is twice that of a PTE in a 32-bit implementation. Finally, the width of the fields used to generate the PTEG addresses are different (different number of bits used in hashing functions, etc...), and the way in which the size of the page table is specified in the SDR1 register is slightly different.

7.6.1.6.1 Example Page Table for 64-Bit Implementation

Figure 7-33 shows the structure of an example page table for a 64-bit implementation. The base address of the page table is defined by SDR1[HTABORG] concatenated with 18 zero bits. In this example, the address is identified by bits 0–41 in SDR1[HTABORG]; note that bits 42–45 of HTABORG must be zero because the HTABSIZE field specifies an integer mask size of four, which decodes to four mask bits of ones. The addresses for individual PTEGs within this page table are then defined by bits 42–56 as an offset from bits 0–41 of this base address. Thus, the size of the page table is defined as 0x7FFF (32K) PTEGs.

Two example PTEG addresses are shown in the figure as PTEGaddr1 and PTEGaddr2. Bits 42–56 of each PTEG address in this example page table are derived from the output of the hashing function (bits 57–63 are zero to start with PTE0 of the PTEG). In this example, the “b” bits in PTEGaddr2 are the one’s complement of the “a” bits in PTEGaddr1. The “n” bits are also the one’s complement of the “m” bits, but these four bits are generated from bits 24–27 of the output of the hashing function, logically ORed with bits 42–45 of the HTABORG field (which must be zero). If bits 42–56 of PTEGaddr1 were derived by using the primary hashing function, PTEGaddr2 corresponds to the secondary PTEG.

Note, however, that bits 42–56 in PTEGaddr2 can also be derived from a combination of effective address bits, segment descriptor bits, and the primary hashing function. In this case, then PTEGaddr1 corresponds to the secondary PTEG. Thus, while a PTEG may be considered a primary PTEG for some effective addresses (and segment descriptor bits), it may also correspond to the secondary PTEG for a different effective address (and segment descriptor value).

It is the value of the H bit in each of the individual PTEs that identifies a particular PTE as either primary or secondary (there may be PTEs that correspond to a primary PTEG and PTEs that correspond to a secondary PTEG, all within the same physical PTEG address space). Thus, only the PTEs that have H = 0 are checked for a hit during a primary PTEG search. Likewise, only PTEs with H = 1 are checked in the case of a secondary PTEG search.
Example:
Given: SDR1

```
0000 0000 1111 0000 0001 1000 0000 0000 1010 0110 0000 0000 0000 0100
```

```
$00F0 1800 A600 0000
```

```
PTEGaddr1 =
0 42 56 63
0000 0000 1111 0000 0001 1000 0000 0000 1010 0110 00mm mmaa aaaa aaaa a000 0000
```

```
PTEGaddr2 =
0 42 56 63
0000 0000 1111 0000 0001 1000 0000 0000 1010 0110 00nn nnbb bbbb bbbb b000 0000
```

**Figure 7-33. Example Page Table Structure—64-Bit Implementations**

### 7.6.1.6.2 Example Page Table for 32-Bit Implementation

Figure 7-34 shows the structure of an example page table for a 32-bit implementation. The base address of the page table is defined by SDR1[HTABORG] concatenated with 16 zero bits. In this example, the address is identified by bits 0–13 in SDR1[HTABORG]; note that bits 14 and 15 of HTABORG must be zero because the lower-order two bits of HTABMASK are ones. The addresses for individual PTEGs within this page table are then
defined by bits 14–25 as an offset from bits 0–13 of this base address. Thus, the size of the page table is defined as 4096 PTEGs.

![Diagram of Page Table]

**Figure 7-34. Example Page Table Structure—32-Bit Implementations**

Two example PTEG addresses are shown in the figure as PTEGaddr1 and PTEGaddr2. Bits 14–25 of each PTEG address in this example page table are derived from the output of the hashing function (bits 26–31 are zero to start with PTE0 of the PTEG). In this example, the “b” bits in PTEGaddr2 are the one’s complement of the “a” bits in PTEGaddr1. The “n” bits are also the one’s complement of the “m” bits, but these two bits are generated from bits 7–8 of the output of the hashing function, logically ORed with bits 14–15 of the HTABORG field (which must be zero). If bits 14–25 of PTEGaddr1 were derived by using the primary hashing function, then PTEGaddr2 corresponds to the secondary PTEG.
Note, however, that bits 14–25 in PTEGaddr2 can also be derived from a combination of effective address bits, segment register bits, and the primary hashing function. In this case, then PTEGaddr1 corresponds to the secondary PTEG. Thus, while a PTEG may be considered a primary PTEG for some effective addresses (and segment register bits), it may also correspond to the secondary PTEG for a different effective address (and segment register value).

It is the value of the H bit in each of the individual PTEs that identifies a particular PTE as either primary or secondary (there may be PTEs that correspond to a primary PTEG and PTEs that correspond to a secondary PTEG, all within the same physical PTEG address space). Thus, only the PTEs that have H = 0 are checked for a hit during a primary PTEG search. Likewise, only PTEs with H = 1 are checked in the case of a secondary PTEG search.

7.6.1.7 PTEG Address Mapping Examples

This section contains two examples of an effective address and how its address translation (the PTE) maps into the primary PTEG in physical memory. The examples illustrate how the processor generates PTEG addresses for a table search operation; this is also the algorithm that must be used by the operating system in creating page tables. There is one example for a 64-bit implementation and a second example for a 32-bit implementation.

7.6.1.7.1 PTEG Address Mapping Example—64-Bit Implementation

In the example shown in Figure 7-35, the value in SDR1 defines a page table at address 0x0F05_8400_0F00_0000 that contains $2^{17}$ PTEGs. The highest order 36 bits of the effective address uniquely map to a segment descriptor. The segment descriptor is then located and the contents of the segment descriptor are used along with bits 36–63 of the effective address to create the 80-bit virtual address.

To generate the address of the primary PTEG, bits 13–51, and bits 52–67 of the virtual address are then used as inputs into the primary hashing function (XOR) to generate hash value 1. The lower-order 17 bits of hash value 1 are then ORed with the higher-order 46 bits of HTABORG (bits 40-45 should be zero) and concatenated with seven lower-order 0 bits, defining the address of the primary PTEG (0x0F05_8400_0F3F_F300).
Figure 7-35. Example Primary PTEG Address Generation—64-Bit Implementation

Figure 7-36 shows the generation of the secondary PTEG address for this example. If the secondary PTEG is required, the secondary hash function is performed and the lower-order 17 bits of hash value 2 are then ORed with the higher-order 46 bits of HTABORG (bits 40–45 should be zero), and concatenated with seven lower-order 0 bits, defining the address of the secondary PTEG (0x0F05_8400_0FC0_0C80).
As described in Figure 7-31, the 11 lower-order bits of the page index field are always used in the generation of a PTEG address (through the hashing function). This is why only the 5-bit abbreviated page index (API) is defined for a PTE (the entire page index field does not need to be checked). For a given effective address, the lower-order 11 bits of the page index (at least) contribute to the PTEG address (both primary and secondary) where the corresponding PTE may reside in memory. Therefore, if the high-order 5 bits (the API field) of the page index match with the API field of a PTE within the specified PTEG, the PTE mapping is guaranteed to be the unique PTE required.

Figure 7-36. Example Secondary PTEG Address Generation—64-Bit Implementation

Note that a given PTEG address does not map back to a unique effective address. Not only can a given PTEG be considered both a primary and a secondary PTEG (as described in Section 7.6.1.6, “Page Table Structure Examples”), but if the mask defined has four 1 bits or less (not the case shown in the example in the figure), some bits of the page index field of the virtual address are not used to generate the PTEG address. Therefore, any combination of these unused bits will map to the same pair of PTEG addresses. (However, these bits are part of the API and are therefore compared for each PTE within the PTEG to
determine if there is a hit.) Furthermore, an effective address can select a different segment
descriptor with a different value such that the output of the primary (or secondary) hashing
function happens to equal the hash values shown in the example. Thus, these effective
addresses would also map to the same PTEG addresses shown.

7.6.1.7.2 PTEG Address Mapping Example—32-Bit Implementation

Figure 7-37 shows an example of PTEG address generation for a 32-bit implementation. In
the example, the value in SDR1 defines a page table at address 0x0F98_0000 that contains
8192 PTEGs. The example effective address selects segment register 0 (SR0) with the
highest order four bits. The contents of SR0 are then used along with bits 4–31 of the
effective address to create the 52-bit virtual address.

To generate the address of the primary PTEG, bits 5–23, and bits 24–39 of the virtual
address are then used as inputs into the primary hashing function (XOR) to generate hash
value 1. The lower-order 13 bits of hash value 1 are then ORed with the higher-order 16
bits of HTABORG (bits 13–15 should be zero), and concatenated with six lower-order 0
bits, defining the address of the primary PTEG (0x0F9F_F980).
Figure 7-37. Example Primary PTEG Address Generation—32-Bit Implementation

Figure 7-38 shows the generation of the secondary PTEG address for this example. If the secondary PTEG is required, the secondary hash function is performed and the lower-order 13 bits of hash value 2 are then ORed with the higher-order 16 bits of HTABORG (bits 13–15 should be zero), and concatenated with six lower-order 0 bits, defining the address of the secondary PTEG (0x0F98_0640).

As described in Figure 7-32, the 10 lower-order bits of the page index field are always used in the generation of a PTEG address (through the hashing function) for a 32-bit implementation. This is why only the abbreviated page index (API) is defined for a PTE (the entire page index field does not need to be checked). For a given effective address, the lower-order 10 bits of the page index (at least) contribute to the PTEG address (both primary and secondary) where the corresponding PTE may reside in memory. Therefore, if
the higher-order 6 bits (the API field as defined for 32-bit implementations) of the page index match with the API field of a PTE within the specified PTEG, the PTE mapping is guaranteed to be the unique PTE required.

![Diagram](image)

**Figure 7-38. Example Secondary PTEG Address Generation—32-Bit Implementations**

Note that a given PTEG address does not map back to a unique effective address. Not only can a given PTEG be considered both a primary and a secondary PTEG (as described in Section 7.6.1.6, “Page Table Structure Examples”), but in this example, bits 24–26 of the page index field of the virtual address are not used to generate the PTEG address. Therefore, any of the eight combinations of these bits will map to the same primary PTEG address. (However, these bits are part of the API and are therefore compared for each PTE within the PTEG to determine if there is a hit.) Furthermore, an effective address can select a different segment register with a different value such that the output of the primary (or secondary) hashing function happens to equal the hash values shown in the example. Thus, these effective addresses would also map to the same PTEG addresses shown.
7.6.2 Page Table Search Operation

The table search process performed by a PowerPC processor in the search of a PTE varies slightly for 64- and 32-bit implementations. The main differences are the address ranges and PTE formats specified.

7.6.2.1 Page Table Search Operation for 64-Bit Implementations

An outline of the page table search process performed by a 64-bit implementation is as follows:

1. The 64-bit physical addresses of the primary and secondary PTEGs are generated as described in Section 7.6.1.4.1, “Page Table Address Generation for 64-Bit Implementations.”

2. As many as 16 PTEs (from the primary and secondary PTEGs) are read from memory (the architecture does not specify the order of these reads, allowing multiple reads to occur in parallel). PTE reads occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable.

3. The PTEs in the selected PTEGs are tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   - PTE[H] = 0 for primary PTEG; PTE[H] = 1 for secondary PTEG
   - PTE[V] = 1
   - PTE[VSID] = VA[0-51]
   - PTE[API] = VA[52-56]

4. If a match is not found within the eight PTEs of the primary PTEG and the eight PTEs of the secondary PTEG, an exception is generated as described in step 8. If a match (or multiple matches) is found, the table search process continues.

5. If multiple matches are found, all of the following must be true:
   - PTE[RPN] is equal for all matching entries
   - PTE[WIMG] is equal for all matching entries
   - PTE[PP] is equal for all matching entries

6. If one of the fields in step 5 does not match, the translation is undefined, and R and C bit of matching entries are undefined. Otherwise, the R and C bits are updated based on one of the matching entries.

7. A copy of the PTE is written into the on-chip TLB (if implemented) and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if necessary) and the table search is complete.

8. If a match is not found within the primary or secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI or DSI exception).
Reads from memory for page table search operations are performed as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required).

### 7.6.2.2 Page Table Search Operation for 32-Bit Implementations

An outline of the page table search process performed by a 32-bit implementation is as follows:

1. The 32-bit physical addresses of the primary and secondary PTEGs are generated as described in Section 7.6.1.4.2, “Page Table Address Generation for 32-Bit Implementations.”

2. As many as 16 PTEs (from the primary and secondary PTEGs) are read from memory (the architecture does not specify the order of these reads, allowing multiple reads to occur in parallel). PTE reads occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable.

3. The PTEs in the selected PTEGs are tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   - PTE[H] = 0 for primary PTEG; PTE[H] = 1 for secondary PTEG
   - PTE[V] = 1
   - PTE[VSID] = VA[0–23]
   - PTE[API] = VA[24–29]

4. If a match is not found within the eight PTEs of the primary PTEG and the eight PTEs of the secondary PTEG, an exception is generated as described in step 8. If a match (or multiple matches) is found, the table search process continues.

5. If multiple matches are found, all of the following must be true:
   - PTE[RPN] is equal for all matching entries
   - PTE[WIMG] is equal for all matching entries
   - PTE[PP] is equal for all matching entries

6. If one of the fields in step 5 does not match, the translation is undefined, and R and C bit of matching entries are undefined. Otherwise, the R and C bits are updated based on one of the matching entries.

7. A copy of the PTE is written into the on-chip TLB (if implemented) and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if necessary) and the table search is complete.

8. If a match is not found within the primary or secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI or DSI exception).
Reads from memory for page table search operations are performed as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required).

7.6.2.3 Flow for Page Table Search Operation

Figure 7-39 provides a detailed flow diagram of a page table search operation. Note that the references to TLBs are shown as optional because TLBs are not required; if they do exist, the specifics of how they are maintained are implementation-specific. Also, Figure 7-39 shows only a few cases of R-bit and C-bit updates. For a complete list of the R- and C-bit updates dictated by the architecture, refer to Table 7-18.
7.6.3 Page Table Updates

This section describes the requirements on the software when updating page tables in memory via some pseudocode examples. Multiprocessor systems must follow the rules described in this section so that all processors operate with a consistent set of page tables. Even single processor systems must follow certain rules, because software changes must...
be synchronized with the other instructions in execution and with automatic updates that may be made by the hardware (referenced and changed bit updates). Updates to the tables include the following operations:

- Adding a PTE
- Modifying a PTE, including modifying the R and C bits of a PTE
- Deleting a PTE

PTEs must be “locked” on multiprocessor systems. Access to PTEs must be appropriately synchronized by software locking of (that is, guaranteeing exclusive access to) PTEs or PTEGs if more than one processor can modify the table at that time. In the examples below, “lock()” and “unlock()” refer to software locks that must be performed to provide exclusive access to the PTE being updated. See Appendix E, “Synchronization Programming Examples,” for more information about the use of the reservation instructions (such as the \texttt{Iwarx} and \texttt{Stwcx} instructions) to perform software locking.

On single processor systems, PTEs need not be locked. To adapt the examples given below for the single processor case, simply delete the “lock()” and “unlock()” lines from the examples. The \texttt{sync} instructions shown are required even for single processor systems (to ensure that all previous changes to the page tables and all preceding \texttt{tlbie} instructions have completed).

When TLBs are implemented they are defined as noncoherent caches of the page tables. TLB entries must be invalidated explicitly with the TLB invalidate entry instruction (\texttt{tlbie}) whenever the corresponding PTE is modified. In a multiprocessor system, the \texttt{tlbie} instruction must be controlled by software locking, so that the \texttt{tlbie} is issued on only one processor at a time. The \texttt{sync} instruction causes the processor to wait until the TLB invalidate operation in progress by this processor is complete.

The PowerPC OEA defines the \texttt{tlbsync} instruction that ensures that TLB invalidate operations executed by this processor have caused all appropriate actions in other processors. In a system that contains multiple processors, the \texttt{tlbsync} functionality must be used in order to ensure proper synchronization with the other PowerPC processors. Note that a \texttt{sync} instruction must also follow the \texttt{tlbsync} to ensure that the \texttt{tlbsync} has completed execution on this processor.

Any processor, including the processor modifying the page table, may access the page table at any time in an attempt to reload a TLB entry. An inconsistent page table entry must never accidentally become visible; thus, there must be synchronization between modifications to the valid bit and any other modifications (to avoid corrupted data). This requires as many as two \texttt{sync} operations for each PTE update.

Because the V, R, and C bits each reside in a distinct byte of a PTE, programs may update these bits with byte store operations (without requiring any higher-level synchronization). However, extreme care must be taken to ensure that no store over-writes one of these bytes.
accidentally. Processors write referenced and changed bits with unsynchronized, atomic byte store operations.

Explicitly altering certain MSR bits (using the \texttt{mtmsrd} instruction), or explicitly altering STEs, PTEs, or certain system registers, may have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. For example, an \texttt{mtmsrd} instruction may change the value of MSR[SF], changing the effective addresses from which the current instruction stream is being fetched, causing an implicit branch. Implicit branches are not supported and an attempt to perform one causes \textit{boundedly undefined} results. Therefore, PTEs and STEs must not be changed in a manner that causes an implicit branch. Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

The pseudocode sequences that follow assume that a context synchronizing operation has occurred before the sequence is executed (for example, that a DSI exception has occurred). For a complete list of context-synchronizing operations see Table 2-22 and Table 2-23. For a complete list of the synchronization requirements for executing the MMU instructions, see Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers.”

The following examples show the required sequence of operations. However, other instructions may be interleaved within the sequences shown.

7.6.3.1 Adding a Page Table Entry
Adding a page table entry requires only a lock on the PTE in a multiprocessor system. The bytes in the PTE are then written, except for the valid bit. A \texttt{sync} instruction then ensures that the updates have been made to memory, and lastly, the valid bit is set.

\begin{verbatim}
lock(PTE)
PTE[VSID,H,API] \leftarrow \text{new values}
PTE[RPN,R,C,WIMG,PP] \leftarrow \text{new values}
\texttt{sync}
PTE[V] \leftarrow 1
unlock(PTE)
\end{verbatim}

7.6.3.2 Modifying a Page Table Entry
This section describes several scenarios for modifying a PTE.

7.6.3.2.1 General Case
Consider the general case where a currently-valid PTE must be changed. To do this, the PTE must be locked, marked invalid, invalidated from the TLB, updated, marked valid again, and unlocked. The \texttt{sync} instruction must be used at appropriate times to wait for modifications to complete.
Note that the \texttt{tlbsync} and the \texttt{sync} instruction that follows it are only required if software consistency must be maintained with other PowerPC processors in a multiprocessor system (and the software is to be used in a multiprocessor environment).

\begin{verbatim}
lock(PTE)
PTE[V] ← 0
sync
tlbie(PTE)
sync
tlbsync
sync
PTE[VSID,H,API] ← new values
PTE[RPN,R,C,WIMG,PP] ← new values
sync
PTE[V] ← 1
unlock(PTE)
\end{verbatim}

\textbf{7.6.3.2.2 Clearing the Referenced (R) Bit}

When the PTE is modified only to clear the R bit to 0, a much simpler algorithm suffices because the R bit need not be maintained exactly.

\begin{verbatim}
lock(PTE)
oldR ← PTE[R]
PTE[R] ← 0
if oldR = 1, then tlbie(PTE)
unlock(PTE)
\end{verbatim}

Since only the R and C bits are modified by the processor, and since they reside in different bytes, the R bit can be cleared by reading the current contents of the byte in the PTE containing R (bits 48–55 of the second double word, or bits 16–23 of the second word for 64- and 32-bit implementations, respectively), ANDing the value with 0xFE, and storing the byte back into the PTE.

\textbf{7.6.3.2.3 Modifying the Virtual Address}

If the virtual address is being changed to a different address within the same hash class (primary or secondary), the following flow suffices:

\begin{verbatim}
lock(PTE)
val ← PTE[VSID,API,H,V]
val[VSID] ← new VSID
PTE[VSID,API,H,V] ← val
sync
tlbie(PTE)
sync
tlbsync
\end{verbatim}
In this pseudocode flow, note that the store into the first double word (for 64-bit implementations) of the PTE is performed atomically. Also, the `tlbsync` and the `sync` instruction that follows it are only required if consistency must be maintained with other PowerPC processors in a multiprocessor system (and the software is to be used in a multiprocessor environment).

In this example, if the new address is not a cache synonym (alias) of the old address, care must be taken to also flush (or invalidate) from an on-chip cache any cache synonyms for the page. Thus, a temporary virtual address that is a cache synonym with the page whose PTE is being modified can be assigned and then used for the cache flushing (or invalidation).

### 7.6.3.3 Deleting a Page Table Entry

In this example, the entry is locked, marked invalid, invalidated in the TLB, and unlocked. Again, note that the `tlbsync` and the `sync` instruction that follows it are only required if consistency must be maintained with other PowerPC processors in a multiprocessor system (and the software is to be used in a multiprocessor environment).

```plaintext
lock(PTE)
PTE[V] ← 0
sync
tlbsync
sync
tlbsync
unlock(PTE)
```

### 7.6.4 ASR and Segment Register Updates

There are certain synchronization requirements for writing to the ASR or using the move to segment register instructions. These are described in Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers.”

### 7.7 Hashed Segment Tables—64-Bit Implementations

Throughout this chapter, the segment information for an access in a 64-bit implementation has been referenced as residing in a segment descriptor. Whereas the segment descriptors reside in on-chip registers for 32-bit implementations, the segment descriptors for 64-bit implementations reside as segment table entries (STEs) in a hashed segment table in memory, analogous to the hashed page tables for PTEs. Also, similar to the optional storing of recently-used PTEs on-chip in a TLB, copies of STEs may optionally be stored in one
or more on-chip segment lookaside buffers (SLBs), for quicker access. Additionally, the hardware may optionally provide dedicated hardware to search the segment table for an STE automatically, or the processor may vector to an exception routine so that the segment table can be searched by the exception handler software when an STE is required. Note that the algorithm for a segment table search operation must be synthesized by the operating system for it to correctly place the STEs in main memory.

If segment table search operations are performed automatically by the hardware, they are performed as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required). Note that the segment table is never updated automatically by the hardware. If the software performs the segment table search operations, the accesses must be performed in real addressing mode (MSR[DR] = 0); this additionally guarantees that M = 1.

This section describes the format of segment tables and the algorithm used to access them. In addition, the constraints imposed on the software in updating the segment tables are described.

**Temporary 64-Bit Bridge**

Because the 64-bit bridge provides access only to 32-bit address space, the entire 4 Gbytes of effective address space can be defined with 16 on-chip segment descriptors, each defining a 256-Mbyte segment.

### 7.7.1 Segment Table Definition

A segment table is a 4-Kbyte (one page) data structure that defines the mapping between effective segments and virtual segments for a process. The segment table must reside on a page boundary. Whereas at any given time the processor can address only the segments that are defined in a particular segment table, many segment tables can exist in memory, and each one can correspond to a unique process. Physical addresses for elements in the active segment table are derived from the value in the address space register (ASR) and some hashed bits of the effective address.

The segment table contains a number of segment table entry groups (STEGs). An STEG contains eight segment table entries (STEs) of 16 bytes each; therefore, each STEG is 128 bytes long. STEG addresses are entry points for segment table search operations. Figure 7-40 shows two STEG addresses (STEGaddr1 and STEGaddr2) where a given STE may reside.
Figure 7-40. Segment Table Definitions

A given STE can reside in one of two possible STEGs. For each STEG address, there is a complementary STEG address—one is the primary STEG and the other is the secondary STEG. Additionally, a given STE can reside in any of the STE locations within an addressed STEG. Thus, a given STE may reside in one of 16 possible locations within the segment table. If a given STE is not resident within either the primary or secondary STEG, a segment table miss occurs, possibly corresponding to a page fault condition.

A segment table search operation is defined as the search for an STE within a primary and secondary STEG. When a segment table search operation commences, the primary and secondary hashing functions are performed on the effective address. The output of the hashing functions are then concatenated with bits programmed into the ASR by the operating system to create the physical addresses of the primary and secondary STEGs. The STEs in the STEGs are then checked to see if there is a hit within one of the STEGs.

Note, however, that although a given STE may reside in one of 16 possible locations, an address that is a primary STEG address for some accesses also functions as a secondary STEG address for a second set of accesses (as defined by the secondary hashing function). Therefore, these 16 possible locations are really shared by two different sets of effective addresses. Section 7.7.1.5, “Segment Table Structure (with Examples),” illustrates how STEs map into the 16 possible locations as primary and secondary STEs.
7.7.1.1 Address Space Register (ASR)

The ASR contains the control information for the segment table structure in that it defines the highest-order bits for the physical base address of the segment table. The format of the ASR is shown in Figure 7-41. The ASR contains bits 0–51 of the 64-bit physical base address of the segment table. Bits 52–56 of the STEG address are derived from the hashing function, and bits 57–63 are zero at the beginning of a segment table search operation to point to the beginning of an STEG. Therefore, the beginning of the segment table lies on a $2^{12}$ byte (4 Kbyte) boundary.

Note that unless all accesses to be performed by the processor can be translated by the BAT mechanism when address translation is enabled (MSR[DR] or MSR[IR] = 1), the ASR must point to a valid segment table. If the processor does not support 64 bits of physical address, software should write zeros to those unsupported bits in the ASR (as the implementation treats them as reserved). Otherwise, a machine check exception can occur.

Additionally, care should be given that segment table addresses not conflict with those that correspond to areas of the exception vector table reserved for implementation-specific purposes, such as 0x0000, 0x1000, and 0x2000. Note that there are certain synchronization requirements for writing to the ASR that are described in Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers.”

![Figure 7-41. ASR Format—64-Bit Implementations Only](image-url)
The STABORG field identifies the 52-bit physical address of the segment table. The remaining bits are reserved.

**TEMPORARY 64-BIT BRIDGE**

The OEA defines an additional, optional bridge to the 64-bit architecture that allows 64-bit implementations to retain certain aspects of the 32-bit architecture that otherwise are not supported, and in some cases not permitted by the 64-bit architecture. In processors that implement this bridge, at least 16 STEs are implemented and are maintained in 16 dedicated SLB entries.

The bridge facilities allow the option of defining bit 63 as ASR[V], the STABORG field valid bit. If this bit is implemented, STABORG is valid only when ASR[V] is set. This bit is optional, but is implemented if any of the following instructions, which are optional to a 64-bit processor, are implemented: mtsr, mtsrin, mfsr, mfsrin, mtsrd, or mtsrdin. If the bit is not implemented it is treated as reserved except that it is assumed to be 1 for address translation.

The following further describes programming considerations that are affected by the ASR[V] bit:

- If ASR[V] is cleared, having the STABORG field refer to a nonexistent memory location does not cause a machine check exception. Also, if ASR[V] is cleared, the segment table in memory is not searched and the result is the same as if the search had failed.

- For a 64-bit operating system that uses the segment register manipulation instructions as if it were running on a 32-bit implementation, if ASR[V] = 0, a segment fault can occur only if the operating system contains a bug that allows the generation of an effective address larger than $2^{32} - 1$ when MSR[SF] = 1 or if the operating system fails to ensure that the first 16 ESIDs are established (that is, that the corresponding SLB entries are valid).

- Note that slbie or slbia can be executed regardless of the setting of ASR[V]; however, the instructions should not be used if ASR[V] is cleared.

If ASR[V] is implemented, the ASR must point to a valid segment table whenever address translation is enabled, the effective address is not covered by BAT translation, and ASR[V] = 1.

**7.7.1.2 Segment Table Hashing Functions**

The MMU uses two different hashing functions, a primary and a secondary, in the creation of the physical addresses used in a segment table search operation. These hashing functions distribute the STEs within the segment table, in that there are two possible STEGs where a given STE can reside. Additionally, there are eight possible STE locations within an STEG where a given STE can reside. If an STE is not found using the primary hashing function, the secondary hashing function is performed, and the secondary STEG is searched. Note
that these two functions must also be used by the operating system to set up the segment tables in memory appropriately.

Typically, the hashing functions provide a high probability that a required STE is resident in the segment table, without requiring the definition of all possible STEs in main memory. However, if an STE is not found in the secondary STEG, an exception is taken. Thus, the required STE can then be placed into either the primary or secondary STEG by the system software, and on the next SLB miss to this segment (in those processors that implement an SLB), the STE will be found.

The address of an STEG is derived from the base address specified in the ASR, and the output of the corresponding hashing function (primary hashing function for primary STEG and secondary hashing function for a secondary STEG).

Figure 7-42 depicts the hashing functions used by the PowerPC OEA for segment tables. The input to the primary hashing function is the lower-order 5 bits of the ESID field of the effective address. This value is also defined as the output of the primary hashing function (hash value 1).

Figure 7-42. Hashing Functions for Segment Tables
When the secondary hashing function is required, the output of the primary hashing function is the one’s complement, to provide hash value 2.

**TEMPORARY 64-BIT BRIDGE**

Note that although processors using the 64-bit bridge implement STEs as defined for 64-bit implementations, the use of the segment table hashing function is not required because only 16 segment descriptors are required to define the entire 32-bit (4 Gbyte) address space. These segment descriptors are defined as STEs and are stored in 16 SLB entries designated for that purpose.

### 7.7.1.3 Segment Table Address Generation

The following sections illustrate the generation of the addresses used for accessing the hashed segment tables. As stated earlier, the operating system must synthesize the segment table search algorithm for setting up the tables.

The base address of the segment table is defined by the higher-order 52 bits of ASR. Bits 52–56 of the STEG address are derived from the hash value. Depending on whether the primary or secondary STEG is to be accessed, the processor uses either the primary or secondary hashing function as described in Section 7.7.1.2, “Segment Table Hashing Functions.” Bits 57–63 of the STEG address are zero. In the process of searching for an STE, the processor first checks STE0 (at the STEG base address). Figure 7-43 provides a graphical description of the generation of the STEG addresses. Note that Figure 7-43 is also an expansion of the virtual address generation shown in Figure 7-17.

In the process of searching for an STE, the processor interprets the values read from memory as described in Section 7.5.2.1.1, “STE Format—64-Bit Implementations.” The entire ESID field of the effective address of the access is compared to the same field of the STEs in memory. In addition, the valid (V) bit is also checked. For a hit to occur, the V bit of the STE in memory must be set. If the ESID field matches and the entry is valid, the STE is considered a hit.

Note that in the case of the segment table, the H bit (defined for PTEs) is not required to distinguish between the primary and secondary STEs. Because the entire ESID field of the access is compared with the entire ESID field of the STE, when there is a hit, the STE should contain the unique mapping of effective to virtual address for the access (provided there are no programming errors).

During a segment table search operation, the processor compares up to 16 STEs: STE0–STE7 of the primary STEG (defined by the primary hashing function) and STE0–STE7 of the secondary STEG (defined by the secondary hashing function). If the ESID field does not match (or if V is not set) for any of these STEs, a page fault exception condition occurs and an exception is taken. Thus, if no matching (and valid) STE is found for an access, the operating system must load the STE into the segment table.

The architecture does not specify the order in which the STEs are checked. Note that for maximum performance, STEs should be allocated by the operating system first beginning...
with the STE0 location within the primary STEG, then STE1, and so on. If more than eight
STEs are required within the address space that defines a STEG address, the secondary
STEG can be used (again, allocation of STE0 of the secondary STEG first, and so on is
recommended). Additionally, it may be desirable to place the STEs that will require most
frequent access at the beginning of a STEG and reserve the STEs in the secondary STEG
for the least frequently accessed STEs.

The architecture also allows for multiple matching STEs to be found within a table search
operation. However, multiple matching STEs must be identical in all fields. Otherwise, the
translation is undefined.
Figure 7-43. Generation of Addresses for Segment Table
7.7.1.4 Segment Table in 32-Bit Mode
As stated earlier, the only effect on the MMU of operating in 32-bit mode (MSR[SF] = 0) is that the upper-order 32 bits of the logical (effective) address are truncated (treated as zero). Thus, only the lower-order four bits of the ESID field of the effective address are used in the address translation. These four bits select one of 16 STEGs in the segment table and correspond to the highest-order four bits of an address that would have been generated by a 32-bit implementation. The 16 STEGs can then be used in a way similar to the 16 segment registers defined for 32-bit implementations.

Temporary 64-bit Bridge
Note that operating systems using features of the 64-bit bridge run in 32-bit mode, and just as is the case for 32-bit mode described in the previous paragraph, only 16 segment descriptors are required. When ASR[V] bit is cleared, the ASR[STABORG], which indicates the starting address of the segment table is considered to be invalid. The 16 segment registers are implemented in 16 SLB entries as required by the 64-bit bridge architecture.

7.7.1.5 Segment Table Structure (with Examples)
This section contains an example of an effective address and how its segment descriptor (the STE) maps into the primary STEG in physical memory. The example illustrates how the processor generates STEG addresses for a segment table search operation; this is also the algorithm that must be used by the operating system in creating the segment tables.

In the example shown in Figure 7-44, the value in ASR defines a segment table at address 0x0000_5C80_42A1_7000 that contains 32 STEGs (all segment tables are defined with a size of 4 Kbytes). The highest-order 36 bits of the effective address are then used to locate the corresponding STE in the segment table. The contents of the STE are then used along with bits 36–63 of the effective address and the 12-bit byte offset to create the 80-bit virtual address.
To locate the primary STEG (in the segment table), EA bits 31–35 are then used as inputs into the primary hashing function (a simple equality function) to generate hash value 1. Hash value 1 is then concatenated with ASR[0–51] and seven lower-order 0 bits, defining the address of the primary STEG (0x0000_5C80_42A1_7480).

Figure 7-45 shows the generation of the secondary STEG address for this example. If the secondary STEG is required, the secondary hash function is performed (one’s complement) and hash value 2 is then concatenated with bits 0–51 of the ASR and seven lower-order 0 bits, defining the address of the secondary STEG (0x0000_5C80_42A1_7B00).
As described earlier, because the entire effective segment ID field of the STE is compared with the effective segment ID field of the effective address, when an STE compare process results in a match (hit) with the effective address, the STE mapping should be the unique STE required (provided there are no programming errors).

Note, however, that a given STEG address does not map back to a unique effective address. Not only can a given STEG be considered both a primary and a secondary STEG, but many of the bits of the effective segment ID in the effective address are not used to generate the STEG address. Therefore, any combination of these unused bits will map to the same pair of STEG addresses.

### 7.7.2 Segment Table Search Operation

The segment table search process performed by a PowerPC processor in the search of an STE is analogous to the page table search algorithm described earlier for PTEs and is as follows:

1. The 64-bit physical addresses of the primary and secondary STEGs are generated as described in Section 7.7.1.3, “Segment Table Address Generation.”

2. As many as 16 STEs (from the primary and secondary STEGs) are read from memory (the architecture does not specify the order of these reads, allowing multiple reads to occur in parallel). STE reads occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable.
3. The STEs in the selected STEGs are tested for a match with the effective segment ID (ESID) of the access. For a match to occur, the following must be true:
   — STE[V] = 1
   — STE[ESID] = EA[0–35]

4. If no match is found within the eight STEs of the primary STEG and the eight STEs of the secondary STEG, an exception is generated as described in step 7. If a match (or multiple matches) is found, the table search process continues.

5. If multiple matches are found, they must be identical in all defined fields. Otherwise, the translation is undefined.

6. If a match is found, the STE is written into the on-chip SLB (if implemented) and the segment table search is complete.

7. If a match is not found within the primary or secondary PTEG, the search fails, and an exception condition (possibly a page fault) occurs (either an ISI or a DSI exception).

Reads from memory for segment table search operations are performed as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required).

Figure 7-46 provides a detailed flow diagram of a segment table search operation. Note that the references to SLBs are shown as optional because SLBs are not required; if they do exist, the specifics of how they are maintained are implementation-specific.
7.7.3 Segment Table Updates

This section describes the requirements on the software when updating segment tables in memory via some pseudocode examples; note that these requirements are very similar to the requirements imposed on the updating of page tables, but do not have the complication of hardware updates to the referenced and changed bits.

Multiprocessor systems must follow the rules described in this section so that all processors operate with a consistent set of segment tables. Even single processor systems must follow certain rules, because software changes must be synchronized with the other instructions in execution. Updates to the tables include the following operations:

- Adding an STE
- Modifying an STE
- Deleting an STE

Figure 7-46. Segment Table Search Flow
STEs must be locked on multiprocessor systems. Access to STEs must be appropriately synchronized by software locking of (that is, guaranteeing exclusive access to) STEs or STEGs if more than one processor can modify the table at that time. In the examples in the following section, lock() and unlock() refer to software locks that must be performed to provide exclusive access to the STE being updated. See Appendix E, “Synchronization Programming Examples,” for more information about the use of the reservation instructions (such as the lwarx and stwcx. instructions) to perform software locking.

On single processor systems, STEs need not be locked. To adapt the examples given below for the single processor case, simply delete the “lock()” and “unlock()” lines from the examples. The sync instructions shown are required even for single processor systems (to ensure that all previous changes to the segment tables have completed).

When SLBs are implemented, they are defined as noncoherent caches of the segment tables. SLB entries must be invalidated explicitly with the SLB invalidate entry instruction (slbie) whenever the corresponding STE is modified. The sync instruction causes the processor to wait until the SLB invalidate operation in progress by this processor is complete.

**Temporary 64-Bit Bridge**

Note that in the 64-bit bridge, 16 SLB entries are used to hold the 16 segment descriptors necessary for defining the 32-bit address space.

Any processor, including the processor modifying the segment table, may access the segment table at any time in an attempt to reload a SLB entry. An inconsistent segment table entry must never accidentally become visible; thus, there must be synchronization between modifications to the valid bit and any other modifications. This requires as many as two sync operations for each STE update.

As is the case with PTEs, STEs must not be changed in a manner that causes an implicit branch. Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed and a complete list of the synchronization requirements for executing the MMU instructions.

The pseudocode sequences that follow assume that a context synchronizing operation has occurred before the sequence is executed (for example, that a DSI exception has occurred). For a complete list of context-synchronizing operations, see Table 2-22 and Table 2-23.

The following examples show the required sequence of operations. However, other instructions may be interleaved within the sequences shown.
7.7.3.1 Adding a Segment Table Entry
Adding a segment table entry requires only a lock on the STE in a multiprocessor system. The bytes in the STE are then written, except for the valid bit. A sync instruction then ensures that the updates have been made to memory, and lastly, the valid bit is set.

```
lock(STE)
if T = 0,
   then
      STE[ESID, T, Ks, Kp, N] ← new values (Note: N bit only for T = 0 segments)
      STE[VSID] ← new value
   else
      STE[ESID, T, Ks, Kp, 0b0] ← new value
      STE[0b1,CNTLR_SPEC] ← new value
sync
STE[V] ← 1
unlock(STE)
```

7.7.3.2 Modifying a Segment Table Entry
To change the contents of a currently-valid STE, the STE must be locked, invalidated from the SLB, updated, marked valid again, and unlocked. The sync instruction must be used at appropriate times to wait for modifications to complete.

```
lock(STE)
STE[V] ← 0
sync
slbie(STE)
sync
if T = 0,
   then
      STE[ESID,T, Ks, Kp, N] ← new values (Note: N bit only for T = 0 segments)
      STE[VSID] ← new value
   else
      STE[ESID, T, Ks, Kp, 0b0] ← new value
      STE[0b1,CNTLR_SPEC] ← new value
sync
STE[V] ← 1
unlock(STE)
```

7.7.3.3 Deleting a Segment Table Entry
In this example, the entry is locked, marked invalid, invalidated in the SLB, and unlocked.

```
lock(STE)
STE[V] ← 0
```
7.8 Direct-Store Segment Address Translation

As described for memory segments, all accesses generated by the processor (with translation enabled) that do not map to a BAT area, map to a segment descriptor. If T = 1 for the selected segment descriptor, the access maps to the direct-store interface, invoking a specific bus protocol for accessing I/O devices. Direct-store segments are provided for POWER compatibility. As the direct-store interface is present only for compatibility with existing I/O devices that used this interface and the direct-store interface protocol is not optimized for performance, its use is discouraged. Applications that require low-latency load/store access to external address space should use memory-mapped I/O, rather than the direct-store interface.

7.8.1 Segment Descriptors for Direct-Store Segments

The format of many of the fields in the segment descriptors depends on the value of the T bit. Figure 7-47 shows the format of segment descriptors (residing as STEs in segment tables) that define direct-store segments for 64-bit implementations (T bit is set).

![Segment Descriptor Format for Direct-Store Segments—64-Bit Implementations](image)
Table 7-26 shows the bit definitions for the segment descriptors when the T bit is set for 64-bit implementations.

**Table 7-26. Segment Descriptor Bit Definitions for Direct-Store Segments—64-Bit Implementations**

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–35</td>
<td>ESID</td>
<td>Effective segment ID</td>
</tr>
<tr>
<td></td>
<td>36–55</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>V</td>
<td>Entry valid (V = 1) or invalid (V = 0)</td>
</tr>
<tr>
<td></td>
<td>57</td>
<td>T</td>
<td>T = 1 selects this format</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>Ks</td>
<td>Supervisor-state protection key</td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>Kp</td>
<td>User-state protection key</td>
</tr>
<tr>
<td></td>
<td>61–63</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>0–24</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>25–31</td>
<td>b1</td>
<td>Bits 2–8 of the BUID</td>
</tr>
<tr>
<td></td>
<td>32–51</td>
<td>CNTLR_SPEC</td>
<td>Controller-specific information</td>
</tr>
<tr>
<td></td>
<td>52–63</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

In 32-bit implementations, the segment descriptors reside in one of 16 on-chip segment registers. Figure 7-48 shows the register format for the segment registers when the T bit is set for 32-bit implementations.

**Figure 7-48. Segment Register Format for Direct-Store Segments—32-Bit Implementations**

<table>
<thead>
<tr>
<th>T</th>
<th>Ks</th>
<th>Kp</th>
<th>BUID</th>
<th>CNTLR_SPEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>11 12 31</td>
</tr>
</tbody>
</table>

Table 7-27 shows the bit definitions for the segment registers when the T bit is set for 32-bit implementations.

**Table 7-27. Segment Register Bit Definitions for Direct-Store Segments**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>T = 1 selects this format.</td>
</tr>
<tr>
<td>1</td>
<td>Ks</td>
<td>Supervisor-state protection key</td>
</tr>
<tr>
<td>2</td>
<td>Kp</td>
<td>User-state protection key</td>
</tr>
<tr>
<td>3–11</td>
<td>BUID</td>
<td>Bus unit ID</td>
</tr>
<tr>
<td>12–31</td>
<td>CNTLR_SPEC</td>
<td>Device specific data for I/O controller</td>
</tr>
</tbody>
</table>
7.8.2 Direct-Store Segment Accesses

When the address translation process determines that the segment descriptor has \( T = 1 \), direct-store segment address translation is selected; no reference is made to the page tables and neither the referenced or changed bits are updated. These accesses are performed as if the WIMG bits were 0b0101; that is, caching is inhibited, the accesses bypass the cache, hardware-enforced coherency is not required, and the accesses are considered guarded.

The specific protocol invoked to perform these accesses involves the transfer of address and data information; however, the PowerPC OEA does not define the exact hardware protocol used for direct-store accesses. Some instructions may cause multiple address/data transactions to occur on the bus. In this case, the address for each transaction is handled individually with respect to the MMU.

7.8.2.1 Direct-Store Transfers for 64-Bit Implementations

The following procedure is provided to describe the data that is typically sent to the memory controller for 64-bit implementations:

- One of the \( K_x \) bits (\( K_s \) or \( K_p \)) is selected to be the key as follows:
  - For supervisor accesses (\( MSR[PR] = 0 \)), the \( K_s \) bit is used and \( K_p \) is ignored.
  - For user accesses (\( MSR[PR] = 1 \)), the \( K_p \) bit is used and \( K_s \) is ignored.
- An implementation-dependent portion of the segment descriptor.
- An implementation-dependent portion of the effective address.

7.8.2.2 Direct-Store Transfers for 32-Bit Implementations

The following procedure is provided to describe the data that is typically sent to the memory controller in 32-bit implementations:

- One of the \( K_x \) bits (\( K_s \) or \( K_p \)) is selected to be the key as follows:
  - For supervisor accesses (\( MSR[PR] = 0 \)), the \( K_s \) bit is used and \( K_p \) is ignored
  - For user accesses (\( MSR[PR] = 1 \)), the \( K_p \) bit is used and \( K_s \) is ignored
- An implementation-dependent portion of the segment descriptor.
- An implementation-dependent portion of the effective address.

7.8.3 Direct-Store Segment Protection

Page-level memory protection as described in Section 7.5.4, “Page Memory Protection,” is not provided for direct-store segments. The appropriate key bit (\( K_s \) or \( K_p \)) from the segment descriptor is sent to the memory controller, and the memory controller implements any protection required. Frequently, no such mechanism is provided; the fact that a direct-store segment is mapped into the address space of a process may be regarded as sufficient authority to access the segment.
7.8.4 Instructions Not Supported in Direct-Store Segments
The following instructions are not supported at all and cause either a DSI exception or boundedly-undefined results when issued with an effective address that selects a segment descriptor that has $T = 1$:

- lwarx and ldarx
- stwcx. and stdcx.
- eciwx
- ecowx

7.8.5 Instructions with No Effect in Direct-Store Segments
The following instructions are executed as no-ops when issued with an effective address that selects a segment where $T = 1$:

- dcbt
- dcbtst
- dcbf
- dbi
- dcbst
- dcbz
- icbi

7.8.6 Direct-Store Segment Translation Summary Flow
Figure 7-49 shows the flow used by the MMU when direct-store segment address translation is selected. This figure expands the Direct-Store Segment Translation stub found in Figure 7-5 for both instruction and data accesses. In the case of a floating-point load or store operation to a direct-store segment, it is implementation-specific whether the alignment exception occurs. In the case of an eciwx, ecowx, lwarx, lda rx, stwcx., or stdcx. instruction, the implementation either sets the DSISR as shown and causes the DSI exception, or causes boundedly-undefined results.
Figure 7-49. Direct-Store Segment Translation Flow

Notes:
*Subtract 32 from bit number for bit setting in 32-bit implementations
Implementation-specific
TEMPORARY 64-BIT BRIDGE

7.9 Migration of Operating Systems from 32-Bit Implementations to 64-Bit Implementations

The facilities and instructions described in this section may optionally be provided by a 64-bit implementation to reduce the amount of software change required to migrate an operating system from a 32-bit implementation to a 64-bit implementation. Using the bridge facility allows the operating system to treat the MSR as a 32-bit register and to continue to use the segment register manipulation instructions (\texttt{mtsr}, \texttt{mtsrin}, \texttt{mfsr}, and \texttt{mfsrin}) which are defined for 32-bit implementations. These instructions are otherwise illegal in the 64-bit architecture. Although the 64-bit bridge does not literally implement the 16 registers as they are defined by the 32-bit portion of the architecture, the segment register manipulation instructions are used to access the 16 predefined segment descriptors stored in the on-chip SLBs.

The bridge features do not conceal the differences in format of the page table, BAT registers, and SDR1 between 32-bit and 64-bit implementations—the operating system must be converted explicitly to use the 64-bit formats. Note that an operating system that uses the bridge features does not take full advantage of the 64-bit implementation (for example, it can generate only 32-bit effective addresses).

An operating system that uses the 64-bit bridge architecture should observe the following:

- The boot process should do the following:
  - Clear MSR[SF] and MSR[ISF].
  - Initialize the ASR, clearing ASR[V].
  - Invalidate all SLB entries.
- The operating system should do the following.
  - Support only 32-bit applications.
  - If any 64-bit instructions are used, for example, to modify a PTE or a 64-bit SPR, ensure either that exceptions cannot occur or that the exception handler saves and restores all 64 bits of the GPRs.
  - Manipulate only the low-order 32 bits of the MSR, leaving the high-order 32 bits unchanged.
  - Always have MSR[ISF] = 0 and ASR[V] = 0.
  - Manage virtual segments using the 32-bit segment register manipulation instructions (\texttt{mtsr}, \texttt{mtsrin}, \texttt{mfsr}, and \texttt{mfsrin}).
  - Always map segments 0–15 in the SLB when translation is enabled. They may be mapped with a VSID for which there are no valid PTEs.
  - Never execute an \texttt{slbie} or \texttt{slbia} instruction.
  - Never generate an effective address greater than $2^{32} - 1$ when MSR[SF] = 1.
7.9.1 ISF Bit of the Machine State Register

MSR[ISF] (bit 2) may optionally be used by a 64-bit implementation to control the mode (64-bit or 32-bit) that is entered when an exception is taken. If MSR[ISF] is implemented, it has the properties described below. If it is not implemented, it is treated as reserved except that ISF is assumed to be set for exception handling.

- When an exception occurs, MSR[ISF] is copied to MSR[SF].
- When an exception occurs, MSR[ISF] is not altered.
- No software synchronization is required before or after altering MSR[ISF] (see Section 2.3.18, “Synchronization Requirements for Special Registers and for Lookaside Buffers.”)

7.9.2 rfi and mtmsr Instructions in a 64-Bit Implementation

The rfi and mtmsr instruction pair may be implemented in some 64-bit implementations, along with the rfid and mtmsrd instructions, which are required by 64-bit implementations. A 64-bit processor must implement either both or neither of these instructions. Attempting to execute either rfi or mtmsr on a 64-bit processor that does not support these instructions causes an illegal instruction type program exception.

Except for the following variances, the operation of these instructions in a 64-bit implementation is identical to their operation in a 32-bit implementation as described in Section 4.4.1, “System Linkage Instructions—OEA (64-Bit Bridge),” and Section 4.4.3.2, “Segment Register Manipulation Instructions.”

- rfi
  - The SRR1 bits that are copied to the corresponding bits of the MSR are bits 32, 37–41, and 48–63. The remaining bits of the MSR, including the high-order 32 bits, are unchanged.
  - If the new MSR value does not enable any pending exceptions, the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–61]||0b00 (when SF is set in the new MSR value) or (32)0||SRR0[32–61]||0b00 (when SF is cleared in the new MSR value).

- mtmsr
  - Bits 32–63 of rS are placed into MSR[32–63]. MSR[0–31] are unchanged.
  - Note that an additional 64-bit–specific instruction for reading the MSR is not needed because the mfmsr instruction copies the entire contents of the MSR to the selected GPR in both 32- and 64-bit implementations.
7.9.3 Segment Register Manipulation Instructions in the 64-Bit Bridge

The four segment register manipulation instructions, **mtsr**, **mtsrin**, **mfsr**, and **mfsrin**, defined as part of the 32-bit portion of the architecture may optionally be provided by a 64-bit implementation that uses the 64-bit bridge. As part of the 64-bit bridge, these instructions operate as described below rather than in the way they are described for 32-bit implementations (as described in Section 4.4.3.2, “Segment Register Manipulation Instructions.”) These instructions are implemented as a group and are not implemented individually. Attempting to execute one of these instructions on a 64-bit processor on which it is not supported causes an illegal instruction type program exception.

These instructions allow software to associate effective segments 0 through 15 with any of virtual segments 0 through $2^{24} - 1$ without altering the segment table in memory. Sixteen indexed SLB entries serve as virtual segment registers. The **mtsr** and **mtsrin** instructions move 32 bits from a selected GPR to a selected SLB entry. The **mfsr** and **mfsrin** instructions move 64 bits from a selected SLB entry to a selected GPR and can be used to read an SLB entry that was created with **mtsr**, **mtsrin**, **mtsrd**, or **mtsrdin**.

The software synchronization requirements for any of the move to segment register instructions in a 64-bit implementation are the same as for those defined by the 32-bit architecture.

To ensure that SLB entries contain unique ESIDs when the bridge is used, an ESID mapped by any of the move to segment register instructions must not have been mapped to that SLB entry by the segment table when ASR[V] was set.

If an SLB entry that software established using one of the move to segment register instructions is overwritten while ASR[V] = 1, software must be able to handle any exception caused when a segment descriptor cannot be located.

Executing an **mfsr** or **mfsrin** instruction may set rD to an undefined value if ASR[V] has been set at any time since execution of the **mtsr**, **mtsrin**, **mtsrd**, or **mtsrdin** instruction that established the selected SLB entry, because that SLB entry may have been overwritten by the processor in the meantime.

Typically, 16 fixed SLB entries are used by the segment register manipulation instructions, while SLB reload from the segment table selects SLB entries based on some other replacement policy such as LRU.

With respect to updating any SLB replacement history used by the SLB replacement policy, implementations will treat the execution of an **mtsr**, **mtsrd**, **mtsrin**, or **mtsrdin** instruction the same as an SLB reload from the segment table.

The following sections describe the move to and move from segment register instructions as they are defined for the 64-bit bridge.
7.9.4 64-Bit Bridge Implementation of Segment Register Instructions
Previously Defined for 32-Bit Implementations Only

The following sections describe the \textbf{mfsr}, \textbf{mfsrin}, \textbf{mtsrr}, and \textbf{mtsrin} instructions that are defined for the 32-bit architecture and are allowed in the 64-bit bridge architecture only if ASR[V] is implemented. Otherwise, attempting to execute one of these instructions is illegal on a 64-bit implementation.

7.9.4.1 Move from Segment Register—\textbf{mfsr}

As in the 32-bit architecture, the \textbf{mfsr} instruction syntax is as follows:

\begin{verbatim}
mfsr \textbf{rD}, SR
\end{verbatim}

The operation of the instruction is described as follows:

\begin{verbatim}
\textbf{rD} \leftarrow \text{SLB}(SR)
\end{verbatim}

When executed as part of the 64-bit bridge, the contents of the SLB entry selected by SR are placed into \textbf{rD}; the contents of \textbf{rD} correspond to a segment table entry containing values as shown in Table 7-29.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Double Word} & \textbf{Bit(s)} & \textbf{Contents} & \textbf{Description} \\
\hline
0 & 0–31 & 0x0000\_0000 & ESID[0–31] \\
 & 36–56 & — & — \\
 & 57–59 & \textbf{rD}[32–34] & T, Ks, Kp \\
 & 60–61 & \textbf{rD}[35–36] & N, reserved bit, or b0 \\
 & 62–63 & — & — \\
1 & 0–24 & \textbf{rD}[7–31] & VSID[0–24] (or reserved if SR[T] = 1) \\
 & 52–63 & — & — \\
\hline
\end{tabular}
\caption{Contents of \textbf{rD} after Executing \textbf{mfsr}}
\end{table}

\textbf{Note:} The contents of \textbf{rD}[0–6] are cleared automatically.

If the SLB entry selected by SR was not created by an \textbf{mtsrr}, \textbf{mtsrc}, or \textbf{mtsrdrin} instruction, the contents of \textbf{rD} are undefined. Formatting for GPR contents is shown in Figure 7-50. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

This is a supervisor-level instruction.
7.9.4.2 Move from Segment Register Indirect—mfsrin

As in the 32-bit architecture, the **mfsrin** instruction syntax is as follows:

\[ \text{mfsrin } r_D, r_B \]

The operation of the instruction is described as follows:

\[ r_D \leftarrow \text{SLB}(r_B[32–35]) \]

The contents of the SLB entry selected by \( r_B[32–35] \) are placed into \( r_D \); the contents of \( r_D \) correspond to a segment table entry containing values as shown in Table 7-29:

### Table 7-29. SLB Entry Following mfsrin

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit(s)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–31</td>
<td>0x0000_0000</td>
<td>ESID[0–31]</td>
</tr>
<tr>
<td></td>
<td>36–56</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>57–59</td>
<td>( r_D[32–34] )</td>
<td>T, Ks, Kp</td>
</tr>
<tr>
<td></td>
<td>60–61</td>
<td>( r_D[35–36] )</td>
<td>N, reserved bit, or b0</td>
</tr>
<tr>
<td>1</td>
<td>0–24</td>
<td>( r_D[7–31] )</td>
<td>VSID[0–24] or reserved</td>
</tr>
<tr>
<td></td>
<td>25–51</td>
<td>( r_D[37–63] )</td>
<td>VSID[25–51], or b1, CNTLR_SPEC</td>
</tr>
<tr>
<td></td>
<td>52–63</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Note:** The contents of \( r_D[0–6] \) are cleared automatically
If the SLB entry selected by \( rB \[32–35] \) was not created by an \texttt{mtsr}, \texttt{mtsrd}, or \texttt{mtsrdin} instruction, the contents of \( rD \) are undefined. Formatting for GPR contents is shown in Figure 7-50. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

This is a supervisor-level instruction.

7.9.4.3 Move to Segment Register—\texttt{mtsr}

As in the 32-bit architecture, the \texttt{mtsr} instruction syntax is as follows:

\[
\texttt{mtsr} \ SR, rS
\]

The operation of the instruction is described as follows:

\[
\text{SLB}(SR) \leftarrow (rS(32–63))
\]

The SLB entry selected by \( SR \) is set as though it were loaded from a segment table entry, as shown in Table 7-30.

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit(s)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–31</td>
<td>0x0000_0000</td>
<td>ESID[0–31]</td>
</tr>
<tr>
<td></td>
<td>32–35</td>
<td>SR</td>
<td>ESID[32–36]</td>
</tr>
<tr>
<td></td>
<td>36–55</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>0b1</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>57–59</td>
<td>( rS[32–34] )</td>
<td>T, Ks, Kp</td>
</tr>
<tr>
<td></td>
<td>60–61</td>
<td>( rS[35–36] )</td>
<td>N, reserved bit, or b0</td>
</tr>
<tr>
<td></td>
<td>62–63</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>0–24</td>
<td>0x0000_00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25–51</td>
<td>( rS[37–63] )</td>
<td>VSID[25–51], or b1, CNTLR_SPEC</td>
</tr>
<tr>
<td></td>
<td>51–63</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

This is a supervisor-level instruction. Formatting for GPR contents is shown in Figure 7-51. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.
Figure 7-51. GPR Contents for mtsr and mtsrin

Note that when creating a memory segment (T = 0) using the mtsr instruction, rS[36–39] should be cleared, as these bits correspond to the reserved bits in the T = 0 format for a segment register.

7.9.4.4 Move to Segment Register Indirect—mtsrin

As in the 32-bit architecture, the mtsrin instruction syntax is as follows:

\[ \text{mtsrin } rS, rB \]

The operation of the instruction is described as follows:

\[ \text{SLB}(rB[32–35]) \leftarrow (rS[32–62]) \]

The SLB entry selected by bits 32–35 of rB is set as though it were loaded from a segment table entry, as shown in Table 7-29.

Table 7-31. SLB Entry Following mtsrin

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit(s)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–31</td>
<td>0x0000_0000</td>
<td>ESID[0–31]</td>
</tr>
<tr>
<td></td>
<td>36–55</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>0b1</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>57–59</td>
<td>rS[32–34]</td>
<td>T, Ks, Kp</td>
</tr>
<tr>
<td></td>
<td>60–61</td>
<td>rS[35–36]</td>
<td>N, reserved bit, or b0</td>
</tr>
<tr>
<td></td>
<td>62–63</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Chapter 7. Memory Management

This is a supervisor-level instruction. Formatting for GPR contents is shown in Figure 7-51. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.

Note that when creating a memory segment (T = 0) using the `mtsrin` instruction, `rS[36–39]` should be cleared, as these bits correspond to the reserved bits in the T = 0 format for a segment register.

### 7.9.5 Segment Register Instructions Defined Exclusively for the 64-Bit Bridge

The following sections describe two instructions `mtsrrd` and `mtsradin`, that are defined for optional use as part of the 64-bit bridge. These instructions support cross-memory operations in a manner similar to that on 32-bit implementations, allowing software to associate effective segments 0–15 (which define the 32-bit address space) with any of virtual segments 0–(2^52 – 1) (which define the 64-bit address space), moving 64 bits from a selected GPR to a selected SLB entry. This allows an operating system to establish addressability to an address space, to copy data to it from another address space, and then to destroy the new addressability, all without altering the segment table in memory.

Note that altering the segment table is slow because of the software synchronization required, as described in Section 7.7.3, “Segment Table Updates.”

If either instruction is provided, both should be. If neither is provided, attempting to execute either causes an illegal instruction type program exception.

Note that because the existing instructions move the entire contents of the selected SLB entry into the selected GPR, additional versions of the move from segment register instructions are not required.

#### 7.9.5.1 Move to Segment Register Double Word—`mtsrrd`

The `mtsrin` instruction syntax is as follows:

`mtsrrd` SR, rS

The operation of the instruction is described as follows:

```
SLB (SR) ← (rS)
```

---

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit(s)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0–24</td>
<td>0x0000_00</td>
<td>VSID[0–24] or reserved</td>
</tr>
<tr>
<td>25–51</td>
<td>rS[37–63]</td>
<td>VSID[25–51], or b1, CNTLR_SPEC</td>
<td></td>
</tr>
<tr>
<td>52–63</td>
<td></td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 7-31. SLB Entry Following mtsrin (Continued)
The contents of $rS$ are placed into the SLB selected by SR. The SLB entry is set as though it were loaded from an STE, as shown in Table 7-29.

### Table 7-32. SLB Entry Following mtsrd

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit(s)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–31</td>
<td>0x0000_0000</td>
<td>ESID[0–31]</td>
</tr>
<tr>
<td></td>
<td>32–35</td>
<td>SR</td>
<td>ESID[32–35]</td>
</tr>
<tr>
<td></td>
<td>36–55</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>0b1</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>57–59</td>
<td>$rS$[32–34]</td>
<td>T, Ks, Kp</td>
</tr>
<tr>
<td></td>
<td>60–61</td>
<td>$rS$[35–36]</td>
<td>N, reserved bit, or b0</td>
</tr>
<tr>
<td></td>
<td>62–63</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>0–24</td>
<td>$rS$[7–31]</td>
<td>VSID[0–24] or reserved</td>
</tr>
<tr>
<td></td>
<td>25–51</td>
<td>$rS$[37–63]</td>
<td>VSID[25–51], or b1, CNTLR_SPEC</td>
</tr>
<tr>
<td></td>
<td>52–63</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

This is a supervisor-level instruction.

This instruction is optional, and defined only for 64-bit implementations. Using it on a 32-bit implementation causes an illegal instruction exception. Formatting for GPR contents is shown in Figure 7-50. Fields shown as zeroes should be cleared. Fields shown as hyphens are ignored.

### 7.9.5.2 Move to Segment Register Double Word Indirect—mtsrdin

The syntax for the **mtsrdin** instruction is as follows:

```
mtsrdin rS, rB
```

The operation of the instruction is described as follows:

```c
SLB(rB[32–35]) ← (rS)
```
The contents of $rS$ are copied to the SLB selected by bits 32–35 of $rB$. The SLB entry is set as though it were loaded from an STE, as shown in Table 7-33.

<table>
<thead>
<tr>
<th>Double Word</th>
<th>Bit(s)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0–31</td>
<td>0x0000_0000</td>
<td>ESID[0–31]</td>
</tr>
<tr>
<td></td>
<td>36–55</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>0b1</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>57–59</td>
<td>$rS[32–34]$</td>
<td>T, Ks, Kp</td>
</tr>
<tr>
<td></td>
<td>60–61</td>
<td>$rS[35–63]$</td>
<td>N, reserved bit, or b0</td>
</tr>
<tr>
<td></td>
<td>62–63</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>0–24</td>
<td>$rS[7–31]$</td>
<td>VSID[0–24] or reserved</td>
</tr>
<tr>
<td></td>
<td>25–51</td>
<td>$rS[37–63]$</td>
<td>VSID[25–51], or b1, CNTLR_SPEC</td>
</tr>
<tr>
<td></td>
<td>52–63</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

This is a supervisor-level instruction.

This instruction is optional, and defined only for 64-bit implementations. Using it on a 32-bit implementation causes an illegal instruction exception. Fields shown as x’s are ignored. Fields shown as slashes correspond to reserved bits in the segment table entry.