2. PROGRAMMING INTERFACE

In this chapter, the internal operations of the IBM 6x86 CPU are described mainly from an application programmer's point of view. Included in this chapter are descriptions of processor initialization, the register set, memory addressing, various types of interrupts and the shutdown and halt process. An overview of real, virtual 8086, and protected operating modes is also included in this chapter. The FPU operations are described separately at the end of the chapter.

This manual does not—and is not intended to—describe the IBM 6x86 microprocessor or its operations at the circuit level.

2.1 Processor Initialization

The IBM 6x86 CPU is initialized when the RESET signal is asserted. The processor is placed in real mode and the registers listed in Table 2-1 (Page 2-2) are set to their initialized values. RESET invalidates and disables the cache and turns off paging. When RESET is asserted, the IBM 6x86 CPU terminates all local bus activity and all internal execution. During the entire time that RESET is asserted, the internal pipelines are flushed and no instruction execution or bus activity occurs.

Approximately 150 to 250 external clock cycles after RESET is negated, the processor begins executing instructions at the top of physical memory (address location FFFF FFF0h). Typically, an intersegment JUMP is placed at FFFF FFF0h. This instruction will force the processor to begin execution in the lowest 1 MByte of address space.

Note: The actual time depends on the clock scaling in use. Also an additional 2^{20} clock cycles are needed if self-test is requested.

Table 2-1. Initialized Register Controls

REGISTER	REGISTER NAME	INITIALIZED CONTENTS	COMMENTS
EAX	Accumulator	xxxx xxxxh	0000 0000h indicates self-test passed.
EBX	Base	xxxx xxxxh	
ECX	Count	xxxx xxxxh	
EDX	Data	05 + Device ID	Device ID = 31h or 33h (2X clock) Device ID = 35h or 37h (3X clock)
EBP	Base Pointer	xxxx xxxxh	
ESI	Source Index	xxxx xxxxh	
EDI	Destination Index	xxxx xxxxh	
ESP	Stack Pointer	xxxx xxxxh	
EFLAGS	Flag Word	0000 0002h	
EIP	Instruction Pointer	0000 FFF0h	
ES	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
CS	Code Segment	F000h	Base address set to FFFF 0000h. Limit set to FFFFh.
SS	Stack Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
DS	Data Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
FS	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
GS	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
IDTR	Interrupt Descriptor Table Register	Base = 0, Limit = 3FFh	
GDTR	Global Descriptor Table Register	xxxx xxxxh, xxxxh	
LDTR	Local Descriptor Table Register	xxxx xxxxh, xxxxh	
TR	Task Register	xxxxh	
CR0	Machine Status Word	6000 0010h	
CR2	Control Register 2	xxxx xxxxh	
CR3	Control Register 3	xxxx xxxxh	
CCR (0-5)	Configuration Control (0-5)	00h	
ARR (0-7)	Address Region Registers (0-7)	00h	
RCR (0-7)	Region Control Registers (0-7)	00h	
DIR0	Device Identification 0	31h or 33h (2X clock) 35h or 37h (3X clock)	
DIR1	Device Identification 1	Step ID + Revision ID	
DR7	Debug Register 7	0000 0400h	

Note: x = Undefined value

2.2 Instruction Set

Overview

The IBM 6x86 CPU instruction set performs nine types of general operations:

- Arithmetic
- Bit Manipulation
- Control Transfer
- Data Transfer
- Floating Point
- High-Level Language Support
- Operating System Support
- Shift/Rotate
- String Manipulation

All IBM 6x86 CPU instructions operate on as few as zero operands and as many as three operands. An NOP instruction (no operation) is an example of a zero operand instruction. Two operand instructions allow the specification of an explicit source and destination pair as part of the instruction. These two operand instructions can be divided into eight groups according to operand types:

- Register to Register
- Register to Memory
- Memory to Register
- Memory to Memory
- Register to I/O
- I/O to Register
- Immediate Data to Register
- Immediate Data to Memory

An operand can be held in the instruction itself (as in the case of an immediate operand), in one of the processor's registers or I/O ports, or in memory. An immediate operand is prefetched as part of the opcode for the instruction.

Operand lengths of 8, 16, or 32 bits are supported as well as 64-or 80-bit associated with floating point instructions. Operand lengths of 8 or 32 bits are generally used when executing code written for 386- or 486-class (32-bit code) processors. Operand lengths of 8 or 16 bits are generally used when executing existing 8086 or 80286 code (16-bit code). The default length of an operand can be overridden by placing one or more instruction prefixes in front of the opcode. For example, by using prefixes, a 32-bit operand can be used with 16-bit code, or a 16-bit operand can be used with 32-bit code.

Chapter 6 of this manual lists each instruction in the IBM 6x86 CPU instruction set along with the associated opcodes, execution clock counts, and effects on the FLAGS register.

2.2.1 Lock Prefix

The LOCK prefix may be placed before certain instructions that read, modify, then write back to memory. The prefix asserts the LOCK# signal to indicate to the external hardware that the CPU is in the process of running multiple indivisible memory accesses. The LOCK prefix can be used with the following instructions:

Bit Test Instructions (BTS, BTR, BTC)
Exchange Instructions (XADD, XCHG, CMPXCHG)
One-operand Arithmetic and Logical Instructions (DEC, INC, NEG, NOT)
Two-operand Arithmetic and Logical Instructions (ADC, ADD, AND, OR, SBB, SUB, XOR).

An invalid opcode exception is generated if the LOCK prefix is used with any other instruction, or with the above instructions when no write operation to memory occurs (i.e., the destination is a register). The LOCK# signal can be negated to allow weak-locking for all of memory or on a regional basis. Refer to the descriptions of the NO-LOCK bit (within CCR1) and the WL bit (within RCRx) later in this chapter.

2.3 Register Sets

From the programmer's point of view there are 58 accessible registers in the IBM 6x86 CPU. These registers are grouped into two sets. The application register set contains the registers frequently used by application programmers, and the system register set contains the registers typically reserved for use by operating system programmers.

The application register set is made up of general purpose registers, segment registers, a flag register, and an instruction pointer register.

The system register set is made up of the remaining registers which include control registers, system address registers, debug registers, configuration registers, and test registers.

Each of the registers is discussed in detail in the following sections.

2.3.1 Application Register Set

The application register set, (Figure 2-1, Page 2-5) consists of the registers most often used by the applications programmer. These registers are generally accessible and are not protected from read or write access.

The **General Purpose Register** contents are frequently modified by assembly language instructions and typically contain arithmetic and logical instruction operands.

Segment Registers in real mode contain the base address for each segment. In protected mode the segment registers contain segment selectors. The segment selectors provide indexing for tables (located in memory) that contain the base address and limit for each segment, as well as access control information.

The **Flag Register** contains control bits used to reflect the status of previously executed instructions. This register also contains control bits that affect the operation of some instructions.

The **Instruction Pointer** register points to the next instruction that the processor will execute. This register is automatically incremented by the processor as execution progresses.

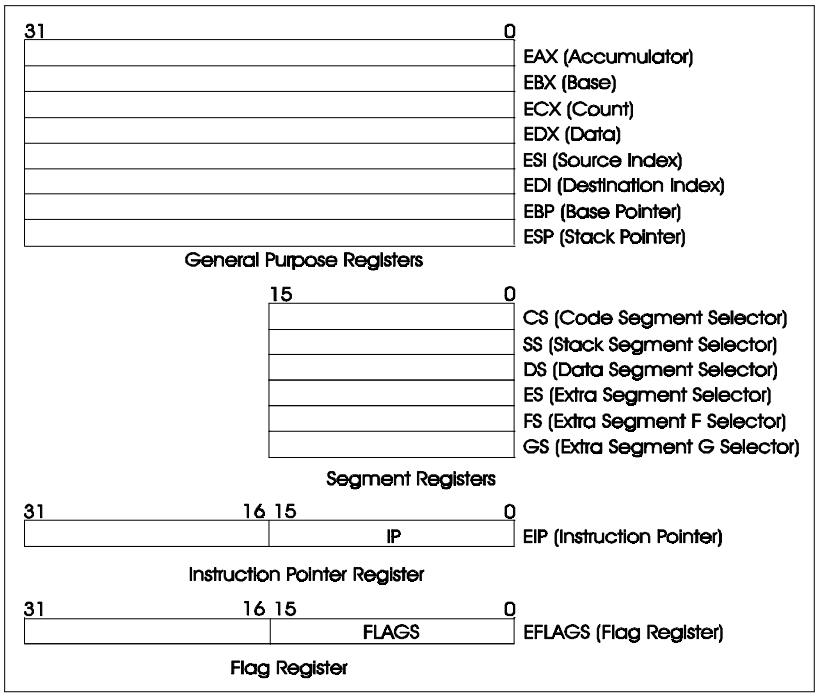

Figure 2-1. Application Register Set

2.3.2 General Purpose Registers

The general purpose registers are divided into four data registers, two pointer registers, and two index registers as shown in Figure 2-2 (Page 2-6).

The **Data Registers** are used by the applications programmer to manipulate data structures and to hold the results of logical and arithmetic operations. Different portions of the general data registers can be addressed by using different names.

An “E” prefix identifies the complete 32-bit register. An “X” suffix without the “E” prefix identifies the lower 16 bits of the register.

The lower two bytes of a data register can be addressed with an “H” suffix (identifies the upper byte) or an “L” suffix (identifies the lower byte). The _L and _H portions of a data registers act as independent registers. For example, if the AH register is written to by an instruction, the AL register bits remain unchanged.

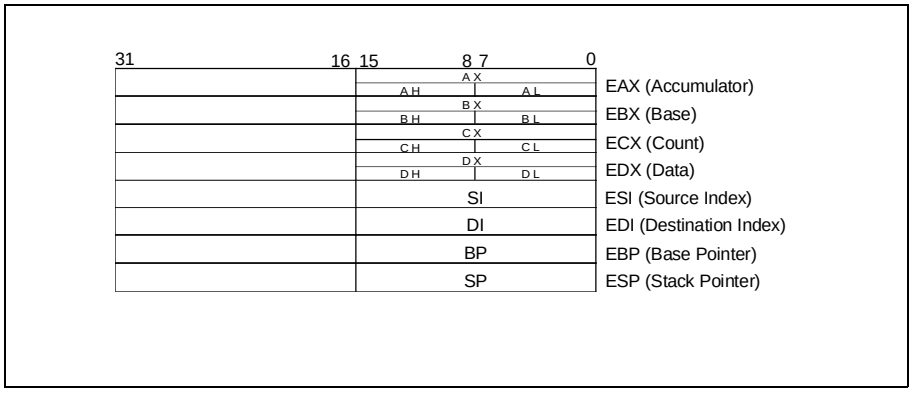

Figure 2-2. General Purpose Registers

The **Pointer and Index Registers** are listed below.

SI or ESI	Source Index
DI or EDI	Destination Index
SP or ESP	Stack Pointer
BP or EBP	Base Pointer

These registers can be addressed as 16- or 32-bit registers, with the “E” prefix indicating 32 bits. The pointer and index registers can be used as general purpose registers, however, some instructions use a fixed assignment of these registers. For example, repeated string operations always use ESI as the source pointer, EDI as the destination pointer, and ECX as the counter. The instructions using fixed registers include multiply and divide, I/O access, string operations, translate, loop, variable shift and rotate, and stack operations.

The IBM 6x86 CPU processor implements a stack using the ESP register. This stack is accessed during the PUSH and POP instructions, procedure calls, procedure returns, interrupts, exceptions, and interrupt/exception returns.

The microprocessor automatically adjusts the value of the ESP during operation of these instructions.The EBP register may be used to reference data passed on the stack during procedure calls. Local data may also be placed on the stack and referenced relative to BP. This register provides a mechanism to access stack data in high-level languages.

2.3.3 Segment Registers and Selectors

Segmentation provides a means of defining data structures inside the memory space of the microprocessor. There are three basic types of segments: code, data, and stack. Segments are used automatically by the processor to determine the location in memory of code, data, and stack references.

There are six 16-bit segment registers:

CS	Code Segment
DS	Data Segment
ES	Extra Segment
SS	Stack Segment
FS	Additional Data Segment
GS	Additional Data Segment.

In real and virtual 8086 operating modes, a segment register holds a 16-bit segment base. The 16-bit segment is multiplied by 16 and a 16-bit or 32-bit offset is then added to it to create a linear address. The offset size is dependent on the current address size. In real mode and in virtual 8086 mode with paging disabled, the linear address is also the physical address. In virtual 8086 mode with paging enabled, the linear address is translated to the physical address using the current page tables. Paging is described in Section 2.6.4 (Page 2-45).

In protected mode a segment register holds a **Segment Selector** containing a 13-bit index, a Table Indicator (TI) bit, and a two-bit Requested Privilege Level (RPL) field as shown in Figure 2-3.

The **Index** points into a descriptor table in memory and selects one of 8192 (2^{13}) segment descriptors contained in the descriptor table.

A segment descriptor is an eight-byte value used to describe a memory segment by defining the segment base, the segment limit, and access control information. To address data within a segment, a 16-bit or 32-bit offset is added to the segment's base address. Once a segment selector has been loaded into a segment register, an instruction needs only to specify the segment register and the offset.

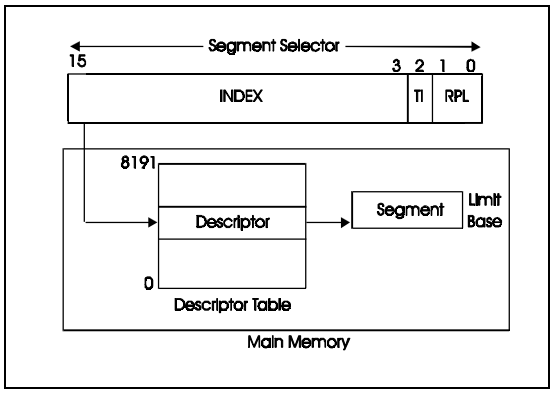

Figure 2-3. Segment Selector in Protected Mode

The **Table Indicator** (TI) bit of the selector defines which descriptor table the index points into. If TI=0, the index references the Global Descriptor Table (GDT). If TI=1, the index references the Local Descriptor Table (LDT). The GDT and LDT are described in more detail in Section 2.4.2. Protected mode addressing is discussed further in Sections 2.6.2 and 2.6.3.

The **Requested Privilege Level** (RPL) field in a segment selector is used to determine the Effective Privilege Level of an instruction (where RPL=0 indicates the most privileged level, and RPL=3 indicates the least privileged level).

If the level requested by RPL is less than the Current Program Level (CPL), the RPL level is accepted and the Effective Privilege Level is changed to the RPL value. If the level requested by RPL is greater than CPL, the CPL overrides the requested RPL and Effective Privilege Level remains unchanged.

When a segment register is loaded with a segment selector, the segment base, segment limit and access rights are loaded from the descriptor table entry into a user-invisible or hidden portion of the segment register (i.e., cached on-chip). The CPU does not access the descriptor table entry again until another segment register load occurs. If the descriptor tables are modified in memory, the segment registers must be reloaded with the new selector values by the software.

The processor automatically selects an implied (default) segment register for memory references. Table 2-2 describes the selection rules. In general, data references use the selector contained in the DS register, stack references use the SS register and instruction fetches use the CS register. While some of these selections may be overridden, instruction fetches, stack operations, and the destination write of string operations cannot be overridden. Special segment override instruction prefixes allow the use of alternate segment registers including the use of the ES, FS, and GS segment registers.

Table 2-2. Segment Register Selection Rules

TYPE OF MEMORY REFERENCE	IMPLIED (DEFAULT) SEGMENT	SEGMENT OVERRIDE PREFIX
Code Fetch	CS	None
Destination of PUSH, PUSHF, INT, CALL, PUSHA instructions	SS	None
Source of POP, POPA, POPF, IRET, RET instructions	SS	None
Destination of STOS, MOVS, REP STOS, REP MOVS instructions	ES	None
Other data references with effective address using base registers of:		
EAX, EBX, ECX, EDX, ESI, EDI	DS	CS, ES, FS, GS, SS
EBP, ESP	SS	CS, DS, ES, FS, GS

2.3.4 Instruction Pointer Register

The Instruction Pointer (EIP) register contains the offset into the current code segment of the next instruction to be executed. The register is normally incremented with each instruction execution unless implicitly modified through an interrupt, exception or an instruction that changes the sequential execution flow (e.g., JMP, CALL).

2.3.5 Flags Register

The Flags Register, EFLAGS, contains status information and controls certain operations on the IBM 6x86 CPU microprocessor. The lower 16 bits of this register are referred to as the FLAGS register that is used when executing 8086 or 80286 code. The flag bits are shown in Figure 2-4 and defined in Table 2-3 (Page 2-10).

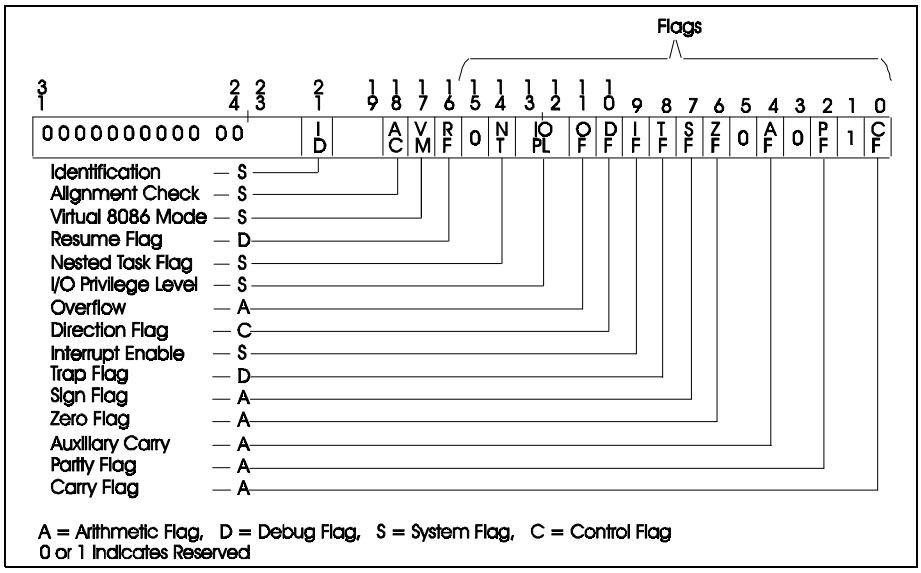

Figure 2-4. EFLAGS Register

Table 2-3. EFLAGS Bit Definitions

BIT POSITION	NAME	FUNCTION
0	CF	Carry Flag: Set when a carry out of (addition) or borrow into (subtraction) the most significant bit of the result occurs; cleared otherwise.
2	PF	Parity Flag: Set when the low-order 8 bits of the result contain an even number of ones; cleared otherwise.
4	AF	Auxiliary Carry Flag: Set when a carry out of (addition) or borrow into (subtraction) bit position 3 of the result occurs; cleared otherwise.
6	ZF	Zero Flag: Set if result is zero; cleared otherwise.
7	SF	Sign Flag: Set equal to high-order bit of result (0 indicates positive, 1 indicates negative).
8	TF	Trap Enable Flag: Once set, a single-step interrupt occurs after the next instruction completes execution. TF is cleared by the single-step interrupt.
9	IF	Interrupt Enable Flag: When set, maskable interrupts (INTR input pin) are acknowledged and serviced by the CPU.
10	DF	Direction Flag: If DF=0, string instructions auto-*in*crement (default) the appropriate index registers (ESI and/or EDI). If DF=1, string instructions auto-*de*crement the appropriate index registers.
11	OF	Overflow Flag: Set if the operation resulted in a carry or borrow into the sign bit of the result but did not result in a carry or borrow out of the high-order bit. Also set if the operation resulted in a carry or borrow out of the high-order bit but did not result in a carry or borrow into the sign bit of the result.
12, 13	IOPL	I/O Privilege Level: While executing in protected mode, IOPL indicates the maximum current privilege level (CPL) permitted to execute I/O instructions without generating an exception 13 fault or consulting the I/O permission bit map. IOPL also indicates the maximum CPL allowing alteration of the IF bit when new values are popped into the EFLAGS register.
14	NT	Nested Task: While executing in protected mode, NT indicates that the execution of the current task is nested within another task.
16	RF	Resume Flag: Used in conjunction with debug register breakpoints. RF is checked at instruction boundaries before breakpoint exception processing. If set, any debug fault is ignored on the next instruction.
17	VM	Virtual 8086 Mode: If set while in protected mode, the microprocessor switches to virtual 8086 operation handling segment loads as the 8086 does, but generating exception 13 faults on privileged opcodes. The VM bit can be set by the IRET instruction (if current privilege level=0) or by task switches at any privilege level.
18	AC	Alignment Check Enable: In conjunction with the AM flag in CR0, the AC flag determines whether or not misaligned accesses to memory cause a fault. If AC is set, alignment faults are enabled.
21	ID	Identification Bit: The ability to set and clear this bit indicates that the CPUID instruction is supported. The ID can be modified only if the CPUID bit in CCR4 is set.

2.4 System Register Set

The system register set, shown in Figure 2-5 (Page 2-12), consists of registers not generally used by application programmers. These registers are typically employed by system level programmers who generate operating systems and memory management programs.

The **Control Registers** control certain aspects of the IBM 6x86 microprocessor such as paging, coprocessor functions, and segment protection. When a paging exception occurs while paging is enabled, some control registers retain the linear address of the access that caused the exception.

The **Descriptor Table Registers** and the **Task Register** can also be referred to as system address or memory management registers. These registers consist of two 48-bit and two 16-bit registers. These registers specify the location of the data structures that control the segmentation used by the IBM 6x86 microprocessor. Segmentation is one available method of memory management.

The **Configuration Registers** are used to configure the IBM 6x86 CPU on-chip cache operation, power management features and System Management Mode. The configuration registers also provide information on the CPU device type and revision.

The **Debug Registers** provide debugging facilities to enable the use of data access breakpoints and code execution breakpoints.

The **Test Registers** provide a mechanism to test the contents of both the on-chip 16 KByte cache and the Translation Lookaside Buffer (TLB). In the following sections, the system register set is described in greater detail.

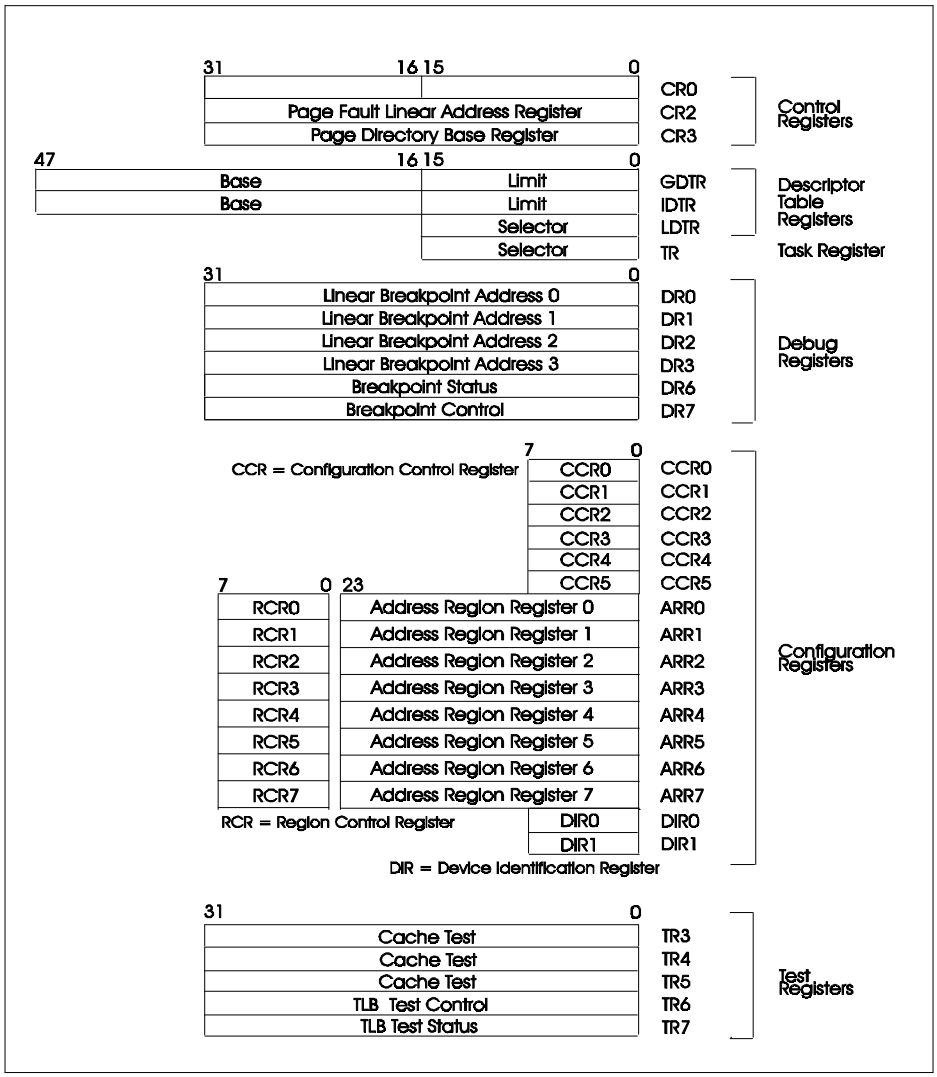

Figure 2-5. System Register Set

2.4.1 Control Registers

The Control Registers (CR0, CR2 and CR3), are shown in Figure 2-6. The CR0 register contains system control bits which configure operating modes and indicate the general state of the CPU. The lower 16 bits of CR0 are referred to as the Machine Status Word (MSW). The CR0 bit definitions are described in Table 2-4 and Table 2-5 (Page 2-14). The reserved bits in CR0 should not be modified.

When paging is enabled and a page fault is generated, the CR2 register retains the 32-bit linear address of the address that caused the fault. When a double page fault occurs, CR2 contains the address for the second fault. Register CR3 contains the 20 most significant bits of the physical base address of the page directory. The page directory must always be aligned to a 4-KByte page boundary, therefore, the lower 12 bits of CR3 are not required to specify the base address.

CR3 contains the Page Cache Disable (PCD) and Page Write Through (PWT) bits. During bus cycles that are not paged, the state of the PCD bit is reflected on the PCD pin and the PWT bit is driven on the PWT pin. These bus cycles include interrupt acknowledge cycles and all bus cycles, when paging is not enabled. The PCD pin should be used to control caching in an external cache. The PWT pin should be used to control write policy in an external cache.

Figure 2-6. Control Registers

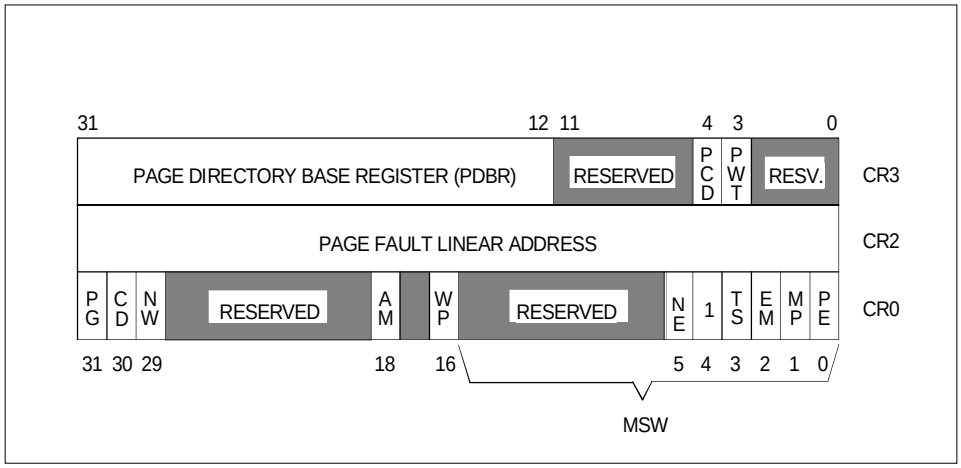

Table 2-4. CR0 Bit Definitions

Table 2-4. CR0 Bit Definitions

BIT POSITION	NAME	FUNCTION
1	MP	Monitor Processor Extension: If MP=1 and TS=1, a WAIT instruction causes Device Not Available (DNA) fault 7. The TS bit is set to 1 on task switches by the CPU. Floating point instructions are not affected by the state of the MP bit. The MP bit should be set to one during normal operations.
2	EM	Emulate Processor Extension: If EM=1, all floating point instructions cause a DNA fault 7.
3	TS	Task Switched: Set whenever a task switch operation is performed. Execution of a floating point instruction with TS=1 causes a DNA fault. If MP=1 and TS=1, a WAIT instruction also causes a DNA fault.
4	1	Reserved: Do not attempt to modify.
5	NE	Numerics Exception. NE=1 to allow FPU exceptions to be handled by interrupt 16. NE=0 if FPU exceptions are to be handled by external interrupts.
16	WP	Write Protect: Protects read-only pages from supervisor write access. WP=0 allows a read-only page to be written from privilege level 0-2. WP=1 forces a fault on a write to a read-only page from any privilege level.
18	AM	Alignment Check Mask: If AM=1, the AC bit in the EFLAGS register is unmasked and allowed to enable alignment check faults. Setting AM=0 prevents AC faults from occurring.
29	NW	Not Write-Back: If NW=1, the on-chip cache operates in write-through mode. In write-through mode, all writes (including cache hits) are issued to the external bus. If NW=0, the on-chip cache operates in write-back mode. In write-back mode, writes are issued to the external bus only for a cache miss, a line replacement of a modified line, or as the result of a cache inquiry cycle.
30	CD	Cache Disable: If CD=1, no further cache line fills occur. However, data already present in the cache continues to be used if the requested address hits in the cache. Writes continue to update the cache and cache invalidations due to inquiry cycles occur normally. The cache must also be invalidated to completely disable any cache activity.
31	PG	Paging Enable Bit: If PG=1 and protected mode is enabled (PE=1), paging is enabled. After changing the state of PG, software must execute an unconditional branch instruction (e.g., JMP, CALL) to have the change take effect.

Table 2-5. Effects of Various Combinations of EM, TS, and MP Bits

CR0 BIT			INSTRUCTION TYPE	
EM	TS	MP	WAIT	ESC
0	0	0	Execute	Execute
0	0	1	Execute	Execute
0	1	0	Execute	Fault 7
0	1	1	Fault 7	Fault 7
1	0	0	Execute	Fault 7
1	0	1	Execute	Fault 7
1	1	0	Execute	Fault 7
1	1	1	Fault 7	Fault 7

2.4.2 Descriptor Table

Registers and Descriptors

Descriptor Table Registers

The Global, Interrupt, and Local Descriptor Table Registers (GDTR, IDTR and LDTR), shown in Figure 2-7, are used to specify the location of the data structures that control segmented memory management. The GDTR, IDTR and LDTR are loaded using the LGDT, LIDT and LLDT instructions, respectively. The values of these registers are stored using the corresponding store instructions. The GDTR and IDTR load instructions are privileged instructions when operating in protected mode. The LDTR can only be accessed in protected mode.

The **Global Descriptor Table Register** (GDTR) holds a 32-bit linear base address and 16-bit limit for the Global Descriptor Table (GDT). The GDT is an array of up to 8192 8-byte descriptors. When a segment register is loaded from memory, the TI bit in the segment selector chooses either the GDT or the Local Descriptor Table (LDT) to locate a descriptor. If TI = 0, the index portion of the selector is used to locate the descriptor within the GDT table. The contents of the GDTR are completely visible to the programmer by using a SGDT instruction. The first descriptor in the GDT (location 0) is not used by the CPU and is referred to as the "null descriptor". The GDTR is initialized using a LGDT instruction.

The **Interrupt Descriptor Table Register** (IDTR) holds a 32-bit linear base address and 16-bit limit for the Interrupt Descriptor Table (IDT). The IDT is an array of 256 interrupt descriptors, each of which is used to point to an interrupt service routine. Every interrupt that may occur in the system must have an associated entry in the IDT. The contents of the IDTR are completely visible to the programmer by using a SIDT instruction. The IDTR is initialized using the LIDT instruction.

The **Local Descriptor Table Register** (LDTR) holds a 16-bit selector for the Local Descriptor Table (LDT). The LDT is an array of up to 8192 8-byte descriptors. When the LDTR is loaded, the LDTR selector indexes an LDT descriptor that must reside in the Global Descriptor Table (GDT). The base address and limit are loaded automatically and cached from the LDT descriptor within the GDT.

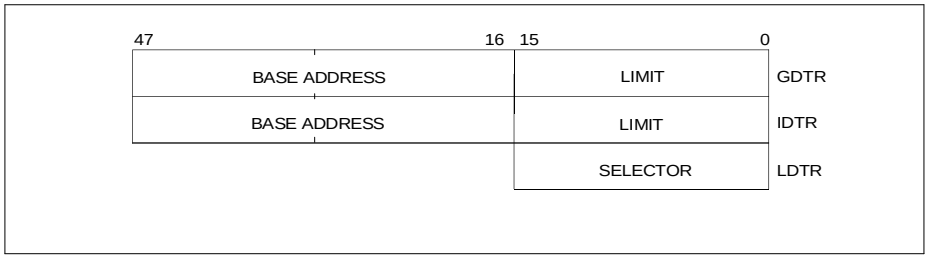

Figure 2-7. Descriptor Table Registers

Subsequent access to entries in the LDT use the hidden LDTR cache to obtain linear addresses. If the LDT descriptor is modified in the GDT, the LDTR must be reloaded to update the hidden portion of the LDTR.

When a segment register is loaded from memory, the TI bit in the segment selector chooses either the GDT or the LDT to locate a segment descriptor. If TI = 1, the index portion of the selector is used to locate a given descriptor within the LDT. Each task in the system may be given its own LDT, managed by the operating system. The LDTs provide a method of isolating a given task's segments from other tasks in the system.

The LDTR can be read or written by the LLDT and SLDT instructions.

Descriptors

There are three types of descriptors:

- Application Segment Descriptors that define code, data and stack segments.
- System Segment Descriptors that define an LDT segment or a Task State Segment (TSS) table described later in this text.
- Gate Descriptors that define task gates, interrupt gates, trap gates and call gates.

Application Segment Descriptors can be located in either the LDT or GDT. System Segment Descriptors can only be located in the GDT. Dependent on the gate type, gate descriptors may be located in either the GDT, LDT or IDT. Figure 2-8 illustrates the descriptor format for both Application Segment Descriptors and System Segment Descriptors. Table 2-6 (Page 2-17) lists the corresponding bit definitions.

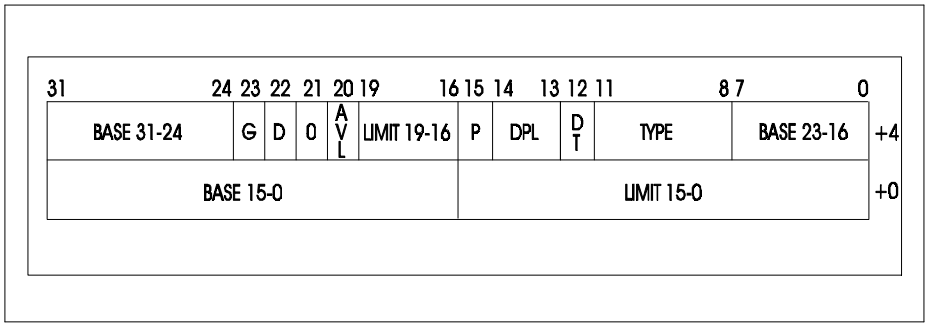

Figure 2-8. Application and System Segment Descriptors

Table 2-6. Segment Descriptor Bit Definitions

BIT POSITION	MEMORY OFFSET	NAME	DESCRIPTION
31-24 7-0 31-16	+4 +4 +0	BASE	Segment base address. 32-bit linear address that points to the beginning of the segment.
19-16 15-0	+4 +0	LIMIT	Segment limit.
23	+4	G	Limit granularity bit: 0 = byte granularity, 1 = 4 KBytes (page) granularity.
22	+4	D	Default length for operands and effective addresses. Valid for code and stack segments only: 0 = 16 bit, 1 = 32-bit.
20	+4	AVL	Segment available.
15	+4	P	Segment present.
14-13	+4	DPL	Descriptor privilege level.
12	+4	DT	Descriptor type: 0 = system, 1 = application.
11-8	+4	TYPE	Segment type. See Tables 2-7 and 2-8.

Table 2-7. TYPE Field Definitions with DT = 0

TYPE (BITS 11-8)	DESCRIPTION
0001	TSS-16 descriptor, task not busy.
0010	LDT descriptor.
0011	TSS-16 descriptor, task busy.
1001	TSS-32 descriptor, task not busy
1011	TSS-32 descriptor, task busy.

Table 2-8. TYPE Field Definitions with DT = 1

TYPE				APPLICATION DECRIPTOR INFORMATION
E	C/D	R/W	A	
0	0	x	x	data, expand up, limit is upper bound of segment
0	1	x	x	data, expand down, limit is lower bound of segment
1	0	x	x	executable, non-conforming
1	1	x	x	executable, conforming (runs at privilege level of calling procedure)
0	x	0	x	data, non-writable
0	x	1	x	data, writable
1	x	0	x	executable, non-readable
1	x	1	x	executable, readable
x	x	x	0	not-accessed
x	x	x	1	accessed