Texas Instruments

TI486 Microprocessor

Reference Guide

1993

PC Systems Logic

TI486 Microprocessor Reference Guide

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments Incorporated (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

About This Manual

This manual describes both the TI486SLC/E and TI486DLC/E product family. Each chapter except for chapters 3 and 4 cover both the TI486SLC/E and TI486DLC/E. Chapter 3 explicitly covers the TI486SLC/E and chapter 4 explicitly covers the TI486DLC/E. This document contains the following chapters:

Chapter 1 **Product Overview**

Chapter 1 introduces the features and itemizes the differences between the TI486SLC/E and TI486DLC/E, both of which are offered in 3-volt versions (TI486xLC/E-V) for battery-powered applications. A functional block diagram, logic symbol, and I/O signal pins are provided for each of the two microprocessors. Additional material describes selected system architectures such as the execution pipeline, the on-chip cache memory, and the power management techniques. The System Management Mode (SMM) permits the TI486 microprocessors to respond to and service interrupts having a higher priority than standard 486 processors.

Chapter 2 **Programming Interface**

Chapter 2 describes the internal operations of the TI486, for both the TI486SLC/E and TI486DLC/E, 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. Also included is an overview of real, virtual 8086, and protected operating modes.

Chapter 3 **TI486SLC/E Bus Interface**

Chapter 3 provides an overview of the TI486SLC/E processor signals, functional description of all pins, functional timing and bus operations (including non-pipelined and pipelined addressing), interfaces, and power management.

Chapter 4

TI486DLC/E Bus Interface

Chapter 4 provides an overview of the TI486DLC/E processor signals, functional description of all pins, functional timing and bus operations (including non-pipelined and pipelined addressing), interfaces, and power management.

Chapter 5

Electrical Specifications

Chapter 5 provides electrical specifications for both the TI486SLC/E and TI486DLC/E, including specifications for the 3-volt versions. The specifications include electrical connection requirements for all package types and pins, maximum ratings, recommended operating conditions, dc electrical, and ac characteristics.

Chapter 6

Mechanical Specifications

Chapter 6 provides mechanical specifications for both the TI486SLC/E and TI486DLC/E that include pin assignments, package physical dimensions, and package thermal characteristics.

Chapter 7

Instruction Set

Chapter 7 summarizes the TI486 instruction set and provides detailed information of the instruction encodings. The instruction set is the same for all TI486 processors.Instructions are listed in an instruction set summary table, that also provides information on the flags affected and the instruction clock counts for each instruction.

Appendix A

TI486 SMM Programmer's Guide

Appendix A provides detailed information, including examples pertinent to programming the TI486 system management mode (SMM). Included are $\overline{\text{SMI}}$ examples, testing/debugging SMM code, power management features, loading SMM programs, detection of CPU type and presence of SMM-capable devices, creating macros, and altering SMM code limits.

Appendix B

TI486 Cache Flush

Appendix B provides general cache invalidation techniques and discusses invalidation in systems with and without secondary cache.

Appendix C

TI486 BIOS Modification Guide

Appendix C discusses some BIOS changes that may need to be considered by the PC designer. The areas considered are power-on and hard reset, protected-mode to real-mode switching, and soft reset. Examples of assembler code for turning the cache on and off are provided.

Appendix D Ordering Information

Appendix D provides detailed ordering information showing what the components of the part number mean, and a description of each microprocessor offered. Versions offered include 5-volt and 3-volt versions, each of which are rated to operate at different speeds. The TI486SLC/E versions are packaged in the quad flat pack, and the TI486DLC/E versions are packaged in the ceramic pin grid array package.

Style and Symbol Conventions

This document uses the following conventions.

❑ Program code listings and program code examples are shown in a `special typeface` similar to a typewriter's.

Here is a sample assembler code program listing:

```
CLI
MOV     EAX, CR0        ; set bit 30, turn off cache
OR      EAX, 40000000h ; for external cache coherency
```

❑ In the instruction syntax descriptions, the instruction is in a **BOLD TYPEFACE** font and a description of the instruction is in *Italic Typeface.* Here is an example of an instruction syntax and description:

RSM *Resume from SMM Mode*

❑ Square brackets ([and]) identify the location and sequence for specifying register and/or memory options in the instruction opcodes. Here's an example of an opcode that requires register and memory parameters:

Reference: Instruction **ADD** *Integer Add* (Register to Memory)

Opcode = 0 [000w] [mod reg r/m]

Information About Cautions and Warnings

This book may contain cautions and warnings.

> CAUTION
>
> **This is an example of a caution statement.**
>
> **A caution statement describes a situation that could potentially damage your software or equipment.**

> WARNING
>
> **This is an example of a warning statement.**
>
> **A warning statement describes a situation that could potentially cause harm to you.**

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Trademarks

EPIC is a trademark of Texas Instruments Incorporated.

Contents

Figures

Tables

Chapter 1

Product Overview

Features

- Provides an immediate upgrade to 486-class performance for 386 footprints
  - TI486SLC/E is up to 2.4 times faster than a 386SL/386SX at same clock frequency (Landmark 2.0 = 107 MHz, Norton SI 6.0 = 52 at 33 MHz)
  - TI486DLC/E is up to 2 times faster than a 386DX at same clock frequency (Landmark 2.0 = 130 MHz, Norton SI 6.0 = 66, PM MIPS = 14 at 40 MHz)
- Advanced power management features for notebook, battery-powered, and reduced-power desktop PC systems
  - System Management Mode (SMM)
  - High priority System Management Interrupt (SMI) with separate memory address space
  - Suspend mode (initiated by either hardware or software)
- 3 V versions available: TI486SLC/E-V and TI486DLC/E-V provide approximately 60 percent power savings
- 486 compatible instruction set
- 386SX pin-compatible (TI486SLC/E) and 386DX pin-compatible (TI486DLC/E) versions)
- High-performance
  - TI486SLC/E clock speeds of 25 MHz and 33 MHz at 5 V
  - TI486SLC/E-V clock speed of 25 MHz at 3 V
  - TI486DLC/E clock speeds of 33 MHz and 40 MHz at 5 V
  - TI486DLC/E-V clock speeds of 25 MHz and 33 MHz at 3 V
- On-chip 1 KByte instruction/data cache can be configured as either direct-mapped or two-way set associative
- Fully static device permits clock stop state.
- Highly optimized variable length pipeline and on-chip 16-bit hardware multiplier
- Texas Instruments EPIC™ submicron CMOS technology
- Available in two packages: a 100 pin plastic bumpered quad flat pack for TI486SLC/E and TI486SLC/E-V and a 132 pin ceramic PGA for TI486DLC/E and TI486DLC/E-V

EPIC is a trademark of Texas Instruments Incorporated.

1.1 Introduction

The Texas Instruments TI486 microprocessor is an advanced x86-compatible processor offering high performance and integrated power management on a single chip.

The 486SLC/E is 486 instruction set compatible and is backward compatible with the 386SX pinout. The TI486SLC/E provides up to 2.4 times the performance of both the 386SL and 386SX at equal clock frequencies. The TI486SLC/E is an ideal solution for battery-powered applications in that it typically draws 0.4 mA supply current while the input clock is stopped in suspend mode. The TI486SLC/E-V version of the TI486SLC/E offers additional power savings because it operates on a 3-V as well as 5-V power supply.

The Texas Instruments TI486DLC/E microprocessor is an advanced 32-bit x86-compatible processor offering high performance and integrated power management on a single chip. The CPU is 486 instruction set compatible and is also compatible with the 386DX pinout. This CPU provides up to twice the performance of the 386DX at equal clock frequencies. The TI486DLC/E is an ideal solution for battery-powered applications in that it typically draws 0.4 mA while the input clock is stopped in suspend mode. The TI486DLC/E-V version of the TI486DLC/E offers additional power savings because it operates on a 3-V as well as 5-V power supply.

The TI486 supports 8-, 16-, and 32-bit data types and operates in real, virtual 8086, and protected modes. The TI486 microprocessor achieves high performance through use of a highly optimized, variable-length pipeline combined with a RISC-like single-cycle execution unit, an on-chip hardware multiplier, and an integrated instruction and data cache.

1.2 Differences Between the TI486SLC/E and TI486DLC/E

The TI486SLC/E and TI486DLC/E are the same except for the pin signals routed and utilized on the processors. Thus, the bus interfaces are different but the CPU core and cache/memory management are the same. The TI486SLC/E has a physical address range of 16 MBytes and the TI486DLC/E has a physical address range of 4 GBytes. Table 1–1 describes the signal differences between the TI486SLC/E and TI486DLC/E.

Table 1–1. TI486SLC/E/DLC/E Signal Differences

DESCRIPTION	TI486SLC/E	TI486DLC/E
Data bus	16-bits wide (D15–D0)	32-bits wide (D31–D0)
Address bus	A23–A1	A31–A2
Byte enables	2-byte enables used ($\overline{\text{BHE}}$, $\overline{\text{BLE}}$)	4-byte enables used ($\overline{\text{BE3}}$–$\overline{\text{BE0}}$)
Float bus signal ($\overline{\text{FLT}}$)	supported	not supported
Bus size 16 signal ($\overline{\text{BS16}}$)	not supported	supported

1.3 TI486SLC/E Overview

The TI486SLC/E microprocessor is implemented using Texas Instruments EPIC submicron CMOS technology and is available in 25-MHz and 33-MHz versions. Both the 5-V TI486SLC/E and 3-V TI486SLC/E-V versions are packaged in a 100-pin bumpered quad flat pack (QFP).

Figure 1–1 is a functional block diagram of the TI486SLC/E. The TI486SLC/E architecture results in up to 2.4 times the performance of conventional 386SX notebook CPUs as listed below.

- ❑ Up to 2.4 times faster than 386SX at same frequency
- ❑ Landmark 2.0 = 107 MHz, Norton SI 6.0 = 52 at 33 MHz

Figure 1–1. TI486SLC/E Functional Block Diagram

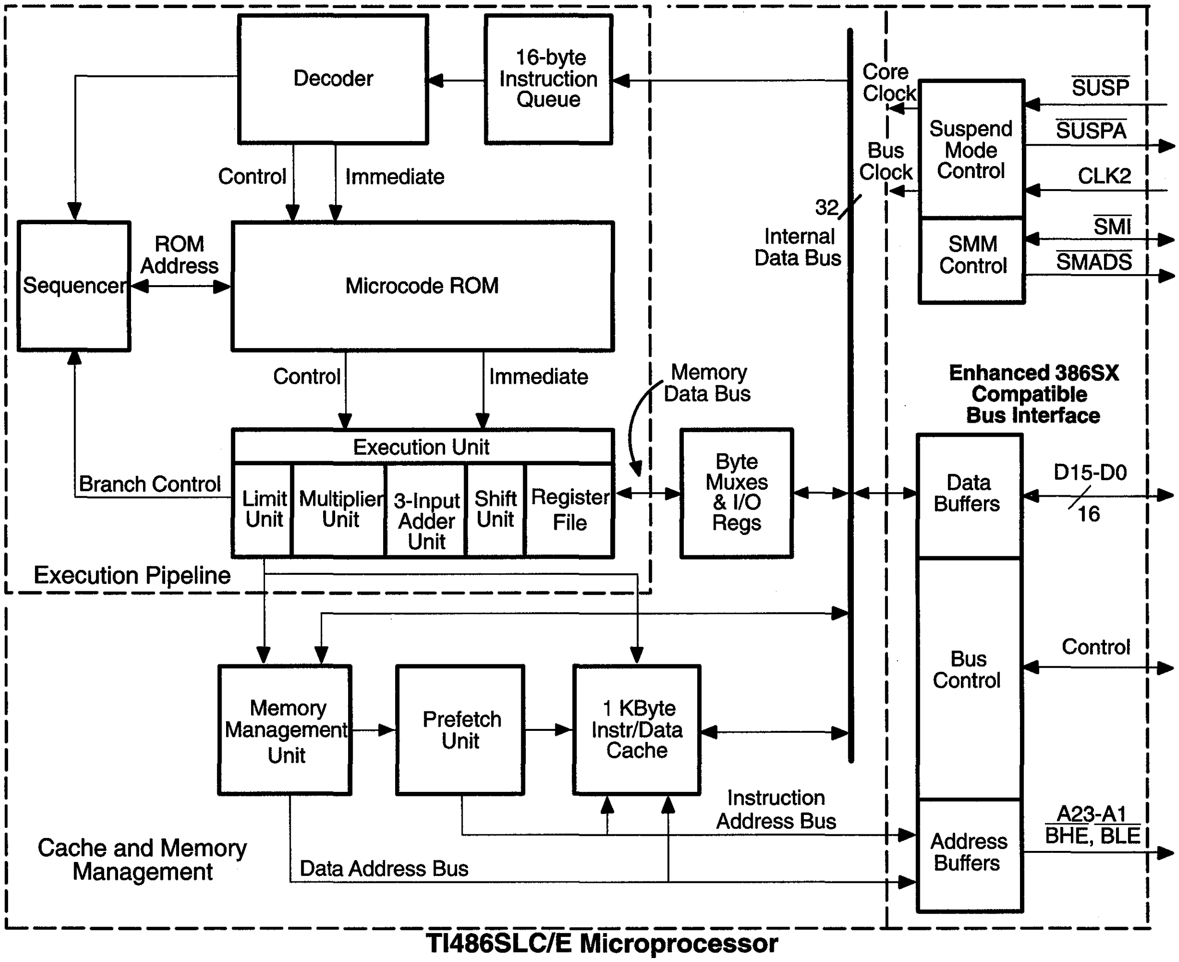
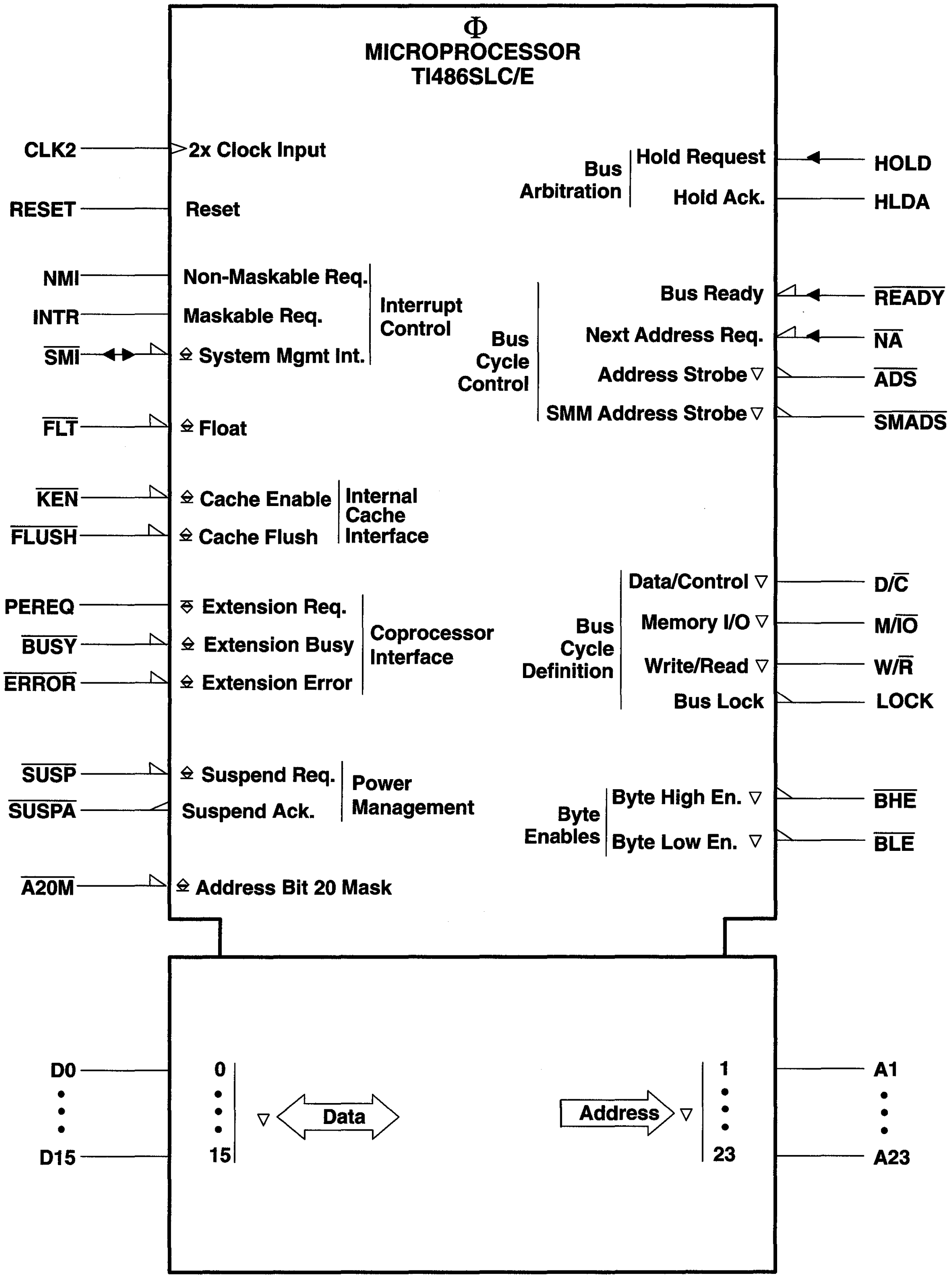


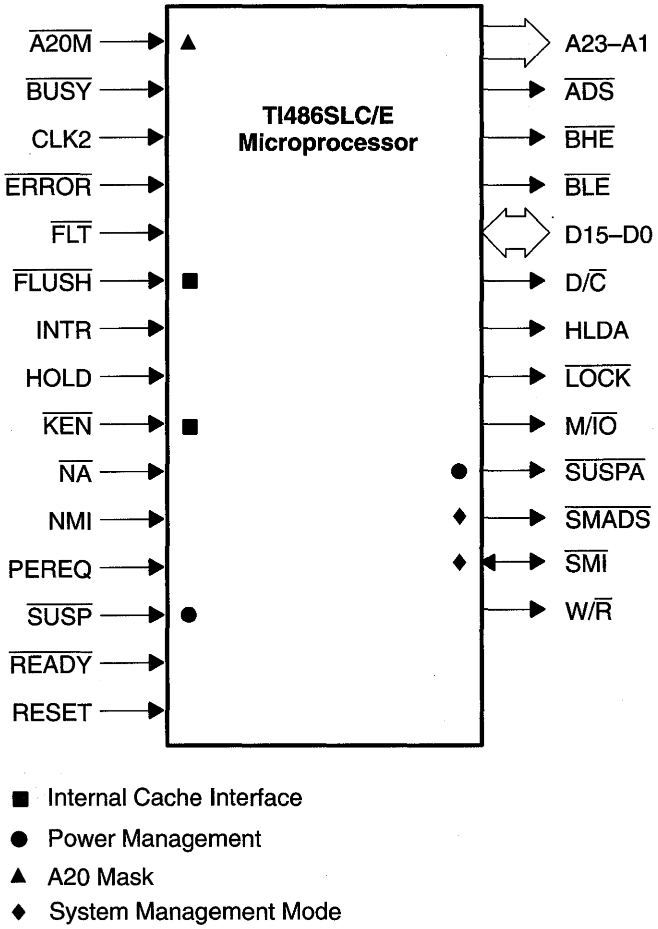
Figure 1–2. TI486SLC/E Logic Symbol



† This symbol is in accordance with ANSI/IEEE Std 91-1984.

The TI486SLC/E includes two power management signals ($\overline{\mathrm{SUSP}}$ and $\overline{\mathrm{SUSPA}}$), two cache interface signals ($\overline{\mathrm{FLUSH}}$ and $\overline{\mathrm{KEN}}$), an A20 mask input ($\overline{\mathrm{A20M}}$), and two SMM signals ($\overline{\mathrm{SMADS}}$ and $\overline{\mathrm{SMI}}$) that are additions to the 386SX signal set. The complete list of TI486SLC/E signals is shown in Figure 1–3.

Figure 1–3. TI486SLC/E Input and Output Signals


TI486SLC/E
Microprocessor
$\overline{\mathrm{A20M}}$ ▲
$\overline{\mathrm{BUSY}}$
CLK2
$\overline{\mathrm{ERROR}}$
$\overline{\mathrm{FLT}}$
$\overline{\mathrm{FLUSH}}$ ■
INTR
HOLD
$\overline{\mathrm{KEN}}$ ■
$\overline{\mathrm{NA}}$
NMI
PEREQ
$\overline{\mathrm{SUSP}}$ ●
$\overline{\mathrm{READY}}$
RESET
A23–A1
$\overline{\mathrm{ADS}}$
$\overline{\mathrm{BHE}}$
$\overline{\mathrm{BLE}}$
D15–D0
D/$\overline{\mathrm{C}}$
HLDA
$\overline{\mathrm{LOCK}}$
M/$\overline{\mathrm{IO}}$
● $\overline{\mathrm{SUSPA}}$
♦ $\overline{\mathrm{SMADS}}$
♦ $\overline{\mathrm{SMI}}$
W/$\overline{\mathrm{R}}$
■ Internal Cache Interface
● Power Management
▲ A20 Mask
♦ System Management Mode

1.4 TI486DLC/E Overview

The TI486DLC/E microprocessor is implemented using Texas Instruments EPIC submicron CMOS technology and is available in 33-MHz and 40-MHz versions. Both the TI486DLC/E and the 3-V TI486DLC/E-V are offered in a 132-pin ceramic PGA package.

Figure 1–4 is a functional block diagram of the TI486DLC/E. The TI486DLC/E typically benchmarks 1.5 to 2 times faster than a 386DX at the same clock frequency as listed below.

□ Landmark 2.0 = 130 MHz at 40 MHz
□ Norton SI 6.0 = 66 at 40 MHz
□ PM MIPS = 14 at 40 MHz

Figure 1–4. TI486DLC/E Functional Block Diagram

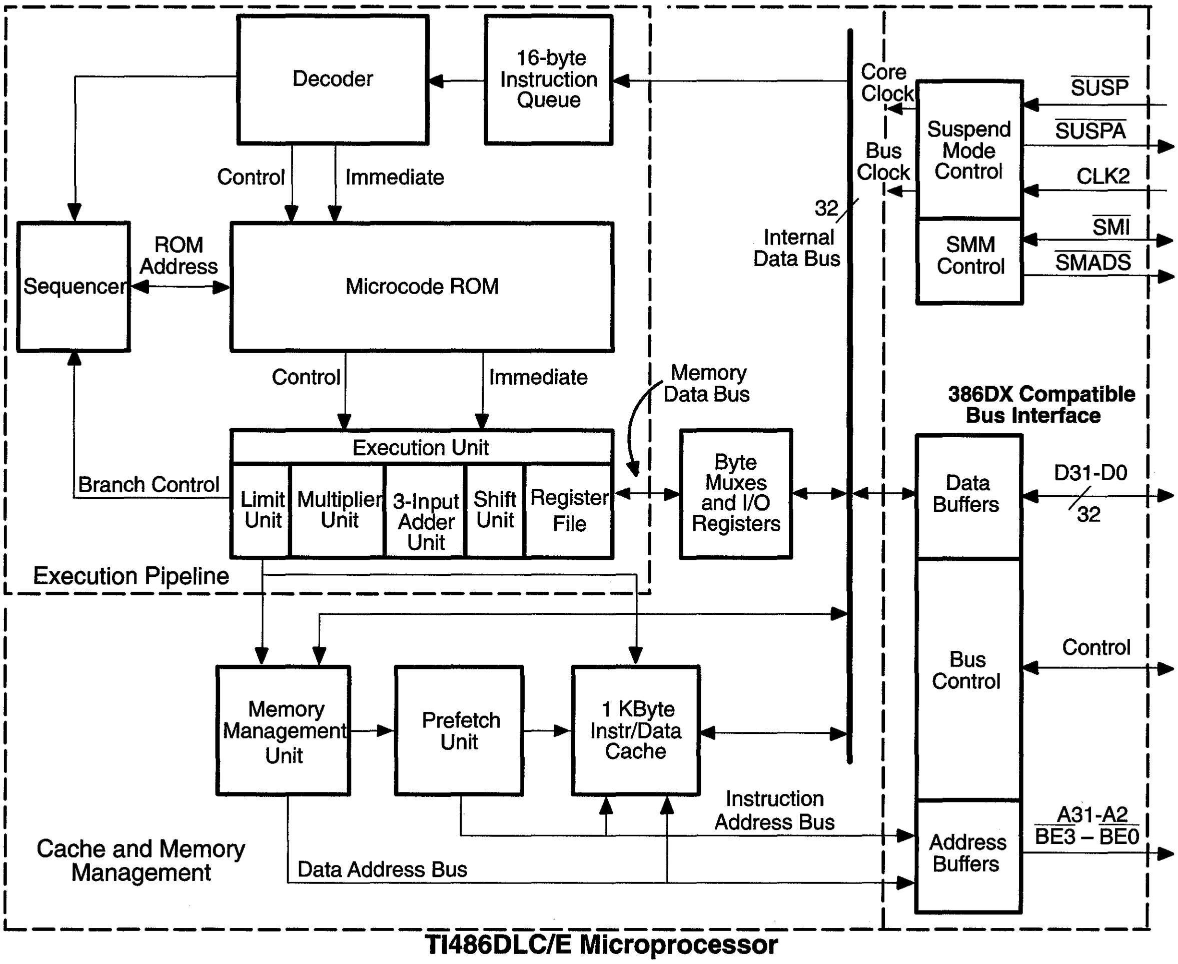
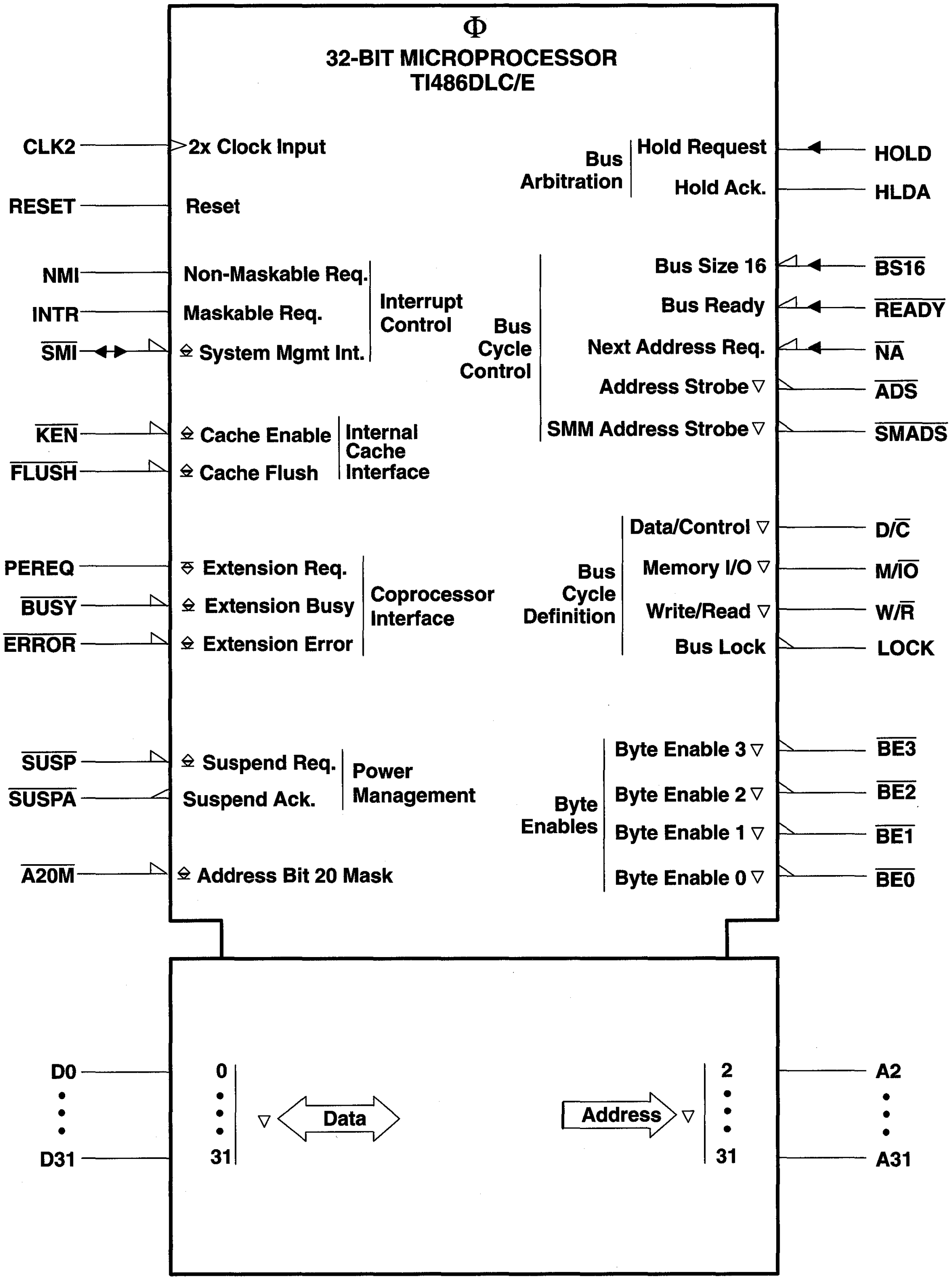


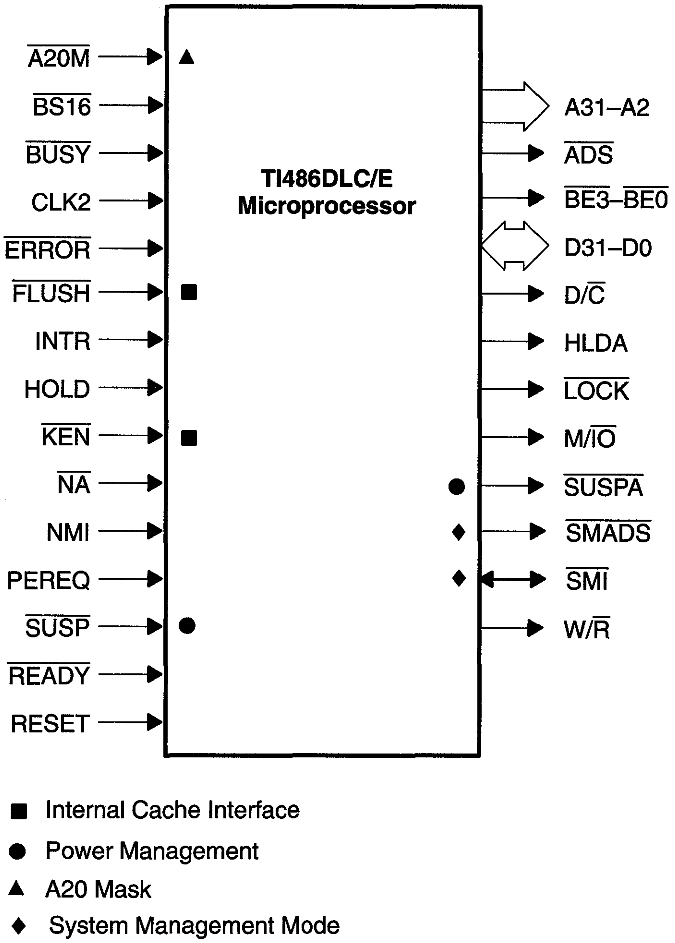
Figure 1–5. TI486DLC/E Logic Symbol



† This symbol is in accordance with ANSI/IEEE Std 91-1984.

The TI486DLC/E includes two power management signals ($\overline{\text{SUSP}}$ and $\overline{\text{SUSPA}}$), two cache interface signals ($\overline{\text{FLUSH}}$ and $\overline{\text{KEN}}$), an A20 mask input ($\overline{\text{A20M}}$), and two SMM signals ($\overline{\text{SMADS}}$ and $\overline{\text{SMI}}$) that are additions to the 386DX signal set. The complete list of TI486DLC/E signals is shown in Figure 1–6.

Figure 1–6. TI486DLC/E Input and Output Signals


TI486DLC/E
Microprocessor
$\overline{\text{A20M}}$ ▲
$\overline{\text{BS16}}$
$\overline{\text{BUSY}}$
CLK2
$\overline{\text{ERROR}}$
$\overline{\text{FLUSH}}$ ■
INTR
HOLD
$\overline{\text{KEN}}$ ■
$\overline{\text{NA}}$
NMI
PEREQ
$\overline{\text{SUSP}}$ ●
$\overline{\text{READY}}$
RESET
A31–A2
$\overline{\text{ADS}}$
$\overline{\text{BE3}}$–$\overline{\text{BE0}}$
D31–D0
D/$\overline{\text{C}}$
HLDA
$\overline{\text{LOCK}}$
M/$\overline{\text{IO}}$
● $\overline{\text{SUSPA}}$
♦ $\overline{\text{SMADS}}$
♦ $\overline{\text{SMI}}$
W/$\overline{\text{R}}$
■ Internal Cache Interface
● Power Management
▲ A20 Mask
♦ System Management Mode

1.5 Execution Pipeline

The TI486 execution path consists of five pipelined stages optimized for minimal instruction cycle times. These five stages are:

- ☐ Code Fetch
- ☐ Instruction Decode
- ☐ Microcode ROM Access
- ☐ Execution
- ☐ Memory/Register File Write-Back

These stages have been designed with hardware interlocks that permit successive instruction execution overlap.

The 16-byte instruction prefetch queue fetches code in advance and prepares it for decode, helping to minimize overall execution time. The instruction decoder then decodes four bytes of instructions per clock eliminating the need for a queue of decoded instructions. Sequential instructions are decoded quickly and provided to the microcode. Non-sequential operations do not have to wait for a queue of decoded instructions to be flushed and refilled before execution continues. As a result, both sequential and non-sequential instruction execution times are minimized.

The execution stage takes advantage of a RISC-like single-cycle execution unit and a 16-bit hardware multiplier. The write-back stage provides single-cycle 32-bit access to the on-chip cache and posts all writes to the cache and system bus using a two-deep write buffer. Posted writes allow the execution unit to proceed with program execution while the bus interface unit completes the write cycle.

1.6 On-Chip Cache

The TI486 on-chip cache maximizes overall performance by quickly supplying instructions and data to the internal execution pipeline. An external memory access takes a minimum of two clock cycles (zero wait states). For cache hits, the TI486 eliminates these two clock cycles by overlapping cache accesses with normal execution pipeline activity. Additional bus bandwidth is gained by presenting instructions and data to the execution pipeline up to 32 bits at a time compared to 16 bits per cycle for an external memory access.

The TI486 cache is a 1-KByte write-through unified instruction and data cache and lines are allocated only during memory read cycles. The cache can be configured as direct-mapped or as two-way set associative. The direct-mapped organization is a single set of 256 four-byte lines. When configured as two-way set associative, the cache organization consists of two sets of 128 four-byte lines and uses a Least Recently Used (LRU) replacement algorithm.

1.7 Power Management

1.7.1 Suspend Mode and Static Operation

The TI486 power management features allow a dramatic reduction in current consumption when the TI486 microprocessor is in suspend mode (typically less than 3 percent of the operating current). Suspend mode is entered either by a hardware or software initiated action. Using the hardware to initiate suspend mode involves a two-pin handshake using the $\overline{\text{SUSP}}$ and $\overline{\text{SUSPA}}$ signals.

The software initiates suspend mode through execution of the HALT instruction. Once in suspend mode, the TI486 power consumption is further reduced by stopping the external clock input. Since the TI486 is a static device, no internal CPU data is lost when the clock input is stopped.

1.7.2 3-V Operation

The TI486SLC/E-V version of the TI486SLC/E operates from either a 3-V or a 5-V supply. While operating with a 3-V supply, the power consumed by the TI486SLC/E-V is typically only 30 percent of the power consumed while operating at 5 V. The TI486SLC/E-V is available in 25-MHz speed.

The TI486DLC/E-V version of the TI486DLC/E operates from either a 3-V or a 5-V supply. While operating with a 3-V supply, the power consumed by the TI486DLC/E-V is typically only 30 percent of the power consumed while operating at 5 V. The TI486DLC/E-V is available in both 25-MHz and 33-MHz speeds.

1.8 System Management Mode (SMM)

System Management Mode (SMM) provides an additional interrupt and a separate address space which can be used for system power management or software transparent emulation of I/O peripherals. SMM is entered using the System Management Interrupt ($\overline{\text{SMI}}$) which has a higher priority than any other interrupt. While running in protected SMM address space, the SMI interrupt routine can execute without interfering with the operating system or application programs.

After reception of an $\overline{\text{SMI}}$, portions of the CPU state are automatically saved, SMM is entered and program execution begins at the base of SMM address space. The location and size of the SMM memory is programmable within the TI486. Seven SMM instructions have been added to the 486 instruction set that permit saving and restoring the total CPU state when in SMM mode.

Chapter 2

Programming Interface

In this chapter, the internal operations of the TI486 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. Also included is an overview of real, virtual 8086, and protected operating modes.

Topic **Page**

2.1 Processor Initialization

The TI486 is initialized when the RESET signal is asserted. The processor is placed in real mode and the registers listed in Table 2–1 and Table 2–2 are set to their initialized values. RESET invalidates and disables the TI486 cache, and turns off paging. When RESET is asserted, the TI486 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 350 to 450 CLK2 clock cycles (additional 2^{20} + 60 if self-test is requested) after de-assertion of RESET, the processor begins executing instructions at the top of physical memory (address location FF FFF0h for the SLC and FFFF FFF0h for the DLC). When the first intersegment JUMP or CALL is executed, address lines A23–A20 for the SLC or A31–A20 for the DLC are driven low for code segment-relative memory access cycles. While these address lines are low, the TI486 executes instructions only in the lowest 1 MByte of physical address space until system-specific initialization occurs via program execution.

Table 2–1. TI486SLC/E Initialized Register Contents

REGISTER	REGISTER NAME	INITIALIZED CONTENTS	COMMENTS
EAX	Accumulator	xx xxxxh	00 0000h indicates self-test passed.
EBX	Base	xx xxxxh	
ECX	Count	xx xxxxh	
EDX	Data	xx 0400 + Revision ID	Revision ID = 10h.
EBP	Base pointer	xx xxxxh	
ESI	Source index	xx xxxxh	
EDI	Destination index	xx xxxxh	
ESP	Stack pointer	xx xxxxh	
EFLAGS	Flag word	00 0002h	
EIP	Instruction pointer	00 FFF0h	
ES	Extra segment	0000h	Base address set to 00 0000h. Limit set to FFFFh.
CS	Code segment	F000h	Base address set to 00 0000h. Limit set to FFFFh.
SS	Stack segment	0000h	
DS	Data segment	0000h	Base address set to 00 0000h. Limit set to FFFFh.
FS	Extra segment	0000h	
GS	Extra segment	0000h	
IDTR	Interrupt Descriptor Table Register	Base=0, Limit=3FFh	
CR0	Machine status word	60 0010h	
CCR0	Configuration Control 0	00h	
CCR1	Configuration Control 1	xx xxx0 (binary)	
ARR1	Address Region 1	000Fh	4 GByte non-cacheable region.
ARR2	Address Region 2	0000h	
ARR3	Address Region 3	0000h	
ARR4	Address Region 4	0000h	
DR7	Debug register DR7	00 0000h	

Note: x = Undefined value

Table 2–2. TI486DLC/E Initialized Register Contents

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 + Revision ID	Revision ID = 10h.
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 0000 0000h. Limit set to FFFFh.
SS	Stack segment	0000h	
DS	Data segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
FS	Extra segment	0000h	
GS	Extra segment	0000h	
IDTR	Interrupt Descriptor Table Register	Base=0, Limit=3FFh	
CR0	Machine status word	6000 0010h	
CCR0	Configuration Control 0	00h	
CCR1	Configuration Control 1	xxxx xxx0 (binary)	
ARR1	Address Region 1	000Fh	4 GByte non-cacheable region.
ARR2	Address Region 2	0000h	
ARR3	Address Region 3	0000h	
ARR4	Address Region 4	0000h	
DR7	Debug register DR7	0000 0000h	

Note: x = Undefined value

2.2 Instruction Set Overview

The TI486 instruction set can be divided into eight types of operations:

- Arithmetic
- Bit manipulation
- Control transfer
- Data transfer
- High-level language support
- Operating system support
- Shift/rotate
- String manipulation

All TI486 instructions operate on as few as 0 operands and as many as 3 operands. A 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. 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 7 of this manual lists each instruction in the TI486 instruction set along with the associated opcodes, execution clock counts, and effects on the FLAGS register.

2.2.1 Lock Prefix

The LOCK prefix may be placed before certain instructions that read, modify, then write back to memory. The prefix asserts the $\overline{\text{LOCK}}$ 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).

2.3 Register Set

There are 43 accessible registers in the TI486 and these registers are grouped into two sets. The application register set contains the registers frequently used by applications 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 32-bit general purpose registers
- Six 16-bit segment registers
- One 32-bit flag register
- One 32-bit instruction pointer register.

The system register set is made up of the remaining registers which include:

- Three 32-bit control registers
- Two 48-bit and two 16-bit system address registers
- Two 8-bit and four 16-bit configuration registers
- Six 32-bit debug registers
- Five 32-bit test registers.

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

2.3.1 Application Register Set

The application register set (Figure 2–1) 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.

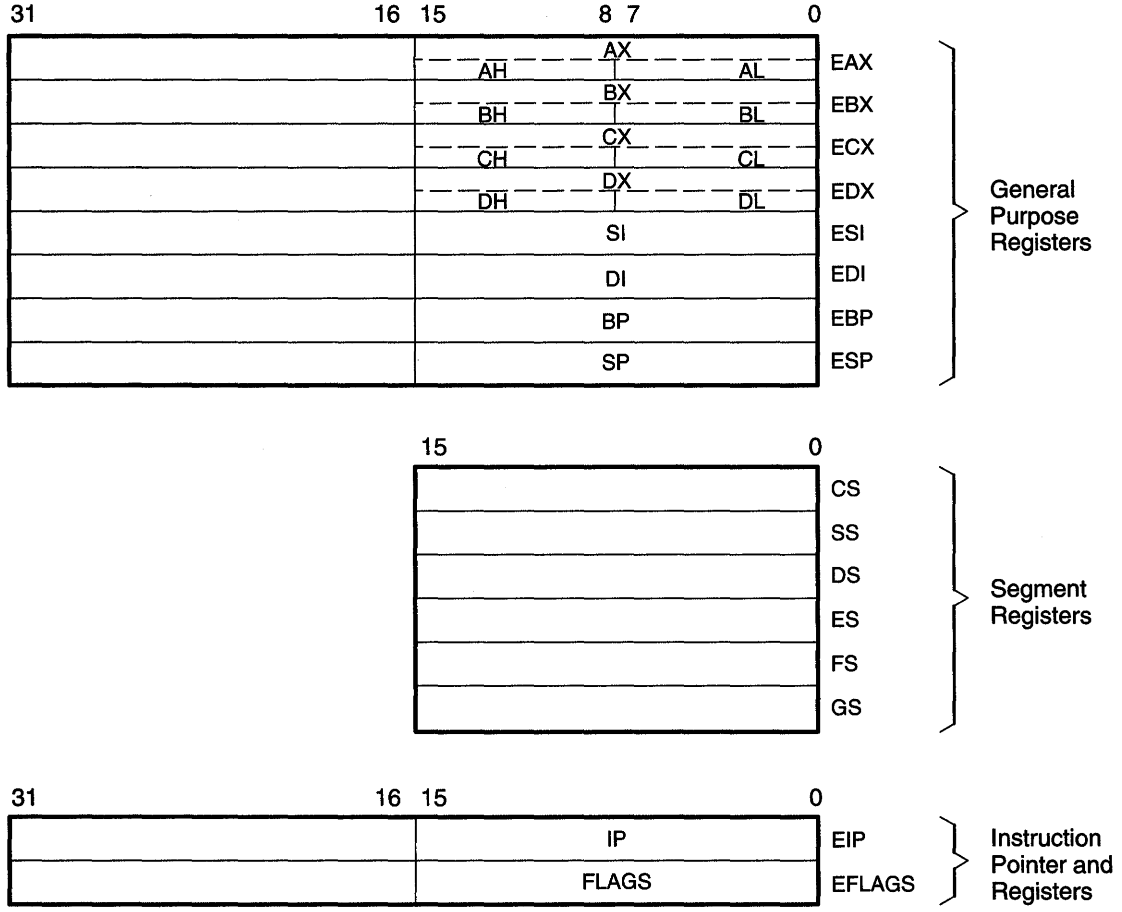
The General Purpose Registers contents are frequently modified by assembly language instructions and typically contain arithmetic and logical instruction operands.

The Segment Registers contain segment selectors, which index into tables located in memory. These tables hold the base address for each segment, as well as other information related to memory addressing.

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

The Instruction Pointer is a 32-bit register that points to the next instruction that the processor will execute. This register is automatically incremented by the processor as execution progresses.

Figure 2–1. Application Register Set



2.3.1.1 General Purpose Registers

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

Data Registers

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 source operand value specified by an instruction is smaller than the specified destination register, the upper bytes of the destination register are not affected when the operand is written to the register.

Pointer and Index Registers

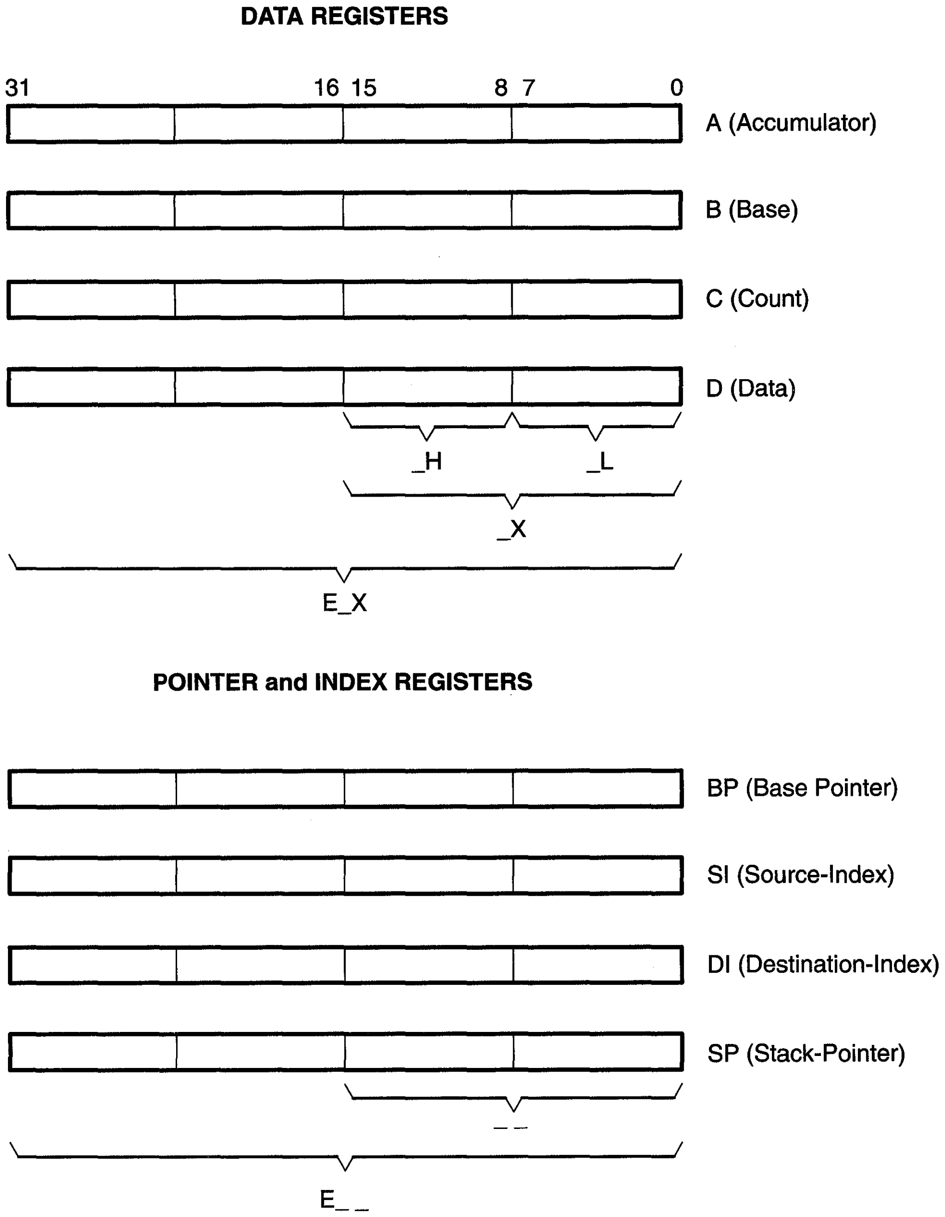
The pointer and index registers are listed below:

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

These registers can be addressed as 16- or 32-bit registers, with the "E" prefix indicating 32 bits. These registers can be used as general purpose registers, however, some instructions use a fixed assignment of these registers. For example, the 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 double-precision multiply and divide, I/O access, string operations, translate, loop, variable shift and rotate, and stack operations.

The TI486 processor implements a stack using the ESP register. This stack is accessed during the PUSH and POP instructions, procedure calls, procedure returns, interrupts, exceptions, and interrupt/exception returns. The microprocessor automatically adjusts the value of the ESP during operation of these instructions. The EBP register may be used to reference data passed on the stack during procedure calls. Local data may also be placed on the stack and referenced relative to BP. This register provides a mechanism to access stack data in high-level languages.

Figure 2–2. General Purpose Registers



2.3.1.2 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 locations 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.

In protected mode, a segment register holds a segment selector containing a 13-bit index, a table indicator (TI) bit, and a two-bit requested privilege level (RPL) field as shown in Figure 2–3.

The Index points into a *descriptor table* in memory and selects one of 8192 (2^{13}) segment descriptors contained in the descriptor table. A *segment descriptor* is an eight-byte value used to describe a memory segment by defining the segment base, the segment limit, and access control information. To address data within a segment, a 16-bit or 32-bit offset is added to the segment's base address. Once a segment selector has been loaded into a segment register, an instruction needs to specify the offset only.

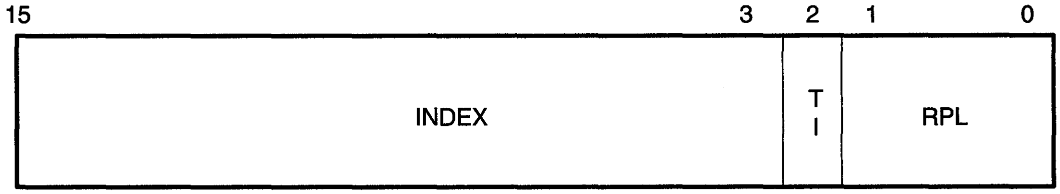
The Table Indicator (TI) bit of the selector defines which descriptor table the index points into. If TI=0, the index references the Global Descriptor Table (GDT). If TI=1, the index references the Local Descriptor Table (LDT). The GDT and LDT are described in more detail later in this chapter.

The Requested Privilege Level (RPL) field contains a 2-bit segment privilege level (00=most privileged, 11=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 program, the RPL overrides the current privilege level and the EPL is the lower privilege level. If the RPL bits indicate more privilege than the program, the current privilege level overrides the RPL and again the EPL is the lower privilege 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.

The processor automatically selects a default segment register for memory references. Table 2–3 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.

Figure 2–3. Segment Selector



TI = Table Indicator

RPL = Requested Privilege Level

Table 2–3. Segment Register Selection Rules

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

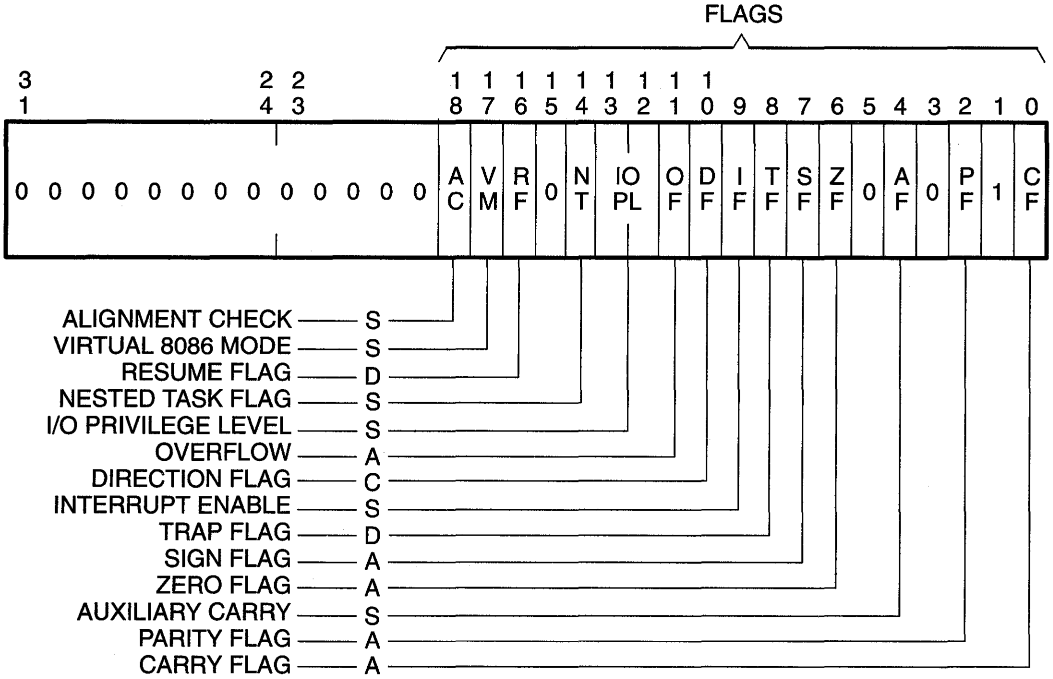
2.3.1.3 Instruction Pointer Register

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).

2.3.1.4 Flags Register

The Flags Register, EFLAGS, contains status information and controls certain operations on the TI486 microprocessor. The lower 16 bits of this register are referred to as the FLAGS register that is used when executing 8086 or 80286 code. The flag bits are shown in Figure 2–4 and defined in Table 2–4.

Figure 2–4. EFLAGS Register



A = Arithmetic Flag, D = Debug Flag, S = System Flag, C = Control Flag
0 or 1 Indicates Reserved

Table 2–4. EFLAGS Definitions

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

2.3.2 System Register Set

The system register set (Figure 2–5) consists of registers not generally used by application programmers. These registers are typically used by system level programmers who generate operating systems and memory management programs.

The Control Registers control certain aspects of the TI486 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.

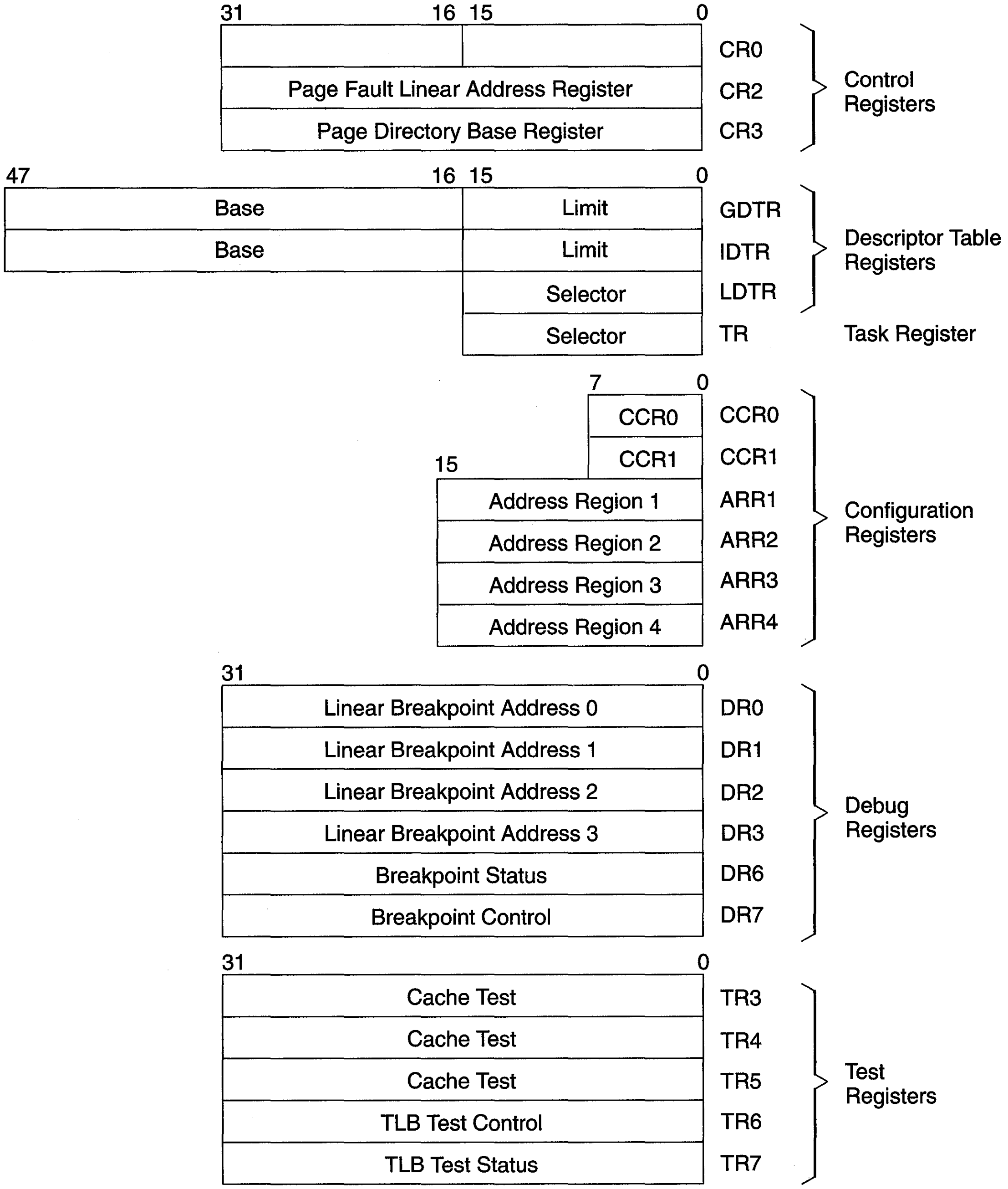
The Descriptor Table Registers and the Task Register can also be referred to as system address or memory management registers. These registers consist of two 48-bit and two 16-bit registers. These registers specify the location of the data structures that control the segmentation used by the TI486 microprocessor. Segmentation is one available method of memory management.

The Configuration Registers are used to control the TI486 on-chip cache operation, power management features, and System Management Mode. The cache, power management, and SMM features can be enabled or disabled by writing to these registers. Non-cacheable areas of physical memory are also defined through the use of these registers.

The Debug Registers provide debugging facilities for the TI486 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 1-KByte cache and the translation lookaside buffer (TLB). The TLB is used as a cache for translating linear addresses to physical addresses when paging is enabled. In the following sections, the system register set is described in greater detail.

Figure 2–5. System Register Set



CCR0 = Configuration Control 0
CCR1 = Configuration Control 1

2.3.2.1 Control Registers

The control registers (CR0, CR2, and CR3) are shown in Figure 2–6. 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 CR0 bit definitions are described in Table 2–5. The reserved bits in the CR0 should not be modified.

When paging is enabled and a page fault is generated, the CR2 register retains the 32-bit linear address of the address that caused the fault. CR3 contains the 20-bit 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 reserved.

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.

Figure 2–6. Control Registers

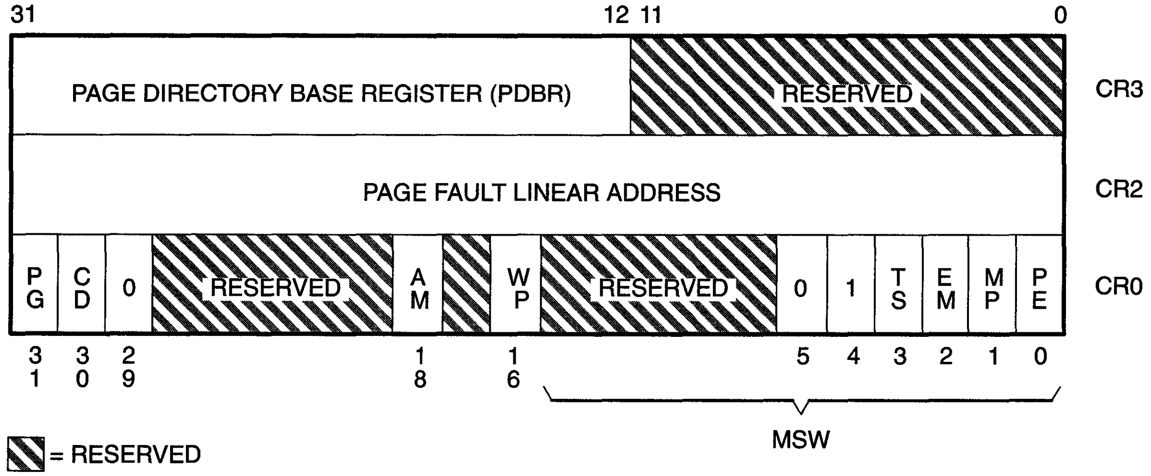


Table 2–5. CR0 Bit Definitions

BIT POSITION	NAME	FUNCTION
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.
1	MP	Monitor Processor Extension. If MP=1 and TS=1, a WAIT instruction causes fault 7. The TS bit is set to 1 on task switches by the CPU. Floating-point instructions are not affected by the state of the MP bit. The MP bit should be set to one during normal operations.
2	EM	Emulate Processor Extension. If EM=1, all floating-point instructions cause a fault 7.
3	TS	Task Switched. Set whenever a task switch operation is performed. Execution of a floating-point instruction with TS=1 causes a device not available (DNA) fault. If MP=1 and TS=1, a WAIT instruction also causes a DNA fault.
4	1	Reserved. Do not attempt to modify.
5	0	Reserved. Do not attempt to modify.
16	WP	Write Protect. Protects read-only pages from supervisor write access. The 386-type CPU allows a read-only page to be written from privilege level 0–2. The TI486 CPU is compatible with the 386-type CPU when WP=0. WP=1 forces a fault on a write to a read-only page from any privilege level.
18	AM	Alignment Check Mask. If AM=1, the AC bit in the EFLAGS register is unmasked and allowed to enable alignment check faults. Setting AM=0 prevents AC faults from occurring.
29	0	Reserved. Do not attempt to modify.
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.
31	PG	Paging Enable Bit. If PG=1 and protected mode is enabled (PE=1), paging is enabled.

2.3.2.2 Descriptor Table Registers and Descriptors

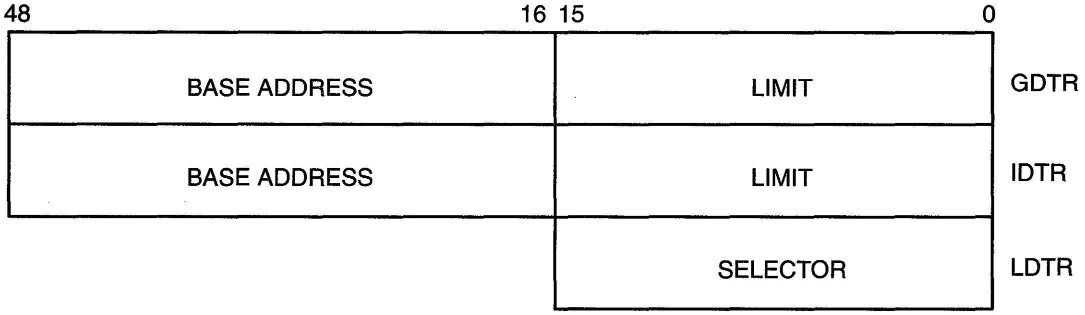
Descriptor Table Registers

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

The Global Descriptor Table Register (GDTR) holds a 32-bit 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 TI486 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 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.

Figure 2–7. Descriptor Table Registers



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 description 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.

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 TSS
- Gate Descriptors that define task gates, interrupt gates, trap gates, and call gates.

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

Figure 2–8. Application and System Segment Descriptors

31 24	23	22	21	20	19 16	15	14 13	12	11 8	7 0	
BASE 31–24	G	D	0	AVL	LIMIT 19–16	P	DPL	DT	TYPE	BASE 23–16	+4
BASE 15–0						LIMIT 15–0					+0

Table 2–6. Segment Descriptor Bit Definitions

BIT POSITION	MEMORY OFFSET	NAME	DESCRIPTION
31–24 7–0 31–16	+4 +4 +0	BASE	Segment base address. 32-bit linear address that points to the beginning of the segment.
19–16 15–0	+4 +0	LIMIT	Segment limit. In real mode, segment limit is always 64 KBytes (0FFFFh)
23	+4	G	Limit granularity bit: 0=byte granularity, 1=4 KBytes (page) granularity.
22	+4	D	Default length for operands and effective addresses. Valid for code and stack segments only: 0=16 bit, 1=32-bit.
20	+4	AVL	Segment available.
15	+4	P	Segment present.
14–13	+4	DPL	Descriptor privilege level.
12	+4	DT	Descriptor type: 0=system, 1=application
11–8	+4	TYPE	Segment type. System descriptor (DT=0): 0010=LDT descriptor 1001=TSS descriptor, task not busy 1011=TSS descriptor, task busy
11		E	Application descriptor (DT=1): 0=data, 1=executable
10		C/D	If E=0: 0=expand up, limit is upper bound of segment. 1=expand down, limit is lower bound of segment If E=1: 0=non-conforming 1=conforming (runs at privilege level of calling procedure)
9		R/W	If E=0: 0=non-readable 1=readable If E=1: 0=non-writable 1=writable
8		A	0=not accessed, 1=accessed

Gate Descriptors provide protection for executable segments operating at different privilege levels. Figure 2–9 illustrates the format for Gate Descriptors and Table 2–7 lists the corresponding bit definitions.

Task Gate descriptors are used to switch the CPU's context during a task switch. The selector portion of the Task Gate descriptor locates a Task State Segment. Task Gate descriptors can be located in the GDT, LDT or IDT.

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 be located only 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 2–9. Gate Descriptor

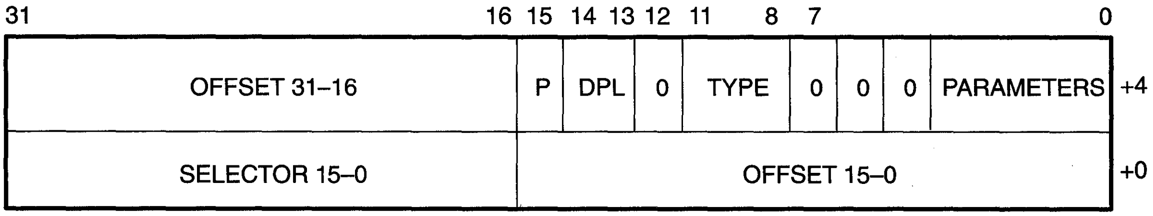


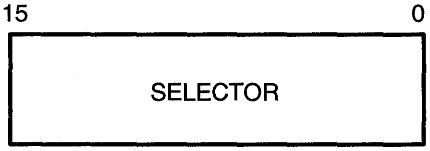
Table 2–7. Gate Descriptor Bit Definitions

BIT POSITION	MEMORY OFFSET	NAME	DESCRIPTION
31–16 15–0	+4 +0	OFFSET	Offset used during a call gate to calculate the branch target.
31–16	+0	SELECTOR	Segment selector used during a call gate to calculate the branch target.
15	+4	P	Segment present
14–13	+4	DPL	Descriptor privilege level
11–8	+4	TYPE	Segment type: 0100=16-bit call gate 0101=tack 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
4–0	+4	Parameters	Number of 32-bit parameters to copy from the caller's stack to the called procedure's stack.

2.3.2.3 Task Register

The Task Register (TR) holds a 16-bit selector for the current Task State Segment (TSS) table as shown in Figure 2–10. The TR is loaded and stored via the LTR and STR instructions, respectively. The TR can be accessed only during protected mode and can be loaded only when the privilege level is 0 (most privileged).

Figure 2–10. Task Register



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 286-type 16-bit TSS or a 386/486-type 32-bit TSS as shown in Figure 2–11 and Figure 2–12. An I/O permission bit map is referenced in the 32-bit TSS by the I/O Map Base Address.

Figure 2–11. 32-Bit Task State Segment (TSS) Table

31 16	15	0	
I/O MAP BASE ADDRESS	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	T	+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

Figure 2–12. 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	+1Ch
SP	+1Ah
BX	+18h
DX	+16h
CX	+14h
AX	+12h
FLAGS	+10h
IP	+Eh
SP FOR PRIVILEGE LEVEL 2	+Ch
SS FOR PRIVILEGE LEVEL 2	+Ah
SP FOR PRIVILEGE LEVEL1	+8h
SS FOR PRIVILEGE LEVEL 1	+6h
SP FOR PRIVILEGE LEVEL 0	+4h
SS FOR PRIVILEGE LEVEL 0	+2h
BACK LINK (OLD TSS SELECTOR)	+0h

2.3.2.4 Configuration Registers

The TI486 contains six registers that do not exist on other 80x86 microprocessors. These registers include two Configuration Control Registers (CCR0 and CCR1) and four Address Region Registers (ARR1 through ARR4) as listed in Table 2–8 and Table 2–9. The CCR and ARR registers exist in I/O memory space and are selected by a “register index” number via I/O port 22h. I/O port 23h is used for data transfer.

Each I/O port 23h data transfer must be preceded by an I/O port 22h register selection, otherwise the second and later I/O port 23h operations are directed off-chip and produce external I/O cycles. If the register index number is outside the C0h–CFh range, external I/O cycles will also occur.

The CCR0 register (Table 2–10) defines the type of cache and determines if the 64-KByte memory area on 1-MByte boundaries and the 640-KByte to 1-MByte area are cacheable. This register also enables certain pins associated with cache control and suspend mode.

The CCR1 register (Table 2–11) is used to set up internal cache operation and System Management Mode (SMM). The ARR registers (Figure 2–13, Figure 2–14, and Table 2–8, Table 2–9) are used to define the location and size of the memory regions associated with the internal cache. ARR1–ARR3 define three write-protected or non-cacheable memory regions as designated by CCR1 bits WP1–WP3. ARR4 defines an SMM memory space or non-cacheable memory region as defined by CCR1 bit SM4. Other CCR1 bits enable RPL and SMM pins and control SMM memory access. The SMAC bit allows access to defined SMM space while not in an SMI service routine. The MMA bit allows access to main memory that overlaps with SMM memory while in an SMI service routine for data access only.

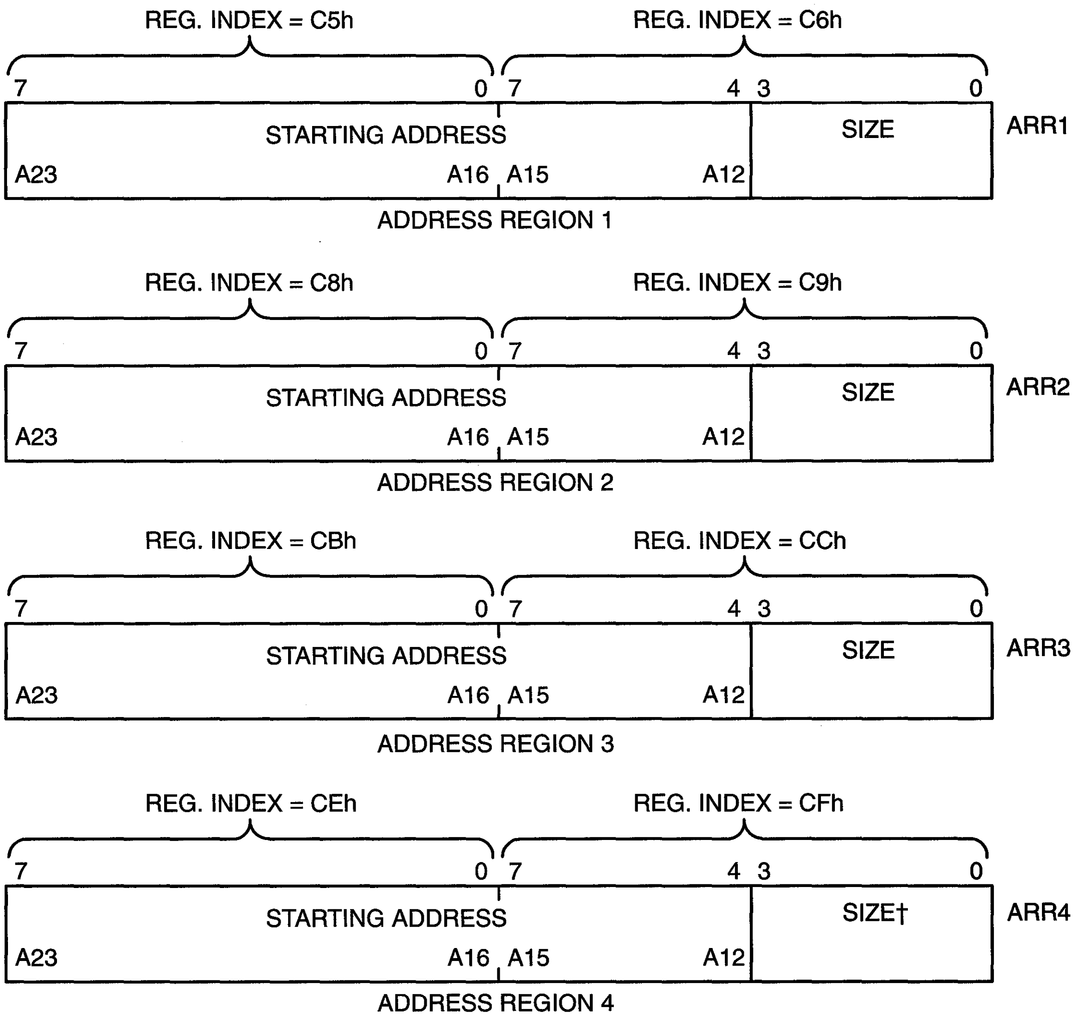
The ARR registers define address regions using a starting address and a block size. The non-cacheable region block sizes range from 4 KBytes to 4 GBytes (Table 2–12). A block size of zero disables the address region. The starting address of the address region must be on a block size boundary. For example, a 128 KByte block is allowed to have a starting address of 0 KBytes, 128 KBytes, 256 KBytes, etc. The SMM memory region size is restricted to a maximum of 16 MBytes. The block size must be defined for $\overline{\text{SMI}}$ to be recognized.

Table 2–8. TI486SLC/E Configuration Control Registers

REGISTER NAME	REGISTER INDEX	WIDTH
Configuration Control 0 CCR0	C0h	8
Configuration Control 1 CCR1	C1h	8
Address Region 1 ARR1	C5h–C6h	16
Address Region 2 ARR2	C8h–C9h	16
Address Region 3 ARR3	CBh–CCh	16
Address Region 4 ARR4	CEh–CFh	16

Note: The following register index numbers are reserved: C2h, C3h, C4h, C7h, CAh, CDh, and D0h through FFh.

Figure 2–13. TI486SLC/E Address Region Registers (ARR1–ARR4)



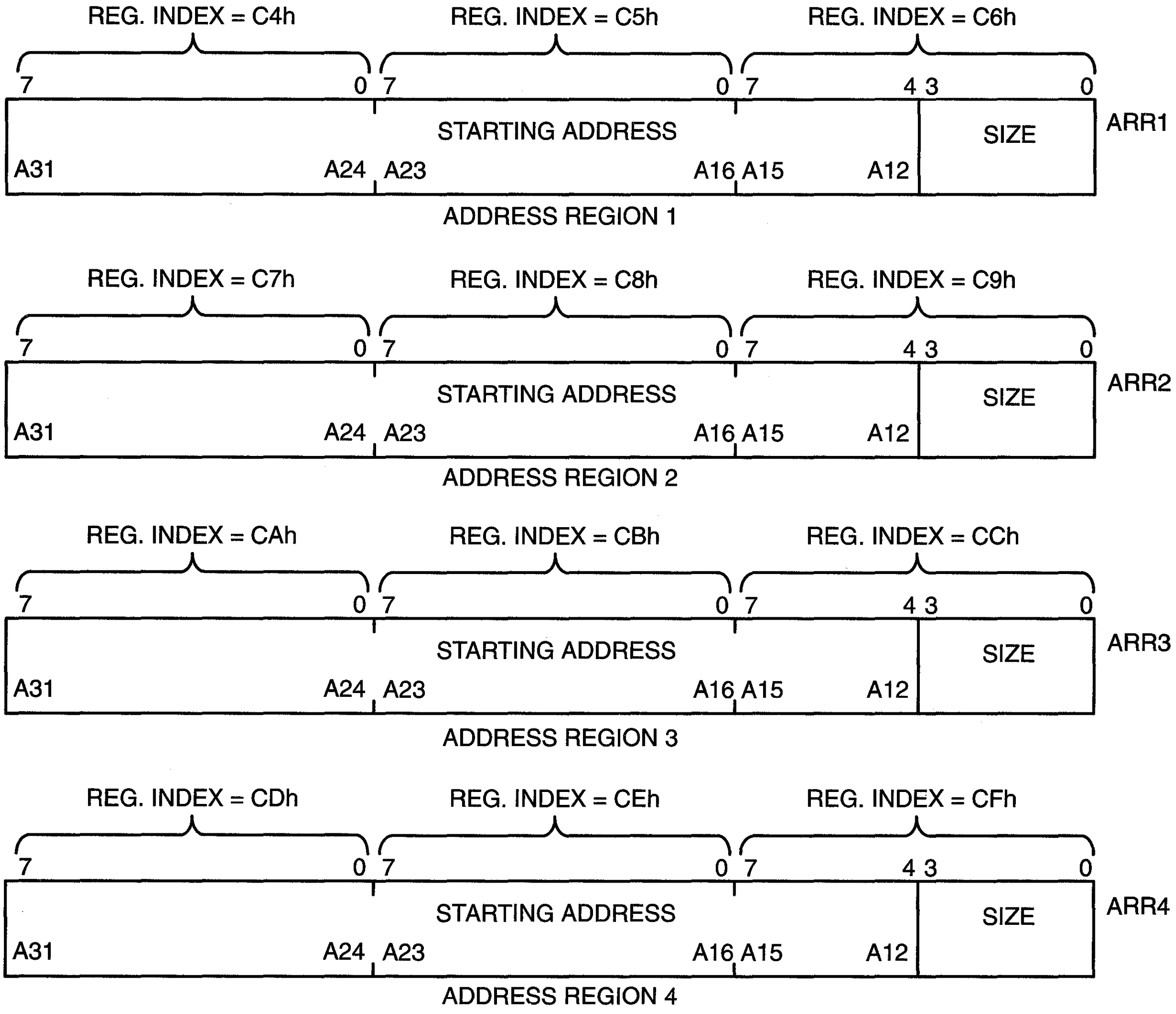
†ARR4 (SIZE) must be 4Kbytes to 16 Mbytes if ARR4 is defined as SMM memory space.

Table 2–9. TI486DLC/E Configuration Control Registers

REGISTER NAME	REGISTER INDEX	WIDTH
Configuration Control 0 CCR0	C0h	8
Configuration Control 1 CCR1	C1h	8
Address Region 1 ARR1	C4h–C6h	24
Address Region 2 ARR2	C7h–C9h	24
Address Region 3 ARRR3	CAh–CCh	24
Address Region 4 ARR4	CDh–CFh	24

Note: The following register index numbers are reserved: C2h, C3h, and D0h through FFh.

Figure 2–14. TI486DLC/E Address Region Registers (ARR1–ARR4)



†ARR4 (SIZE) must be 4Kbytes to 16 Mbytes if ARR4 is defined as SMM memory space.

Table 2–10. CCR0 Bit Definitions

BIT POSITION	REGISTER INDEX	DESCRIPTION
0	NC0	Non-cacheable 1-MByte Boundaries If=1: Sets the first 64 KBytes at each 1-MByte boundary as non-cacheable.
1	NC1	Non-cacheable Upper Memory Area If=1: Sets 640-KByte to 1–MByte memory region non-cacheable.
2	A20M	Enable $\overline{\text{A20M}}$ pin If=1: Enables $\overline{\text{A20M}}$ input pin; otherwise pin is ignored.
3	KEN	Enable $\overline{\text{KEN}}$ pin If = 1: Enables $\overline{\text{KEN}}$ input pin; otherwise pin is ignored.
4	FLUSH	Enable $\overline{\text{FLUSH}}$ pin If = 1: Enables $\overline{\text{FLUSH}}$ input pin; otherwise pin is ignored.
5	BARB	Enable Cache Flush during Hold If = 1: Enables flushing of the internal cache when hold state is entered.

6	CO	Cache Type Select
		If = 1: Selects direct-mapped cache.
		If = 0: Selects 2-way set-associative cache.

7	SUS	Enable Suspend Pins If =1: Enables $\overline{\text{SUSP}}$ input pin and $\overline{\text{SUSPA}}$ output pin. If = 0: $\overline{\text{SUSPA}}$ output pin floats; $\overline{\text{SUSP}}$ input pin is ignored.

Table 2–11. CCR1 Bit Definitions

BIT POSITION	REGISTER INDEX	DESCRIPTION
0	—	Reserved
1	SMI	Enable SMM Pins. If=1: $\overline{\text{SMI}}$ input/output pin and $\overline{\text{SMADS}}$ output pin are enabled. If= 0: $\overline{\text{SMI}}$ input pin ignored and $\overline{\text{SMADS}}$ output pin floats.
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 $\overline{\text{SMADS}}$ output active. $\overline{\text{SMI}}$ input is ignored. If = 0: No effect on access.
3	NMAC	Main Memory Access. If = 1: All data accesses which occur within an SMI service routine (or when SMAC = 1) will access main memory instead of SMM memory space. If = 0: No effect on access.
4	WP1	Access Region 1 Control If = 1: Region 1 is write protected and cacheable. If = 0: Region 1 is non-cacheable.
5	WP2	Access Region 2 Control If = 1: Region 2 is write protected and cacheable. If = 0: Region 2 is non-cacheable.
6	WP3	Access Region 3 Control If = 1: Region 3 is write protected and cacheable. If = 0: Region 3 is non-cacheable.

7	SM4	Access Region 4 Control
		If = 1: Region 4 is non-cacheable SMM memory space.
		If = 0: Region 4 is non-cacheable. $\overline{\text{SMI}}$ input ignored.

Table 2–12. ARR1–ARR4 Block 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 GBytes

2.3.2.5 Debug Registers

Six debug registers (DR0–DR3, DR6 and DR7), shown in Figure 2–15 and Figure 2–16, support debugging on the TI486. 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.

Figure 2–15. TI486SLC/E 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	G D	0	0	1	G E	L E	G 3	L 3	G 2	L 2	G 1	L 1	G 0	L 0	DR7
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	B T	B S	0	0	1	1	1	1	1	1	1	1	B 3	B 2	B 1	B 0	DR6
RESERVED																																DR5
RESERVED																																DR4
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.

Figure 2–16. TI486DLC/E Debug Registers

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15	14	13	12 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	000	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	1	011	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.

Table 2–13. DR6 and DR7 Field Definitions

REGISTER	FIELD	NUMBER OF BITS	DESCRIPTION
DR6	Bi	1	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	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	BS is set by the processor if the debug exception was triggered by the single-step execution mode (TF flag in EFLAGS set).
DR7	R/Wi	2	Applies to the DRi breakpoint address register: 00 – Break on instruction execution only 01 – Break on data writes only 10 – Not used 11 – Break on data reads or writes
	LENi	2	Applies to the DRi breakpoint address register: 00 – One byte length 01 – Two byte length 10 – Not used 11 – Four byte length
	Gi	1	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	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	Global disable of debug register access. GD bit is cleared whenever a debug exception occurs.

The debug address registers DR0–DR3 each contain the linear address for one of four possible breakpoints. 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 execution as well as data access breakpoints. Instruction execution and data access breakpoints are always taken before execution of the instruction that matches the breakpoint.

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. Table 2–13 lists the field definitions for the DR6 and DR7 registers.

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.

2.3.2.6 Test Registers

The five test registers, shown in Figure 2–17, 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. Table 2–14 and Table 2–15 list the bit definitions for the TR6 and TR7 registers.

TLB Test Registers

The TI486 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.

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 are 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.

The TLB Test Data Register (TR7) contains the upper 20 bits of the physical address (TLB data field), two 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.

Figure 2–17. Test Registers

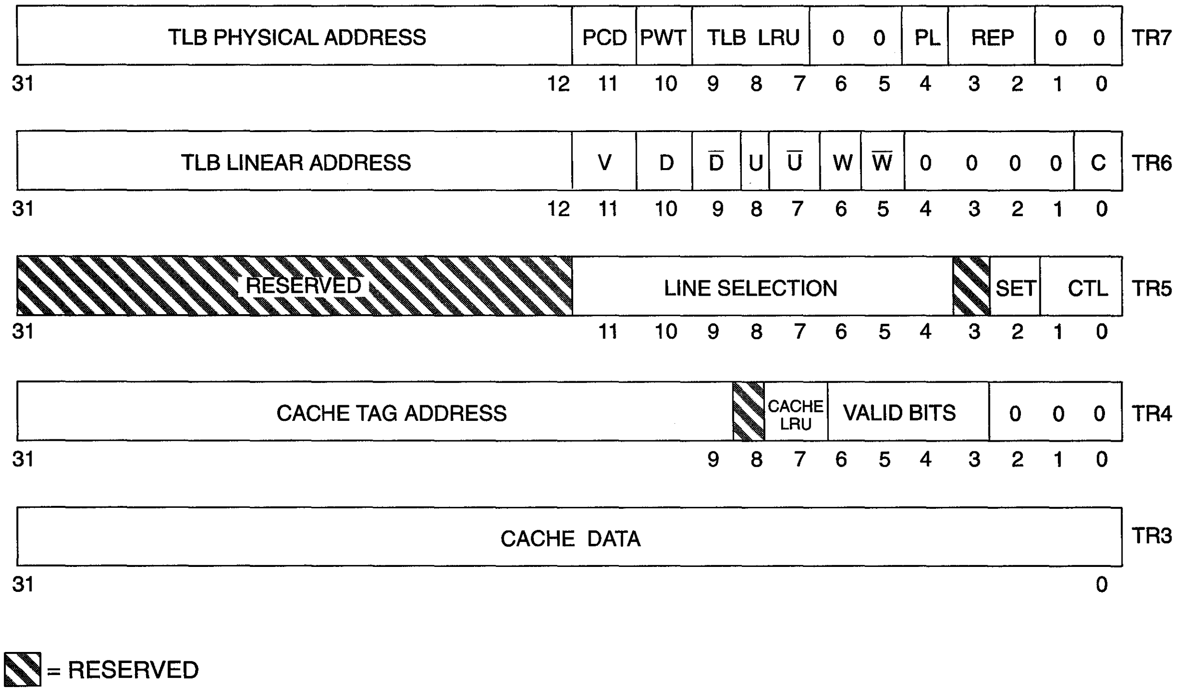


Table 2–14. TR6 and TR7 Bit Definitions

REGISTER NAME	BIT POSITION	DESCRIPTION
TR6	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.
	11	Valid bit (V). TLB lookup: Always set to 1. TLB write: If set, indicates that the TLB entry contains valid data. If clear, target entry is invalidated.
	10–9	Dirty attribute bit and its complement (D, $\overline{D}$). (Refer to Table 2–15).
	8–7	User/supervisor attribute bit and its complement (U, $\overline{U}$). (Refer to Table 2–15).
	6–5	Read/write attribute bit and its complement (R, $\overline{R}$). (Refer to Table 2–15).

	0	Command bit (C).
		If=0: TLB write.
		If=1: TLB lookup.

TR7	31–12	Physical address. TLB lookup: data field from the TLB. TLB write: data field written into the TLB.
	11	Page-level cache disable bit (PCD). Corresponds to the PCD bit of a page table entry.
	10	Page-level cache write-through bit (PWT). Corresponds to the PWT bit of a page table entry.
	9–7	LRU bits. TLB lookup: LRU bits associated with the TLB entry prior to the TLB lookup. TLB write: ignored.
	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.
	3–2	Set selection (REP). 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 2–15. TR6 Attribute Bit Pairs

BIT (B)	BIT COMPLEMENT ($\overline{B}$)	EFFECT ON TLB LOOKUP	EFFECT ON TLB WRITE
0	0	Do not match	Undefined
0	1	Match if the bit is 0	Clear the bit
1	0	Match if the bit is 1	Set the bit
1	1	Match is the bit is 1 or 0	Undefined

Cache Test Registers

The TI486 on-chip cache can be configured either as a direct-mapped (256 entries) or as a two-way set associative memory (128 entries per set). Each entry consists of a 23-bit tag, 32-bit data field, four valid bits, and an LRU bit. The 23-bit tag represents the high-order 23 bits of the physical address. The 32-bit data represents the four bytes of data currently in memory at the physical address represented by the tag. The four valid bits indicate which of the four data bytes contain valid data. The LRU bit is accessed only when the cache is configured as two-way set associative and indicates which of the two sets was most recently accessed.

The TI486 contains three test registers that allow testing of its internal cache. Using these registers, cache test writes and reads may be performed. Cache test writes cause the data in TR3 to be written to the selected set and entry in the cache. Cache test reads allow inspection of the data, valid bits and the LRU bit for the cache entry. For data to be written to the allocated entry, the valid bits for the entry must be set prior to the write of the data. Bit definitions for the cache test registers are shown in Table 2–16.

Table 2–16. TR3–TR5 Bit Definitions

REGISTER NAME	BIT POSITION	DESCRIPTION
TR3	31–10	Cache data. Cache read: data accessed from the cache. Cache write: to be written into the cache.
TR4	31–9	Tag address. Cache read: tag address from which data is read. Cache write: data written into the tag address of the selected line.
	7	LRU Cache read: the LRU bit associated with the cache line. Cache write: ignored.
	6–3	Valid bits Cache reads: four valid bits for the accessed line, (one bit per byte). Cache writes: valid bits written into the line.
TR5	10–4	Line selection. Selects one of 128 lines.

	2	Set selection
		If=0: set 0 is selected
		If=1: set 1 is selected

	1–0	Control bits. These bits control reading or writing the cache.
		If=00: Ignored
		If=01: Cache write
		If=10: Cache read
		If=11: Cache flush (marks all entries as invalid).

2.4 Address Spaces

The TI486 can directly address either memory or I/O space. Figure 2–18 and Figure 2–19 illustrate the range of addresses available for memory address space and I/O address space. For the TI486SLC/E, the addresses for physical memory range between 00 0000h and FF FFFFh (16 MBytes). For the TI486DLC/E, the addresses for physical memory range between 0000 0000h and FFFF FFFFh (4 GBytes). The accessible I/O addresses space for both the TI486SLC/E and TI486DLC/E ranges between 00 0000h and 00 FFFFh (64 KBytes). The coprocessor communication space for the TI486SLC/E exists in upper I/O space between 80 00F8h and 80 00FFh. The coprocessor communication space for the TI486DLC/E exists in the upper I/O space between 8000 00F8h and 8000 00FFh. These coprocessor I/O ports are automatically accessed by the CPU whenever an ESC opcode is executed. The I/O locations 22h and 23h are used for TI486SLC/E and TI486DLC/E configuration register access.

Figure 2–18. TI486SLC/E Memory and I/O Address Spaces

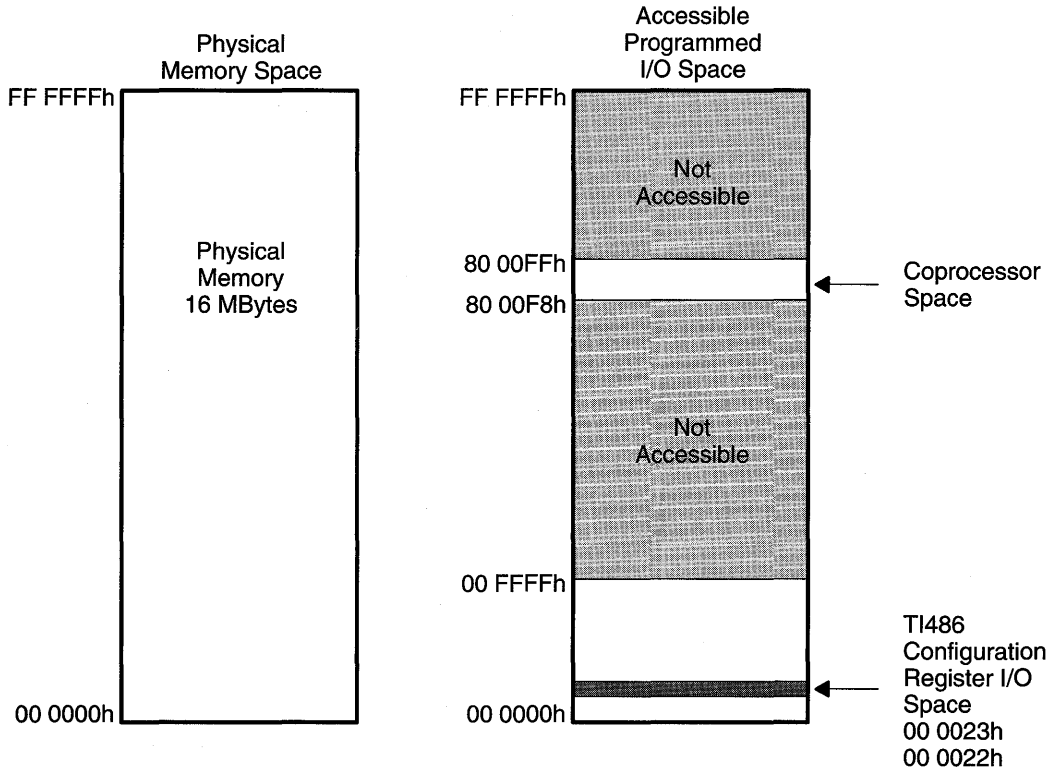
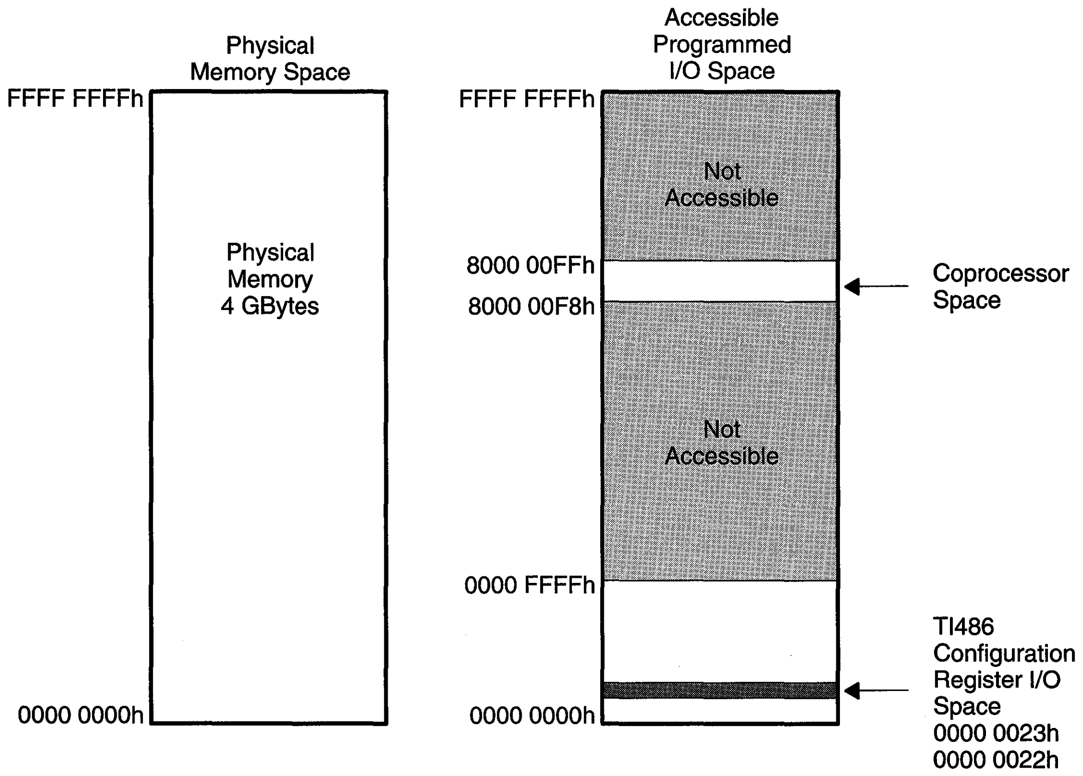


Figure 2–19. TI486DLC/E Memory and I/O Address Spaces



2.4.1 I/O Address Space

The TI486 I/O address space is accessed using IN and OUT instructions to addresses referred to as "ports". The accessible I/O 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/$\overline{\text{IO}}$ pin to be driven low, thereby selecting the I/O space instead of memory space for loading or storing data. The upper 8 address bits are always driven low during IN and OUT instruction port accesses.

The TI486 configuration registers reside within the I/O 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 TI486 index range (C0h to CFh) result in external I/O cycles and do not affect the on-chip configuration registers. Reads of port 22h are always directed off-chip.

2.4.2 Memory Address Space

The TI486SLC/E directly addresses up to 16 MBytes of physical memory and the TI486DLC/E 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 TI486, 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 and address a physical memory location.

Memory management mechanisms on the TI486 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 TI486 memory address space.

2.4.2.1 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 determined from 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 2–20 illustrates the calculation of the offset address.

Nine valid combinations of the base, index, scale factor, and displacement can be used with the TI486 instruction set. These combinations are listed in Table 2–17. The base and index both refer to contents of a register as indicated by [Base] and [Index].

Figure 2–20. Offset Address Calculation

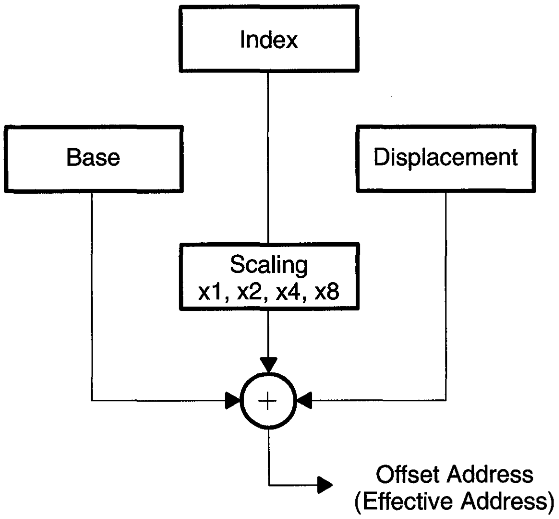


Table 2–17. Memory Addressing Modes

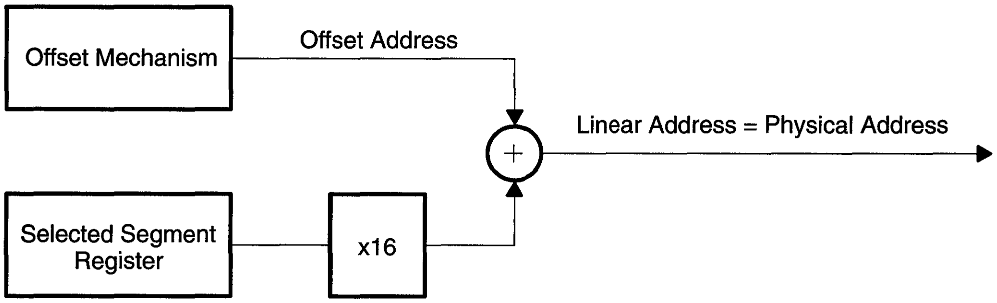
ADDRESSING MODE	BASE	INDEX	SCALE FACTOR (SF)	DISPLACEMENT (DP)	OFFSET ADDRESS (OA) CALCULATION
Direct				X	OA = DP
Register indirect	X				OA = [BASE]
Based	X			X	OA = [BASE] + DP
Index		X		X	OA = [INDEX] + DP
Scaled index		X	X	X	OA = ([INDEX] * SF) + DP
Based index	X	X			OA = [BASE] + [INDEX]
Based scaled index	X	X	X		OA = [BASE] + ([INDEX] * SF)
Based index with displacement	X	X		X	OA = [BASE] + [INDEX] + DP
Based scaled index with displacement	X	X	X	X	OA = [BASE] + ([INDEX] * SF) + DP

2.4.2.2 Real Mode Memory Addressing

In real mode operation, the TI486 addresses only the lowest 1 MByte (2^{20}) of memory. To calculate a physical memory address, the 16-bit segment base address located in the selected segment register is shifted left by four bits and then the 16-bit offset address is added. For the TI486SLC/E, the resulting 20-bit address is then extended with four zeros in the upper address bits to create the 24-bit physical address. For the TI486DLC/E, the resulting 20-bit address is then extended with 12 zeros in the upper address bits to create the 32-bit physical address. Figure 2–21 illustrates the real mode address calculation. Physical addresses beyond 1 MByte cause a segment limit overrun exception.

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 address memory in the first 64 KBytes above 1 MByte.

Figure 2–21. Real Mode Address Calculation



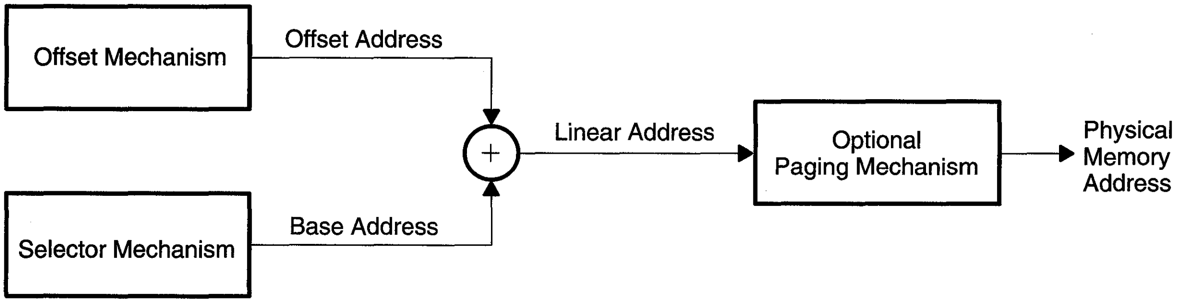
2.4.2.3 Protected Mode Memory Addressing

In protected mode, three mechanisms calculate a physical memory address (Figure 2–22).

- □ 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 the real and protected mode. The selector and paging mechanisms are described in the following paragraphs.

Figure 2–22. Protected Mode Address Calculation

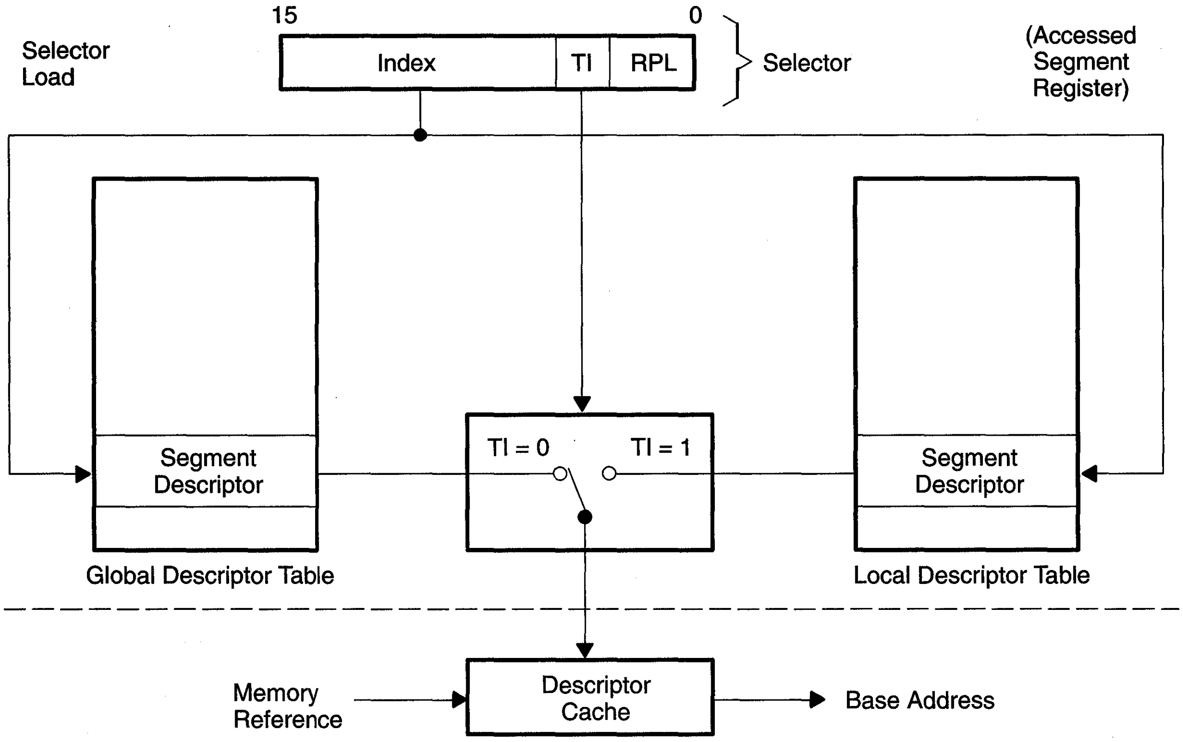


Selector Mechanism

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 TI486 as a result of loading the selector. The cached descriptor contents are not visible to the programmer. When a memory reference occurs in protected mode, the linear address is generated by adding the segment base address in the hidden portion of the segment register to the offset address. If paging is not enabled, this linear address is used as the physical memory address. Figure 2–23 illustrates the operation of the selector mechanism.

Figure 2–23. Selector Mechanism



Paging Mechanism

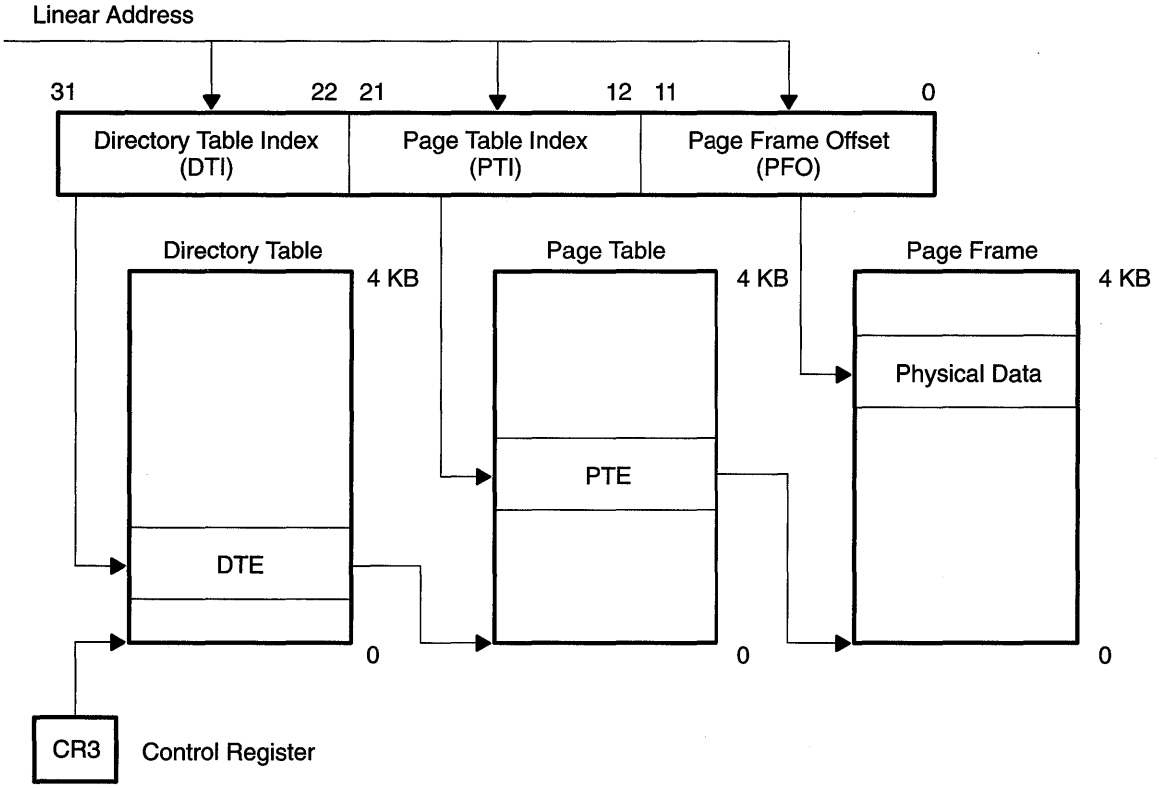
The paging mechanism supports a memory subsystem that simulates a large address space with a small amount of RAM and disk storage. The paging mechanism either translates a linear address to its corresponding physical address or generates an exception if the required page is not currently present in RAM.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

A page is addressed by using two levels of tables as illustrated in Figure 2–24. 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 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, locate the desired data within the page frame.

Since the page directory table can point to 1K page tables, and each page table can point to 1 K of page frames, a total of 1M of page frames can be implemented. Since each page contains 4 KBytes, up to 4 GBytes of virtual memory can be addressed by the TI486 with a single page directory table.

Figure 2–24. Paging Mechanism



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 2–25 and listed in Table 2–18.

Figure 2–25. Directory and Page Table Entry (DTE and PTE) Format

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

▨ = RESERVED

Table 2–18. Directory and Page Table Entry (DTE and PTE) Bit Definitions

BIT POSITION	FIELD NAME	DESCRIPTION
31–12	Base Address	Specifies the base address of the page or page table.
11–9	—	Undefined and available to the programmer.
8–7	—	Reserved and not available to the programmer.
6	D	Dirty bit. If set, indicates that a write access has occurred to the page (PTE only, undefined in DTE).
5	A	Accessed flag. If set, indicates that a read access or write access has occurred to the page.
4	PCD	Page caching disable flag. If set, indicates that the page is not cacheable in the on-chip cache.
3	—	Reserved and not available to the programmer.
2	U/S	User/supervisor attribute. If set (user), page is accessible at all privilege levels. If clear (supervisor), page is accessible only when CPL ≤ 2.
1	W/R	Write/read attribute. If set (write), page is writable. If clear (read), page is read only.
0	P	Present flag. If set, indicates that the page is present in RAM memory, and validates the remaining DTE/PTE bits. If clear, indicates that the page is not present in memory and the remaining DTE/PTE bits can be used by the programmer.

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 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.

Translation Look-Aside Buffer

The translation look-aside buffer (TLB) is a cache for the paging mechanism and replaces the two-level page table lookup procedure for cache hits. The TLB is a four-way set associative 32-entry page table cache that automatically keeps the most commonly used page table entries in the processor. The 32-entry TLB, coupled with a 4K page size, results in coverage of 128 KBytes of memory addresses.

The TLB must be flushed when entries in the page tables are changed. The TLB is flushed whenever the CR3 register is loaded. An individual entry in the TLB can be flushed using the INVLPG instruction.

2.5 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.

All 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.

2.5.1 Interrupts

External events can interrupt normal program execution by using one of the three interrupt pins on the TI486.

- Non-maskable Interrupt (NMI pin)
- Maskable Interrupt (INTR pin)
- SMM Interrupt ($\overline{\text{SMI}}$ 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 usually 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 the 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 are still 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 TI486 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 the INTR or NMI. After $\overline{\text{SMI}}$ 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.

2.5.2 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 during single stepping, at a breakpoint, or by software interrupt instruction (INT 0, 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 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. Abort exceptions include the double fault (vector 8) and coprocessor segment overrun (vector 9).

2.5.3 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 $\overline{\text{SMI}}$) and exception is assigned one of 256 interrupt vector numbers (Table 2–19). The first 32 interrupt vector assignments are defined or reserved. INT instructions acting as software interrupts may use any of the interrupt vectors, 0 through 255. The non-maskable hardware interrupt (NMI) is assigned vector 2.

In response to a maskable hardware interrupt (INTR), the TI486 issues interrupt acknowledge bus cycles used to read the vector number from external hardware. These vectors should be in the range 32–255 because vectors 0–31 are predefined.

Table 2–19. Interrupt Vector Assignments

INTERRUPT VECTOR	FUNCTION	EXCEPTION TYPE
0	Divide error	FAULT
1	Debug exception	TRAP (see Note)
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	Coprocessor segment overrun	ABORT
10	Invalid TSS	FAULT
11	Segment not present	FAULT
12	Stack fault	FAULT
13	General protection fault	FAULT
14	Page fault	FAULT/TRAP
15	Reserved	—
16	Coprocessor error	FAULT
17	Alignment check exception	FAULT
18–31	Reserved	—
32–255	Maskable hardware interrupts	TRAP
0–255	Programmed interrupt	TRAP

Note: Some debug exceptions may report both traps on the previous instruction and faults on the next instruction.

Interrupt Descriptor Table

The interrupt vector number is used by the TI486 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 mainly to enter a hardware interrupt handler. Trap gates are generally used to enter an exception handler or software interrupt handler. If an interrupt gate is used, the Interrupt Enable Flag (IF) in the EFLAGS register is cleared before the interrupt handler is entered. Task gates are used to make the transition to a new task.

2.5.4 Interrupt and Exception Priorities

As the TI486 executes instructions, it follows a consistent policy for prioritizing exceptions and hardware interrupts as listed in Table 2–20. SMM interrupts always take precedence. Debug traps for the previous instruction and next instruction are handled in the next priority. When NMI and maskable INTR interrupts are both detected at the same instruction boundary, the TI486 microprocessor services the NMI interrupt first.

The TI486 checks for exceptions in parallel with instruction decoding and execution. Several exceptions can result in 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 that way, exceptions can be serviced until the instruction executes properly.

The TI486 supports instruction restart after all faults, except when an instruction causes a task switch to a task whose task state segment (TSS) is partially not present. A TSS can be partially not present if the TSS is not page aligned and one of the pages (where the TSS resides) is not currently in memory.

Table 2–20. Interrupt and Exception Priorities

PRIORITY	DESCRIPTION	NOTES
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	Coprocessor error exception.	Caused by $\overline{\text{ERROR}}$ asserted.
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.	

2.5.5 Exceptions in Real Mode

Many of the exceptions described in Table 2–19 are not applicable in real mode. Exceptions 10, 11, and 14 do not occur in real mode. Other exceptions have slightly different meanings in real mode as listed in Table 2–21.

Table 2–21. Exception Changes in Real Mode

VECTOR NUMBER	PROTECTED MODE FUNCTION	READ 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	—

2.5.6 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 2–26 and the error code bit definitions are listed in Table 2–22. 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 2–26. Error Code Format

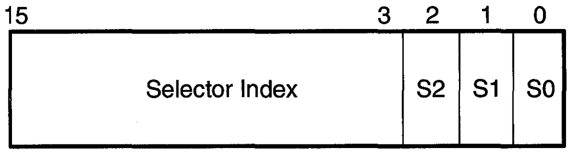


Table 2–22. Error Code Bit Definitions

FAULT TYPE	SELECTOR INDEX (BITS 15–3)	S2 (BIT 2)	S1 (BIT 1)	SO (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 set exception occurred while trying to invoke exception or hardware interrupt handler.
Segment fault	Index of faulty selector	TI bit of faulty selector	0	If set exception occurred while trying to invoke exception or hardware interrupt handler.

2.6 System Management Mode

2.6.1 Introduction

System Management Mode (SMM) provides an additional interrupt which can be used for system power management or software transparent emulation of I/O peripherals. SMM is entered using the Software Management Interrupt (SMI) which has a higher priority than any other interrupt, including NMI. After receiving an SMI, portions of the CPU state are automatically saved, SMM is entered and program execution begins at the base of SMM space (Figure 2–27 and Figure 2–28). Running in protected SMM address space, the interrupt routine does not interfere with the operating system or any application program.

Seven SMM instructions have been added to the TI486 instruction set that permit saving and restoring of the total CPU state when in SMM mode. Two new pins, SMI and SMADS, support SMM functions.

Figure 2–27. TI486SLC/E Memory and I/O Address Spaces

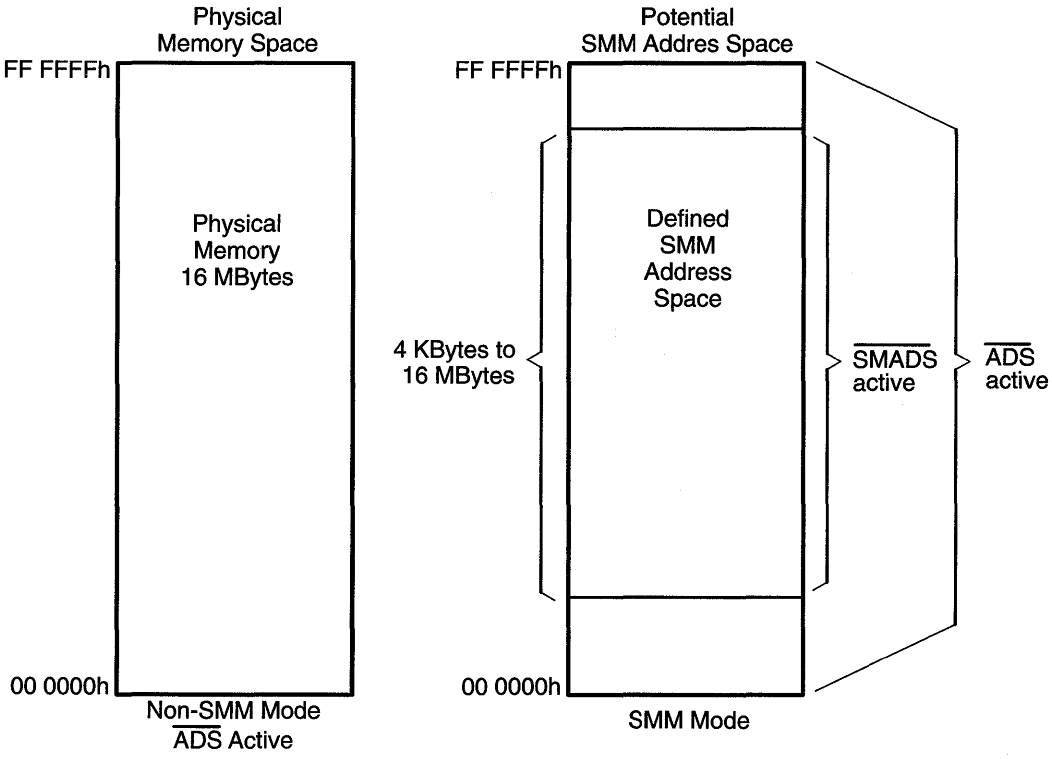
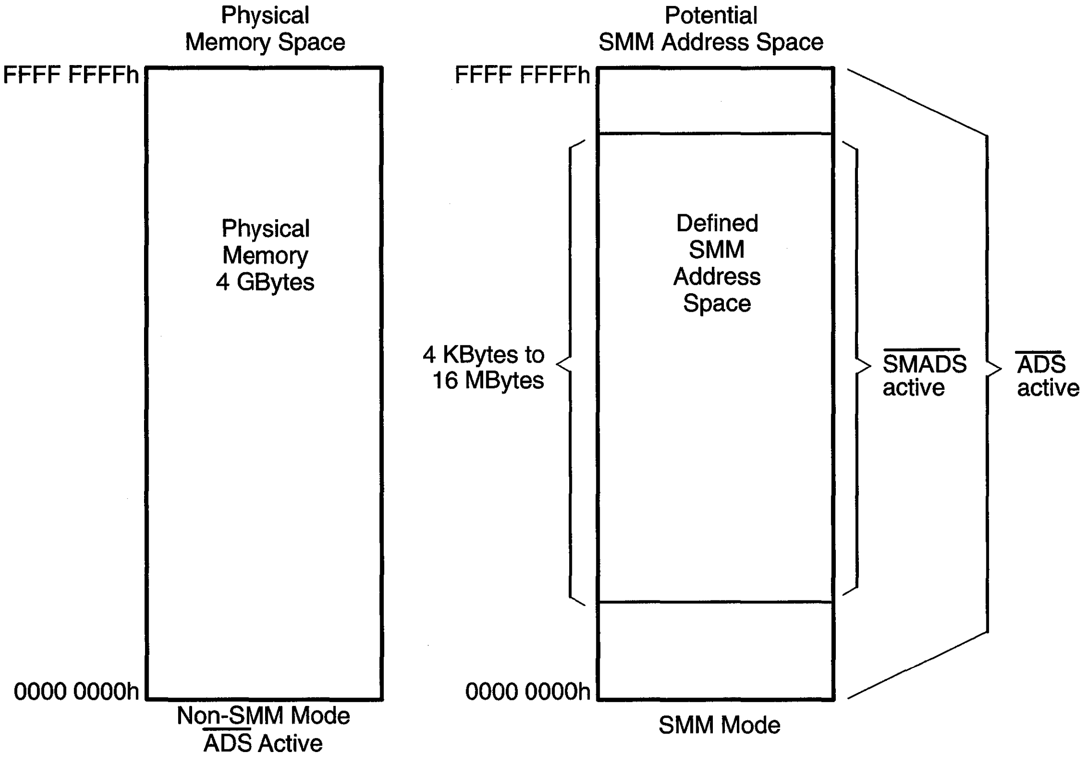


Figure 2–28. TI486DLC/E Memory and I/O Address Spaces


Physical Memory Space
FFFF FFFFh
Physical Memory 4 GBytes
0000 0000h
Non-SMM Mode
$\overline{\text{ADS}}$ Active
Potential SMM Address Space
FFFF FFFFh
Defined SMM Address Space
4 KBytes to 16 MBytes
0000 0000h
SMM Mode
$\overline{\text{SMADS}}$ active
$\overline{\text{ADS}}$ active

2.6.2 SMM Operations

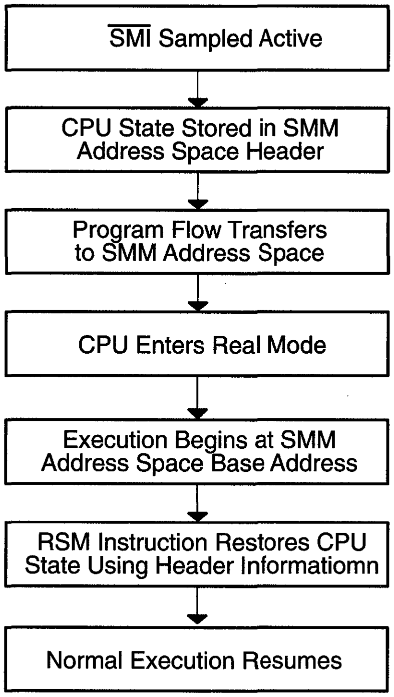
SMM operation is summarized in Figure 2–29. Entering SMM requires the assertion of the $\overline{\text{SMI}}$ pin for at least four CLK2 periods. For the $\overline{\text{SMI}}$ input to be recognized, the following configuration register bits must be set as shown below:

SMI	CCR1(1)	= 1
SMAC	CCR1(2)	= 0
SM4	CCR1(7)	= 1
ARR4	SIZE(3–0)	> 0

The configuration registers are discussed in detail earlier in this chapter. After recognizing $\overline{\text{SMI}}$ 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 the 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 if the SMI service routine modifies more than what is automatically saved, the complete CPU state information must be saved.

Figure 2–29. SMM Execution Flow Diagram


$\overline{\text{SMI}}$ Sampled Active
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 Informatiomn
Normal Execution Resumes

A complete CPU state save is performed by using MOV instructions to save normally accessible information, and by using the SMM instructions to save CPU information that is not normally accessible to the programmer. As will be explained, SMM instructions (SVDC, SVLDT, and SVTS) are used to store the LDTR, TSR and segment registers and their associated descriptor cache entries in 80-bit memory locations. After power up or at the end of the SMI service routine, the MOV and additional SMM instructions (RSDC, RSLDT, and RSTS) are used to restore the CPU state. The SMM RSM instruction returns the CPU to normal execution.

2.6.3 SMM Memory Space Header

With every SMI interrupt, certain CPU state information is automatically saved in the SMM memory space header located at the top of SMM address space (Figure 2–30 and Table 2–23). 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. 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 (Table 2–23) contain the same addresses and the I and P fields contain valid information.

Figure 2–30. SMM Memory Space Header

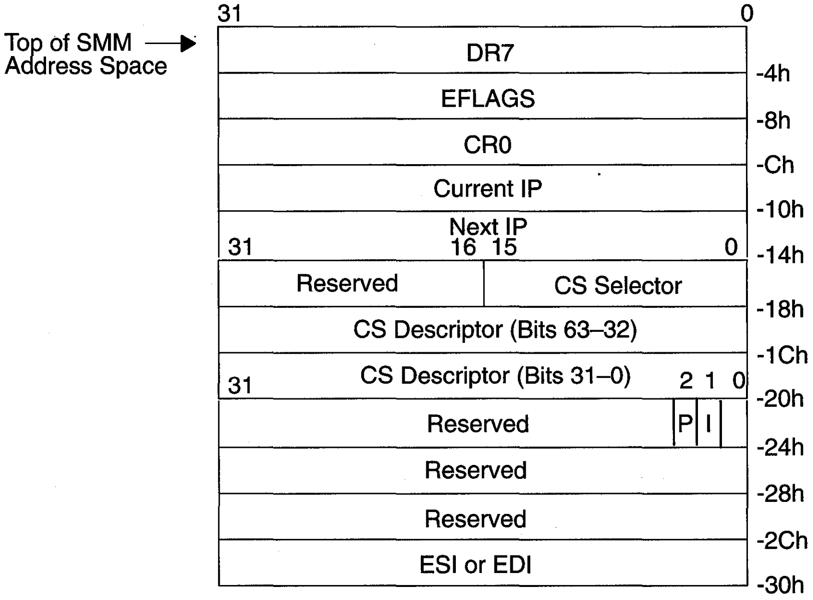


Table 2–23. SMM Memory Space Header

NAME	DESCRIPTION	SIZE
DR7	The contents of the debug register 7.	4 Bytes
EFLAGS	The contents of the extended flag register.	4 Bytes
CR0	The contents of the control register 0.	4 Bytes
Current IP	The address of the instruction executed prior to servicing the SMI interrupt.	4 Bytes
Next IP	The address of the next instruction that will be executed after exiting the SMM mode.	4 Bytes
CS Selector	Code segment register selector for the current code segment.	2 Bytes
CS Descriptor	Code register descriptor for the current code segment.	8 Bytes
P	REP INSx/OUTSx Indicator P = 1 if current instruction has a REP prefix P = 0 if current instruction does not have 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
ESI or EDI	Restored ESI or EDI value. Used when it is necessary to repeat an REP OUTSx or REP INSx instruction when one of the I/O cycles caused an $\overline{\text{SMI}}$ trap	4 Bytes

Note: INSx = INS, INSB, INSW, or INSD instruction.

Note: OUTSx = OUTS, OUTSB, OUTSW, or OUTSD instruction.

2.6.4 SMM Instructions

The TI486 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 either modifies more than what is automatically saved or forces the CPU to power down, the complete CPU state information must be saved. Since the TI486 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 2–24, can be executed only if: (a) the Current Privilege Level (CPL) = 0 and the SMAC bit (CCR1, bit 2) is set; or (b) CPL =0 and the CPU is in an SMI service routine ($\overline{\text{SMI}}$ = 0). If both these conditions are not met and an attempt is made to execute an SVDC, RSDC, SVLDT, RSLDT, SVTS, RSTS, 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. All of the SMM instructions (except RSM) save or restore 80 bits of data, allowing the saved values to include the hidden portion of the register contents.

Table 2–24. 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. *(CS is automatically restored with RSM)*
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* Save 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.
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.

2.6.5 SMM Memory Space

SMM memory space is defined by assigning Address Region 4 to SMM memory space. This assignment is made by setting bit 7 (SM4) in the on-chip CCR1 register. ARR4, also an on-chip configuration register, specifies the base address and size of the SMM memory space. 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 16 MBytes.

SMM memory space accesses can use address pipelining, and are always non-cacheable. SMM accesses ignore the state of the $\overline{\text{A20M}}$ 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 System Management Access (SMAC) bit in the CCR1 register. This feature may be used to initialize the SMM memory space.

While in SMM mode, $\overline{\text{SMADS}}$ address strobes are generated instead of $\overline{\text{ADS}}$ for SMM memory accesses. Any memory accesses outside the defined SMM space result in normal memory accesses and $\overline{\text{ADS}}$ strobes. Data (non-code) accesses to main memory that overlap with defined SMM memory space are allowed if bit 3 in CCR1 (MMAC) is set. In this case, $\overline{\text{ADS}}$ strobes are generated for data accesses only and $\overline{\text{SMADS}}$ strobes continue to be generated for code accesses.

2.6.6 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 and limits defined by the ARR4 register and the SMI service routine begins execution at the SMM base address in real mode.

The programmer must then 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'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 an SMI service routine. If interrupts are to be serviced while operating 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. However, NMIs remain enabled. If it is desired to disable NMI, it should be done immediately after entering the SMI service routine by the system hardware logic.

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

To exit the SMI service routine, a Resume (RSM) instruction, rather than an IRET, is executed. The RSM instruction causes the TI486 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 and the RSDC, RSLDT, and RSTS instructions.

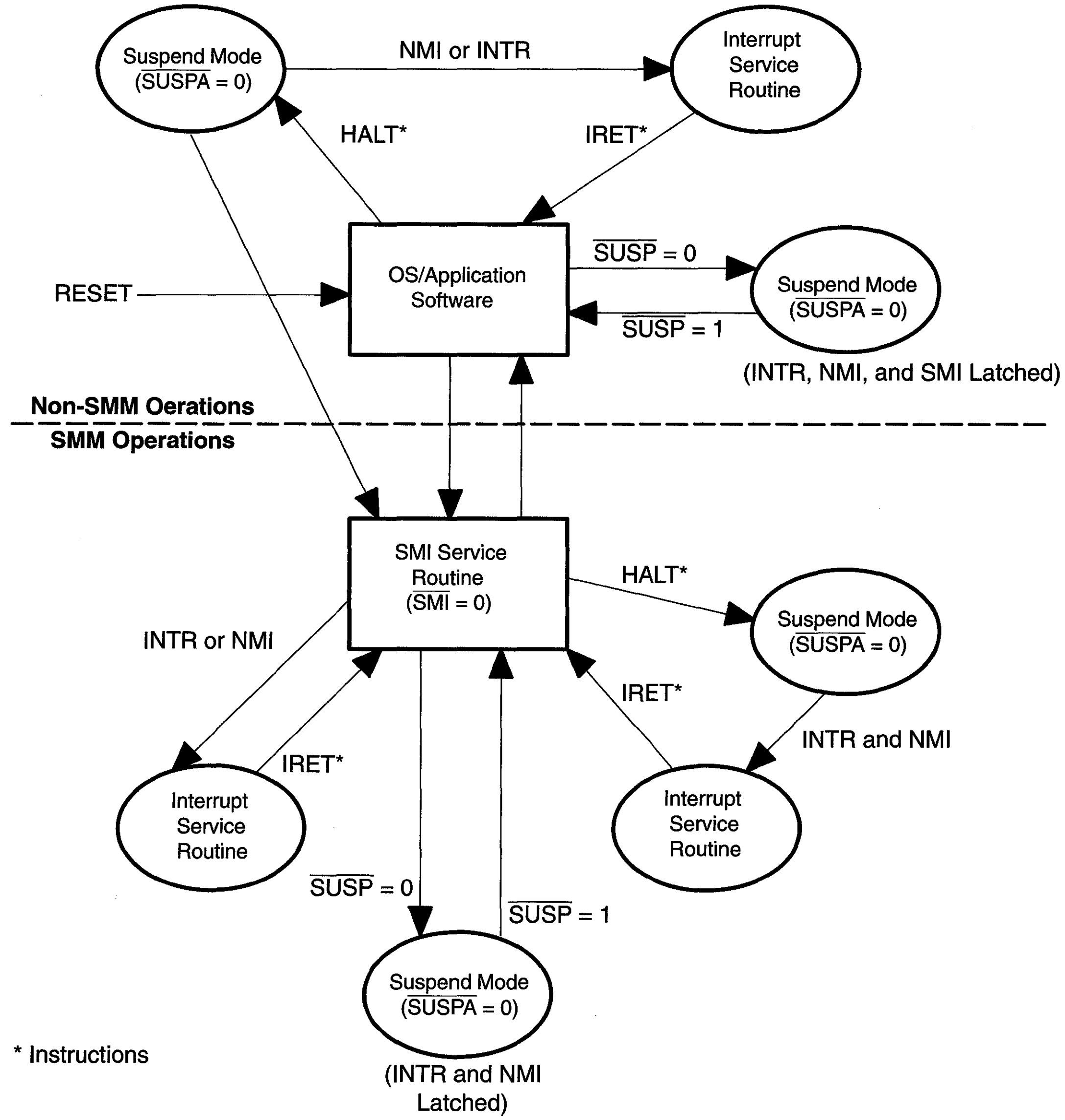
CPU States Related to SMM and Suspend Mode

The state diagram shown in Figure 2–31 illustrates the various CPU states associated with SMM and suspend mode. While in the SMI service routine, the TI486 can enter suspend mode either by (1) executing a HALT instruction or (2) by asserting the $\overline{\text{SUSP}}$ input.

During SMM operation and while in $\overline{\text{SUSP}}$ initiated suspend mode, an occurrence of either NMI or INTR is latched. In order for INTR to be latched, the IF flag must have been set. The INTR or NMI is serviced after exiting suspend mode.

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

Figure 2–31. SMM and Suspended Mode Flow Diagram



* Instructions

2.7 Shutdown and Halt

The halt instruction (HLT) stops program execution and prevents the processor from using the local bus until restarted. The TI486 then enters a low-power suspend mode. INTR with interrupts enabled (IF bit in EFLAGS = 1), SMI, NMI, 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 be exited only by a processor reset.

2.8 Protection

Segment protection and page protection are safeguards built into the TI486 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 privilege instructions, I/O 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

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 be called only by a task executing at the same or a less privileged level.

2.8.1 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 succeeds only if executed with an EPL ≥ 2.

2.8.2 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.

2.8.3 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.

The five types of control transfers are summarized in Table 2–25. 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 2–25. 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	Call gate	GDT or LDT
	Interrupt instruction, Exception, External interrupt	Trap or interrupt gate	IDT
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

2.8.3.1 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).

2.8.4 Initialization and Transition to Protected Mode

The TI486 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.

2.9 Virtual 8086 Mode

Both Real Mode and Virtual 8086 (V86) Mode are supported by the TI486 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 TI486 protection mechanism. V86 tasks run at privilege level 3. Upon entry, all segment limits are set to FFFFh (64K) as in real mode.

2.9.1 Memory Addressing

While in V86 mode, segment registers are used in an identical fashion to Real Mode. The contents of the segment register are shifted left four bits and added to the offset to form the segment base linear address. The TI486 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 TI486 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 TI486 CPU. As in real mode, linear addresses that exceed 1 MByte cause a segment limit overrun exception.

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.

2.9.2 Protection

All V86 tasks operate with the least amount of privilege (level 3) and are subject to all of the TI486 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 is sensitive to the I/O 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.

2.9.3 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.

2.9.4 Entering and Leaving V86 Mode

V86 mode is entered from protected mode either by 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 be exited only 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.

Chapter 3

TI486SLC/E Bus Interface

In this chapter, an overview of the TI486 provides a summary of the processor signals, functional description of all pins, functional timing and bus operations (including non-pipelined and pipelined addressing), various interfaces, and power management.

Topic **Page**

3.1 Overview

The following sections describe the TI486SLC/E input and output signals. The discussion of these signals is arranged by functional groups as shown in Figure 3–1. Table 3–1 gives a brief description of each of the TI486SLC/E signals.

Figure 3–1. TI486SLC/E Functional Signal Groupings

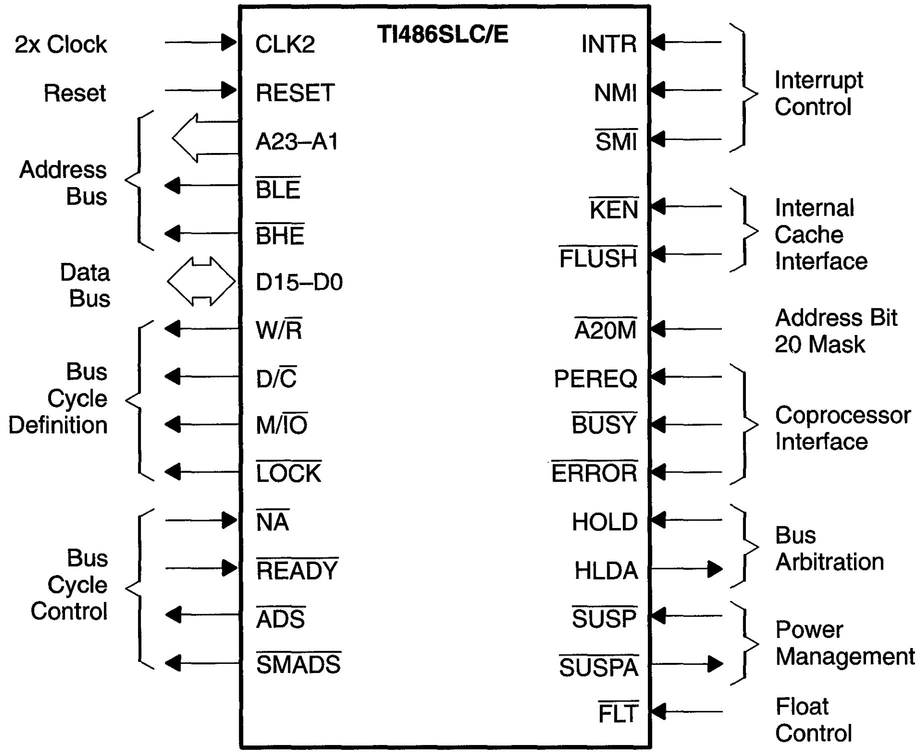


Table 3–1. TI486SLC/E Signal Summary

SIGNAL	SIGNAL NAME	SIGNAL GROUP
$\overline{\text{A20M}}$	Address bit 20 mask	
A23–A1	Address bus lines	Address bus
$\overline{\text{ADS}}$	Address strobe	Bus cycle control
$\overline{\text{BHE}}$	Byte high enable	Address bus
$\overline{\text{BLE}}$	Byte low enable	Address bus
$\overline{\text{BUSY}}$	Processor extension busy	Coprocessor interface
CLK2	2X clock input	
D15–D0	Data bus lines	
D/$\overline{\text{C}}$	Data/control	Bus cycle definition
$\overline{\text{ERROR}}$	Processor extension error	Coprocessor interface
$\overline{\text{FLT}}$	Float	
$\overline{\text{FLUSH}}$	Cache flush	Internal cache interface
HLDA	Hold acknowledge	Bus arbitration
HOLD	Hold request	Bus arbitration
INTR	Maskable interrupt request	Interrupt control
$\overline{\text{KEN}}$	Cache enable	Internal cache interface
$\overline{\text{LOCK}}$	Bus lock	Bus cycle definition
M/$\overline{\text{IO}}$	Memory/input-output	Bus cycle definition
$\overline{\text{NA}}$	Next address request	Bus cycle control
NMI	Non-maskable interrupt request	Interrupt control
PEREQ	Processor extension request	Coprocessor interface
$\overline{\text{READY}}$	Bus ready	Bus cycle control
RESET	Reset	
$\overline{\text{SMADS}}$	SMM address strobe	Bus cycle control
$\overline{\text{SMI}}$	System management interrupt	Interrupt control
$\overline{\text{SUSP}}$	Suspend request	Power management
$\overline{\text{SUSPA}}$	Suspend acknowledge	Power management
W/$\overline{\text{R}}$	Write/read	Bus cycle definition

The following sections describe the signals and their functional timing characteristics. Additional signal information may be found in Chapter 5, Electrical Specifications. Chapter 5 documents the dc and ac characteristics for the signals including voltage levels, propagation delays, setup times, and hold times. Specified setup and hold times must be met for proper operation of the TI486.

Table 3–2. Terminal Functions

PIN		I/O	DESCRIPTION
NAME	NO.		
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 A17 A18 A19 A20 A21 A22 A23	18 51 52 53 54 55 56 58 59 60 61 62 64 65 66 70 72 73 74 75 76 79 80	O/Z	**Address Bus** (active high). The address bus (A23–A1) signals are 3-state outputs that provide addresses for physical memory and I/O ports. All address lines can be used for addressing physical memory allowing a 16 MByte address space (00 0000h to FF FFFFh). During I/O port accesses, A23–A16 are driven low (except for coprocessor accesses). This permits a 64 KByte I/O address space (00 0000h to 00 FFFFh). During all coprocessor I/O access address lines A22–A16 are driven low and A23 is driven high. This allows A23 to be used by external logic to generate a coprocessor select signal. Coprocessor command transfers occur with address 80 00F8h and coprocessor data transfers occur with addresses 80 00FCh and 90 00FEh. A23–A1 float while the CPU is in a hold acknowledge or float state.
$\overline{ADS}$	16	O/Z	**Address Strobe** (active low). This is a 3-state output that indicates the TI486 has driven a valid address (A23–A1, $\overline{BHE}$, $\overline{BLE}$) and bus cycle definition (M/$\overline{IO}$), D/$\overline{C}$, W/$\overline{R}$) on the appropriate TI486SLC/E output pins. During non-pipelined bus cycles, $\overline{ADS}$ is active for the first clock of the bus cycle. During address pipelining, $\overline{ADS}$ is asserted during the previous bus cycle and remains asserted until $\overline{READY}$ is returned for that cycle. $\overline{ADS}$ floats while the TI486SLC/E is in a hold acknowledge or float state.
$\overline{A20M}$	31	I	**Address Bit 20 Mask** (active low). This input causes the TI486SLC/E to mask (force low) physical address bit 20 when driving the external address bus or performing an internal cache access. When the processor is in real mode, asserting $\overline{A20M}$ emulates the 1 MByte address wrap around that occurs on the 8086. The A20 signal is never masked when paging is enabled regardless of the state of the $\overline{A20M}$ input. The $\overline{A20M}$ input is ignored following reset and can be enabled using the A20M bit in the CCR0 configuration register. $\overline{A20M}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{BHE}$ $\overline{BLE}$	19 17	O/Z	**Byte Enables** (active low). Byte Low Enable ($\overline{BLE}$) and Byte High Enable ($\overline{BHE}$) are 3-state outputs that indicate which byte(s) of the 16-bit data bus will be selected for data transfer during the current bus cycle. $\overline{BLE}$ selects the low byte (D7–D0) and $\overline{BHE}$ selects the high byte (D15–D8). When $\overline{BHE}$ and $\overline{BLE}$ are asserted, both bytes (all 16 bits) of the data bus are selected. $\overline{BLE}$ and $\overline{BHE}$ float while the CPU is in a hold acknowledge or float state. $\overline{BHE}$ = $\overline{BLE}$ = 1 never occurs during a bus cycle.

Table 3–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
$\overline{\text{BUSY}}$	34	I	**Coprocessor Busy** (active low). This is an input from the coprocessor that indicates to the TI486SLC/E that the coprocessor is currently executing an instruction and is not yet able to accept another opcode. When the TI486SLC/E processor encounters a WAIT instruction or any coprocessor instruction that operates on the coprocessor stack (i.e., load, pop, arithmetic operation), $\overline{\text{BUSY}}$ is sampled. $\overline{\text{BUSY}}$ is continually sampled and must be recognized as inactive before the CPU will supply the coprocessor with another instruction. However, the following coprocessor instructions are allowed to execute even if $\overline{\text{BUSY}}$ is active since these instructions are used for coprocessor initialization and exception clearing: FNINIT, FNCLEX. $\overline{\text{BUSY}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
CLK2	15	I	**2X Clock Input** (active high). This signal is the basic timing reference for the TI486SLC/E microprocessor. The CLK2 input is internally divided by two to generate the internal processor clock. The external CLK2 is synchronized to a known phase of the internal processor clock by the falling edge of the RESET signal. External timing parameters are defined with respect to the rising edge of CLK2.
D0	1	I/O/Z	**Data Bus** (active high). The Data Bus (D15–D0) signals are 3-state bidirectional signals that provide the data path between the TI486SLC/E and external memory and I/O devices. The data bus inputs data during memory read, I/O read and interrupt acknowledge cycles and outputs data during memory and I/O write cycles. Data read operations require that specified data setup and hold times be met for correct operation. The data bus signals are high active and float while the CPU is in a hold acknowledge or float state.
D1	100		
D2	99		
D3	96		
D4	95		
D5	94		
D6	93		
D7	92		
D8	90		
D9	89		
D10	88		
D11	87		
D12	86		
D13	83		
D14	82		
D15	81		
D/$\overline{\text{C}}$	24	O/Z	**Data/Control.** This signal is low during control cycles and is high during data cycles. Control cycles are issued during functions such as a halt instruction, interrupt servicing and code fetching. Data bus cycles include data access from either memory or I/O.
$\overline{\text{ERROR}}$	36	I	**Coprocessor Error** (active low). This is an input used to indicate that the coprocessor generated an error during execution of a coprocessor instruction. $\overline{\text{ERROR}}$ is sampled by the TI486SLC/E processor whenever a coprocessor instruction is executed. If $\overline{\text{ERROR}}$ is sampled active, the processor generates exception 16 which is then serviced by the exception handling software. Certain coprocessor instructions do not generate an exception 16 even if $\overline{\text{ERROR}}$ is active. These instructions, which involve clearing coprocessor error flags and saving the coprocessor state, are listed as follows: FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV, FNSAVE. $\overline{\text{ERROR}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected. $\overline{\text{ERROR}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\text{FLT}}$	28	I	**Float** (active low). This input forces all bidirectional and output signals to a 3-state condition. Floating the signals allows the TI486SLC/E signals to be externally driven without physically removing the device from the circuit. The TI486SLC/E CPU must be reset following assertion or deassertion of $\overline{\text{FLT}}$. It is recommended that $\overline{\text{FLT}}$ be used only for test purposes. $\overline{\text{FLT}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.

Table 3–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
$\overline{\text{FLUSH}}$	30	I	**Cache Flush** (active low). This is an input that invalidates (flushes) the entire cache. Use of $\overline{\text{FLUSH}}$ to maintain cache coherency is optional. The cache may also be invalidated during each hold acknowledge cycle by setting the BARB bit in the CCR0 configuration register. The $\overline{\text{FLUSH}}$ input is ignored following reset and can be enabled using the FLUSH bit in the CCR0 configuration register. $\overline{\text{FLUSH}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
HOLD	4	I	**Hold Request** (active high). This input is used to indicate that another bus master requests control of the local bus. The bus arbitration (HOLD, HLDA) signals allow the TI486SLC/E to relinquish control of its local bus when requested by another bus master device. Once the processor has relinquished its bus (3-stated), the bus master device can then drive the local bus signals. After recognizing the HOLD request and completing the current bus cycle or sequence of locked bus cycles, the TI486SLC/E responds by floating the local bus and asserting the hold acknowledge (HLDA) output. Once HLDA is asserted, the bus remains granted to the requesting bus master until HOLD becomes inactive. When the TI486SLC/E recognizes HOLD is inactive, it simultaneously drives the local bus and drives HLDA inactive. External pullup resistors may be required on some of the TI486SLC/E 3-state outputs to guarantee that they remain inactive while in a hold acknowledge state. The HOLD input is not recognized while RESET is active. If HOLD is asserted while RESET is active, RESET has priority and the TI486SLC/E places the bus into an idle state instead of a hold acknowledge state. The HOLD input is also recognized during suspend mode provided that the CLK2 input has not been stopped. HOLD is level-sensitive and must meet specified setup and hold times for correct operation.
HLDA	3	O	**Hold Acknowledge** (active high). This output indicates that the TI486SLC/E is in a hold acknowledge state and has relinquished control of its local bus. While in the hold acknowledge state, the TI486SLC/E drives HLDA active and continues to drive $\overline{\text{SUSPA}}$, if enabled. The other TI486SLC/E outputs are in a high-impedance state allowing the requesting bus master to drive these signals. If the on-chip cache can satisfy bus requests, the TI486SLC/E continues to operate during hold acknowledge states. $\overline{\text{A20M}}$ is internally recognized during this time. The processor deactivates HLDA when the HOLD request is driven inactive. The TI486SLC/E stores an NMI rising edge during a hold acknowledge state for processing after HOLD is inactive. The $\overline{\text{FLUSH}}$ input is also recognized during a hold acknowledge state. If $\overline{\text{SUSP}}$ is asserted during a hold acknowledge state, the TI486SLC/E may or may not enter suspend mode depending on the state of the internal execution pipeline. Table 3–3 summarizes the state of the TI486SLC/E signals during hold acknowledge.
INTR	40	I	**Maskable Interrupt Request**. This is a level-sensitive input that causes the processor to suspend execution of the current instruction stream and begin execution of an interrupt service routine. The INTR input can be masked (ignored) through the Flags Register IF bit. When unmasked, the TI486SLC/E responds to the INTR input by issuing two locked interrupt acknowledge cycles. To assure recognition of the INTR request, INTR must remain active until the start of the first interrupt acknowledge cycle.

Table 3–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
$\overline{\text{KEN}}$	29	I	**Cache Enable** (active low). This is an input which indicates that the data being returned during the current cycle is cacheable. When $\overline{\text{KEN}}$ is active and the TI486SLC/E is performing a cacheable code fetch or memory data read cycle, the cycle is transformed into a cache fill. Use of the $\overline{\text{KEN}}$ input to control cacheability is optional. The non-cacheable region registers can also be used to control cacheablity. Memory addresses specified by the non-cacheable region registers are not cacheable regardless of the state of $\overline{\text{KEN}}$. I/O accesses, locked reads, SMM address space accesses, and interrupt acknowledge cycles are never cached. During cached code fetches, two contiguous read cycles are performed to completely fill the 4-byte cache line. $\overline{\text{KEN}}$ must be asserted during both read cycles in order to cause a cache line fill. During cached data reads, the TI486SLC/E performs only those bus cycles necessary to supply the required data to complete the current operation. Valid bits are maintained for each byte in the cache line, thus allowing data operands of less than 4 bytes to reside in the cache. During any cache fill cycle with $\overline{\text{KEN}}$ asserted, the TI486SLC/E ignores the state of the byte enables ($\overline{\text{BHE}}$ and $\overline{\text{BLE}}$) and always writes two bytes of data into the cache. The $\overline{\text{KEN}}$ input is ignored following reset and can be enabled using the KEN bit in the CCR0 configuration register. $\overline{\text{KEN}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\text{LOCK}}$	26	I	**LOCK** (active low). $\overline{\text{LOCK}}$ is asserted to deny control of the CPU bus to other bus masters. The $\overline{\text{LOCK}}$ signal may be explicitly activated during bus operations by including the LOCK prefix on certain instructions. $\overline{\text{LOCK}}$ is always asserted during descriptor and page table updates, interrupt acknowledge sequences and when executing the XCHG instruction. The TI486SLC/E does not enter the hold acknowledge state in response to HOLD while the $\overline{\text{LOCK}}$ input is active.
M/$\overline{\text{IO}}$	23	O/Z	**Memory/IO**. This signal is low during I/O read and write cycles and is high during memory cycles.
$\overline{\text{NA}}$	6	I	**Next Address Request** (active low). This is an input used to request address pipelining by the system hardware. When asserted, the system indicates that it is prepared to accept new bus cycle definition and address signals (M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$, W/$\overline{\text{R}}$, A23–A1, $\overline{\text{BHE}}$, and $\overline{\text{BLE}}$) from the microprocessor even if the current bus cycle has not been terminated by assertion of $\overline{\text{READY}}$. If the TI486SLC/E has an internal bus request pending and the $\overline{\text{NA}}$ input is sampled active, the next bus cycle definition and address signals are driven onto the bus.
NC	27, 45, 46	—	No connection. Should be left disconnected.
NMI	38	I	**Non-maskable Interrupt Request**. This is a rising-edge-sensitive input that causes the processor to suspend execution of the current instruction stream and begin execution of an NMI interrupt service routine. The NMI interrupt service request cannot be masked by software. Asserting NMI causes an interrupt which internally supplies interrupt vector 2h to the CPU core. External interrupt acknowledge cycles are not necessary since the NMI interrupt vector is supplied internally. The TI486SLC/E samples NMI at the beginning of each phase 2. To assure recognition, NMI must be inactive for at least eight CLK2 periods and then be active for at least eight CLK2 periods. Additionally, specified setup and hold times must be met to guarantee recognition at a particular clock edge.

Table 3–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
PEREQ	37	I	**Coprocessor Request** (active high). This is an input that indicates the coprocessor is ready to transfer data to or from the CPU. The coprocessor may assert PEREQ in the process of executing a coprocessor instruction. The TI486SLC/E internally stores the current coprocessor opcode and performs the correct data transfers to support coprocessor operations using PEREQ to synchronize the transfer of required operands. PEREQ is internally connected to a pulldown resistor to prevent this signal from floating active when left unconnected.
$\overline{\text{READY}}$	7	I	**Ready**. This is an input generated by the system hardware that indicates the current bus cycle can be terminated. During a read cycle, assertion of $\overline{\text{READY}}$ indicates that the system hardware has presented valid data to the CPU. When $\overline{\text{READY}}$ is sampled active, the TI486SLC/E latches the input data and terminates the cycle. During a write cycle, $\overline{\text{READY}}$ assertion indicates that the system hardware has accepted the TI486SLC/E output data. $\overline{\text{READY}}$ must be asserted to terminate every bus cycle, including halt and shutdown indication cycles.
RESET	33	I	**Reset** (active high). When asserted, RESET suspends all operations in progress and places the TI486SLC/E into a reset state. RESET is a level-sensitive synchronous input and must meet specified setup and hold times to be properly recognized by the TI486SLC/E. The TI486SLC/E begins executing instructions at physical address location FF FFF0h approximately 400 CLK2s after RESET is driven inactive (low). While RESET is active all other input pins, except $\overline{\text{FLT}}$, are ignored. The remaining signals are initialized to their reset state during the internal processor reset sequence. The reset signal states for the TI486SLC/E are shown in Table 3–3.
$\overline{\text{SMADS}}$	20	O/Z	**SMM Address Strobe** (active low). $\overline{\text{SMADS}}$ is asserted instead of the $\overline{\text{ADS}}$ during SMM bus cycles and indicates that SMM memory is being accessed. $\overline{\text{SMADS}}$ floats while the CPU is in a hold acknowledge or float state. The $\overline{\text{SMADS}}$ output is disabled (floated) following reset and can be enabled using the SMI bit in the CCR1 configuration register.
$\overline{\text{SMI}}$	47	I/O	**System Management Interrupt** (active low). This is a bidirectional signal and level sensitive interrupt with higher priority than the NMI interrupt. $\overline{\text{SMI}}$ must be active for at least four CLK2 clock periods to be recognized by the TI486SLC/E. After the $\overline{\text{SMI}}$ interrupt is acknowledged, the $\overline{\text{SMI}}$ pin is driven low by the TI486SLC/E for the duration of the SMI service routine. The $\overline{\text{SMI}}$ input is ignored following reset and can be enabled using the SMI bit in the CCR1 configuration register. $\overline{\text{SMI}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\text{SUSP}}$	43	I	**Suspend Request** (active low). This is an input that requests the TI486SLC/E enter suspend mode. After recognizing $\overline{\text{SUSP}}$ active, the processor completes execution of the current instruction, any pending decoded instructions and associated bus cycles. In addition, the TI486SLC/E waits for the coprocessor to indicate a not busy status ($\overline{\text{BUSY}}$ = 1) before entering suspend mode and asserting suspend acknowledge ($\overline{\text{SUSPA}}$). $\overline{\text{SUSP}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\text{SUSPA}}$	44	O	**Suspend Acknowledge** (active low). This output indicates that the TI486SLC/E has entered the suspend mode as a result of $\overline{\text{SUSP}}$ assertion or execution of a HALT instruction.

Table 3–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
V_{CC}	8 9 10 21 32 39 42 48 57 69 71 84 91 97	I	**5-V Power Supply.** All pins must be connected and used.
V_{SS}	2 5 11 12 13 14 22 35 41 49 50 63 67 68 77 78 85 98	I	**Ground Pins.** All pins must be connected and used.
W/$\overline{R}$	25	O/Z	**Write/Read.** W/$\overline{R}$ is low during read cycles (data is read from memory or I/O) and is high during write bus cycles (data is written to memory or I/O).

Table 3–3. Signal States During RESET and Hold Acknowledge

SIGNAL NAME	SIGNAL STATE DURING RESET	SIGNAL STATE DURING HOLD ACKNOWLEDGE
$\overline{\text{A20M}}$	Ignored	Input recognized
A23–A1	1	Float
$\overline{\text{ADS}}$	1	Float
$\overline{\text{BHE}}$, $\overline{\text{BLE}}$	0	Float
$\overline{\text{BUSY}}$	Initiates self test	Ignored
D15–D0	Float	Float
D/$\overline{\text{C}}$	1	Float
$\overline{\text{ERROR}}$	Ignored	Ignored
$\overline{\text{FLT}}$	Input recognized	Input recognized
$\overline{\text{FLUSH}}$	Ignored	Input recognized
HLDA	0	1
HOLD	Ignored	Input recognized
INTR	Ignored	Input recognized
$\overline{\text{KEN}}$	Ignored	Ignored
$\overline{\text{LOCK}}$	1	Float
M/$\overline{\text{IO}}$	0	Float
$\overline{\text{NA}}$	Ignored	Ignored
NMI	Ignored	Input recognized
PEREQ	Ignored	Ignored
$\overline{\text{READY}}$	Ignored	Ignored
RESET	Input recognized	Input recognized
$\overline{\text{SMADS}}$	Float	Float
$\overline{\text{SMI}}$	Ignored	Input recognized
$\overline{\text{SUSP}}$	Ignored	Input recognized
$\overline{\text{SUSPA}}$	Float	Driven
W/$\overline{\text{R}}$	0	Float

3.1.1 Bus Cycle Definition

The bus cycle definition (M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$, W/$\overline{\text{R}}$, $\overline{\text{LOCK}}$) signals consist of four 3-state outputs that define the type of bus cycle operation being performed. Table 3–4 defines the bus cycles for the possible states of these signals. M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$ and W/$\overline{\text{R}}$ are the primary bus cycle definition signals and are driven valid as $\overline{\text{ADS}}$ (Address Strobe) becomes active. During non-pipelined cycles, the $\overline{\text{LOCK}}$ output is driven valid along with M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$ and W/$\overline{\text{R}}$. During pipelined addressing, $\overline{\text{LOCK}}$ is driven at the beginning of the bus cycle, which is after $\overline{\text{ADS}}$ becomes active for that cycle. The bus cycle definition signals are active low and float while the TI486SLC/E is in a hold acknowledge or float state.

Table 3–4. Bus Cycle Types

M/$\overline{\text{IO}}$	D/$\overline{\text{C}}$	W/$\overline{\text{R}}$	$\overline{\text{LOCK}}$	BUS CYCLE TYPE
0	0	0	0	Interrupt acknowledge
0	0	0	1	—
0	0	1	X	—
0	1	X	0	—
0	1	0	1	I/O data read
0	1	1	1	I/O data write
1	0	X	0	—
1	0	0	1	Memory code read
1	0	1	1	Halt: A23–A1=2h, $\overline{\text{BHE}}$=1 and $\overline{\text{BLE}}$=0 Shutdown: A23–A1=0h, $\overline{\text{BHE}}$=1 and $\overline{\text{BLE}}$=0
1	1	0	0	Locked memory data read
1	1	0	1	Memory data read
1	1	1	0	Locked memory data write
1	1	1	1	Memory data write

X = don't care
— = does not occur

3.1.2 Power Management

The power management signals allow the TI486SLC/E to enter suspend mode. Suspend mode circuitry allows the TI486SLC/E to consume minimal power while maintaining the entire internal CPU state.

3.1.2.1 *Suspend Request ($\overline{\text{SUSP}}$)*

Suspend Request ($\overline{\text{SUSP}}$) is an active-low input that requests the TI486SLC/E to enter suspend mode. After recognizing $\overline{\text{SUSP}}$ is active, the processor completes execution of the current instruction, any pending decoded instructions and associated bus cycles. In addition, the TI486SLC/E waits for the coprocessor to indicate a not busy condition ($\overline{\text{BUSY}}$=1) before entering suspend mode and asserting suspend acknowledge ($\overline{\text{SUSPA}}$). During suspend mode, internal clocks are stopped and only the logic associated with monitoring RESET, HOLD and $\overline{\text{FLUSH}}$ remains active. With $\overline{\text{SUSPA}}$ asserted, the CLK2 input to the TI486SLC/E can be stopped in either phase. Stopping the CLK2 input further reduces current consumption of the TI486SLC/E.

To resume operation, the CLK2 input is restarted (if stopped), followed by deassertion of the $\overline{\text{SUSP}}$ input. The processor then resumes instruction fetching and begins execution in the instruction stream at the point it had stopped. The $\overline{\text{SUSP}}$ input is level sensitive and must meet specified setup and hold times to be recognized at a particular clock edge. The $\overline{\text{SUSP}}$ input is ignored following reset and can be enabled using the SUSP bit in the CCR0 configuration register.

3.1.2.2 *Suspend Acknowledge ($\overline{\text{SUSPA}}$)*

The Suspend Acknowledge ($\overline{\text{SUSPA}}$) output indicates that the TI486SLC/E has entered the suspend mode as a result of $\overline{\text{SUSP}}$ assertion or execution of a HALT instruction. If $\overline{\text{SUSPA}}$ is asserted and the CLK2 input is switching, the TI486SLC/E continues to recognize $\overline{\text{FLT}}$, RESET, HOLD, and $\overline{\text{FLUSH}}$. If suspend mode was entered as the result of a HALT instruction, the TI486SLC/E also continues to monitor the NMI input and an unmasked INTR input. Detection of INTR or NMI forces the TI486SLC/E to exit suspend mode and begin execution of the appropriate interrupt service routine. The CLK2 input to the processor may be stopped after $\overline{\text{SUSPA}}$ has been asserted to further reduce the power consumption of the TI486SLC/E. The $\overline{\text{SUSPA}}$ output is disabled (floated) following reset and can be enabled using the SUSP bit in the CCR0 configuration register.

Table 3–5 shows the state of the TI486SLC/E signals when the device is in suspend mode.

Table 3–5. Signal States During Suspend Mode

SIGNAL NAME	SIGNAL STATE DURING HOLD ACKNOWLEDGE	SIGNAL STATE DURING HALT INITIATED SUSPEND MODE
$\overline{\mathrm{A20M}}$	Ignored	Ignored
A23–A1	1	1
$\overline{\mathrm{ADS}}$	1	1
$\overline{\mathrm{BHE}}$, $\overline{\mathrm{BLE}}$	0	0
$\overline{\mathrm{BUSY}}$	Ignored	Ignored
D15–D0	Float	Float
D/$\overline{\mathrm{C}}$	1	1
$\overline{\mathrm{ERROR}}$	Ignored	Ignored
$\overline{\mathrm{FLT}}$	Input recognized	Input recognized
$\overline{\mathrm{FLUSH}}$	Input recognized	Input recognized
HLDA	0	0
HOLD	Input recognized	Input recognized
INTR	Latched	Input recognized
$\overline{\mathrm{KEN}}$	Ignored	Ignored
$\overline{\mathrm{LOCK}}$	1	1
M/$\overline{\mathrm{IO}}$	0	0
$\overline{\mathrm{NA}}$	Ignored	Ignored
NMI	Latched	Input recognized
PEREQ	Ignored	Ignored
$\overline{\mathrm{READY}}$	Ignored	Ignored
RESET	Input recognized	Input recognized
$\overline{\mathrm{SMADS}}$	1	1
$\overline{\mathrm{SMI}}$	Latched	Input recognized
$\overline{\mathrm{SUSP}}$	Input recognized	Ignored
$\overline{\mathrm{SUSPA}}$	0	0
W/$\overline{\mathrm{R}}$	0	0

3.1.2.3 Coprocessor Interface

The data bus, address bus, and bus cycle definition signals, as well as the coprocessor interface signals (PREQ, $\overline{\mathrm{BUSY}}$, $\overline{\mathrm{ERROR}}$), are used to control communication between the TI486SLC/E and a coprocessor. Coprocessor or ESC opcodes are decoded by the TI486SLC/E and the opcode and operands are then transferred to the coprocessor via I/O port accesses to addresses 80 00F8h, 80 00FCh, or 80 00FEh. Address 80 00F8h functions as the control port address and 80 00 FCh and 80 00FEh are used for operand transfers.

3.2 Functional Timing

3.2.1 Reset Timing and Internal Clock Synchronization

RESET is the highest priority input signal and is capable of interrupting any processor activity when it is asserted. When RESET is asserted, the TI486SLC/E aborts any bus cycle. Idle, hold acknowledge, and suspend states are also discontinued and the reset state is established. RESET is used when the TI486SLC/E microprocessor is powered up to initialize the CPU to a known valid state and to synchronize the internal CPU clock with external clocks.

RESET must be asserted for at least 15 CLK2 periods to ensure recognition by the TI486SLC/E microprocessor. If the self-test feature is to be invoked, RESET must be asserted for at least 80 CLK2 periods. RESET pulses less than 15 CLK2 periods may not have sufficient time to propagate throughout the TI486SLC/E and may not be recognized. RESET pulses less than 80 CLK2 periods followed by a self-test request may incorrectly report a self-test failure when no true failure exists.

Provided the RESET falling edge meets specified setup and hold times, the internal processor clock phase is synchronized as illustrated in Figure 3–2. The internal processor clock is half the frequency of the CLK2 input and each CLK2 cycle corresponds to an internal CPU clock phase. Phase 2 of the internal clock is defined to be the second rising edge of CLK2 following the falling edge of RESET.

Following the falling edge of REST (and after self-test if it was requested), the TI486SLC/E microprocessor performs an internal initialization sequence for approximately 400 CLK2 periods. The TI486SLC/E self-test feature is invoked if the $\overline{\text{BUSY}}$ input is in an active-low state when RESET falls inactive. The self-test sequence requires approximately (2^{20} + 60) CLK2 periods to complete. Even if the self-test indicates a problem, the TI486SLC/E microprocessor attempts to proceed with the reset sequence. Figure 3–3 illustrates the bus activity and timing during the TI486SLC/E reset sequence.

Upon completion of self-test, the EAX register contains 0000 0000h if the TI486SLC/E microprocessor passed its internal self-test with no problems detected. Any non-zero value in the EAX register indicates that the microprocessor is faulty.

Figure 3–2. Internal Processor Clock Synchronization

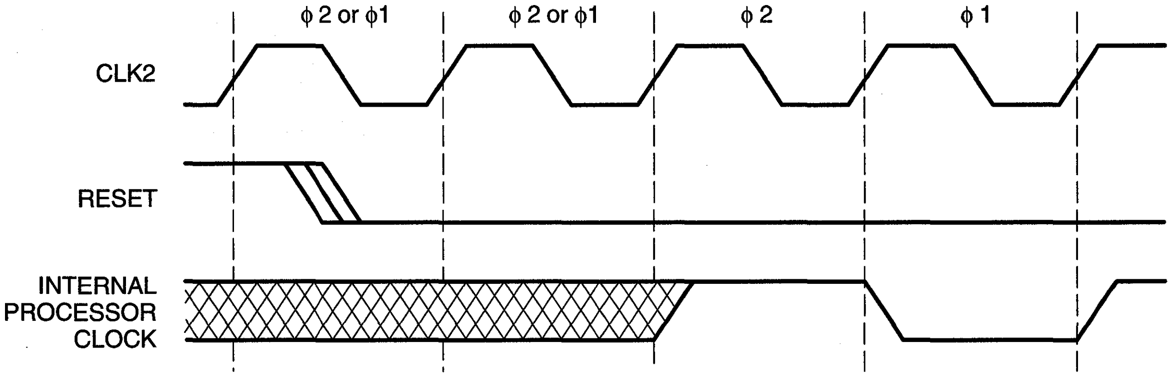
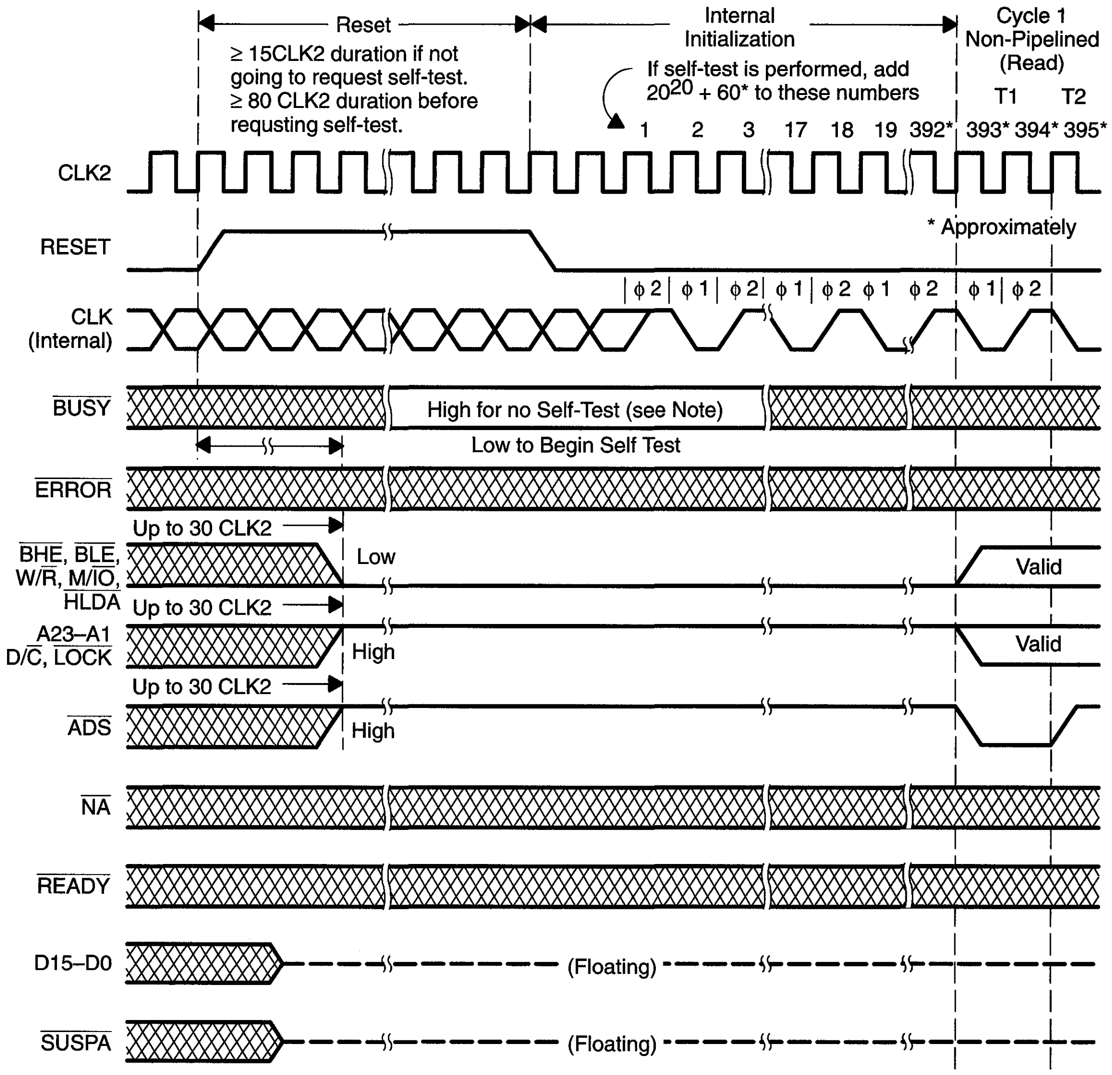


Figure 3–3. Bus Activity from RESET until First Code Fetch


Reset
≥ 15CLK2 duration if not going to request self-test.
≥ 80 CLK2 duration before requsting self-test.
Internal Initialization
If self-test is performed, add 2^{20} + 60* to these numbers
1 2 3 17 18 19 392*
Cycle 1 Non-Pipelined (Read)
T1 T2
393* 394* 395*
CLK2
* Approximately
RESET
CLK (Internal)
φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2
BUSY
High for no Self-Test (see Note)
Low to Begin Self Test
ERROR
Up to 30 CLK2
BHE, BLE, W/R, M/IO, HLDA
Low
Valid
Up to 30 CLK2
A23–A1 D/C, LOCK
High
Valid
Up to 30 CLK2
ADS
High
NA
READY
D15–D0
(Floating)
SUSPA
(Floating)

Note: BUSY should be held stable for 80 CLK2 periods before and after the CLK2 period in which RESET falling edge occurs.

3.2.2 Bus Operation

The TI486SLC/E microprocessor communicates with the external system through separate, parallel buses for data and address. This is commonly called a demultiplexed address/data bus. This demultiplexed bus eliminates the need for address latches required in multiplexed address/data bus configurations where the address and data are presented on the same pins at different times.

TI486SLC/E instructions can act on memory data operands consisting of 8-bit bytes, 16-bit words or 32-bit double words. The TI486SLC/E bus architecture allows for bus transfers of these operands without restrictions on physical address alignment. Any byte boundary may require more than one bus cycle to transfer the operand. This feature is transparent to the programmer.

The TI486SLC/E data bus (D15–D0) is a 16-bit-wide bidirectional bus. The TI486SLC/E drives the data bus during write bus cycles, and the external system hardware drives the data bus during read bus cycles. The address bus provides a 24-bit value using 23 signals for the 23 upper-order address bits (A23–A1), defining which 16-bit word is being accessed, and two byte enable signals ($\overline{\text{BHE}}$ and $\overline{\text{BLE}}$) to directly indicate which of the two bytes within the word are active.

Every bus cycle begins with the assertion of the address strobe ($\overline{\text{ADS}}$). $\overline{\text{ADS}}$ indicates that the TI486SLC/E has issued a new address and new bus cycle definition signals. A bus cycle is defined by four signals: M/$\overline{\text{IO}}$, W/$\overline{\text{R}}$, D/$\overline{\text{C}}$ and $\overline{\text{LOCK}}$. M/$\overline{\text{IO}}$ defines if a memory or I/O operation is occurring, W/$\overline{\text{R}}$ defines the cycle to be read or write, and D/$\overline{\text{C}}$ indicates whether a data or control cycle is in effect. $\overline{\text{LOCK}}$ indicates that the current cycle is a locked bus cycle. Every bus cycle completes when the system hardware returns $\overline{\text{READY}}$ asserted.

The TI486SLC/E performs the following bus cycle types:

- Memory read
- Locked memory read
- Memory write
- Locked memory write
- I/O read (or coprocessor read)
- I/O write (or coprocessor write)
- Interrupt acknowledge (always locked)
- Halt/shutdown

When the TI486SLC/E microprocessor has no pending bus requests, the bus enters the idle state. There is no encoding of the idle state on the bus cycle definition signals; however, the idle state can be identified by the absence of further assertions of $\overline{\text{ADS}}$ following a completed bus cycle.

3.2.2.1 Bus Cycles Using Non-Pipelined Addressing

Non-Pipelined Bus States

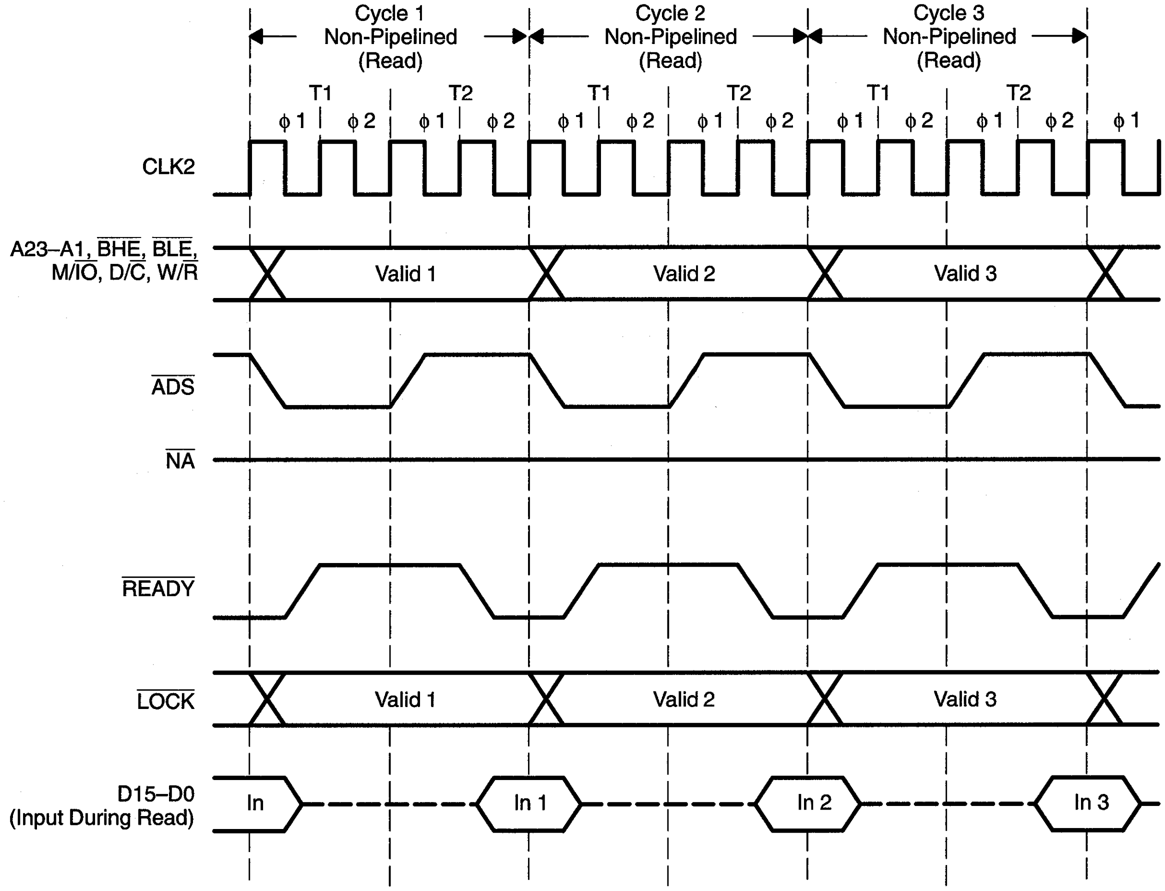
The shortest time unit of bus activity is a bus state, commonly called a T state. A bus state is one internal processor clock period (two CLK2 periods) in duration. A complete data transfer occurs during a bus cycle, composed of two or more bus states.

The first state of a non-pipelined bus cycle is called T1. During phase one (first CLK2) of T1, the address bus and bus cycle definition signals are driven valid and, to signal their availability, address strobe ($\overline{\text{ADS}}$) is simultaneously asserted.

The second bus state of a non-pipelined cycle is called T2. T2 terminates a bus cycle with the assertion of the $\overline{\text{READY}}$ input and valid data is either input or output depending on the bus cycle type. The fastest TI486SLC/E microprocessor bus cycle requires only these two bus states. $\overline{\text{READY}}$ is ignored at the end of the T1 state.

Three consecutive bus read cycles, each consisting of two bus states, are shown in Figure 3–4.

Figure 3–4. Fastest Non-Pipelined Read Cycles



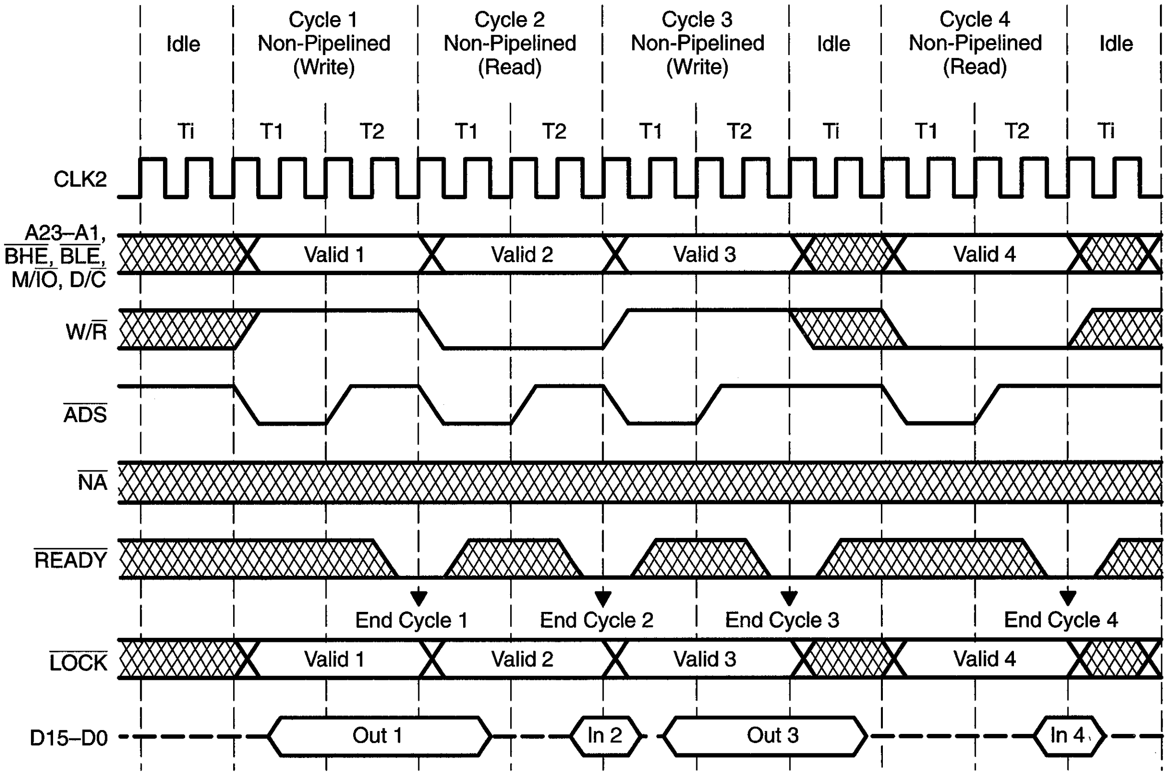
Note: Fastest non-pipelined bus cycles consist of T1 and T2.

Non-Pipelined Read and Write Cycles

Any bus cycle may be performed with non-pipelined address timing. Figure 3–5 shows a mixture of read and write cycles with non-pipelined address timing. When a read cycle is performed, the TI486SLC/E microprocessor floats its data bus and the externally addressed device then drives the data. The TI486SLC/E microprocessor requires that all data bus pins be driven to a valid logic state (high or low) at the end of each read cycle, when $\overline{\text{READY}}$ is asserted. When a read cycle is acknowledged by $\overline{\text{READY}}$ asserted in the T2 bus state, the TI486SLC/E CPU latches the information present at its data pins and terminates the cycle.

When a write cycle is performed, the data bus is driven by the TI486SLC/E CPU beginning in phase two of T1. When a write cycle is acknowledged, the TI486SLC/E write data remains valid throughout phase one of the next bus state to provide write data hold time.

Figure 3–5. Various Non-Pipelined Bus Cycles (No Wait States)



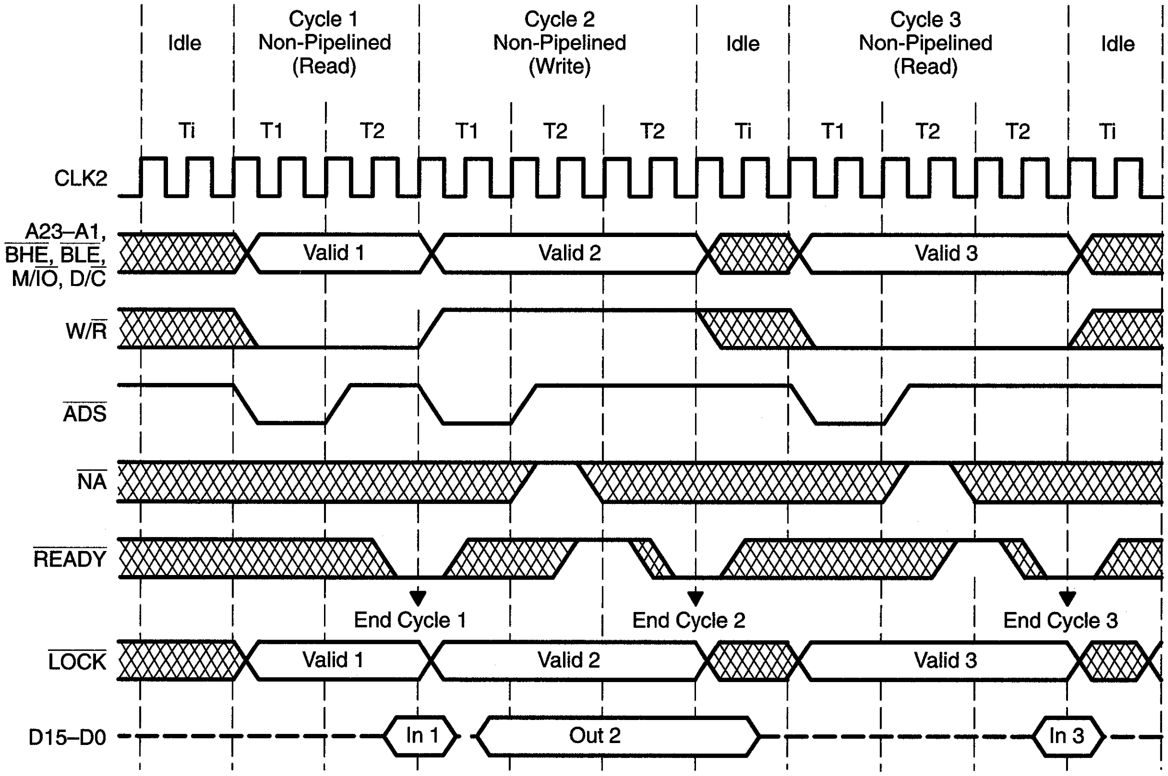
Note: Idle states are shown here for diagram variety only.

Non-Pipelined Wait States

Once a bus cycle begins, it continues until acknowledged by the external system hardware using the TI486SLC/E $\overline{\text{READY}}$ input. Acknowledging the bus cycle at the end of the first T2 results in the shortest possible bus cycle, requiring only T1 and T2. If $\overline{\text{READY}}$ is not immediately asserted however, T2 states are repeated indefinitely until the $\overline{\text{READY}}$ input is sampled active. These intermediate T2 states are referred to as wait states. If the external system hardware is not able to receive or deliver data in two bus states, it withholds the $\overline{\text{READY}}$ signal and at least one wait state is added to the bus cycle. Thus, on an address-by-address basis the system is able to define how fast a bus cycle completes.

Figure 3–6 illustrates non-pipelined bus cycles with one wait state added to cycles 2 and 3. $\overline{\text{READY}}$ is sampled inactive at the end of the first T2 state in cycles 2 and 3. Therefore, the T2 state is repeated until $\overline{\text{READY}}$ is sampled active at the end of the second T2 and the cycle is then terminated. The TI486SLC/E ignores the $\overline{\text{READY}}$ input at the end of the T1 state.

Figure 3–6. Various Non-Pipelined Bus Cycles with Different Numbers of Wait States

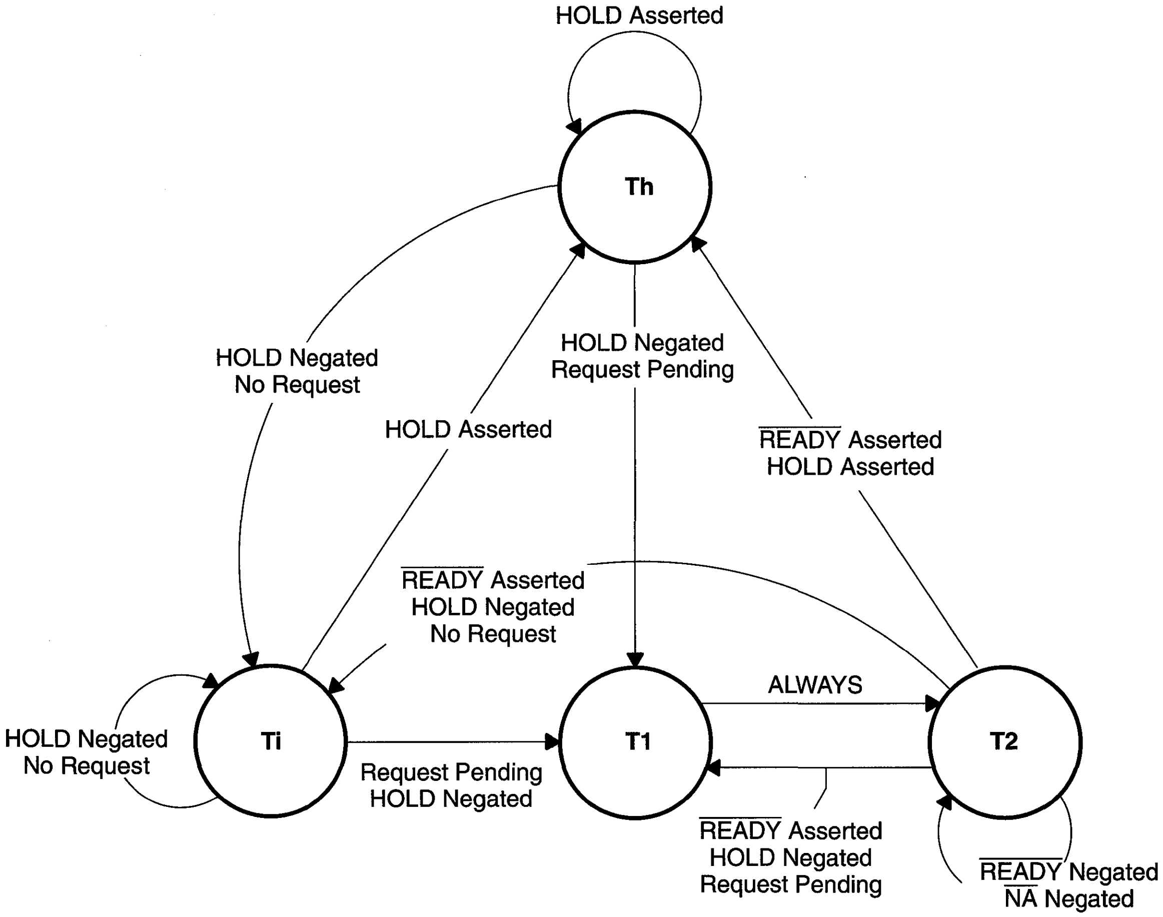


Note: Idle states are shown here for diagram variety only.

Initiating and Maintaining Non-Pipelined Cycles

The bus states and transitions for non-pipelined addressing are illustrated in Figure 3–7. The bus transitions between four possible states: T1, T2, Ti, and Th. Active bus cycles consist of T1 and T2 states, with T2 being repeated for wait states. Bus cycles always begin with a single T1 state. T1 is always followed by a T2 state. If a bus cycle is not acknowledged during a given T2 and $\overline{\text{NA}}$ is inactive, T2 is repeated resulting in a wait state. When a cycle is acknowledged during T2, the following state is T1 of the next bus cycle if a bus request is pending internally. If no internal bus request is pending, the Ti state is entered. If the HOLD input is asserted and the TI486SLC/E is ready to enter the hold acknowledge state, the Th state is entered.

Figure 3–7. Non-Pipelined Bus States



Bus States:

T1 – First clock of a non-pipelined bus cycle (CPU drives new address and asserts $\overline{\text{ADS}}$)
T2 – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has not been sampled asserted in the current bus cycle.
Ti – Idle State
Th – Hold Acknowledge (CPU asserts HLDA)

The fastest bus cycle consists of two states: T1 and T2.

Because of the demultiplexed nature of the bus, the address pipelining option provides a mechanism for the external hardware to have an additional T state of access time without inserting a wait state. After the reset sequence and following any idle bus state, the processor always uses non-pipelined address timing. Pipelined or non-pipelined address timing is then determined on a cycle-by-cycle basis using the $\overline{\text{NA}}$ input. When address pipelining is not used, the address and bus cycle definition remain valid during all wait states. When wait states are added and it is desirable to maintain non-pipelined address timing, it is necessary to negate $\overline{\text{NA}}$ during each T2 state of the bus cycle except the last one.

3.2.2.2 Bus Cycles Using Pipelined Addressing

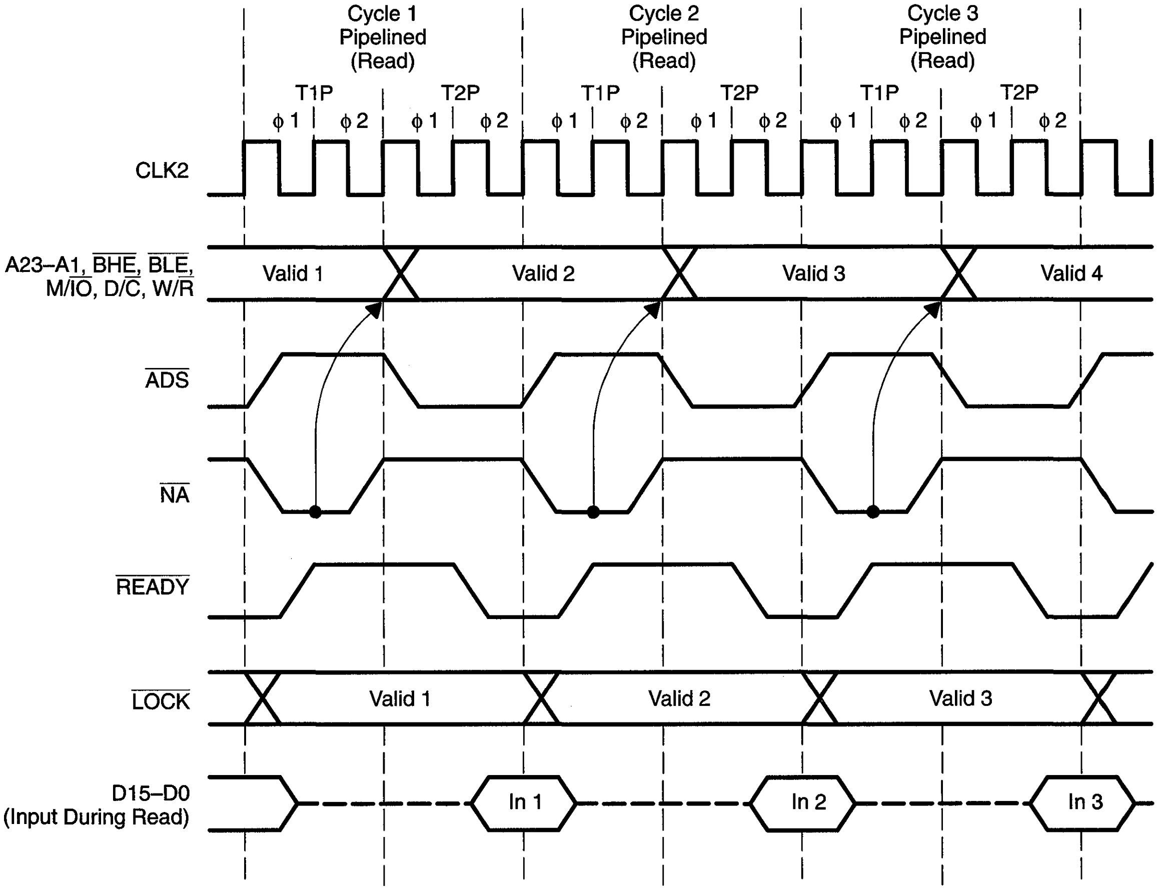
The address pipelining option allows the system to request the address and bus cycle definition of the next internally pending bus cycle before the current bus cycle is acknowledged with $\overline{\text{READY}}$ asserted. If address pipelining is used, the external system hardware has an extra T state of access time to transfer data. The address pipelining option is controlled on a cycle-by-cycle basis by the state of the $\overline{\text{NA}}$ input.

Pipelined Bus States

Pipelined addressing is always initiated by asserting $\overline{\text{NA}}$ during a non-pipelined bus cycle. Within the non-pipelined bus cycle, $\overline{\text{NA}}$ is sampled at the beginning of phase 2 of each T2 state and is only acknowledged by the TI486SLC/E during wait states. When address pipelining is acknowledged, the address ($\overline{\text{BHE}}$, $\overline{\text{BLE}}$, and A23–A1) and bus cycle definition (W/$\overline{\text{R}}$, D/$\overline{\text{C}}$, and M/$\overline{\text{IO}}$) of the next bus cycle are driven before the end of the non-pipelined cycle. The address status output ($\overline{\text{ADS}}$) is asserted simultaneously to indicate validity of the above signals. Once in effect, address pipelining is maintained in successive bus cycles by continuing to assert $\overline{\text{NA}}$ during the pipelined bus cycles.

As in non-pipelined bus cycles, the fastest bus cycles using pipelined address require only two bus states. Figure 3–8 illustrates the fastest read cycles using pipelined address timing. The two bus states for pipelined addressing are T1P and T2P or T1P and T2I. The T1P state is entered following completion of the bus cycle in which the pipelined address and bus cycle definition information was made available and is the first bus state of every pipelined bus cycle. In other words, the T1P state follows a T2 state if the previous cycle was non-pipelined, and follows a T2P state if the previous cycle was pipelined.

Figure 3–8. Fastest Pipelined Read Cycles



Note: Fastest pipelined bus cycles consist of T1P and T2P.

Within the pipelined bus cycle, NA is sampled at the beginning of phase 2 of the T1P state. If the TI486SLC/E has an internally pending bus request and NA is asserted, the T1P state is followed by a T2P state and the address and bus cycle definition for the next pending bus request is made available. If no pending bus request exists, the T1P state is followed by a T2I state regardless of the state of NA and no new address or bus cycle information is driven.

The pipelined bus cycle is terminated in either the T2P or T2I states with the assertion of the READY input and valid data is either input or output depending on the bus cycle type. READY is ignored at the end of the T1P state.

Pipelined Read and Write Cycles

Any bus cycle may be performed with pipelined address timing. When a read cycle is performed, the TI486SLC/E microprocessor floats its data bus and the externally addressed device then drives the data. When a read cycle is acknowledged by READY asserted in either the T2P or T2I bus state, the TI486SLC/E CPU latches the information present at its data pins and terminates the cycle.

When a write cycle is performed, the data bus is driven by the TI486SLC/E CPU beginning in phase 2 of T1P. When a write cycle is acknowledged, the TI486SLC/E write data remains valid throughout phase 1 of the next bus state to provide write data hold time.

Pipelined Wait States

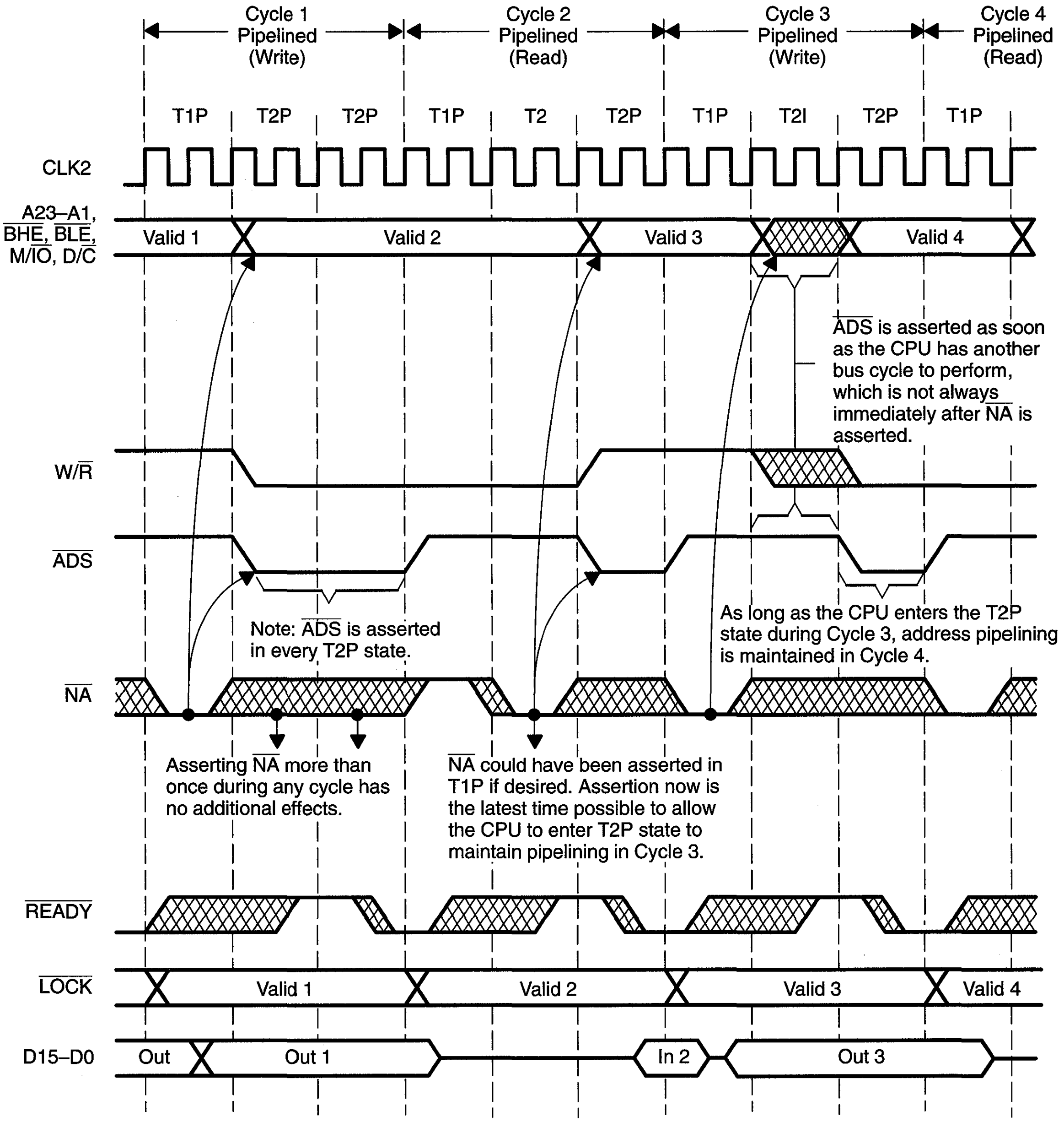
Once a pipelined bus cycle begins, it continues until acknowledged by the external system hardware using the TI486SLC/E $\overline{\text{READY}}$ input. Acknowledging the bus cycle at the end of the first T2P or T2I state results in the shortest possible pipelined bus cycle. If READY is not immediately asserted, however, T2P or T2I states are repeated indefinitely until the $\overline{\text{READY}}$ input is sampled active. Additional T2P or T2I states are referred to as wait states.

Figure 3–9 illustrates pipelined bus cycles with one wait state added to cycles 1 through 3. Cycle 1 is a pipelined cycle with $\overline{\text{NA}}$ asserted during T1P and a pending bus request. READY is sampled inactive at the end of the first T2P state in cycle 1. Therefore, the T2P state is repeated until READY is sampled active at the end of the second T2P and the cycle is then terminated. The TI486SLC/E ignores the $\overline{\text{READY}}$ input at the end of the T1P state. Note that $\overline{\text{ADS}}$, the address and the bus cycle definition signals for the pending bus cycle are all valid during each of the T2P states. Also, asserting NA more than once during the cycle has no additional effects. Pipelined addressing can only output information for the very next bus cycle.

Cycle 2 in Figure 3–9 illustrates a pipelined cycle, with one wait state, where $\overline{\text{NA}}$ is not asserted until the second bus state in the cycle. In this case, the CPU enters the T2 state following T1P because $\overline{\text{NA}}$ is not asserted. During the T2 state, the TI486SLC/E samples $\overline{\text{NA}}$ asserted. Because a bus request is pending internally and $\overline{\text{READY}}$ is not active, the CPU enters the T2P state and asserts $\overline{\text{ADS}}$, valid address and bus cycle definition information for the pending bus cycle. The cycle is then terminated by an active READY at the end of the T2P state.

Cycle 3 of Figure 3–9 illustrates the case where no internal bus request exists until the last state of a pipelined cycle with wait states. In cycle 3, $\overline{\text{NA}}$ is asserted in T1P requesting the next address. Because the CPU does not have an internal bus request pending, The T2I state is entered. However, by the end of the T2I state, a bus request exists. Because $\overline{\text{READY}}$ is not asserted, a wait state is added. The CPU then enters the T2P and asserts $\overline{\text{ADS}}$ and valid address and bus cycle definition information for the pending bus cycle. As long as the CPU enters the T2P state at some point during the bus cycle, pipelined addressing is maintained. $\overline{\text{NA}}$ needs to be asserted only once during the bus cycle to request pipelined addressing.

Figure 3–9. Various Pipelined Cycles (One Wait State)



ADS is asserted as soon as the CPU has another bus cycle to perform, which is not always immediately after NA is asserted.

Note: ADS is asserted in every T2P state.

As long as the CPU enters the T2P state during Cycle 3, address pipelining is maintained in Cycle 4.

Asserting NA more than once during any cycle has no additional effects.

NA could have been asserted in T1P if desired. Assertion now is the latest time possible to allow the CPU to enter T2P state to maintain pipelining in Cycle 3.

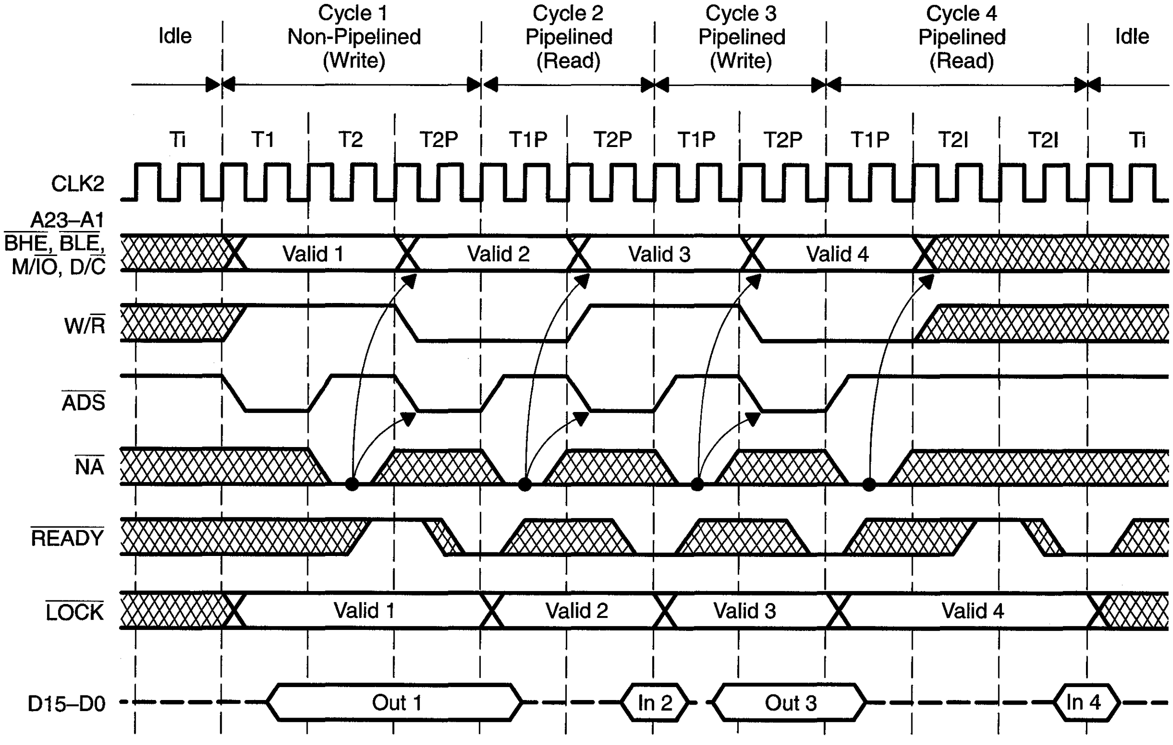
Initiating and Maintaining Pipelined Cycles

Pipelined addressing is always initiated by asserting $\overline{NA}$ during a non-pipelined bus cycle with at least one wait state. For the first bus cycle following RESET, an idle bus, or a hold acknowledge state is always non-pipelined. Therefore, the TI486SLC/E always issues at least one non-pipelined bus cycle following RESET, idle, or hold acknowledge before pipelined addressing takes effect.

Once a bus cycle is in progress and the current address has been valid for one entire bus state, the $\overline{NA}$ input is sampled at the end of every phase one until the bus cycle is acknowledged. Once $\overline{NA}$ is sampled active, the TI486SLC/E microprocessor is free to drive a new address and bus cycle definition on the bus as early as the next bus state and as late as the last bus state in the cycle.

Figure 3–10 illustrates the fastest transition possible to pipelined addressing following an idle bus state. In Cycle 1, $\overline{NA}$ is driven during state T2. Thus, Cycle 1 makes the transition to pipelined address timing, since it begins with T1 but ends with T2P. Because the address for Cycle 2 is available before Cycle 2 begins, Cycle 2 is called a pipelined bus cycle, and it begins with a T1P state. Cycle 2 begins as soon as $\overline{READY}$ asserted terminates Cycle 1.

Figure 3–10. Fastest Transition to Pipelined Address Following Idle Bus State

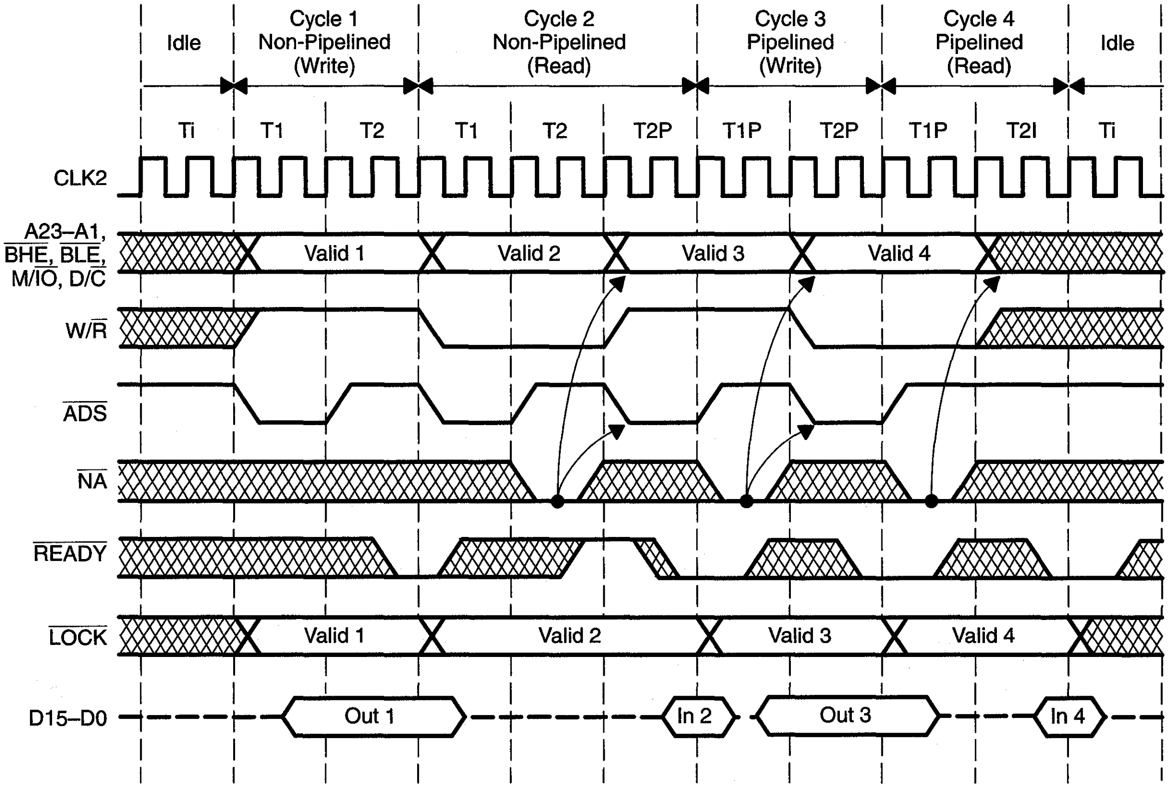


Note: Following any idle bus state (Ti) the address is always non-pipelined and $\overline{NA}$ is sampled only during wait states. To start address pipelining after an idle state requires a non-pipelined cycle with at least one wait state (Cycle 1 above). The pipelined cycles (2, 3, and 4 above) are shown with various numbers of wait states.

Figure 3–11 illustrates transitioning to pipelined addressing during a burst of bus cycles. Cycle 2 makes the transition to pipelined addressing. Comparing Cycle 2 to Cycle 1 of Figure 3–10 illustrates that a transition cycle is the same whenever it occurs consisting of at least T1, T2 ($\overline{NA}$ is asserted at that time), and T2P (provided the TI486SLC/E microprocessor has an internal bus request already pending). T2P states are repeated if wait states are added to the cycle. Cycles 2, 3, and 4 in Figure 3–11 show that once address pipelining is achieved it can be maintained with two-state bus cycles consisting only of T1P and T2P.

Once a pipelined bus cycle is in progress, pipelined timing is maintained for the next cycle by asserting $\overline{NA}$ and detecting that the TI486SLC/E microprocessor enters T2P during the current bus cycle. The current bus cycle must end in state T2P for pipelining to be maintained in the next cycle. T2P is identified by the assertion of $\overline{ADS}$. Figure 3–10 and Figure 3–11 each show pipelining ending after Cycle 4. This occurred because the TI486SLC/E CPU did not have an internal bus request prior to the acknowledgment of Cycle 4.

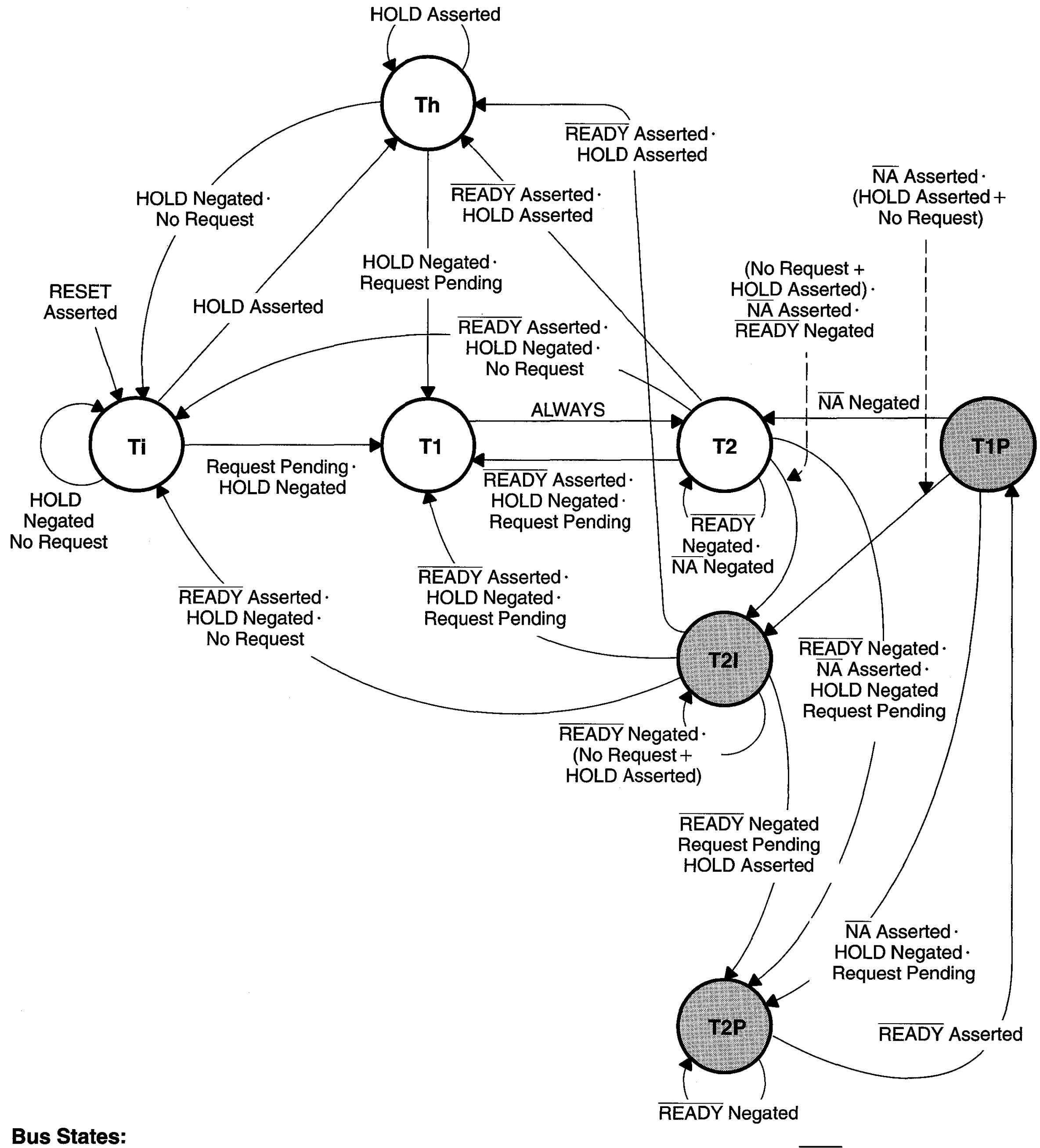
Figure 3–11. Transitioning to Pipelined Address During Burst of Bus Cycles



Note: Following any idle bus state (Ti), addresses are non-pipelined bus cycles, $\overline{NA}$ is sampled only during wait states. Therefore, to begin address pipelining during a group of non-pipelined bus cycles requires a non-pipelined cycle with at least one wait state (Cycle 2 above).

The complete bus state transition diagram, including operation with pipelined address is given in Figure 3–12. This is a superset of the diagram for non-pipelined address. The three additional bus states for pipelined address are shaded.

Figure 3–12. Complete Bus States


HOLD Asserted
Th
READY Asserted · HOLD Asserted
NA Asserted · (HOLD Asserted + No Request)
HOLD Negated · No Request
READY Asserted · HOLD Asserted
HOLD Negated · Request Pending
(No Request + HOLD Asserted) · NA Asserted · READY Negated
RESET Asserted
HOLD Asserted
READY Asserted · HOLD Negated · No Request
NA Negated
ALWAYS
Ti
T1
T2
T1P
Request Pending · HOLD Negated
READY Asserted · HOLD Negated · Request Pending
HOLD Negated No Request
READY Negated · NA Negated
READY Asserted · HOLD Negated · Request Pending
READY Asserted · HOLD Negated · No Request
T2I
READY Negated · NA Asserted · HOLD Negated Request Pending
READY Negated · (No Request + HOLD Asserted)
READY Negated Request Pending HOLD Asserted
NA Asserted · HOLD Negated · Request Pending
T2P
READY Asserted
READY Negated

Bus States:

T1 – First clock of a non-pipelined bus cycle (CPU drives new address and asserts $\overline{\text{ADS}}$).
T2 – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has not been sampled asserted in the current bus cycle.
T2I – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has been sampled asserted in the current bus cycle but there is not yet an internal bus request pending (CPU drives new address and asserts $\overline{\text{ADS}}$).
T2P – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has been sampled asserted in the current bus cycle and there is an internal bus request pending (CPU drives new address and asserts $\overline{\text{ADS}}$).
T1P – First clock of a pipelined bus cycle.
Ti – Idle state.
Th – Hold Acknowledge state (CPU asserts HLDA).

3.2.3 Locked Bus Cycles

When the $\overline{\text{LOCK}}$ signal is asserted the TI486SLC/E microprocessor does not allow other bus master devices to gain control of the system bus. $\overline{\text{LOCK}}$ is driven active in response to executing certain instructions with the LOCK prefix. The LOCK prefix allows indivisible read/modify/write operations on memory operands. $\overline{\text{LOCK}}$ is also active during interrupt acknowledge cycles.

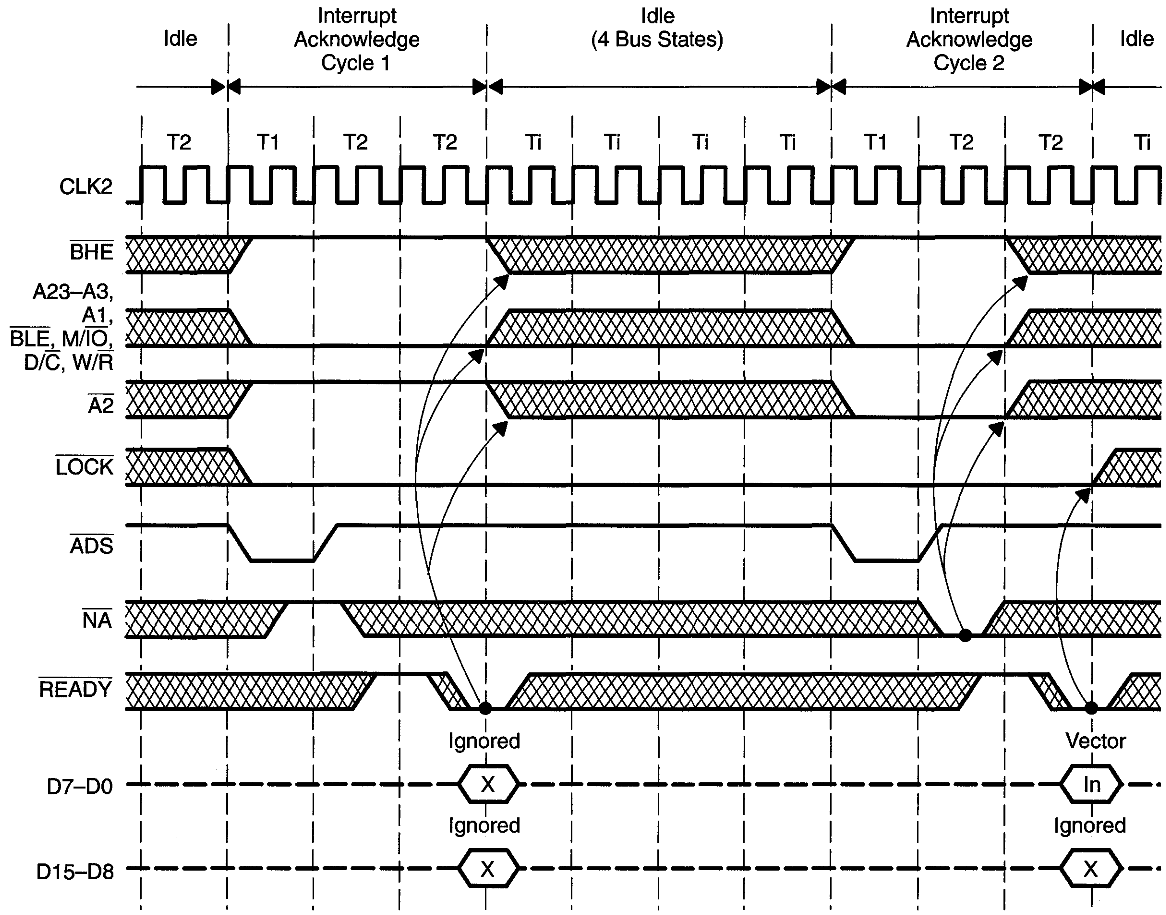
$\overline{\text{LOCK}}$ is activated on the CLK2 edge that begins the first locked bus cycle and is deactivated when $\overline{\text{READY}}$ is returned at the end of the last locked bus cycle. When using non-pipelined addressing, $\overline{\text{LOCK}}$ is asserted during phase 1 of T1. When using pipelined addressing, $\overline{\text{LOCK}}$ is driven valid during phase 1 of T1P.

Figure 3–4 through Figure 3–6 illustrate $\overline{\text{LOCK}}$ timing during non-pipelined cycles and Figure 3–8 through Figure 3–11 cover the pipelined address case.

3.2.4 Interrupt Acknowledge (INTA) Cycles

The TI486SLC/E microprocessor is interrupted by an external source via an input request on the INTR input (when interrupts are enabled). The TI486SLC/E microprocessor responds with two locked interrupt acknowledge cycles. These bus cycles are similar to read cycles. Each cycle is terminated by $\overline{\text{READY}}$ sampled active as shown in Figure 3–13.

Figure 3–13. Interrupt Acknowledge Cycles



Note: Interrupt Vector (0–255) is read on D7–D0 at end of second interrupt acknowledge bus cycle. Because each Interrupt Acknowledge bus cycle is followed by idle bus states, asserting NA has no practical effect.

The state of A2 distinguishes the first and second interrupt acknowledge cycles. The address driven during the first interrupt acknowledge cycle is 4h (A23–A3, A1, BLE=0; A2, BHE=1). the address driven during the second interrupt acknowledge cycle is 0h (A23–A1, BLE=0; BHE=1).

To assure that the interrupt acknowledge cycles are executed indivisibly, the LOCK output is asserted from the beginning of the first interrupt acknowledge cycle until the end of the second interrupt acknowledge cycle. Four idle bus states (Ti) are always inserted by the TI486SLC/E microprocessor between the two interrupt acknowledge cycles.

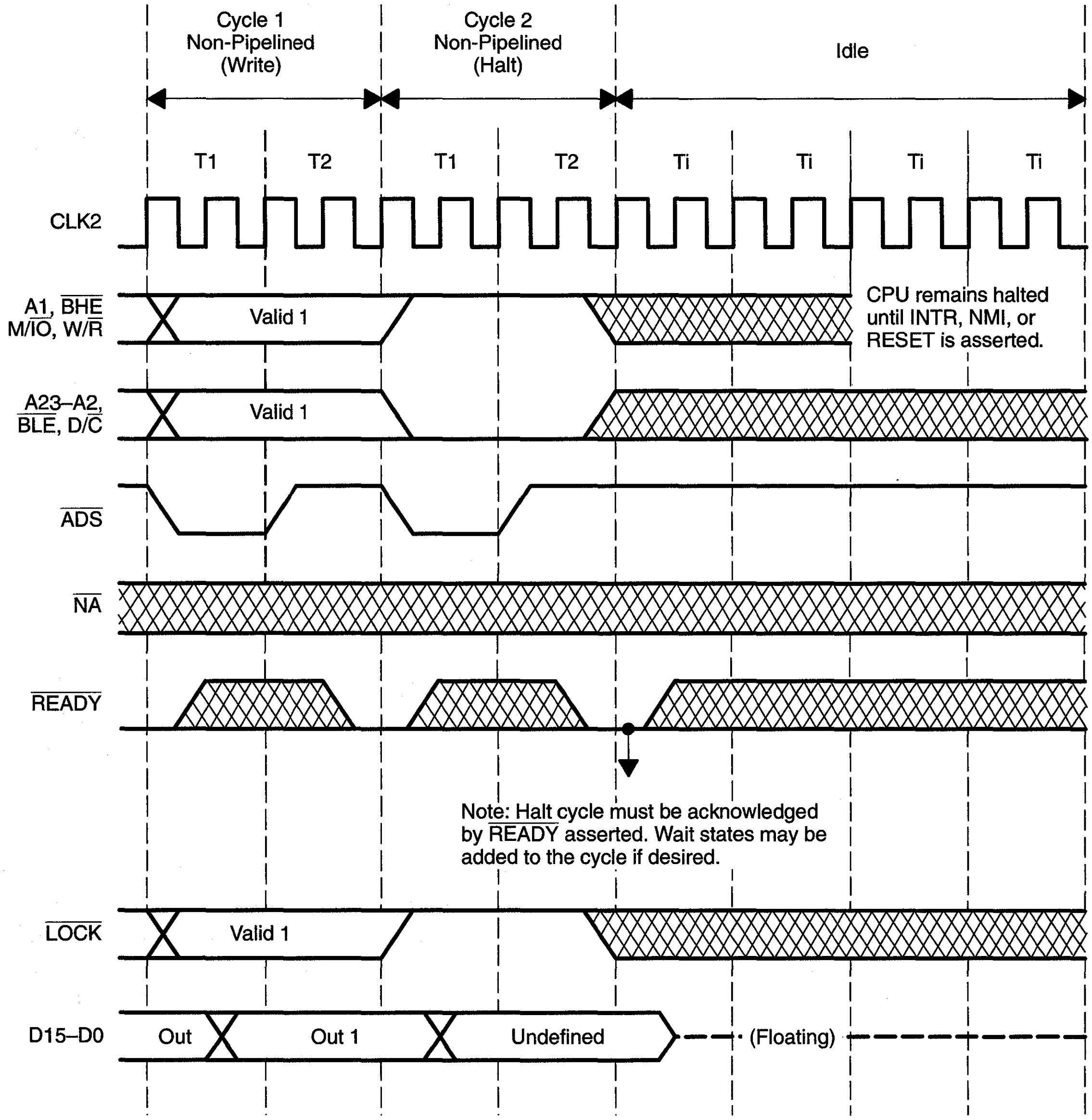
The interrupt vector is read at the end of the second interrupt cycle. The vector is read by the TI486SLC/E microprocessor from D7–D0 of the data bus. The vector indicates the specific interrupt number (from 0–255) requiring service. Throughout the balance of the two interrupt cycles, D15–D0 float. At the end of the first interrupt acknowledge cycle, any data presented to the TI486SLC/E is ignored.

3.2.5 Halt and Shutdown Cycles

Halt Indication Cycle

Executing the HLT instruction causes the TI486SLC/E execution unit to cease operation. Signaling its entrance into the halt state, a halt indication cycle is performed. The halt indication cycle is identified by the state of the bus cycle definition signals (M/$\overline{\text{IO}}$=1, D/$\overline{\text{C}}$=0, W/$\overline{\text{R}}$=1, $\overline{\text{LOCK}}$=1) and an address of 2h (A23–A2=0, A1=1, $\overline{\text{BHE}}$=1, $\overline{\text{BLE}}$=0). The halt indication cycle must be acknowledged by $\overline{\text{READY}}$ asserted. A halted TI486SLC/E microprocessor resumes execution when INTR (if interrupts are enabled), NMI, or RESET is asserted. Figure 3–14 illustrates a non-pipelined halt cycle.

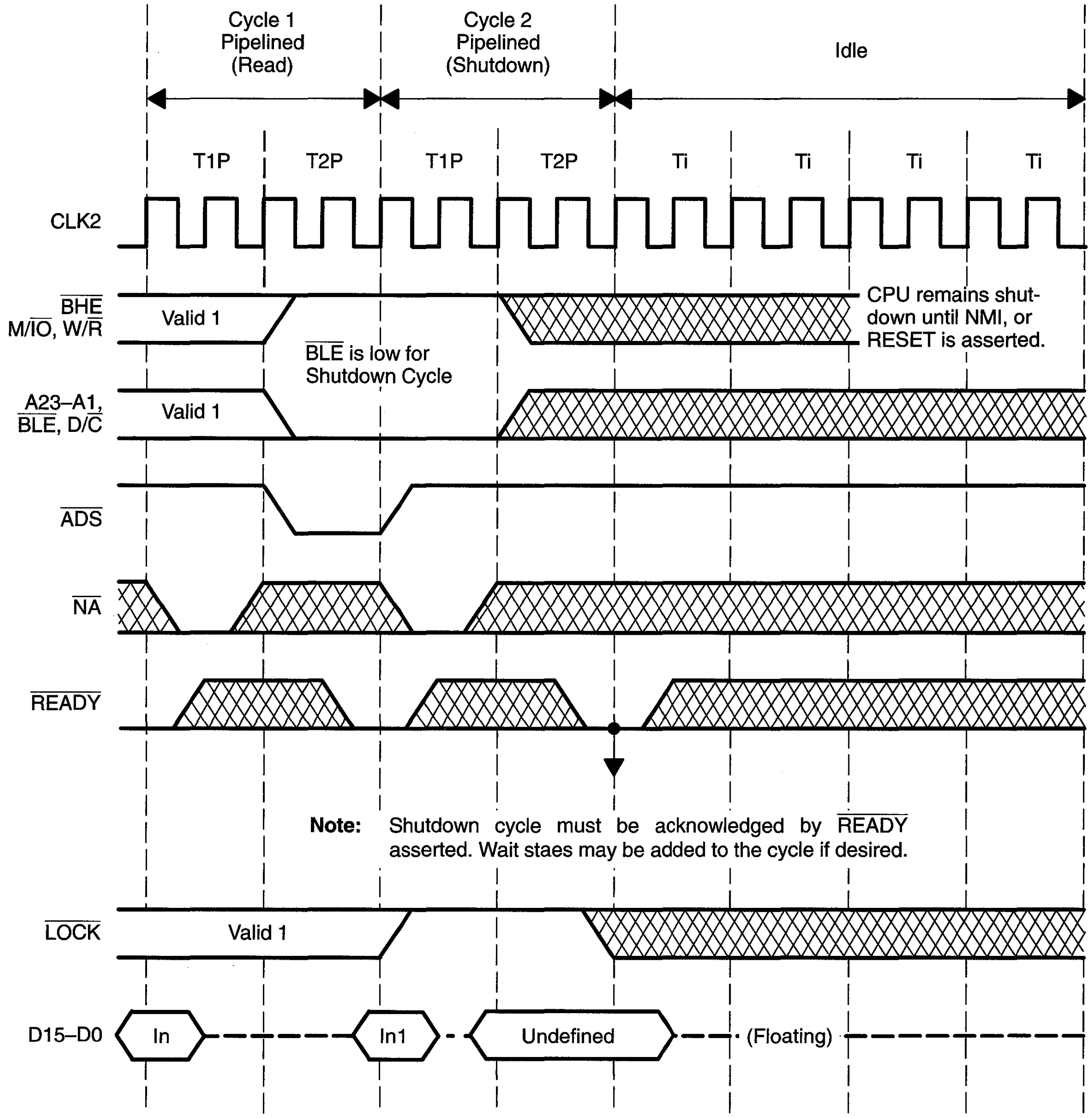
Figure 3–14. Non-Pipelined Halt Cycle



Shutdown Indication Cycle

Shutdown occurs when a severe error is detected that prevents further processing. The TI486SLC/E microprocessor shuts down as a result of a protection fault while attempting to process a double fault as well as the conditions referenced in Chapter 2. Signaling its entrance into the shutdown state, a shutdown indication cycle is performed. The shutdown indication cycle is identified by the state of the bus cycle definition signals (M/$\overline{\text{IO}}$=1, D/$\overline{\text{C}}$=0, W/$\overline{\text{R}}$=1, $\overline{\text{LOCK}}$=1) and an address of 0h (A23–A1=0, $\overline{\text{BHE}}$=1, $\overline{\text{BLE}}$=0). The shutdown indication cycle must be acknowledged by $\overline{\text{READY}}$ asserted. A shutdown TI486SLC/E microprocessor resumes execution only when NMI or RESET is asserted. Figure 3–15 illustrates a shutdown cycle using pipelined addressing.

Figure 3–15. Pipelined Shutdown Cycle



$\overline{\text{BLE}}$ is low for Shutdown Cycle

CPU remains shutdown until NMI, or RESET is asserted.

Note: Shutdown cycle must be acknowledged by $\overline{\text{READY}}$ asserted. Wait staes may be added to the cycle if desired.

3.2.6 Internal Cache Interface

3.2.6.1 Cache Fills

Any unlocked memory read cycle can be cached by the TI486SLC/E. The TI486SLC/E automatically does not cache accesses to memory addresses specified by the non-cacheable region registers. Additionally, the KEN input can be used to enable caching of memory accesses on a cycle-by-cycle basis. The TI486SLC/E acknowledges the KEN input only if the KEN enable bit is set in the CCR0 configuration register.

As shown in Figure 3–16 and Figure 3–17, the TI486SLC/E samples the KEN input one CLK2 before READY is sampled active. If KEN is asserted and the current address is not set as non-cacheable per the non-cacheable region registers, then the TI486SLC/E fills two bytes of a line in the cache with the data present on the data bus pins. The states of BHE and BLE are ignored if KEN is asserted for the cycle.

Figure 3–16. Non-Pipelined Cache Fills Using KEN (With Different Numbers of Wait States)

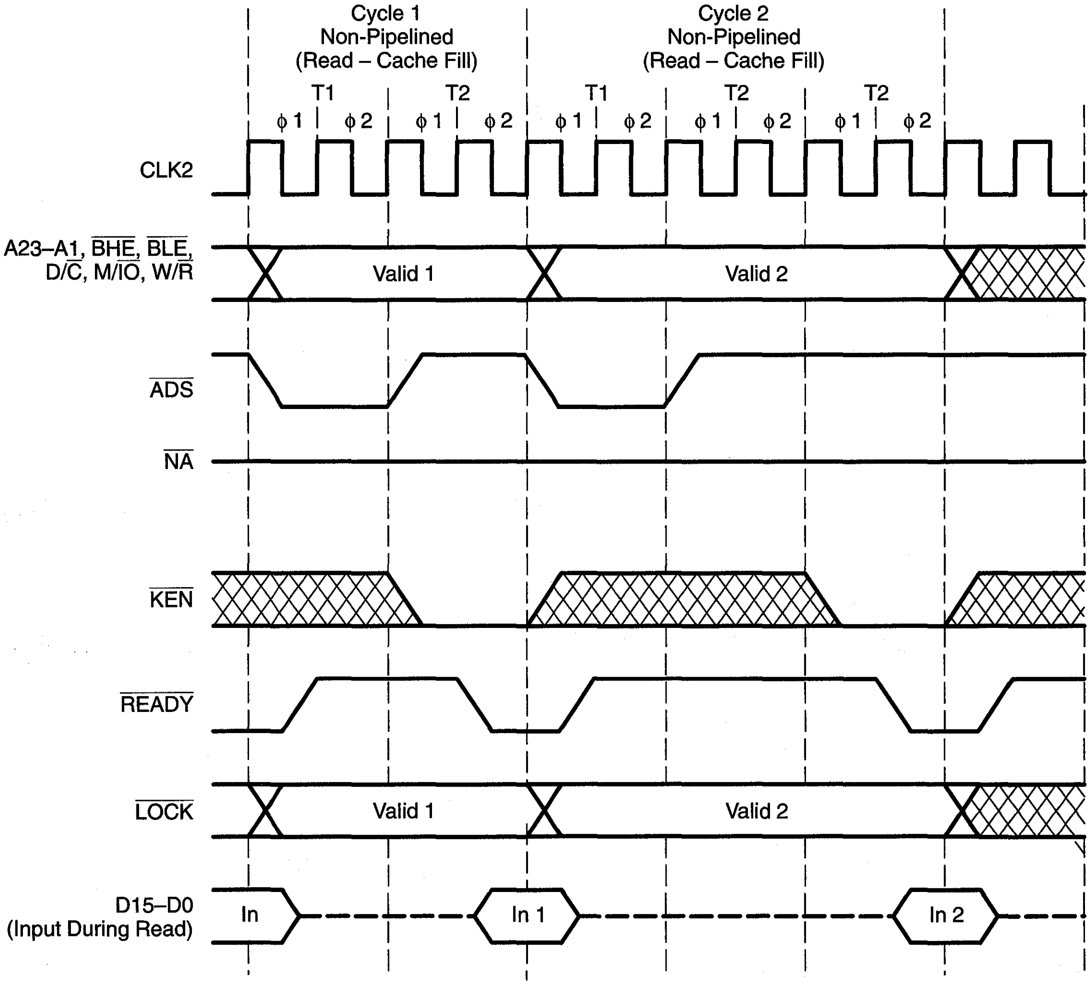
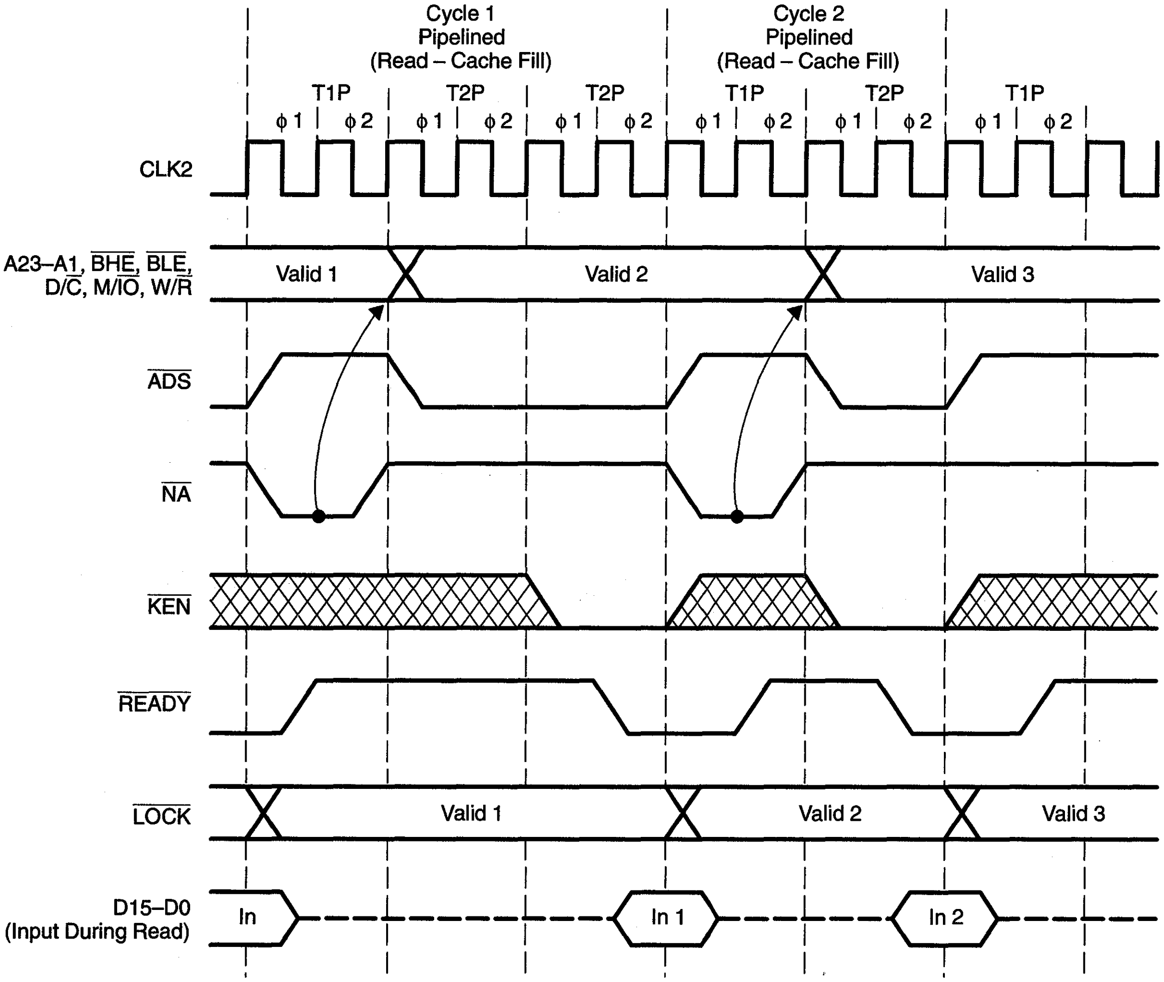


Figure 3–17. Pipelined Cache Fills Using KEN (With Different Numbers of Wait States)



3.2.6.2 Flushing the Cache

To maintain cache coherency with external memory, the TI486SLC/E cache contents should be invalidated when previously cached data is modified in external memory by another bus master. The TI486SLC/E invalidates the internal cache contents during execution of the INVD and WBINVD instructions, following assertion of HLDA if the BARB bit is set in the CCR0 configuration register, or following assertion of FLUSH if the FLUSH bit is set in CCR0.

The TI486SLC/E samples the $\overline{\text{FLUSH}}$ input on the rising edge of CLK2 corresponding to the beginning of phase 2 of the internal processor clock. If $\overline{\text{FLUSH}}$ is asserted, the TI486SLC/E invalidates the entire contents of the internal cache. The actual point in time where the cache is invalidated depends upon the internal state of the execution pipeline. $\overline{\text{FLUSH}}$ must be asserted for at least two CLK2 periods and must meet specified setup and hold times to be recognized on a specific CLK2 edge.

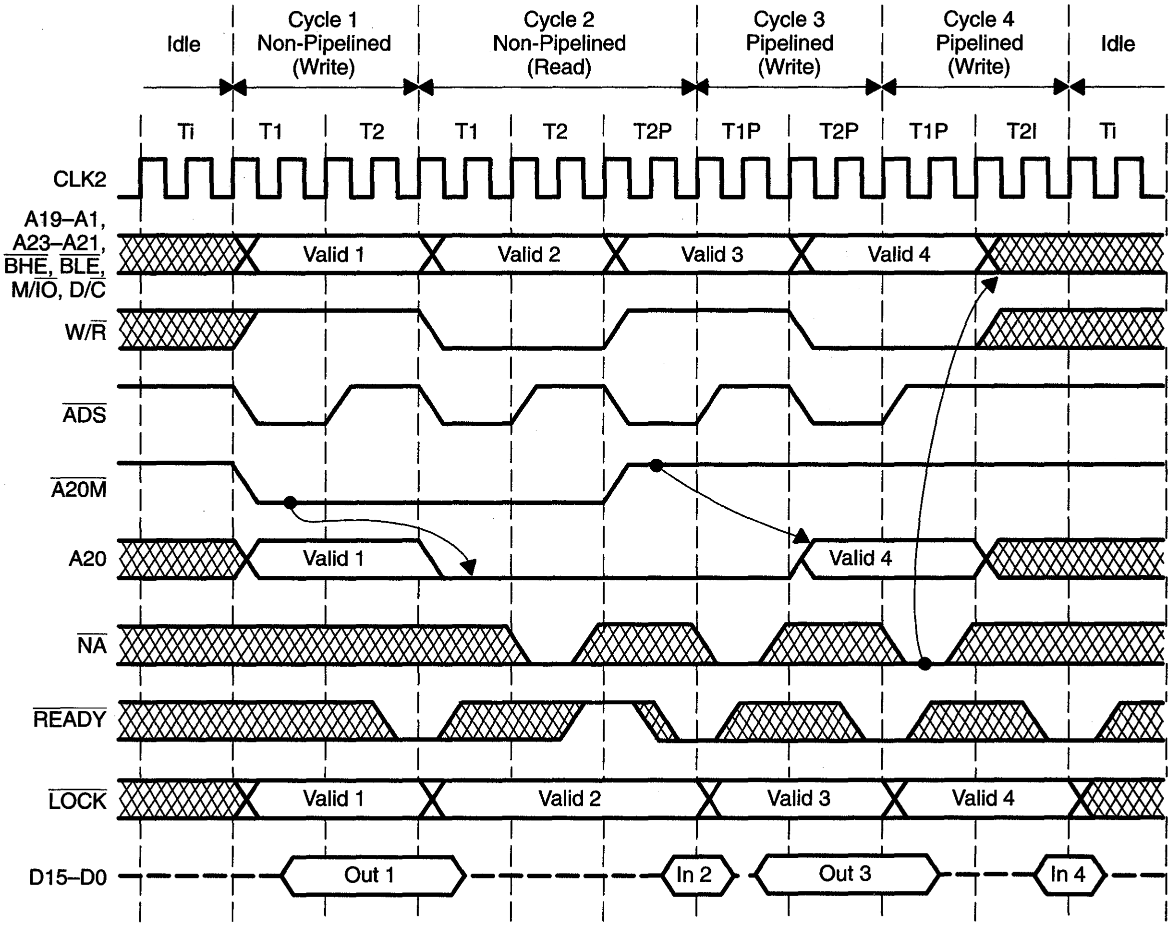
3.2.7 Address Bit 20 Masking

The TI486SLC/E can be forced to provide 8086 1-MByte address wraparound compatibility by setting the A20 bit in the CCR0 configuration register and asserting the $\overline{\text{A20M}}$ input. When the $\overline{\text{A20M}}$ is asserted, the 20th bit in the address to both the internal cache and the external bus pin is masked (zeroed).

As shown in Figure 3–18, the TI486SLC/E samples the $\overline{\text{A20M}}$ input on the rising edge of CLK2 corresponding to the beginning of phase 2 of the internal processor clock. If $\overline{\text{A20M}}$ is asserted and paging is not enabled, the TI486SLC/E masks the A20 signal internally starting with the next cache access and externally starting with the next bus cycle. If paging is enabled, the A20 signal is not masked regardless of the state of $\overline{\text{A20M}}$. A20 remains masked until the access following detection of an inactive state on the $\overline{\text{A20M}}$ pin. $\overline{\text{A20M}}$ must be asserted for a minimum of two CLK2 periods and must meet specified setup and hold times to be recognized on a specific CLK2 edge.

An alternative to using the $\overline{\text{A20M}}$ pin is provided by the NC0 bit in the CCR0 configuration register. The TI486SLC/E automatically does not cache accesses, to the first 64 KBytes and to 1 MByte + 64 KBytes, if the NC0 bit is set. This prevents data within the wraparound memory area from residing in the internal cache and thus eliminates the need for masking A20 to the internal cache.

Figure 3–18. Masking A20 Using $\overline{\text{A20M}}$ During Burst of Bus Cycles



3.2.8 Hold Acknowledge State

The hold acknowledge state provides the mechanism for an external device in a TI486SLC/E system to acquire the TI486SLC/E system bus while the TI486SLC/E is held in an inactive bus state. This allows external bus masters to take control of the TI486SLC/E bus and directly access system hardware in a shared manner with the TI486SLC/E. The TI486SLC/E continues to execute instructions out of the cache (if enabled) until a system bus cycle is required.

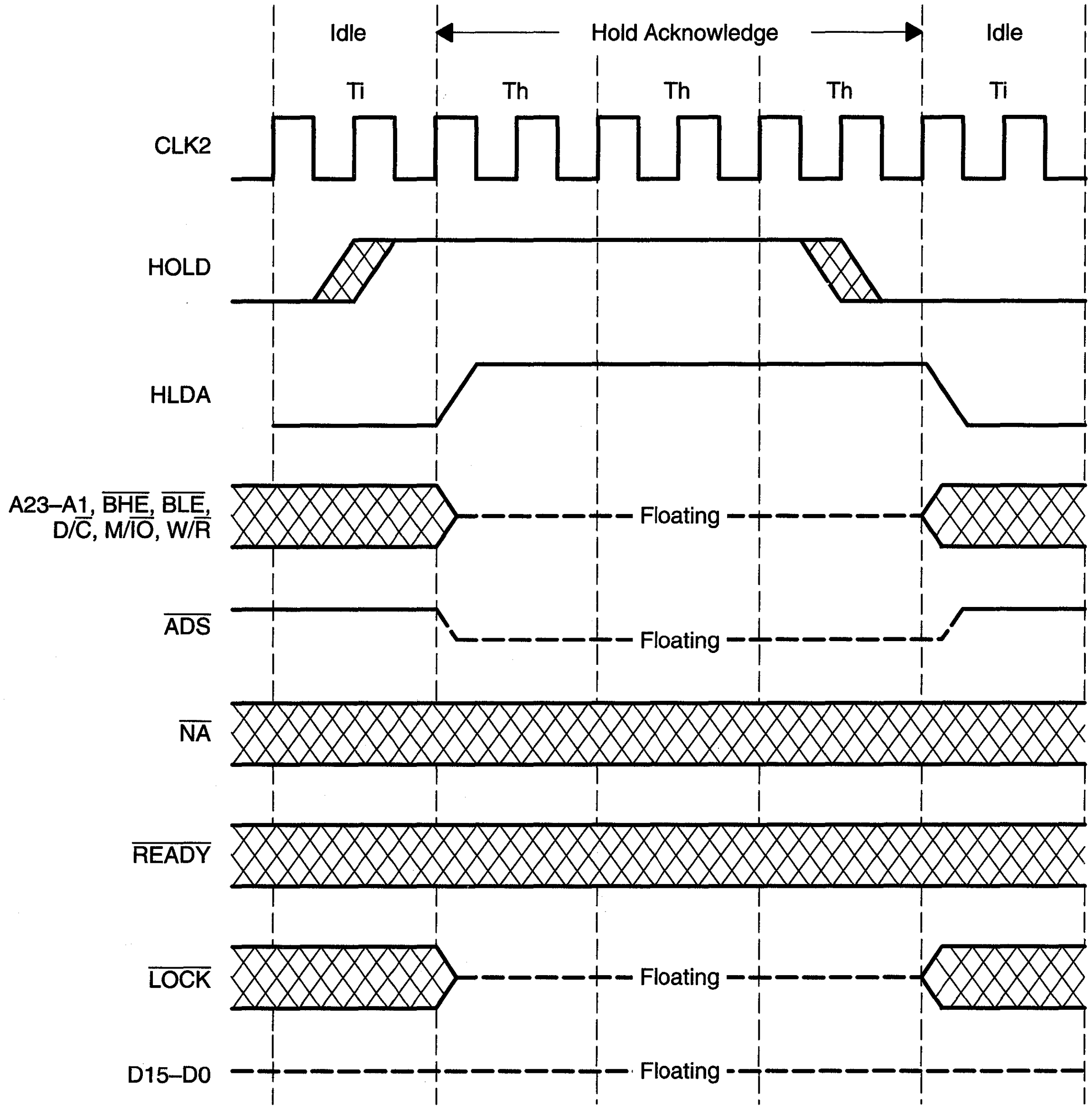
The hold acknowledge state (Th) is entered in response to assertion of the HOLD input. in the hold acknowledge state, the TI486SLC/E microprocessor floats all output and bidirectional signals, except for HLDA and $\overline{\text{SUSPA}}$. HLDA is asserted as long as the TI486SLC/E CPU remains in the hold acknowledge state and all inputs except HOLD, $\overline{\text{FLUSH}}$, $\overline{\text{FLT}}$, $\overline{\text{SUSP}}$ and RESET are ignored.

State Th may be entered directly from a bus idle state, as in Figure 3–19, or after the completion of the current physical bus cycle if the LOCK signal is not asserted, as in Figure 3–20 and Figure 3–21. The CPU samples the HOLD input on the rising edge of CLK2 corresponding to the beginning of phase 1 of internal processor clock. HOLD must meet specified setup and hold times to be recognized at a given CLK2 edge.

The hold acknowledge state is exited in response to the HOLD input being negated. The next bus start is an idle state (Ti) if no bus request is pending, as in Figure 3–19. If a bus request is internally pending, as in Figure 3–20 and Figure 3–21, the next bus state is T1. State Th is also exited in response to RESET being asserted. If HOLD remains asserted when RESET goes inactive, the TI486SLC/E enters the hold acknowledge state before performing any bus cycles provided HOLD is still asserted when the CPU is ready to perform its first bus cycle.

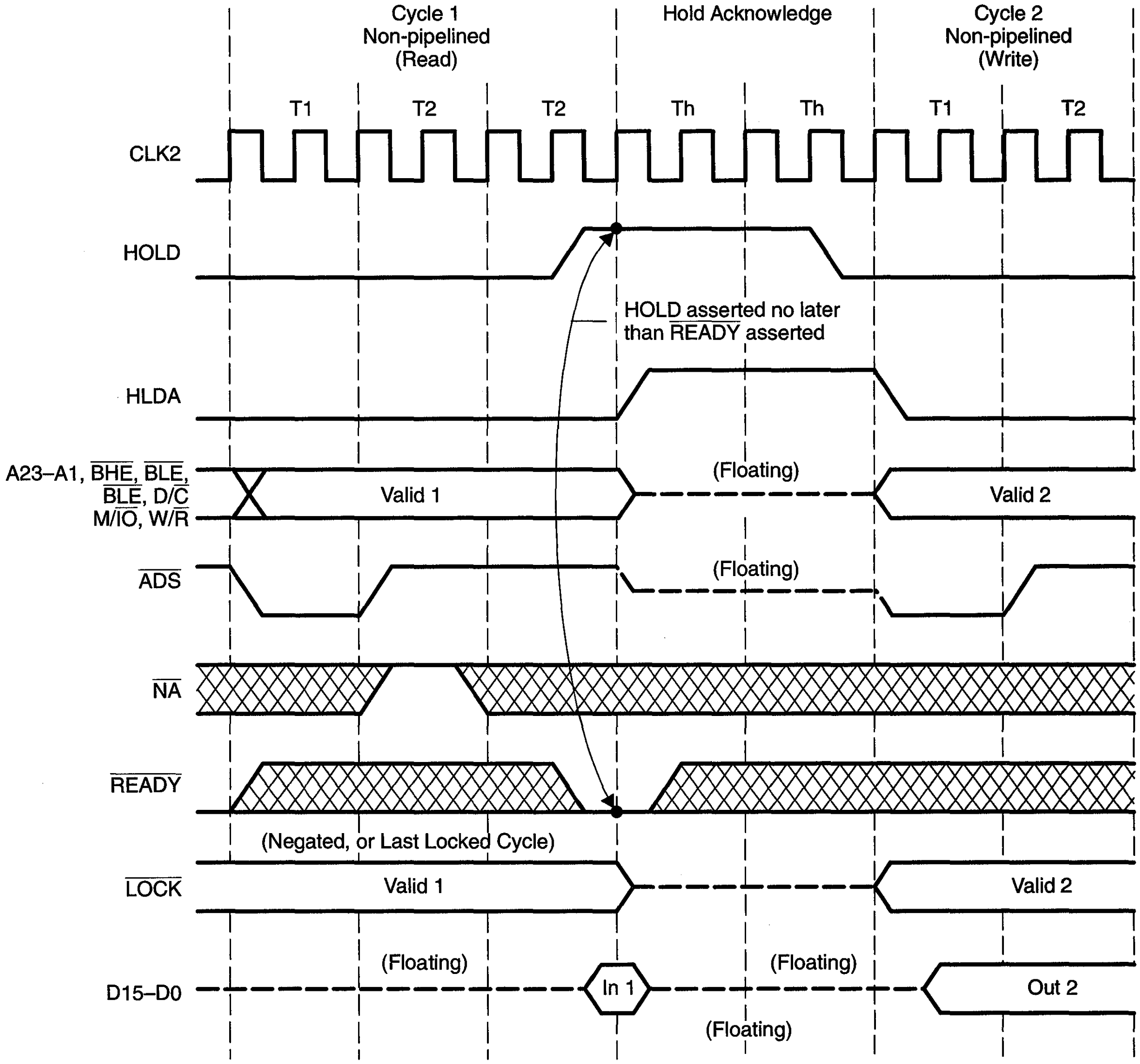
If a rising edge occurs on the edge-triggered NMI input while in state Th, the event is remembered as a non-maskable interrupt 2 and is serviced when the state is exited.

Figure 3–19. Requesting Hold from Idle Bus State



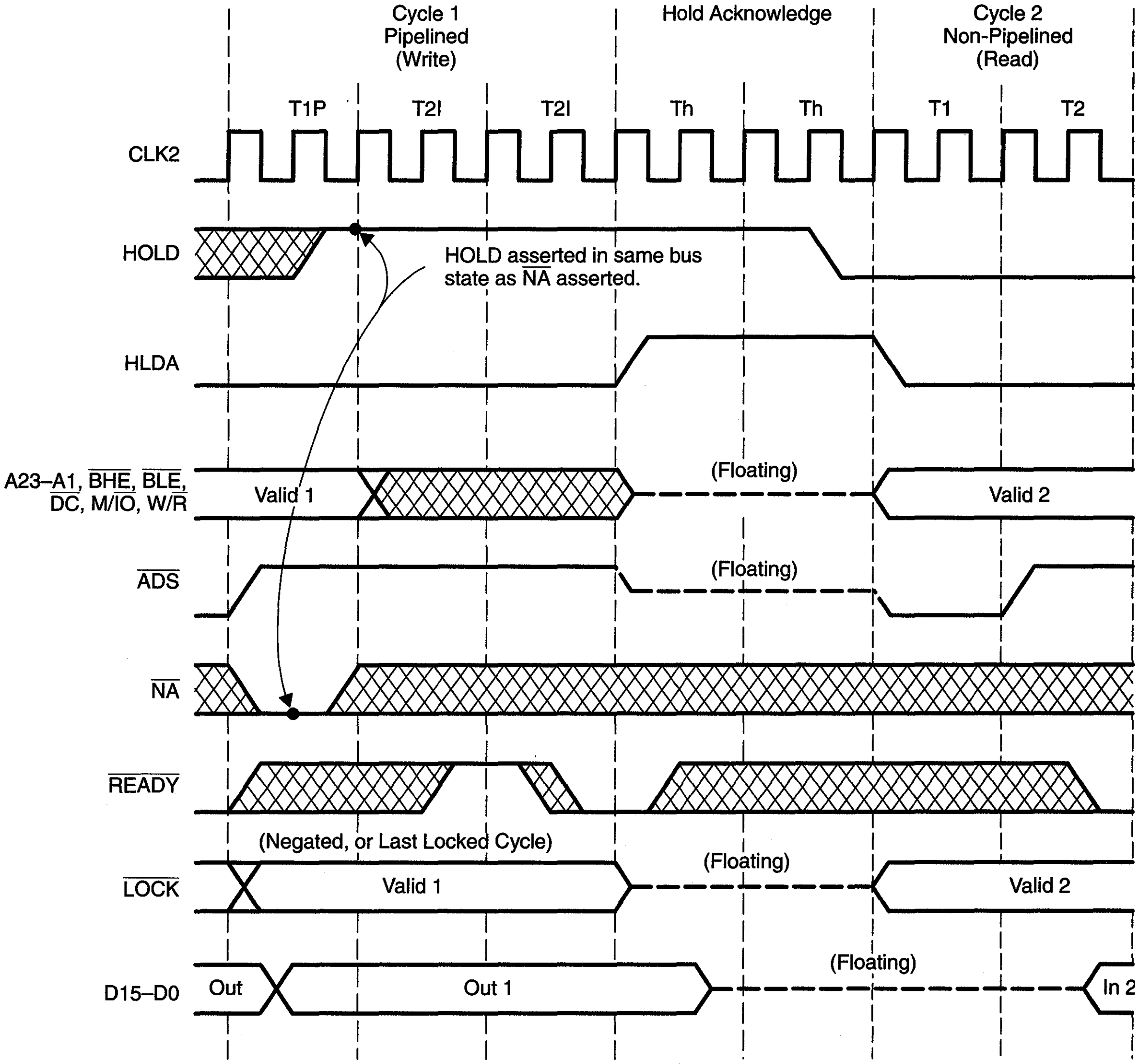
Note: For maximum design flexibility the CPU has no internal pullup resistors on its outputs. External pullups may be required on $\overline{ADS}$ and other output to keep them negated during hold acknowledge period.

Figure 3–20. Requesting Hold from Active Non-Pipelined Bus



Note: HOLD is a synchronous input and can be asserted at any CLK2 edge, provided setup and hold requirements are met. This waveform is useful for determining hold acknowledge latency.

Figure 3–21. Requesting Hold from Active Pipelined Bus



Note: HOLD is a synchronous input and can be asserted at any CLK2 edge, provided setup and hold requirements are met. This waveform is useful for determining hold acknowledge latency.

3.2.9 Coprocessor Interface

The coprocessor interface consists of the data bus, address bus, bus cycle definition signals, and the coprocessor interface signals ($\overline{\text{BUSY}}$, $\overline{\text{ERROR}}$ and PEREQ). The TI486SLC/E automatically accesses dedicated coprocessor I/O addresses 80 00F8h, 80 00FCh, and 80 00FEh to transfer opcodes and operands to/from the coprocessor whenever a coprocessor instruction is decoded. Coprocessor cycles can be either read or write and can be either non-pipelined or pipelined. Coprocessor cycles must be terminated by $\overline{\text{READY}}$ and, as with any other bus cycle, can be terminated as early as the second bus state of the cycle.

$\overline{\text{BUSY}}$, $\overline{\text{ERROR}}$ and PEREQ are asynchronous level-sensitive inputs used to synchronize CPU and coprocessor operation. All three signals are sampled at the beginning of phase 1 and must meet specified setup and hold times to be recognized at a given CLK2 edge.

3.2.10 SMM Interface

System Management Mode (SMM) uses two TI486SLC/E pins, $\overline{\text{SMI}}$ and $\overline{\text{SMADS}}$. the bidirectional $\overline{\text{SMI}}$ pin is a non-maskable interrupt that is higher priority than the NMI input. $\overline{\text{SMI}}$ must be active for at least four CLK2 periods to be recognized by the TI486SLC/E. Once the TI486SLC/E recognizes the active $\overline{\text{SMI}}$ input, the CPU drives the $\overline{\text{SMI}}$ pin low for the duration of the SMI service routine.

The $\overline{\text{SMADS}}$ pin outputs the SMM Address Strobe that indicates a SMM memory bus cycle is in progress and a valid SMM address is on the address bus. The $\overline{\text{SMADS}}$ functional timing, output delay times and float delay times are identical to the main memory address strobe ($\overline{\text{ADS}}$) timing.

3.2.10.1 SMI Handshake

The functional timing for $\overline{\text{SMI}}$ interrupt is shown in Figure 3–22. Five significant events take place during a TI486SLC/E $\overline{\text{SMI}}$ handshake:

1) The $\overline{\text{SMI}}$ input pin is driven active (low) by the system logic.

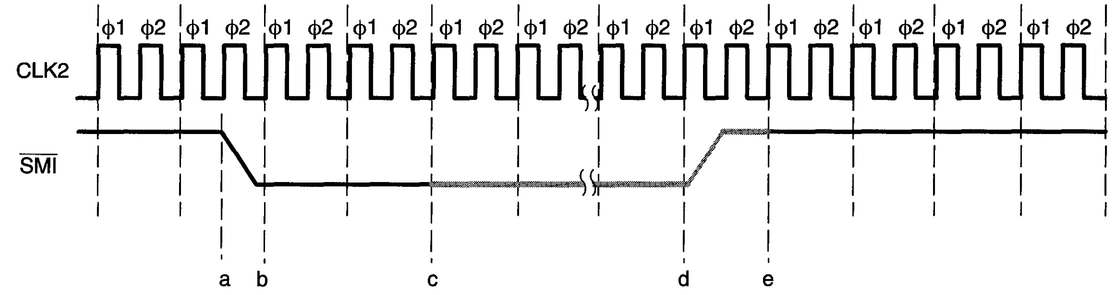
2) The CPU samples $\overline{\text{SMI}}$ active on the rising edge of CLK2 phase 1.

3) Four CLK2s after sampling the $\overline{\text{SMI}}$ active, the CPU switches the $\overline{\text{SMI}}$ pin to an output and drives $\overline{\text{SMI}}$ low.

4) Following execution of the RSM instruction, the CPU drives the $\overline{\text{SMI}}$ pin high for two CLK2s indicating completion of the SMI service routine.

5) The CPU stops driving the $\overline{\text{SMI}}$ pin high and switches the $\overline{\text{SMI}}$ pin to an input in preparation for the next SMI interrupt. The system logic is responsible for maintaining the $\overline{\text{SMI}}$ pin at an inactive (high) level after the pin has been changed to an input.

Figure 3–22. $\overline{\text{SMI}}$ Timing

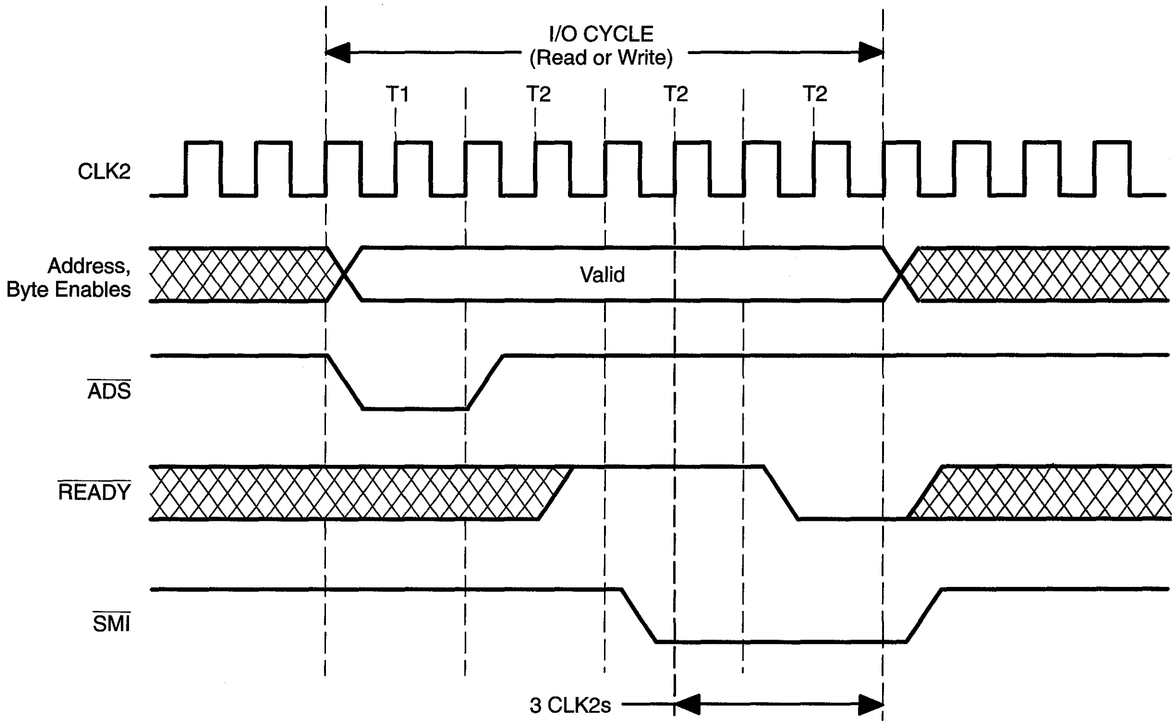

φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2
CLK2
SMI
a
b
c
d
e

▬ Indicates that the TI486SLC/E drives the $\overline{\text{SMI}}$ pin.

3.2.10.2 I/O Trapping

The TI486SLC/E provides I/O trapping that can be used to facilitate power management of I/O peripherals. When an I/O bus cycle is issued, the I/O address is driven onto the address bus and can be decoded by external logic. If a trap to the SMI handler is required, the $\overline{\text{SMI}}$ input should be activated at least three CLK2 edges prior to returning the $\overline{\text{READY}}$ input for the I/O cycle. The timing for creating an I/O trap via the $\overline{\text{SMI}}$ input is shown in Figure 3–23. The TI486SLC/E immediately traps to the SMI interrupt handler following execution of the I/O instruction, and no other instructions are executed between completion of the I/O instruction and entering the SMI service routine. The I/O trap mechanism is not active during coprocessor accesses.

Figure 3–23. I/O Trap Timing



3.2.11 Power Management

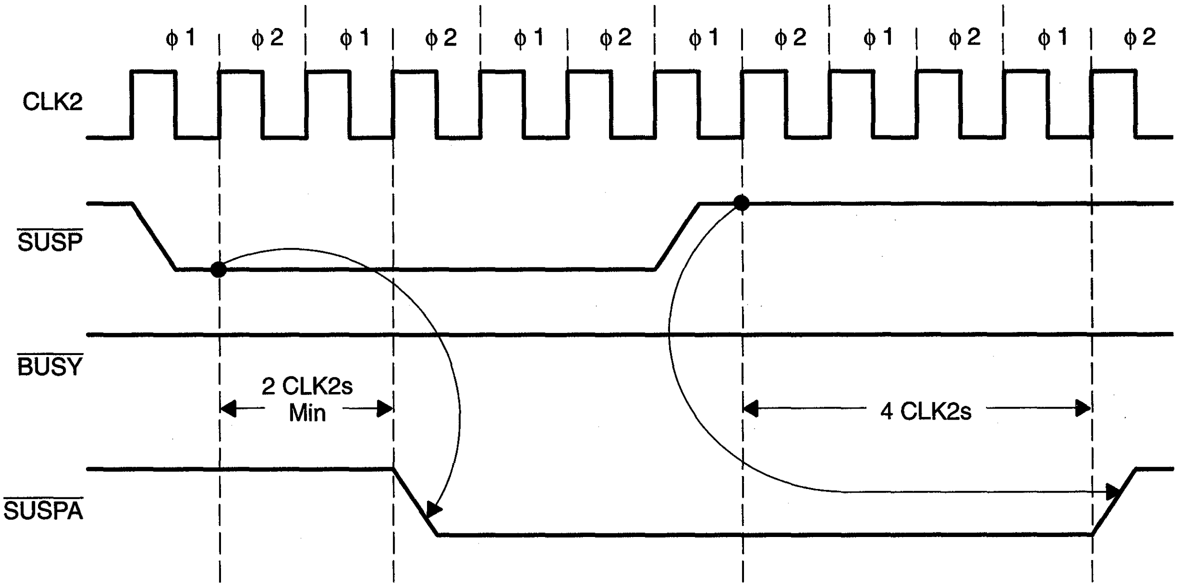
$\overline{\text{SUSP}}$ Initiated Suspend Mode

The TI486SLC/E enters suspend mode when the $\overline{\text{SUSP}}$ input is asserted and execution of the current instruction, any pending decoded instructions and associated bus cycles are completed. The TI486SLC/E also waits for the coprocessor to indicate a not busy status ($\overline{\text{BUSY}}$=1) prior to entering suspend mode. The $\overline{\text{SUSPA}}$ output is then asserted. The TI486SLC/E responds to $\overline{\text{SUSP}}$ and asserts $\overline{\text{SUSPA}}$ only if the SUSP bit is set in the CCR0 configuration register.

Figure 3–24 illustrates the TI486SLC/E functional timing for $\overline{\text{SUSP}}$ initiated suspend mode. $\overline{\text{SUSP}}$ is sampled on the phase 2 CLK2 rising edge and must meet specified setup and hold times to be recognized at a particular CLK2 edge. The time from assertion of $\overline{\text{SUSP}}$ to activation of $\overline{\text{SUSPA}}$ varies depending on which instructions were decoded prior to assertion of $\overline{\text{SUSP}}$. The minimum time from $\overline{\text{SUSP}}$ sampled active to $\overline{\text{SUSPA}}$ asserted is 2 CLK2s. As a maximum, the TI486SLC/E may execute up to two instructions and associated bus cycles prior to asserting $\overline{\text{SUSPA}}$. The time required for the TI486SLC/E to deactivate $\overline{\text{SUSPA}}$ once $\overline{\text{SUSP}}$ has been sampled inactive is 4 CLK2s.

If the TI486SLC/E is in a hold acknowledge state and $\overline{\text{SUSP}}$ is asserted, the processor may or may not enter suspend mode depending on the state of the TI486SLC/E internal execution pipeline. If the TI486SLC/E is in a SUSP initiated suspend state and the CLK2 input is not stopped, the processor recognizes and acknowledges the HOLD input and stores the occurrence of $\overline{\text{FLUSH}}$, NMI and INTR (if enabled) for execution once suspend mode is exited.

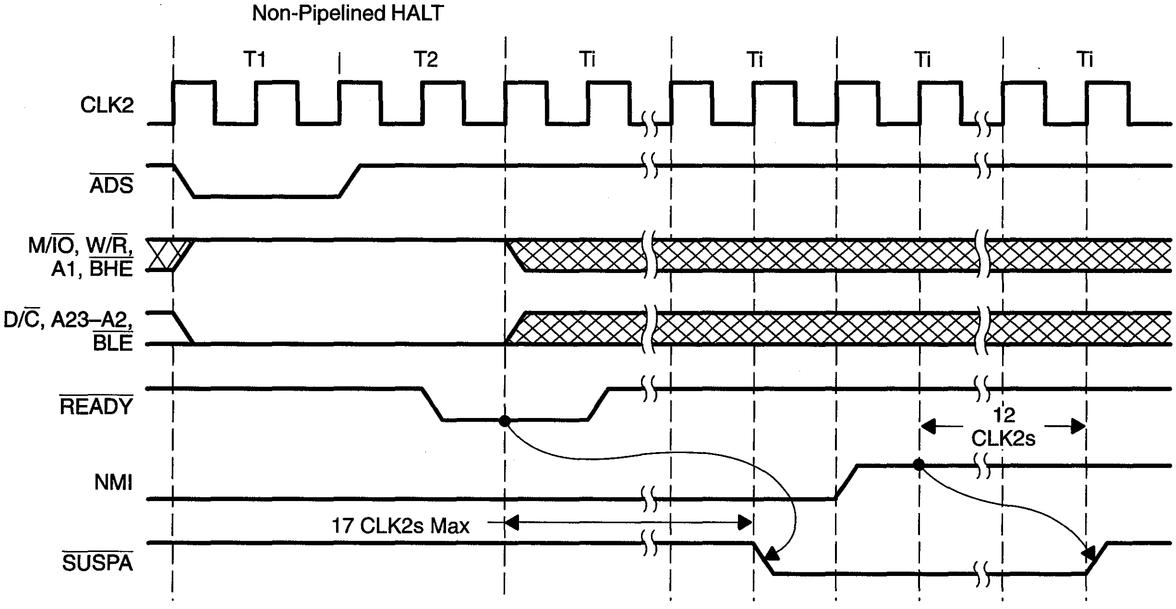
Figure 3–24. $\overline{\text{SUSP}}$ Initiated Suspend Mode



HALT Initiated Suspend Mode

The TI486SLC/E also enters suspend mode as a result of executing a HALT instruction. The SUSPA output is asserted no more than 17 CLK2s following a READY sampled active for the HALT bus cycle as shown in Figure 3–25. Suspend mode is then exited upon recognition of an NMI or an unmasked INTR. SUSPA is deactivated 12 CLK2s after sampling of an active NMI or unmasked INTR. If the TI486SLC/E is in a HALT initiated suspend mode and the CLK2 input is not stopped, the processor recognizes and acknowledges the HOLD input and stores the occurrence of FLUSH for execution once suspend mode is exited.

Figure 3–25. Halt Initiated Suspend Mode

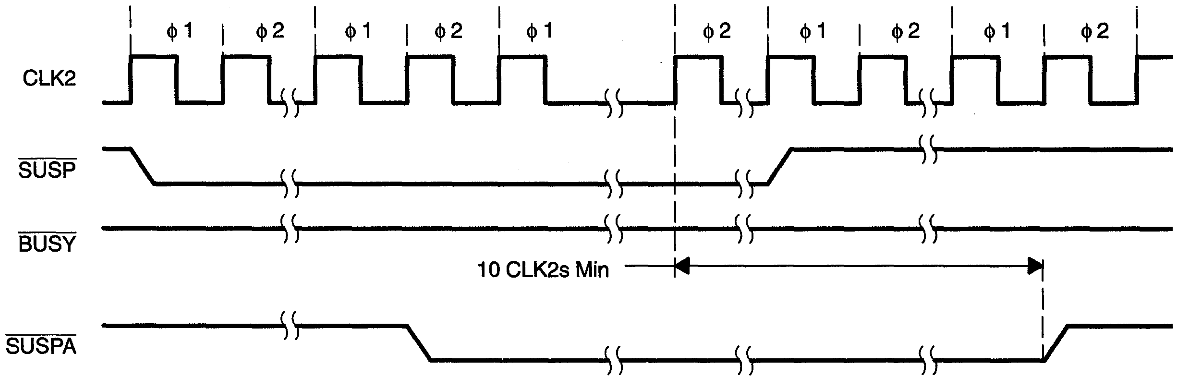


Stopping the Input Clock

Because the TI486SLC/E is a static device, the input clock (CLK2) can be stopped and restarted without loss of any internal CPU data. CLK2 can be stopped in either phase 1 or phase 2 of the clock and in either a logic high or logic low state. However, entering suspend mode prior to stopping CLK2 dramatically reduces the CPU current requirements. Therefore, the recommended sequence for stopping CLK2 is to initiate TI486SLC/E suspend mode, wait for assertion of $\overline{\text{SUSPA}}$ by the processor and then stop the input clock.

The TI486SLC/E remains suspended until CLK2 is restarted and suspend mode is exited as described above. While CLK2 is stopped, the TI486SLC/E can no longer sample and respond to any input stimulus including the HOLD, $\overline{\text{FLUSH}}$, NMI, INTR and RESET inputs. Figure 3–26 illustrates the recommended sequence for stopping CLK2 using $\overline{\text{SUSP}}$ to initiate suspend mode. CLK2 should be stable for a minimum of 10 clock periods before $\overline{\text{SUSP}}$ is deasserted.

Figure 3–26. Stopping CLK2 During Suspend Mode



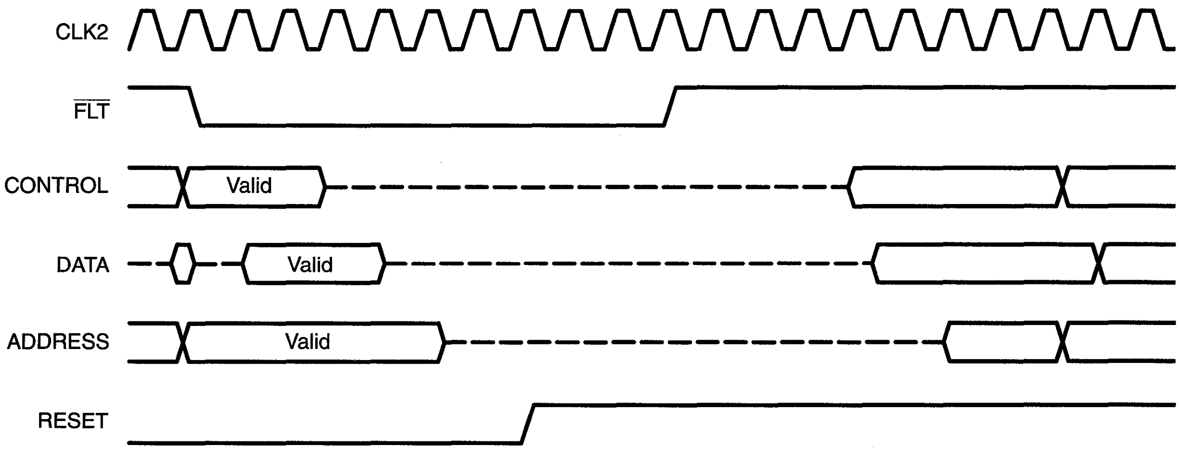
3.2.12 Float

Activating the $\overline{\text{FLT}}$ input floats all TI486SLC/E microprocessor bidirectional and output signals. Asserting $\overline{\text{FLT}}$ electrically isolates the TI486SLC/E microprocessor from the surrounding circuitry. This feature is useful in board-level test environments. As the TI486SLC/E microprocessor is packaged in a surface mount PQFP, it is not usually socketed and cannot be removed from the motherboard when In-Circuit Emulation (ICE) is needed. Float capability allows connection of an emulator by clamping the emulator probe onto the TI486SLC/E microprocessor PQFP without removing it from the circuit board.

$\overline{\text{FLT}}$ is an asynchronous, active-low input. It is recognized on the rising edge of CLK2. When recognized, it aborts the current bus state and floats the outputs of the TI486SLC/E microprocessor as shown in Figure 3–27. $\overline{\text{FLT}}$ must be asserted for a minimum of 16 CLK2 cycles. To exit the float condition, RESET should be asserted and held asserted until after $\overline{\text{FLT}}$ is deasserted.

Asserting the $\overline{\text{FLT}}$ input unconditionally aborts the current bus cycle and forces the TI486SLC/E microprocessor into the float mode. As a result, the TI486SLC/E microprocessor is not guaranteed to enter float in a valid state. After deactivating $\overline{\text{FLT}}$, the TI486SLC/E CPU is not guaranteed to exit float in a valid state. The TI486SLC/E microprocessor RESET input must be asserted prior to exiting float to guarantee that the TI486SLC/E is reset and that it returns in a valid state.

Figure 3–27. Entering and Exiting Float



Chapter 4

TI486DLC/E Bus Interface

In this chapter, an overview of the TI486DLC/E provides a summary of the processor signals, functional description of all pins, functional timing and bus operations (including non-pipelined and pipelined addressing), various interfaces, and power management.

Topic **Page**

4.1 Overview

The following sections describe the TI486DLC/E input and output signals. The discussion of these signals is arranged by functional groups as shown in Figure 4–1. Table 4–1 gives a brief description of each of the TI486DLC/E signals.

Figure 4–1. TI486DLC/E Functional Signal Groupings

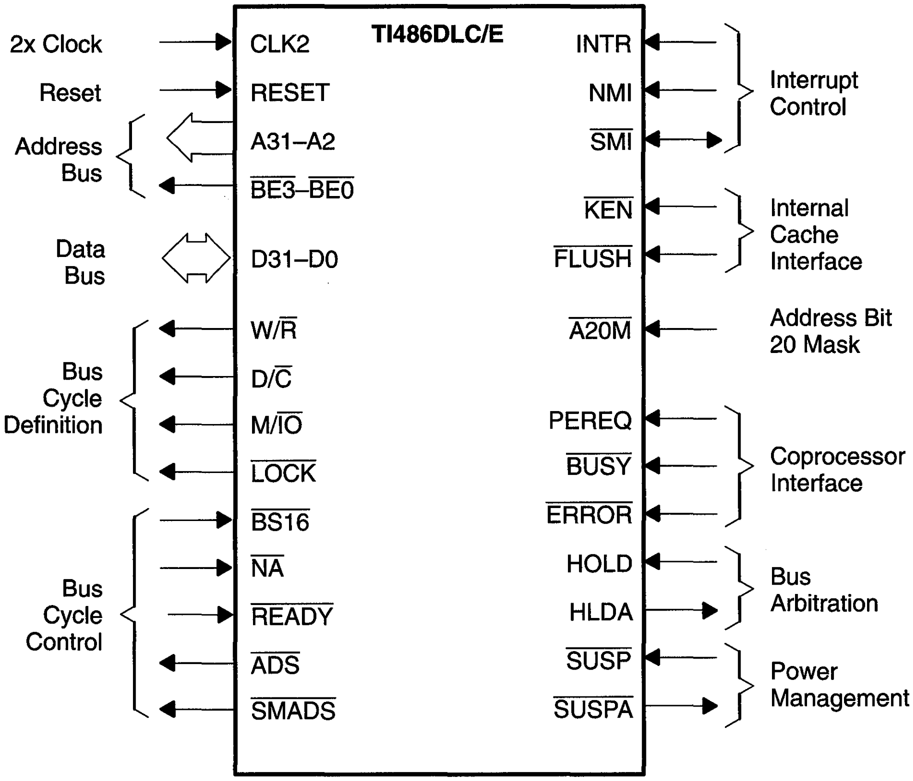


Table 4–1. TI486DLC/E Signal Summary

SIGNAL	SIGNAL NAME	SIGNAL GROUP
$\overline{\text{A20M}}$	Address Bit 20 Mask	
A31–A2	Address Bus Lines	Address bus
$\overline{\text{ADS}}$	Address Strobe	Bus cycle control
$\overline{\text{BE3}}$–$\overline{\text{BE0}}$	Byte enables	Address bus
$\overline{\text{BS16}}$	Bus size 16	Bus cycle control
$\overline{\text{BUSY}}$	Processor extension busy	Coprocessor interface
CLK2	2X clock input	
D31–D0	Data bus	
D/$\overline{\text{C}}$	Data/control	Bus cycle definition
$\overline{\text{ERROR}}$	Processor extension error	Coprocessor interface
$\overline{\text{FLUSH}}$	Cache flush	Internal cache interface
HLDA	Hold acknowledge	Bus arbitration
HOLD	Hold request	Bus arbitration
INTR	Maskable interrupt request	Interrupt control
$\overline{\text{KEN}}$	Cache enable	Internal Cache interface
$\overline{\text{LOCK}}$	Bus lock	Bus cycle definition
M/$\overline{\text{IO}}$	Memory/input-output	Bus cycle definition
$\overline{\text{NA}}$	Next address request	Bus cycle control
NMI	Non-maskable interrupt request	Interrupt control
PEREQ	Processor extension request	Coprocessor interface
$\overline{\text{READY}}$	Bus ready	Bus cycle control
RESET	Reset	
$\overline{\text{SMADS}}$	SMM address strobe	Bus cycle control
$\overline{\text{SMI}}$	System management interrupt	Interrupt control
$\overline{\text{SUSP}}$	Suspend request	Power management
$\overline{\text{SUSPA}}$	Suspend acknowledge	Power management
W/$\overline{\text{R}}$	Write/read	Bus cycle definition

The following sections describe the signals and their functional timing characteristics. Additional signal information may be found in Chapter 5, Electrical Specifications. Chapter 5 documents the dc and ac characteristics for the signals including voltage levels, propagation delays, setup times, and hold times. Specified setup and hold times must be met for proper operation of the TI486DLC/E.

Table 4–2. Terminal Functions

PIN		I/O	DESCRIPTION
NAME	NO.		
A2	C4	O/Z	**Address Bus** (active high). The address bus (A31–A2) signals are 3-state outputs that provide addresses for physical memory and I/O ports. All address lines can be used for addressing physical memory allowing a 16 MByte address space (0000 0000h to FFFF FFFFh). During I/O port accesses, A31–A16 are driven low (except for coprocessor accesses). This permits a 64 KByte I/O address space (0000 0000h to 0000 FFFFh). During all coprocessor I/O access address lines A30–A16 are driven low and A31 is driven high. This allows A31 to be used by external logic to generate a coprocessor select signal. Coprocessor command transfers occur with address 8000 00F8h and coprocessor data transfers occur with address 8000 00FCh. A31–A2 float while the CPU is in a hold acknowledge state.
A3	A3		
A4	B3		
A5	B2		
A6	C3		
A7	C2		
A8	C1		
A9	D3		
A10	D2		
A11	D1		
A12	E3		
A13	E2		
A14	E1		
A15	F1		
A16	G1		
A17	H1		
A18	H2		
A19	H3		
A20	J1		
A21	K1		
A22	K2		
A23	L1		
A24	L2		
A25	K3		
A26	M1		
A27	N1		
A28	L3		
A29	M2		
A30	P1		
A31	N2		
$\overline{\mathrm{ADS}}$	E14	O/Z	**Address Strobe** (active low). This is a 3-state output that indicates the TI486DLC/E has driven a valid address (A31–A2, $\overline{\mathrm{BE3}}$–$\overline{\mathrm{BE0}}$) and bus cycle definition (M/$\overline{\mathrm{IO}}$), D/$\overline{\mathrm{C}}$, W/$\overline{\mathrm{R}}$) on the appropriate TI486DLC/E output pins. During non-pipelined bus cycles, $\overline{\mathrm{ADS}}$ is active for the first clock of the bus cycle. During address pipelining, $\overline{\mathrm{ADS}}$ is asserted during the previous bus cycle and remains asserted until $\overline{\mathrm{READY}}$ is returned for that cycle. $\overline{\mathrm{ADS}}$ floats while the TI486DLC/E is in a hold acknowledge state.
$\overline{\mathrm{A20M}}$	F13	I	**Address Bit 20 Mask** (active low). This input causes the TI486DLC/E to mask (force low) physical address bit 20 when driving the external address bus or performing an internal cache access. When the processor is in real mode, asserting $\overline{\mathrm{A20M}}$ emulates the 1 MByte address wrap around that occurs on the 8086. The A20 signal is never masked when paging is enabled regardless of the state of the $\overline{\mathrm{A20M}}$ input. The $\overline{\mathrm{A20M}}$ input is ignored following reset and can be enabled using the A20M bit in the CCR0 configuration register. $\overline{\mathrm{A20M}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\mathrm{BE3}}$	A13	O/Z	**Byte Enables $\overline{\mathbf{BE3}}$–$\overline{\mathbf{BE0}}$** (active low). These are 3-state outputs that determine which bytes within the 32-bit data bus will be transferred during a memory or I/O access (Table 4–3). During a memory write, one or both of the upper bytes (D and C) of the data bus may be duplicated in the lower bytes (B and A) of the bus. This duplication is dependent on $\overline{\mathrm{BE3}}$–$\overline{\mathrm{BE0}}$ as listed in Table 4–4. Generating A1–A0 using $\overline{\mathrm{BE3}}$–$\overline{\mathrm{BE0}}$ can be achieved by using the following equations: $A0 = (\overline{\mathrm{BE0}} \bullet \overline{\mathrm{BE2}}) + (\overline{\mathrm{BE0}} \bullet \overline{\mathrm{BE1}})$ $A1 = \overline{\mathrm{BE0}} \bullet \overline{\mathrm{BE1}}$ The relationship between A1–A0 and $\overline{\mathrm{BE3}}$–$\overline{\mathrm{BE0}}$ is shown in Table 4–5.
$\overline{\mathrm{BE2}}$	B13		
$\overline{\mathrm{BE1}}$	C13		
$\overline{\mathrm{BE0}}$	E12		

Table 4–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
$\overline{\text{BS16}}$	C14	I	**Bus Size 16** (active low). This is an input that allows connection of the 32-bit TI486DLC/E data bus to an external 16-bit data bus. When this input is activated, the microprocessor performs multiple bus cycles to couple read and write accesses from devices that cannot provide (accept) 32 bits of data in a single cycle. During bus cycles with $\overline{\text{BS16}}$ active, data is transferred using data bus signals D15–D0 only.
$\overline{\text{BUSY}}$	B9	I	**Coprocessor Busy** (active low). This is an input from the coprocessor that indicates to the TI486DLC/E that the coprocessor is currently executing an instruction and is not yet able to accept another opcode. When the TI486DLC/E processor encounters a WAIT instruction or any coprocessor instruction that operates on the coprocessor stack (i.e., load, pop, arithmetic operation), $\overline{\text{BUSY}}$ is sampled. $\overline{\text{BUSY}}$ is continually sampled and must be recognized as inactive before the CPU will supply the coprocessor with another instruction. However, the following coprocessor instructions are allowed to execute even if $\overline{\text{BUSY}}$ is active since these instructions are used for coprocessor initialization and exception clearing: FNINIT, FNCLEX. $\overline{\text{BUSY}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
CLK2	F12	I	**2X Clock Input** (active high). This signal is the basic timing reference for the TI486DLC/E microprocessor. The CLK2 input is internally divided by two to generate the internal processor clock. The external CLK2 is synchronized to a known phase of the internal processor clock by the falling edge of the RESET signal. External timing parameters are defined with respect to the rising edge of CLK2.
D0	H12	I/O/Z	**Data Bus** (active high). The Data Bus (D31–D0) signals are 3-state bidirectional signals that provide the data path between the TI486DLC/E and external memory and I/O devices. The data bus inputs data during memory read, I/O read and interrupt acknowledge cycles and outputs data during memory and I/O write cycles. Data read operations require that specified data setup and hold times be met for correct operation. The data bus signals are high active and float while the CPU is in a hold acknowledge or float state.
D1	H13		
D2	H14		
D3	J14		
D4	K14		
D5	K13		
D6	L14		
D7	K12		
D8	L13		
D9	N14		
D10	M12		
D11	N13		
D12	N12		
D13	P13		
D14	P12		
D15	M11		
D16	N11		
D17	N10		
D18	P11		
D19	P10		
D20	M9		
D21	N9		
D22	P9		
D23	N8		
D24	P7		
D25	N6		
D26	P5		
D27	N5		
D28	M6		
D29	P4		
D30	P3		
D31	M5		

Table 4–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
D/$\overline{\text{C}}$	A11	O/Z	**Data/Control**. This signal is low during control cycles and is high during data cycles. Control cycles are issued during functions such as a halt instruction, interrupt servicing and code fetching. Data bus cycles include data access from either memory or I/O.
$\overline{\text{ERROR}}$	A8	I	**Coprocessor Error** (active low). This is an input used to indicate that the coprocessor generated an error during execution of a coprocessor instruction. $\overline{\text{ERROR}}$ is sampled by the TI486DLC/E processor whenever a coprocessor instruction is executed. If $\overline{\text{ERROR}}$ is sampled active, the processor generates exception 16 which is then serviced by the exception handling software. Certain coprocessor instructions do not generate an exception 16 even if $\overline{\text{ERROR}}$ is active. These instructions, which involve clearing coprocessor error flags and saving the coprocessor state, are listed as follows: FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV, FNSAVE. $\overline{\text{ERROR}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected. $\overline{\text{ERROR}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\text{FLUSH}}$	E13	I	**Cache Flush** (active low). This is an input that invalidates (flushes) the entire cache. Use of $\overline{\text{FLUSH}}$ to maintain cache coherency is optional. The cache may also be invalidated during each hold acknowledge cycle by setting the BARB bit in the CCR0 configuration register. The $\overline{\text{FLUSH}}$ input is ignored following reset and can be enabled using the FLUSH bit in the CCR0 configuration register. $\overline{\text{FLUSH}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
HLDA	M14	O	**Hold Acknowledge** (active high). This output indicates that the TI486DLC/E is in a hold acknowledge state and has relinquished control of its local bus. While in the hold acknowledge state, the TI486DLC/E drives HLDA active and continues to drive $\overline{\text{SUSPA}}$, if enabled. The other TI486DLC/E outputs are in a high-impedance state allowing the requesting bus master to drive these signals. If the on-chip cache can satisfy bus requests, the TI486DLC/E continues to operate during hold acknowledge states. $\overline{\text{A20M}}$ is internally recognized during this time. The processor deactivates HLDA when the HOLD request is driven inactive. The TI486DLC/E stores on NMI rising edge during a hold acknowledge state for processing after HOLD is inactive. The $\overline{\text{FLUSH}}$ input is also recognized during a hold acknowledge state. If $\overline{\text{SUSP}}$ is asserted during a hold acknowledge state, the TI486DLC/E may or may not enter suspend mode depending on the state of the internal execution pipeline. Table 3–3 summarizes the state of the TI486DLC/E signals during hold acknowledge.

Table 4–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
HOLD	D14	I	**Hold Request** (active high). This input is used to indicate that another bus master requests control of the local bus. The bus arbitration (HOLD, HLDA) signals allow the TI486DLC/E to relinquish control of its local bus when requested by another bus master device. Once the processor has relinquished its bus (3-stated), the bus master device can then drive the local bus signals. After recognizing the HOLD request and completing the current bus cycle or sequence of locked bus cycles, the TI486DLC/E responds by floating the local bus and asserting the hold acknowledge (HLDA) output. Once HLDA is asserted, the bus remains granted to the requesting bus master until HOLD becomes inactive. When the TI486DLC/E recognizes HOLD is inactive, it simultaneously drives the local bus and drives HLDA inactive. External pullup resistors may be required on some of the TI486DLC/E 3-state outputs to guarantee that they remain inactive while in a hold acknowledge state. The HOLD input is not recognized while RESET is active. If HOLD is asserted while RESET is active, RESET has priority and the TI486DLC/E places the bus into an idle state instead of a hold acknowledge state. The HOLD input is also recognized during suspend mode provided that the CLK2 input has not been stopped. HOLD is level-sensitive and must meet specified setup and hold times for correct operation.
INTR	B7	I	**Maskable Interrupt Request.** This is a level-sensitive input that causes the processor to suspend execution of the current instruction stream and begin execution of an interrupt service routine. The INTR input can be masked (ignored) through the Flags Register IF bit. When unmasked, the TI486DLC/E responds to the INTR input by issuing two locked interrupt acknowledge cycles. To assure recognition of the INTR request, INTR must remain active until the start of the first interrupt acknowledge cycle.
$\overline{\mathrm{KEN}}$	B12	I	**Cache Enable** (active low). This is an input which indicates that the data being returned during the current cycle is cacheable. When $\overline{\mathrm{KEN}}$ is active and the TI486DLC/E is performing a cacheable code fetch or memory data read cycle, the cycle is transformed into a cache fill. Use of the $\overline{\mathrm{KEN}}$ input to control cacheability is optional. The non-cacheable region registers can also be used to control cacheablity. Memory addresses specified by the non-cacheable region registers are not cacheable regardless of the state of $\overline{\mathrm{KEN}}$. I/O accesses, locked reads, SMM address space accesses, and interrupt acknowledge cycles are never cached. During cached code fetches, two contiguous read cycles are performed to completely fill the 4-byte cache line. $\overline{\mathrm{KEN}}$ must be asserted during both read cycles in order to cause a cache line fill. During cached data reads, the TI486DLC/E performs only those bus cycles necessary to supply the required data to complete the current operation. Valid bits are maintained for each byte in the cache line, thus allowing data operands of less than 4 bytes to reside in the cache. During any cache fill cycle with $\overline{\mathrm{KEN}}$ asserted, the TI486DLC/E ignores the state of the byte enables ($\overline{\mathrm{BE3}}$ – $\overline{\mathrm{BE0}}$) and always writes two bytes of data into the cache. The $\overline{\mathrm{KEN}}$ input is ignored following reset and can be enabled using the KEN bit in the CCR0 configuration register. $\overline{\mathrm{KEN}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\mathrm{LOCK}}$	C10	I	**LOCK** (active low). $\overline{\mathrm{LOCK}}$ is asserted to deny control of the CPU bus to other bus masters. The $\overline{\mathrm{LOCK}}$ signal may be explicitly activated during bus operations by including the LOCK prefix on certain instructions. $\overline{\mathrm{LOCK}}$ is always asserted during descriptor and page table updates, interrupt acknowledge sequences and when executing the XCHG instruction. The TI486DLC/E does not enter the hold acknowledge state in response to HOLD while the $\overline{\mathrm{LOCK}}$ input is active.
$\mathrm{M}/\overline{\mathrm{IO}}$	A12	O/Z	**Memory/IO.** This signal is low during I/O read and write cycles and is high during memory cycles.

Table 4–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
$\overline{\text{NA}}$	D13	I	**Next Address Request** (active low). This is an input used to request address pipelining by the system hardware. When asserted, the system indicates that it is prepared to accept new bus cycle definition and address signals (M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$, W/$\overline{\text{R}}$, A31–A2, $\overline{\text{BS16}}$, and $\overline{\text{BE3}}$–$\overline{\text{BE0}}$) from the microprocessor even if the current bus cycle has not been terminated by assertion of $\overline{\text{READY}}$. If the TI486DLC/E has an internal bus request pending and the $\overline{\text{NA}}$ input is sampled active, the next bus cycle definition and address signals are driven onto the bus.
N/C	B6	—	No connection. Should be left disconnected.
NMI	B8	I	**Non-maskable Interrupt Request**. This is a rising-edge-sensitive input that causes the processor to suspend execution of the current instruction stream and begin execution of an NMI interrupt service routine. The NMI interrupt service request cannot be masked by software. Asserting NMI causes an interrupt which internally supplies interrupt vector 2h to the CPU core. External interrupt acknowledge cycles are not necessary since the NMI interrupt vector is supplied internally. The TI486DLC/E samples NMI at the beginning of each phase 2. To assure recognition, NMI must be inactive for at least eight CLK2 periods and then be active for at least eight CLK2 periods. Additionally, specified setup and hold times must be met to guarantee recognition at a particular clock edge.
PEREQ	C8	I	**Coprocessor Request** (active high). This is an input that indicates the coprocessor is ready to transfer data to or from the CPU. The coprocessor may assert PEREQ in the process of executing a coprocessor instruction. The TI486DLC/E internally stores the current coprocessor opcode and performs the correct data transfers to support coprocessor operations using PEREQ to synchronize the transfer of required operands. PEREQ is internally connected to a pulldown resistor to prevent this signal from floating active when left unconnected.
$\overline{\text{READY}}$	G13	I	**Ready**. This is an input generated by the system hardware that indicates the current bus cycle can be terminated. During a read cycle, assertion of $\overline{\text{READY}}$ indicates that the system hardware has presented valid data to the CPU. When $\overline{\text{READY}}$ is sampled active, the TI486DLC/E latches the input data and terminates the cycle. During a write cycle, $\overline{\text{READY}}$ assertion indicates that the system hardware has accepted the TI486DLC/E output data. $\overline{\text{READY}}$ must be asserted to terminate every bus cycle, including halt and shutdown indication cycles.
RESET	C9	I	**Reset** (active high). When asserted, RESET suspends all operations in progress and places the TI486DLC/E into a reset state. RESET is a level-sensitive synchronous input and must meet specified setup and hold times to be properly recognized by the TI486DLC/E. The TI486DLC/E begins executing instructions at physical address location FF FFF0h approximately 400 CLK2s after RESET is driven inactive (low). While RESET is active all other input pins are ignored. The remaining signals are initialized to their reset state during the internal processor reset sequence. The reset signal states for the TI486DLC/E are shown in Table 4–6.
$\overline{\text{SMADS}}$	C6	O/Z	**SMM Address Strobe** (active low). $\overline{\text{SMADS}}$ is asserted instead of the $\overline{\text{ADS}}$ during SMM bus cycles and indicates that SMM memory is being accessed. $\overline{\text{SMADS}}$ floats while the CPU is in a hold acknowledge or float state. The $\overline{\text{SMADS}}$ output is disabled (floated) following reset and can be enabled using the SMI bit in the CCR1 configuration register.
$\overline{\text{SMI}}$	C7	I/O	**System Management Interrupt** (active low). This is a bidirectional signal and level sensitive interrupt with higher priority than the NMI interrupt. $\overline{\text{SMI}}$ must be active for at least four CLK2 clock periods to be recognized by the TI486DLC/E. After the $\overline{\text{SMI}}$ interrupt is acknowledged, the $\overline{\text{SMI}}$ pin is driven low by the TI486DLC/E for the duration of the SMI service routine. The $\overline{\text{SMI}}$ input is ignored following reset and can be enabled using the SMI bit in the CCR1 configuration register. $\overline{\text{SMI}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.

Table 4–2. Terminal Functions (Continued)

PIN		I/O	DESCRIPTION
NAME	NO.		
$\overline{\text{SUSP}}$	A4	I	**Suspend Request** (active low). This is an input that requests the TI486DLC/E enter suspend mode. After recognizing $\overline{\text{SUSP}}$ active, the processor completes execution of the current instruction, any pending decoded instructions and associated bus cycles. In addition, the TI486DLC/E waits for the coprocessor to indicate a not busy status ($\overline{\text{BUSY}}$ = 1) before entering suspend mode and asserting suspend acknowledge ($\overline{\text{SUSPA}}$). $\overline{\text{SUSP}}$ is internally connected to a pullup resistor to prevent it from floating active when left unconnected.
$\overline{\text{SUSPA}}$	B4	O	**Suspend Acknowledge** (active low). This output indicates that the TI486DLC/E has entered the suspend mode as a result of $\overline{\text{SUSP}}$ assertion or execution of a HALT instruction.
V_{CC}	A1 A5 A7 A10 A14 C5 C12 D12 G2 G3 G12 G14 L12 M3 M7 M13 N4 N7 P2 P8	I	**5-V Power Supply**. All pins must be connected and used.
V_{SS}	A2 A6 A9 B1 B5 B11 B14 C11 F2 F3 F14 J2 J3 J12 J13 M4 M8 M10 N3 P6 P14	I	**Ground Pins**. All pins must be connected and used.
W/$\overline{\text{R}}$	B10	O/Z	**Write/Read**. W/$\overline{\text{R}}$ is low during read cycles (data is read from memory or I/O) and is high during write bus cycles (data is written to memory or I/O).

Table 4–3. Byte Enable Line Definitions

BYTE ENABLE LINE	BYTE TRANSFERRED
$\overline{\text{BE0}}$	D7–D0
$\overline{\text{BE1}}$	D15–D8
$\overline{\text{BE2}}$	D23–D16
$\overline{\text{BE3}}$	D31–D24

Table 4–4. Write Duplication as a Function of $\overline{\text{BE3}}$–$\overline{\text{BE0}}$

$\overline{\text{BE3}}$–$\overline{\text{BE0}}$	D31–D24	D23–D16	D15–D8	D7–D0	DUPLICATED DATA
0000	D	C	B	A	no
0001	D	C	B	x	no
0011	D	C	D	C	yes
0111	D	x	D	x	yes
1000	x	C	B	A	no
1001	x	C	B	x	no
1011	x	C	x	C	yes
1100	x	x	B	A	no
1101	x	x	B	x	no
1110	x	x	x	A	no

NOTE: $\overline{\text{BE3}}$ – $\overline{\text{BE0}}$ combinations not listed, do not occur during TI486DLC/E bus cycles.
A = logical write data D7 – D0.
B = logical write data D15 – D8.
C = logical write data D23 – D16.
D = logical write data D31 – D24.
x = not defined.

Table 4–5. Generating A1–A0 Using $\overline{\text{BE3}}$–$\overline{\text{BE0}}$

A31–A2	A1	A0	$\overline{\text{BE3}}$	$\overline{\text{BE2}}$	$\overline{\text{BE1}}$	$\overline{\text{BE0}}$
———	0	0	x	x	x	0
———	0	1	x	x	0	1
———	1	0	x	0	1	1
———	1	1	0	1	1	1

Table 4–6. Signal States During RESET and Hold Acknowledge

SIGNAL NAME	SIGNAL STATE DURING RESET	SIGNAL STATE DURING HOLD ACKNOWLEDGE
$\overline{\text{A20M}}$	Ignored	Input recognized
A31–A2	1	Float
$\overline{\text{ADS}}$	1	Float
$\overline{\text{BE3}}$–$\overline{\text{BE0}}$	0	Float
$\overline{\text{BS16}}$	Ignored	Ignored
$\overline{\text{BUSY}}$	Initiates self test	Ignored
D31–D0	Float	Float
D/$\overline{\text{C}}$	1	Float
$\overline{\text{ERROR}}$	Ignored	Ignored
$\overline{\text{FLUSH}}$	Ignored	Input recognized
HLDA	0	1
HOLD	Ignored	Input recognized
INTR	Ignored	Input recognized
$\overline{\text{KEN}}$	Ignored	Ignored
$\overline{\text{LOCK}}$	1	Float
M/$\overline{\text{IO}}$	0	Float
$\overline{\text{NA}}$	Ignored	Ignored
NMI	Ignored	Input recognized
PEREQ	Ignored	Ignored
$\overline{\text{READY}}$	Ignored	Ignored
RESET	Input recognized	Input recognized
$\overline{\text{SMADS}}$	Float	Float
$\overline{\text{SMI}}$	Ignored	Input recognized
$\overline{\text{SUSP}}$	Ignored	Input recognized
$\overline{\text{SUSPA}}$	Float	Driven
W/$\overline{\text{R}}$	0	Float

4.1.1 Bus Cycle Definition

The bus cycle definition (M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$, W/$\overline{\text{R}}$, $\overline{\text{LOCK}}$) signals consist of four 3-state outputs that define the type of bus cycle operation being performed. Table 4–7 defines the bus cycles for the possible states of these signals. M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$ and W/$\overline{\text{R}}$ are the primary bus cycle definition signals and are driven valid as $\overline{\text{ADS}}$ (Address Strobe) becomes active. During non-pipelined cycles, the $\overline{\text{LOCK}}$ output is driven valid along with M/$\overline{\text{IO}}$, D/$\overline{\text{C}}$ and W/$\overline{\text{R}}$. During pipelined addressing, $\overline{\text{LOCK}}$ is driven at the beginning of the bus cycle, which is after $\overline{\text{ADS}}$ becomes active for that cycle. The bus cycle definition signals are active low and float while the TI486DLC/E is in a hold acknowledge or float state.

Table 4–7. Bus Cycle Types

M/$\overline{\text{IO}}$	D/$\overline{\text{C}}$	W/$\overline{\text{R}}$	$\overline{\text{LOCK}}$	BUS CYCLE TYPE
0	0	0	0	Interrupt acknowledge
0	0	0	1	—
0	0	1	X	—
0	1	X	0	—
0	1	0	1	I/O data read
0	1	1	1	I/O data write
1	0	X	0	—
1	0	0	1	Memory code read
1	0	1	1	Halt: A31–A2=0h, $\overline{\text{BE3}}$–$\overline{\text{BE0}}$=1011 Shutdown: A31–A2=0h, $\overline{\text{BE3}}$–$\overline{\text{BE0}}$=1110
1	1	0	0	Locked memory data read
1	1	0	1	Memory data read
1	1	1	0	Locked memory data write
1	1	1	1	Memory data write

X = don't care
— = does not occur

4.1.2 Power Management

The power management signals allow the TI486DLC/E to enter suspend mode. Suspend mode circuitry allows the TI486DLC/E to consume minimal power while maintaining the entire internal CPU state.

4.1.2.1 Suspend Request ($\overline{\text{SUSP}}$)

Suspend Request ($\overline{\text{SUSP}}$) is an active-low input that requests the TI486DLC/E to enter suspend mode. After recognizing $\overline{\text{SUSP}}$ is active, the processor completes execution of the current instruction, any pending decoded instructions and associated bus cycles. In addition, the TI486DLC/E waits for the coprocessor to indicate a not busy condition ($\overline{\text{BUSY}}$=1) before entering suspend mode and asserting suspend acknowledge ($\overline{\text{SUSPA}}$). During suspend mode, internal clocks are stopped and only the logic associated with monitoring RESET, HOLD and $\overline{\text{FLUSH}}$ remains active. With $\overline{\text{SUSPA}}$ asserted, the CLK2 input to the TI486DLC/E can be stopped in either phase. Stopping the CLK2 input further reduces current consumption of the TI486DLC/E.

To resume operation, the CLK2 input is restarted (if stopped), followed by deassertion of the $\overline{\text{SUSP}}$ input. The processor then resumes instruction fetching and begins execution in the instruction stream at the point it had stopped. The $\overline{\text{SUSP}}$ input is level sensitive and must meet specified setup and hold times to be recognized at a particular clock edge. The $\overline{\text{SUSP}}$ input is ignored following reset and can be enabled using the SUSP bit in the CCR0 configuration register.

4.1.2.2 Suspend Acknowledge ($\overline{\text{SUSPA}}$)

The Suspend Acknowledge ($\overline{\text{SUSPA}}$) output indicates that the TI486DLC/E has entered the suspend mode as a result of $\overline{\text{SUSP}}$ assertion or execution of a HALT instruction. If $\overline{\text{SUSPA}}$ is asserted and the CLK2 input is switching, the TI486DLC/E continues to recognize RESET, HOLD, and $\overline{\text{FLUSH}}$. If suspend mode was entered as the result of a HALT instruction, the TI486DLC/E also continues to monitor the NMI input and an unmasked INTR input. Detection of INTR or NMI forces the TI486DLC/E to exit suspend mode and begin execution of the appropriate interrupt service routine. The CLK2 input to the processor may be stopped after $\overline{\text{SUSPA}}$ has been asserted to further reduce the power consumption of the TI486DLC/E. The $\overline{\text{SUSPA}}$ output is disabled (floated) following reset and can be enabled using the SUSP bit in the CCR0 configuration register.

Table 4–8 shows the state of the TI486DLC/E signals when the device is in suspend mode.

Table 4–8. Signal States During Suspend Mode

SIGNAL NAME	SIGNAL STATE DURING HOLD ACKNOWLEDGE	SIGNAL STATE DURING HALT INITIATED SUSPEND MODE
$\overline{\text{A20M}}$	Ignored	Ignored
A31–A2	1	1
$\overline{\text{ADS}}$	1	1
$\overline{\text{BE3}}$–$\overline{\text{BE0}}$	0	0
$\overline{\text{BS16}}$	Ignored	Ignored
$\overline{\text{BUSY}}$	Ignored	Ignored
D31–D0	Float	Float
D/$\overline{\text{C}}$	1	1
$\overline{\text{ERROR}}$	Ignored	Ignored
$\overline{\text{FLUSH}}$	Input recognized	Input recognized
HLDA	0	0
HOLD	Input recognized	Input recognized
INTR	Latched	Input recognized
$\overline{\text{KEN}}$	Ignored	Ignored
$\overline{\text{LOCK}}$	1	1
M/$\overline{\text{IO}}$	0	0
$\overline{\text{NA}}$	Ignored	Ignored
NMI	Latched	Input recognized
PEREQ	Ignored	Ignored
$\overline{\text{READY}}$	Ignored	Ignored
RESET	Input recognized	Input recognized
$\overline{\text{SMADS}}$	1	1
$\overline{\text{SMI}}$	Latched	Input recognized
$\overline{\text{SUSP}}$	Input recognized	Ignored
$\overline{\text{SUSPA}}$	0	0
W/$\overline{\text{R}}$	0	0

4.1.2.3 Coprocessor Interface

The data bus, address bus, and bus cycle definition signals, as well as the coprocessor interface signals (PEREQ, $\overline{\text{BUSY}}$, $\overline{\text{ERROR}}$), are used to control communication between the TI486DLC/E and a coprocessor. Coprocessor or ESC opcodes are decoded by the TI486DLC/E and the opcode and operands are then transferred to the coprocessor via I/O port accesses to addresses 8000 00F8h and 8000 00FCh. Address 8000 00F8h functions as the control port address and 8000 00FCh is used for operand transfers.

4.2 Functional Timing

4.2.1 Reset Timing and Internal Clock Synchronization

RESET is the highest priority input signal and is capable of interrupting any processor activity when it is asserted. When RESET is asserted, the TI486DLC/E aborts any bus cycle. Idle, hold acknowledge, and suspend states are also discontinued and the reset state is established. RESET is used when the TI486DLC/E microprocessor is powered up to initialize the CPU to a known valid state and to synchronize the internal CPU clock with external clocks.

RESET must be asserted for at least 15 CLK2 periods to ensure recognition by the TI486DLC/E microprocessor. If the self-test feature is to be invoked, RESET must be asserted for at least 80 CLK2 periods. RESET pulses less than 15 CLK2 periods may not have sufficient time to propagate throughout the TI486DLC/E and may not be recognized. RESET pulses less than 80 CLK2 periods followed by a self-test request may incorrectly report a self-test failure when no true failure exists.

Provided the RESET falling edge meets specified setup and hold times, the internal processor clock phase is synchronized as illustrated in Figure 4–2. The internal processor clock is half the frequency of the CLK2 input and each CLK2 cycle corresponds to an internal CPU clock phase. Phase 2 of the internal clock is defined to be the second rising edge of CLK2 following the falling edge of RESET.

Following the falling edge of REST (and after self-test if it was requested), the TI486DLC/E microprocessor performs an internal initialization sequence for approximately 400 CLK2 periods. The TI486DLC/E self-test feature is invoked if the $\overline{\text{BUSY}}$ input is in an active-low state when RESET falls inactive. The self-test sequence requires approximately (2^{20} + 60) CLK2 periods to complete. Even if the self-test indicates a problem, the TI486DLC/E microprocessor attempts to proceed with the reset sequence. Figure 4–3 illustrates the bus activity and timing during the TI486DLC/E reset sequence.

Upon completion of self-test, the EAX register contains 0000 0000h if the TI486DLC/E microprocessor passed its internal self-test with no problems detected. Any non-zero value in the EAX register indicates that the microprocessor is faulty.

Figure 4–2. Internal Processor Clock Synchronization

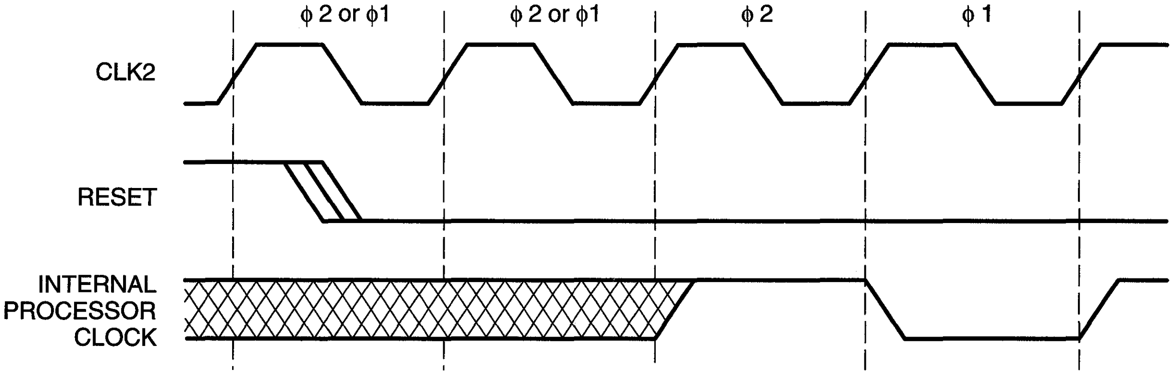
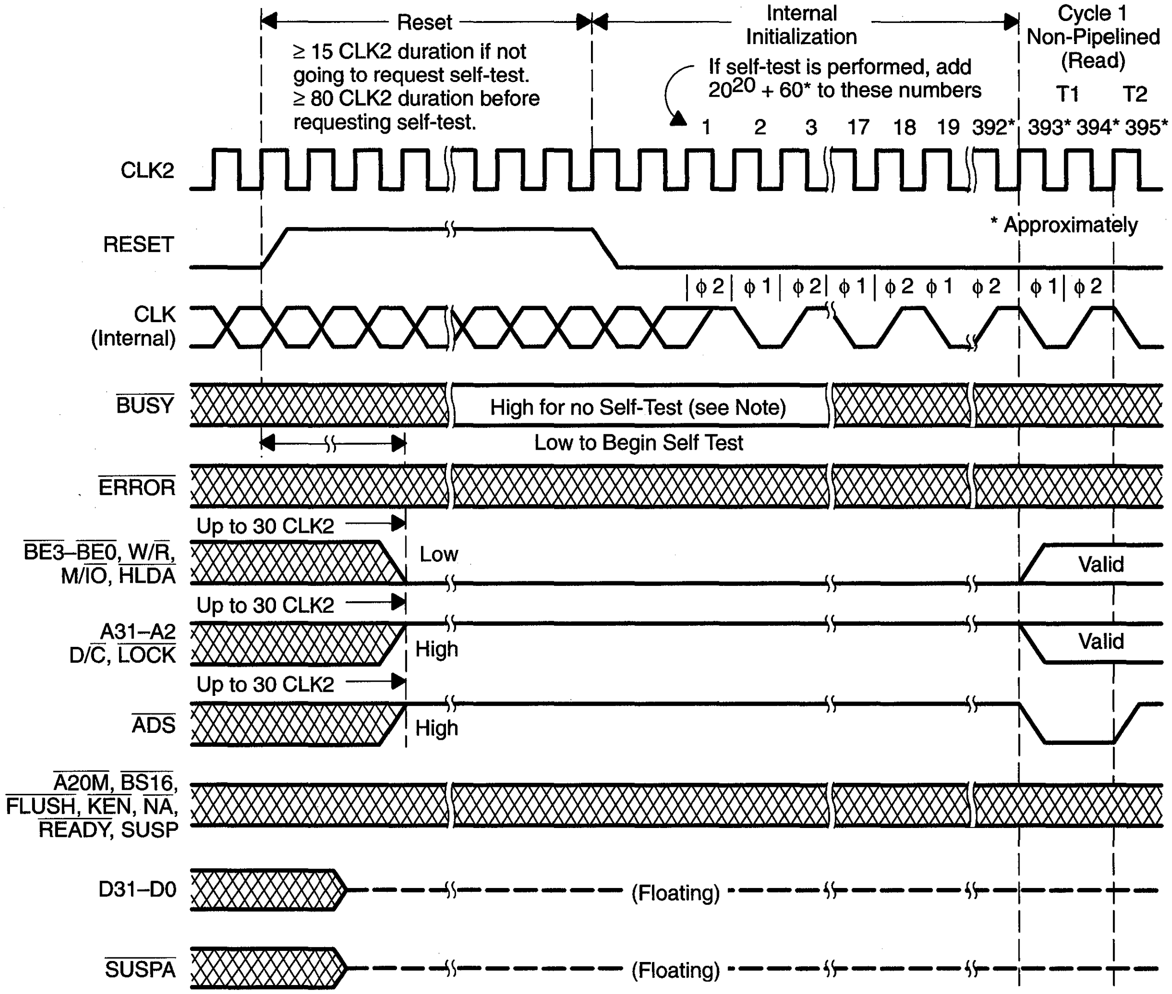


Figure 4–3. Bus Activity from RESET until First Code Fetch


Reset
≥ 15 CLK2 duration if not going to request self-test.
≥ 80 CLK2 duration before requesting self-test.
Internal Initialization
If self-test is performed, add 2^{20} + 60* to these numbers
1 2 3 17 18 19 392*
Cycle 1 Non-Pipelined (Read)
T1 T2
393* 394* 395*
CLK2
RESET
* Approximately
CLK (Internal)
ϕ2 ϕ1 ϕ2 ϕ1 ϕ2 ϕ1 ϕ2 ϕ1 ϕ2
BUSY
High for no Self-Test (see Note)
Low to Begin Self Test
ERROR
Up to 30 CLK2
BE3–BE0, W/R, M/IO, HLDA
Low
Valid
Up to 30 CLK2
A31–A2 D/C, LOCK
High
Valid
Up to 30 CLK2
ADS
High
A20M, BS16, FLUSH, KEN, NA, READY, SUSP
D31–D0
(Floating)
SUSPA
(Floating)

Note: BUSY should be held stable for 80 CLK2 periods before and after the CLK2 period in which RESET falling edge occurs.

4.2.2 Bus Operation

The TI486DLC/E microprocessor communicates with the external system through separate, parallel buses for data and address. This is commonly called a demultiplexed address/data bus. This demultiplexed bus eliminates the need for address latches required in multiplexed address/data bus configurations where the address and data are presented on the same pins at different times.

TI486DLC/E instructions can act on memory data operands consisting of 8-bit bytes, 16-bit words or 32-bit double words. The TI486DLC/E bus architecture allows for bus transfers of these operands without restrictions on physical address alignment. Any byte boundary may require more than one bus cycle to transfer the operand. This feature is transparent to the programmer.

The TI486DLC/E data bus (D31–D0) is a bidirectional bus that can be configured as either a 16-bit or 32-bit wide bus as determined by $\overline{\text{BS16}}$. The bus is 16 bits wide when $\overline{\text{BS16}}$ is asserted. When 32 bits wide, memory and I/O spaces are physically addressed as arrays of 32-bit double words. The TI486DLC/E drives the data bus during write bus cycles, and the external system hardware drives the data bus during read bus cycles.

Every bus cycle begins with the assertion of the address strobe ($\overline{\text{ADS}}$). $\overline{\text{ADS}}$ indicates that the TI486DLC/E has issued a new address and new bus cycle definition signals. A bus cycle is defined by four signals: M/$\overline{\text{IO}}$, W/$\overline{\text{R}}$, D/$\overline{\text{C}}$ and $\overline{\text{LOCK}}$. M/$\overline{\text{IO}}$ defines if a memory or I/O operation is occurring, W/$\overline{\text{R}}$ defines the cycle to be read or write, and D/$\overline{\text{C}}$ indicates whether a data or control cycle is in effect. $\overline{\text{LOCK}}$ indicates that the current cycle is a locked bus cycle. Every bus cycle completes when the system hardware returns $\overline{\text{READY}}$ asserted.

The TI486DLC/E performs the following bus cycle types:

- Memory read
- Locked memory read
- Memory write
- Locked memory write
- I/O read (or coprocessor read)
- I/O write (or coprocessor write)
- Interrupt acknowledge (always locked)
- Halt/shutdown

When the TI486DLC/E microprocessor has no pending bus requests, the bus enters the idle state. There is no encoding of the idle state on the bus cycle definition signals; however, the idle state can be identified by the absence of further assertions of $\overline{\text{ADS}}$ following a completed bus cycle.

4.2.2.1 Bus Cycles Using Non-Pipelined Addressing

Non-Pipelined Bus States

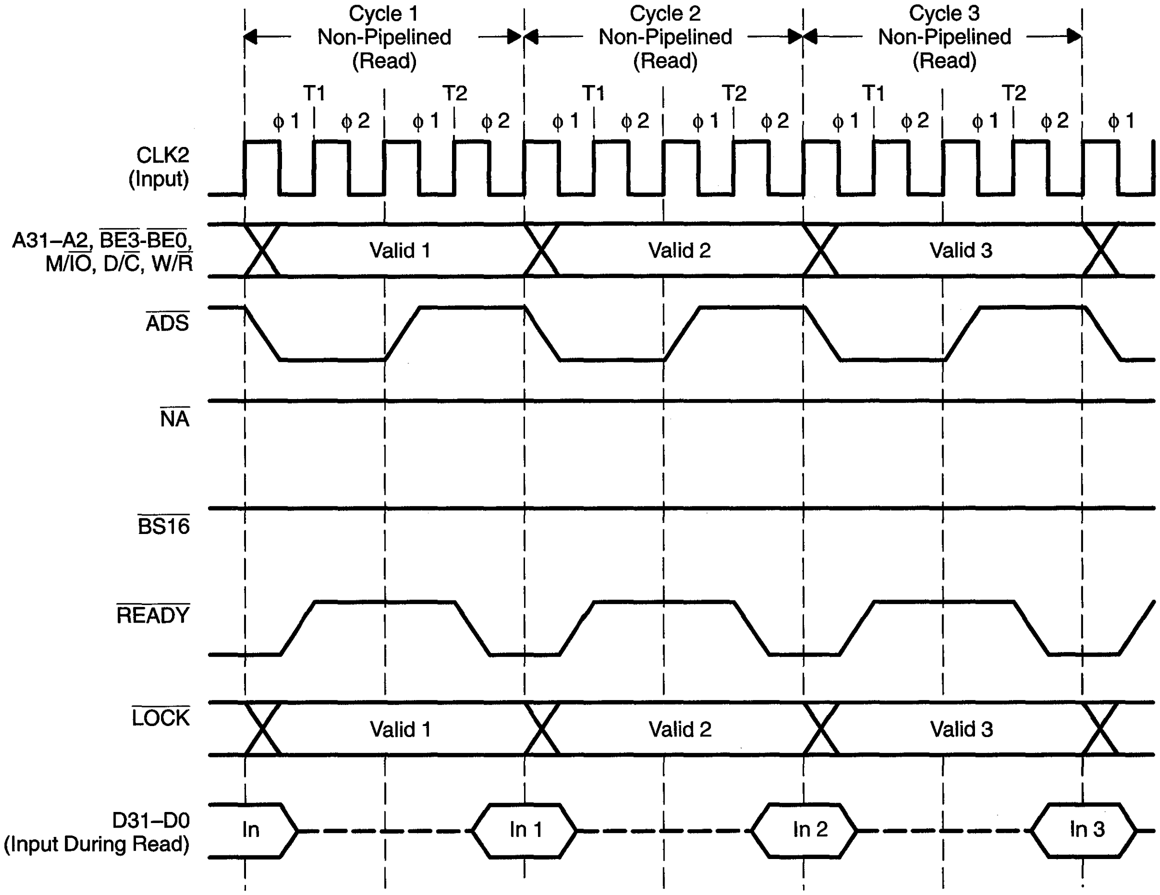
The shortest time unit of bus activity is a bus state, commonly called a T state. A bus state is one internal processor clock period (two CLK2 periods) in duration. A complete data transfer occurs during a bus cycle, composed of two or more bus states.

The first state of a non-pipelined bus cycle is called T1. During phase one (first CLK2) of T1, the address bus and bus cycle definition signals are driven valid and, to signal their availability, address strobe (ADS) is simultaneously asserted.

The second bus state of a non-pipelined cycle is called T2. T2 terminates a bus cycle with the assertion of the READY input and valid data is either input or output depending on the bus cycle type. The fastest TI486DLC/E microprocessor bus cycle requires only these two bus states. READY is ignored at the end of the T1 state.

Three consecutive bus read cycles, each consisting of two bus states, are shown in Figure 4–4.

Figure 4–4. Fastest Non-Pipelined Read Cycles


Cycle 1 Non-Pipelined (Read)
Cycle 2 Non-Pipelined (Read)
Cycle 3 Non-Pipelined (Read)
T1
T2
ϕ1
ϕ2
CLK2 (Input)
A31–A2, BE3-BE0, M/IO, D/C, W/R
Valid 1
Valid 2
Valid 3
ADS
NA
BS16
READY
LOCK
D31–D0 (Input During Read)
In
In 1
In 2
In 3

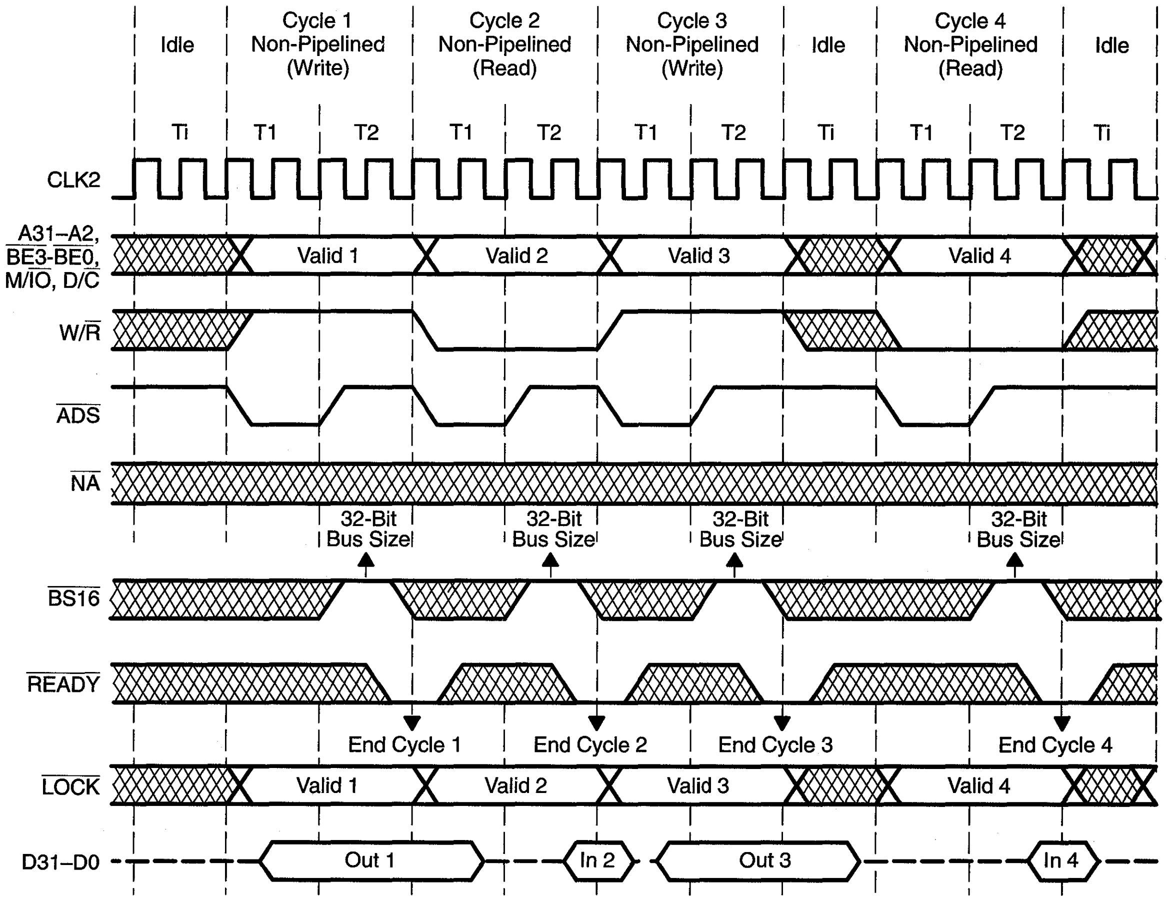
Note: Fastest non-pipelined bus cycles consist of T1 and T2.

Non-Pipelined Read and Write Cycles

Any bus cycle may be performed with non-pipelined address timing. Figure 4–5 shows a mixture of read and write cycles with non-pipelined address timing. When a read cycle is performed, the TI486DLC/E microprocessor floats its data bus and the externally addressed device then drives the data. The TI486DLC/E microprocessor requires that all data bus pins be driven to a valid logic state (high or low) at the end of each read cycle, when $\overline{\text{READY}}$ is asserted. When a read cycle is acknowledged by $\overline{\text{READY}}$ asserted in the T2 bus state, the TI486DLC/E CPU latches the information present at its data pins and terminates the cycle.

When a write cycle is performed, the data bus is driven by the TI486DLC/E CPU beginning in phase two of T1. When a write cycle is acknowledged, the TI486DLC/E write data remains valid throughout phase one of the next bus state to provide write data hold time.

Figure 4–5. Various Non-Pipelined Bus Cycles (No Wait States)



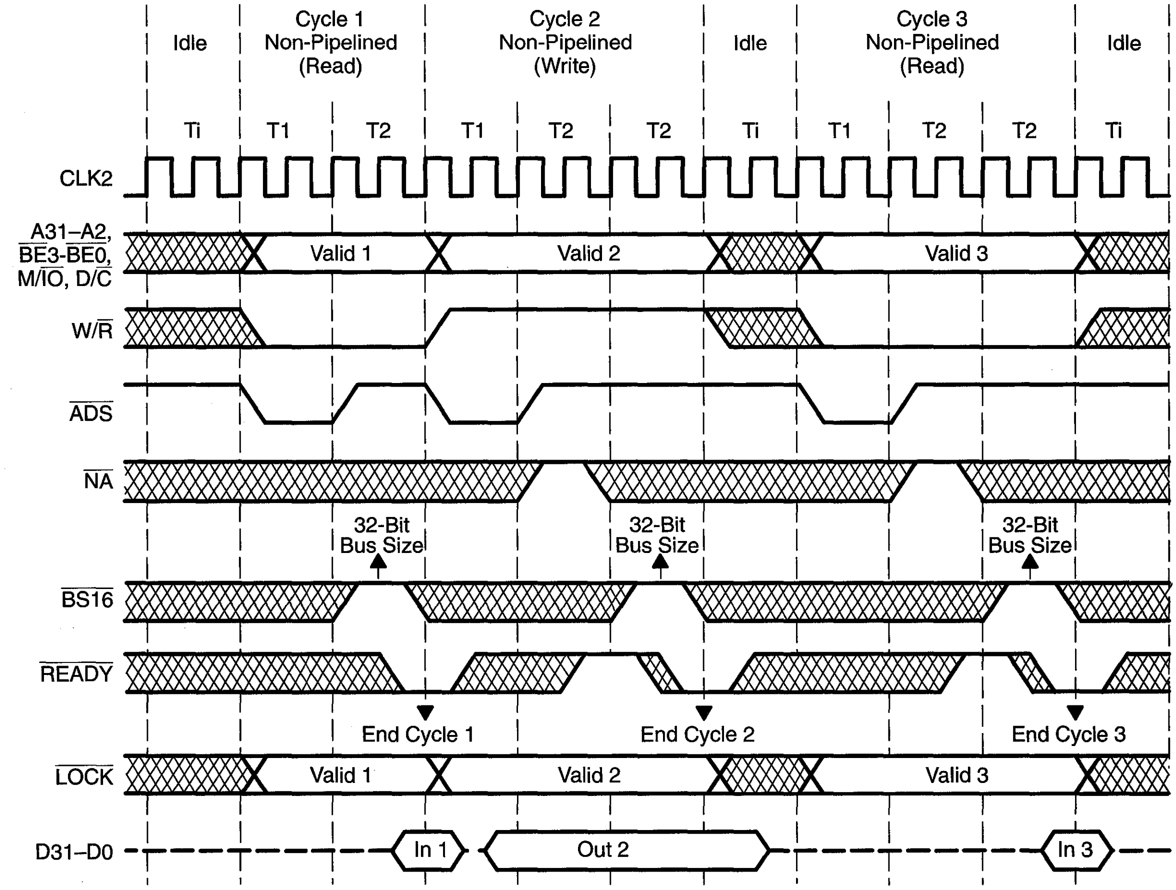
Note: Idle states are shown here for diagram variety only.

Non-Pipelined Wait States

Once a bus cycle begins, it continues until acknowledged by the external system hardware using the TI486DLC/E $\overline{\text{READY}}$ input. Acknowledging the bus cycle at the end of the first T2 results in the shortest possible bus cycle, requiring only T1 and T2. If $\overline{\text{READY}}$ is not immediately asserted however, T2 states are repeated indefinitely until the $\overline{\text{READY}}$ input is sampled active. These intermediate T2 states are referred to as wait states. If the external system hardware is not able to receive or deliver data in two bus states, it withholds the $\overline{\text{READY}}$ signal and at least one wait state is added to the bus cycle. Thus, on an address-by-address basis the system is able to define how fast a bus cycle completes.

Figure 4–6 illustrates non-pipelined bus cycles with one wait state added to cycles 2 and 3. $\overline{\text{READY}}$ is sampled inactive at the end of the first T2 state in cycles 2 and 3. Therefore, the T2 state is repeated until $\overline{\text{READY}}$ is sampled active at the end of the second T2 and the cycle is then terminated. The TI486DLC/E ignores the $\overline{\text{READY}}$ input at the end of the T1 state.

Figure 4–6. Various Non-Pipelined Bus Cycles with Different Numbers of Wait States

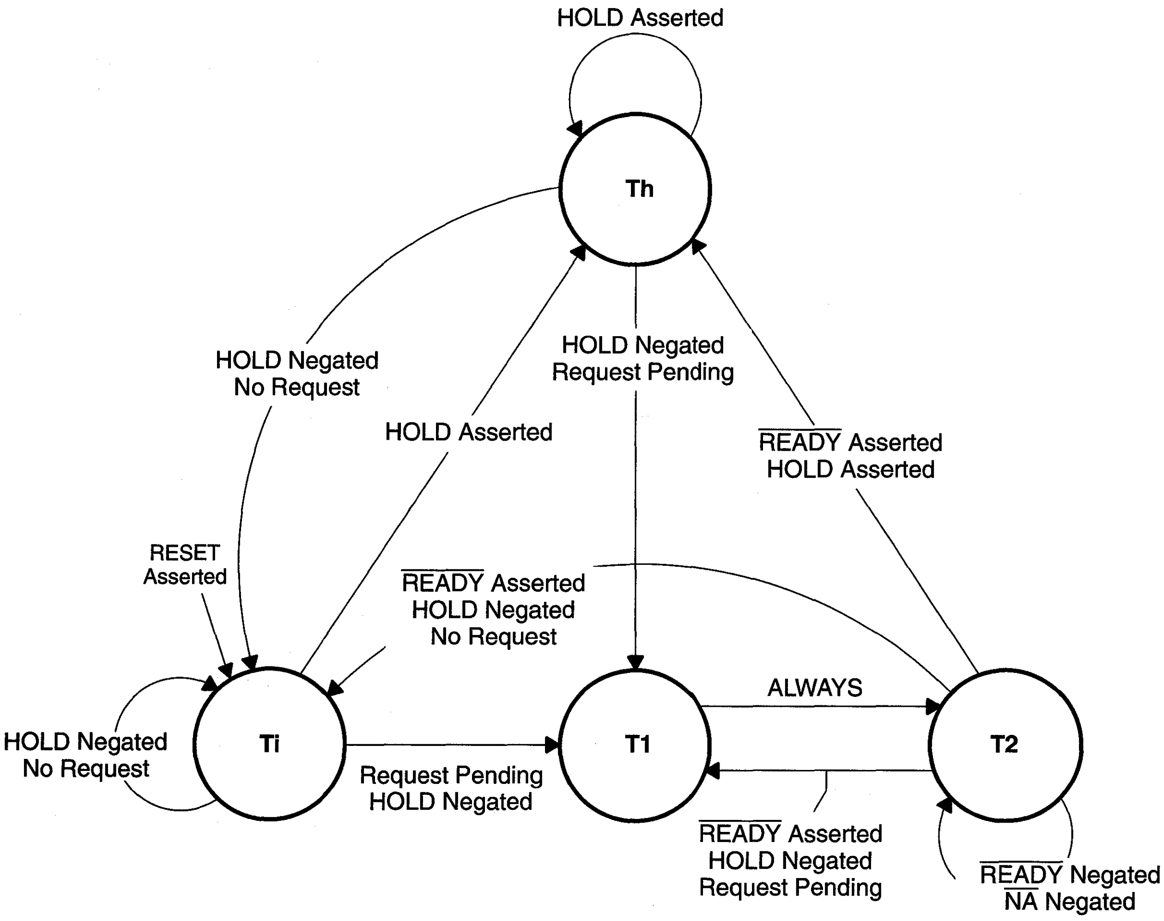


Note: Idle states are shown here for diagram variety only.

Initiating and Maintaining Non-Pipelined Cycles

The bus states and transitions for non-pipelined addressing are illustrated in Figure 4–7. The bus transitions between four possible states: T1, T2, Ti, and Th. Active bus cycles consist of T1 and T2 states, with T2 being repeated for wait states. Bus cycles always begin with a single T1 state. T1 is always followed by a T2 state. If a bus cycle is not acknowledged during a given T2 and $\overline{\text{NA}}$ is inactive, T2 is repeated resulting in a wait state. When a cycle is acknowledged during T2, the following state is T1 of the next bus cycle if a bus request is pending internally. If no internal bus request is pending, the Ti state is entered. If the HOLD input is asserted and the TI486DLC/E is ready to enter the hold acknowledge state, the Th state is entered.

Figure 4–7. Non-Pipelined Bus States



Bus States:

T1 – First clock of a non-pipelined bus cycle (CPU drives new address and asserts $\overline{\text{ADS}}$)
T2 – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has not been sampled asserted in the current bus cycle.
Ti – Idle State
Th – Hold Acknowledge (CPU asserts HLDA)

The fastest bus cycle consists of two states: T1 and T2.

Because of the demultiplexed nature of the bus, the address pipelining option provides a mechanism for the external hardware to have an additional T state of access time without inserting a wait state. After the reset sequence and following any idle bus state, the processor always uses non-pipelined address timing. Pipelined or non-pipelined address timing is then determined on a cycle-by-cycle basis using the $\overline{\text{NA}}$ input. When address pipelining is not used, the address and bus cycle definition remain valid during all wait states. When wait states are added and it is desirable to maintain non-pipelined address timing, it is necessary to negate $\overline{\text{NA}}$ during each T2 state of the bus cycle except the last one.

4.2.2.2 Bus Cycles Using Pipelined Addressing

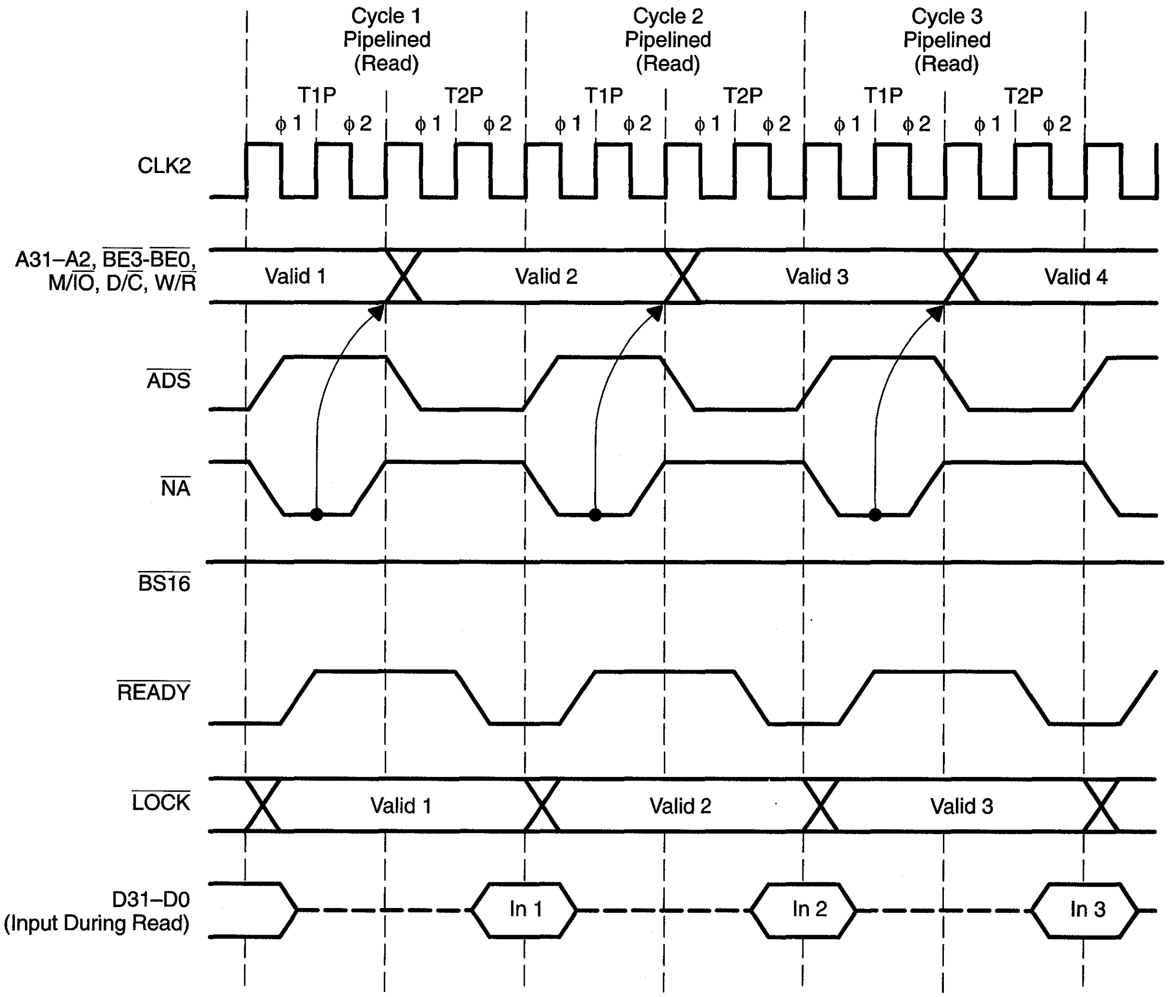
The address pipelining option allows the system to request the address and bus cycle definition of the next internally pending bus cycle before the current bus cycle is acknowledged with $\overline{\text{READY}}$ asserted. If address pipelining is used, the external system hardware has an extra T state of access time to transfer data. The address pipelining option is controlled on a cycle-by-cycle basis by the state of the $\overline{\text{NA}}$ input.

Pipelined Bus States

Pipelined addressing is always initiated by asserting $\overline{\text{NA}}$ during a non-pipelined bus cycle. Within the non-pipelined bus cycle, $\overline{\text{NA}}$ is sampled at the beginning of phase 2 of each T2 state and is only acknowledged by the TI486DLC/E during wait states. When address pipelining is acknowledged, the address ($\overline{\text{BE3}}$–$\overline{\text{BE0}}$, and A31–A2) and bus cycle definition (W/$\overline{\text{R}}$, D/$\overline{\text{C}}$, and M/$\overline{\text{IO}}$) of the next bus cycle are driven before the end of the non-pipelined cycle. The address status output ($\overline{\text{ADS}}$) is asserted simultaneously to indicate validity of the above signals. Once in effect, address pipelining is maintained in successive bus cycles by continuing to assert $\overline{\text{NA}}$ during the pipelined bus cycles.

As in non-pipelined bus cycles, the fastest bus cycles using pipelined address require only two bus states. Figure 4–8 illustrates the fastest read cycles using pipelined address timing. The two bus states for pipelined addressing are T1P and T2P or T1P and T2I. The T1P state is entered following completion of the bus cycle in which the pipelined address and bus cycle definition information was made available and is the first bus state of every pipelined bus cycle. In other words, the T1P state follows a T2 state if the previous cycle was non-pipelined, and follows a T2P state if the previous cycle was pipelined.

Figure 4–8. Fastest Pipelined Read Cycles



Note: Fastest pipelined bus cycles consist of T1P and T2P.

Within the pipelined bus cycle, $\overline{NA}$ is sampled at the beginning of phase 2 of the T1P state. If the TI486DLC/E has an internally pending bus request and $\overline{NA}$ is asserted, the T1P state is followed by a T2P state and the address and bus cycle definition for the next pending bus request is made available. If no pending bus request exists, the T1P state is followed by a T2I state regardless of the state of $\overline{NA}$ and no new address or bus cycle information is driven.

The pipelined bus cycle is terminated in either the T2P or T2I states with the assertion of the $\overline{READY}$ input and valid data is either input or output depending on the bus cycle type. $\overline{READY}$ is ignored at the end of the T1P state.

Pipelined Read and Write Cycles

Any bus cycle may be performed with pipelined address timing. When a read cycle is performed, the TI486DLC/E microprocessor floats its data bus and the externally addressed device then drives the data. When a read cycle is acknowledged by $\overline{READY}$ asserted in either the T2P or T2I bus state, the TI486DLC/E CPU latches the information present at its data pins and terminates the cycle.

When a write cycle is performed, the data bus is driven by the TI486DLC/E CPU beginning in phase 2 of T1P. When a write cycle is acknowledged, the TI486DLC/E write data remains valid throughout phase 1 of the next bus state to provide write data hold time.

Pipelined Wait States

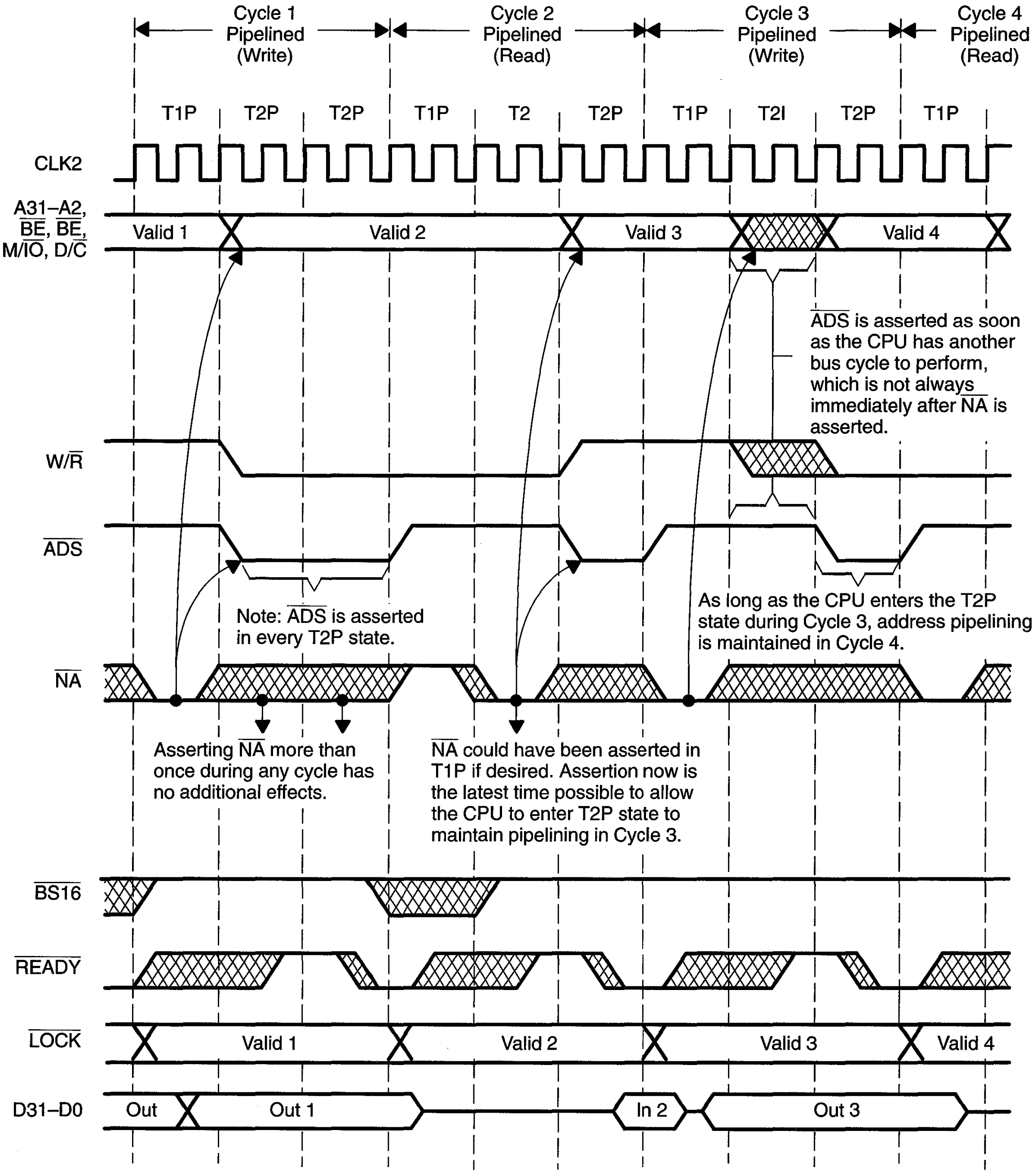
Once a pipelined bus cycle begins, it continues until acknowledged by the external system hardware using the TI486DLC/E $\overline{\text{READY}}$ input. Acknowledging the bus cycle at the end of the first T2P or T2I state results in the shortest possible pipelined bus cycle. If READY is not immediately asserted, however, T2P or T2I states are repeated indefinitely until the $\overline{\text{READY}}$ input is sampled active. Additional T2P or T2I states are referred to as wait states.

Figure 4–9 illustrates pipelined bus cycles with one wait state added to cycles 1 through 3. Cycle 1 is a pipelined cycle with $\overline{\text{NA}}$ asserted during T1P and a pending bus request. READY is sampled inactive at the end of the first T2P state in cycle 1. Therefore, the T2P state is repeated until READY is sampled active at the end of the second T2P and the cycle is then terminated. The TI486DLC/E ignores the $\overline{\text{READY}}$ input at the end of the T1P state. Note that $\overline{\text{ADS}}$, the address and the bus cycle definition signals for the pending bus cycle are all valid during each of the T2P states. Also, asserting NA more than once during the cycle has no additional effects. Pipelined addressing can only output information for the very next bus cycle.

Cycle 2 in Figure 4–9 illustrates a pipelined cycle, with one wait state, where $\overline{\text{NA}}$ is not asserted until the second bus state in the cycle. In this case, the CPU enters the T2 state following T1P because $\overline{\text{NA}}$ is not asserted. During the T2 state, the TI486DLC/E samples $\overline{\text{NA}}$ asserted. Because a bus request is pending internally and $\overline{\text{READY}}$ is not active, the CPU enters the T2P state and asserts $\overline{\text{ADS}}$, valid address and bus cycle definition information for the pending bus cycle. The cycle is then terminated by an active READY at the end of the T2P state.

Cycle 3 of Figure 4–9 illustrates the case where no internal bus request exists until the last state of a pipelined cycle with wait states. In cycle 3, $\overline{\text{NA}}$ is asserted in T1P requesting the next address. Because the CPU does not have an internal bus request pending, The T2I state is entered. However, by the end of the T2I state, a bus request exists. Because $\overline{\text{READY}}$ is not asserted, a wait state is added. The CPU then enters the T2P and asserts $\overline{\text{ADS}}$ and valid address and bus cycle definition information for the pending bus cycle. As long as the CPU enters the T2P state at some point during the bus cycle, pipelined addressing is maintained. $\overline{\text{NA}}$ needs to be asserted only once during the bus cycle to request pipelined addressing.

Figure 4–9. Various Pipelined Cycles (One Wait State)


Cycle 1
Pipelined
(Write)
Cycle 2
Pipelined
(Read)
Cycle 3
Pipelined
(Write)
Cycle 4
Pipelined
(Read)
T1P
T2P
T2P
T1P
T2
T2P
T1P
T2I
T2P
T1P
CLK2
A31–A2, BE, BE, M/IO, D/C
Valid 1
Valid 2
Valid 3
Valid 4
ADS is asserted as soon as the CPU has another bus cycle to perform, which is not always immediately after NA is asserted.
W/R
ADS
Note: ADS is asserted in every T2P state.
As long as the CPU enters the T2P state during Cycle 3, address pipelining is maintained in Cycle 4.
NA
Asserting NA more than once during any cycle has no additional effects.
NA could have been asserted in T1P if desired. Assertion now is the latest time possible to allow the CPU to enter T2P state to maintain pipelining in Cycle 3.
BS16
READY
LOCK
Valid 1
Valid 2
Valid 3
Valid 4
D31–D0
Out
Out 1
In 2
Out 3

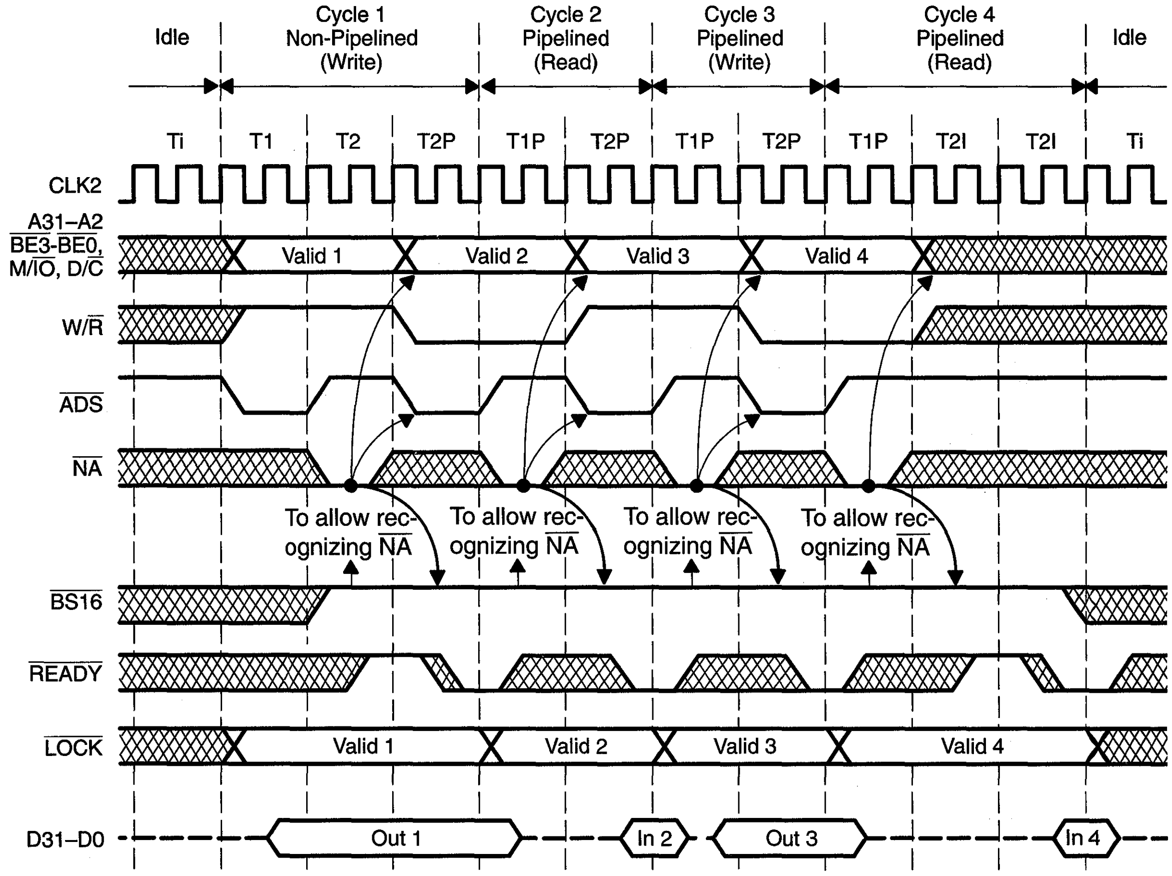
Initiating and Maintaining Pipelined Cycles

Pipelined addressing is always initiated by asserting NA during a non-pipelined bus cycle with at least one wait state. The first bus cycle following RESET, an idle bus, or a hold acknowledge state is always non-pipelined. Therefore, the TI486DLC/E always issues at least one non-pipelined bus cycle following RESET, idle, or hold acknowledge before pipelined addressing takes effect.

Once a bus cycle is in progress and the current address has been valid for one entire bus state, the $\overline{\text{NA}}$ input is sampled at the end of every phase one until the bus cycle is acknowledged. Once $\overline{\text{NA}}$ is sampled active, the TI486DLC/E microprocessor is free to drive a new address and bus cycle definition on the bus as early as the next bus state and as late as the last bus state in the cycle.

Figure 4–10 illustrates the fastest transition possible to pipelined addressing following an idle bus state. In Cycle 1, $\overline{\text{NA}}$ is driven during state T2. Thus, Cycle 1 makes the transition to pipelined address timing, since it begins with T1 but ends with T2P. Because the address for Cycle 2 is available before Cycle 2 begins, Cycle 2 is called a pipelined bus cycle, and it begins with a T1P state. Cycle 2 begins as soon as $\overline{\text{READY}}$ asserted terminates Cycle 1.

Figure 4–10. Fastest Transition to Pipelined Address Following Idle Bus State

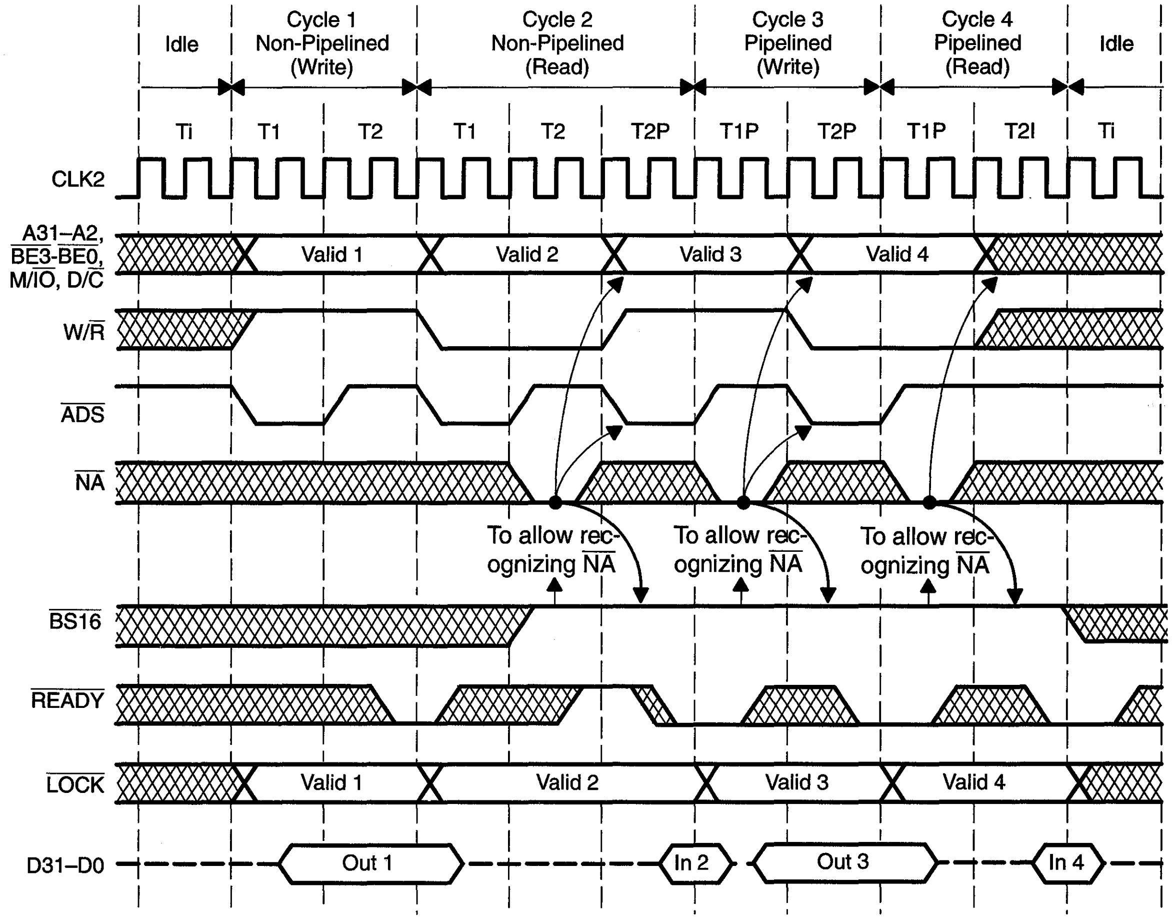


Note: Following any idle bus state (Ti) the address is always non-pipelined and $\overline{\text{NA}}$ is sampled only during wait states. To start address pipelining after an idle state requires a non-pipelined cycle with at least one wait state (Cycle 1 above). The pipelined cycles (2, 3, and 4 above) are shown with various numbers of wait states.

Figure 4–11 illustrates transitioning to pipelined addressing during a burst of bus cycles. Cycle 2 makes the transition to pipelined addressing. Comparing Cycle 2 to Cycle 1 of Figure 3–10 illustrates that a transition cycle is the same whenever it occurs consisting of at least T1, T2 ($\overline{\text{NA}}$ is asserted at that time), and T2P (provided the TI486DLC/E microprocessor has an internal bus request already pending). T2P states are repeated if wait states are added to the cycle. Cycles 2, 3, and 4 in Figure 4–11 show that once address pipelining is achieved it can be maintained with two-state bus cycles consisting only of T1P and T2P.

Once a pipelined bus cycle is in progress, pipelined timing is maintained for the next cycle by asserting $\overline{NA}$ and detecting that the TI486DLC/E microprocessor enters T2P during the current bus cycle. The current bus cycle must end in state T2P for pipelining to be maintained in the next cycle. T2P is identified by the assertion of $\overline{ADS}$. Figure 4–10 and Figure 4–11 each show pipelining ending after Cycle 4. This occurred because the TI486DLC/E CPU did not have an internal bus request prior to the acknowledgment of Cycle 4.

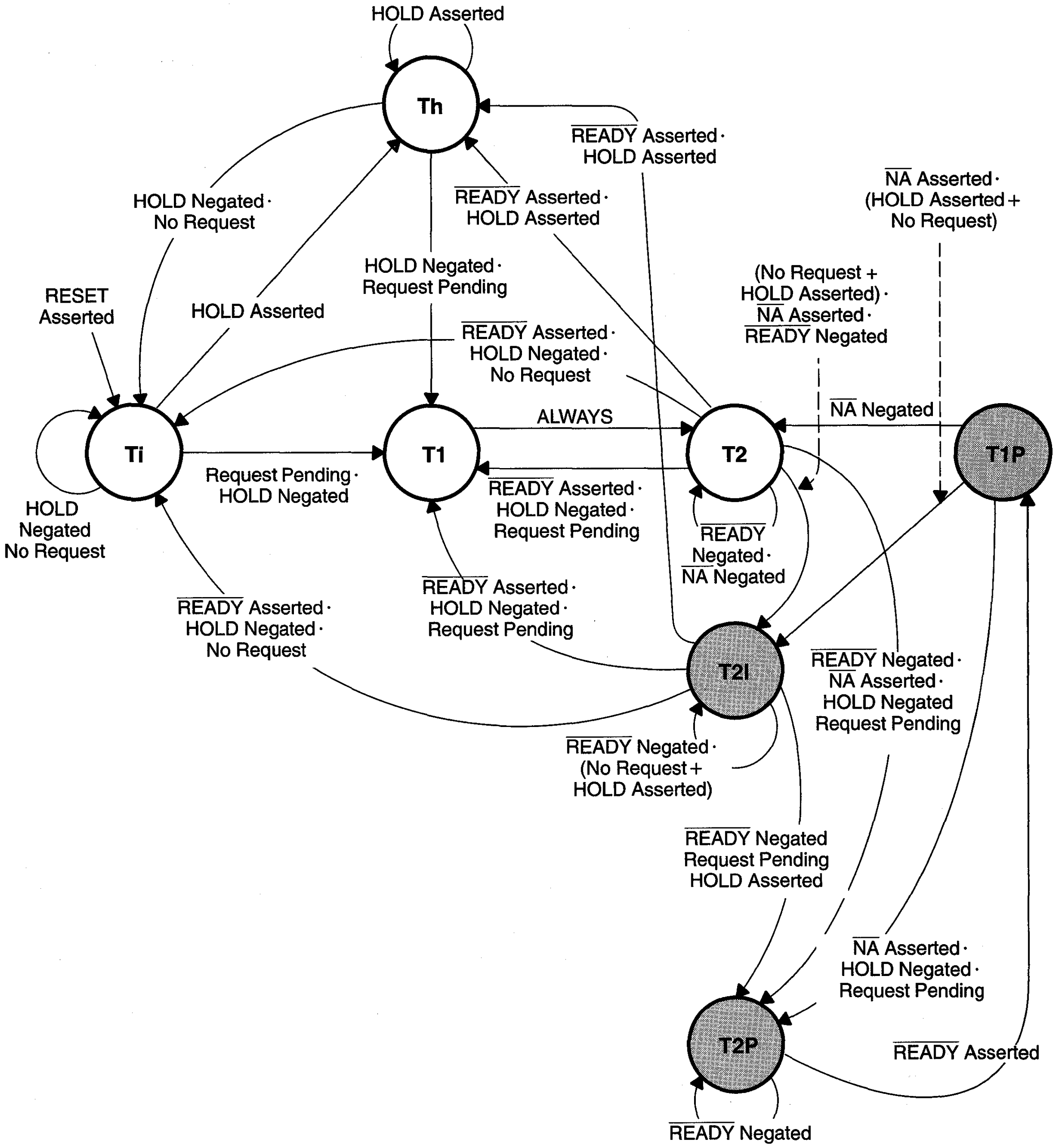
Figure 4–11. Transitioning to Pipelined Address During Burst of Bus Cycles



Note: Following any idle bus state (Ti), addresses are non-pipelined bus cycles, $\overline{NA}$ is sampled only during wait states. Therefore, to begin address pipelining during a group of non-pipelined bus cycles requires a non-pipelined cycle with at least one wait state (Cycle 2 above).

The complete bus state transition diagram, including operation with pipelined address is given in Figure 3–12. This is a superset of the diagram for non-pipelined address. The three additional bus states for pipelined address are shaded.

Figure 4–12. Complete Bus States


HOLD Asserted
Th
READY Asserted · HOLD Asserted
HOLD Negated · No Request
READY Asserted · HOLD Asserted
NA Asserted · (HOLD Asserted + No Request)
HOLD Negated · Request Pending
(No Request + HOLD Asserted) · NA Asserted · READY Negated
RESET Asserted
HOLD Asserted
READY Asserted · HOLD Negated · No Request
NA Negated
ALWAYS
Ti
T1
T2
T1P
HOLD Negated No Request
Request Pending · HOLD Negated
READY Asserted · HOLD Negated · Request Pending
READY Negated · NA Negated
READY Asserted · HOLD Negated · Request Pending
READY Asserted · HOLD Negated · No Request
T2I
READY Negated · NA Asserted · HOLD Negated Request Pending
READY Negated · (No Request + HOLD Asserted)
READY Negated Request Pending HOLD Asserted
NA Asserted · HOLD Negated · Request Pending
T2P
READY Asserted
READY Negated

Bus States:

T1 – First clock of a non-pipelined bus cycle (CPU drives new address and asserts $\overline{\text{ADS}}$).
T2 – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has not been sampled asserted in the current bus cycle.
T2I – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has been sampled asserted in the current bus cycle but there is not yet an internal bus request pending (CPU drives new address and asserts $\overline{\text{ADS}}$).
T2P – Subsequent clocks of a bus cycle when $\overline{\text{NA}}$ has been sampled asserted in the current bus cycle and there is an internal bus request pending (CPU drives new address and asserts $\overline{\text{ADS}}$).
T1P – First clock of a pipelined bus cycle.
Ti – Idle state.
Th – Hold Acknowledge state (CPU asserts HLDA).

4.2.3 Bus Cycles Using $\overline{\text{BS16}}$

Assertion of $\overline{\text{BS16}}$ during a bus cycle effectively changes the TI486DLC/E 32-bit bus into a 16-bit data bus. Although slower, the 16-bit data bus usually requires less hardware interface circuitry and generally offers greater compatibility with 16-bit devices.

Non-Pipelined Cycles

With $\overline{\text{BS16}}$ asserted, all operand transfers physically occur on data bus lines D15-D0. With $\overline{\text{BS16}}$ asserted during a 32-bit non-pipelined read or write, additional bus cycles are issued by the CPU to transfer the data.

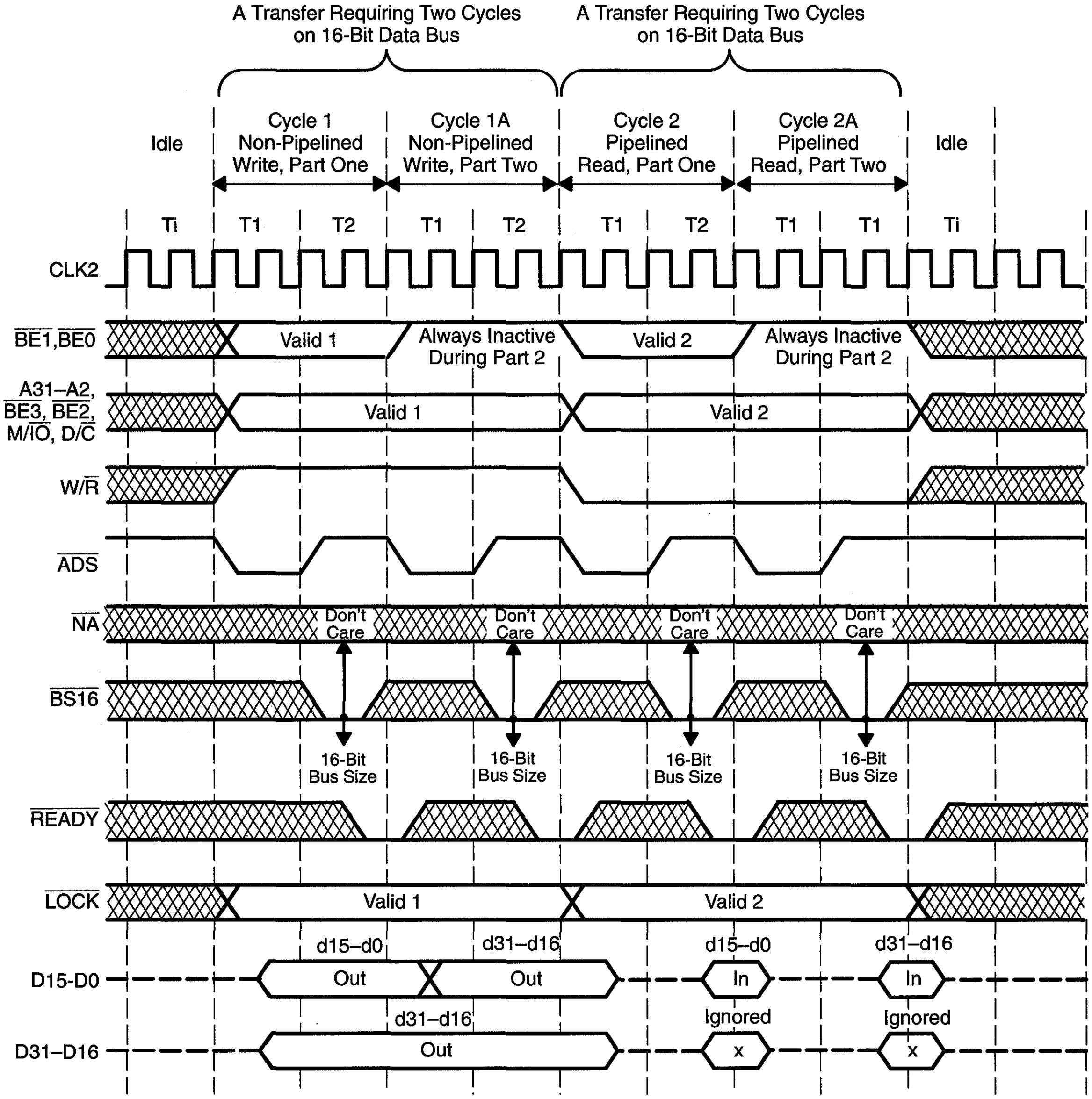
For data reads with only the two upper bytes selected ($\overline{\text{BE3}}$ and/or $\overline{\text{BE2}}$ asserted), data is read from D15-D0.

For data writes with only the two upper bytes selected ($\overline{\text{BE3}}$ and/or $\overline{\text{BE2}}$ asserted), data is duplicated on D15-D0 and no further action is required.

For data reads with all four bytes selected (at least $\overline{\text{BE1}}$, $\overline{\text{BE2}}$ asserted and possibly $\overline{\text{BE0}}$ and/or $\overline{\text{BE3}}$ also asserted), the CPU performs two 16-bit read cycles using data lines D15-D0. Lines D31-D16 are ignored.

Data writes with all four bytes selected (at least $\overline{\text{BE1}}$, $\overline{\text{BE2}}$ asserted and possibly $\overline{\text{BE0}}$ and/or $\overline{\text{BE3}}$ also asserted), the CPU performs two 16-bit write cycles using data lines D15-D0. Bytes 0 and 1 (corresponding to $\overline{\text{BE0}}$, $\overline{\text{BE1}}$) are sent on the first bus cycle and bytes 2 and 3 (corresponding to $\overline{\text{BE2}}$, $\overline{\text{BE3}}$) are sent on the second bus cycle. $\overline{\text{BE0}}$ and $\overline{\text{BE1}}$ are always negated during the second 16-bit bus cycle. Figure 4–13 illustrates two non-pipelined bus cycles using $\overline{\text{BS16}}$.

Figure 4–13. Non-Pipelined Bus Cycles Using $\overline{BS16}$


A Transfer Requiring Two Cycles on 16-Bit Data Bus
A Transfer Requiring Two Cycles on 16-Bit Data Bus
Idle
Cycle 1 Non-Pipelined Write, Part One
Cycle 1A Non-Pipelined Write, Part Two
Cycle 2 Pipelined Read, Part One
Cycle 2A Pipelined Read, Part Two
Idle
Ti
T1
T2
T1
T2
T1
T2
T1
T1
Ti
CLK2
BE1,BE0
Valid 1
Always Inactive During Part 2
Valid 2
Always Inactive During Part 2
A31–A2, BE3, BE2, M/IO, D/C
Valid 1
Valid 2
W/R
ADS
NA
Don't Care
Don't Care
Don't Care
Don't Care
BS16
16-Bit Bus Size
16-Bit Bus Size
16-Bit Bus Size
16-Bit Bus Size
READY
LOCK
Valid 1
Valid 2
D15-D0
d15–d0
Out
d31–d16
Out
d15–d0
In
d31–d16
In
D31–D16
d31–d16
Out
Ignored
x
Ignored
x

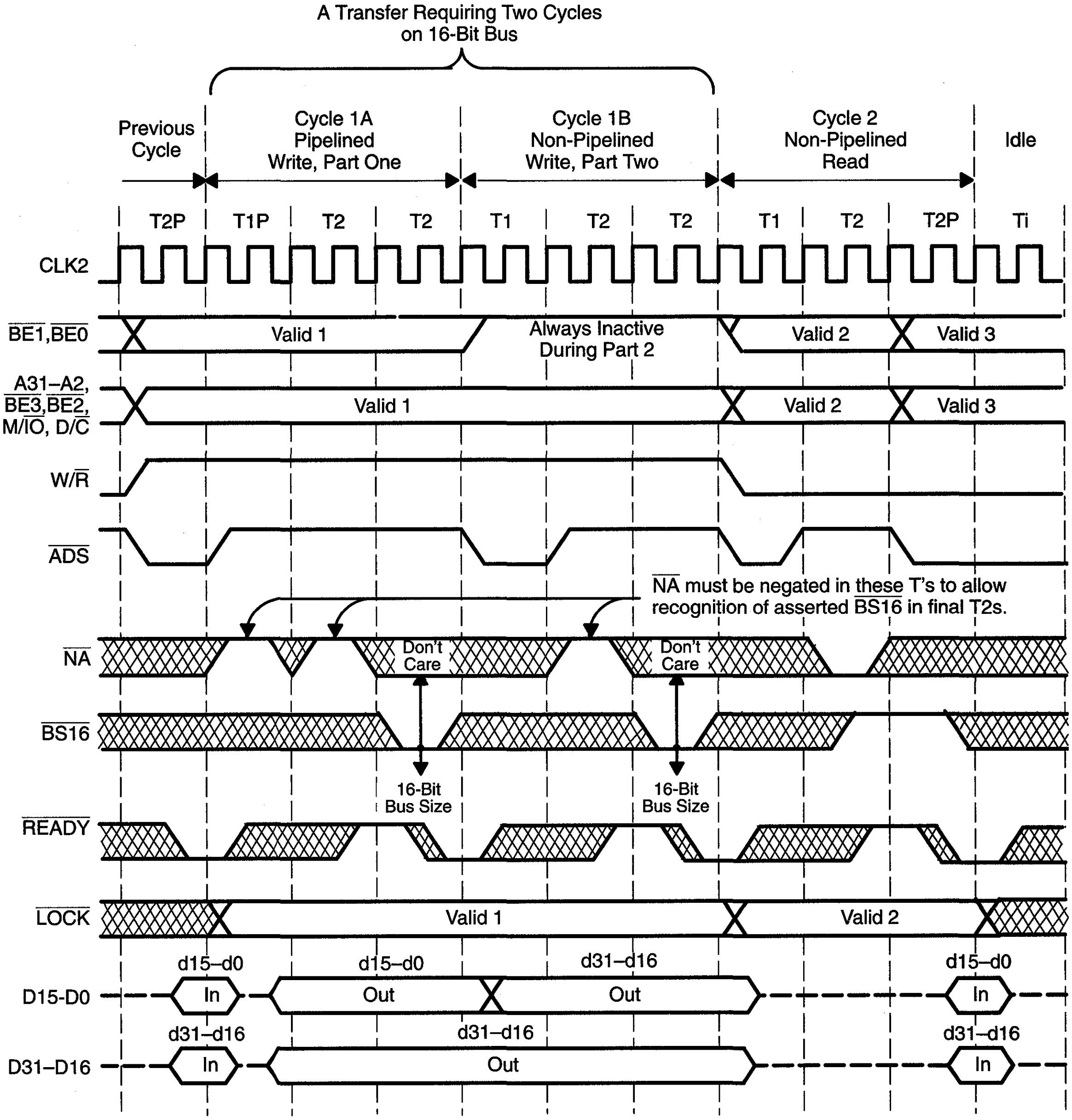
Note: Dn = physical data pin n.
dn = logical data bit n.

Pipelined Cycles

The input signal $\overline{\text{NA}}$ is a request to the CPU to drive the address, byte enables, and bus status signals for the next bus cycle as soon as they become internally available. “Pipelining” this address allows the system logic to anticipate the next bus cycle operation.

The CPU cannot acknowledge both address pipelining and $\overline{\text{BS16}}$ for the same bus cycle. If $\overline{\text{NA}}$ is already sampled when $\overline{\text{BS16}}$ is asserted, the data bus remains 32-bits wide. If $\overline{\text{NA}}$ and $\overline{\text{BS16}}$ are asserted in the same window, $\overline{\text{NA}}$ is ignored and $\overline{\text{BS16}}$ remains effective (the data bus becomes 16-bits wide). Figure 4–14 illustrates the interaction between $\overline{\text{NA}}$ and $\overline{\text{BS16}}$.

Figure 4–14. Pipelining and $\overline{\text{BS16}}$


A Transfer Requiring Two Cycles on 16-Bit Bus
Previous Cycle
Cycle 1A Pipelined Write, Part One
Cycle 1B Non-Pipelined Write, Part Two
Cycle 2 Non-Pipelined Read
Idle
T2P
T1P
T2
T2
T1
T2
T2
T1
T2
T2P
Ti
CLK2
BE1,BE0
Valid 1
Always Inactive During Part 2
Valid 2
Valid 3
A31–A2, BE3,BE2, M/IO, D/C
Valid 1
Valid 2
Valid 3
W/R
ADS
NA must be negated in these T's to allow recognition of asserted BS16 in final T2s.
NA
Don't Care
Don't Care
BS16
16-Bit Bus Size
16-Bit Bus Size
READY
LOCK
Valid 1
Valid 2
d15–d0
d15–d0
d31–d16
d15–d0
D15-D0
In
Out
Out
In
d31–d16
d31–d16
d31–d16
D31–D16
In
Out
In

Dn = physical data pin n.
dn = logical data bit n.
Cycle 1A is pipelined. Cycle 1B cannot be pipelined, but its address can be inferred from cycle 1 to externally simulate address pipelining during cycle 1B.

4.2.4 Locked Bus Cycles

When the $\overline{\text{LOCK}}$ signal is asserted, the TI486DLC/E microprocessor does not allow other bus master devices to gain control of the system bus. $\overline{\text{LOCK}}$ is driven active in response to executing certain instructions with the LOCK prefix. The LOCK prefix allows indivisible read/modify/write operations on memory operands. $\overline{\text{LOCK}}$ is also active during interrupt acknowledge cycles.

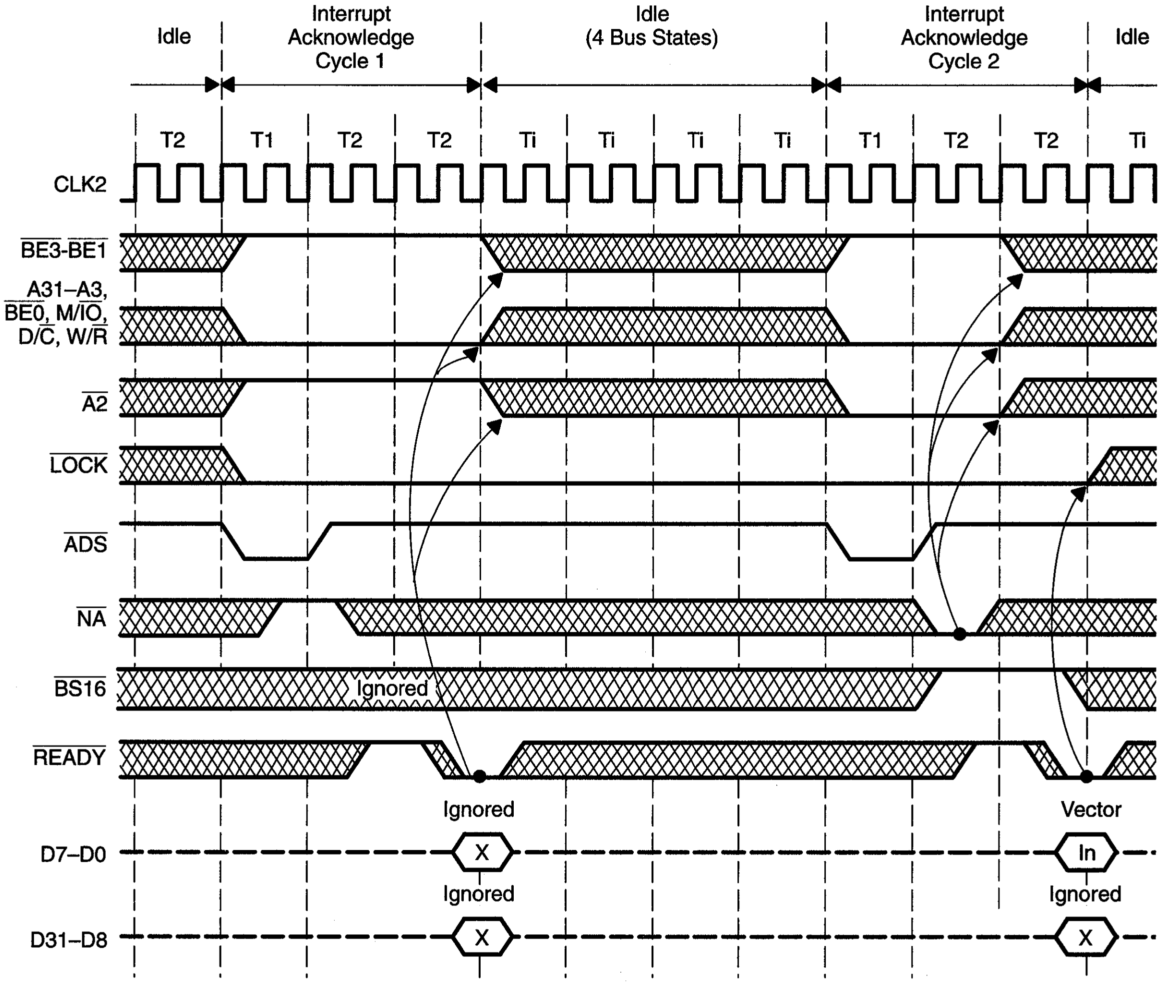
LOCK is activated on the CLK2 edge that begins the first locked bus cycle and is deactivated when READY is returned at the end of the last locked bus cycle. When using non-pipelined addressing, LOCK is asserted during phase 1 of T1. When using pipelined addressing, LOCK is driven valid during phase 1 of T1P.

Figure 4–4 through Figure 4–6 and Figure 4–13 illustrate LOCK timing during non-pipelined cycles and Figure 4–8 through Figure 4–11 and Figure 4–14 cover the pipelined address case.

4.2.5 Interrupt Acknowledge (INTA) Cycles

The TI486DLC/E microprocessor is interrupted by an external source via an input request on the INTR input (when interrupts are enabled). The TI486DLC/E microprocessor responds with two locked interrupt acknowledge cycles. These bus cycles are similar to read cycles. Each cycle is terminated by READY sampled active as shown in Figure 4–15.

Figure 4–15. Interrupt Acknowledge Cycles



Note: Interrupt Vector (0–255) is read on D7–D0 at end of second interrupt acknowledge bus cycle. Because each Interrupt Acknowledge bus cycle is followed by idle bus states, asserting NA has no practical effect.

The state of A2 distinguishes the first and second interrupt acknowledge cycles. The address driven during the first interrupt acknowledge cycle is 4h (A31–A3=0, A2=1, $\overline{\text{BE3}}$–$\overline{\text{BE1}}$=1, and $\overline{\text{BE0}}$=0). The address driven during the second interrupt acknowledge cycle is 0h (A31–A2=0, $\overline{\text{BE3}}$–$\overline{\text{BE1}}$=1, and $\overline{\text{BE0}}$=0).

To assure that the interrupt acknowledge cycles are executed indivisibly, the $\overline{\text{LOCK}}$ output is asserted from the beginning of the first interrupt acknowledge cycle until the end of the second interrupt acknowledge cycle. Four idle bus states (Ti) are always inserted by the TI486DLC/E microprocessor between the two interrupt acknowledge cycles.

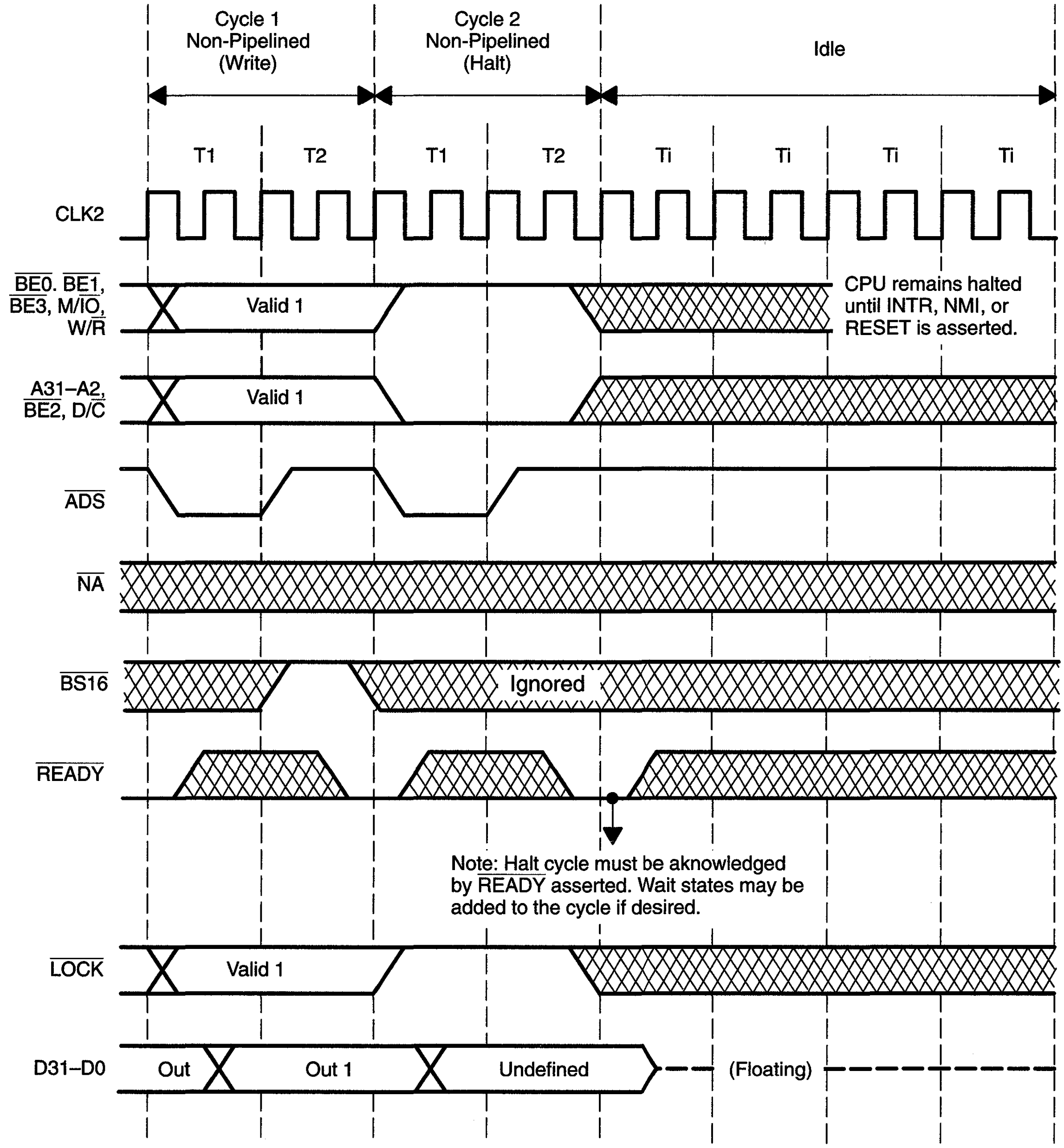
The interrupt vector is read at the end of the second interrupt cycle. The vector is read by the TI486DLC/E microprocessor from D7–D0 of the data bus. The vector indicates the specific interrupt number (from 0–255) requiring service. Throughout the balance of the two interrupt cycles, D31–D0 float. At the end of the first interrupt acknowledge cycle, any data presented to the TI486DLC/E is ignored.

4.2.6 Halt and Shutdown Cycles

Halt Indication Cycle

Executing the HLT instruction causes the TI486DLC/E execution unit to cease operation. Signaling its entrance into the halt state, a halt indication cycle is performed. The halt indication cycle is identified by the state of the bus cycle definition signals (M/$\overline{\text{IO}}$=1, D/$\overline{\text{C}}$=0, W/$\overline{\text{R}}$=1, $\overline{\text{LOCK}}$=1) and an address of 2h (A31–A2=0, $\overline{\text{BE3}}$=1, $\overline{\text{BE2}}$=0, $\overline{\text{BE1}}$–$\overline{\text{BE0}}$=1). The halt indication cycle must be acknowledged by $\overline{\text{READY}}$ asserted. A halted TI486DLC/E microprocessor resumes execution when INTR (if interrupts are enabled), NMI, or RESET is asserted. Figure 4–16 illustrates a non-pipelined halt cycle.

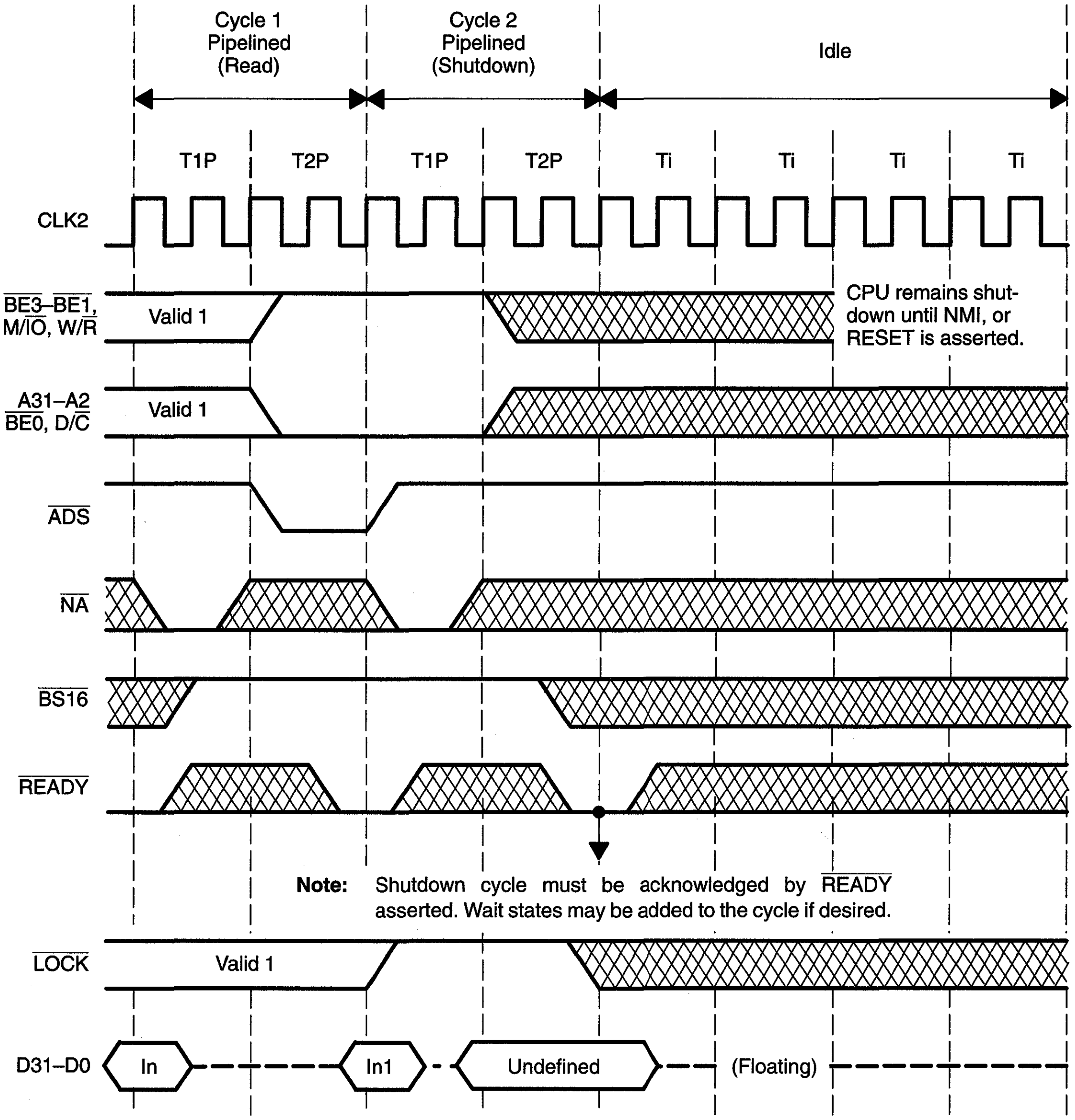
Figure 4–16. Non-Pipelined Halt Cycle



Shutdown Indication Cycle

Shutdown occurs when a severe error is detected that prevents further processing. The TI486DLC/E microprocessor shuts down as a result of a protection fault while attempting to process a double fault as well as the conditions referenced in Chapter 2. Signaling its entrance into the shutdown state, a shutdown indication cycle is performed. The shutdown indication cycle is identified by the state of the bus cycle definition signals ($M/\overline{IO}$=1, $D/\overline{C}$=0, $W/\overline{R}$=1, $\overline{LOCK}$=1) and an address of 0h (A31–A2=0, $\overline{BE3}$–$\overline{BE1}$=1, and $\overline{BE0}$=0). The shutdown indication cycle must be acknowledged by $\overline{READY}$ asserted. A shutdown TI486DLC/E microprocessor resumes execution only when NMI or RESET is asserted. Figure 4–17 illustrates a shutdown cycle using pipelined addressing.

Figure 4–17. Pipelined Shutdown Cycle



Note: Shutdown cycle must be acknowledged by READY asserted. Wait states may be added to the cycle if desired.

4.2.7 Internal Cache Interface

4.2.7.1 Cache Fills

Any unlocked memory read cycle can be cached by the TI486DLC/E. The TI486DLC/E automatically does not cache accesses to memory addresses specified by the non-cacheable region registers. Additionally, the $\overline{\text{KEN}}$ input can be used to enable caching of memory accesses on a cycle-by-cycle basis. The TI486DLC/E acknowledges the $\overline{\text{KEN}}$ input only if the KEN enable bit is set in the CCR0 configuration register.

As shown in Figure 4–19 and Figure 4–20, the TI486DLC/E samples the $\overline{\text{KEN}}$ input one CLK2 before $\overline{\text{READY}}$ is sampled active. If $\overline{\text{KEN}}$ is asserted and the current address is not set as non-cacheable per the non-cacheable region registers, then the TI486DLC/E fills two bytes of a line in the cache with the data present on the data bus pins. The states of $\overline{\text{BE3}}$–$\overline{\text{BE0}}$ are ignored if $\overline{\text{KEN}}$ is asserted for the cycle.

Figure 4–18. Non-Pipelined Cache Fills Using $\overline{\text{KEN}}$

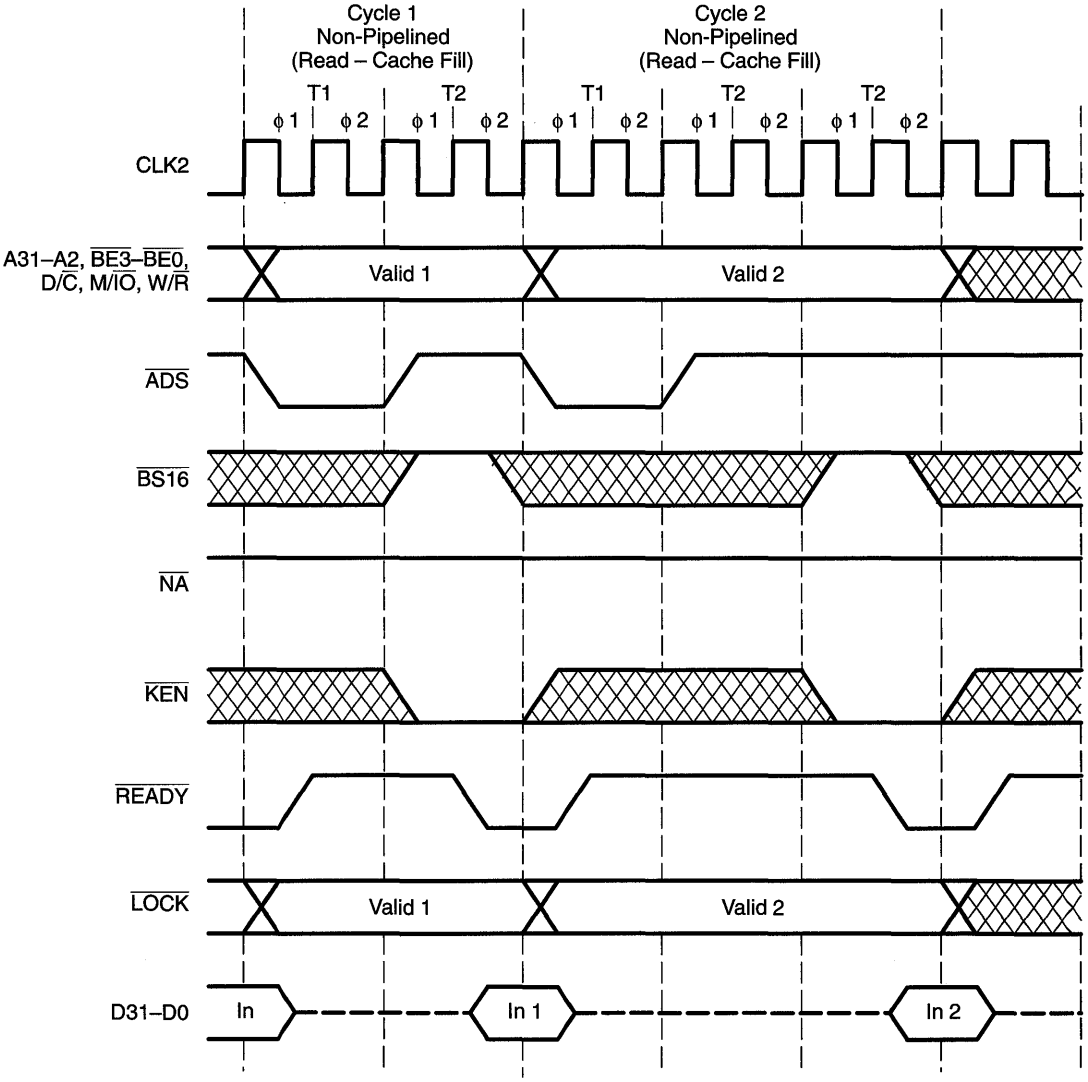


Figure 4–19. Non-Pipelined Cache Fills Using $\overline{KEN}$ and $\overline{BS16}$

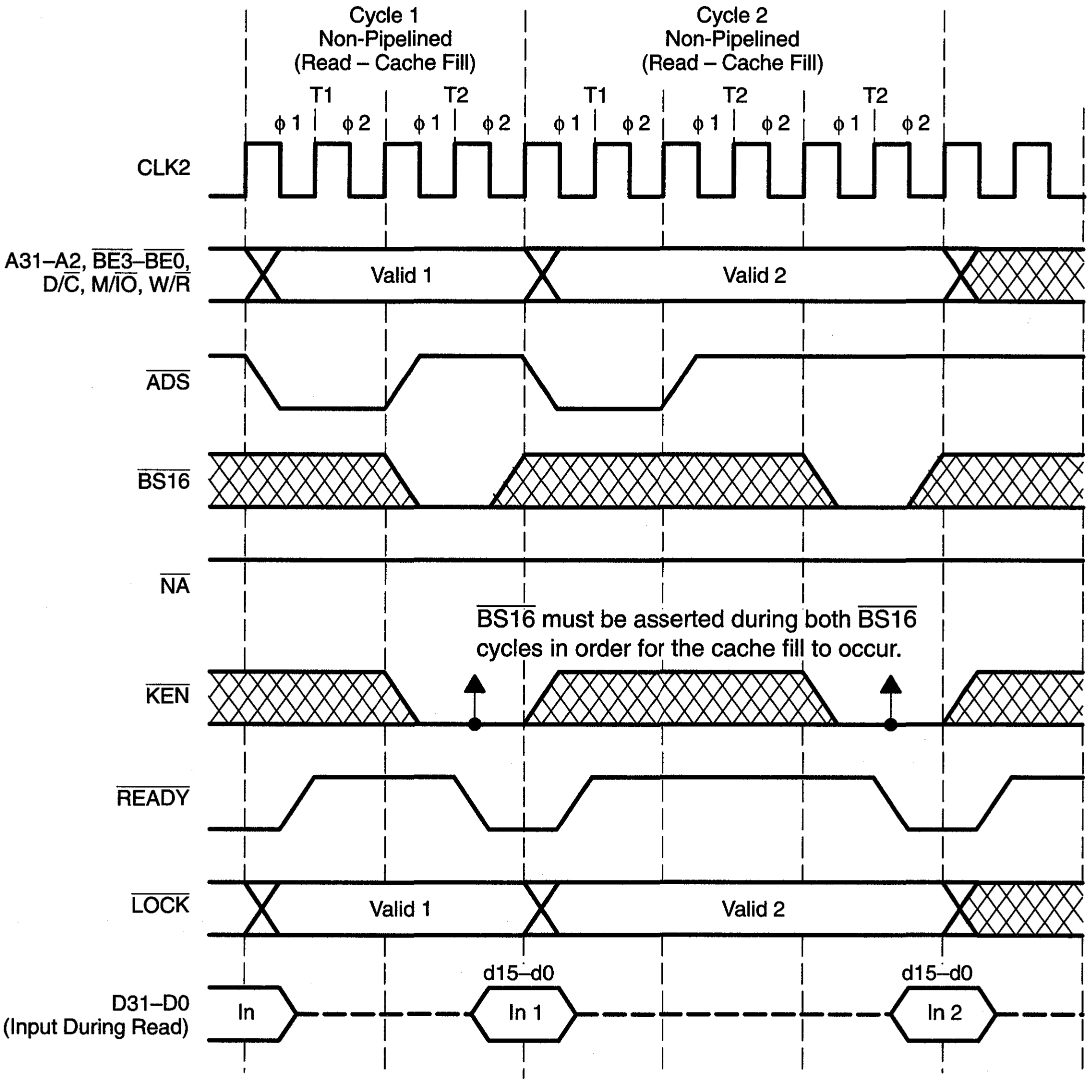
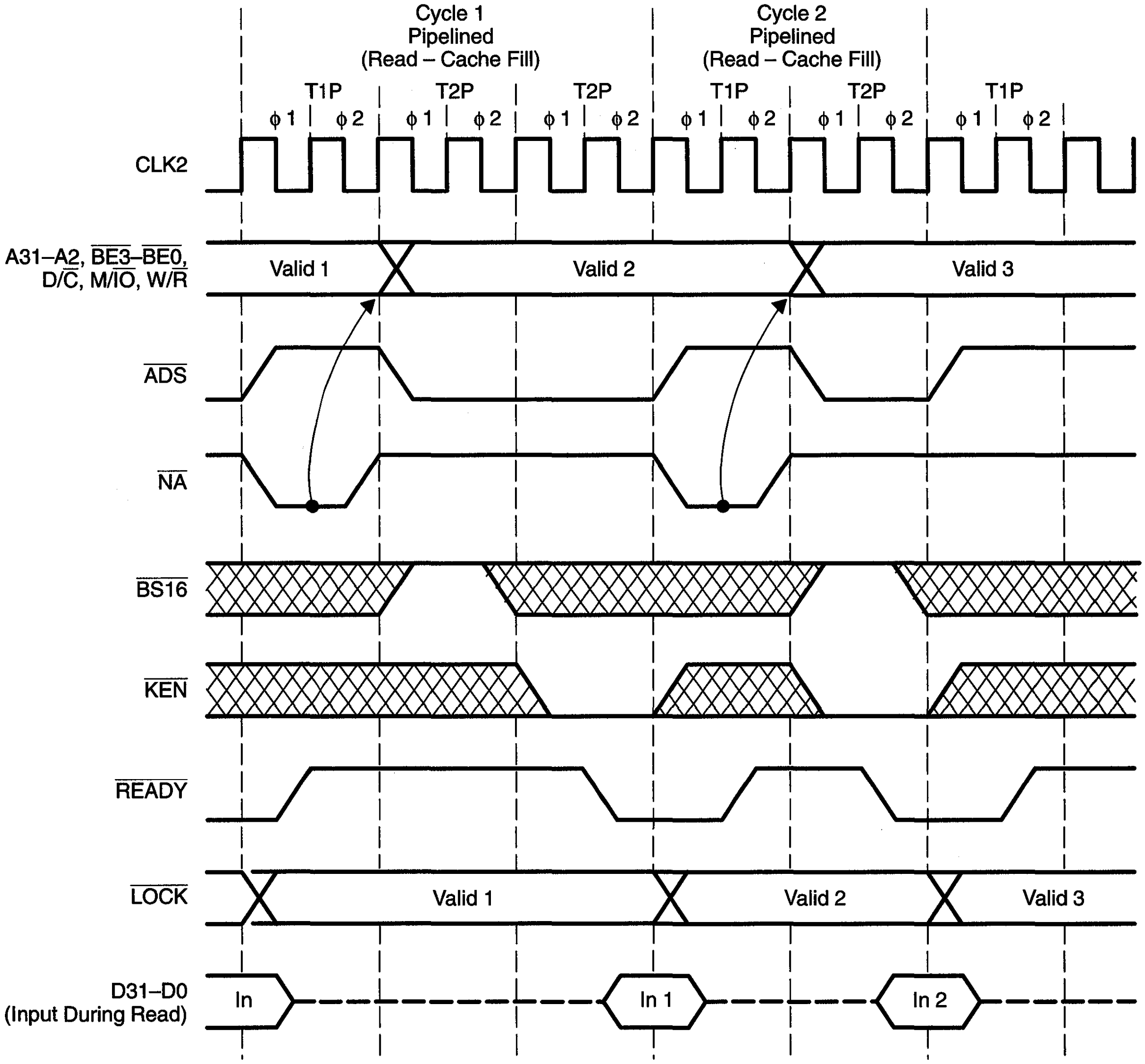


Figure 4–20. Pipelined Cache Fills Using $\overline{KEN}$



4.2.7.2 Flushing the Cache

To maintain cache coherency with external memory, the TI486DLC/E cache contents should be invalidated when previously cached data is modified in external memory by another bus master. The TI486DLC/E invalidates the internal cache contents during execution of the INVD and WBINVD instructions, following assertion of HLDA if the BARB bit is set in the CCR0 configuration register, or following assertion of $\overline{\text{FLUSH}}$ if the FLUSH bit is set in CCR0.

The TI486DLC/E samples the $\overline{\text{FLUSH}}$ input on the rising edge of CLK2 corresponding to the beginning of phase 2 of the internal processor clock. If $\overline{\text{FLUSH}}$ is asserted, the TI486DLC/E invalidates the entire contents of the internal cache. The actual point in time where the cache is invalidated depends upon the internal state of the execution pipeline. $\overline{\text{FLUSH}}$ must be asserted for at least two CLK2 periods and must meet specified setup and hold times to be recognized on a specific CLK2 edge.

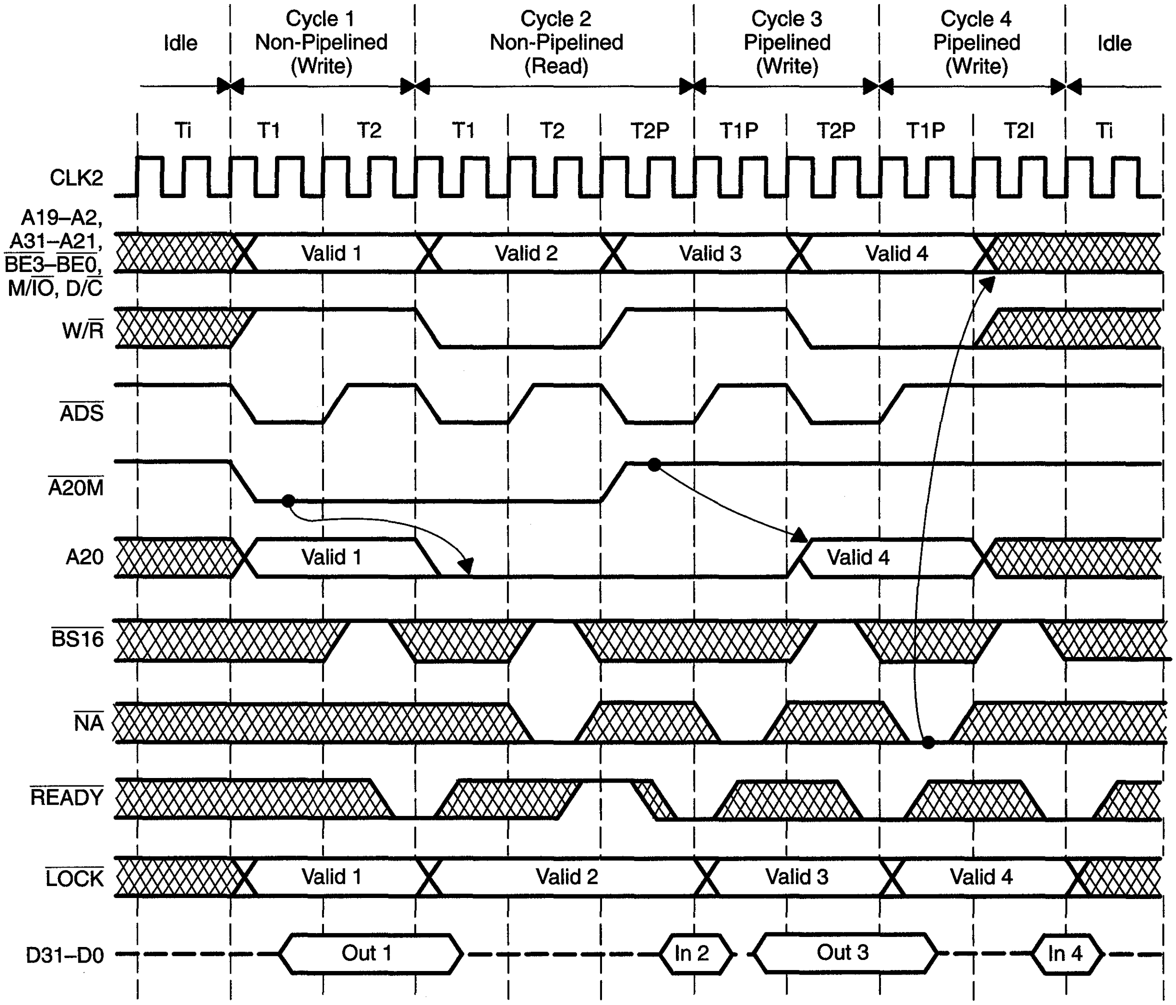
4.2.8 Address Bit 20 Masking

The TI486DLC/E can be forced to provide 8086 1-MByte address wraparound compatibility by setting the A20 bit in the CCR0 configuration register and asserting the $\overline{\text{A20M}}$ input. When the $\overline{\text{A20M}}$ is asserted, the 20th bit in the address to both the internal cache and the external bus pin is masked (zeroed).

As shown in Figure 4–21, the TI486DLC/E samples the A20M input on the rising edge of CLK2 corresponding to the beginning of phase 2 of the internal processor clock. If A20M is asserted and paging is not enabled, the TI486DLC/E masks the A20 signal internally starting with the next cache access and externally starting with the next bus cycle. If paging is enabled, the A20 signal is not masked regardless of the state of A20M. A20 remains masked until the access following detection of an inactive state on the A20M pin. A20M must be asserted for a minimum of two CLK2 periods and must meet specified setup and hold times to be recognized on a specific CLK2 edge.

An alternative to using the A20M pin is provided by the NC0 bit in the CCR0 configuration register. The TI486DLC/E automatically does not cache accesses, to the first 64 KBytes and to 1 MByte + 64 KBytes, if the NC0 bit is set. This prevents data within the wraparound memory area from residing in the internal cache and thus eliminates the need for masking A20 to the internal cache.

Figure 4–21. Masking A20 Using A20M During Burst of Bus Cycles



4.2.9 Hold Acknowledge State

The hold acknowledge state provides the mechanism for an external device in a TI486DLC/E system to acquire the TI486DLC/E system bus while the TI486DLC/E is held in an inactive bus state. This allows external bus masters to take control of the TI486DLC/E bus and directly access system hardware in a shared manner with the TI486DLC/E. The TI486DLC/E continues to execute instructions out of the cache (if enabled) until a system bus cycle is required.

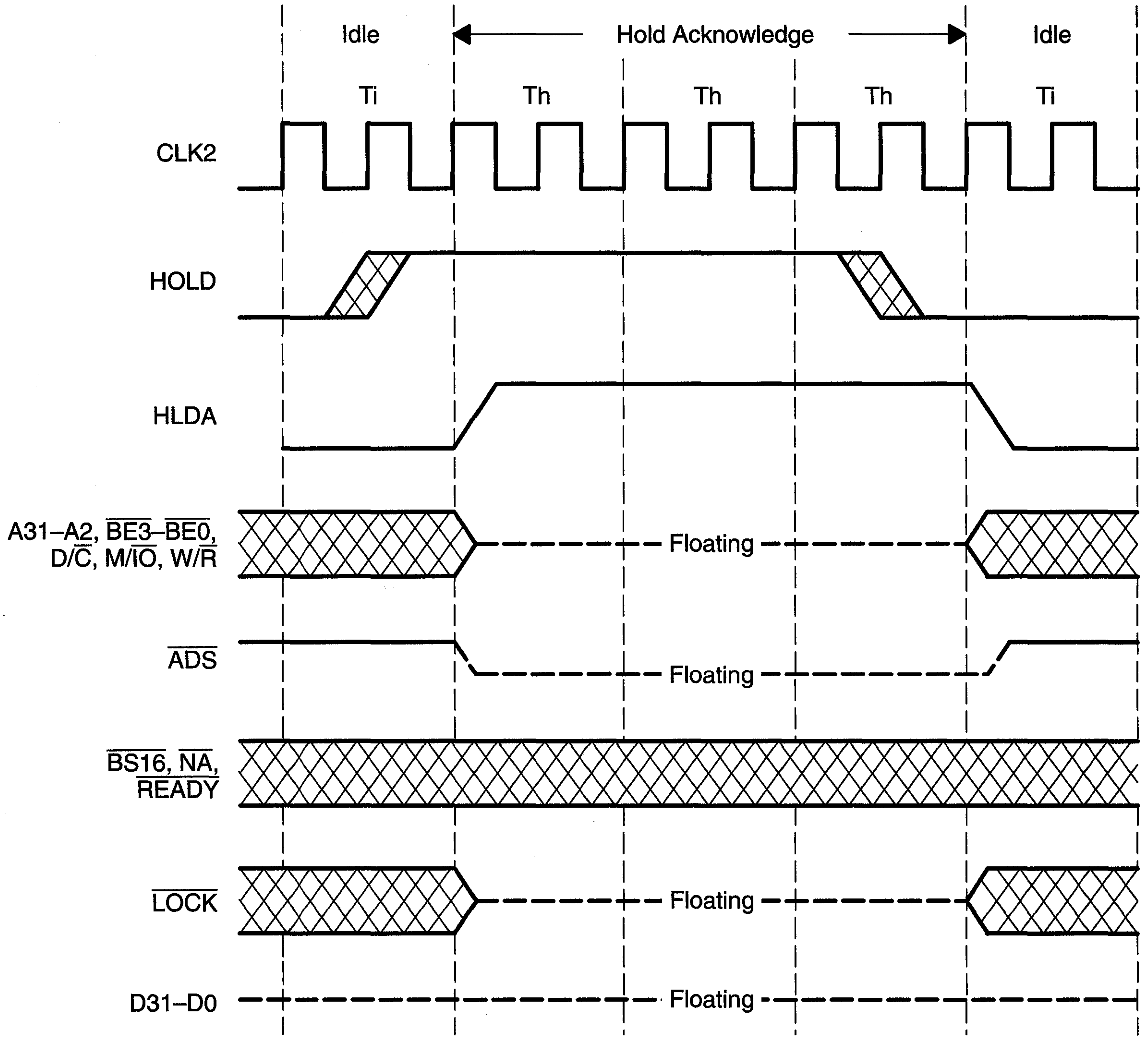
The hold acknowledge state (Th) is entered in response to assertion of the HOLD input. in the hold acknowledge state, the TI486DLC/E microprocessor floats all output and bidirectional signals, except for HLDA and $\overline{\text{SUSPA}}$. HLDA is asserted as long as the TI486DLC/E CPU remains in the hold acknowledge state and all inputs except HOLD, $\overline{\text{FLUSH}}$, $\overline{\text{SUSP}}$ and RESET are ignored.

Th may be entered directly from a bus idle state, as in Figure 4–22, or after the completion of the current physical bus cycle if the LOCK signal is not asserted, as in Figure 4–23 and Figure 4–24. The CPU samples the HOLD input on the rising edge of CLK2 corresponding to the beginning of phase 1 of internal processor clock. HOLD must meet specified setup and hold times to be recognized at a given CLK2 edge.

The hold acknowledge state is exited in response to the HOLD input being negated. The next bus start is an idle state (Ti) if no bus request is pending, as in Figure 4–22. If a bus request is internally pending, as in Figure 4–23 and Figure 4–24, the next bus state is T1. Th is also exited in response to RESET being asserted. If HOLD remains asserted when RESET goes inactive, the TI486DLC/E enters the hold acknowledge state before performing any bus cycles provided HOLD is still asserted when the CPU is ready to perform its first bus cycle.

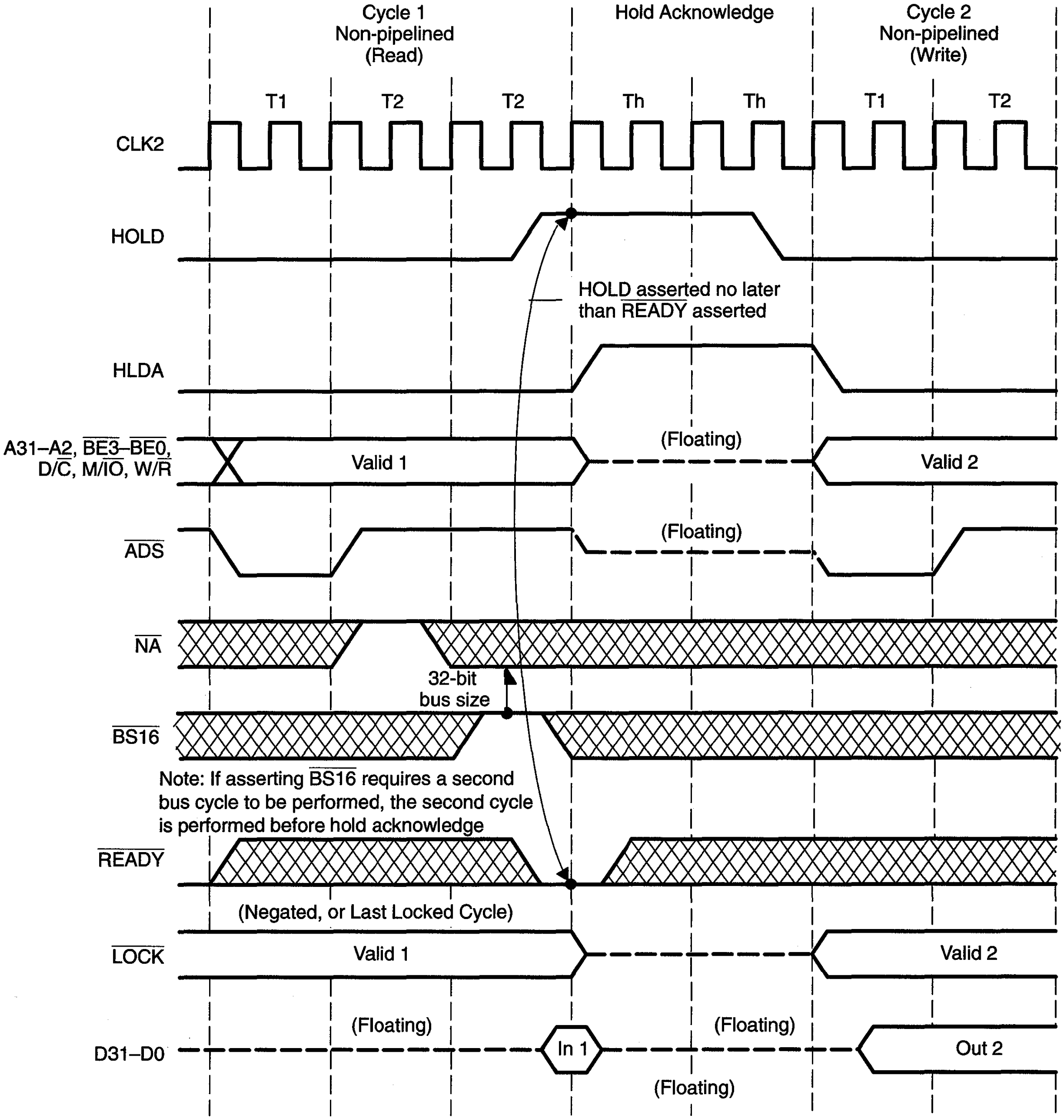
If a rising edge occurs on the edge-triggered NMI input while in Th state, the event is remembered as a non-maskable interrupt 2 and is serviced when the state is exited.

Figure 4–22. Requesting Hold from Idle Bus State


Idle
Hold Acknowledge
Idle
Ti
Th
Th
Th
Ti
CLK2
HOLD
HLDA
A31–A2, $\overline{BE3}$–$\overline{BE0}$, D/$\overline{C}$, M/$\overline{IO}$, W/$\overline{R}$
Floating
$\overline{ADS}$
Floating
$\overline{BS16}$, $\overline{NA}$, $\overline{READY}$
$\overline{LOCK}$
Floating
D31–D0
Floating

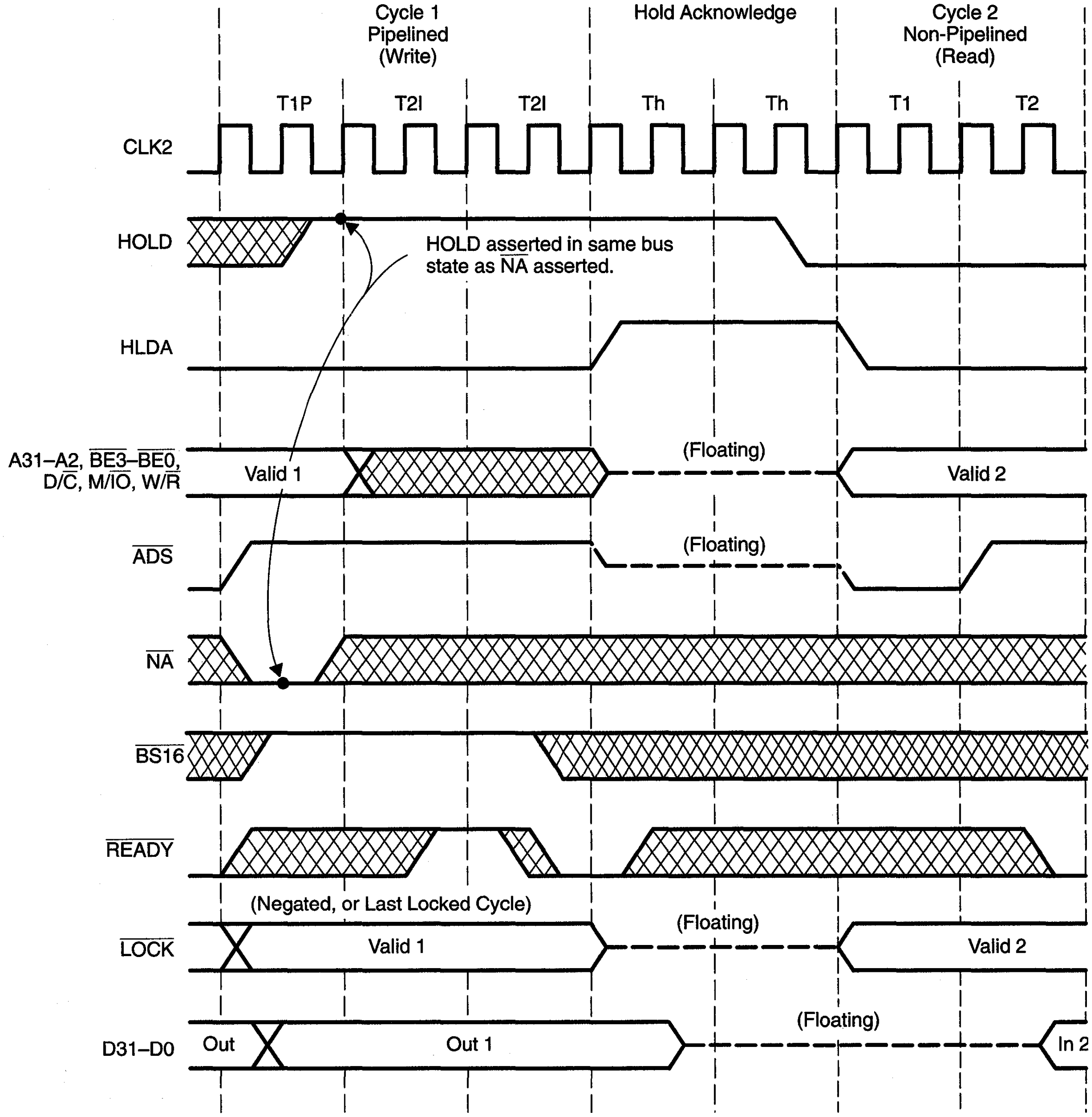
Note: For maximum design flexibility, the CPU has no internal pullup resistors on its outputs. External pullups may be required on $\overline{ADS}$ and other output to keep them negated during hold acknowledge period.

Figure 4–23. Requesting Hold from Active Non-Pipelined Bus


Cycle 1
Non-pipelined
(Read)
Hold Acknowledge
Cycle 2
Non-pipelined
(Write)
T1
T2
T2
Th
Th
T1
T2
CLK2
HOLD
HOLD asserted no later than READY asserted
HLDA
A31–A2, BE3–BE0, D/C, M/IO, W/R
Valid 1
(Floating)
Valid 2
ADS
(Floating)
NA
32-bit bus size
BS16
Note: If asserting BS16 requires a second bus cycle to be performed, the second cycle is performed before hold acknowledge
READY
(Negated, or Last Locked Cycle)
LOCK
Valid 1
Valid 2
D31–D0
(Floating)
In 1
(Floating)
Out 2
(Floating)

Note: HOLD is a synchronous input and can be asserted at any CLK2 edge, provided setup and hold requirements are met. This waveform is useful for determining hold acknowledge latency.

Figure 4–24. Requesting Hold from Active Pipelined Bus



Note: HOLD is a synchronous input and can be asserted at any CLK2 edge, provided setup and hold requirements are met. This waveform is useful for determining hold acknowledge latency.

4.2.10 Coprocessor Interface

The coprocessor interface consists of the data bus, address bus, bus cycle definition signals, and the coprocessor interface signals (BUSY, ERROR and PEREQ). The TI486DLC/E automatically accesses dedicated coprocessor I/O address 8000 00F8h and 80 00FCh to transfer opcodes and operands to or from the coprocessor whenever a coprocessor instruction is decoded. Coprocessor cycles can be either read or write and can be either non-pipelined or pipelined. Coprocessor cycles must be terminated by READY and, as with any other bus cycle, can be terminated as early as the second bus state of the cycle.

BUSY, ERROR, and PEREQ are asynchronous level-sensitive inputs used to synchronize CPU and coprocessor operation. All three signals are sampled at the beginning of phase 1 and must meet specified setup and hold times to be recognized at a given CLK2 edge.

4.2.11 SMM Interface

System Management Mode (SMM) uses two TI486DLC/E pins, SMI and SMADS. the bidirectional SMI pin is a non-maskable interrupt that is higher priority than the NMI input. SMI must be active for at least four CLK2 periods to be recognized by the TI486DLC/E. Once the TI486DLC/E recognizes the active SMI input, the CPU drives the SMI pin low for the duration of the SMI service routine.

The SMADS pin outputs the SMM Address Strobe that indicates a SMM memory bus cycle is in progress and a valid SMM address is on the address bus. The SMADS functional timing, output delay times and float delay times are identical to the main memory address strobe (ADS) timing.

4.2.11.1 SMI Handshake

The functional timing for SMI interrupt is shown in Figure 4–25. Five significant events take place during a TI486DLC/E SMI handshake:

1) The SMI input pin is driven active (low) by the system logic.

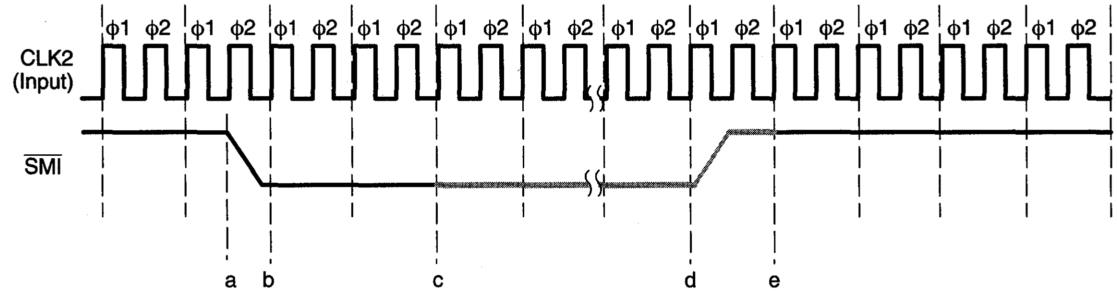
2) The CPU samples SMI active on the rising edge of CLK2 phase 1.

3) Four CLK2s after sampling the SMI active, the CPU switches the SMI pin to an output and drives SMI low.

4) Following execution of the RSM instruction, the CPU drives the SMI pin high for two CLK2s indicating completion of the SMI service routine.

5) The CPU stops driving the SMI pin high and switches the SMI pin to an input in preparation for the next SMI interrupt. The system logic is responsible for maintaining the SMI pin at an inactive (high) level after the pin has been changed to an input.

Figure 4–25. $\overline{\text{SMI}}$ Timing

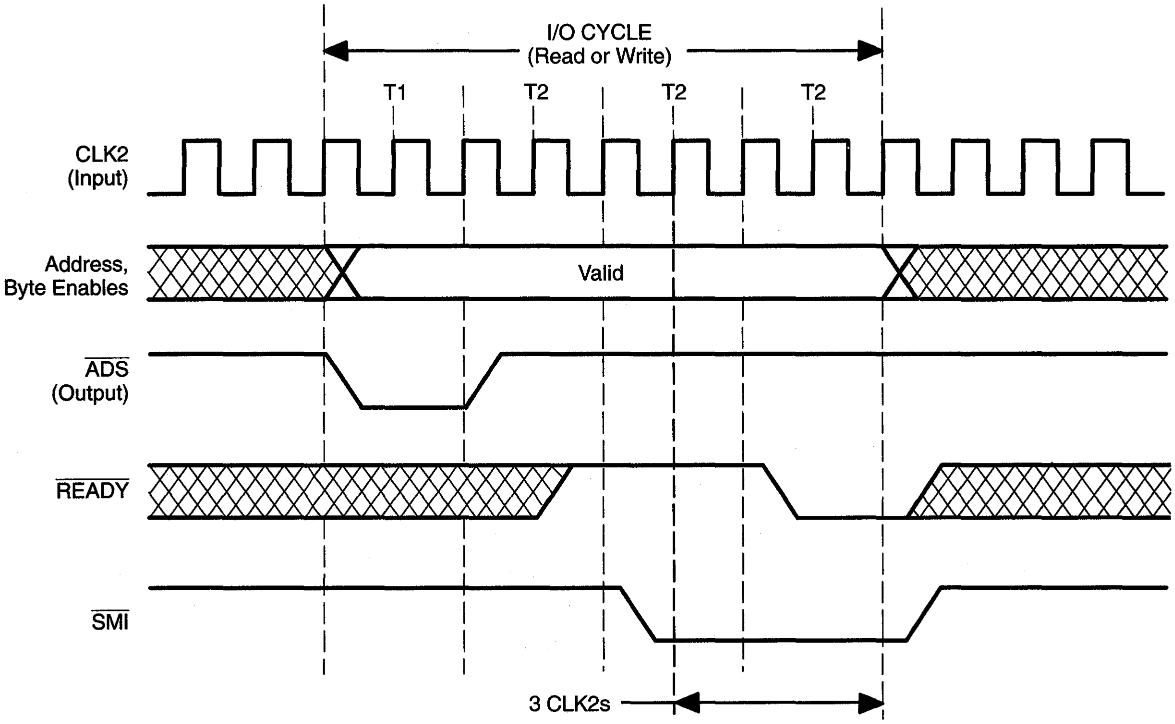

φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2 φ1 φ2
CLK2 (Input)
SMI
a
b
c
d
e

▬ Indicates that the TI486DLC/E drives the $\overline{\text{SMI}}$ pin.

4.2.11.2 I/O Trapping

The TI486DLC/E provides I/O trapping that can be used to facilitate power management of I/O peripherals. When an I/O bus cycle is issued, the I/O address is driven onto the address bus and can be decoded by external logic. If a trap to the SMI handler is required, the $\overline{\text{SMI}}$ input should be activated at least three CLK2 edges prior to returning the $\overline{\text{READY}}$ input for the I/O cycle. The timing for creating an I/O trap via the $\overline{\text{SMI}}$ input is shown in Figure 4–26. The TI486DLC/E immediately traps to the SMI interrupt handler following execution of the I/O instruction, and no other instructions are executed between completion of the I/O instruction and entering the SMI service routine. The I/O trap mechanism is not active during coprocessor accesses.

Figure 4–26. I/O Trap Timing



4.2.12 Power Management

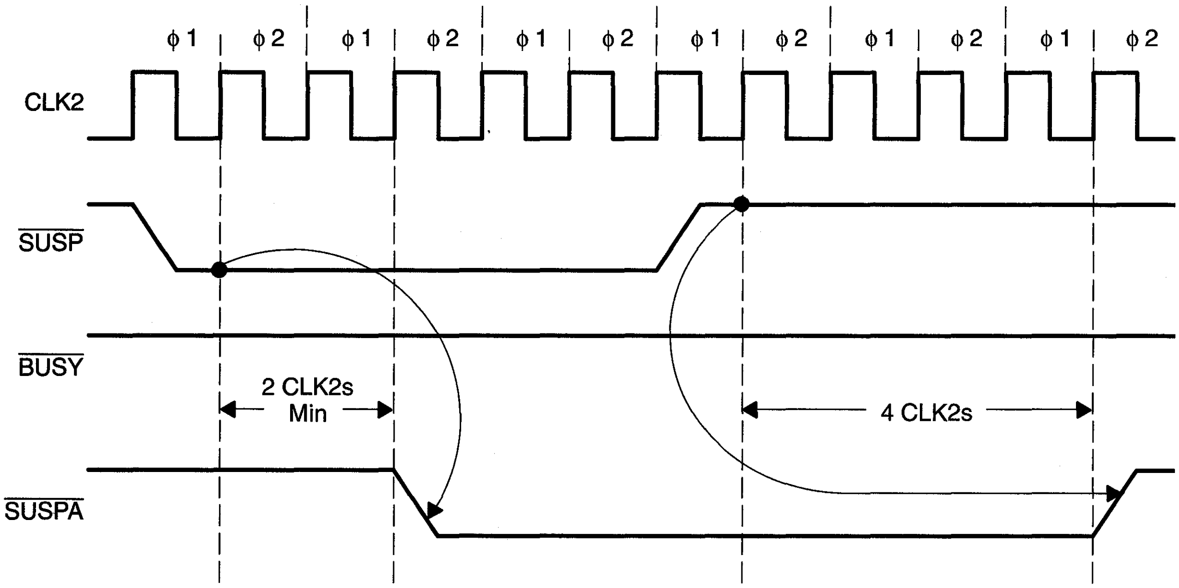
SUSP Initiated Suspend Mode

The TI486DLC/E enters suspend mode when the SUSP input is asserted and execution of the current instruction, any pending decoded instructions and associated bus cycles are completed. The TI486DLC/E also waits for the coprocessor to indicate a not busy status (BUSY=1) prior to entering suspend mode. The SUSPA output is then asserted. The TI486DLC/E responds to SUSP and asserts SUSPA only if the SUSP bit is set in the CCR0 configuration register.

Figure 4–27 illustrates the TI486DLC/E functional timing for SUSP initiated suspend mode. SUSP is sampled on the phase 2 CLK2 rising edge and must meet specified setup and hold times to be recognized at a particular CLK2 edge. The time from assertion of SUSP to activation of SUSPA varies depending on which instructions were decoded prior to assertion of SUSP. The minimum time from SUSP sampled active to SUSPA asserted is 2 CLK2s. As a maximum, the TI486DLC/E may execute up to two instructions and associated bus cycles prior to asserting SUSPA. The time required for the TI486DLC/E to deactivate SUSPA once SUSP has been sampled inactive is 4 CLK2s.

If the TI486DLC/E is in a hold acknowledge state and SUSP is asserted, the processor may or may not enter suspend mode depending on the state of the TI486DLC/E internal execution pipeline. If the TI486DLC/E is in a SUSP initiated suspend state and the CLK2 input is not stopped, the processor recognizes and acknowledges the HOLD input and stores the occurrence of FLUSH, NMI and INTR (if enabled) for execution once suspend mode is exited.

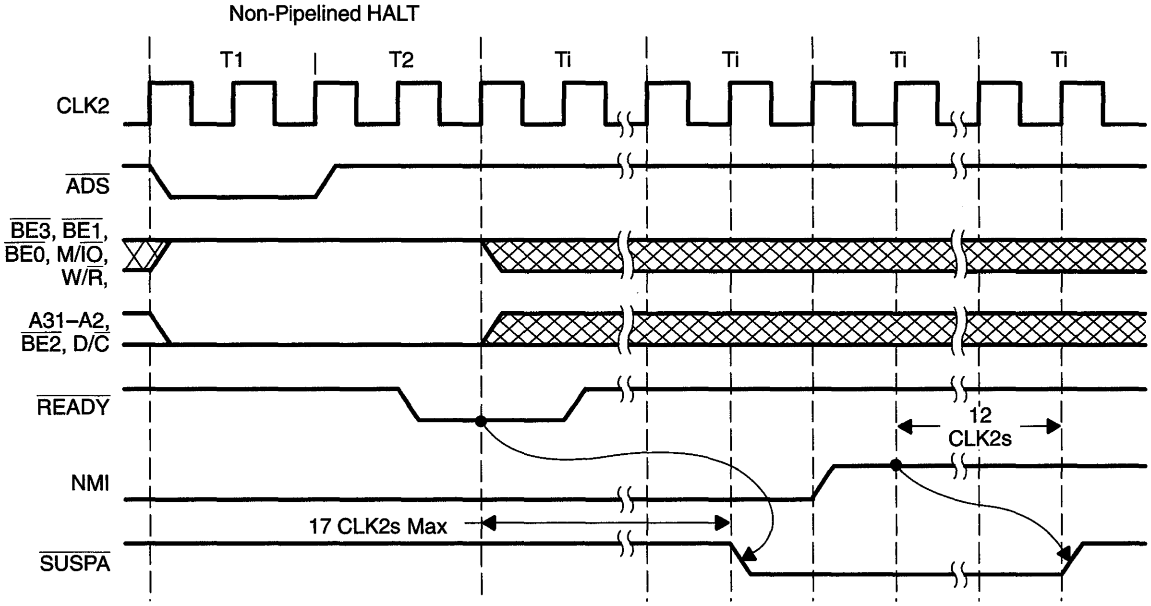
Figure 4–27. SUSP Initiated Suspend Mode



HALT Initiated Suspend Mode

The TI486DLC/E also enters suspend mode as a result of executing a HALT instruction. The SUSPA output is asserted no more than 17 CLK2s following READY sampled active for the HALT bus cycle as shown in Figure 4–28. Suspend mode is then exited upon recognition of an NMI or an unmasked INTR. SUSPA is deactivated 12 CLK2s after sampling of an active NMI or unmasked INTR. If the TI486DLC/E is in a HALT initiated suspend mode and the CLK2 input is not stopped, the processor recognizes and acknowledges the HOLD input and stores the occurrence of FLUSH for execution once suspend mode is exited.

Figure 4–28. Halt Initiated Suspend Mode

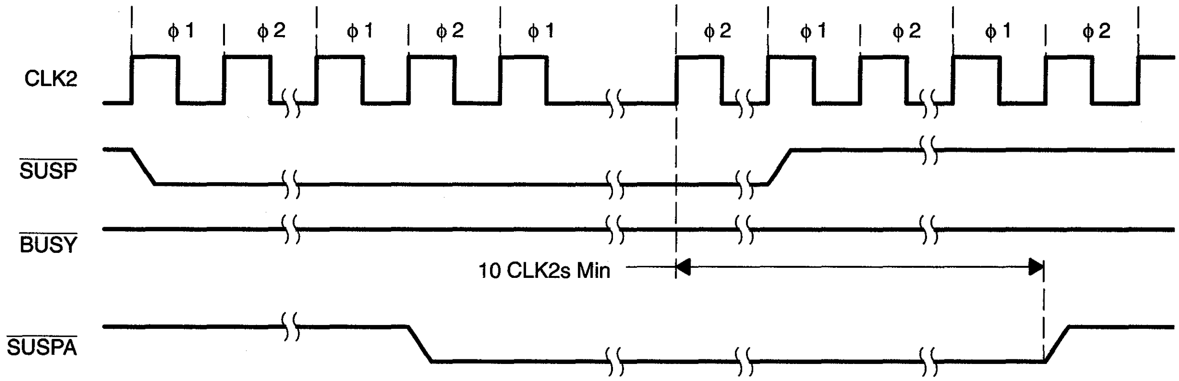


Stopping the Input Clock

Because the TI486DLC/E is a static device, the input clock (CLK2) can be stopped and restarted without loss of any internal CPU data. CLK2 can be stopped in either phase 1 or phase 2 of the clock and in either a logic high or logic low state. However, entering suspend mode prior to stopping CLK2 dramatically reduces the CPU current requirements. Therefore, the recommended sequence for stopping CLK2 is to initiate TI486DLC/E suspend mode, wait for assertion of $\overline{\text{SUSPA}}$ by the processor and then stop the input clock.

The TI486DLC/E remains suspended until CLK2 is restarted and suspend mode is exited as described above. While CLK2 is stopped, the TI486DLC/E can no longer sample and respond to any input stimulus including the HOLD, $\overline{\text{FLUSH}}$, NMI, INTR and RESET inputs. Figure 3–26 illustrates the recommended sequence for stopping CLK2 using $\overline{\text{SUSP}}$ to initiate suspend mode. CLK2 should be stable for a minimum of 10 clock periods before $\overline{\text{SUSP}}$ is deasserted.

Figure 4–29. Stopping CLK2 During Suspend Mode



Chapter 5

Electrical Specifications

Electrical specifications for the TI486 are provided in this chapter. The specifications include electrical connection requirements for all package pins, maximum ratings, recommended operating conditions, dc electrical, and ac characteristics.

Electrical connection requirements provides the designer with specific requirements for power and ground connections decoupling, termination of inputs having internal pullup/pulldown resistors, termination of system functional inputs requiring external pullup resistors, termination of unused inputs, and termination of inputs designated NC.

The absolute maximum ratings provide the designer with specific limits regarding power supply and input voltages, input and output current limits, and operating and storage temperatures.

Recommended operating conditions provide the designer with specific values for power supply and input voltages, required input threshold ranges, output drive currents available for system interfacing, and operating levels for clamp currents and case temperature.

The dc electrical characteristics provides specific data regarding the capabilities of the TI486 devices to interface directly with either CMOS or TTL type system functions.

The ac characteristics provide detailed information regarding measurement points, specific timing requirements for setup and hold times, and propagation delay times of the TI486 processors.

Topic **Page**

5.1 Electrical Connections

5.1.1 Power and Ground Connections and Decoupling

Due to the high frequency operation of the TI486, it is necessary to install and test this device using standard high-frequency techniques. The high clock frequencies used in the TI486 and its output buffer circuits can cause transient power surges when several output buffers switch output levels simultaneously. These effects can be minimized by filtering the dc power leads with low-inductance decoupling capacitors, using low-impedance wiring, and by connecting all of the V_{CC} and GND (V_{SS}) pins. There are 14 V_{CC} and 18 Vss pins on the 100-pin quad flat package, and 20 V_{CC} and 21 V_{SS} pins on the 132-pin pin grid array package.

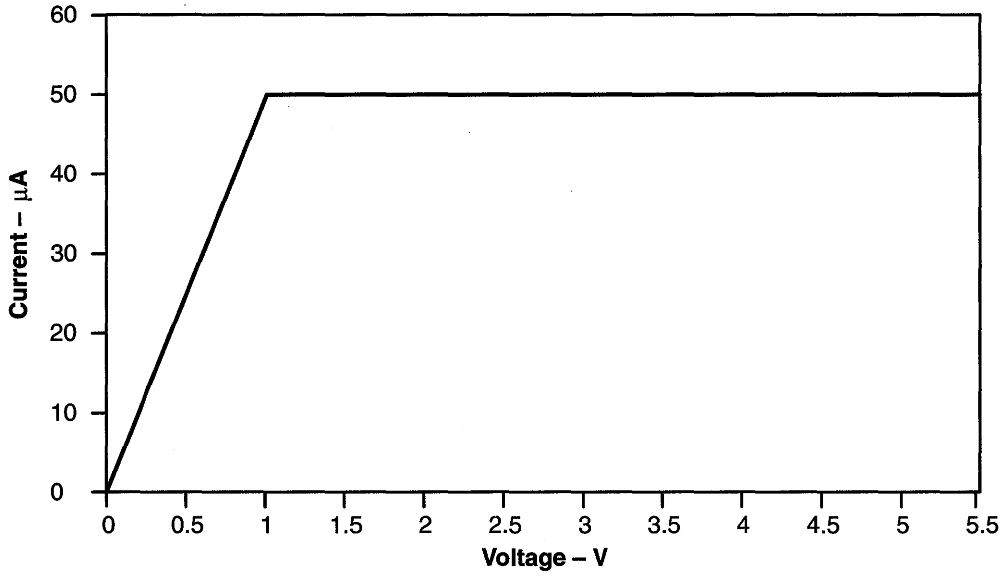
5.1.2 Pullup/Pulldown Resistors

Table 5–1 lists the input pins that are internally connected to pullup and pulldown resistors (See Figure 5–1). The pullup resistors are connected to V_{CC} and the pulldown resistors are connected to V_{SS}. When unused, these inputs do not require connection to external pullup or pulldown resistors.

Table 5–1. Pins Connected to Internal Pullup and Pulldown Resistors

SIGNAL	TI486SLC/E PIN	TI486DLC/E PIN	RESISTOR
$\overline{\text{A20M}}$	31	F13	pullup
$\overline{\text{BUSY}}$	34	B9	pullup
$\overline{\text{ERROR}}$	36	A8	pullup
$\overline{\text{FLT}}$	28	–	pullup
$\overline{\text{FLUSH}}$	30	E13	pullup
$\overline{\text{KEN}}$	29	B12	pullup
PEREQ	37	C8	pulldown
$\overline{\text{SMI}}$	47	C7	pullup
$\overline{\text{SUSP}}$	43	A4	pullup

Figure 5–1. Internal Pullup/Pulldown-IV Characteristic



It is recommended that the $\overline{\text{ADS}}$ and $\overline{\text{LOCK}}$ output pins be connected to pullup resistors, as indicated in Table 5–2. The external pullups guarantee that the signals will remain negated during hold acknowledge states.

Table 5–2. Pins Requiring External Pullup Resistors

SIGNAL	TI486SLC/E PIN	TI486DLC/E PIN	EXTERNAL RESISTOR
$\overline{\text{ADS}}$	16	E14	20-kΩ pullup
$\overline{\text{LOCK}}$	26	C10	20-kΩ pullup

5.1.3 Unused Input Pins

All inputs not used by the system designer and not listed in Table 5–1 should be connected either to ground or to V_{CC}. Connect active-high inputs to ground through a 20-kΩ (± 10%) pulldown resistor and active-low inputs to V_{CC} through a 20-kΩ (± 10%) pullup resistor to prevent possible spurious operation.

5.1.4 NC Designated Pins

Pins designated NC should be left disconnected. Connecting an NC pin to a pullup resistor, pulldown resistor, or an active signal could cause unexpected results and possible circuit malfunctions.

5.2 Absolute Maximum Ratings

Table 5–3 specifies the absolute maximum ratings for the TI486SLC/E, TI486SLC/E-V, TI486DLC/E, and TI486DLC/E-V microprocessors.

Table 5–3. Absolute Maximum Ratings Over Operating Free-Air Temperature Range (Unless Otherwise Noted)†

PARAMETER		MIN	MAX	UNIT
Supply voltage, V_{CC}	With respect to V_{SS}	–0.5	6.5	V
Voltage on any pin	With respect to V_{SS}	–0.5	V_{CC}+0.5	V
Input clamp current, I_{IK}	Power applied		10	mA
Output clamp current, I_{OK}	Power applied		25	mA
Case temperature	Power applied	–65	110	°C
Storage temperature	No bias	–65	150	°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

5.3 Recommended Operating Conditions

Table 5–4 and Table 5–5 presents the recommended operating conditions for the TI486SLC/E, TI486SLC/E-V, TI486DLC/E, and TI486DLC/E-V processors.

Table 5–4. TI486 SLC/E Recommended Operating Conditions

PARAMETER			TI486SLC/E		TI486SLC/E-V		UNIT
			MIN	MAX	MIN	MAX	
V_{CC}	Supply voltage	With respect to V_{SS}	4.75	5.25	3	3.6	V
V_{IH}	High-level input voltage		2	V_{CC}+0.3	2	V_{CC}+0.3	V
V_{IL}	Low-level input voltage		–0.3	0.8	–0.3	0.6	V
V_{ILC}	CLK2 low-level input voltage		–0.3	0.8	–0.3	0.5	V
V_{IHC}	CLK2 high-level input voltage		3.7	V_{CC}+0.3	V_{CC}–0.5	V_{CC}+0.3	V
I_{OH}	High-level output current	$V_{OH}=V_{OH(min)}$		–1		–1	mA
I_{OL}	Low-level output current	$V_{OL}=V_{OL(max)}$		5		3	mA
I_{IK}	Input clamp current	$V_{IN}<V_{SS}$ or $V_{IN}>V_{CC}$		10		10	mA
I_{OK}	Output clamp current	$V_{OUT}<V_{SS}$ or $V_{OUT}>V_{CC}$		25		25	mA
t_c	Case temperature	Power applied	0	100	0	85	°C

Table 5–5. TI486DLC/E Recommended Operating Conditions

PARAMETER			TI486DLC/E		TI486DLC/E-V		UNIT
			MIN	MAX	MIN	MAX	
V_{CC}	Supply voltage	With respect to V_{SS}	4.75	5.25	3	3.6	V
V_{IH}	High-level input voltage		2	V_{CC}+0.3	2	V_{CC}+0.3	V
V_{IL}	Low-level input voltage		–0.3	0.8	–0.3	0.6	V
V_{ILC}	CLK2 low-level input voltage		–0.3	0.8	–0.3	0.5	V
V_{IHC}	CLK2 high-level input voltage		3.7	V_{CC}+0.3	V_{CC}–0.5	V_{CC}+0.3	V
I_{OH}	High-level output current	$V_{OH}=V_{OH(min)}$		–1		–1	mA
I_{OL}	Low-level output current	$V_{OL}=V_{OL(max)}$		5		3	mA
I_{IK}	Input clamp current	$V_{IN}<V_{SS}$ or $V_{IN}>V_{CC}$		10		10	mA
I_{OK}	Output clamp current	$V_{OUT}<V_{SS}$ or $V_{OUT}>V_{CC}$		25		25	mA
t_c	Case temperature	Power applied	0	85	0	85	°C

5.4 DC Electrical Characteristics

Table 5–6 and Table 5–7 presents the dc electrical characteristics for the TI486SLC/E, TI486SLC/E-V, TI486DLC/E, and TI486DLC/E-V processors.

Table 5–6. TI486SLC/E DC Electrical Characteristics at Recommended Operating Conditions (Typical values are at nominal V_{CC} (5 V or 3.3 V) and T_A = 25°C)

PARAMETER		TEST CONDITIONS	TI486SLC/E			TI486SLC/E-V			UNIT
			MIN	TYP	MAX	MIN	TYP	MAX	
V_{OL}	Low-level output voltage	I_{OL} = 3 mA						0.35	V
		I_{OL} = 5 mA			0.45				
V_{OH}	High-level output voltage	I_{OH} = –1 mA	2.4			V_{CC}–0.4			V
		I_{OH} = –0.2 mA	V_{CC}–0.5			V_{CC}–0.4			
I_I	Input current (leakage)	0 < V_{IN} < V_{CC}, See Note 1			±15			±15	µA
I_{IH}	High-level input current at PEREQ	V_{IN} = 2.4, See Note 2			200			200	µA
I_{IL}	Low-level input current	V_{IL} = 0.45 V, See Note 3			–400			–400	µA
I_{CC}	Supply current (Active mode)	25 MHz (CLK2 = 50 MHz)		395	495		225	285	mA
		33 MHz (CLK2 = 66 MHz)		495	615		—	—	
I_{CCSM}	Supply current (Suspend mode)	25 MHz (CLK2 = 50 MHz)		9	15		6	10	mA
		33 MHz (CLK2 = 66 MHz)		10	18		—	—	
I_{CCSS}	Standby supply current	0 MHz, Suspended/ CLK2 stopped, See Note 3		0.4	2		0.3	2	mA
C_{IN}	Input capacitance	f_C = 1 MHz, See Note 5			10			10	pF
C_{OUT}	Output or I/O capacitance	f_C = 1 MHz, See Note 5			12			12	pF
C_{CLK}	Input capacitance CLK2	f_C = 1 MHz, See Note 5			20			20	pF

Notes:

1) Applicable for all input pins except those listed in Note 3.
2) PEREQ input has an internal pulldown resistor.
3) Applicable for $\overline{A20M}$, $\overline{BUSY}$, $\overline{ERROR}$, $\overline{FLT}$, $\overline{FLUSH}$, $\overline{KEN}$, $\overline{SMI}$, and $\overline{SUSP}$ inputs that have an internal pullup resistor.
4) All inputs at 0.4 or V_{CC}–0.4 (CMOS levels). All inputs held static, (except CLK2 as indicated). All outputs unloaded (static I_{OUT} = 0 mA).
5) Not 100% tested.

Table 5–7. TI486DLC/E DC Electrical Characteristics at Recommended Operating Conditions (Typical values are at nominal V_{CC} (5 V or 3.3 V) and T_A = 25°C)

PARAMETER		TEST CONDITIONS	TI486DLC/E			TI486DLC/E-V			UNIT
			MIN	TYP	MAX	MIN	TYP	MAX	
V_{OL}	Low-level output voltage	I_{OL} = 3 mA						0.35	V
		I_{OL} = 5 mA			0.45				
V_{OH}	High-level output voltage	I_{OH} = –1 mA	2.4			V_{CC}–0.4			V
		I_{OH} = –0.2 mA	V_{CC}–0.5			V_{CC}–0.4			
I_I	Input current (leakage)	0 < V_{IN} < V_{CC}, See Note 1			±15			±15	µA
I_{IH}	High-level input current at PEREQ	V_{IN} = 2.4, See Note 2			200			200	µA
I_{IL}	Low-level input current	V_{IL} = 0.45 V, See Note 3			–400			–400	µA
I_{CC}	Supply current (Active mode)	25 MHz (CLK2 = 50 MHz)		—	—		240	305	mA
		33 MHz (CLK2 = 66 MHz)		520	650		300	375	
		40 MHz (CLK2 = 80 MHz)		560	700		—	—	
I_{CCSM}	Supply current (Suspend mode)	25 MHz (CLK2 = 50 MHz)		—	—		6	10	mA
		33 MHz (CLK2 = 66 MHz)		7.5	15		7	12	
		40 MHz (CLK2 = 80 MHz)		10	20		—	—	
I_{CCSS}	Standby supply current	0 MHz, Suspended/ CLK2 stopped, See Note 3		0.4	2		0.3	2	mA
C_{IN}	Input capacitance	f_C = 1 MHz, See Note 5			10			10	pF
C_{OUT}	Output or I/O capacitance	f_C = 1 MHz, See Note 5			12			12	pF
C_{CLK}	Input capacitance CLK2	f_C = 1 MHz, See Note 5			20			20	pF

Notes:

1) Applicable for all input pins except those listed in Note 3.
2) PEREQ input has an internal pulldown resistor.
3) Applicable for $\overline{\text{A20M}}$, $\overline{\text{BUSY}}$, $\overline{\text{ERROR}}$, $\overline{\text{FLUSH}}$, $\overline{\text{KEN}}$, $\overline{\text{SMI}}$, and $\overline{\text{SUSP}}$ inputs that have an internal pullup resistor.
4) All inputs at 0.4 or V_{CC}–0.4 (CMOS levels). All inputs held static, (except CLK2 as indicated). All outputs unloaded (static I_{OUT} = 0 mA).
5) Not 100% tested.

5.5 AC Characteristics

5.5.1 Measurement Points for Switching Characteristics

The rising clock edge reference level V_{REFC}, and other reference levels are specified in Table 5–8 for the TI486SLC/E, TI486SLC/E-V, TI486DLC/E, and TI486DLC/E-V. Input or output signals must cross these levels during testing. Table 5–9, Table 5–10, Table 5–11, and , Table 5–12 list the ac characteristics including output delays, input setup requirements, input hold requirements, and output float delays. These measurements are based on the measurement points identified in Figure 5–2, Figure 5–3, and Figure 5–4.

Figure 5–2 and Figure 5–3 show delays (A and B) and input setup and hold times (C and D). Input setup and hold times (C and D) are specified minimums, defining the smallest acceptable sampling window a synchronous input signal must be stable for correct operation.

The TI486SLC/E and TI486SLC/E-V outputs A23–A1, $\overline{\text{ADS}}$, $\overline{\text{BHE}}$, $\overline{\text{BLE}}$, D/$\overline{\text{C}}$, HLDA, $\overline{\text{LOCK}}$, M/$\overline{\text{IO}}$, $\overline{\text{SMADS}}$, $\overline{\text{SMI}}$, and W/$\overline{\text{R}}$ change only at the beginning of phase one (Figure 5–2, ϕ1). Outputs D15–D0 (write cycles) and $\overline{\text{SUSPA}}$ change at the beginning of phase two, ϕ2.

The TI486SLC/E and TI486SLC/E-V inputs $\overline{\text{BUSY}}$, D15–D0 (read cycles), $\overline{\text{ERROR}}$, $\overline{\text{FLT}}$, HOLD, PEREQ, and $\overline{\text{READY}}$ are sampled at the beginning of phase one (Figure 5–2, ϕ1). Inputs $\overline{\text{A20M}}$, $\overline{\text{FLUSH}}$, INTR, $\overline{\text{KEN}}$, $\overline{\text{NA}}$, NMI, $\overline{\text{SMI}}$ and $\overline{\text{SUSP}}$ are sampled at the beginning of phase two, ϕ2.

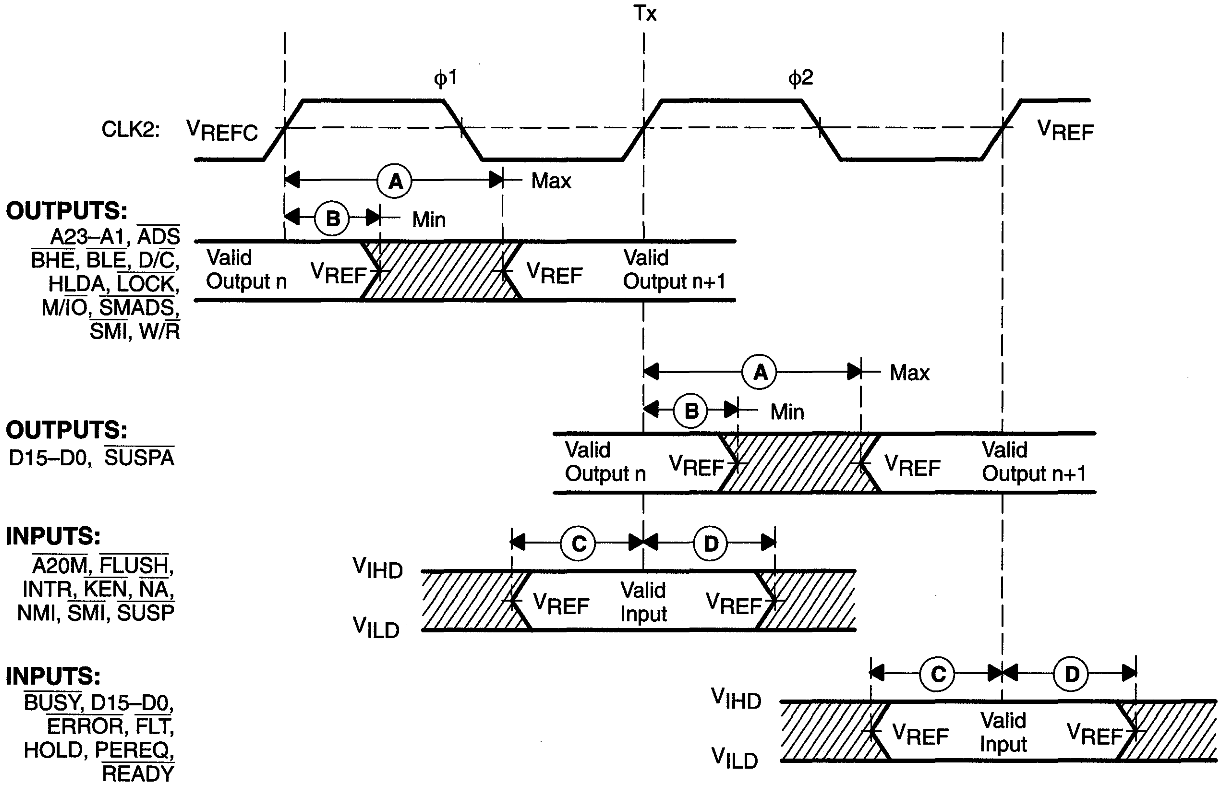
The TI486DLC/E and TI486DLC/E-V outputs A31–A2, $\overline{\text{ADS}}$, $\overline{\text{BE3}}$ – $\overline{\text{BE0}}$, D/$\overline{\text{C}}$, HLDA, $\overline{\text{LOCK}}$, M/$\overline{\text{IO}}$, $\overline{\text{SMADS}}$, $\overline{\text{SMI}}$, and W/$\overline{\text{R}}$ change only at the beginning of phase one (Figure 5–3, ϕ1). Outputs D31–D0 (write cycles) and $\overline{\text{SUSPA}}$ change at the beginning of phase two, ϕ2.

The TI486DLC/E and TI486DLC/E-V inputs $\overline{\text{BUSY}}$, D31–D0 (read cycles), $\overline{\text{ERROR}}$, HOLD, PEREQ, and $\overline{\text{READY}}$ are sampled at the beginning of phase one (Figure 5–3, ϕ1). Inputs $\overline{\text{A20M}}$, $\overline{\text{BS16}}$, $\overline{\text{FLUSH}}$, INTR, $\overline{\text{KEN}}$, $\overline{\text{NA}}$, NMI, $\overline{\text{SMI}}$ and $\overline{\text{SUSP}}$ are sampled at the beginning of phase two, ϕ2.

Table 5–8. Measurement Points for Switching Characteristics

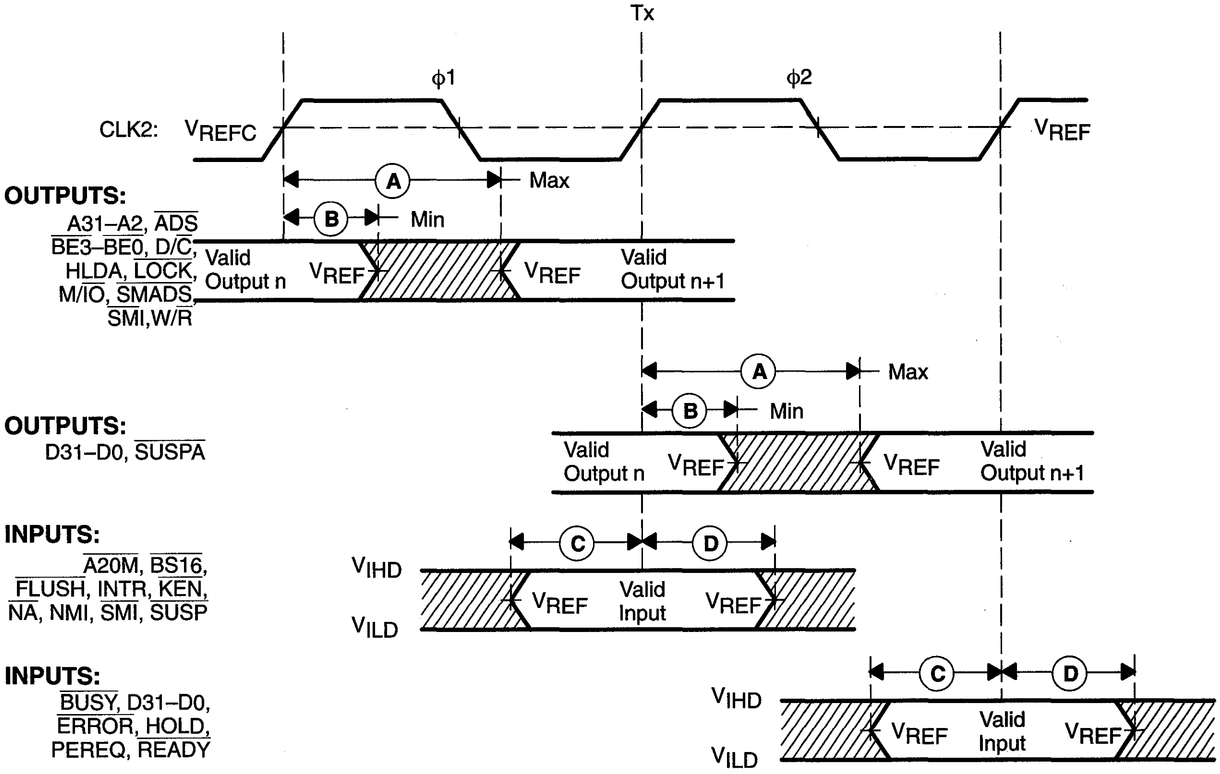
SYMBOL	TI486SLC/E	TI486SLC/E-V	TI486DLC/E	TI486DLC/E-V	UNIT
V_{REFC}	2	1.5	2	1.5	V
V_{REF}	1.5	1.2	1.5	1.2	V
V_{IHC}	V_{CC}–0.8	V_{CC}–0.5	V_{CC}–0.8	V_{CC}–0.5	V
V_{ILC}	0.8	0.6	0.8	0.6	V
V_{IHD}	3	2.3	3	2.3	V
V_{ILD}	0	0	0	0	V

Figure 5–2. TI486SLC/E and TI486SLC/E-V Drive Level and Measurement Points for Switching Characteristics



LEGEND: A – Maximum Output Delay Specification
B – Maximum Output Delay Specification
C – Minimum Input Setup Specification
D – Minimum Input Hold Specificaton

Figure 5–3. TI486DLC/E Drive Level and Measurement Points for Switching Characteristics



LEGEND: A – Maximum Output Delay Specification
B – Maximum Output Delay Specification
C – Minimum Input Setup Specification
D – Minimum Input Hold Specificaton

5.5.2 CLK2 Timing Measurement Points

The CLK2 timing measurement points are illustrated in Figure 5–4 for the TI486SLC/E, TI486SLC/E-V, TI486DLC/E, and TI486DLC/E-V.

Figure 5–4. CLK2 Timing Measurement Points

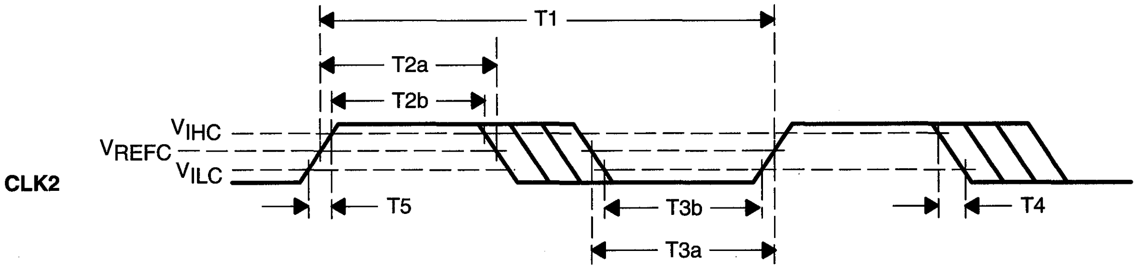


Table 5–9. AC Characteristics for TI486SLC/E-25 and TI486SLC/E-33, V_{CC} = 4.75 V to 5.25 V, T_C = 0°C to 100°C

SYMBOL	PARAMETER	TI486SLC/E-25		TI486SLC/E-33		FIGURE	NOTES
		MIN (ns)	MAX (ns)	MIN (ns)	MAX (ns)		
T1	CLK2 period	20		15		5-4	Note 1
T2a	CLK2 high time	7		6.25		5-4	Note 2
T2b	CLK2 high time	4		4.5		5-4	Note 2
T3a	CLK2 low time	7		6.25		5-4	Note 2
T3b	CLK2 low time	5		4.5		5-4	Note 2
T4	CLK2 fall time		7		4	5-4	Note 2
T5	CLK2 rise time		7		4	5-4	Note 2
T6	A23–A1 valid delay	4	21	4	15	5-7, 5-10	C_L = 50 pF
T6a	$\overline{\text{SMI}}$ valid delay	4	21	4	15	5-7, 5-10	C_L = 50 pF
T7	A23–A1 float delay	4	30	4	20	5-10	Note 3
T8	$\overline{\text{BHE}}$, $\overline{\text{BLE}}$, $\overline{\text{LOCK}}$ valid delay	4	21	4	15	5-7, 5-10	C_L = 50 pF
T9	$\overline{\text{BHE}}$, $\overline{\text{BLE}}$, $\overline{\text{LOCK}}$ float delay	4	30	4	20	5-10	Note 3
T10	$\overline{\text{ADS}}$, D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$, W/$\overline{\text{R}}$ valid delay	4	21	4	15	5-7, 5-10	C_L = 50 pF
T10a	$\overline{\text{SMADS}}$ valid delay	4	21	4	15	5-7, 5-10	C_L = 50 pF
T11	$\overline{\text{ADS}}$, D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$, W/$\overline{\text{R}}$ float delay	4	30	4	20	5-10	Note 3
T11a	$\overline{\text{SMADS}}$ float delay	4	30	4	20	5-10	Note 3
T12	D15–D0 write data, $\overline{\text{SUSPA}}$ valid delay	7	27	7	24	5-7, 5-8 5-10	C_L = 50 pF, Note 5
T12a	D15–D0 write data hold time	2		2		5-9	
T13	D15–D0 write data, $\overline{\text{SUSPA}}$ float delay	4	22	4	17	5-10	Note 3, Note 6
T14	HDLA valid delay	4	22	4	20	5-10	C_L = 50 pF
T15	$\overline{\text{NA}}$, $\overline{\text{SUSP}}$, $\overline{\text{FLUSH}}$, $\overline{\text{KEN}}$, $\overline{\text{A20M}}$ setup time	5		5		5-6	
T16	$\overline{\text{NA}}$, $\overline{\text{SUSP}}$, $\overline{\text{FLUSH}}$, $\overline{\text{KEN}}$, $\overline{\text{A20M}}$ hold time	3		3		5-6	
T19	$\overline{\text{READY}}$ setup time	9		7		5-6	
T20	$\overline{\text{READY}}$ hold time	4		4		5-6	
T21	D15–D0 read data setup time	7		5		5-6	
T22	D15–D0 read data hold time	5		3		5-6	
T23	HOLD setup time	9		11		5-6	
T24	HOLD hold time	3		2		5-6	
T25	RESET setup time	8		5		5-5	
T26	RESET hold time	3		2		5-5	
T27	$\overline{\text{NMI}}$, INTR setup time	6		5		5-6	Note 4
T27a	$\overline{\text{SMI}}$ setup time	6		5		5-6	Note 4
T28	$\overline{\text{NMI}}$, INTR hold time	6		5		5-6	Note 4
T28a	$\overline{\text{SMI}}$ hold time	6		5		5-6	Note 4
T29	PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ setup time	6		5		5-6	Note 4
T30	PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ hold time	5		4		5-6	Note 4

Notes:

1) Input clock can be stopped, therefore minimum CLK2 frequency is 0 MHz.
2) These parameters are not tested. They are guaranteed by design characterization.
3) Float condition occurs when maximum output current becomes less than I_l in magnitude. Float is not 100% tested.
4) These inputs are allowed to be asynchronous to CLK2. The setup and hold specifications are given for testing purposes, to assure recognition within a specific CLK2 period.
5) T12 minimum time is not 100% tested.
6) $\overline{\text{SUSPA}}$ floats only in response to activation of $\overline{\text{FLT}}$. $\overline{\text{SUSPA}}$ does not float during a hold acknowledge state.

Table 5–10. AC Characteristics for TI486SLC/E-V25, V_{CC} = 3 V to 3.6 V, T_C = 0°C to 85°C

SYMBOL	PARAMETER	TI486SLC/E-V25		FIGURE	NOTES
		MIN (ns)	MAX (ns)		
T1	CLK2 period	20		5-4	Note 1
T2a	CLK2 high time	7		5-4	Note 2
T2b	CLK2 high time	4		5-4	Note 2
T3a	CLK2 low time	7		5-4	Note 2
T3b	CLK2 low time	5		5-4	Note 2
T4	CLK2 fall time		7	5-4	Note 2
T5	CLK2 rise time		7	5-4	Note 2
T6	A23–A1 valid delay	3	21	5-7, 5-10	C_L = 50 pF
T6a	SMI valid delay	3	21	5-7, 5-10	C_L = 50 pF
T7	A23–A1 float delay	4	30	5-10	Note 3
T8	BHE, BLE, LOCK valid delay	2.5	18	5-7, 5-10	C_L = 50 pF
T9	BHE, BLE, LOCK float delay	4	30	5-10	Note 3
T10	ADS, D/C, M/IO, W/R valid delay	4	19	5-7, 5-10	C_L = 50 pF
T10a	SMADS valid delay	4	19	5-7,5-10	C_L = 50 pF
T11	ADS, D/C, M/IO, W/R float delay	4	30	5-10	Note 3
T11a	SMADS float delay	4	30	5-10	Note 3
T12	D15–D0 write data, SUSPA valid delay	3.5	27	5-7, 5-8	C_L = 50 pF, Note 5
T12a	D15–D0 write data hold time	2		5-9	
T13	D15–D0 write data, SUSPAfloat delay	4	22	5-10	Note 3, Note 6
T14	HDLA valid delay	2	22	5-10	C_L = 50 pF
T15	NA, SUSP, FLUSH, KEN, A20M setup time	5		5-6	
T16	NA, SUSP, FLUSH, KEN, A20M hold time	3.5		5-6	
T19	READY setup time	9		5-6	
T20	READY hold time	4		5-6	
T21	D15–D0 read data setup time	7		5-6	
T22	D15–D0 read data hold time	5		5-6	
T23	HOLD setup time	9		5-6	
T24	HOLD hold time	3.5		5-6	
T25	RESET setup time	8		5-5	
T26	RESET hold time	3		5-5	
T27	NMI, INTR setup time	6		5-6	Note 4
T27a	SMI setup time	6		5-6	Note 4
T28	NMI, INTR hold time	6		5-6	Note 4
T28a	SMI hold time	6		5-6	Note 4
T29	PEREQ, ERROR, BUSY setup time	6		5-6	Note 4
T30	PEREQ, ERROR, BUSY hold time	5		5-6	Note 4

Notes:
1) Input clock can be stopped, therefore minimum CLK2 frequency is 0 MHz.
2) These parameters are not tested. They are guaranteed by design characterization.
3) Float condition occurs when maximum output current becomes less than I_I in magnitude. Float is not 100% tested.
4) These inputs are allowed to be asynchronous to CLK2. The setup and hold specifications are given for testing purposes, to assure recognition within a specific CLK2 period.
5) T12 minimum time is not 100% tested.
6) SUSPA floats only in response to activation of FLT. SUSPA does not float during a hold acknowledge state.

ADVANCE INFORMATION concerns new products in the sampling or preproduction phase of development. Characteristic data and other specifications are subject to change without notice.

Table 5–11. AC Characteristics for TI486DLC/E-33 and TI486DLC/E-40
V_{CC} = 4.75 V to 5.25 V, T_C = 0°C to 85°C

SYMBOL	PARAMETER	TI486DLC/E-33		TI486DLC/E-40		FIGURE	NOTES
		MIN (ns)	MAX (ns)	MIN (ns)	MAX (ns)		
T1	CLK2 period	15		12.5		5-4	Note 1
T2a	CLK2 high time	6.25		5		5-4	Note 2
T2b	CLK2 high time	4.5		3.25		5-4	Note 2
T3a	CLK2 low time	6.25		5		5-4	Note 2
T3b	CLK2 low time	4.5		3.25		5-4	Note 2
T4	CLK2 fall time		4		4	5-4	Note 2
T5	CLK2 rise time		4		4	5-4	Note 2
T6	A31–A2 valid delay	4	15	3	12.5	5-12, 5-15	C_L = 50 pF
T6a	SMI valid delay	4	15	3	12.5	5-12, 5-15	C_L = 50 pF
T7	A31–A2 float delay	4	20	3	17	5-15	Note 3
T8	BE3 – BE0, LOCK valid delay	4	15	3	12.5	5-15, 5-15	C_L = 50 pF
T9	BE3 – BE0, LOCK float delay	4	20	3	17	5-15	Note 3
T10	ADS, D/C, M/IO, W/R valid delay	4	15	3	12.5	5-12, 5-15	C_L = 50 pF
T10a	SMADS valid delay	4	15	3	12.5	5-12, 5-15	C_L = 50 pF
T11	ADS, D/C, M/IO, W/R float delay	4	20	3	17	5-15	Note 3
T11a	SMADS float delay	4	20	3	17	5-15	Note 3
T12	D31–D0 write data, SUSPA valid delay	7	24	5	20	5-12, 5-13	C_L = 50 pF, Note 5
T12a	D31–D0 write data hold time	2		2		5-14	
T13	D31–D0 write data, SUSPA float delay	4	17	3	14.5	5-15	Note 3
T14	HDLA valid delay	4	20	3	17	5-15	C_L = 50 pF
T15	A20M, FLUSH, KEN, NA, SUSP setup time	5		5		5-11	
T16	A20M, FLUSH, KEN, NA, SUSP hold time	2		2		5-11	
T17	BS16 setup time	5		5		5-11	
T18	BS16 hold time	2		2		5-11	
T19	READY setup time	7		5		5-11	
T20	READY hold time	4		3		5-11	
T21	D31–D0 read data setup time	5		5		5-11	
T22	D31–D0 read data hold time	3		3		5-11	
T23	HOLD setup time	7		4		5-11	
T24	HOLD hold time	2		2		5-11	
T25	RESET setup time	5		4.5		5-5	
T26	RESET hold time	2		2		5-5	
T27	NMI, INTR setup time	5		5		5-11	Note 4
T27a	SMI setup time	5		5		5-11	Note 4
T28	NMI, INTR hold time	5		5		5-11	Note 4
T28a	SMI hold time	5		5		5-11	Note 4
T29	PEREQ, ERROR, BUSY setup time	5		5		5-11	Note 4
T30	PEREQ, ERROR, BUSY hold time	4		3		5-11	Note 4

Notes:
1) Input clock can be stopped, therefore minimum CLK2 frequency is 0 MHz.
2) These parameters are not tested. They are guaranteed by design characterization.
3) Float condition occurs when maximum output current becomes less than I_L in magnitude. Float is not 100% tested.
4) These following inputs are allowed to be asynchronous to CLK2. The setup and hold specifications are given for testing purposes, to assure recognition within a specific CLK2 period.
5) T12 minimum time is not 100% tested.

Table 5–12. AC Characteristics for TI486DLC/E-V25 and TI486DLC/E-V33 V_{CC} = 3 V to 3.6 V, T_C = 0°C to 85°C

SYMBOL	PARAMETER	TI486DLC/E-V25		TI486DLC/E-V33		FIGURE	NOTES
		MIN (ns)	MAX (ns)	MIN (ns)	MAX (ns)		
T1	CLK2 period	20		15		5-4	Note 1
T2a	CLK2 high time	7		6.25		5-4	Note 2
T2b	CLK2 high time	4		4.5		5-4	Note 2
T3a	CLK2 low time	7		6.25		5-4	Note 2
T3b	CLK2 low time	5		4.5		5-4	Note 2
T4	CLK2 fall time		7		4	5-4	Note 2
T5	CLK2 rise time		7		4	5-4	Note 2
T6	A31–A2 valid delay	3	21	3	15	5-12, 5-15	C_L = 50 pF
T6a	$\overline{\text{SMI}}$ valid delay	3	21	3	15	5-12, 5-15	C_L = 50 pF
T7	A31–A2 float delay	4	30	4	20	5-15	Note 3
T8	$\overline{\text{BE3}}$ – $\overline{\text{BE0}}$, $\overline{\text{LOCK}}$ valid delay	2.5	18	2.5	18	5-12, 5-15	C_L = 50 pF
T9	$\overline{\text{BE3}}$ – $\overline{\text{BE0}}$, $\overline{\text{LOCK}}$ float delay	4	30	4	20	5-15	Note 3
T10	$\overline{\text{ADS}}$, D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$, W/$\overline{\text{R}}$ valid delay	4	19	4	19	5-12, 5-15	C_L = 50 pF
T10a	$\overline{\text{SMADS}}$ valid delay	4	19	4	19	5-12, 5-15	C_L = 50 pF
T11	$\overline{\text{ADS}}$, D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$, W/$\overline{\text{R}}$ float delay	4	30	4	20	5-15	Note 3
T11a	$\overline{\text{SMADS}}$ float delay	4	30	4	20	5-15	Note 3
T12	D31–D0 write data, $\overline{\text{SUSPA}}$ valid delay	3.5	27	3.5	24	5-12, 5-13	C_L = 50 pF, Note 5
T12a	D31–D0 write data hold time	2		2		5-14	
T13	D31–D0 write data, $\overline{\text{SUSPA}}$ float delay	4	22	4	17	5-15	Note 3
T14	HDLA valid delay	2	22	2	20	5-15	C_L = 50 pF
T15	$\overline{\text{A20M}}$, $\overline{\text{FLUSH}}$, $\overline{\text{KEN}}$, $\overline{\text{NA}}$, $\overline{\text{SUSP}}$ setup time	5		5		5-11	
T16	$\overline{\text{A20M}}$, $\overline{\text{FLUSH}}$, $\overline{\text{KEN}}$, $\overline{\text{NA}}$, $\overline{\text{SUSP}}$ hold time	3.5		3.5		5-11	
T17	$\overline{\text{BS16}}$ setup time	7		5		5-11	
T18	$\overline{\text{BS16}}$ hold time	2		2		5-11	
T19	$\overline{\text{READY}}$ setup time	9		7		5-11	
T20	$\overline{\text{READY}}$ hold time	4		4		5-11	
T21	D31–D0 read data setup time	7		7		5-11	
T22	D31–D0 read data hold time	5		4		5-11	
T23	HOLD setup time	15		12		5-11	
T24	HOLD hold time	4		4		5-11	
T25	RESET setup time	8		5		5-4	
T26	RESET hold time	3		2		5-4	
T27	$\overline{\text{NMI}}$, INTR setup time	6		5		5-10	Note 4
T27a	$\overline{\text{SMI}}$ setup time	6		5		5-10	Note 4
T28	$\overline{\text{NMI}}$, INTR hold time	6		5		5-10	Note 4
T28a	$\overline{\text{SMI}}$ hold time	6		5		5-10	Note 4
T29	PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ setup time	6		5		5-10	Note 4
T30	PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ hold time	5		4		5-10	Note 4

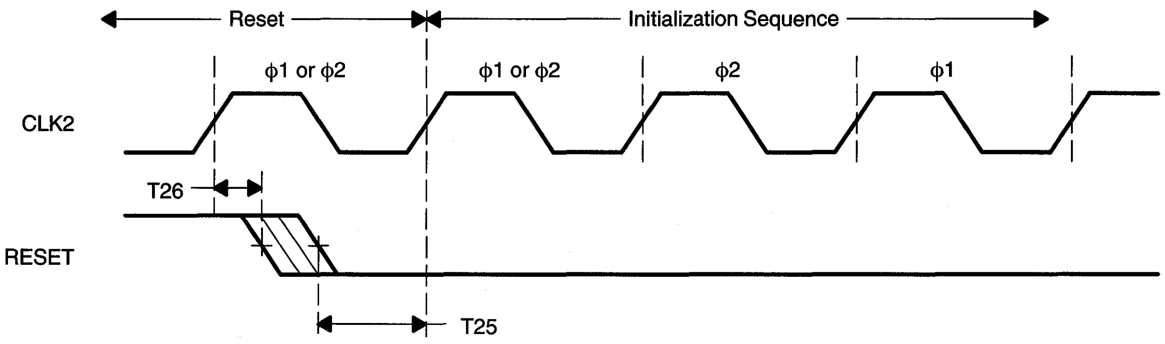
Notes:
1) Input clock can be stopped, therefore minimum CLK2 frequency is 0 MHz.
2) These parameters are not tested. They are guaranteed by design characterization.
3) Float condition occurs when maximum output current becomes less than I_l in magnitude. Float is not 100% tested.
4) These following inputs are allowed to be asynchronous to CLK2. The setup and hold specifications are given for testing purposes, to assure recognition within a specific CLK2 period.
5) T12 minimum time is not 100% tested.

ADVANCE INFORMATION concerns new products in the sampling or preproduction phase of development. Characteristic data and other specifications are subject to change without notice.

5.5.3 RESET Setup and Hold Timing

RESET and Hold timing for the TI486SLC/E, TI486SLC/E-V, and TI486DLC/E are illustrated in Figure 5–5.

Figure 5–5. RESET Setup and Hold Timing



5.5.4 TI486SLC/E and TI486SLC/E-V Switching Waveforms

Switching waveforms for the TI486SLC/E and TI486SLC/E-V are illustrated in Figure 5–6 through Figure 5–10.

Figure 5–6. TI486SLC/E and TI486SLC/E-V Input Signal Setup and Hold Timing

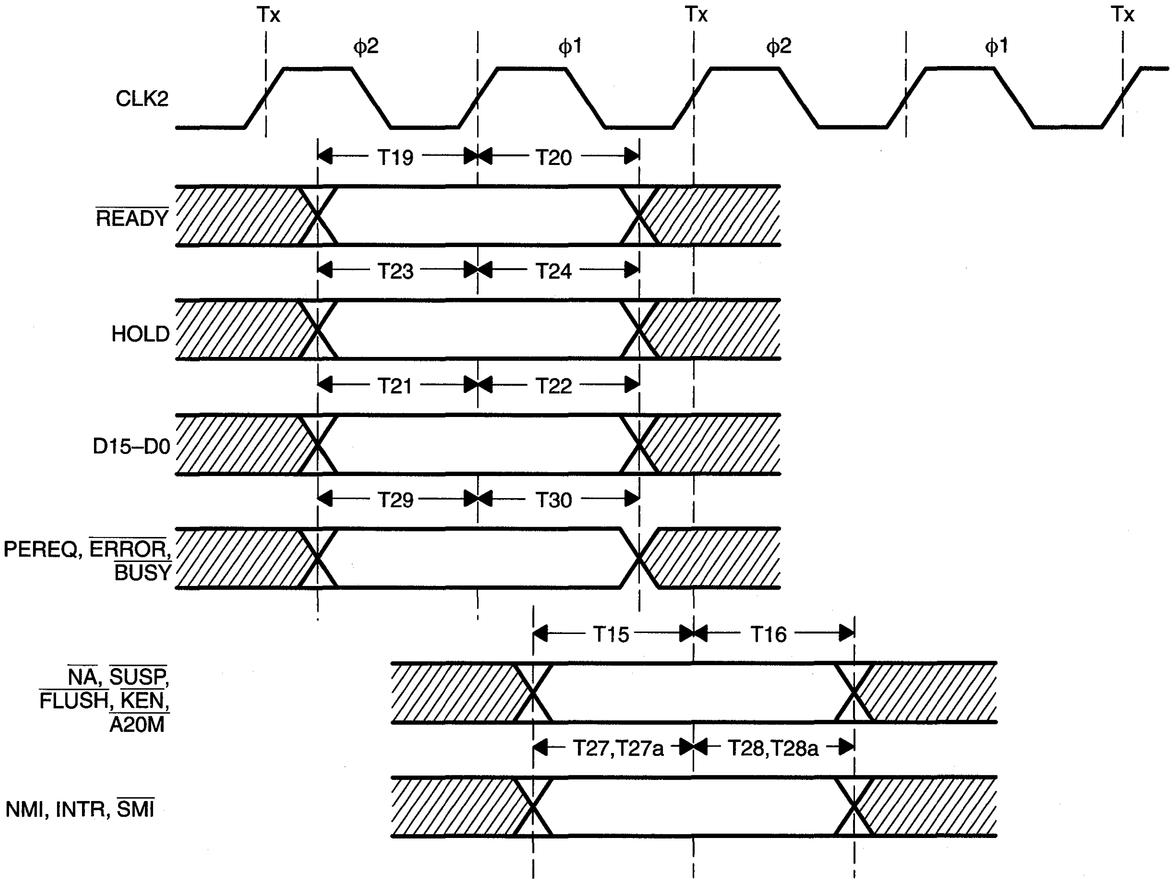


Figure 5–7. TI486SLC/E and TI486SLC/E-V Output Signal Valid Delay Timing

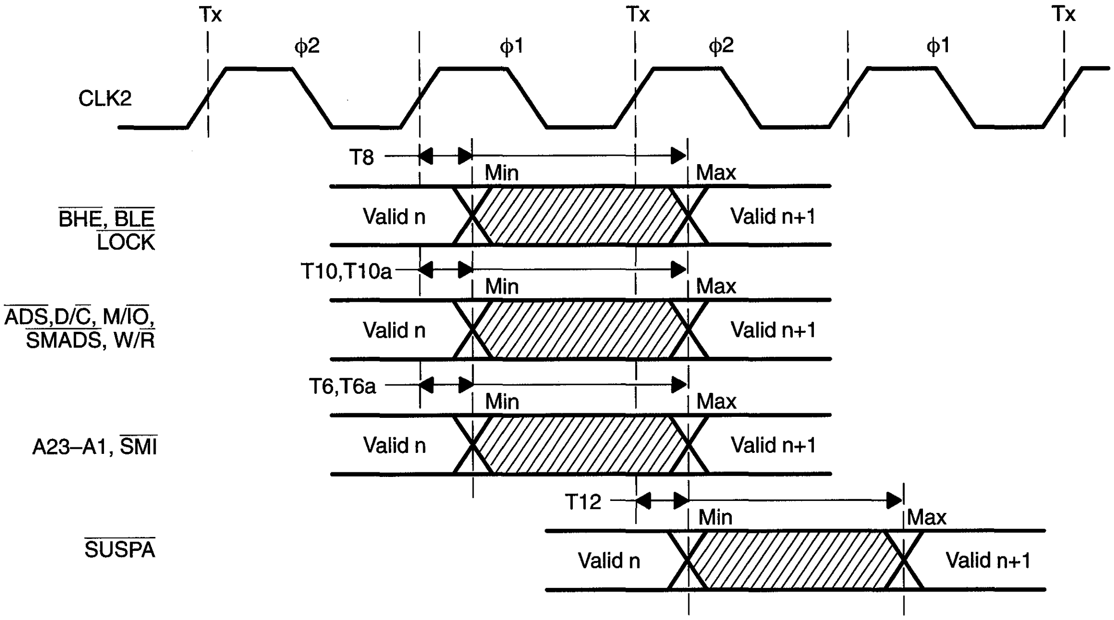


Figure 5–8. TI486SLC/E and TI486SLC/E-V Data Write Cycle Valid Delay Timing

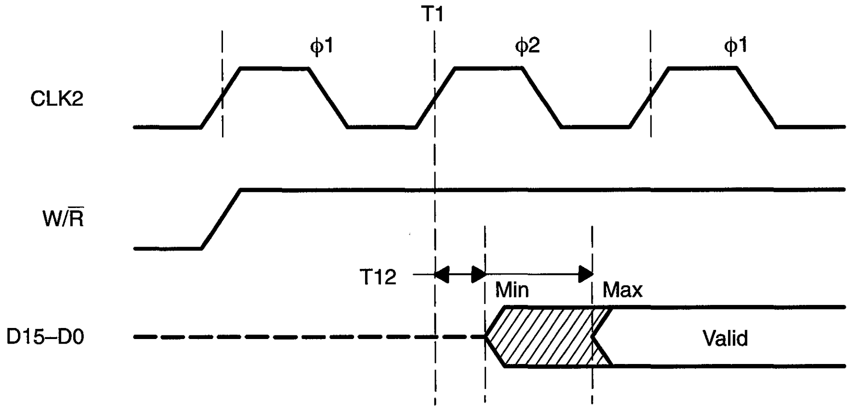


Figure 5–9. TI486SLC/E and TI486SLC/E-V Data Write Cycle Hold Timing

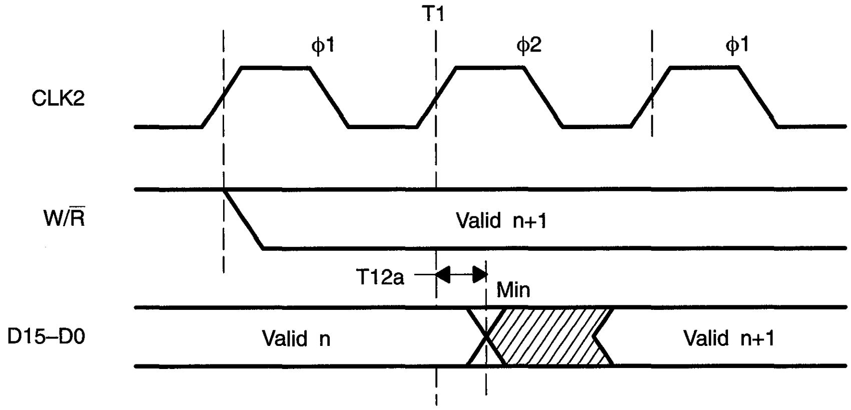
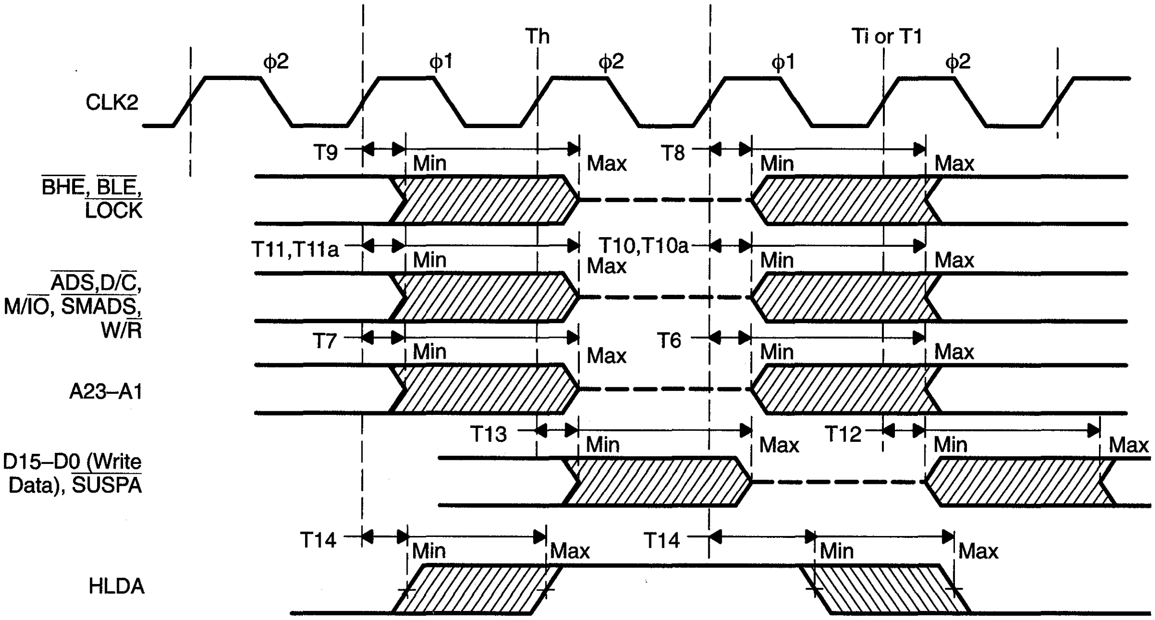


Figure 5–10. TI486SLC/E and TI486SLC/E-V Output Signal Float Delay and HLDA Valid Delay Timing



5.5.5 TI486DLC/E Switching Waveforms

Switching waveforms for the TI486DLC/E and TI486DLC/E-V are illustrated in Figure 5–11 through Figure 5–15.

Figure 5–11. TI486DLC/E and TI486DLC/E-V Input Signal Setup and Hold Timing

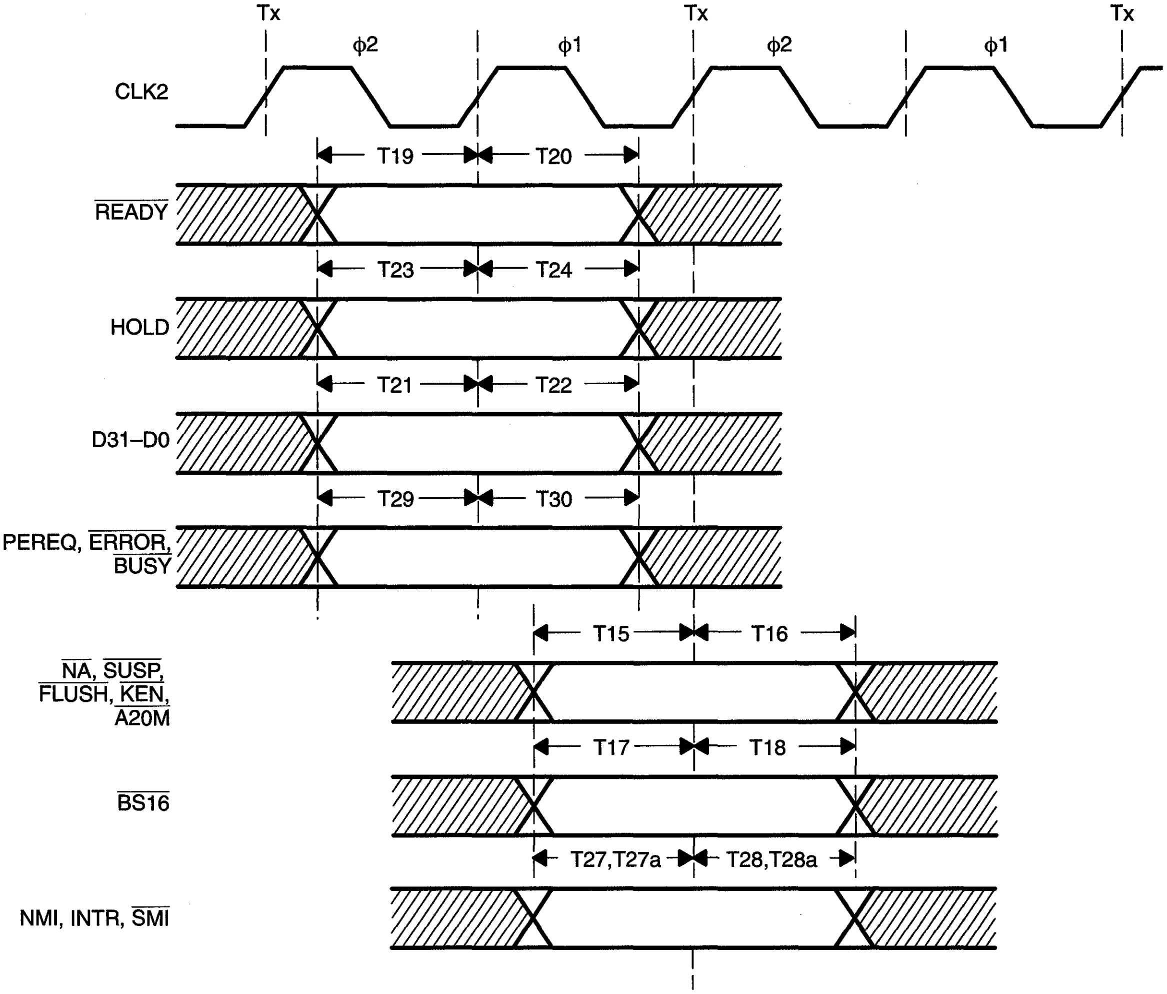


Figure 5–12. TI486DLC/E and TI486DLC/E-V Output Signal Valid Delay Timing

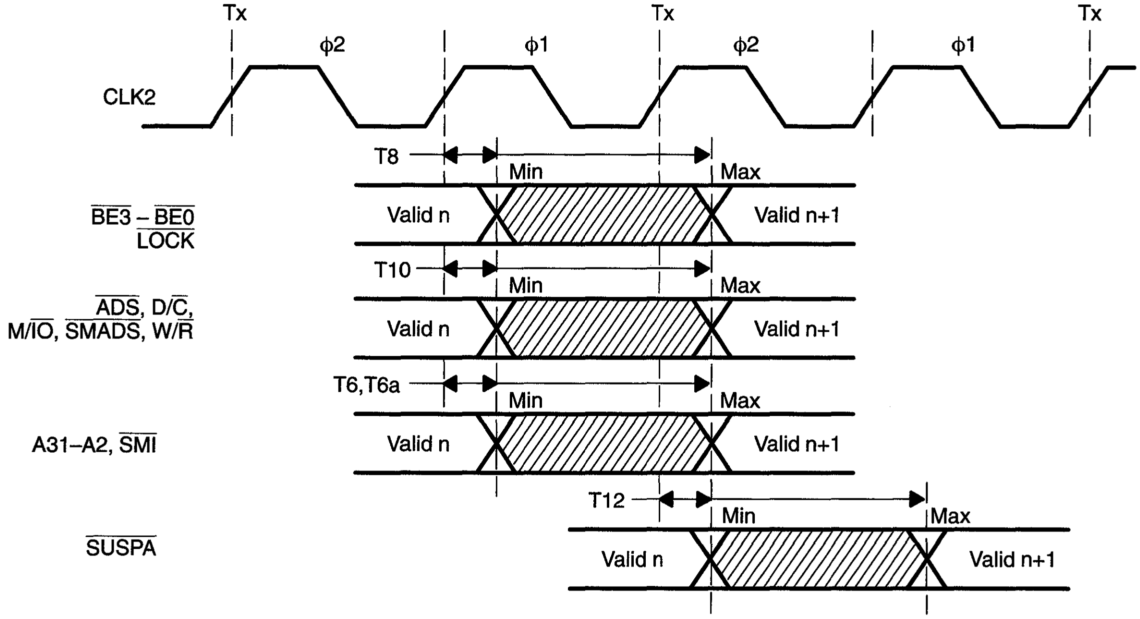


Figure 5–13. TI486DLC/E and TI486DLC/E-V Data Write Cycle Valid Delay Timing

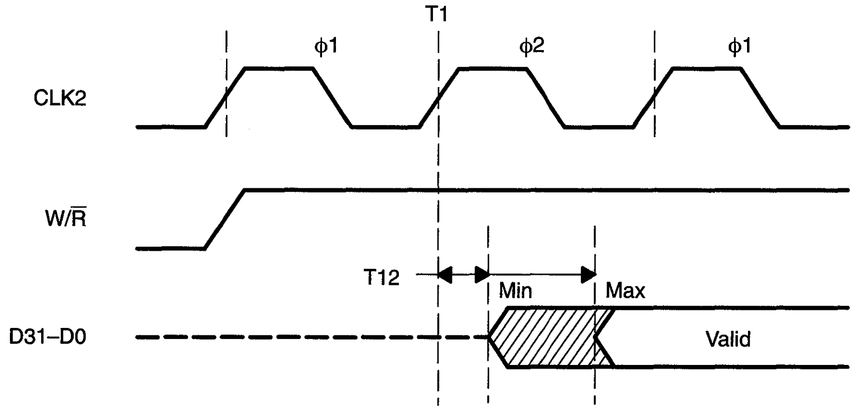


Figure 5–14. TI486DLC/E and TI486DLC/E-V Data Write Cycle Hold Timing

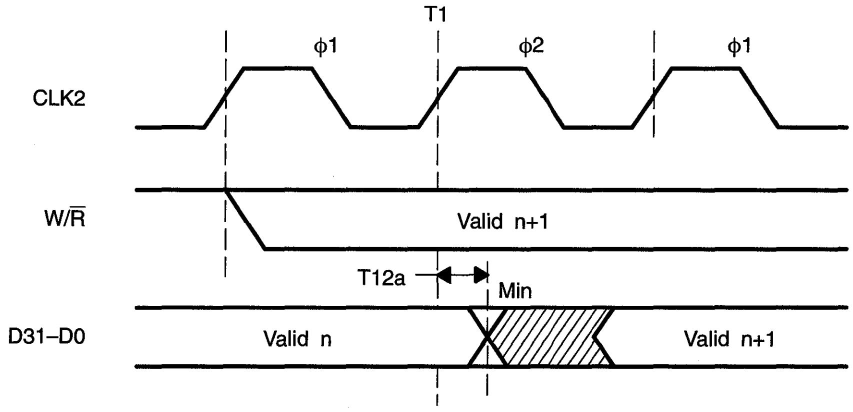
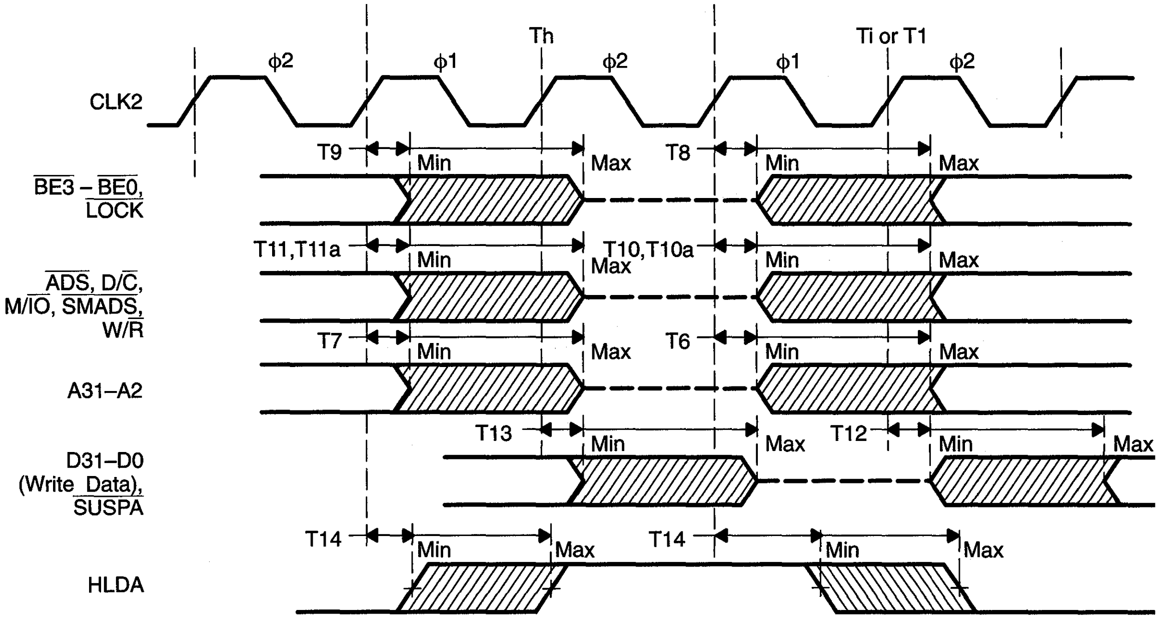


Figure 5–15. TI486DLC/E Output Signal Float Delay and HLDA Valid Delay Timing



Chapter 6

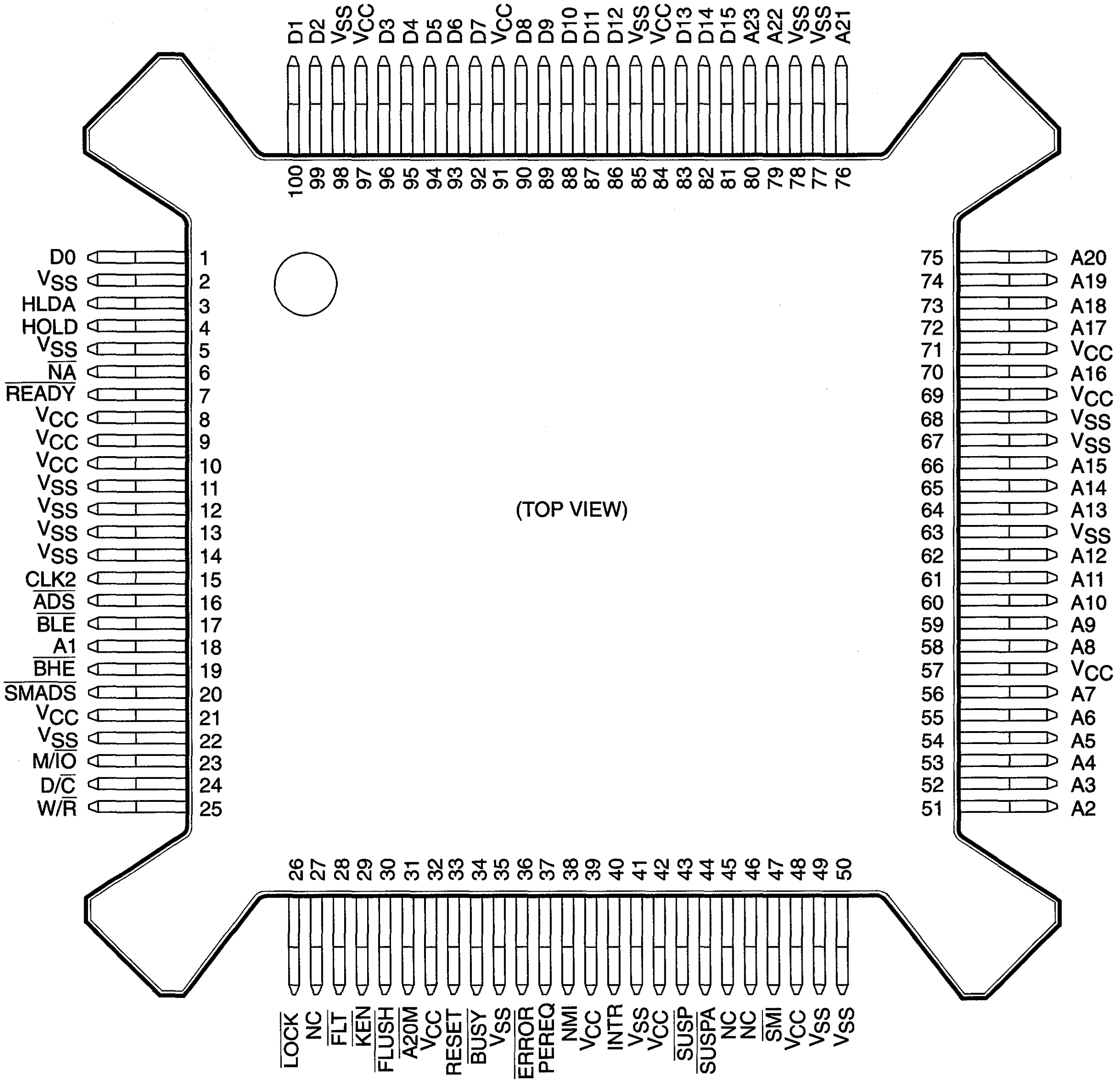
Mechanical Specifications

Topic **Page**

6.1 Pin Assignments

The pin assignments for the TI486SLC/E and TI486SLC/E-V are shown in Figure 6–1. The signal names are shown in Table 6–1 sorted by pin numbers and in Table 6–2 sorted by signal names.

Figure 6–1. TI486SLC/E and TI486SLC/E-V Pin Assignments



NC — No internal connection

Table 6–1. TI486SLC/E and TI486SLC/E-V Signal Names Sorted by Pin Number

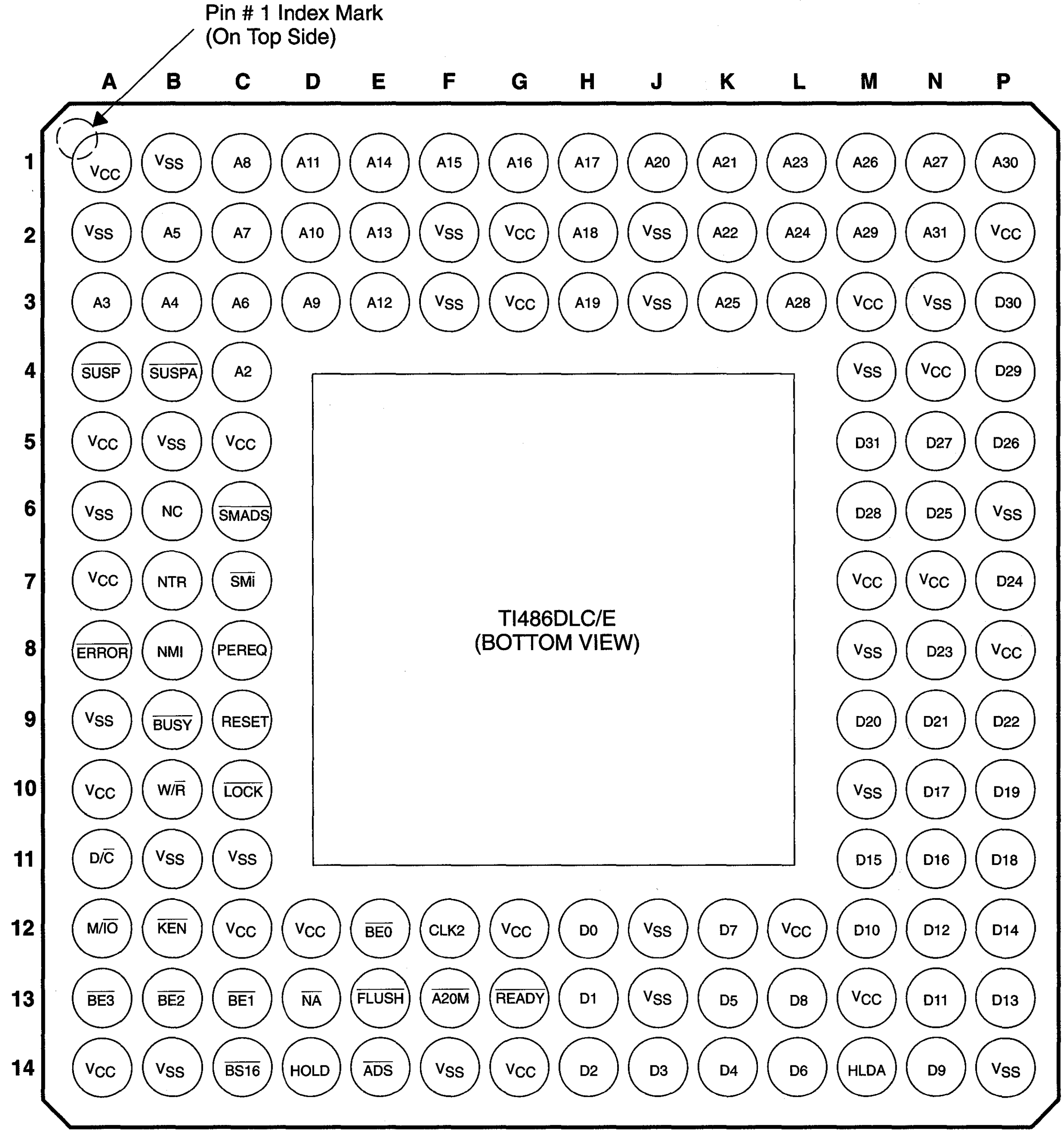
PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME
1	D0	21	V_{CC}	41	V_{SS}	61	A11	81	D15
2	V_{SS}	22	V_{SS}	42	V_{CC}	62	A12	82	D14
3	HLDA	23	M/IO	43	SUSP	63	V_{SS}	83	D13
4	HOLD	24	D/C	44	SUSPA	64	A13	84	V_{CC}
5	V_{SS}	25	W/R	45	NC	65	A14	85	V_{SS}
6	NA	26	LOCK	46	NC	66	A15	86	D12
7	READY	27	NC	47	SMI	67	V_{SS}	87	D11
8	V_{CC}	28	FLT	48	V_{CC}	68	V_{SS}	88	D10
9	V_{CC}	29	KEN	49	V_{SS}	69	V_{CC}	89	D9
10	V_{CC}	30	FLUSH	50	V_{SS}	70	A16	90	D8
11	V_{SS}	31	A20M	51	A2	71	V_{CC}	91	V_{CC}
12	V_{SS}	32	V_{CC}	52	A3	72	A17	92	D7
13	V_{SS}	33	RESET	53	A4	73	A18	93	D6
14	V_{SS}	34	BUSY	54	A5	74	A19	94	D5
15	CLK2	35	V_{SS}	55	A6	75	A20	95	D4
16	ADS	36	ERROR	56	A7	76	A21	96	D3
17	BLE	37	PEREQ	57	V_{CC}	77	V_{SS}	97	V_{CC}
18	A1	38	NMI	58	A8	78	V_{SS}	98	V_{SS}
19	BHE	39	V_{CC}	59	A9	79	A22	99	D2
20	SMADS	40	INTR	60	A10	80	A23	100	D1

Table 6–2. TI486SLC/E and TI486SLC/E-V Pin Numbers Sorted by Signal Name

SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.
A1	18	A21	76	D11	87	PEREQ	37	V_{CC}	97
A2	51	A22	79	D12	86	READY	7	V_{SS}	2
A3	52	A23	80	D13	83	RESET	33	V_{SS}	5
A4	53	ADS	16	D14	82	SMADS	20	V_{SS}	11
A5	54	A20M	31	D15	81	SMI	47	V_{SS}	12
A6	55	BHE	19	D/C	24	SUSP	43	V_{SS}	13
A7	56	BLE	17	ERROR	36	SUSPA	44	V_{SS}	14
A8	58	BUSY	34	FLT	28	V_{CC}	8	V_{SS}	22
A9	59	CLK2	15	FLUSH	30	V_{CC}	9	V_{SS}	35
A10	60	D0	1	HOLD	4	V_{CC}	10	V_{SS}	41
A11	61	D1	100	HLDA	3	V_{CC}	21	V_{SS}	49
A12	62	D2	99	INTR	40	V_{CC}	32	V_{SS}	50
A13	64	D3	96	KEN	29	V_{CC}	39	V_{SS}	63
A14	65	D4	95	LOCK	26	V_{CC}	42	V_{SS}	67
A15	66	D5	94	M/IO	23	V_{CC}	48	V_{SS}	68
A16	70	D6	93	NA	6	V_{CC}	57	V_{SS}	77
A17	72	D7	92	NC	27	V_{CC}	69	V_{SS}	78
A18	73	D8	90	NC	45	V_{CC}	71	V_{SS}	85
A19	74	D9	89	NC	46	V_{CC}	84	V_{SS}	98
A20	75	D10	88	NMI	38	V_{CC}	91	W/R	25

The pin assignments for the TI486DLC/E and TI486DLC/E-V are shown as viewed from the pin side in Figure 6–2 and as viewed from the top side (component side when mounted on a PC board) in Figure 6–3. The signal names are listed in Table 6–3 and Table 6–4, sorted by pin number and signal name respectively.

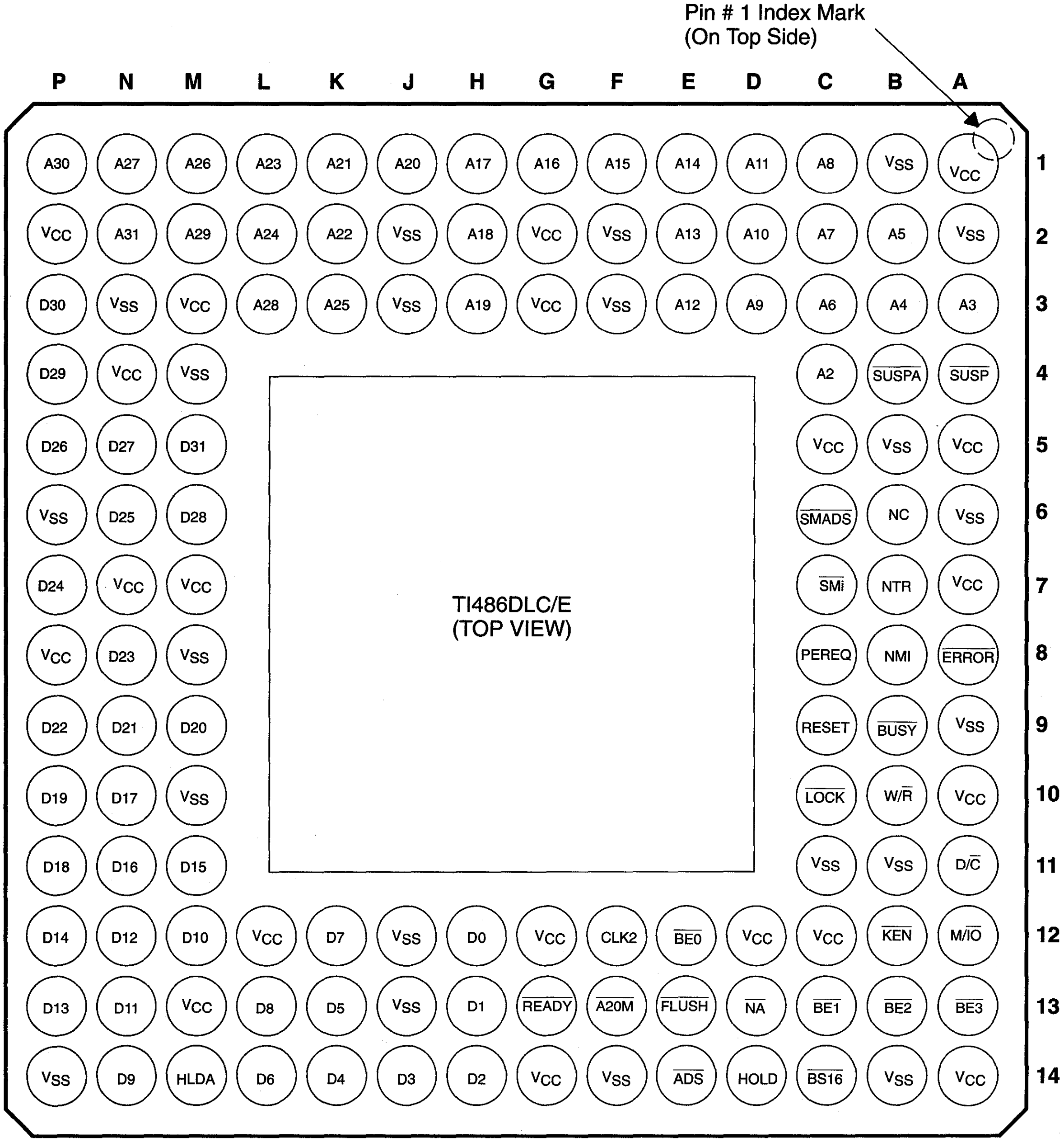
Figure 6–2. TI486DLC/E and TI486DLC/E-V Package Pins (Bottom View)



	A	B	C	D	E	F	G	H	J	K	L	M	N	P
1	V_{CC}	V_{SS}	A8	A11	A14	A15	A16	A17	A20	A21	A23	A26	A27	A30
2	V_{SS}	A5	A7	A10	A13	V_{SS}	V_{CC}	A18	V_{SS}	A22	A24	A29	A31	V_{CC}
3	A3	A4	A6	A9	A12	V_{SS}	V_{CC}	A19	V_{SS}	A25	A28	V_{CC}	V_{SS}	D30
4	$\overline{\text{SUSP}}$	$\overline{\text{SUSPA}}$	A2									V_{SS}	V_{CC}	D29
5	V_{CC}	V_{SS}	V_{CC}									D31	D27	D26
6	V_{SS}	NC	$\overline{\text{SMADS}}$									D28	D25	V_{SS}
7	V_{CC}	NTR	$\overline{\text{SMI}}$									V_{CC}	V_{CC}	D24
8	$\overline{\text{ERROR}}$	NMI	PEREQ									V_{SS}	D23	V_{CC}
9	V_{SS}	$\overline{\text{BUSY}}$	RESET									D20	D21	D22
10	V_{CC}	W/$\overline{\text{R}}$	$\overline{\text{LOCK}}$									V_{SS}	D17	D19
11	D/$\overline{\text{C}}$	V_{SS}	V_{SS}									D15	D16	D18
12	M/$\overline{\text{IO}}$	$\overline{\text{KEN}}$	V_{CC}	V_{CC}	$\overline{\text{BE0}}$	CLK2	V_{CC}	D0	V_{SS}	D7	V_{CC}	D10	D12	D14
13	$\overline{\text{BE3}}$	$\overline{\text{BE2}}$	$\overline{\text{BE1}}$	$\overline{\text{NA}}$	$\overline{\text{FLUSH}}$	$\overline{\text{A20M}}$	$\overline{\text{READY}}$	D1	V_{SS}	D5	D8	V_{CC}	D11	D13
14	V_{CC}	V_{SS}	$\overline{\text{BS16}}$	HOLD	$\overline{\text{ADS}}$	V_{SS}	V_{CC}	D2	D3	D4	D6	HLDA	D9	V_{SS}

NC — No internal connection

Figure 6–3. TI486DLC/E and TI486DLC/E-V Package Pins (Top View)



	P	N	M	L	K	J	H	G	F	E	D	C	B	A
1	A30	A27	A26	A23	A21	A20	A17	A16	A15	A14	A11	A8	V_{SS}	V_{CC}
2	V_{CC}	A31	A29	A24	A22	V_{SS}	A18	V_{CC}	V_{SS}	A13	A10	A7	A5	V_{SS}
3	D30	V_{SS}	V_{CC}	A28	A25	V_{SS}	A19	V_{CC}	V_{SS}	A12	A9	A6	A4	A3
4	D29	V_{CC}	V_{SS}									A2	$\overline{\text{SUSPA}}$	$\overline{\text{SUSP}}$
5	D26	D27	D31									V_{CC}	V_{SS}	V_{CC}
6	V_{SS}	D25	D28									$\overline{\text{SMADS}}$	NC	V_{SS}
7	D24	V_{CC}	V_{CC}									$\overline{\text{SMI}}$	NTR	V_{CC}
8	V_{CC}	D23	V_{SS}									PEREQ	NMI	$\overline{\text{ERROR}}$
9	D22	D21	D20									RESET	$\overline{\text{BUSY}}$	V_{SS}
10	D19	D17	V_{SS}									$\overline{\text{LOCK}}$	W/$\overline{\text{R}}$	V_{CC}
11	D18	D16	D15									V_{SS}	V_{SS}	D/$\overline{\text{C}}$
12	D14	D12	D10	V_{CC}	D7	V_{SS}	D0	V_{CC}	CLK2	$\overline{\text{BE0}}$	V_{CC}	V_{CC}	$\overline{\text{KEN}}$	M/$\overline{\text{IO}}$
13	D13	D11	V_{CC}	D8	D5	V_{SS}	D1	$\overline{\text{READY}}$	$\overline{\text{A20M}}$	$\overline{\text{FLUSH}}$	$\overline{\text{NA}}$	$\overline{\text{BE1}}$	$\overline{\text{BE2}}$	$\overline{\text{BE3}}$
14	V_{SS}	D9	HLDA	D6	D4	D3	D2	V_{CC}	V_{SS}	$\overline{\text{ADS}}$	HOLD	$\overline{\text{BS16}}$	V_{SS}	V_{CC}

NC — No internal connection

Table 6–3. TI486DLC/Eand TI486DLC/E-V Signal Names Sorted by Pin Number

PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME
A1	V_{CC}	B9	$\overline{\text{BUSY}}$	D3	A9	H1	A17	L13	D8	N7	V_{CC}
A2	V_{SS}	B10	W/$\overline{\text{R}}$	D12	V_{CC}	H2	A18	L14	D6	N8	D23
A3	A3	B11	V_{SS}	D13	$\overline{\text{NA}}$	H3	A19	M1	A26	N9	D21
A4	$\overline{\text{SUSP}}$	B12	$\overline{\text{KEN}}$	D14	HOLD	H12	D0	M2	A29	N10	D17
A5	V_{CC}	B13	$\overline{\text{BE2}}$	E1	A14	H13	D1	M3	V_{CC}	N11	D16
A6	V_{SS}	B14	V_{SS}	E2	A13	H14	D2	M4	V_{SS}	N12	D12
A7	V_{CC}	C1	A8	E3	A12	J1	A20	M5	D31	N13	D11
A8	$\overline{\text{ERROR}}$	C2	A7	E12	$\overline{\text{BEO}}$	J2	V_{SS}	M6	D28	N14	D9
A9	V_{SS}	C3	A6	E13	$\overline{\text{FLUSH}}$	J3	V_{SS}	M7	V_{CC}	P1	A30
A10	V_{CC}	C4	A2	E14	$\overline{\text{ADS}}$	J12	V_{SS}	M8	V_{SS}	P2	V_{CC}
A11	D/$\overline{\text{C}}$	C5	V_{CC}	F1	A15	J13	V_{SS}	M9	D20	P3	D30
A12	M/$\overline{\text{IO}}$	C6	$\overline{\text{SMADS}}$	F2	V_{SS}	J14	D3	M10	V_{SS}	P4	D29
A13	$\overline{\text{BE3}}$	C7	$\overline{\text{SMi}}$	F3	V_{SS}	K1	A21	M11	D15	P5	D26
A14	V_{CC}	C8	PEREQ	F12	CLK2	K2	A22	M12	D10	P6	V_{SS}
B1	V_{SS}	C9	RESET	F13	$\overline{\text{A20M}}$	K3	A25	M13	V_{CC}	P7	D24
B2	A5	C10	$\overline{\text{LOCK}}$	F14	V_{SS}	K12	D7	M14	HLDA	P8	V_{CC}
B3	A4	C11	V_{SS}	G1	A16	K13	D5	N1	A27	P9	D22
B4	$\overline{\text{SUSPA}}$	C12	V_{CC}	G2	V_{CC}	K14	D4	N2	A31	P10	D19
B5	V_{SS}	C13	$\overline{\text{BE1}}$	G3	V_{CC}	L1	A23	N3	V_{SS}	P11	D18
B6	NC	C14	$\overline{\text{BS16}}$	G12	V_{CC}	L2	A24	N4	V_{CC}	P12	D14
B7	INTR	D1	A11	G13	$\overline{\text{READY}}$	L3	A28	N5	D27	P13	D13
B8	NMI	D2	A10	G14	V_{CC}	L12	V_{CC}	N6	D25	P14	V_{SS}

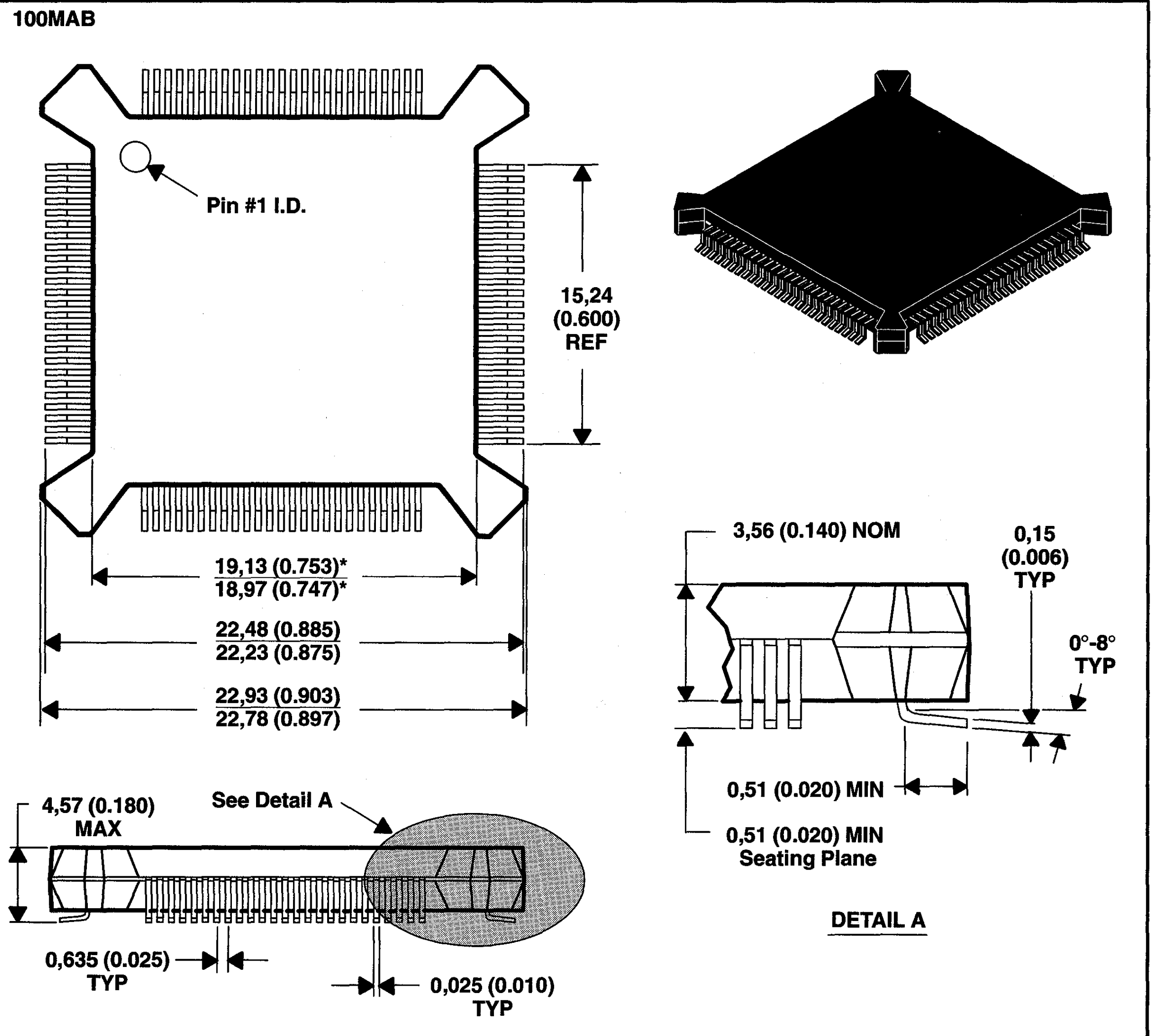
Table 6–4. TI486DLC/E and TI486DLC/E-V Pin Numbers Sorted by Signal Name

SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.	SIGNAL NAME	PIN NO.
A2	C4	A23	L1	D4	K14	D26	P5	$\overline{\text{SUSP}}$	A4	V_{SS}	A2
A3	A3	A24	L2	D5	K13	D27	N5	$\overline{\text{SUSPA}}$	B4	V_{SS}	A6
A4	B3	A25	K3	D6	L14	D28	M6	V_{CC}	A1	V_{SS}	A9
A5	B2	A26	M1	D7	K12	D29	P4	V_{CC}	A5	V_{SS}	B1
A6	C3	A27	N1	D8	L13	D30	P3	V_{CC}	A7	V_{SS}	B5
A7	C2	A28	L3	D9	N14	D31	M5	V_{CC}	A10	V_{SS}	B11
A8	C1	A29	M2	D10	M12	$\overline{\text{ERROR}}$	A8	V_{CC}	A14	V_{SS}	B14
A9	D3	A30	P1	D11	N13	$\overline{\text{FLUSH}}$	E13	V_{CC}	C5	V_{SS}	C11
A10	D2	A31	N2	D12	N12	HLDA	M14	V_{CC}	C12	V_{SS}	F2
A11	D1	$\overline{\text{ADS}}$	E14	D13	P13	HOLD	D14	V_{CC}	D12	V_{SS}	F3
A12	E3	$\overline{\text{BE0}}$	E12	D14	P12	INTR	B7	V_{CC}	G2	V_{SS}	F14
A13	E2	$\overline{\text{BE1}}$	C13	D15	M11	$\overline{\text{KEN}}$	B12	V_{CC}	G3	V_{SS}	J2
A14	E1	$\overline{\text{BE2}}$	B13	D16	N11	$\overline{\text{LOCK}}$	C10	V_{CC}	G12	V_{SS}	J3
A15	F1	$\overline{\text{BE3}}$	A13	D17	N10	M/$\overline{\text{IO}}$	A12	V_{CC}	G14	V_{SS}	J12
A16	G1	$\overline{\text{BS16}}$	C14	D18	P11	$\overline{\text{NA}}$	D13	V_{CC}	L12	V_{SS}	J13
A17	H1	$\overline{\text{BUSY}}$	B9	D19	P10	NC	B6	V_{CC}	M3	V_{SS}	M4
A18	H2	CLK2	F12	D20	M9	NMI	B8	V_{CC}	M7	V_{SS}	M8
A19	H3	D/$\overline{\text{C}}$	A11	D21	N9	PEREQ	C8	V_{CC}	M13	V_{SS}	M10
A20	J1	D0	H12	D22	P9	$\overline{\text{READY}}$	G13	V_{CC}	N4	V_{SS}	N3
$\overline{\text{A20M}}$	F13	D1	H13	D23	N8	RESET	C9	V_{CC}	N7	V_{SS}	P6
A21	K1	D2	H14	D24	P7	$\overline{\text{SMI}}$	C7	V_{CC}	P2	V_{SS}	P14
A22	K2	D3	J14	D25	N6	$\overline{\text{SMADS}}$	C6	V_{CC}	P8	W/$\overline{\text{R}}$	B10

6.2 Package Dimensions

The package dimensions for the TI486SLC/E and TI486SLC/E-V are shown in Figure 6–4 and the package dimensions for the TI486DLC/E and TI486DLC/E-V are shown in Figure 6–5.

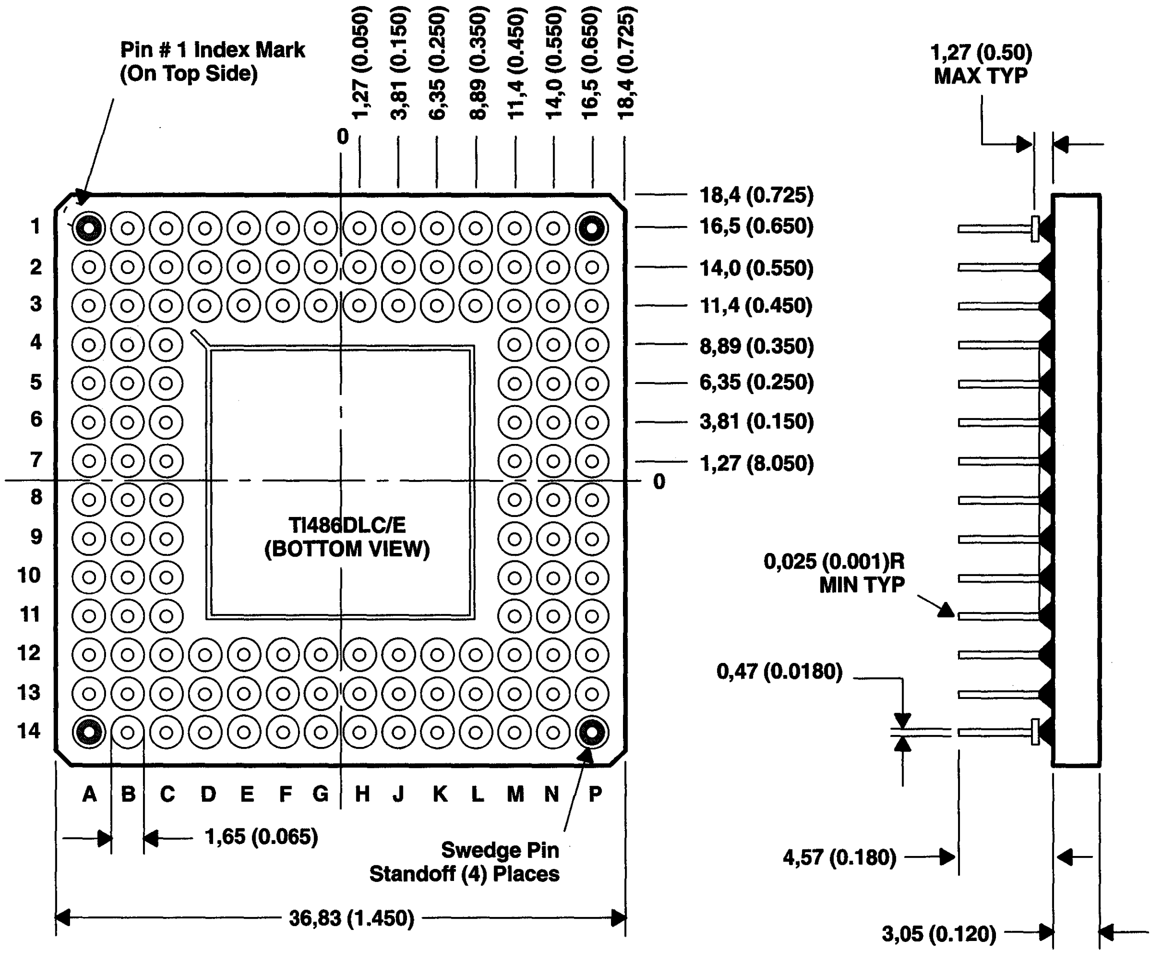
Figure 6–4. 100-Pin Plastic Bumpered QFP Package Dimensions (TI486SLC/E and TI486SLC/E-V)



* Note: For metal BQFP package only, this dimension is: 18,75 (0.738) / 18,59(0.732)

ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

Figure 6–5. 132-Pin PGA Package Dimensions (TI486DLC/E and TI486DLC/E-V)



ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

6.3 Thermal Characteristics

The TI486SLC/E is designed to operate when the case temperature is between 0°C and 100°C. The TI486SLC/E-V, TI486DLC/E and TI486DLC/E-V are designed to operate when the case temperature is between 0°C and 85°C. The case temperatures are measured on the top center of the package. The maximum die temperature (T_{jmax}) and the maximum ambient temperature (T_{amax}) can be calculated using the following equations.

$$T_{jmax} = T_c + (P_{max} \times \theta_{jc})$$
$$T_{amax} = T_j - (P_{max} \times \theta_{ja})$$

where:

T_{jmax} = Maximum average junction temperature (°C)
T_C = Case temperature at top center of package (°C)
P_{max} = Maximum device power dissipation (W)
θ_{jc} = Junction-to-case thermal resistance (°C/W)
T_{amax} = Maximum ambient temperature (°C)
T_j = Average junction temperature (°C)
θ_{ja} = Junction-to-ambient thermal resistance (°C/W)

Values for θ_{ja} and θ_{jc} are given in Table 6–5 for various airflows.

Table 6–5. Package Thermal Resistance and Airflow

AIRFLOW (FT/SEC)	THERMAL RESISTANCE (°C/W)			
	100-LEAD PLASTIC BQFP		132-PIN CERAMIC PGA PACKAGE	
	θ_{ja}	θ_{jc}	θ_{ja}	θ_{jc}
0	21	2	20	3
100	19	2	18	3
250	16	2	14	3
500	13	2	10	3

Chapter 7

Instruction Set

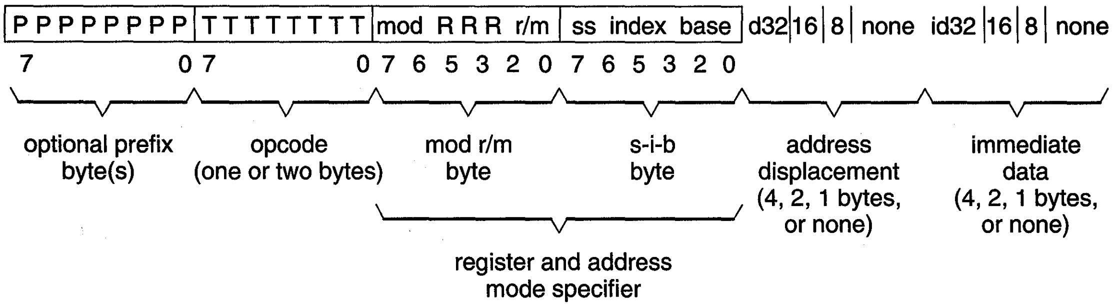
This section summarizes the TI486SLC/DLC instruction set and provides detailed information on the instruction encodings. All instructions are listed in the Instruction Set Summary Table (Table 7–17), which provides information on the instruction encoding, which flags are affected, and the instruction clock counts for each instruction. The clock count values are based on the assumptions described in subsection 7.4.1.

Topic **Page**

7.1 General Instruction Format

All of the TI486SLC/DLC machine instructions follow the general instruction format shown in Figure 7–1. 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 7–1. General Instruction Format


PPPPPPPP
TTTTTTTT
mod RRR r/m
ss index base
d32 16 8 none
id32 16 8 none
7 0 7 0 7 6 5 3 2 0 7 6 5 3 2 0
optional prefix byte(s)
opcode (one or two bytes)
mod r/m byte
s-i-b byte
address displacement (4, 2, 1 bytes, or none)
immediate data (4, 2, 1 bytes, or none)
register and address mode specifier

P – prefix bit
T – opcode bit
R – opcode bit or reg bit

7.2 Instruction Fields

The general instruction format shows the larger fields that make up an instruction. Certain instructions have smaller encoding fields that vary according to the class of operation. These fields define information such as the direction of the operation, the size of the displacements, register encoding and sign extension. All the fields are described in Table 7–1 and the subsequent paragraphs provide greater detail.

Table 7–1. Instruction Fields

FIELD NAME	DESCRIPTION	NUMBER OF BITS
Prefix	Specifies segment register override, address and operand size, repeat elements in string instruction, $\overline{\text{LOCK}}$ assertion.	8 per byte
Opcode	Identifies instruction operation.	1 or 2 bytes
w	Specifies if data is byte or full size (full size is either 16 or 32 bits).	1
d	Specifies direction of data operation.	1
s	Specifies if an immediate data field must be sign-extended.	1
reg	General register specifier.	3
mod r/m	Address mode specifier.	2 for mod; 3 for r/m
ss	Scale factor for scaled index address mode.	2
index	General register to be used as index register.	3
base	General register to be used as base register.	2
sreg2	Segment register for CS, SS, DS, and ES.	2
sreg3	Segment register for CS, SS, DS, ES, FS, and GS.	3
eee	Control, debug and test register specifier.	3
Address displacement	Address displacement operand.	1, 2 or 4 bytes
Immediate data	Immediate data operand.	1, 2 or 4 bytes

7.2.1 Prefixes

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

1) Segment override explicitly specifies which segment register an instruction will use.

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 addressing. 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 $\overline{\text{LOCK}}$ signal during execution of the instruction.

Table 7–2 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 7–2. 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 $\overline{\text{LOCK}}$ hardware signal.
REPNE	F2h	Repeat the following string instruction.
REP/REPE	F3h	Repeat the following string instruction.

7.2.2 Opcode Field

The opcode field is either one or two bytes in length and 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 0x80 names a group of operations that have an immediate operand, and a register or memory operand. The group opcodes use an opcode extension field of 3 bits in the following byte, called the MOD R/M byte, to resolve the operation type. Opcodes for the entire TI486SLC/DLC instruction set are listed in the Instruction Set Summary Table. The opcodes are given in hex values unless shown within brackets ([ ]). Values shown in brackets are binary values.

7.2.3 w Field

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

Table 7–3. w Field Encoding

w FIELD	OPERAND SIZE 16-BIT DATA OPERATIONS	OPERAND SIZE 32-BIT DATA OPERATIONS
0	8 bits	8 bits
1	16 bits	32 bits

7.2.4 d Field

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

Table 7–4. d Field Encoding

d FIELD	DIRECTION OF OPERATION	SOURCE OPERAND	DESIGNATION OPERAND
0	Register → Register/Memory	reg	mod r/m or mod ss-index-base
1	Register/Memory → Register	mod r/m or mod ss-index-base	reg

7.2.5 reg Field

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

Table 7–5. reg Field Encoding

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

7.2.6 mod and r/m Field

The mod and r/m sub-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 7–6 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 7–7.

Table 7–6. mod r/m Field Encoding

mod r/m	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	SSS:[BP+SI]	DS:[EDX]
00 011	SS:[BP+DI]	DS:[EBX]
00 100	DS:[SI]	s-i-b is present (see subsection 6.2.7)
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:[BXI+DI+d8]	DS:[EAX+d8]
01 010	SS:[BP+SI+d8]	DS:[EDX+d8]
01 011	SS:[BP+DI+d8]	DS:[EBX+d8]
01 100	DS:[SI+d8]	s-i-b is present (see subsection 6.2.7)
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	SS:[BP+SI+d16]	DS:[EDX+d32]
10 011	SS:[BP+DI+d16]	DS:[EBX+d32]
10 100	DS:[SI+d16]	s-i-b is present (see subsection 6.2.7)
10 101	DS:[DI+d16]	SS:[EBP+d32]
10 110	SS:[BP+d16]	DS:[ESI+d32]
10 111	DS:[BX+d16]	DS:[EDI+d32]
11 000–11 111	See Table 7–7	See Table 7–7

Table 7–7. 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

7.2.7 mod and base Fields

In Table 7–7, the note “s-i-b present” for certain entries forces the use of the mod and base field as listed in Table 7–8.

Table 7–8. mod base Field Encoding

mod r/m	32-BIT ADDRESS MODE WITH mod r/m BYTE AND NO s-i-b BYTE 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]

7.2.8 ss Field

The ss field (Table 7–9) 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 7–9. ss Field Encoding

ss FIELD	SCALE FACTOR
00	x1
01	x2
10	x4
11	x8

7.2.9 index Field

The index field (Table 7–10) specifies the index register used by the offset mechanism for offset address calculation. When no index register is used (index field=100), the ss value must be 00 or the effective address is undefined.

Table 7–10. index Field Encoding

index FIELD	INDEX REGISTER
000	EAX
001	ECX
010	EDX
011	EBX
100	none
101	EBP
110	ESI
111	EDI

7.2.10 sreg2 Field

The sreg2 field (Table 7–11) is a 2-bit field that allows one of the four 286-type segment registers to be specified.

Table 7–11. sreg2 Field encoding

sreg2 FIELD	SEGMENT REGISTER SELECTED
00	ES
01	CS
10	SS
11	DS

7.2.11 sreg3 Field

The sreg3 field (Table 7–12) is 3-bit field that is similar to the sreg2 field, but allows use of the FS and GS segment registers.

Table 7–12. sreg3 Field Encoding

sreg3 FIELD	SEGMENT REGISTER SELECTED
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	undefined
111	undefined

7.2.12 eee Field

The eee field is used to select the control, debug, and test registers as indicated in Table 7–13. The values shown are the only valid encodings for the eee bits.

Table 7–13. 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

7.3 Flags

The Instruction Set Summary Table lists nine flags that are affected by the execution of instructions. The conventions shown in Table 7–14 are used to identify the different flags. Table 7–15 lists the conventions used to indicate what action the instruction has on the particular flag.

Table 7–14. 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 7–15. Action of Instruction on Flag

INSTRUCTION TABLE SYMBOL	ACTION
m	Flag is modified by the instruction
u	Flag is not changed by the instruction
0	Flag is reset to “0”
1	Flag is set to “1”

7.4 Clock Count Summary

7.4.1 Assumptions

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

1) The instruction has been prefetched, decoded and is ready for execution.
2) Bus cycles do not require wait states.
3) There are no local bus HOLD requests delaying processor access to the bus.
4) No exceptions are detected during instruction execution.
5) 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 1 clock to the clock count shown.
6) All clock counts assume aligned 16-bit memory/IO operands for cache miss counts.
7) If instructions access a misaligned 16-bit operand or a 32-bit operand on even addresses, add 2 clocks for read or write, and add 4 clock counts for read and write.
8) If instructions access a 32-bit operand on odd addresses, add 4 clocks for read or write, and add 8 clocks for read and write.

7.4.2 Abbreviations

The clock counts listed in the 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 7–16.

Table 7–16. Clock Count Abbreviations

CLOCK COUNT SYMBOL	EXPLANATION
/	Register operand/memory operand
n	Number of times operation is repeated
L	Level of the stack frame
|	Condition jump taken | conditional jump not taken
\	CPL ≤ IOPL \ CPL > IOPL

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
AAA *ASCII Adjust AL after Add*	37	u	u	u	u	u	u	m	u	m	4		4			
AAD *ASCII Adjust AX before Divide*	D5 0A	u	u	u	u	m	m	u	m	u	4		4			
AAM *ASCII Adjust AX after Multiply*	D4 0A	u	u	u	u	m	m	u	m	u	16		16			
AAS *ASCII Adjust AL after Subtract*	3F	u	u	u	u	u	u	m	u	m	4		4			
ADC *Add with Carry*		m	u	u	u	m	m	m	m	m					1	2
Register to Register	1 [00dw] [11 reg r/m]										1		1			
Register to Memory	1 [000w] [mod reg r/m]										3	5	3	5		
Memory to Register	1 [001w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 010 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator	1 [010w]†										1		1			
ADD *Integer Add*		m	u	u	u	m	m	m	m	m					1	2
Register to Register	0 [00dw] [11 reg r/m]										1		1			
Register to Memory	0 [000w] [mod reg r/m]										3	5	3	5		
Memory to Register	0 [001w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 000 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator	0 [010w]†										1		1			
AND *Boolean AND*		0	u	u	u	m	m	u	m	0					1	2
Register to Register	2 [00dw] [11 reg r/m]										1		1			
Register to Memory	2 [000w] [mod reg r/m]										3	5	3	5		
Memory to Register	2 [001w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 100 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator	2 [010w]†										1		1			
ARPL *Adjust Requested Privilege Level*		u	u	u	u	u	m	u	u	u					3	2
From Register/Memory	63 [mod reg r/m]												6/10	10		
BOUND *Check Array Boundaries*	62 [mod reg r/m]	u	u	u	u	u	u	u	u	u					1,4	2,5,6,7,8
If Out of range (Int 5)											11+int		11+int			
If In Range											11		11			
BSF *Scan Bit Forward*		u	u	u	u	u	m	u	u	u					1	2
Register/Memory, Register	0F BC[mod reg r/m]										5/7+n	9+n	5/7+n	9+n		

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
BSR *Scan Bit Reverse*		u	u	u	u	u	m	u	u	u					1	2
Register/Memory, Register	0F BC[mod reg r/m]										5/7+n	9+n	5/7+n	9+n		
BSWAP *Byte Swap*	0F C[1 reg]	u	u	u	u	u	u	u	u	u	4		4			
BT *Test Bit*		u	u	u	u	u	u	u	u	m					1	2
Register/Memory, Immediate	0F BA[mod 100 r/m]†										3/4	5	3/4	5		
Register/Memory, Register	0F A3[mod reg r/m]										3/6	7	3/6	7		
BTC *Test Bit and Complement*		u	u	u	u	u	u	u	u	m					1	2
Register/Memory, Immediate	0F BA[mod 111 r/m]†										4/5	6	4/5	6		
Register/Memory, Register	0F BB[mod reg r/m]										5/8	9	5/8	9		
BTR *Test Bit and Reset*		u	u	u	u	u	u	u	u	m					1	2
Register/Memory, Immediate	0F BA[mod 110 r/m]†										4/5	6	4/5	6		
Register/Memory, Register	0F B3[mod reg r/m]										5/8	9	5/8	9		
BTS *Test Bit and Set*		u	u	u	u	u	u	u	u	m					1	2
Register/Memory	0F BA[mod 101 r/m]										3/5	6	3/5	6		
Register (short form)	0F AB[mod reg r/m]										4/7	8	4/7	8		

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:
1) 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) $\overline{\text{LOCK}}$ is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert $\overline{\text{LOCK}}$ to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
CALL *Subroutine Call*		u	u	u	u	u	u	u	u	u					1	2,6,7,8
Direct within Segment	E8¶										7		7			
Register/Memory Indirect within Segment	FF [mod 010 r/m]										8/9	10	8/9	10		
Direct Intersegment	9A [unsigned full offset,										12		30			
Call Gate to Same Privilege	selector]												41	49		
Call Gate to Different Privilege No P													83	97		
Call Gate to Different Privilege Ps													81+4x	95+4x		
16-Bit Task to 16-bit TSS													262	263		
16-Bit Task to 32-bit TSS													293	317		
16-Bit Task to V86 Task													179	206		
32-Bit Task to 16-bit TSS													238	258		
32-Bit Task to 32-bit TSS													296	340		
32-Bit Task to V86 Task													182	229		
Indirect Intersegment	FF [mod 011 r/m]										14	17	14	34		
Call Gate to Same Privilege													43	51		
Call Gate to Different Privilege No P													85	99		
Call Gate to Different Privilege Ps													86+4x	100+4x		
16-Bit Task to 16-bit TSS													267	268		
16-Bit Task to 32-bit TSS													298	322		
16-Bit Task to V86 Task													181	211		
32-Bit Task to 16-bit TSS													243	263		
32-Bit Task to 32-bit TSS													301	345		
32-Bit Task to V86 Task													184	230		
P = Parameters																
CBW *Convert Byte to Word*	98	u	u	u	u	u	u	u	u	u	3		3			
CDQ *Convert Doubleword to Quadword*	99	u	u	u	u	u	u	u	u	u	1		1			
CLC *Clear Carry Flag*	F8	u	u	u	u	u	u	u	u	0	1		1			
CLD *Clear Direction Flag*	FC	u	0	u	u	u	u	u	u	u	1		1			
CLI *Clear Interrupt Flag*	FA	u	u	0	u	u	u	u	u	u	7		7			9
CLTS *Clear Task Switched Flag*	0F 06	u	u	u	u	u	u	u	u	u	5		5		10	11
CMC *Complement the Carry Flag*	F5	u	u	u	u	u	u	u	u	m	1		1			
CMP *Compare Integers*		m	u	u	u	m	m	m	m	m					1	2
Register to Register	3 [10dw] [11 reg r/m]										1		1			
Register to Memory	3 [101w] [mod reg r/m]										3	5	3	5		
Memory to Register	3 [100w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 111 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator	3 [110w]†										1		1			

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
CMPS *Compare String*	A [011w]	m	u	u	u	m	m	m	m	m	7	8	7	8	1	2
CMPXCHG *Compare and Exchange* Register1, Register2 Memory, Register	 0F B[000w] [11 reg2 reg1] 0F B[000w] [mod reg r/m]	m	u	u	u	m	m	m	m	m	 5 7	 8	 5 7	 8		
CWD *Convert Word to Doubleword*	99	u	u	u	u	u	u	u	u	u	1		1			
CWDE *Convert Word to Doubleword Extended*	98	u	u	u	u	u	u	u	u	u	3		3			
DAA *Decimal Adjust AL after Add*	27	u	u	u	u	m	m	m	m	m	4		4			
DAS *Decimal Adjust AL after Subtract*	2F	u	u	u	u	m	m	m	m	m	4		4			
DEC *Decrement by 1* Register/Memory Register (short form)	 F [111w] [mod 001 r/m] 4 [1 reg]	m	u	u	u	m	m	m	m	u	 1/3 1	 5	 1/3 1	 5	1	2
DIV *Unsigned Divide* Accumulator by Register/Memory Divisor: Byte Word Doubleword	F [011w] [mod 110 r/m]	u	u	u	u	u	u	u	u	u	 14/15 22/23 38/39	 17 24 40	 14/15 22/23 38/39	 17 24 40	1,4	2,4
ENTER *Enter New Stack Frame* Level = 0 Level = 1 Level (L) > 1	C8 [8-bit level]§	u	u	u	u	u	u	u	u	u	 7 10 6+4*L	 10 6+4*L	 7 10 6+4*L	 10 6+4*L	1	2
HLT Halt	F4	u	u	u	u	u	u	u	u	u	3		3			11

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:

1) 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) LOCK is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.
9) An exception 13 fault occurs if CPL is greater than IOPL.
10) This instruction may be executed in Real Mode. in Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.
11) An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
IDIV *Integer (Signed) Divide*		u	u	u	u	u	u	u	u	u					1,4	2,4
Accumulator by Register/Memory	F [011w] [mod 111 r/m]															
Divisor: Byte											19/20	22	19/20	22		
Word											27/28	29	27/28	29		
Doubleword											43/44	47	43/44	47		
IMUL *Integer (Signed) Multiply*		m	u	u	u	u	u	u	u	m					1	2
Accumulator by Register/Memory	F [011w] [mod 101 r/m]															
Multiplier: Byte											3/5	7	3/5	7		
Word											3/5	7	3/5	7		
Doubleword											7/9	13	7/9	13		
Register with Register/Memory	0F AF[mod reg r/m]															
Multiplier: Byte											3/5	7	3/5	7		
Word											3/5	7	3/5	7		
Doubleword											7/9	13	7/9	13		
Register/Memory with Immediate to Register2	6 [10s1] [mod reg r/m]†															
Multiplier: Byte											3/5	7	3/5	7		
Word											3/5	7	3/5	7		
Doubleword											7/9	13	7/9	13		
IN *Input from I/O Port*		u	u	u	u	u	u	u	u	u						9
Fixed Port	E [010w] [port number]										16	16	6/19	6/20		
Variable Port	E [110w]										16	16	6/19	6/20		
INC *Increment by 1*		m	u	u	u	m	m	m	m	u					1	2
Register/Memory	F [111w] [mod 000 r/m]										1/3	5	1/3	5		
Register (short from)	4 [0 reg]										1		1			
INS *Input String from I/O Port*	6 [110w]	u	u	u	u	u	u	u	u	u	20	20	6/19	6/20	1	2,9
INT *Software Interrupt*		u	m	0	u	u	u	u	u	u					1,4	5,6,7,8
INT i	CD [i]										14	16				
Protected Mode:																
Interrupt or Trap to Same Privilege													57	58		
Interrupt or Trap to Different Privilege													91	92		
16-Bit Task to 16-bit TSS by Task Gate													265	266		
16-Bit Task to 32-bit TSS by Task Gate													296	320		
16-Bit Task to V86 Task by Task Gate													177	205		
32-Bit Task to 16-bit TSS by Task Gate													241	261		
32-Bit Task to 32-bit TSS by Task Gate													299	343		
32-Bit Task to V86 Task by Task Gate													180	232		
V86 to 16-bit TSS by Task Gate													241	261		
V86 to 32-bit TSS by Task Gate													299	343		
V86 to Privilege 0 by Trap Gate/Int Gate													106	114		
Continued on next page . . .																

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
INT *Software Interrupt* **(Continued)**		u	m	0	u	u	u	u	u	u					1,4	5,6,7,8
INT3	CC										14	16				
Protected Mode:																
Interrupt or Trap to Same Privilege													57	58		
Interrupt or Trap to Different Privilege													91	92		
16-Bit Task to 16-bit TSS by Task Gate													265	266		
16-Bit Task to 32-bit TSS by Task Gate													296	320		
16-Bit Task to V86 by Task Gate													177	205		
32-Bit Task to 16-bit TSS by Task Gate													241	261		
32-Bit Task to 32-bit TSS by Task Gate													299	343		
32-Bit Task to V86 by Task Gate													180	232		
V86 to 16-bit TSS by Task Gate													241	261		
V86 to 32-bit TSS by Task Gate													299	343		
V86 to Privilege 0 by Trap Gate/Int Gate													106	114		
INTO	CE	u	u	m	0	u	u	u	u	u						
If 0F == 0											1	1	1	1		
If 0F == 1 (INT4)											15	17				
Protected Mode:																
Interrupt or Trap to Same Privilege													57	58		
Interrupt or Trap to Different Privilege													91	92		
16-Bit Task to 16-bit TSS by Task Gate													265	266		
16-Bit Task to 32-bit TSS by Task Gate													296	320		
16-Bit Task to V86 by Task Gate													177	205		
32-Bit Task to 16-bit TSS by Task Gate													241	261		
32-Bit Task to 32-bit TSS by Task Gate													299	343		
32-Bit Task to V86 by Task Gate													180	232		
V86 to 16-bit TSS by Task Gate													241	261		
V86 to 32-bit TSS by Task Gate													299	343		
V86 to Privilege 0 by Trap Gate/Int Gate													106	114		

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:

1) Exception 13 fault (general protection) will occur in Read 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) $\overline{\text{LOCK}}$ is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert $\overline{\text{LOCK}}$ to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.
9) An exception 13 fault occurs if CPL is greater than IOPL.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
INVD *Invalidate Cache*	0F 08	u	u	u	u	u	u	u	u	u	4		4			
INVLPG *Invalidate TLB Entry*	0F 01[mod 111 r/m]	u	u	u	u	u	u	u	u	u	4		4			
IRET *Interrupt Return*	CF	m	m	m	m	m	m	m	m	m						2,5,6,7,8
Read Mode											14	14				
Protected Mode																
Within Task to Same Privilege													35	37		
Within Task to Different Privilege													74	78		
16-Bit Task to 16-bit TSS													259	260		
16-Bit Task to 32-bit TSS													290	314		
16-Bit Task to V86 Task													173	203		
32-Bit Task to 16-bit TSS													235	255		
32-Bit Task to 32-bit TSS													295	339		
32-Bit Task to V86 Task													176	226		
JB/JNAE/JC *Jump on Below/Not Above or Equal/Carry*		u	u	u	u	u	u	u	u	u						8
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*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	76‡										4|1		6|1			
Full displacement	0F 86¶										4|1		6|1			
JCXZ *Jump on CX Zero*	E3‡	u	u	u	u	u	u	u	u	u	7|3		7|3			8
JE/JZ *Jump on Equal/Zero*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	74‡										4|1		6|1			
Full displacement	0F 84¶										4|1		6|1			
JECXZ *Jump on ECX Zero*	E3‡	u	u	u	u	u	u	u	u	u	7|3		7|3			8
JL/JNGE *Jump on Less/Not Greater or Equal*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	7C‡										4|1		6|1			
Full displacement	0F 8C¶										4|1		6|1			

Table 7–17. *Instructions, Opcodes, Flags, and Clock Summary (Continued)*

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
JLE/JNG *Jump on Less or Equal/Not Greater*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	7E‡										4|1		6|1			
Full displacement	0F 8E¶										4|1		6|1			
JMP *Unconditional Jump*		u	u	u	u	u	u	u	u	u					1	2,6,7,8
Short	EB‡										4		6			
Direct within Segment	E9¶										4		6			
Register/Memory Indirect within Segment	FF [mod 100 r/m]										6/8	10	6/8	10		
Direct Intersegment	EA [full offset, selector]										9		26			
Call Gate Same Privilege Level													45	45		
16-Bit Task to 16-bit TSS													265	266		
16-Bit Task to 32-bit TSS													296	320		
16-Bit Task to V86 Task													182	209		
32-Bit Task to 16-bit TSS													241	261		
32-Bit Task to 32-bit TSS													299	343		
32-Bit Task to V86 Task													185	232		
Indirect Intersegment	FF [mod 101 r/m]										11	14	30	30		
Call Gate Same Privilege Level													47	47		
16-Bit Task to 16-bit TSS													270	271		
16-Bit Task to 32-bit TSS													301	325		
16-Bit Task to V86 Task													184	214		
32-Bit Task to 16-bit TSS													246	268		
32-Bit Task to 32-bit TSS													304	348		
32-Bit Task to V86 Task													187	237		
JNB/JAE/JNC *Jump on Not Below/ Above or Equal/Not Carry*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	73‡										4|1		6|1			
Full displacement	0F 83¶										4|1		6|1			

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:
1) Exception 13 fault (general protection) will occur in Read 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) $\overline{\text{LOCK}}$ is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert $\overline{\text{LOCK}}$ to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
JNBE/JA *Jump on Not Below or Equal/Above*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	77‡										4|1		6|1			
Full displacement	0F 87¶										4|1		6|1			
JNE/JNZ *Jump on Not Equal/Not Zero*		u	u	u	u	u	u	u	u	u						8
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*		u	u	u	u	u	u	u	u	u						8
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*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	7F‡										4|1		6|1			
Full displacement	0F 8F¶										4|1		6|1			
JNO *Jump on Not Overflow*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	71‡										4|1		6|1			
Full displacement	0F 81¶										4|1		6|1			
JNP/JPO *Jump on Not Parity/Parity Odd*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	7B‡										4|1		6|1			
Full displacement	0F 8B¶										4|1		6|1			
JNS *Jump on Not Sign*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	79‡										4|1		6|1			
Full displacement	0F 89¶										4|1		6|1			
JO *Jump on Overflow*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	70‡										4|1		6|1			
Full displacement	0F 80¶										4|1		6|1			
JP/JPE *Jump on Parity/Parity Even*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	7A‡										4|1		6|1			
Full displacement	0F 8A¶										4|1		6|1			
JS *Jump on Sign*		u	u	u	u	u	u	u	u	u						8
8-Bit displacement	78‡										4|1		6|1			
Full displacement	0F 88¶										4|1		6|1			

Table 7–17. *Instructions, Opcodes, Flags, and Clock Summary (Continued)*

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
LAHF *Load AH with Flags*	9F	u	u	u	u	u	u	u	u	u	2		2			
LAR *Load Access Rights* From Register/Memory	0F 02[mod reg r/m]	u	u	u	u	u	m	u	u	u			11/12	14	3	2,5,6,12
LDS *Load Pointer to DS*	C5 [mod reg r/m]	u	u	u	u	u	u	u	u	u	6	7	23	24	1	2,6,13
LEA *Load Effective Address* No Index Register With Index Register	8D [mod reg r/m]	u	u	u	u	u	u	u	u	u	2 3		2 3			
LEAVE *Leave Current Stack Frame*	C9	u	u	u	u	u	u	u	u	u	3	4	3	4	1	2
LES *Load Pointer to ES*	C4 [mod reg r/m]	u	u	u	u	u	u	u	u	u	6	7	23	24	1	2,6,13
LFS *Load Pointer to FS*	0F B4[mod reg r/m]	u	u	u	u	u	u	u	u	u	6	7	23	24	1	2,6,13
LGDT *Load GDT Register*	0F 01[mod 010 r/m]	u	u	u	u	u	u	u	u	u	9	9	9	9	1,10	2,11
LGS *Load Pointer to GS*	0F B5[mod reg r/m]	u	u	u	u	u	u	u	u	u	6	7	23	24	1	2,6,13
LIDT *Load IDT Register*	0F 01[mod 011 r/m]	u	u	u	u	u	u	u	u	u	9	9	9	9	1,10	2,11
LLDT *Load LDT Register* From Register/Memory	0F 00[mod 010 r/m]	u	u	u	u	u	u	u	u	u			16/17	18	3	2,5,6,11
LMSW *Load Machine Status Word* From Register/Memory	0F 01[mod 110 r/m]	u	u	u	u	u	u	u	u	u	5	8	5	8	1,10	2,11
LODS *Load String*	A [110w]	u	u	u	u	u	u	u	u	u	4	4	4	4	1	2
LOOP *Offset Loop/No Loop*	E2‡	u	u	u	u	u	u	u	u	u	7|4		9|3			8

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:
1) Exception 13 fault (general protection) will occur in Read 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) LOCK is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.
9) An exception 13 fault occurs if CPL is greater than IOPL.
10) This instruction may be executed in Real Mode. in Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.
11) An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).
12) Any violation of privilege rules as apply to the selector operand does not cause a Protection exception, rather, the zero flag is cleared.
13) For segment load operations, the CPL, RPL, and DPL must agree witht he privolege rules to avoid an exception 13 fault. The segment's descriptor must indicate "present" or exception 11 (DS, DS, ES, FS, GS not present). If the SS register is loaded and a stack segment not present is detected, and exception 12 occurs.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
LOOPNZ/LOOPNE *Offset*	E0‡	u	u	u	u	u	u	u	u	u	7|4		9|3			8
LOOPZ/LOOPE *Offset*	E1‡	u	u	u	u	u	u	u	u	u	7|4		9|3			8
LSL *Load Segment Limit*		u	u	u	u	u	m	u	u	u					3	2,5,6,12
From Register/Memory	0F 03[mod reg r/m]												22/23	25		
LSS *Load Pointer to SS*	0F B2[mod reg r/m]	u	u	u	u	u	u	u	u	u	6	7	23	24	3	2,6,13
LTR *Load Task Register*	0F 00[mod reg r/m]	u	u	u	u	u	u	u	u	u					3	2,5,6,11
From Register/Memory													16/17	18		
MOV *Move Data*		u	u	u	u	u	u	u	u	u					1	2,6,13
Register to Register/Memory	8 [110w] [mod reg r/m]										1/2	2	1/2	2		
Register/Memory to Register	8 [101w] [mod reg r/m]										1/2	4	1/2	4		
Immediate to Register/Memory	C [011w] [mod 000 r/m]†										1/2	2	1/2	2		
Immediate to Register (short form)	B [w reg]†										1		1			
Memory to Accumulator (short form)	A [000w]¶										2	4	2	4		
Accumulator to Memory (short form)	A [001w]¶										1/2	2	1/2	2		
Register/Memory to Segment Register	8E [mod sreg3 r/m]										2/3	5	15/16	18		
Segment Register to Register/Memory	8C [mod reg r/m]										1/2	2	1/2	2		
MOV *Move to/from Control/Debug/Test Registers*		u	u	u	u	u	u	u	u	u						11
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			
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]	u	u	u	u	u	u	u	u	u	5	5	5	5	1	2
MOVSX *Move with Sign Extension*		u	u	u	u	u	u	u	u	u					1	2
Register from Register/Memory	0F B[111w] [mod reg r/m]										1/3	5	1/3	5		
MOVZX *Move with Zero Extension*		u	u	u	u	u	u	u	u	u					1	2
Register from Register/Memory	0F B[011w] [mod reg r/m]										2/3	5	2/3	5		

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
MUL *Unsigned Multiply*	F [011w] [mod 100 r/m]	m	u	u	u	u	u	u	u	m					1	2
Accumulator with Register/Memory																
Multiplier: Byte											3/5	7	3/5	7		
Word											3/5	7	3/5	7		
Doubleword											7/9	13	7/9	13		
NEG *Negate Integer*	F [011w] [mod 011 r/m]	m	u	u	u	m	m	m	m	m	1/3	5	1/3	5	1	2
NOP *No Operation*	90	u	u	u	u	u	u	u	u	u	3		3			
NOT *Boolean Complement*	F [011w] [mod 010 r/m]	u	u	u	u	u	u	u	u	u	1/3	5	1/3	5	1	2
OR *Boolean OR*		0	u	u	u	m	m	m	m	0					1	2
Register to Register	0 [10dw] [11 reg r/m]										1		1			
Register to Memory	0 [100w] [mod reg r/m]										3	5	3	5		
Memory to Register	0 [101w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [000w] [mod 001 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator	0 [110w]†										1		1			
OUT *Output to Port*		u	u	u	u	u	u	u	u	u						9
Fixed Port	E [011w] [port number]										18	18	4\17	4\18		
Variable Port	E [111w]										18	18	4\17	4\18		
OUTS *Output String*	6 [111w]	u	u	u	u	u	u	u	u	u	20	20	6\19	6\19	1	2,9

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:
1) Exception 13 fault (general protection) will occur in Read 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) LOCK is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.
9) An exception 13 fault occurs if CPL is greater than IOPL.
10)This instruction may be executed in Real Mode. in Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.
11)An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).
12)Any violation of privilege rules as apply to the selector operand does not cause a Protection exception, rather, the zero flag is cleared.
13)For segment load operations, the CPL, RPL, and DPL must agree witht he privolege rules to avoid an exception 13 fault. The segment's descriptor must indicate "present" or exception 11 (DS, DS, ES, FS, GS not present). If the SS register is loaded and a stack segment not present is detected, and exception 12 occurs.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
POP *Pop Value off Stack*		u	u	u	u	u	u	u	u	u					1	2,6,13
Register/Memory	8F [mod 000 r/m]										3/5	4/5	3/5	4/5		
Register (short form)	5 [1 reg]										3	4	3	4		
Segment Register (ES, CS, SS, DS)	[000 sreg2 110]										4	5	18	19		
Segment Register (ES, CS, SS, DS, FS, GS)	0F [10 sreg3 001]										4	5	18	19		
POPA *Pop All General Registers*	61	u	u	u	u	u	u	u	u	u	18	18	18	18	1	2
POPF *Pop Stack into FLAGS*	9D	m	m	m	m	m	m	m	m	m	4	5	4	5	1	2,14
PREFIX *BYTES*		u	u	u	u	u	u	u	u	u						9
Assert Hardware LOCK Prefix	F0															
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*		u	u	u	u	u	u	u	u	u					1	2
Register/Memory	FF [mod 110 r/m]										2/4	4	2/4	4		
Register (short form)	5 [0 reg]										2	2	2	2		
Segment Register (ES, CS, SS, DS)	[000 sreg2 110]										2	2	2	2		
Segment Register (ES, CS, SS, DS, FS, GS)	0F [10 sreg3 000]										2	2	2	2		
Immediate	6 [10s0]†										2	2	2	2		
PUSHA *Push All General Registers*	60	u	u	u	u	u	u	u	u	u	17	17	17	17	1	2
PUSHF *Push FLAGS Register*	9C	u	u	u	u	u	u	u	u	u	2	2	2	2	1	2
RCL *Rotate Through Carry Left*		m	u	u	u	u	u	u	u	m					1	2
Register/Memory by 1	D [000w] [mod 010 r/m]										9/9	10	9/9	10		
Register/Memory by CL	D [001w] [mod 010 r/m]										9/9	10	9/9	10		
Register/Memory by Immediate	C [000w] [mod 010 r/m]†										9/9	10	9/9	10		

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
RCR *Rotate Through Carry Right* Register/Memory by 1 Register/Memory by CL Register/Memory by Immediate	 D [000w] [mod 011 r/m] D [001w] [mod 011 r/m] C [000w] [mod 011 r/m]†	m	u	u	u	u	u	u	u	m	 9/9 9/9 9/9	 10 10 10	 9/9 9/9 9/9	 10 10 10	1	2
REP INS *Input String*	F2 6[110w]	u	u	u	u	u	u	u	u	u	20+9n	20+9n	5+9n\\ 18+9n	5+9n\\ 19+9n	1	2,9
REP LODS *Load String*	F2 A[110w]	u	u	u	u	u	u	u	u	u	4+5n	4+5n	4+5n	4+5n	1	2
REP MOVS *Move String*	F2 A[010w]	u	u	u	u	u	u	u	u	u	5+4n	5+4n	5+4n	5+4n	1	2
REP OUTS *Output String*	F2 6[111w]	u	u	u	u	u	u	u	u	u	20+4n	20+4n	5+4n\\ 18+4n	5+4n\\ 19+4n	1	2,9
REP STOS *Store String*	F2 A[101w]	u	u	u	u	u	u	u	u	u	3+4n	3+4n	3+4n	3+4n	1	2
REPE CMPS *Compare String* (Find non-match)	F3 A[011w]	m	u	u	u	m	m	m	m	m	5+8n	5+8n	5+8n	5+8n	1	2
REPE SCAS *Scan String* (Find non-AL/AX/EAX)	F3 A[111w]	m	u	u	u	m	m	m	m	m	4+5n	4+6n	4+5n	4+6n	1	2
REPNE CMPS *Compare String* (Find match)	F2 A[011w]	m	u	u	u	m	m	m	m	m	5+8n	5+8n	5+8n	5+8n	1	2
REPNE SCAS *Scan String* (Find AL/AX/EAX)	F2 A[111w]	m	u	u	u	m	m	m	m	m	4+5n	4+6n	4+5n	4+6n	1	2

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:
1) Exception 13 fault (general protection) will occur in Read 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) LOCK is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.
9) An exception 13 fault occurs if CPL is greater than IOPL.
10) This instruction may be executed in Real Mode. in Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.
11) An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).
12) Any violation of privilege rules as apply to the selector operand does not cause a Protection exception, rather, the zero flag is cleared.
13) For segment load operations, the CPL, RPL, and DPL must agree witht he privolege rules to avoid an exception 13 fault. The segment's descriptor must indicate "present" or exception 11 (DS, DS, ES, FS, GS not present). If the SS register is loaded and a stack segment not present is detected, and exception 12 occurs.
14) 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.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
RET *Return from Subroutine*		u	u	u	u	u	u	u	u	u					1	2,5,6,7,8
Within Segment	C3										10		10			
Within Segment Add Immediate to SP	C2§										10		10			
Intersegment	CB										13	13	26	26		
Intersegment Add Immediate to SP	CA§										13	13	26	27		
Protected Mode: Different Privilege Level																
Intersegment													69	72		
Intersegment Add Immediate to SP													69	72		
ROL *Rotate Left*		m	u	u	u	u	u	u	u	m					1	2
Register/Memory by 1	D [000w] [mod 000 r/m]										2/4	6	2/4	6		
Register/Memory by CL	D [001w] [mod 000 r/m]										3/5	7	3/5	7		
Register/Memory by Immediate	C [000w] [mod 000 r/m]†										2/4	6	2/4	6		
ROR *Rotate Right*		m	u	u	u	u	u	u	u	m					1	2
Register/Memory by 1	D [000w] [mod 001 r/m]										2/4	6	2/4	6		
Register/Memory by CL	D [001w] [mod 001 r/m]										3/5	7	3/5	7		
Register/Memory by Immediate	C [000w] [mod 001 r/m]†										2/4	6	2/4	6		
RSDC *Restore Segment Register and Descriptor*	0F 79 [mod sreg3 r/m]	u	u	u	u	u	u	u	u	u		14		14	16	16
RSLDT *Restore LDTR and Descriptor*	0F 78 [mod 000 r/m]	u	u	u	u	u	u	u	u	u		14		14	16	16
RSM *Resume from SMM Mode*	oF AA	u	u	u	u	u	u	u	u	u		58		58	16	16
RSTS *Restore TSR and Descriptor*	0F 7D [mod 000 r/m]	u	u	u	u	u	u	u	u	u		14		14	16	16
SAHF *Store AH in FLAGS*	9E	u	u	u	u	m	m	u	m	m	2		2			
SAL *Shift Left Arithmetic*		m	u	u	u	m	m	u	m	m						
Register/Memory by 1	D [000w] [mod 100 r/m]										2/4	6	2/4	6		
Register/Memory by CL	D [001w] [mod 100 r/m]										3/5	7	3/5	7		
Register/Memory by Immediate	C [000w] [mod 100 r/m]†										2/4	6	2/4	6		
SAR *Shift Right Arithmetic*		m	u	u	u	m	m	m	m	m						
Register/Memory by 1	D [000w] [mod 111 r/m]										2/4	6	2/4	6		
Register/Memory by CL	D [001w] [mod 111 r/m]										3/5	7	3/5	7		
Register/Memory by Immediate	C [000w] [mod 111 r/m]†										2/4	5	2/4	8		

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
SBB *Integer Subtract with Borrow*		m	u	u	u	m	m	m	m	m					1	2
Register to Register	1 [10dw] [11 reg r/m]										1		1			
Register to Memory	1 [100w] [mod reg r/m]										3	5	3	5		
Memory to Register	1 [101w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 011 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator (short form)	1 [110w]†										1		1			
SCAS *Scan String*	A [111w]	m	u	u	u	m	m	m	m	m	5	5	5	5	1	2
SETB/SETNAE/SETC *Set Byte on Below/ Not Above or Equal/Carry*		u	u	u	u	u	u	u	u	u						2
To Register/Memory	0F 92[mod 000 r/m]										2/2	2	2/2	2		
SETBE/SETNA *Set Byte on Below or Equal/ Not Above*		u	u	u	u	u	u	u	u	u						2
To Register/Memory	0F 96 [mod 000 r/m]										2/2	2	2/2	2		
SETE/SETZ *Set Byte on Equal/Zero Register/ Memory*	0F 94 [mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETL/SETNGE *Set Byte on Less/ Not Greater or Equal*		u	u	u	u	u	u	u	u	u						2
To Register/Memory	0F 9C[mod 000 r/m]										2/2	2	2/2	2		
SETLE/SETNG *Set Byte on Less or Equal/ Not Greater*		u	u	u	u	u	u	u	u	u						2
To Register/Memory	0F 9E[mod 000 r/m]										2/2	2	2/2	2		

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:

1) 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) $\overline{\text{LOCK}}$ is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert $\overline{\text{LOCK}}$ to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.

16) All memory accesses using this instruction are non-cacheable as this instruction uses SMM address space.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
SETNB/SETAE/SETNC *Set Byte on Not Below/ Above or Equal/Not Carry* To Register/Memory	0F 93[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNBE/SETA *Set Byte on Not Below or Equal/ Above* To Register Memory	0F 97[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNE/SETNZ *Set Byte on Not Equal/ Not Zero* To Register/Memory	0F 95[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNL/SETGE *Set Byte on Not Less/ Greater or Equal* To Register/Memory	0F 9D [mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNLE/SETG *Set Byte on Not Less or Equal/Greater* To Register/Memory	0F 9F[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNO *Set Byte on Not Overflow* To Register/Memory	0F 91[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNP/SETPO *Set Byte on Not Parity/ Parity Odd* To Register/Memory	0F 9B[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETNS *Set Byte on Not Sign* To Register/Memory	0F 99[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETO *Set Byte on Overflow* To Register/Memory	0F 90[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETP/SETPE *Set Byte on Parity/Parity Even* To Register/Memory	0F 9A[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SETS *Set Byte on Sign* To Register/Memory	0F 98[mod 000 r/m]	u	u	u	u	u	u	u	u	u	2/2	2	2/2	2		2
SGDT *Store GDT Register* To Register/Memory	0F 01[mod 00 r/m]	u	u	u	u	u	u	u	u	u	6	6	6	6	1,10	2
SHL *Shift Left Logical* Register/Memory by 1 Register/Memory by CL Register/memory by Immediate	D [000w] [mod 100 r/m] D [001w] [mod 100 r/m] C [000w] [mod 100 r/m]†	m	u	u	u	m	m	u	m	m	2/4 3/5 2/4	6 7 6	2/4 3/5 2/4	6 7 6	1	2

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
SHLD *Shift Left Double*		u	u	u	u	m	m	u	m	m						
Register/memory by Immediate	0F A4[mod reg r/m]†										1/3	5	1/3	5		
Register/Memory by CL	0F A5[mod reg r/m]										3/5	7	3/5	7		
SHR *Shift Right Logical*		m	u	u	u	m	m	u	m	m					1	2
Register/Memory by 1	D [000w] [mod 101 r/m]										2/4	6	2/4	6		
Register/Memory by CL	D [001w] [mod 101 r/m]										3/5	7	3/5	7		
Register/Memory by Immediate	C [000w] [mod 101 r/m]†										2/4	5	2/4	6		
SHRD *Shift Right Double*		u	u	u	u	m	m	u	m	m						
Register/Memory by Immediate	0F AC[mod reg r/m]†										1/3	5	1/3	5		
Register/Memory by CL	0F AD[mod reg r/m]										3/5	7	3/5	7		
SIDT *Store IDT Register*		u	u	u	u	u	u	u	u	u					1,10	2
To Register/Memory	0F 01[mod 001 r/m]										6	6	6	6		
SLDT *Store LDT Register*		u	u	u	u	u	u	u	u	u					3	2
To Register/Memory	0F 00[mod 000 r/m]												1/2	2		
SMSW *Store Machine Status Word*	0F 01[mod 100 r/m]	u	u	u	u	u	u	u	u	u	1/2	2	1/2	2	1,10	2,11
STC *Set Carry Flag*	F9	u	u	u	u	u	u	u	u	1	1		1			
STD *Set Direction Flag*	FD	u	1	u	u	u	u	u	u	u	1		1			
STI *Set Interrupt Flag*	FB	u	u	1	u	u	u	u	u	u	7		7			9

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes:
1) 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.
2) 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.
3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
4) An exception may occur, depending on the value of the operand.
5) $\overline{\text{LOCK}}$ is asserted during descriptor table accesses.
6) All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert $\overline{\text{LOCK}}$ to maintain descriptor integrity in multiprocessor systems.
7) JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
8) 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.
9) An exception 13 fault occurs if CPL is greater than IOPL.
10) This instruction may be executed in Real Mode. in Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.
11) An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).
16) All memory accesses using this instruction are non-cacheable as this instruction uses SMM address space.

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		O F	D F	I F	T F	S F	Z F	A F	P F	C F	REG/ CACHE HIT	CACHE MISS	REG/ CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
STOS *Store String*	A [101w]	u	u	u	u	u	u	u	u	u	3	3	3	3	1	2
STR *Store Task Register*		u	u	u	u	u	u	u	u	u					3	2
To Register/Memory	0F 00[mod 001 r/m]												1/2	2		
SUB *Integer Subtract*		m	u	u	u	m	m	m	m	m					1	2
Register to Register	2 [10dw] [11 reg r/m]										1		1			
Register to memory	2 [100w] [mod reg r/m]										3	5	3	5		
Memory to Register	2 [101w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 101 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator (short form)	2 [110w]†										1		1			
SVDC *Save Segment Register and Descriptor*	0F 78 [mod sreg3 r/m]	u	u	u	u	u	u	u	u	u		22		22	16	16
SVLDT *Save LDTR and Descriptor*	0F 7A [mod 000 r/m]	u	u	u	u	u	u	u	u	u		22		22	16	16
SVTS *Save TSR and Descriptor*	0F 7C [mod 000 r/m]	u	u	u	u	u	u	u	u	u		22		22	16	16
TEST *Test Bits*		0	u	u	u	m	m	u	m	0					1	2
Register/Memory and Register	8 [010w] [mod reg r/m]										1/3	5	1/3	5		
Immediate Data and Register/Memory	F [011w] [mod 000 r/m]†										1/3	5	1/3	5		
Immediate Data and Accumulator	A [100w]†										1		1			
VERR *Verify Read Access*		u	u	u	u	u	m	u	u	u					3	2,5,6,12
To Register/Memory	0F 00[mod 100 r/m]												9/10	12		
VERW *Verify Write Access*		u	u	u	u	u	m	u	u	u					3	2,5,6,12
To Register/Memory	0F 00[mod 101 r/m]												9/10	12		
WAIT *Wait Until FPU Not Busy*	9B	u	u	u	u	u	u	u	u	u	5	5	5	5		
WBINVD *Write-Back and Invalidate Cache*	0F 09	u	u	u	u	u	u	u	u	u	4		4			
XADD *Exchange and Add*		m	u	u	u	m	m	m	m	m						
Register1, Register2	0FC[000w] [11 reg2 reg1]										3		3			
Memory, Register	0FC[000w] [mod reg r/m]										6	6	6	6		
XCHG *Exchange*		u	u	u	u	u	u	u	u	u					1,15	2,15
Register/Memory with Register	8 [011w] [mod reg r/m]										3/4	4	3/4	4		
Register with Accumulator	9 [0 reg]										3		3			

Table 7–17. Instructions, Opcodes, Flags, and Clock Summary (Continued)

INSTRUCTION	OPCODE	FLAGS									REAL MODE CLOCKS		PROTECTED MODE CLOCKS		NOTES	
		OF	DF	IF	TF	SF	ZF	AF	PF	CF	REG/CACHE HIT	CACHE MISS	REG/CACHE HIT	CACHE MISS	READ MODE	PROTECTED MODE
XLAT *Translate Byte*	D7	u	u	u	u	u	u	u	u	u	3	5	3	5		2
XOR *Boolean Exclusive OR*		0	u	u	u	m	m	u	m	0					1	2
Register to Register	3 [00dw] [11 reg r/m]										1		1			
Register to Memory	3 [000w] [mod reg r/m]										3	5	3	5		
Memory to Register	3 [001w] [mod reg r/m]										3	5	3	5		
Immediate to Register/Memory	8 [00sw] [mod 110 r/m]†										1/3	5	1/3	5		
Immediate to Accumulator (short form)	3 [010w]†										1		1			

† = immediate data ‡ = 8-bit displacement § = 16-bit displacement ¶ = 32-bit displacement m = Flag modified u = Flag unchanged

Notes: 1) Exception 13 fault (general protection) will occur in Read 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.

2) 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.

3) This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).

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

5) LOCK is asserted during descriptor table accesses.

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

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

8) 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.

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

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

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

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

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

14)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.

15)LOCK is automatically asserted, regardless of the presence or absence of the LOCK prefix.

16)All memory accesses using this instruction are non-cacheable as this instruction uses SMM address space.

Appendix A

TI486 SMM Programmer's Guide

This programmers guide provides detailed information, including example code listings, macros, and instructions, pertinent to the TI486 system management modes (SMM).

Topic **Page**

A.1 SMM Overview

A.1.1 Introduction

This programmer's guide has been written to aid programmers in the creation of software using the TI486 system management mode (SMM). SMM is currently implemented in both the TI486SLC/E and the TI486DLC/E, and the 3-volt versions of each (TI486xLC/E-V) microprocessor.

For an introduction to SMM and additional information, refer to the *TI486 32-Bit Microprocessor Reference Guide*. Section A.13 describes the differences between the TI486SLC/E and the TI486DLC/E with SMM. Section A.16 contains important information concerning SMM programming.

A.1.2 SMM Implementation

SMM operation in the TI486 microprocessors is similar to related operations performed by the AMD and Intel microprocessors. Each of these three microprocessors switches into real mode upon entry into the SMM interrupt handler. Each manufacturer's CPU has unique SMM code locations. The TI CPU has a programmable location and size for the SMM memory region. Each of the manufacturer's processors saves the programmer-visible register contents upon entry and also saves the non-programmer visible register contents. The TI486 CPU automatically saves the minimal register information, reducing the entry and exit clock count to 140. This compares with Intel's clock overhead for entry and exit of 804 clocks and AMD's minimum of 694 clocks. (See Section A.11 for a comparison of SMM overhead.)

The TI486 SMM implementation provides unique instructions that save additional segment registers as required by the programmer, in addition to the x86 MOV instruction that saves the general purpose registers.

Although all three manufacturer's CPUs provide I/O trapping, the TI486 SMM simplifies identification of I/O type and instruction restarting. The TI486 SMM process is unique in its ability to permit software relocation and sizing of the SMM address region. This flexibility facilitates run time changes to SMM support. This software flexibility allows an operating system or debugger to change, modify, or disable the SMM code.

A.2 SMM Implementation

The following sections provide an overview of TI486 SMM coding and information helpful in developing SMM code.

A.2.1 Hardware Background

A.2.1.1 SMM Pins

The $\overline{\text{SMI}}$ and $\overline{\text{SMADS}}$ pins are used to implement SMM. The bidirectional $\overline{\text{SMI}}$ pin is used by the chip set to signal the CPU that an SMI has occurred. While the CPU is in the process of servicing an SMM interrupt, the same pin is used to send a signal to the chip set to indicate that the SMM processing is occurring. The $\overline{\text{SMADS}}$ address strobe is generated instead of the $\overline{\text{ADS}}$ address strobe while executing or accessing data in SMM address space.

A.2.1.2 $\overline{\text{SMI}}$ Pin Timing

In order to enter TI486 SMM mode, the $\overline{\text{SMI}}$ pin must be asserted for at least 4 CLK2 periods. Once the CPU recognizes the active $\overline{\text{SMI}}$ input, the CPU drives the SMI input low for the duration of the SMI routine. The SMI routine is terminated with an SMI specific resume RSM instruction. When the RSM instruction is executed, the CPU drives the $\overline{\text{SMI}}$ pin high for 2 CLK2 periods. The $\overline{\text{SMI}}$ pin bidirectional design:

1) Prohibits more than one SMI interrupt from becoming active.

2) Provides feedback to the chip-set/core logic that an SMI is in process.

3) Provides compatibility with other SMM hardware interfaces.

A.2.1.3 Address Strobes

The TI486 CPU has two address strobes, $\overline{\text{ADS}}$ and $\overline{\text{SMADS}}$. $\overline{\text{ADS}}$ is the address strobe used during normal operations. The $\overline{\text{SMADS}}$ address strobe replaces $\overline{\text{ADS}}$ during SMM operations when data is written, read, or fetched in the SMM defined region. Using a separate address strobe increases chip set compatibility and control.

During an SMM interrupt routine, control can be transferred to main memory via a JMP, CALL, Jcc instruction or execution of a software interrupt (INT). Execution in main memory will cause $\overline{\text{ADS}}$ to be generated for code and data outside of the defined SMM address region. (It is assumed, but not required, that the chip set ultimately translates $\overline{\text{SMADS}}$ and a particular address to some other address.) To access code in main memory that overlaps the SMM address space, the MMAC bit (CCR1, bit 3) must be set. This allows $\overline{\text{ADS}}$ strobes to be generated for MOV instructions that overlap main memory while in SMM mode. It is not possible to execute code in main memory that overlaps SMM space while in the SMM mode.

$\overline{\text{SMADS}}$ can also be generated for memory reads/writes and code fetches within the defined SMM region when the SMAC bit (CCR1, bit 2) is set while in normal mode. The generation of $\overline{\text{SMADS}}$ would permit a program in normal space to jump into SMM code space. Care should be taken to be in real mode before the jump occurs into SMM space. A routine should be followed to initialize used registers to their real-mode state. The RSM instruction should not be used after jumping into SMM space unless return information is first written into the SMM context area before the RSM instruction is executed.

A.2.1.4 Chip-Set $\overline{\textit{READY}}$

The TI486 CPU has one $\overline{\text{READY}}$ input. Chip sets that implement the dual READY lines can OR the two ready lines together for the single $\overline{\text{READY}}$. The AMD implementation of SMM provides for two READY lines from the chip set, one for SMM space ($\overline{\text{SREADY}}$) and one for the normal $\overline{\text{READY}}$.

A.2.2 SMM Software Considerations

A.2.2.1 SMM Handler Entry State

At the start of the SMM routine, before control is transferred to code executing at SMM base, some of the CPU state is saved at the end of SMM memory. This is one area where the TI486 CPU SMM state is unique. The CPU saves the minimum CPU state information necessary for an interrupt handler to execute and return to the interrupted context. The information is saved at the top of the defined SMM region (starting at SMM base + size – 30h). Of the typically used program registers only the CS, EFLAGS, CR0, and DR7 are saved upon entry. This requires that data accesses use a CS segment override to save other registers and access data. To use any other register the SMM programmer must first save the contents using the SVDC instruction for segment registers or MOV operations for general purpose registers (See SMM Instructions, Section A.12). It is possible to save all the CPU registers as needed.

Unique to the TI486 CPU is the saving of the previous IP, before the SMI, and the next IP to be executed after exiting the SMI handler. Upon execution of an RSM instruction, control is returned to the “next IP”. The value of the “next IP” may need to be modified for restarting OUTSx/INSx instructions; this modification is a simple move (MOV) of the “previous IP” value to the “next IP” location. Execution is then returned to the I/O instruction, rather than the instruction after the next I/O instruction. (The restarting of I/O instructions may also require modifications to the ESI, ECX, and EDI depending on the instruction. See Section A.5 for an example.)

Figure A–1 and Table A–1 describe the SMM memory space header. The P and I bits indicate whether a INSx/OUTSx and REP prefix were being executed. IN/OUT instructions will be restarted by changing “NEXT IP” and leaving the SMI handler.

The only area in the SMM header that the programmer should consider altering is the “NEXT IP”. Altering any other header values can have unpredictable results.

The EFLAGS, CR0, and DR7 registers are set to the reset values upon entry to the SMI handler. This has implications for setting break points using the debug registers. Break points can not be set prior to the SMI using debug registers. The INT 3 debug code trap technique can be used, however, prior to the occurrence of the SMI in SMM space. Once the SMI has occurred and the debugger has control in SMM space, the debug registers can be used for the remaining SMI execution.

Figure A–1. SMM Memory Space Header

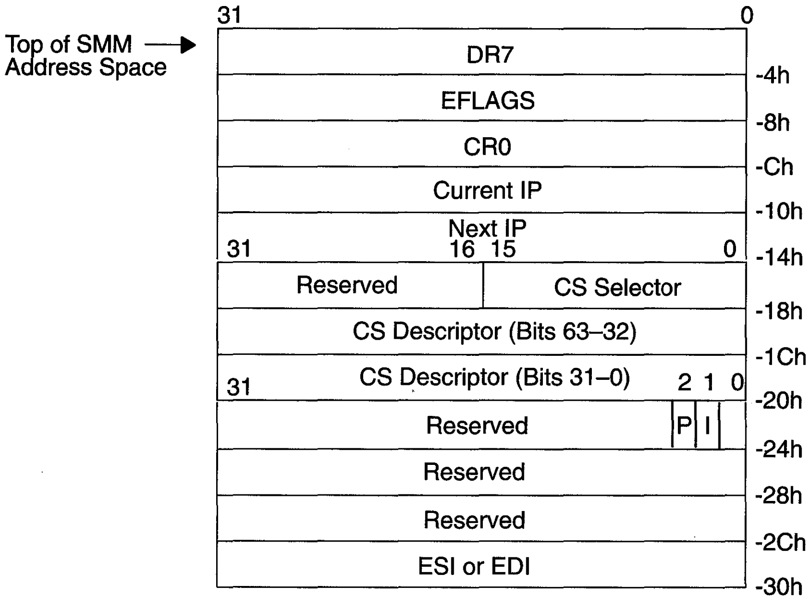


Table A–1. SMM Memory Space Header

NAME	DESCRIPTION	SIZE
DR7	The contents of the debug register 7.	4 Bytes
EFLAGS	The contents of the extended flag register.	4 Bytes
CR0	The contents of the control register 0.	4 Bytes
Current IP	The address of the instruction executed prior to servicing the SMI interrupt.	4 Bytes
Next IP	The address of the next instruction that will be executed after exiting the SMM mode.	4 Bytes
CS Selector	Code segment register selector for the current code segment.	2 Bytes
CS Descriptor	Code register descriptor for the current code segment.	8 Bytes
P	REP INSx/OUTSx Indicator P = 1 if current instruction has a REP prefix P = 0 if current instruction does not have 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
ESI or EDI	Restored ESI or EDI value. Used when it is necessary to repeat an REP OUTSx or REP INSx instruction when one of the I/O cycles caused an $\overline{\text{SMI}}$ trap	4 Bytes

Note: INSx = INS, INSB, INSW, or INSD instruction.

Note: OUTSx = OUTS, OUTSB, OUTSW, or OUTSD instruction.

A.2.2.2 Exiting the SMI Handler

When the RSM instruction is executed at the end of the SMI handler the IP is loaded from the top of the SMM at the address (SMMbase +SMMsize – 14h) called SMI_NEXTIP. This permits the instruction to be restarted. The values of ECX, ESI, and EDI, prior to the execution of the instruction that was interrupted by SMI, can be restored from information in the header that pertains to the INx and OUTx instructions. The only registers that are restored from the SMM header are CS, NEXT_IP, EFLAGS, CR0, and DR7.

A.2.2.3 Addressing SMM Code With the Same Address as Main Memory

To access main memory overlapping the SMM space (i.e., generate $\overline{\text{ADS}}$ from memory MOV instructions rather than $\overline{\text{SMADS}}$) set the MMAC bit in CCR1. The following code will enable MMAC:

```
MMAC = shl 3
      mov   al, 0c1h       ;select CCR1
      out   22h, al
      in    al, 23h        ;get CCR1 current value
      mov   ah, al         ;save it
      mov   al, 0c1h
      out   22h, al
      mov   al, ah
      or    al, MMAC       ;set MMAC
      out   23h, al
;Now all data memory access will use ADS#, Code fetches will continue to
be done ;
;     with SMADS# from SMM memory
;
;Disable MMAC
      mov              al, 0c1h       ;select CCR1
      out              22h, al
      mov              al, ah         ;get old value of CCR1
      out              23h, al        ;and restore it
```

A.2.2.4 Miscellaneous Execution Details

1) Execution of SMM code begins at the start of SMM space. This is the value entered onto the base portion of AAR4. The CS base will be set to the ARR4 SMM base, and EIP will be equal to 0.

2) The $\overline{\text{A20}}$ input to the CPU is ignored for all SMM space accesses. These are all accesses which use $\overline{\text{SMADS}}$.

3) All SMM instructions can be executed outside the SMM defined space, provided that SMAC bit is set in CCR1 or if execution of an SMI handler is in progress. (An SMI handler is “in progress” during the time the CPU is driving the SMI pin low.)

4) Setting the MMAC bit permits the reading and writing of main memory addresses that overlaps SMM memory while an SMI is in progress.

5) It is not possible to execute code in main memory that overlaps SMM memory addresses.

6) NMI is the only enabled interrupt at the entry to the SMI handler. It is advised that system designers provide latches to disable NMI while the SMI is in progress.

7) The SMI handler can execute calls, jumps, and other changes of flow and will generate software interrupts and faults using the current definition of the IDT. (Note that on entry to the SMI handler the IDT is not set to the reset real mode value of 0:0.)

8) The SMI handler can go from real mode to protected mode and vice-versa. Almost anything that can be done normally may be done during the SMI service routine.

9) Data from SMM memory is not cached.

10) If the location of SMM space is beyond 1 MByte, the value in CS will truncate the segment above 16 bits. This would prohibit doing calls or INTS from real mode without restoring the 32-bit features of the 486 because of the incorrect return address on the stack.

11) An undefined opcode exception will typically be generated when conditions are not correct to permit the execution of SMM instructions.

12) To execute outside the SMM region (BIOS, debugger, etc.) the CS limit must be changed after entry to the SMI handler. The limit of the CS segment register is set to the size of the SMM region in AAR4. This means that EIP cannot become larger than the SMM region size. Since jumps in real mode do not change the CS limit, this has implications for software interrupts and jumps out of SMM space. (See Section A.15 for details and options.

13) Segment registers other than the CS have the limits set in the non-programmer-visible portion that were present before the SMI. To avoid a protection error due to limit or other violation, the RSDC SMM instruction should be used to change the limit of the register in use. (See Section A.14.)

A.3 Enabling SMM

The enabling and setup of SMM in the TI486 CPU is done by setting all 4 of the SMM registers/bits to:

Register/Bit	Location†	Value	Description
SMI	CCR1 bit 1	1	Enable SMI pin
SM4	CCR1 bit 7	1	Make ARR4 as SMM space
SM_loc	ARR4 bits 12–4	Start SMM region	SMM base address
SM_size	ARR bits 3–0	≥ 4KB and ≤ 16MB	SMM size

† See subsection 2.3.2.4 for further information on CCR1 and ARR4.

Setup example

```
SMM Location   =   0C8000H
SMM Size       =   8KB

mov            al, 0c1h       ; index to CCR1
out            22h, al        ; select CCR1 register
in             ah, 23h        ; read current CCR1 value
or             ah, 082h       ; enable SMI and SM4 region
mov            al, 0c1h       ; index to CCR1
out            22h, al        ; select CCR1 register
out            23h, ah        ; write new value to CCR1
mov            al, 0ceh       ; index ARR4 SMM base address bits <23-16>
out            22h, al        ; select
mov            al, 0ch        ; set ARR4 SMM base address upper bits
out            23h, al        ; write value
mov            al, 0cfh       ; index ARR4 SMM base address bits <15-12>
out            22h, al        ; and 4 bits for SMM size
mov            al, 082h       ; set SMM lower address bits and SMM size
out            23h, al        ; write value
```

A.4 Instruction Summary

These instructions are valid only when CPL is 0 and either:

1) The SMAC bit is set and a valid SMM region is defined (the SMM size defined to be greater that 0), and the SMI bit of CCR1 is set and the SM4 bit of CCR1 is set.

2) The $\overline{\text{SMI}}$ pin is driven low by the CPU. (The CPU drives $\overline{\text{SMI}}$ low after it recognizes the SMI interrupt, and continues to drive it low until an RSM is executed.)

The CPU will always generate an undefined opcode fault when the above conditions are not met and one of the SMM instructions are executed.

The MACROs used in the descriptions below are excerpted from the file smimac.inc, Section A.12.

Instruction	Opcode	Parameters	Clocks
rsdc	0F 79 [mod sreg3 r/m]	sreg3, mem80	14
rsldt	0F 7B [mod 000 r/m]	mem80	14
rsm	0F AA	<none>	58
rsts	0F 7D [mod 000 r/m]	mem80	14
svdc	0F78 [mod sreg3 r/m]	mem80, sreg3	22
svldt	0F 7A [mod 000 r/m]	mem80	22
svts	0F 7C	mem80	22

A.4.1 Restore Register and Descriptor

Instruction	Opcode	Parameters	Clocks
rsdc	0F 79 [mod sreg3 r/m]	sreg3, mem80	14

Description:

Load the information at the mem80 into the sreg register and its associated descripton.

Example:

```
_ds   equ   3
$rsdc        MACRO
             db 0fh, 79h
             ENDM
cs_over      MACRO
             db 02eh
             ENDM
;
; rsdc                  ds, cs:[590h]
cs_over                            ; cs segment override
$rsdc                              ; opcode for restore register
db (06 or (_ds SHL 3)              ; build [mod sreg3 r/m]
dw   0590h                         ; displacement
```

A.4.2 Restore LDTR and Descriptor

Instruction	Opcode	Parameters	Clocks
rsldt	0F 7B [mod 000 r/m]	mem80	14

Description:

Load the local descriptor table register with the selector and the associated descriptor at the mem80 parameter

Example:

```
$rsldt        MACRO
              db 0fh, 7bh
              ENDM
cs_over       MACRO
              db 02eh
              ENDM
;
;  rsldt cs:[590]
cs_over
$rsldt
db 6                               ; build [mod sreg3 r/m]
dw  590h                           ; displacement
```

A.4.3 Resume Normal Mode

Instruction	Opcode	Parameters	Clocks
rsm	0F AA	<none>	58

Description:

RSM restores the state of the CPU from the SMM header at the top of the SMM space and exit SMM mode. This is the last instruction to be executed in SMI handler. This instruction can be executed in either SM memory or main memory.

Example:

```
$rsm        MACRO
            db 0fh, 0aah
            ENDM
;
mov ax, word ptr cs: [520]          ; restore
mov bx, word ptr cs: [522]          ; some
mov cx, word ptr cs: [524]          ; registers
rsm                                 ;  last instruction in
                                    ;  SMI handler
```

CAUTION

This instruction should be executed only after SMI has occurred or if the programmer has created the correct SMM header at the top of SMM space.

A.4.4 Restore TSR and Descriptor

Instruction	Opcode	Parameters	Clocks
rsts	0F 7D [mod 000 r/m]	mem80	14

Description:

This instruction restores the task state register in the CPU from the mem80 location.

Example:

```
$rsts       MACRO
            db 0fh, 7dh
            ENDM
cs_over     MACRO
            db 02eh
            ENDM
;
;  rsts cs: [600h]
cs_over
$rsts                       ; build [mod sreg3 r/m], 6 == DS: [d16]
db 6                        ; displacement
```

A.4.5 Save Register and Descriptor

Instruction	Opcode	Parameters	Clocks
svdc	0F 78 [mod sreg3 r/m]	mem80, sreg3	22

Description:

This instruction saves the 80486 segment register into the 80-bit mem80 location.

Example:

```
$svdc          MACRO
               db 0fh, 78h
               ENDM
cs_over        MACRO
               db 02eh
               ENDM
_DS            EQU 3
;
;  svdc cs: [680h] , DS
cs_over
$svdc
db (06 or (_DS SHL 3))                   ;  build [mod sreg3 r/m]
dw   680h                                ;  displacement
```

Note: The non-programmer-visible information of the segment register is also saved. See Section A.14 for a description of the storage format.

A.4.6 Save LDTR and Descriptor

Instruction	Opcode	Parameters	Clocks
svldt	0F 7A [mod 000 r/m]	mem80	22

Description:

This instruction stores local descriptor table register to the 80-bit mem80 location.

Example:

```
$svldt     MACRO
           db 0fh, 7ah
           ENDM
cs_over    MACRO
           db 02eh
           ENDM
;
;  svldt cs: [700h]
cs_over
$svldt
db 6                               ;  build [mod sreg3 r/m]
dw  700h                           ;  displacement
```

Note: The non-programmer-visible information about the register is also saved.

A.4.7 Save TS and Descriptor

Instruction	Opcode	Parameters	Clocks
svts	0F 7C	mem80	22

Description:

Save the task register and associated descriptor into the 80-bit mem80 location.

Example:

```
$svts      MACRO
           db 0fh, 7ch
           ENDM
cs_over    MACRO
           db 02eh
           ENDM
;
;  svts cs: [780h]
cs_over
$svts
db 6                              ;  build [mod sreg3 r/m]
dw  780h                          ;  displacement
```

Note: The non-programmer-visible information about the segment register is also saved.

A.5 SMI Handler Example

This section contains fragments of typical coding found within TI486 SMI handlers.

```
      SMBASE= 0C8000H                 ; base address of SMM space
      SMSIZE= 2                       ; SMM space size is 8k bytes
      SMEND = SMSIZE SHL (SMSIZE-1)   ;works for most cases

      INCLUDE smimac.inc              ;see Section A.12
      .MODEL SMALL
      .386P
      .CODE

COMMENT ^
Execution begins here in real mode, with CS defined at the SMBASE and EIP=0
^
      public smi_start
      smi_start:
            jmp      $skipdata        ;skip data area, makes it easy for
                                      ;assembler
      EAXsavedd      ?
      DSsavedt       ?
      DStempb        0ffh, 0ffh, 0,0,0,92h,8fh,0,0,0  ;4gig present segment
      $skipdata:
            mov      dword ptr cs:[EAXsave],eax; save EAX
            svdc     cs:,[DSsave], ds     ; save DS
            rsdc     ds,cs:,[DStemp]      ; setDS
            mov      ax, cs
            mov      ds, ax
COMMENT ^
We need to extend the limits of DS so that we don't get a fault when we use
it to access low memory. It may be not present with a limit of 0, and these
values won't be changed when we set it using a real mode load.
```

```
;Determine Why Are We In The SMI Handler

;
COMMENT ^
Chip set/Core logic unique instructions will follow. The chip set will be
used to determine what caused the SMM interrupt to occur. The BIOS could also
"jump" to this point in the SMM region.
^
;
;Decision Tree:

        ;
        ;a)     If timer, go to timer_expired
        ;
        ;b)     If port i/o occurred to a trapped location, go to port_io_caused
        ;
        ;c)     If the cpu was turned off, go to cpu_turned_off
;
;timer_expired;

;
COMMENT ^
A chip set timer has expired. Unique code would appear to determine which
timer. Depending on the purpose of the timer the handler could;
;
;       1)      Reduce the clock frequency
;       2)      Execute a halt instruction and enter suspend mode
;       3)      Turn current off to the CPU
;       4)      Turn off a peripheral device
;       5)      Reset the timer and increment a counter
;
^
reduce_clock:

COMMENT ^
To go to a lower CPU current requirement the CPU clock can be reduced. The
CPU clock can be reduced from its current setting to a lower value. That
value could be zero. Since the CPU is a static device and will maintain the
state of all its registers in the absence of a clock input there is no state
saving requirement. It is assumed that by writing to the chip set it will
reduce or zero the clock. If the clock is stopped then the next instruction
to be executed will be one in this SMI handler immediately following the
point where the chip set turned the clock off.
^

        jmp end_timer:
```

execute_halt:

```
COMMENT ^
To go to a lower CPU current consumption the SMI handler will now execute a
HALT instruction. The HALT instruction will put the CPU into a low power
sleep mode until a non-SMI interrupt occurs. Interrupt(s) will need to be
enabled to permit the interrupt to wake-up the CPU. A common choice would be
the keyboard interrupt. A flag would need to be set in main memory to
indicate that the SMI handler should be jumped into or SMI created, to permit
it to restore the state/context of the CPU, prior to the halt for servicing
the interrupt. The interrupt in low memory must point to the BIOS handler for
the return to be made to the SMI handler. An interrupt handler in SMM space
could also service the interrupt rather than a BIOS routine.
^
;
    ;[ Alternatively the chip set could pull the SUSP# CPU pin low to enter ]
    ;[ suspend mode. The chip set would have to pull SUSP# high to exit ]
    ;[ suspend mode. ]

    :To be sure that BIOS will get control on intr
    ;check for keyboard interrupt vector pointing to BIOS
    ;if not BIOS, save existing and set to BIOS vector or jump to
            can_not_halt
    ;Set a flag in main memory indicating SMI HALT executed
    ;If an SMM space interrupt handler is used then IDTR and/or the vector
    ;would need to be updated to the SMM space routine.
    mov ax, 0        ; point to bottom segment
    mov ds, ax       ; ds segment is now in main memory
    mov [485], 1     ; set BIOS flag in main memory
                     ;<set cpu state for bios int>
    halt             ; last instruction executed here
    ;<the chip set could remove the clock to go to suspend mode now>
    nop

    can_not_halt:    ;CPU state will not be correct at interrupt

    jmp end_timer
```

turn_off_cpu:

```
; set bit in main memory to indicate to the BIOS that SMI handler
; turned power off to CPU and CPU state should be restored by
; the SMI handler
;
     mov ax, 0              ; point to bottom segment
     mov ds, ax             ; ds segment is now in main memory
     mov [485], 1           ; set BIOS flag in memory
;
;    (save entire CPU state. See Restore CPU state label)
;    (Chip set specific instructions to be executed to remove power to
;    cpu)
;    jmp end_timer
```

turn_off_peripheral:

```
; Chip set specific instructions to turn off peripheral and enable
; chip set I/O trapping of the devices io range or enable timer
; to allow polling of peripheral requirements.
jmp end_timer
```

reset_timer:

```
; Chip set specific instructions to be executed to reset a timer and
; possibly increment a counter to maintain number to time out occurred
; for a particular device.
     jmp end_timer
```

end_timer:

```
jmp done
```

port_io_caused:

```
COMMENT ^
The TI486 SMM support for I/O being interrupted provides information that
permits the restarting of the I/O instruction without investigating the
actual code where the instruction is located.

Many things can be done at this point beyond turning on a powered down
peripheral. The CPU clock could now be speeded up in anticipation of heavy
CPU processing requirements, timers could be reset, etc.
^
      ;** Restart the interrupted instruction

            mov                       eax,dword ptr [SMEND+SMI_PREVIOUSIP]
            mov                       dword ptr [SMEND+SMI_NEXTIP],eax
            mov                       al,byte ptr cs:[SMEND+SMI_BITS]
      ;test for REP instruction
            bt                        al,2        ;rep instruction?
                                                  ;(result to Carry)
            adc                       ecx,0       ;if so, increment ecx
            test                      al,1 shl 1  ;test bit 1 to see
                                                  ;if an OUTS or INS
            jnz                       out_instr

      COMMENT ^
      ** A port read (INx) instruction caused the chip set to generate an
SMI instruction. Restore EDI saved by SMI microcode.
      ^
            mov                       edi, dword ptr cs:[SMEND+SMI_EDIESI]
            jmp                       common1
      out_instr:

      COMMENT ^
      ** A port write (OUTx) instruction caused the chip set to generate an
SMI instruction. Restore ESI saved by SMI microcode.
      ^
            mov                       esi, dword ptr cs:[SMEND+SMI_EDIESI]
      common1:
            jmp     done
```

cpu_turned_off:

```
COMMENT ^
This handler turned off the current to the CPU. Before it did, the handler
set a bit in main memory or battery backed-up CMOS indicating that this event
happened. At reset, BIOS will determine that this was the case and "jump"
into the SMI handler. SMI handler will then restore the entire state/context
of the CPU prior to current being removed. The bit in main memory would also
be cleared indicating that the SMI handler had removed current.
^
      mov ax, 0                        ; point to bottom segment
      mov ds, ax                       ; ds segment is now in main memory
      mov [485], 0                     ; clear BIOS flag in main memory
      mov ax, cs                       ; restore ds to SMM area
      mov ds, ax
```

{Restore Complete CPU State}

```
      ;     eax
      ;     ebx
      ;     ecx
      ;     edx
      ;     edi
      ;     esi
      ;     ebp
      ;     esp
      ;     cs       ;use rsdc
      ;     ds       ;use rsdc
      ;     ss       ;use rsdc
      ;     es       ;use rsdc
      ;     fs       ;use rsdc
      ;     gs       ;use rsdc
      ;     ldtr
      ;     gdtr
      ;     idtr
      ;     tr
      ;     eflags
      ;     cr0
      ;     cr2
      ;     cr3
      ;     dr0
      ;     dr1
      ;     dr2
      ;     dr3
      ;     dr6
      ;     dr7
      ;     ccr0
      ;     ccr1
      ;     ccr2
      ;     Save the configuration registers with index C3h through FFh
      ;      for future TI486 product compatibility

      ;      arr1
      ;      arr2
      ;      arr3
      ;      arr4
      jmp done

      done:
             mov      eax,cs:[EAXsave]
             rsdc     ds,cs:,[DSsave]
             rsm
return
```

A.6 Testing/Debugging SMM Code

There are several ways to debug SMM code:

1) Emulation Technology TI486SLC/E pod with an HP 16500/550 Logic Analyzer.
    - Supports selective trace capture
    - SMM instruction disassembly
2) Periscope – Software only
    - Full screen debugging
    - TSR
    - Single stepping and break points
3) DOS debug – Software only
    - Single stepping and break points
4) Other selected logic analyzers

A.6.1 Software Only Debugging

It is possible to write an SMI handler and debug it as a TSR. You will need to use a debugger that can set break points at any address in memory. Use the following code sequence as a model of how to build your SMI handler as a TSR. This code sequence also contains a section that loads the CS non-programmer-visible section to change the limit. This is required so that a protection error will not occur whenever code is executed outside of the SMM region. It is assumed that $\overline{\text{ADS}}$ and $\overline{\text{SMADS}}$ from the CPU are ORed together by the chip set or external logic. Also, the chip set should support programmable SMM locations.

This code will mark the SMI handler address in the user interrupt INT 66 location (0:198h). This is done so that the programmer can determine the location of the SMM region and set break points.

The debugger will only be able to set a code break point using INT 3 outside of the SMI handler. This is because the debug control register DR7 is set to the reset value upon entry to the SMI handler. This causes break conditions in DR0–3 to be disabled. Debug registers can be used if set after entry to the SMI handler and DR0–3 are saved.

Using a TSR to debug SMI has some limitations:

- Other code could overwrite the region.
- Jumps or calls must be to known offsets.

What follows is an example that can be used for the first step in debugging of SMI code:

```
      .MODEL SMALL
      .STACK
      .386P
      INCLUDE smimac.inc

RD_WR EQU            12h               ;read/write
EX_RD EQU            1Ah               ;execute/readable

COMMENT ^
This is an example of SMI code which can exist below the 1 MByte boundary. It
must be before the 1 MByte boundary because it uses the value in the cs
register in order to form fixups based on its location as well as for the
jump to return to real mode.
^

      .CODE

smi_handler:
      jmp    $over                     ;pass data area for assembler
      db     100 dup (?)
stacksmilabel
;
;our smi handler gdt
;
gdt   dq     0                         ;null

ADDR = 0
LIMT = 100000h
g_big  = $ - gdt
      dw     (LIMT-1 and 0ffffh)
      dw     (ADDR and 0ffffh)
      db     ((ADDR SHR 16) and 0ffh)
      db     RD_WR OR (0 SHL 5) OR (1 SHL 7)
      db     (((LIMT-1) SHR 16) AND 0fh) OR (0 SHL 6) OR (1 SHL 7)
      db     ((ADDR SHR 24) and 0ffh)
g_code  = $-gdt
ADDR = 0
LIMT = 100000h
      dw     (LIMT-1 and 0ffffh)
      dw     (ADDR and 0ffffh)
      db     ((ADDR SHR 16) and 0ffh)
      db     EX_RD OR (0 SHL 5) OR (1 SHL 7)
      db     (((LIMT-1) SHR 16) AND 0fh) OR (0 SHL 6) OR (1 SHL 7)
      db     ((ADDR SHR 24) and 0ffh)
```

```
GDTSIZE = ($-gdt)

csareadb            10 dup (?)
dsareadb            10 dup (?)
ssareadb            10 dup (?)
esareadb            10 dup (?)
fsareadb            10 dup (?)
gsareadb            10 dup (?)
tsareadb            10 dup (?)

gdtsave df?
gdtnewdw     GDTSIZE - 1
             dd  ?                    ;address

eaxsave dd?
ebxsave dd?
ecxsave dd?
edxsave dd?
espsave dd?

$over:
COMMENT ^
The debugger may want to use ss,ds,es,fs,gs. The limits may be shortened if
the program had been running in protected mode. We therefore extend the
limits of these registers before we enable the debugger.
^
      svdc   cs:,[ssarea],ss          ;save the stack pointer
      svdc   cs:,[dsarea],ds
      svdc   cs:,[esarea],es
      svdc   cs:,[fsarea],fs
      svdc   cs:,[gsarea],gs
      mov    cs:[eaxsave],eax
      mov    cs:[ebxsave],ebx
      mov    cs:[espsave],esp

COMMENT ^
Clear VM flag in Eflags (See Section A.16).
^

      rsdc   ss, cs:, [gdt+g_big]
      mov    esp, offset smistack
      mov    ax, cs
      mov    ss, ax
      mov    eax, 0
      push   eax
      mov    eax, cs
      push   eax, offset @F
      push   eax
      iretd
@@:
      sgdt   fword ptr cl: [gdtsave]
COMMENT ^
fixup code for smi base
^
;patch gdt
      mov    eax,cs                             ;segment of us here
      shl    eax,4
      mov    ebx,offset gdt                     ;offset to here
      add    ebx,eax
      mov    dword ptr [gdtnew+2],ebx           ;define gdt base
;patch far jump into protected mode
      mov    ebx,offset $next0
      add    ebx,eax
      mov    dword ptr cs:[patch1],ebx
;patch far jump back to real mode
      mov    word ptr cs:[patch2],cs

start here

COMMENT ^
extend the limits for the code segment
^
      db     66h
      lgdt   fword ptr [gdtnew]
      mov    eax,cr0
      or     al,1
      mov    cr0,eax
      db     66h
      db     0eah
patch1dd     ?
      dw     g_code

$next0:      mov     bx,g_big        ;extend the limits of the data segments
      mov    ss,bx
      mov    ds,bx
      mov    es,bx
      mov    fs,bx
      mov    gs,bx
      xor    al,1
      mov    cr0,eax                 ;back to real mode
      db     0eah
      dw     offset $next1
patch2dw     ?                       ;far jump to set cs and writable bit
$next1:


COMMENT ^
define a valid stack
^
      mov    ax,cs
      mov    ss,ax
      mov    esp,offset stacksmi
```

```
COMMENT ^
****** Insert user specific smi code here & set breakpoints. ******
^
      db    66h
      lgdt  fword ptr cs:[gdtsave]
      rsdc  ss,cs:,[ssarea]
      rsdc  ds,cs:,[dsarea]
      rsdc  es,cs:,[esarea]
      rsdc  fs,cs:,f[sarea]
      rsdc  gs,cs:,[gsarea]
      mov   eax,dword ptr cs:[eaxsave]
      mov   ebx,dword ptr cs:[ebxsave]
      mov   esp,dword ptr cs:[espsave]
      rsm
smi_handlere:
SMI_SIZE = offset smi_handlere - offset smi_handler
Install PROC

;***** Enable SMM Region ******
; Don't enable SMI yet because we're not ready for it.
      mov   al, 0c1h                  ;select CCR1
      out   22h,al
      in    al, 23h                   ;read CCR1
      or    al, 80h                   ;enable SMADS# and SMM region (not SMI)
      mov   ah, al
      mov   al, 0c1h                  ;select CCR1
      out   22h, al
      mov   al, ah
      out   23h, al                   ;write new CCR1 value

      mov   eax,offset endresident
      mov   ebx,cs
      shl   ebx,4
      add   eax,ebx
      add   eax,0fffh
      and   eax,NOT 0fffh             ;eax = start of smi space
      mov   edx,eax
      push  edx
```

```
;****************************************************************************
; * Load SMI address and size into ARR4
;
;******                          cd         ce            cf
;******                    -----------  -----------  ------------
;****** Config Reg         31-28 27-24, 23-20 19-16, 15-12 <size>
;****** Address            31-28 27-24, 23-20 19-16, 15-12 11-8,  7-4 3-0

        mov     al, 0cdh             ;region 4 1st word
        out     22h, al
        mov     eax, edx             ; get smi handler address
        shr     eax, 24              ;move address <31-24> to al
        out     23h, al              ; [7-0]=>smbase[31-24]

        mov     al, 0ceh             ;region 4 2nd word
        out     22h, al
        mov     eax, edx             ; get smi handler address
        shr     eax, 16              ; move address <23-16> to al
        out     23h, al              ; [7-0]=>smbase[23-16]

        mov     al, 0cfh             ;region 4 3rd word
        out     22h, al
        mov     eax, edx             ; get smi handler address
        shr     eax, 8               ; move address <15-12> to al
        and     al, 0f0h             ; clear bottom nibble
        or      al, 1                ; select 4KB SMI size
        out     23h, al              ; and [3-0]=>smsize
;****************************************************************************
        pop     edx                  ;start of smi area
        mov     eax,edx
        add     edx,1000h            ;reserve 4k for smi handler
        mov     ebx,es               ;current psp
        shl     ebx,4                ;
        sub     edx,ebx              ;bytes to reserve
        she     edx,4                ;paragraphs to reserve in dx
        push    dx
        shr     eax,4                ;paragraph of smi handler
        mov     es,ax                ;save for later
        mov     ds,ax
        mov     dx,0                 ;always starts at 0
        mov     ax, 2566h            ;int 66h vector at 0:198h
        int     21h
        pop     dx                   ;tsr address
```

```
;move the code to the smi_area
        mov      al, 0c1h              ;select CCR1
        out      22h, al
        in       al, 23h               ;read CCR1
        mov      ah, al                ;save old value
        mov      al, 0c1h              ;select CCR1
        out      22h, al
        mov      al, ah                ;get old value
        or       al, 04h               ;enable SMAC
        out      23h,al                ;be clean on ah for later
RELOCATE = 0
IF RELOCATE
        sub      esi,esi
        sub      edi,edi
        mov      cx,cs
        mov      ds,cx
        mov      ecx, (SMI_SIZE+3)/4
        rep      movs dword ptr es:[edi],dword ptr ds:[esi]
ELSE
;put the far jump at the start of the smi_area to above code
        mov      byte ptr es:[0],0eah
        mov      word ptr ex:[1],offset smi_handler
        mov      word ptr ex:[3],cs
ENDIF
;restore smi state and enable SMI
        mov      al, 0c1h              ;select CCR1
        out      22h, al
        mov      al, ah                ;get old value
        or       al, 02h               ;set SMI bit to enable SMI
        out      23h,al                ;be clean on ah for later
COMMENT ^
SMIs may happen at any time now.
^
;dx = offset in this segment to tsr
        mov      ax, 3100h             ; Request function 31h, error code=0
        int      21h                   ; Terminate-and-Stay-Resident
Install ENDP
;----end of resident code----
endresident      label byte

        db  2000h  dup (?)

        END      Install

;***************************************************************************
```

A.7 TI486 Power Management Features

The TI486 CPU provides several methods and levels of power management. The fully static design, suspend mode, system management mode (SMM), and 3.3-volt operation can be used to achieve optimum CPU and system power management. The following table summarizes the various power management options for the TI486:

Option	Power Savings
Reduced Clock Frequency	$I_{CC} = (12 \times f_{CLK2\ (MHz)}) + 150$ mA @ 5 V
Lower Supply Voltage (V_{CC})	$I_{CC} = (130 \times V_{CC}) - 256$ mA @ 25 MHz
Suspend Mode	2% of typical I_{CC}
Remove Clock	25% of typical I_{CC}
Suspend Mode and Remove Clock	400 μA
Remove Power	0 μA

A.7.1 Reducing the Clock Frequency

The TI486 CPU is a fully static design meaning that the input clock frequency can be reduced or stopped without a loss of internal CPU data or state. The system designer can make decisions to reduce the clock by utilizing the SMM capabilities to support Advanced Power Management (APM) software API in concert with chip set capabilities. When the clock is removed then restarted, CPU execution will begin with the instruction where the clock was removed.

A.7.2 Suspend Mode

The TI486 CPU supports suspend mode operation that can be entered either through software or hardware initiation.

Software initiates suspend mode through execution of a HALT instruction. After HALT is executed, the CPU enters suspend mode and asserts suspend acknowledge ($\overline{\text{SUSPA}}$), if enabled.

Hardware initiates suspend mode by using the $\overline{\text{SUSP}}$ and $\overline{\text{SUSPA}}$ pins of the TI486. When $\overline{\text{SUSP}}$ is asserted the CPU completes any pending instructions and bus cycles and then enters suspend mode. Once in suspend mode, the $\overline{\text{SUSPA}}$ pin is asserted by the CPU.

A.8 Loading SMM Memory With an SMM Program from Main Memory

To load SMM memory with an SMI interrupt handler it is important that the SMI interrupt does not occur before the handler is ready to accept it. This can be done by not having SMAC = 0 and SMI = 1 (in the CCR1 register) before the SMI handler is installed. It is necessary to set SM4 = 1 and ARR4 with appropriate values before using the SMM memory. To load SMM memory with a program it is first necessary to enable SMM with the exception of the $\overline{\text{SMI}}$ pin by setting SMAC. (See Section A.3.) The SMM region is then mapped over main memory at the same location. This is done by the generation of $\overline{\text{SMADS}}$ for memory access for the SMI. A REP MOV instruction can then be used to transfer the program to the location. Then, turn off SMAC to activate potential SMIs.

```
SMI          = 1 shl 1
SMAC         = 1 shl 2
MMAC         = 1 shl 3
SM4          = 1 shl 7

      mov                al, 0c1h             ; index to CCr1
      out                22h, al              ; select CCR1 register
      in                 al, 23h              ; read current CCR1 value
      mov                ah,al                ;
      mov                al, 0c1h             ; index to CCR1
      out                22h, al              ; select CCR1 register
      mov                al, ah
      or                 al, SMI or SMAC;
      out                23h, al              ; write new value to CCR1
      mov                ax, SMI_SEGMENT
      mov                es, ax
      mov                edi, 0               ;start of the SMM area
      mov                esi, offset SMI_ROUTINE
      mov                ds,seg SMI_ROUTINE
      mov                ecx, (SMI_ROUTINE_LENGTH+3)/4
      rep                movs dword ptr es:[edi],dword ptr ds:[esi]
      mov                al,0c1h              ; index to CCR1
      out                22h, al              ; select CCR1 register
      in                 al, 23h              ; read current CCR1 value
      mov                ah,al
      mov                al, 0c1h             ; index to CCR1
      out                22h, al              ; select CCR1 register
      mov                al, ah
      and                al,NOT SMAC          ;disable SMAC, enable SMI#
      out                23h, al              ; write new value to CCR1
```

A.9 Detection of TI486 CPU

```
COMMENT ^
Name      detect.ASM

Purpose:  * Detect the presence of a TI486 micprocessor.
          * The undefined flags on a TI486 remain unchanged
            following a divide. The Intel part will modify some of the
            undefined flags. In this example the ZF flag should change.
          *       pseudocode
                     save flags before
                     load dividend and divisor
                     do unsigned divide
                     save flags after
                     cmp flags before with flags after
                     return a 1 if flags are unchanged
                               ( TI486 part detected )

Called by: Main();

Inputs:                none

Returns:  ax = 0 if Intel is detected
          ax = 1 if TI486 is detected
^

     DOSSEG                      ;select Intel-convention segment ordering
     .MODEL  SMALL               ;select small model (nearcode and data)
     .DATA                       ;TC-compatible initialized data segment

     flags_mask       dw 08D5H   ;mask to isolate the undefined bits
                                 ;masks all but OF,SF,ZF,AF,PF,CF flags
     flags_before     DW         ?                ; flags before div
     flags_after      DW         ?                ; flags after div
     dividend         DW         0FFFFh           ; dividend
     divisor          DW         4h               ; divisor
     result           DW         0                ; results of flags compare
                                                  ; 0=different (not TI486)
                                                  ; 1=same (TI486)

          .DATA?                 ;TC-compatible uninitialized data segment
          .CODE                  ;TC-compatible code segment

          PUBLIC  _TI486_detect

     _TI486_detect    PROC       ;function (near-callable in small model)

          .286

          push bp                ; "C" calling convention
          mov bp,sp

          pusha                  ; save processor state
          pushf
```

```
                                  ; set flags to a known value
          mov          ax,0
          cmp          ax,ax

          pushf                   ; load flags into ax
          pop          ax
          mov          ds:[word ptr flags_before],ax     ;save flags to mem

     ; do a div instruction so that the signature of the undefined flags
     ; can be observed
          mov          ax, dividend
          mov          dx, 0
          mov          bx, divisor
          div          bx

          pushf                                    ;load flags into ax
          pop          ax
          mov          flags_after, ax             ;save flags to mem

     ; recall flags_before and clear unwanted bits
          mov          ax, flags_mask
          and          ax, flags_before

     ; recall flags_after and clear unwanted bits
          mov          bx, flags_mask
          and          bx, flags_after

     ; compare the signature of the undefined bits before and after
          cmp          ax,bx
          jnz          Diff
          mov          result,1       ; set if flag bits are unchanged
                                      ; TI486 part found
          jmp  Done

Diff: mov  result,0                   ; clear if flag bits are changed
                                      ; TI486 part not found

Done:

     popf                             ;restore processor state
     popa

     mov  ax, result                  ; return value in ax

     pop  bp                          ; "C" calling convention
     ret

     _M5_detect        ENDP

     END                              ;end detect.ASM module
```

A.10 Detection of SMM Capable Version

At powerup/reset the EDX register will contain part type and stepping information.

EDX	Stepping	SMM Available
0410h	A	No
0411h	B	Yes

The following technique can be used to identify the stepping of a TI486 CPU after the reset information in EDX is lost. The method uses two functions: the mixed C and assembler function isb() and assembly language illegal opcode handler interrupt handler ill_op. The function isb() will return a 1 to indicate when a B step part is present, 0 otherwise. The function isb() installs an illegal opcode handler, ill_op. Then isb() sets up conditions to execute an SMM segment save instruction, SVDC. If an A step part is present the illegal opcode handler will be invoked. The ill_op process will then modify the return address on the stack to return to the instruction after the SVDC instruction. The storage location used by the SVDC instruction is then checked to see if it changed. If it has changed the part being tested is a B step part. This detection technique must be run at protection ring 0.

```
//****************************************************************************
//********************************* isb.c ***********************************
//****************************************************************************
#define TRUE 1
#defube FALSE 0

int old_off;
int old_seg;
extern ill_op();
//****************************************************************************
//             Function: isb ()
//             Returns:       1 if TI486 B step
//                   0 if TI486 A step
//****************************************************************************

isb ()
    {
    int i, b_step;
    char mem[10];

    for (i=0; i<10; mem[i++]=0;

    asm  {

         .386
         extrn _ill_op:near
```

```
;*********************************************
;****** get present illegal opcode handler
;*********************************************
push  es
push  bx
mov   ax, 3506h
int   21h
mov   old_seg, es
mov   old_off, bx
pop   bx
pop   es

;*********************************************
;****** install new illegal opcode handler
;*********************************************
push  dx
push  bx
push  ds
mov   ax, 2506h
mov   dx, OFFSET _ill_op
mov   bx, cs
mov   ds, bx
int   21h
pop   ds
pop   bx
pop   dx

char save_ccr1, save_cf, save_ce, save_cd;

;*************************************************************
;****** Set SM4 and SMAC and SMI bit to allow SMM instructions
;*************************************************************
mov   al, 0c1h
out   22h, al
in    al, 23h
mov   byte ptr [save_ccr1, al
or    al, 86h
mov   ah, al
mov   al, 0c1h
out   22h, al
mov   al, ah
out   23h, al

;*********************************************
;****** Setup non-zero SMM region
;*********************************************
mov   al, 0cfh
out   22h, al
in    al, 23h
mov   byte ptr [save_cf], al
mov   al, 0cfh
out   22h, al
mov   al, 1
out   23h, al
```

```
     ;*********************************************
     ;****** Set SMM region to the top of memory to
     ;****** avoid overlapping with this program
     ;*********************************************
     mov   al, 0cdh
     out   22h, al
     in    al, 23h
     mov   byte ptr [save_cd], al
     mov   al, 0ceh
     out   22h, al
     in    al, 23h
     mov   byte ptr [save_ce], al
     mov   al, 0cdh
     out   22h, al
     mov   al, 0ffh
     out   23h, al
     mov   al, 0ceh
     out   22h, al
     mov   al, 0h
     out   23h, al
     mov   al, 0cfh
     out   22h, al
     in    al, 23h
     and   al, 0fh
     out   23h, al

;****** flush prefetch after changing configuration
     jmp   $+2

     ;*********************************************
     ;****** Execute SMM instruction svdc
     ;*********************************************
     ;svdc word ptr mem, ds
           Word ptr mem == ss:[bx]
     lea   bx, mem
     db 36h 0fh 78h 1fh

     ;*********************************************
     ;****** restore configuration registers
     ;*********************************************
     mov   al, 0cdh
     out   22h, al
     mov   al, byte ptr save_cd
     out   23h, al
     mov   al, 0ceh
     out   22h, al
     mov   al, byte ptr save_ce
     out   23h, al
     mov   al, 0cfh
     out   22h, al
     mov   al byte ptr save_cf
     out   23h, al
     mov   al, 0c1h
     out   22h, al
     mov   al byte ptr save_ccr1
     out   23h, al
```

```
               ;*********************************************
               ;****** restore old illegal opcode handler
               ;*********************************************
               push  dx
               push  bx
               push  ds
               mov   ax, 2506h
               mov   dx, OFFSET old_off
               mov   bx, OFFSET old_seg
               mov   ds, bx
               int   21h
               pop   ds
               pop   bx
               pop   dx
         ) // isb asm region

    for (i=0, b_step=FALSE; i<10; ++i)
         if (mem[i] != 0)
               {
               b_step = TRUE;
               break;
               }

    return (b_step);
         } // isb ()

;********************** bad_op.asm ***********************
public _ill_op

assumecs:_TEXT

_TEXT segment byte public 'CODE'
_ill_op  proc near
         pop   ax
         add   ax, 5
         push  ax
         iret
_ill_op endp
_TEXT ends

end
```

A.11 SMM Feature Comparison

Feature	TI486	386SL	AMD
SMM Entry Point	Base of SMM Space	38000h	Reset Vector
CPU State Save Area	Top of SMM Space	3FFA8h–3FFFFh	60000h–600CAh and 60100h–60126h
SMM Space	Programmable (4K to 32M	38000/30000h (32K/64K)	Entire Address Space
Data Auto-Saved	8 32-bit registers 1 16-bit register 1 4-bit register	44 32-bit registers 9 16-bit registers	53 32-bit registers 8 16-bit registers
SMM Memory Restrictions	None	8-bit on 8 MHz XD Bus	Non-pipelined No dynamic bus-sizing
Normal Mode SMM Memory Access	Yes	Yes	No
Hardware Pins	2	NA – Must use 82360	4
Incremental CPU State Save Instructions	Yes	No	No
I/O Trapping	Yes	Yes	Yes
$\overline{\text{SMI}}$ Input Maskin g	Yes	Yes	No

A.12 SMM Instruction Macros – SMIMAC.INC

COMMENT ^

SMM Macro Implementation, smimac.inc (by Dean C. Wills)

This Section provides a complex set of macros that generate SMM opcodes containing the appropriate mod/rm bytes. For explicit SMM opcode definition, basic macros that define the opcode byte alone are provided with a '$' prefix to distinguish them from the other macros.

The complex macros require that the labels they access correspond to the segment specified or the macros will be inoperative. Segment overrides must by passed to the macro as an argument. If an address size override is used, a final argument of '1' must be passed to the macro. Segment and address size overrides must be presented explicitly to prevent the assembler from generating them automatically and breaking the macros. Examples of these macros are provided in the file smitest.asm.

```
^
COMMENT ^
Basic macros which allow you to create your own mod/rm bytes
^
cs_over MACRO
        db      2eh
        ENDM
$svdc   MACRO
        db      0fh,78h
        ENDM
$rsdc   MACRO
        db      0fh,79h
        ENDM
$svldt  MACRO
        db      0fh,7Ah
        ENDM
$rsldt  MACRO
        db      0fh,7Bh
        ENDM
$svts   MACRO
        db      0fh,7Ch
        ENDM
$rsts   MACRO
        db      0fh,7Dh
        ENDM
$rsm    MACRO
        db      0fh,AAh
        ENDM
```

```
COMMENT ^
Complex macros which gererate mod/rm automatically
^
svdc     MACRO    segover,addr,reg,adover
         domac    segover,addr,reg,adover,78h
         ENDM
rsdc     MACRO    segover,addr,adover
         domac    segover,addr,reg,adover,79h
         ENDM
svldt    MACRO    segover,addr,adover
         domac    segover,addr,es,adover,7ah
         ENDM
rsldt    MACRO    segover,addr,adover
         domac    segover,addr,es,adover,7bh
         ENDM
svts     MACRO    segover,addr,adover
         domac    segover,addr,es,adover,7ch
         ENDM
rsts     MACRO    segover,addr,adover
         domac    segover,addr,es,adover,7dh
         ENDM
rsm      MACRO
         db       ofh,0aah
         ENDM

COMMENT ^
Sub-Macro used by the above macro
^
domac    MACRO    segover,addr,reg,adover,op
         local    place1,place2,count
         count    = 0
         ifnb     <adover>
             count=count+1
         endif
         if       (count eq 0)
                  nop                     ;we're expanding the opcode one byte
         endif
         place1   = $
;pull off the proper prefix byte count
         mov      word ptr segover addr,reg
         org      place1+count
         mov      word ptr segover addr,reg
         place 2  = $
;patch the opcode
         org      place1+(count*2)-1
         db       0Fh,op
         org      place2
ENDM
```

```
COMMENT ^
Offset Definition for access into SMM space
^
SMI_SAVE  STRUC
        $EDIESI       DD          ?
        $RES3         DD          ?
        $RES2         DD          ?
        $BITS         DD          ?
        $CSDES        DQ          ?
        $CSSEL        DW          ?
        $RES1         DW          ?
        $NEXTIP       DD          ?
        $PREVIOUSIP   DD          ?
        $CR0          DD          ?
        $EFLAGS       DD          ?
        $DR7          DD          ?
SMI_SAVE ENDS

SMI_EDIESI            EQU ($EDIESI -         SIZE SMI_SAVE)
SMI_RES3              EQU ($RES3 -           SIZE SMI_SAVE)
SMI_RES2              EQU ($RES2 -           SIZE SMI_SAVE)
SMI_BITS              EQU ($BITS -           SIZE SMI_SAVE)
SMI_CSDES             EQU ($CSDES -          SIZE SMI_SAVE)
SMI_CSSEL             EQU ($CSSEL -          SIZE SMI_SAVE)
SMI_RES1              EQU ($RES1 -           SIZE SMI_SAVE)
SMI_NEXTIP            EQU ($NEXTIP -         SIZE SMI_SAVE)
SMI_PREVIOUSIP        EQU ($PREVIOUSIP -     SIZE SMI_SAVE)
SMI_CR0               EQU ($CR0 -            SIZE SMI_SAVE)
SMI_EFLAGS            EQU ($EFLAGS -         SIZE SMI_SAVE)
SMI_DR7               EQU ($DR7 -            SIZE SMI_SAVE)
```

SMM Instruction macro example: TEST.ASM

```
.MODEL  SMALL
.386
COMMENT ^
SMM Macro Examples

by Dean C. Wills

^

include smimac.inc

.DATA

public  hello, there ; so they'll be easy to find in map file.

there   db      10 dup (?)

.CODE

        svdc    cs:,hello,ds
        rsdc    ds,cs:,hello
        rsdc    gs,cs:,hello
        svdc    cs:,[eax+ebx*2+hello],ds,1
                      ;address size override here
        svdc    ,[ebx],fs,1
                      ;address size override
        svdc    ,there,gs
        svldt   cs:,hello
        rsldt   cs:,hello
        rsts    cs:,hello
        svts    cs:,[eax+ebx*2+hello],1;address size override here
        svldt   ,[ebx],1
                      ;address size override
        svts                       ,there
        hello                      db                         10 dup (?)
        align                      16  ;align so we'll create a more legible
                                       ;map file

end
```

A.13 TI486DLC/E and SMM

With respect to SMM programming, the TI486DLC/E with SMM differs from the TI486SLC/E in the following ways:

1) The SMM memory region size ranges from 4 KBytes to 4 GBytes.

2) The SMM memory base location can be from 0 Bytes to 4 GBytes less 4 KBytes (FFFF EFFFh).

3) Address region 4 is eight bits wider to support a 4 GByte physical address space by adding address lines 24 to 31. The additional lines are indexed by 0CDh.

A.14 Format of Data Used by SVDC/RSDC Instructions

The SVDC/REDC instructions should be used to change limits and r/w priveleges of segment registers before they are used by SMM code. The instructions use a 10 byte area that is comprised of two major portions of the segment register value/contents and the non-programmable visible internal descriptor that has the following format:

```
|Segment Register Descripton <8 bytes>|Segment Register Selector <2 bytes>|
```

1) Segment Register Selector: This is the segment if the segment register was loaded in real mode or the selector if the segment register was loaded in protected mode. In real mode, this is also equal to the Segment base divided by 10h and clipped to 16 bits.

```
        dw      |Selector or Segment |
```

2) Segment Register Descriptor, which is the actual descriptor if the segment was loaded in protected mode, or a psuedo-descriptor if the segment register was loaded in real mode.

```
        dw      | Limit [15:0] |
        dw      | Base [15:0] |
        db      | Base [23:16] |
        db      | P | DPL | 1 | E | DscTy[2:0] | A |
        db      |G | D | r | AVL | Limit [19:16] |
        db      | Base [31:24] |

        Limit             Max size
        Base              Starting Address
        A                 Segment Accessed Flag
        DscTy                 E == 1: Executable,      E == 0: Data,
                              C|R                      ED|W
                              C == 1: Conforming       ED == 1: Expand Down
                              R == 1: Readable         W == 1: Writable

        DPL               Protection Level
        P                 1 Segment present, 0 not present
        AVL
        D                 0 16 bit address and operand size
        G                 0 byte, 1 page granular
```

Example:

```
;Load SS descriptor (non-programmer-visible region) values appropriate to
REAL mode.

INCLUDE smimac.inc

        old_val         dt        ?           ; location to store old ss
value
        real_mode:dw    0ffffh                ; limit
                        dw        0           ; base
                        db        0           ; base
                        db        10010011B   ; 92h, data segment
                        db        0           ; G=0, D=0, upper limit=0
                        db        0           ; high portion of base
                        dw        0           ; selector/segment

        svdc            cs:,[old_val], ss
        rsdc            ss, cs:,[real_mode]
        mov             ax, cs
        mov             ds, ax
```

A.15 Altering SMM Code Limits

Since it is not possible to use the rsdc instruction to modify the non-programmer-visible portion of the CS information, a switch into protected mode becomes necessary and is demonstrated here.

```
SMMBASE = 15000H

.386P
        jmp        $skip
gdt     dq         0                    ;null
G_4gig= $-gdt
        dw         0ffffh               ;limit
        dw         0                    ;linear low
        db         0                    ;linear high
        db         12h or (0 shl 5) or 80h:read/write, p10 present
        db         0fh or 80h           ;G=1, high limit = 0Fh
        db         0                    ;extra high (0c0h for EMC chip)
g_code     = $-gdt
        dw         0ffffh               ;low limit
        dw         0                    ;base
        db         0                    ;base
        db         1ah or (0 shl 5) or 80h;
        db         8fh                  ;4gig limit
        db         0                    ;base
GDTSIZE = ($-gdt)
gdtinit DW         GDTSIZE-1
        DD         ?                    ;base
$skip:
        mov        eax, SMMBASE
        mov        word ptr cs:[gdt+g_code+2],ax;low base
        shr        eax,16
        mov        byte ptr cl:[gdt+g_code+4],al
        mov        byte ptr cs:[gdt+g_code+7],ah;base
        mov        eax,offset gdt
        add        eeax,SMMBASE
        mov        dword ptr cs:[gdtinit+2],eax;
        db         66h
        lgdt       cs: fword ptr [gdtinit];load gdt
        mov        eax, cr0             ;get death register
        mov        ebx, eax             ;save in ebx
        or         eax, 1               ;turn on protected mode
        mov        cr0, eax         ;go to protected mode
        db         0eah             ;load new descriptor with far jump
        dw         offset pmode
        dw         g_code
pmode:
        mov        cr0,ebx              ;back to real mode
        db         0eah
        dw         offset pmode2
        dw         SMMBASE / 10h    ;we could patch this run time if desired
pmode2:
```

```
COMMENT ^
now we are back to real mode with the limits set as desired
^

;        <<user smi code>>
;        rsm

         db        0fh
         db        0aah
```

Setting other registers to the value of the CS register during the SMI

```
COMMENT ^
load DS register with the same value as CS register. If our base is beyond 1
MByte, we can't rely on the CS selector to be accurate so we need to use svdc
and rsdc. svdc may be used on the CS segment to determine the base and limit.
We need to set the segment type ourselves.
```

A.16 SMM Errata

The following condition is known to exist:

If the CPU is in V86 mode and is interrupted by an SMI, the VM bit in the EFLAGS register is not cleared as it should be during real mode operation. Not clearing this bit can cause protection errors of valid instructions that are being executed in the SMI handler. This can be resolved by adding the following code after saving all used registers:

```
rsdc    ss, cs:, [gdt+g_big]          ; change ss limit to 4 GBytes
mov     esp, offset smistack          ; create new stack pointer
mov     ax, cs
mov     ss, ax                        ; new stack segment
mov     eax, 0
push    eax                           ; flags after iretd
mov     eax, cs
push    eax                           ; segment after iretd
mov     eax, offset @F
push    eax                           ; offset after iretd
iretd
     @@:
```

Note: See Section A.6, debugging example, for usage of above code.

Appendix B

TI486 Cache Flush

B.1 General Cache Invalidation

When the FLUSH bit in CCR0 is set, the $\overline{\text{FLUSH}}$ input invalidates the entire contents of the TI486 internal cache when asserted low. This may be used to assure that data stored in the TI486 internal cache does not differ from data stored in system memory. Additionally, the cache may be invalidated by execution of the 486-compatible invalidate instructions (INVD,WBINVD) or in response to a Hold Acknowledge state if the BARB bit in CCR0 is set. The method chosen for invalidating the TI486 internal cache may be different, depending on whether or not the system has a serial secondary cache. Invalidation methods are described for systems with and without a serial secondary cache.

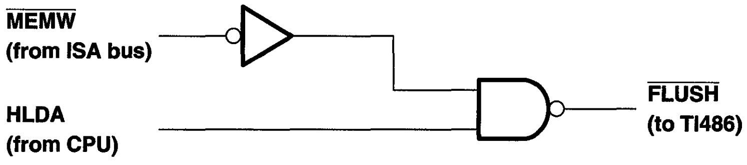
B.1.1 Cache Invalidation for Systems With No Secondary Cache or a Parallel Secondary Cache

When the only cache memory in the system is the TI486 internal cache, or when the secondary cache has a parallel (or look-aside) architecture, there are two general methods of invalidating the cache and maintaining cache coherency.

Method 1 Invalidate the TI486 every time the CPU enters a HOLD state. By setting the BARB bit in CCR0, automatic cache flush occurs when the TI486 is placed in a HOLD state. If the chip set does not support hidden refresh, this may lead to very frequent cache invalidation, since it will put the CPU in hold during DRAM refresh cycles, which occur approximately every 15 μs. If the chip set supports hidden refresh, this may be an acceptable solution, since the cache will only be invalidated during DMA or bus master reads from or writes to memory.

Method 2 Invalidate the TI486 internal cache when a DMA or bus master writes to system memory. This requires external hardware to drive the TI486 $\overline{\text{FLUSH}}$ input when DMA or bus masters are detected writing to system memory. This can be done with the simple circuit shown in Figure B–1. The circuit will generate an active-low $\overline{\text{FLUSH}}$ to the CPU every time a HOLD state is entered (defined by HLDA = 1) and memory write occurs (defined by $\overline{\text{MEMW}}$ = 0).

Figure B–1. $\overline{\text{FLUSH}}$ Logic

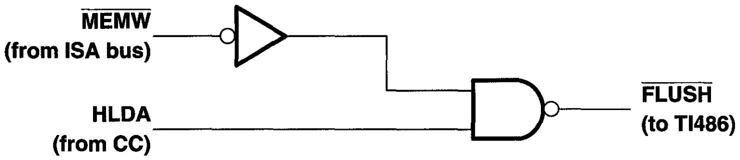


B.1.2 Cache Invalidation for Systems With A Serial Secondary Cache

In a system with a serial (or look-through) secondary cache, flushing the cache cannot be accomplished by setting the BARB bit in CCR0 because bus arbitration occurs between the serial cache controller and the system. This allows the CPU to continue executing out of cache.

The secondary cache controller arbitrates the bus between itself and DMA controllers or bus masters and asserts HLDA to the chip set when the bus has been granted. Each time a DMA or bus master write is detected, the $\overline{\text{FLUSH}}$ pin on the TI486 must be asserted. The circuit shown in Figure B–2 may be used. Note that the HLDA signal is now generated by the secondary cache controller rather than the CPU. This is the preferred solution since in many cases with secondary serial caches, the CPU is not put in HOLD so that it can continue execution from cache while DMA or bus master activity is occurring on the system bus.

Figure B–2. $\overline{\text{FLUSH}}$ Logic



Appendix C

TI486 BIOS Modification Guide

C.1 Introduction

In order to reap full benefit from the TI486 microprocessors, the system BIOS should be modified to support the internal registers that control the on-chip cache and other extra features. This Appendix serves as a guide to some of the changes that need to be considered, and includes sample assembler code for controlling the cache.

There are three main areas of consideration that will be discussed in relation to the internal cache registers:

- Power-on and hard reset
- Protected-mode to real-mode switching
- Soft reset/CTRL-ALT-DEL

In each case, the state of the CPU cache registers and when and how to change their values must be known.

C.1.1 Power-On and Hard Reset

In these two cases, the system will be booted into the operating system. Due to the reset line to the CPU going active, the internal cache will be disabled, making the CPU act much like a 386. At some point the cache must be turned on before the OS is booted. A convenient time to turn on the cache may be during final chip set initialization, understanding that the cache should remain off during memory sizing. Many BIOSs provide the user an option to disable the system cache using the setup screen. As most user cache control options are stored in non-volatile RAM, the flag responses, and potentially other flags, should be checked before turning the cache on.

C.1.2 Protected-Mode to Real-Mode Switching

Protected-mode to real-mode switching can be implemented to handle cases where the OS has been booted, applications have been running, and the CPU needs to be reset to switch from protected mode to real mode. The objective is to switch CPU modes and jump back into the OS or application at some saved return address. Because the CPU was reset, the internal cache wll have been disabled. Before returning control to the application the cache should be turned back on, but only if it was on before the reset occurred, This is accomplished by checking the cache enable flag in the non-volatile RAM, to see if the user enabled caching from the setup screen. However, if the BIOS allows the user to turn off the cache by a hot-key combination (perhaps as part of speed switching), other checks may need to be performed to see if the cache should be turned back on.

C.1.3 Soft Reset/CTRL-ALT-DEL

The objective of a soft reset is to reset the system and reboot the OS, similar to power-on and hard-reset, but a hard reset of the CPU is not generated. Thus, the CPU's internal cache is not disabled (if it was on). This can have a negative impact on memory sizing code, such as generating memory size mismatch errors. In this situation, disable the internal cache and enable it prior to booting, if enabled by the user in setup.

C.1.4 Turning On and Off the Internal Cache

When the TI486 internal cache is turned on or off, the following guidelines should be observed:

1) Turn off interrupts — CLI

2) Turn off cache using CR0 bit 30 and flush using WBINVD

3) Manipulate cache registers

4) Turn on cache and flush using WBINVD

5) Turn on interrupts — STI

The above sequence ensures that the process is not interrupted until complete and that no cache coherency issues arise when the cach is turned back on. When manipulating the cache registers it is a good idea to explicitly set each register instead of relying on default values.

Some example assembler code for turning the cache off follows:

```
CacheOut    MACRO             index, value
            MOV               AL, index
            OUT               22h, AL
            MOV               AL, value
            OUT               23h, AL
ENDM

CLI
MOV         EAX, CR0
OR          EAX, 40000000h        ; set bit 30, turn off cache
MOV         CR0, EAX
WBINVD                            ; for external cache coherency

CacheOut    0C0h,     00h
CacheOut    0C1h,     00h

CacheOut    0C4h,     00h
CacheOut    0C5h,     00h
CacheOut    0C6h,     0Fh

CacheOut    0C7h,     00h
CacheOut    0C8h,     00h
CacheOut    0C9h,     00h

CacheOut    0CAh,     00h
CacheOut    0CBh,     00h
CacheOut    0CCh,     00h

CacheOut    0CDh,     00h
CacheOut    0CEh,     00h
CacheOut    0CFh,     00h

WBINVD
MOV         EAX, CR0
AND         EAX, 0BFFFFFFFh
MOV         CR0, EAX
STI
```

Turning on the TI486 cache can be done by modifying some of the register values as shown below. The CacheOut macro definition remains the same:

```
CLI
MOV         EAX, CR0
OR          EAX, 40000000h          ; set bit 30, turn off cache
MOV         CR0, EAX
WBINVD                              ; for external cache coherency

CacheOut    0C0h,     01h           ; set NC0 bit
CacheOut    OC1h,     00h

CacheOut    0C4h,     00h
CacheOut    0C5h,     0Ah           ; non-cache region at A0000
CacheOut    0C6h,     06h           ; that is 128K in size

CacheOut    0C7h,     00h
CacheOut    0C8h,     0Ch           ; non-cache region at C0000
CacheOut    0C9h,     07h           ; that is 256K in size

CacheOut    0CAh,     00h
CacheOut    0CBh,     00h
CacheOut    0CCh,     00h

CacheOut    0CDh,     00h
CacheOut    0CEh,     00h
CacheOut    0CFh,     00h

WBINVD
MOV         EAX, CR0
AND         EAX, 0BFFFFFFFh
MOV         CR0, EAX
STI
```

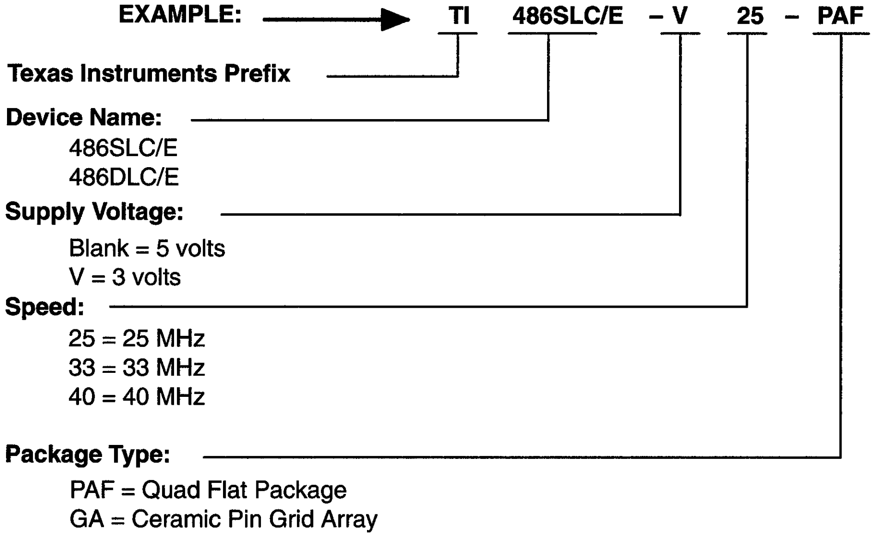
Appendix D

Ordering Information

D.1 Ordering Information

D.1.1 Part Number Components

Components of the TI486 processor part number are diagrammed in the following example.



EXAMPLE: → **TI 486SLC/E – V 25 – PAF**

Texas Instruments Prefix

Device Name:
486SLC/E
486DLC/E

Supply Voltage:
Blank = 5 volts
V = 3 volts

Speed:
25 = 25 MHz
33 = 33 MHz
40 = 40 MHz

Package Type:
PAF = Quad Flat Package
GA = Ceramic Pin Grid Array

D.1.2 Part Numbers for TI486 Processors Offered

The following table lists the complete part number for each version of the TI486 processor offered and provides a short description consisting of the supply voltage, performance capabilities, and the mechanical package offered for each.

TI486SLC/E/DLC/E Part Numbers

PART NUMBER	DESCRIPTION
TI486SLC/E-25-PAF	5 V, 25 MHZ, QFP Package
TI486SLC/E-33-PAF	5 V, 33 MHZ, QFP Package
TI486SLC/E-V25-PAF	3.3 V, 25 MHZ, QFP Package
TI486DLC/E-33-GA	5 V, 33 MHz, PGA Package
TI486DLC/E-40-GA	5 V, 40 MHz, PGA Package
TI486DLC/E-V25-GA	3.3 V, 25 MHZ, PGA Package
TI486DLC/E-V33-GA	3.3 V, 33 MHZ, PGA Package

TEXAS INSTRUMENTS

Printed in U.S.A.