Intel Architectural Support for Memory Management

The memory management facilities of the Intel Architecture are divided into two parts: segmentation and paging.

- Segmentation provides a mechanism of isolating individual code, data, and runtime stack memory segments, so that multiple programs (or tasks) can run on the same processor without interfering with one another.
- Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks.

As shown in Figure 1, segmentation provides a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**. Segments can be used to hold the code, data, and stack for a program or to hold system data structures (such as a TSS (task state segment for per process hardware context) or LDT (Local Descriptor Table)). If more than one program (or task) is running on a processor, each program can be assigned its own set of segments. The processor then enforces the boundaries between these segments and insures that one program does not interfere with the execution of another program by writing into the other program's segments.

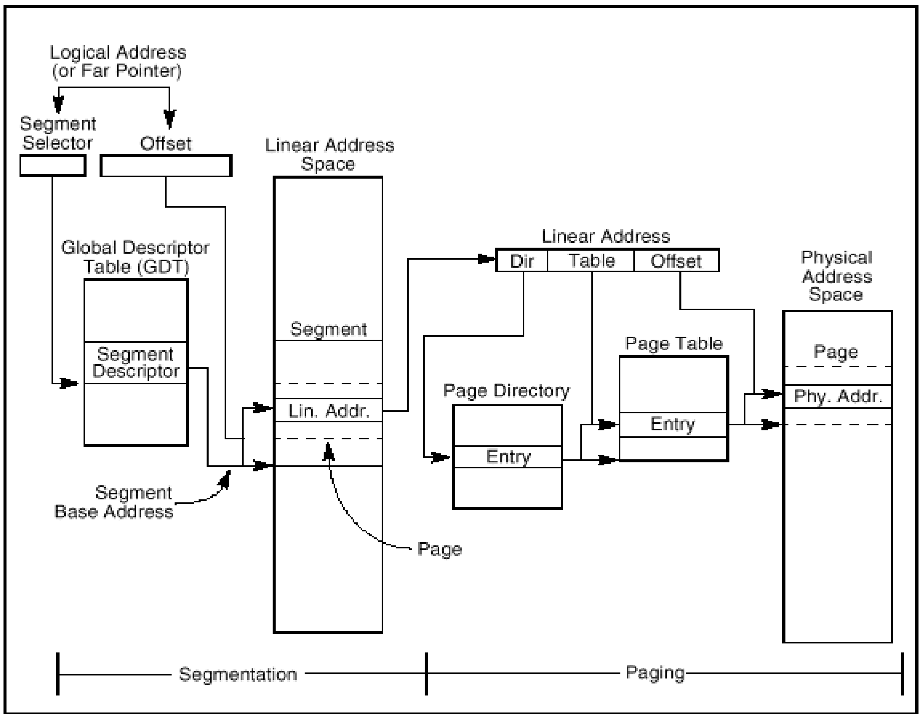


Figure 1: Segmentation and Paging

All of the segments within a system are contained in the processor's linear address space. To locate a byte in a particular segment, a **logical address** (sometimes called a far pointer) must be provided. A logical address consists of a segment selector and an offset. The segment selector is a unique identifier for a segment. Among other things it provides an offset into a descriptor table (such as the global descriptor table, GDT) to an entry (a data structure) called a segment descriptor. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor's linear address space.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (ordinarily 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location. If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program.

1. Physical address space

In protected mode (real mode is used to maintain processor compatibility with older models than Intel 80286), the Intel Architecture provides a normal physical address space of 4 Gbytes (2^{32} bytes). This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFFFFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

(Introduced in the Pentium Pro processor.) The Intel Architecture also supports an extension of the physical address space to 2^{36} bytes (64 GBytes), with a maximum physical address of FFFFFFFFFH. This extension is invoked with the physical address extension (PAE) flag, located in bit 5 of control register CR4.

2. Logical and linear addresses

At the system-architecture level in protected mode, the processor uses two stages of address translation to arrive at a physical address: logical-address translation and linear address space paging. A logical address consists of a 16-bit segment selector and a 32-bit offset (see Figure

2). The segment selector identifies the segment in which the byte is located, and the offset specifies the location of the byte in the segment relative to the base address of the segment.

The processor translates every logical address into a linear address. A linear address is a 32-bit address in the processor's linear address space. Like the physical address space, the linear address space is a flat (unsegmented), 2^{32}-byte address space, with addresses ranging from 0 to FFFFFFFFH. The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor.
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.

Following are some related system registers and data structures:

<u>Segment selectors</u>

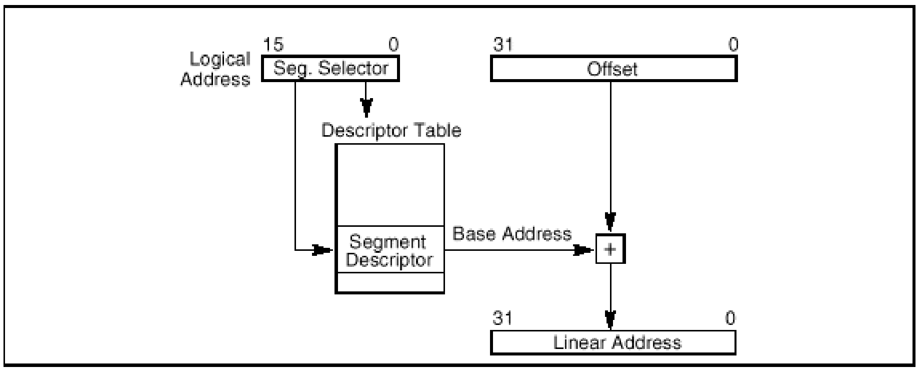


Figure 2: Logical address to linear address translation

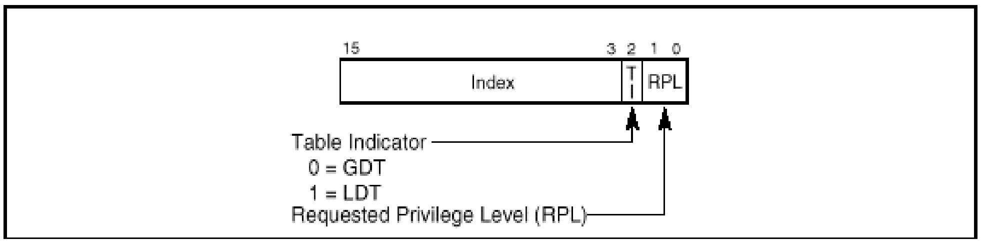


Figure 3: Segment selector

A segment selector is a 16-bit identifier for a segment (see Figure 3). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index (Bits 3 through 15). Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

TI (table indicator) flag (Bit 2). Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

Requested Privilege Level (RPL) (Bits 0 and 1). Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level.

The first entry of the GDT is not used by the processor. A segment selector that points to this entry of the GDT (that is, a segment selector with an index of 0 and the TI flag set to 0) is used as a “null segment selector.” The processor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. A null selector can be used to initialize unused segment registers. Loading the CS or SS register with a null segment selector causes a general-protection exception (#GP) to be generated

Segment registers

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 4). Each of these segment registers supports a specific kind of memory reference (code, stack, or data). For virtually any kind of program execution to take place, at least the code-segment (CS), data-segment (DS), and stack-segment (SS) registers must be loaded with valid segment selectors. The processor also provides three additional data-segment registers (ES, FS, and GS), which can be used to make additional data segments available to the currently executing program (or task).

For a program to access a segment, the segment selector for the segment must have been loaded in one of the segment registers. So, although a system can define thousands of segments, only 6 can be available for immediate use. Other segments can be made available by loading their segment selectors into these registers during program execution.

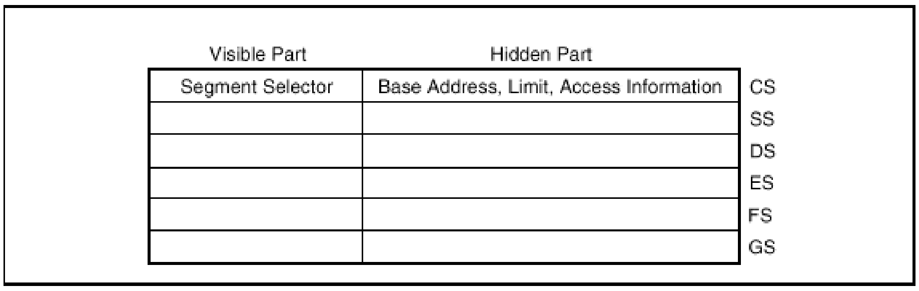


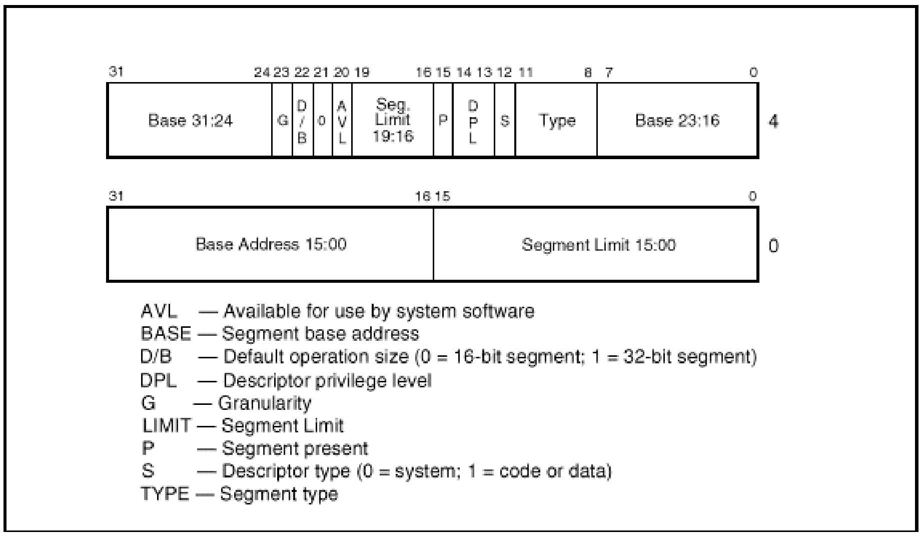
Figure 4: Segment registers

Every segment register has a “visible” part and a “hidden” part. When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Segment descriptors

A segment descriptor is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information. Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs. Figure 5 illustrates the general descriptor format for all types of segment descriptors.

The flags and fields in a segment descriptor are as follows:



31
24 23 22 21 20 19
16 15 14 13 12 11
8 7
0
Base 31:24
G
D/B
0
AVL
Seg. Limit 19:16
P
DPL
S
Type
Base 23:16
4
31
16 15
0
Base Address 15:00
Segment Limit 15:00
0
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 5: Segment descriptor

Segment limit field

Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

Base address fields

Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries.

Type field

Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor.

S (descriptor type) flag

Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

DPL (descriptor privilege level) field

Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment.

P (segment-present) flag

Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register. Memory management software can use this flag to control which segments are actually loaded into physical memory at a given time. It offers a control in addition to paging for managing virtual memory.

D/B (default operation size/default stack pointer size and/or upper bound) flag

Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

- **Executable code segment.** The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed. The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.
- **Stack segment (data segment pointed to by the SS register).** The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the

flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.

- **Expand-down data segment.** The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).

G (granularity) flag

Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units.

Available and reserved bits

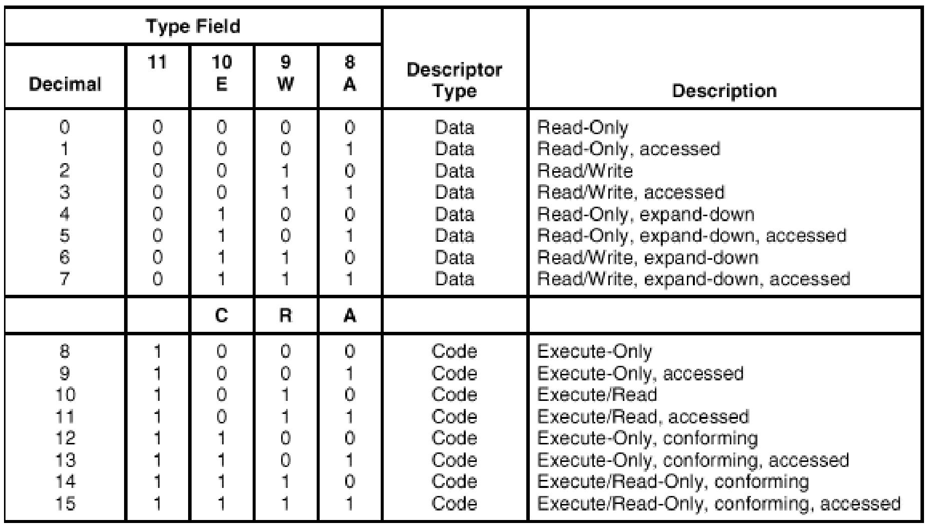
Bit 20 of the second double word of the segment descriptor is available for use by system software; bit 21 is reserved and should always be set to 0.

Code- and Data-segment descriptor types

When the S (descriptor type) flag in a segment descriptor is set, the descriptor is for either a code or a data segment. The highest order bit of the type field (bit 11 of the second double word of the segment descriptor) then determines whether the descriptor is for a data segment (clear) or a code segment (set).

For data segments, the three low-order bits of the type field (bits 8, 9, and 10) are interpreted as accessed (A), write-enable (W), and expansion-direction (E). See Table 1 for a description of the encoding of the bits in the type field for code and data segments.

Table 1



Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

Segment descriptor types

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

- Local descriptor-table (LDT) segment descriptor.
- Task-state segment (TSS) descriptor.
- Call-gate descriptor.
- Interrupt-gate descriptor.
- Trap-gate descriptor.
- Task-gate descriptor.

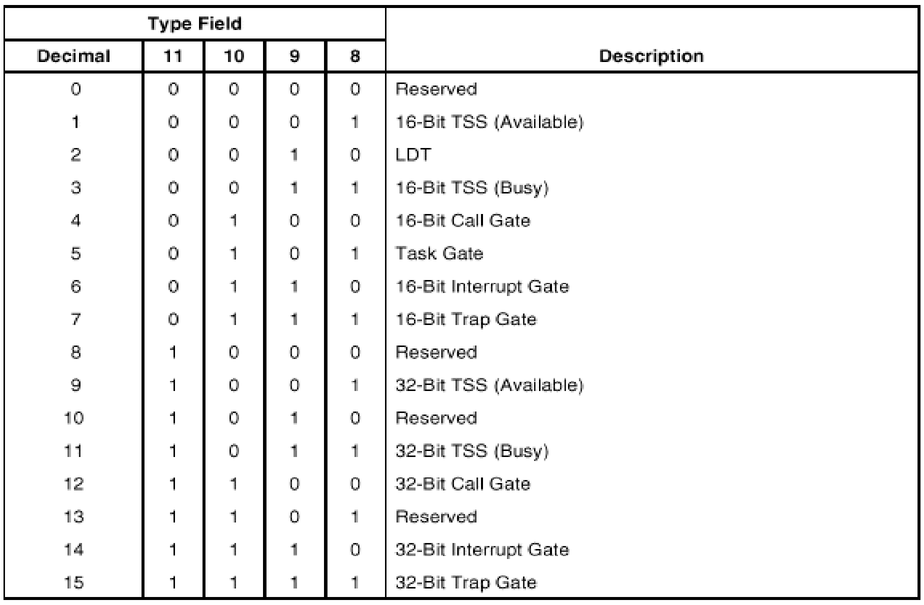
These descriptor types fall into two categories: system-segment descriptors and gate descriptors. System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves "gates," which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS's (task gates). Table 2 shows the encoding of the type field for system-segment descriptors and gate descriptors.

<u>Segment descriptor tables</u>

A segment descriptor table is an array of segment descriptors (see Figure 6). A descriptor table is variable in length and can contain up to 8192 (2^{13}) 8-byte descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT)

Table 2



Type Field					
Decimal	11	10	9	8	Description
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

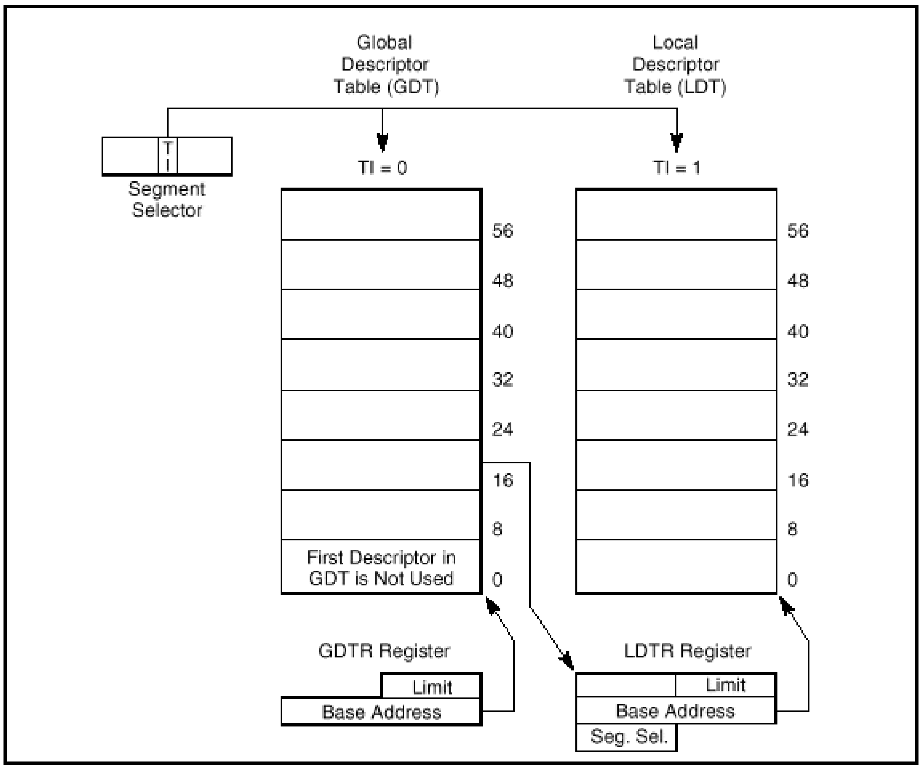


Figure 6 GDT and LDT

- The local descriptor tables (LDT)

Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in the linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register.

The LDT is located in a system segment of the LDT type. The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. The segment descriptor for an LDT can be located anywhere in the GDT. An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register

3. Paging (Linear address to physical address)

When paging is used, the processor divides the linear address space into fixed-size pages (generally 4 KBytes in length) that can be mapped into physical memory and/or disk storage. When a program (or task) references a logical address in memory, the processor translates the address into a linear address and then uses its paging mechanism to translate the linear address into a corresponding physical address. If the page containing the linear address is not currently in physical memory, the processor generates a page-fault exception (#PF). The exception handler for the page-fault exception typically directs the operating system or executive to load the page from disk storage into physical memory (perhaps writing a different page from physical memory out to disk in the process). When the page has been loaded in physical memory, a return from the exception handler causes the instruction that generated the exception to be restarted. The information that the processor uses to map linear addresses into the physical address space and to generate page-fault exceptions (when necessary) is contained in page directories and page tables stored in memory.

Paging is controlled by three flags in the processor’s control registers:

- PG (paging) flag, bit 31 of CR0: The PG flag enables the page-translation mechanism.
- PSE (page size extensions) flag, bit 4: The PSE flag enables large page sizes: 4-MByte pages or 2-MByte pages (when the PAE flag is set). When the PSE flag is clear, the more common page length of 4 KBytes is used.
- PAE (physical address extension) flag, bit 5 of CR4: The PAE flag enables 36-bit physical addresses.

The information that the processor uses to translate linear addresses into physical addresses (when paging is enabled) is contained in four data structures:

- Page directory: an array of 32-bit page-directory entries (PDEs) contained in a 4-Kbyte page. Up to 1024 page-directory entries can be held in a page directory.
- Page table: an array of 32-bit page-table entries (PTEs) contained in a 4-KByte page. Up to 1024 page-table entries can be held in a page table. (Page tables are not used for 2-MByte or 4-MByte pages. These page sizes are mapped directly from one or more page-directory entries.)
- Page: a 4-KByte, 2-MByte, or 4-MByte flat address space.
- Page-Directory-Pointer Table: an array of four 64-bit entries, each of which points to a page directory. This data structure is only used when the physical address extension is enabled.

Figure 7 shows the page directory and page-table hierarchy when mapping linear addresses to 4-KByte pages. The entries in the page directory point to page tables, and the entries in a page table point to pages in physical memory. This paging method can be used to address up to 2^{20} pages, which spans a linear address space of 2^{32} bytes (4 GBytes).

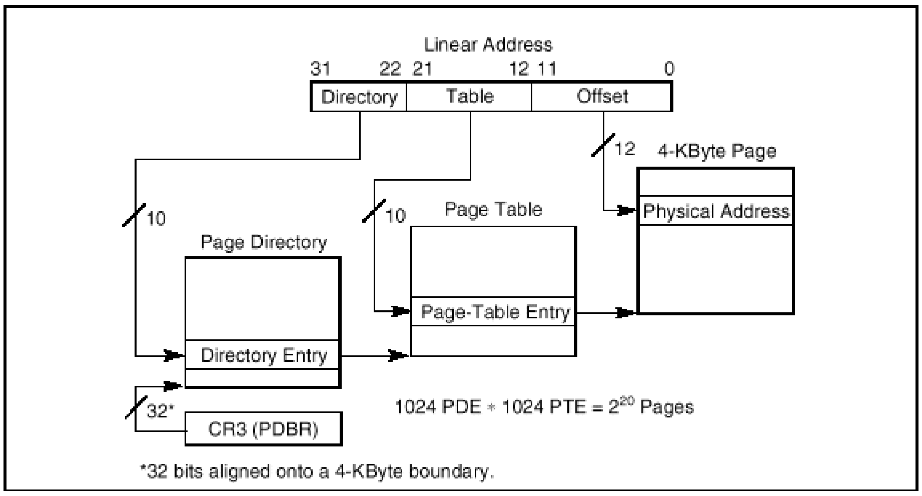


Figure 7: Linear address to physical address translation

To select the various table entries, the linear address is divided into three sections:

- Page-directory entry: Bits 22 through 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a page table.
- Page-table entry: Bits 12 through 21 of the linear address provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory.
- Page offset: Bits 0 through 11 provides an offset to a physical address in the page.

Figure 8 shows the format for the page-directory and page-table entries when 4-Kbyte pages and 32-bit physical addresses are being used. The functions of the flags and fields in the entries are as follows:

Page base address, bits 12 through 32

(Page-table entries for 4-KByte pages.) Specifies the physical address of the first byte of a 4-KByte page. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces pages to be aligned on 4-KByte boundaries.

(Page-directory entries for 4-KByte page tables.) Specifies the physical address of the first byte of a page table. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces page tables to be aligned on 4-KByte boundaries.

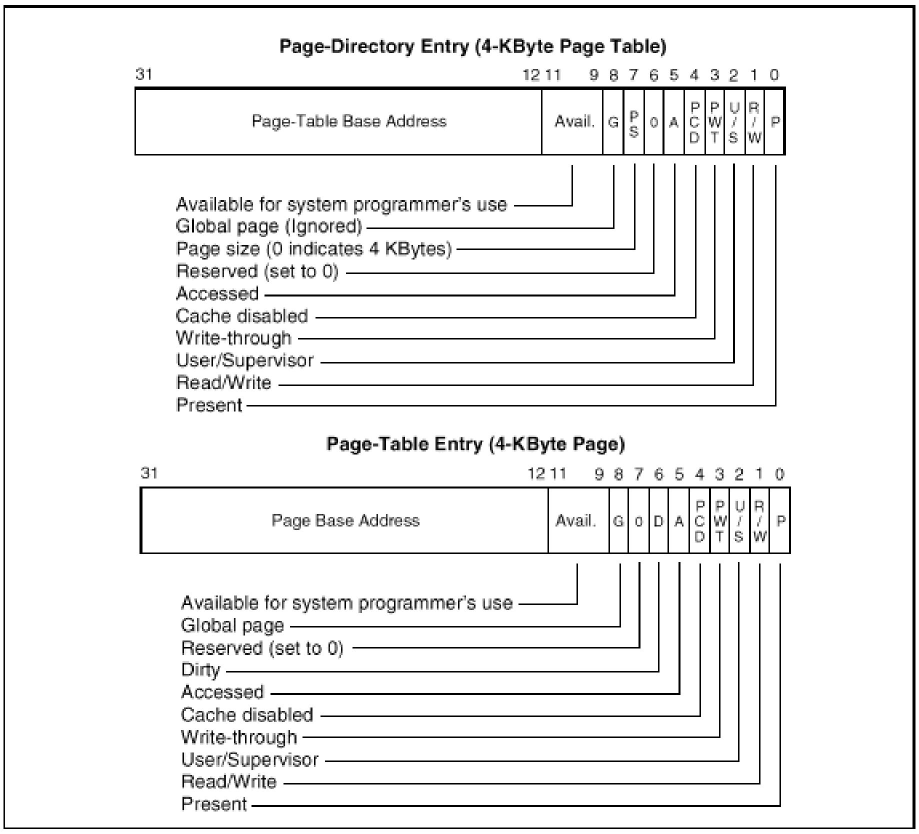


Figure 8: Page-Directory and Page-Table Entries

Present (P) flag, bit 0

Indicates whether the page or page table being pointed to by the entry is currently loaded in physical memory. When the flag is set, the page is in physical memory and address translation is carried out. When the flag is clear, the page is not in memory and, if the processor attempts to access the page, it generates a page-fault exception (#PF).

Read/write (R/W) flag, bit 1

Specifies the read-write privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When this flag is clear, the page is read only; when the flag is set, the page can be read and written into.

User/supervisor (U/S) flag, bit 2

Specifies the user-supervisor privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When this flag is clear, the page is assigned the supervisor privilege level; when the flag is set, the page is assigned the user privilege level.

Page-level write-through (PWT) flag, bit 3

Controls the write-through or write-back caching policy of individual pages or page tables. When the PWT flag is set, write-through caching is enabled for the associated page or page table; when the flag is clear, write-back caching is enabled for the associated page or page table.

Page-level cache disable (PCD) flag, bit 4

Controls the caching of individual pages or page tables. When the PCD flag is set, caching of the associated page or page table is prevented; when the flag is clear, the page or page table can be cached. This flag permits caching to be disabled for pages that contain memory-mapped I/O ports or that do not provide a performance benefit when cached.

Accessed (A) flag, bit 5

Indicates whether a page or page table has been accessed (read from or written to) when set. Memory management software typically clears this flag when a page or page table is initially loaded into physical memory. The processor then sets this flag the first time a page or page table is accessed. This flag is a "sticky" flag, meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The accessed and dirty flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

Dirty (D) flag, bit 6

Indicates whether a page has been written to when set. (This flag is not used in page-directory entries that point to page tables.) Memory management software typically clears this flag when a page is initially loaded into physical memory. The processor then sets this flag the first time a page is accessed for a write operation. This flag is "sticky," meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The dirty and accessed flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

Page size (PS) flag, bit 7

Determines the page size. This flag is only used in page-directory entries. When this flag is clear, the page size is 4 KBytes and the page-directory entry points to a page table. When the flag is set, the page size is 4 MBytes for normal 32-bit addressing (and 2 MBytes if extended physical addressing is enabled) and the page-directory entry points to a page. If the page-directory entry points to a page table, all the pages associated with that page table will be 4-Kbyte pages.

Global (G) flag, bit 8

(Introduced in the Pentium Pro processor.) Indicates a global page when set. When a page is marked global and the page global enable (PGE) flag in register CR4 is set, the page-table or page-directory entry for the page is not invalidated in the TLB when register CR3 is loaded or a task switch occurs. This flag is provided to prevent frequently used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. Only software can set or clear this flag. For page-directory entries that point to page tables, this flag is ignored and the global characteristics of a page are set in the page-table entries.

Reserved and available-to-software bits

In a page-table entry, bit 7 is reserved and should be set to 0; in a page-directory entry that points to a page table, bit 6 is reserved and should be set to 0. For a page-directory entry for a 4-MByte page, bits 12 through 21 are reserved and must be set to 0, for Intel Architecture processors through the Pentium II processor. For both types of entries, bits 9, 10, and 11 are available for use by software. (When the present bit is clear, bits 1 through 31 are available to software.) When the PSE and PAE flags in control register CR4 are set, the processor generates a page fault if reserved bits are not set to 0.

4. Virtual address space details

Although virtual addresses are 64 bits wide in 64-bit mode, current implementations (and any chips known to be in the planning stages) do not allow the entire virtual address space of 16 EB (18,446,744,073,709,551,616 bytes) to be used. Most operating systems and applications will not need such a large address space for the foreseeable future (for example, Windows implementations for AMD64 are only populating 16 TB (17,592,186,044,416 bytes), or 44 bits' worth), so implementing such wide virtual addresses would simply increase the complexity and cost of address translation with no real benefit. AMD therefore decided that, in the first implementations of the architecture, only the least significant 48 bits of a virtual address would actually be used in address translation (page table lookup). However, bits 48 through 63 of any virtual address must be copies of bit 47 (in a manner akin to sign extension), or the processor will raise an exception. Addresses complying with this rule are referred to as "canonical form." Canonical form addresses run from 0 through 00007FFF`FFFFFFFF, and from FFFF8000`00000000 through FFFFFFFF`FFFFFFFF, for a total of 256 TB (281,474,976,710,656 bytes) of usable virtual address space.

This scheme allows an important feature for later scalability to true 64-bit addressing: many operating systems (including, but not limited to, the Windows NT family) take the higher-addressed half of the address space (named kernel space) for themselves and leave the lower-addressed half (user space) for application code, user mode stacks, heaps, and other data regions. The "canonical address" design ensures that every AMD64 compliant implementation has, in effect, two memory halves: the lower half starts at 00000000`00000000 and "grows upwards" as more virtual address bits become available, while the higher half is "docked" to the top of the address space and grows downwards. Also, fixing the contents of the unused address bits prevents their use by operating system as flags, privilege markers, etc., as such use could become problematic when the architecture is indeed extended to 52, 56, 60 and 64 bits.

The 64-bit addressing mode ("long mode") is a superset of Physical Address Extensions (PAE); because of this, page sizes may be 4 KB (4096 bytes), 2 MB (2,097,152 bytes), or 1 GB (1,073,741,824 bytes). However, rather than the three-level page table system used by systems in PAE mode, systems running in long mode use four levels of page table: PAE's *Page-Directory Pointer Table* is extended from 4 entries to 512, and an additional *Page-Map Level 4 Table* is added, containing 512 entries in 48-bit implementations. In implementations providing larger virtual addresses, this latter table would either grow to accommodate sufficient entries to describe the entire address range, up to a theoretical maximum of 33,554,432 entries for a 64-bit implementation, or be over ranked by a new mapping level, such as a PML5. A full mapping hierarchy of 4 KB (4096 bytes) pages for the whole 48-bit space would take a bit more than 512 GB (549,755,813,888 bytes) of RAM (about 0.196% of the 256 TB [281,474,976,710,656 bytes] virtual space).