2.5 Address Space

The IBM 6x86 CPU can directly address 64 KBytes of I/O space and 4 GBytes of physical memory (Figure 2-24).

Memory Address Space. Access can be made to memory addresses between 0000 0000h and FFFF FFFFh. This 4 GByte memory space can be accessed using byte, word (16 bits), or doubleword (32 bits) format. Words and doublewords are stored in consecutive memory bytes with the low-order byte located in the lowest address. The physical address of a word or doubleword is the byte address of the low-order byte.

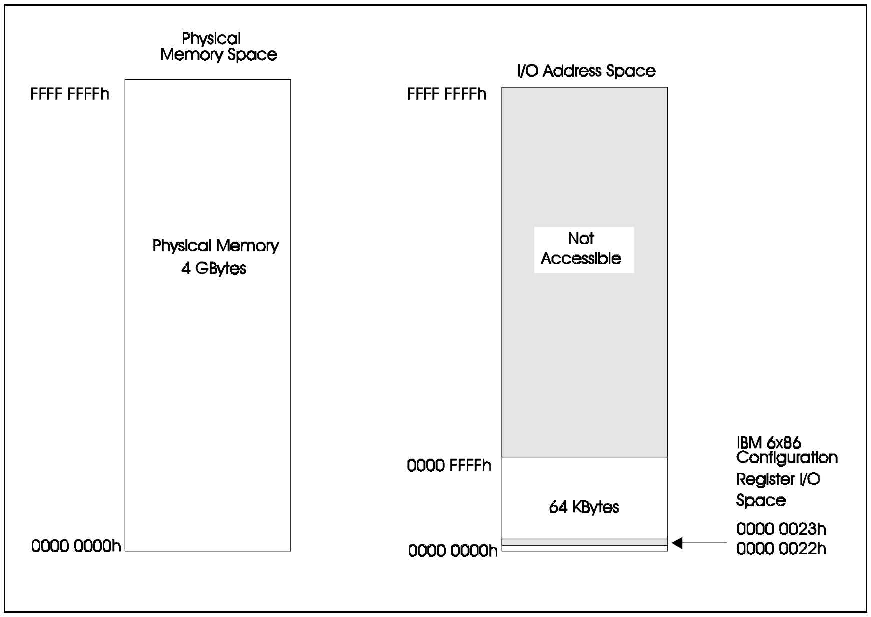

Figure 2-24. Memory and I/O Address Spaces

I/O Address Space

The IBM 6x86 I/O address space is accessed using IN and OUT instructions to addresses referred to as “ports”. The accessible I/O address space size is 64 KBytes and can be accessed through 8-bit, 16-bit or 32-bit ports. The execution of any IN or OUT instruction causes the M/IO# pin to be driven low, thereby selecting the I/O space instead of memory space.

The accessible I/O address space ranges between locations 0000 0000h and 0000 FFFFh (64 KBytes). The I/O locations (ports) 22h and 23h can be used to access the IBM 6x86 configuration registers.

2.6 Memory Addressing Methods

With the IBM 6x86 CPU, memory can be addressed using nine different addressing modes (Table 2-23, Page 2-42). These addressing modes are used to calculate an offset address often referred to as an effective address. Depending on the operating mode of the CPU, the offset is then combined using memory management mechanisms to create a physical address that actually addresses the physical memory devices.

Memory management mechanisms on the IBM 6x86 CPU consist of segmentation and paging. Segmentation allows each program to use several independent, protected address spaces. Paging supports a memory subsystem that simulates a large address space using a small amount of RAM and disk storage for physical memory. Either or both of these mechanisms can be used for management of the IBM 6x86 CPU memory address space.

2.6.1 Offset Mechanism

The offset mechanism computes an offset (effective) address by adding together one or more of three values: a base, an index and a displacement. When present, the base is the value of one of the eight 32-bit general registers. The index if present, like the base, is a value that is in one of the eight 32-bit general purpose registers (not including the ESP register). The index differs from the base in that the index is first multiplied by a scale factor of 1, 2, 4 or 8 before the summation is made. The third component added to the memory address calculation is the displacement. The displacement is a value of up to 32-bits in length supplied as part of the instruction. Figure 2-25 illustrates the calculation of the offset address.

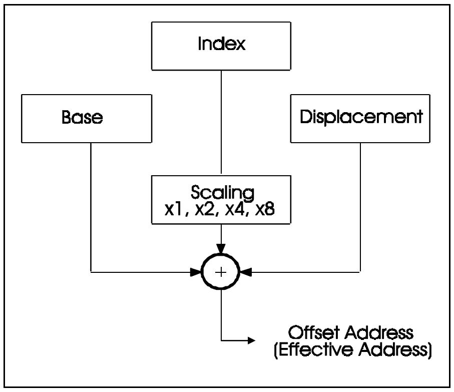

Figure 2-25. Offset Address Calculation

Nine valid combinations of the base, index, scale factor and displacement can be used with the IBM 6x86 CPU instruction set. These combinations are listed in Table 2-23. The base and index both refer to contents of a register as indicated by [Base] and [Index].

Table 2-23. Memory Addressing Modes

ADDRESSING MODE	BASE	INDEX	SCALE FACTOR (SF)	DISPLACEMENT (DP)	OFFSET ADDRESS (OA) CALCULATION
Direct				x	OA = DP
Register Indirect	x				OA = [BASE]
Based	x			x	OA = [BASE] + DP
Index		x		x	OA = [INDEX] + DP
Scaled Index		x	x	x	OA = ([INDEX] * SF) + DP
Based Index	x	x			OA = [BASE] + [INDEX]
Based Scaled Index	x	x	x		OA = [BASE] + ([INDEX] * SF)
Based Index with Displacement	x	x		x	OA = [BASE] + [INDEX] + DP
Based Scaled Index with Displacement	x	x	x	x	OA = [BASE] + ([INDEX] * SF) + DP

2.6.2 Memory Addressing

Real Mode Memory Addressing

In real mode operation, the IBM 6x86 CPU only addresses the lowest 1 MByte of memory. To calculate a physical memory address, the 16-bit segment base address located in the selected segment register is multiplied by 16 and then the 16-bit offset address is added. The resulting 20-bit address is then extended. Three hexadecimal zeros are added as upper address bits to create the 32-bit physical address. Figure 2-26 illustrates the real mode address calculation.

The addition of the base address and the offset address may result in a carry. Therefore, the resulting address may actually contain up to 21 significant address bits that can address memory in the first 64 KBytes above 1 MByte.

Protected Mode Memory Addressing

In protected mode three mechanisms calculate a physical memory address (Figure 2-27, Page 2-44).

- **Offset Mechanism** that produces the offset or effective address as in real mode.
- **Selector Mechanism** that produces the base address.
- Optional **Paging Mechanism** that translates a linear address to the physical memory address.

The offset and base address are added together to produce the linear address. If paging is not enabled, the linear address is used as the physical memory address. If paging is enabled, the paging mechanism is used to translate the linear address into the physical address. The offset mechanism is described earlier in this section and applies to both real and protected mode. The selector and paging mechanisms are described in the following paragraphs.

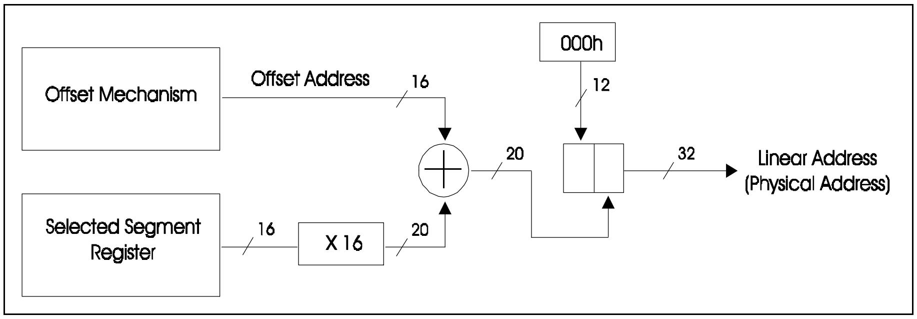

Figure 2-26. Real Mode Address Calculation

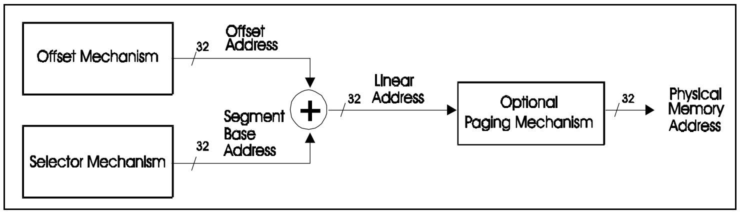

Figure 2-27. Protected Mode Address Calculation

2.6.3 Selector Mechanism

Using segmentation, memory is divided into an arbitrary number of segments, each containing usually much less than the 2^{32} byte (4 GByte) maximum.

The six segment registers (CS, DS, SS, ES, FS and GS) each contain a 16-bit selector that is used when the register is loaded to locate a segment descriptor in either the global descriptor table (GDT) or the local descriptor table (LDT). The segment descriptor defines the base address, limit, and attributes of the selected segment and is cached on the IBM 6x86 CPU as a result of loading the selector. The cached descriptor contents are not visible to the programmer. When a memory reference occurs in protected mode, the linear address is generated by adding the segment base address in the hidden portion of the segment register to the offset address. If paging is not enabled, this linear address is used as the physical memory address. Figure 2-28 illustrates the operation of the selector mechanism.

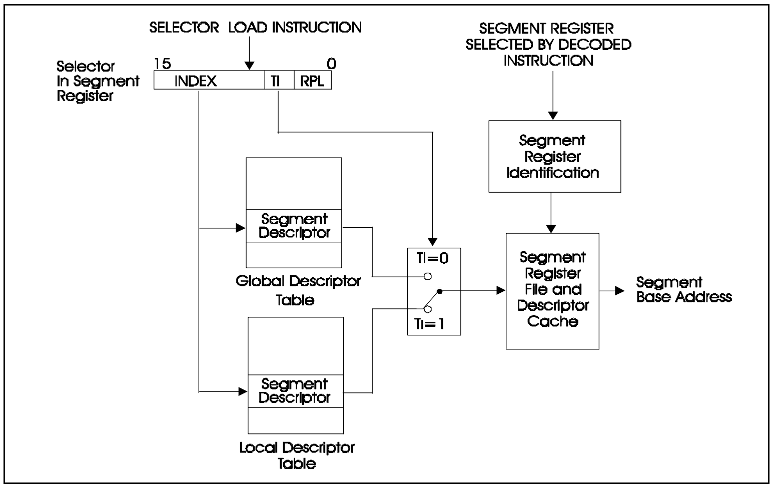
SELECTOR LOAD INSTRUCTION
Selector In Segment Register
15
0
INDEX
TI
RPL
SEGMENT REGISTER SELECTED BY DECODED INSTRUCTION
Segment Register Identification
Segment Descriptor
Global Descriptor Table
TI=0
TI=1
Segment Register File and Descriptor Cache
Segment Base Address
Segment Descriptor
Local Descriptor Table

Figure 2-28. Selector Mechanism

2.6.4 Paging Mechanisms

The paging mechanisms (Figure 2-29) translate linear addresses to their corresponding physical addresses. For traditional paging, the page size is always 4 KBytes. If IBM 6x86 Variable-Size Paging is selected, a page size may be as large as 4 GBytes. Use of larger page sizes allows large memory areas such as video memory to be placed in a single page, eliminating page table thrashing.

Paging is activated when the PG and the PE bits within the CR0 register are set.

2.6.4.1 Traditional Paging Mechanism

The traditional paging mechanism translates the 20 most significant bits of a linear address to a physical address. The linear address is divided into three fields DTI, PTI, PFO (Figure 2-30, Page 2-46). These fields respectively select:

- an entry in the directory table,
- an entry in the page table selected by the directory table
- the offset in the physical page selected by the page table

The directory table and all the page tables can be considered as pages as they are 4-KBytes in size and are aligned on 4-KByte boundaries. Each entry in these tables is 32 bits in length. The fields within the entries are detailed in Figure 2-31 (Page 2-46) and Table 2-24 (Page 2-47).

A single page directory table can address up to 4 GBytes of virtual memory (1,024 page tables—each table can select 1,024 pages and each page contains 4 KBytes).

Translation Lookaside Buffer (TLB) is made up of three caches (Figure 2-30, Page 2-46).

- the DTE Cache caches directory table entries
- the Main TLB caches page tables entries
- the Victim TLB stores PTEs that have been evicted from the Main TLB

The DTE cache is a 4-entry fully associative cache, the main TLB is a 128-entry direct mapped cache and the victim TLB is an 8-entry fully associative cache.The DTE cache caches the four most recent DTEs so that future TLB misses only require a single page table read to calculate the physical address. The DTE cache is disabled following RESET and is enabled by setting the DTE_EN bit (CCR4 bit4).

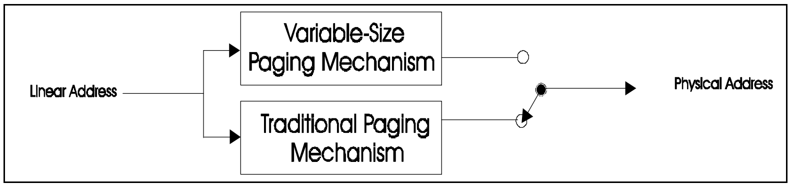

Figure 2-29. Paging Mechanisms

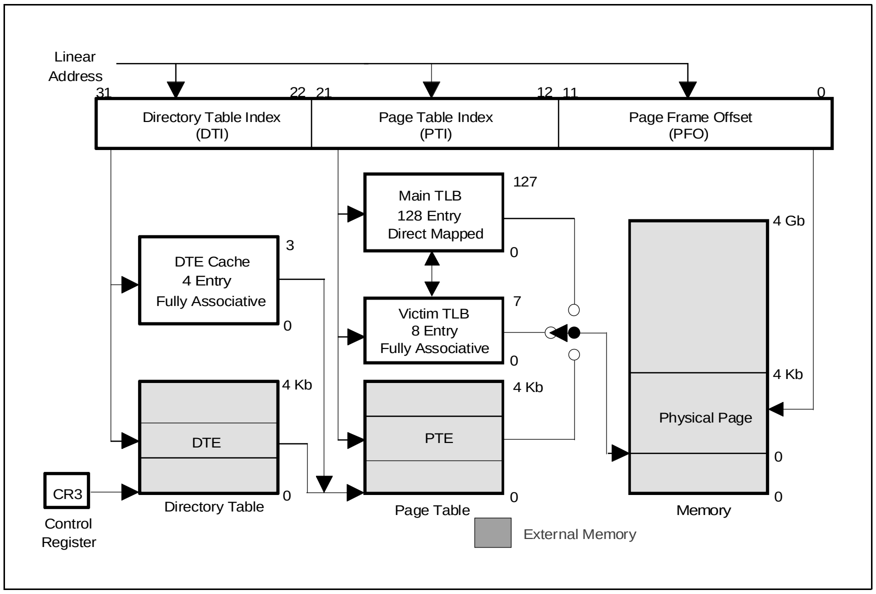

Figure 2-30. Traditional Paging Mechanism

31 12	11 10 9	8 7	6	5	4	3	2	1	0
BASE ADDRESS	AVAILABLE	RESERVED	D	A	P C D	P W T	U / S	W / R	P

Note: In DTE format, bit 6 is reserved

Figure 2-31. Directory and Page Table Entry (DTE and PTE) Format

Table 2-24. Directory and Page Table Entry (DTE and PTE) Bit Definitions

BIT POSITION	FIELD NAME	DESCRIPTION
31-12	BASE ADDRESS	Specifies the base address of the page or page table.
11-9	--	Undefined and available to the programmer.
8-7	--	Reserved and not available to the programmer.
6	D	Dirty Bit. If set, indicates that a write access has occurred to the page (PTE only, undefined in DTE).
5	A	Accessed Flag. If set, indicates that a read access or write access has occurred to the page.
4	PCD	Page Caching Disable Flag. If set, indicates that the page is not cacheable in the on-chip cache.
3	PWT	Page Write-Through Flag. If set, indicates that writes to the page or page tables that hit in the on-chip cache must update both the cache and external memory.
2	U/S	User/Supervisor Attribute. If set (user), page is accessible at privilege level 3. If clear (supervisor), page is accessible only when CPL ≤ 2.
1	W/R	Write/Read Attribute. If set (write), page is writable. If clear (read), page is read only.
0	P	Present Flag. If set, indicates that the page is present in RAM memory, and validates the remaining DTE/PTE bits. If clear, indicates that the page is not present in memory and the remaining DTE/PTE bits can be used by the programmer.

For a TLB hit, the TLB eliminates accesses to external directory and page tables.

The victim TLB increases the apparent associativity of the main TLB and helps eliminate TLB trashing (unproductive TLB management). When an entry in the main TLB is replaced, a copy of the replaced entry is sent to the victim TLB before the entry in the main TLB is overwritten. If the victim TLB receives a hit, its entry is swapped with a main TLB entry.

The TLB must be flushed by the software when entries in the page tables are changed. The TLB is flushed whenever the CR3 register is loaded. A particular page can be flushed from the TLB by using the INVLPG instruction. This instruction also flushes the entire DTE cache.

2.6.4.2 Translation Lookaside Buffer Testing

The TLB can be tested by writing to a main TLB followed by performing a TLB lookup (TLB read) to see if the expected contents are within the TLB. TLB test operations are performed using test register TR6 and TR7 shown in Figure 2-32 (Page 2-48). Tables 2-25 through 2-27 list the bit definitions for TR6 and TR7.

Main TLB Write. To perform a direct write to a main TLB entry, the TR7 register is configured with the desired physical address as well as the PCD and PWT bits. The BI, HV, HD and HB bits are not used. The TR6 register is then configured with the linear address, D, U, W and V bits. The D, U, and W bits must be complements of the D#, U#, and W# bits during a write. When the TR6 register is configured, the IBM 6x86 CPU writes the linear and physical address into the main TLB along with the A, D, U, and W bits. The main TLB entry is selected by bits 12 through 18 of the linear address field.

TLB Lookup. During a TLB lookup, the IBM 6x86 CPU queries the TLB with a given linear address and expected A, W, U and D values. The query returns a corresponding physical address, and the source of the address. The address source could be from the main TLB, from the victim TLB or from the variable-size paging mechanism.

The TLB lookup involves a single TR6 register write. The CMD bits are set to 0x1. The D, U, W, D#, U# and W# bits are not used during TLB lookups.

After a TLB lookup, the HV, HD and HB bits in TR7 indicate which (if any) PTEs were found with the requested linear address. If a TLB entry was found for a PTE in the victim or variable size-paging cache, the BI bit in the TR7 register will contain the index of the particular entry. If multiple entries respond, only the HV, HD and HB bits are valid and all TR7 fields are undefined.

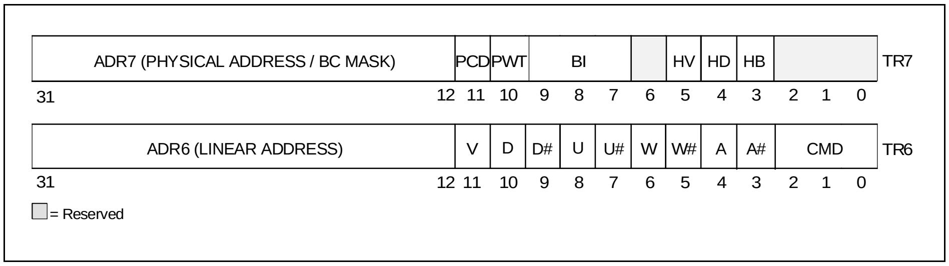

Figure 2-32. TLB Test Registers

Table 2-25. TLB Test Register Bit Definitions

REGISTER NAME	NAME	RANGE	DESCRIPTION
TR7	ADR7	31-12	Physical address or variable page size mechanism mask. TLB lookup: data field from the TLB. TLB write: data field written into the TLB.
	PCD	11	Page-level cache disable bit (PCD). Corresponds to the PCD bit of a page table entry.
	PWT	10	Page-level cache write-through bit (PWT). Corresponds to the PWT bit of a page table entry.
	BI	9-7	Cell index for victim TLB and block cache operations.
	HV	5	Victim TLB hit.
	HD	4	Main TLB hit.
	HB	3	Variable-Size Paging Mechanism hit.
TR6	ADR6	31-12	Linear Address. TLB lookup: The TLB is interrogated per this address. If one and only one match occurs in the TLB, the rest of the fields in TR6 and TR7 are updated per the matching TLB entry. TLB write: A TLB entry is allocated to this linear address.
	V	11	PTE Valid. TLB write: If set, indicates that the TLB entry contains valid data. If clear, target entry is invalidated.
	D, D#	10-9	Dirty Attribute Bit and its complement. Refer to Table 2-26., Page 2-50.
	U, U#	8-7	User/Supervisor Attribute Bit and its complement. Refer to Table 2-26., Page 2-50.
	W, W#	6-5	Write Protect bit and its complement. Refer to Table 2-26., Page 2-50.
	A, A#	4-3	Accessed Bit and its complement. Used for block cache entries only. Refer to Table 2-26., Page 2-50.
	CMD	2-0	Array Command Select. Determines TLB array command. Refer to Table 2-27, Page 2-50.

Table 2-26. TR6 Attribute Bit Pairs

BIT	BIT#	EFFECT ON TLB LOOKUP	EFFECT ON TLB WRITE
0	0	Do not match.	Undefined.
0	1	If bit = 0, match.	Bit is cleared.
1	0	If bit = 1, match.	The bit is set.
1	1	If bit = 0 or 1, match.	Undefined.

Note: “BIT” applies to A, D, U or W fields in TR6; “BIT#” applies to A#, D#, U#, or W# fields in TR6.

Table 2-27. TR6 Command Bits

CMD	Command
0x0	Direct write to main TLB.
0x1	TLB lookup for a linear address in all arrays.
100	Write to variable page size mask only.
110	Write to variable page size linear and physical address fields.
101	Read variable page size mask and linear address.
111	Read variable page size cache physical and linear address.

Note: x = don’t care

2.6.5 Variable-Size Paging Mechanism

The Variable-Size Paging Mechanism (VSPM) is an advanced alternative to traditional paging. As shown in Figure 2-33, VSPM allows the creation of pages ranging in size from 4 KBytes to 4 GBytes. The larger page size nearly eliminates page table thrashing associated with using multiple 4-KByte pages.

For example, paging 1 MByte of memory requires 256 4-KByte pages using traditional paging. The software not only incurs overhead during setting up the 256 pages, but also incurs additional overhead accessing the page tables each time a page is not found in the on-chip TLB. In contrast, a single 1-MByte page virtually eliminates the overhead.

Configuring Variable-Size Pages. The VSPM is configured using TLB test registers, TR6 and TR7 (These registers are also used to test the TLB). The VSPM configuration is performed in much the same manner as when writing to a line of the TLB (Refer to Section 2.6.4.2.). The major exception to this, is that a mask field is written to the VSPM as part of the VSPM configuration.

The physical address, linear address, valid bit and attribute bits in a main TLB write all have the same meaning as in a main TLB read except that CMD=110. The BI field is used to select the VSPM cell to be written.

A VSPM mask setup operation is performed when CMD=100 and a test register write is performed. During a VSPM mask setup, the TR7 address field is used as the mask field. The mask field selectively masks linear address bits 31-12 from the VSPM tag compare. This has the effect of allowing the VSPM to map pages greater than 4 KBytes.

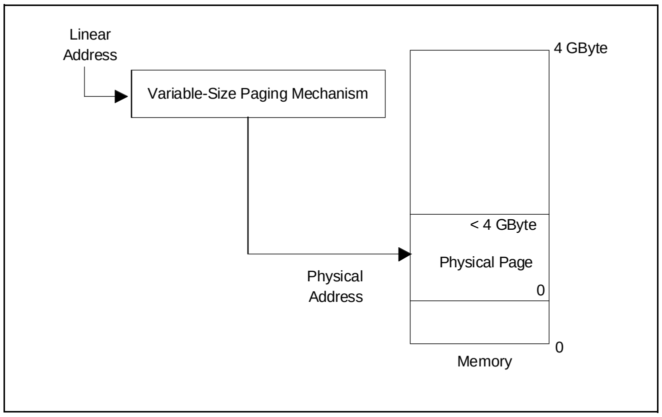

Figure 2-33. Variable-Size Paging Mechanism

After a VSPM mask setup, the valid bit, attribute bits, and the linear address are left in undefined states. Therefore, the VSPM mask setup should be performed prior to other VSPM operations.

Unlike the victim and main TLBs, the VSPM operations make use of the accessed bit. During a VSPM mask or physical address write the A and A# fields are written to the VSPM.

VSPM Reads. VSPM reads are performed with the address of the entry to be read in the BI field of the TR7 register and with CMD=111. The entry's and physical address is read into the TR6 and TR7 address fields as well as the valid bit, and attribute bits.

If CMD=101, the linear address, mask, valid bit and attribute bits are read.

2.7 Memory Caches

The IBM 6x86 CPU contains two memory caches as described in Chapter 1. The Unified Cache acts the primary data cache, and secondary instruction cache. The Instruction Line Cache is the primary instruction cache and provides a high speed instruction stream for the Integer Unit.

The unified cache is dual-ported allowing simultaneous access to any two unique banks. Two different banks may be accessed at the same time permitting any two of the following operations to occur in parallel:

- Code fetch
- Data read (X pipe, Y pipe or FPU)
- Data write (X pipe, Y pipe or FPU).

2.7.1 Unified Cache MESI States

The unified cache lines are assigned one of four MESI states as determined by MESI bits stored in tag memory. Each 32-byte cache line is divided into two 16-byte sectors. Each sector contains its own MESI bits. The four MESI states are described below:

Modified MESI cache lines are those that have been updated by the CPU, but the corresponding main memory location has not yet been updated by an external write cycle. Modified cache lines are referred to as dirty cache lines.

Exclusive MESI lines are lines that are exclusive to the IBM 6x86 CPU and are not duplicated within another caching agent's cache within the same system. A write to this cache line may be performed without issuing an external write cycle.

Shared MESI lines may be present in another caching agent's cache within the same system. A write to this cache line forces a corresponding external write cycle.

Invalid MESI lines are cache lines that do not contain any valid data.

2.7.1.1 Unified Cache Testing

The unified cache can be tested through the use of TR3, TR4, and TR5 on-chip test registers. Fields within these test registers identify which area of the cache will be selected for testing.

Cache Organization. The unified cache (Figure 2-34) is divided into 32-bytes lines. This cache is divided into four sets. Since a set (as well as the cache) is smaller than main memory, each line in the set corresponds to more than one line in main memory. When a cache line is allocated, bits A31-A12 of the main memory address are stored in the cache line tag. The remaining address bits are used to identify the specific 32-byte cache line (A11-A5), and the specific 4-byte entry within the cache line (A4-A2).

Test Initiation. A test register operation is initiated by writing to the TR5 register shown in Figure 2-35 (Page 2-54) using a special MOV instruction. The TR5 CTL field, detailed in Table 2-28 (Page 2-54), determines the function to be performed. For cache writes, the registers TR4 and TR3 must be initialized before a write is made to TR5. Eight 4-byte accesses are required to access a complete cache line.

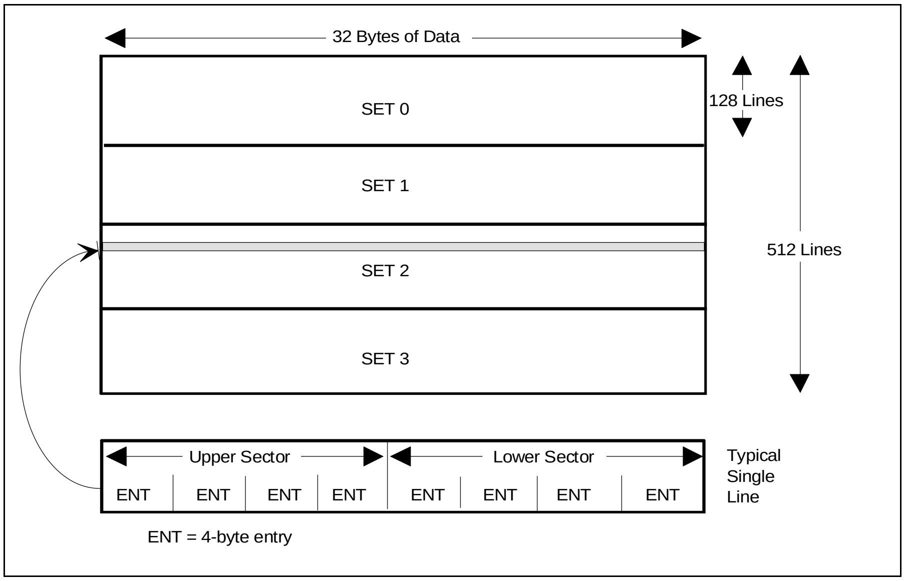

Figure 2-34. Unified Cache

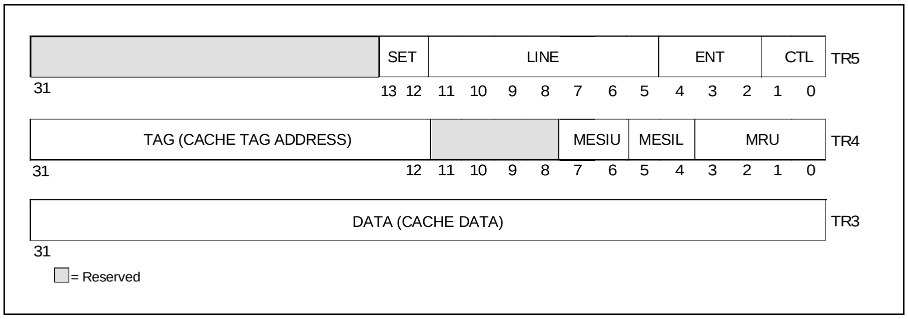

Figure 2-35. Cache Test Registers

Table 2-28. Cache Test Register Bit Definitions

REGISTER NAME	FIELD NAME	RANGE	DESCRIPTION
TR5	SET	13 - 12	Cache set selection (one of four “sets”).
	LINE	11 - 5	Cache line selection (one of 128 lines).
	ENT	4 - 2	Entry selection (one of eight 4-byte entries in a line).
	CTL	1 - 0	Control field If = 00: flush cache without invalidate If = 01: write cache If = 10: read cache If = 11: no cache or test register modification
TR4	TAG	31 - 12	Physical address for selected line

	MESIU	7 - 6	If = 00, Modified Upper Sector MESI bits
			If = 01, Shared Upper Sector MESI bits
			If = 10, Exclusive Upper Sector MESI bits
			If = 11, Invalid Upper Sector MESI bits*

	MESIL	5 - 4	If = 00, Modified Lower Sector MESI bits
			If = 01, Shared Lower Sector MESI bits
			If = 10, Exclusive Lower Sector MESI bits
			If = 11, Invalid Lower Sector MESI bits*

	MRU	3 - 0	Used to determine the Least Recently Used (LRU) line.
TR3	DATA	31 - 0	Data written or read during a cache test.

*Note: All 32 bytes should contain valid data before a line is marked as valid.

Write Operations. During a write, the TR3 DATA (32-bits) and TAG field information is written to the address selected by the SET, LINE, and ENT fields in TR5.

Read Operations. During a read, the cache address selected by the SET, LINE and ENT fields in TR5 are used to read data into the TR3 DATA (32-bits) field. The TAG, MESI and MRU fields in TR4 are updated with the information from the selected line. TR3 holds the selected read data.

Cache Flushing. A cache flush occurs during a TR5 write if the CTL field is set to zero. During flushing, the CPU's cache controller reads through all the lines in the cache. "Modified" lines are redefined as "shared" by setting the shared MESI bit. Clean lines are left in their original state.

2.8 Interrupts and Exceptions

The processing of either an interrupt or an exception changes the normal sequential flow of a program by transferring program control to a selected service routine. Except for SMM interrupts, the location of the selected service routine is determined by one of the interrupt vectors stored in the interrupt descriptor table.

Hardware interrupts are generated by signal sources external to the CPU. All exceptions (including so-called software interrupts) are produced internally by the CPU.

2.8.1 Interrupts

External events can interrupt normal program execution by using one of the three interrupt pins on the IBM 6x86 CPU.

- Non-maskable Interrupt (NMI pin)
- Maskable Interrupt (INTR pin)
- SMM Interrupt (SMI# pin).

For most interrupts, program transfer to the interrupt routine occurs after the current instruction has been completed. When the execution returns to the original program, it begins immediately following the last completed instruction.

With the exception of string operations, interrupts are acknowledged between instructions. Long string operations have interrupt windows between memory moves that allow interrupts to be acknowledged.

The **NMI interrupt** cannot be masked by software and always uses interrupt vector 2 to locate its service routine. Since the interrupt vector is fixed and is supplied internally, no interrupt acknowledge bus cycles are performed. This interrupt is normally reserved for unusual situations such as parity errors and has priority over INTR interrupts.

Once NMI processing has started, no additional NMIs are processed until an IRET instruction is executed, typically at the end of the NMI service routine. If NMI is re-asserted prior to execution of the IRET instruction, one and only one NMI rising edge is stored and processed after execution of the next IRET.

During the NMI service routine, maskable interrupts may be enabled (unmasked). If an unmasked INTR occurs during the NMI service routine, the INTR is serviced and execution returns to the NMI service routine following the next IRET. If a HALT instruction is executed within the NMI service routine, the IBM 6x86 CPU restarts execution only in response to RESET, an unmasked INTR or an SMM interrupt. NMI does not restart CPU execution under this condition.

The **INTR interrupt** is unmasked when the Interrupt Enable Flag (IF) in the EFLAGS register is set to 1. When an INTR interrupt occurs, the CPU performs two locked interrupt acknowledge bus cycles. During the second cycle, the CPU reads an 8-bit vector that is supplied by an external interrupt controller. This vector selects one of the 256 possible interrupt handlers which will be executed in response to the interrupt.

The **SMM interrupt** has higher priority than either INTR or NMI. After SMI# is asserted, program execution is passed to an SMI service routine that runs in SMM address space reserved for this purpose. The remainder of this section does not apply to the SMM interrupts. SMM interrupts are described in greater detail later in this chapter.

2.8.2 Exceptions

Exceptions are generated by an interrupt instruction or a program error. Exceptions are classified as traps, faults or aborts depending on the mechanism used to report them and the restartability of the instruction that first caused the exception.

A **Trap Exception** is reported immediately following the instruction that generated the trap exception. Trap exceptions are generated by execution of a software interrupt instruction (INTO, INT 3, INT n, BOUND), by a single-step operation or by a data breakpoint.

Software interrupts can be used to simulate hardware interrupts. For example, an INT n instruction causes the processor to execute the interrupt service routine pointed to by the nth vector in the interrupt table. Execution of the interrupt service routine occurs regardless of the state of the IF flag in the EFLAGS register.

The one byte INT 3, or breakpoint interrupt (vector 3), is a particular case of the INT n instruction. By inserting this one byte instruction in a program, the user can set breakpoints in the code that can be used during debug.

Single-step operation is enabled by setting the TF bit in the EFLAGS register. When TF is set, the CPU generates a debug exception (vector 1) after the execution of every instruction. Data breakpoints also generate a debug exception and are specified by loading the debug registers (DR0-DR7) with the appropriate values.

A **Fault Exception** is reported prior to completion of the instruction that generated the exception. By reporting the fault prior to instruction completion, the CPU is left in a state that allows the instruction to be restarted and the effects of the faulting instruction to be nullified. Fault exceptions include divide-by-zero errors, invalid opcodes, page faults and coprocessor errors. Instruction breakpoints (vector 1) are also handled as faults. After execution of the fault service routine, the instruction pointer points to the instruction that caused the fault.

An **Abort Exception** is a type of fault exception that is severe enough that the CPU cannot restart the program at the faulting instruction. The double fault (vector 8) is the only abort exception that occurs on the IBM 6x86 CPU.

2.8.3 Interrupt Vectors

When the CPU services an interrupt or exception, the current program's FLAGS, code segment and instruction pointer are pushed onto the stack to allow resumption of execution of the interrupted program. In protected mode, the processor also saves an error code for some exceptions. Program control is then transferred to the interrupt handler (also called the interrupt service routine). Upon execution of an IRET at the end of the service routine, program execution resumes by popping from the stack, the instruction pointer, code segment, and FLAGS.

Interrupt Vector Assignments

Each interrupt (except SMI#) and exception is assigned one of 256 interrupt vector numbers (Table 2-29). The first 32 interrupt vector assignments are defined or reserved. INT instructions acting as software interrupts may use any of the interrupt vectors, 0 through 255.

Table 2-29. Interrupt Vector Assignments

INTERRUPT VECTOR	FUNCTION	EXCEPTION TYPE
0	Divide error	FAULT
1	Debug exception	TRAP/FAULT*
2	NMI interrupt	
3	Breakpoint	TRAP
4	Interrupt on overflow	TRAP
5	BOUND range exceeded	FAULT
6	Invalid opcode	FAULT
7	Device not available	FAULT
8	Double fault	ABORT
9	Reserved	
10	Invalid TSS	FAULT
11	Segment not present	FAULT
12	Stack fault	FAULT
13	General protection fault	TRAP/FAULT
14	Page fault	FAULT
15	Reserved	
16	FPU error	FAULT
17	Alignment check exception	FAULT
18-31	Reserved	
32-255	Maskable hardware interrupts	TRAP
0-255	Programmed interrupt	TRAP

*Note: Data breakpoints and single-steps are traps. All other debug exceptions are faults.

In response to a maskable hardware interrupt (INTR), the IBM 6x86 CPU issues interrupt acknowledge bus cycles used to read the vector number from external hardware. These vectors should be in the range 32 - 255 as vectors 0 - 31 are reserved.

Interrupt Descriptor Table

The interrupt vector number is used by the IBM 6x86 CPU to locate an entry in the interrupt descriptor table (IDT). In real mode, each IDT entry consists of a four-byte far pointer to the beginning of the corresponding interrupt service routine. In protected mode, each IDT entry is an eight-byte descriptor. The Interrupt Descriptor Table Register (IDTR) specifies the beginning address and limit of the IDT. Following reset, the IDTR contains a base address of 0h with a limit of 3FFh.

The IDT can be located anywhere in physical memory as determined by the IDTR register. The IDT may contain different types of descriptors: interrupt gates, trap gates and task gates. Interrupt gates are used primarily to enter a hardware interrupt handler. Trap gates are generally used to enter an exception handler or software interrupt handler. If an interrupt gate is used, the Interrupt Enable Flag (IF) in the EFLAGS register is cleared before the interrupt handler is entered. Task gates are used to make the transition to a new task.

2.8.4 Interrupt and Exception Priorities

As the IBM 6x86™ CPU executes instructions, it follows a consistent policy for prioritizing exceptions and hardware interrupts. The priorities for competing interrupts and exceptions are listed in Table 2-30 (Page 2-60). Debug traps for the previous instruction and the next instructions always take precedence. SMM interrupts are the next priority. When NMI and maskable INTR interrupts are both detected at the same instruction boundary, the IBM 6x86 microprocessor services the NMI interrupt first.

The IBM 6x86 CPU checks for exceptions in parallel with instruction decoding and execution. Several exceptions can result from a single instruction. However, only one exception is generated upon each attempt to execute the instruction. Each exception service routine should make the appropriate corrections to the instruction and then restart the instruction. In this way, exceptions can be serviced until the instruction executes properly.

The IBM 6x86 CPU supports instruction restart after all faults, except when an instruction causes a task switch to a task whose task state segment (TSS) is partially not present. A TSS can be partially not present if the TSS is not page aligned and one of the pages where the TSS resides is not currently in memory.

Table 2-30. Interrupt and Exception Priorities

PRIORITY	DESCRIPTION	NOTES
0	Warm Reset	Caused by the assertion of WM_RST.
1	Debug traps and faults from previous instruction.	Includes single-step trap and data breakpoints specified in the debug registers.
2	Debug traps for next instruction.	Includes instruction execution breakpoints specified in the debug registers.
3	Hardware Cache Flush	Caused by the assertion of FLUSH#.
4	SMM hardware interrupt.	SMM interrupts are caused by SMI# asserted and always have highest priority.
5	Non-maskable hardware interrupt.	Caused by NMI asserted.
6	Maskable hardware interrupt.	Caused by INTR asserted and IF = 1.
7	Faults resulting from fetching the next instruction.	Includes segment not present, general protection fault and page fault.
8	Faults resulting from instruction decoding.	Includes illegal opcode, instruction too long, or privilege violation.
9	WAIT instruction and TS = 1 and MP = 1.	Device not available exception generated.
10	ESC instruction and EM = 1 or TS = 1.	Device not available exception generated.
11	Floating point error exception.	Caused by unmasked floating point exception with NE = 1.
12	Segmentation faults (for each memory reference required by the instruction) that prevent transferring the entire memory operand.	Includes segment not present, stack fault, and general protection fault.
13	Page Faults that prevent transferring the entire memory operand.	
14	Alignment check fault.	

2.8.5 Exceptions in Real Mode

Many of the exceptions described in Table 2-30 (Page 2-60) are not applicable in real mode. Exceptions 10, 11, and 14 do not occur in real mode. Other exceptions have slightly different meanings in real mode as listed in Table 2-31.

Table 2-31. Exception Changes in Real Mode

VECTOR NUMBER	PROTECTED MODE FUNCTION	REAL MODE FUNCTION
8	Double fault.	Interrupt table limit overrun.
10	Invalid TSS.	x
11	Segment not present.	x
12	Stack fault.	SS segment limit overrun.
13	General protection fault.	CS, DS, ES, FS, GS segment limit overrun.
14	Page fault.	x

Note: x = does not occur

2.8.6 Error Codes

When operating in protected mode, the following exceptions generate a 16-bit error code:

Double Fault	Invalid TSS
Alignment Check	Segment Not Present
Page Fault	Stack Fault
	General Protection Fault

The error code is pushed onto the stack prior to entering the exception handler. The error code format is shown in Figure 2-36 and the error code bit definitions are listed in Table 2-32. Bits 15-3 (selector index) are not meaningful if the error code was generated as the result of a page fault. The error code is always zero for double faults and alignment check exceptions.

15 3	2	1	0
Selector Index	S2	S1	S0

Figure 2-36. Error Code Format

Table 2-32. Error Code Bit Definitions

FAULT TYPE	SELECTOR INDEX (BITS 15-3)	S2 (BIT 2)	S1 (BIT 1)	S0 (BIT 0)
Double Fault or Alignment Check	0	0	0	0
Page Fault	Reserved.	Fault caused by: 0 = not present page 1 = page-level protection violation.	Fault occurred during: 0 = read access 1 = write access.	Fault occurred during: 0 = supervisor access 1 = user access.
IDT Fault	Index of faulty IDT selector.	Reserved.	1	If = 1, exception occurred while trying to invoke exception or hardware interrupt handler.
Segment Fault	Index of faulty selector.	TI bit of faulty selector.	0	If =1, exception occurred while trying to invoke exception or hardware interrupt handler.