ST486 CORE

Standard 486 Processor Core

FEATURES

- INDUSTRY STANDARD 486 COMPATIBILITY
- ON-CHIP FPU
- ON-CHIP 8KBYTE WRITE BACK L1 CACHE
- DX / DX2 MODE OF OPERATION
- ADVANCED POWER MANAGEMENT

1.1 DESCRIPTION

The ST486 CPU is an advanced 486DX/DX2 compatible processor. It incorporates an on-chip 8-KByte write-back cache and an integrated math coprocessor.

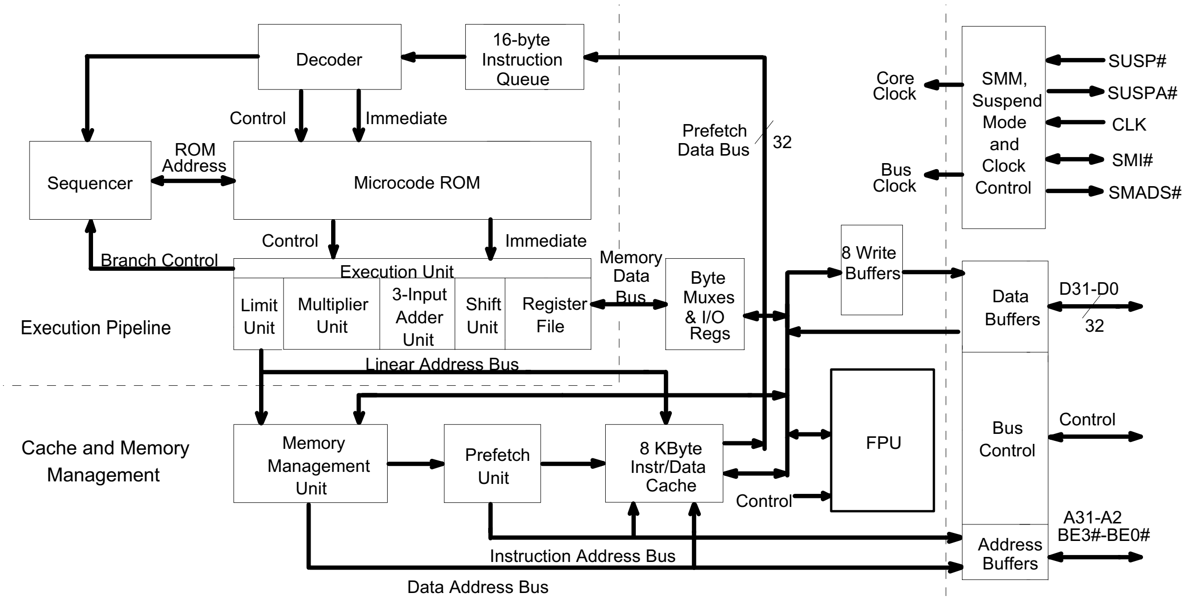
The on-chip write-back cache allows up to 15% higher performance by eliminating unnecessary external write cycles. On traditonnal write-trough CPUs, these external write cycles can create bus bottlenecks affecting system-wide performance.

The integrated Floating Point Unit, based on ST's FasMath architecture, improves performance up to 10% over the 80486DX as measured using Power Meter Whetstone test.

This processor is designed to meet the power management requirements in the newest generation of low-power desktops and notebooks. Power is saved not only by usinglow power supply, but by taking advantage of advanced power management features such as static circuitry, SMM, and automatic FPU power-down. Fast entry and exit of SMM allows frequent use of the SMM feature without noticeable performance degradation.

This CPU maintains compatibility with the installed base of x86 software.

Figure 1-1. Internal architecture.



HOW TO USE THIS MANUAL

HOW TO USE THIS MANUAL

2.3 INTRODUCTION

This manual provides full technical documentation for the ST486 CPU Core. It is recommended that the reader is familiar with the x86 series processors and PC compatible architectures before reading this document. Many terms are used which related directly to the PC architecture.

The manual itself is split into chapters. These chapters hold the information for a particular functional block of the ST486 Core.

2.4 SPECIFIC NOTES

2.4.1 RESERVED BITS

Many bits in the register descriptions are noted as reserved. These bits are not internally connected, physically not present or are used for testing purposes. In all cases these bits should be set to a ‘0’ when writing to a register with reserved bits. When reading from a register with reserved bits, these specific bits should be masked from the data value before action is taken on the data.

Any functionality found by setting the reserved bits to levels other than ‘0’ cannot and will not be guaranteed on future revisions of the circuit design. Thus it is not recommended to use the bits marked as reserved in any way different from noted above.

2.4.2 SIGNAL ACTIVE STATE

The pound symbol (#) following a signal name indicates that when the signal is in its active (asserted) state, the signal is at a logic low level. When the “#” is not present at the end of a signal name, the logic high level represents the active state.

2.4.3 HEX NOTATION

In this manual Hexadecimal (Hex) numbers (numbers to the base 16: [0-9,A-F]) are denoted by the postfix ‘h’.

For example a memory address 7830A hexadecimal will be written 783Ah.

2.4.4 ENDIAN

In common with the x86 architecture, values in memory are little-endian, that is the lower part of the memory contains the least significant byte.

For an 8-bit value

N	7	6	5	4	3	2	1	0

For a 16-bit (word) value

N	7	6	5	4	3	2	1	0
N+1	15	14	13	12	11	10	9	8

For a 24-bit value

N	7	6	5	4	3	2	1	0
N+1	15	14	13	12	11	10	9	8
N+2	23	22	21	20	19	18	17	16

For a 32-bit (long word) value

N	7	6	5	4	3	2	1	0
N+1	15	14	13	12	11	10	9	8
N+2	23	22	21	20	19	18	17	16
N+3	31	30	29	28	27	26	25	24

For a 64-bit (QUAD word) value

N	7	6	5	4	3	2	1	0
N+1	15	14	13	12	11	10	9	8
N+2	23	22	21	20	19	18	17	16
N+3	31	30	29	28	27	26	25	24
N+4	39	38	37	36	35	34	33	32
N+5	47	46	45	44	43	42	41	40
N+6	55	54	53	52	51	50	49	48
N+6	63	62	61	60	59	58	57	56

486 CORE

486 CORE

3.1 Introduction

In this chapter, the internal operations of the 486 core 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. Included is an overview of real, virtual 8086, and protected operating modes. The FPU operations are described separately at the end of this chapter.

3.2 Processor Initialization

The 486 core is initialized when the RESET signal is asserted. The processor is placed in real mode and the registers listed in Table 3-1 are set to their initialized values. RESET invalidates and disables the 486 core cache, and turns off paging. When RESET is asserted, the 486 core terminates all local bus activity and all internal execution. During the entire time that RESET is asserted, the internal pipeline is flushed and no instruction execution or bus activity occurs.

Approximately 150 to 250 external clock cycles (additional 2^{20} + 60 if self-test is requested) after RESET is negated, the processor begins executing instructions at the top of physical memory (address location FFFF FFF0h). When the first intersegment JUMP or CALL is executed, memory address lines A31-A20 are driven low for code segment-relative memory access cycles. While A31-A20 are low, the 486 core executes instructions only in the lowest 1 MByte of physical address space until system-specific initialization occurs via program execution.

Table 3-1. Initialized Core 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	xxxx 0400 + Device ID	Device ID = 80h.
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	
CR0	Machine Status Word	6000 0010h	
CCR1	Configuration Control 1	00h	
CCR2	Configuration Control 2	00h	
CCR3	Configuration Control 3	00h	
SMAR	SMM Address Region	0000h	
DIR0	Device Identification 0	586DX = 1Ah	
DIR1	Device Identification 1	Step ID + Revision ID	
DR7	Debug Register 7	0000 0400h	
Note: x = Undefined value			

3.3 Instruction Set Overview

The 486 core instruction set can be divided into eight types of operations:

– Arithmetic

– Bit Manipulation

– Control Transfer

– Data Transfer

– Floating Point

– High-Level Language Support

– Operating System Support

– Shift/Rotate

– String Manipulation.

All 486 core instructions operate on as few as 0 operands and as many as 3 operands. An NOP instruction (no operation) is an example of a 0 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 a register, in an I/O port 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 12 of this manual lists each instruction in the 486 core instruction set along with the associated opcodes, execution clock counts and effects on the FLAGS register.

3.3.1 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 prefix function may be disabled by setting the NO_LOCK bit in Configuration Control Register 1 (CCR1).

3.4 Register Set

There are 40 accessible registers in the 486 core and 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 systems programmers.

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

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

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

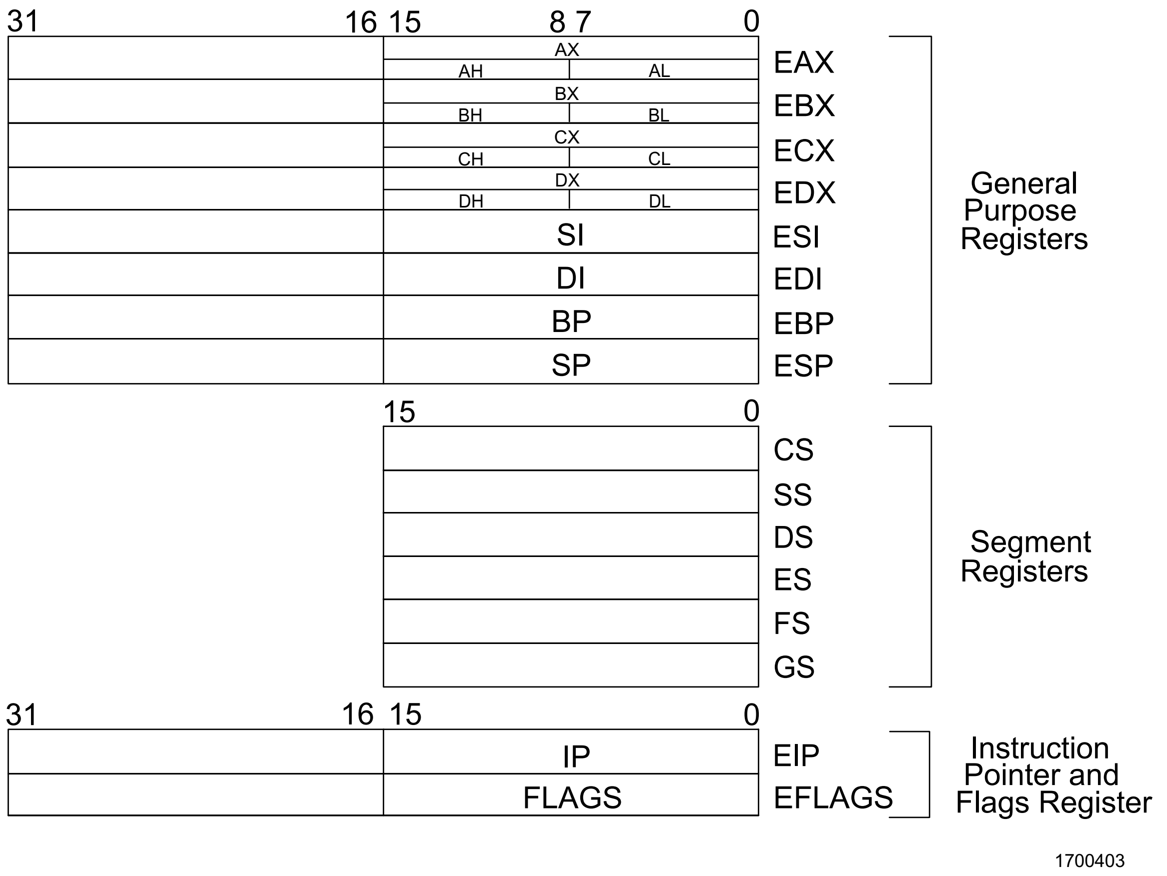
3.4.1 Application Register Set

The application register set, Figure 3-1, 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.

Figure 3-1. Application Register Set



The **Segment Register** in real mode contains 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 for each segment, as well as other memory addressing information.

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

3.4.2 General Purpose Registers

The general purpose registers are divided into four data registers, two pointer registers, and two index registers as shown in Figure 3-2.

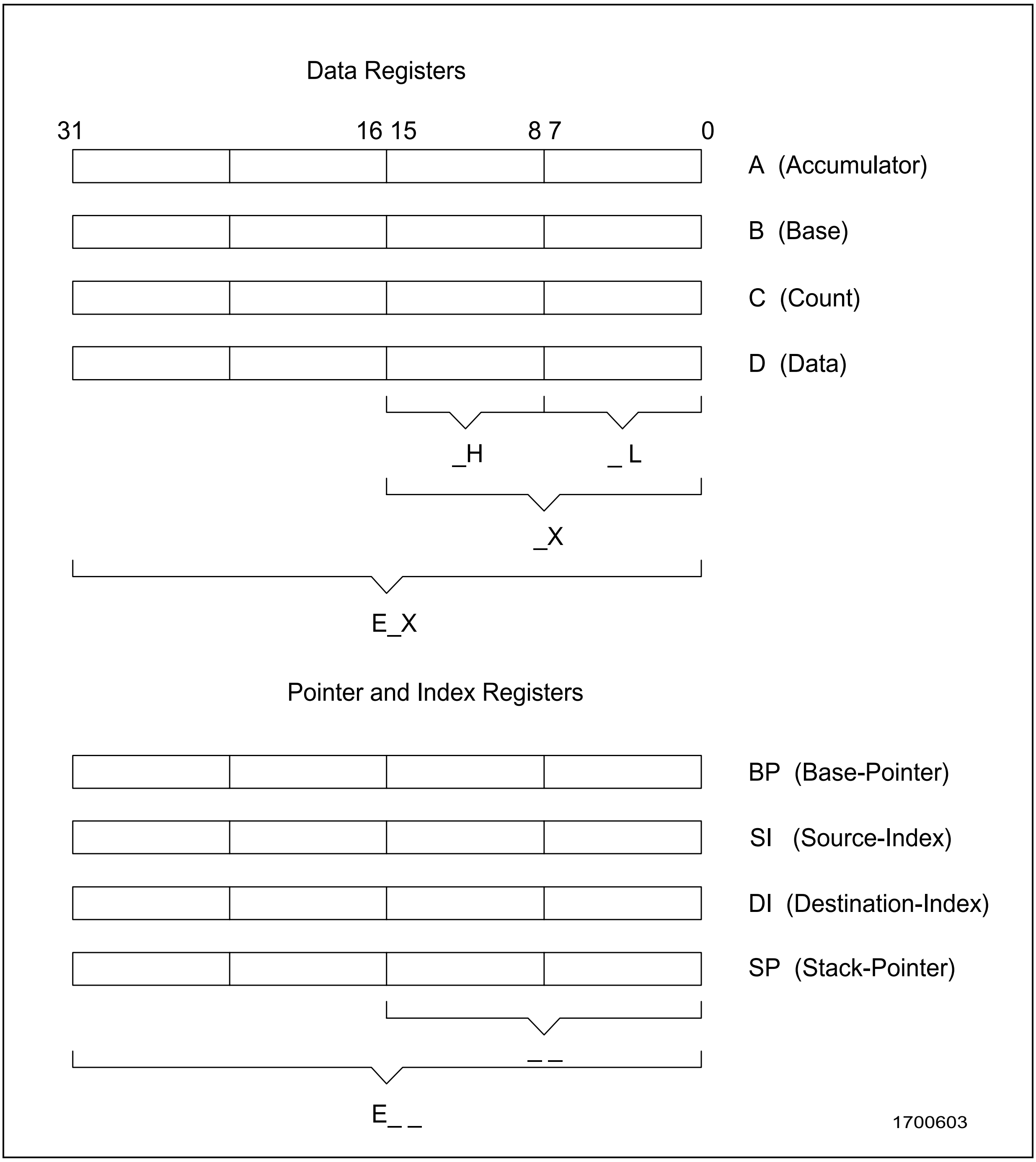
The **Data Registers** are used by the applications programmer to manipulate data structures and to hold the results of logical and arithmetic operations. Different portions of the general data registers can be addressed by using different names. An “E” prefix identifies the complete 32-bit register. An “X” suffix without the “E” prefix identifies the lower 16 bits of the register. The lower two bytes of the register can be addressed with an “H” suffix to identify the upper byte or an “L” suffix to identify the lower byte. When a destination operand size specified by an instruction is smaller than the specified destination register, the other bytes of the destination register are not affected when the operand is written to the register.

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 a counter. The instructions using fixed registers include multiply and divide, I/O access, string operations, translate, loop, variable shift and rotate, and stack operations instructions. The 486 core 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.

Figure 3-2. General Purpose Registers



3.4.3 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 base 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.

3.4.3.1 Segment Selector Register

In protected mode, a 16_bit 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.

Bits 15-3 **Index.** These bits point 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 to specify the offset only.

Bit 2 **TI**, **Table Indicator.** This 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 later in this chapter.

Bits 1-0 **RPL**, **Requested Privilege Level.** This field contains a 2-bit segment privilege level (0 = most privileged, 3 = least privileged). The RPL bits are used when the segment register is loaded to determine the Effective Privilege Level (EPL). If the RPL bits indicate less privilege than the Current Program Level (CPL), the RPL overrides the CPL and the EPL is the less privileged level. If the RPL bits indicate more privilege than the program, the CPL overrides the RPL and again the EPL is the less privileged level.

When a segment register is loaded with a segment selector, the segment base, segment limit and access rights are also loaded from the descriptor table into a user-invisible or hidden portion of the segment register, i.e., cached on-chip. The CPU does not access the descriptor table 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 a default segment register for memory references. Table 3-2 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 prefixes allow the use of alternate segment registers including the use of the ES, FS, and GS segment registers.

Table 3-2. Segment Register Selection Rules

Type of Memory References	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, EBP, ESP	DS SS	CS, ES, FS, GS, SS CS, ES, FS, GS

3.4.3.2 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., jump, call).

3.4.3.3 Flags Register

The **Flags Register**, EFLAGS, contains status information and controls certain operations on the 486 core 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 3-3.

Bits 31-19 *Reserved.* These bits should be set to '0'

Bit 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.

Bit 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.

Bit 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.

Bit 15 *Reserved.* This bit should be set to '0'

Bit 14 **NT, Nested Task.** While executing in protected mode, NT indicates that the execution of the current task is nested within another task.

Bits 13-12, **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.

Bit 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.

Bit 10 **DF, Direction Flag.** When cleared, DF causes string instructions to auto-increment (default) the appropriate index registers (ESI and/or EDI). Setting DF causes auto-decrement of the index registers to occur.

Bit 9 **IF, Interrupt Enable Flag.** When set, maskable interrupts (INTR input pin) are acknowledged and serviced by the CPU.

Bit 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.

Bit 7 **SF, Sign Flag.** Set equal to high-order bit of result (0 indicates positive, 1 indicates negative).

Bit 6 **ZF, Zero Flag.** Set if result is zero; cleared otherwise.

Bit 5 *Reserved.* This bit should be set to '0'

Bit 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.

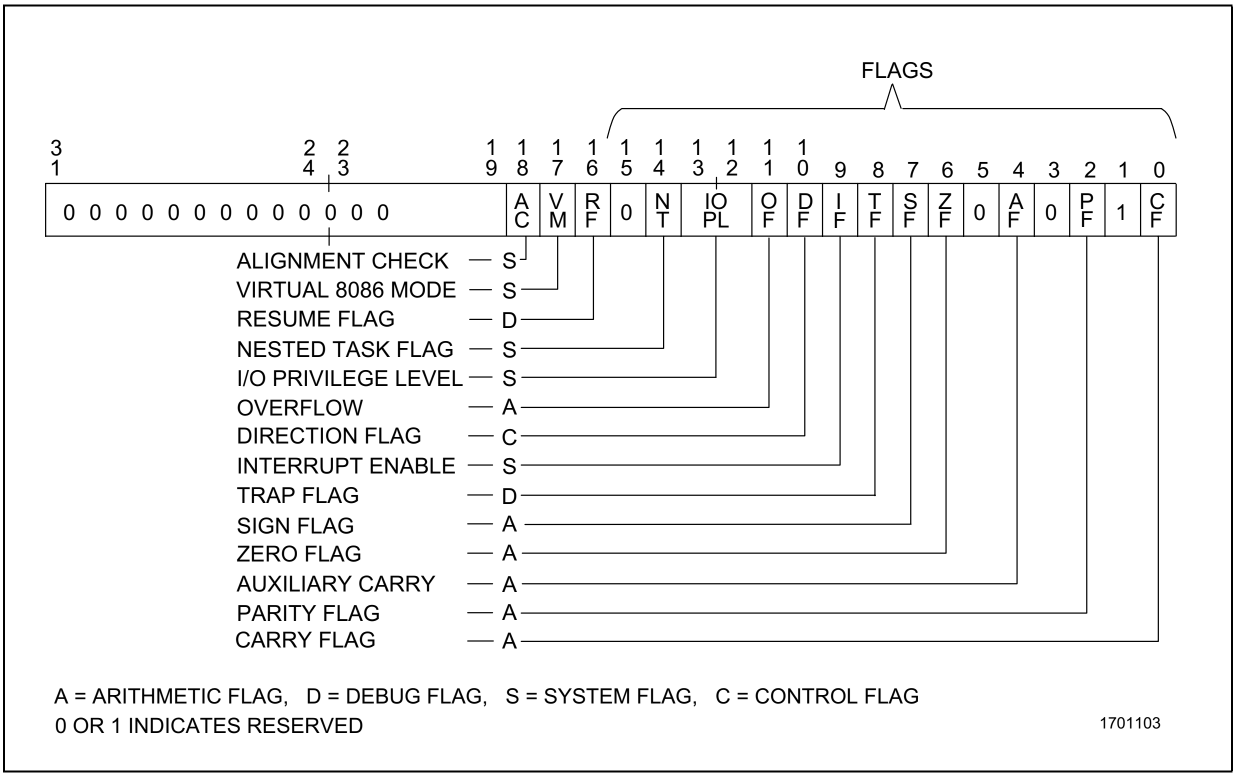
Bit 3 *Reserved.* This bit should be set to '0'

Bit 2 **PF, Parity Flag.** Set when the low-order 8 bits of the result contain an even number of ones; cleared otherwise.

Bit 1 *Reserved.* This bit should be set to '1'

Bit 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.

Figure 3-3. EFLAGS Register



3.4.4 System Register Set

The system register set, shown in Figure 3-4, 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 486 core microprocessor such as paging, coprocessor functions, and segment protection. When a paging exception occurs while paging is enabled, the 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 486 core microprocessor. Segmentation is one available method of memory management.

The **Configuration Registers** are used to configure the 486 core on-chip cache operation, coprocessor interface, 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 for the 486 core microprocessor and 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 8 KByte cache and the Translation Lookaside Buffer (TLB). The TLB is used as a cache for the tables which are used in translating linear addresses to physical addresses while paging is enabled. In the following sections, the system register set is described in greater detail.

3.4.4.1 Control Registers

The Control Registers (CR0, CR2 and CR3), are shown in Figure 3-5.

When operating in protected mode, any program can read the control registers. However, only privilege level 0 (most privileged) programs can modify the contents of these registers.

CONTROL REGISTER CR0

The CR0 register contains system control flags which control 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 effects of combinations of EM, TS, MP bits of CR0 are described in Table 3-3. The reserved bits in CR0 should not be modified.

Bit 31 **PG, Paging Enable Bit.** If PG='1' and protected mode is enabled (PE='1'), paging is enabled.

Bit 30 **CD, Cache Disable.** If CD='1', no further cache fills occur. However, data already present in the cache continues to be used if the requested address hits in the cache. The cache must also be invalidated to completely disable any cache activity.

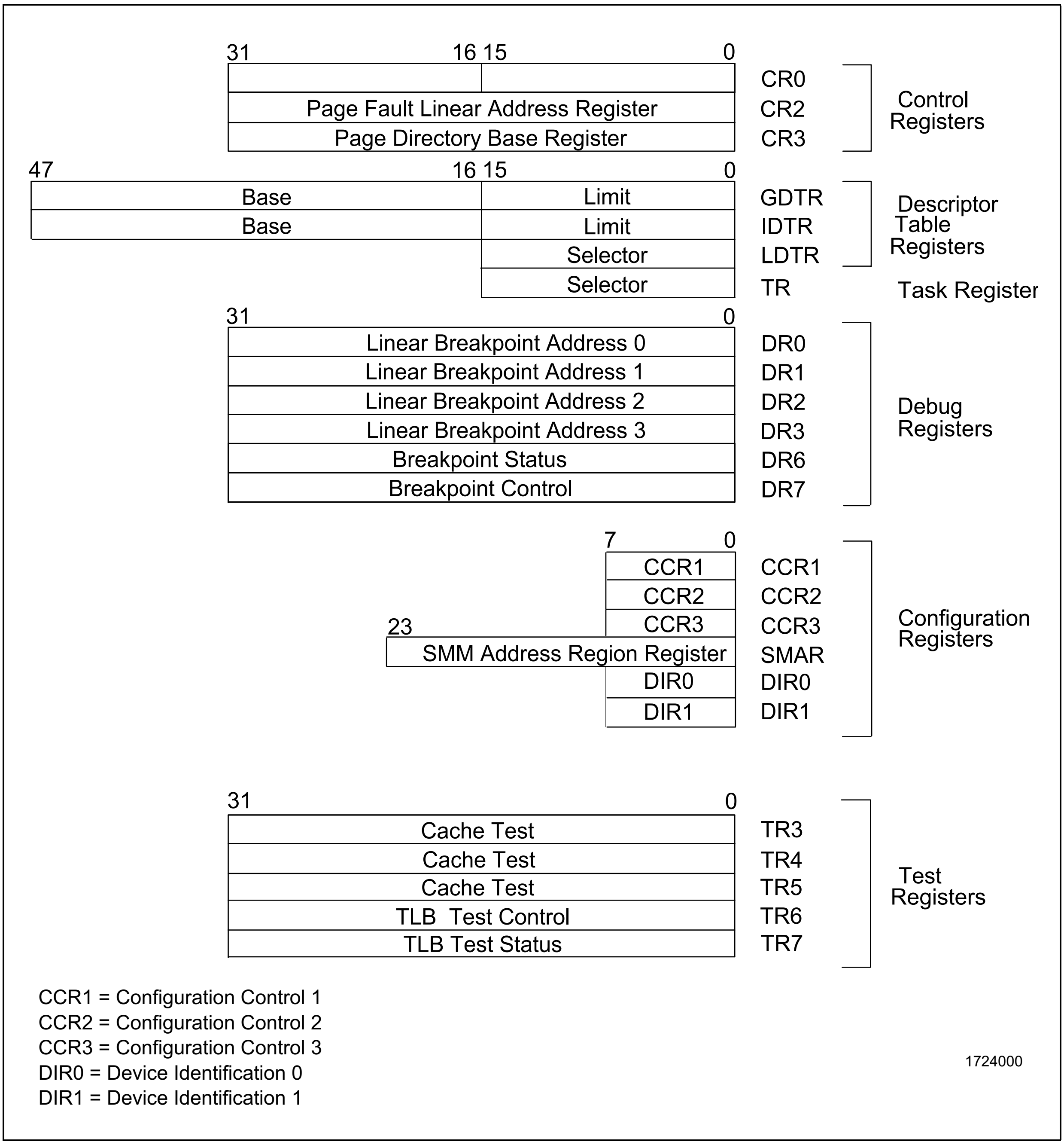
Bit 29 **NW, Not Write-Through.** If NW='1', the on-chip cache operates in write-back mode. In write-back mode, writes are issued to the external bus only for a cache miss, a line replacement of a modified line, or as the result of a cache inquiry cycle. If NW='0', the on-chip cache operates in write-through mode. In write-through mode, all writes (including cache hits) are issued to the external bus.

Bits 28-19 *Reserved.* Do not attempt to modify.

Bit 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.

Bits 17 *Reserved.* Do not attempt to modify.

Figure 3-4. System Register Set



CCR1 = Configuration Control 1
CCR2 = Configuration Control 2
CCR3 = Configuration Control 3
DIR0 = Device Identification 0
DIR1 = Device Identification 1

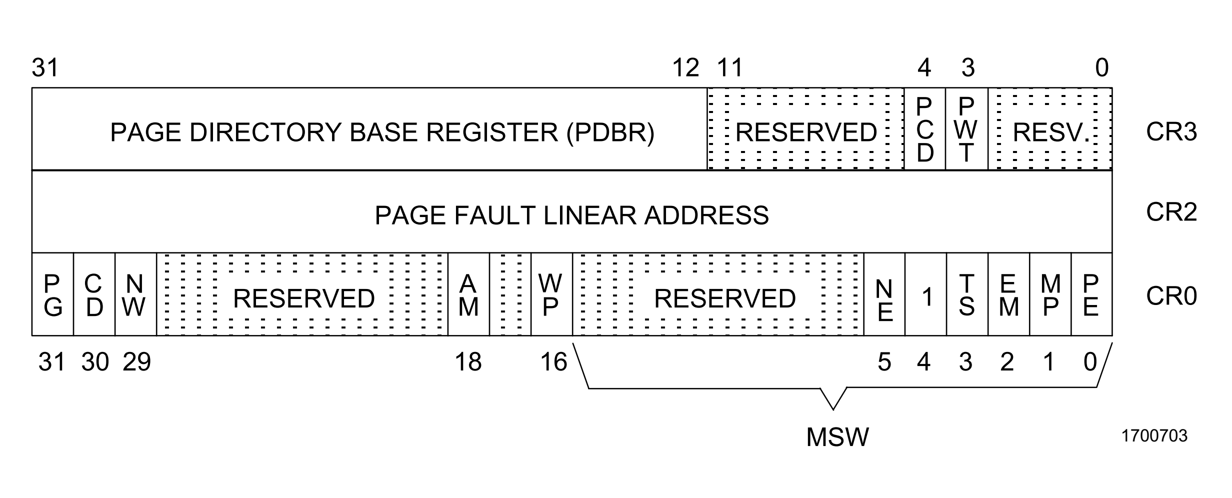
Bit 16 **WP, Write Protect.** Protects read-only pages from supervisor write access. WP='0' allows a read-only page to be written from privilege level 0-2. WP='1' forces a fault on a write to a read-only page from any privilege level.

Bits 15-6 *Reserved.* Do not attempt to modify. This bit must be set to ‘1’.

Bit 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.

Bit 4 *Reserved.* Do not attempt to modify. This bit must be maintained at '1'

Figure 3-5. Control Registers



Bit 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.

Bit 2 **EM, Emulate Processor Extension.** If EM='1', all floating point instructions cause a DNA fault 7.

Bit 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.

Bit 0 **PE, Protected Mode Enable.** Enables the segment based protection mechanism. If PE='1', protected mode is enabled. If PE='0', the CPU operates in real mode, with segment based protection disabled, and addresses are formed as in an 8086-class CPU.

Table 3-3. Effects of combinations of EM, TS, 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

CONTROL REGISTER CR2

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.

Bits 31-0 **Page Fault Linear Address.**

CONTROL REGISTER CR3

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.

Bits 31-12 **PDBR, Page Directory Base.**

Bits 11-5 *Reserved.*

Bit 4 **PCD.**

Bit 3 **PWT.**

Bits 3-0 *Reserved.*

3.4.4.2 Descriptor Table Registers and Descriptors

Descriptor Table Registers

The Global, Interrupt and Local Descriptor Table Registers (GDTR, IDTR and LDTR), shown in Figure 3-6, 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.

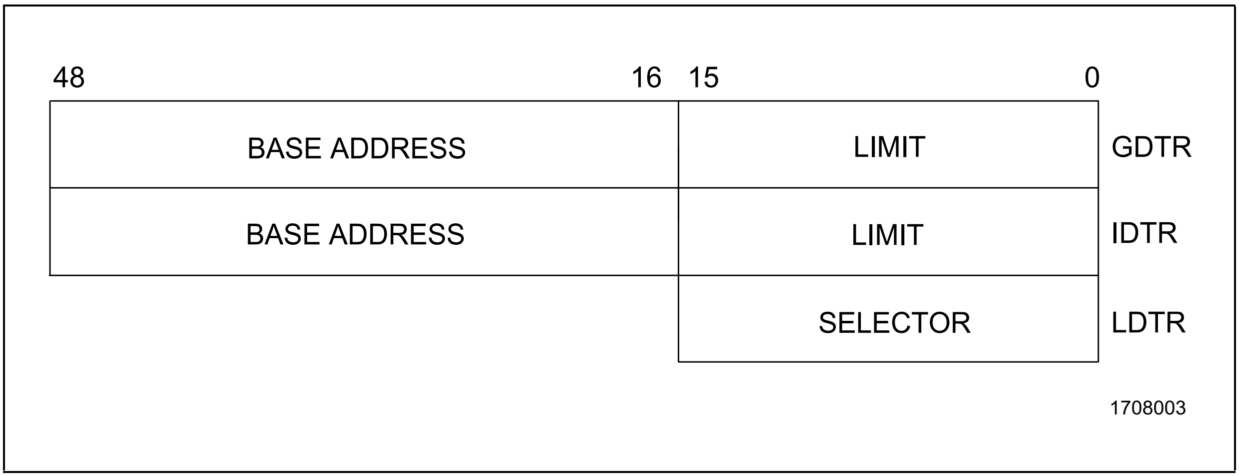
The **Global Descriptor Table Register** (GDTR) holds a 32-bit linear base address and 16-bit limit for the Global Descriptor Table (GDT). The GDT is an array of up to 8192 _8-byte descriptors. When a segment register is loaded from memory, the TI bit in the segment selector chooses either the GDT or the Local Descriptor Table (LDT) to locate a descriptor. If TI = ‘0’, the index portion of the selector is used to locate a given descriptor within the GDT table. The contents of the GDTR are completely visible to the programmer. The first descriptor in the GDT (location 0) is not used by the CPU and is referred to as the “null descriptor”. If the GDTR is loaded while operating in 16-bit operand mode, the 486 core accesses a 32-bit base value but the upper 8 bits are ignored resulting in a 24-bit base address.

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 8-byte 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.

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 contents of the selected descriptor are cached on-chip in the hidden portion of the LDTR. The CPU does not access the GDT again until the LDTR is reloaded. If the LDT descriptor is modified in memory 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.

Figure 3-6. Descriptor Table Registers



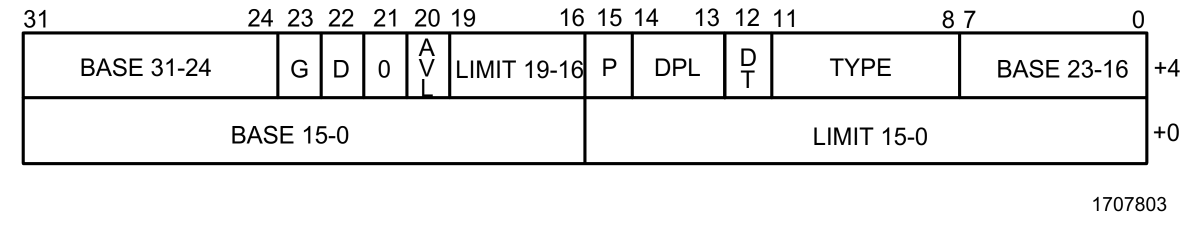
Descriptors

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 Interrupt Descriptor Table (IDT) described later in this text. Figure 3-7 illustrates the descriptor format for both Application Segment Descriptors and System Segment Descriptors.

Figure 3-7. Application and System Segment Descriptors



SEGMENT DESCRIPTOR REGISTER

MEMORY OFFSET +4

Bits 31-24 **BASE, Segment base address Bits 31-24.** 32-bit linear address that points to the beginning of the segment.

Bits 19-16 **LIMIT, Segment limit Bits 19-16.** In real mode, the default segment limit is always 64KBytes (0FFFFh).

Bit 23 **G, Limit Granularity.** '0' = byte granularity, '1' = 4 KBytes (page) granularity.

Bit 22 **D, Default length for operands and effective addresses.** Valid for code and stack segments only: '0' = 16 bit, '1' = 32-bit.

Bit 21 *Reserved.* Must be set to '0'.

Bit 20 **AVL, Segment available.**

Bit 15 **P, Segment present.**

Bits 14-13 **DPL, Descriptor privilege level.**

Bit 12 **DT, Descriptor type.** '0' = system, '1' = application.

Bits 11-8, **TYPE Segment type.**

– System descriptor (DT = '0'):

0010 = LDT descriptor.

1001 = TSS descriptor, task not busy.

1011 = TSS descriptor, task busy.

– Application descriptor (DT = '1'):

Bit 11 E. '0' = data, '1' = executable.

Bit 10 C/D.

E	C/D	Function
0	0	expand up, limit is upper bound of segment.
	1	expand down, limit is lower bound of segment.
1	0	non-conforming.
	1	conforming (runs at privilege level of calling procedure).

SEGMENT DESCRIPTOR (continued)

Bit 9 **R/W.** If E = '0':
'0' = non-writable, '1' = writable.

Bit 8 **A.** '0' = not accessed, '1' = accessed.

Bits 7-0 **BASE, Segment base address Bits 23-16.**

MEMORY OFFSET +0

Bits 31-16 **BASE, Segment base address Bits 15-0.**

Bits 15-0 **LIMIT, Segment limit Bits 15-0.**

GATE DESCRIPTORS

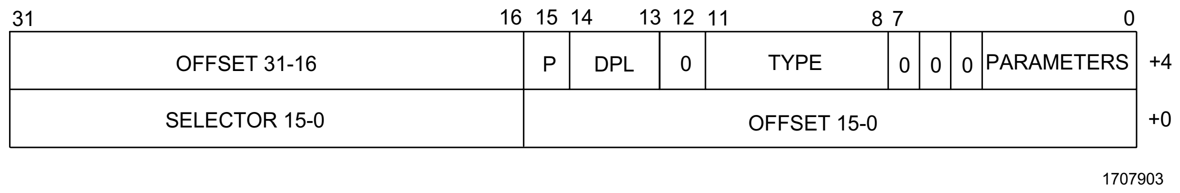
Gate Descriptors provide protection for executable segments operating at different privilege levels. Figure 3-8 illustrates the format for Gate Descriptors.

Task Gate Descriptors (TGD) are used to switch the CPU’s context during a task switch. The selector portion of the TGD locates a Task State Segment. TGDs can be located in the GDT, LDT or IDT tables.

Interrupt Gate Descriptors are used to enter a hardware interrupt service routine. Trap Gate Descriptors are used to enter exceptions or software interrupt service routines. Trap Gate and Interrupt Gate Descriptors can only be located in the IDT.

Call Gate Descriptors are used to enter a procedure (subroutine) that executes at the same or a more privileged level. A Call Gate Descriptor primarily defines the procedure entry point and the procedure’s privilege level.

Figure 3-8. Gate Descriptor



GATE DESCRIPTOR REGISTER

MEMORY OFFSET +4

Bits 31-16 **OFFSET Bits 31-16.** Offset used during a call gate to calculate the branch target.

Bits 15 **P. Segment present.**

Bits 14-13 **DPL. Descriptor privilege level.**

Bit 12 *Reserved.* This bit must be set to '0'.

Bits 11-8 TYPE. Segment type:

Bits 11-8	Segment type
0100	16-bit call gate
0101	task gate
0110	16-bit interrupt gate
0111	16-bit trap gate
1100	32-bit call gate
1110	32-bit interrupt gate
1111	32-bit trap gate

Bits 7-5 *Reserved.* These bits must be set to '0'.

Bits 4-0 PARAMETERS. Number of 32-bit parameters to copy from the caller's stack to the called procedure's stack.

MEMORY OFFSET +0

Bits 31-16 SELECTOR. Segment selector used during a call gate to calculate the branch target.

Bits 15-0 OFFSET Bits 15-0. Offset used during a call gate to calculate the branch target.

4.4.4.3 Task Register

The **Task Register** (TR) holds a 16-bit selector for the current Task State Segment (TSS) table as shown in Figure 3-9. The TR is loaded and stored via the LTR and STR instructions, respectively. The TR can only be accessed during protected mode and can only be loaded when the privilege level is 0 (most privileged). When the TR is loaded, the TR selector field indexes a TSS descriptor that must reside in the Global Descriptor Table (GDT). The contents of the selected descriptor are cached on-chip in the hidden portion of the TR.

During task switching, the processor saves the current CPU state in the TSS before starting a new task. The TR points to the current TSS. The TSS can be either a 386/486-type 32-bit TSS (Figure 3-10) or a 286-type 16-bit TSS type (Figure 3-11). An I/O permission bit map is referenced in the 32-bit TSS by the I/O Map Base Address.

Figure 3-9. Task Register

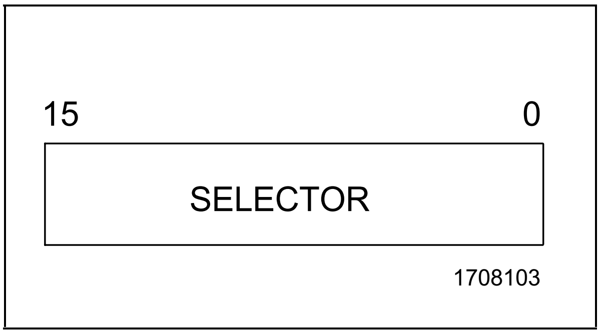
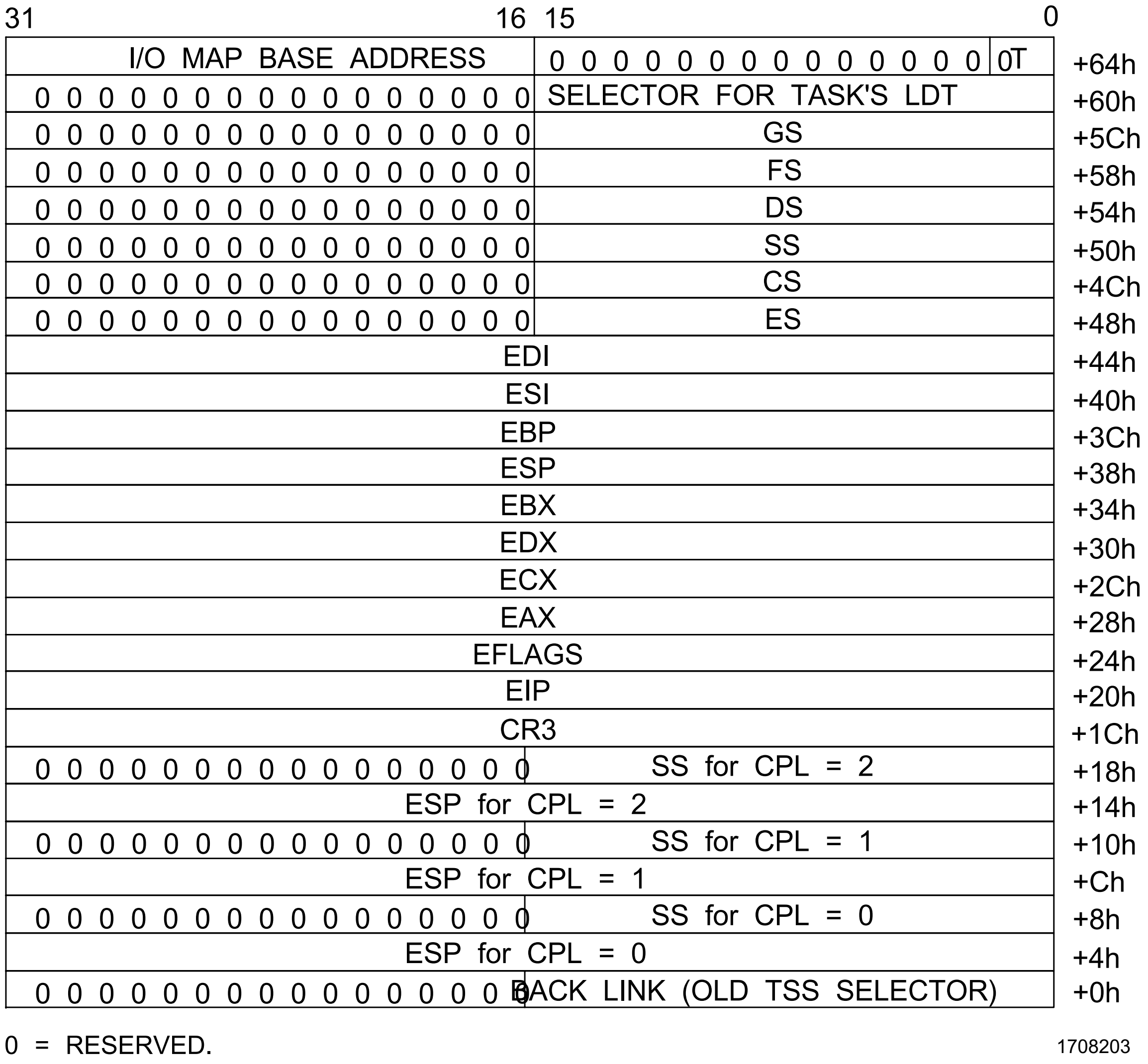


Figure 3-10. 32-Bit Task State Segment (TSS) Table

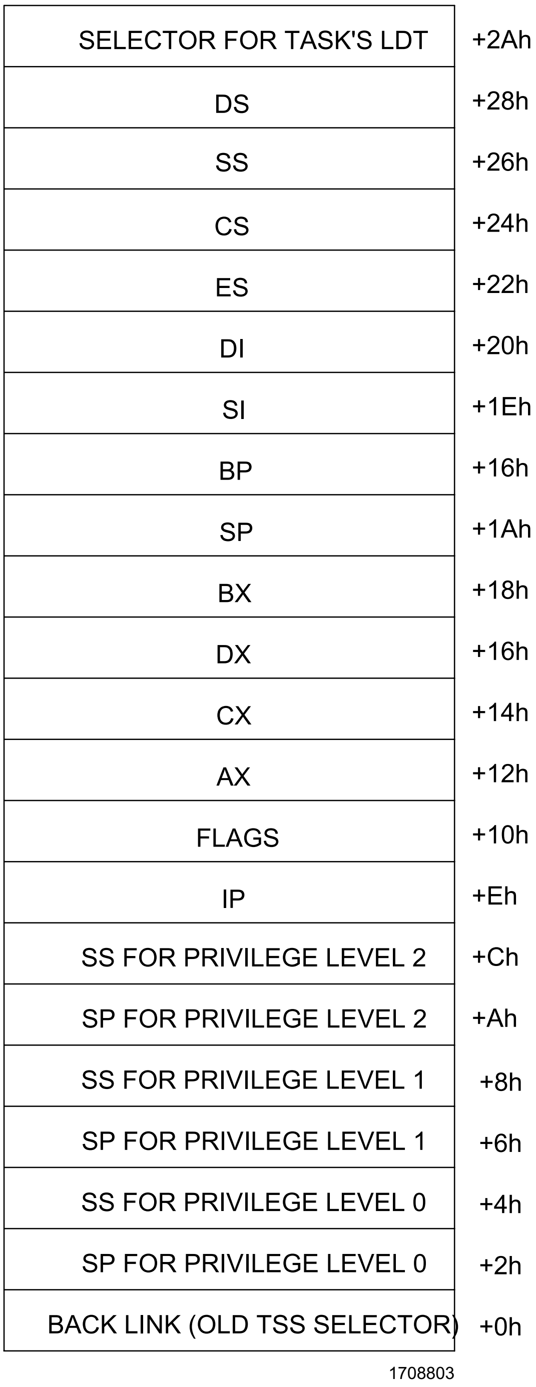


31 … 16	15 … 1	0	
I/O MAP BASE ADDRESS	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0T	+64h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SELECTOR FOR TASK'S LDT		+60h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS		+5Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS		+58h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS		+54h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS		+50h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS		+4Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES		+48h
EDI			+44h
ESI			+40h
EBP			+3Ch
ESP			+38h
EBX			+34h
EDX			+30h
ECX			+2Ch
EAX			+28h
EFLAGS			+24h
EIP			+20h
CR3			+1Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL = 2		+18h
ESP for CPL = 2			+14h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL = 1		+10h
ESP for CPL = 1			+Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL = 0		+8h
ESP for CPL = 0			+4h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	BACK LINK (OLD TSS SELECTOR)		+0h

0 = RESERVED.

1708203

Figure 3-11. 16-Bit Task State Segment (TSS) Table



SELECTOR FOR TASK'S LDT	+2Ah
DS	+28h
SS	+26h
CS	+24h
ES	+22h
DI	+20h
SI	+1Eh
BP	+16h
SP	+1Ah
BX	+18h
DX	+16h
CX	+14h
AX	+12h
FLAGS	+10h
IP	+Eh
SS FOR PRIVILEGE LEVEL 2	+Ch
SP FOR PRIVILEGE LEVEL 2	+Ah
SS FOR PRIVILEGE LEVEL 1	+8h
SP FOR PRIVILEGE LEVEL 1	+6h
SS FOR PRIVILEGE LEVEL 0	+4h
SP FOR PRIVILEGE LEVEL 0	+2h
BACK LINK (OLD TSS SELECTOR)	+0h

4.4.4.4 Configuration Registers

The ST486 Core provides three 8-bit Configuration Control Registers (CCR1, CCR2 and CCR3) used to control the on-chip write-back cache, the coprocessor interface and SMM features. The ST486 Core also provides two 8-bit internal read-only device identification registers (DIR0 and DIR1) and one 24-bit SMM Address Region Register (SMAR). The CCR, DIR and SMAR registers exist in IO memory spaceand are selected by a “register index” number as listed in Table 3-4.

Access to these registers is achieved by writing the index of the register to IO port 22h. IO port 23H is then used for data transfer. Each IO port 23h data transfer musr be preceded by an IO port 22h register index selection, otherwise the second and later IO port 23H operations are directed off-chip and produce external IO cycles. If the register index number is outside the C0h-CFh, FEh-FFh range, external IO cycles will also occur.

Table 3-4. Configuration registers

Register Name	Register Index	Width
Configuration Control 1 CCR1	C1h	8 bits
Configuration Control 2 CCR2	C2h	8 bits
Configuration Control 3 CCR3	C3h	8 bits
SMM Address Region SMAR	CDh, CEh, CFh	24 bits
Device Identification 0 DIR0	FEh	8 bits
Device Identification 1 DIR1	FFh	8 bits
Note: The following register index numbers are reserved for future use: C0h through CFh and FEh, FFh.		

CONTROL REGISTER 1, CCR1

The 8-bit CCR1 register (Figure 3-12) controls SMM features and enables SMM and cache interface pins.

Bits 7-5 *Reserved.*

Bit 4 **NO_LOCK, Negate LOCK#.** If = '1': All bus cycles are issued with LOCK# internal signal negated except page table accesses. Interrupt acknowledge cycles are executed as locked cycles even though LOCK# is negated. With NO_LOCK set, previously noncacheable locked cycles are executed as unlocked cycles and therefore, may be cached. This results in higher CPU performance.

Bit 3 **MMAC, Main Memory Access.** If = '1': All data accesses which occur within an SMI service routine (or when SMAC = '1') access main memory instead of SMM memory space. If = '0': No effect on access.

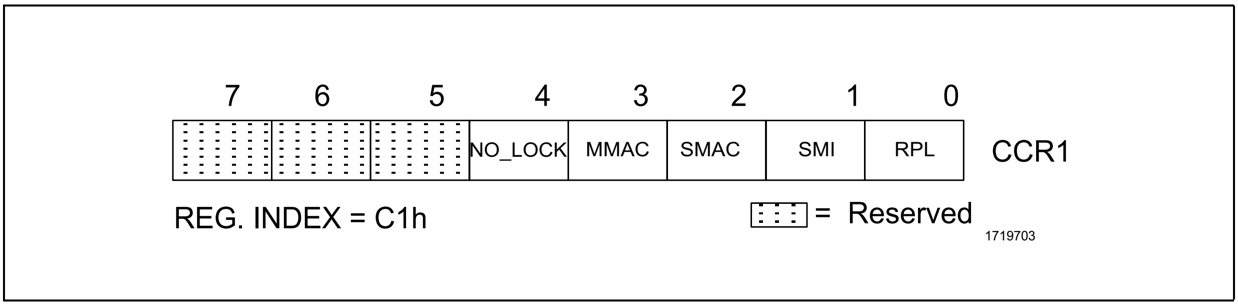
Bit 2 **SMAC, System Management Memory Access.** If = '1': Any access to addresses within the SMM memory space cause external bus cycles to be issued with SMADS# output active. SMI# input is ignored. If = '0': No effect on access.

Bit 1 **SMI, Enable SMM internal signals.** If = '1': SMI# input/output pin and SMADS# output internal signal are enabled. If = '0': SMI# input is ignored and SMADS# output floats.

Bit 0 **RPL, Enable RPL Pins.** If = '1': Enable output pins RPLSET(1-0) and RPLVAL#. If = '0': Output pins RPLSET(1-0) and RPLVAL# float.

Note: Bits 4-0 are cleared to '0' at reset.

Figure 3-12. Configuration Control Register 1



CONTROL REGISTER 2, CCR2

The CCR2 register (Figure 3-13) is used to setup internal cache operation and enable suspend control pins.

Bit 7 **SUSP, Enable Suspend Pins.** If = '1': SUSP# input and SUSPA# output are enabled. If = '0': SUSP# input is ignored and SUSPA# output floats.

Bit 6 **BWRT, Enable Burst Write Cycles.** If = '1': Enables use of 16-byte burst write-back cycles.

Bit 5 **BARB, Enable Cache Coherency on Bus Arbitration.** If = '1': Enable write-back of all dirty cache data when HOLD is requested and prior to asserting HLDA.

Bit 4 **WT1, Write-Through Region 1.** If = '1': Forces all writes to the 640 KBytes to 1 MByte address region that hit in the on-chip cache to be issued on the external bus.

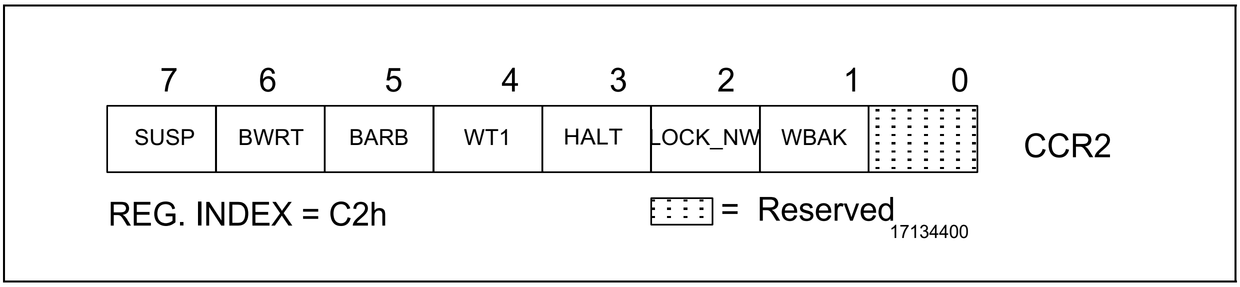
Bit 3 **HALT, Suspend on HALT.** If = '1': CPU enters suspend mode following execution of a HALT instruction.

Bit 2 **LOCK_NW, LOCK NW Bit**. If = '1': Prohibits changing the state of the NW bit in CRO.

Bit 1 **WBAK, EnableWrite-Back Cache Interface Pins.** If = '1': Enable INVAL and WM_RST input pins, and HITM# output pin. If = '0': INVAL and WM_RST input pins are ignored, and HITM# output pin floats.

Note: All bits are cleared to zero at reset

Figure 3-13. Configuration Control Register 2



7	6	5	4	3	2	1	0
SUSP	BWRT	BARB	WT1	HALT	LOCK_NW	WBAK	▨

CONTROL REGISTER 3, CCR3

The CCR3 register (Figure 3-14) controls additional SMM features.

Bits 7-4 *Reserved.*

Bit 3 **SMM_MODE, SMM Interface Mode.** If = ‘0’: the ST Interface mode is used for the SMM mode. If = ‘1’: the SL-compatible mode is used. (Table 3-5)

Bit 2 *Reserved.*

Bit 1 **NMIEN, NMI Enable.** If = '1': NMI is enabled during SMM. If = '0': NMI is not recognized during SMM.

Bit 0 **SMI_LOCK, SMM Register Lock.** If = '1': the following SMM control bits cannot be modified:

CCR1 bits: 1, 2, and 3
CCR3: bits 1 and 3
all SMAR bits.

However, while operating within a SMI handler these SMM control bits can be modified.

Once set,, the SMI_LOCK bit can only be cleared by asserting the $\overline{\text{RESET}}$ pin.

Note: Bits 3,1-0 are cleared to zero at reset.

Figure 3-14. Configuration Control Register 3

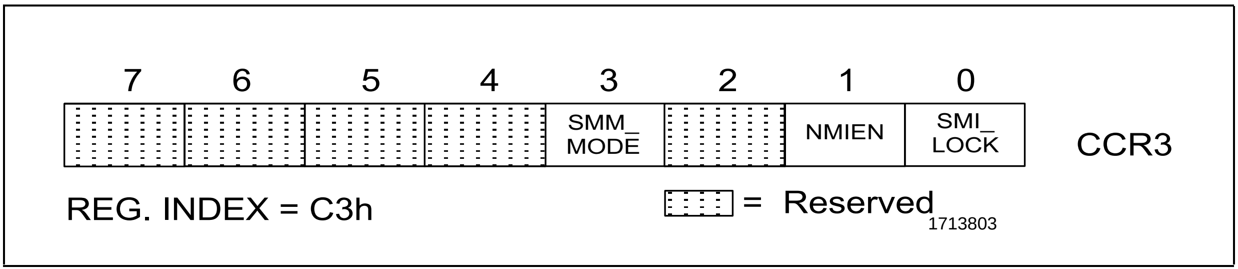


Table 3-5. SMM Pin Definitions

ST MODE	SL-Compatible Mode
$\overline{\text{SMI}}$: Bidirectional System management pin. Asserted by the system logic to request an SMI interrupt. Sampled by the CPU on each rising edge. Causes an I/O trap to occur if sampled asserted at least two clocks prior to RDY# sampled asserted for an IO cycle. Asserted by the CPU during execution of an SMI service routine or in response to SMINT if SMAC is set.	$\overline{\text{SMI}}$: System Management Interrupt input pin Asserted by the system logic to request an SMI interrupt. Sampled by the CPU on each rising edge. SMI# is falling edge sensitive and causes an I/O trap to occur if sampled asserted at least three clocks prior to $\overline{\text{RDY}}$/$\overline{\text{BRDY}}$ sampled for any I/O cycle.
$\overline{\text{SMADS}}$: SMI Address Strobe output is used to indicate that the current bus cycle in an SMM memory access.	$\overline{\text{SMIACT}}$: SMI Active output asserted by the CPU during execution of an SMI service routine

SMM ADDRESS REGION, SMAR

The SMAR register (Figure 3-15), is used to define the location and size of the memory region associated with SMM memory space. The starting address of the SMM address region must be on a block size boundary. For example, a 128 KByte block is allowed to have a starting address of 0 KBytes, 128 KBytes, 256 KBytes, etc. The SMM block size must be defined for SMI# to be recognized (Table 3-6).

Figure 3-15. SMM Address Region Register (SMAR)

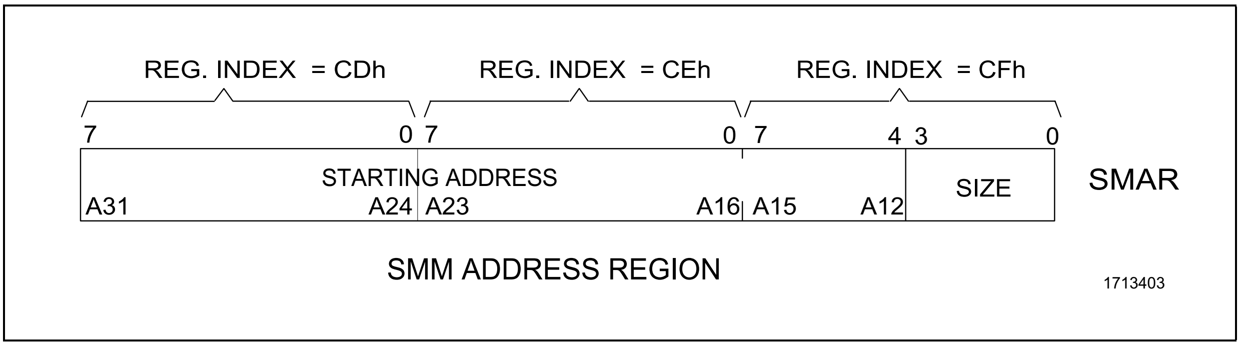


Table 3-6. SMAR Size Field

Bits 3-0	Block Size	Bits 3-0	Block Size
0h	Disabled	8h	512 KBytes
1h	4 KBytes	9h	1 MBytes
2h	8 KBytes	Ah	2 MBytes
3h	16 KBytes	Bh	4 MBytes
4h	32 KBytes	Ch	8 MBytes
5h	64 KBytes	Dh	16 MBytes
6h	128 KBytes	Eh	32 MBytes
7h	256 KBytes	Fh	4 KBytes (same as 1h)

DEVICE IDENTIFICATION 0, DIR0

DIR0 (Figure 3-16) contains an 8-bit value that defines the device type.

Bits 7-0 **DEVID, Device Identification.** DEVID(7-0) bits define the CPU type. These bits are read only. ST5X86 Core = 1Ah for DX, 1Bh for DX2 and 1Fh for DX4.

Figure 3-16. Device Identification Register 0

7	6	5	4	3	2	1	0
DEVID7	DEVID6	DEVID5	DEVID4	DEVID3	DEVID2	DEVID1	DEVID0

DIR0

REG. INDEX = FEh

1723900

DEVICE IDENTIFICATION 1, DIR1

DIR1 (Figure 3-17) contains additional device type information. The upper 4 bits of DIR1 represent the stepping number of the device and the lower 4 bits of DIR1 represent the particular revision number of the stepping. Actual values for DIR0 and DIR1 are shown in Initialized Register Controls, Table 3-1 earlier in this chapter.

Bits 7-4 **SID, Stepping Identification.** SID bits are read only. and indicate device stepping number

Bits 3-0 **RID, Revision Identification.** RID bits are read only. and indicate device revision number

Note: DIR1 value is greater than or equal to 40h.

Figure 3-17. Device Identification Register 1

7	6	5	4	3	2	1	0
SID3	SID2	SID1	SID0	RID3	RID2	RID1	RID0

DIR1

REG. INDEX = FFh

1723800

4.4.4.5 Debug Registers

Six debug registers (DR0-DR3, DR6 and DR7), shown in Figure 3-18, support debugging on the ST486 core. Memory addresses loaded in the debug registers, referred to as “breakpoints”, generate a debug exception when a memory access of the specified type occurs to the specified address. A breakpoint can be specified for a particular kind of memory access such as a read or a write. Code and data breakpoints can also be set allowing debug exceptions to occur whenever a given data access (read or write) or code access (execute) occurs. The size of the debug target can be set to 1, 2, or 4 bytes. The debug registers are accessed via MOV instructions which can be executed only at privilege level 0.

DEBUG REGISTERS DR0-DR3

The Debug Address Registers (DR0-DR3) each contain the linear address for one of four possible breakpoints.

Figure 3-18. Debug Registers

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
LEN 3		R/W 3		LEN 2		R/W 2		LEN 1		R/W 1		LEN 0		R/W 0		0	0	GD	0	0	1	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0	DR7
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	BT	BS	0	0	1	1	1	1	1	1	1	1	B3	B2	B1	B0	DR6
BREAKPOINT 3 LINEAR ADDRESS																																DR3
BREAKPOINT 2 LINEAR ADDRESS																																DR2
BREAKPOINT 1 LINEAR ADDRESS																																DR1
BREAKPOINT 0 LINEAR ADDRESS																																DR0

ALL BITS MARKED AS 0 OR 1 ARE RESERVED AND SHOULD NOT BE MODIFIED.

1703203

DEBUG REGISTER DR7

Each breakpoint is further specified by bits in the Debug Control Register (DR7). For each breakpoint address in DR0-DR3, there are corresponding fields L, R/W, and LEN in DR7 that specify the type of memory access associated with the breakpoint.

The R/W field can be used to specify instruction execution as well as data access breakpoints. Instruction execution breakpoints are always taken before execution of the instruction that matches the breakpoint.

R/Wi. (2 bits, i = 0..3) Applies to the DRi breakpoint address register:

RWi	Breakpoint on...
00	Break on instruction execution only
01	Break on data writes only
10	Not used
11	Break on data reads or writes

LENi (2 bits, i = 0..3) Applies to the DRi breakpoint address register:

LENi	Byte Length
00	One byte length
01	Two byte length
10	Not used
11	Four byte length

Gi (1 bit, i = 0..3) If set to a ‘1’, breakpoint in DRi is globally enabled for all tasks and is not cleared by the processor as the result of a task switch.

Li (1 bit, i = 0..3) If set to a ‘1’, breakpoint in DRi is locally enabled for the current task and is cleared by the processor as the result of a task switch.

GD (1 bit, i = 0..3) Global disable of debug register access. GD bit is cleared whenever a debug exception occurs.

DEBUG REGISTER DR6

The Debug Status Register (DR6) reflects conditions that were in effect at the time the debug exception occurred. The contents of the DR6 register are not automatically cleared by the processor after a debug exception occurs and, therefore, should be cleared by software at the appropriate time.

Bi (1 bit, i = 0..3) Bi is set by the processor if the conditions described by DRi, R/Wi, and LENi occurred when the debug exception occurred, even if the breakpoint is not enabled via the Gi or Li bits.

BT (1 bit) BT is set by the processor before entering the debug handler if a task switch has occurred to a task with the T bit in the TSS set.

BS (1 bit) BS is set by the processor if the debug exception was triggered by the single-step execution mode (TF flag in EFLAGS set).

Code execution breakpoints may also be generated by placing the breakpoint instruction (INT 3) at the location where control is to be regained. The single-step feature may be enabled by setting the TF flag in the EFLAGS register. This causes the processor to perform a debug exception after the execution of every instruction.

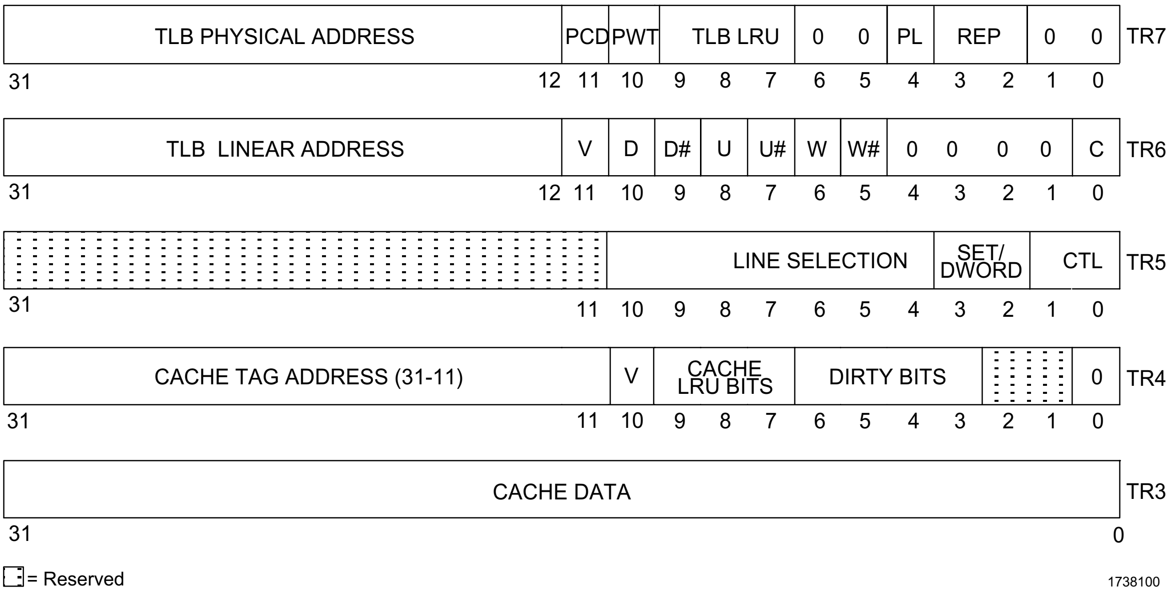
4.4.4.6 Test Registers

The five test registers, shown in Figure 3-19, are used in testing the CPU’s Translation Look-aside Buffer (TLB) and on-chip cache. TR6 and TR7 are used for TLB testing, and TR3-TR5 are used for cache testing. The bit definitions for the TR6 and TR7 registers follow on the next page.

The ST486 core TLB is a four-way set associative memory with eight entries per set. Each TLB entry consists of a 24-bit tag and 20-bit data. The 24-bit tag represents the high-order 20 bits of the linear address, a valid bit, and three attribute bits. The 20-bit data portion represents the upper 20 bits of the physical address that corresponds to the linear address.

TLB Test Registers

Figure 3-19. Test Registers



TLB TEST CONTROL REGISTER, TR6

The TLB Test Control Register (TR6) contains a command bit, the upper 20 bits of a linear address, a valid bit and the attribute bits used in the test operation. The contents of TR6 is used to create the 24-bit TLB tag during both write and read (TLB lookup) test operations. The command bit defines whether the test operation is a read or a write.

Bits 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.

Bit 11 **Valid bit (V).**

TLB write: If set, indicates that the TLB entry contains valid data. If clear, target entry is invalidated.

Bits 10-9 **D, D#, Dirty attribute bit and its complement.** Refer to Table 3-7.

Bits 8-7 **U, U#, User/supervisor attribute bit and its complement.** Refer to Table 3-7.

Bits 6-5 **R, R#, Read/write attribute bit and its complement.** Refer to Table 3-7.

Bit 0 **C, Command bit.** If = 0': TLB write, If = '1': TLB lookup.

TLB TEST CONTROL REGISTER, TR7

The TLB Test Data Register (TR7) contains the upper 20 bits of the physical address (TLB data field), three LRU bits and a control bit. During TLB write operations, the physical address in TR7 is written into the TLB entry selected by the contents of TR6. During TLB lookup operations, the TLB data selected by the contents of TR6 is loaded into TR7.

Bits 31-12 **Physical address.**

TLB lookup: data field from the TLB.

TLB write: data field written into the TLB.

Bit 11 **PCD, Page-level cache disable bit.** This bit corresponds to the PCD bit of a page table entry.

Bit 10 **PWT, Page-level cache write-through bit.**

Corresponds to the PWT bit of a page table entry.

Bits 9-7 **LRU bits.**

TLB lookup: LRU bits associated with the TLB entry prior to the TLB lookup.

TLB write: ignored.

Bit 4 **PL bit.**

TLB lookup: If = '1', read hit occurred. If = '0', read miss occurred.

TLB write: If = '1', REP field is used to select the set. If = '0', the pseudo-LRU replacement algorithm is used to select the set.

Bits 3-2 **REP, Set selection.**

TLB lookup: If PL = '1', set in which the tag was found. If PL = '0', undefined data.

TLB write: If PL = '1', selects one of the four sets for replacement. If PL = '0', ignored.

Table 3-7. TR6 Attribute Bit Pairs

Bit (D, U or R)	Bit Complement (D#, U#, or R#)	Effect On TLB Loolup	Effect On TLB Write
0	0	Do not match.	Undefined.
0	1	Match if D, U or R bit is a '0'.	Clear the bit.
1	0	Match if D, U or R bit is a '1'.	Set the bit.
1	1	Match if D, U or R bit is either a '1' or '0'.	Undefined.

Cache Test Registers

The ST486 core 8-KByte on-chip cache is a four-way set associative memory that can be configured as either write-back or write-through. Each cache set contains 128 entries. Each entry consists of a 21-bit tag address, a 16-byte data field, a valid bit, and four dirty bits.

The 21-bit tag represents the high-order 21 bits of the physical address. The 16-byte data represents the 16 bytes of data currently in memory at the physical address represented by the tag. The valid bit indicates whether the data bytes in the cache actually contain valid data. The four dirty bits indicate if the data bytes in the cache have been modified internally without updating external memory (write-back configuration). Each dirty bit indicates the status for one double-word (4 bytes) within the 16-byte data field.

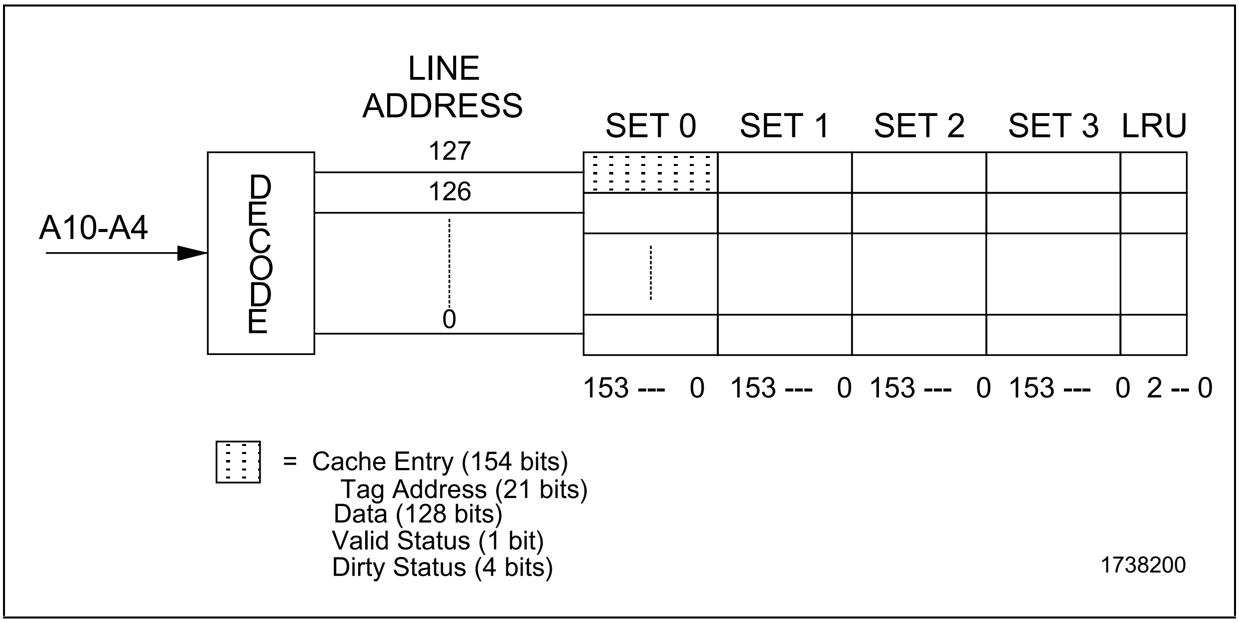
For each line in the cache, there are three LRU bits that indicate which of the four sets was most recently accessed. A line is selected using bits 0 - 4 of the physical address. Figure 3-20 illustrates the ST486 core cache architecture.

The ST486 core contains three test registers that allow testing of its internal cache. Using these registers, cache writes and reads may be performed.

Cache test writes cause the data in the cache fill buffer to be written to the selected set and entry in the cache. Data must be written to TR3 (32-bit register) four times in order to fill the cache fill buffer. Once the cache fill buffer has been loaded, a cache test write can be performed. For data to be written to the allocated entry, the valid bit for the entry must be set prior to the write of the data.

Cache test reads cause the data in the selected set and entry to be loaded into the cache flush buffer. Once the buffer has been loaded, data must be read from TR3 four times in order to empty the cache flush buffer. For proper operation, cache tests should be performed only when the cache is disabled (CD bit in CR0 =’1’).

Figure 3-20. ST486 Cache Architecture



CACHE TEST REGISTER 5, TR5

Bits 10-4 **Line Selection.** Physical address bits 10-4 used to select one of 128 lines.

Bits 3-2 **Set/DWord Selection.**

Cache read: selects which of the four sets is used as the source for data transferred to the cache flush buffer.

Cache write: selects which of the four sets is used as the destination for data transferred from the cache fill buffer.

Flush buffer read: selects which of the four dwords in the flush buffer is loaded into TR3.

Fill buffer write: selects which of the four dwords in TR3 is written to the fill buffer.

Bits 1-0 **Control Bits.**

Bits 1-0	Cache Test Function
00	Flush read or fill buffer write. Writing to TR3 fill buffer write. Reading TR3 initiates flush buffer read
01	Cache write
10	Cache read
11	Cache flush

CACHE TEST REGISTER 4, TR4

Bits 31-11 **Upper Tag Address.**

Cache read: upper 21 bits of tag address of the selected entry.

Cache write: data written into the upper 21 bits of the tag address of the selected entry.

Bit 10 **Valid Bit.**

Cache read: valid bit for the selected entry.

Cache write: data written into the valid bit for the selected entry.

Bits 9-7 **LRU Bits.**

Cache read: the LRU bits for the selected line.

xx1 = Set 0 or Set 1 most recently accessed.

xx0 = Set 2 or Set 3 most recently accessed.

x1x = Most recent access to Set 0 or Set 1 was to Set 0.

x0x = Most recent access to Set 0 or Set 1 was to Set 1.

1xx = Most recent access to Set 2 or Set 3 was to Set 2.

0xx = Most recent access to Set 2 or Set 3 was to Set 3.

Cache write: ignored.

Bits 6-3 **Dirty Bits.**

Cache read: the dirty bits for the selected entry (one bit per dword).

Cache write: data written into the dirty bits for the selected entry.

CACHE TEST REGISTER 3, TR3

Bits 31-0 **Cache data.**

Flush buffer read: data accessed from the cache flush buffer.

Fill buffer write: data to be written into the cache fill buffer.

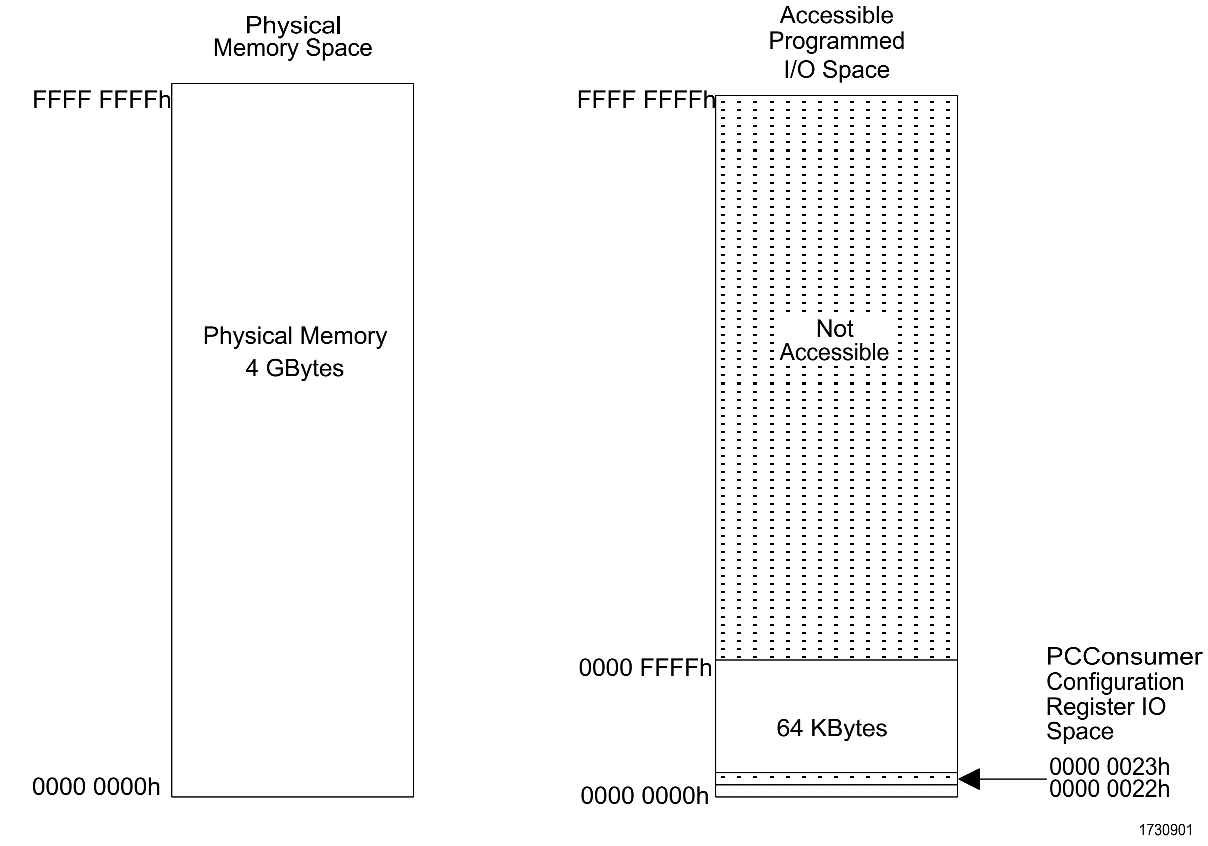
3.5 Address Spaces

The STPC can directly address either memory or IO space. Figure 3-21 illustrates the range of addresses available for memory and IO address space. The addresses for physical memory range between 0000 0000h and FFFF FFFFh (4 GBytes). The accessible IO addresses space ranges between 0000 0000h and 0000 FFFFh

(64KBytes). The ST486 Core does not use co-processor space in upper IO space between 8000 00F8h and 8000 00FFh as do the 386-style CPUs. The IO locations 22h and 23h are used for STPC configuration and on-chip peripheral configuration registers

.

Figure 3-21. Memory and IO Address Spaces



4.5.1 I/O Address Space

The ST486 IO address space is accessed using IN and OUT instructions to addresses referred to as “ports”. The accessible IO address space is 64 KBytes and can be accessed as 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 IO space instead of memory space.

The ST486 core configuration registers reside within the IO address space at port addresses 22h and 23h and are accessed using the standard IN and OUT instructions. The configuration registers are modified by writing the index of the configuration register to port 22h and then transferring the data through port 23h. Accesses to the on-chip configuration registers do not generate external I/O cycles. However, each port 23h operation must be preceded by a port 22h write with a valid index value.

Otherwise, the second and later port 23h operations are directed off-chip and generate external I/O cycles without modifying the on-chip configuration registers. Also, writes to port 22h outside of the ST486 core index range (C0h-CFh and FEh-FFh) result in external I/O cycles and do not effect the _on-chip configuration registers. Reads of port 22h are always directed off-chip.

4.5.2 Memory Address Space

The ST486 core directly addresses up to 4 GBytes of physical memory. Memory address space is accessed as bytes, words (16-bits) or doublewords (32-bits). 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.

With the ST486 core, memory can be addressed using nine different addressing modes. 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 ST486 core 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 ST486 core memory address space.

4.5.2.1 Offset Mechanism

The offset mechanism computes an offset (effective) address by adding together up to three values: a base, an index and a displacement. The base, if present, is the value in one of eight 32-bit general registers at the time of the execution of the instruction. The index, like the base, is a value that is contained in one of the 32-bit general registers (except the ESP register) when the instruction is executed. 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 which is a value of up to 32-bits in length supplied as part of the instruction. Figure 3-22 illustrates the calculation of the offset address.

Nine valid combinations of the base, index, scale factor and displacement can be used with the ST486 core instruction set. These combinations are listed in Table 3-8. The base and index both refer to contents of a register as indicated by [Base] and [Index].

Figure 3-22. Offset Address Calculation

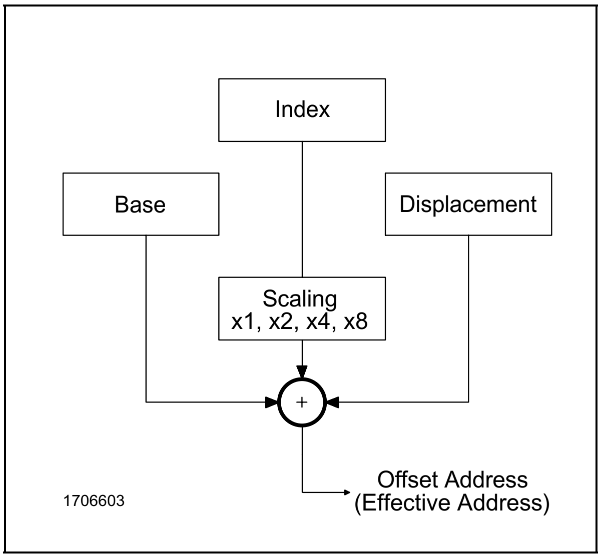


Table 3-8. Memory Addressing Modes

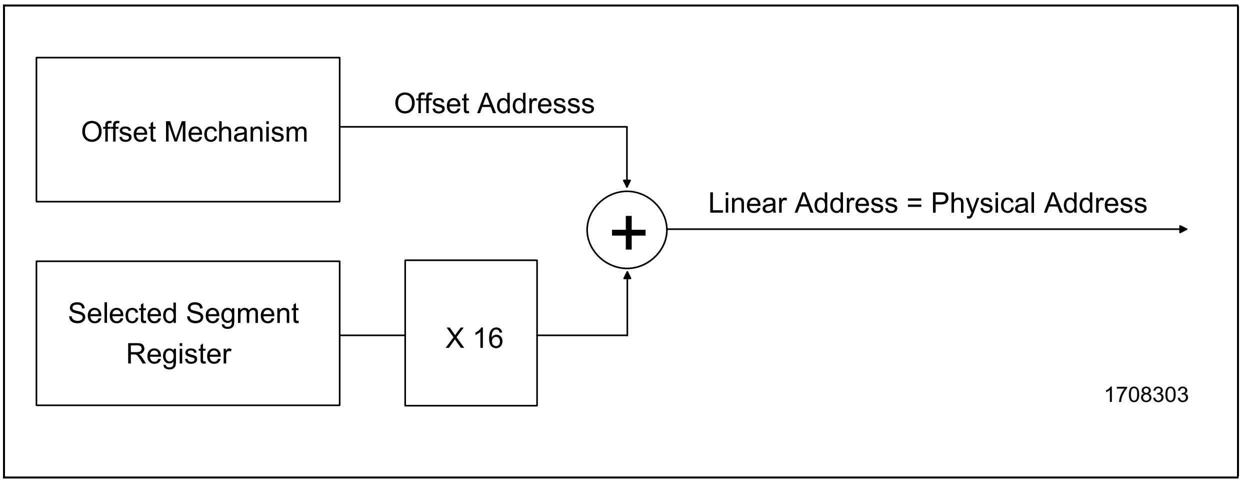
Addressing Mode	Base	Index	Scale Factor (SF)	Displacement (DP)	Offset Address (OA) Calculation
Direct				x	OA = DP
Register Indirect	x				OA = [BASE]
Based	x			x	OA = [BASE] + DP
Index		x		x	OA = [INDEX] + DP
Scaled Index		x	x	x	OA = ([INDEX] * SF) + DP
Based Index	x	x			OA = [BASE] + [INDEX]
Based Scaled 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

4.5.2.2 Real Mode Memory Addressing

In real mode operation, the ST486 core 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 with twelve zeros in the upper address bits to create the 32-bit physical address. Figure 3-23 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.

Figure 3-23. Real Mode Address Calculation



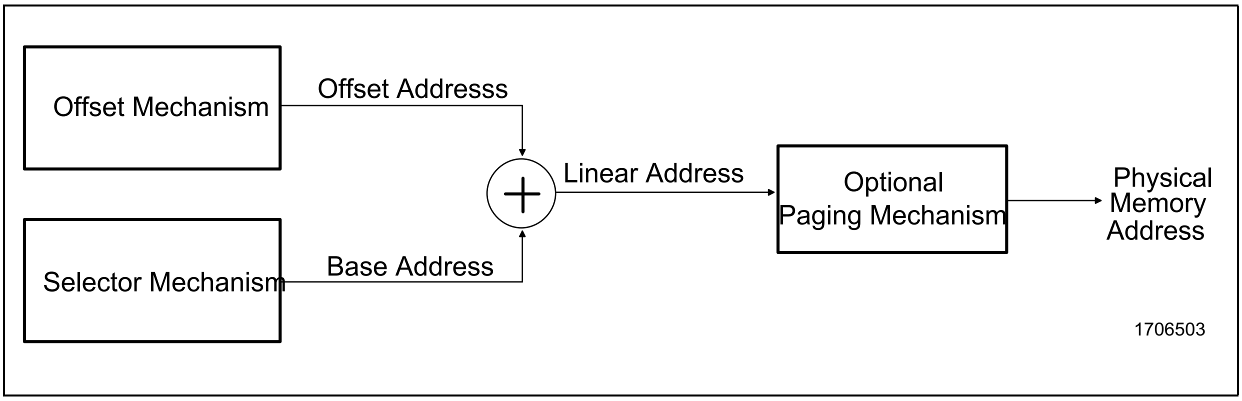
4.5.2.3 Protected Mode Memory Addressing

In protected mode three mechanisms calculate a physical memory address (Figure 3-24).

- 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 used, 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 3-24. Protected Mode Address Calculation

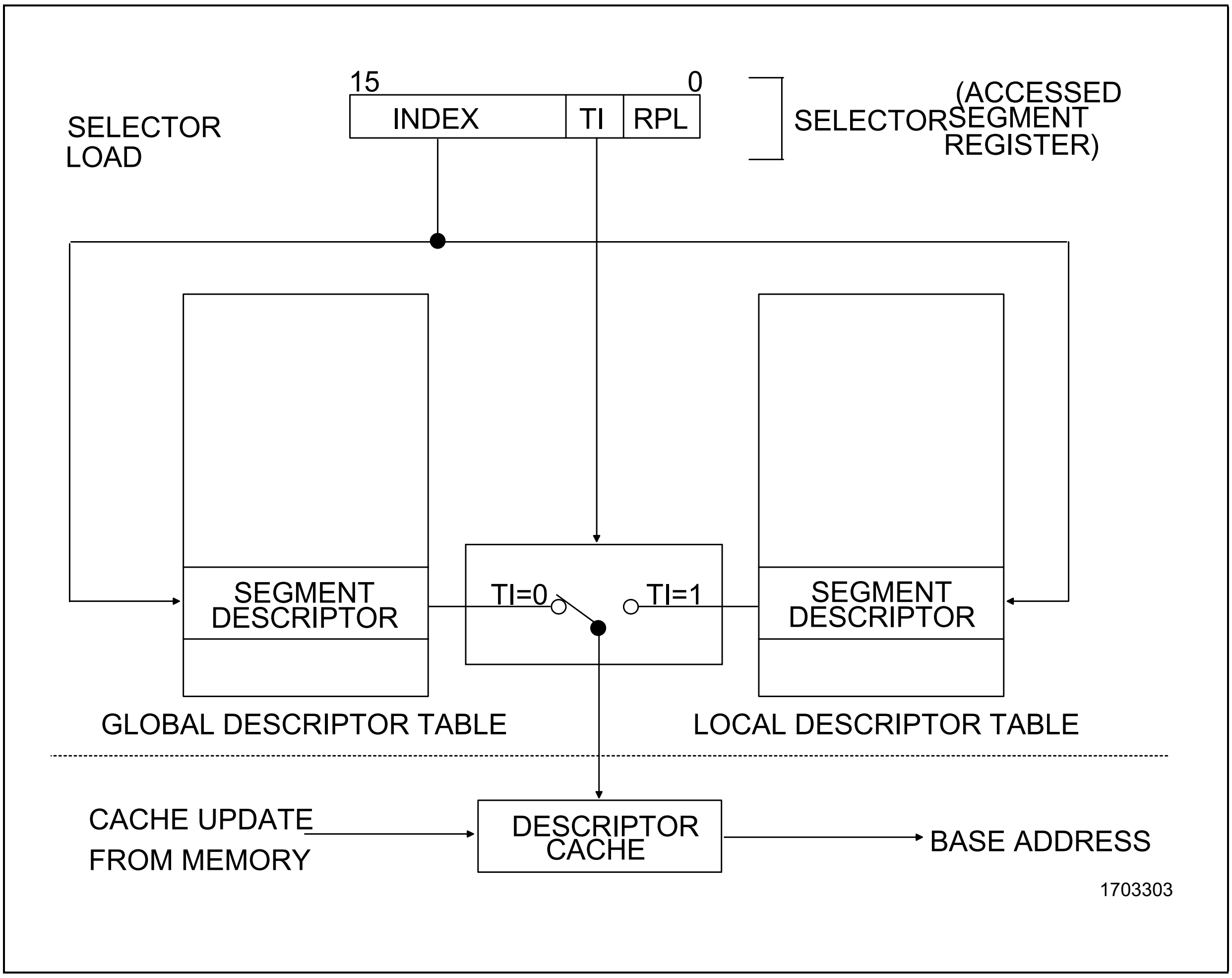


4.5.2.4 Selector Mechanism

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 ST486 core 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 3-25 illustrates the operation of the selector mechanism.

Figure 3-25. Selector Mechanism



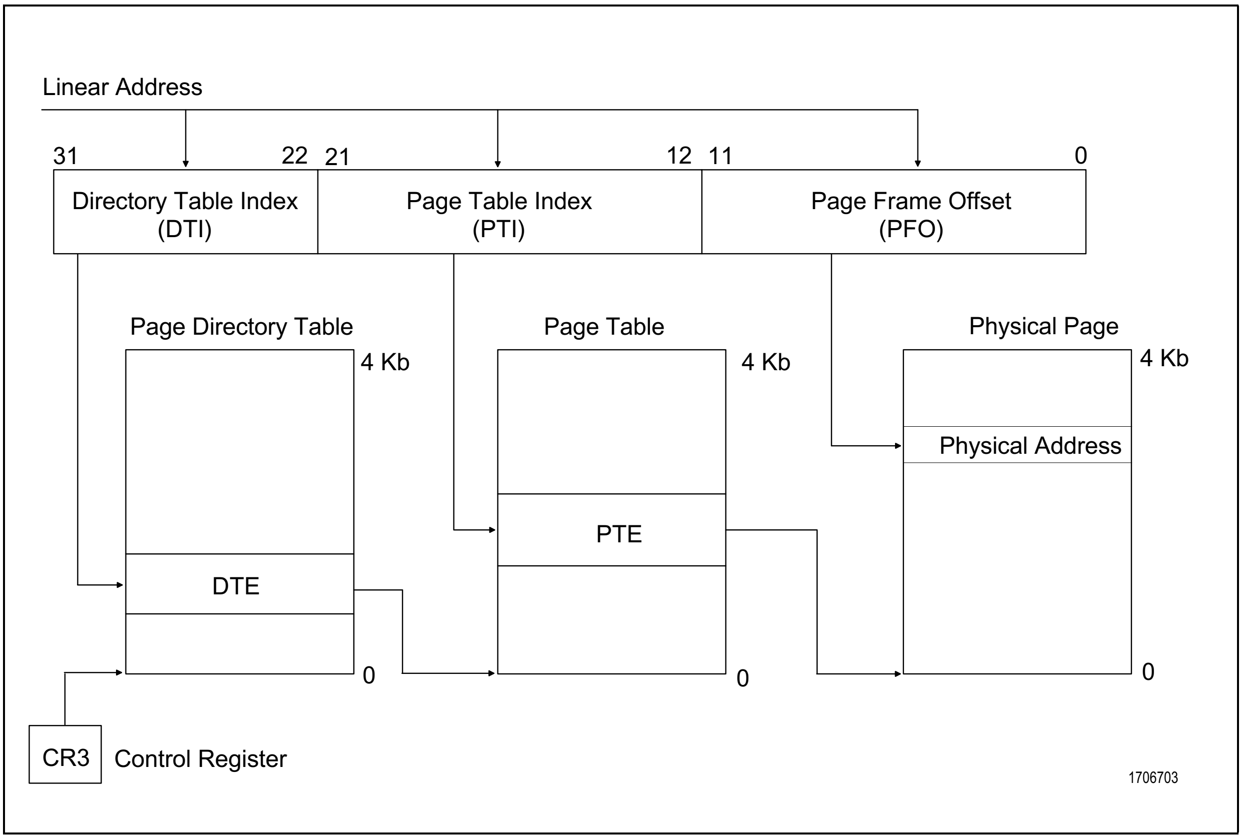
4.5.2.5 Paging Mechanism

The paging mechanism supports a memory subsystem that simulates a large address space with a small amount of RAM and disk storage. The paging mechanism either translates a linear address to its corresponding physical address or generates an exception if the required page is not currently present in RAM. When the operating system services the exception, the required page is loaded into memory and the instruction is then restarted. Pages are either 4 KBytes or 1 MByte in size. The CPU defaults to 4 KByte pages that are aligned to 4 KByte boundaries.

A page is addressed by using two levels of tables as illustrated in Figure 3-26. The upper 10 bits of the 32-bit linear address are used to locate an entry in the page directory table. The page directory table acts as a 32-bit master index to up to 1K individual second-level page tables. The selected entry in the page directory table, referred to as the directory table entry, identifies the starting address of the second-level page table. The page directory table itself is a page and is, therefore, aligned to a 4 KByte boundary. The physical address of the current page directory table is stored in the CR3 control register, also referred to as the Page Directory Base Register (PDBR).

Bits 12-21 of the 32-bit linear address, referred to as the Page Table Index, locate a 32-bit entry in the second-level page table. This Page Table Entry (PTE) contains the base address of the desired page frame. The second-level page table addresses up to 1K individual page frames. A second-level page table is 4 KBytes in size and is itself a page. The lower 12 bits of the 32-bit linear address, referred to as the Page Frame Offset (PFO), locate the desired physical data within the page frame.

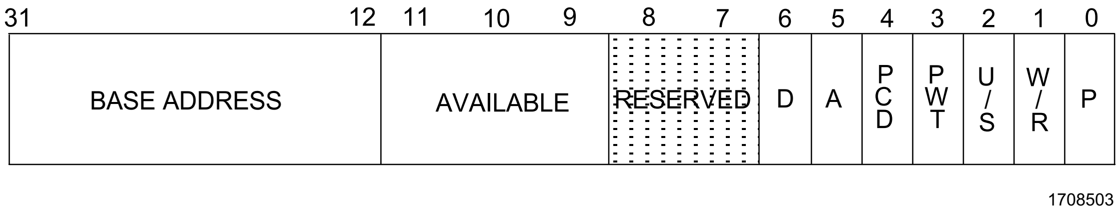
Figure 3-26. Paging Mechanism



Since the page directory table can point to 1K page tables, and each page table can point to 1K of page frames, a total of 1M of page frames can be implemented. Since each page frame contains 4 KBytes, up to 4 GBytes of virtual memory can be addressed by the ST486 core with a single page directory table.

In addition to the base address of the page table or the page frame, each directory table entry or page table entry contains attribute bits and a present bit as illustrated in Figure 3-27.

Figure 3-27. Directory and Page Table Entry (DTE and PTE) Format



DIRECTORY AND PAGE TABLE ENTRY

Bits 31-12 **Base Address.** Specifies the base address of the page or page table.

Bits 11-9 **User**. These bits are undefined and are available to the programmer.

Bits 8-7 *Reserved.* These bits are not available to the programmer.

Bit 6 **D, Dirty Bit.** If set, indicates that a write access has occurred to the page (PTE only, undefined in DTE).

Bitb 5 **A, Accessed Flag.** If set, indicates that a read access or write access has occurred to the page.

Bit 4 **PCD, Page Caching Disable Flag.** If set, indicates that the page is not cacheable in the on-chip cache.

Bit 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.

Bit 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.

Bit 1 **W/R, Write/Read Attribute.** If set (write), page is writable. If clear (read), page is read only.

Bit 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.

If the present bit (P) is set in the DTE, the page table is present and the appropriate page table entry is read. If P='1' in the corresponding PTE (indicating that the page is in memory), the accessed and dirty bits are updated, if necessary, and the operand is fetched. Both accessed bits are set (DTE and PTE), if necessary, to indicate that the table and the page have been used to translate a linear address. The dirty bit (D) is set before the first write is made to a page.

The present bits must be set to validate the remaining bits in the DTE and PTE. If either of the present bits are not set, a page fault is generated when the DTE or PTE is accessed. If P='0', the remaining DTE/PTE bits are available for use by the operating system. For example, the operating system can use these bits to record where on the hard disk the pages are located. A page fault is also generated if the memory reference violates the page protection attributes.

3.6 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.

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

3.6.1 Interrupts

External events can interrupt normal program execution by using one of the three interrupt pins on the ST486 core.

- 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 interrupted instruction.

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 then processed after execution of the next IRET.

During the NMI service routine, maskable interrupts may be enabled. 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 ST486 core 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’. With the exception of string operations, INTR interrupts are acknowledged between instructions. Long string operations have interrupt windows between memory moves that allow INTR interrupts to be acknowledged.

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 which is supplied by an external interrupt controller. This vector selects which of the 256 possible interrupt handlers 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 which 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

3.6.2 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 which 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 caused by a program error and 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 which 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. Debug exceptions (vector 1) are also handled as faults (except for data breakpoints and single-step operations). 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 ST486 core.

3.6.3 Interrupt Vectors

When the CPU services an interrupt or exception, the current program's instruction pointer and flags 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 at the instruction pointer address saved on the stack when the interrupt was serviced.

Interrupt Vector Assignments

Each interrupt (except SMI#) and exception is assigned one of 256 interrupt vector numbers Table 3-9. The first 32 interrupt vector assignments are defined or reserved. INT instructions acting as software interrupts may use any of interrupt vectors, 0 through 255.

The non-maskable hardware interrupt (NMI) is assigned vector 2. Illegal opcodes including faulty FPU instructions will cause an illegal opcode exception, interrupt vector 6.

Table 3-9. 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 ST486 core 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 pre-defined.

Interrupt Descriptor Table

The interrupt vector number is used by the ST486 core 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.

4.6.4 Interrupt and Exception Priorities

As the ST486 core executes instructions, it follows a consistent policy for prioritizing exceptions and hardware interrupts. The priorities for competing interrupts and exceptions are listed in Table 3-10. SMM interrupts always take precedence. Debug traps for the previous instruction and next instructions are handled as the next priority. When NMI and maskable INTR interrupts are both detected at the same instruction boundary, the ST486 core microprocessor services the NMI interrupt first.

The ST486 core 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 ST486 core 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

Table 3-10. Interrupt and Exception Priorities

Priority	Description	Notes
0	SMM hardware interrupt.	SMM interrupts are caused by SMI# asserted and always have highest priority.
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	Non-maskable hardware interrupt.	Caused by NMI asserted.
4	Maskable hardware interrupt.	Caused by INTR asserted and IF = ‘1’.
5	Faults resulting from fetching the next instruction.	Includes segment not present, general protection fault and page fault.
6	Faults resulting from instruction decoding.	Includes illegal opcode, instruction too long, or privilege violation.
7	WAIT instruction and TS = ‘1’ and MP = ‘1’.	Device not available exception generated.
8	ESC instruction and EM = ‘1’ or TS = ‘1’.	Device not available exception generated.
9	Floating point error exception.	Caused by unmasked floating point exception with NE = ‘1’.
10	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.
11	Page Faults that prevent transferring the entire memory operand.	
12	Alignment check fault.	

4.6.5 Exceptions in Real Mode

Many of the exceptions described in Table 3-10 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 3-11.

Table 3-11. Exception Changes in Real Mode

Vector Number	Protected Mode Function	Real Mode Function
8	Double fault.	Interrupt table limit overrun.
10	Invalid TSS.	--
11	Segment not present.	--
12	Stack fault.	SS segment limit overrun.
13	General protection fault.	CS, DS, ES, FS, GS segment limit overrun.
14	Page fault.	--
Note: -- = does not occur		

4.6.6 Error Codes

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

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

The error code format is shown in Figure 3-28 and the error code bit definitions are listed in Table 3-12. 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.

Figure 3-28. Error Code Format

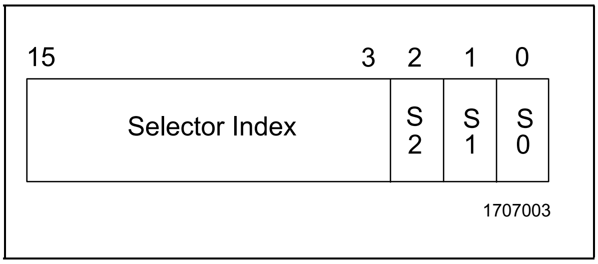


Table 3-12. Error Code Bit Definitions

Fault Type	Selector Index (Bits 15-3)	S2 (Bit 2)	S1 (Bit 1)	S0 (Bit 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.

3.7 System Management Mode

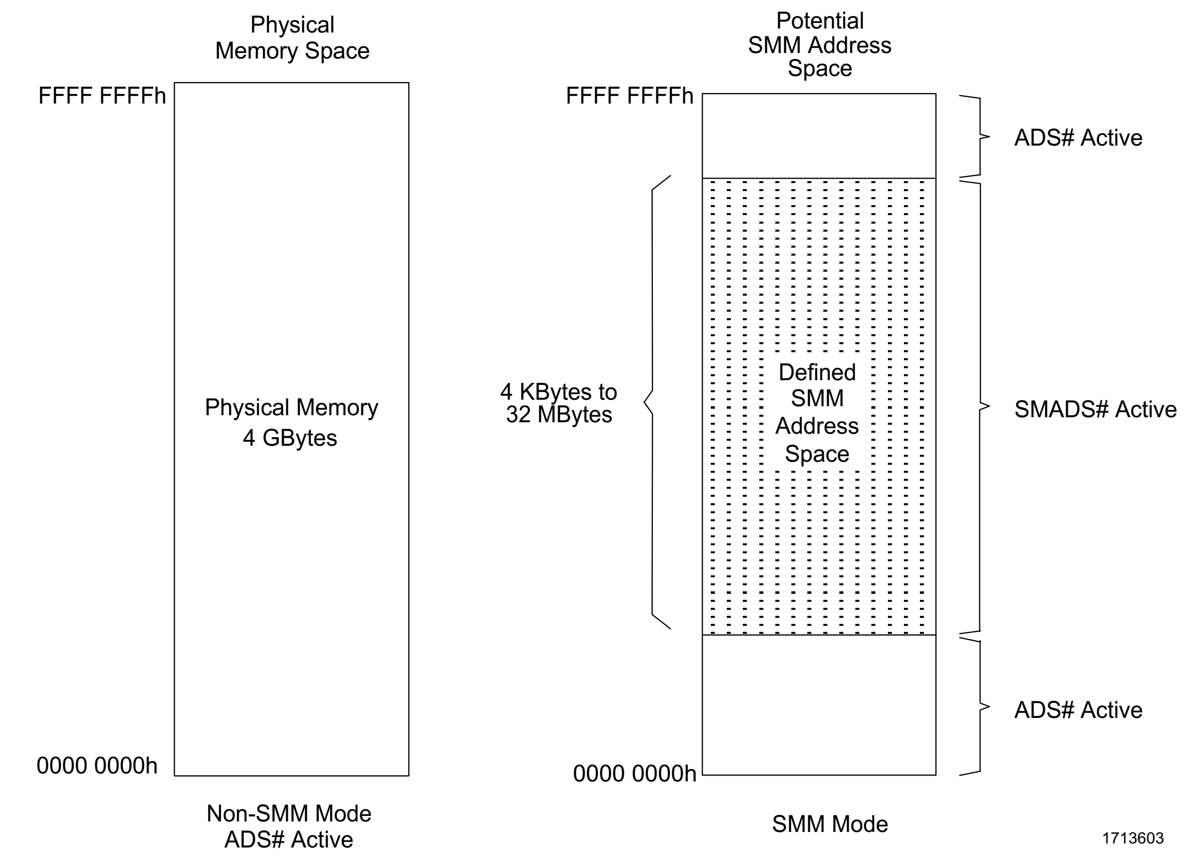
System Management Mode (SMM) provides an additional interrupt which can be used for system power management or software transparent emulation of IO peripherals. SMM is entered using the System Management Interrupt (SMI#) that has a higher priority than any other interrupt, including NMI. An SMI interrupt can also be triggered via the software using an SMINT instruction. After an SMI interrupt, portions of the CPU state are automatically saved, SMM is entered, and program execution begins at the base of SMM address space (Figure 3-29). Running in protected SMM address space, the interrupt routine does not interfere with the operating system or any application program.

Eight SMM instructions have been added to the ST486 instruction set that permit software initiated SMM, and saving and restoring of the total CPU state when in SMM mode.

SMM Operation is specific to ST and is not compatible with Intel one.

See ST486 SMM PROGRAMMING MANUAL for more informations.

Figure 3-29. System Management Memory Address Space



3.7.1 SMM Operation

SMM operation is summarized in Figure 3-30. Entering SMM requires the assertion of the SMI# pin for at least two CLK periods or execution of the SMINT instruction. For the SMI# or SMINT instruction to be recognized, the following configuration register bits must be set as shown in Table 3-13. The configuration registers are discussed in detail earlier in this chapter.

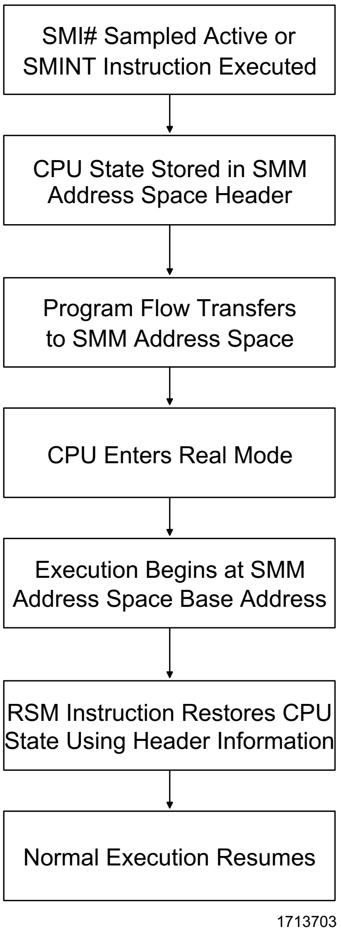
Table 3-13. Requirement for Recognizing SMI# and SMINT

Register (Bit)		SMI#	SMINT
SMI	CCR1 (1)	1	1
SMAC	CCR1 (2)	0	1
SMAR	SIZE (3-0)	> 0	> 0

After recognizing SMI# or SMINT and prior to executing the SMI service routine, some of the CPU state information is changed. Prior to modification, this information is automatically saved in the SMM memory space header located at the top of SMM memory space. After the header is saved, the CPU enters real mode and begins executing the SMI service routine starting at the SMM memory base address.

The SMI service routine is user definable and may contain system or power management software. If the power management software forces the CPU to power down, or the SMI service routine modifies more than what is automatically saved, the complete CPU state information can be saved.

Figure 3-30. SMI Execution Flow Diagram

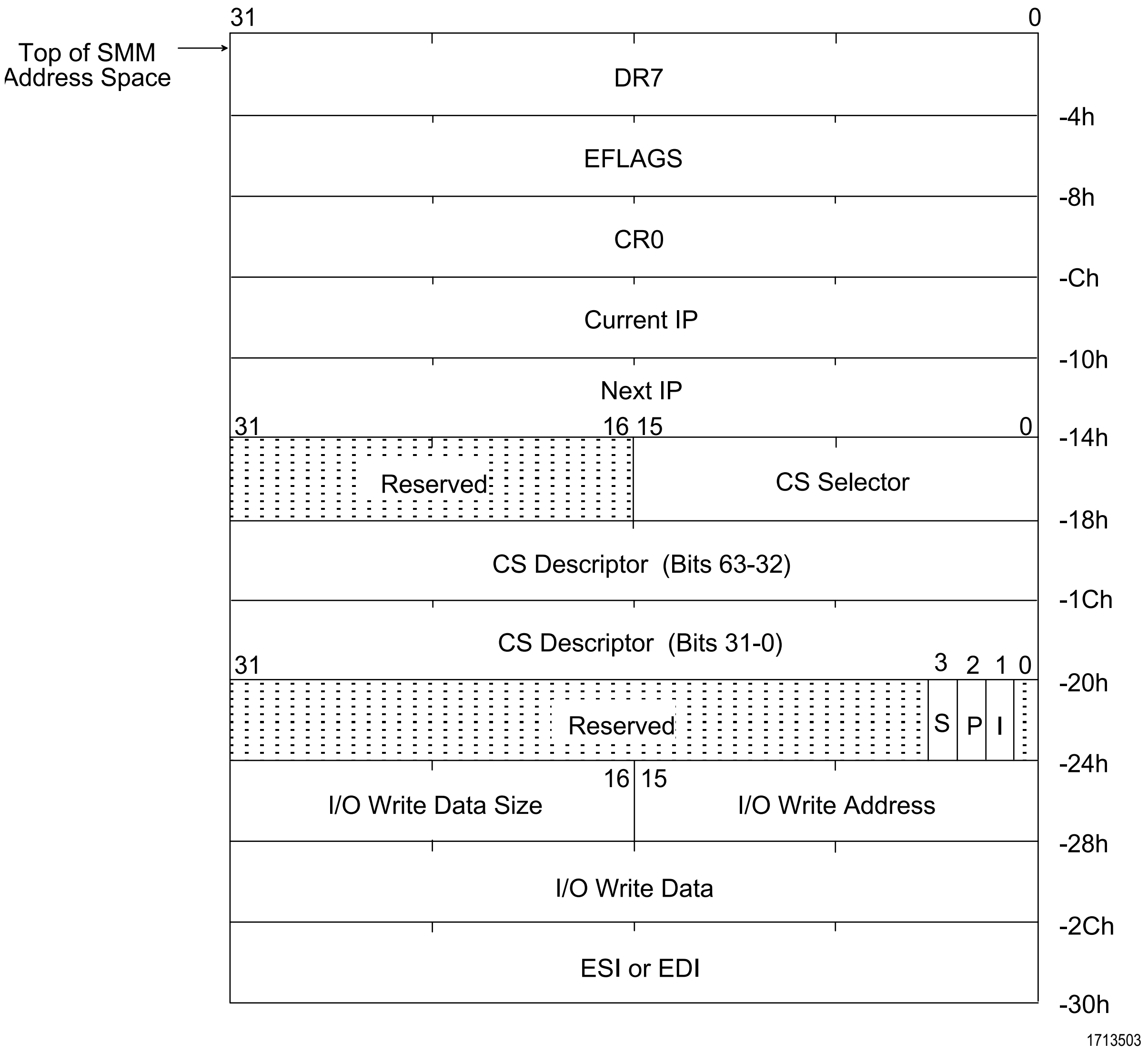

SMI# Sampled Active or SMINT Instruction Executed
CPU State Stored in SMM Address Space Header
Program Flow Transfers to SMM Address Space
CPU Enters Real Mode
Execution Begins at SMM Address Space Base Address
RSM Instruction Restores CPU State Using Header Information
Normal Execution Resumes
1713703

3.7.2 SMM Memory Space Header

With every SMI interrupt or SMINT instruction, certain CPU state information is automatically saved in the SMM memory space header located at the top of SMM address space (Figure 3-31).

The header contains CPU state information that is modified when servicing an SMI interrupt. Included in this information are two pointers. The Current IP points to the instruction executing when the SMI was detected.

Figure 3-31. SMM Memory Space Header



The Next IP points to the instruction that will be executed after exiting SMM. Also saved are the contents of debug register 7 (DR7), the extended flags register (EFLAGS), and control register 0 (CR0). If SMM has been entered due to an I/O trap for a REP INSx or REP OUTSx instruction, the Current IP and Next IP fields contain the same addresses and the I and P field contain valid information.

If entry into SMM was caused by an I/O trap, it is useful for the programmer to know the port address, data size and data value associated with that I/O operation. This information is also saved in the header and is only valid for an I/O write operation. The I/O write information is not restored within the CPU when executing a RSM instruction.

Table 3-14. SMM Memory Space Header

Name	Description	Size
DR7	The contents of Debug Register 7.	4 Bytes
EFLAGS	The contents of Extended Flags Register.	4 Bytes
CR0	The contents of Control Register 0.	4 Bytes
Current IP	The address of the instruction executed prior to servicing SMI interrupt.	4 Bytes
Next IP	The address of the next instruction that will be executed after exiting SMM mode.	4 Bytes
CS Selector	Code segment register selector for the current code segment.	2 Bytes
CS Descriptor	Code segment register descriptor for the current code segment.	8 Bytes
S	Software SMM Entry Indicator. S = ‘1’, if current SMM is the result of an SMINT instruction. S = ‘0’, if current SMM is not the result of an SMINT instruction.	1 Bit
P	REP INSx/OUTSx Indicator. P = ‘1’ if current instruction has a REP prefix. P = ‘0’ if current instruction does not have a REP prefix.	1 Bit
I	IN, INSx, OUT, or OUTSx Indicator. I = ‘1’ if current instruction performed is an I/O WRITE. I = ‘0’ if current instruction performed is an I/O READ.	1 Bit
I/O Write Data Size	Indicates size of data for the trapped IO write. 01h = byte 03h = word 0Fh = dword	2 Bytes
I/O Write Address	Address of the trapped I/O write.	2 Bytes
I/O Write Data	Data associated with the trapped IO write.	4 Bytes
ESI or EDI	Restored ESI or EDI value. Used when it is necessary to repeat a REP OUTSx or REP INSx instruction when one of the IO cycles caused an SMI# trap.	4 Bytes
Note: INSx = INS, INSB, INSW or INSD instruction. Note: OUTSx = OUTS, OUTSB, OUTSW and OUTSD instruction.		

3.7.3 SMM Instructions

The ST486 core automatically saves the minimal amount of CPU state information when entering SMM which allows fast SMI service routine entry and exit. After entering the SMI service routine, the MOV, SVDC, SVLDT and SVTS instructions can be used to save the complete CPU state information. If the SMI service routine modifies more than what is automatically saved or forces the CPU to power down, the complete CPU state information must be saved. Since the CPU is a static device, its internal state is retained when the input clock is stopped. Therefore, an entire CPU state save is not necessary prior to stopping the input clock.

The new SMM instructions, listed in Table 3-15, can only be executed if:

SMI# is enabled and

SMAR SIZE 0 and

the Current Privilege Level (CPL) = 0 and

the SMAC bit (CCR1, bit 2) is set] or

[the Current Privilege Level (CPL) = 0 and

the CPU is in an SMI service routine (SMI# = 0)].

If the above conditions are not met and an attempt is made to execute an SVDC, RSDC, SVLDT, RSLDT, SVTS, RSTS, SMINT or RSM instruction, an invalid opcode exception is generated. These instructions can be executed outside of defined SMM space provided the above conditions are met.

Table 3-15. SMM Instruction Set

Instruction	Opcode	Format	Description
SVDC	0F 78 [mod sreg3 r/m]	SVDC mem80, sreg3	*Save Segment Register and Descriptor* Saves reg (DS, ES, FS, GS, or SS) to mem80.
RSDC	0F 79 [mod sreg3 r/m]	RSDC sreg3, mem80	*Restore Segment Register and Descriptor* Restores reg (DS, ES, FS, GS, or SS) from mem80. Use RSM to restore CS. Note: Processing “RSDC CS, Mem80” will produce an exception.
SVLDT	0F 7A [mod 000 r/m]	SVLDT mem80	*Save LDTR and Descriptor* Saves Local Descriptor Table (LDTR) to mem80.
RSLDT	0F 7B [mod 000 r/m]	RSLDT mem80	*Restore LDTR and Descriptor* Restores Local Descriptor Table (LDTR) from mem80.
SVTS	0F 7C [mod 000 r/m]	SVTS mem80	*Save TSR and Descriptor* Saves Task State Register (TSR) to mem80.
RSTS	0F 7D [mod 000 r/m]	RSTS mem80	*Restore TSR and Descriptor* Restores Task State Register (TSR) from mem80.
SMINT	0F 7E	SMINT	*Software SMM Entry* CPU enters SMM mode. CPU state information is saved in SMM memory space header and execution begins at SMM base address.
RSM	0F AA	RSM	*Resume Normal Mode* Exits SMM mode. The CPU state is restored using the SMM memory space header and execution resumes at interrupted point.
Note: mem80 = 80-bit memory location			

The SMINT instruction can be used by software to enter SMM. The CPU will not drive the SMI# output low during the software initiated SMM.

However, if the SMI# is asserted to the CPU during a software SMM, the SMI# handshake occurs normally. The hardware SMI# is serviced after the software SMM has been exited by execution of the RSM instruction.

All of the SMM instructions (except RSM and SMINT) save or restore 80 bits of data, allowing the saved values to include the hidden portion of the register contents.

3.7.4 SMM Memory Space

SMM memory space is defined by specifying the base address and size of the SMM memory space in the SMAR register. The base address must be a multiple of the SMM memory space size. For example, a 32 KByte SMM memory space must be located at a 32 KByte address boundary. The memory space size can range from 4 KBytes to 32 MBytes.

SMM memory space accesses are always non-cacheable. SMM accesses ignore the state of the A20M# input pin and drive the A20 address bit to the unmasked value.

Access to the SMM memory space can be made while not in SMM mode by setting the SMAC bit in the CCR1 register. This feature may be used to initialize the SMM memory space.

While in SMM mode, SMADS# address strobes are generated instead of ADS# for SMM memory accesses. Any memory accesses outside the defined SMM space result in normal memory accesses and ADS# strobes. Data (non-code) accesses to main memory that overlap with defined SMM memory space are allowed if MMAC in CCR1 is set. In this case, ADS# strobes are generated for data accesses only and SMADS# strobes continue to be generated for code accesses.

3.7.5 SMI Service Routine Execution

Upon entry into SMM, after the SMM header has been saved, the CR0, EFLAGS, and DR7 registers are set to their reset values. The Code Segment (CS) register is loaded with the base, as defined by the SMAR register, and a limit of 4 GBytes. The SMI service routine then begins execution at the SMM base address in real mode.

The programmer must save the value of any registers that may be changed by the SMI service routine. For data accesses immediately after entering the SMI service routine, the programmer must use CS as a segment override. I/O port access is possible during the routine but care must be taken to save registers modified by the I/O instructions. Before using a segment register, the register and the register's descriptor cache contents should be saved using the SVDC instruction. While executing in the SMM space, execution flow can transfer to normal memory locations.

Hardware interrupts, (INTRs and NMIs), may be serviced during a SMI service routine. If interrupts are to be serviced while executing in the SMM memory space, the SMM memory space must be within the 0 to 1 MByte address range to guarantee proper return to the SMI service routine after handling the interrupt.

INTRs are automatically disabled when entering SMM since the IF flag is set to its reset value. Once in SMM, the INTR can be enabled by setting the IF flag. An NMI event in SMM mode can be enabled by setting NMIEN in the CCR3 register. If NMI is not enabled while in SMM mode, the CPU latches one NMI event and services the interrupt after NMI has been enabled or after exiting SMM mode through the RSM instruction.

Within the SMI service routine, protected mode may be entered and exited as required, and real or protected mode device drivers may be called.

To exit the SMI service routine, a Resume (RSM) instruction, rather than an IRET, is executed. The RSM instruction causes the ST486 core to restore the CPU state using the SMM header information and resume execution at the interrupted point. If the full CPU state was saved by the programmer, the stored values should be reloaded prior to executing the RSM instruction using the MOV, RSDC, RSLDT and RSTS instructions.

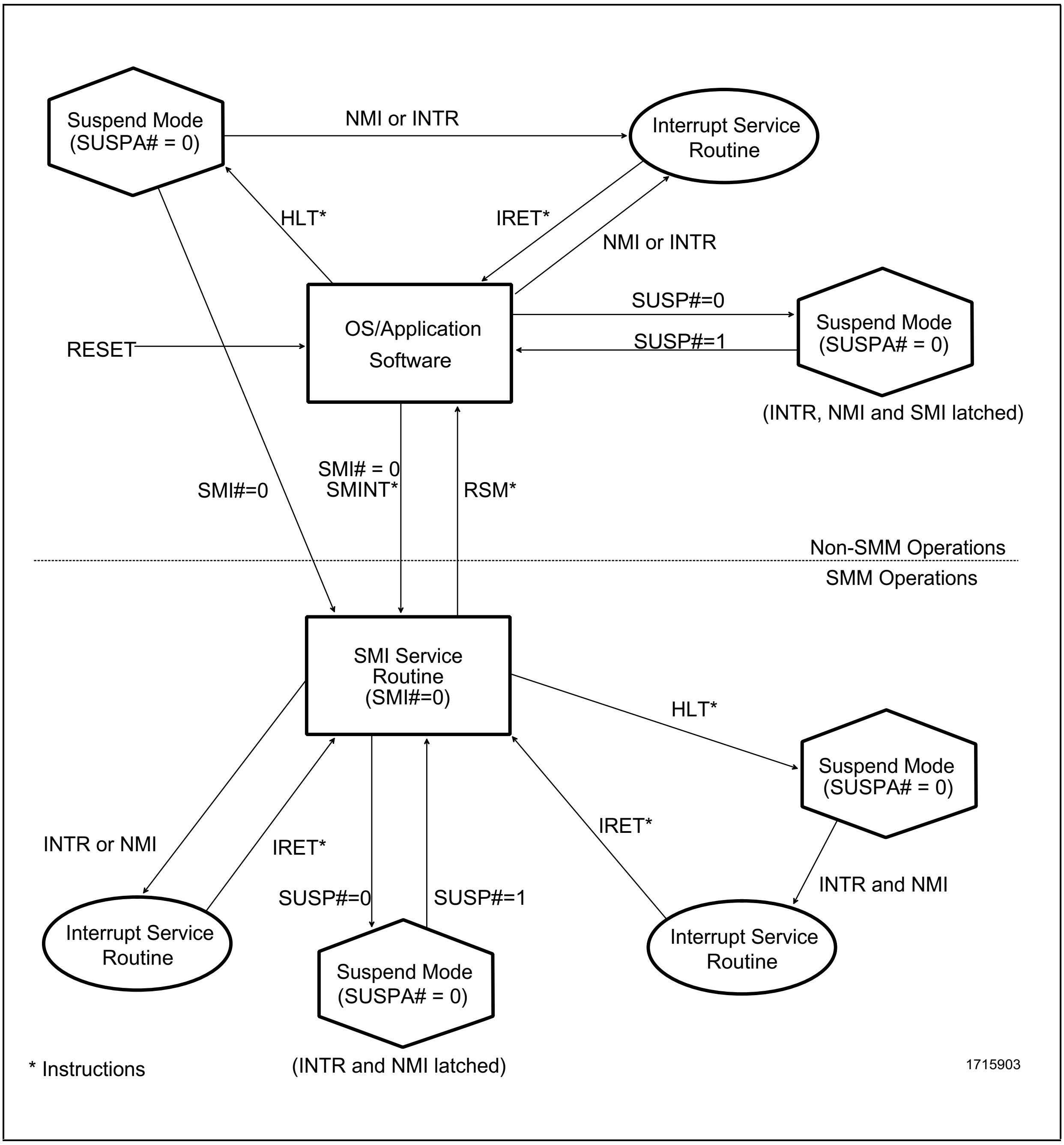
3.7.6 CPU States Related to SMM and Suspend Mode

The state diagram shown in Figure 3-32 illustrates the various CPU states associated with SMM and suspend mode. While in the SMI service routine, the ST486 core can enter suspend mode either by (1) executing a halt (HLT) instruction or (2) by asserting the SUSP# input.

During SMM operations and while in SUSP# initiated suspend mode, an occurrence of either NMI or INTR is latched. (In order for INTR to be latched, the IF flag must be set.) The INTR or NMI is serviced after exiting suspend mode.

If suspend mode is entered via a HLT instruction from the operating system or application software, the reception of an SMI# interrupt causes the CPU to exit suspend mode and enter SMM. If suspend mode is entered via the hardware (SUSP# = '0') while the operating system or application software is active, the CPU latches one occurrence of INTR, NMI and SMI#.

Figure 3-32. SMM and Suspend Mode State Diagram


Suspend Mode (SUSPA# = 0)
NMI or INTR
Interrupt Service Routine
HLT*
IRET*
NMI or INTR
OS/Application Software
RESET
SUSP#=0
SUSP#=1
Suspend Mode (SUSPA# = 0)
(INTR, NMI and SMI latched)
SMI#=0
SMI# = 0
SMINT*
RSM*
Non-SMM Operations
SMM Operations
SMI Service Routine (SMI#=0)
HLT*
Suspend Mode (SUSPA# = 0)
INTR or NMI
IRET*
IRET*
SUSP#=0
SUSP#=1
INTR and NMI
Interrupt Service Routine
Suspend Mode (SUSPA# = 0)
Interrupt Service Routine
* Instructions
(INTR and NMI latched)
1715903

3.8 Shutdown and Halt

The Halt Instruction (HLT) stops program execution and prevents the processor from using the local bus until restarted. The ST486 core then enters a low-power suspend mode if the HLT bit in CCR2 is set. SMI, NMI, INTR with interrupts enabled (IF bit in EFLAGS='1'), or RESET forces the CPU out of the halt state. If interrupted, the saved code segment and instruction pointer specify the instruction following the HLT.

Shutdown occurs when a severe error is detected that prevents further processing. An NMI input can bring the processor out of shutdown if the IDT limit is large enough to contain the NMI interrupt vector (at least 000Fh) and the stack has enough room to contain the vector and flag information (i.e., stack pointer is greater than 0005h). Otherwise, shutdown can only be exited by a processor reset.

3.9 Protection

Segment protection and page protection are safeguards built into the ST486 core protected mode architecture which deny unauthorized or incorrect access to selected memory addresses. These safeguards allow multitasking programs to be isolated from each other and from the operating system. Page protection is discussed earlier in this chapter in Section 2.4. This section concentrates on segment protection.

Selectors and descriptors are the key elements in the segment protection mechanism. The segment base address, size, and privilege level are established by a segment descriptor. Privilege levels control the use of privileged instructions, IO instructions and access to segments and segment descriptors. Selectors are used to locate segment descriptors.

Segment accesses are divided into two basic types, those involving code segments (e.g., control transfers) and those involving data accesses. The ability of a task to access a segment depends on:

- the segment type
- the instruction requesting access
- the type of descriptor used to define the segment
- the associated privilege levels (described below).

Data stored in a segment can be accessed only by code executing at the same or a more privileged level. A code segment or procedure can only be called by a task executing at the same or a less privileged level.

3.9.1 Privilege Levels

The values for privilege levels range between 0 and 3. Level 0 is the highest privilege level (most privileged), and level 3 is the lowest privilege level (least privileged). The privilege level in real mode is effectively 0.

The **Descriptor Privilege Level** (DPL) is the privilege level defined for a segment in the segment descriptor. The DPL field specifies the minimum privilege level needed to access the memory segment pointed to by the descriptor.

The **Current Privilege Level** (CPL) is defined as the current task's privilege level. The CPL of an executing task is stored in the hidden portion of the code segment register and essentially is the DPL for the current code segment.

The **Requested Privilege Level** (RPL) specifies a selector's privilege level and is used to distinguish between the privilege level of a routine actually accessing memory (the CPL), and the privilege level of the original requestor (the RPL) of the memory access. The lesser of the RPL and CPL is called the effective privilege level (EPL). Therefore, if RPL = 0 in a segment selector, the effective privilege level is always determined by the CPL. If RPL = 3, the effective privilege level is always 3 regardless of the CPL.

For a memory access to succeed, the effective privilege level (EPL) must be at least as privileged as the descriptor privilege level (EPL ≤ DPL). If the EPL is less privileged than the DPL (EPL>DPL), a general protection fault is generated. For example, if a segment has a DPL = 2, an instruction accessing the segment only succeeds if executed with an EPL ≤ 2.

3.9.2 I/O Privilege Levels

The I/O Privilege Level (IOPL) allows the operating system executing at CPL=0 to define the least privileged level at which IOPL-sensitive instructions can unconditionally be used. The IOPL-sensitive instructions include CLI, IN, OUT, INS, OUTS, REP INS, REP OUTS, and STI. Modification of the IF bit in the EFLAGS register is also sensitive to the I/O privilege level.

The IOPL is stored in the EFLAGS register. An I/O permission bit map is available as defined by the 32-bit Task State Segment (TSS). Since each task can have its own TSS, access to individual I/O ports can be granted through separate I/O permission bit maps.

If CPL ≤ IOPL, IOPL-sensitive operations can be performed. If CPL > IOPL, a general protection fault is generated if the current task is associated with a 16-bit TSS. If the current task is associated with a 32-bit TSS and CPL > IOPL, the CPU consults the I/O permission bitmap in the TSS to determine on a port-by-port basis whether or not I/O instructions (IN, OUT, INS, OUTS, REP INS, REP OUTS) are permitted, and the remaining IOPL-sensitive operations generate a general protection fault.

3.9.3 Privilege Level Transfers

A task's CPL can be changed only through intersegment control transfers using gates or task switches to a code segment with a different privilege level. Control transfers result from exception and interrupt servicing and from execution of the CALL, JMP, INT, IRET and RET instructions.

There are five types of control transfers that are summarized in Table 3-16. Control transfers can be made only when the operation causing the control transfer references the correct descriptor type. Any violation of these descriptor usage rules causes a general protection fault.

Any control transfer that changes the CPL within a task results in a change of stack. The initial values for the stack segment (SS) and stack pointer (ESP) for privilege levels 0, 1, and 2 are stored in the TSS. During a JMP or CALL control transfer, the SS and ESP are loaded with the new stack pointer and the previous stack pointer is saved on the new stack. When returning to the original privilege level, the RET or IRET instruction restores the less-privileged stack.

Table 3-16. Descriptor Types Used for Control Transfer

Type of Control Transfer	Operation Types	Descriptor Referenced	Descriptor Table
Intersegment within the same privilege level.	JMP, CALL, RET, IRET*	Code Segment	GDT or LDT
Intersegment to the same or a more privileged level. Interrupt within task (could change CPL level).	CALL	Gate Call	GDT or LDT
	Interrupt Instruction, Exception, External Interrupt	Trap or Interrupt Gate	LDT
Intersegment to a less privileged level (changes task CPL).	RET, IRET*	Code Segment	GDT or LDT
Task Switch via TSS	CALL, JMP	Task State Segment	GDT
Task Switch via Task Gate	CALL, JMP	Task Gate	GDT or LDT
	IRET**, Interrupt Instruction, Exception, External Interrupt	Task Gate	IDT
* NT (Nested Task bit in EFLAGS) = '0' ** NT (Nested Task bit in EFLAGS) = '1'			

3.9.3.1 Gates

Gate descriptors provide protection for privilege transfers among executable segments. Gates are used to transition to routines of the same or a more privileged level. Call gates, interrupt gates and trap gates are used for privilege transfers within a task. Task gates are used to transfer between tasks.

Gates conform to the standard rules of privilege. In other words, gates can be accessed by a task if the effective privilege level (EPL) is the same or more privileged than the gate descriptor's privilege level (DPL).

3.9.4 Initialization and Transition to Protected Mode

The ST486 core microprocessor switches to real mode immediately after RESET. While operating in real mode, the system tables and registers should be initialized. The GDTR and IDTR must point to a valid GDT and IDT, respectively. The size of the IDT should be at least 256 bytes, and the GDT must contain descriptors which describe the initial code and data segments.

The processor can be placed in protected mode by setting the PE bit in the CR0 register. After enabling protected mode, the CS register should be loaded and the instruction decode queue should be flushed by executing an intersegment JMP. Finally, all data segment registers should be initialized with appropriate selector values.

3.10 Virtual 8086 Mode

Both real mode and virtual 8086 (V86) mode are supported by the ST486 core CPU allowing execution of 8086 application programs and 8086 operating systems. V86 Mode allows the execution of 8086-type applications, yet still permits use of the ST486 core protection mechanism. V86 tasks run at privilege level 3. Upon entry, all segment limits are set to FFFFh (64K) as in real mode.

3.10.1 Memory Addressing

While in V86 mode, segment registers are used in an identical fashion to real mode. The contents of the segment register are multiplied by 16 and added to the offset to form the segment base linear address. The ST486 core CPU permits the operating system to select which programs use the V86 address mechanism and which programs use protected mode addressing for each task.

The ST486 core also permits the use of paging when operating in V86 mode. Using paging, the 1-MByte address space of the V86 task can be mapped to anywhere in the 4-GByte linear address space of the ST486 core CPU.

The paging hardware allows multiple V86 tasks to run concurrently, and provides protection and operating system isolation. The paging hardware must be enabled to run multiple V86 tasks or to relocate the address space of a V86 task to physical address space greater than 1 MByte.

3.10.2 Protection

All V86 tasks operate with the least amount of privilege (level 3) and are subject to all of the ST486 core protected mode protection checks. As a result, any attempt to execute a privileged instruction within a V86 task results in a general protection fault.

In V86 mode, a slightly different set of instructions are sensitive to the IO privilege level (IOPL) than in protected mode. These instructions are: CLI, INT n, IRET, POPF, PUSHF, and STI. The INT3, INTO and BOUND variations of the INT instruction are not IOPL sensitive.

3.10.3 Interrupt Handling

To fully support the emulation of an 8086-type machine, interrupts in V86 mode are handled as follows. When an interrupt or exception is serviced in V86 mode, program execution transfers to the interrupt service routine at privilege level 0 (i.e., transition from V86 to protected mode occurs) and the VM bit in the EFLAGS register is cleared. The protected mode interrupt service routine then determines if the interrupt came from a protected mode or V86 application by examining the VM bit in the EFLAGS image stored on the stack. The interrupt service routine may then choose to allow the 8086 operating system to handle the interrupt or may emulate the function of the interrupt handler. Following completion of the interrupt service routine, an IRET instruction restores the EFLAGS register (restores VM=’1’) and segment selectors and control returns to the interrupted V86 task.

3.10.4 Entering and Leaving V86 Mode

V86 mode is entered from protected mode by either executing an IRET instruction at CPL = ‘0’ or by task switching. If an IRET is used, the stack must contain an EFLAGS image with VM = ‘1’. If a task switch is used, the TSS must contain an EFLAGS image containing a 1 in the VM bit position. The POPF instruction cannot be used to enter V86 mode since the state of the VM bit is not affected. V86 mode can only be exited as the result of an interrupt or exception. The transition out must use a 32-bit trap or interrupt gate which must point to a non-conforming privilege level 0 segment (DPL = 0), or a 32-bit TSS. These restrictions are required to permit the trap handler to IRET back to the V86 program.

3.11 FPU Operations

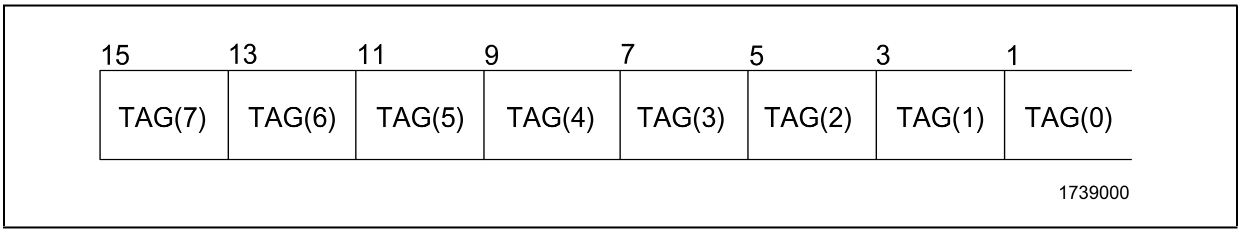
3.11.1 FPU Register Set

In addition to the registers described to this point, the FPU circuitry within the ST486 core provides the user eight data registers accessed in a stack-like manner, a control register, and a status register. The ST486 core also provides a data register tag word which improves context switching and stack performance by maintaining empty/non-empty status for each of the eight data registers. In addition, registers in the CPU contain pointers to (a) the memory location containing the current instruction word and (b) the memory location containing the operand associated with the current instruction word (if any).

FPU Tag Word Register.

The ST486 core maintains a tag word register comprised of two bits for each physical data register. Tag values are maintained transparently by the ST486 core and are only available to the programmer indirectly through the FSTENV and FSAVE instructions. The tag word with tag fields for each associated physical register, tag(n), is shown in Figure 3-33.

Figure 3-33. FPU Tag Word Register



TAGn Tag Word Fields (n=0..7). These fields provide status information for the Data register n.

TAGn	TAG value
00	Valid
01	Zero
10	Special
11	Empty

Note: Denormal, Infinity, QNaN, SNaN and unsupported formats are tagged as “Special”.

FPU Status Register.

The FPU circuitry communicates information about its status and the results of operations to the ST486 core via the FPU status register (Figure 3-34). This register is continuously accessible to the ST486 CPU core regardless of the state of the Control or Execution Units.

Figure 3-34. FPU Status Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	S	S	S	C2	C1	C0	ES	SF	P	U	O	Z	D	I

1739003

Bit 15 **B, Copy of the ES bit.** (ES is bit 7)

Bit 14 **C3, Condition code bit 3.**

Bits 13-11 **SSS, Top of stack register number.** The value held in this field points to the current TOS.

Bits 10-8 **C2-C0, Condition code bits 2-0.**

Bit 7 **ES, Error indicator.** Set to ‘1’ if an unmasked exception is detected.

Bit 6 **SF, Stack Fault or invalid register operation Flag.** Set to ‘1’ if an unmasked stack fault of invalid exception is detected.

Bit 5 **P, Precision error exception flag.** Set to ‘1’ if an unmasked precision error exception is detected.

Bit 4 **U, Underflow error exception flag.** Set to ‘1’ if an unmasked underflow exception is detected.

Bit 3 **O, Overflow error exception flag.** Set to ‘1’ if an unmasked overflow exception is detected.

Bit 2 **Z, Divide by zero exception flag.** Set to ‘1’ if an unmasked divide by zero exception is detected.

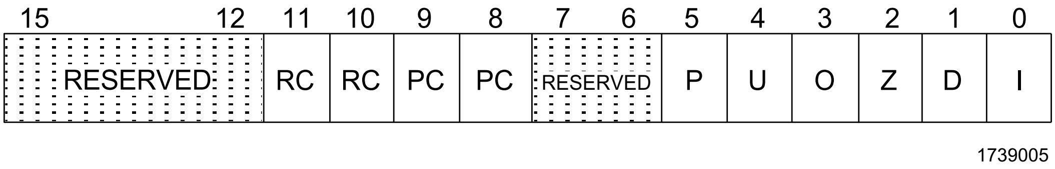
Bit 1 **D, Denormalized operand error exception flag.** Set to ‘1’ if an unmasked denormalized operand error exception is detected.

Bit 0 **I, Invalid operation exception flag.** Set to ‘1’ if an unmasked invalid operation exception is detected.

FPU Mode Control Register.

The FPU Mode Control Register (MCR) (Figure 3-35) is used by the CPU to specify the operating mode of the FPU. The MCR contains bit fields which specify the rounding mode to be used, the precision by which to calculate results, and the exception conditions which should be reported to the CPU via traps. The user controls precision, rounding, and exception reporting by setting or clearing appropriate bits in the MCR.

Figure 3-35. FPU Mode Control Register



Bits 15-12 *Reserved.*

Bits 11-10 **RC, Rounding Control bits:**

RC value	Rounding Control
00	Round to nearest or even
01	Round towards minus infinity
10	Round towards plus infinity
11	Truncate

Bits 9-8 **PC, Precision Control bits:**

PC Value	Precision Control
00	24-bit mantissa
01	Reserved
10	53-bit mantissa
11	64-bit mantissa

Bits 7-6 *Reserved.*

Bit 5 **P, Precision error exception bit mask.**

Bit 4 **U, Underflow error exception bit mask.**

Bit 3 **O, Overflow error exception bit mask.**

Bit 2 **Z, Divide by zero exception bit mask.**

Bit 1 **D, Denormalized operand error exception bit mask.**

Bit 0 **I, Invalid operation exception bit mask.**

INSTRUCTION SET

INSTRUCTION SET

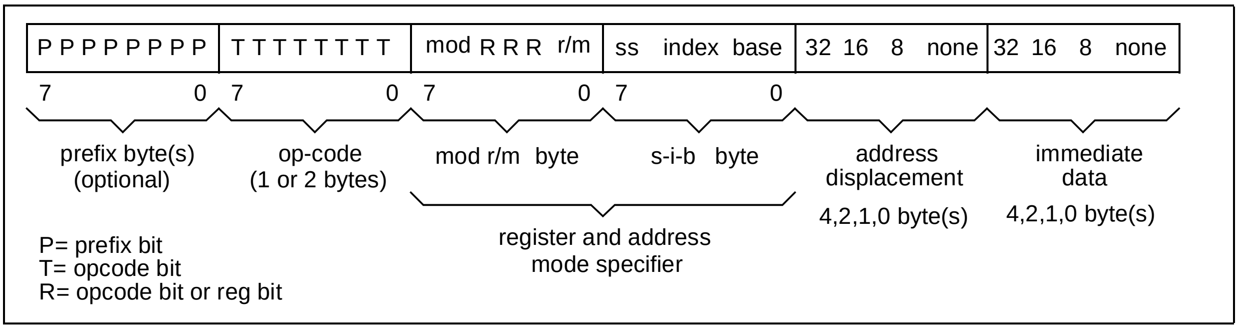
4.1 Introduction

This section summarizes the ST486 instruction set and provides detailed information on the instruction encodings. The instructions are compatible and functionally identical to those of the ST486DX/DX2 CPU. All instructions are listed in the CPU Instruction Set Summary Table (Table 13-17, Page 13-12), and the FPU Instruction Set Summary Table (Table 13-19, Page 13-28). These tables provide information on the instruction encoding, and the instruction clock counts for each instruction. The clock count values for both tables are based on the assumptions described in Section 13.3.

4.2 Instruction Set Summary

Depending on the instruction, the ST486 CPU instructions follow the general instruction format shown in Figure 4-1. These instructions vary in length and can start at any byte address. An instruction consists of one or more bytes that can include: prefix byte(s), at least one opcode byte(s), mod r/m byte, s-i-b byte, address displacement byte(s) and immediate data byte(s). An instruction can be as short as one byte and as long as 15 bytes. If there are more than 15 bytes in the instruction a general protection fault (error code of 0) is generated.

Figure 4-1. Instruction Set Format



4.3 General Instruction Fields

The fields in the general instruction format at the byte level are listed in Table 4-1.

Table 4-1. Instruction Fields

Field Name	Description	Width
Optional Prefix Byte(s)	Specifies segment register override, address and operand size, repeat elements in string instruction, LOCK# assertion.	1 or more bytes
Opcode Byte(s)	Identifies instruction operation.	1 or 2 bytes
mod and r/m Byte	Address mode specifier.	1 byte
s-i-b Byte	Scale factor, Index and Base fields.	1 byte
Address Displacement	Address displacement operand.	1, 2 or 4 bytes
Immediate data	Immediate data operand.	1, 2 or 4 bytes

4.3.1 Optional Prefix Bytes(s)

Prefix bytes can be placed in front of any instruction. The prefix modifies the operation of the next instruction only. When more than one prefix is used, the order is not important. There are five type of prefixes as follows:

1. Segment Override explicitly specifies which segment register an instruction will use for effective address calculation.
2. Address Size switches between 16- and 32-bit addressing. Selects the inverse of the default.
3. Operand Size switches between 16- and 32-bit operand size. Selects the inverse of the default.
4. Repeat is used with a string instruction which causes the instruction to be repeated for each element of the string.
5. Lock is used to assert the hardware LOCK# signal during execution of the instruction.

Table 4-2 lists the encodings for each of the available prefix bytes. The operand size and address size prefixes allow the individual overriding of the default value for operand size and effective address size. The presence of these prefixes select the opposite (non-default) operand size and/or effective address size as the case may be.

Table 4-2. Instruction Prefix Summary

Prefix	Encoding	Description
ES:	26h	Override segment default, use ES for memory operand
CS:	2Eh	Override segment default, use CS for memory operand
SS:	36h	Override segment default, use SS for memory operand
DS:	3Eh	Override segment default, use DS for memory operand
FS:	64h	Override segment default, use FS for memory operand
GS:	65h	Override segment default, use GS for memory operand
Operand Size	66h	Make operand size attribute the inverse of the default
Address Size	67h	Make address size attribute the inverse of the default
LOCK	F0h	Assert LOCK# hardware signal.
REPNE	F2h	Repeat the following string instruction.
REP/REPE	F3h	Repeat the following string instruction.

4.3.2 Opcode Byte(s)

The opcode field is either one or two bytes in length and may be further defined by additional bits in the mod r/m byte. The opcode field specifies the operation to be performed by the instruction. Some operations have more than one opcode, each specifying a different form of the operation. Some opcodes name instruction groups. For example, opcode 80h names a group of operations that have an immediate operand, and a register or memory operand. The opcodes are given in hex values unless shown within brackets ([ ]). Values within brackets are given in binary. The reg field may appear in the second opcode byte or in the mod r/m byte.

4.3.2.1 w Field

The 1-bit w field selects the operand size during 16 and 32 bit data operations.

Table 4-3. w Field Encoding

w Field	Operand Size	
	16-Bit Data Operations	32-Bit Data Operations
0	8 Bits	8 Bits
1	16 Bits	32 Bits

4.3.2.2 d Field

The d field determinds which operand is taken as the source operand and which operand is taken as the destination.

Table 4-4. d Field Encoding

d Field	Direction of Operation	Source Operand	Dest Operand
0	Register → Register or Register → Memory	reg	mod r or mod ss-index-base
1	Register → Register or Memory → Register	mod r/m or mod ss-index-base	reg

4.3.2.3 eee Field

The eee field is used to select the control, debug and test registers in the MOV instructions. The type of register and base registers selected by the eee field are listed in Table 4-5. The values shown in Table 4-5 are the only valid encodings for the eee bits.

Table 4-5. eee Field Encoding

eee Field	Register Type	Base Register
000	Control Register	CR0
010	Control Register	CR2
011	Control Register	CR3
000	Debug Register	DR0
001	Debug Register	DR1
010	Debug Register	DR2
011	Debug Register	DR3
110	Debug Register	DR6
111	Debug Register	DR7
011	Test Register	TR3
100	Test Register	TR4
101	Test Register	TR5
110	Test Register	TR6
111	Test Register	TR7

4.3.3 mod and r/m Byte

The mod and r/m fields, within the mod r/m byte, select the type of memory addressing to be used. Some instructions use a fixed addressing mode (e.g., PUSH or POP) and therefore, these fields are not present. Table 4-6 (next page) lists the addressing method when 16-bit addressing is used and a mod r/m byte is present. Some mod r/m field encodings are dependent on the w field and are shown in Table 4-7 (next page).

Table 4-6. mod r/m Field Encoding

mod and r/m fields	16-Bit Address Mode with mod r/m Byte	32-Bit Address Mode with mod r/m Byte and No s-i-b Byte Present
00 000	DS:[BX+SI]	DS:[EAX]
00 001	DS:[BX+DI]	DS:[ECX]
00 010	DS:[BP+SI]	DS:[EDX]
00 011	DS:[BP+DI]	DS:[EBX]
00 100	DS:[SI]	s-i-b is present (Section 4.3.4)
00 101	DS:[DI]	DS:[d32]
00 110	DS:[d16]	DS:[ESI]
00 111	DS:[BX]	DS:[EDI]
01 000	DS:[BX+SI+d8]	DS:[EAX+d8]
01 001	DS:[BX+DI+d8]	DS:[ECX+d8]
01 010	DS:[BP+SI+d8]	DS:[EDX+d8]
01 011	DS:[BP+DI+d8]	DS:[EBX+d8]
01 100	DS:[SI+d8]	s-i-b is present (Section 4.3.4)
01 101	DS:[DI+d8]	SS:[EBP+d8]
01 110	SS:[BP+d8]	DS:[ESI+d8]
01 111	DS:[BX+d8]	DS:[EDI+d8]
10 000	DS:[BX+SI+d16]	DS:[EAX+d32]
10 001	DS:[BX+DI+d16]	DS:[ECX+d32]
10 010	DS:[BP+SI+d16]	DS:[EDX+d32]
10 011	DS:[BP+DI+d16]	DS:[EBX+d32]
10 100	DS:[SI+d16]	s-i-b is present (Section 4.3.4)
10 101	DS:[DI+d16]	SS:[EBP+d32]
10 110	SS:[BP+d16]	DS:[ESI+d32]
10 111	DS:[BX+d16]	DS:[EDI+d32]
11000-11111	See Table 4-7	See Table 4-7

Table 4-7. mod r/m Field Encoding Dependent on w Field

mod r/m	16-Bit Operation w = 0	16-Bit Operation w = 1	32-Bit Operation w = 0	32-Bit Operation w = 1
11 000	AL	AX	AL	EAX
11 001	CL	CX	CL	ECX
11 010	DL	DX	DL	EDX
11 011	BL	BX	BL	EBX
11 100	AH	SP	AH	ESP
11 101	CH	BP	CH	EBP
11 110	DH	SI	DH	ESI
11 111	BH	DI	BH	EDI

4.3.3.1 reg Field

The reg field determines which general registers are to be used. The selected register is dependent on whether a 16- or 32-bit operation is current and the status of the w bit..

Table 4-8. reg Field

reg	16-Bit Operation w Field Not Present	32-Bit Operation w Field Not Present	16-Bit Operation w=0	16-Bit Operation w=1	32-Bit Operation w=0	32-Bit Operation w=1
000	AX	EAX	AL	AX	AL	EAX
001	CX	ECX	CL	CX	CL	ECX
010	DX	EDX	DL	DX	DL	EDX
011	BX	EBX	BL	BX	BL	EBX
100	SP	ESP	AH	SP	AH	ESP
101	BP	EBP	CH	BP	CH	EBP
110	SI	ESI	DH	SI	DH	ESI
111	DI	EDI	BH	DI	BH	EDI

4.3.3.2 sreg3 Field

The sreg3 field (Table 4-9) is 3-bit field that is similar to the sreg2 field, but allows use of the FS and GS segment registers.

Table 4-9. sreg3 Field Encoding

sreg3 Field	Segment Register Selected
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	undefined
111	undefined

4.3.3.3 sreg2 Field

The sreg2 field (Table 4-10) is a 2-bit field that allows one of the four 286-type segment registers to be specified.

Table 4-10. sreg2 Field Encoding

sreg2 Field	Segment Register Selected
00	ES
01	CS
10	SS
11	DS

4.3.4 s-i-b Byte

The s-i-b fields provide scale factor, indexing and a base field for address selection.

4.3.4.1 ss Field

The ss field (Table 4-11) specifies the scale factor used in the offset mechanism for address calculation. The scale factor multiplies the index value to provide one of the components used to calculate the offset address.

Table 4-11. ss Field Encoding

ss Field	Scale Factor
00	x1
01	x2
01	x4
11	x8

4.3.4.2 Index Field

The index field (Table 4-12) specifies the index register used by the offset mechanism for offset address calculation. When no register is used (index field = 100), the ss value must be 00 or the effective address is undefined

4.3.4.3 Base Field

In Table 4-6, the note "s-i-b present" for certain entries forces the use of the mod and base field as listed in Table 4-13. The first two digits in the first column of Table 4-13 identifies the mod bits in the mod r/m byte. The last three digits in the first column of this table identifies the base fields in the s-i-b byte..

Table 4-12. index Field Encoding

Index Field	Index Register
000	EAX
001	ECX
010	EDX
011	EBX
100	none
101	EBP
110	ESI
111	EDI

Table 4-13. mod base Field Encoding

mod Field within mode/rm Byte	base Field within s-i-b Byte	32-Bit Address Mode with mod r/m and s-i-b Bytes Present
00	000	DS:[EAX+(scaled index)]
00	001	DS:[ECX+(scaled index)]
00	010	DS:[EDX+(scaled index)]
00	011	DS:[EBX+(scaled index)]
00	100	SS:[ESP+(scaled index)]
00	101	DS:[d32+(scaled index)]
00	110	DS:[ESI+(scaled index)]
00	111	DS:[EDI+(scaled index)]
01	000	DS:[EAX+(scaled index)+d8]
01	001	DS:[ECX+(scaled index)+d8]
01	010	DS:[EDX+(scaled index)+d8]
01	011	DS:[EBX+(scaled index)+d8]
01	100	SS:[ESP+(scaled index)+d8]
01	101	SS:[EBP+(scaled index)+d8]
01	110	DS:[ESI+(scaled index)+d8]
01	111	DS:[EDI+(scaled index)+d8]
10	000	DS:[EAX+(scaled index)+d32]
10	001	DS:[ECX+(scaled index)+d32]
10	010	DS:[EDX+(scaled index)+d32]
10	011	DS:[EBX+(scaled index)+d32]
10	100	SS:[ESP+(scaled index)+d32]
10	101	SS:[EBP+(scaled index)+d32]
10	110	DS:[ESI+(scaled index)+d32]
10	111	DS:[EDI+(scaled index)+d32]

4.4 Instruction Set Tables

The ST486 instruction set is presented in two tables, the CPU Instruction Set (Table 13-17) and the FPU Clock Count (Table 13-18). Additional information concerning the FPU Clock Count Table is presented on page 13-28.

4.4.1 Assumptions Made in Determining Instruction Clock Count

The following assumptions have been made in presenting the clock count values for the individual instructions:

1. All clock counts refer to the internal CPU internal clock frequency.

2. The instruction has been prefetched, decoded and is ready for execution.

3. Bus cycles do not require wait states.

4. There are no local bus HOLD requests delaying processor access to the bus.

5. No exceptions are detected during instruction execution.

6. If an effective address is calculated, it does not use two general register components. One register, scaling and displacement can be used within the clock count shown. However, if the effective address calculation uses two general register components, add one clock to the clock count shown.

7. All clock counts assume aligned 32-bit memory/IO operands.

8. If instructions access a 32-bit operand on odd addresses, add one clock for read or write and add two clocks for read and write.

9. For non-cached memory accesses, add two clocks, assuming zero wait state memory accesses.

10. Locked cycles are not cacheable. Therefore, using the LOCK prefix with an instruction adds additional clocks as specified in paragraph 9 above.

4.4.2 CPU Instruction Set Summary Table Abbreviations

The clock counts listed in the CPU Instruction Set Summary Table are grouped by operating mode and whether there is a register/cache hit or a cache miss. In some cases, more than one clock count is shown in a column for a given instruction, or a variable is used in the clock count. The abbreviations used for these conditions are listed in Table 4-14..

Table 4-14. CPU Clock Count Abbreviations

Clock Count SymbolL	Explanation
/	Register operand/memory operand.
n	Number of times operation is repeated.
L	Level of the stack frame.
|	Conditional jump taken | Conditional jump not taken. (e.g. “4|1“ = 4 clocks if jump taken, 1 clock if jump not taken)
\	CPL ≤ IOPL \ CPL IOPL (where CPL = Current Privilege Level, IOPL = I/O Privilege Level)

4.4.3 CPU Instruction Set Summary Table Flags Table

The CPU Instruction Set Summary Table lists nine flags that are affected by the execution of instructions. The conventions shown in Table 4-15 are used to identify the different flags.

Table 4-15. Flag Abbreviations

Abbreviation	Name of Flag
OF	Overflow Flag
DF	Direction Flag
IF	Interrupt Enable Flag
TF	Trap Flag
SF	Sign Flag
ZF	Zero Flag
AF	Auxiliary Flag
PF	Parity Flag
CF	Carry Flag

Table 4-16 lists the conventions used to indicate what action the instruction has on the particular flag.

Table 4-16. Action of Instruction on Flag

Instruction Table Symbol	Action
x	Flag is modified by the instruction.
-	Flag is not changed by the instruction.
0	Flag is reset to “0”.
1	Flag is set to “1”.

Table 4-17. Instruction Set Summary

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
AAA *ASCII Adjust AL after Add*	37	-	-	-	-	-	-	x	-	x	4	4		
AAD *ASCII Adjust AX before Divide*	D5 0A	-	-	-	-	x	x	-	x	-	4	4		
AAM *ASCII Adjust AX after Multiply*	D4 0A	-	-	-	-	x	x	-	x	-	16	16		
AAS *ASCII Adjust AL after Subtract*	3F	-	-	-	-	-	-	x	-	x	4	4		
ADC *Add with Carry*		x	-	-	-	x	x	x	x	x			b	h
Register to Register	1 [00dw] [11 reg r/m]										1			
Register to Memory	1 [000w] [mod reg r/m]										3			
Memory to Register	1 [001w] [mod reg r/m]										3			
Immediate to Register/Memory	8 [00sw] [mod 010 r/m]#										1/3			
Immediate to Accumulator	1 [010w] #										1			
ADD *Integer Add*		x	-	-	-	x	x	x	x	x			b	h
Register to Register	0 [00dw] [11 reg r/m]										1	1		
Register to Memory	0 [000w] [mod reg r/m]										3	3		
Memory to Register	0 [001w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8 [00sw] [mod 000 r/m]#										1/3	1/3		
Immediate to Accumulator	0 [010w] #										1	1		
AND *Boolean AND*		0	-	-	-	x	x	-	x	0				h
Register to Register	2 [00dw] [11 reg r/m]										1	1		
Register to Memory	2 [000w] [mod reg r/m]										3	3		
Memory to Register	2 [001w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8 [00sw] [mod 100 r/m]#										1/3	1/3		
Immediate to Accumulator	2 [010w] #										1	1		
ARPL *Adjust Requested Privilege Level*		-	-	-	-	-	x	-	-	-			a	h
From Register/Memory	63 [mod reg r/m]											6/10		
BOUND *Check Array Boundaries*	62 [mod reg r/m]	-	-	-	-	-	-	-	-	-			b,e	g,h,j,k,r
If Out of Range (Int 5)											11+int	11+int		
If In Range											11	11		
BSF *Scan Bit Forward*		-	-	-	-	-	x	-	-	-			b	h
Register/Memory, Register	0F BC [mod reg r/m]										5/7+n	5/7+n		
BSR *Scan Bit Reverse*		-	-	-	-	-	x	-	-	-			b	h
Register/Memory, Register	0F BC [mod reg r/m]										5/7+n	5/7+n		
BSWAP *Byte Swap*	0F C[1 reg]	-	-	-	-	-	-	-	-	-	4	4		
BT *Test Bit*		-	-	-	-	-	-	-	-	x				
Register/Memory, Immediate	0F BA [mod 100 r/m]#										3/4	3/4		
Register/Memory, Register	0F A3 [mod reg r/m]											3/6		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
BTC *Test Bit and Complement*		-	-	-	-	-	-	-	-	x			b	h
Register/Memory, Immediate	0F BA [mod 111 r/m]#										4/5	4/5		
Register/Memory, Register	0F BB [mod reg r/m]										5/8	5/8		
BTR *Test Bit and Reset*		-	-	-	-	-	-	-	-	x			b	h
Register/Memory, Immediate	0F BA [mod 110 r/m]#										4/5	4/5		
Register/Memory, Register	0F B3 [mod reg r/m]										5/8	5/8		
BTS *Test Bit and Set*		-	-	-	-	-	-	-	-	x			b	h
Register/Memory	0F BA [mod 101 r/m]										3/5	3/5		
Register (short form)	0F AB [mod reg r/m]										4/7	4/7		
CALL *Subroutine Call*		-	-	-	-	-	-	-	-	-			b	h,j,k,r
Direct Within Segment	E8 +++										7	7		
Register/Memory Indirect Within Segment	FF [mod 010 r/m]										8/9	8/9		
Direct Intersegment	9A [unsigned full offset, selector]										12	30		
Call Gate to Same Privilege												41		
Call Gate to Different Privilege No P												83		
Call Gate to Different Privilege x P’s												81+4x		
16-bit Task to 16-bit TSS												235		
16-bit Task to 32-bit TSS												262		
16-bit Task to V86 Task												179		
32-bit Task to 16-bit TSS												238		
32-bit Task to 32-bit TSS												265		
32-bit Task to V86 Task												182		
Indirect Intersegment	FF [mod 011 r/m]										14	14		
Call Gate to Same Privilege												43		
Call Gate to Different Privilege No P												85		
Call Gate to Different Privilege Level x P’s												86+4x		
16-bit Task to 16-bit TSS												237		
16-bit Task to 32-bit TSS												264		
16-bit Task to V86 Task												181		
32-bit Task to 16-bit TSS												240		
32-bit Task to 32-bit TSS												267		
32-bit Task to V86 Task												184		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
CBW *Convert Byte to Word*	98	-	-	-	-	-	-	-	-	-	3	3		
CDQ *Convert Doubleword to Quadword*	99	-	-	-	-	-	-	-	-	-	1	1		
CLC *Clear Carry Flag*	F8	-	-	-	-	-	-	-	-	0	1	1		
CLD *Clear Direction Flag*	FC	-	0	-	-	-	-	-	-	-	1	1		
CLI *Clear Interrupt Flag*	FA	-	-	0	-	-	-	-	-	-	7	7		m
CLTS *Clear Task Switched Flag*	0F 06	-	-	-	-	-	-	-	-	-	5	5	c	l
CMC *Complement the Carry Flag*	F5	-	-	-	-	-	-	-	-	x	1	1		
CMP *Compare Integers*		x	-	-	-	x	x	x	x	x				
Register to Register	3 [10dw] [11 reg r/m]										1	1	b	h
Register to Memory	3 [101w] [mod reg r/m]										3	3		
Memory to Register	3 [100w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8 [00sw] [mod 111 r/m] #										1/3	1/3		
Immediate to Accumulator	3 [110w] #										1	1		
CMPS *Compare String*	A [011w]	x	-	-	-	x	x	x	x	x	7	7	b	h
CMPXCHG *Compare and Exchange*		x	-	-	-	x	x	x	x	x				
Register1, Register2	0F B [000w] [11 reg2 reg1]										5	5		
Memory, Register	0F B [000w] [mod reg r/m]										7	7		
CWD *Convert Word to Doubleword*	99	-	-	-	-	-	-	-	-	-	1	1		
CWDE *Convert Word to Doubleword*	98	-	-	-	-	-	-	-	-	-	3	3		
DAA *Decimal Adjust AL after Add*	27	-	-	-	-	x	x	x	x	x	4	4		
DAS *Decimal Adjust AL after Subtract*	2F	-	-	-	-	x	x	x	x	x	4	4		
DEC *Decrement by 1*		x	-	-	-	x	x	x	x	-			b	h
Register/Memory	F [111w] [mod 001 r/m]										1/3	1/3		
Register (short form)	4 [1 reg]										1	1		
DIV *Unsigned Divide*	F [011w] [mod 110 r/m]	-	-	-	-	-	-	-	-	-			b,e	e,h
Accumulator by Register/Memory											14/15	14/15		
Divisor: Byte											22/23	22/23		
Word											38/39	38/39		
Doubleword														
ENTER *Enter New Stack Frame*	C8 ++[8-bit Level]	-	-	-	-	-	-	-	-	-			b	h
Level = 0											7	7		
Level = 1											10	10		
Level (L) > 1											6+4*L	6+4*L		
HLT *Halt*	F4	-	-	-	-	-	-	-	-	-	3	3		l

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
IDIV *Integer (Signed) Divide*	F [011w] [mod 111 r/m]	-	-	-	-	-	-	-	-	-			b,e	e,h
Accumulator by Register/Memory														
Divisor: Byte											19/20	19/20		
Word											27/28	27/28		
Doubleword											43/44	43/44		
IMUL *Integer (Signed) Multiply*		x	-	-	-	-	-	-	-	x			b	h
Accumulator by Register/Memory	F [011w] [mod 101 r/m]													
Multiplier: Byte											3/5	3/5		
Word											3/5	3/5		
Doubleword											7/9	7/9		
Register with Register/Memory	0F AF [mod reg r/m]													
Multiplier: Byte											3/5	3/5		
Word											3/5	3/5		
Doubleword											7/9	7/9		
Register/Memory with Immediate to Register2	6 [10s1] [mod reg r/m] #													
Multiplier: Byte											3/5	3/5		
Word											3/5	3/5		
Doubleword											7/9	7/9		
IN *Input from I/O Port*		-	-	-	-	-	-	-	-	-				m
Fixed Port	E [010w] [port number]										16	6/19		
Variable Port	E [110w]										16	6/19		
INC *Increment by 1*		x	-	-	-	x	x	x	x	-			b	h
Register/Memory	F [111w] [mod 000 r/m]										1/3	1/3		
Register (short form)	4 [0 reg]										1	1		
INS *Input String from I/O Port*	6 [110w]	-	-	-	-	-	-	-	-	-	20	6/19	b	h,m
INT *Software Interrupt*		-	x	0	-	-	-	-	-	-			b,e	g,j,k,r
INT i	CD [i]										14			
Protected Mode:														
Interrupt or Trap to Same Privilege												49		
Interrupt or Trap to Different Privilege												77		

Continued on next page...

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
INT *Software Interrupt* **(Continued)**		-	x	0	-	-	-	-	-	-			b,e	g,j,k,r
16-bit Task to 16-bit TSS by Task Gate												233		
16-bit Task to 32-bit TSS by Task Gate												260		
16-bit Task to V86 by Task Gate												177		
16-bit Task to 16-bit TSS by Task Gate												236		
32-bit Task to 32-bit TSS by Task Gate												263		
32-bit Task to V86 by Task Gate												180		
V86 to 16-bit TSS by Task Gate												236		
V86 to 32-bit TSS by Task Gate												263		
V86 to Privilege 0 by Trap Gate/Int Gate												93		
INT 3	CC										14			
Protected Mode:														
Interrupt or Trap to Same Privilege												49		
Interrupt or Trap to Different Privilege												77		
16-bit Task to 16-bit TSS by Task Gate												233		
16-bit Task to 32-bit TSS by Task Gate												260		
16-bit Task to V86 by Task Gate												177		
32-bit Task to 16-bit TSS by Task Gate												236		
32-bit Task to 32-bit TSS by Task Gate												180		
32-bit Task to V86 by Task Gate												236		
V86 to 16-bit TSS by Task Gate												263		
V86 to 32-bit TSS by Task Gate												93		
V86 to Privilege 0 by Trap Gate/Int Gate														
INT 0	CE	-	x	0	-	-	-	-	-	-		1		
If OF==0											1			
If OF==1 (INT 4)											15			
Protected Mode:												49		
Interrupt or Trap to Same Privilege												77		
Interrupt or Trap to Different Privilege												233		
16-bit Task to 16-bit TSS by Task Gate												260		
16-bit Task to 32-bit TSS by Task Gate												177		

Continued on next page...

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
INT *Software Interrupt* **(Continued)**		-	x	0	-	-	-	-	-	-			b,e	g,j,k,r
16-bit Task to V86 by Task Gate												236		
32-bit Task to 16-bit TSS by Task Gate												263		
32-bit Task to 32-bit TSS by Task Gate											180			
32-bit Task to V86 by Task Gate											236			
V86 to 16-bit TSS by Task Gate											263			
V86 to 32-bit TSS by Task Gate											93			
V86 to Privilege 0 by Trap Gate/Int Gate,														
INVD *Invalidate Cache*	0F 08	-	-	-	-	-	-	-	-	-	4	4		
INVLPG *Invalidate TLB Entry*	0F 01 [mod 111 r/m]	-	-	-	-	-	-	-	-	-	4	4		
RET *Interrupt Return*	CF	x	x	x	x	x	x	x	x	x				g,h,j,k,r
Real Mode											14			
Protected Mode:														
Within Task to Same Privilege												31		
Within Task to Different Privilege												66		
16-bit Task to 16-bit Task												229		
16-bit Task to 32-bit TSS												256		
16-bit Task to V86 Task												173		
32-bit Task to 16-bit TSS												232		
32-bit Task to 32-bit TSS												259		
32-bit Task to V86 Task												176		
JB/JNAE/JC *Jump on Below/Not Above or Equal/Carry*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	72 +										4 | 1	6 | 1		
Full Displacement	0F 82 +++										4 | 1	6 | 1		
JBE/JNA *Jump on Below or Equal/Not Above*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	76 +										4 | 1	6 | 1		
Full Displacement	0F 86 +++										4 | 1	6 | 1		
JCXZ *Jump on CX Zero*	E3 +	-	-	-	-	-	-	-	-	-	7 | 3	7 | 3		r
JE/JZ *Jump on Equal/Zero*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	74 +										4 | 1	6 | 1		
Full Displacement	0F 84 +++										4 | 1	6 | 1		
JECXZ *Jump on ECX Zero*	E3 +	-	-	-	-	-	-	-	-	-	7 | 3	7 | 3		r

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
JL/JNGE *Jump on Less/Not Greater or Equal*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	7C +										4 | 1	6 | 1		
Full Displacement	0F 8C +++										4 | 1	6 | 1		
JLE/JNG *Jump on Less or Equal/Not Greater*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	7E +										4 | 1	6 | 1		
Full Displacement	0F 8E +++										4 | 1	6 | 1		
JMP *Unconditional Jump*		-	-	-	-	-	-	-	-	-			b	h,j,k,r
Short	EB +										4	6		
Direct within Segment	E9 +++										4	6		
Register/Memory Indirect Within Segment	FF [mod 100 r/m]										6/8	6/8		
Direct Intersegment	EA [full offset, selector]										9	26		
Call Gate Same Privilege Level												37		
16-bit Task to 16-bit TSS												238		
16-bit Task to 32-bit TSS												265		
16-bit Task to V86 Task												182		
32-bit Task to 16-bit TSS												241		
32-bit Task to 32-bit TSS												268		
32-bit Task to V86 Task												185		
Indirect Intersegment	FF [mod 101 r/m]										11	30		
Call Gate Same Privilege Level												39		
16-bit Task to 16-bit TSS												240		
16-bit Task to 32-bit TSS												267		
16-bit Task to V86 Task												184		
32-bit Task to 16-bit TSS												243		
32-bit Task to 32-bit TSS												270		
32-bit Task to V86 Task												187		
JNB/JAE/JNC *Jump on Not Below/Above or Equal/Not Carry*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	73 +										4 | 1	6 | 1		
Full Displacement	0F 83 +++										4 | 1	6 | 1		
JNBE/JA *Jump on Not Below or Equal/Above*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	77 +										4 | 1	6 | 1		
Full Displacement	0F 87 +++										4 | 1	6 | 1		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
JNE/JNZ *Jump on Not Equal/Not Zero*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	75 +										4 | 1	6 | 1		
Full Displacement	0F 85 +++										4 | 1	6 | 1		
JNL/JGE *Jump on Not Less/Greater or Equal*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	7D +										4 | 1	6 | 1		
Full Displacement	0F 8D +++										4 | 1	6 | 1		
JNLE/JG *Jump on Not Less or Equal/Greater*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	7F +										4 | 1	6 | 1		
Full Displacement	0F 8F +++										4 | 1	6 | 1		
JNO *Jump on Not Overflow*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	71 +										4 | 1	6 | 1		
Full Displacement	0F 81 +++										4 | 1	6 | 1		
JNP/JPO *Jump on Not Parity/Parity Odd*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	7B +										4 | 1	6 | 1		
Full Displacement	0F 8B +++										4 | 1	6 | 1		
JNS Jump on Not Sign		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	79 +										4 | 1	6 | 1		
Full Displacement	0F 89 +++										4 | 1	6 | 1		
JO Jump on Overflow		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	70 +										4 | 1	6 | 1		
Full Displacement	0F 80 +++										4 | 1	6 | 1		
JP/JPE *Jump on Parity/Parity Even*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	7A +										4 | 1	6 | 1		
Full Displacement	0F 8A +++										4 | 1	6 | 1		
JS *Jump on Sign*		-	-	-	-	-	-	-	-	-				r
8-bit Displacement	78 +										4 | 1	6 | 1		
Full Displacement	0F 88 +++										4 | 1	6 | 1		
LAHF *Load AH with Flags*	9F	-	-	-	-	-	-	-	-	-	2	2		
LAR *Load Access Rights*		-	-	-	-	-	x	-	-	-			a	g,h,j,p
From Register/Memory	0F 02 [mod reg r/m]											11/12		
LDS *Load Pointer to DS*	C5 [mod reg r/m]	-	-	-	-	-	-	-	-	-	6	19	b	h,i,j
LEA *Load Effective Address*	8D [mod reg r/m]	-	-	-	-	-	-	-	-	-				
No Index Register											2	2		
With Index Register											3	3		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
LEAVE *Leave Current Stack Frame*	C9	-	-	-	-	-	-	-	-	-	3	3	b	h
LES *Load Pointer to ES*	C4 [mod reg r/m]	-	-	-	-	-	-	-	-	-	6	19	b	h,i,j
LFS *Load Pointer to FS*	0F B4 [mod reg r/m]	-	-	-	-	-	-	-	-	-	6	19	b	h,i,j
LGDT *Load GDT Register*	0F 01 [mod 010 r/m]	-	-	-	-	-	-	-	-	-	9	9	b,c	h,l
LGS *Load Pointer to GS*	0F B5 [mod reg r/m]	-	-	-	-	-	-	-	-	-	6	19	b	h,i,j
LIDT *Load IDT Register*	0F 01 [mod 011 r/m]	-	-	-	-	-	-	-	-	-	9	9	b,c	h,l
LLDT *Load LDT Register*		-	-	-	-	-	-	-	-	-			a	g,h,j,l
From Register/Memory	0F 00 [mod 010 r/m]											16/17		
LMSW *Load Machine Status Word*		-	-	-	-	-	-	-	-	-			b,c	h,l
From Register/Memory	0F 01 [mod 110 r/m]										5	5		
LODS *Load String*	A [110 w]	-	-	-	-	-	-	-	-	-	4	4	b	h
LOOP *Offset Loop/No Loop*	E2 +	-	-	-	-	-	-	-	-	-	7 | 3	9 | 3		r
LOOPNZ/LOOPNE *Offset*	E0 +	-	-	-	-	-	-	-	-	-	7 | 3	9 | 3		r
LOOPZ/LOOPE *Offset*	E1 +	-	-	-	-	-	-	-	-	-	7 | 3	9 | 3		r
LSL *Load Segment Limit*		-	-	-	-	-	x	-	-	-		14/15	a	g,h,j,p
From Register/Memory	0F 03 [mod reg r/m]													
LSS *Load Pointer to SS*	0F B2 [mod reg r/m]	-	-	-	-	-	-	-	-	-	6	19	a	h,i,j
LTR *Load Task Register*		-	-	-	-	-	-	-	-	-			a	g,h,j,l
From Register/Memory	0F 00 [mod reg r/m]											16/17		
MOV *Move Data*		-	-	-	-	-	-	-	-	-			b	h,i,j
Register to Register/Memory	8 [100w] [mod reg r/m]										1/2	1/2		
Register/Memory to Register	8 [101w] [mod reg r/m]										1/2	1/2		
Immediate to Register/Memory	C [011w] [mod 000 r/m] #										1/2	1/2		
Immediate to Register (short form)	B [w reg] #										1	1		
Memory to Accumulator (short form)	A [000w] +++										2	2		
Accumulator to Memory (short form)	A [001w] +++										1/2	1/2		
Register/Memory to Segment Register	8E [mod sreg3 r/m]										2/3	15/16		
Segment Register to Register/Memory	8C [mod reg r/m]										1/2	1/2		
MOV *Move to/from Control/Debug/Test Regs*		-	-	-	-	-	-	-	-	-				l
Register to CR0/CR2/CR3	0F 22 [11 eee reg]										11/3/3	11/3/3		
CR0/CR2/CR3 to Register	0F 20 [11 eee reg]										1/3/3	1/3/3		

Continued on next page...

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
MOV *Move to/from Control/Debug/Test Regs*		-	-	-	-	-	-	-	-	-				l
Register to DR0-DR3	0F 23 [11 eee reg]										1	1		
DR0-DR3 to Register	0F 21 [11 eee reg]										3	3		
Register to DR6-DR7	0F 23 [11 eee reg]										1	1		
DR6-DR7 to Register	0F 21 [11 eee reg]										3	3		
Register to TR3-5	0F 26 [11 eee reg]										5	5		
TR3-5 to Register	0F 24 [11 eee reg]										5	5		
Register to TR6-TR7	0F 26 [11 eee reg]										1	1		
TR6-TR7 to Register	0F 24 [11 eee reg]										3	3		
MOVS *Move String*	A [010w]	-	-	-	-	-	-	-	-	-	5	5	b	h
MOVSX *Move with Sign Extension*		-	-	-	-	-	-	-	-	-			b	h
Register from Register/Memory	0F B[111w] [mod reg r/m]										1/3	1/3		
MOVZX *Move with Zero Extension*		-	-	-	-	-	-	-	-	-			b	h
Register from Register/Memory	0F B[011w] [mod reg r/m]										2/3	2/3		
MUL *Unsigned Multiply*	F [011w] [mod 100 r/m]	x	-	-	-	-	-	-	-	x			b	h
Accumulator with Register/Memory														
Multiplier - Byte											3/5	3/5		
- Word											3/5	3/5		
- Doubleword											7/9	7/9		
NEG *Negate Integer*	F [011w] [mod 011 r/m]	x	-	-	-	x	x	x	x	x			b	h
NOP *No Operation*	90	-	-	-	-	-	-	-	-	-	1	1		
NOT *Boolean Complement*	F [011w] [mod 010 r/m]	-	-	-	-	-	-	-	-	-	1/3	1/3	b	h
OR *Boolean OR*		0	-	-	-	x	x	x	x	0			b	h
Register to Register	0 [10dw] [11 reg r/m]										1	1		
Register to Memory	0 [100w] [mod reg r/m]										3	3		
Memory to Register	0 [101w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8 [000w] [mod 001 r/m] #										1/3	1/3		
Immediate to Accumulator	0 [110w] #										1	1		
OUT *Output to Port*		-	-	-	-	-	-	-	-	-				m
Fixed Port	E [011w] [port number]										18	4\17		
Variable Port	E [111w]										18	4\17		
OUTS *Output String*	6 [111w]	-	-	-	-	-	-	-	-	-	20	6\19	b	h,m

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
POP *Pop Value off Stack*		-	-	-	-	-	-	-	-	-			b	h,i,j
Register/Memory	8F [mod 000 r/m]										3/5	3/5		
Register (short form)	5 [1 reg]										3	3		
Segment Register (ES, CS, SS, DS)	[000 sreg2 110]										4	18		
Segment Register (ES, CS, SS, DS, FS, GS)	0F [10 sreg3 001]										4	18		
POPA *Pop All General Registers*	61	-	-	-	-	-	-	-	-	-	18	18	b	h
POPF *Pop Stack into FLAGS*	9D	x	x	x	x	x	x	x	x	x	4	4	b	h,n
PREFIX BYTES		-	-	-	-	-	-	-	-	-				
Assert Hardware LOCK Prefix	F0													m
Address Size Prefix	67													
Operand Size Prefix	66													
Segment Override Prefix														
CS	2E													
DS	3E													
ES	26													
FS	64													
GS	65													
SS	36													
PUSH *Push Value onto Stack*		-	-	-	-	-	-	-	-	-			b	h
Register/Memory	FF [mod 110 r/m]										2/4	2/4		
Register (short form)	5 [0 reg]										2	2		
Segment Register (ES, CS, SS, DS)	[000 sreg2 110]										2	2		
Segment Register (ES, CS, SS, DS, FS, GS)	0F [10 sreg3 000]										2	2		
Immediate	6 [10s0] #										2	2		
PUSHA *Push All General Registers*	60	-	-	-	-	-	-	-	-	-	17	17	b	h
PUSHF *Push FLAGS Register*	9C	-	-	-	-	-	-	-	-	-	2	2	b	h
RCL *Rotate Through Carry Left*		x	-	-	-	-	-	-	-	x			b	h
Register/Memory by 1	D [000w] [mod 010 r/m]										9/9	9/9		
Register/Memory by CL	D [001w] [mod 010 r/m]										9/9	9/9		
Register/Memory by Immediate	C [000w] [mod 010 r/m] #										9/9	9/9		
RCR *Rotate Through Carry Right*		x	-	-	-	-	-	-	-	x			b	h
Register/Memory by 1	D [000w] [mod 011 r/m]										9/9	9/9		
Register/Memory by CL	D [001w] [mod 011 r/m]										9/9	9/9		
Register/Memory by Immediate	C [000w] [mod 011 r/m] #										9/9	9/9		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
REP INS *Input String*	F2 6[110w]	-	-	-	-	-	-	-	-	-	20+9n	5+9n\ 18+9n	b	h,m
REP LODS *Load String*	F2 A[110w]	-	-	-	-	-	-	-	-	-	4+5n	4+5n	b	h
REP MOVS *Move String*	F2 A[010w]	-	-	-	-	-	-	-	-	-	5+4n	5+4n	b	h
REP OUTS *Output String*	F2 6[111w]	-	-	-	-	-	-	-	-	-	20+4n	5+4n\ 18+4n		b
REP STOS *Store String*	F2 A[101w]	-	-	-	-	-	-	-	-	-	3+4n	3+4n	b	h
REPE CMPS *Compare String* (Find non-match)	F3 A[011w]	x	-	-	-	x	x	x	x	x	5+8n	5+8n	b	h
REPE SCAS *Scan String* (Find non-AL/AX/EAX)	F3 A[111w]	x	-	-	-	x	x	x	x	x	4+5n	4+5n	b	h
REPNE CMPS *Compare String* (Find match)	F2 A[011w]	x	-	-	-	x	x	x	x	x	5+8n	5+8n	b	h
REPNE SCAS *Scan String* (Find AL/AX/EAX)	F2 A[111w]	x	-	-	-	x	x	x	x	x	4+5n	4+5n	b	h
RET *Return from Subroutine*		-	-	-	-	-	-	-	-	-			b	g,h,j,k,r
Within Segment	C3										10	10		
Within Segment Adding Immediate to SP	C2 ++										10	10		
Intersegment	CB										13	26		
Intersegment Adding Immediate to SP	CA ++										13	26		
Protected Mode: Different Privilege Level														
Intersegment												61		
Intersegment Adding Immediate to SP												61		
ROL *Rotate Left*		x	-	-	-	-	-	-	-	x			b	h
Register/Memory by 1	D[000w] [mod 000 r/m]										2/4	2/4		
Register/Memory by CL	D[001w] [mod 000 r/m]										3/5	3/5		
Register/Memory by Immediate	C[000w] [mod 000 r/m] #										2/4	2/4		
ROR *Rotate Right*		x	-	-	-	-	-	-	-	x			b	h
Register/Memory by 1	D[000w] [mod 001 r/m]										2/4	2/4		
Register/Memory by CL	D[001w] [mod 001 r/m]										3/5	3/5		
Register/Memory by Immediate	C[000w] [mod 001 r/m] #										2/4	2/4		
RSDC *Restore Segment Register and Descriptor*	0F 79 [mod sreg3 r/m]	-	-	-	-	-	-	-	-	-	10	10	s	s
RSLDT *Restore LDTR and Descriptor*	0F 7B [mod 000 r/m]	-	-	-	-	-	-	-	-	-	10	10	s	s
RSM *Resume from SMM Mode*	0F AA	-	-	-	-	-	-	-	-	-	76	76	s	s

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
RSTS *Restore TSR and Descriptor*	0F 7D [mod 000 r/m]	-	-	-	-	-	-	-	-	-	10	10	s	s
SAHF *Store AH in FLAGS*	9E	x	-	-	-	x	x	-	x	x	2	2		
SAL *Shift Left Arithmetic*		x	-	-	-	x	x	-	x	x				
Register/Memory by 1	D[000w] [mod 100 r/m]										2/4	2/4		
Register/Memory by CL	D[001w] [mod 100 r/m]										3/5	3/5		
Register/Memory by Immediate	C[000w] [mod 100 r/m] #										2/4	2/4		
SAR *Shift Right Arithmetic*		x	-	-	-	x	x	x	x	x				
Register/Memory by 1	D[000w] [mod 111 r/m]										2/4	2/4		
Register/Memory by CL	D[001w] [mod 111 r/m]										3/5	3/5		
Register/Memory by Immediate	C[000w] [mod 111 r/m] #										2/4	2/4		
SBB *Integer Subtract with Borrow*		x	-	-	-	x	x	x	x	x				
Register to Register	1[10dw] [11 reg r/m]										1	1		
Register to Memory	1[100w] [mod reg r/m]										3	3		
Memory to Register	1[101w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8[00sw] [mod 001 r/m] #										1/3	1/3		
Immediate to Accumulator (short form)	1[110w] #										1	1		
SCAS *Scan String*	A [111w]	x	-	-	-	x	x	x	x	x	5	5	b	h
SETB/SETNAE/SETC *Set Byte on Below/Not Above or Equal/Carry*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 92 [mod 000 r/m]										2/2	2/2		
SETBE/SETNA *Set Byte on Below or Equal/Not Above*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 96 [mod 000 r/m]										2/2	2/2		
SETE/SETZ *Set Byte on Equal/Zero*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 94 [mod 000 r/m]										2/2	2/2		
SETL/SETNGE *Set Byte on Less/Not Greater or Equal*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 9C [mod 000 r/m]										2/2	2/2		
SETLE/SETNG *Set Byte on Less or Equal/Not Greater*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 9E [mod 000 r/m]										2/2	2/2		
SETNB/SETAE/SETNC *Set Byte on Not Below/Above or Equal/Not Carry*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 93 [mod 000 r/m]										2/2	2/2		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
SETNBE/SETA *Set Byte on Not Below or Equal/ Above*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 97 [mod 000 r/m]										2/2	2/2		
SETNE/SETNZ *Set Byte on Not Equal/Not Zero*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 95 [mod 000 r/m]										2/2	2/2		
SETNL/SETGE *Set Byte on Not Less/Greater or Equal*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 9D [mod 000 r/m]										2/2	2/2		
SETNLE/SETG *Set Byte on Not Less or Equal/ Greater*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 9F [mod 000 r/m]										2/2	2/2		
SETNO *Set Byte on Not Overflow*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 91 [mod 000 r/m]										2/2	2/2		
SETNP/SETPO *Set Byte on Not Parity/Parity Odd*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 9B [mod 000 r/m]										2/2	2/2		
SETNS *Set Byte on Not Sign*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 99 [mod 000 r/m]										2/2	2/2		
SETO *Set Byte on Overflow*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 90 [mod 000 r/m]										2/2	2/2		
SETP/SETPE *Set Byte on Parity/Parity Even*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 9A [mod 000 r/m]										2/2	2/2		
SETS *Set Byte on Sign*		-	-	-	-	-	-	-	-	-				h
To Register/Memory	0F 98 [mod 000 r/m]										2/2	2/2		
SGDT *Store GDT Register*		-	-	-	-	-	-	-	-	-			b,c	h
To Register/Memory	0F 01 [mod 000 r/m]										6	6		
SHL *Shift Left Logical*		x	-	-	-	x	x	-	x	x			b	h
Register/Memory by 1	D [000w] [mod 100 r/m]										1/3	1/3		
Register/Memory by CL	D [001w] [mod 100 r/m]										2/4	2/4		
Register/Memory by Immediate	C [000w] [mod 100 r/m] #										1/3	1/3		
SHLD *Shift Left Double*		-	-	-	-	x	x	-	x	x				
Register/Memory by Immediate	0F A4 [mod reg r/m] #										1/3	1/3		
Register/Memory by CL	0F A5 [mod reg r/m]										3/5	3/5		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
SHR *Shift Right Logical*		x	-	-	-	x	x	-	x	x			b	h
Register/Memory by 1	D [000w] [mod 101 r/m]										1/3	1/3		
Register/Memory by CL	D [001w] [mod 101 r/m]										2/4	2/4		
Register/Memory by Immediate	C [000w] [mod 101 r/m] #										1/3	1/3		
SHRD *Shift Right Double*		-	-	-	-	x	x	-	x	x				
Register/Memory by Immediate	0F AC [mod reg r/m] #										1/3	1/3		
Register/Memory by CL	0F AD [mod reg r/m]										3/5	3/5		
SIDT *Store IDT Register*		-	-	-	-	-	-	-	-	-			b,c	h
To Register/Memory	0F 01 [mod 001 r/m]										6	6		
SLDT *Store LDT Register*		-	-	-	-	-	-	-	-	-			a	h
To Register/Memory	0F 01 [mod 000 r/m]											1/2		
SMINT *Software SMM Entry*	0F 7E	-	-	-	-	-	-	-	-	-	24	24	s	s
SMSW *Store Machine Status Word*	0F 01 [mod 100 r/m]	-	-	-	-	-	-	-	-	-	1/2	1/2	b,c	h
STC *Set Carry Flag*	F9	-	-	-	-	-	-	-	-	1	1	1		
STD *Set Direction Flag*	FD	-	1	-	-	-	-	-	-	-	1	1		
STI *Set Interrupt Flag*	FB	-	-	1	-	-	-	-	-	-	7	7		m
STOS *Store String*	A [101w]	-	-	-	-	-	-	-	-	-	3	3	b	h
STR *Store Task Register*		-	-	-	-	-	-	-	-	-			a	h
To Register/Memory	0F 00 [mod 001 r/m]											1/2		
SUB *Integer Subtract*		x	-	-	-	x	x	x	x	x			b	h
Register to Register	2 [10dw] [11 reg r/m]										1	1		
Register to Memory	2 [100w] [mod reg r/m]										3	3		
Memory to Register	2 [101w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8 [00sw] [mod 101 r/m] #										1/3	1/3		
Immediate to Accumulator (short form)	2 [110w] #										1	1		
SVDC *Save Segment Register and Descriptor*	0F 78 [mod sreg3 r/m]	-	-	-	-	-	-	-	-	-	18	18	s	s
SVLDT *Save LDTR and Descriptor*	0F 7A [mod 000 r/m]	-	-	-	-	-	-	-	-	-	18	18	s	s
SVTS *Save TSR and Descriptor*	0F 7C [mod 000 r/m]	-	-	-	-	-	-	-	-	-	18	18	s	s
TEST *Test Bits*		0	-	-	-	x	x	-	x	0			b	h
Register/Memory and Register	8 [010w] [mod reg r/m]										1/3	1/3		
Immediate Data and Register/Memory	F [011w] [mod 000 r/m] #										1/3	1/3		
Immediate Data and Accumulator	A [100w] #										1	1		
VERR *Verify Read Access*		-	-	-	-	-	x	-	-	-			a	g,h,j,p
To Register/Memory	0F 00 [mod 100 r/m]											9/10		

Table 4-17. Instruction Set Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCK COUNT	PROT. MODE CLOCK COUNT	NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	Reg/Cache Hit	Reg/Cache Hit	Real Mode	Protected Mode
VERW *Verify Write Access*		-	-	-	-	-	x	-	-	-			a	g,h,j,p
To Register/Memory	0F 00 [mod 101 r/m]											9/10		
WAIT *Wait Until FPU Not Busy*	9B	-	-	-	-	-	-	-	-	-	5	5		
WBINVD *Write-Back and Invalidate Cache*	0F 09	-	-	-	-	-	-	-	-	-	4	4		
XADD *Exchange and Add*		x	-	-	-	x	x	x	x	x				
Register1, Register2	0F C[000w] [11 reg2 reg1]										3	3		
Memory, Register	0F C[000w] [mod reg r/m]										6	6		
XCHG *Exchange*		-	-	-	-	-	-	-	-	-			b,f	f,h
Register/Memory with Register	8[011w] [mod reg r/m]										3/4	3/4		
Register with Accumulator	9[0 reg]										3	3		
XLAT *Translate Byte*	D7	-	-	-	-	-	-	-	-	-	3	3		h
XOR *Boolean Exclusive OR*		0	-	-	-	x	x	-	x	0			b	h
Register to Register	3 [00dw] [11 reg r/m]										1	1		
Register to Memory	3 [000w] [mod reg r/m]										3	3		
Memory to Register	3 [001w] [mod reg r/m]										3	3		
Immediate to Register/Memory	8 [00sw] [mod 110 r/m] #										1/3	1/3		
Immediate to Accumulator (short form)	3 [010w] #										1	1		

Instruction Notes for Instruction Set Summary

# = immediate data
++ = 16-bit displacement
x = modified
+ = 8-bit displacement
+++ = 32 bit displacement (full)
- = unchanged

Notes a through c apply to Real Address Mode only:

a. This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid op-code).

b. Exception 13 fault (general protection) will occur in Real Mode if an operand reference is made that partially or fully extends beyond the maximum CS, DS, ES, FS, or GS segment limit (FFFFH). Exception 12 fault (stack segment limit violation or not present) will occur in Real Mode if an operand reference is made that partially or fully extends beyond the maximum SS limit.

c. This instruction may be executed in Real Mode. In Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.

Notes e through g apply to Real Address Mode and Protected Virtual Address Mode:

e. An exception may occur, depending on the value of the operand.

f. $\overline{\text{LOCK}}$ is automatically asserted, regardless of the presence or absence of the LOCK prefix.

g. $\overline{\text{LOCK}}$ is asserted during descriptor table accesses.

Notes h through r apply to Protected Virtual Address Mode only:

h. Exception 13 fault will occur if the memory operand in CS, DS, ES, FS, or GS cannot be used due to either a segment limit violation or an access rights violation. If a stack limit is violated, an exception 12 occurs.

i. For segment load operations, the CPL, RPL, and DPL must agree with the privilege rules to avoid an exception 13 fault. The segment's descriptor must indicate "present" or exception 11 (CS, DS, ES, FS, GS not present). If the SS register is loaded and a stack segment not present is detected, an exception 12 occurs.

j. All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK# to maintain descriptor integrity in multiprocessor systems.

k. JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.

l. An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).

m. An exception 13 fault occurs if CPL is greater than IOPL.

n. The IF bit of the flag register is not updated if CPL is greater than IOPL. The IOPL and VM fields of the flag register are updated only if CPL = 0.

o. The PE bit of the MSW (CR0) cannot be reset by this instruction. Use MOV into CRO if desiring to reset the PE bit.

p. Any violation of privilege rules as apply to the selector operand does not cause a Protection exception, rather, the zero flag is cleared.

q. If the coprocessor's memory operand violates a segment limit or segment access rights, an exception 13 fault will occur before the ESC instruction is executed. An exception 12 fault will occur if the stack limit is violated by the operand's starting address.

r. The destination of a JMP, CALL, INT, RET, or IRET must be in the defined limit of a code segment or an exception 13 fault will occur.

Note s applies to SGS THOMSON specific SMM instructions:

s. All memory accesses to SMM space are non-cacheable. An invalid opcode exception 6 occurs unless SMI is enabled and SMAR size 0, and CPL = 0 and [SMAC is set or if in an SMI handler].

4.5 FPU Clock Counts

The CPU can be divided into the FPU which processes floating point instructions and the remaining circuity collectively called the integer unit. The FPU can execute instructions independently of the integer unit. For example, the integer unit can issue a floating point instruction without memory operands, in two clock cycles and then pass the operation to the FPU to execute. The integer unit will continue to execute instructions until the next floating point instruction is encountered. The FPU loads from memory are similar in that the integer unit issues the FPU instruction, transfers data to the FPU and then is free to execute integer instructions. However, when executing a floating point store, the resources of both the FPU and integer unit are used.

4.5.1 Instruction Set Summary

Table 4-19 summarizes the operation and allowed forms of the ST486 FPU instruction set.

4.5.2 Abbreviations

The abbreviations used in Table 4-19 are listed in the table below:

Table 4-18. FPU Table Abbreviations

Abbreviations	Meaning
n	Stack register number
TOS	Top of stack register pointed to by SSS in the status register.
ST(1)	FPU register next to TOS
ST(n)	A specific FPU register, relative to TOS
M.WI	16-bit integer operand from memory
M.SI	32-bit integer operand from memory
M.LI	64-bit integer operand from memory
M.SR	32-bit real operand from memory
M.DR	64-bit real operand from memory
M.XR	80-bit real operand from memory
M.BCD	18-digit BCD integer operand from memory
CC	FPU condition code
Env Regs	Status, Mode Control and Tag Registers, Instruction Pointer and Operand Pointer

Table 4-19. FPU Instruction Set Summary

FPU Instruction	OP Code	Operation	Clock Count	Notes
F2XM1 *Function Evaluation 2x-1*	D9 F0	TOS ← 2^{TOS}-1	98 -114	See Note 2
FABS *Floating Absolute Value*	D9 E1	TOS ← |TOS |	5	
FADD *Floating Point Add*				
Top of Stack	DC [1100 0 n]	ST(n) ← ST(n) + TOS	10 - 16	
80-bit Register	D8 [1100 0 n]	TOS ← TOS + ST(n)	10 - 16	
64-bit Real	DC [mod 000 r/m]	TOS ← TOS + M.DR	11 - 17	
32-bit Real	D8 [mod 000 r/m]	TOS ← TOS + M.SR	13 - 19	
FADDP *Floating Point Add, Pop*	DE [1100 0 n]	ST(n) ← ST(n) + TOS; then pop TOS	10 - 16	
FIADD *Floating Point Integer Add*				
32-bit integer	DA [mod 000 r/m]	TOS ← TOS + M.SI	18 - 27	
16-bit integer	DE [mod 000 r/m]	TOS ← TOS + M.WI	18 - 26	
FCHS *Floating Change Sign*	D9 E0	TOS ← TOS	5	
FCLEX *Clear Exceptions*	(9B) DB E2	Wait then Clear Exceptions	8	
FNCLEX *Clear Exceptions*	DB E2	Clear Exceptions	5	
FCOM *Floating Point Compare*				
80-bit Register	D8 [1101 0 n]	CC set by TOS - ST(n)	8	
64-bit Real	DC [mod 010 r/m]	CC set by TOS - M.DR	12	
32-bit Real	D8 [mod 010 r/m]	CC set by TOS - M.SR	10	
FCOMP *Floating Point Compare, Pop*				
80-bit Register	D8 [1101 1 n]	CC set by TOS - ST(n); then pop TOS	8	
64-bit Real	DC [mod 011 r/m]	CC set by TOS - M.DR; then pop TOS	12	
32-bit Real	D8 [mod 011 r/m]	CC set by TOS - M.SR; then pop TOS	10	
FCOMPP *Floating Point Compare, Pop* Two Stack Elements	DE D9	CC set by TOS - ST(1); then pop TOS and ST(1)	8	
FICOM *Floating Point Compare*				
32-bit integer	DA [mod 010 r/m]	CC set by TOS - M.WI	15 - 17	
16-bit integer	DE [mod 010 r/m]	CC set by TOS - M.SI	15 - 16	
FICOMP *Floating Point Compare*				
32-bit integer	DA [mod 011 r/m]	CC set by TOS - M.WI; then pop TOS	15 - 17	
16-bit integer	DE [mod 011 r/m]	CC set by TOS - M.SI; then pop TOS	15 - 16	
FCOS *Function Evaluation: Cos(x)*	D9 FF	TOS ← COS(TOS)	98 - 143	See Note 1
FDECSTP *Decrement Stack Pointer*	D9 F6	Decrement top of stack pointer	5	
FDIV *Floating Point Divide*				
Top of Stack	DC [1111 1 n]	ST(n) ← ST(n) / TOS	28 -34	
80-bit Register	D8 [1111 0 n]	TOS ← TOS / ST(n)	28 - 34	
64-bit Real	DC [mod 110 r/m]	TOS ← TOS / M.DR	35 - 41	
32-bit Real	D8 [mod 110 r/m]	TOS ← TOS / M.SR	33 - 39	

Table 4-19. FPU Instruction Set Summary

FPU Instruction	OP Code	Operation	Clock Count	Notes
FDIVP F*loating Point Divide, Pop*	DE [1111 1 n]	ST(n) ← ST(n) / TOS; then pop TOS	28 - 34	
FDIVR *Floating Point Divide Reversed*				
Top of Stack	DC [1111 0 n]	TOS ← ST(n) / TOS	28 -34	
80-bit Register	D8 [1111 1 n]	ST(n) ←TOS / ST(n)	28 - 34	
64-bit Real	DC [mod 111 r/m]	TOS ← M.DR / TOS	35 - 41	
32-bit Real	D8 [mod 111 r/m]	TOS ← M.SR / TOS	33 - 39	
FIDIVRP *Floating Point Integer Divide*				
Reversed, Pop	DE [1111 0 n]	ST(n) ← TOS / ST(n); then pop TOS	28 -34	
FIDIV *Floating Point Integer Divide*				
32-bit Integer				
16-bit Integer	DA [mod 110 r/m]	TOS ← TOS / M.SI	36 - 44	
FIDIVR *Floating Point Integer*	DE [mod 110 r/m]	TOS ← TOS / M.WI	36 - 43	
Reversed				
32-bit Integer	DA [mod 111 r/m]	TOS ← M.SI / TOS	36 - 44	
16-bit Integer	DE [mod 111 r/m]	TOS ← M.WI / TOS	36 - 43	
FFREE *Free Floating Point Register*	DD [1100 0 n]	TAG(n) ← Empty	5	
FINCSTP *Increment Stack Pointer*	D9 F7	Increment top of stack pointer	5	
FINIT *Initialize FPU*	(9B)DB E3	Wait then initialize	8	
FNINIT *Initialize FPU*	DB E3	Initialize	5	
FLD *Load Data to FPU Reg.*				
Top of Stack	D9 [1100 0 n]	Push ST(n) onto stack	4	
80-bit Real	DB [mod 101 r/m]	Push M.XR onto stack	9	
64-bit Real	DD [mod 000 r/m]	Push M.DR onto stack	7	
32-bit Real	D9 [mod 000 r/m]	Push M.SR onto stack	5	
FBLD *Load Packed BCD Data to FPU Reg.*	DF [mod 100 r/m]	Push M.BCD onto stack	49 - 53	
FILD *Load Integer Data to FPU Reg.*				
64-bit Integer	DF [mod 101 r/m]	Push M.LI onto stack	9 - 13	
32-bit Integer	DB [mod 000 r/m]	Push M.SI onto stack	8 - 10	
16-bit Integer	DF [mod 000 r/m]	Push M.WI onto stack	8 - 9	
FLD1 *Load Floating Const.= 1.0*	D9 E8	Push 1.0 onto stack	6	
FLDCW *Load FPU Mode Control Register*	D9 [mod 101 r/m]	Ctl Word ← Memory	5	
FLDENV *Load FPU Environment*	D9 [mod 100 r/m]	Env Regs ← Memory	28 - 38	
FLDL2E *Load Floating Const.= Log2(e)*	D9 EA	Push Log^{2}(e) onto stack	6	
FLDL2T *Load Floating Const.= Log2(10)*	D9 E9	Push Log^{2}(10) onto stack	6	
FLDLG2 *Load Floating Const.= Log10(2)*	D9 EC	Push Log^{10}(2) onto stack	6	
FLDLN2 *Load Floating Const.= Ln(2)*	D9 ED	Push Log^{e}(2) onto stack	6	
FLDPI *Load Floating Const.=* π	D9 EB	Push π onto stack	6	
FLDZ *Load Floating Const.= 0.0*	D9 EE	Push 0.0 onto stack	6	

Table 4-19. FPU Instruction Set Summary

FPU Instruction	OP Code	Operation	Clock Count	Notes
FMUL *Floating Point Multiply*				
Top of Stack	DC [1100 1 n]	ST(n) ← ST(n) / TOS	12	
80-bit Register	D8 [1100 1 n]	TOS ← TOS / ST(n)	12	
64-bit Real	DC [mod 001 r/m]	TOS ← TOS / M.DR	15	
32-bit Real	D8 [mod 001 r/m]	TOS ← TOS / M.SR	13	
FMULP *Floating Point Multiply & Pop*	DE [1100 1 n]	ST(n) ← ST(n) / TOS; then pop TOS	12	
FIMUL *Floating Point Integer Multiply*				
32-bit Integer	DA [mod 001 r/m]	TOS ← TOS/ M.SI	21 - 25	
16-bit Integer	DE [mod 001 r/m]	TOS ← TOS / M.WI	21 - 24	
FNOP *No Operation*	D9 D0	No Operation	3	
FPATAN *Function Eval:* $Tan^{-1}(y/x)$	D9 F3	ST(1) ← ATAN[ST(1) / TOS]; then pop TOS	97 - 161	See Note 3
FPREM *Floating Point Remainder*	D9 F8	TOS ← Rem[TOS / ST(1)]	82 - 93	
FPREM1 *Floating Point Remainder IEEE*	D9 F5	TOS ← Rem[TOS / ST(1)]	82 - 93	
FPTAN *Function Eval: Tan(x)*	D9 F2	TOS ← TAN(TOS); then push 1.0 onto stack	123 - 140	See Note 1
FRNDINT *Round to Integer*	D9 FC	TOS ← Round(TOS)	12 - 21	
FRSTOR *Load FPU Environment and Reg.*	DD [mod 100 r/m]	Restore state.	110 - 120	
FSAVE *Save FPU Environment and Reg*	(9B)DD [mod 110 r/m]	Wait then save state.	143 - 153	
FNSAVE *Save FPU Environment and Reg*	DD [mod 110 r/m]	Save state.	140 - 150	
FSCALE *Floating Multiply by* 2^n	D9 FD	TOS ← TOS x $2^{(ST(1))}$	10 - 15	
FSIN *Function Evaluation: Sin(x)*	D9 FE	TOS ← SIN(TOS)	81 - 159	See Note 1

FSINCOS *Function Eval.: Sin(x)& Cos(x)*	D9 FB	temp ← TOS;
		TOS ← SIN(temp); then
		push COS(temp) onto stack
			150 - 165	See Note 1

FSQRT *Floating Point Square Root*	D9 FA	TOS ← Square Root of TOS	61 - 62	
FST *Store FPU Register*				
80-bit Register	DD [1101 0 n]	ST(n) ← TOS	5	
80-bit Real	DB [mod 111 r/m]	M.XR ← TOS	15	
64-bit Real	DD [mod 010 r/m]	M.DR ← TOS	12	
32-bit Real	D9 [mod 010 r/m]	M.SR ← TOS	9	
FSTP *Store FPU Register, Pop*				
Top of Stack	DB [1101 1 n]	ST(n) ← TOS; then pop TOS	5	
80-bit Real	DB [mod 111 r/m]	M.XR ← TOS; then pop TOS	15	
64-bit Real	DD [mod 011 r/m]	M.DR ← TOS; then pop TOS	12	
32-bit Real	D9 [mod 011 r/m]	M.SR ← TOS; then pop TOS	9	
FBSTP *Store BCD Data, Pop*	DF [mod 110 r/m]	M.BCD ← TOS; then pop TOS	77 - 82	

Table 4-19. FPU Instruction Set Summary

FPU Instruction	OP Code	Operation	Clock Count	Notes
FIST *Store Integer FPU Register*				
32-bit Integer	DB [mod 010 r/m]	M.SI ← TOS	16 - 22	
16-bit Integer	DF [mod 010 r/m]	M.WI ← TOS	12 - 18	
FISTP *Store Integer FPU Register, Pop*				
64-bit Integer	DF [mod 111 r/m]	M.LI ← TOS; then pop TOS	19 - 27	
32-bit Integer	DB [mod 011 r/m]	M.SI ← TOS; then pop TOS	16 - 22	
16-bit Integer	DF [mod 011 r/m]	M.WI ← TOS; then pop TOS	12 - 18	
FSTCW *Store FPU Mode Control Register*	(9B) D9 [mod 111 r/m]	Wait Memory ← Control Mode Register	6	
FNSTCW *Store FPU Mode Control Register*	D9 [mod 111 r/m]	Memory ← Control Mode Register	3	
FSTENV *Store FPU Environment*	(9B) D9 [mod 110 r/m]	Wait Memory ← Env. Registers	30 - 40	
FNSTENV *Store FPU Environment*	D9 [mod 110 r/m]	Memory ← Env. Registers	27 - 37	
FSTSW *Store FPU Status Register*	(9B) DD [mod 111 r/m]	Wait Memory ← Status Register	6	
FNSTSW *Store FPU Status Register*	DD [mod 111 r/m]	Memory ← Status Register	3	
FSTSW AX *Store FPU Status Register to AX*	E0	Wait AX ← Status Register	6	
FNSTSW AX *Store FPU Status Register to AX*	E0	AX ← Status Register	3	
FSUB *Floating Point Subtract*				
Top of Stack	DC [1110 1 n]	ST(n) ← ST(n) - TOS	10 - 16	
80-bit Register	D8 [1110 0 n]	TOS ← TOS - ST(n)	10 - 16	
64-bit Real	DC [mod 100 r/m]	TOS ← TOS - M.DR	13 - 19	
32-bit Real	D8 [mod 100 r/m]	TOS ← TOS - M.SR	11 - 17	
FSUBP *Floating Point Subtract, Pop*	DE [1110 1 n]	ST(n) ← ST(n) - TOS; then pop TOS	10 - 16	
FSUBR *Floating Point Subtract Reverse*				
Top of Stack	DC [1110 0 n]	TOSST(n) - TOS	10 - 16	
80-bit Register	D8 [1110 1 n]	ST(n) ← TOS - ST(n)	10 - 16	
64-bit Real	DC [mod 101 r/m]	TOS ← M.DR - TOS	13 - 19	
32-bit Real	D8 [mod 101 r/m]	TOS ← M.SR - TOS	11 - 17	
FSUBRP *Floating Point Subtract Reverse, Pop*	DE [1110 0 n]	ST(n) ← TOS - ST(n); then pop TOS	10 - 16	
FISUB *Floating Point Integer Subtract*				
32-bit Integer	DA [mod 100 r/m]	TOS ← TOS - M.SI	18 - 27	
16-bit Integer	DE [mod 100 r/m]	TOS ← TOS - M.WI	18 - 26	
FISUBR *Floating Point Integer Subtract Reverse*				
32-bit Integer Reversed	DA [mod 101 r/m]	TOS ← M.SI - TOS	18 - 27	
16-bit Integer Reversed	DE [mod 101 r/m]	TOS ← M.WI - TOS	18 - 26	
FTST *Test Top of Stack*	D9 E4	CC set by TOS - 0.0	10	
FUCOM *Unordered Compare*	DD [1110 0 n]	CC set by TOS - ST(n)	8	

Table 4-19. FPU Instruction Set Summary

FPU Instruction	OP Code	Operation	Clock Count	Notes
FUCOMP *Unordered Compare, Pop*	DD [1110 1 n]	CC set by TOS - ST(n); then pop TOS	8	
FUCOMPP *Unordered Compare,* *Pop two elements*	DA E9	CC set by TOS - ST(I); then pop TOS and ST(1)	8	
FWAIT *Wait*	9B	Wait for FPU not busy	3	
FXAM *Report Class of Operand*	D9 E5	CC ← Class of TOS	4	
FXCH *Exchange Register with TOS*	D9 [1100 1 n]	TOS ← ST(n) Exchange	9	

FXTRACT *Extract Exponent*	D9 F4	temp ← TOS;
		TOS ← exponent (temp); then
		TOS ← push significant (temp) onto stack
			11 - 16	

FLY2X *Function Eval. y x Log2(x)*	D9 F1	ST(1) ← ST(1) x Log_2(TOS); then pop TOS	145 - 154	
FLY2XP1 *Function Eval. y x Log2(x+1)*	D9 F9	ST(1) ← ST(1) x Log_2(1+TOS); then pop TOS	131 - 133	See Note 4

FPU Instruction Summary Notes

All references to TOS and ST(n) refer to stack layout prior to execution.

Values popped off the stack are discarded.

A pop from the stack increments the top stack pointer.

A push to the stack decrements the top of the stack pointer.

Note 1:

For FCOS, FSIN, FSINCOS and FPTAN, time shown is for absolute value of TOS < 3π /4.
Add 90 clock counts for argument reduction if outside this range.

For FCOS, clock count is 143 if TOS < π /4 and clock count is 98 if π /4 < TOS > π /2
For FSIN, clock count is 81 to 82 if absolute value of TOS < π /4.

Note 2:

For F2XM1, clock count is 98 if absolute value of TOS < 0.5.

Note 3:

For FPATAN, clock count is 97 if ST(1)/TOS < π /32.

Note 4:

For FYL2XP1, clock count is 170 if TOS is out of range and regular FYL2X is called.