Embedded Intel486™ Processor Hardware Reference Manual

Release Date: July 1997
Order Number: 273025-001

The embedded Intel486™ processors may contain design defects known as errata which may cause the products to deviate from published specifications. Currently characterized errata are available on request.

CHAPTER 3
INTERNAL ARCHITECTURE

CHAPTER 4
BUS OPERATION

CHAPTER 5
MEMORY SUBSYSTEM DESIGN

CHAPTER 6
CACHE SUBSYSTEM

CHAPTER 7
PERIPHERAL SUBSYSTEM

CHAPTER 10
PHYSICAL DESIGN AND SYSTEM DEBUGGING

FIGURES

FIGURES

FIGURES

FIGURES

TABLES

1

GUIDE TO THIS MANUAL

Chapter Contents

CHAPTER 1
GUIDE TO THIS MANUAL

This manual describes the embedded Intel486™ processors. It is intended for use by hardware designers familiar with the principles of embedded microprocessors and with the Intel486 processor architecture.

1.1 MANUAL CONTENTS

This manual contains 10 chapters and an index. This section summarizes the contents of the remaining chapters. The remainder of this chapter describes conventions and special terminology used throughout the manual and provides references to related documentation.

Chapter 2: "Introduction"	This chapter provides an overview of the current embedded Intel486 processor family, including product features, system components, system architecture, and applications. This chapter also lists product frequency, voltage and package offerings.
Chapter 3: "Internal Architecture"	This chapter describes the Intel486 processor internal architecture, with a description of the processor's functional units.
Chapter 4: "Bus Operation"	This chapter describes the features of the processor bus, including bus cycle handling, interrupt and reset signals, cache control, and floating-point error control.
Chapter 5: "Memory Subsystem Design"	This chapter designing a memory subsystem that supports features of the Intel486 processor such as burst cycles and cache. This chapter also discusses using write-posting and interleaving to reduce bus cycle latency.
Chapter 6: "Cache Subsystem"	This chapter discusses cache theory and the impact of caches on performance. This chapter details different cache configurations, including direct-mapped, set associative, and fully associative. In addition, write-back and write-through methods for updating main memory are described.

Chapter 7: "Peripheral Subsystem"	This chapter describes the connection of peripheral devices to the Intel486 processor bus. Design techniques are discussed for interfacing a variety of devices, including a LAN controller and an interrupt controller.
Chapter 8: "System Bus Design"	This chapter provides an overview of system bus design considerations, including implementing of the EISA and PCI system buses.
Chapter 9: "Performance Considerations"	This chapter focuses on the system parameters that affect performance. External (L2) caches are also examined as a means of improving memory system performance.
Chapter 10: "Physical Design and System Debugging"	The higher clock speeds of Intel486 processor systems require design guidelines. This chapter outlines basic design considerations, including power and ground, thermal environment, and system debugging issues.

1.2 TEXT CONVENTIONS

The following notations are used throughout this manual.

#	The pound symbol (#) appended to a signal name indicates that the signal is active low.
Variables	Variables are shown in italics. Variables must be replaced with correct values.
New Terms	New terms are shown in italics. See the Glossary for a brief definition of commonly used terms.
Instructions	Instruction mnemonics are shown in uppercase. When you are programming, instructions are not case-sensitive. You may use either upper- or lowercase.
Numbers	Hexadecimal numbers are represented by a string of hexadecimal digits followed by the character *H*. A zero prefix is added to numbers that begin with *A* through *F*. (For example, *FF* is shown as *0FFH*.) Decimal and binary numbers are represented by their customary notations. (That is, 255 is a decimal number and 1111 1111 is a binary number. In some cases, the letter *B* is added for clarity.)
Units of Measure	The following abbreviations are used to represent units of measure:

A	amps, amperes
Gbyte	gigabytes
Kbyte	kilobytes
KΩ	kilo-ohms
mA	milliamps, milliamperes
Mbyte	megabytes
MHz	megahertz
ms	milliseconds
mW	milliwatts
ns	nanoseconds
pF	picofarads
W	watts
V	volts
µA	microamps, microamperes
µF	microfarads
µs	microseconds
µW	microwatts

Register Bits When the text refers to more that one bit, the range of bits is represented by the highest and lowest numbered bits, separated by a long dash (example: A15–A8). The first bit shown (15 in the example) is the most-significant bit and the second bit shown (8) is the least-significant bit.

Register Names Register names are shown in uppercase. If a register name contains a lowercase italic character, it represents more than one register. For example, P*n*CFG represents three registers: P1CFG, P2CFG, and P3CFG.

Signal Names Signal names are shown in uppercase. When several signals share a common name, an individual signal is represented by the signal name followed by a number, while the group is represented by the signal name followed by a variable (*n*). For example, the lower chip-select signals are named CS0#, CS1#, CS2#, and so on; they are collectively called CS*n*#. A pound symbol (#) appended to a signal name identifies an active-low signal. Port pins are represented by the port abbreviation, a period, and the pin number (e.g., P1.0, P1.1).

1.3 SPECIAL TERMINOLOGY

The following terms have special meanings in this manual.

Assert and Deassert The terms *assert* and *deassert* refer to the acts of making a signal active and inactive, respectively. The active polarity (high/low) is defined by the signal name. Active-low signals are designated by the pound symbol (#) suffix; active-high signals have no suffix. To assert RD# is to drive it low; to assert HOLD is to drive it high; to deassert RD# is to drive it high; to deassert HOLD is to drive it low.

DOS I/O Address Peripherals that are compatible with PC/AT system architecture can be mapped into DOS (or PC/AT) addresses 0H–03FFH. In this manual, the terms *DOS address* and *PC/AT address* are synonymous.

Expanded I/O Address All peripheral registers reside at I/O addresses 0F000H–0FFFFH. PC/AT-compatible integrated peripherals can also be mapped into DOS (or PC/AT) address space (0H–03FFH).

PC/AT Address Integrated peripherals that are compatible with PC/AT system architecture can be mapped into PC/AT (or DOS) addresses 0H–03FFH. In this manual, the terms *DOS address* and *PC/AT address are synonymous.*

Set and Clear The terms *set* and *clear* refer to the value of a bit or the act of giving it a value. If a bit is *set*, its value is “1”; *setting* a bit gives it a “1” value. If a bit is *clear*, its value is “0”; *clearing* a bit gives it a “0” value.

1.4 ELECTRONIC SUPPORT SYSTEMS

Intel's FaxBack* service provides up-to-date technical information. Intel also offers a variety of information on the World Wide Web. These systems are available 24 hours a day, 7 days a week, providing technical information whenever you need it.

1.4.1 FaxBack Service

FaxBack is an on-demand publishing system that sends documents to your fax machine. You can get product announcements, change notifications, product literature, device characteristics, design recommendations, and quality and reliability information from FaxBack 24 hours a day, 7 days a week.

1-800-525-3019 (US or Canada)

+44-1793-496646 (Europe)

+65-256-5350 (Singapore)

+852-2-844-4448 (Hong Kong)

+886-2-514-0815 (Taiwan)

+822-767-2594 (Korea)

+61-2-975-3922 (Australia)

1-503-264-6835 (Worldwide)

Think of the FaxBack service as a library of technical documents that you can access with your phone. Just dial the telephone number and respond to the system prompts. After you select a document, the system sends a copy to your fax machine.

1.4.2 World Wide Web

Intel offers a variety of information through the World Wide Web (http://www.intel.com/).

1.5 TECHNICAL SUPPORT

In the U.S. and Canada, technical support representatives are available to answer your questions between 5 a.m. and 5 p.m. PST. You can also fax your questions to us. (Please include your voice telephone number and indicate whether you prefer a response by phone or by fax). Outside the U.S. and Canada, please contact your local distributor.

1-800-628-8686	U.S. and Canada
916-356-7599	U.S. and Canada
916-356-6100 (fax)	U.S. and Canada

1.6 PRODUCT LITERATURE

You can order product literature from the following Intel literature centers.

1-800-548-4725	U.S. and Canada
708-296-9333	U.S. (from overseas)
44(0)1793-431155	Europe (U.K.)
44(0)1793-421333	Germany
44(0)1793-421777	France
81(0)120-47-88-32	Japan (fax only)

1.6.1 Related Documents

The following Intel documents contain additional information on designing systems that incorporate the Intel486 processors.

Intel Document Name	Intel Order Number
Datasheets	
Embedded Intel486™ SX Processor datasheet	272769-001
Embedded IntelDX2™ Processor datasheet	272770-001
Embedded Ultra-Low Power Intel486™ SX Processor datasheet	272731-001
Embedded Ultra Low-Power Intel486™ GX Processor datasheet	272755-001
Embedded Write-Back Enhanced IntelDX4™ Processor datasheet	272771-001
MultiProcessor Specification	242016-005
Manuals	
Intel Architecture Software Developer's Manual, Volumes 1 and 2	243190-001 243191-001
Embedded Intel486™ Processor Family Developer's Manual	273021.001
Ultra-Low Power Intel486™ SX Processor Evaluation Board Manual	272815-001
Intel486™ Processor Family Programmer's Reference Manual	240486-003
Application Notes/Performance Briefs	
AP-505–Picking Up the Pace: Designing the IntelDX4™ Processor into Intel486™ Processor-Based Designs	242034-001
Intel486™ Microprocessor Performance Brief	241254-002
IntelDX4™ Processor Performance Brief	242446-001

You can obtain the following resources from the Word Wide Web at the sites listed.

Document Name	Web Site
Standard 1149.1—1990, IEEE Standard Test Access Port and Boundary-Scan Architecture and its supplement, *Standard 1149.1a—1993*	Contact the IEEE at http://www.ieee.org.
PCI Local Bus Specification, Revisions 2.0 and 2.1	Contact the PCI Special Interest Group at http://www.pcisig.com

2

Introduction

Chapter Contents

CHAPTER 2
INTRODUCTION

The Intel486™ processor family enables a range of low-cost, high-performance embedded system designs capable of running the entire installed base of DOS*, Windows*, OS/2*, and UNIX* applications written for the Intel architecture. This family includes the following processors:

- The **IntelDX4™ processor** is the fastest Intel486 processor (up to 50% faster than an IntelDX2™ processor). The IntelDX4 processor integrates a 16-Kbyte unified cache and floating-point hardware on-chip for improved performance.

  The IntelDX4 processor is also available with a write-back on-chip cache for improved entry-level performance.

- The **IntelDX2™ processor** integrates an 8-Kbyte unified cache and floating-point hardware on-chip.

  The IntelDX4 and IntelDX2 processors use Intel's speed-multiplying technology, allowing the processor core to operate at frequencies higher than the external memory bus.

- **The Intel486 SX** processor offers the features of the IntelDX2 processor without floating-point hardware and clock multiplying.

- The **Ultra-Low Power Ultra-Low Power Intel486 SX** and **Ultra-Low Power Intel486 GX** processors provide additional power-saving features for use in battery-operated and hand-held embedded designs. The Ultra-Low Power Intel486 SX processor, like the other Intel486 processors, supports dynamic data bus sizing for 8-, 16-, or 32-bit bus sizes, whereas the Ultra-Low Power Intel486 GX processor has a 16-bit external data bus.

The entire Intel486 processor family incorporates energy efficient "SL Technology" for mobile and fixed embedded computing. SL Technology enables system designs that exceed the Environmental Protection Agency's (EPA) Energy Star program guidelines without compromising performance. It also increases system design flexibility and improves battery life in all Intel486 processor-based hand-held applications. SL Technology allows system designers to differentiate their power management schemes with a variety of energy efficient, battery life enhancing features.

Intel486 processors provide power management features that are transparent to application and operating system software. Stop Clock, Auto HALT Power Down, and Auto Idle Power Down allow software-transparent control over processor power management.

Equally important is the capability of the processor to manage system power consumption. Intel486 processor System Management Mode (SMM) incorporates a non-maskable System Management Interrupt (SMI#), a corresponding Resume (RSM) instruction and a new memory space for system management code. Although transparent to any application or operating system, Intel's SMM ensures seamless power control of the processor core, system logic, main memory, and one or more peripheral devices.

Intel486 processors are available in a full range of speeds (16 MHz to 100 MHz), packages (PGA, SQFP, PQFP, TQFP), and voltages (5 V, 3.3 V, 3.0 V and 2.0 V) to meet many system design requirements.

2.1 PROCESSOR FEATURES

All Intel486 processors consist of a 32-bit integer processing unit, an on-chip cache, and a memory management unit. These ensure full binary compatibility with the 8086, 8088, 80186, 80286, Intel386™ SX, and Intel386 DX processors, and with all versions of Intel486 processors. All Intel486 processors offer the following features:

- *32-bit RISC integer core* — The Intel486 processor performs a complete set of arithmetic and logical operations on 8-, 16-, and 32-bit data types using a full-width ALU and eight general purpose registers.
- *Single Cycle Execution* — Many instructions execute in a single clock cycle.
- *Instruction Pipelining* — The fetching, decoding, address translation, and execution of instructions are overlapped within the Intel486 processor.
- *On-Chip Floating-Point Unit* — The IntelDX2 and Intel DX4 processors support the 32-, 64-, and 80-bit formats specified in IEEE standard 754. The unit is binary compatible with the 8087, Intel287, and Intel387 coprocessors, and with the Intel OverDrive® processor.
- *On-Chip Cache with Cache Consistency Support* — An 8-Kbyte (16-Kbyte on the IntelDX4 processor) internal cache is used for both data and instructions. Cache hits provide zero wait state access times for data within the cache. Bus activity is tracked to detect alterations in the memory represented by the internal cache. The internal cache can be invalidated or flushed so that an external cache controller can maintain cache consistency.
- *External Cache Control* — Write-back and flush controls for an external cache are provided so the processor can maintain cache consistency.
- *On-Chip Memory Management Unit* — Address management and memory space protection mechanisms maintain the integrity of memory in a multi-tasking and virtual memory environment. The memory management unit supports both segmentation and paging.
- *Burst Cycles* — Burst transfers allow a new doubleword to be read from memory on each bus clock cycle. This capability is especially useful for instruction prefetch and for filling the internal cache.
- *Write Buffers* — The processor contains four write buffers to enhance the performance of consecutive writes to memory. The processor can continue internal operations after a write to these buffers, without waiting for the write to be completed on the external bus.
- *Bus Backoff* — If another bus master needs control of the bus during a processor-initiated bus cycle, the Intel486 processor floats its bus signals, then restarts the cycle when the bus becomes available again.
- *Instruction Restart* — Programs can continue execution following an exception that is generated by an unsuccessful attempt to access memory. This feature is important for supporting demand-paged virtual memory applications.

- *Dynamic Bus Sizing* — External controllers can dynamically alter the effective width of the data bus. Bus widths of 8, 16, or 32 bits can be used (the 8-bit and 32-bit bus widths are not available on the Ultra-Low Power Intel486 GX processor).
- *Boundary Scan (JTAG)* — Boundary Scan provides in-circuit testing of components on printed circuit boards. The Intel Boundary Scan implementation conforms with the IEEE Standard Test Access Port and Boundary Scan Architecture.

SL Technology provides the following features:

- *Intel System Management Mode* — A unique Intel architecture operating mode provides a dedicated special purpose interrupt and address space that can be used to implement intelligent power management and other enhanced functions in a manner that is completely transparent to the operating system and applications software.
- *I/O Restart* — An I/O instruction interrupted by a System Management Interrupt (SMI#) can automatically be restarted following the execution of the RSM instruction.
- *Stop Clock* — The Intel486 processor has a stop clock control mechanism that provides two low-power states: a “fast wake-up” Stop Grant state and a “slow wake-up” Stop Clock state with CLK frequency at 0 MHz.
- *Auto HALT Power Down* — After the execution of a HALT instruction, the Intel486 processor issues a normal Halt bus cycle and the clock input to the Intel486 processor core is automatically stopped, causing the processor to enter the Auto HALT Power Down state.
- *Upgrade Power Down Mode* — When an Intel486 processor upgrade is installed, the Upgrade Power Down Mode detects the presence of the upgrade, powers down the core, and three-states all outputs of the original processor, so the Intel486 processor enters a very low current mode.
- *Auto Idle Power Down* — This function allows the processor to reduce the core frequency to the bus frequency when both the core and bus are idle. Auto Idle Power Down is software-transparent and does not affect processor performance. Auto Idle Power Down provides an average power savings of 10% and is only applicable to clock-multiplied processors.

Enhanced Bus Mode Features (for the Write-Back Enhanced IntelDX4 processor only):

- *Write Back Internal Cache* — The Write-Back Enhanced IntelDX4 processor adds write-back support to the unified cache. The on-chip cache is configurable to be write-back or write-through on a line-by-line basis. The internal cache implements a modified MESI protocol, which is most applicable to single processor systems.
- *Enhanced Bus Mode* — The definitions of some signals have been changed to support the new Enhanced Bus Mode (Write-Back Mode).
- *Write Bursting* — Data written from the processor to memory can be burst to provide zero wait state transfers.

2.2 Intel486™ PROCESSOR PRODUCT FAMILY

Table 2-1 shows the Intel486 processors available by clock mode, supply voltage, maximum frequency, and package. An individual product has either a 5 V supply voltage or a 3.3 V supply voltage, but not both. Likewise, an individual product may have 1x, 2x, or 3x clock. Please contact Intel for the latest product availability and specifications.

Table 2-1. Product Options

Intel486™ Processor	V_{CC}		Processor Frequency (MHz)								168-Pin PGA	208-Lead SQFP	196-Lead PQFP	176-Lead TQFP
	V_{CCP}	16	20	25	33	40	50	66	75	100				
1x Clock														
Intel486 SX Processor	3.3 V			✓	✓							✓		
	5 V			✓	✓						✓		✓	
Ultra-Low Power Intel486 SX Processor	2.4-3.3			✓										✓
	2.7-3.3				✓									✓
Ultra-Low Power Intel486 GX Processor	2.0-3.3	✓												✓
	2.2-3.3		✓											✓
	2.4-3.3			✓										✓
	2.7-3.3				✓									✓
2x Clock														
IntelDX2™ Processor	3.3						✓					✓		
	5						✓	✓			✓			
3x Clock														
Write-Back Enhanced IntelDX4™ Processor	3.3								✓	✓	✓	✓		

2.2.1 Operating Modes and Compatibility

The Intel486 processor can run in modes that give it object-code compatibility with software written for the 8086, 80286, and Intel386 processor families. The operating mode is set in software as one of the following:

- Real Mode: When the processor is powered up or reset, it is initialized in Real Mode. This mode has the same base architecture as the 8086 processor but allows access to the 32-bit register set of the Intel486 processor. The address mechanism, maximum memory size (1 Mbyte), and interrupt handling are identical to the Real Mode of the 80286 processor. Nearly all Intel486 processor instructions are available, but the default operand size is 16 bits; in order to use the 32-bit registers and addressing modes, override instruction prefixes must be used. The primary purpose of Real Mode is to set up the processor for Protected Mode operation.
- Protected Mode (also called Protected Virtual Address Mode): The complete capabilities of the Intel486 processor become available when programs are run in Protected Mode. In addition to segmentation protection, paging can be used in Protected Mode. The linear address space is four gigabytes and virtual memory programs of up to 64 terabytes can be run. All existing 8086, 80286, and Intel386 processor software can be run under the Intel486 processor's hardware-assisted protection mechanism. The addressing mechanism is more sophisticated in Protected Mode than in Real Mode.
- Virtual 8086 Mode, a sub-mode of Protected Mode, allows 8086 programs to be run with the segmentation and paging protection mechanisms of Protected Mode. This mode offers more flexibility than the Real Mode for running 8086 programs. Using this mode, the Intel486 processor can execute 8086 operating systems and applications simultaneously with an Intel486 operating system and both 80286 and Intel486 processor applications.

The hardware offers additional modes, which are described in greater detail in the *Embedded Intel486™ Processor Family Developer's Manual*.

2.2.2 Memory Management

The memory management unit supports both segmentation and paging. Segmentation provides several independent, protected address spaces. This security feature limits the damage a program error can cause. For example, a program's stack space should be prevented from growing into its code space. The segmentation unit maps the separate address spaces seen by programmers into one unsegmented, linear address space.

Paging provides access to data structures larger than the available memory space by keeping them partly in memory and partly on disk. Paging breaks the linear address space into units of 4 Kbytes called pages. When a program makes its first reference to a page, the program can be stopped, the new page copied from disk, and the program restarted. Programs tend to use only a few pages at a time, so a processor with paging can simulate a large address space in RAM using a small amount of RAM plus storage on a disk.

2.2.3 On-chip Cache

A software-transparent 8-Kbyte cache (16-Kbyte on the IntelDX4 processor) stores recently accessed information on the processor. Both instructions and data can be cached. If the processor needs to read data that is available in the cache, the cache responds, thereby avoiding a time-consuming external memory cycle. This allows the processor to complete transfers faster and reduces traffic on the processor bus.

The internal cache on all members of the Intel486 processor family uses a write-through protocol. The IntelDX4 processor can also be configured to implement a write-back protocol. With a write-through protocol, all writes to the cache are immediately written to the external memory that the cache represents. With a write-back protocol, writes to the cache are stored for future memory updating. To reduce the impact of writes on performance, the processor can buffer its write cycles; an operation that writes data to memory can finish before the write cycle is actually performed on the processor bus.

The processor performs a cache line fill to place new information into the on-chip cache. This operation reads four doublewords into a cache line, the smallest unit of storage that can be allocated in the cache. Most read cycles on the processor bus result from cache misses, which cause cache line fills.

The Intel486 processor provides mechanisms to maintain cache consistency between memory and cached data in multiple bus master environments. These mechanisms protect the Intel486 processor from reading invalid data from its own internal cache or from external caches. For example, when the Intel486 processor attempts to read an operand from memory that is also held in the cache of another bus master, the other bus master is forced to write its cached data back to memory before the Intel486 processor can complete its read from memory. This is done because the cached version of the data may have been updated, and so may now be different from the version stored in memory.

Most memory systems optimize the speed of access on a read cycle. This is because the large majority of all memory accesses in a typical system are read accesses. The Intel486 processor's internal cache changes this ratio. Most read requests result in cache hits, so most memory accesses on the processor bus are write cycles. Memory optimization should be done with this in mind.

2.2.4 Floating-Point Unit

The internal floating-point unit performs floating-point operations on the 32-, 64- and 80-bit arithmetic formats as specified in IEEE Standard 754. Like the integer processing unit, the floating-point unit architecture is binary-compatible with the 8087 and 80287 coprocessors. The architecture is 100% compatible with the Intel387 DX and Intel387 SX coprocessors.

Floating-point instructions execute fastest when they are entirely internal to the processor. This occurs when all operands are in the internal registers or cache. When data needs to be read from or written to external locations, burst transfers minimize the time required and a bus locking mechanism ensures that the bus is not relinquished to other bus masters during the transfer. Bus signals are provided to monitor errors in floating-point operations and to control the processor's response to such errors.

2.2.5 Upgrade Power Down Mode

Upgrade Power Down Mode on the Intel486 processor is initiated by the Intel OverDrive® processor using the UP# (Upgrade Present) pin. Upon sensing the presence of the Intel OverDrive Processor, the Intel486 processor three-states its outputs and enters the "Upgrade Power Down Mode," lowering its power consumption. The UP# pin of the Intel486 processor is driven active (low) by the UP# pin of the Intel OverDrive processor. (In the *embedded* Intel486 processor family, the UP# pin has been renamed Reserved, with no changes in functionality.)

2.3 SYSTEM COMPONENTS

Intel offers several chips that are highly compatible with the Intel486 processor. These components can be used to design high-performance embedded systems with a minimum of effort and cost. For components not directly connectable to the Intel486 processor bus, industry-standard interfaces can be used.

The Intel486 processor provides all integer and floating-point CPU functions plus many of the peripheral functions required in a typical computer system. It executes the complete instruction set of the Intel386 processor and Intel387 DX numerics coprocessor, with some extensions. The processor eliminates the need for an external memory management unit, and the on-chip cache minimizes the need for external cache and associated control logic.

The remaining chapters of this manual detail the Intel486 processor's architecture, hardware functions, and interfacing. For more information on the architecture and software interface, see the *Embedded Intel486™ Processor Family Developer's Manual* and the *Intel Architecture Software Developer's Manual*, Volumes 1 and 2.

2.4 SYSTEM ARCHITECTURE

The Intel486 processor can be the foundation for single-processor or multi-processor embedded systems. A single-processor system might be an embedded personal computer designed to use the Intel486 processor. A system design of this type offers higher performance through the integration of floating-point processing, memory management, and caching. More complex embedded systems may use multiple processors that provide, at chip-level, the equivalent of board-level functions. Designs of this type are typically used in multi-user machines, scientific workstations, and engineering workstations.

A typical Intel486 design is shown in Figure 2-1. This example uses a single Intel486 processor with external cache. Other examples of system design are illustrated in the figures that follow.

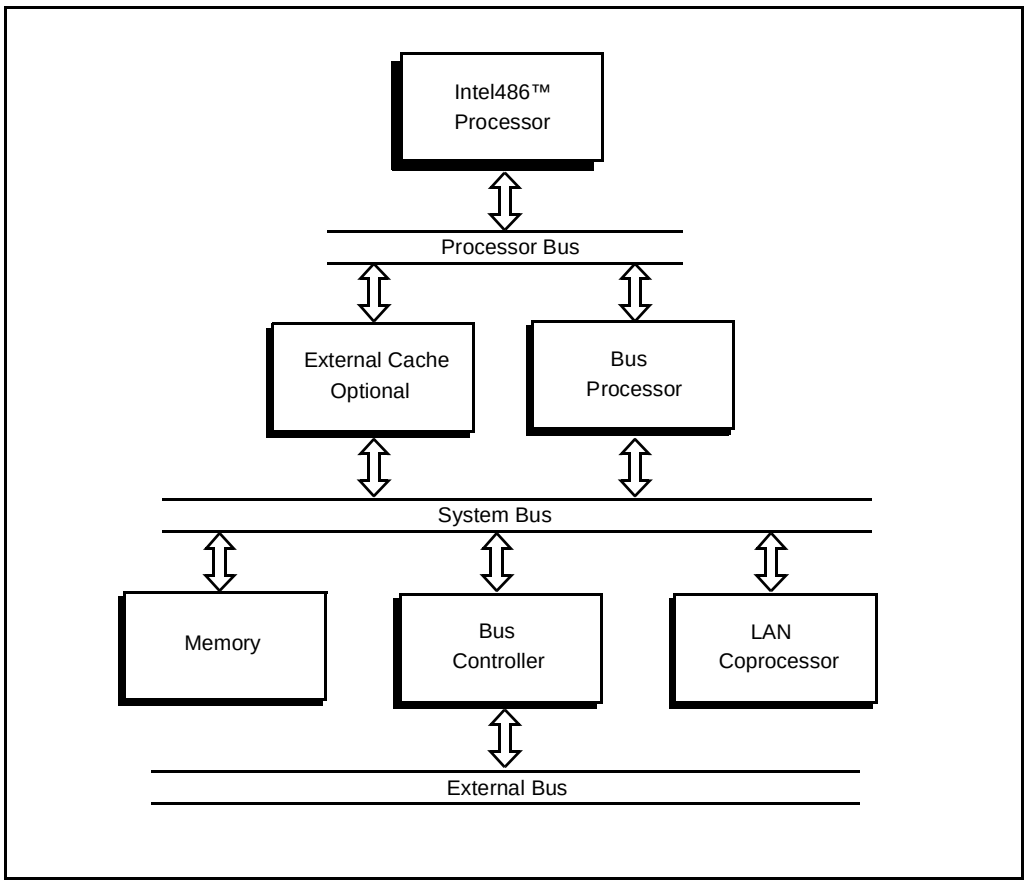


Figure 2-1. A Typical Intel486™ Processor System

2.4.1 Single Processor System

In single-processor systems, the processor handles all peripheral resources and intelligent devices, and executes all software. The Intel486 processor does this in a more efficient way and for a wider range of task complexity than earlier processors. Single-processor systems offer small size and low cost in exchange for flexibility in upgrading or expanding the system. Typical applications include personal computers, small desktop workstations, and embedded controllers. Such applications are implemented as a single board, usually called a motherboard; the processor bus does not extend beyond the board occupied by the Intel486 processor.

Figure 2-2 shows an example of such a system. In a single-processor system, devices that share the processor bus must be selected carefully. All components must interact directly with the processor bus or have interface logic that allows them to do so. The total bus bandwidth requirements

of other components should be no more than 50% of the available processor-bus bandwidth. Traffic above 50% degrades performance of the processor.

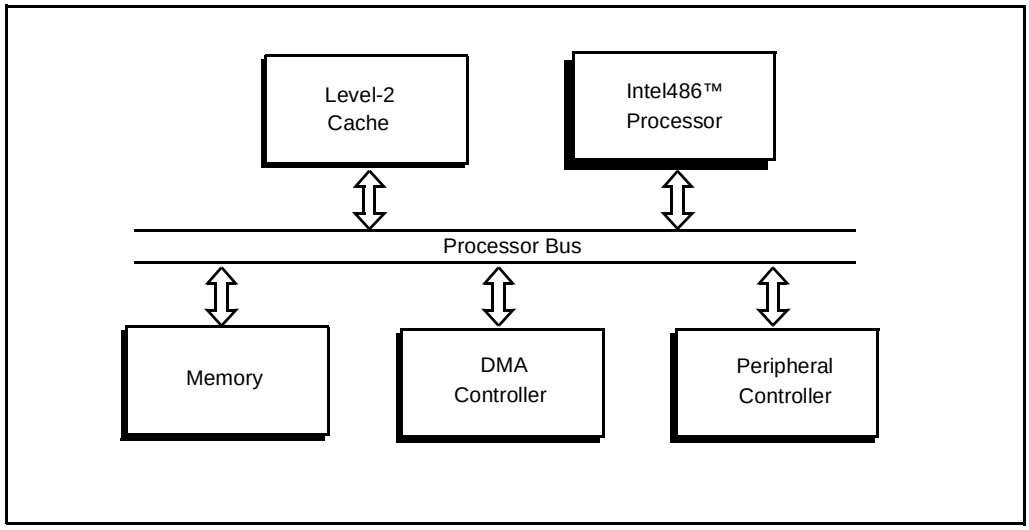


Figure 2-2. Single-Processor System

Two basic design approaches are used to elaborate the single-processor system into a more complex system. The first approach is to add more devices to the processor bus. This can be done up to the limit mentioned above: no more than 50% of the processor-bus bandwidth should be used by devices other than the Intel486 processor. The second design approach is to add more buses to the system. By adding buses, greater bus bandwidth is created in the system as a whole, which in turn allows more devices to be added to the system. The two approaches go hand-in-hand to expand the capabilities of a system. The sections below give only a few examples of the great variety of designs that are possible with Intel486 processor-compatible devices.

2.4.2 Loosely Coupled Multi-Processor System

Loosely coupled multi-processor systems include board-level products that communicate with one another through a standard system bus. In this architecture, each board contains a processor and associated logic. There is typically only one processor per board. Components within each board communicate on either a processor bus or on the buffered system bus. The system bus usually provides extra bandwidth beyond the processor bus.

A typical system is shown in Figure 2-3. Such system-bus boards typically occur in higher-end personal computers and embedded systems that allow for modular expansion. A typical design would include a coprocessor or LAN interface board in a personal computer, or a network-interface board in a file server or gateway. Systems built from these boards can contain a mix of processor types. Devices attached to the processor bus on a given board make demands that may affect system performance. For example, a typical system may use up to 3% of the bus bandwidth to handle 10-Mbit/second Ethernet traffic.

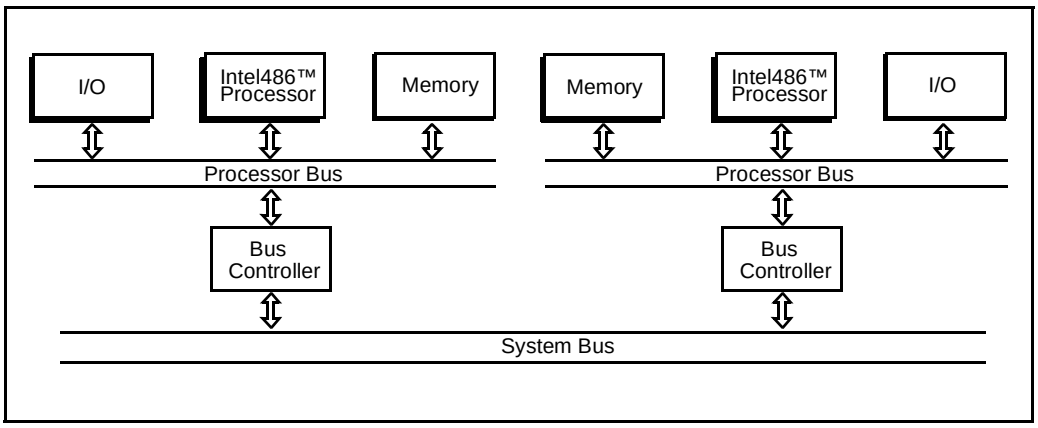


Figure 2-3. Loosely Coupled Multi-processor System

2.4.3 External Cache

External cache allows a system to achieve maximum performance. This cache is essential in tightly coupled multi-processor embedded systems. The external cache consists of cache memory (usually fast SRAM) and cache control logic.

External cache systems typically provide access to the cache from both the processor and the system buses. This is shown in Figure 2-4. These caches typically monitor processor memory accesses, processor access time, and consistency between cache and memory. The cache controller is responsible for maintaining an optimal mix of data and instructions in cache.

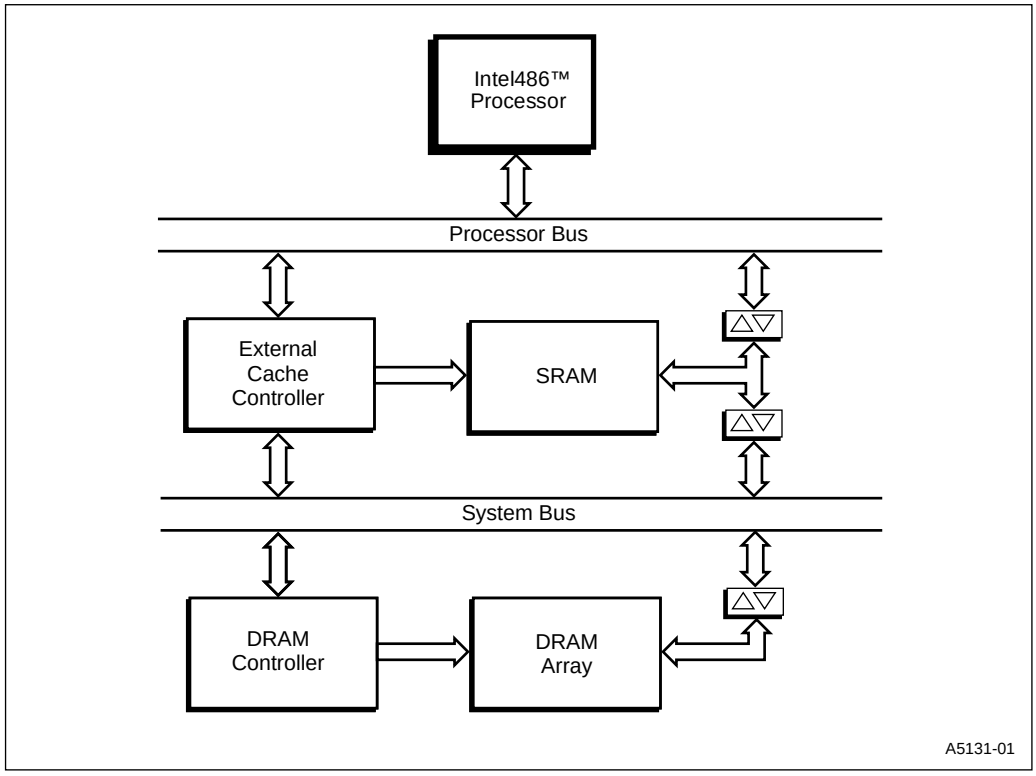


Figure 2-4. External Cache

2.5 SYSTEMS APPLICATIONS

Most Intel486 processor systems can be grouped as one of these types:

- Embedded Personal Computer
- Embedded Controller

Each type of system has distinct design goals and constraints, as described in the following sections. Software running on the processor, even in stand-alone embedded applications, should use a standard operating system such as DOS*, Windows 95*, Windows NT*, OS/2*, or UNIX System V/386*, to facilitate debugging, documentation, and transportability.

2.5.1 Embedded Personal Computers

In single-processor embedded systems, the processor interacts directly with I/O devices and DRAM memory. Other bus masters such as a LAN coprocessor typically reside on the system bus; conventional personal computer architecture puts most peripherals on separate plug-in boards. Expansion is typically limited to memory boards and I/O boards. A standard I/O architecture such as MCA or EISA is used. System cost and size are very important. Figure 2-5 shows an example of an embedded personal computer or an embedded controller application.

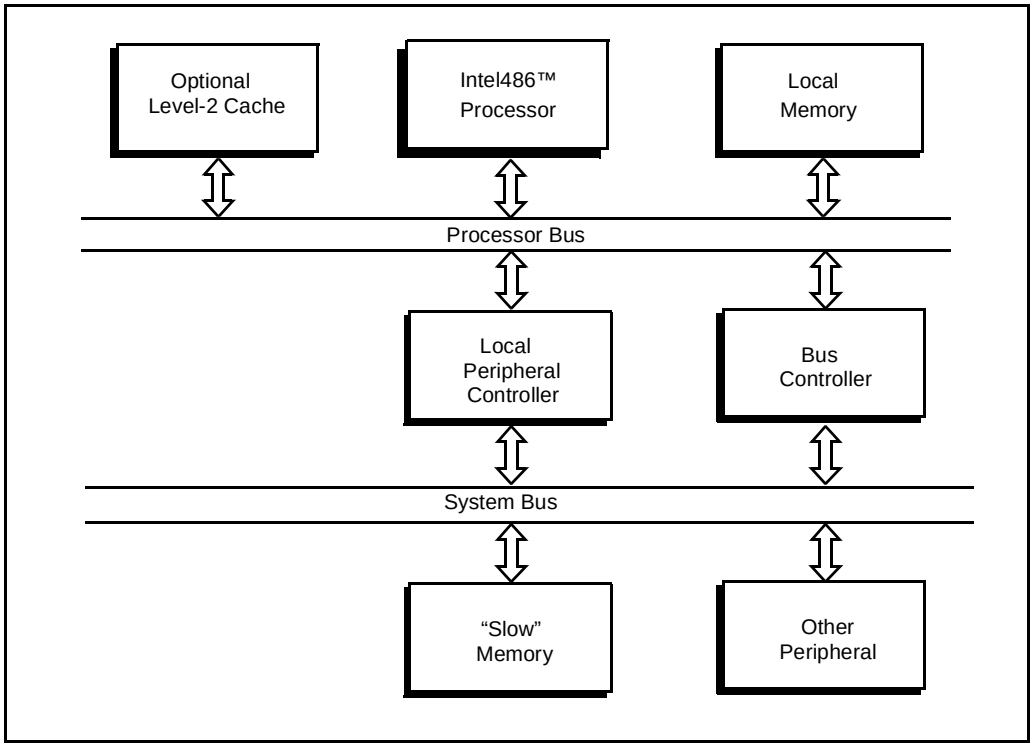


Figure 2-5. Embedded Personal Computer and Embedded Controller Example

External cache is optional in such environments, particularly if system performance is not a critical parameter. Where an external cache is used, memory-access speeds improve only if the cache is designed as a write-back system and memory access has zero to one wait states.

2.5.2 Embedded Controllers

Most embedded controllers perform real-time tasks. The performance of the Intel486 processor and its compatibility with the extensive installed base of Intel386 processors are important factors in its choice. Embedded controllers are usually implemented as stand-alone systems, with less ex-

pansion capability than other applications because they are tailored specifically to a single environment.

If code must be stored in EPROM, ROM, or Flash for non-volatility, but performance is also a critical issue, then the code should be copied into RAM provided specifically for this purpose. Frequently used routines and variables, such as interrupt handlers and interrupt stacks, can be locked in the processor's internal cache so they are always available quickly.

Embedded controllers usually require less memory than other applications, and control programs are usually tightly written machine-level routines that need optimal performance in a limited variety of tasks. The processor typically interacts directly with I/O devices and DRAM memory. Other peripherals connect to the system bus.

3

Internal Architecture

Chapter Contents

CHAPTER 3
INTERNAL ARCHITECTURE

The Intel486™ SX processor has a 32-bit architecture with on-chip memory management and level-1 cache.

The IntelDX2™ and IntelDX4™ processors also have a 32-bit architecture with on-chip memory management and cache, but add clock multiplier and floating-point units. The Intel486 SX and Intel486 DX processors support dynamic bus sizing for the external data bus; that is, the bus size can be specified as 8-, 16-, or 32-bits wide.

Internally, the ultra-low power processors are similar to the Intel486 SX processor, but add a clock control unit. Although the Ultra-Low Power Intel486 SX supports dynamic bus sizing, the Ultra-Low Power Intel486 GX supports only a 16-bit external data bus. The Ultra-Low Power Intel486 GX also has advanced power management features.

Table 3-1 lists the functional units of the embedded Intel486 processors.

Table 3-1. Intel486™ Processor Family Functional Units

Functional Unit	IntelDX2™ and IntelDX4™ Processors	Intel486™ SX Processor	Ultra-Low Power Intel486 SX and Ultra-Low Power Intel486 GX Processors
Bus Interface	✓	✓	✓
Cache (L1)	✓	✓	✓
Instruction Prefetch	✓	✓	✓
Instruction Decode	✓	✓	✓
Control	✓	✓	✓
Integer and Datapath	✓	✓	✓
Segmentation	✓	✓	✓
Paging	✓	✓	✓
Floating-Point	✓		
Clock Multiplier	✓		
Clock Control			✓

Figure 3-1 is a block diagram of the embedded IntelDX2 and IntelDX4 processors. Note that the cache unit is 8-Kbytes for the IntelDX2 processor and 16 Kbytes for the IntelDX4 processor.

Figure 3-2 is a block diagram of the embedded Intel486 SX processor and Figure 3-3 is a block diagram of the Ultra-Low Power Intel486 SX and the Ultra-Low Power Intel486 GX processors.

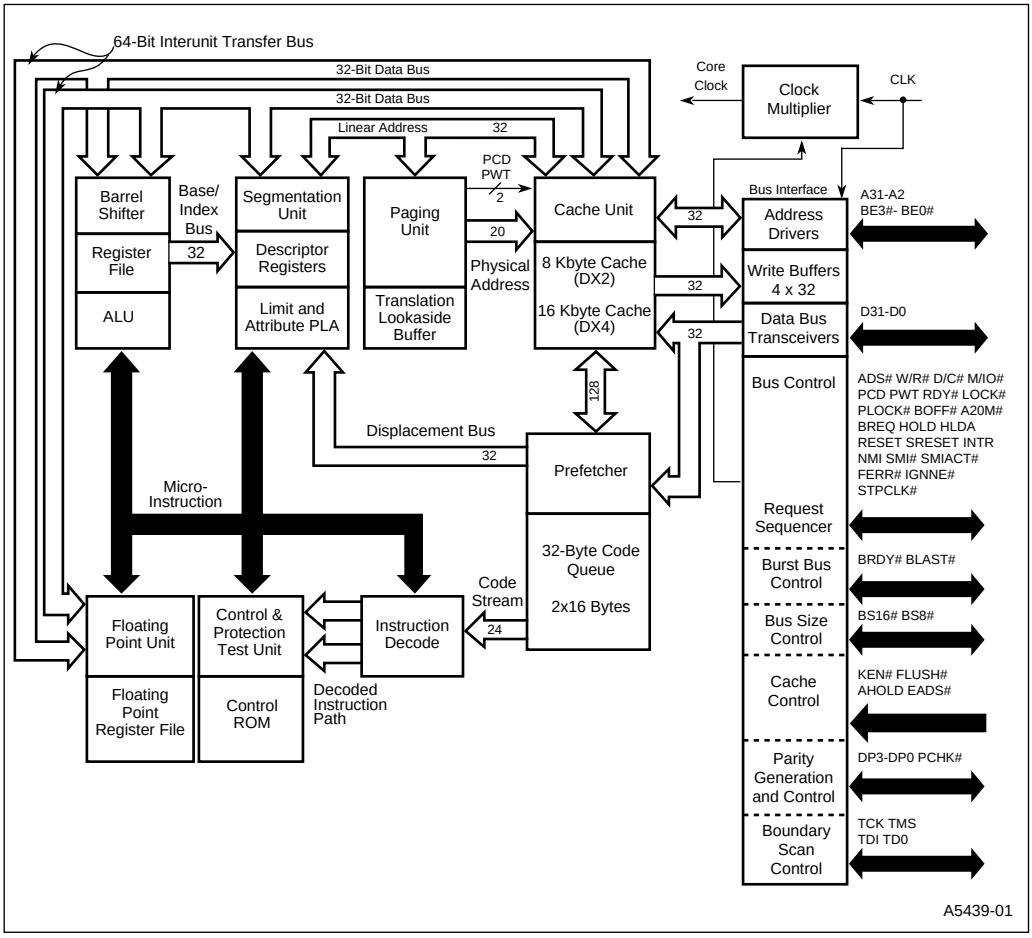


Figure 3-1. IntelDX2™ and IntelDX4™ Processors Block Diagram

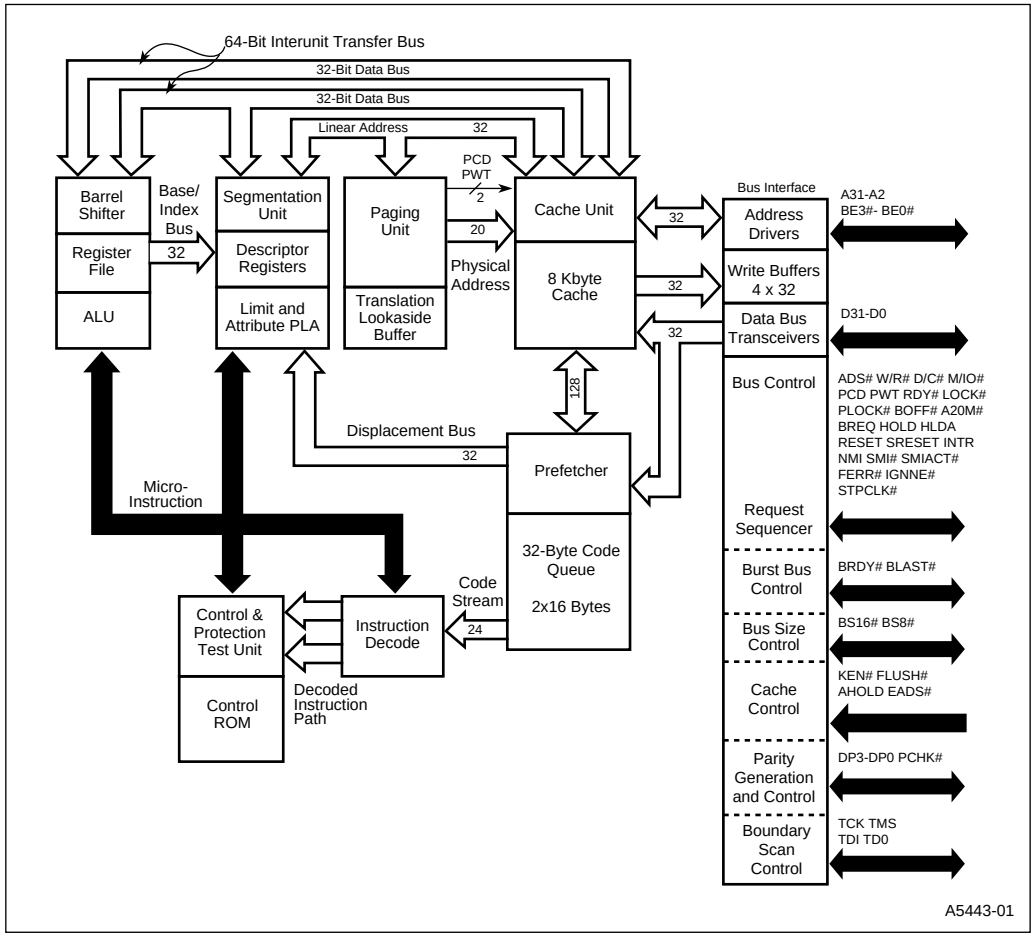


Figure 3-2. Intel486™ SX Processor Block Diagram

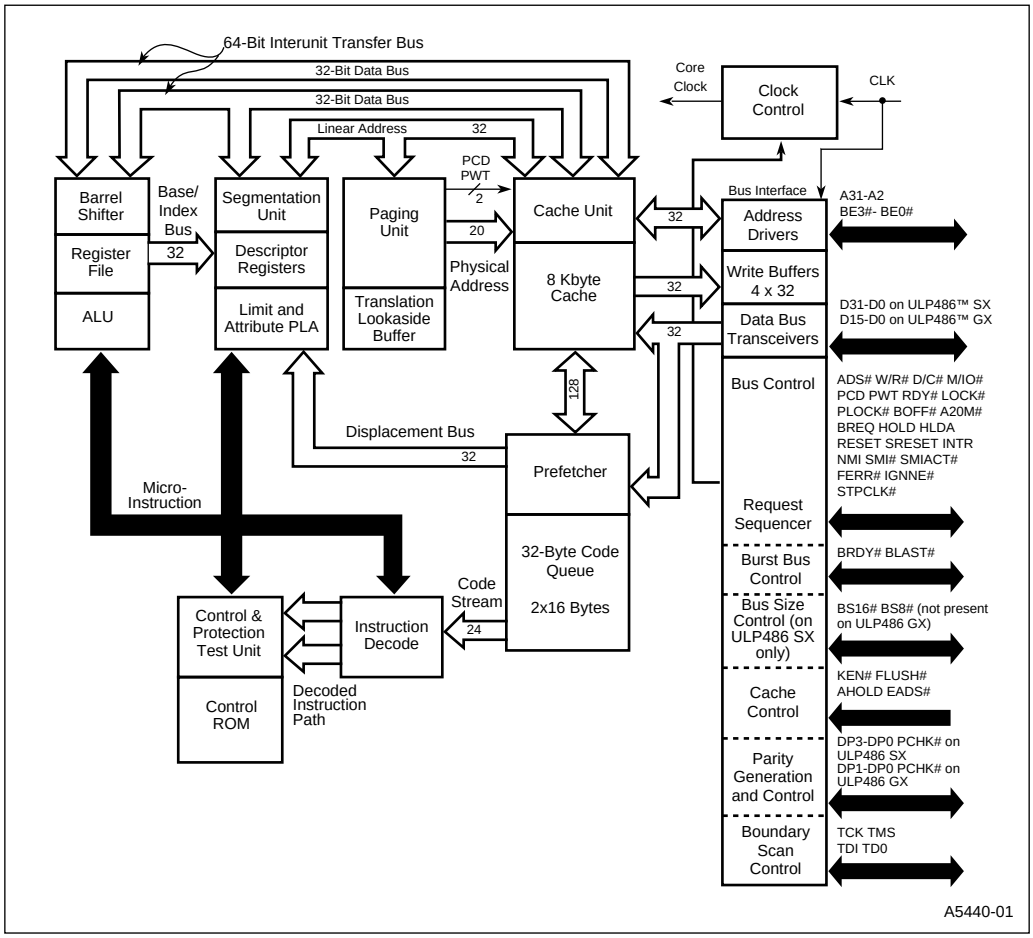


Figure 3-3. Ultra-Low Power Intel486™ SX and Ultra-Low Power Intel486 GX Processors Block Diagram

Signals from the external 32-bit processor bus reach the internal units through the bus interface unit. On the internal side, the bus interface unit and cache unit pass addresses bi-directionally through a 32-bit bus. Data is passed from the cache to the bus interface unit on a 32-bit data bus. The closely coupled cache and instruction prefetch units simultaneously receive instruction prefetches from the bus interface unit over a shared 32-bit data bus, which the cache also uses to receive operands and other types of data. Instructions in the cache are accessible to the instruction prefetch unit, which contains a 32-byte queue of instructions waiting to be executed.

The on-chip cache is 16 Kbytes for the IntelDX4 processor and 8 Kbytes for all other members of the Intel486 processor family. It is 4-way set associative and follows a write-through policy. The Write-Back Enhanced IntelDX4 processor can be set to use an on-chip write-back cache pol-

icy. The on-chip cache includes features to provide flexibility in external memory system design. Individual pages can be designated as cacheable or non-cacheable by software or hardware. The cache can also be enabled and disabled by software or hardware.

Internal cache memory allows frequently used data and code to be stored on-chip, reducing accesses to the external bus. RISC design techniques reduce instruction cycle times. A burst bus feature enables fast cache fills.

When internal requests for data or instructions can be satisfied from the cache, time-consuming cycles on the external processor bus are avoided. The bus interface unit is only involved when an operation needs access to the processor bus. Many internal operations are therefore transparent to the external system.

The instruction decode unit translates instructions into low-level control signals and microcode entry points. The control unit executes microcode and controls the integer, floating-point, and segmentation units. Computation results are placed in internal registers within the integer or floating-point units, or in the cache. Internal storage locations (datapaths) are kept in the integer unit.

The cache shares two 32-bit data buses with the segmentation, integer, and floating-point units. These two buses can be used together as a 64-bit inter-unit transfer bus. When 64-bit segment descriptors are passed from the cache to the segmentation unit, 32 bits are passed directly over one data bus and the other 32 bits are passed through the integer unit, so that all 64 bits reach the segmentation unit simultaneously.

The memory management unit (MMU) consists of a segmentation unit and a paging unit which perform address generation. The segmentation unit translates logical addresses and passes them to the paging and cache units on a 32-bit linear address bus. Segmentation allows management of the logical address space by providing easy relocation of data and code and efficient sharing of global resources.

The paging mechanism operates beneath segmentation and is transparent to the segmentation process. The paging unit translates linear addresses into physical addresses, which are passed to the cache on a 20-bit bus. Paging is optional and can be disabled by system software. To implement a virtual memory system, the Intel486 processor supports full restartability for all page and segment faults.

The Intel486 processor instruction set includes the complete Intel386™ processor instruction set along with extensions to serve new applications and increase performance. The on-chip memory MMU is completely compatible with the Intel386 processor MMU. Software written for previous members of the Intel architecture family runs on the Intel486 processor without modification.

Memory is organized into one or more variable length segments, each up to four Gbytes (2^{32} bytes). A segment can have attributes associated with it that include its location, size, type (i.e., stack, code, or data), and protection characteristics. Each task on an Intel486 processor can have a maximum of 16,381 segments and each are up to four Gbytes in size. Thus, each task has a maximum of 64 terabytes (trillion bytes) of virtual memory.

The segmentation unit provides four levels of protection for isolating and protecting applications and the operating system from each other. The hardware-enforced protection allows the design of systems with a high degree of software integrity.

The Intel486 processor has four modes of operation: Real Address Mode (Real Mode), Protected Mode, Virtual Mode (within Protected Mode), and System Management Mode (SMM). In Real Mode the Intel486 processor operates as a very fast 8086. Real Mode is required primarily to set up the Intel486 processor for Protected Mode operation.

Protected Mode provides access to the sophisticated memory management paging and privilege capabilities of the processor. Within Protected Mode, software can perform a task switch to enter into tasks designated as Virtual 8086 Mode tasks. Each Virtual 8086 task behaves with 8086 semantics, allowing 8086 processor software (an application program or an entire operating system) to execute.

System Management Mode (SMM) provides system designers with a means of adding new software-controlled features to their computer products that always operate transparently to the operating system (OS) and software applications. SMM is intended for use only by system firmware, not by applications software or general purpose systems software.

The Intel486 processor also has features that facilitate high-performance hardware designs. The 1X bus clock input eases high-frequency board-level designs. The clock multiplier on IntelDX2 and IntelDX4 processors improves execution performance without increasing board design complexity. The clock multiplier enhances all operations operating out of the cache that are not blocked by external bus accesses. The burst bus feature enables fast cache fills.

3.1 INSTRUCTION PIPELINING

Not every instruction involves all internal units. When an instruction needs the participation of several units, each unit operates in parallel with others on instructions at different stages of execution. Although each instruction is processed sequentially, several instructions are at varying stages of execution in the processor at any given time. This is called *instruction pipelining*. Instruction prefetch, instruction decode, microcode execution, integer operations, floating-point operations, segmentation, paging, cache management, and bus interface operations are all performed simultaneously. Figure 3-4 shows some of this parallelism for a single instruction: the instruction fetch, two-stage decode, execution, and register write-back of the execution result. Each stage in this pipeline can occur in one clock cycle.

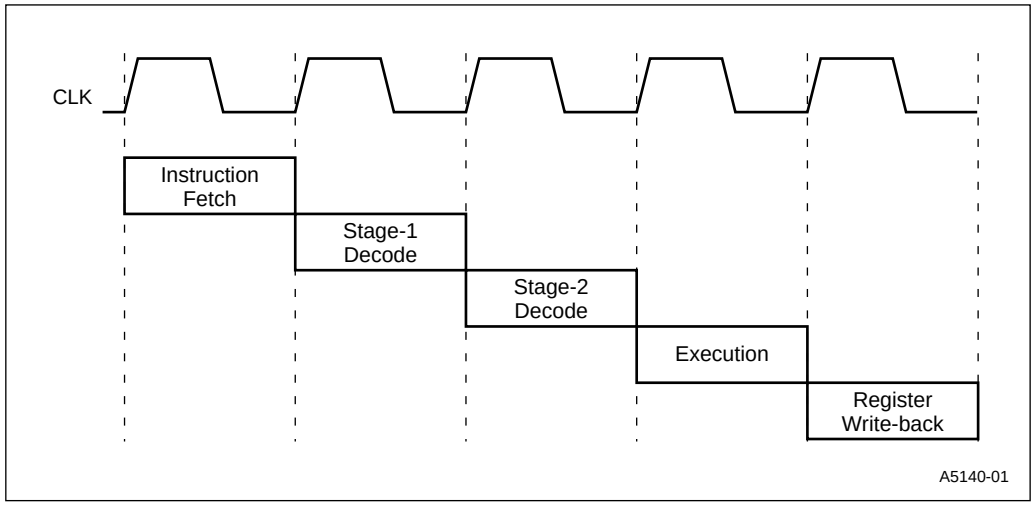


Figure 3-4. Internal Pipelining

The internal pipelining on the Intel486 processor offers an important performance advantage over many single-clock RISC processors: in the Intel486 processor, data can be loaded from the cache with one instruction and used by the next instruction in the next clock. This performance advantage results from the stage-1 decode step, which initiates memory accesses before the execution cycle. Because most compilers and application programs follow load instructions with instructions that operate on the loaded data, this method optimizes the execution of existing binary code.

The method has a performance trade-off: an instruction sequence that changes register contents and then uses that register in the next instruction to access memory takes three clocks rather than two. This trade-off is only a minor disadvantage, however, since most instructions that access memory use the stable contents of the stack pointer or frame pointer, and the additional clock is not used very often. Compilers often place an unrelated instruction between one that changes an addressing register and one that uses the register. Such code is compatible with the Intel386 processor, and the Intel486 processor provides special stack increment/decrement hardware and an extra register port to execute back-to-back stack push/pop instructions in a single clock.

3.2 BUS INTERFACE UNIT

The bus interface unit prioritizes and coordinates data transfers, instruction prefetches, and control functions between the processor's internal units and the outside system. Internally, the bus interface unit communicates with the cache and the instruction prefetch units through three 32-bit buses, as shown in Figure 3-1. Externally, the bus interface unit provides the processor bus signals, described in Chapter 3. Except for cycle definition signals, all external bus cycles, memory reads, instruction prefetches, cache line fills, etc., look like conventional microprocessor cycles to external hardware, with all cycles having the same bus timing.

The bus interface unit contains the following architectural features:

- Address Transceivers and Drivers — The A31–A2 address signals are driven on the processor bus, together with their corresponding byte-enable signals, BE3#–BE0#. The high-order 28 address signals are bidirectional, allowing external logic to drive cache invalidation addresses into the processor.
- Data Bus Transceivers — The D31–D0 data signals are driven onto and received from the processor bus (for the Ultra-Low Power Intel486 GX processor, signals D15–D0 comprise the data bus transceivers).
- Bus Size Control — Three sizes of external data bus can be used: 32, 16, and 8 bits wide. Two inputs from external logic specify the width to be used. Bus size can be changed on a cycle-by-cycle basis. The Ultra-Low Power Intel486 GX does not support dynamic bus sizing; its external data bus is 16 bits wide.
- Write Buffering — Up to four write requests can be buffered, allowing many internal operations to continue without waiting for write cycles to be completed on the processor bus.
- Bus Cycles and Bus Control — A large selection of bus cycles and control functions are supported, including burst transfers, non-burst transfers (single- and multiple-cycle), bus arbitration (bus request, bus hold, bus hold acknowledge, bus locking, bus pseudo-locking, and bus backoff), floating-point error signalling, interrupts, and reset. Two software-controlled outputs enable page caching on a cycle-by-cycle basis. One input and one output are provided for controlling burst read transfers.
- Parity Generation and Control — Even parity is generated on writes to the processor and checked on reads. An error signal indicates a read parity error.
- Cache Control — Cache control and consistency operations are supported. Three inputs allow the external system to control the consistency of data stored in the internal cache unit. Two special bus cycles allow the processor to control the consistency of external cache.

3.2.1 Data Transfers

To support the cache, the bus interface unit reads 16-byte cacheable transfers of operands, instructions, and other data on the processor bus and passes them to the cache unit. When cache contents are updated from an internal source, such as a register, the bus interface unit writes the updated cache information to the external system. Non-cacheable read transfers are passed through the cache to the integer or floating-point units.

During instruction prefetch, the bus interface unit reads instructions on the processor bus and passes them to both the instruction prefetch unit and the cache. The instruction prefetch unit may then obtain its inputs directly from the cache.

3.2.2 Write Buffers

The bus interface unit has temporary storage for buffering up to four 32-bit write transfers to memory. Addresses, data, or control information can be buffered. Single I/O-mapped writes are not buffered, although multiple I/O writes may be buffered. The buffers can accept memory

writes as fast as one per clock. Once a write request is buffered, the internal unit that generated the request is free to continue processing. If no higher-priority request is pending and the bus is free, the transfer is propagated as an immediate write cycle to the processor bus. When all four write buffers are full, any subsequent write transfer stalls inside the processor until a write buffer becomes available.

The bus interface unit can re-order pending reads in front of buffered writes. This is done because pending reads can prevent an internal unit from continuing, whereas buffered writes need not have a detrimental effect on processing speed.

Writes are propagated to the processor bus in the first-in-first-out order in which they are received from the internal unit. However, a subsequently generated read request (data or instruction) may be re-ordered in front of buffered writes. As a protection against reading invalid data, this re-ordering of reads in front of buffered writes occurs only if all buffered writes are cache hits. Because an external read is generated only for a cache miss, and is re-ordered in front of buffered writes only if all such buffered writes are cache hits, any read generated on the external bus with this protection never reads a location that is about to be written by a buffered write. This re-ordering can only happen once for a given set of buffered writes, because the data returned by the read cycle could otherwise replace data about to be written from the write buffers.

To ensure that no more than one such re-ordering is done for a given set of buffered writes, all buffered writes are re-flagged as cache misses when a read request is re-ordered ahead of them. Buffered writes thus marked are propagated to the processor bus before the next read request is acted upon. Invalidation of data in the internal cache also causes all pending writes to be flagged as cache misses. Disabling the cache unit disables the write buffers, which eliminates any possibility of re-ordering bus cycles.

3.2.3 Locked Cycles

The processor can generate signals to lock a contiguous series of bus cycles. These cycles can then be performed without interference from other bus masters, if external logic observes these lock signals. One example of a locked operation is a semaphor read-modify-write update, where a resource control register is updated. No other operations should be allowed on the bus until the entire locked semaphor update is completed.

When a locked read cycle is generated, the internal cache is not read. All pending writes in the buffer are completed first. Only then is the read part of the locked operation performed, the data modified, the result placed in a write buffer, and a write cycle performed on the processor bus. This sequence of operations ensures that all writes are performed in the order in which they were generated.

3.2.4 I/O Transfers

Transfers to and from I/O locations have some restrictions to ensure data integrity:

- Caching — I/O reads are never cached.
- Read Re-ordering — I/O reads are never re-ordered ahead of buffered writes to memory. This ensures that the processor has completed updating all memory locations before reading status from a device.

- Writes — Single I/O writes are never buffered. When processing an OUT instruction, internal execution stops until all buffered writes and the I/O write are completed on the processor bus. This allows time for external logic to drive a cache invalidate cycle or mask interrupts before the processor executes the next instruction. The processor completes updating all memory locations before writing to the I/O location. Repeated OUT instructions may be buffered.

The write buffers and the cache unit determine I/O device recovery time. In the Intel386 processor, back-to-back write recovery time could be guaranteed to exceed a certain value by inserting a jump to the next instruction that writes to the I/O device. This forced an instruction prefetch cycle that could only be performed after the preceding write was completed. This technique is not used in the Intel486 processor because a prefetch can be satisfied internally by the cache and recovery time may be too short. The same effect is achieved in the Intel486 processor by explicitly generating a read to an area of memory that is not cacheable. Because the Intel486 processor does not buffer single I/O writes, such a read is not done until the I/O write is completed.

3.3 CACHE UNIT

The cache unit stores copies of recently read instructions, operands, and other data. When the processor requests information already in the cache, called a cache hit, no processor-bus cycle is required. When the processor requests information not in the cache, called a cache miss, the information is read into the cache in one or more 16-byte cacheable data transfers, called cache line fills. An internal write request to an area currently in the cache causes two distinct actions if the cache is using a write-through policy: the cache is updated, and the write is also passed through the cache to memory. If the cache is using a write-back policy, then the internal write request only causes the cache to be updated and the write is stored for future main memory updating.

The cache transfers data to other units on two 32-bit buses, as shown in Figure 3-1. The cache receives linear addresses on a 32-bit bus and the corresponding physical addresses on a 20-bit bus. The cache and instruction prefetch units are closely coupled. 16-Byte blocks of instructions in the cache can be passed quickly to the instruction prefetch unit. Both units read information in 16-byte blocks.

The cache can be accessed as often as once each clock. The cache acts on physical addresses, which minimizes the number of times the cache must be flushed. When both the cache and the cache write-through functions are disabled, the cache may be used as a high-speed RAM.

3.3.1 Cache Structure

The cache has a four-way set associative organization. There are four possible cache locations to store data from a given area of memory. Four-way association is a compromise between the speed of a direct-mapped cache during cache hits and the high cache-hit ratio of a fully associative cache. As shown in Figure 3-5, the 8-Kbyte data block is divided into four data ways, each containing 128 16-byte sets, or cache lines (the DX4 processor has 256 16-byte sets). Each cache line holds data from 16 successive byte addresses in memory, beginning with an address divisible by 16.

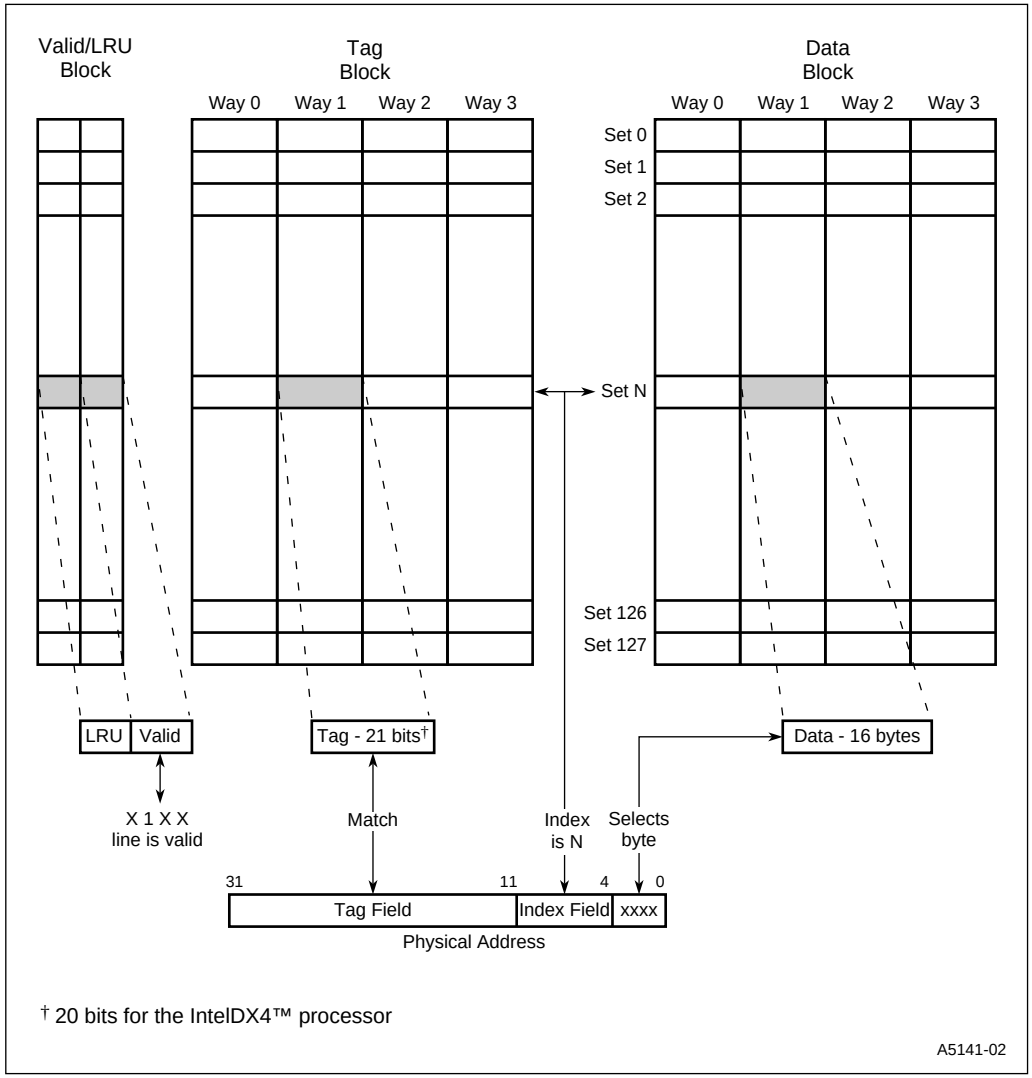


Figure 3-5. Cache Organization

Cache addressing is performed by dividing the high-order 28 bits of the physical address into three parts, as shown in Figure 3-5. The 7 bits of the index field specify the set number, one of 128, within the cache. The high-order 21 bits (20 on the IntelDX4 processor) are the tag field; these bits are compared with tags for each cache line in the indexed set, and they indicate whether a 16-byte cache line is stored for that physical address. The low-order 4 bits of the physical address select the byte within the cache line. Finally, a 4-bit valid field, one for each way within a given set, indicates whether the cached data at that physical address is currently valid.

3.3.2 Cache Updating

When a cache miss occurs on a read, the 16-byte block containing the requested information is written into the cache. Data in the neighborhood of the required data is also read into the cache, but the exact position of data within the cache line depends on its location in memory with respect to addresses divisible by 16.

Any area of memory can be cacheable, but any page of memory can be declared not cacheable by setting a bit in its page table entry. The I/O region of memory is non-cacheable. When a read from memory is initiated on the bus, external logic can indicate whether the data may be placed in cache, as discussed in Chapter 4, "Bus Operation." If the read is cacheable, the processor attempts to read an entire 16-byte cache line.

The cache unit follows a write-through cache policy. The unit on the IntelDX4 processor can be configured to be a write-through or write-back cache. Cache line fills are performed only for read misses, never for write misses. When the processor is enabled for normal caching and write-through operation, every internal write to the cache (cache hit) not only updates the cache but is also passed along to the bus interface unit and propagated through the processor bus to memory. The only conditions under which data in the cache differs from the corresponding data in memory occur when a processor write cycle to memory is delayed by buffering in the bus interface unit, or when an external bus master alters the memory area mapped to the internal cache. When the IntelDX4 processor is enabled for normal caching and write-back operation, an internal write only causes the cache to be updated. The modified data is stored for the future update of main memory and is not immediately written to memory.

3.3.3 Cache Replacement

Replacement in the cache is handled by a pseudo-LRU (least recently used) mechanism. This mechanism maintains three bits for each set in the valid/LRU block, as shown in Figure 3-5. The LRU bits are updated on each cache hit or cache line fill. Each cache line (four per set) also has an associated valid bit that indicates whether the line contains valid data. When the cache is flushed or the processor is reset, all of the valid bits are cleared. When a cache line is to be filled, a location for the fill is selected by simply finding any cache line that is invalid. If no cache line is invalid, the LRU bits select the line to be overwritten. Valid bits are not set for lines that are only partially valid.

Cache lines can be invalidated individually by a cache line invalidation operation on the processor bus. When such an operation is initiated, the cache unit compares the address to be invalidated with tags for the lines currently in cache and clears the valid bit if a match is found. A cache flush operation is also available. This invalidates the entire contents of the internal cache unit.

3.3.4 Cache Configuration

Configuration of the cache unit is controlled by two bits in the processor's machine status register (CR0). One of these bits enables caching (cache line fills). The other bit enables memory write-through. Table 3-2 shows the four configuration options. Chapter 4, "Bus Operation," gives details.

Table 3-2. Cache Configuration Options

Cache Enabled	Write-through Enabled	Operating Mode
no	no	Cache line fills, cache write-throughs, and cache invalidations are disabled. This configuration allows the internal cache to be used as high-speed static RAM.
no	yes	Cache line fills are disabled, and cache write-throughs and cache invalidations are enabled. This configuration allows software to disable the cache for a short time, then re-enable it without flushing the original contents.
yes	no	INVALID
yes	yes	Cache line fills, cache write-throughs, and cache invalidations are enabled. This is the normal operating configuration.

When caching is enabled, memory reads and instruction prefetches are cacheable. These transfers are cached if external logic asserts the cache enable input in that bus cycle, and if the current page table entry allows caching. During cycles in which caching is disabled, cache lines are not filled on cache misses. However, the cache remains active even though it is disabled for further filling. Data already in the cache is used if it is still valid. When all data in the cache is flagged invalid, as happens in a cache flush, all internal read requests are propagated as bus cycles to the external system.

When cache write-through is enabled, all writes, including those that are cache hits, are written through to memory. Invalidation operations remove a line from cache if the invalidate address maps to a cache line. When cache write-throughs are disabled, an internal write request that is a cache hit does not cause a write-through to memory, and cache invalidation operations are disabled. With both caching and cache write-through disabled, the cache can be used as a high-speed static RAM. In this configuration, the only write cycles that are propagated to the processor bus are cache misses, and cache invalidation operations are ignored.

The IntelDX4 processor can also be configured to use a write-back cache policy. For detailed information on the Intel486 processor cache feature, and on the Write-Back Enhanced IntelDX4 processor, refer to Chapter 6, “Cache Subsystem.”

3.4 INSTRUCTION PREFETCH UNIT

When the bus interface unit is not performing bus cycles to execute an instruction, the instruction prefetch unit uses the bus interface unit to prefetch instructions. By reading instructions before they are needed, the processor rarely needs to wait for an instruction prefetch cycle on the processor bus.

Instruction prefetch cycles read 16-byte blocks of instructions, starting at addresses numerically greater than the last-fetched instruction. The prefetch unit, which has a direct connection (not shown in Figure 3-1) to the paging unit, generates the starting address. The 16-byte prefetched blocks are read into both the prefetch and cache units simultaneously. The prefetch queue in the prefetch unit stores 32 bytes of instructions. As each instruction is fetched from the queue, the code part is sent to the instruction decode unit and (depending on the instruction) the displacement part is sent to the segmentation unit, where it is used for address calculation. If loops are

encountered in the program being executed, the prefetch unit gets copies of previously executed instructions from the cache.

The prefetch unit has the lowest priority for processor bus access. Assuming zero wait-state memory access, prefetch activity never delays execution. However, if there is no pending data transfer, prefetching may use bus cycles that would otherwise be idle. The prefetch unit is flushed whenever the next instruction needed is not in numerical sequence with the previous instruction; for example, during jumps, task switches, exceptions, and interrupts.

The prefetch unit never accesses beyond the end of a code segment and it never accesses a page that is not present. However, prefetching may cause problems for some hardware mechanisms. For example, prefetching may cause an interrupt when program execution nears the end of memory. To keep prefetching from reading past a given address, instructions should come no closer to that address than one byte plus one aligned 16-byte block.

3.5 INSTRUCTION DECODE UNIT

The instruction decode unit receives instructions from the instruction prefetch unit and translates them in a two-stage process into low-level control signals and microcode entry points, as shown in Figure 3-1. Most instructions can be decoded at a rate of one per clock. Stage 1 of the decode, shown in Figure 3-4, initiates a memory access. This allows execution of a two-instruction sequence that loads and operates on data in just two clocks, as described in Section 3.2.

The decode unit simultaneously processes instruction prefix bytes, opcodes, modR/M bytes, and displacements. The outputs include hardwired microinstructions to the segmentation, integer, and floating-point units. The instruction decode unit is flushed whenever the instruction prefetch unit is flushed.

3.6 CONTROL UNIT

The control unit interprets the instruction word and microcode entry points received from the instruction decode unit. The control unit has outputs with which it controls the integer and floating-point processing units. It also controls segmentation because segment selection may be specified by instructions.

The control unit contains the processor's microcode. Many instructions have only one line of microcode, so they can execute in an average of one clock cycle. Figure 3-4 shows how execution fits into the internal pipelining mechanism.

3.7 INTEGER (DATAPATH) UNIT

The integer and datapath unit identifies where data is stored and performs all of the arithmetic and logical operations available in the Intel386 processor's instruction set, plus a few new instructions. It has eight 32-bit general-purpose registers, several specialized registers, an ALU, and a barrel shifter. Single load, store, addition, subtraction, logic, and shift instructions execute in one clock.

Two 32-bit bidirectional buses connect the integer and floating-point units. These buses are used together for transferring 64-bit operands. The same buses also connect the processing units with

the cache unit. The contents of the general purpose registers are sent to the segmentation unit on a separate 32-bit bus for generation of effective addresses.

3.8 FLOATING-POINT UNIT

The floating-point unit executes the same instruction set as the 387 math coprocessor. The unit contains a push-down register stack and dedicated hardware for interpreting the 32-, 64-, and 80-bit formats as specified in IEEE Standard 754. An output signal passed through to the processor bus indicates floating-point errors to the external system, which in turn can assert an input to the processor indicating that the processor should ignore these errors and continue normal operations.

3.8.1 IntelDX2™ and IntelDX4™ Processor On-Chip Floating-Point Unit

The IntelDX2 and IntelDX4 processors incorporate the basic Intel486 processor 32-bit architecture, with on-chip memory management and cache memory units. They also have an on-chip floating-point unit (FPU) that operates in parallel with the arithmetic and logic unit. The FPU provides arithmetic instructions for a variety of numeric data types and executes numerous built-in transcendental functions (e.g., tangent, sine, cosine, and log functions). The floating-point unit fully conforms to the ANSI/IEEE standard 754-1985 for floating-point arithmetic.

All software written for the Intel386 processor, Intel387 math coprocessor and previous members of the 86/87 architectural family runs on these processors without modifications.

3.9 SEGMENTATION UNIT

A segment is a protected, independent address space. Segmentation is used to enforce isolation among application programs, to invoke recovery procedures, and to isolate the effects of programming errors.

The segmentation unit translates a segmented address issued by a program, called a logical address, into an unsegmented address, called a linear address. The locations of segments in the linear address space are stored in data structures called segment descriptors. The segmentation unit performs its address calculations using segment descriptors and displacements (offsets) extracted from instructions. Linear addresses are sent to the paging and cache units. When a segment is accessed for the first time, its segment descriptor is copied into a processor register. A program can have as many as 16,383 segments. Up to six segment descriptors can be held in processor registers at a time. Figure 3-6 shows the relationships between logical, linear, and physical addresses.

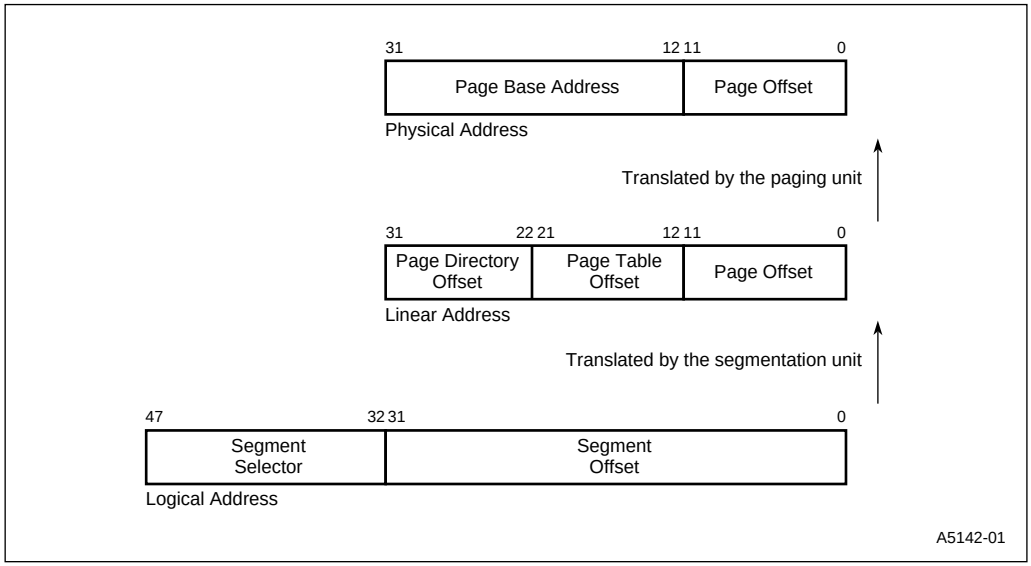


Figure 3-6. Segmentation and Paging Address Formats

3.10 PAGING UNIT

The paging unit allows access to data structures larger than the available memory space by keeping them partly in memory and partly on disk. Paging divides the linear address space into 4-Kbyte blocks called pages. Paging uses data structures in memory called page tables for mapping a linear address to a physical address. The cache uses physical addresses and puts them on the processor bus. The paging unit also identifies problems, such as accesses to a page that is not resident in memory, and raises exceptions called page faults. When a page fault occurs, the operating system has a chance to bring the required page into memory from disk. If necessary, it can free space in memory by sending another page out to disk. If paging is not enabled, the physical address is identical to the linear address.

The paging unit includes a translation lookaside buffer (TLB) that stores the 32 most recently used page table entries. Figure 3-7 shows the TLB data structures. The paging unit looks up linear addresses in the TLB. If the paging unit does not find a linear address in the TLB, the unit generates requests to fill the TLB with the correct physical address contained in a page table in memory. Only when the correct page table entry is in the TLB does the bus cycle take place. When the paging unit maps a page in the linear address space to a page in physical memory, it maps only the upper 20 bits of the linear address. The lowest 12 bits of the physical address come unchanged from the linear address.

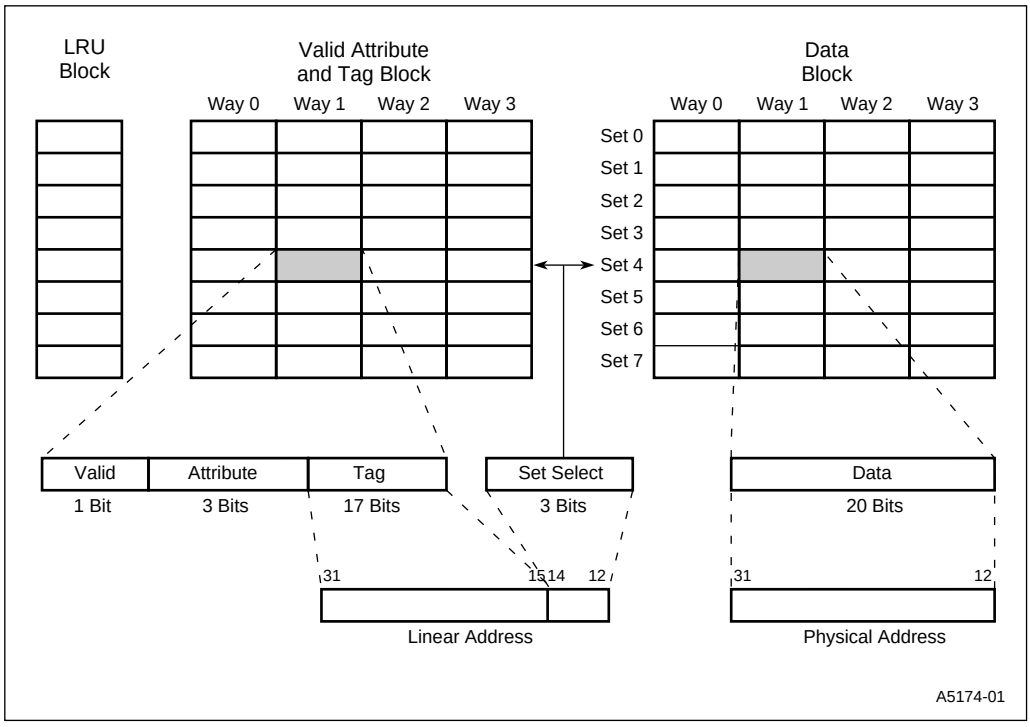


Figure 3-7. Translation Lookaside Buffer

Most programs access only a small number of pages during any short span of time. When this is true, the pages stay in memory and the address translation information stays in the TLB. In typical systems, the TLB satisfies 99% of the requests to access the page tables. The TLB uses a pseudo-LRU algorithm, similar to the cache, as a content-replacement strategy.

The TLB is flushed whenever the page directory base register (CR3) is loaded. Page faults can occur during either a page directory read or a page table read. The cache can be used to supply data for the TLB, although this may not be desirable when external logic monitors TLB updates.

Unlike segmentation, paging is invisible to application programs and does not provide the same kind of protection against programs altering data outside a restricted part of memory. Paging is visible to the operating system, which uses it to satisfy application program memory requirements. For more information on paging and segmentation, see the *Embedded Intel486™ Developer's Manual*.

4

Bus Operation

Chapter Contents

CHAPTER 4
BUS OPERATION

All Intel486™ processors operate in Standard Bus (write-through) mode. However, when the internal cache of the Write-Back Enhanced IntelDX4™ processor is configured in write-back mode, the processor bus operates in the Enhanced Bus mode, which is described in Section 4.4. When the internal cache of the Write-Back Enhanced IntelDX4 processor is configured in write-through mode, the processor bus operates in Standard Bus mode, identical to the other embedded Intel486 processors.

4.1 DATA TRANSFER MECHANISM

All data transfers occur as a result of one or more bus cycles. Logical data operands of byte, word and doubleword lengths may be transferred without restrictions on physical address alignment. Data may be accessed at any byte boundary but two or three cycles may be required for unaligned data transfers. (See Section 4.1.2, "Dynamic Data Bus Sizing," and Section 4.1.5, "Operand Alignment.")

The Intel486 processor address signals are split into two components. High-order address bits are provided by the address lines, A31–A2. The byte enables, BE3#–BE0#, form the low-order address and provide linear selects for the four bytes of the 32-bit address bus.

The byte enable outputs are asserted when their associated data bus bytes are involved with the present bus cycle, as listed in Table 4-1. Byte enable patterns that have a deasserted byte enable separating two or three asserted byte enables never occur (see Table 4-5 on page 4-7). All other byte enable patterns are possible.

Table 4-1. Byte Enables and Associated Data and Operand Bytes

Byte Enable Signal	Associated Data Bus Signals	
BE0#	D7–D0	(byte 0–least significant)
BE1#	D15–D8	(byte 1)
BE2#	D23–D16	(byte 2)
BE3#	D31–D24	(byte 3–most significant)

Address bits A0 and A1 of the physical operand's base address can be created when necessary. Use of the byte enables to create A0 and A1 is shown in Table 4-2. The byte enables can also be decoded to generate BLE# (byte low enable) and BHE# (byte high enable). These signals are needed to address 16-bit memory systems. (See Section 4.1.3, "Interfacing with 8-, 16-, and 32-Bit Memories.")

4.1.1 Memory and I/O Spaces

Bus cycles may access physical memory space or I/O space. Peripheral devices in the system can be either memory-mapped, I/O-mapped, or both. Physical memory addresses range from

00000000H to FFFFFFFFH (4 gigabytes). I/O addresses range from 00000000H to 0000FFFFH (64 Kbytes) for programmed I/O. (See Figure 4-1.)

Table 4-2. Generating A31–A0 from BE3#–BE0# and A31–A2

Intel486™ Processor Address Signals								
A31	through	A2			BE3#	BE2#	BE1#	BE0#
Physical Address								
A31	...	A2	A1	A0				
A31	...	A2	0	0	X	X	X	0
A31	...	A2	0	1	X	X	0	1
A31	...	A2	1	0	X	0	1	1
A31	...	A2	1	1	0	1	1	1

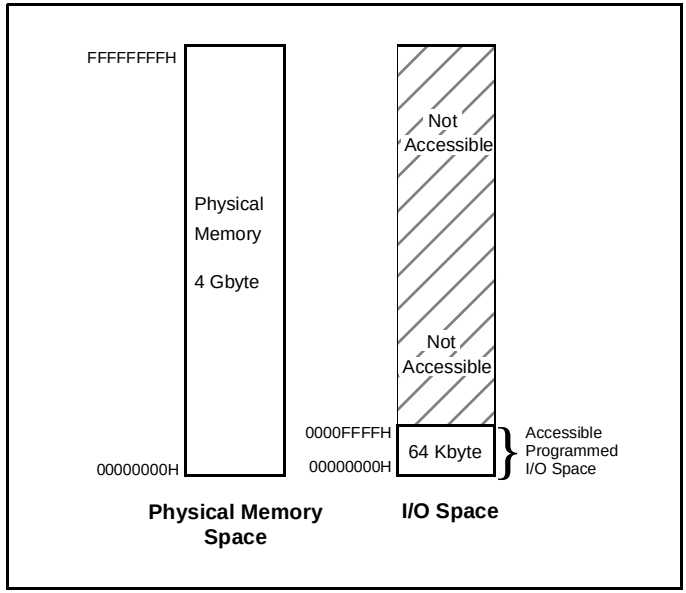


Figure 4-1. Physical Memory and I/O Spaces

4.1.1.1 Memory and I/O Space Organization

The Intel486 processor datapath to memory and input/output (I/O) spaces can be 32, 16, or 8 bits wide. The byte enable signals, BE3#–BE0#, allow byte granularity when addressing any memory or I/O structure, whether 8, 16, or 32 bits wide.

The Intel486 processor includes bus control pins, BS16# and BS8#, which allow direct connection to 16- and 8-bit memories and I/O devices. Cycles of 32-, 16- and 8-bits may occur in any sequence, since the BS8# and BS16# signals are sampled during each bus cycle.

NOTE

The Ultra-Low Power Intel486 GX processor has a 16-bit external data bus. All data transfers are done on the low order data bits (D15-D0) and parity is generated and checked on pins DP0 and DP1. For this reason, dynamic data bus sizing (using pins BS16# and BS8#) is not supported.

Memory and I/O spaces that are 32-bit wide are organized as arrays of four bytes each. Each four bytes consists of four individually addressable bytes at consecutive byte addresses (see Figure 4-2). The lowest addressed byte is associated with data signals D7–D0; the highest-addressed byte with D31–D24. Each 4 bytes begin at an address that is divisible by four.

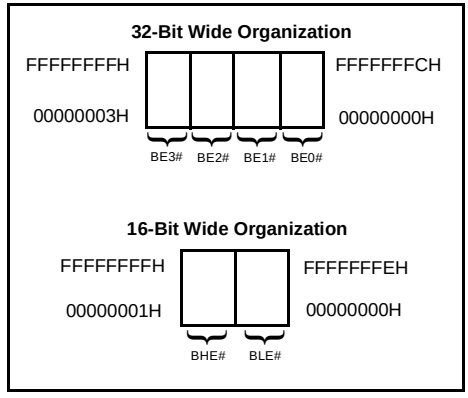


Figure 4-2. Physical Memory and I/O Space Organization

16-bit memories are organized as arrays of two bytes each. Each two bytes begins at addresses divisible by two. The byte enables BE3#–BE0#, must be decoded to A1, BLE# and BHE# to address 16-bit memories.

To address 8-bit memories, the two low order address bits A0 and A1 must be decoded from BE3#–BE0#. The same logic can be used for 8- and 16-bit memories, because the decoding logic for BLE# and A0 are the same. (See Section 4.1.3, “Interfacing with 8-, 16-, and 32-Bit Memories.”)

4.1.2 Dynamic Data Bus Sizing

Dynamic data bus sizing is a feature that allows processor connection to 32-, 16- or 8-bit buses for memory or I/O. The Intel486 processors can connect to all three bus sizes, except for the Ultra-Low Power Intel486 GX processor, uses a 16-bit data bus. Transfers to or from 32-, 16- or 8-bit devices are supported by dynamically determining the bus width during each bus cycle. Address decoding circuitry may assert BS16# for 16-bit devices or BS8# for 8-bit devices during

each bus cycle. BS8# and BS16# must be deasserted when addressing 32-bit devices. An 8-bit bus width is selected if both BS16# and BS8# are asserted.

BS16# and BS8# force the Intel486 processor to run additional bus cycles to complete requests larger than 16 or 8 bits. A 32-bit transfer is converted into two 16-bit transfers (or 3 transfers if the data is misaligned) when BS16# is asserted. Asserting BS8# converts a 32-bit transfer into four 8-bit transfers.

Extra cycles forced by BS16# or BS8# should be viewed as independent bus cycles. BS16# or BS8# must be asserted during each of the extra cycles unless the addressed device has the ability to change the number of bytes it can return between cycles.

The Intel486 processor drives the byte enables appropriately during extra cycles forced by BS8# and BS16#. A31–A2 does not change if accesses are to a 32-bit aligned area. Table 4-3 shows the set of byte enables that is generated on the next cycle for each of the valid possibilities of the byte enables on the current cycle.

The dynamic bus sizing feature of the Intel486 processor is significantly different than that of the Intel386™ processor. Unlike the Intel386 processor, the Intel486 processor requires that data bytes be driven on the addressed data pins. The simplest example of this function is a 32-bit aligned, BS16# read. When the Intel486 processor reads the two high order bytes, they must be driven on the data bus pins D31–D16. The Intel486 processor expects the two low order bytes on D15–D0. The Intel386 processor expects both the high and low order bytes on D15–D0. The Intel386 processor always reads or writes data on the lower 16 bits of the data bus when BS16# is asserted.

The external system must contain buffers to enable the Intel486 processor to read and write data on the appropriate data bus pins. Table 4-4 shows the data bus lines to which the Intel486 processor expects data to be returned for each valid combination of byte enables and bus sizing options.

Table 4-3. Next Byte Enable Values for BS*x*# Cycles

Current				Next with				Next with BS16#			
BE3#	BE2#	BE1#	BE0#	BE3#	BE2#	BE1#	BE0#	BE3#	BE2#	BE1#	BE0#
1	1	1	0	N	N	N	N	N	N	N	N
1	1	0	0	1	1	0	1	N	N	N	N
1	0	0	0	1	0	0	1	1	0	1	1
0	0	0	0	0	0	0	1	0	0	1	1
1	1	0	1	N	N	N	N	N	N	N	N
1	0	0	1	1	0	1	1	1	0	1	1
0	0	0	1	0	0	1	1	0	0	1	1
1	0	1	1	N	N	N	N	N	N	N	N
0	0	1	1	0	1	1	1	N	N	N	N
0	1	1	1	N	N	N	N	N	N	N	N

NOTE: "N" means that another bus cycle is not required to satisfy the request.

Table 4-4. Data Pins Read with Different Bus Sizes

BE3#	BE2#	BE1#	BE0#	w/o BS8#/BS16#	w BS8#	w BS16#
1	1	1	0	D7–D0	D7–D0	D7–D0
1	1	0	0	D15–D0	D7–D0	D15–D0
1	0	0	0	D23–D0	D7–D0	D15–D0
0	0	0	0	D31–D0	D7–D0	D15–D0
1	1	0	1	D15–D8	D15–D8	D15–D8
1	0	0	1	D23–D8	D15–D8	D15–D8
0	0	0	1	D31–D8	D15–D8	D15–D8
1	0	1	1	D23–D16	D23–D16	D23–D16
0	0	1	1	D31–D16	D23–D16	D31–D16
0	1	1	1	D31–D24	D31–D24	D31–D24

Valid data is only driven onto data bus pins corresponding to asserted byte enables during write cycles. Other pins in the data bus are driven but they contain no valid data. Unlike the Intel386 processor, the Intel486 processor does not duplicate write data onto parts of the data bus for which the corresponding byte enable is deasserted.

4.1.3 Interfacing with 8-, 16-, and 32-Bit Memories

In 32-bit physical memories, such as the one shown in Figure 4-3, each 4-byte word begins at a byte address that is a multiple of four. A31–A2 are used as a 4-byte word select. BE3#–BE0# select individual bytes within the 4-byte word. BS8# and BS16# are deasserted for all bus cycles involving the 32-bit array.

For 16- and 8-bit memories, byte swapping logic is required for routing data to the appropriate data lines and logic is required for generating BHE#, BLE# and A1. In systems where mixed memory widths are used, extra address decoding logic is necessary to assert BS16# or BS8#.

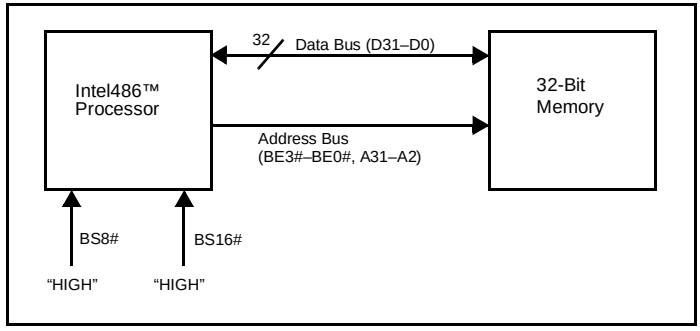


Figure 4-3. Intel486™ Processor with 32-Bit Memory

Figure 4-4 shows the Intel486 processor address bus interface to 32-, 16- and 8-bit memories. To address 16-bit memories the byte enables must be decoded to produce A1, BHE# and BLE# (A0). For 8-bit wide memories the byte enables must be decoded to produce A0 and A1. The same byte select logic can be used in 16- and 8-bit systems, because BLE# is exactly the same as A0 (see Table 4-5).

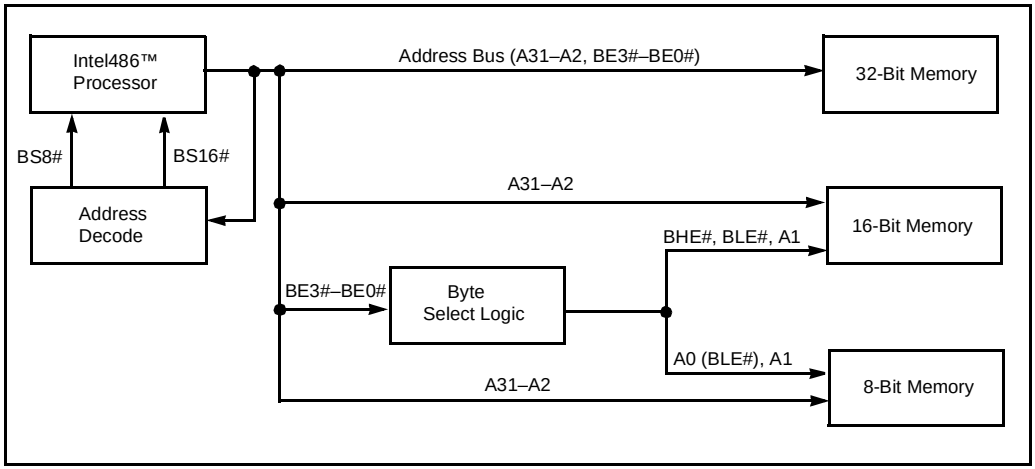


Figure 4-4. Addressing 16- and 8-Bit Memories

BE3#–BE0# can be decoded as shown in Table 4-5. The byte select logic necessary to generate BHE# and BLE# is shown in Figure 4-5.

Table 4-5. Generating A1, BHE# and BLE# for Addressing 16-Bit Devices

Intel486™ Processor				8-, 16-Bit Bus Signals			Comments
BE3#	BE2#	BE1#	BE0#	A1[3]	BHE#[2]	BLE# (A0)[1]	
1†	1†	1†	1†	x	x	x	x–no asserted bytes
1	1	1	0	0	1	0	
1	1	0	1	0	0	1	
1	1	0	0	0	0	0	
1	0	1	1	1	1	0	
1†	0†	1†	0†	x	x	x	x–not contiguous bytes
1	0	0	1	0	0	1	
1	0	0	0	0	0	0	
0	1	1	1	1	0	1	
0†	1†	1†	0†	x	x	x	x–not contiguous bytes
0†	1†	0†	1†	x	x	x	x–not contiguous bytes
0†	1†	0†	0†	x	x	x	x–not contiguous bytes
0		1	1	1	0	0	
0†	0†	1†	0†	x	x	x	x–not contiguous bytes
0	0	0	1	0	0	1	
0	0	0	0	0	0	0	

NOTES:
1. BLE# asserted when D7–D0 of 16-bit bus is asserted.
2. BHE# asserted when D15–D8 of 16-bit bus is asserted.
3. A1 low for all even words; A1 high for all odd words.

KEY:

x = don't care

† = a non-occurring pattern of byte enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes

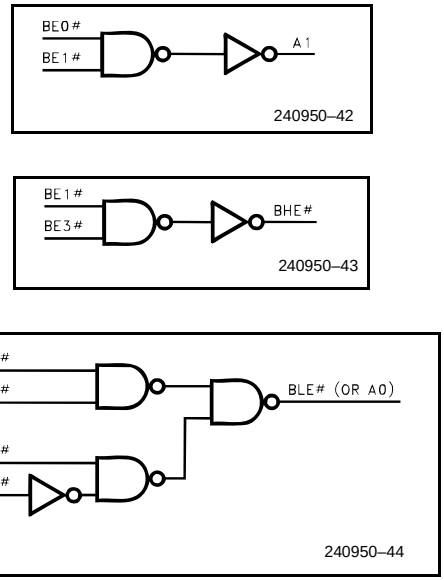


Figure 4-5. Logic to Generate A1, BHE# and BLE# for 16-Bit Buses

Combinations of BE3#–BE0# that never occur are those in which two or three asserted byte enables are separated by one or more deasserted byte enables. These combinations are “don't care” conditions in the decoder. A decoder can use the non-occurring BE3#–BE0# combinations to its best advantage.

Figure 4-6 shows an Intel486 processor data bus interface to 16- and 8-bit wide memories. External byte swapping logic is needed on the data lines so that data is supplied to and received from the Intel486 processor on the correct data pins (see Table 4-4).

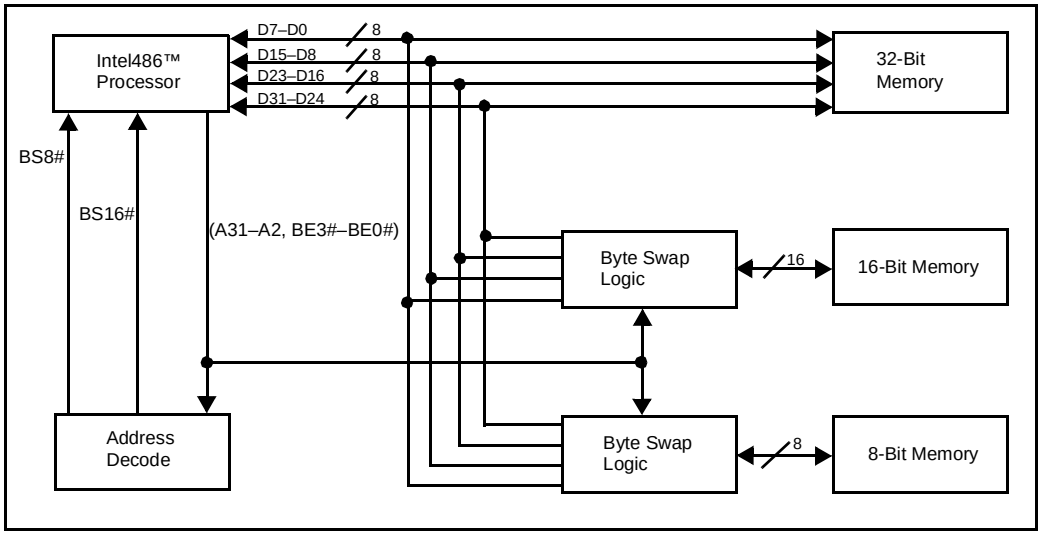


Figure 4-6. Data Bus Interface to 16- and 8-Bit Memories

4.1.4 Dynamic Bus Sizing During Cache Line Fills

BS8# and BS16# can be driven during cache line fills. The Intel486 processor generates enough 8- or 16-bit cycles to fill the cache line. This can be up to sixteen 8-bit cycles.

The external system should assume that all byte enables are asserted for the first cycle of a cache line fill. The Intel486 processor generates proper byte enables for subsequent cycles in the line fill. Table 4-6 shows the appropriate A0 (BLE#), A1 and BHE# for the various combinations of the Intel486 processor byte enables on both the first and subsequent cycles of the cache line fill. The "†" marks all combinations of byte enables that are generated by the Intel486 processor during a cache line fill.

Table 4-6. Generating A0, A1 and BHE# from the Intel486™ Processor Byte Enables

BE3#	BE2#	BE1#	BE0#	First Cache Fill Cycle			Any Other Cycle		
				A0	A1	BHE#	A0	A1	BHE#
1	1	1	0	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
†0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	1	0	0
1	0	0	1	0	0	0	1	0	0
†0	0	0	1	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	1
†0	0	1	1	0	0	0	0	1	0
†0	1	1	1	0	0	0	1	1	0

KEY:

† = a non-occurring pattern of Byte Enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes

4.1.5 Operand Alignment

Physical 4-byte words begin at addresses that are multiples of four. It is possible to transfer a logical operand that spans more than one physical 4-byte word of memory or I/O at the expense of extra cycles. Examples are 4-byte operands beginning at addresses that are not evenly divisible by 4, or 2-byte words split between two physical 4-byte words. These are referred to as unaligned transfers.

Operand alignment and data bus size dictate when multiple bus cycles are required. Table 4-7 describes the transfer cycles generated for all combinations of logical operand lengths, alignment, and data bus sizing. When multiple cycles are required to transfer a multibyte logical operand, the highest-order bytes are transferred first. For example, when the processor executes a 4-byte unaligned read beginning at byte location 11 in the 4-byte aligned space, the three high-order bytes are read in the first bus cycle. The low byte is read in a subsequent bus cycle.

Table 4-7. Transfer Bus Cycles for Bytes, Words and Dwords

	Byte-Length of Logical Operand								
	1	2				4			
Physical Byte Address in Memory (Low Order Bits)	xx	00	01	10	11	00	01	10	11
Transfer Cycles over 32-Bit Bus	b	w	w	w	hb lb	d	hb l3	hw lw	h3 lb
Transfer Cycles over 16-Bit Bus († = BS#16 asserted)	b	w	lb † hb †	w	hb lb	lw † hw †	hb lb † mw †	hw lw	mw † hb † lb
Transfer Cycles over 8-Bit Bus (‡ = BS8# Asserted)	b	lb ‡ hb ‡	lb ‡ hb‡	lb ‡ hb ‡	hb lb	lb ‡ mlb ‡ mhb ‡ hb ‡	hb lb ‡ mlb ‡ mhb ‡	mhb ‡ hb ‡ lb ‡ mlb ‡	mlb ‡ mhb ‡ hb ‡ lb

KEY:

b = byte transfer
w = 2-byte transfer
3 = 3-byte transfer
d = 4-byte transfer

h = high-order portion
l = low-order portion
m = mid-order portion

4-Byte Operand

lb	mlb	mhb	hb

↑ byte with lowest address

↑byte with highest address

The function of unaligned transfers with dynamic bus sizing is not obvious. When the external systems asserts BS16# or BS8#, forcing extra cycles, low-order bytes or words are transferred first (opposite to the example above). When the Intel486 processor requests a 4-byte read and the external system asserts BS16#, the lower two bytes are read first followed by the upper two bytes.

In the unaligned transfer described above, the processor requested three bytes on the first cycle. When the external system asserts BS16# during this 3-byte transfer, the lower word is transferred first followed by the upper byte. In the final cycle, the lower byte of the 4-byte operand is transferred, as shown in the 32-bit example above.

4.2 BUS ARBITRATION LOGIC

Bus arbitration logic is needed with multiple bus masters. Hardware implementations range from single-master designs to those with multiple masters and DMA devices.

Figure 4-7 shows a simple system in which only one master controls the bus and accesses the memory and I/O devices. Here, no arbitration is required.

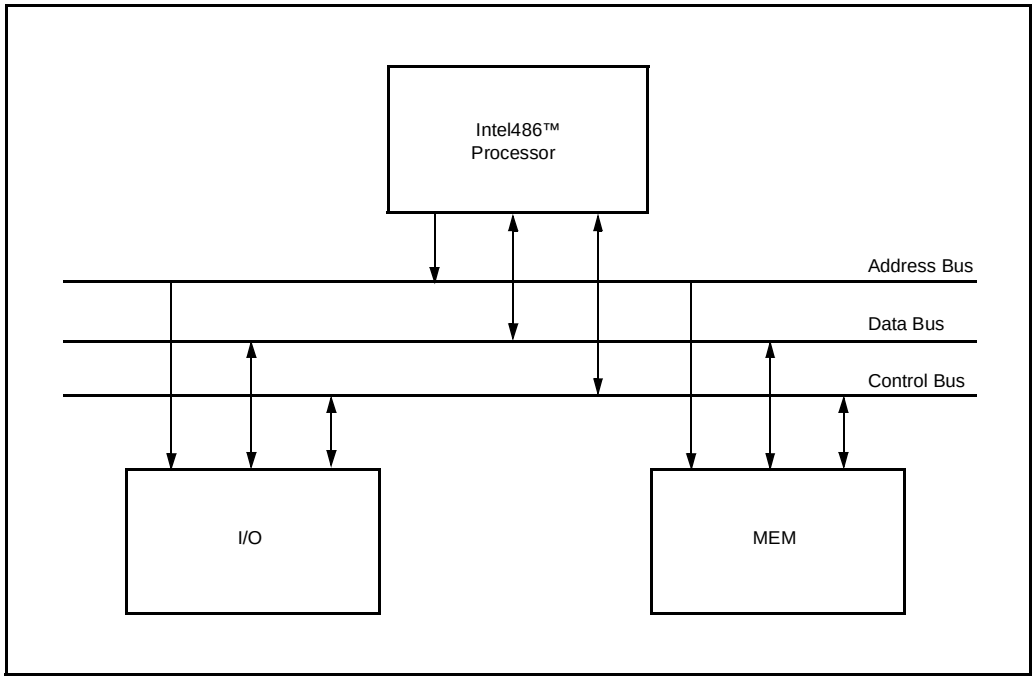


Figure 4-7. Single Master Intel486™ Processor System

Figure 4-8 shows a single processor and a DMA device. Here, arbitration is required to determine whether the processor, which acts as a master most of the time, or a DMA controller has control of the bus. When the DMA wants control of the bus, it asserts the HOLD request to the processor. The processor then responds with a HLDA output when it is ready to relinquish bus control to the DMA device. Once the DMA device completes its bus activity cycles, it negates the HOLD signal to relinquish the bus and return control to the processor.

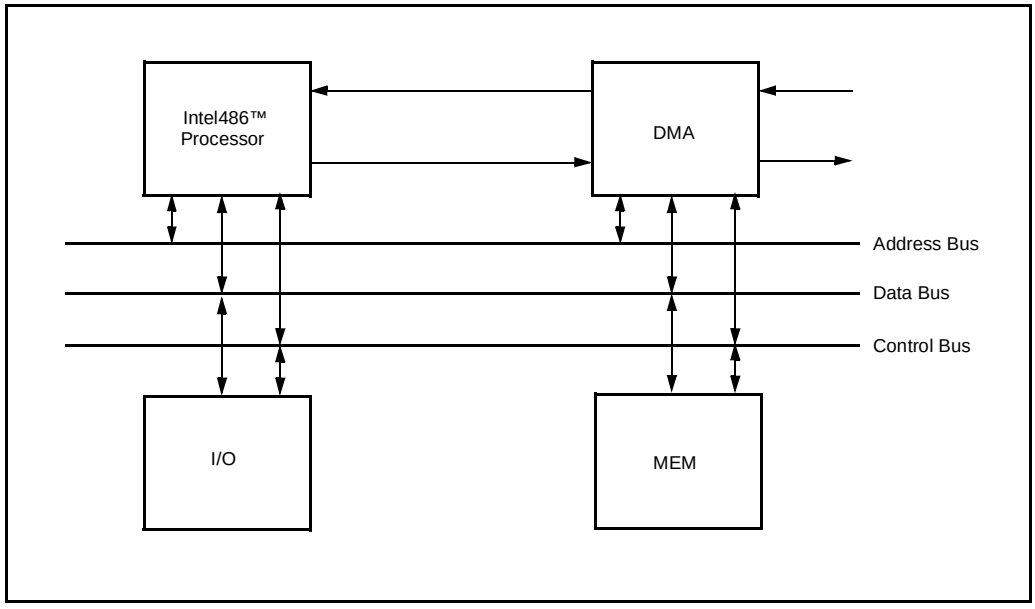


Figure 4-8. Single Intel486™ Processor with DMA

Figure 4-9 shows more than one primary bus master and two secondary masters, and the arbitration logic is more complex. The arbitration logic resolves bus contention by ensuring that all device requests are serviced one at a time using either a fixed or a rotating scheme. The arbitration logic then passes information to the Intel486 processor, which ultimately releases the bus. The arbitration logic receives bus control status information via the HOLD and HLDA signals and relays it to the requesting devices.

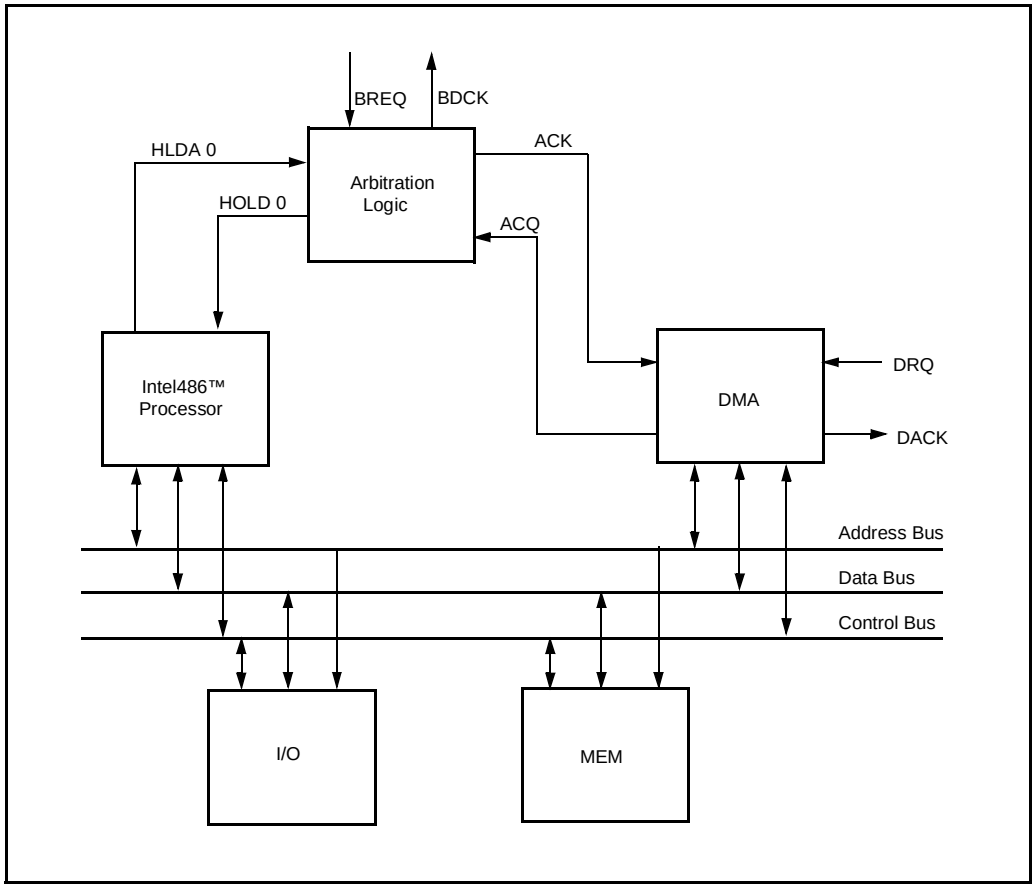


Figure 4-9. Single Intel486™ Processor with Multiple Secondary Masters

As systems become more complex and include multiple bus masters, hardware must be added to arbitrate and assign the management of bus time to each master. The second master may be a DMA controller that requires bus time to perform memory transfers or it may be a second processor that requires the bus to perform memory or I/O cycles. Any of these devices may act as a bus master. The arbitration logic must assign only one bus master at a time so that there is no contention between devices when accessing main memory.

The arbitration logic may be implemented in several different ways. The first technique is to "round-robin" or to "time slice" each master. Each master is given a block of time on the bus to match their priority and need for the bus.

Another method of arbitration is to assign the bus to a master when the bus is needed. Assigning the bus requires the arbitration logic to sample the BREQ or HOLD outputs from the potential masters and to assign the bus to the requestor. A priority scheme must be included to handle cases where more than one device is requesting the bus. The arbitration logic must assert HOLD to the device that must relinquish the bus. Once HLDA is asserted by all of these devices, the arbitration logic may assert HLDA or BACK# to the device requesting the bus. The requestor remains the bus master until another device needs the bus.

These two arbitration techniques can be combined to create a more elaborate arbitration scheme that is driven by a device that needs the bus but guarantees that every device gets time on the bus. It is important that an arbitration scheme be selected to best fit the needs of each system's implementation.

The Intel486 processor asserts BREQ when it requires control of the bus. BREQ notifies the arbitration logic that the processor has pending bus activity and requests the bus. When its HOLD input is inactive and its HLDA signal is deasserted, the Intel486 processor can acquire the bus. Otherwise if HOLD is asserted, then the Intel486 processor has to wait for HOLD to be deasserted before acquiring the bus. If the Intel486 processor does not have the bus, then its address, data, and status pins are 3-stated. However, the processor can execute instructions out of the internal cache or instruction queue, and does not need control of the bus to remain active.

The address buses shown in Figure 4-8 and Figure 4-9 are bidirectional to allow cache invalidations to the processors during memory writes on the bus.

4.3 BUS FUNCTIONAL DESCRIPTION

The Intel486 processor supports a wide variety of bus transfers to meet the needs of high performance systems. Bus transfers can be single cycle or multiple cycle, burst or non-burst, cacheable or non-cacheable, 8-, 16- or 32-bit, and pseudo-locked. Cache invalidation cycles and locked cycles provide support for multiprocessor systems.

This section explains basic non-cacheable, non-burst single cycle transfers. It also details multiple cycle transfers and introduces the burst mode. Cacheability is introduced in Section 4.3.3, "Cacheable Cycles." The remaining sections describe locked, pseudo-locked, invalidate, bus hold, and interrupt cycles.

Bus cycles and data cycles are discussed in this section. A bus cycle is at least two clocks long and begins with ADS# asserted in the first clock and RDY# or BRDY# asserted in the last clock. Data is transferred to or from the Intel486 processor during a data cycle. A bus cycle contains one or more data cycles.

Refer to Section 4.3.13, "Bus States," for a description of the bus states shown in the timing diagrams.

4.3.1 Non-Cacheable Non-Burst Single Cycle

4.3.1.1 No Wait States

The fastest non-burst bus cycle that the Intel486 processor supports is two clocks. These cycles are called 2-2 cycles because reads and writes take two cycles each. The first “2” refers to reads and the second “2” to writes. If a wait state needs to be added to the write, the cycle is called “2-3.”

Basic two-clock read and write cycles are shown in Figure 4-10. The Intel486 processor initiates a cycle by asserting the address status signal (ADS#) at the rising edge of the first clock. The ADS# output indicates that a valid bus cycle definition and address is available on the cycle definition lines and address bus.

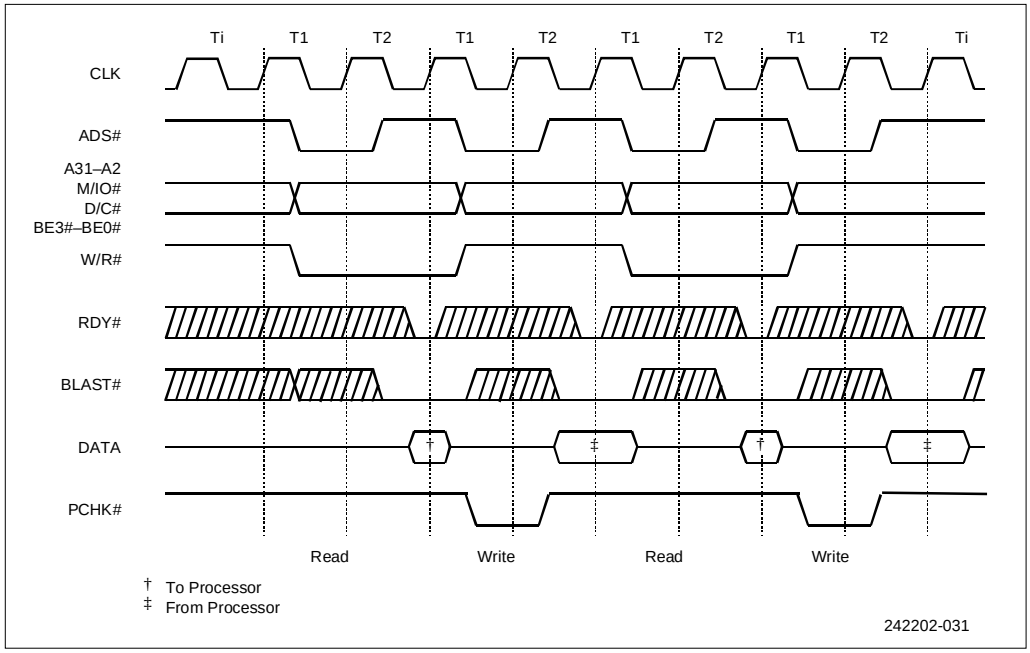


Figure 4-10. Basic 2-2 Bus Cycle

The non-burst ready input (RDY#) is asserted by the external system in the second clock. RDY# indicates that the external system has presented valid data on the data pins in response to a read or the external system has accepted data in response to a write.

The Intel486 processor samples RDY# at the end of the second clock. The cycle is complete if RDY# is asserted (LOW) when sampled. Note that RDY# is ignored at the end of the first clock of the bus cycle.

The burst last signal (BLAST#) is asserted (LOW) by the Intel486 processor during the second clock of the first cycle in all bus transfers illustrated in Figure 4-10. This indicates that each trans-

fer is complete after a single cycle. The Intel486 processor asserts BLAST# in the last cycle, “T2”, of a bus transfer.

The timing of the parity check output (PCHK#) is shown in Figure 4-10. The Intel486 processor drives the PCHK# output one clock after RDY# or BRDY# terminates a read cycle. PCHK# indicates the parity status for the data sampled at the end of the previous clock. The PCHK# signal can be used by the external system. The Intel486 processor does nothing in response to the PCHK# output.

4.3.1.2 Inserting Wait States

The external system can insert wait states into the basic 2-2 cycle by deasserting RDY# at the end of the second clock. RDY# must be deasserted to insert a wait state. Figure 4-11 illustrates a simple non-burst, non-cacheable signal with one wait state added. Any number of wait states can be added to an Intel486 processor bus cycle by maintaining RDY# deasserted.

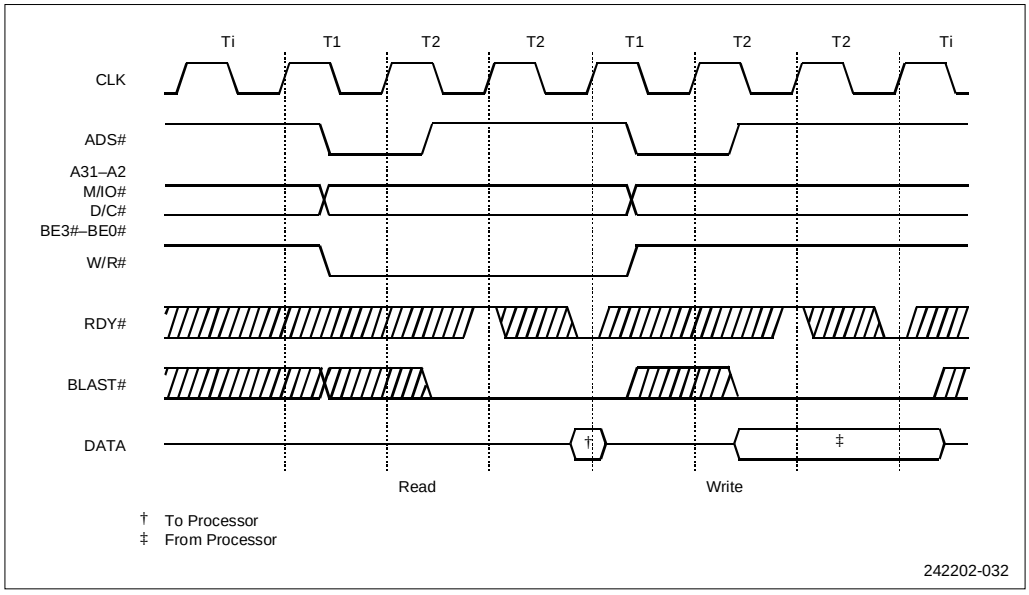


Figure 4-11. Basic 3-3 Bus Cycle

The burst ready input (BRDY#) must be deasserted on all clock edges where RDY# is deasserted for proper operation of these simple non-burst cycles.

4.3.2 Multiple and Burst Cycle Bus Transfers

Multiple cycle bus transfers can be caused by internal requests from the Intel486 processor or by the external memory system. An internal request for a 128-bit pre-fetch requires more than one cycle. Internal requests for unaligned data may also require multiple bus cycles. A cache line fill requires multiple cycles to complete.

The external system can cause a multiple cycle transfer when it can only supply 8- or 16-bits per cycle.

Only multiple cycle transfers caused by internal requests are considered in this section. Cacheable cycles and 8- and 16-bit transfers are covered in Section 4.3.3, “Cacheable Cycles,” and Section 4.3.5, “8- and 16-Bit Cycles.”

Internal Requests from IntelDX2 and IntelDX4 Processors

An internal request by an IntelDX2 or IntelDX4 processor for a 64-bit floating-point load must take more than one internal cycle.

4.3.2.1 Burst Cycles

The Intel486 processor can accept burst cycles for any bus requests that require more than a single data cycle. During burst cycles, a new data item is strobed into the Intel486 processor every clock rather than every other clock as in non-burst cycles. The fastest burst cycle requires two clocks for the first data item, with subsequent data items returned every clock.

The Intel486 processor is capable of bursting a maximum of 32 bits during a write. Burst writes can only occur if BS8# or BS16# is asserted. For example, the Intel486 processor can burst write four 8-bit operands or two 16-bit operands in a single burst cycle. But the Intel486 processor cannot burst multiple 32-bit writes in a single burst cycle.

Burst cycles begin with the Intel486 processor driving out an address and asserting ADS# in the same manner as non-burst cycles. The Intel486 processor indicates that it is willing to perform a burst cycle by holding the burst last signal (BLAST#) deasserted in the second clock of the cycle. The external system indicates its willingness to do a burst cycle by asserting the burst ready signal (BRDY#).

The addresses of the data items in a burst cycle all fall within the same 16-byte aligned area (corresponding to an internal Intel486 processor cache line). A 16-byte aligned area begins at location XXXXXXX0 and ends at location XXXXXXXF. During a burst cycle, only BE3#–BE0#, A2, and A3 may change. A31–A4, M/IO#, D/C#, and W/R# remain stable throughout a burst. Given the first address in a burst, external hardware can easily calculate the address of subsequent transfers in advance. An external memory system can be designed to quickly fill the Intel486 processor internal cache lines.

Burst cycles are not limited to cache line fills. Any multiple cycle read request by the Intel486 processor can be converted into a burst cycle. The Intel486 processor only bursts the number of bytes needed to complete a transfer. For example, the IntelDX2 and Write-Back Enhanced IntelDX4 processors burst eight bytes for a 64-bit floating-point non-cacheable read.

The external system converts a multiple cycle request into a burst cycle by asserting BRDY# rather than RDY# (non-burst ready) in the first cycle of a transfer. For cycles that cannot be burst, such as interrupt acknowledge and halt, BRDY# has the same effect as RDY#. BRDY# is ignored if both BRDY# and RDY# are asserted in the same clock. Memory areas and peripheral devices that cannot perform bursting must terminate cycles with RDY#.

4.3.2.2 Terminating Multiple and Burst Cycle Transfers

The Intel486 processor deasserts BLAST# for all but the last cycle in a multiple cycle transfer. BLAST# is deasserted in the first cycle to inform the external system that the transfer could take additional cycles. BLAST# is asserted in the last cycle of the transfer to indicate that the next time BRDY# or RDY# is asserted the transfer is complete.

BLAST# is not valid in the first clock of a bus cycle. It should be sampled only in the second and subsequent clocks when RDY# or BRDY# is asserted.

The number of cycles in a transfer is a function of several factors including the number of bytes the Intel486 processor needs to complete an internal request (1, 2, 4, 8, or 16), the state of the bus size inputs (BS8# and BS16#), the state of the cache enable input (KEN#) and the alignment of the data to be transferred.

When the Intel486 processor initiates a request, it knows how many bytes are transferred and if the data is aligned. The external system must indicate whether the data is cacheable (if the transfer is a read) and the width of the bus by returning the state of the KEN#, BS8# and BS16# inputs one clock before RDY# or BRDY# is asserted. The Intel486 processor determines how many cycles a transfer will take based on its internal information and inputs from the external system.

BLAST# is not valid in the first clock of a bus cycle because the Intel486 processor cannot determine the number of cycles a transfer will take until the external system asserts KEN#, BS8# and BS16#. BLAST# should only be sampled in the second T2 state and subsequent T2 states of a cycle when the external system asserts RDY# or BRDY#.

The system may terminate a burst cycle by asserting RDY# instead of BRDY#. BLAST# remains deasserted until the last transfer. However, any transfers required to complete a cache line fill follow the burst order; for example, if burst order was 4, 0, C, 8 and RDY# was asserted after 0, the next transfers are from C and 8.

4.3.2.3 Non-Cacheable, Non-Burst, Multiple Cycle Transfers

Figure 4-12 illustrates a two-cycle, non-burst, non-cacheable read. This transfer is simply a sequence of two single cycle transfers. The Intel486 processor indicates to the external system that this is a multiple cycle transfer by deasserting BLAST# during the second clock of the first cycle. The external system asserts RDY# to indicate that it will not burst the data. The external system also indicates that the data is not cacheable by deasserting KEN# one clock before it asserts RDY#. When the Intel486 processor samples RDY# asserted, it ignores BRDY#.

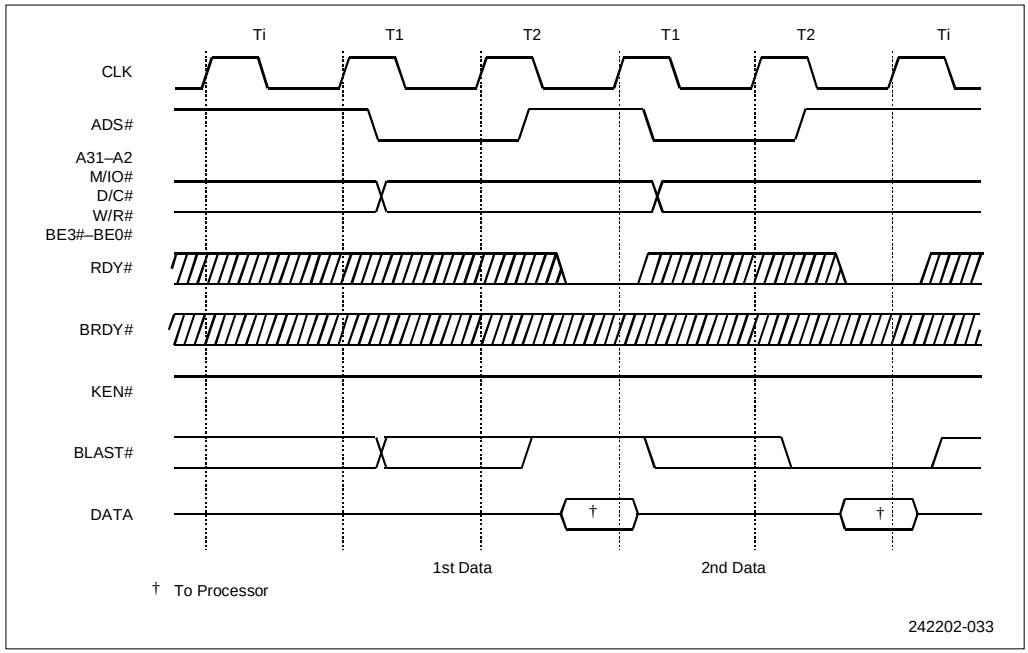


Figure 4-12. Non-Cacheable, Non-Burst, Multiple-Cycle Transfers

Each cycle in the transfer begins when ADS# is asserted and the cycle is complete when the external system asserts RDY#.

The Intel486 processor indicates the last cycle of the transfer by asserting BLAST#. The next RDY# asserted by the external system terminates the transfer.

4.3.2.4 Non-Cacheable Burst Cycles

The external system converts a multiple cycle request into a burst cycle by asserting BRDY# rather than RDY# in the first cycle of the transfer. This is illustrated in Figure 4-13.

There are several features to note in the burst read. ADS# is asserted only during the first cycle of the transfer. RDY# must be deasserted when BRDY# is asserted.

BLAST# behaves exactly as it does in the non-burst read. BLAST# is deasserted in the second clock of the first cycle of the transfer, indicating more cycles to follow. In the last cycle, BLAST# is asserted, prompting the external memory system to end the burst after asserting the next BRDY#.

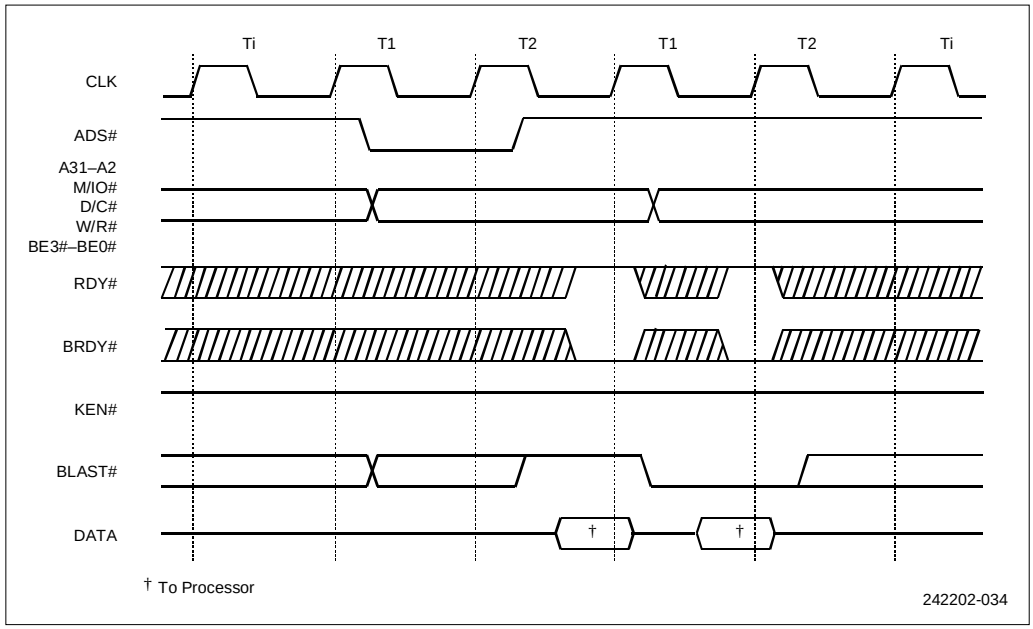


Figure 4-13. Non-Cacheable Burst Cycle

4.3.3 Cacheable Cycles

Any memory read can become a cache fill operation. The external memory system can allow a read request to fill a cache line by asserting KEN# one clock before RDY# or BRDY# during the first cycle of the transfer on the external bus. Once KEN# is asserted and the remaining three requirements described below are met, the Intel486 processor fetches an entire cache line regardless of the state of KEN#. KEN# must be asserted in the last cycle of the transfer for the data to be written into the internal cache. The Intel486 processor converts only memory reads or prefetches into a cache fill.

KEN# is ignored during write or I/O cycles. Memory writes are stored only in the on-chip cache if there is a cache hit. I/O space is never cached in the internal cache.

To transform a read or a prefetch into a cache line fill, the following conditions must be met:

1. The KEN# pin must be asserted one clock prior to RDY# or BRDY# being asserted for the first data cycle.
2. The cycle must be of a type that can be internally cached. (Locked reads, I/O reads, and interrupt acknowledge cycles are never cached.)
3. The page table entry must have the page cache disable bit (PCD) set to 0. To cache a page table entry, the page directory must have PCD=0. To cache reads or prefetches when

paging is disabled, or to cache the page directory entry, control register 3 (CR3) must have PCD=0.

4. The cache disable (CD) bit in control register 0 (CR0) must be clear.

External hardware can determine when the Intel486 processor has transformed a read or prefetch into a cache fill by examining the KEN#, M/IO#, D/C#, W/R#, LOCK#, and PCD pins. These pins convey to the system the outcome of conditions 1–3 in the above list. In addition, the Intel486 processor drives PCD high whenever the CD bit in CR0 is set, so that external hardware can evaluate condition 4.

Cacheable cycles can be burst or non-burst.

4.3.3.1 Byte Enables during a Cache Line Fill

For the first cycle in the line fill, the state of the byte enables should be ignored. In a non-cacheable memory read, the byte enables indicate the bytes actually required by the memory or code fetch.

The Intel486 processor expects to receive valid data on its entire bus (32 bits) in the first cycle of a cache line fill. Data should be returned with the assumption that all the byte enable pins are asserted. However if BS8# is asserted, only one byte should be returned on data lines D7–D0. Similarly if BS16# is asserted, two bytes should be returned on D15–D0.

The Intel486 processor generates the addresses and byte enables for all subsequent cycles in the line fill. The order in which data is read during a line fill depends on the address of the first item read. Byte ordering is discussed in Section 4.3.4, “Burst Mode Details.”

4.3.3.2 Non-Burst Cacheable Cycles

Figure 4-14 shows a non-burst cacheable cycle. The cycle becomes a cache fill when the Intel486 processor samples KEN# asserted at the end of the first clock. The Intel486 processor deasserts BLAST# in the second clock in response to KEN#. BLAST# is deasserted because a cache fill requires three additional cycles to complete. BLAST# remains deasserted until the last transfer in the cache line fill. KEN# must be asserted in the last cycle of the transfer for the data to be written into the internal cache.

Note that this cycle would be a single bus cycle if KEN# was not sampled asserted at the end of the first clock. The subsequent three reads would not have happened since a cache fill was not requested.

The BLAST# output is invalid in the first clock of a cycle. BLAST# may be asserted during the first clock due to earlier inputs. Ignore BLAST# until the second clock.

During the first cycle of the cache line fill the external system should treat the byte enables as if they are all asserted. In subsequent cycles in the burst, the Intel486 processor drives the address lines and byte enables. (See Section 4.3.4.2, “Burst and Cache Line Fill Order.”)

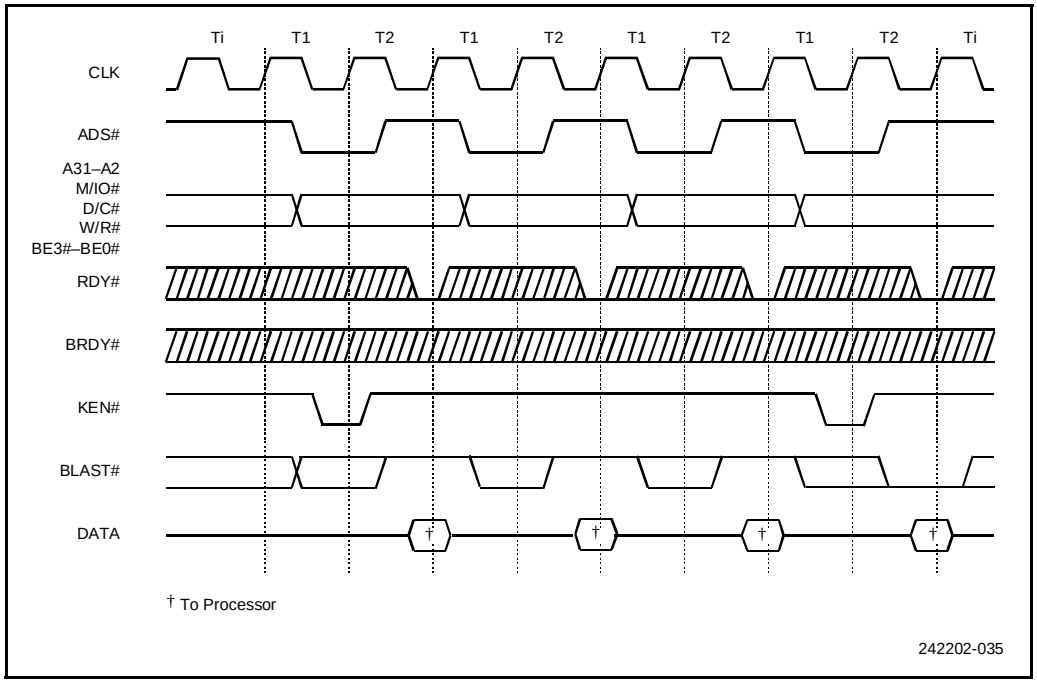


Figure 4-14. Non-Burst, Cacheable Cycles

4.3.3.3 Burst Cacheable Cycles

Figure 4-15 illustrates a burst mode cache fill. As in Figure 4-14, the transfer becomes a cache line fill when the external system asserts KEN# at the end of the first clock in the cycle.

The external system informs the Intel486 processor that it will burst the line in by asserting BRDY# at the end of the first cycle in the transfer.

Note that during a burst cycle, ADS# is only driven with the first address.

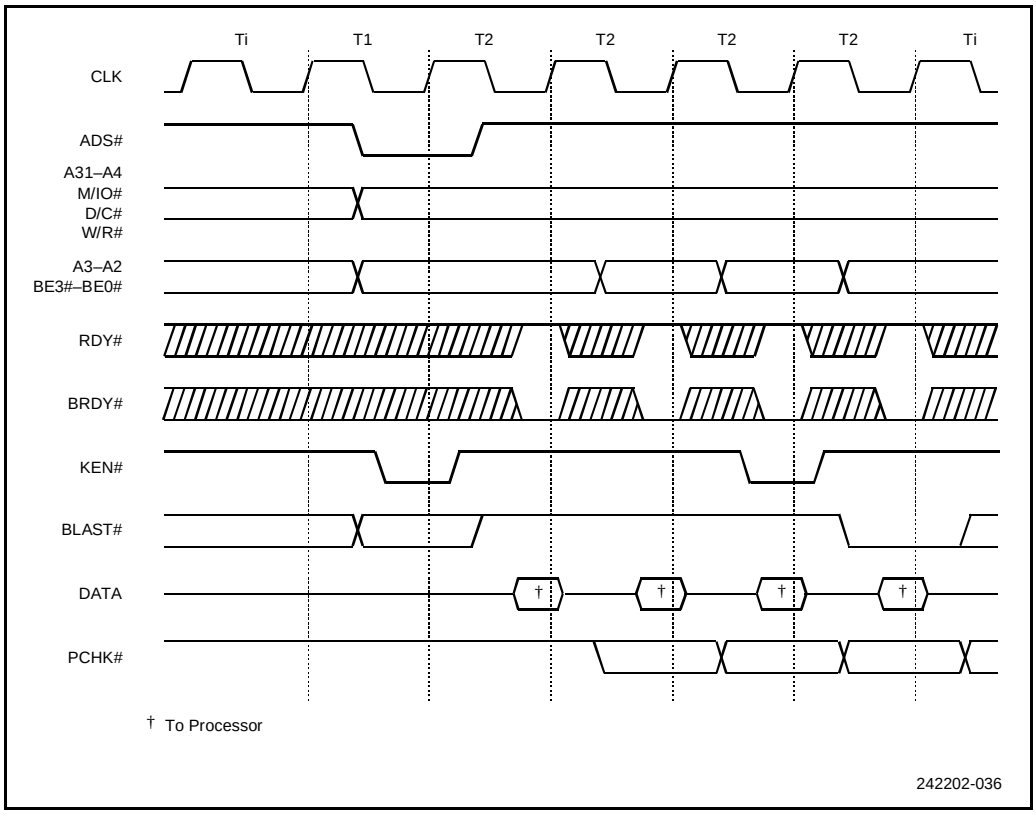


Figure 4-15. Burst Cacheable Cycle

4.3.3.4 Effect of Changing KEN# during a Cache Line Fill

KEN# can change multiple times as long as it arrives at its final value in the clock before RDY# or BRDY# is asserted. This is illustrated in Figure 4-16. Note that the timing of BLAST# follows that of KEN# by one clock. The Intel486 processor samples KEN# every clock and uses the value returned in the clock before BRDY# or RDY# to determine if a bus cycle would be a cache line fill. Similarly, it uses the value of KEN# in the last cycle before early RDY# to load the line just retrieved from memory into the cache. KEN# is sampled every clock and it must satisfy setup and hold times.

KEN# can also change multiple times before a burst cycle, as long as it arrives at its final value one clock before BRDY# or RDY# is asserted.

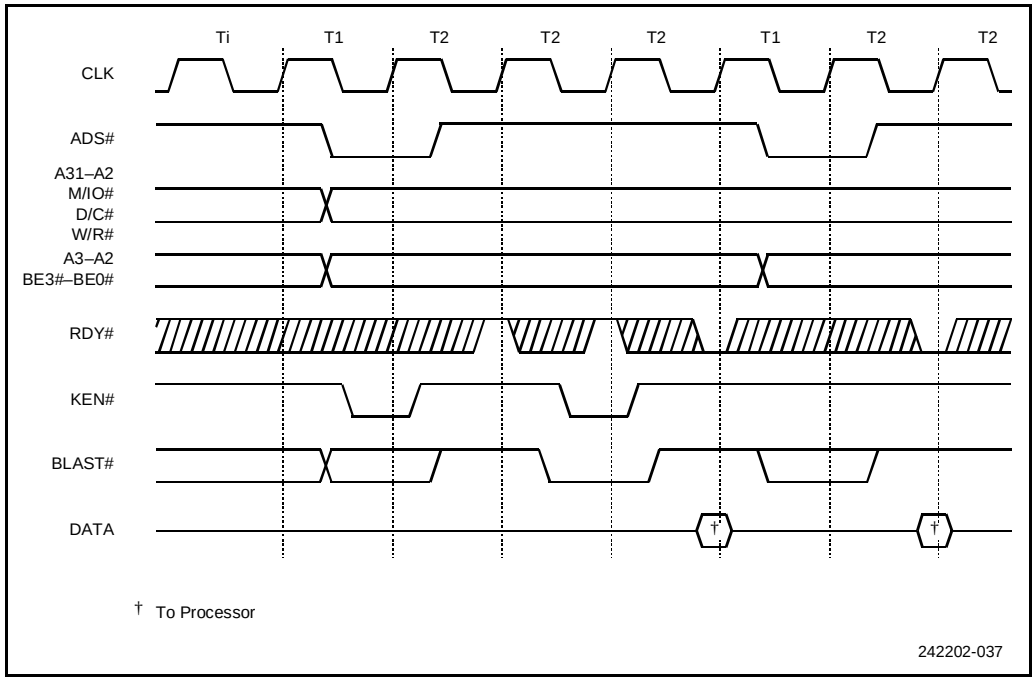


Figure 4-16. Effect of Changing KEN#

4.3.4 Burst Mode Details

4.3.4.1 Adding Wait States to Burst Cycles

Burst cycles need not return data on every clock. The Intel486 processor strobes data into the chip only when either RDY# or BRDY# is asserted. Deasserting BRDY# and RDY# adds a wait state to the transfer. A burst cycle where two clocks are required for every burst item is shown in Figure 4-17.

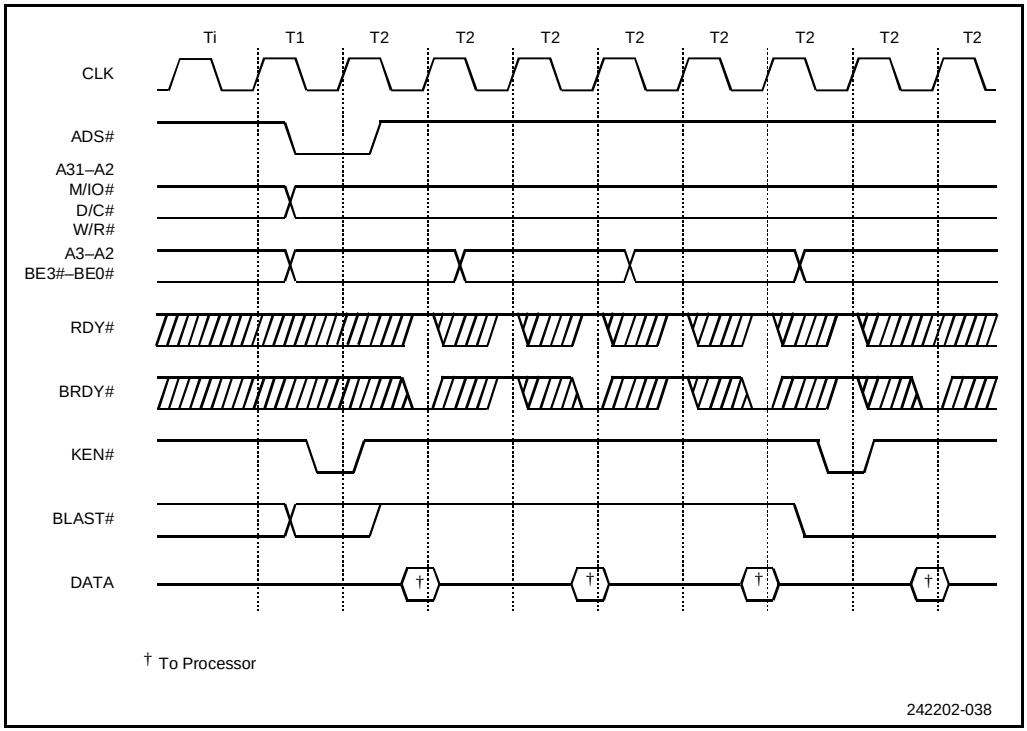


Figure 4-17. Slow Burst Cycle

4.3.4.2 Burst and Cache Line Fill Order

The burst order used by the Intel486 processor is shown in Table 4-8. This burst order is followed by any burst cycle (cache or not), cache line fill (burst or not) or code prefetch.

The Intel486 processor presents each request for data in an order determined by the first address in the transfer. For example, if the first address was 104 the next three addresses in the burst will be 100, 10C and 108. An example of burst address sequencing is shown in Figure 4-18.

Table 4-8. Burst Order (Both Read and Write Bursts)

First Address	Second Address	Third Address	Fourth Address
0	4	8	C
4	0	C	8
8	C	0	4
C	8	4	0

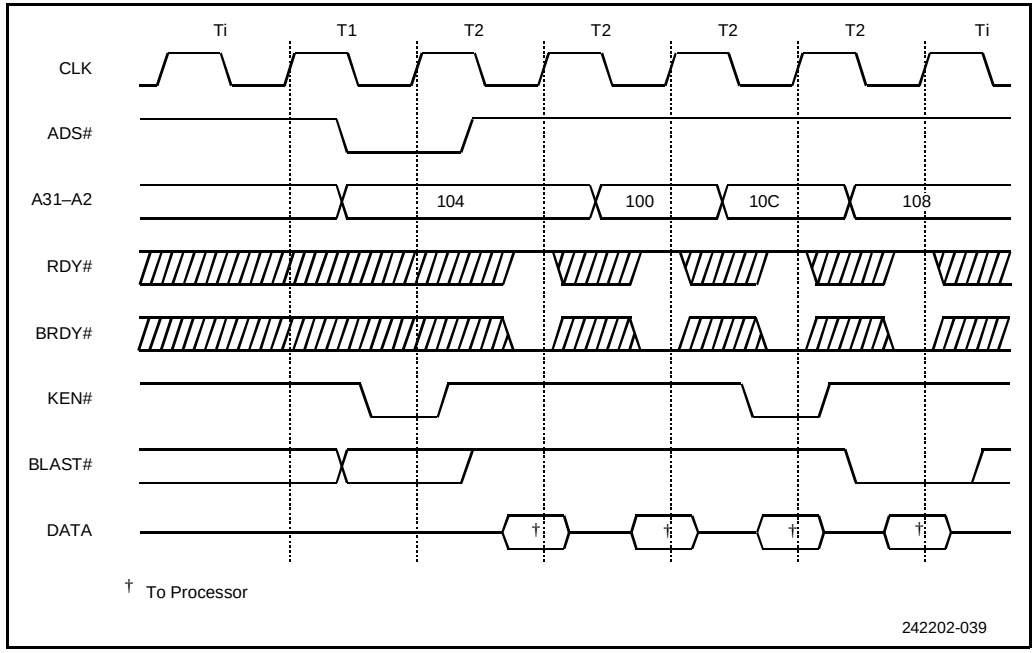


Figure 4-18. Burst Cycle Showing Order of Addresses

The sequences shown in Table 4-8 accommodate systems with 64-bit buses as well as systems with 32-bit data buses. The sequence applies to all bursts, regardless of whether the purpose of the burst is to fill a cache line, perform a 64-bit read, or perform a pre-fetch. If either BS8# or BS16# is asserted, the Intel486 processor completes the transfer of the current 32-bit word before

progressing to the next 32-bit word. For example, a BS16# burst to address 4 has the following order: 4-6-0-2-C-E-8-A.

4.3.4.3 Interrupted Burst Cycles

Some memory systems may not be able to respond with burst cycles in the order defined in Table 4-8. To support these systems, the Intel486 processor allows a burst cycle to be interrupted at any time. The Intel486 processor automatically generates another normal bus cycle after being interrupted to complete the data transfer. This is called an interrupted burst cycle. The external system can respond to an interrupted burst cycle with another burst cycle.

The external system can interrupt a burst cycle by asserting RDY# instead of BRDY#. RDY# can be asserted after any number of data cycles terminated with BRDY#.

An example of an interrupted burst cycle is shown in Figure 4-19. The Intel486 processor immediately asserts ADS# to initiate a new bus cycle after RDY# is asserted. BLAST# is deasserted one clock after ADS# begins the second bus cycle, indicating that the transfer is not complete.

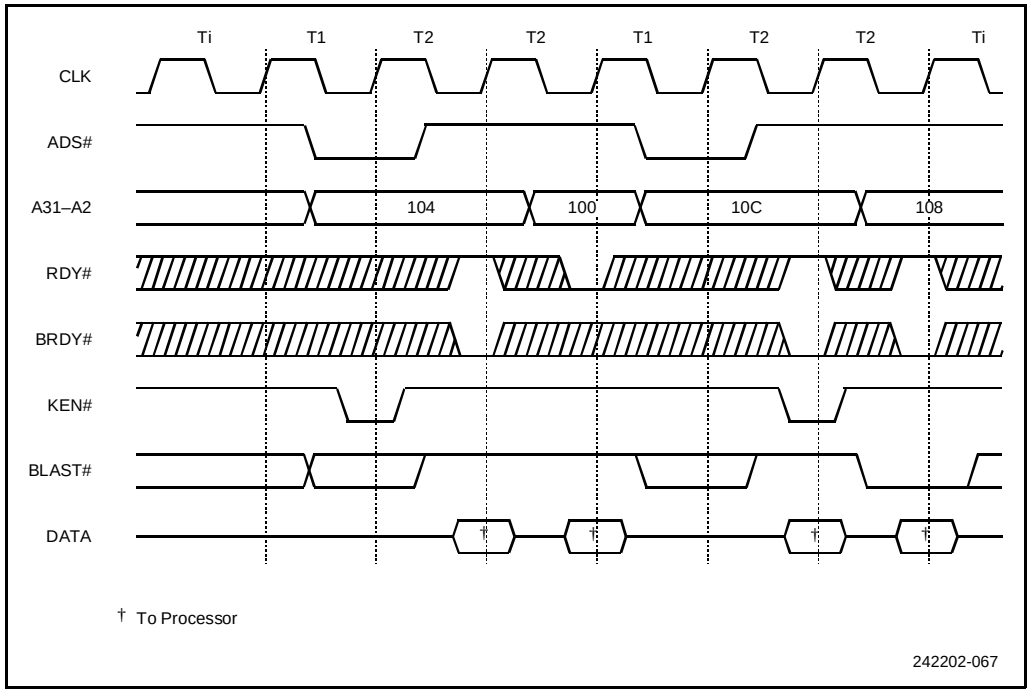


Figure 4-19. Interrupted Burst Cycle

KEN# need not be asserted in the first data cycle of the second part of the transfer shown in Figure 4-20. The cycle had been converted to a cache fill in the first part of the transfer and the Intel486 processor expects the cache fill to be completed. Note that the first half and second half of the transfer in Figure 4-19 are both two-cycle burst transfers.

The order in which the Intel486 processor requests operands during an interrupted burst transfer is shown by Table 4-7 on page 4-11. Mixing RDY# and BRDY# does not change the order in which operand addresses are requested by the Intel486 processor.

An example of the order in which the Intel486 processor requests operands during a cycle in which the external system mixes RDY# and BRDY# is shown in Figure 4-20. The Intel486 processor initially requests a transfer beginning at location 104. The transfer becomes a cache line fill when the external system asserts KEN#. The first cycle of the cache fill transfers the contents of location 104 and is terminated with RDY#. The Intel486 processor drives out a new request (by asserting ADS#) to address 100. If the external system terminates the second cycle with BRDY#, the Intel486 processor next requests/expects address 10C. The correct order is determined by the first cycle in the transfer, which may not be the first cycle in the burst if the system mixes RDY# with BRDY#.

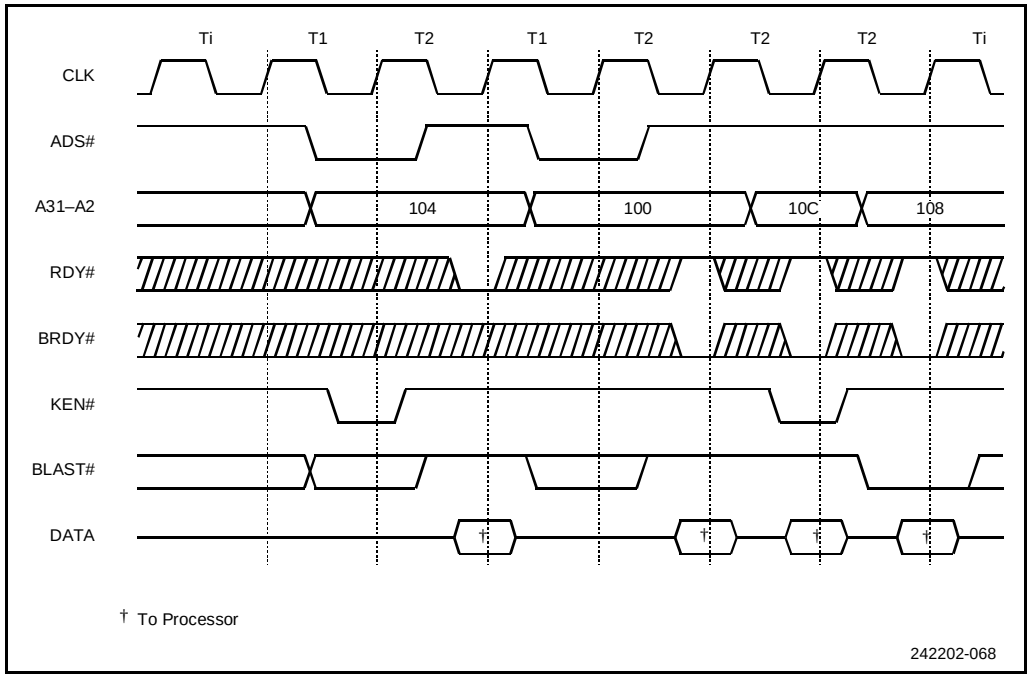


Figure 4-20. Interrupted Burst Cycle with Non-Obvious Order of Addresses

4.3.5 8- and 16-Bit Cycles

The Intel486 processor supports both 16- and 8-bit external buses through the BS16# and BS8# inputs. BS16# and BS8# allow the external system to specify, on a cycle-by-cycle basis, whether the addressed component can supply 8, 16 or 32 bits. BS16# and BS8# can be used in burst cycles as well as non-burst cycles. If both BS16# and BS8# are asserted for any bus cycle, the Intel486 processor responds as if only BS8# is asserted.

The timing of BS16# and BS8# is the same as that of KEN#. BS16# and BS8# must be asserted before the first RDY# or BRDY# is asserted. Asserting BS16# and BS8# can force the Intel486 processor to run additional cycles to complete what would have been only a single 32-bit cycle. BS8# and BS16# may change the state of BLAST# when they force subsequent cycles from the transfer.

Figure 4-21 shows an example in which BS8# forces the Intel486 processor to run two extra cycles to complete a transfer. The Intel486 processor issues a request for 24 bits of information. The external system asserts BS8#, indicating that only eight bits of data can be supplied per cycle. The Intel486 processor issues two extra cycles to complete the transfer.

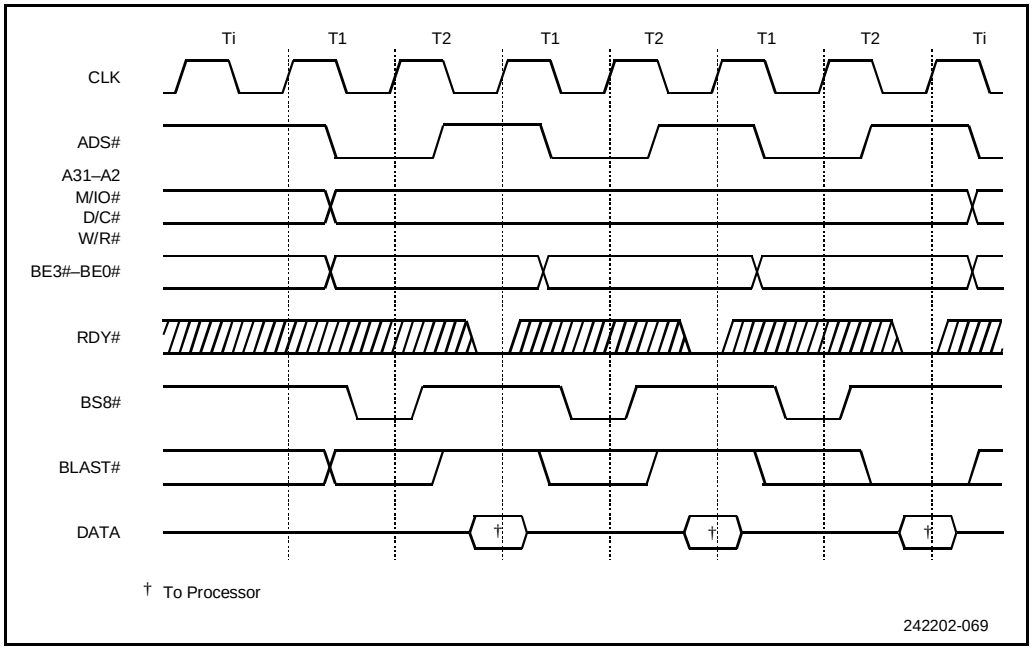


Figure 4-21. 8-Bit Bus Size Cycle

Extra cycles forced by BS16# and BS8# signals should be viewed as independent bus cycles. BS16# and BS8# should be asserted for each additional cycle unless the addressed device can change the number of bytes it can return between cycles. The Intel486 processor deasserts BLAST# until the last cycle before the transfer is complete.

Refer to Section 4.1.2, "Dynamic Data Bus Sizing," for the sequencing of addresses when BS8# or BS16# are asserted.

During burst cycles, BS8# and BS16# operate in the same manner as during non-burst cycles. For example, a single non-cacheable read could be transferred by the Intel486 processor as four 8-bit burst data cycles. Similarly, a single 32-bit write could be written as four 8-bit burst data cycles. An example of a burst write is shown in Figure 4-22. Burst writes can only occur if BS8# or BS16# is asserted.

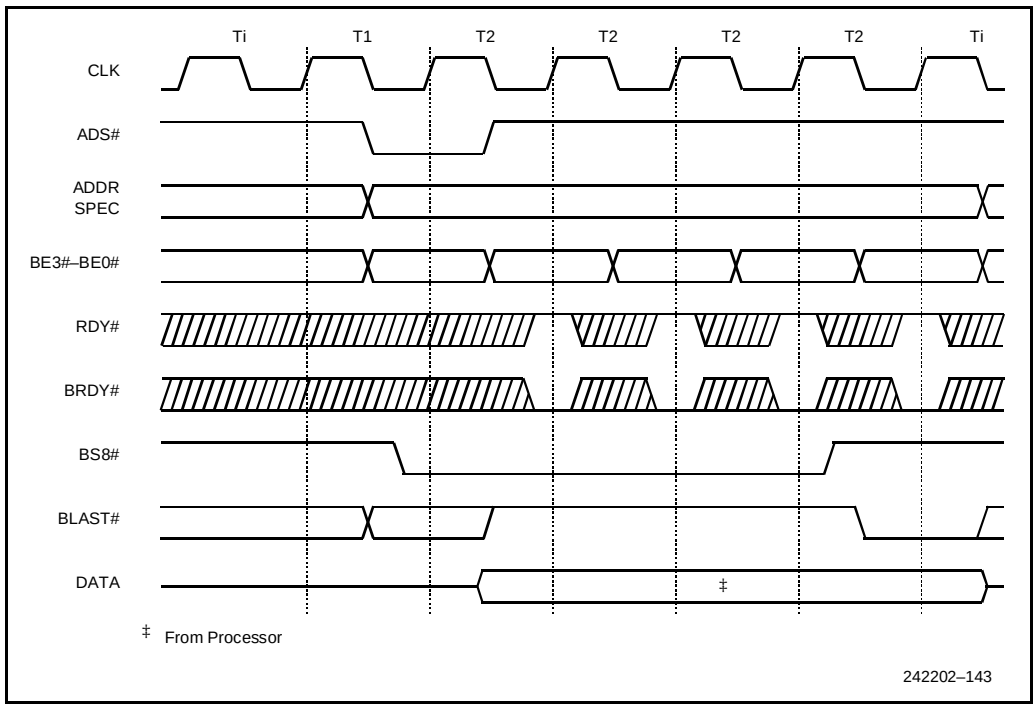


Figure 4-22. Burst Write as a Result of BS8# or BS16#

4.3.6 Locked Cycles

Locked cycles are generated in software for any instruction that performs a read-modify-write operation. During a read-modify-write operation, the Intel486 processor can read and modify a variable in external memory and ensure that the variable is not accessed between the read and write.

Locked cycles are automatically generated during certain bus transfers. The XCHG (exchange) instruction generates a locked cycle when one of its operands is memory-based. Locked cycles are generated when a segment or page table entry is updated and during interrupt acknowledge cycles. Locked cycles are also generated when the LOCK instruction prefix is used with selected instructions.

Locked cycles are implemented in hardware with the LOCK# pin. When LOCK# is asserted, the Intel486 processor is performing a read-modify-write operation and the external bus should not be relinquished until the cycle is complete. Multiple reads or writes can be locked. A locked cycle is shown in Figure 4-23. LOCK# is asserted with the address and bus definition pins at the beginning of the first read cycle and remains asserted until RDY# is asserted for the last write cycle. For unaligned 32-bit read-modify-write operations, the LOCK# remains asserted for the entire duration of the multiple cycle. It deasserts when RDY# is asserted for the last write cycle.

When LOCK# is asserted, the Intel486 processor recognizes address hold and backoff but does not recognize bus hold. It is left to the external system to properly arbitrate a central bus when the Intel486 processor generates LOCK#.

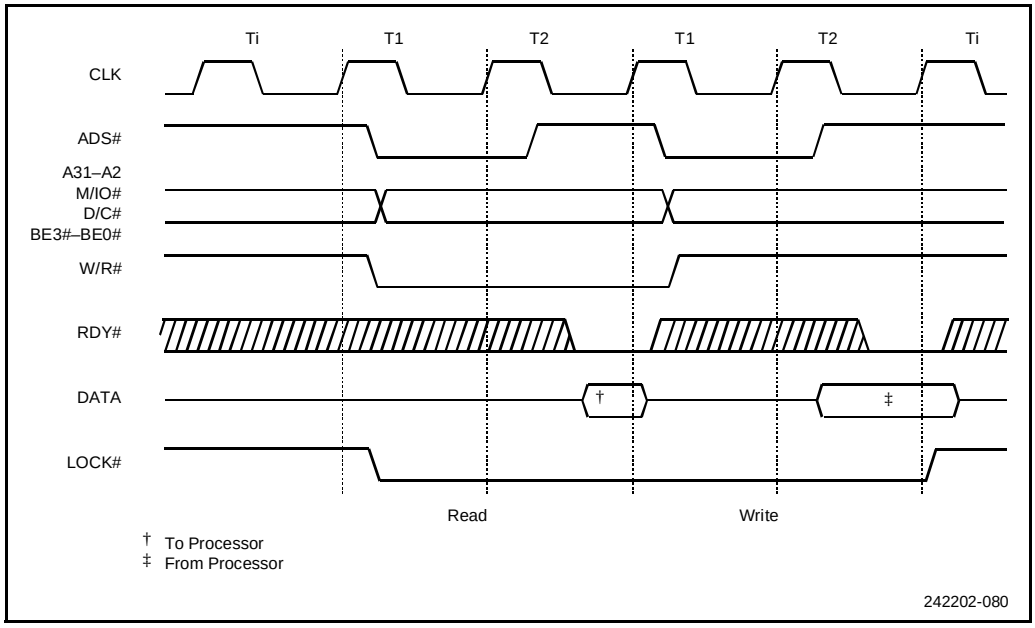


Figure 4-23. Locked Bus Cycle

4.3.7 Pseudo-Locked Cycles

Pseudo-locked cycles assure that no other master is given control of the bus during operand transfers that take more than one bus cycle.

For the Intel486 processor, examples include 64-bit description loads and cache line fills.

Pseudo-locked transfers are indicated by the PLOCK# pin. The memory operands must be aligned for correct operation of a pseudo-locked cycle.

PLOCK# need not be examined during burst reads. A 64-bit aligned operand can be retrieved in one burst (note that this is only valid in systems that do not interrupt bursts).

The system must examine PLOCK# during 64-bit writes since the Intel486 processor cannot burst write more than 32 bits. However, burst can be used within each 32-bit write cycle if BS8# or BS16# is asserted. BLAST is de-asserted in response to BS8# or BS16#. A 64-bit write is driven out as two non-burst bus cycles. BLAST# is asserted during both 32-bit writes, because a burst is not possible. PLOCK# is asserted during the first write to indicate that another write follows. This behavior is shown in Figure 4-24.

The first cycle of a 64-bit floating-point write is the only case in which both PLOCK# and BLAST# are asserted. Normally PLOCK# and BLAST# are the inverse of each other.

During all of the cycles in which PLOCK# is asserted, HOLD is not acknowledged until the cycle completes. This results in a large HOLD latency, especially when BS8# or BS16# is asserted. To reduce the HOLD latency during these cycles, windows are available between transfers to allow HOLD to be acknowledged during non-cacheable code prefetches. PLOCK# is asserted because BLAST# is deasserted, but PLOCK# is ignored and HOLD is recognized during the prefetch.

PLOCK# can change several times during a cycle, settling to its final value in the clock in which RDY# is asserted.

4.3.7.1 Floating-Point Read and Write Cycles

For IntelDX2 and Write-Back Enhanced IntelDX4 processors, 64-bit floating-point read and write cycles are also examples of operand transfers that take more than one bus cycle.

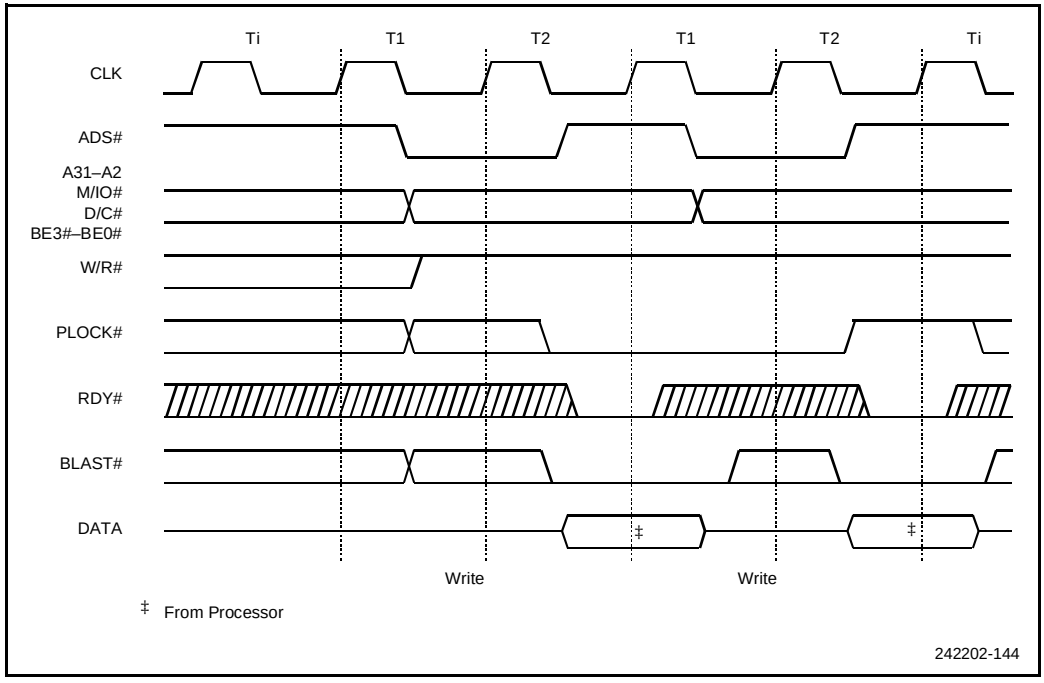


Figure 4-24. Pseudo Lock Timing

4.3.8 Invalidate Cycles

Invalidate cycles keep the Intel486 processor internal cache contents consistent with external memory. The Intel486 processor contains a mechanism for monitoring writes by other devices to external memory. When the Intel486 processor finds a write to a section of external memory contained in its internal cache, the Intel486 processor's internal copy is invalidated.

Invalidations use two pins, address hold request (AHOLD) and valid external address (EADS#). There are two steps in an invalidation cycle. First, the external system asserts the AHOLD input forcing the Intel486 processor to immediately relinquish its address bus. Next, the external system asserts EADS#, indicating that a valid address is on the Intel486 processor address bus. Figure 4-25 shows the fastest possible invalidation cycle. The Intel486 processor recognizes AHOLD on one CLK edge and floats the address bus in response. To allow the address bus to float and avoid contention, EADS# and the invalidation address should not be driven until the following CLK edge. The Intel486 processor reads the address over its address lines. If the Intel486 processor finds this address in its internal cache, the cache entry is invalidated. Note that the Intel486 processor address bus is input/output, unlike the Intel386 processor's bus, which is output only.

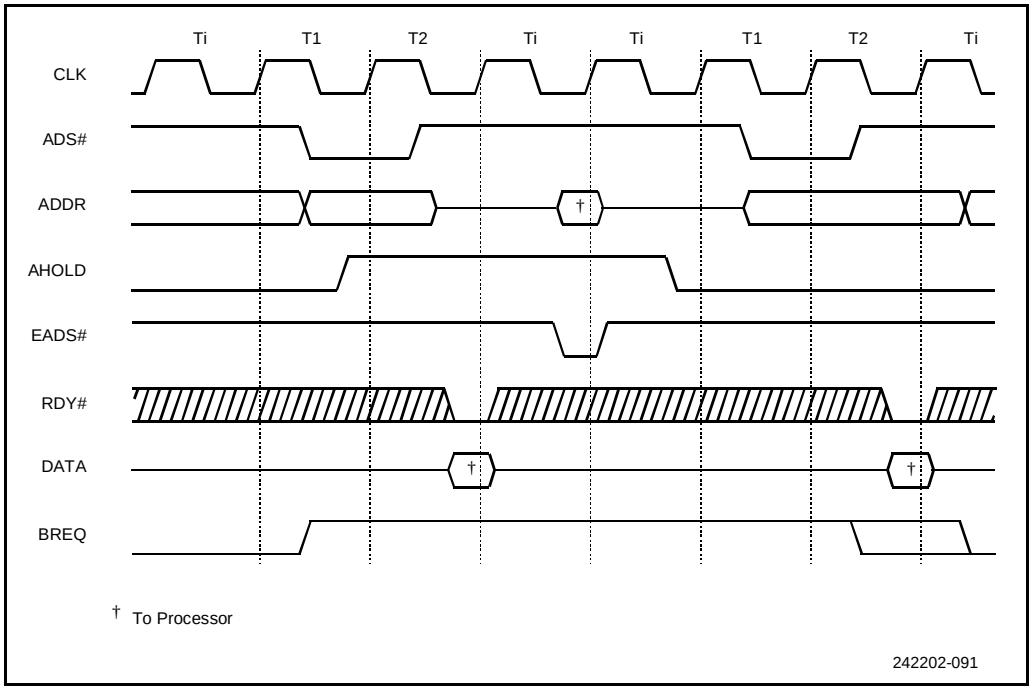


Figure 4-25. Fast Internal Cache Invalidation Cycle

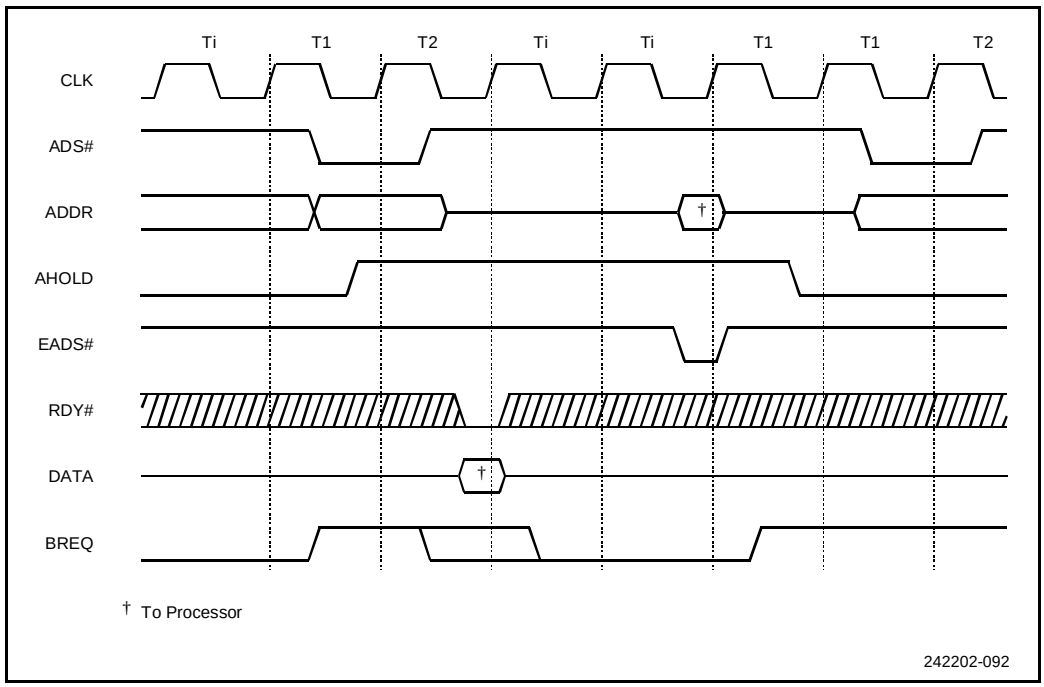


Figure 4-26. Typical Internal Cache Invalidation Cycle

4.3.8.1 Rate of Invalidate Cycles

The Intel486 processor can accept one invalidate per clock except in the last clock of a line fill. One invalidate per clock is possible as long as EADS# is deasserted in ONE or BOTH of the following cases:

1. In the clock in which RDY# or BRDY# is asserted for the last time.
2. In the clock following the clock in which RDY# or BRDY# is asserted for the last time.

This definition allows two system designs. Simple designs can restrict invalidates to one every other clock. The simple design need not track bus activity. Alternatively, systems can request one invalidate per clock provided that the bus is monitored.

4.3.8.2 Running Invalidate Cycles Concurrently with Line Fills

Precautions are necessary to avoid caching stale data in the Intel486 processor cache in a system with a second-level cache. An example of a system with a second-level cache is shown in Figure 4-27.

An external device can write to main memory over the system bus while the Intel486 processor is retrieving data from the second-level cache. The Intel486 processor must invalidate a line in its internal cache if the external device is writing to a main memory address that is also contained in the Intel486 processor cache.

A potential problem exists if the external device is writing to an address in external memory, and at the same time the Intel486 processor is reading data from the same address in the second-level cache. The system must force an invalidation cycle to invalidate the data that the Intel486 processor has requested during the line fill.

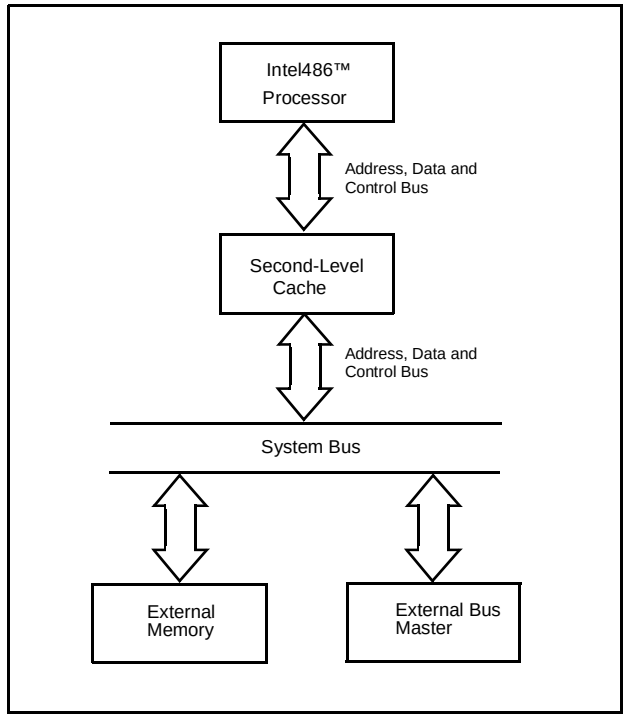


Figure 4-27. System with Second-Level Cache

If the system asserts EADS# before the first data in the line fill is returned to the Intel486 processor, the system must return data consistent with the new data in the external memory upon resumption of the line fill after the invalidation cycle. This is illustrated by the asserted EADS# signal labeled “1” in Figure 4-28.

If the system asserts EADS# at the same time or after the first data in the line fill is returned (in the same clock that the first RDY# or BRDY# is asserted or any subsequent clock in the line fill) the data is read into the Intel486 processor input buffers but it is not stored in the on-chip cache. This is illustrated by asserted EADS# signal labeled “2” in Figure 4-28. The stale data is used to satisfy the request that initiated the cache fill cycle.

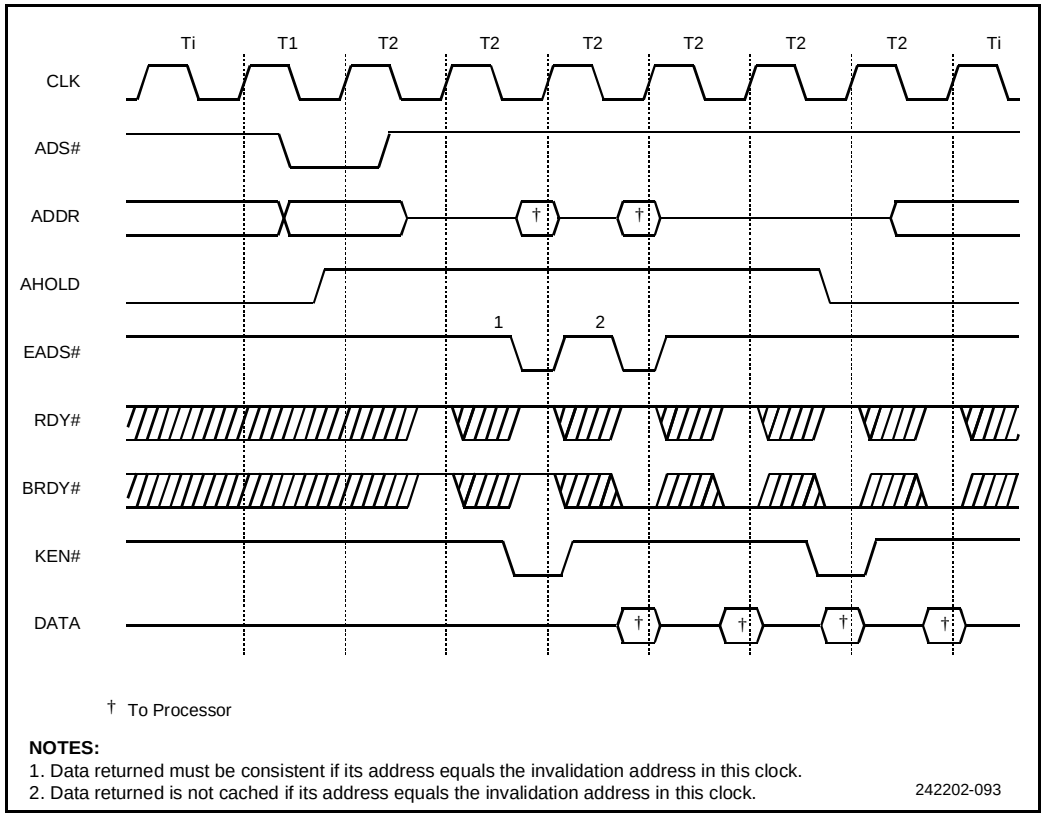


Figure 4-28. Cache Invalidation Cycle Concurrent with Line Fill

4.3.9 Bus Hold

The Intel486 processor provides a bus hold, hold acknowledge protocol using the bus hold request (HOLD) and bus hold acknowledge (HLDA) pins. Asserting the HOLD input indicates that another bus master has requested control of the Intel486 processor bus. The Intel486 processor responds by floating its bus and asserting HLDA when the current bus cycle, or sequence of locked cycles, is complete. An example of a HOLD/HLDA transaction is shown in Figure 4-29. Unlike the Intel386 processor, the Intel486 processor can respond to HOLD by floating its bus and asserting HLDA while RESET is asserted.

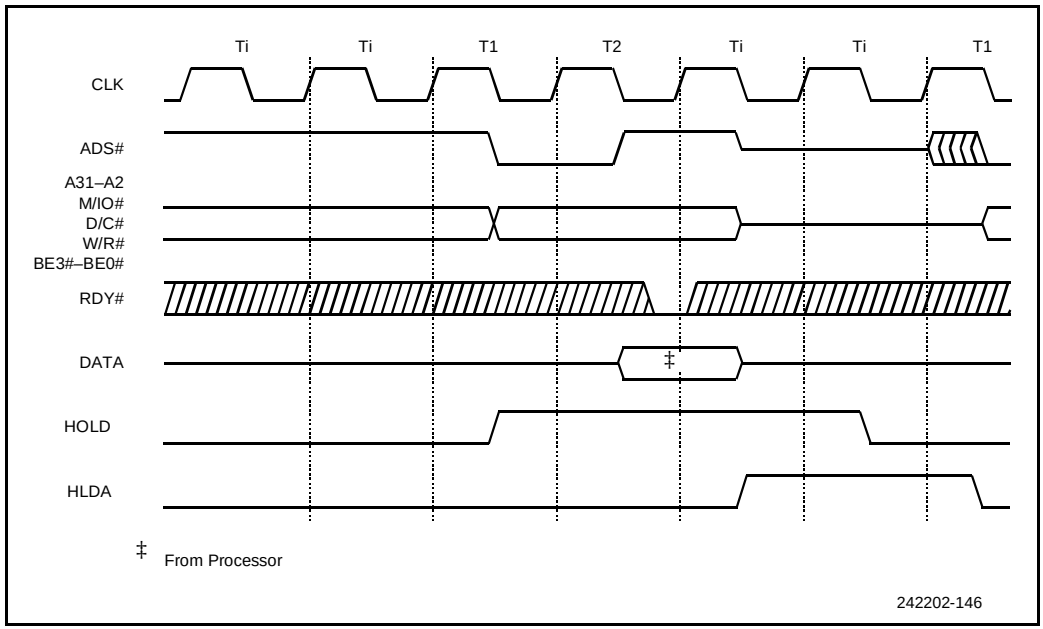


Figure 4-29. HOLD/HLDA Cycles

Note that HOLD is recognized during un-aligned writes (less than or equal to 32 bits) with BLAST# being asserted for each write. For a write greater than 32-bits or an un-aligned write, HOLD# recognition is prevented by PLOCK# getting asserted. However, HOLD is recognized during non-cacheable, non-burstable code prefetches even though PLOCK# is asserted.

For cacheable and non-burst or burst cycles, HOLD is acknowledged during backoff only if HOLD and BOFF# are asserted during an active bus cycle (after ADS# asserted) and before the first RDY# or BRDY# has been asserted (see Figure 4-30). The order in which HOLD and BOFF# are asserted is unimportant (as long as both are asserted prior to the first RDY#/BRDY# asserted by the system). Figure 4-30 shows the case where HOLD is asserted first; HOLD could be asserted simultaneously or after BOFF# and still be acknowledged.

The pins floated during bus hold are: BE3#–BE0#, PCD, PWT, W/R#, D/C#, M/O#, LOCK#, PLOCK#, ADS#, BLAST#, D31–D0, A31–A2, and DP3–DP0.

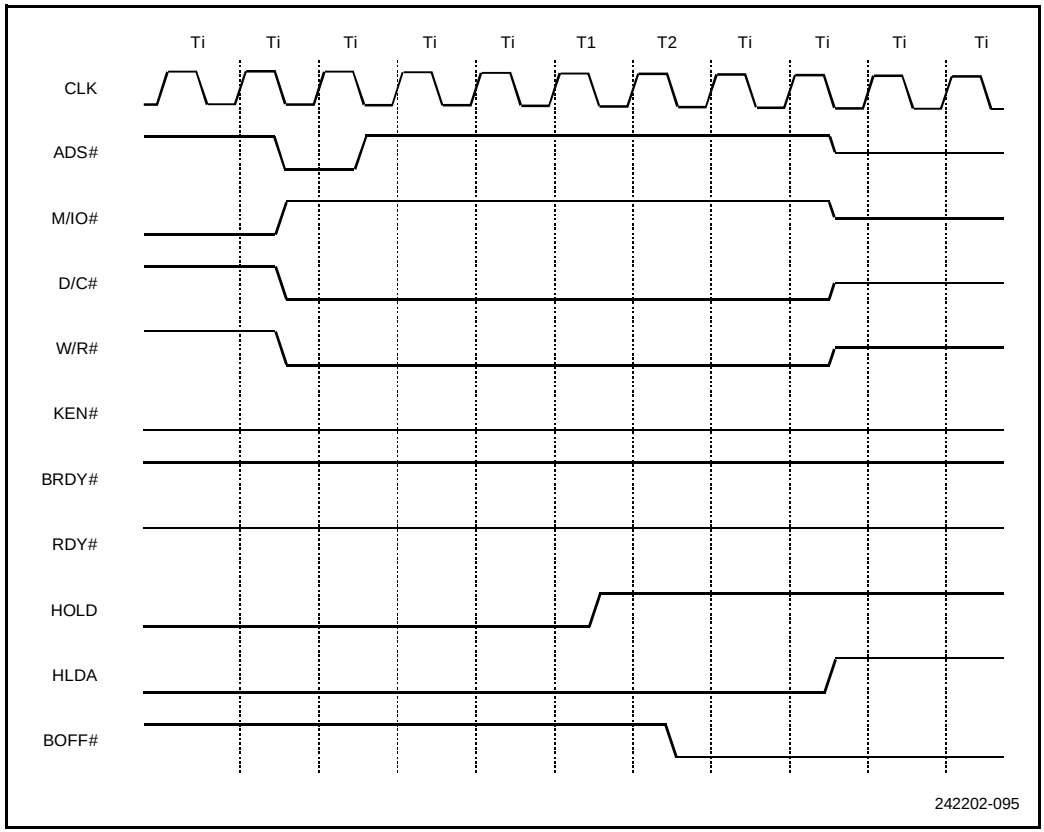


Figure 4-30. HOLD Request Acknowledged during BOFF#

4.3.10 Interrupt Acknowledge

The Intel486 processor generates interrupt acknowledge cycles in response to maskable interrupt requests that are generated on the interrupt request input (INTR) pin. Interrupt acknowledge cycles have a unique cycle type generated on the cycle type pins.

An example of an interrupt acknowledge transaction is shown in Figure 4-31. Interrupt acknowledge cycles are generated in locked pairs. Data returned during the first cycle is ignored. The interrupt vector is returned during the second cycle on the lower 8 bits of the data bus. The Intel486 processor has 256 possible interrupt vectors.

The state of A2 distinguishes the first and second interrupt acknowledge cycles. The byte address driven during the first interrupt acknowledge cycle is 4 (A31–A3 low, A2 high, BE3#–BE1# high, and BE0# low). The address driven during the second interrupt acknowledge cycle is 0 (A31–A2 low, BE3#–BE1# high, BE0# low).

Each of the interrupt acknowledge cycles is terminated when the external system asserts RDY# or BRDY#. Wait states can be added by holding RDY# or BRDY# deasserted. The Intel486 processor automatically generates four idle clocks between the first and second cycles to allow for 8259A recovery time.

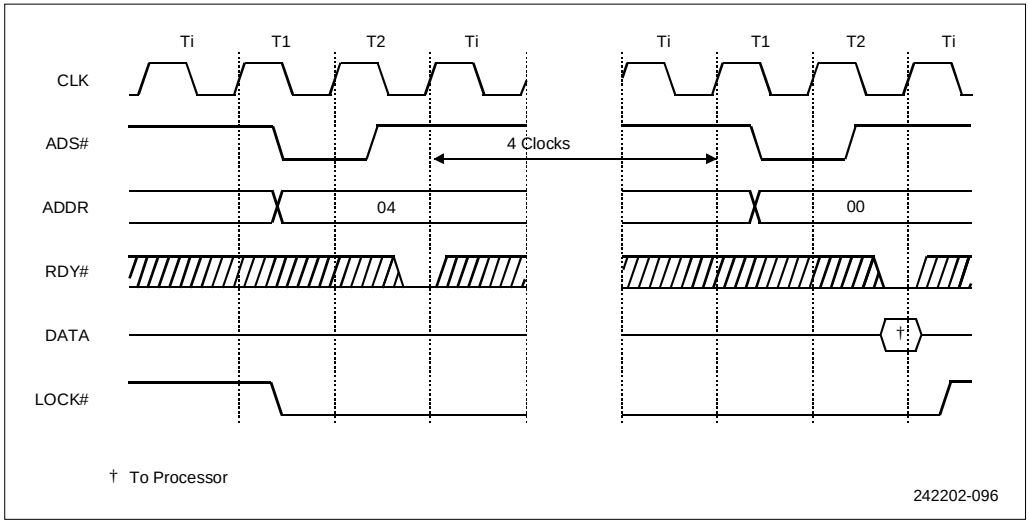


Figure 4-31. Interrupt Acknowledge Cycles

4.3.11 Special Bus Cycles

The Intel486 processor provides special bus cycles to indicate that certain instructions have been executed, or certain conditions have occurred internally. The special bus cycles are identified by the status of the pins shown in Table 4-9.

During these cycles the address bus is driven low while the data bus is undefined.

Two of the special cycles indicate halt or shutdown. Another special cycle is generated when the Intel486 processor executes an INVD (invalidate data cache) instruction and could be used to flush an external cache. The Write Back cycle is generated when the Intel486 processor executes the WBINVD (write-back invalidate data cache) instruction and could be used to synchronize an external write-back cache.

The external hardware must acknowledge these special bus cycles by asserting RDY# or BRDY#.

4.3.11.1 HALT Indication Cycle

The Intel486 processor halts as a result of executing a HALT instruction. A HALT indication cycle is performed to signal that the processor has entered into the HALT state. The HALT indication cycle is identified by the bus definition signals in special bus cycle state and by a byte address of 2. BE0# and BE2# are the only signals that distinguish HALT indication from shutdown indication, which drives an address of 0. During the HALT cycle, undefined data is driven on D31–D0. The HALT indication cycle must be acknowledged by RDY# asserted.

A halted Intel486 processor resumes execution when INTR (if interrupts are enabled), NMI, or RESET is asserted.

4.3.11.2 Shutdown Indication Cycle

The Intel486 processor shuts down as a result of a protection fault while attempting to process a double fault. A shutdown indication cycle is performed to indicate that the processor has entered a shutdown state. The shutdown indication cycle is identified by the bus definition signals in special bus cycle state and a byte address of 0.

4.3.11.3 Stop Grant Indication Cycle

A special Stop Grant bus cycle is driven to the bus after the processor recognizes the STPCLK# interrupt. The definition of this bus cycle is the same as the HALT cycle definition for the Intel486 processor, with the exception that the Stop Grant bus cycle drives the value 0000 0010H on the address pins. The system hardware must acknowledge this cycle by asserting RDY# or BRDY#. The processor does not enter the Stop Grant state until either RDY# or BRDY# has been asserted. (See Figure 4-32.)

The Stop Grant Bus Cycle is defined as follows:

M/IO# = 0, D/C# = 0, W/R# = 1, Address Bus = 0000 0010H (A4 = 1), BE3#–BE0# = 1011, Data bus = undefined.

The latency between a STPCLK# request and the Stop Grant bus cycle is dependent on the current instruction, the amount of data in the processor write buffers, and the system memory performance.

Table 4-9. Special Bus Cycle Encoding

Cycle Name	M/IO#	D/C#	W/R#	BE3#–BE0#	A4-A2
Write-Back†	0	0	1	0111	000
First Flush Ack Cycle†	0	0	1	0111	001
Flush†	0	0	1	1101	000
Second Flush Ack Cycle†	0	0	1	1101	001
Shutdown	0	0	1	1110	000
HALT	0	0	1	1011	000
Stop Grant Ack Cycle	0	0	1	1011	100

† These cycles are specific to the Write-Back Enhanced IntelDX4™ processor. The FLUSH# cycle is applicable to all Intel486™ processors. See appropriate sections for details.

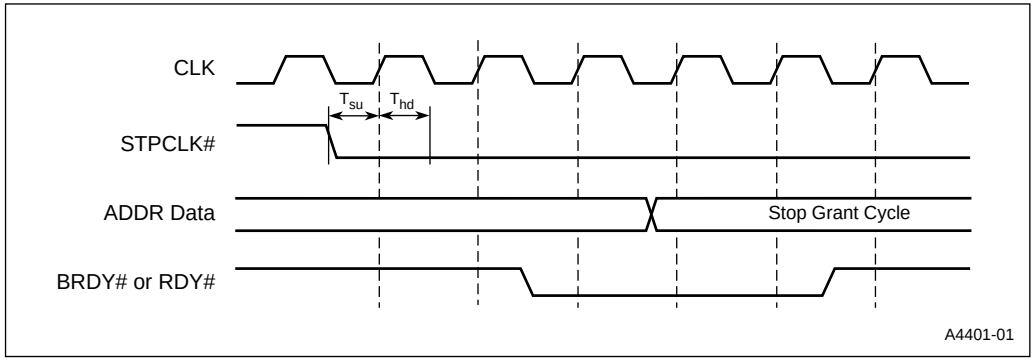


Figure 4-32. Stop Grant Bus Cycle

4.3.12 Bus Cycle Restart

In a multi-master system, another bus master may require the use of the bus to enable the Intel486 processor to complete its current bus request. In this situation, the Intel486 processor must restart its bus cycle after the other bus master has completed its bus transaction.

A bus cycle may be restarted if the external system asserts the backoff (BOFF#) input. The Intel486 processor samples the BOFF# pin every clock cycle. When BOFF# is asserted, the Intel486 processor floats its address, data, and status pins in the next clock (see Figures 4-33 and 4-34). Any bus cycle in progress when BOFF# is asserted is aborted and any data returned to the processor is ignored. The pins that are floated in response to BOFF# are the same as those that are floated in response to HOLD. HLDA is not generated in response to BOFF#. BOFF# has higher priority than RDY# or BRDY#. If either RDY# or BRDY# are asserted in the same clock as BOFF#, BOFF# takes effect.

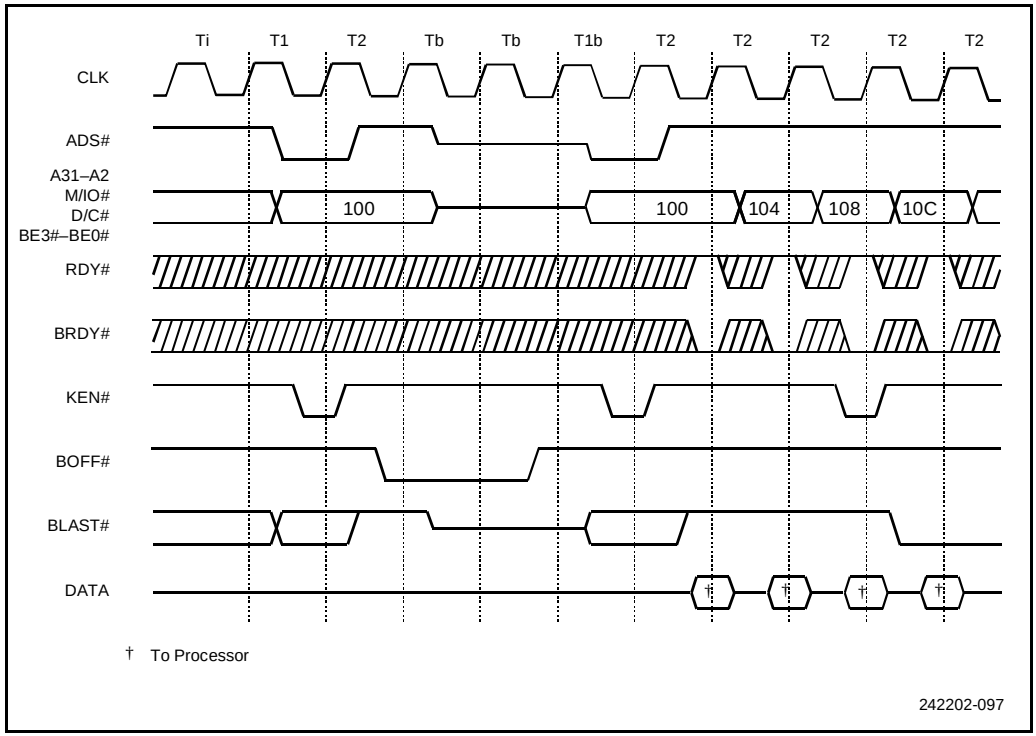


Figure 4-33. Restarted Read Cycle

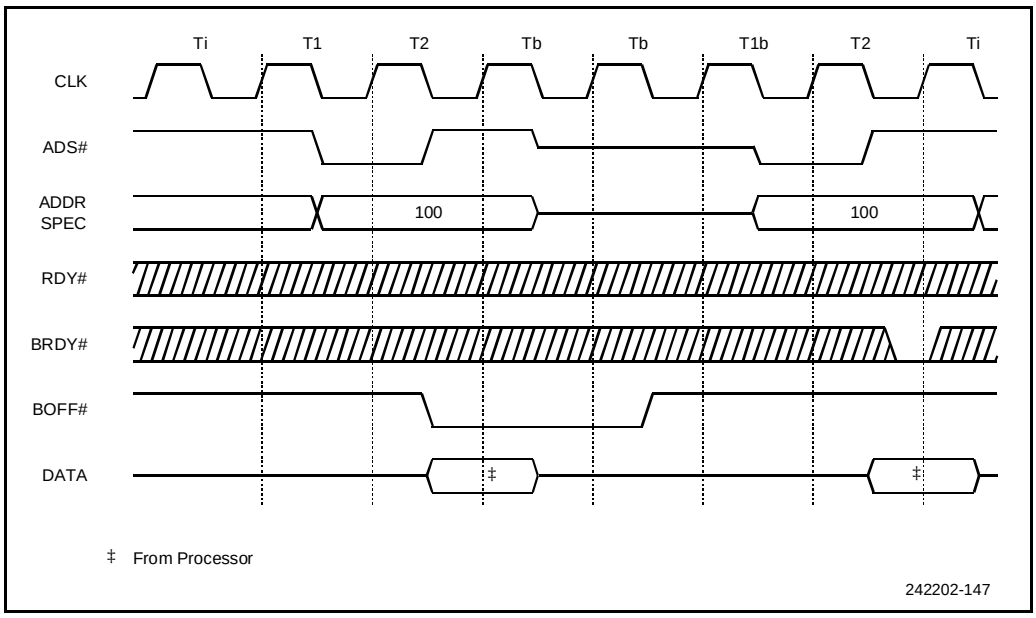


Figure 4-34. Restarted Write Cycle

The device asserting BOFF# is free to run cycles while the Intel486 processor bus is in its high impedance state. If backoff is requested after the Intel486 processor has started a cycle, the new master should wait for memory to assert RDY# or BRDY# before assuming control of the bus. Waiting for RDY# or BRDY# provides a handshake to ensure that the memory system is ready to accept a new cycle. If the bus is idle when BOFF# is asserted, the new master can start its cycle two clocks after issuing BOFF#.

The external memory can view BOFF# in the same manner as BLAST#. Asserting BOFF# tells the external memory system that the current cycle is the last cycle in a transfer.

The bus remains in the high impedance state until BOFF# is deasserted. Upon negation, the Intel486 processor restarts its bus cycle by driving out the address and status and asserting ADS#. The bus cycle then continues as usual.

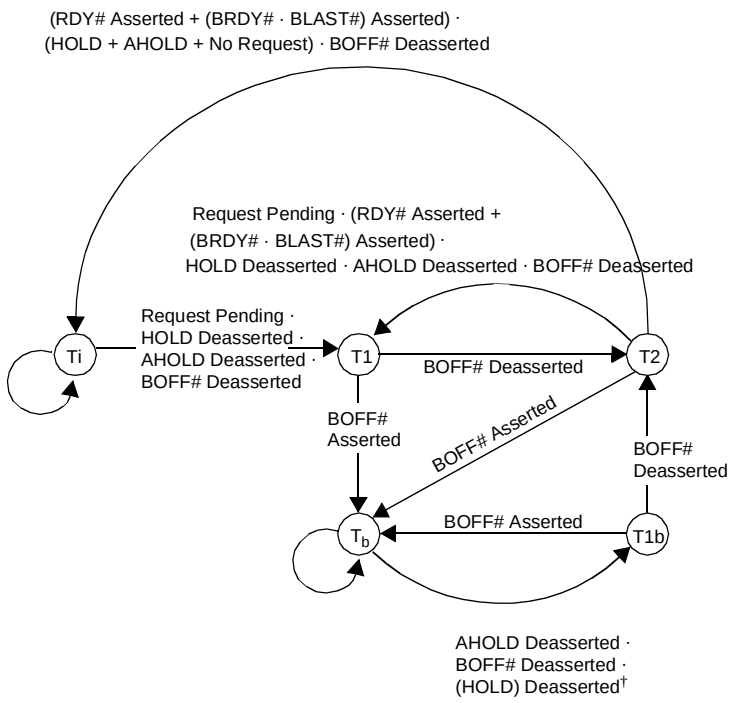
Asserting BOFF# during a burst, BS8#, or BS16# cycle forces the Intel486 processor to ignore data returned for that cycle only. Data from previous cycles is still valid. For example, if BOFF# is asserted on the third BRDY# of a burst, the Intel486 processor assumes the data returned with the first and second BRDY# is correct and restarts the burst beginning with the third item. The same rule applies to transfers broken into multiple cycles by BS8# or BS16#.

Asserting BOFF# in the same clock as ADS# causes the Intel486 processor to float its bus in the next clock and leave ADS# floating low. Because ADS# is floating low, a peripheral may think that a new bus cycle has begun even though the cycle was aborted. There are two possible solutions to this problem. The first is to have all devices recognize this condition and ignore ADS# until RDY# is asserted. The second approach is to use a “two clock” backoff: in the first clock

AHOLD is asserted, and in the second clock BOFF# is asserted. This guarantees that ADS# is not floating low. This is necessary only in systems where BOFF# may be asserted in the same clock as ADS#.

4.3.13 Bus States

A bus state diagram is shown in Figure 4-35. A description of the signals used in the diagram is given in Table 4-10.



† HOLD is only factored into this state transition if T_b was entered while a non-cacheable. non-burst, code prefetch was in progress. Otherwise, ignore HOLD.

Figure 4-35. Bus State Diagram

Table 4-10. Bus State Description

State	Means
Ti	Bus is idle. Address and status signals may be driven to undefined values, or the bus may be floated to a high impedance state.
T1	First clock cycle of a bus cycle. Valid address and status are driven and ADS# is asserted.
T2	Second and subsequent clock cycles of a bus cycle. Data is driven if the cycle is a write, or data is expected if the cycle is a read. RDY# and BRDY# are sampled.
$T1_b$	First clock cycle of a restarted bus cycle. Valid address and status are driven and ADS# is asserted.
T_b	Second and subsequent clock cycles of an aborted bus cycle.

4.3.14 Floating-Point Error Handling for the IntelDX2™ and IntelDX4™ Processors

The IntelDX2 and IntelDX4 processors provide two options for reporting floating-point errors. The simplest method is to raise interrupt 16 whenever an unmasked floating-point error occurs. This option may be enabled by setting the NE bit in control register 0 (CR0).

The IntelDX2 and IntelDX4 processors also provide the option of allowing external hardware to determine how floating-point errors are reported. This option is necessary for compatibility with the error reporting scheme used in DOS-based systems. The NE bit must be cleared in CR0 to enable user-defined error reporting. User-defined error reporting is the default condition because the NE bit is cleared on reset.

Two pins, floating-point error (FERR#, an output) and ignore numeric error (IGNNE#, an input) are provided to direct the actions of hardware if user-defined error reporting is used. The IntelDX2 and IntelDX4 processors assert the FERR# output to indicate that a floating-point error has occurred. FERR# corresponds to the ERROR# pin on the Intel387™ math coprocessor. However, there is a difference in the behavior of the two.

In some cases FERR# is asserted when the next floating-point instruction is encountered, and in other cases it is asserted before the next floating-point instruction is encountered, depending upon the execution state of the instruction causing the exception.

4.3.14.1 Floating-Point Exceptions

The following class of floating-point exceptions drive FERR# at the time the exception occurs (i.e., before encountering the next floating-point instruction).

1. The stack fault, invalid operation, and denormal exceptions on all transcendental instructions, integer arithmetic instructions, FSQRT, FSEALE, FPREM(1), FXTRACT, FBLD, and FBSTP.
2. Any exceptions on store instructions (including integer store instructions).

The following class of floating-point exceptions drive FERR# only after encountering the next floating-point instruction.

3. Exceptions other than on all transcendental instructions, integer arithmetic instructions, FSQRT, FSCALE, FPREM(1), FXTRACT, FBLD, and FBSTP.
4. Any exception on all basic arithmetic, load, compare, and control instructions (i.e., all other instructions).

For both sets of exceptions above, the Intel387 math coprocessor asserts ERROR# when the error occurs and does not wait for the next floating-point instruction to be encountered.

IGNNE# is an input to the IntelDX2 and IntelDX4 processors. When the NE bit in CR0 is cleared, and IGNNE# is asserted, the IntelDX2 and IntelDX4 processors ignore user floating-point errors and continue executing floating-point instructions. When IGNNE# is deasserted, the IGNNE# is an input to these processors that freeze on floating-point instructions that get errors (except for the control instructions FNCLEX, FNINIT, FNSAVE, FNSTENV, FNSTCW, FNSTSW, FNSTSW AX, FNENI, FNDISI and FNSETPM). IGNNE# may be asynchronous to the IntelDX2 and IntelDX4 processor clock.

In systems with user-defined error reporting, the FERR# pin is connected to the interrupt controller. When an unmasked floating-point error occurs, an interrupt is raised. If IGNNE# is high at the time of this interrupt, the IntelDX2 and IntelDX4 processors freeze (disallowing execution of a subsequent floating-point instruction) until the interrupt handler is invoked. By driving the IGNNE# pin low (when clearing the interrupt request), the interrupt handler can allow execution of a floating-point instruction, within the interrupt handler, before the error condition is cleared (by FNCLEX, FNINIT, FNSAVE or FNSTENV). If execution of a non-control floating-point instruction, within the floating-point interrupt handler, is not needed, the IGNNE# pin can be tied high.

4.3.15 IntelDX2™ and IntelDX4™ Processors Floating-Point Error Handling in AT-Compatible Systems

The IntelDX2 and IntelDX4 processors provide special features to allow the implementation of an AT-compatible numerics error reporting scheme. These features DO NOT replace the external circuit. Logic is still required that decodes the OUT F0 instruction and latches the FERR# signal. The use of these Intel Processor features is described below.

- The NE bit in the Machine Status Register
- The IGNNE# pin
- The FERR# pin

The NE bit determines the action taken by the IntelDX2 and IntelDX4 processors when a numerics error is detected. When set, this bit signals that non-DOS compatible error handling is implemented. In this mode the IntelDX2 and IntelDX4 processors take a software exception (16) if a numerics error is detected.

If the NE bit is reset, the IntelDX2 and IntelDX4 processors use the IGNNE# pin to allow an external circuit to control the time at which non-control numerics instructions are allowed to execute. Note that floating-point control instructions such as FNINIT and FNSAVE can be executed during a floating-point error condition regardless of the state of IGNNE#.

To process a floating-point error in the DOS environment, the following sequence must take place:

1. The error is detected by the IntelDX2 and IntelDX4 processor that activates the FERR# pin.
2. FERR# is latched so that it can be cleared by the OUT F0 instruction.
3. The latched FERR# signal activates an interrupt at the interrupt controller. This interrupt is usually handled on IRQ13.
4. The Interrupt Service Routine (ISR) handles the error and then clears the interrupt by executing an OUT instruction to port F0. The address F0 is decoded externally to clear the FERR# latch. The IGNNE# signal is also activated by the decoder output.
5. Usually the ISR then executes an FNINIT instruction or other control instruction before restarting the program. FNINIT clears the FERR# output.

Figure 4-36 illustrates a sample circuit that performs the function described above. Note that this circuit has not been tested and is included as an example of required error handling logic.

Note that the IGNNE# input allows non-control instructions to be executed prior to the time the FERR# signal is reset by the IntelDX2 and IntelDX4 processors. This function is implemented to allow exact compatibility with the AT implementation. Most programs re-initialize the Floating-Point Unit (FPU) before continuing after an error is detected. The FPU can be re-initialized using one of the following four instructions: FCLEX, FINIT, FSAVE and FSTENV.

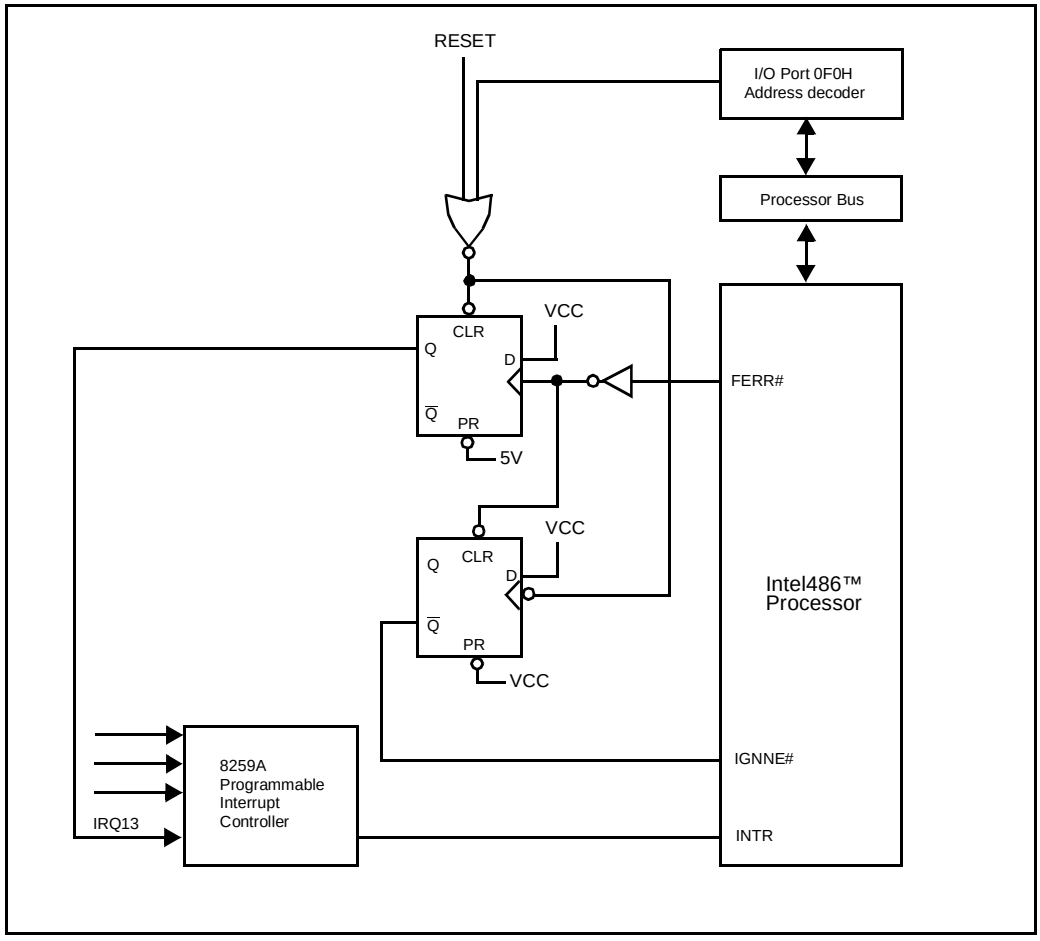


Figure 4-36. DOS-Compatible Numerics Error Circuit

4.4 ENHANCED BUS MODE OPERATION (WRITE-BACK MODE) FOR THE WRITE-BACK ENHANCED IntelDX4™ PROCESSOR

All Intel486™ processors operate in Standard Bus (write-through) mode. However, when the internal cache of the Write-Back Enhanced IntelDX4 processor is configured in write-back mode, the processor bus operates in the Enhanced Bus mode. This section describes how the Write-Back Enhanced Intel486 processor bus operation changes for the Enhanced Bus mode when the internal cache is configured in write-back mode.

4.4.1 Summary of Bus Differences

The following is a list of the differences between the Enhanced Bus and Standard Bus modes. In Enhanced Bus mode:

1. Burst write capability is extended to four doubleword burst cycles (for write-back cycles only).
2. Four new signals: INV, WB/WT#, HITM#, and CACHE#, have been added to support the write-back operation of the internal cache. These signals function the same as the equivalent signals on the Pentium® OverDrive® processor pins.
3. The SRESET signal has been modified so that it does not write back, invalidate, or disable the cache. Special test modes are also not initiated through SRESET.
4. The FLUSH# signal behaves the same as the WBINVD instruction. Upon assertion, FLUSH# writes back all modified lines, invalidates the cache, and issues two special bus cycles.
5. The PLOCK# signal remains deasserted.

4.4.2 Burst Cycles

Figure 4-37 shows a basic burst read cycle of the Write-Back Enhanced IntelDX4 processor. In the Enhanced Bus mode, both PCD and CACHE# are asserted if the cycle is internally cacheable. The Write-Back Enhanced IntelDX4 processor samples KEN# in the clock before the first BRDY#. If KEN# is asserted by the system, this cycle is transformed into a multiple-transfer cycle. With each data item returned from external memory, the data is “cached” only if KEN# is asserted again in the clock before the last BRDY# signal. Data is sampled only in the clock in which BRDY# is asserted. If the data is not sent to the processor every clock, it causes a “slow burst” cycle.

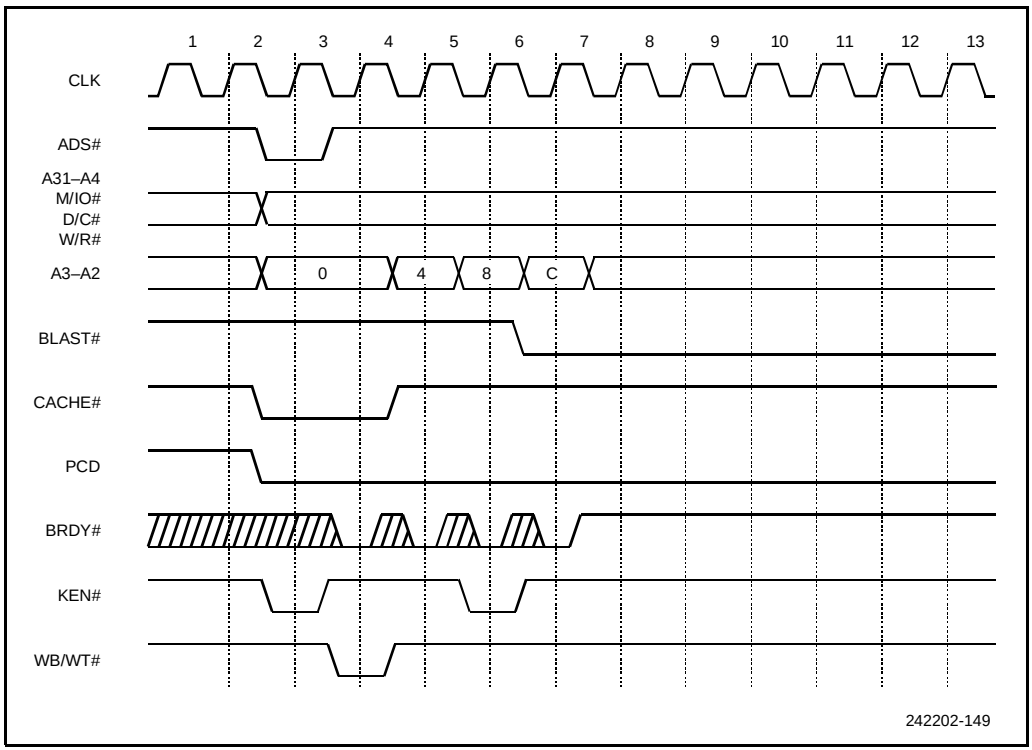


Figure 4-37. Basic Burst Read Cycle

4.4.2.1 Non-Cacheable Burst Operation

When CACHE# is asserted on a read cycle, the processor follows with BLAST# high when KEN# is asserted. However, the converse is not true. The Write-Back Enhanced IntelDX4 processor may elect to read burst data that are identified as non-cacheable by either CACHE# or KEN#. In this case, BLAST# is also high in the same cycle as the first BRDY# (in clock four). To improve performance, the memory controller should try to complete the cycle as a burst cycle.

The assertion of CACHE# on a write cycle signifies a replacement or snoop write-back cycle. These cycles consist of four doubleword transfers (either bursts or non-burst). The signals KEN# and WB/WT# are not sampled during write-back cycles because the processor does not attempt to redefine the cacheability of the line.

4.4.2.2 Burst Cycle Signal Protocol

The signals from ADS# through BLAST#, which are shown in Figure 4-37, have the same function and timing in both Standard Bus and Enhanced Bus modes. Burst cycles can be up to 16-bytes long (four aligned doublewords) and can start with any one of the four doublewords. The sequence of the addresses is determined by the first address and the sequence follows the order

shown in Table 4-8 on page 4-27. The burst order for reads is the same as the burst order for writes. (See Section 4.3.4.2, "Burst and Cache Line Fill Order.")

An attempted line fill caused by a read miss is indicated by the assertion of CACHE# and W/R#. For a line fill to occur, the system must assert KEN# twice: one clock prior to the first BRDY# and one clock prior to last BRDY#. It takes only one deassertion of KEN# to mark the line as non-cacheable. A write-back cycle of a cache line, due to replacement or snoop, is indicated by the assertion of CACHE# low and W/R# high. KEN# has no effect during write-back cycles. CACHE# is valid from the assertion of ADS# through the clock in which the first RDY# or BRDY# is asserted. CACHE# is deasserted at all other times. PCD behaves the same in Enhanced Bus mode as in Standard Bus mode, except that it is low during write-back cycles.

The Write-Back Enhanced IntelDX4 processor samples WB/WT# once, in the *same* clock as the first BRDY#. This sampled value of WB/WT# is combined with PWT to bring the line into the internal cache, either as a write-back line or write-through line.

4.4.3 Cache Consistency Cycles

The system performs snooping to maintain cache consistency. Snoop cycles can be performed under AHOLD, BOFF#, or HOLD, as described in Table 4-11.

Table 4-11. Snoop Cycles under AHOLD, BOFF#, or HOLD

AHOLD	Floats the address bus. ADS# is asserted under AHOLD only to initiate a snoop write-back cycle. An ongoing burst cycle is completed under AHOLD. For non-burst cycles, a specific non-burst transfer (ADS#-RDY# transfer) is completed under AHOLD and fractured before the next assertion of ADS#. A snoop write-back cycle is reordered ahead of a fractured non-burst cycle and the non-burst cycle is completed only after the snoop write-back cycle is completed, provided there are no other snoop write-back cycles scheduled.
BOFF#	Overrides AHOLD and takes effect in the next clock. On-going bus cycles will stop in the clock following the assertion of BOFF# and resume when BOFF# is de-asserted. The snoop write-back cycle begins after BOFF# is de-asserted followed by the backed-off cycle.
HOLD	HOLD is acknowledged only between bus cycles, except for a non-cacheable, non-burst code prefetch cycle. In a non-cacheable, non-burst code prefetch cycle, HOLD is acknowledged after the system asserts RDY#. Once HOLD is asserted, the processor blocks all bus activities until the system releases the bus (by de-asserting HOLD).

The snoop cycle begins by checking whether a particular cache line has been "cached" and invalidates the line based on the state of the INV pin. If the Write-Back Enhanced IntelDX4 processor is configured in Enhanced Bus mode, the system must drive INV high to invalidate a particular cache line. The Write-Back Enhanced IntelDX4 processor does not have an output pin to indicate a snoop hit to an S-state line or an E-state line. However, the Write-Back Enhanced IntelDX4 processor invalidates the line if the system snoop hits an S-state, E-state, or M-state line, provided INV was driven high during snooping. If INV is driven low during a snoop cycle, a modified line is written back to memory and remains in the cache as a write-back line; a write-through line also remains in the cache as a write-through line.

After asserting AHOLD or BOFF#, the external bus master driving the snoop cycle must wait for two clocks before driving the snoop address and asserting EADS#. If snooping is done under HOLD, the master performing the snoop must wait for at least one clock cycle before driving the

snoop addresses and asserting EADS#. INV should be driven low during read operations to minimize invalidations, and INV should be driven high to invalidate a cache line during write operations. The Write-Back Enhanced IntelDX4 processor asserts HITM# if the cycle hits a modified line in the cache. This output signal becomes valid two clock periods after EADS# is valid on the bus. HITM# remains asserted until the modified line is written back and remains asserted until the RDY# or BRDY# of the snoop cycle is asserted. Snoop operations could interrupt an ongoing bus operation in both the Standard Bus and Enhanced Bus modes. The Write-Back Enhanced IntelDX4 processor can accept EADS# in every clock period while in Standard Bus mode. In Enhanced Bus mode, the Write-Back Enhanced IntelDX4 processor can accept EADS# every other clock period or until a snoop hits an M-state line. The Write-Back Enhanced IntelDX4 processor does not accept any further snoop cycles inputs until the previous snoop write-back operation is completed.

All write-back cycles adhere to the burst address sequence of 0-4-8-C. The CACHE#, PWT, and PCD output pins are asserted and the KEN# and WB/WT# input pins are ignored. Write-back cycles can be either burst or non-burst. All write-back operations write 16 bytes of data to memory corresponding to the modified line that hit during the snoop.

NOTE

Note that the Write-Back Enhanced IntelDX4 processor accepts BS8# and BS16# line-fill cycles, but not on replacement or snoop-forced write-back cycles.

4.4.3.1 Snoop Collision with a Current Cache Line Operation

The system can also perform snooping concurrent with a cache access and may collide with a current cache bus cycle. Table 4-12 lists some scenarios and the results of a snoop operation colliding with an on-going cache fill or replacement cycle.

Table 4-12. Various Scenarios of a Snoop Write-Back Cycle Colliding with an On-Going Cache Fill or Replacement Cycle

Arbitration Control	Snoop to the Line That Is Being Filled	Snoop to a Different Line than the Line Being Filled	Snoop to the Line That Is Being Replaced	Snoop to a Different Line than the Line Being Replaced
AHOLD	Read all line fill data into cache line buffer. Update cache only if snoop occurred with INV = 0 No write-back cycle because the line has not been modified yet.	Complete fill if the cycle is burst. Start snoop write-back. If the cycle is non-burst, the snoop write-back is reordered ahead of the line fill. After the snoop write-back cycle is completed, continue with line fill.	Complete replacement write-back if the cycle is burst. Processor does not initiate a snoop write-back, but asserts HITM# until the replacement write-back is completed. If the replacement cycle is non-burst, the snoop write-back is re-ordered ahead of the replacement write-back cycle. The processor does not continue with the replacement write-back cycle.	Complete replacement write-back if it is a burst cycle. Initiate snoop write-back. If the replacement write-back is a non-burst cycle, the snoop write-back cycle is re-ordered in front of the replacement cycle. After the snoop write-back, the replacement write-back is continued from the interrupt point.
BOFF#	Stop reading line fill data Wait for BOFF# to be deasserted. Continue read from backed off point Update cache only if snoop occurred with INV = '0'.	Stop fill Wait for BOFF# to be deasserted. Do snoop write-back Continue fill from interrupt point.	Stop replacement write-back Wait for BOFF# to be deasserted. Initiate snoop write-back Processor does not continue replacement write-back.	Stop replacement write-back Wait for BOFF# to be de-asserted Initiate snoop write-back Continue replacement write-back from point of interrupt.
HOLD	HOLD is not acknowledged until the current bus cycle (i.e., the line operation) is completed, except for a non-cacheable, non-burst code prefetch cycle. Consequently there can be no collision with the snoop cycles using HOLD, except as mentioned earlier. In this case the snoop write-back is re-ordered ahead of an on-going non-burst, non-cached code prefetch cycle. After the write-back cycle is completed, the code prefetch cycle continues from the point of interrupt.			

4.4.3.2 Snoop under AHOLD

Snooping under AHOLD begins by asserting AHOLD to force the Write-Back Enhanced IntelDX4 processor to float the address bus, as shown in Figure 4-38. The ADS# for the write-back cycle is guaranteed to occur no sooner than the second clock following the assertion of HITM# (i.e., there is a dead clock between the assertion of HITM# and the first ADS# of the snoop write-back cycle).

When a line is written back, KEN#, WB/WT#, BS8#, and BS16# are ignored, and PWT and PCD are always low during write-back cycles.

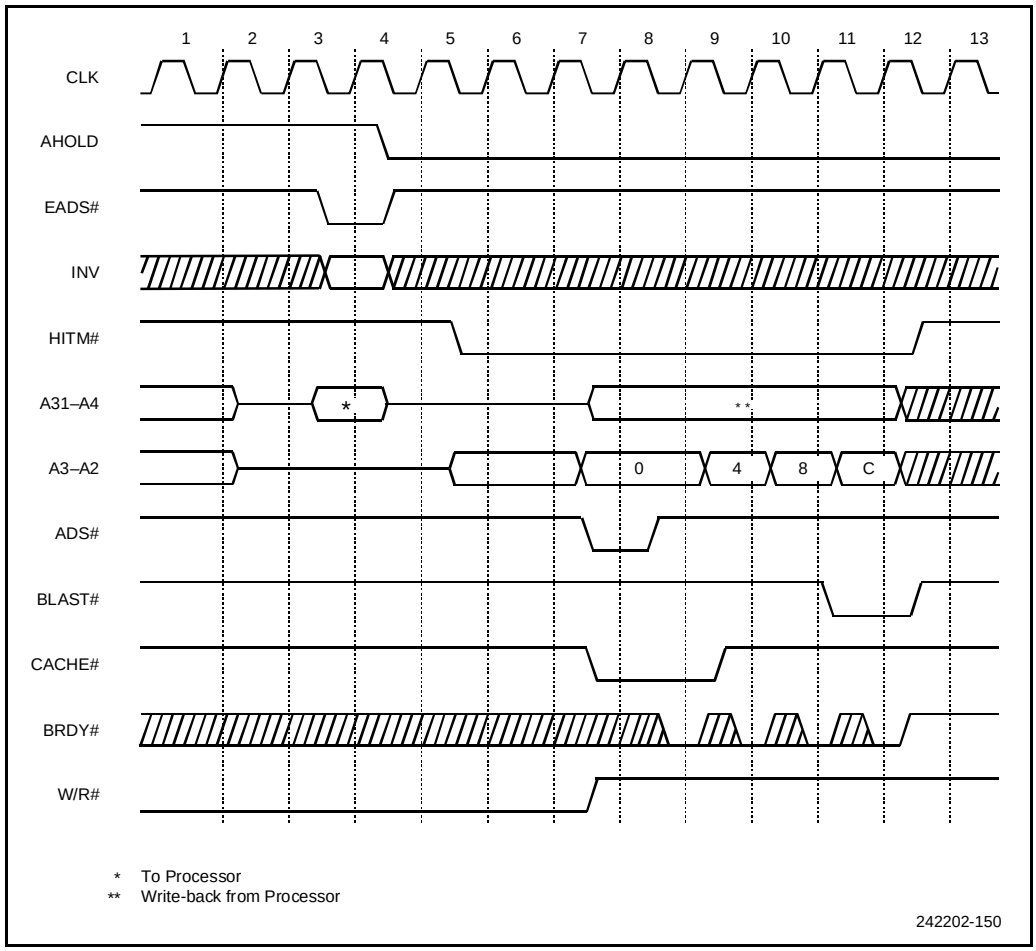


Figure 4-38. Snoop Cycle Invalidating a Modified Line

The next ADS# for a new cycle can occur immediately after the last RDY# or BRDY# of the write-back cycle. The Write-Back Enhanced IntelDX4 processor does not guarantee a dead clock between cycles unless the second cycle is a snoop-forced write-back cycle. This allows snoop-forced write-backs to be backed off (BOFF#) when snooping under AHOLD.

HITM# is guaranteed to remain asserted until the RDY# or BRDY# signals corresponding to the last doubleword of the write-back cycle is returned. HITM# is de-asserted from the clock edge in which the last BRDY# or RDY# for the snoop write-back cycle is asserted. The write-back cycle could be a burst or non-burst cycle. In either case, 16 bytes of data corresponding to the modified line that has a snoop hit is written back.

Snoop under AHOLD Overlaying a Line-Fill Cycle

The assertion of AHOLD during a line fill is allowed on the Write-Back Enhanced IntelDX4 processor. In this case, when a snoop cycle is overlaid by an on-going line-fill cycle, the chipset must generate the burst addresses internally for the line fill to complete, because the address bus has the valid snoop address. The write-back mode is more complex compared to the write-through mode because of the possibility of a line being written back. Figure 4-39 shows a snoop cycle overlaying a line-fill cycle, when the snooped line is not the same as the line being filled.

In Figure 4-39, the snoop to an M-state line causes a snoop write-back cycle. The Write-Back Enhanced IntelDX4 processor asserts HITM# two clocks after the EADS#, but delays the snoop write-back cycle until the line fill is completed, because the line fill shown in Figure 4-39 is a burst cycle. In this figure, AHOLD is asserted one clock after ADS#. In the clock after AHOLD is asserted, the Write-Back Enhanced IntelDX4 processor floats the address bus (not the Byte Enables). Hence, the memory controller must determine burst addresses in this period. The chipset must comprehend the special ordering required by all burst sequences of the Write-Back Enhanced IntelDX4 processor. HITM# is guaranteed to remain asserted until the write-back cycle completes.

If AHOLD continues to be asserted over the forced write-back cycle, the memory controller also must supply the write-back addresses to the memory. The Write-Back Enhanced IntelDX4 processor always runs the write-back with an address sequence of 0-4-8-C.

In general, if the snoop cycle overlays any burst cycle (not necessarily a line-fill cycle) the snoop write-back is delayed because of the on-going burst cycle. First, the burst cycle goes to completion and only then does the snoop write-back cycle start.

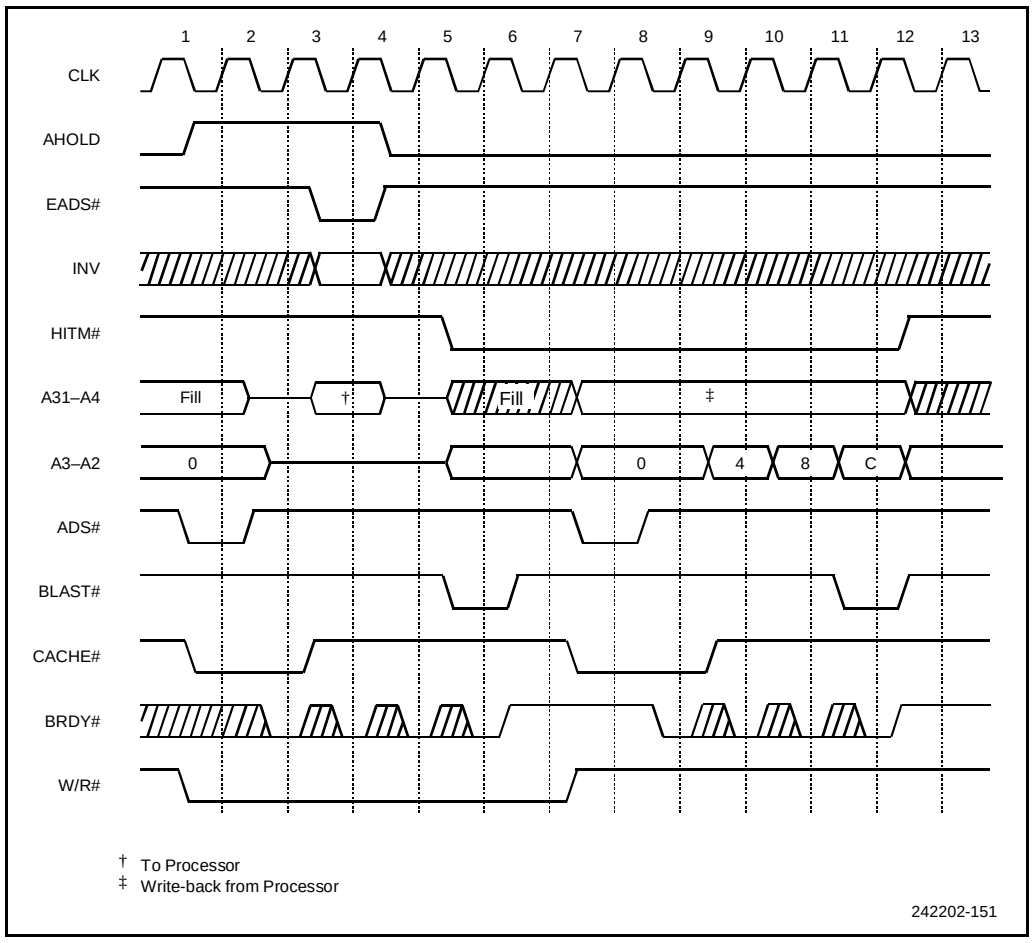


Figure 4-39. Snoop Cycle Overlaying a Line-Fill Cycle

AHOLD Snoop Overlaying a Non-Burst Cycle

When AHOLD overlays a non-burst cycle, snooping is based on the completion of the current non-burst transfer (ADS#-RDY# transfer). Figure 4-40 shows a snoop cycle under AHOLD overlaying a non-burst line-fill cycle. HITM# is asserted two clocks after EADS#, and the non-burst cycle is fractured after the RDY# for a specific single transfer is asserted. The snoop write-back cycle is re-ordered ahead of an ongoing non-burst cycle. After the write-back cycle is completed, the fractured non-burst cycle continues. The snoop write-back ALWAYS precedes the completion of a fractured cycle, regardless of the point at which AHOLD is de-asserted, and AHOLD must be de-asserted before the fractured non-burst cycle can complete.

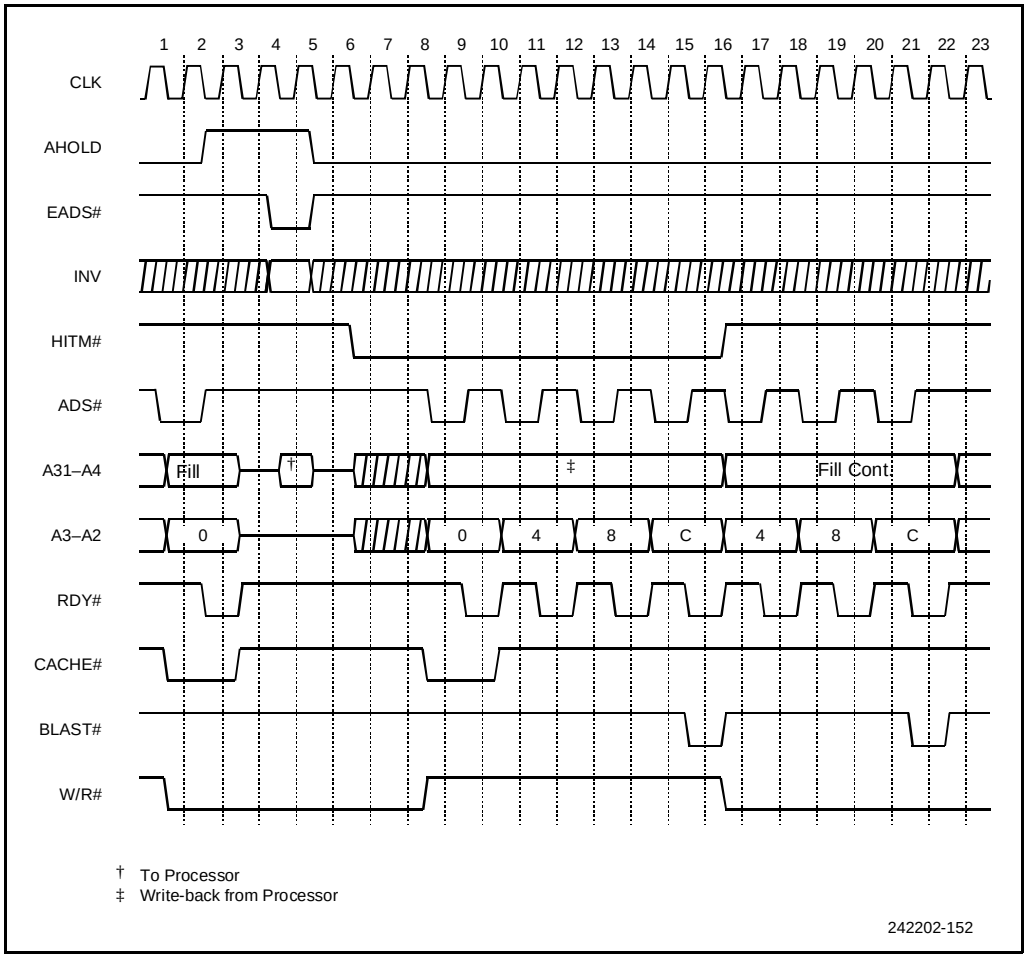


Figure 4-40. Snoop Cycle Overlaying a Non-Burst Cycle

AHOLD Snoop to the Same Line that is being Filled

A system snoop does not cause a write-back cycle to occur if the snoop hits a line while the line is being filled. The processor does not allow a line to be modified until the fill is completed (and a snoop only produces a write-back cycle for a modified line). Although a snoop to a line that is being filled does not produce a write-back cycle, the snoop still has an effect based on the following rules:

1. The processor always snoops the line being filled.
2. In all cases, the processor uses the operand that triggered the line fill.
3. If the snoop occurs when INV = “1”, the processor never updates the cache with the fill data.
4. If the snoop occurs when INV = “0”, the processor loads the line into the internal cache.

4.4.3.3 Snoop During Replacement Write-Back

If the cache contains valid data during a line fill, one of the cache lines may be replaced as determined by the Least Recently Used (LRU) algorithm. Refer to Chapter 6, “Cache Subsystem” for a detailed discussion of the LRU algorithm. If the line being replaced is modified, this line is written back to maintain cache coherency. When a replacement write-back cycle is in progress, it might be necessary to snoop the line that is being written back. (See Figure 4-41.)

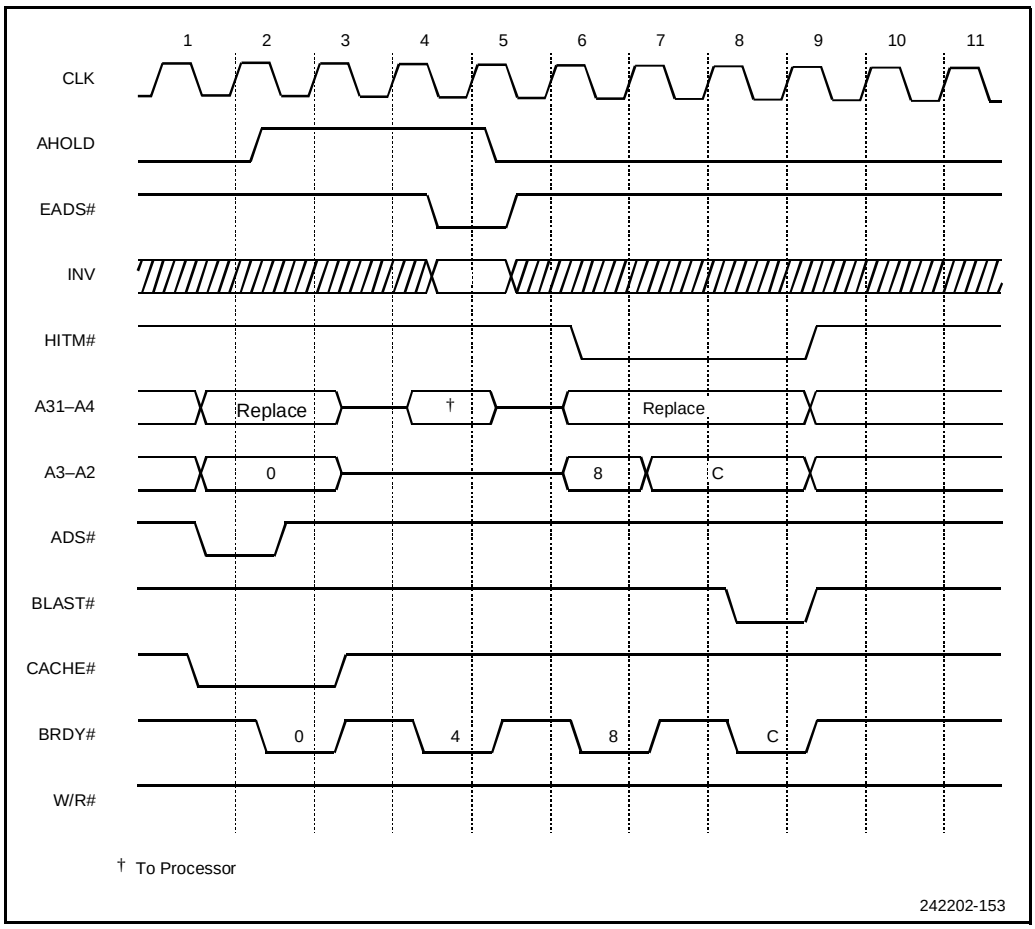


Figure 4-41. Snoop to the Line that is Being Replaced

If the replacement write-back cycle is burst and there is a snoop hit to the same line as the line that is being replaced, the on-going replacement cycle runs to completion. HITM# is asserted until the line is written back and the snoop write-back is not initiated. In this case, the replacement write-back is converted to the snoop write-back, and HITM# is asserted and de-asserted without a specific ADS# to initiate the write-back cycle.

If there is a snoop hit to a different line from the line being replaced, and if the replacement write-back cycle is burst, the replacement cycle goes to completion. Only then is the snoop write-back cycle initiated.

If the replacement write-back cycle is a non-burst cycle, and if there is a snoop hit to the same line as the line being replaced, it fractures the replacement write-back cycle after RDY# is asserted for the current non-burst transfer. The snoop write-back cycle is reordered in front of the frac-

tured replacement write-back cycle and is completed under HITM#. However, after AHOLD is deasserted, the replacement write-back cycle is not completed.

If there is a snoop hit to a line that is different from the one being replaced, the non-burst replacement write-back cycle is fractured, and the snoop write-back cycle is reordered ahead of the replacement write-back cycle. After the snoop write-back is completed, the replacement write-back cycle continues.

4.4.3.4 Snoop under BOFF#

BOFF# is capable of fracturing any transfer, burst or non-burst. The output pins (see Table 4-8 and Table 4-12) of the Write-Back Enhanced IntelDX4 processor are floated in the clock period following the assertion of BOFF#. If the system snoop hits a modified line using BOFF#, the snoop write-back cycle is reordered ahead of the current cycle. BOFF# must be de-asserted for the processor to perform a snoop write-back cycle and resume the fractured cycle. The fractured cycle resumes with a new ADS# and begins with the first uncompleted transfer. Snoops are permitted under BOFF#, but write-back cycles are not started until BOFF# is de-asserted. Consequently, multiple snoop cycles can occur under a continuously asserted BOFF#, but only up to the first asserted HITM#.

Snoop under BOFF# during Cache Line Fill

As shown in Figure 4-42, BOFF# fractures the second transfer of a non-burst cache line-fill cycle. The system begins snooping by driving EADS# and INV in clock six. The assertion of HITM# in clock eight indicates that the snoop cycle hit a modified line and the cache line is written back to memory. The assertion of HITM# in clock eight and CACHE# and ADS# in clock ten identifies the beginning of the snoop write-back cycle. ADS# is guaranteed to be asserted no sooner than two clock periods after the assertion of HITM#. Write-back cycles always use the four-doubleword address sequence of 0-4-8-C (burst or non-burst). The snoop write-back cycle begins upon the de-assertion of BOFF# with HITM# asserted throughout the duration of the snoop write-back cycle.

If the snoop cycle hits a line that is different from the line being filled, the cache line fill resumes after the snoop write-back cycle completes, as shown in Figure 4-42.

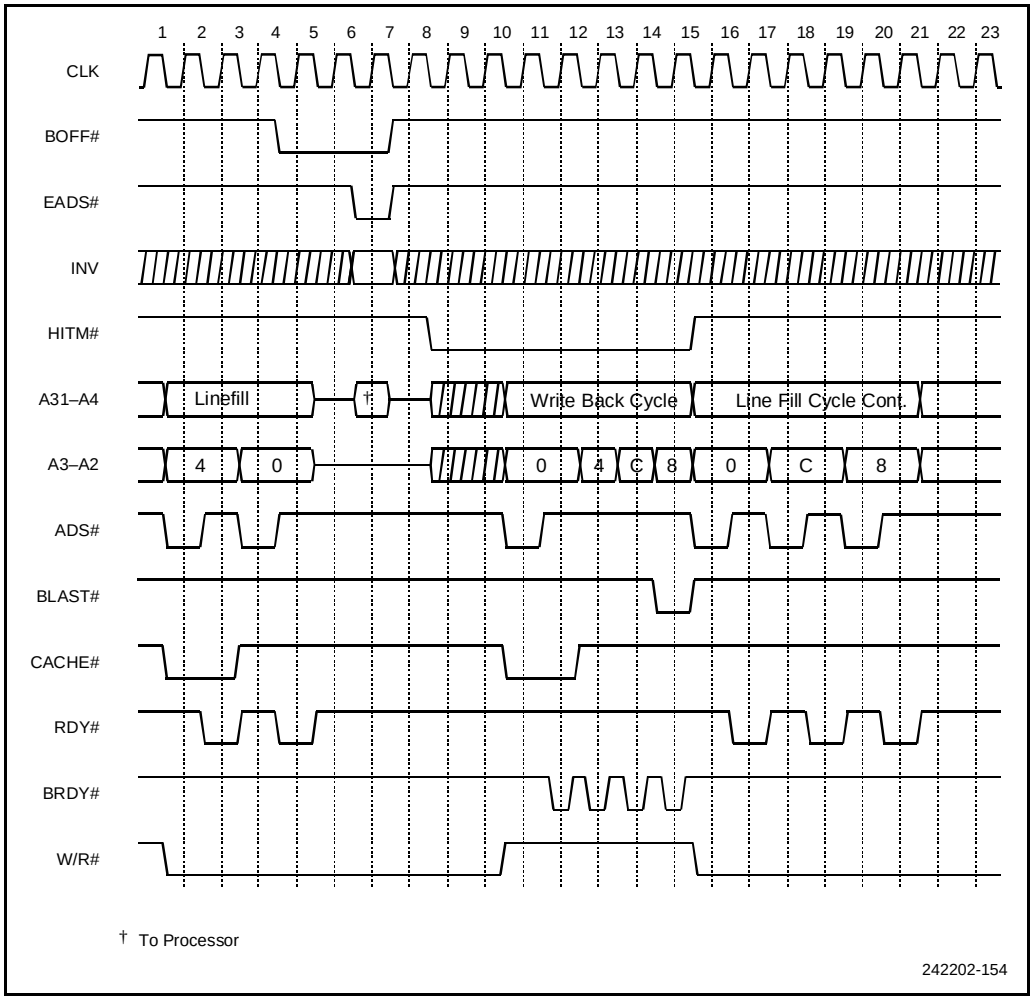


Figure 4-42. Snoop under BOFF# during a Cache Line-Fill Cycle

An ADS# is always issued when a cycle resumes after being fractured by BOFF#. The address of the fractured data transfer is reissued under this ADS#, and CACHE# is not issued unless the fractured operation resumes from the first transfer (e.g., first doubleword). If the system asserts BOFF# and RDY# simultaneously, as shown in clock four on Figure 4-42, BOFF# dominates and RDY# is ignored. Consequently, the Write-Back Enhanced IntelDX4 processor accepts only up to the x4h doubleword, and the line fill resumes with the x0h doubleword. ADS# initiates the re-sumption of the line-fill operation in clock period 15. HITM# is de-asserted in the clock period following the clock period in which the last RDY# or BRDY# of the write-back cycle is asserted. Hence, HITM# is guaranteed to be de-asserted before the ADS# of the next cycle.

Figure 4-42 also shows the system asserting RDY# to indicate a non-burst line-fill cycle. Burst cache line-fill cycles behave similarly to non-burst cache line-fill cycles when snooping using BOFF#. If the system snoop hits the same line as the line being filled (burst or non-burst), the Write-Back Enhanced IntelDX4 processor does not assert HITM# and does not issue a snoop write-back cycle, because the line was not modified, and the line fill resumes upon the de-assertion of BOFF#. However, the line fill is cached only if INV is driven low during the snoop cycle.

Snoop under BOFF# during Replacement Write-Back

If the system snoop under BOFF# hits the line that is currently being replaced (burst or non-burst), the entire line is written back as a snoop write-back line, and the replacement write-back cycle is not continued. However, if the system snoop hits a different line than the one currently being replaced, the replacement write-back cycle continues after the snoop write-back cycle has been completed. Figure 4-43 shows a system snoop hit to the same line as the one being replaced (non-burst).

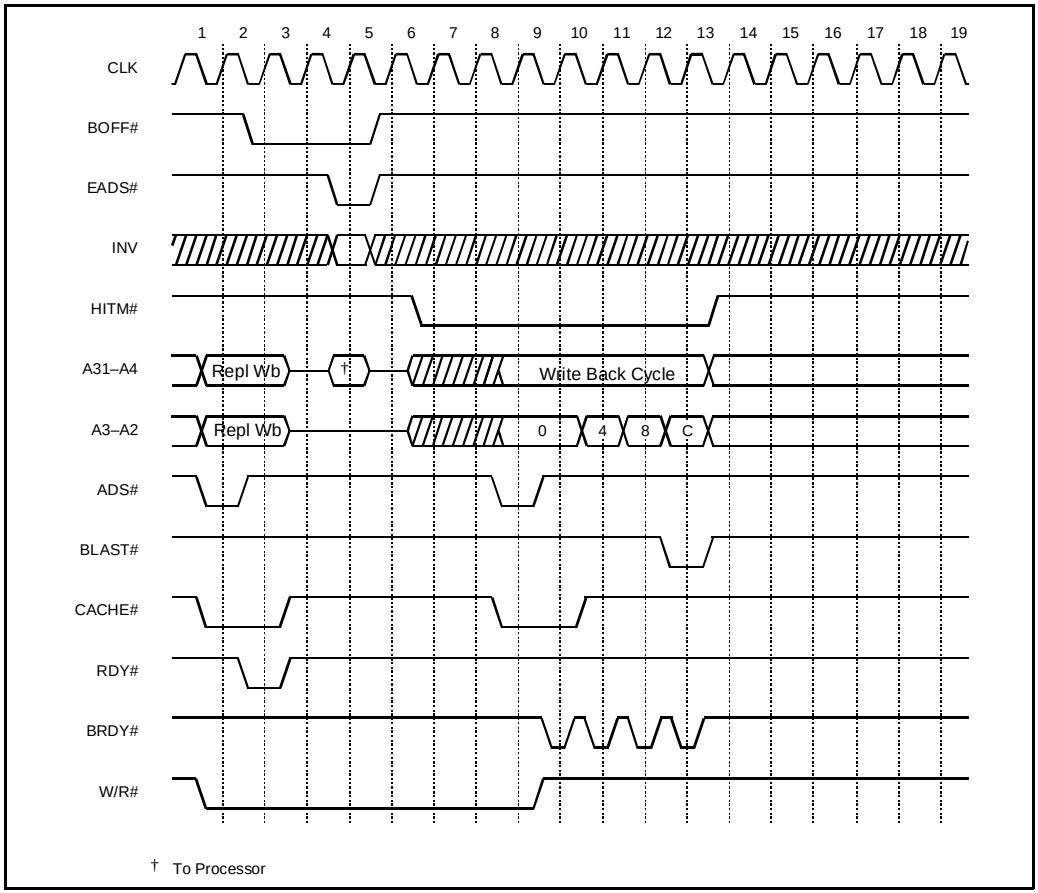


Figure 4-43. Snoop under BOFF# to the Line that is Being Replaced

4.4.3.5 Snoop under HOLD

HOLD can only fracture a non-cacheable, non-burst code prefetch cycle. For all other cycles, the Write-Back Enhanced IntelDX4 processor does not assert HLDA until the entire current cycle is completed. If the system snoop hits a modified line under HLDA during a non-cacheable, non-burstable code prefetch, the snoop write-back cycle is reordered ahead of the fractured cycle. The fractured non-cacheable, non-burst code prefetch resumes with an ADS# and begins with the first uncompleted transfer. Snoops are permitted under HLDA, but write-back cycles do not occur until HOLD is de-asserted. Consequently, multiple snoop cycles are permitted under a continuously asserted HLDA only up to the first asserted HITM#.

Snoop under HOLD during Cache Line Fill

As shown in Figure 4-44, HOLD (asserted in clock two) does not fracture the burst cache line-fill cycle until the line fill is completed (in clock five). Upon completing the line fill in clock five, the Write-Back Enhanced IntelDX4 processor asserts HLDA and the system begins snooping by driving EADS# and INV in the following clock period. The assertion of HITM# in clock nine indicates that the snoop cycle has hit a modified line and the cache line is written back to memory. The assertion of HITM# in clock nine and CACHE# and ADS# in clock 11 identifies the beginning of the snoop write-back cycle. The snoop write-back cycle begins upon the de-assertion of HOLD, and HITM# is asserted throughout the duration of the snoop write-back cycle.

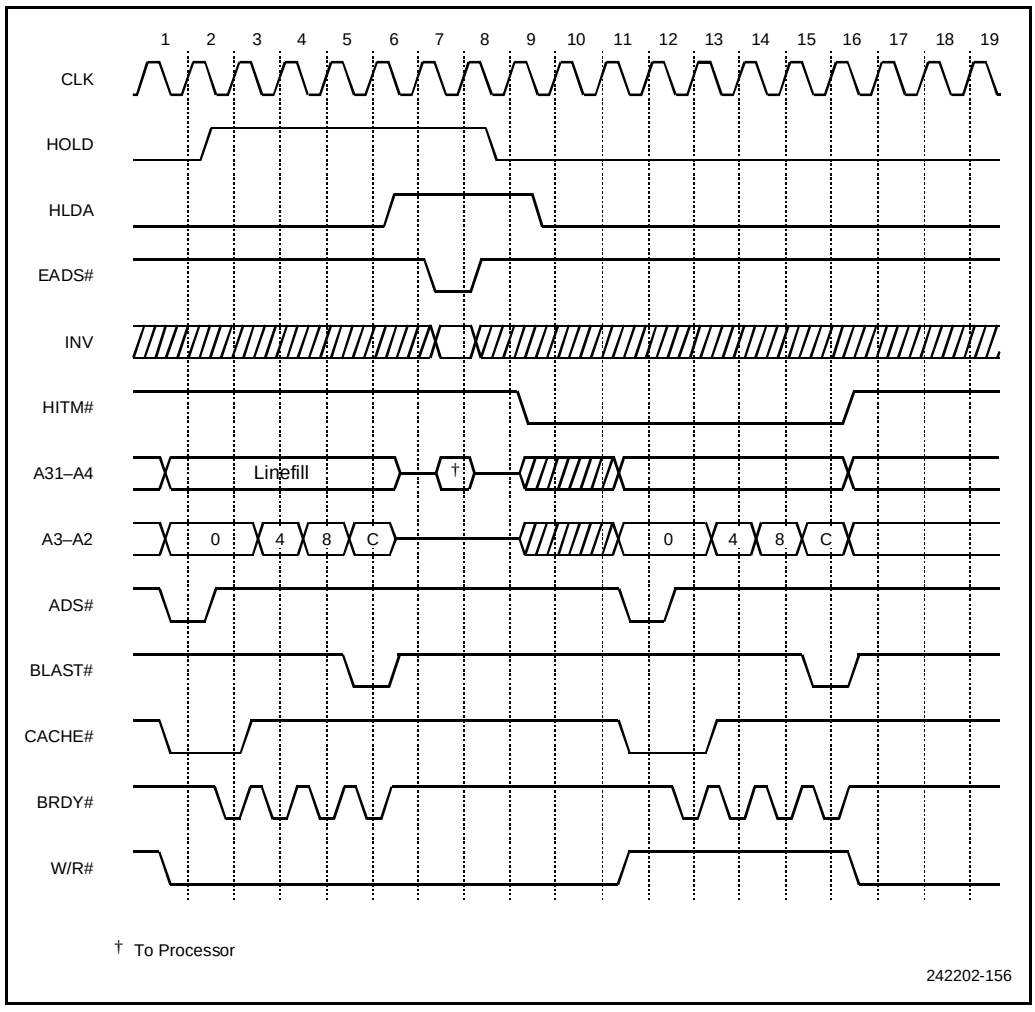


Figure 4-44. Snoop under HOLD during Line Fill

If HOLD is asserted during a non-cacheable, non-burst code prefetch cycle, as shown in Figure 4-45, the Write-Back Enhanced IntelDX4 processor issues HLDA in clock seven (which is the clock period in which the next RDY# is asserted). If the system snoop hits a modified line, the snoop write-back cycle begins after HOLD is released. After the snoop write-back cycle is completed, an ADS# is issued and the code prefetch cycle resumes.

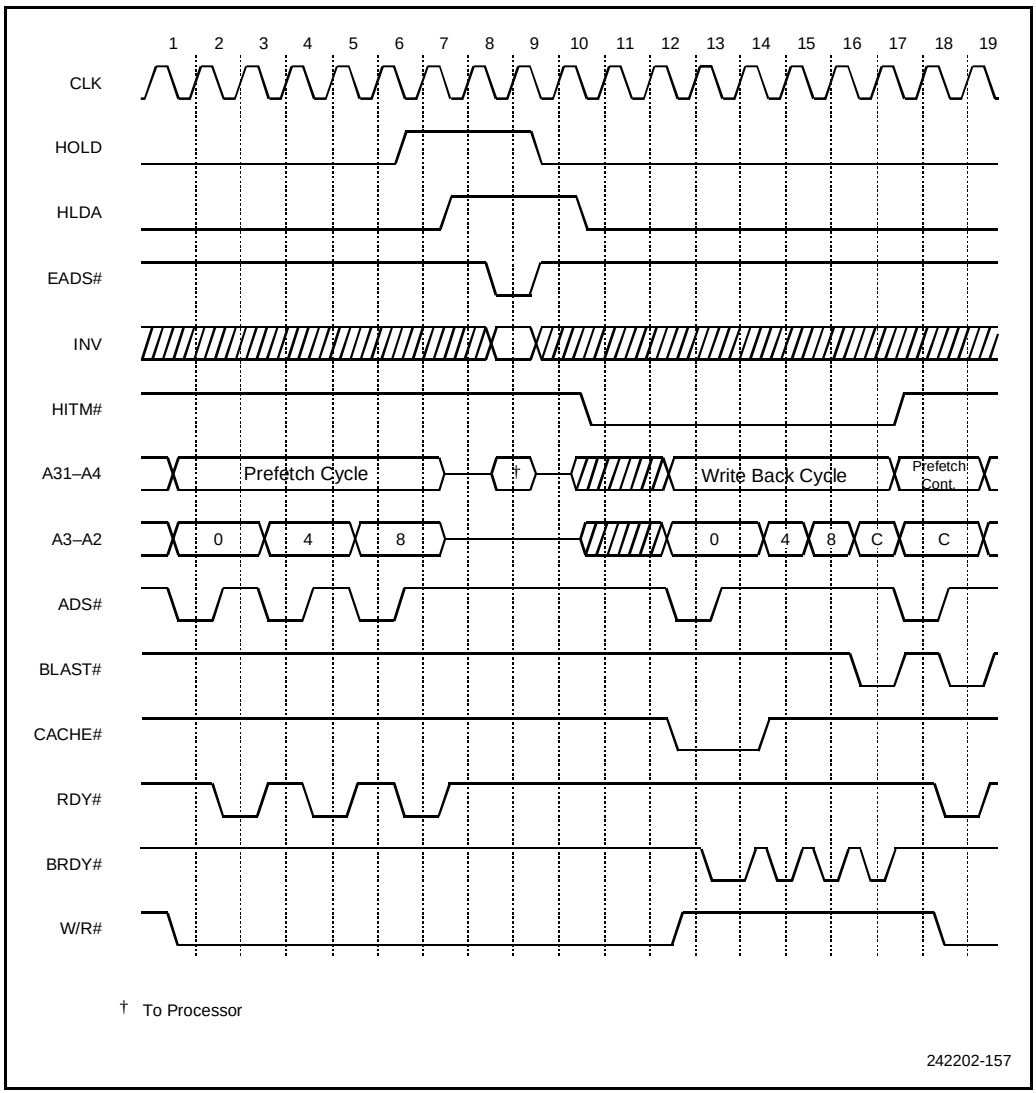


Figure 4-45. Snoop using HOLD during a Non-Cacheable, Non-Burstable Code Prefetch

4.4.3.6 Snoop under HOLD during Replacement Write-Back

Collision of snoop cycles under a HOLD during the replacement write-back cycle can never occur, because HLDA is asserted only after the replacement write-back cycle (burst or non-burst) is completed.

4.4.4 Locked Cycles

In both Standard and Enhanced Bus modes, the Write-Back Enhanced IntelDX4 processor architecture supports atomic memory access. A programmer can modify the contents of a memory variable and be assured that the variable is not accessed by another bus master between the read of the variable and the update of that variable. This function is provided for instructions that contain a LOCK prefix, and also for instructions that implicitly perform locked read modify write cycles. In hardware, the LOCK function is implemented through the LOCK# pin, which indicates to the system that the processor is performing this sequence of cycles, and that the processor should be allowed atomic access for the location accessed during the first locked cycle.

A locked operation is a combination of one or more read cycles followed by one or more write cycles with the LOCK# pin asserted. Before a locked read cycle is run, the processor first determines if the corresponding line is in the cache. If the line is present in the cache, and is in an E or S state, it is invalidated. If the line is in the M state, the processor does a write-back and then invalidates the line. A locked cycle to an M, S, or E state line is always forced out to the bus. If the operand is misaligned across cache lines, the processor could potentially run two write back cycles before starting the first locked read. In this case the sequence of bus cycles is: write back, write back, locked read, locked read, locked write and the final locked write. Note that although a total of six cycles are generated, the LOCK# pin is asserted only during the last four cycles, as shown in Figure 4-46.

LOCK# is not deasserted if AHOLD is asserted in the middle of a locked cycle. LOCK# remains asserted even if there is a snoop write-back during a locked cycle. LOCK# is floated if BOFF# is asserted in the middle of a locked cycle. However, it is driven LOW again when the cycle restarts after BOFF#. Locked read cycles are never transformed into line fills, even if KEN# is asserted. If there are back-to-back locked cycles, the Write-Back Enhanced IntelDX4 processor does not insert a dead clock between these two cycles. HOLD is recognized if there are two back-to-back locked cycles, and LOCK# floats when HLDA is asserted.

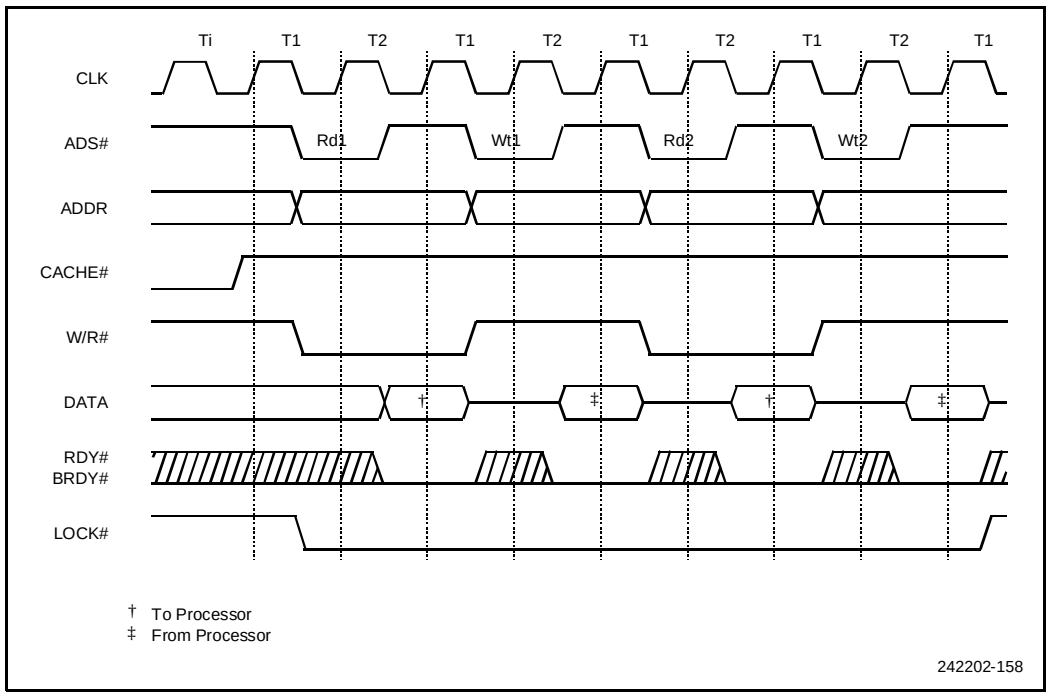


Figure 4-46. Locked Cycles (Back-to-Back)

4.4.4.1 Snoop/Lock Collision

If there is a snoop cycle overlaying a locked cycle, the snoop write-back cycle fractures the locked cycle. As shown in Figure 4-47, after the read portion of the locked cycle is completed, the snoop write-back starts under HITM#. After the write-back is completed, the locked cycle continues. But during all this time (including the write-back cycle), the LOCK# signal remains asserted.

Because HOLD is not acknowledged if LOCK# is asserted, snoop-lock collisions are restricted to AHOLD and BOFF# snooping.

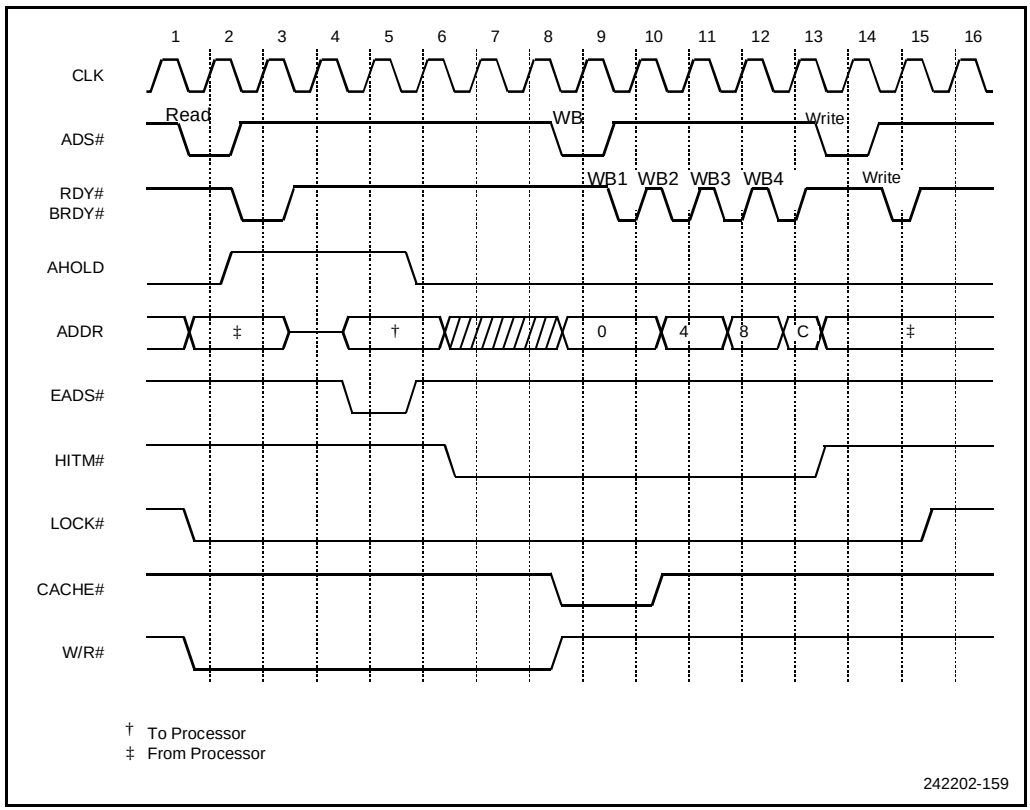


Figure 4-47. Snoop Cycle Overlaying a Locked Cycle

4.4.5 Flush Operation

The Write-Back Enhanced IntelDX4 processor executes a flush operation when the FLUSH# pin is asserted, and no outstanding bus cycles, such as a line fill or write back, are being processed. In the Enhanced Bus mode, the processor first writes back all the modified lines to external memory. After the write-back is completed, two special cycles are generated, indicating to the external system that the write-back is done. All lines in the internal cache are invalidated after all the write-back cycles are done. Depending on the number of modified lines in the cache, the flush could take a minimum of 1280 bus clocks (2560 processor clocks) and up to a maximum of 5000+ bus clocks to scan the cache, perform the write backs, invalidate the cache, and run the flush acknowledge cycles. FLUSH# is implemented as an interrupt in the Enhanced Bus mode, and is recognized only on an instruction boundary. Write-back system designs should look for the flush acknowledge cycles to recognize the end of the flush operation. Figure 4-48 shows the flush operation of the Write-Back Enhanced IntelDX4 processor when configured in the Enhanced Bus mode.

If the processor is in Standard Bus mode, the processor does not issue special acknowledge cycles in response to the FLUSH# input, although the internal cache is invalidated. The invalidation of the cache in this case, takes only two bus clocks.

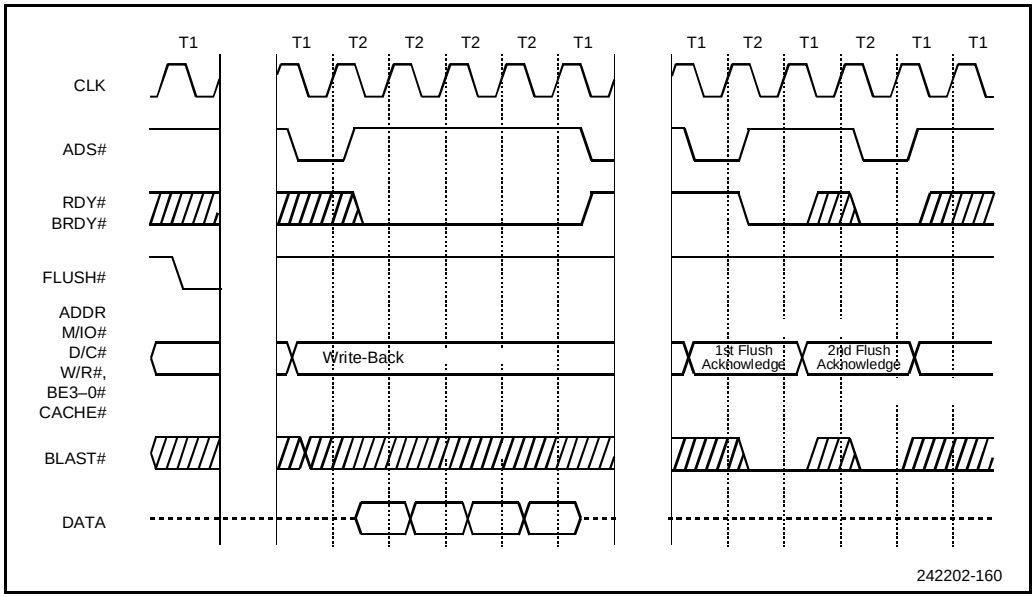


Figure 4-48. Flush Cycle

4.4.6 Pseudo Locked Cycles

In Enhanced Bus mode, PLOCK# is always deasserted for both burst and non-burst cycles. Hence, it is possible for other bus masters to gain control of the bus during operand transfers that take more than one bus cycle. A 64-bit aligned operand can be read in one burst cycle or two non-burst cycles if BS8# and BS16# are not asserted. Figure 4-49 shows a 64-bit floating-point operand or Segment Descriptor read cycle, which is burst by the system asserting BRDY#.

4.4.6.1 Snoop under AHOLD during Pseudo-Locked Cycles

AHOLD can fracture a 64-bit transfer if it is a non-burst cycle. If the 64-bit cycle is burst, as shown in Figure 4-49, the entire transfer goes to completion and only then does the snoop write-back cycle start.

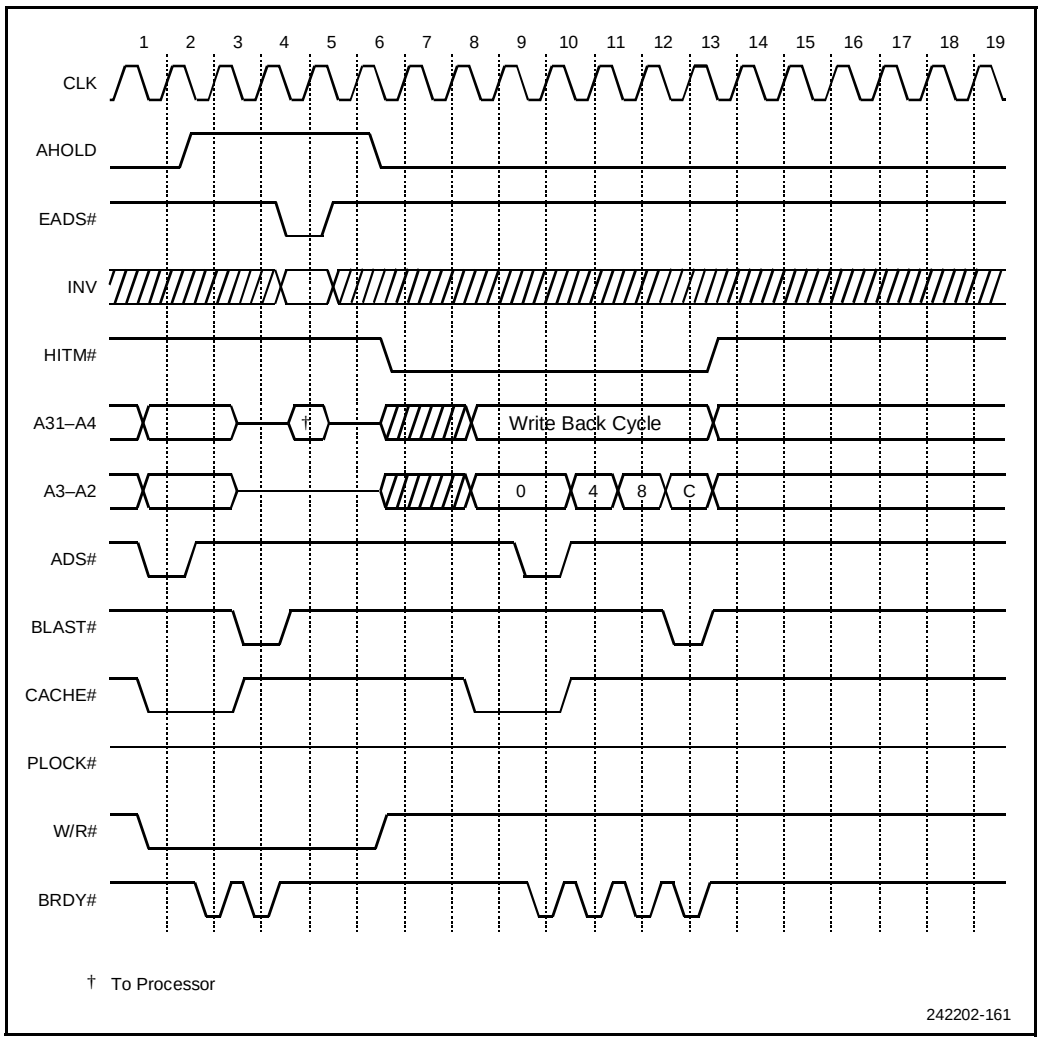


Figure 4-49. Snoop under AHOLD Overlaying Pseudo-Locked Cycle

4.4.6.2 Snoop under Hold during Pseudo-Locked Cycles

As shown in Figure 4-50, HOLD does not fracture the 64-bit burst transfer. The Write-Back Enhanced IntelDX4 processor does not issue HLDA until clock four. After the 64-bit transfer is completed, the Write-Back Enhanced IntelDX4 processor writes back the modified line to memory (if snoop hits a modified line). If the 64-bit transfer is non-burst, the Write-Back Enhanced IntelDX4 processor can issue HLDA in between bus cycles for a 64-bit transfer.

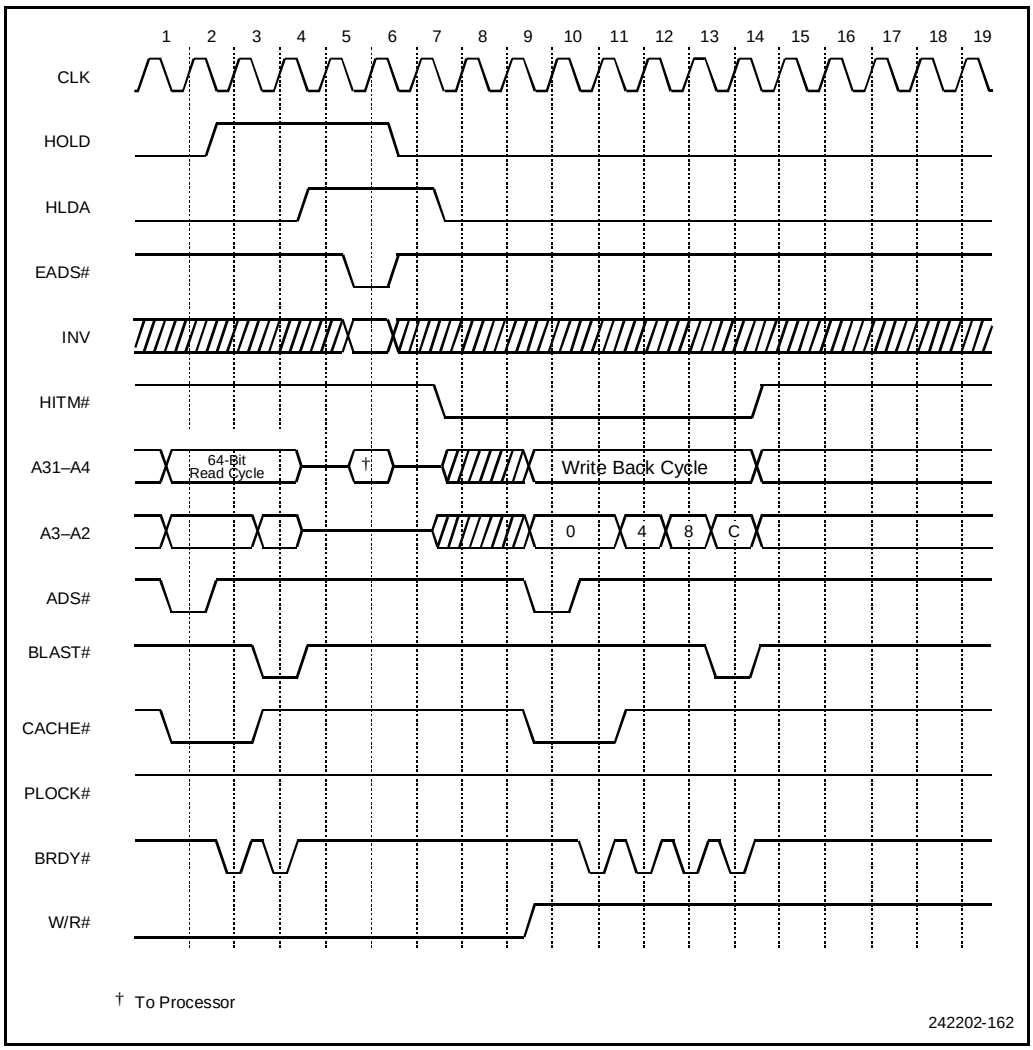


Figure 4-50. Snoop under HOLD Overlaying Pseudo-Locked Cycle

4.4.6.3 Snoop under BOFF# Overlaying a Pseudo-Locked Cycle

BOFF# is capable of fracturing any bus operation. In Figure 4-51, BOFF# fractured a current 64-bit read cycle in clock four. If there is a snoop hit under BOFF#, the snoop write-back operation begins after BOFF# is deasserted. The 64-bit write cycle resumes after the snoop write-back operation completes.

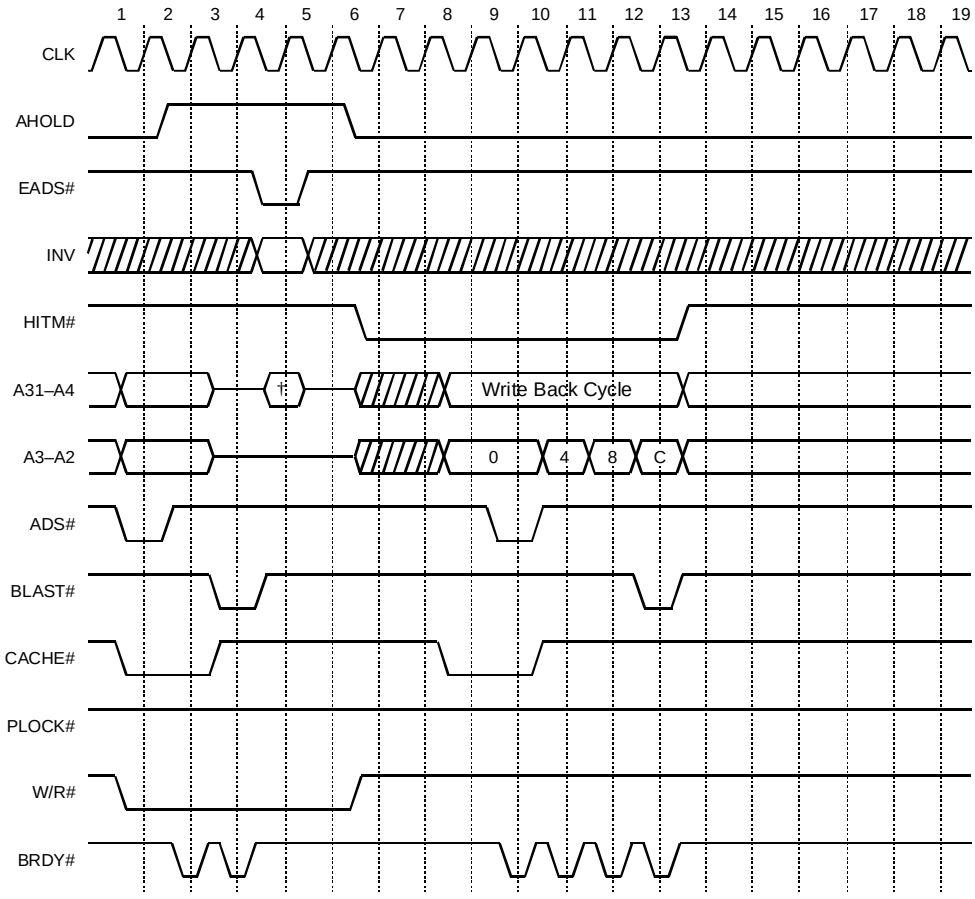


Figure 4-51. Snoop under BOFF# Overlaying a Pseudo-Locked Cycle

5

Memory Subsystem Design

Chapter Contents

CHAPTER 5
MEMORY SUBSYSTEM DESIGN

5.1 INTRODUCTION

The Intel486™ processor contains several improvements over its predecessor, the highly successful Intel386™ processor. One of the most important of these is the processor's data access rate. The Intel486 processor can access instructions and data from its on-chip cache in the same clock cycle. To support the processor's redesigned internal data path, the external bus has also been optimized and can access external memory at twice the rate of the Intel386 CPU. The internal cache requires rapid access to entire cache lines. Invalidation cycles must be supported to maintain consistency with external memory. All of these functions must be supported by the external memory system. Without them, the full performance potential of the CPU cannot be attained.

The requirements of multi-tasking and multiprocessor operating systems also place increased demand on the external memory system. OS support functions such as paging and context switching can degrade reference locality. Without efficient access to external memory, the performance of these functions is degraded.

Second-level (also known as L2) caching is a technique used to improve the memory interface. Some applications, such as multi-user office computers, require this feature to meet performance goals. Single-user systems, on the other hand, may not warrant the extra cost. Due to the variety of applications incorporating the Intel486 processor, memory system architecture is very diverse.

5.2 PROCESSOR AND CACHE FEATURE OVERVIEW

The improvements made to the processor bus interface impact the memory subsystem design. It is important to understand the impact of these features before attempting to define a memory subsystem. This section reviews the bus features that affect the memory interface.

NOTE

The Ultra-Low Power Intel486 GX processor supports only a 16-bit external data bus. The other Intel486 processors discussed in this manual feature dynamic bus sizing to accommodate 32-, 16-, and 8-bit devices.

5.2.1 The Burst Cycle

The Intel486 processor's burst bus cycle feature has more impact on the memory logic than any other feature. A large portion of the control logic is dedicated to supporting this feature. The L2 cache control is also primarily dedicated to supporting burst cycles.

To understand why the logic is designed this way, we must first understand the function of the burst cycle. Burst cycles are generated by the CPU only when two events occur. First, the CPU must request a cycle which is longer in bytes than the data bus can accommodate. Second, the BRDY# signal must be activated to terminate the cycle. When these two events occur a burst cy-

cle takes place. Note that this cycle occurs regardless of the state of the KEN# input. The KEN# input's function is discussed in the next section.

With this definition we see that several cases are included as "burstable." Some examples of burstable cycles are listed in Table 5-1. These cycle lengths are shown in bytes to clarify the case listed.

Table 5-1. Access Length of Typical CPU Functions

Bus Cycle	Size (Bytes)
All code fetches	16
Descriptor loads	8
Cacheable reads	16
Floating-point operand loads	8
Bus size 8 (16) writes	4 (Max)

The last two cases show that write cycles are burstable. In the last case a write cycle is transferred on an 8- or 16-bit bus. If BRDY# is returned to terminate this cycle, the CPU generates another write cycle without activating ADS#.

Using the burst write feature has debatable performance benefit. Some systems may implement special functions that benefit from the use of burst writes. However, the Intel486 processor does not write cache lines. Therefore, all write cycles are 4 bytes long. Most of the devices that use dynamic bus sizing are read-only. This fact further reduces the utility of burst writes.

Due to these facts, a memory subsystem design normally does not implement burst write cycles. The BRDY# input is asserted only during main memory read cycles. RDY# is used to terminate all memory write cycles. RDY# is also used for all cycles that are not in the memory subsystem or are not capable of supporting burst cycles. The RDY# input is used, for example, to terminate an EPROM or I/O cycle.

5.2.2 The KEN# Input

The primary purpose of the KEN# input is to determine whether a cycle is to be cached. Only read data and code cycles can be cached. Therefore, these cycles are the only cycles affected by the KEN# input.

Figure 5-1 shows a typical burst cycle. In this sequence, the value of KEN# is important in two different places. First, to begin a cacheable cycle, KEN# must be active the clock before BRDY# is returned. Second, KEN# is sampled the clock before BLAST# is active. At this time the CPU determines whether this line is written to the cache.

The state of KEN# also determines when read cycles can be burst. Most read cycles are initiated as 4 bytes long from the processor's cache unit. When KEN# is sampled active, the clock before BRDY# or RDY# is asserted, the cycle is converted to a 16-byte cache line fill by the bus unit. This way, a cycle which would not have been burst can now be burst by activating BRDY#.

Some read cycles can be burst without activating KEN#. The most prevalent example of this type of read cycle is a code fetch. All code fetches are generated as 16-byte cycles from the processor's

cache unit. So, regardless of the state of KEN#, code fetches are always burstable. In addition, several types of data read cycles are generated as 8-byte cycles. These cycles, mentioned previously, are descriptor loads and floating-point operand loads. These cycles can be burst at any time.

The use of the KEN# input affects performance. The design example used in Figure 5-1 illustrates one way to use this signal effectively.

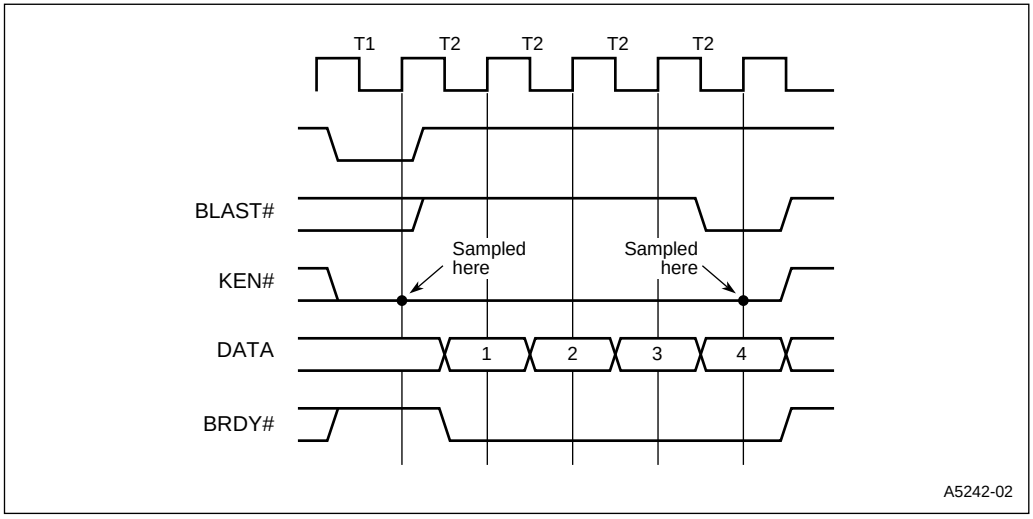


Figure 5-1. Typical Burst Cycle

The primary concern when using KEN# is generating it in time for zero wait state read cycles. Most main memory cycles are zero wait state if an L2 cache is implemented. The access to main memory is one wait state during most read cycles. Any cache access takes place with zero wait states. KEN# must, therefore, be valid during the first T2 of any read cycle.

Once this requirement is established, a problem arises. Decode functions are inherently asynchronous. Therefore, the decoded output that generates KEN# must be synchronized. If it is not, the CPU's setup and hold times are violated and internal metastability results. With synchronization, the delay required to generate KEN# will be at least three clocks. In the example shown, four clocks are required. In either case the KEN# signal will not be valid before BRDY# is returned for zero or one wait state cycles.

This problem is resolved if KEN# is made active. Figure 5-2 illustrates this function. In this diagram KEN# is active during the first two clocks of the burst cycle. If this is a data read cycle, KEN# being active at this time causes it to be converted to a 16-byte length. The decode and synchronization of KEN# takes place during the first two T2 states of the cycle. If the cycle turns out to be non-cacheable, KEN# is deactivated in the third T2. Otherwise KEN# is left active and the retrieved data is written to the cache.

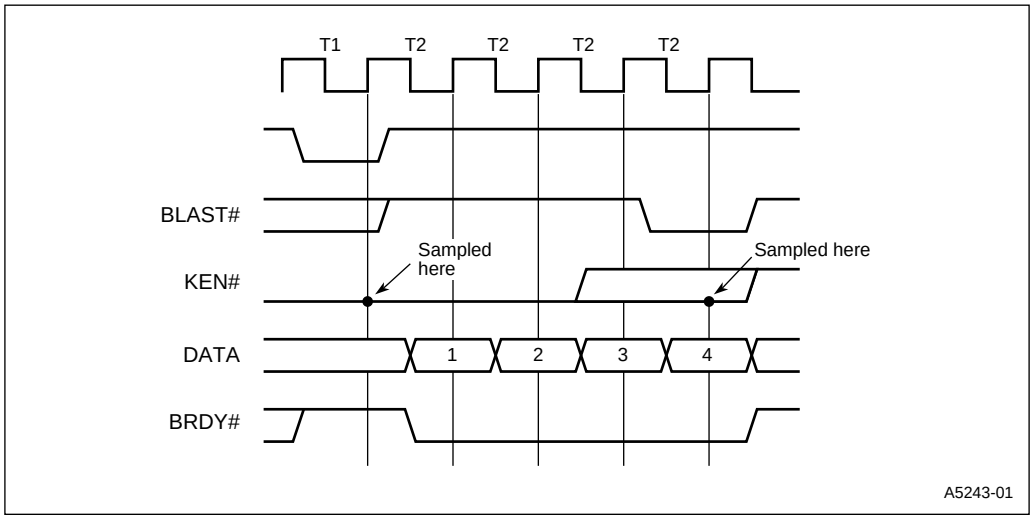


Figure 5-2. Burst Cycle: KEN# Normally Active

Some memory devices may be slow enough that 16-byte cycles are undesirable. In this case more than three wait states exist. The KEN# signal can be deactivated prior to returning RDY# or BRDY# if three or more wait states are present. As a result, these slow cycles are not converted to 16-byte cache line fills.

5.2.3 Bus Characteristics

The internal cache causes other effects that impact the memory subsystem design. Perhaps the most obvious of these is the effect on bus traffic. The fact that the internal cache uses the write-through policy dramatically increases the number of write bus cycles. Figure 5-3 illustrates this effect. The chart on the left shows the bus cycle mix for an application executed with the Intel386 DX CPU. The chart on the right shows the same application executed with the Intel486 processor. The percentage of write bus cycles jumps to 70% from 30% when this application is executed with the Intel486 processor.

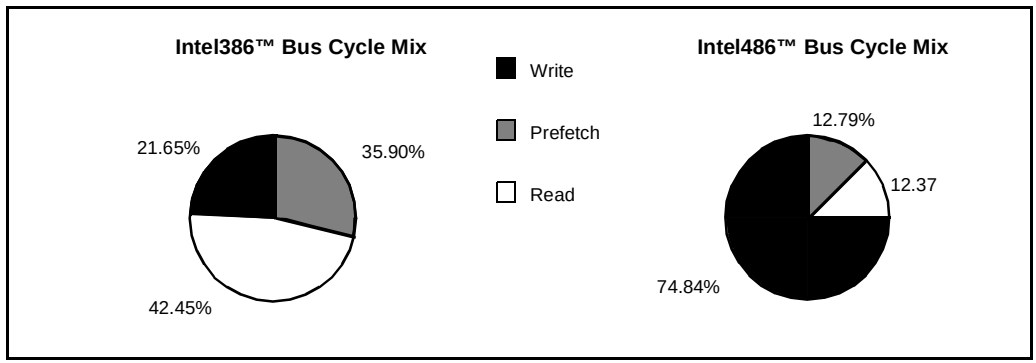


Figure 5-3. Intel386™ Processor Bus Cycle Mix/Intel486™ Processor Bus Cycle Mix

It seems obvious that many of these write cycles would be consecutive. In fact, 70% of all write cycles are consecutive. Furthermore, 50% of all write cycles occur three in-a-row. It is obvious from these statistics that optimizing the memory subsystem for write cycles can improve performance. But it is important to optimize the memory system for consecutive write cycles. Improving individual write cycle latency does not buy much performance improvement if subsequent write cycles suffer.

5.2.4 Improving Write Cycle Latency

5.2.4.1 Interleaving

The interleaving technique is used to support the burst bus feature of the Intel486 processor. The use of this technique allows the DRAM to supply a dword every clock during burst cycles. Interleaving proves to be very useful in Intel486 processor memory designs. Without its use, DRAM timings such as T_{PC} (Page Mode Cycle time) and T_{CP} (CAS Precharge time) would prevent zero wait state access at 33 MHz.

5.2.4.2 Write Posting

Analysis has shown that, in general, 6% degradation in performance can be expected for every additional wait state added to write cycles. This analysis was performed by measuring the CPU clocks required to execute several applications.

A technique called write posting can be used to improve write cycle latency. Write posting uses data registers that hold write data during write cycles. This technique allows consecutive write cycles to be overlapped. It also allows write cycles to be overlapped with L2 cache cycles and reduces overall write miss latency.

Using the write posting technique adds complexity to the system logic. It is important to determine the performance improvement realized by using this technique. This question is especially pertinent when we consider the logic already implemented in the Intel486 processor to improve write performance. The internal Intel486 write buffers decouple the processor execution unit from the external bus.

Write posting can improve average write latency to under 3 clocks for many applications. This improvement is important in Intel486 processor-based systems because approximately 70% of all bus cycles are writes. Without using a latency improvement technique such as write posting, average write latency is above 15 clocks. From this data we can conclude that approximately a 9% performance improvement can be obtained using write posting.

This improvement may increase due to other effects. Write cycles, particularly DRAM page misses, can be overlapped with read hit cycles in the L2 cache. This fact greatly reduces the delay caused by read cycles which immediately follow write cycles.

Analysis of this memory subsystem design has shown that use of these features has resulted in a low latency response to the CPU. The following characteristics have been recorded over several important applications. The average clock cycles required to complete the first read is 3.5 clocks. Subsequent cycles of a burst are always processed in one clock. Write cycles average 2.5 clocks. These average counts result from the DRAM access rates in Table 5-2. Read accesses from the cache always occur in zero wait states.

Table 5-2. Clock Latencies for DRAM Functions

DRAM Function	First Access Burst	Subsequent Burst	Write Cycles
Page hit	3	1	2
Page miss	7	1	5*

*Latency only incurred for back-to-back cycles.

5.2.5 Second-Level Cache

Several different types of L2 cache architectures are possible candidates for use with the Intel486 processor. For single CPU systems the different architectures offer similar performance benefits in most cases. The reason they are so similar is the mechanism which improves performance. The primary benefit of the L2 cache is bus cycle latency reduction.

In most systems that incorporate a single Intel486 processor, bus traffic from other bus masters is minimal. With most memory systems, the CPU uses at most 50% to 70% of the bus. Therefore reduction of bus cycle latency is the only performance benefit external logic can offer.

An L2 cache is an economical method of reducing read cycle latency and can be implemented as a system option. To provide this capability, a cache device can be configured as a look-aside cache that monitors the CPU address and control signals. When a cycle occurs in which the cache can supply data, it intervenes. The cache device could then supply an entire 16-byte line with no wait states.

The performance improvement offered by an L2 cache is substantial in some environments. This performance improvement is particularly obvious when executing multi-tasking, multi-user operating systems such as UNIX*, OS/2*, Windows 95*, Windows NT*, and Windows CE*. Some applications, however, may not require the performance improvement offered by the cache. In these cases, implementing the L2 cache as a system option is attractive.

By designing the cache subsystem as an option both users requirements can be met. A single-system design can be manufactured for both customers. The operating system user can add the cache module. Users can choose the system configuration which meets their price-performance needs.

6

Cache Subsystem

Chapter Contents

CHAPTER 6
CACHE SUBSYSTEM

6.1 INTRODUCTION

Cache is an important means of improving overall system performance. The Intel486™ processors have an on-chip, unified code and data cache. The on-chip cache is used for both instruction and data accesses and operates on physical addresses. The Intel486 processor and most variants have an 8-Kbyte cache (the IntelDX4 processor has a 16-Kbyte cache) which is organized in a 4-way set associative manner. To understand cache philosophy and the system advantages of a cache, many issues must be considered.

This chapter discusses the following related cache issues:

- Cache theory and the impact of cache on performance.
- The relationship between cache size and hit rates when using a first-level cache.
- Issues in mapping (or associativity) that arise when main memory is cached. Different cache configurations including direct-mapped, set associative, and fully associative. They are discussed along with the performance trade-offs inherent to each configuration.
- The impact of cache line sizes and cache re-filling algorithms on performance.
- Write-back and write-through methods for updating main memory. How each method maintain cache consistency and the impact on external bus utilization.
- Cache consistency issues that arise when a DMA occurs while the Intel486 processor's cache is enabled. Methods that ensure cache and main memory consistency during cache accesses.
- Cache used in single versus multiple CPU systems.

6.2 CACHE MEMORY

Cache memory is high-speed memory that is placed between microprocessors and main memory. Cache memory keeps copies of main memory that are currently in use to speed microprocessor access to requested data and instructions. When properly implemented, cache access time can be three to eight times faster than that of main memory, and thus can reduce the overall access time. Cache also reduces the number of accesses to main memory DRAM, which is important to systems with multiple bus masters that all access that same memory. This section introduces the cache concept and discusses memory performance benefits provided by a cache.

6.2.1 What is a Cache?

A cache memory is a smaller high-speed memory that fits between a CPU and slower main memory. Cache memory is important in increasing computer performance by reducing total memory latency. A cache memory consists of a directory (or tag), and a data memory. Whenever the CPU is required to read or write data, it first accesses the tag memory and determines if a cache hit has

occurred, implying that the requested word is present in the cache. If the tags do not match, the data word is not present in the cache. This is called a cache miss. On a cache hit, the cache data memory allows a read operation to be completed more quickly from its faster memory than from a slower main memory access. The hit rate is the percentage of the accesses that are hits, and is affected by the size and organization of the cache, the cache algorithm used, and the program running. An effective cache system maintains data in a way that increases the hit rate. Different cache organizations are discussed later in this chapter. The main advantage of cache is that a larger main memory appears to have the high speed of a cache. For example, a zero-wait state cache that has a hit rate of 90 percent makes main memory appear to be zero-wait state memory for 9 out of 10 accesses.

Programs usually address memory in the neighborhood of recently accessed locations. This is called program locality or locality of reference and it is locality that makes cache systems possible. Code, data character strings, and vectors tend to be sequentially scanned items or items accessed repeatedly, and cache helps the performance in these cases. In some cases the program locality principle does not apply. Jumps in code sequences and context switching are some examples.

6.2.2 Why Add an External Cache?

System designers must take into account several factors when deciding whether to incorporate a Level II cache subsystem in an embedded Intel486 processor design. These considerations include the performance expectations, operating system used, DRAM cycle speed, possible future upgrades to the initial application, and system costs. Although the Intel486 processor-based personal computer often required a 256-K to 512-K L2 cache for optimal performance, embedded applications have a wide variety of performance and cost requirements and their L2 cache needs vary accordingly. In many applications, an inexpensive 32-K or 64-K cache provides good performance, whereas the additional performance provided by a 512-K cache would be too costly to justify. When possible, system designers should run the application code on a standard Intel486 processor-based personal computer (assuming the operating system is compatible) and take performance measurements with the L2 cache first enabled, then disabled in the BIOS. Although this technique for performance evaluation is not perfect, it gives the applications team a good basis upon which to make design decisions.

6.3 CACHE TRADE-OFFS

Cache efficiency is the cache's ability to keep the code and data most frequently used by the microprocessor. Cache efficiency is measured in terms of the hit rate. Another indication of cache efficiency is system performance; this is the time in which the microprocessor can perform a certain task and is measured in effective bus cycles. An efficient cache reduces external bus cycles and enhances overall system performance. Hit rates are discussed in the next section.

Factors that can affect a cache's performance are:

- Size: Increasing the cache size allows more items to be contained in the cache. Cost is increased, however, and a larger cache cannot operate as quickly as a smaller one.

- Associativity (discussed in Section 6.2.2, “Why Add an External Cache?”): Increased associativity increases the cache hit rate but also increases its complexity and reduces its speed.
- Line Size: The amount of data the cache must fetch during each cache line replacement (every miss) affects performance. More data takes more time to fill a cache line, but then more data is available and the hit rate increases.
- Write-Back and Write Posting: The ability to write quickly to the cache and have the cache then write to the slower memory increases performance. Implementing these types of cache designs can be very complex, however.
- Features: Adding features such as write-protection (to be able to cache ROM memory), bus watching, and multiprocessing protocols can speed a cache but increases cost and complexity.
- Speed: Not all cache return data to the CPU as quickly as possible. It is less expensive and complex to use slower cache memories and cache logic.

6.3.1 Cache Size and Performance

Hit rates for various first-level cache configurations are shown in Table 6-1. These statistics are conservative because they illustrate the lowest hit rates generated by analyzing several main-frame traces. The hit rates are not absolute quantities, and the hit rate of a particular configuration is software-dependent. However, the table allows a meaningful comparison of the various cache configurations. It also indicates the degree of hardware complexity needed to arrive at a particular cache efficiency. Table 6-1 presents direct-mapped, 2-way, and 4-way set associative cache, which are all discussed in the next section.

Table 6-1. Level-1 Cache Hit Rates (Sheet 1 of 2)

Cache Configurations			Hit Rate
Size	Associativity	Line Size	
1 Kbyte	direct	4 bytes	41%
8 Kbyte	direct	4 bytes	73%
16 Kbyte	direct	4 bytes	81%
32 Kbyte	direct	4 bytes	86%

Table 6-1. Level-1 Cache Hit Rates (Sheet 2 of 2)

Cache Configurations			Hit Rate
Size	Associativity	Line Size	
32 Kbyte	2-way	4 bytes	87%
32 Kbyte	direct	8 bytes	91%
64 Kbyte	direct	4 bytes	88%
64 Kbyte	2-way	4 bytes	89%
64 Kbyte	4-way	4 bytes	89%
64 Kbyte	direct	8 bytes	92%
64 Kbyte	2-way	8 bytes	93%
128 Kbyte	direct	4 bytes	89%
128 Kbyte	2-way	4 bytes	89%
128 Kbyte	direct	8 bytes	93%

Program behavior is another important factor in determining cache efficiency. If a program uses a piece of data only once, then the cache may spend all its time thrashing or replacing itself with new data from memory. This is common in vector processing. The processor receives no added efficiency from the cache because main memory is being requested frequently. In such instances, the user can consider mapping the data entries as non-cacheable.

Cache system performance can be calculated based on the main memory access time, the cache access time, the miss rate, and the write cycle time.

C_S is defined as the ratio of the cache system access time to the main memory access time. C_S is a dimensionless number but provides a useful measure of the cache performance.

$C_a = (1-M)T_c + MT_m$
$C_s = C_a/T_m = (1-M)(T_c/T_m) + M = (1-M)C_m+M$

where:

C_a = average cache system cycle time averaged over reads and writes
T_c = cache cycle time
T_m = main memory cycle time
M = miss rate = 1-hit rate
C_s = cache system access time as a fraction of main memory access time
C_m = cache memory access time as compared to main memory cycle time

If the cache always misses, then M=1 and C_m=1, and the main memory access is equal to the effective access time of the cache. If the cache is infinitely fast, then C_m is equal to the miss rate. Because the cache access time is finite, the cache system access time approaches the cache access time as the miss rate approaches zero.

While the above discussion applies to read operations, it can be easily extended to write operations, which also affect system performance. When memory is written to, the CPU must wait for the completion of the write cycle before proceeding to the next instruction. In a buffered memory system, where posted writes occur, data can be loaded in a register, and the memory can be up-

dated later. This allows the CPU to begin the next cycle without being delayed by the main memory write access time. Both these memory updating techniques are discussed later in this chapter.

6.3.2 Associativity and Performance Issues

Data and instructions are written into the cache by a function that maps the main memory address into a cache location. The placement policy determines the mapping function from the main memory address to the cache location. There are four policies to consider: fully associative, direct-mapped, set associative, and sector buffering.

Fully Associative: A fully associative cache system provides maximum flexibility in determining which blocks are stored in the cache at any time. Ideally, the blocks of words in the cache would contain the main memory locations needed most by the processor regardless of the distance between the words in main memory. The size of a block in the cache is also known as the line size, and corresponds to the width of a cache word. For example, a block can be eight bytes for a 32-bit processor, in which case two doublewords are accessed each time the cache line is filled. In the example shown in Figure 6-1, the block size is one doubleword.

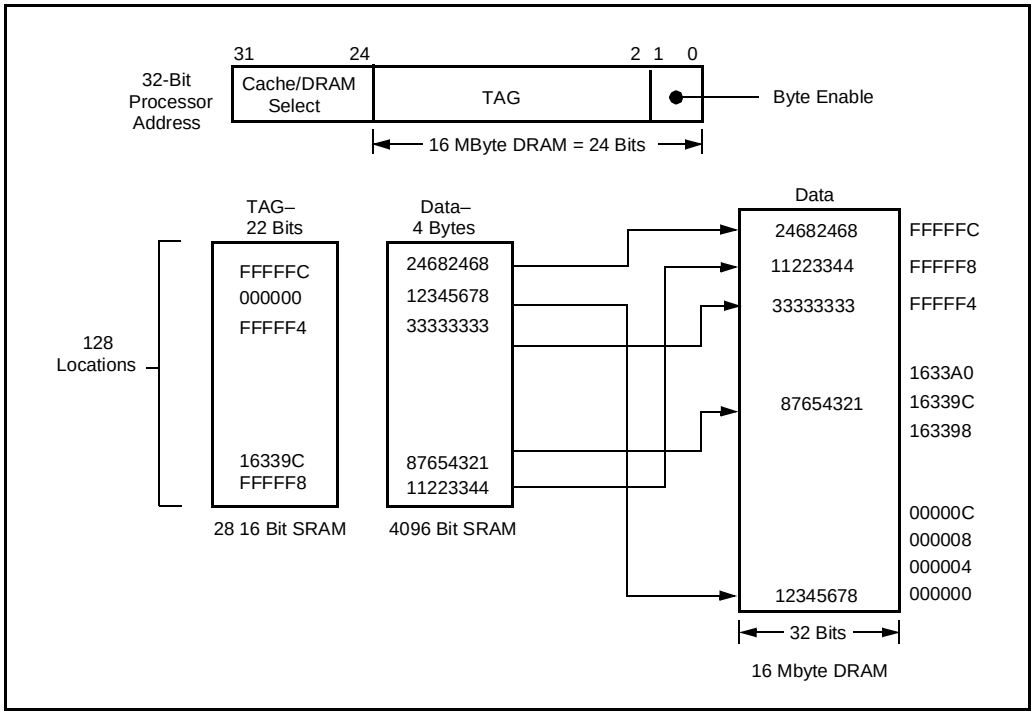


Figure 6-1. A Fully Associative Cache Organization

Because there is no single relationship between all of the addresses in the 64 blocks, the cache would have to store the entire address of each block. When the processor requests data, the cache

controller would have to compare the address with each of the 64 addresses in the cache for a match condition. This organization, shown in Figure 6-1, is called fully associative.

Direct Mapped: In a direct mapped cache, the simplest of the three policies, only one address comparison is required to determine if the requested word is in the cache. This is because each block in the cache maps to only one location in the cache. A direct mapped cache address has two parts: a cache index field, which specifies the block's location in the cache, and a tag field that distinguishes blocks within a particular cache location.

For example, consider a 64-Kbyte direct mapped cache that contains 16-Kbyte 32-bit locations and cache 16 Mbytes of memory. The cache index field must include 14 bits to select one of the 16-Kbyte blocks in cache plus two bits to decode one of the four byte enables. The tag field must be eight bits wide to identify one of the 256 blocks that can occupy the selected cache location. The most significant eight bits of the address are decoded to select the cache subsystem from other memories in the memory space. The direct-mapped cache organization is shown in Figure 6-2.

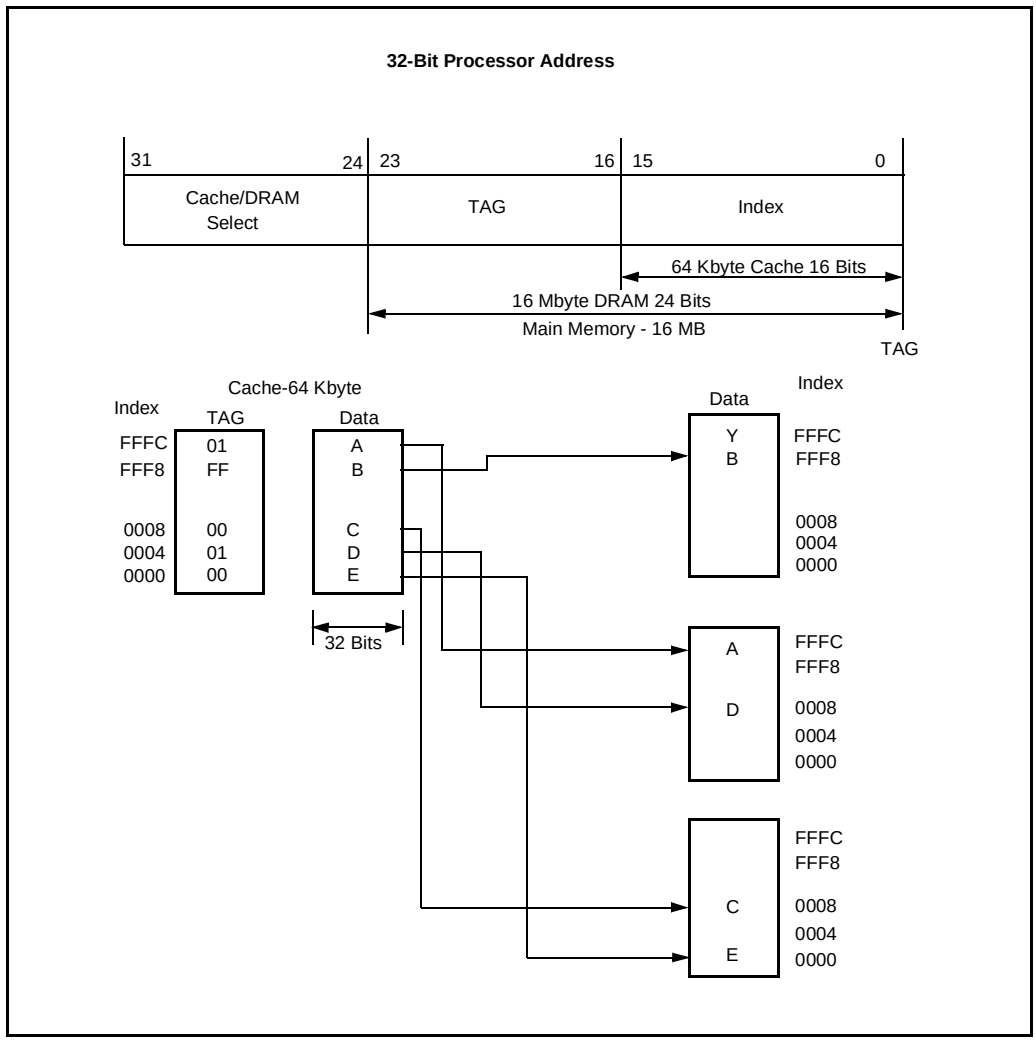


Figure 6-2. Direct Mapped Cache Organization

If the processor requests data at FFFFF8, the first step is to send the least significant 14 bits of FFF8 to the cache tag RAM. If the tag field stored at FFF8 is FF (as shown in the diagram), then a hit has occurred and the data word ‘‘B’’ is sent to the CPU. If the requested word has 020004, then the tags would not match. In this case the tag RAM would be updated with the value 02 corresponding to the index 0004, and the data ‘‘D’’ would be replaced by the word at location 020004.

If the processor accesses locations that have the same index bits, then the cache would have to be updated constantly. This type of program behavior is infrequent, however, so a direct mapped

cache may provide acceptable performance at a lower cost when compared to a fully associative cache memory.

Set Associative: The set-associative cache is a compromise between the fully associative and direct-mapped cache. The set-associative cache has more than one set and it is equivalent to several direct mapped cache operating in parallel. For each cache index there are several block locations allowed, and the block can be placed in any set or retrieved from any set. Figure 6-3 shows a two-way set associative cache memory.

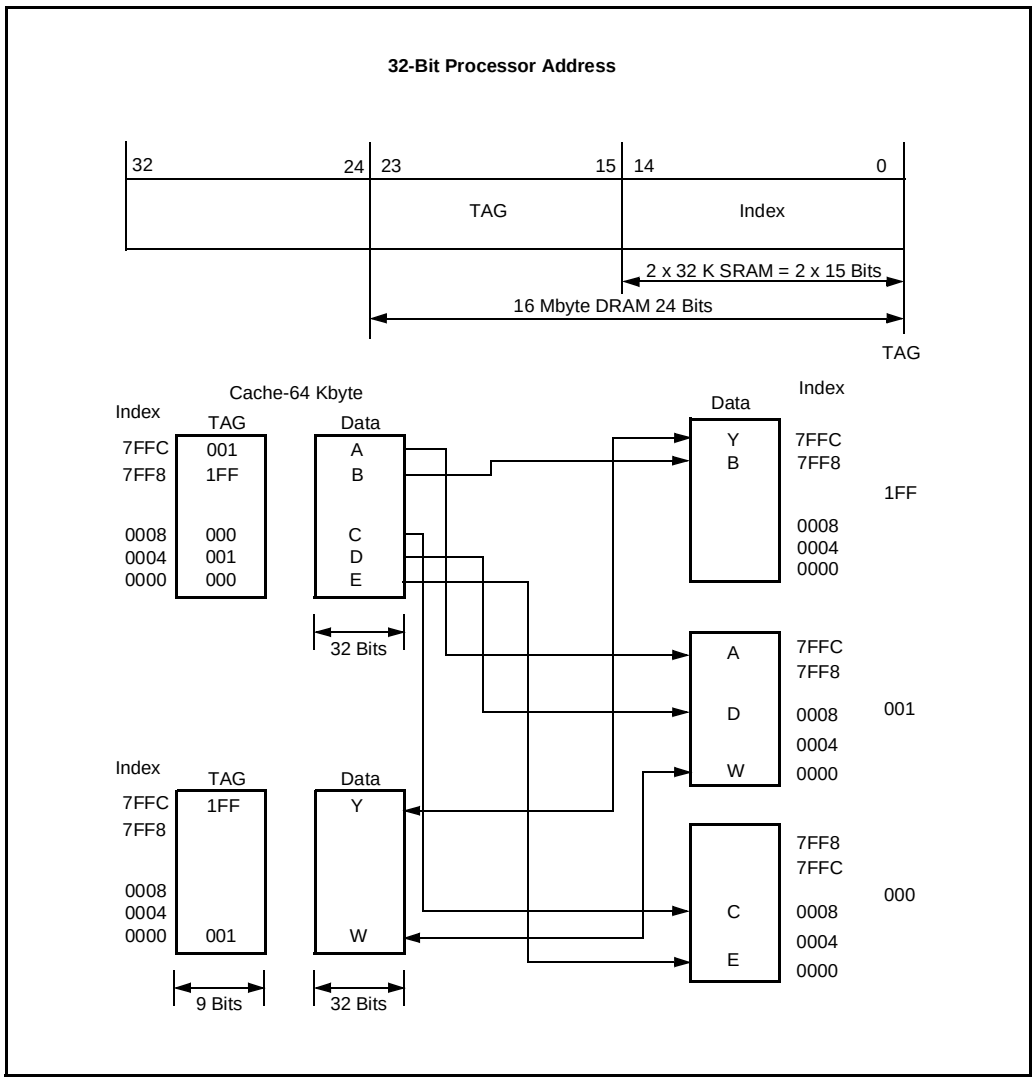


Figure 6-3. Two-Way Set Associative Cache Organization

Given an equal amount of cache memory as in the direct mapped example, the set associative cache has half as many locations, and the extra address bit becomes part of the tag field. Because the set-associative cache has several places for a block with the same cache index, the hit rate is increased. The set associative cache performs more efficiently than a direct mapped cache, but it needs a wider tag field and additional logic to determine which set should receive the data. This function is determined by the replacement policy, which is covered later in this section.

Sector Buffering: Another cache configuration uses a sector buffer and is sometimes called a sub-block cache. The cache is a number of sectors, and the sectors in turn are a number of blocks. Each block can have its own valid bit, but only one tag address exists per sector. When a word is accessed whose sector is in the cache but the block is not, then the block is fetched from the main memory. Sector buffering has its own trade-offs associated with miss ratios and bus utilization. Having smaller blocks increases the miss ratio, but reduces the number of external bus accesses. Conversely, having a large number of blocks increases the hit ratio but also increases the external bus utilization. Figure 6-4 shows the cache organization in sector buffering.

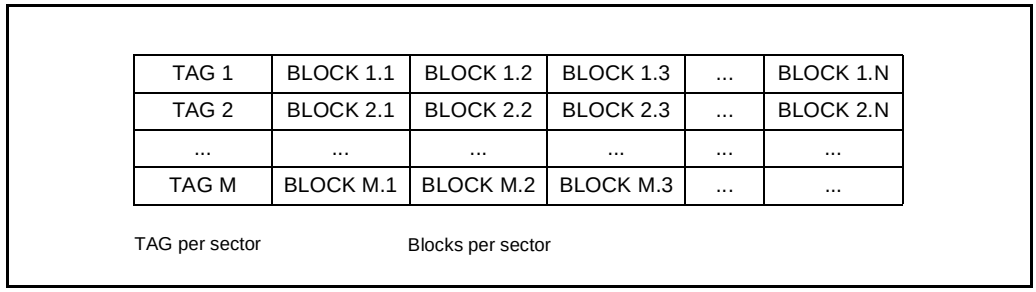


Figure 6-4. Sector Buffer Cache Organization

The Intel486 processor's on-board cache is organized 4-way set associative with a line size of 16 bytes. The 8-Kbyte cache is organized as four 2-Kbyte sets. Each 2-Kbyte set is comprised of 128 16-byte lines. Figure 6-5 shows the cache organization. An application can achieve an extremely high hit rate with the 4-way associativity. The cache is transparent so that the Intel486 processor remains software-compatible with its non-cache predecessors.

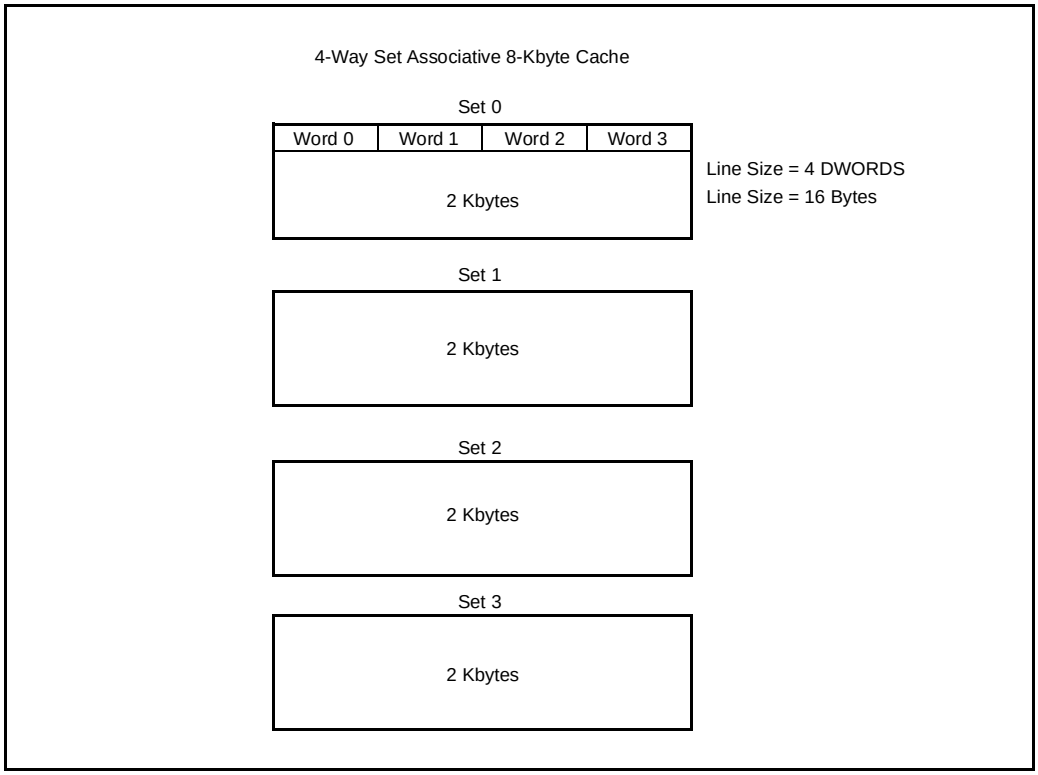


Figure 6-5. The Cache Data Organization for the Intel486™ Processor's On-Chip Cache

6.3.3 Block/Line Size

Block size is an important consideration in cache memory design. Block size is also referred to as the *line size*, or the width of the cache data word. The block size may be larger than the word, and this can impact the performance, because the cache may be fetching and storing more information than the CPU needs.

As the block size increases, the number of blocks that fit in the cache is reduced. Because each block fetch overwrites the older cache contents, some blocks are overwritten shortly after being fetched. In addition, as block size increases, additional words are fetched with the requested word. Because of program locality, the additional words are less likely to be needed by the processor.

When a cache is refilled with four dwords or eight words on a miss, the performance is dramatically better than a cache size that employs single-word refills. Those extra words that are read into the cache, because they are subsequent words and because programs are generally sequential in nature, are likely be hits in subsequent cache accesses. Also, the cache refill algorithm is a significant performance factor in systems in which the delay in transferring the first word from the main memory is long but in which several subsequent words can be transferred in a shorter time.

This situation applies when using page mode accesses in dynamic RAM; and the initial word is read after the normal access time, whereas subsequent words can be accessed quickly by changing only the column addresses. Taking advantage of this situation while selecting the optimum line size can greatly increase cache performance.

6.3.4 Replacement Policy

In a set-associative cache configuration, a replacement policy is needed to determine which set should receive new data when the cache is updated. There are four common approaches for choosing which block (or single word) within a set is be overwritten. These are the least recently used (LRU) method, the pseudo LRU method, the first-in first-out (FIFO) method, and the random method.

In the LRU method, the set that was least recently accessed is overwritten. The control logic must maintain least recently used bits and must examine the bits before an update occurs. In the pseudo LRU method, the set that was assumed to be the least recently accessed is overwritten. In the FIFO method, the cache overwrites the block that is resident for the longest time. In the random method, the cache arbitrarily replaces a block. The performance of the algorithms depends on the program behavior. The LRU method is preferred because it provides the best hit rate.

6.4 UPDATING MAIN MEMORY

When the processor executes instructions that modify the contents of the cache, changes have to be made in the main memory as well; otherwise, the cache is only a temporary buffer and it is possible for data inconsistencies to arise between the main memory and the cache. If only one of the cache or the main memory is altered and the other is not, two different sets of data become associated with the same address. A potential situation of incorrect or stale data is shown in Figure 6-6. There are two general approaches to updating the main memory. The first is the write-through method; and the second is the write-back, also known as copy-back method. Memory traffic issues are discussed for both methods.

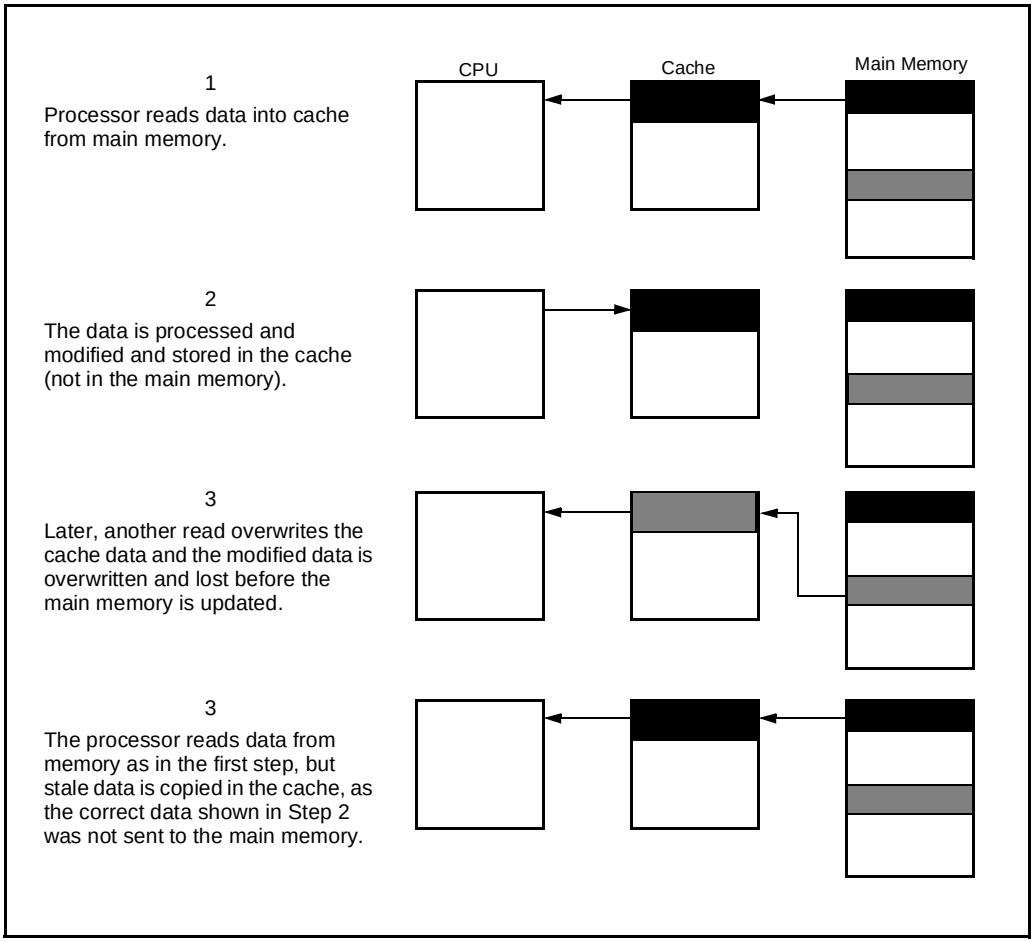


Figure 6-6. Stale Data Problem in the Cache/Main Memory

6.4.1 Write-Through and Buffered Write-Through Systems

In a write-through system, data is written to the main memory immediately after or while it is written into the cache. As a result, the main memory always contains valid data. The advantage to this approach is that any block in the cache can be overwritten without data loss, while the hardware implementation remains fairly straightforward. There is a memory traffic trade-off, however, because every write cycle increases the bus traffic on a slower memory bus. This can create contention for use of the memory bus by other bus masters. Even in a buffered write-through scheme, each write eventually goes to memory. Thus, bus utilization for write cycles is not reduced by using a write-through or buffered write-through cache.

Users sometimes adopt a buffered write-through approach in which the write accesses to the main memory can be buffered with a N-deep pipeline. A number of words are stored in pipelined registers, and will subsequently be written to the main memory. The processor can begin a new operation before the write operation to main memory is completed. If a read access follows a write access, and a cache hit occurs, then data can be accessed from the cache memory while the main memory is updated. When the N-deep pipeline is full, the processor must wait if another write access occurs and the main memory has not yet been updated. A write access followed by a read miss also requires the processor to wait because the main memory has to be updated before the next read access.

Pipeline configurations must account for multiprocessor complications when another processor accesses a shared main memory location which has not been updated by the pipeline. This means the main memory hasn't been updated, and the memory controller must take the appropriate action to prevent data inconsistencies.

6.4.2 Write-Back System

In a write-back system, the processor writes data into the cache and sets a "dirty bit" which indicates that a word had been written into the cache but not into the main memory. The cache data is written into the main memory at a later time and the dirty bit is cleared. Before overwriting any word or block in the cache, the cache controller looks for a dirty bit and updates the main memory before loading the cache with the new data.

A write-back cache accesses memory less often than a write-through cache because the number of times that the main memory must be updated with altered cache locations is usually lower than the number of write accesses. This results in reduced traffic on the main memory bus.

A write-back cache can offer higher performance than a write-through cache if writes to main memory are slow. The primary use of the a write-back cache is in a multiprocessing environment. Since many processors must share the main memory, a write-back cache may be required to limit each processor's bus activity, and thus reduce accesses to main memory. It has been shown that in a single-CPU environment with up to four clock memory writes, there is no significant performance difference between a write-through and write-back cache.

There are some disadvantages to a write-back system. The cache control logic is more complex because addresses have to be reconstructed from the tag RAM and the main memory has to be updated along with the pending request. For DMA and multiprocessor operations, all locations with an asserted dirty bit must be written to the main memory before another device can access the corresponding main memory locations.

6.4.3 Cache Consistency

Write-through and write-back systems require mechanisms to eliminate the problem of stale main memory in a multiprocessing system or in a system with a DMA controller. If the main memory is updated by one processor, the cache data maintained by another processor may contain stale data. A system that prevents the stale data problem is said to maintain cache consistency. There are four methods commonly used to maintain cache consistency: snooping (or bus watching), broadcasting (or hardware transparency), non-cacheable memory designation, and cache flushing.

In snooping, cache controllers monitor the bus lines and invalidate any shared locations that are written by another processor. The common cache location is invalidated and cache consistency is maintained. This method is shown in Figure 6-7.

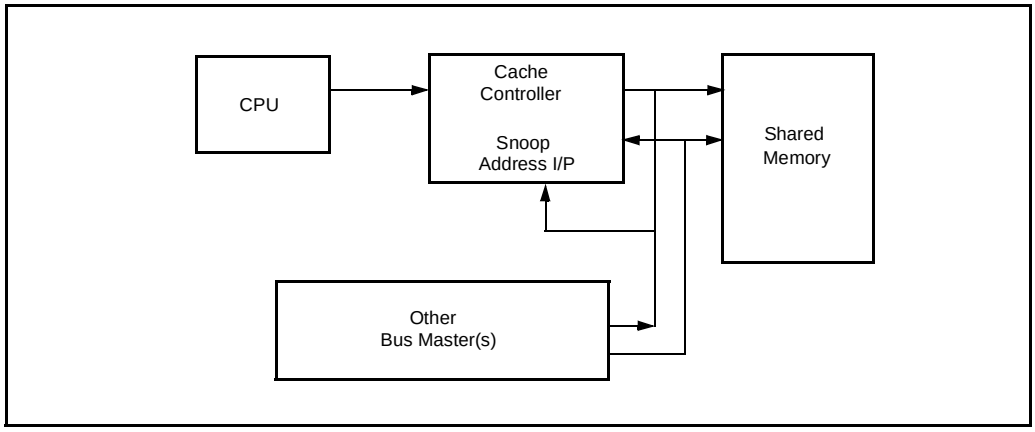


Figure 6-7. Bus Watching/Snooping for Shared Memory Systems

In broadcasting/hardware transparency, the addresses of all stores are transmitted to all the other cache so that all copies are updated. This is accomplished by routing the accesses of all devices to main memory through the same cache. Another method is by copying all cache writes to main memory and to all of the cache that share main memory. A hardware transparent system is shown in Figure 6-8.

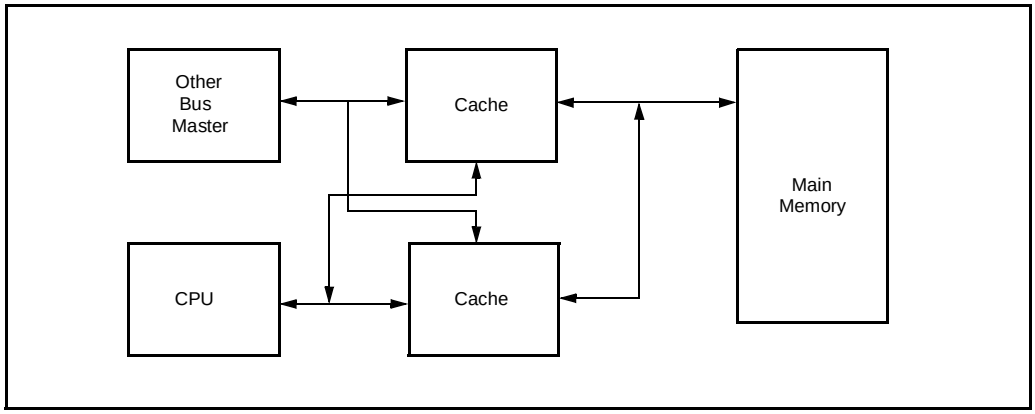


Figure 6-8. Hardware Transparency

In non-cacheable memory systems, all shared memory locations are considered non-cacheable. In such systems, access to the shared memory is never copied in the cache, and the cache never

receives stale data. This can be implemented with chip select logic or with the high order address bits. Figure 6-9 shows non-cacheable memory.

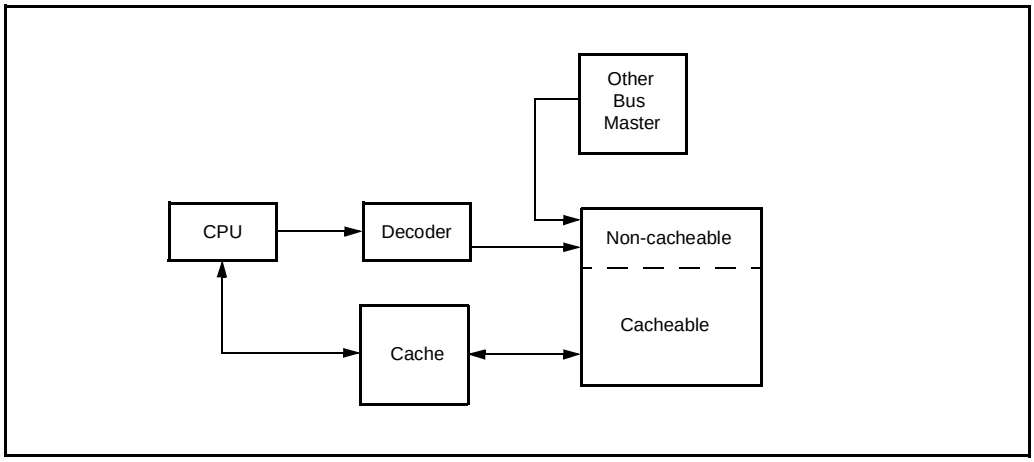


Figure 6-9. Non-Cacheable Share Memory

In cache flushing, all cache locations with set dirty bits are written to main memory (for write-back systems), then cache contents are cleared. If all of the devices are flushed before another bus master writes to shared memory, cache consistency is maintained.

Combinations of various cache coherency techniques may be used in a system to provide an optimal solution. A system may use hardware transparency for time critical I/O operations such as paging, and it may partition the memory as non-cacheable for slower I/O operations such as printing.

6.5 NON-CACHEABLE MEMORY LOCATIONS

To avoid cache consistency problems, certain memory locations must not be cached. The PC architecture has several special memory areas which may not be cached. If ROM locations on add-in cards are cached, for example, write operations to the ROM can alter the cache while main memory contents remain the same. Further, if the mode of a video RAM subsystem is switched, it can produce altered versions of the original data when a read-back is performed. Expanded memory cards may change their mapping, and hence memory contents, with an I/O write operation. LAN or disk controllers with local memory may change the memory contents independent of the Intel486 processor. This altering of certain memory locations can cause a cache consistency problem. For these reasons, the video RAM, shadowed BIOSROMs, expanded memory boards, add-in cards, and shadowed expansion ROMs should be non-cacheable locations. Depending on the system design, ROM locations may be cacheable in a second-level cache if write protection is allowed.

6.6 CACHE AND DMA OPERATIONS

Some of the issues related to cache consistency in systems employing DMA have already been covered in the preceding section. Because a DMA controller or other bus master can update main memory, there is a possibility of stale data in the cache. The problem can be avoided through snooping, cache transparency, and non-cacheable designs.

In snooping, the cache controller monitors the system address bus and invalidates cache locations that will be written to during a DMA cycle. This method is advantageous in that the processor can access its cache during DMA operations to main memory. Only a “snoop hit” causes an invalidation cycle (or update cycle) to occur.

In cache transparency, memory accesses through the CPU and the DMA controller are directed through the cache, requiring minimal hardware. However, the main disadvantage is that while a DMA operation is in progress, the CPU bus is placed in HOLD. The concurrency of CPU/cache and DMA controller/main memory operations is not supported.

In non-cacheable designs, a separate dual-ported memory can be used as the non-cacheable portion of the memory, and the DMA device is tightly coupled to this memory. In this way, the problem of stale data cannot occur.

In all of the approaches, the cache should be made software transparent so that DMA cycles do not require special software programming to ensure cache coherency.

6.7 CACHE FOR SINGLE VERSUS MULTIPLE PROCESSOR SYSTEMS

6.7.1 Cache in Single Processor Systems

In single CPU systems, a write-through cache is an ideal cache solution. Write-through cache solves consistency issues, may be designed as a plug-in option, and is less expensive. The main drawback of a write-through cache is its inability to reduce main memory utilization for write cycles. However, this is not as critical a consideration to single CPU designs.

6.7.2 Cache in Multiple Processor Systems

The Intel486 processor is designed for multiple-processor applications. The BREQ output permits a simple hardware interface for bus arbitration. The on-board and second-level caches have a high hit rate and reduce main memory accesses for reads. Each microprocessor may have its own local cache or all the microprocessors may share a global cache. With multi-masters, bus utilization is critical. When a write-back cache is used, the bus utilization is reduced compared to a write-through cache for write operations.

The multi-processor system illustrated in Figure 6-10 shows two processors and a DMA controller that are connected over the system bus. The address bus on the Intel486 processor and the L2 cache controller are bidirectional to allow cache invalidation on system bus memory writes by other masters. The arbitration logic arbitrates between the processors and the DMA controller. The CPUs and their second-level cache monitor the system bus to identify cache writes. The system must have the mechanisms to support invalidation cycles and to ensure consistency between the contents of the two caches and memory. Coherency is achieved by snooping the address bus. When a write is identified by one processor to a location contained in the other's cache, an inval-

idation cycle must be generated by asserting AHOLD and EADS# to the second processor and its cache. This type of invalidation is true for the write-through cache such as the one shown in Figure 6-10. If the caches are write-back caches the invalidation protocol may be different.

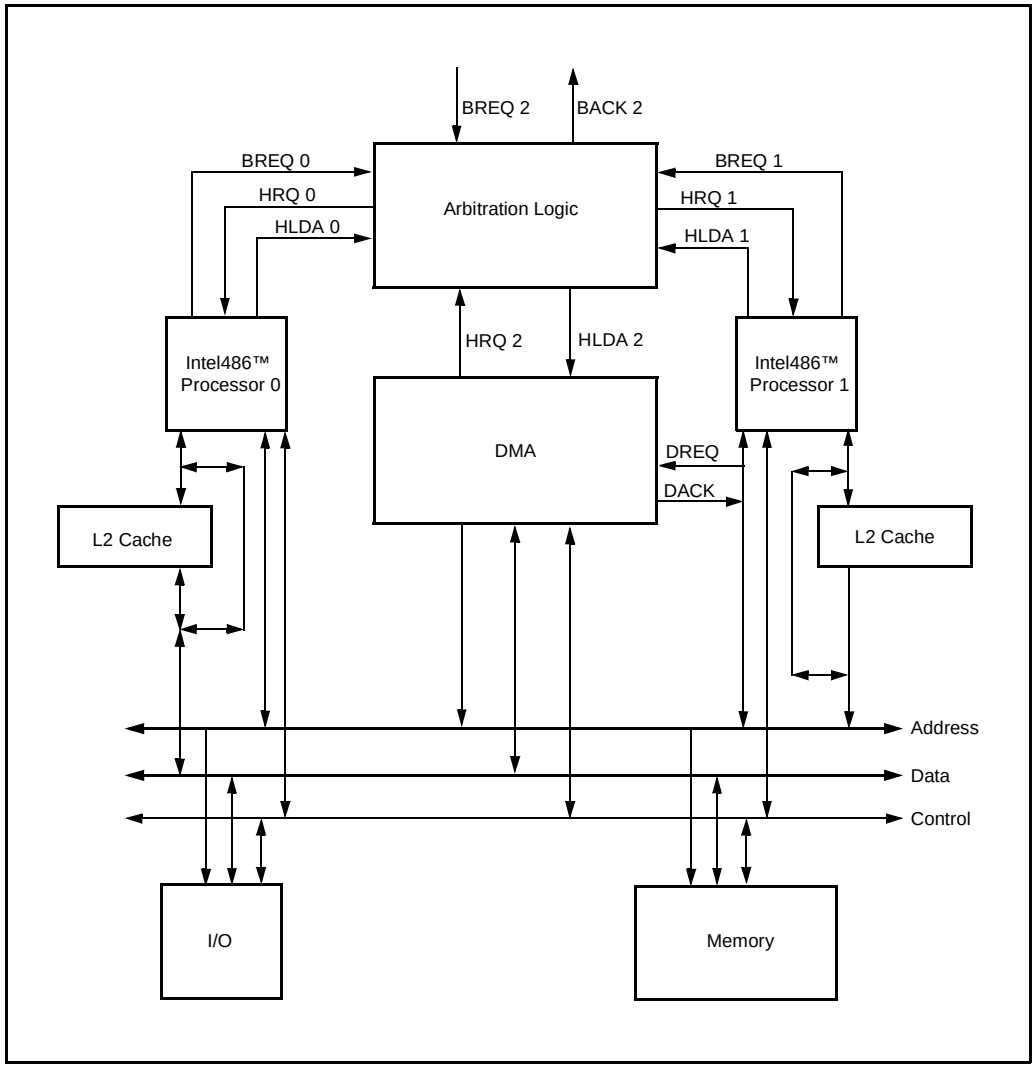


Figure 6-10. Intel486™ Processor System Arbitration

Memory bus utilization in multiple CPU systems may be the most important performance consideration. In this type of system, a cache should have a very high hit rate for both reads and writes. Accesses to main, shared memory must be minimized. Write-back cache is best-suited for

these multiprocessor environments. A write-back cache will, however, be more complex in its architecture and coherency mechanisms.

6.8 AN Intel486™ PROCESSOR SYSTEM EXAMPLE

A typical Intel486 processor system is shown in Figure 6-11. The Intel486 processor has a local bus that consists of address, data and control buses. These buses are either buffered, registered or latched to comprise the system bus.

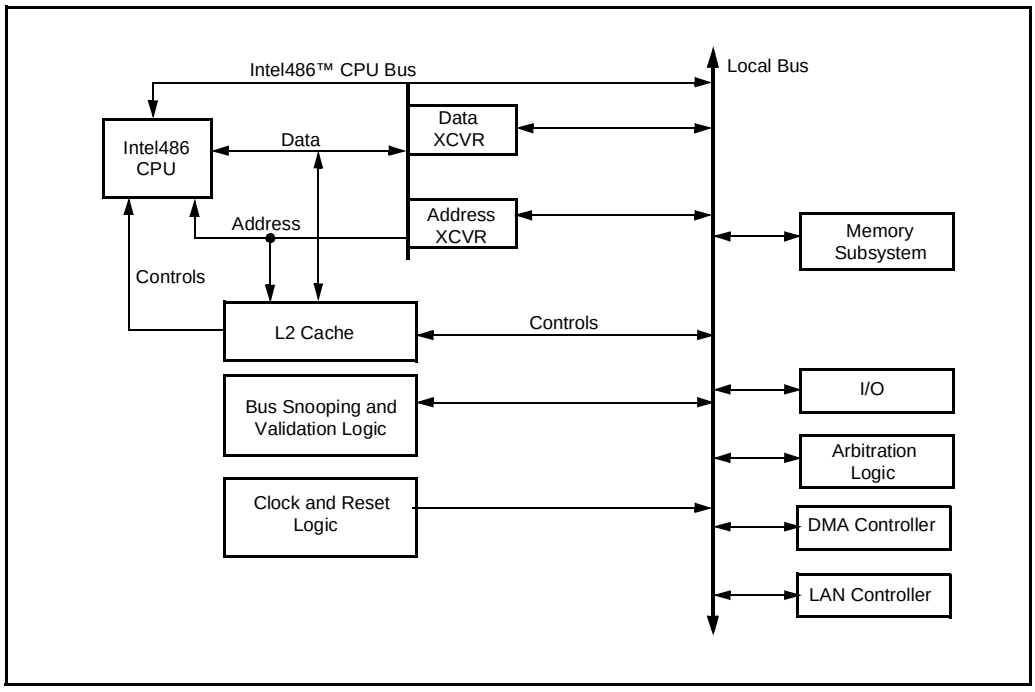


Figure 6-11. A Typical Intel486™ Processor System

The memory subsystem is made up of DRAMs, SRAMs, Flash and EPROMs. Main memory accesses are usually addressed to a DRAM subsystem; however, the I/O subsystem can communicate with the Intel486 processor and with the memory subsystem during DMA operations.

Cache consistency must be maintained whenever main memory accesses occur during DMA operations. Bus snooping and validation logic can monitor the bus to detect memory writes that may be initiated by other bus masters. If such writes are detected, portions of the processor and the L2 cache may have to be invalidated. The Intel486 processor has mechanisms that can invalidate cache entries; the L2 cache device should also have this capability.

The typical L2 cache is closely coupled to the Intel486 processor: the address, data, and control signals are connected to the processor's local bus, and L2 cache control signals interface to the system bus as well. The system bus control signals interface to the processor and the L2 cache in

a similar manner, allowing the L2 cache to be implemented into an Intel486 processor system with ease.

6.8.1 The Memory Hierarchy and Advantages of a Second-level Cache

The Intel486 processor has an on-chip cache and a high-speed register set. These registers are accorded the first level of memory hierarchy. Instructions can be executed in a single clock, and at an average cycles-per-instruction rate of 1.8 (CPI). The next level of hierarchy is accorded to the second-level cache, which can consist of one or more L2 cache devices. These sustain a high level of performance by supporting the fastest possible memory accesses, requiring only two clock cycles for the first read and one clock cycle for each of the subsequent three reads in a burst cycle. System performance degrades if main memory accesses are required. However, with the on-chip L1 cache and the external L2 cache, the number of main memory read accesses is reduced considerably. Figure 6-12 shows the memory hierarchy in a typical Intel486 processor system.

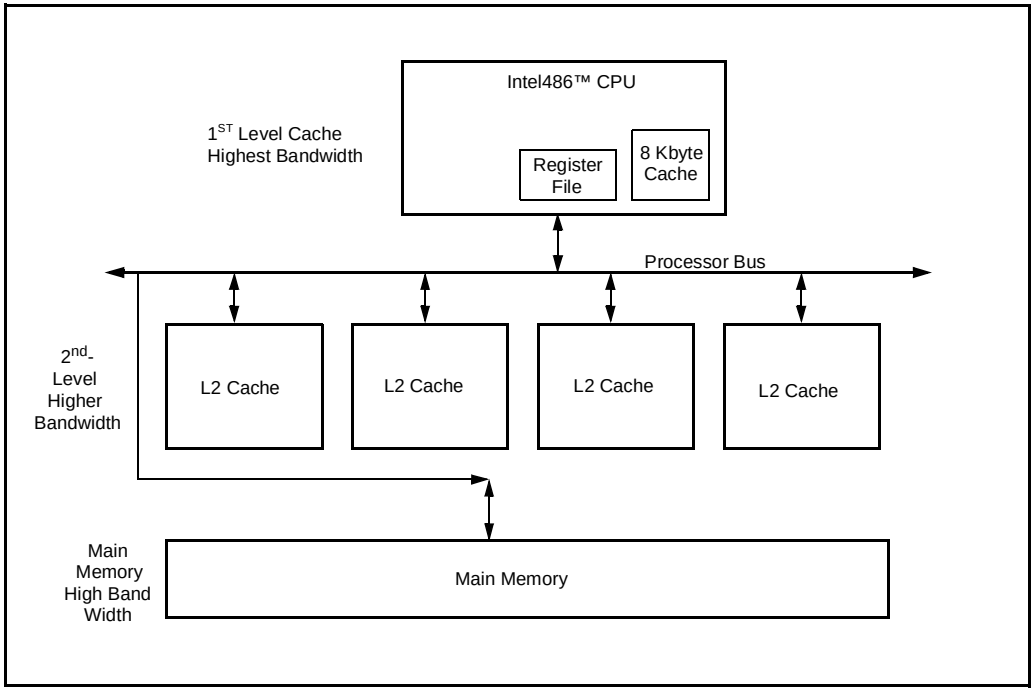


Figure 6-12. Intel486™ Processor System Memory Hierarchy

Because the Intel486 processor internal cache is so efficient, most external CPU bus cycles are DRAM page misses. An L2 cache improves the bus latency problem, as data is available a large percentage of the time from the cache for read operations. A large main memory can have an access time of six to eight cycles on a page miss. On page hits data can be provided in three or four cycles.

7

Peripheral Subsystem

Chapter Contents

CHAPTER 7
PERIPHERAL SUBSYSTEM

The peripheral (I/O) interface is an essential part of any embedded processor system. It supports communications between the microprocessor and the peripherals. Given the variety of existing peripheral devices, a peripheral system must allow a variety of interfaces. An important part of a microprocessor system is the bus that connects all major parts of the system. This chapter describes the connection of peripheral devices to the Intel486™ processor microprocessor bus. This chapter presents design techniques for interfacing different devices with the Intel486 processor, such as LAN controllers and EISA, VESA local bus, and PCI chip sets.

The peripheral subsystem must provide sufficient data bandwidth to support the Intel486 processor. High-speed devices like disks must be able to transfer data to memory with minimal CPU overhead or interaction. The on-chip cache of the Intel486 processor requires further considerations to avoid stale data problems. These subjects are also covered in this chapter.

The Intel486 processor supports 8-bit, 16-bit and 32-bit I/O devices, which can be I/O-mapped, memory-mapped, or both. It has a 106 Mbyte/sec memory bandwidth at 33 MHz. Cache coherency is supported by cache line invalidation and cache flush cycles. I/O devices can be accessed by dedicated I/O instructions for I/O-mapped devices, or by memory operand instructions for memory-mapped devices. In addition, the Intel486 processor always synchronizes I/O instruction execution with external bus activity. All previous instructions are completed before an I/O operation begins. In particular, all writes pending in the write buffers are completed before an I/O read or write is performed. These functions are described in this chapter.

7.1 PERIPHERAL/PROCESSOR BUS INTERFACE

Because the Intel486 processor supports both memory-mapped and I/O-mapped devices, this section discusses the types of mapping, support for dynamic bus sizing, byte swap logic, and critical timings. An example of a basic I/O controller implementation is also included. Some system-oriented interface considerations are discussed because they can have a significant influence on overall system performance.

7.1.1 Mapping Techniques

The system designer should have a thorough understanding of the system application and its use of peripherals in order to design the optional mapping scheme. Two techniques can be used to control the transmission of data between the computer and its peripherals. The most straightforward approach is I/O mapping.

The Intel486 processor can interface with 8-bit, 16-bit or 32-bit I/O devices, which can be I/O-mapped, memory-mapped, or both. All I/O devices can be mapped into physical memory addresses ranging from 00000000H to FFFFFFFFH (four-gigabytes) or I/O addresses ranging from 00000000H to 0000FFFFH (64 Kbytes) for programmed I/O, as shown in Figure 7-1.

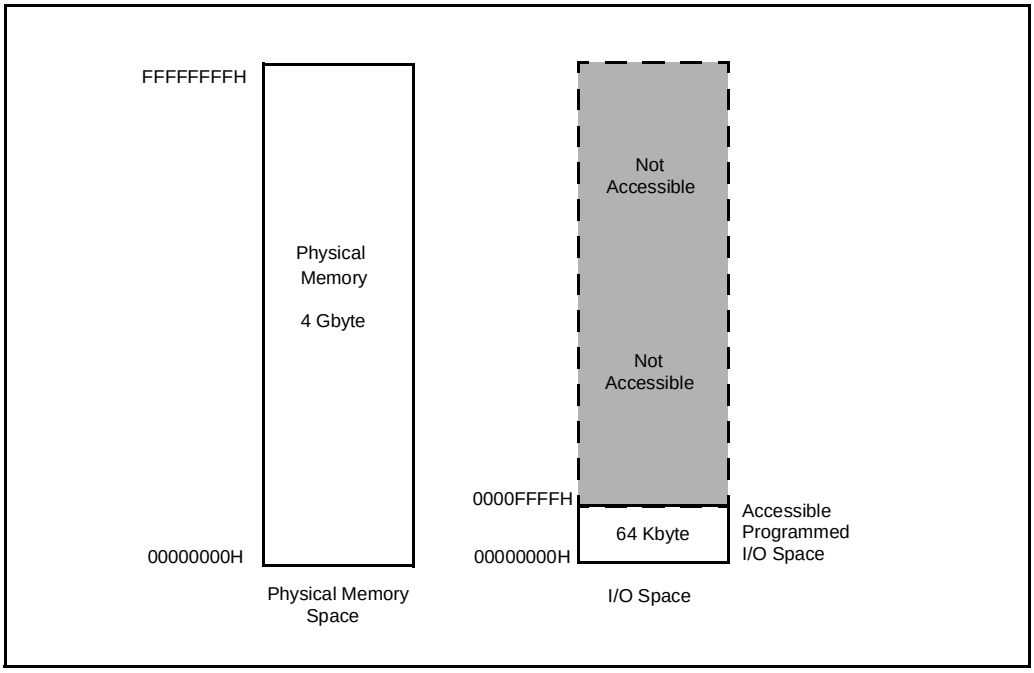


Figure 7-1. Mapping Scheme

I/O mapping and memory-mapping differ in the following respects:

- The address decoding required to generate chip selects for the I/O-mapped devices is much simpler than that required for memory-mapped devices. I/O-mapped devices reside within the I/O space of the Intel486 processor (64 Kbytes); memory-mapped devices reside in a much larger Intel486 processor memory space (4-gigabytes), which requires more address lines to decode.
- The I/O space is 64 Kbytes and can be divided into 64 K of 8-bit ports, 32 K of 16-bit ports, 16 K of 32-bit ports or any combinations of ports which add up to less than 64 Kbytes. The 64 Kbytes of I/O address space refers to physical memory because I/O instructions do not utilize the segmentation or paging hardware and are directly addressable using DX registers.
- Memory-mapped devices can be accessed using the Intel486 processor's instructions, so that I/O to memory, memory-to-I/O, and I/O-to-I/O transfers, as well as compare and test operations, can be coded efficiently.
- The I/O-mapped device can be accessed only with IN, OUT, INS, and OUTS instructions. I/O instruction execution is synchronized with external bus activity. All I/O transfers are performed using the AL (8-bit), AX (16-bit), or EAX (32-bit) registers.

- Memory mapping offers more flexibility in Protected Mode than I/O mapping. Memory-mapped devices are protected by the memory management and protection features. A device can be inaccessible to a task, visible but protected, or fully accessible, depending on where it is mapped. Paging and segmentation provide the same protection levels for 4-Kbyte pages or variable length segments, which can be swapped to the disk or shared between programs. The Intel486 processor supports pages and segments to provide the designer with maximum flexibility.
- The I/O privilege level of the Intel486 processor protects I/O-mapped devices by either preventing a task from accessing any I/O devices or by allowing a task to access all I/O devices. A virtual-8086 mode I/O permission bitmap can be used to select the privilege level for a combination of I/O bytes.

7.1.2 Dynamic Bus Sizing

Dynamic data bus sizing allows a direct processor connection to 32-, 16- or 8-bit buses for memory or I/O devices. The Intel486 processors support dynamic data bus sizing, except for the Ultra-Low Power Intel486 GX processor, which has a 16-bit data bus only. With dynamic bus sizing, the bus width is determined during each bus cycle to accommodate data transfers to or from 32-bit, 16-bit or 8-bit devices. The decoding circuitry can assert BS16# for 16-bit devices, or BS8# for 8-bit devices for each bus cycle. For addressing 32-bit devices, both BS16# and BS8# are deasserted. If both BS16# and BS8# are asserted, an 8-bit bus width is assumed.

Appropriate selection of BS16# and BS8# drives the Intel486 processor to run additional bus cycles to complete requests larger than 16-bits or 8-bits. When BS16# is asserted, a 32-bit transfer is converted into two 16-bit transfers (or three transfers if the data is misaligned). Similarly, asserting BS8# converts 32-bit transfers into four 8-bit transfers. The extra cycles forced by the BS16# or BS8# signals should be viewed as independent cycles. BS16# or BS8# are normally driven active during the independent cycles. The only exception is when the addressed device can vary the number of bytes that it can return between the cycles.

The Intel486 processor drives the appropriate byte enables during the independent cycles initiated by BS8# and BS16#. Addresses A31–A2 do not change if accesses are to a 32-bit aligned area. Table 7-1 shows the set of byte enables that is generated on the next cycle for each of the valid possibilities of the byte enables on the current cycle. BE*x*# must be ignored for 16-byte cycles to memory-mapped devices.

Table 7-1. Next Byte-Enable Values for the BS*x*# Cycles

Current				Next with BS8#				Next with BS16#			
BE3#	BE2#	BE1#	BE0#	BE3#	BE2#	BE1#	BE0#	BE3#	BE2#	BE1#	BE0#
1	1	1	0	N	N	N	N	N	N	N	N
1	1	0	0	1	1	0	1	N	N	N	N
1	0	0	0	1	0	0	1	1	0	1	1
0	0	0	0	0	0	0	1	0	0	1	1
1	1	0	1	N	N	N	N	N	N	N	N
1	0	0	1	1	0	1	1	1	0	1	1
0	0	0	1	0	0	1	1	0	0	1	1
1	0	1	1	N	N	N	N	N	N	N	N
0	0	1	1	0	1	1	1	N	N	N	N
0	1	1	1	N	N	N	N	N	N	N	N

NOTE: "N" means that another bus cycle is not required to satisfy the request.

The dynamic bus sizing feature of Intel486 processor is significantly different than that of the Intel386™ DX processor. The Intel486 processor requires that the data bytes be driven on the addressed lines only, unlike the Intel386 DX processor, which expects both high and low order bytes on D15–D0. The simplest example of this function is a 32-bit aligned BS16# read. When the Intel486 processor reads the two higher order bytes, they must be driven on D31–D16 data bus, and it expects the two low order bytes on D15–D0. The Intel386 DX processor always reads or writes data on the lower 16-bits of the data bus when BS16# is asserted.

The external system design must provide buffers to allow the Intel486 processor to read or write data on the appropriate data bus pins. Table 7-2 shows the data bus lines where the Intel486 processor expects valid data to be returned for each valid combination of byte enables and bus sizing options. Valid data is driven only on data bus pins which correspond to byte enable signals that are active during write cycles. Other data pins are also driven, but they do not contain valid data. Unlike the Intel386 DX processor, the Intel486 processor does not duplicate write data on the data bus when corresponding byte enables are deasserted.

Table 7-2. Valid Data Lines for Valid Byte Enable Combinations

BE3#	BE23	BE1#	BE0#	w/o BS8#/BS16#	w BS8#	w BS16#
1	1	1	0	D7–D0	D7–D0	D7–D0
1	1	0	0	D15–D0	D7–D0	D15–D0
1	0	0	0	D23–D0	D7–D0	D15–D0
0	0	0	0	D31–D0	D7–D0	D15–D0
1	1	0	1	D15–D8	D15–D8	D15–D8
1	0	0	1	D23–D8	D15–D8	D15–D8
0	0	0	1	D31–D8	D15–D8	D15–D8
1	0	1	1	D23–D16	D23–D16	D23–D16
0	0	1	1	D31–D16	D23–D16	D31–D16
0	1	1	1	D31–D24	D31–D24	D31–D24

The BS16# and BS8# inputs allow external 16- and 8-bit buses to be supported using fewer external components. The Intel486 processor samples these pins every clock cycle. This value is sampled on the clock before RDY# to determine the bus size. When BS8# or BS16# is asserted, only 16-bits or 8-bits of data are transferred in a clock cycle. When both BS8# and BS16# are asserted, an 8-bit bus width is used.

Dynamic bus sizing allows the power-up or boot-up programs to be stored in 8-bit non-volatile memory devices (e.g., PROM, EPROM, E2PROM, Flash, and ROM) while program execution uses 32-bit DRAM or variants.

7.1.3 Address Decoding for I/O Devices

Address decoding for I/O devices resembles address decoding for memories. The primary difference is that the block size (range of addresses) for each address signal is much smaller. The minimum block size depends on the number of addresses used by the I/O device. In most processors, where I/O instructions are separate, I/O addresses are shorter than memory addresses. Typically, processors with a 16-bit address bus use an 8-bit address for I/O.

One technique for decoding memory-mapped I/O addressed is to map the entire I/O space of the Intel486 processor into a 64-Kbyte region of the memory space. The address decoding logic can be reconfigured so that each I/O device responds to a memory address and an I/O address. This configuration is compatible with software that uses either I/O instructions or memory-mapped techniques.

Addresses can be assigned arbitrarily within the I/O or memory space. Addresses for either I/O-mapped or memory-mapped devices should be selected so as to minimize the number of address lines needed.

7.1.3.1 Address Bus Interface

Figure 7-2 shows the Intel486 processor address interface to 32-, 16- and 8-bit devices. To address 16-bit devices, the byte enables must be decoded to produce A1, BHE# and BLE# (A0) signals.

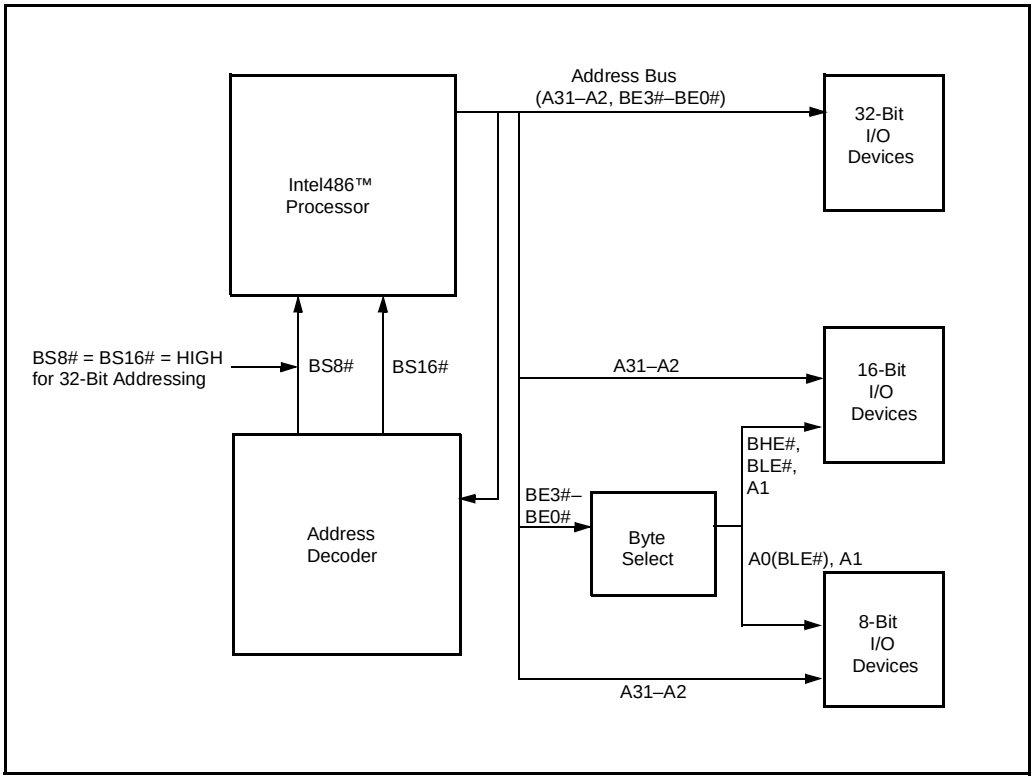


Figure 7-2. Intel486™ Processor Interface to I/O Devices

To access to 8-bit devices, the byte enable signals must be decoded to generate A0 and A1. Because A0 and BLE# are the same, the same generation logic can be used. For 32-bit memory/mapped devices A31–A2 can be used in conjunction with BE3#–BE0#. This logic is shown in Figure 7-3.

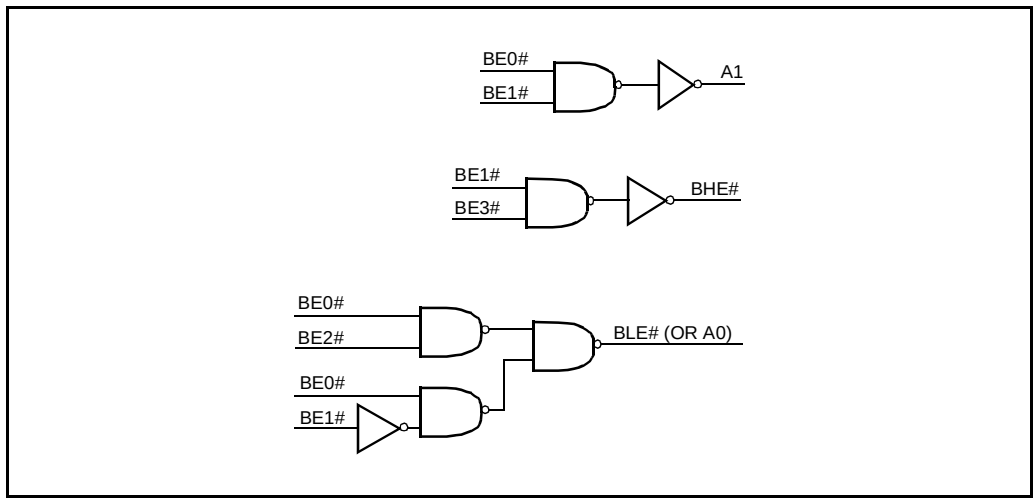


Figure 7-3. Logic to Generate A1, BHE# and BLE# for 16-Bit Buses

7.1.3.2 8-Bit I/O Interface

Due to the presence of dynamic data bus sizing and the variety of byte-enable pin combinations (Table 7-2), byte swapping logic for 32-to-8-bit conversions can be implemented in various ways.

This section discusses an example in which BE3#–BE0# are low and D7–D0 are used when BS8# is enabled.

Figure 7-4 shows the interfacing of an Intel486 processor to an 8-bit device. This implementation requires seven 8-bit bidirectional data buffers.

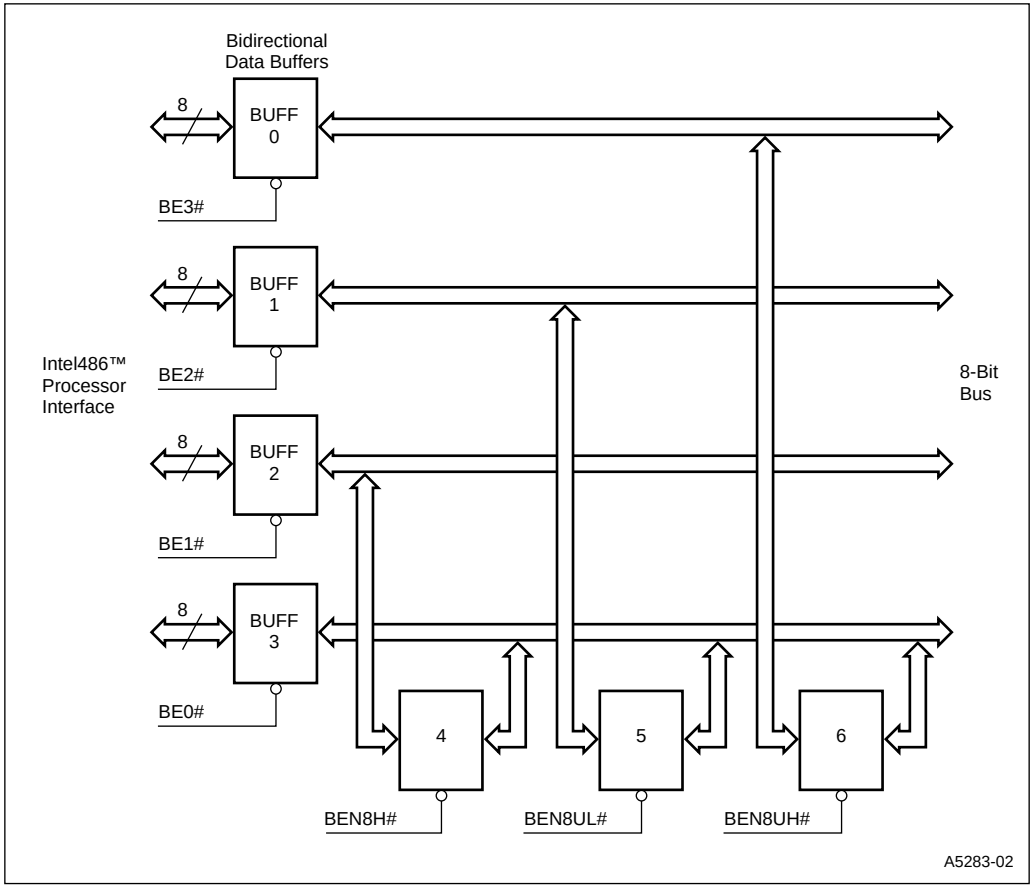


Figure 7-4. Intel486™ Processor Interface to 8-Bit Device

In this example of a 32-bit write, the BE3#–BE0# are enabled; hence 32 bits of data reside on the data buffer outputs. This data is then swapped based on the control signals. Buffers are enabled in the following manner:

For Byte # 0 Buffer 3 is enabled (BE0# is true)
For Byte # 1 Buffer 2 and 4 are enabled (BE1# and BEN8H#)
For Byte # 2 Buffer 1 and 5 are enabled (BE2# and BEN8UL#)
For Byte # 3 Buffer 0 and 6 are enabled (BE3# and BEN8UH#)

Table 7-5 shows the truth table for 8-bit I/O interface to the Intel486 processor. The table also contains the values of the control signals used to enable the second set of buffers. The PLD equations used to implement these signals are shown in Tables 7-3 and 7-4.

Table 7-3. PLD Input Signals

BS8#	The signal is from an 8-bit device or from the system logic that interfaces to an 8-bit device.
BE3#–BE0#	When processor drives all of these signals Low, external logic should look only for BE0# while in 8-bit mode.
ADS#	An address strobe from the Intel486™ processor indicates a valid processor cycle.
OUTPUTS BEN8H#, BEN8UH#, BEN8UL#	Byte enables for 8-bit interface.

Table 7-4. Equations

BEN8H = ADS * BE1 * /BE0 * BS8 + /ADS * BEN8H	;Swapping second byte for 8-bit interface
BEN8UL = ADS * BE2 * /BE1 * /BE0 * BS8 + /ADS * BEN8UL	;Swapping third byte for 8-bit interface
BEN8UH = ADS * BE3 * /BE2 * /BE1 * /BE0 * BS8 + /ADS * BEN8UH	;Swapping fourth byte for 8-bit interface

Table 7-5. 32-Bit to 8-Bit Steering (Sheet 1 of 2)

Intel486™ Processor[3]					8-Bit Interface[1]					
BE3#	BE2#	BE1#	BE0#	BEN16#	BEN8UH#	BEN8UL#	BEN8H#	BHE#[2]	A1	A0
0	0	0	0	1	1	1	1	X	0	0
1	0	0	0	1	1	1	1	X	0	0
0	1	0	0†	1	1	1	1	X	X	X
1	1	0	0	1	1	1	1	X	0	0
0	0	1	0†	1	1	1	1	X	X	X
1	0	1	0†	1	1	1	1	X	X	X
Inputs					Outputs					

NOTES:
1. X implies “do not care” (either 0 or 1).
2. BHE# (byte high enable) is not needed in 8-bit interface.
3. † indicates a non-occurring pattern of byte enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes.

Table 7-5. 32-Bit to 8-Bit Steering (Sheet 2 of 2)

Intel486™ Processor[3]					8-Bit Interface[1]					
BE3#	BE2#	BE1#	BE0#	BEN16#	BEN8UH#	BEN8UL#	BEN8H#	BHE#[2]	A1	A0
0	1	1	0†	1	1	1	1	X	X	X
1	1	1	0	1	1	1	1	X	0	0
0	0	0	1	1	1	1	1	X	0	1
1	0	0	1	1	1	1	0	X	0	1
0	1	0	1†	1	1	1	0	X	X	X
1	1	0	1	1	1	1	0	X	0	1
0	0	1	1	1	1	0	0	X	1	0
1	0	1	1	1	1	0	1	X	1	0
0	1	1	1	1	0	1	1	X	1	1
1	1	1	1	1	1	1	1	X	X	X
Inputs					Outputs					

NOTES:
1. X implies “do not care” (either 0 or 1).
2. BHE# (byte high enable) is not needed in 8-bit interface.
3. † indicates a non-occurring pattern of byte enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes.

7.1.3.3 16-Bit I/O Interface

16-bit I/O interface byte swap logic requires six 8-bit bidirectional I/O data buffers as shown in Figure 7-5. Buffers 3 through 0 are controlled by BE3#–BE0# respectively. Buffers 4 and 5 are monitored by BEN16#.

To transfer data on the lower 16-bits, buffers 2 and 3 are enabled. While the higher 16-bits are transferred through Buffer 0, 1, 4, and 5.

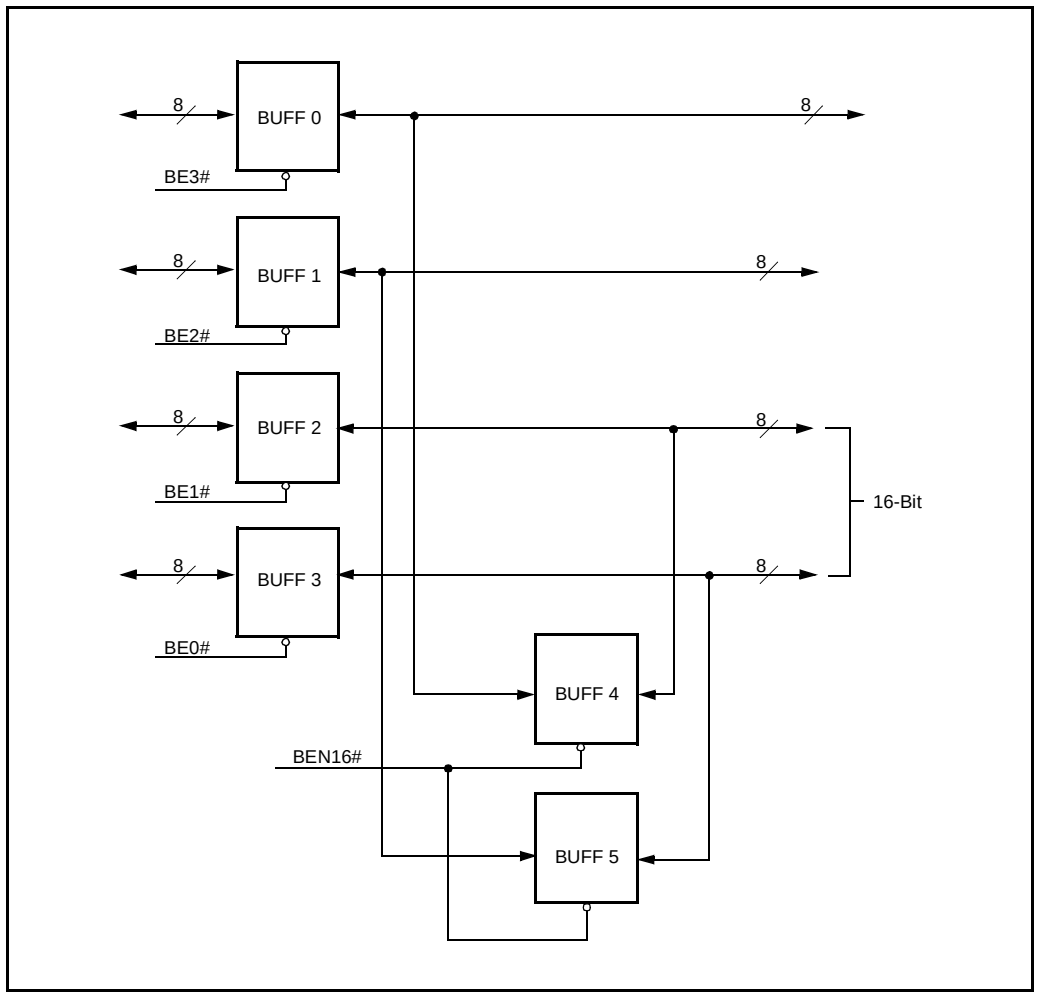


Figure 7-5. Bus Swapping 16-Bit Interface

Table 7-9 shows the truth table for 32-to-16-bit bus swapping logic and A0, A1 and BHE# generation.

The PLD equation used to implement 32-bit-to-16-bit byte swap logic is shown in Tables 7-6 and 7-7.

Table 7-6. PLD Input Signals

BS16#	Either from a 16-bit device or from system logic which indicates a 16-bit transfer.
BE3#–BE0#	Byte enable inputs from Intel486™ processor. In 16-bit mode, the external logic should look at BE0# and BE1# only.
ADS#	Address strobe from an Intel486 processor indicating a valid CPU cycle.

Table 7-7. PLD Output Signals

BS16#	Word enable for 16-bit interface.

Table 7-8. Equation

```
BEN16 = ADS * BE2 * /BE1 * /BE0 * BS16 * /BS8                ;swapping upper 16-bits
+ ADS * BE3 * /BE1 * /BE0 * BS16 * /BS8
+ /ADS * BEN16
```

Table 7-9. 32-Bit to 16-Bit Bus Swapping Logic Truth Table (Sheet 1 of 2)

Intel486™ Processor[3]					8-Bit Interface[1]					
BE3#	BE2#	BE1#	BE0#	BEN16#	BEN8UH#	BEN8UL#	BEN8H#	BHE#[2]	A1	A0
0	0	0	0	1	1	1	1	1	0	1
1	0	0	0	1	1	1	1	1	0	1
0	1	0	0†	1	1	1	1	X	X	X
1	1	0	0	1	1	1	1	1	0	1
0	0	1	0†	0	1	1	1	0	X	0
1	0	1	0†	0	1	1	1	X	X	0
Inputs					Outputs					

NOTES:
1. X implies “do not care” (either 0 or 1).
2. BHE# (byte high enable) is not needed in 8-bit interface.
3. † indicates a non-occurring pattern of byte enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes.

Table 7-9. 32-Bit to 16-Bit Bus Swapping Logic Truth Table (Sheet 2 of 2)

Intel486™ Processor[3]					8-Bit Interface[1]					
BE3#	BE2#	BE1#	BE0#	BEN16#	BEN8UH#	BEN8UL#	BEN8H#	BHE#[2]	A1	A0
0	1	1	0†	0	1	1	1	X	X	X
1	1	1	0	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	0	1
0	1	0	1†	1	1	1	1	X	X	X
1	1	0	1	1	1	1	1	1	0	1
0	0	1	1	0	1	1	1	0	1	0
1	0	1	1	0	1	1	1	1	1	0
0	1	1	1	0	1	1	1	1	1	1
1	1	1	1†	1	1	1	1	X	X	X
Inputs					Outputs					

NOTES:
1. X implies “do not care” (either 0 or 1).
2. BHE# (byte high enable) is not needed in 8-bit interface.
3. † indicates a non-occurring pattern of byte enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes.

The logic needed to generate the byte-swapping control signals for 32-bit-to-8-bit and 32-bit-to-16-bit data transfer can be implemented in PLDs. Propagation delay of the PLD and the bidirectional buffer propagation delay of 9 ns maximum must be taken into consideration. This delay adds into data set-up time for CPU read cycles and data valid delay for the CPU write cycle. The byte-swapping and address bit generation logic is shown in Figure 7-6.

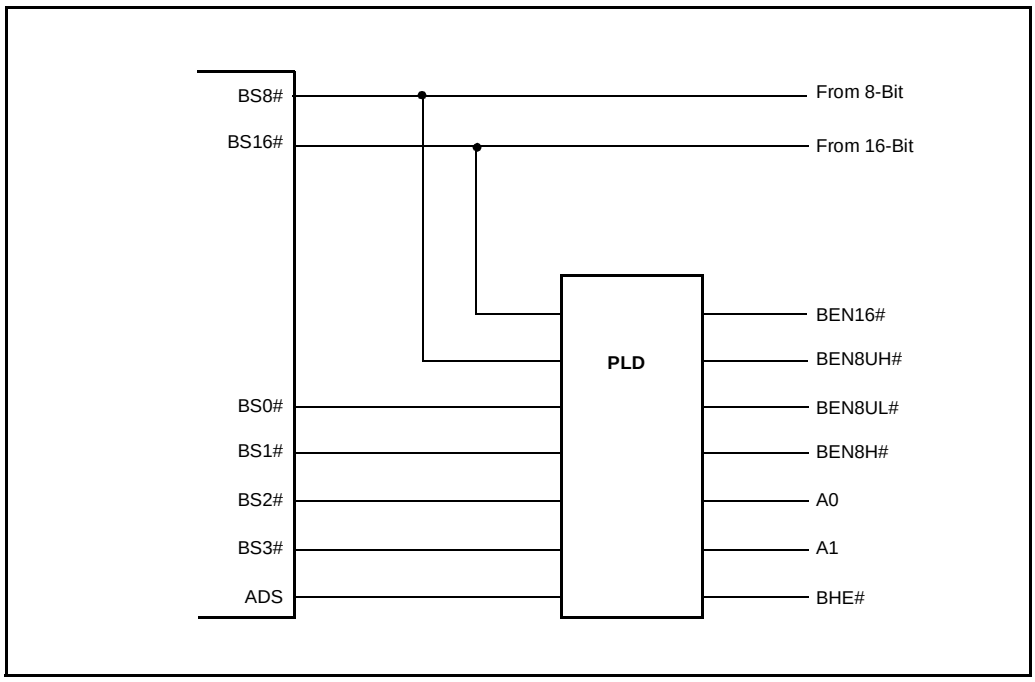


Figure 7-6. Bus Swapping and Low Address Bit Generating Control Logic

7.1.3.4 32-Bit I/O Interface

A simple 32-bit I/O interface is shown in Figure 7-7. The example uses only four 8-bit wide bi-directional buffers which are enabled by BE3#–BE0#. Table 7-2 provides different combinations of BE3#–BE0#. To provide greater flexibility in I/O interface implementation, the design should include interfaces for 32-, 16- and 8-bit devices. The truth table for a 32-to-32-bit interface is shown in Table 7-10.

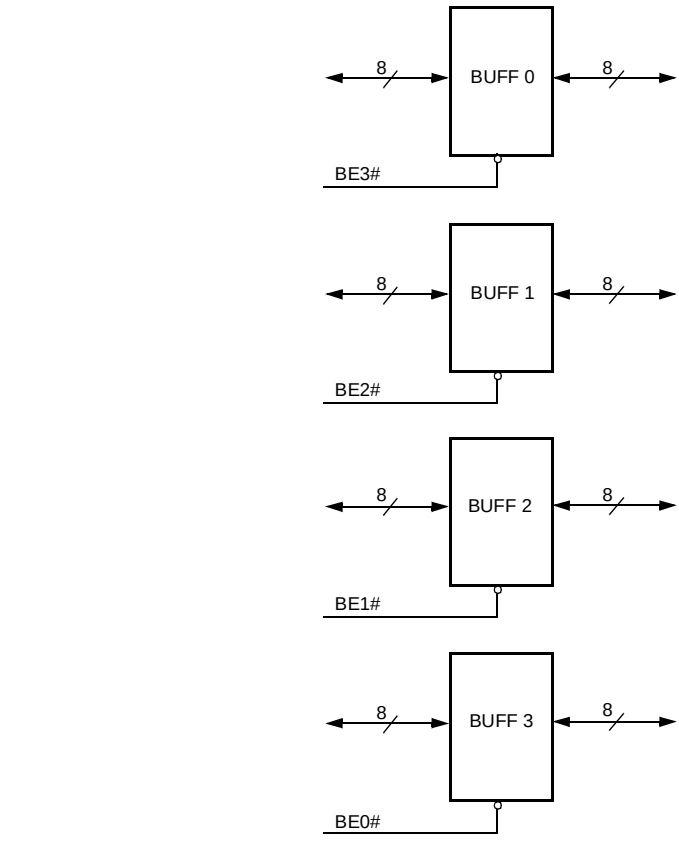


Figure 7-7. 32-Bit I/O Interface

Table 7-10. 32-Bit to 32-Bit Bus Swapping Logic Truth Table

Intel486™ Processor[3]					8-Bit Interface[1]					
BE3#	BE2#	BE1#	BE0#	BEN16#	BEN8UH#	BEN8UL#	BEN8H#	BHE#[2]	A1	A0
0	0	0	0	1	1	1	1	X	X	X
1	0	0	0	1	1	1	1	X	X	X
0	1	0	0†	1	1	1	1	X	X	X
1	1	0	0	1	1	1	1	X	X	X
0	0	1	0†	1	1	1	1	X	X	X
1	0	1	0†	1	1	1	1	X	X	X
0	1	1	0†	1	1	1	1	X	X	X
1	1	1	0	1	1	1	1	X	X	X
0	0	0	1	1	1	1	1	X	X	X
1	0	0	1	1	1	1	1	X	X	X
0	1	0	1†	1	1	1	1	X	X	X
1	1	0	1	1	1	1	1	X	X	X
0	0	1	1	1	1	1	1	X	X	X
1	0	1	1	1	1	1	1	X	X	X
0	1	1	1	1	1	1	1	X	X	X
1	1	1	1†	1	1	1	1	X	X	X
Inputs					Outputs					

NOTES:
1. X implies “do not care” (either 0 or 1).
2. BHE# (byte high enable) is not needed in 8-bit interface.
3. † indicates a non-occurring pattern of byte enables; either none are asserted or the pattern has byte enables asserted for non-contiguous bytes.

7.2 BASIC PERIPHERAL SUBSYSTEM

All microprocessor systems include a CPU, memory and I/O devices which are linked by the address, data and control buses. Figure 7-8 illustrates the system block diagram of a typical Intel486 processor-based system.

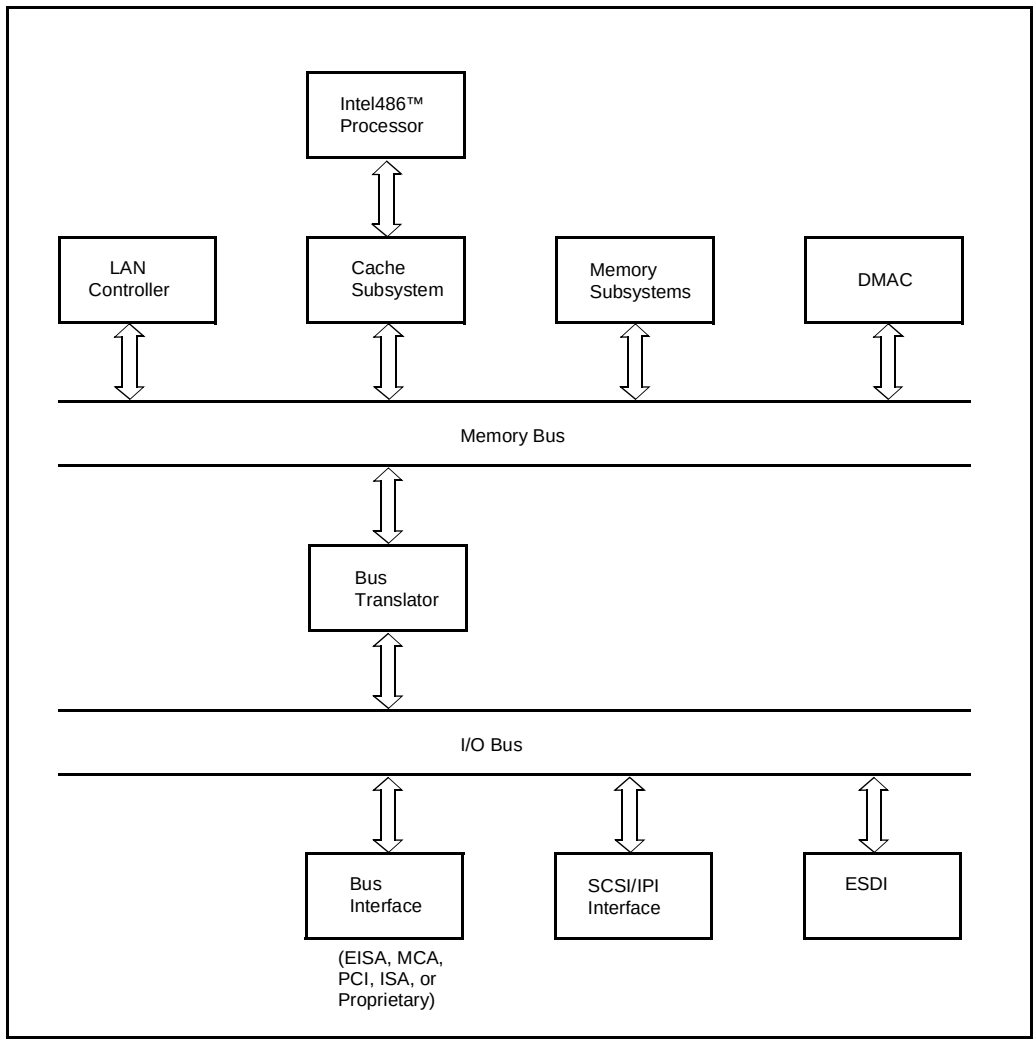


Figure 7-8. System Block Diagram

An embedded Intel486 processor system may consist of several subsystems. The heart of the system is the processor. The memory subsystem is also important and must be efficient and optimized to provide peak system level performance. As described in Chapter 5, "Memory

Subsystem Design," it is necessary to utilize the burst-bus feature of the Intel486 processor for the DRAM control implementation. The cache subsystem, as described in Chapter 6, "Cache Subsystem," also plays an important role in overall system performance. For many systems however, the on-chip cache provides sufficient performance.

A high-performance Intel486 processor-based system, requires an efficient peripheral subsystem. This section describes the elements of this system, including the I/O devices on the expansion bus (the memory bus) and the local I/O bus. In a typical system, a number of slave I/O devices can be controlled through the same local bus interface. Complex peripheral devices which can act as bus masters may require a more complex interface.

The bus interface control logic is shown in Figure 7-9 and consists of three main blocks: the bus control and ready logic, the data transceiver and byte swap logic, and the address decoder.

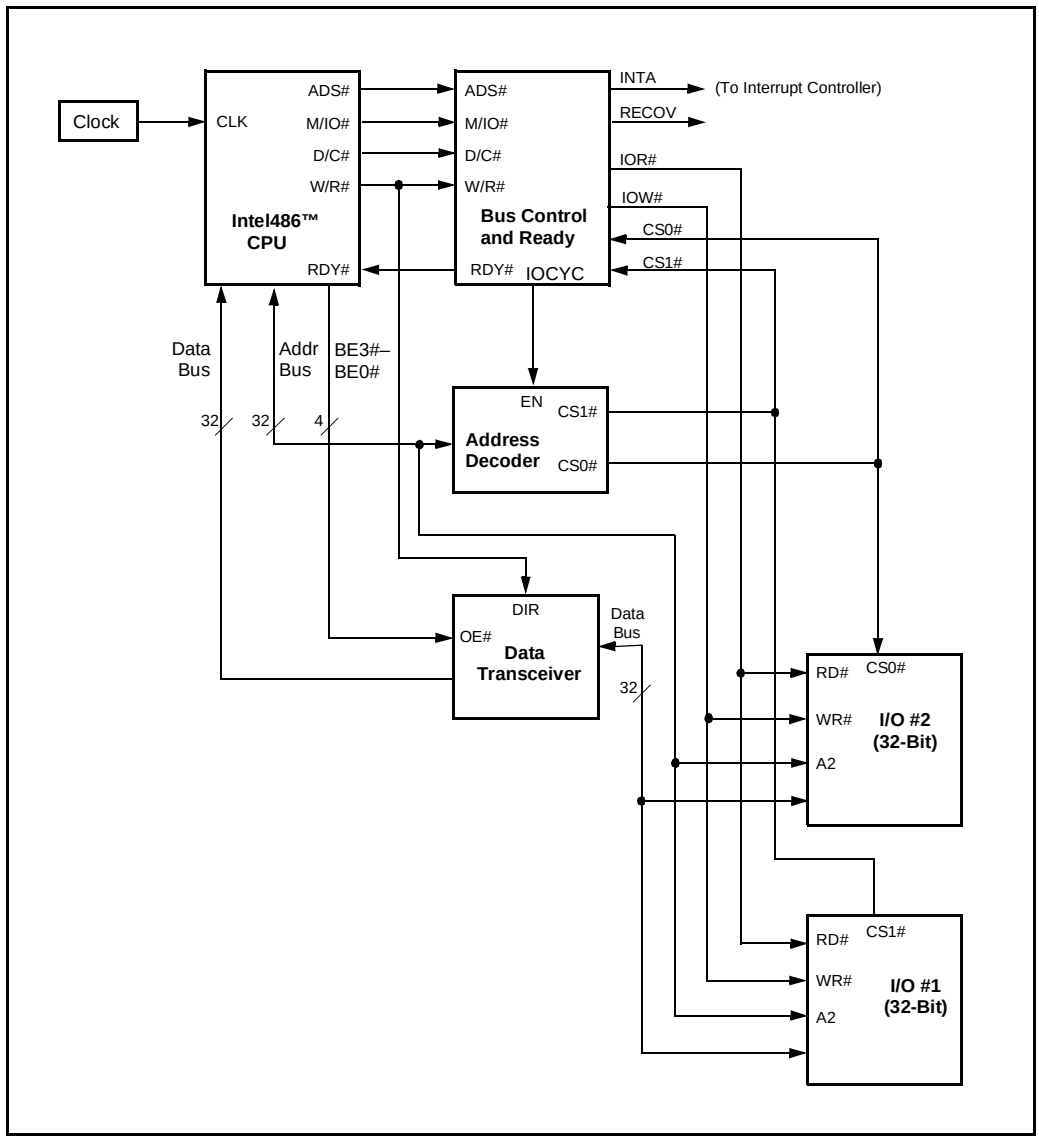


Figure 7-9. Basic I/O Interface Block Diagram

7.2.1 Bus Control and Ready Logic

A typical peripheral device has address inputs which the processor uses to select the device's internal registers. It also has a chip select (CS#) signal which enables it to read data from and write data to the data bus, as controlled by the READ (RD#) and WRITE (WR#) control signals. For a processor that has separate memory and I/O addressing, either memory or I/O read and write signals can be used. As discussed in Section 7.1.1, "Mapping Techniques," when memory read and write signals are used to access the peripheral device, the device is called a memory-mapped I/O device.

Many peripheral devices also generate an interrupt output which is asserted when a response is required from the processor. Here, the processor must generate an interrupt acknowledge (INTA#) signal.

The bus controller decodes the Intel486 processor's status outputs (W/R#, M/IO# and D/C#) and activates command signals according to the type of bus cycle requested.

The bus controller can be used to do the following:

1. Generate an EPROM data read when the control logic generates a signal such as a memory read command (EPRD#). The command forces the selected memory device to output data. Chapter 8, "System Bus Design," provides further explanation.
2. Generate the IOCYC# signal which indicates to the address decoder that a valid I/O cycle is taking place. As a result, the relevant chip select (CS#) signal should be enabled for the I/O device. Once IOCYC is generated, the IOR# and IOW# signals are asserted according to the decoded Intel486 processor status signals (explained later).
3. Initiate I/O read cycles when W/R# is low and M/IO# is low. The I/O read command (IOR#) is generated. IOR# selects the I/O device which is to output the data.
4. Initiate an I/O write cycle when W/R# is high and M/IO# is low. The I/O write command signal (IOW#) is generated. This signal instructs a selected I/O device to receive data from the Intel486 processor.
5. Generate a RECOV signal which is used for recovery and to avoid data contention. This signal is detailed in Section 7.2.6, "Recovery and Bus Contention."
6. Generates the interrupt acknowledge signal (INTA#). This signal is sent to the 82C59A programmable interrupt controller to enable 82C59A interrupt vector data onto the Intel486 processor data bus using a sequence of interrupt acknowledge pulses that are issued by the control logic. This signal is detailed in Section 7.5, "Interfacing to x86 Peripherals."

7.2.2 Bus Control Signal Description

The following list describes the input/output signals for the bus control logic.

7.2.2.1 Processor Interface

ADS#—Address Status. This input signal to the bus controller is connected directly to the processor's ADS# output. It indicates that a valid bus cycle definition and address are available on the cycle definition lines and address bus. ADS# is driven active at the same time when addresses are driven.

M/IO#—Memory/Input-Output Signal
D/C#—Data/Control
W/R#—Write/Read (Input signals to bus controller)

These signals are connected directly to the Intel486 processor's bus cycle status outputs. For the Intel486 processor, they are valid when ADS# is asserted. Table 7-11 describes the bus cycles of various combinations of M/IO#, D/C# and W/R# signals.

Table 7-11. Bus Cycle Definitions

M/IO#	D/C#	W/R#	ADS#	Bus Cycle Initiated
0	0	0	0	Interrupt acknowledge
0	0	1	0	Halt/special cycle
0	1	0	0	I/O read
0	1	1	0	I/O write
1	0	0	0	Code read
1	0	1	0	Reserved†
1	1	0	0	Memory read
1	1	1	0	Memory write

NOTE: † Intel reserved. Do not use.

RDY#—Ready Output Signal. This signal is connected directly to the Intel486 processor's RDY# input and indicates that the current bus cycle is complete. It also indicates that the I/O device has returned valid data to the Intel486 processor's data pins following an I/O write cycle. For the Intel486 processor, RDY# is ignored when the bus is idle and at the end of the first clock of the bus cycle. The signal is utilized in wait state generation which is covered in the next section.

CLK#—Clock Input Signal. This signal provides the fundamental timings for the bus control logic and is synchronous with the processor's clock. All of the external timings are specified with respect to the rising edge of the clock.

IOCYC—I/O Interface Signals. The IO cycle output signal is generated at the rising clock edge following ADS#, M/IO#, D/C and W/R# being active. The signal indicates that an I/O cycle is taking place and is used to enable the address decoder.

IOR#—The I/O Read Signal. This signal is active low and is generated when the Intel486 processor's W/R# output signal is low, indicating a read cycle. When IOR# is low, data can be read from the peripheral device. The signal is deasserted with the rising edge of the RDY# signal.

IOW#—Interrupt Write Signal. This signal is generated by the controller logic and is active low when W/R# status signal from the Intel486 processor is high, indicating that the processor will write to the I/O device which has its present address on the address bus. When IOW# is low, data from the Intel486 processor can be written to the peripheral device. The signal is valid until the rising edge of the RDY# signal.

INTA—Interrupt Acknowledge Signal. This signal is active high and is generated to acknowledge an interrupt from peripheral devices such as 82C59A, etc. The signal function is discussed in Section 7.5, "Interfacing to x86 Peripherals."

7.2.2.2 Wait State Generation Signals

SEL0, SEL1, SEL2. These programmable wait state select inputs can be controlled by DIP switches or can be programmed by the processor. In the control logic example, a seven-state wait state generator is implemented. The purpose and functionality of a wait state generator is described in the next section.

C0, C1, C2—Counter Outputs 0, 1, and 2. These outputs are internally decoded to generate a RDY# signal and they represent the number of wait states implemented by the bus control logic. The wait state generation logic is used to patch timing differences between the peripheral device and the Intel486 processor. The next section discusses this issue in detail.

7.2.3 Wait State Generator Logic

When the memory subsystem or the I/O device cannot respond to the processor in time, wait states are added to the bus cycles. During wait states the processor freezes the state of the bus. On the Intel486 processor, wait states are activated by the RDY# signal (when asserted). Additional wait states are generated as long as RDY# stays deasserted, and the processor resumes its operations once RDY# is asserted.

Timing differences between microprocessors and peripheral devices are common, but can be compensated for by using wait states or some other delay techniques. The following major timing parameters must be accounted for:

1. Minimum pulse width for read and write timings
2. Chip select access time
3. Address access time
4. Access time from read strobe

It is possible to adjust the minimum pulse width and chip select access time by adding wait states. Refer to Section CHAPTER 4, "Bus Operation" for more detailed information on adding wait states to basic bus cycles.

Figure 7-10 shows PLD equations for basic I/O control logic. A wait state generator should be implemented to optimize wait state generation.

```
Inputs    ADS#, M/IO#, D/C#, W/R#, SEL0, SEL1, SEL2
Outputs   IOCYC, 0 C1, C2, IOR#, IOW#, RDY#
          IOCYC = IOCYCLE VALID
          C0, C1, C2 = Outputs of a 3-bit counter
          Sel 0, 1, 2 = Programmable wait state select input


PLD Equation:
          IO VALID CYCLE;                          ; start I/O cycle
IOCYC : =ADS * M/IO# *D/C                          ;END when ready
Wait State Counter;
          C0 : = IOCYC * C0#                       ;Counter bit 0
          C1 : = IOCYC * C0 * C1#                  ;Counter bit 1
                 + IOCYC * C0# * C1
          C2 : = IOCYC * C0 * C1 * C2#             ;Counter bit 2
                 + IOCYC * C0# * C2
                 + IOCYC * C0# * C1 * C2


I/O Read; I/O Write
IOR : = ADS * M/IO# * D/C * W/R#
        + IOR * RDY
IOW : = ADS * M/IO * D/C * W/R
         + IOW * RDY#
READY (3 Wait States)
          RDY = C0 * C1 * C2#
```

Figure 7-10. PLD Equations for Basic I/O Control Logic

The equation in Figure 7-10 shows an implementation of a seven-state wait state controller. The wait state logic inserts the needed wait states according to the number required by the device being accessed. In a simple design, I/O accesses can be designated as being equal to the number of wait states required by the slowest device.

7.2.4 Address Decoder

The function of the address decoder is to decode the most significant address bits and generate address select signals for each system device. The address space is divided into blocks, and the address select signals indicate whether the address on the address bus is within the predetermined range. The block size usually represents the amount of address space that can be accessed within a particular device and the address select signal is asserted for any address within that range.

Address select signals are asserted within the range of addresses which is determined by the decoded address lines. The relationship between memory and I/O mapping and address decoding is given by the following equation:

Given thatn = bits to decoder
m = bits to I/O or memory

then # of chip selects = 2^n
address range = 2^m = # of least significant address lines

For example, when the address decoder decodes A13 through the most significant address bits, the least significant 13 address bits A2 to A12 are ignored. Hence the address select can be asserted for a 2-Kbyte address range.

For I/O-mapped devices, the maximum I/O space is 64 Kbytes. When using I/O instructions the block size (range of addresses) for each address select signal is much smaller than the address space of the memory-mapped devices. The minimum block size is determined according to the number of addresses being used by the peripheral device.

A typical address decoding circuit for a basic I/O interface implementation is shown in Figure 7-11. It uses 74S138. Only one output is asserted at a time. The signal corresponding to the binary number present at the A, B and C inputs and value of the gate enable signals.

Figure 7-12 shows the internal logic and truth table of the 74S138. It has three enable inputs; two are active low, and one is active high. All three inputs must be asserted; otherwise the outputs are deasserted. Since all of the outputs are active low, the selected output is low and the others are high.

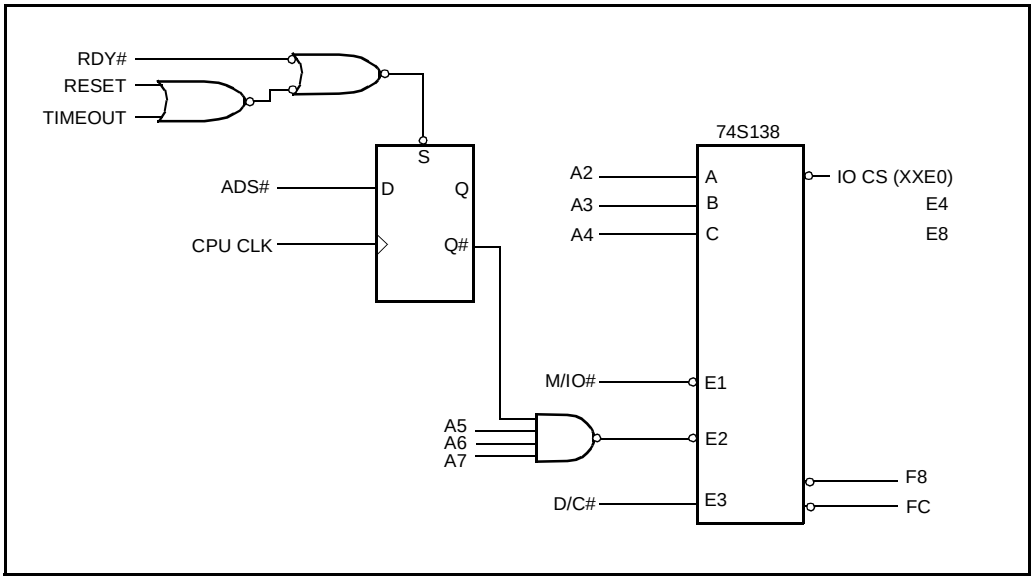
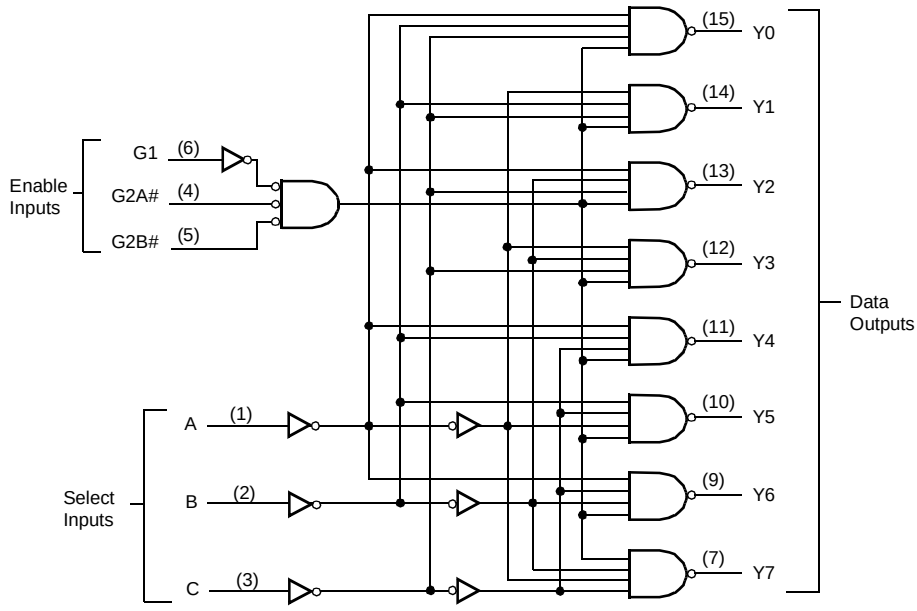


Figure 7-11. I/O Address Example

In Figure 7-11, address lines A15–A8 are ignored to maintain simplicity. Lines A7–A2 are decoded to generate addresses XXE0–XXFC. When a valid cycle begins, ADS# is latched in the flip-flop.



Function Table

Inputs					Outputs							
Enable		Select										
G1	G2#†	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

†G2# = G2A# + G2B#
1 = High 0= Low Level
X = Don't Care

Figure 7-12. Internal Logic and Truth Table of 74S138

When A5, A6 and A7 are high and ADS# is strobed, E2 on the decoder is enabled. Here, M/IO# is low and D/C# is high, enabling inputs E1 and E3 of the decoder. When RDY# is active, E2 is disabled and the address is no longer valid. Reset and timeout signals may also disable the address decoding logic.

Because of its non-pipelined address bus, the basic I/O interface design for the Intel486 processor does not require address latches following the decoder.

The number of decoders needed is usually a factor of memory mapping complexity.

7.2.5 Data Transceivers

Data transceivers are used for isolating the processor's data bus from the external data bus and increasing the drive capability for larger fanouts. Transceivers are used to avoid the contention on the data bus caused when slow devices perform a delayed read on the databus following a read cycle. When a write cycle follows a read cycle, the Intel486 processor may drive the data bus before a slow device can remove its outputs from the bus, creating potential bus contention. If the load on the Intel486 processor's data pins meets device specifications, and if the data float time of the device is short enough, the transceivers can be omitted from the system.

There should be enough transceivers in the bus interface to accommodate the device with the most inputs and outputs on the data bus. Only eight transceivers are needed if the widest device has 16 data bits and if the I/O device addresses are connected only to the lower byte of the data bus.

The 74S245 transceiver is controlled through two input signals:

- Data Transmit/Receive (DT/R#)—The transceiver for write cycles is enabled when this signal is high, and a read cycle is enabled when it is low. This signal is simply a latched version of the Intel486 processor's W/R# output.
- Data Enable (DEN#)—When low, this input enables the transceiver outputs. It is generated by the byte swapping logic and by the BE3#–BE0# signals.

Data transceivers may be combined with byte swapping logic, depending upon whether a 32 bit to 8/16/32-bit transfer is used. The implementation details of this logic are discussed in previous sections.

7.2.6 Recovery and Bus Contention

Although data transceivers help to avoid data bus contention, I/O devices may still require a recovery period between back-to-back accesses. At higher Intel486 processor clock frequencies, bus contention is more problematic, particularly because of the long float delay of I/O devices, which can conflict with read data from other I/O devices or write data from the CPU. To ensure proper operation, I/O devices require a recovery time between consecutive accesses. All slave devices stop driving data on the bus on the rising edge of IOR#. After a delay which follows this rising edge, the data bus floats.

When other devices drive data on to the bus before the data from the previous access floats, bus contention occurs. The Intel486 processor has a fast cycle time (30 ns at 33 MHz), and the probability of bus contentions must be addressed.

Bus control logic should implement recovery to eliminate bus contention. The logic generates a RECOV signal until the data from the previous read floats. It may or may not be possible to enforce this recovery with the hardware counter. The hardware counter method may not be feasible when recovery times are too fast for the hardware counter (i.e., when recovery time is in nanoseconds). In this case, recovery time can be enforced in software using NOPs and delay loops or using a programmable timer.

The advantages of using hardware-enforced recovery are transparency and reliability. When moving to higher processor clock speeds, no change is needed in the I/O device drivers. For these reasons, hardware enforced I/O recovery time is recommended.

7.2.7 Write Buffers and I/O Cycles

The Intel486 processor contains four write buffers to enhance the performance of consecutive writes to memory. Writes are driven onto the external bus in the same order in which they are received by the write buffers. Under certain conditions, a memory read is reordered in front of the writes pending in the write buffer even though the writes occurred earlier in program execution (see Chapter 4, “Bus Operation” for details).

However, I/O cycles must be handled in a different manner by the write buffers. I/O reads are never reordered in front of buffered memory writes. This ensures that the Intel486 processor updates all memory locations before reading status from an I/O device.

The Intel486 processor never buffers single I/O writes. When processing an I/O write instruction (OUT, OUTS), internal execution stops until the I/O write actually completes on the external bus. This allows time for the external system to drive an invalidate into the Intel486 processor or to mask interrupts before the processor continues to the instruction following the write instruction. Repeated OUTS (REP OUTS) instructions are buffered and the next instruction is not executed until the REP OUTS finishes executing.

7.2.7.1 Write Buffers and Recovery Time

The write buffers, in association with the cache, have certain implications for I/O device recovery times. Back-to-back write recovery times must be guaranteed by explicitly generating a read cycle to a non-cacheable area in between the writes. Since the Intel486 processor does not buffer I/O writes, the inserted read does not proceed to the bus until the first write is completed. Then, the read cycle executes on the external bus. During this time, the I/O device recovers and allows the next write.

7.2.8 Non-Cacheability of Memory-Mapped I/O Devices

To avoid problems caused by I/O “read arounds,” memory-mapped I/O should not be cached. A read around occurs when a read cycle is reordered in front of a write cycle. If the memory-mapped I/O device is cached, it is possible to read the status of the I/O device before all previous writes to that device are completed. This causes a problem when the read initiates an action requiring memory to be up-to-date.

An example of when a read around could cause a problem follows:

- The interrupt controller is memory-mapped in cacheable memory.
- The write buffer is filled with write cache hits, so a read is reordered in front of the writes.
- One of the pending writes is a write to the interrupt controller control register.
- The read that was reordered (and performed before the write) was to the interrupt controller's status register.

Because the reading of the status register occurred before the write to the control register, the wrong status was read. This can be avoided by not caching memory-mapped I/O devices.

7.2.9 Intel486™ Processor On-Chip Cache Consistency

Some peripheral devices can write to cacheable main memory. If this is the case, cache consistency must be maintained to prevent stale data from being left in the on-chip cache. Cache consistency is maintained by adding bus snooping logic to the system and invalidating any line in the on-chip cache that another bus master writes to.

Cache line invalidations are usually performed by asserting AHOLD to allow another bus master to drive the address of the line to be invalidated, and then asserting EADS# to invalidate the line. Cache line invalidations may also be performed when BOFF# or HOLD is asserted instead of AHOLD. If AHOLD, BOFF# and HOLD are all deasserted when EADS# is issued, the Intel486 processor invalidates the cache line at the address that happens to be on the bus. Cache line invalidations and cache consistency are explained more fully in Chapter 6, "Cache Subsystem."

7.3 I/O CYCLES

The I/O read and write cycles used in a system are a factor of the I/O control logic implementation. Figures 7-13 through 7-16 illustrate an I/O read and write cycle for a typical implementation.

7.3.1 Read Cycle Timing

A new processor read cycle is initiated when ADS# is asserted in T1. The address and status signals (M/IO# = low, W/R# = low, D/C# = high) are asserted. The IOCYC signal is generated by the control logic by decoding ADS#, M/IO#, W/R# and D/C#. IOCYC indicates to an external device that an I/O cycle is pending. The IOR# signal is asserted in the T2 state when IOCYC is valid and RECOV is inactive. The RECOV signal indicates that the new cycle must be delayed to meet the I/O device recovery time or to prevent data bus contention. The I/O read signal (IOR#) signal is not asserted until RECOV is deasserted. Data becomes valid after IOR# is asserted, with the timing dependent on the number of wait states implemented.

In the example, two wait states are required for the slowest I/O device to do a read, and the bus control logic keeps IOR# active to meet the minimum active time requirement. The worst case timing values are calculated by assuming maximum delay in the decode logic and through data transceivers. The following equations show the fastest possible cycle implementation. Wait States should be added to meet the access times of the I/O devices used. Figure 7-13 and 7-14 show the I/O read cycle timing and the critical analysis.

TR_{VD} Read Signal Valid Delay
$TR_{VD} = T_{PLDpd}$
$= 10$ ns

TD_{SU} Read Data Setup Time
$TD_{SU} = T_{BUFpd} + T_{su}^{\dagger}$
$= 9 + 5 = 14$ ns

TD_{HD} Read Data Hold Time
$TD_{HD} = T_{HD}^{\dagger} - T_{BUFpd}$
$= 3 - 9 = -6$ ns

$^{\dagger}T_{SU} = T_{22}$ = Intel486™ processor time (33 MHz)
T_{HD} = Intel486 processor read hold time (33 MHz)

Figure 7-13. I/O Read Timing Analysis

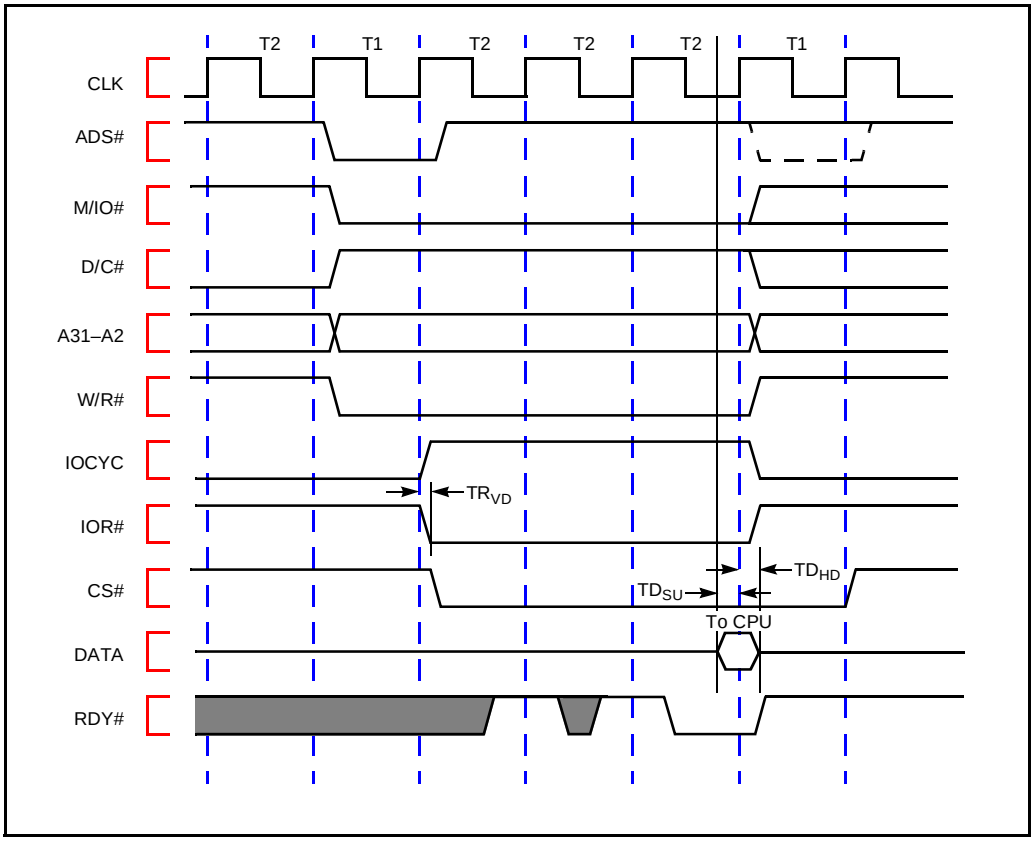


Figure 7-14. I/O Read Timings

7.3.2 Write Cycle Timings

The I/O write cycle is similar to the I/O read cycle with the exception of W/R# being asserted high when sampled rather than low from the Intel486 processor side. This is shown in Figures 7-15 and 7-16.

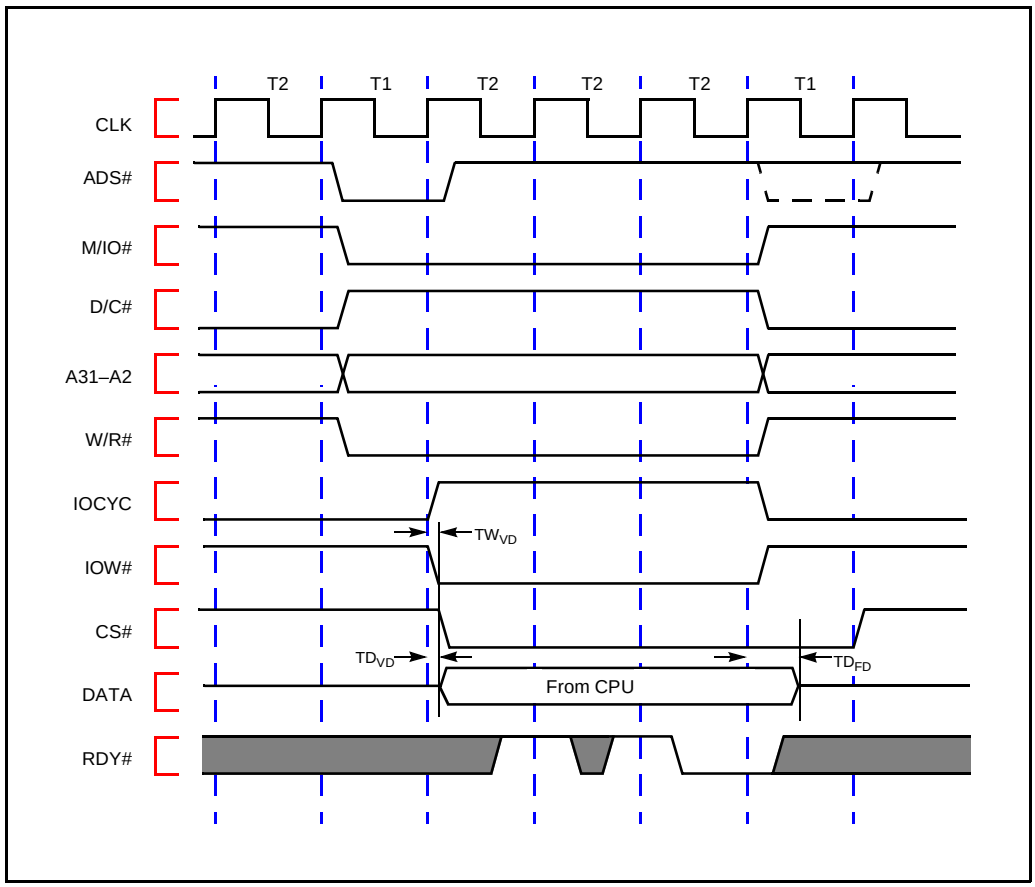


Figure 7-15. I/O Write Cycle Timings

The timing of the remaining signals (the address and status signals) is similar to that of I/O read cycle timings. The processor outputs data in T2. The I/O write signal (IOW#) may be asserted one or two clocks after the chip select. The exact delay between the chip select and the IOW# varies according to the write requirements of the I/O device. Data is written into the I/O device on the rising edge of IOW#, and the processor stops driving data once RDY# data is sampled active. The critical timings for the I/O write cycle are shown in Figure 7-16.

TW_{VD} Write Signal Valid Delay
$TW_{VD} = T_{PLDpd}$
$= 10$ ns

TD_{VD} Write Data Valid Delay
$TD_{VD} = T_{VD}^{\dagger} + T_{BUFpd}$
$= 19 + 9 = 28$ ns

TD_{FD} Write Data Float Time
$TD_{FD} = T_{FD}^{\dagger} - T_{BUFpd}$
$= 0 + 9 = 9$ ns

$^{\dagger}T_{VD} = T_{10}$ = Intel486™ processor write data valid delay (33 MHz)
$T_{FD} = T_{11}$ = Intel486 processor write data float delay (33 MHz)

Figure 7-16. I/O Write Cycle Timing Analysis

Latches and data buffers can improve processor write performance. In Figure 7-17, I/O addresses and data are both latched in a configuration called a posted write. Posted writes help increase system performance by allowing the processor to complete a cycle without wait states. Once the data and address are latched, RDY# can be asserted during the first T2 of an I/O write cycle. Thus, the processor operation and the write cycle to the peripheral device can continue simultaneously. This is illustrated in Figure 7-18. The write cycle appears to be only two clocks long (from ADS# to RDY#) because the actual write overlaps other CPU bus cycles.

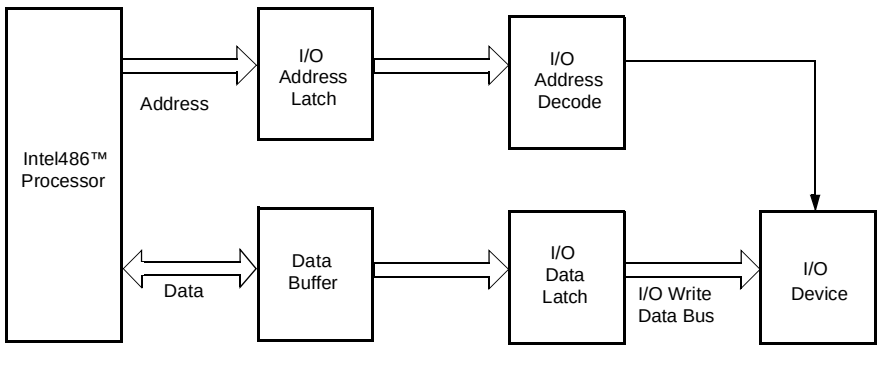


Figure 7-17. Posted Write Circuit

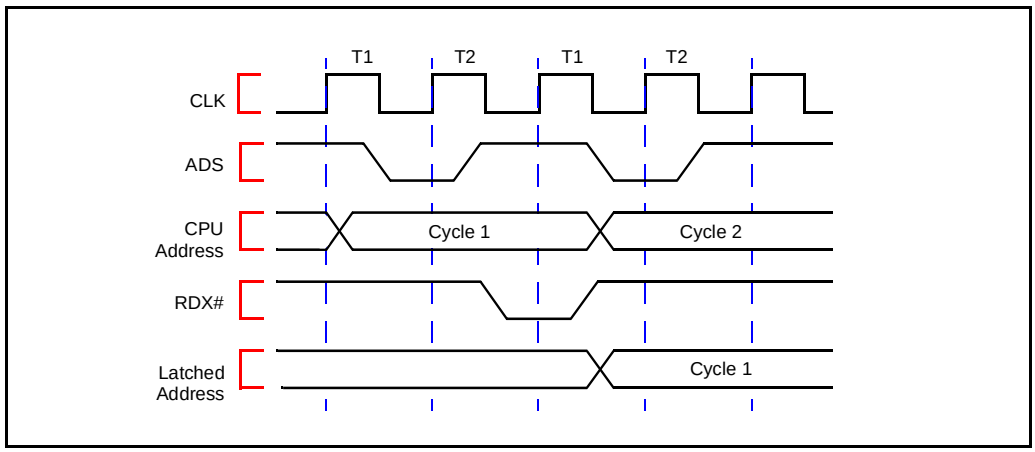


Figure 7-18. Timing of a Posted Write

7.4 DIFFERENCE BETWEEN THE Intel486 DX PROCESSOR FAMILY AND Intel386 PROCESSORS

The IntelDX2 and IntelDX4 processors are integrated chips that is include a CPU, a math coprocessor, and a cache controller. It is fully compatible with its predecessor, the Intel386 DX processor, yet has the following differences:

- Intel486 processor offers dynamic bus sizing to support 8-, 16-, and 32-bit bus sizes, except for the Ultra-Low Power Intel486 GX processor, which supports a 16-bit data bus only. Dynamic bus sizing requires external swapping logic. The Intel386 DX processor supports only 16-bit and 32-bit bus sizes and does not require swapping logic.
- The Intel486 processor has a burst transfer mode which can transfer four 32-bit words from external memory to the on-chip cache using only five clock cycles. The Intel386 DX processor requires at least eight clock cycles to transfer the same amount of data.
- The Intel486 processor has a BREQ output which supports multi-processor environments.
- The Intel486 processor's bus is significantly faster than the Intel386 processor's bus. New features include a 1x clock, parity support[†], burst cycles, cacheable cycles, cache invalidate cycles and 8-bit support. The Hardware Interface and Bus Operation chapters of the *Embedded Intel486™ Processor Family Developer's Manual* explains of the bus functionality and its hardware interface.
- To support the on-chip cache, new bits have been added to control register 0 (CD and NW), new pins have been added to the bus, and new bus cycle types have been added. The on-chip cache must be enabled after reset by clearing the CD and NW bit in CR0.

[†] Not available in the ULP486SX or ULP486GX processors.

- The complete Intel387™ math coprocessor instruction set and register set have been added. No I/O cycles are performed during floating-point instruction execution. The instruction and data pointers are set to zero after FINIT/FSAVE. Interrupt 9 cannot occur, and interrupt 13 occurs instead.
- The Intel486 processor supports new floating-point error reporting modes to ensure DOS compatibility. These new modes require a new bit in Control Register 0 (NE) as well as new pins.
- Six new instructions have been added: Byte Swap (BSWAP), Exchange and Add (XADD), Compare and Exchange (CMPXCHG), Invalidate data cache (INVD), Write-back and Invalidate Data Cache (WBINVD) and Invalidate TLB Entry (INVLPG).
- Two new bits are defined in control register 3 for page table entries and page directory entries.
- A new page protection feature has been added, requiring a new bit in Control Register 0.
- A new alignment check feature has been added, requiring a new bit in the flags register and a new bit in the control register 0.
- The replacement algorithm for the translation lookaside buffer (TLB) is a pseudo least-recently-used algorithm (PLRU), like the one used in the on-chip cache.
- Three new testability registers TR5, TR6 and TR7 have been added for testing of the on-chip cache. TLB testability has been enhanced.
- The prefetch queue has been increased from 16 bytes to 32 bytes. A jump must always execute after code modification to ensure proper execution of the new instruction.
- After reset, the ID in the upper byte of the DX register is 04. The contents of the base register, including the floating-point registers, may be different after reset.

Refer to the individual Intel486 processor datasheets for more information about these features.

7.5 INTERFACING TO x86 PERIPHERALS

This section discusses the Intel486 processor interface to two peripheral devices from the x86 family: the 8041 and the 82C59A. Not all systems use these separate devices, however the examples explain in detail many of the issues surrounding slave I/O and interrupts.

7.5.1 Universal Peripheral Interface

Universal peripheral interface (UPI) devices allow customized solutions for peripheral device control. These microcontrollers have a slave interface on-board and include an 8-bit CPU, ROM, RAM, an I/O timer/counter and a clock. Intel supplies an EPROM implementation, which includes the 8741 and 8742 microcontrollers. The 8742 has a 2 K x 8-bit ROM and 256 K x 8-bit RAM, an eight-bit timer/counter and 18 programmable I/O pins. It also has an 8-bit status register and two data registers for asynchronous slave-to-master interfacing. The 8742 supports DMA, interrupt and polled operations.

The 32-bit Intel486 processor requires 32-bit-to-8-bit byte-steering logic to interface to an 8-bit UPI device.

7.5.2 82C59A Interface

The following discussion of interrupt-driven processor environments is a helpful preface to the section on interfacing Intel486 processor systems to the 82C59A programmable interrupt controller. It also provides a context to review other interrupt controller implementations.

In a microcomputer system, the CPU must efficiently service I/O devices such as keyboards and display monitors to minimize overhead. One technique is polling, in which the processor tests each device in sequence to determine whether servicing is needed. A large portion of the main program must be devoted to polling, at a cost of system throughput.

Interrupts provide a more efficient and desirable alternative for servicing I/O devices. Using interrupts, a hardware signal can cause the main program to change its execution path. These interrupts are acknowledged only between instructions—with the exception of the bus error signal. The Intel486 processor reacts to interrupts by saving the program address and then performing special interrupt processing (as explained in the *Embedded Intel486™ Processor Family Developer's Manual*). Once the current program address and flags are saved on a stack, the Intel486 processor receives an eight-bit vector identifying an entry in the interrupt table that contains the starting address of the interrupt service routine. The vector interrupt allows a hardware mechanism to select a separate service routine for each interrupt source. Once the interrupt service routine is executed, the previous processor state is restored, and program execution resumes. The Intel486 processor can handle up to 256 interrupts/exceptions. Refer to the *Embedded Intel486™ Processor Family Developer's Manual* for the interrupt table.

The interrupt-driven environment increases system throughput and allows more tasks to be accomplished by the processor, thus increasing overall cost-effectiveness.

The 82C59A is a high performance CMOS programmable interrupt controller which manages the interrupt-driven Intel486 processor system environment. It accepts requests from peripheral devices and determines device priorities. The 82C59A provides the processor with an eight-bit vector interrupt. The interrupt points to an address in the vector table, and the processor's INTA# signal (generated by the bus controller logic) enables the vector data on the data bus.

Individual 82C59A devices can be cascaded to accommodate up to 64 interrupts. Later sections discuss how to implement such configurations.

7.5.2.1 Single Interrupt Controller

Figure 7-19 shows a basic I/O interface between the Intel486 processor and a single 82C59A device. The address decoder generates the chip select (CS#) signal, while the bus control ready logic generates the interrupt acknowledge (INTA#), write (WR#) and read (RD#) signals. In this example, the 82C59A is used in the master mode since the SP/EN# pin is high. The A0 address pin is used to decipher various processor command words and to determine the status that the processor wishes to read. The A0 pin is connected to the processor's A2 pin and is also used to distinguish between two consecutive interrupt acknowledge cycles. The 82C59A register address must therefore be located at two consecutive doubleword boundaries.

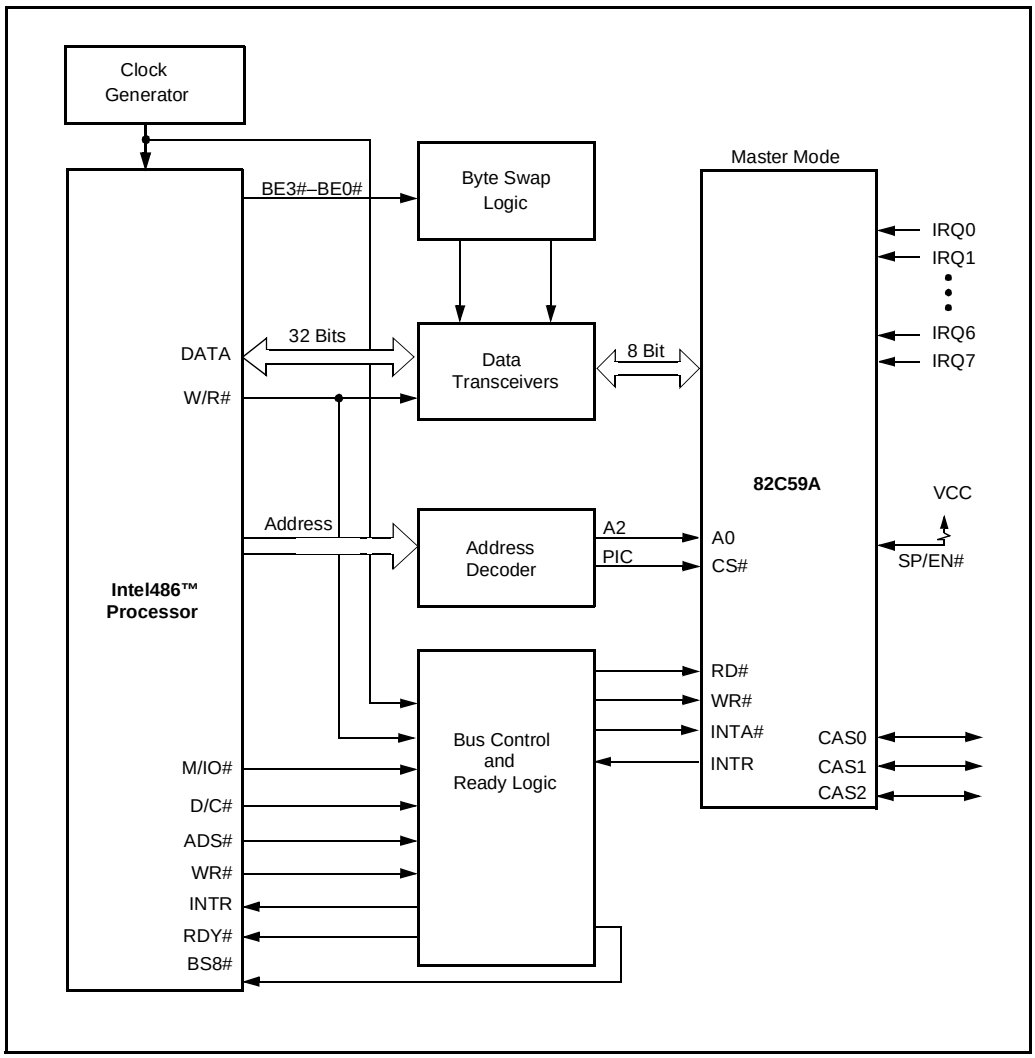


Figure 7-19. Intel486™ Processor Interface to the 82C59A

An interrupt activates the Interrupt output of the 82C59A, which is connected to the INTR input (interrupt request) of the Intel486 processor. The processor automatically performs two consecutive interrupt acknowledge cycles. The 82C59A device's timings are as follows:

- Each interrupt acknowledge cycle must be extended by at least one wait state, which is implemented by the wait state generator logic described in Section 7.2, "Basic Peripheral Subsystem."
- Four idle cycles must be inserted between two interrupt acknowledge cycles.

7.5.2.2 Cascaded Interrupt Controllers

Figure 7-20 shows how several interrupt controllers can be cascaded to handle up to 64 interrupt requests. One device acts as the master and the rest as slaves. The interface between these devices resembles the single device interface with the following additional features:

- The cascaded address outputs are used to provide address and chip select signals for the slave controllers.
- The interrupt request lines (IR7–IR0) of the master controller are connected to the INTR outputs of the slave devices.

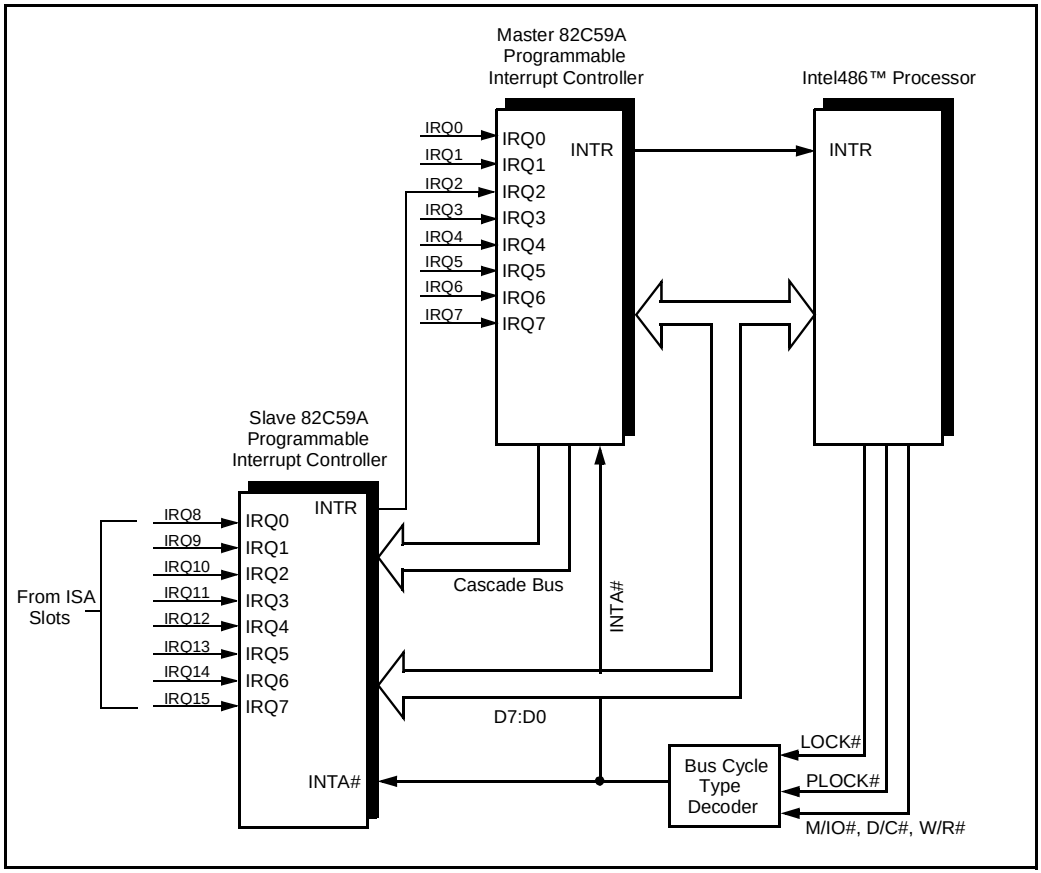


Figure 7-20. Cascaded Interrupt Controller

The function of each slave controller is to identify the priorities among eight interrupt requests and generate a single interrupt request for the master controller. The master controller must identify the priorities among eight slave controllers and transmit a single interrupt request to the Intel486 processor.

The timing interface resembles that used for single devices. During the first interrupt acknowledge cycle, all the 82C59A devices freeze the states of their interrupt request inputs. The master controller outputs the cascaded address to select the slave controller that is generating the request with the highest priority. During the second interrupt acknowledge cycle, the selected slave controller outputs an interrupt vector to the Intel486 processor.

7.5.2.3 Handling More than 64 Interrupts

If an Intel486 processor-based system requires more than 64 interrupt request lines, a third 82C59A device level in polled mode is added to the configuration shown in Figure 7-19. Once the third-level interrupt controller receives an interrupt, it drives an interrupt request input to the slave controller on the second level. The second-level slave controller then sends an interrupt request to the master controller, which in turn interrupts the processor. The slave controller then returns a service routine vector to the Intel486 processor. The service routine must include commands to poll the third level to determine the source of the interrupt request.

The additional hardware required to implement this configuration includes additional 82C59A devices and the chip-select logic.

7.6 Intel486™ PROCESSOR LAN CONTROLLER INTERFACE

This section describes two LAN interface solutions using Intel controllers: the 82596CA coprocessor and the PCI-compliant 82557 controller for fast Ethernet networks.

7.6.1 82596CA Coprocessor

The 82596CA coprocessor (hereafter referred to generically as the 82596 coprocessor) is a 32-bit multitasking LAN coprocessor which implements the carrier-sense, multiple-access and collision-detect (CSMA/CD) link access protocol. The coprocessor supports a wide variety of networks. It executes high-level commands, and it performs command chaining and inter-processor communication via memory shared with the Intel486 processor. This relieves the processor of all time-critical local-network control functions.

The coprocessor's features include:

- Complete CSMA/CD Functions
  - — Complete media access control (MAC) functions
  - — High-level command interface
  - — Manchester encoding or NRZ encoding and decoding
  - — IEEE 802.3 or CCITT HDLC frame delimiting

- Industry-Standard Network Support
  - IEEE 802.3 (Ethernet, Ethernet Twisted Pair, Cheapernet, StarLAN, etc.)
  - IBM PC Network (baseband and broadband)
  - Proprietary CSMA/CD networks up to 20 Mbits/second
  - HDLC frame delimiting
- Compatible Intel486 Processor Interface
  - Optimized interface to the Intel486 processor bus
  - Shared Intel486 processor bus signals and memory timing
  - Support for Intel486 processor byte ordering
- Architectural Features
  - On-chip DMA
  - Bus Throttle
  - 128-byte receive FIFO, 64-byte transmit FIFO
  - On-chip memory management
  - Network management and diagnostics
  - 82586 software-compatible mode
- Performance Features
  - 9.6 msec interframe spacing for back-to-back frame transmission and reception
  - 80/106 Mbytes/second bus transfer rate (burst) at 25/33 MHz
  - 50/66 Mbytes/second bus transfer rate (non-burst) at 25/33 MHz

Figure 7-21 is a block diagram of the 82596 coprocessor. A serial subsystem interfaces to the physical-layer device for the network. This subsystem performs CSMA/CD media access-control and channel-interface functions. It supports the full set of IEEE 802.3 and other industry-standard and proprietary network functions. A parallel subsystem interfaces to the Intel486 processor. This subsystem contains a data interface unit, a bus interface unit, a 4-channel DMA unit, and a micro-machine command processor. A FIFO subsystem connects the serial and parallel subsystems, allowing them to run asynchronously to one another through a 128-byte receive FIFO and a 64-byte transmit FIFO.

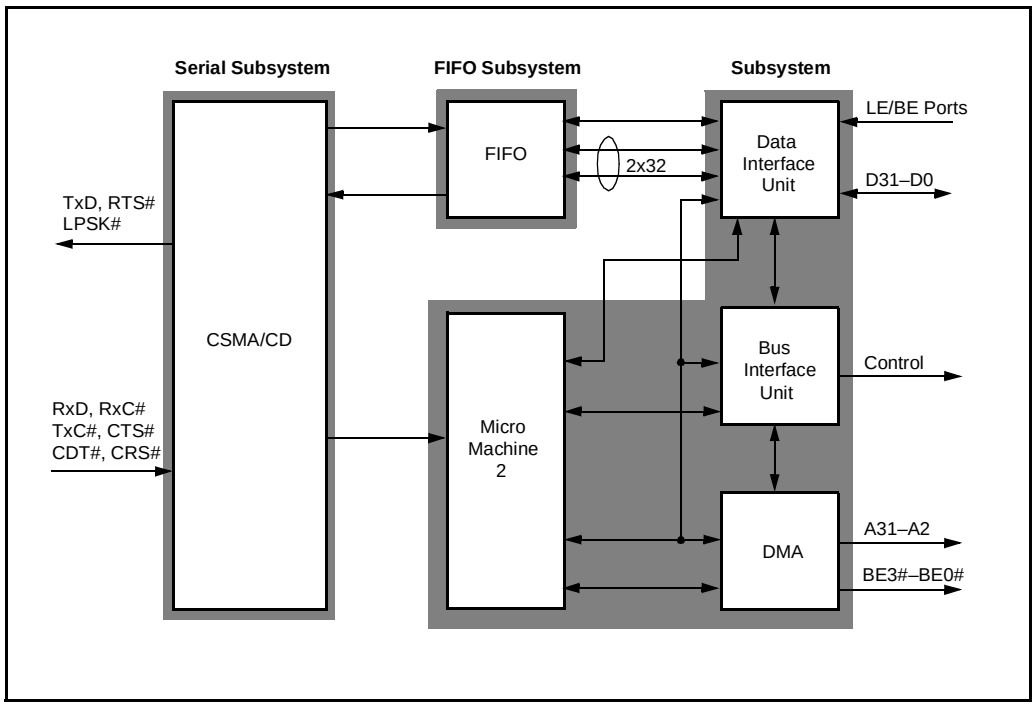


Figure 7-21. 82596CA Coprocessor Block Diagram

The coprocessor can be used in either baseband or broadband networks. It can be configured for maximum network efficiency (minimum contention overhead) for networks of any physical cable length operating at any data rate up to 20 Mbits/second. It features a highly flexible CSMA/CD unit, supporting address lengths from zero to six bytes. It supports 16- or 32-bit CRC. The CRC field can optionally be transferred directly to memory on receive and dynamically inserted on transmit. The CSMA/CD unit can also be configured for full duplex operation or for CSMA/DCR (deterministic collision resolution).

The coprocessor provides a rich set of diagnostic and network management functions, including internal and external loopback, exception condition tallies, channel activity indicators, optional capture of all frames (promiscuous mode), optional capture of erroneous or collided frames, and time-domain reflectometry; for locating fault points on the network cable. The 32-bit statistical counters; monitor CRC errors, alignment errors, overrun errors, resource errors, short frames, and receive collisions.

The coprocessor also features a monitor mode for network analysis. This mode can capture status bytes and update statistical counters of frames monitored, without transferring the contents of the frames to memory. It does this concurrently with frame transmission and frame transfers to memory destined to that station.

The 82596 coprocessor is an extension of the earlier 82586 LAN coprocessor, which interfaces an Ethernet network to a 16-bit Intel bus. The 82596 coprocessor can be configured to run software drivers written for the 82586 device without modification.

7.6.1.1 Hardware Interface

The 82596 coprocessor communicates with the rest of the system via two hardware interfaces: the Intel486 processor bus (parallel) interface and the network (serial) interface, as shown in Figure 7-22. The signals for both interfaces are listed in Table 7-12. The coprocessor's bus cycles (including burst cycles), bus interface timing, bus arbitration method, and signal definitions are compatible with the Intel486 processor. When the coprocessor is not holding the bus, its bus interface signals are floated. The state machines for the Intel486 processor and the 82596 coprocessor are very similar.

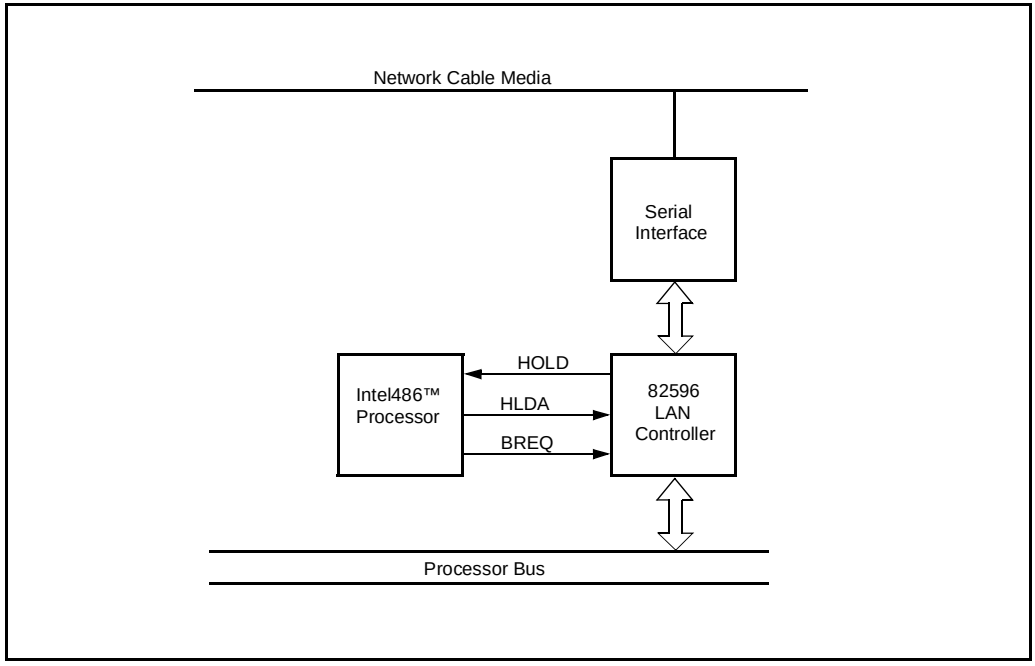


Figure 7-22. 82596CA Application Example

Table 7-12. 82596 Signals (Sheet 1 of 2)

Signal	Type	Description
Address and Data Buses		
A31–A2	O	Address
D31–D0	I/O	Data
BE3#–BE0#	O	Byte-enables
BS16#	I	16-bit data bus size
LE/BE#	I	Little endian or big endian byte ordering
DP3–DP0	I/O	Data parity
PCHK#	O	Parity error
Cycle Definition and Control		
ADS#	O	Address status
W/R#	O	Write or read
PORT#†	I	Port access
RDY#	I	Non-burst data ready
BRDY#	I	Burst data ready
BLAST#	O	Last burst cycle
Bus Control		
CLK	I	Clock
RESET†	I	Reset
INT/INT#	O	Interrupt
BREQ	I	Bus request
HOLD	O	Bus hold request
HLDA	I	Bus hold acknowledgment
AHOLD†	I	Address hold request
BOFF#	I	Bus backoff
LOCK#	O	Bus lock
CA#†	I	Channel attention

†Signals marked with a dagger are not included on, or operate differently than, the Intel486™ processor bus.

Table 7-12. 82596 Signals (Sheet 2 of 2)

Signal	Type	Description
Network (Serial) Interface		
TxD†	O	Transmit data
TxC#†	O	Transmit clock
LPBK#	O	Loopback
RxD	I	Receive data
RxC#	I	Receive clock
RTS#	O	Request to send
CTS#	I	Clear to send
CRS#	I	Carrier sense
CDT#	I	Collision detect

†Signals marked with a dagger are not included on, or operate differently than, the Intel486™ processor bus.

These similarities between the Intel486 processor and the 82596 coprocessor simplify bus arbitration when the processor and the coprocessor are the only two bus masters on the processor bus. The HOLD and HLDA signals can be used for handshake arbitration and BREQ from the processor can trigger the coprocessor's bus throttle timers when needed, as shown in Figure 7-23.

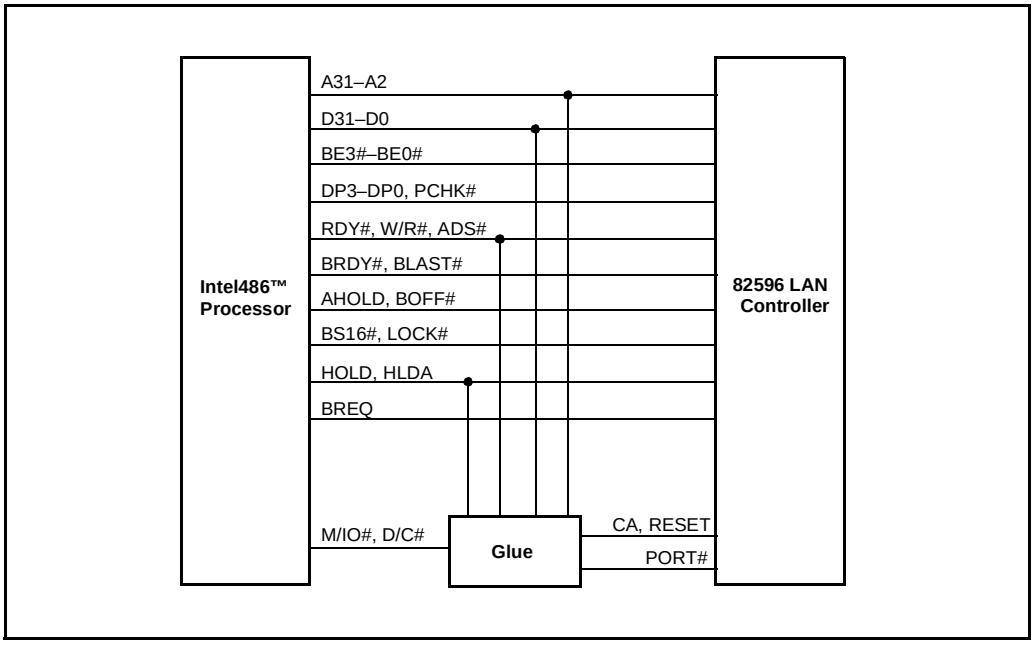


Figure 7-23. 82596-to-Processor Interfacing

Access to memory and I/O resources can be overlapped between the processor and the coprocessor with the bus backoff (BOFF#) output to the processor. The BOFF# overlapping method avoids the need for a time-consuming bus hold arbitration (HOLD and HLDA) and it is done without the risk of deadlock.

The coprocessor signals have the same significance as on the Intel486 processor bus, except for the AHOLD signal. Because there is no internal cache to invalidate on the coprocessor, this input is used to release the coprocessor address bus when an external cache controller needs to perform a cache invalidation cycle.

7.6.1.2 Processor and Coprocessor Interaction

The 82596 coprocessor interacts with the processor bus as either a bus master or a slave (port access mode). In normal operation, it is a bus master which moves data between the system memory and the coprocessor's control registers or internal FIFOs. The coprocessor can use the same burst cycles, bus hold, address hold, bus backoff, and bus lock operations that the Intel486 processor uses.

The coprocessor and the processor communicate through shared memory, as shown in Figure 7-24. The processor and the coprocessor normally use the interrupt (INT/INT#) and channel attention (CA) signals to initiate communication, using a system control block of memory for command and status storage. INT/INT# alerts the processor to a change of contents in the system control block. By asserting CA, the processor causes the coprocessor to examine the system control block contents for the change.

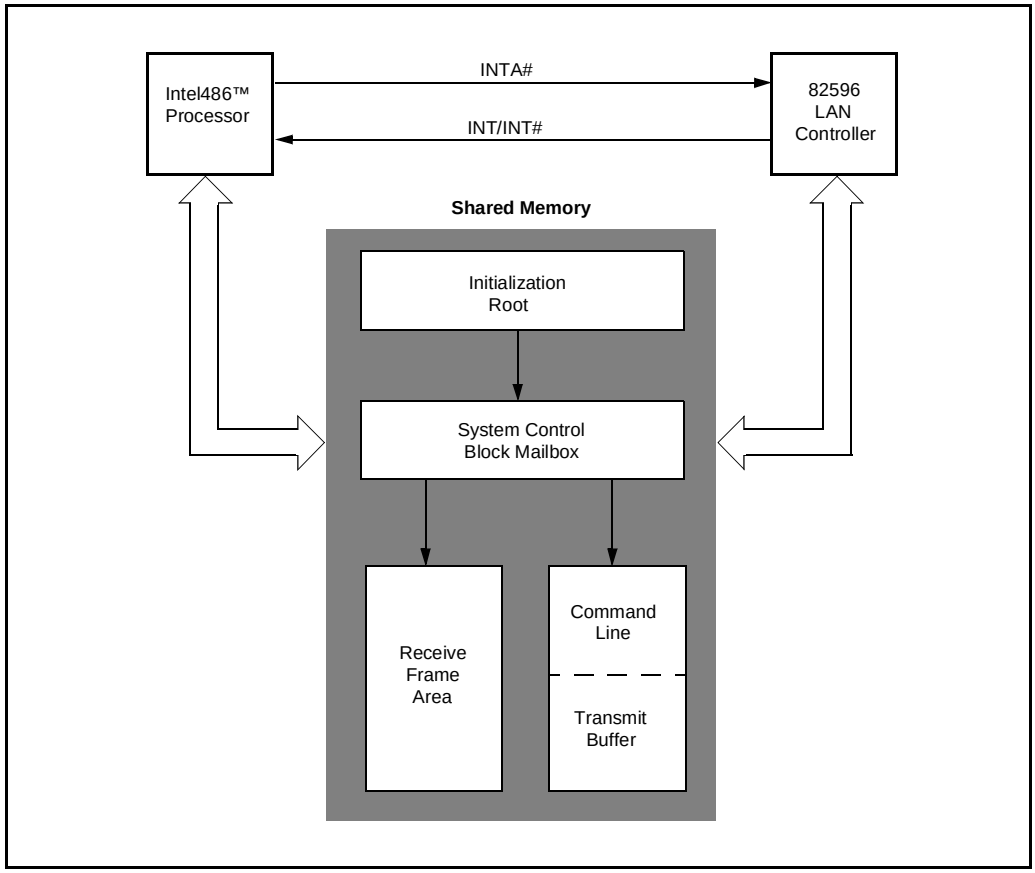


Figure 7-24. 82596 Shared Memory

The coprocessor executes its command list from shared memory and, in parallel, receives frames from the network and places them in shared memory. The processor manages the shared memory, which contains command chains and bidirectional data chains. The coprocessor executes the command chains. An on-chip DMA controls four channels, which allow autonomous transfers of data blocks independently of the processor. Buffers containing erroneous or collided frames can be automatically recovered without processor intervention. The processor becomes involved only after a command sequence has finished executing, or after a sequence of frames has been received and stored, ready for processing.

In addition to this normal operating mode, the processor can initiate a port access in the coprocessor. This mode may be entered whenever the coprocessor is not actively driving the bus. It allows the processor to write an alternate system configuration pointer, write an alternate dump command and pointer (used for troubleshooting a no-response problem), perform a software reset, or perform a self-test.

7.6.1.3 Memory Structure

The memory shared by the processor and 82596 coprocessor consists of four parts.

- Initialization root
- System control block
- Command list (including transmit buffer)
- Receive frame area

The command list functions as a program. Individual commands are placed in blocks of memory called command blocks. These command blocks contain the parameters and status of high-level commands used by the processor to control the operation of the coprocessor.

One of three memory addressing modes can be used:

- 82586 Mode: Uses 24-bit addresses with all shared memory structures residing in one 64-Kbyte segment.
- 32-bit Segmented Mode: Uses 32-bit addresses with all shared memory structures residing in one 64-Kbyte segment.
- Linear Mode: Uses 32-bit addresses with no restrictions on the placement of any shared memory structure.

Big-endian and little-endian byte ordering schemes are supported. For compatibility with the Intel486 processor, the little-endian scheme should be used.

7.6.1.4 Media Access

The 82596 coprocessor accesses the cable-media network through the serial subsystem. This subsystem performs the full set of IEEE 802.3 CSMA/CD media access control (MAC) sublayer and channel interface functions, including framing, preamble generation and stripping, source address generation, destination address checking, short-frame (runt packet) detection, and automatic-length field handling. Data rates up to 20 Mbits per second on the cable media are supported. IEEE 802.3 and HDLC CRC generation and checking is supported.

The following media access methods are supported:

- CSMA/CD
- Deterministic collision resolution
- Full duplex

The following IEEE standards are supported:

- 1BASE5
- 10BASE5
- 10BASE2
- 10BROAD36

- Proposed 10BASE-F
- Proposed 10BASE-T

7.6.1.5 Transmit and Receive Operation

Most of the bus traffic initiated by the coprocessor consists of DMA transfers of frame data. The coprocessor transmits data as a series of frames by executing a series of high-level commands from the command list in memory. These commands are fetched by the coprocessor and executed in parallel with processor operations. A single transmit command contains all the information necessary to prepare and execute the transmission of one or more data frames.

The data consists of a buffer descriptor and a data buffer containing the actual data. These may also be chained into a linked list of buffer descriptors and associated data buffers. A frame with a long data field can therefore be transmitted using several shorter buffers chained together. This is useful when assembling frames which include nested headers generated by independent software modules.

In order for the coprocessor to receive frames, the processor must first dedicate an area of memory as a receive buffer space and enable the coprocessor for reception. Frames arrive unsolicited at the coprocessor network interface. The coprocessor must always be prepared to store them in an buffer area of memory known as the free frame area. The receive frame area is a list of free frame descriptors and a list of user-prepared buffers. The coprocessor fills the buffers as frames are received, and it reformats the free buffer list into received frame structures. The frame structure stored is the same as that for frames to be transmitted. The data contained in the buffers is transferred by means of the on-chip DMA controller. This allows bidirectional, autonomous transfer of data blocks partitioned as buffers or chained into frames. Buffers which contain errors are recovered automatically without processor intervention.

The coprocessor monitors the frames presented on the serial interface for a destination address which corresponds to its own unique address, one or more multicast addresses, or the broadcast address. When a match is found, the frame's destination, source addresses, and length field are stored, and the data field is placed in the next available buffer. As one buffer is filled, the device automatically links the next available buffer until the entire frame is stored. This technique accommodates buffer sizes which are much shorter than the maximum permitted frame length.

When a frame has been received without error, several housekeeping tasks are performed by the coprocessor. If a frame error occurs, the coprocessor re-initializes the DMA pointers and reclaims any buffers to which the frame had been allocated.

7.6.1.6 Bus Throttle Timers

The 82596 coprocessor's use of the processor bus is regulated with the coprocessor's bus throttle timer logic. These timers are independently programmed and can be triggered internally or externally. The operation of the timers is shown in Figure 7-25. Two timers are associated with the bus throttle function:

- TON Timer: Defines the maximum time the coprocessor can remain bus master.
- TOFF Timer: Defines the minimum time the coprocessor must wait before re-asserting the HOLD output to request the bus again.

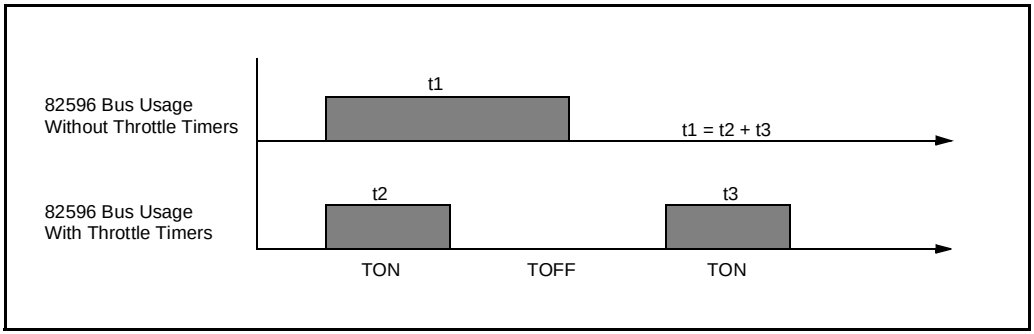


Figure 7-25. Bus Throttle Timers

If the timers are configured to be triggered internally, the coprocessor monitors the length of time that the HLDA input is held asserted. When this time exceeds the time programmed in the TON timer, the coprocessor relinquishes the bus by de-asserting HOLD; and starts the TOFF timer.

If the timers are configured externally, assertion of the BREQ; input causes the coprocessor to start the TON timer. Upon timeout, the coprocessor relinquishes the bus and starts the TOFF timer. This latter configuration is particularly useful in the Intel486 processor environment, where the processor's BREQ output can be tied directly to the coprocessor's BREQ input.

7.6.1.7 Design Considerations

The glue logic for interfacing the 82596 coprocessor to the Intel486 processor can be contained in a single Intel 85C220 PLD, as shown in Figure 7-26. This logic provides four functions:

- Generate channel attention (CA) input to the coprocessor.
- Generate reset (RESET) input to the coprocessor.
- Generate processor port access (PORT#) input to the coprocessor.
- Drive the M/IO# and D/C# processor bus signals when the coprocessor is bus master.

The coprocessor's RESET input is referred to in Figure 7-26 and the text below as "596RESET" to distinguish it from the processor's RESET.

To assert the CA or 596RESET signals, the processor drives a memory-mapped I/O cycle. During such a cycle, address decode is done while monitoring CLK, ADS#, HLDA, and D0 to distinguish CA from 596RESET. A similar memory-mapped cycle is used to de-assert the signal. The HLDA input to the 85C220 PLD gates the logic, so that CA or 596RESET is generated only when HLDA is de-asserted (i.e., when the coprocessor is not bus master).

The PORT# input to the coprocessor can be generated by combinatorial logic which has an address decode qualified by ADS# and CLK. This asserts the PORT# output for one clock. While PORT# is asserted, the coprocessor treats the data bus as containing slave control information. System software must ensure that the coprocessor is idle while the processor executes a port access. This guarantees that the coprocessor does not attempt to acquire the bus by asserting HOLD. Failure to comply with this restriction may result in the coprocessor entering an undefined state.

The CA, 596RESET, and PORT# signals are generated according to the equations shown in Figure 7-26. The M/IO# and D/C# signals are also generated by the glue logic. When both HOLD and HLDA are asserted, indicating that the coprocessor has requested and been granted the bus, M/IO# and D/C# must be driven high.

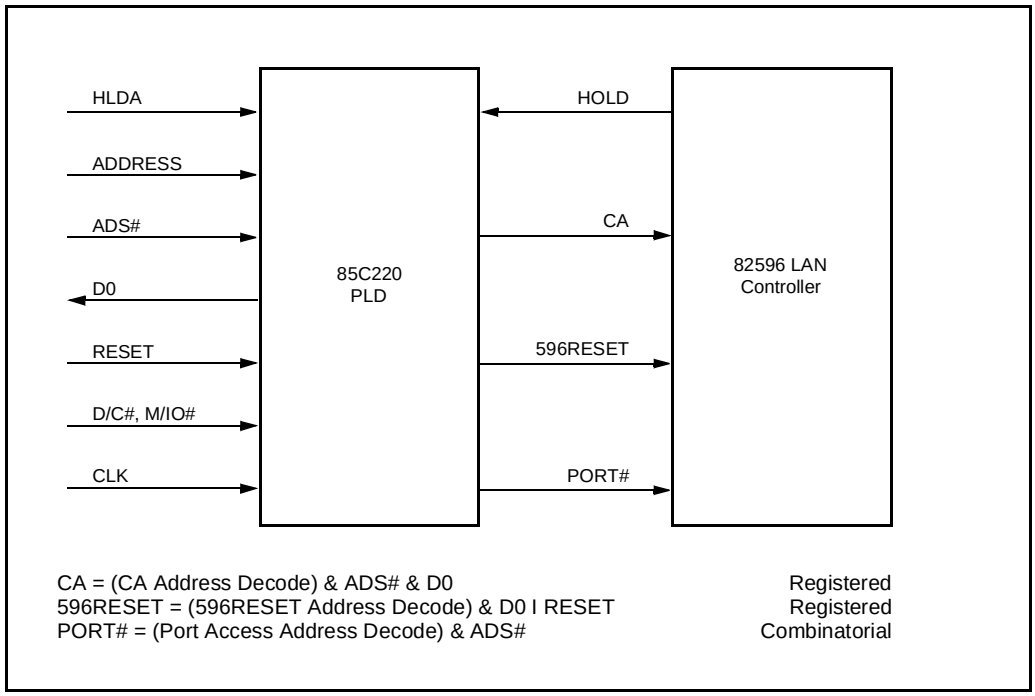


Figure 7-26. 596RESET, CA, and PORT# Equations

Caching of the coprocessor memory structures in the Intel486 processor internal cache may be disadvantageous, because these memory structures are not directly executable by the processor. Typically, most coprocessor bus activity consists of receiving and transmitting frames, managing the receive frame area, and prefetching descriptor pointers. The system control block is typically accessed only once by the processor for every update of this area made by the coprocessor. The processor gains no advantage from caching locations which are used only once. Also, each time a cached memory location is written to by the coprocessor, a cache invalidation cycle must be performed.

For systems in which caching is obligatory, external logic must monitor ADS# and W/R# and drive the EADS# cache invalidation input to the processor.

7.6.1.8 82596 Co-processor Performance

With a 25-MHz clock, the 82596 coprocessor can transfer data at up to 80 Mbyte/second in burst cycles, or 50 Mbytes/second in non-burst cycles. With a 33-MHz clock, the rates are 106 Mbytes/second for burst and 66 Mbytes/second for non-burst. Most transfers in a Intel486

processor environment can be in burst mode. Ethernet provides data at a maximum instantaneous rate of 1.25 Mbytes/second. The coprocessor, however, requires approximately 0.25 Mbytes/second additional bandwidth for frame processing, updating various command blocks, and descriptors. This brings the maximum bus bandwidth requirement to approximately 1.5 Mbytes/second. The coprocessor therefore requires only a small fraction of the available processor bus bandwidth.

Several variables affect the total bandwidth required. The main factors are:

- Use of burst cycles for memory transfers
- Number of memory wait states per transfer
- Processor bus clock (CLK) frequency
- Frame and buffer size

Table 7-13 compares the percentage of 32-bit bus bandwidth used under some of these conditions. These are worst-case numbers and over-estimate typical network loading. Typical bus utilization numbers at non-peak rates are lower.

Table 7-13. 82596 Bus Bandwidth Utilization

Bus Frequency	Frame Size	Burst (0 ws)	Non-Burst (0 ws)	Non-Burst (1 ws)
25 MHz	64 bytes	3.33%	4.05%	5.65%
	1,518 bytes	1.70%	2.63%	3.90%
33 MHz	64 bytes	2.52%	3.07%	4.29%
	1,518 bytes	1.29%	1.99%	2.95%

7.6.2 82557 High Speed LAN Controller Interface

7.6.2.1 82557 Overview

The 82557 is Intel's first highly-integrated 32-bit PCI LAN controller for 10 or 100 Mbps Fast Ethernet networks. The 82557 offers a high performance LAN solution while maintaining low-cost through its high integration. It contains a 32-bit PCI Bus Master interface to fully utilize the high bandwidth (up to 132 Mbytes per second) available to masters on the PCI bus. The bus master interface can eliminate the intermediate copy step in Receive (RCV) and Transmit (XMT) frame copies, resulting in faster processing of these frames. The 82557 maintains a similar memory structure to the 82596 LAN Co-processor; however, these memory structures have been streamlined for better network operating system (NOS) interaction and improved performance.

The 82557 contains two large receive and transmit FIFOs (3 Kbytes each) which prevent data overruns or underruns while waiting for access to the PCI bus, and enables back-to-back frame transmission within the minimum 960 nanosecond inter-frame spacing. Full support for up to 1 Mbyte of FLASH enables network management support via Intel FlashWorks utilities as well as remote boot capability (a BIOS extension stored in the FLASH which could allow a node to boot itself off of a network drive). For 100 Mbps applications, the 82557 contains an IEEE MII compliant interface to the Intel 82553 serial interface device (or other MII compliant PHYs)

which will allow connection to 100/10 Mbps networks. For 10 Mbps networks, the 82557 can be interfaced to a standard ENDEC device (such as the Intel 82503 Serial Interface), while maintaining software compatibility with 100 Mbps solutions.

The 82557 is designed to implement cost effective, high performance PCI add-in adapters, embedded PCs, or other interconnect devices such as hubs or bridges. Its combination of high integration and low cost make it ideal for these applications.

7.6.2.2 Features and Enhancements

The following list summarizes the main features of the Intel 82557 controller:

- Glueless 32-bit PCI Bus Master Interface (Direct Drive of Bus), compatible with *PCI Bus Specification*, Revision 2.1
- 82596-like chained memory structure
- Improved dynamic transmit chaining for enhanced performance
- Programmable transmit threshold for improved bus utilization
- Early receive interrupt for concurrent processing of receive data
- FLASH support up to 1 Mbyte
- Large on-chip receive and transmit FIFOs (3 Kbytes each)
- On-chip counters for network management
- Back-to-back transmit at 100 Mbps
- EEPROM support
- Support for both 10 Mbps and 100 Mbps networks
- Interface to MII-compliant PHY devices, including Intel 82553 Physical Interface component for 10/100 Mbps designs
- Compatible with IEEE 802.3u 100Base-T, TX, and T4
- Interface to Intel 82503 or other serial device for 10 Mbps designs: Compatible with IEEE 802.3 10Base-T
- Autodetect and autoswitching for 10 or 100 Mbps network speeds
- Capable of full or half duplex at 10 and 100 Mbps
- 160-Lead QFP package

Figure 7-27 shows a high level block diagram of the 82557 part. It is divided into three main subsystems: a parallel subsystem, a FIFO subsystem and the 10/100 Mbps CSMA/CD unit.

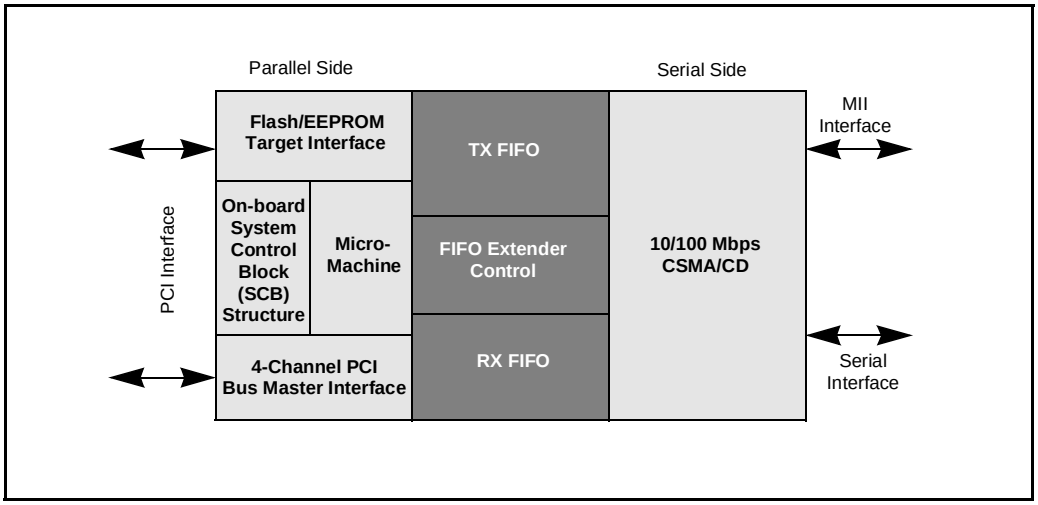


Figure 7-27. Intel 82557 Block Diagram

7.6.2.3 PCI Bus Interface

The PCI bus interface enables the 82557 to interact with the host system via the PCI bus. It provides the control, address and data interface to implement a PCI-compliant device. The 82557 operates as both a master and slave on the PCI bus. As a master, the 82557 interacts with the system main memory to access data for transmission or deposit received data.

As a slave, some 82557 control structures are accessed by the host CPU which reads or writes to these on-chip registers. The CPU provides the 82557 with the necessary action commands, control commands, and pointers which enable the 82557 to process RCV and XMT data. The PCI bus interface also provides the means for configuring PCI parameters in the 82557.

7.6.2.4 82557 Bus Operations

After configuration, the 82557 is ready for its normal operation. As a Fast Ethernet Controller, the role of the 82557 is to access transmitted data or deposit received data. In both cases the 82557, as a bus master device, initiates memory cycles via the PCI bus to fetch/deposit the required data. In order to perform these actions, the 82557 is controlled and examined by the CPU via its control and status structures and registers. Some of these control and status structures reside on-chip and some reside in system memory. For access to its Control/Status Registers (CSR), the 82557 serves as a slave (target). The 82557 serves as a slave also while the CPU accesses its 1 Mbyte Flash buffer or its EEPROM.

7.6.2.5 Initializing the 82557

A power-on or software reset prepares the 82557 for normal operation. Because the PCI specification already provides for auto-configuration of many critical parameters such as I/O, memory mapping and interrupt assignment, the 82557 is set to an operational default state after reset. However, the 82557 cannot transmit or receive frames until a Configure command is issued

7.6.2.6 Controlling the 82557

The CPU issues control commands to the Command Unit (CU) and Receive Unit (RU) through the SCB, which is part of the CSR. The CPU instructs the 82557 to Activate, Suspend, Resume or Idle the CU or RU by placing the appropriate control command in the CU or RU control field. A CPU write access to the SCB causes the 82557 to read the SCB, including the Status word, Command word, CU and RU Control fields, and the SCB General Pointer. Activating the CU causes the 82557 to begin executing the CBL. When execution is complete, the 82557 updates the SCB with the CU status, then interrupts the CPU, if configured to do so. Activating the RU causes the 82557 to access the RFA and go into the READY state for frame reception. When a frame is received, the RU updates the SCB with the RU status and interrupts the CPU. It also automatically advances to the next free RFD in the RFA. This interaction between the CPU and 82557 can continue until a software reset is issued to the 82557, at which point the initialization process must be executed again. The CPU can also perform certain 82557 functions directly through a CPU PORT interface.

8

System Bus Design

Chapter Contents

CHAPTER 8
SYSTEM BUS DESIGN

8.1 INTRODUCTION

With the increasing speed of microprocessors, there is a need for efficient input/output devices (such as disks, video controllers and local area network controllers). The key to successfully supporting I/O options is to have a standard means of connecting them to the motherboard. Each computer supports a standard system bus. System bus types include ISA, MCA, EISA, PCI, etc. To exercise the full potential of the Intel486™ processor's 32-bit system buses, support for 32-bit I/O devices is required. This chapter discusses two standards supported by the embedded Intel486 processors: the PCI (Peripheral Component Interconnect) and EISA (Extended Industry Standard Architecture) system buses.

A typical embedded Intel486 processor system includes of a system bus that connects various subsystems. Each subsystem can have its own local bus with local resources and can share global resources. This approach allows each subsystem to perform operations simultaneously on its local bus to yield a significant throughput improvement over single-bus systems.

Intel486 processor system designs may be divided into several subsystems. The first level is the CPU core, which consists of CPU and second-level cache subsystem memory, cache, and I/O control. Each of these subsystems have been described in detail in the previous chapters. The system bus is the vehicle by which the Intel486 processor communicates with other processing subsystems that perform operations simultaneously on their own local buses.

A major concern when designing a system with various subsystems is how to divide the allocated resources. A designer has to decide which resources should be shared by all the subsystems on the system bus and which should be located on the local bus. The choice is based on the individual system's needs in the areas of reliability, integrity, throughput, and performance. Duplicating resources on each local bus, for example, may increase system integrity and local bus performance, but increase system cost.

8.2 SYSTEM BUS INTERFACE

Subsystems must communicate with one another. Each may be able to stand alone as a processing unit but must share information. The system bus is the vehicle by which information may be transferred. In addition, a standard system bus provides a format for all vendors to follow when building boards or subsystems. This standard allows boards from multiple suppliers to be used in a system. For a subsystem to access the system bus, the protocol signals associated with that bus must be provided. In addition, buffers and drivers are needed to provide the necessary AC and DC drive capability for the address, data, and control signals.

8.3 EISA BUS: SYSTEM DESIGN EXAMPLE

8.3.1 Introduction to the EISA Architecture

EISA represents an extension to the Industry Standard Architecture (ISA) but maintains compatibility with ISA expansion boards and software. EISA provides the following important enhancements:

- 32-bit address bus for CPU, DMA and bus masters
- 32-bit data transfers for CPU, DMA, I/O devices and bus masters
- High speed synchronous bus transfers of 1.5 cycles per doubleword
- Automatic translation of bus cycles between EISA and ISA master and slaves
- Up to 33 Mbytes/second transfer rates for bus masters and DMA devices
- Interrupts are programmable to be edge- or level-sensitive
- Support of intelligent bus master peripheral controllers

The EISA bus is designed to handle wider address and data buses than those of ISA. All EISA connector, performance, and function enhancements are a superset to those of ISA. EISA maintains full compatibility with ISA expansion boards and software.

Bus masters and multiple processors on the EISA bus can be synchronized to a common clock for greater performance. Burst cycles can be executed at 33 Mbytes/sec transfer rate and a standard EISA cycle can transfer data in two cycles. However, CPUs are permitted to generate 1.5-clock “compressed” cycles for slaves that request such cycles.

EISA systems can support DMA transfers with 32-bit addressability, and with 8-, 16-, or 32-bit data. 32-bit DMA devices can transfer data at 33 Mbytes/sec using burst cycles.

EISA-based computers support a bus master architecture for intelligent peripherals. The bus master provides a high-speed channel with data rates up to 33 Mbytes/sec. The bus master provides localized intelligence with a dedicated I/O processor and local memory to relieve the host of sophisticated memory access functions. Peripherals that use bus mastering techniques include disk controllers, LAN interfaces, data acquisition systems and certain classes of graphic controllers.

The EISA bus provides a mechanism for data size translation which is useful when it is transferring data between 16-bit ISA bus masters and 8-bit or 16-bit memory, I/O slaves, or DMA devices. The system board also provides a mechanism for transactions between 16-bit ISA devices and 32-bit EISA devices.

EISA systems provide a centralized arbiter that allows efficient bus sharing between multiple EISA bus masters and DMA devices. An active bus master or DMA device may be preempted when another device needs the bus. Further, if a device does not release the bus once it has been preempted, then the centralized arbiter can reset the device. The EISA arbitration method grants the bus to the DMA devices, the memory controller for DRAM refreshes, the bus masters, and the host CPU in an efficient rotational manner. The rotational scheme provides shorter latencies for DMA devices to ensure compatibility with ISA devices. Bus masters and CPUs have longer latencies because often they have buffers.

8.3.2 An Example EISA Chip Set

Figure 8-1 shows a high-performance system with an Intel486 processor residing on the host bus. Three EISA support devices, an EISA bus controller (EBC), an integrated system peripheral (ISP), and EISA bus buffers (EBB), interface the host bus to the EISA bus. The three devices also communicate with each other.

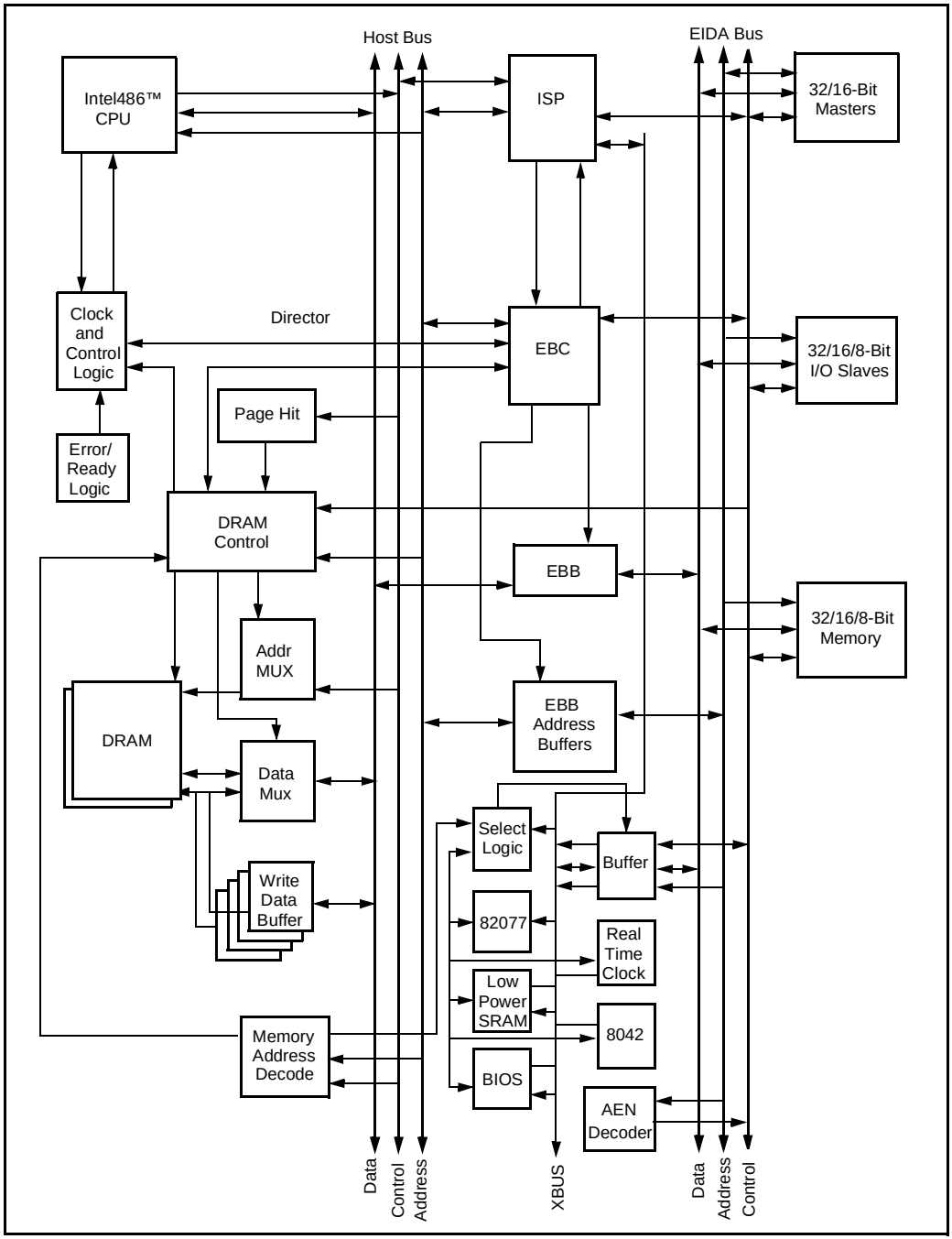


Figure 8-1. Intel486™ Processor System

The EBC interfaces the host bus to the EISA/ISA bus. It provides compatibility with EISA/ISA bus cycles for EISA/ISA standard memory or I/O cycles, zero-wait state cycles, compressed cycles and burst cycles. It also translates host bus cycles to EISA/ISA bus cycles and vice versa. It generates ISA signals for EISA masters and EISA signals for ISA masters and it supports host and EISA/ISA refresh cycles. The EBC supports 8-, 16-, and 32-bit DMA transfers and interacts with the DMA controller. It provides byte-assembly and disassembly for 8-, 16-, and 32-bit data transfers. The EBC generates the appropriate data conversion and assembly control signals to facilitate transfers of various data widths between the host and ISA and EISA buses. The EBC posts processor-to-EISA/ISA write cycles to improve system performance and provides I/O recovery time between back-to-back I/O cycles. Figure 8-2 shows a detailed block diagram of the EBC and its various interface signals to the host, the EISA, ISA, ISP units and the data and address controls. The interfaces are discussed later in this chapter.

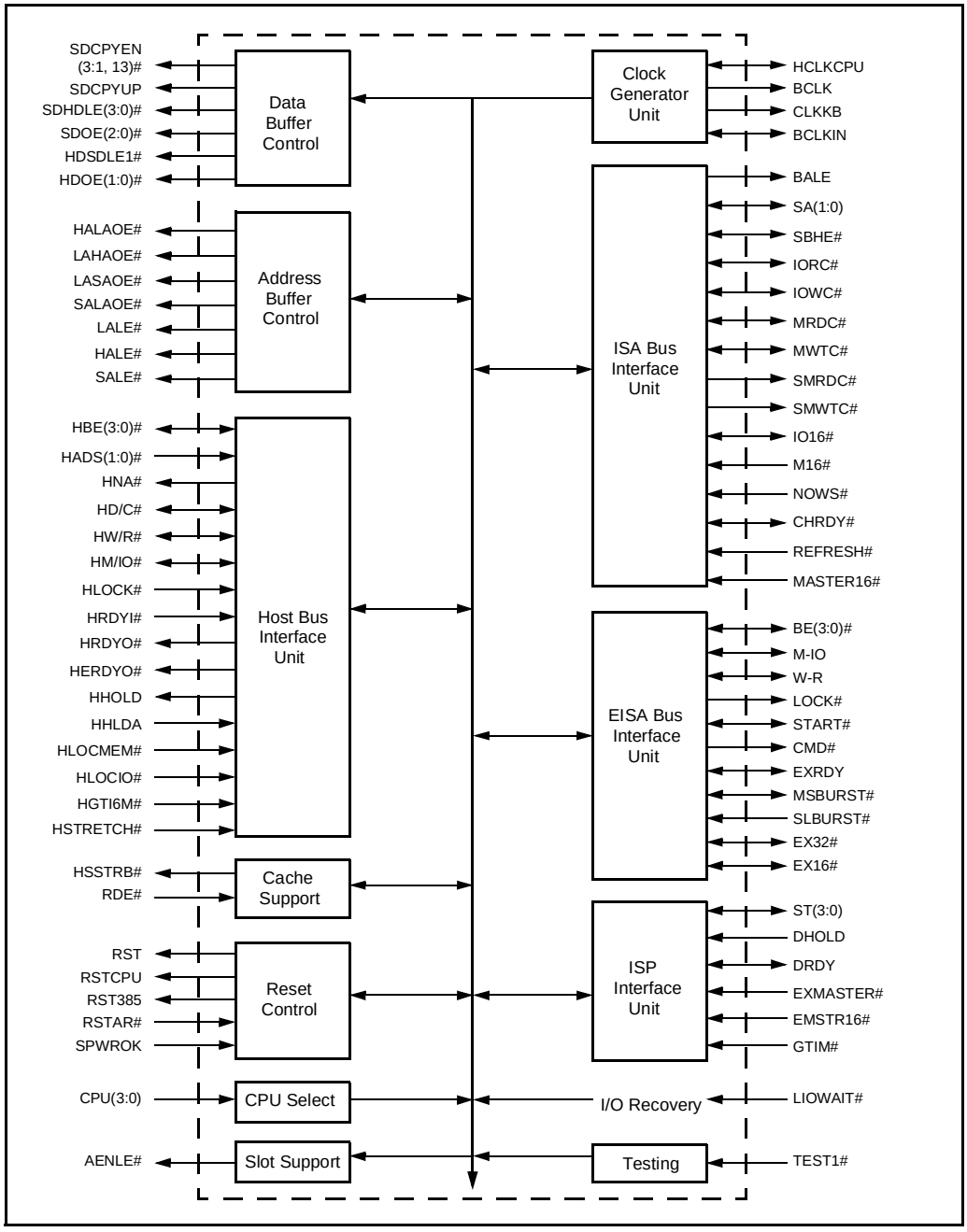


Figure 8-2. Block Diagram of EISA Bus Controller (EBC)

The integrated system peripheral (ISP), shown in Figure 8-3, is a multi-function support peripheral device that integrates many system functions that are normally distributed in several VLSI and LSI components. The ISP supports high-performance DMA operations with a programmable seven-channel controller. It has an arbiter that provides efficient bus sharing among multiple EISA masters and DMA devices. A programmable interrupt controller provides 15 levels of interrupts which can be edge-triggered or level-sensitive on a channel-by-channel basis. Non-maskable interrupts (NMI) are also supported. The ISP has five counters/timers that can provide system timer interrupts for a time of day, a diskette timeout, DRAM refresh requests and other system timing operations. The DMA controller is integrated in the ISP, and it has the necessary logic to set up, initiate, and complete DMA transfers. Various types of DMA transfers are provided for, including single transfer, block transfer, demand transfer, and cascade modes. Buffer chaining is also supported. The DMA controller provides the necessary timing signals to support a 33 Mbytes/sec transfer rate. Also supported are full 32-bit addressability on all functions and control signal support for data transfer between devices of different data widths. Each channel can operate independently in several modes.

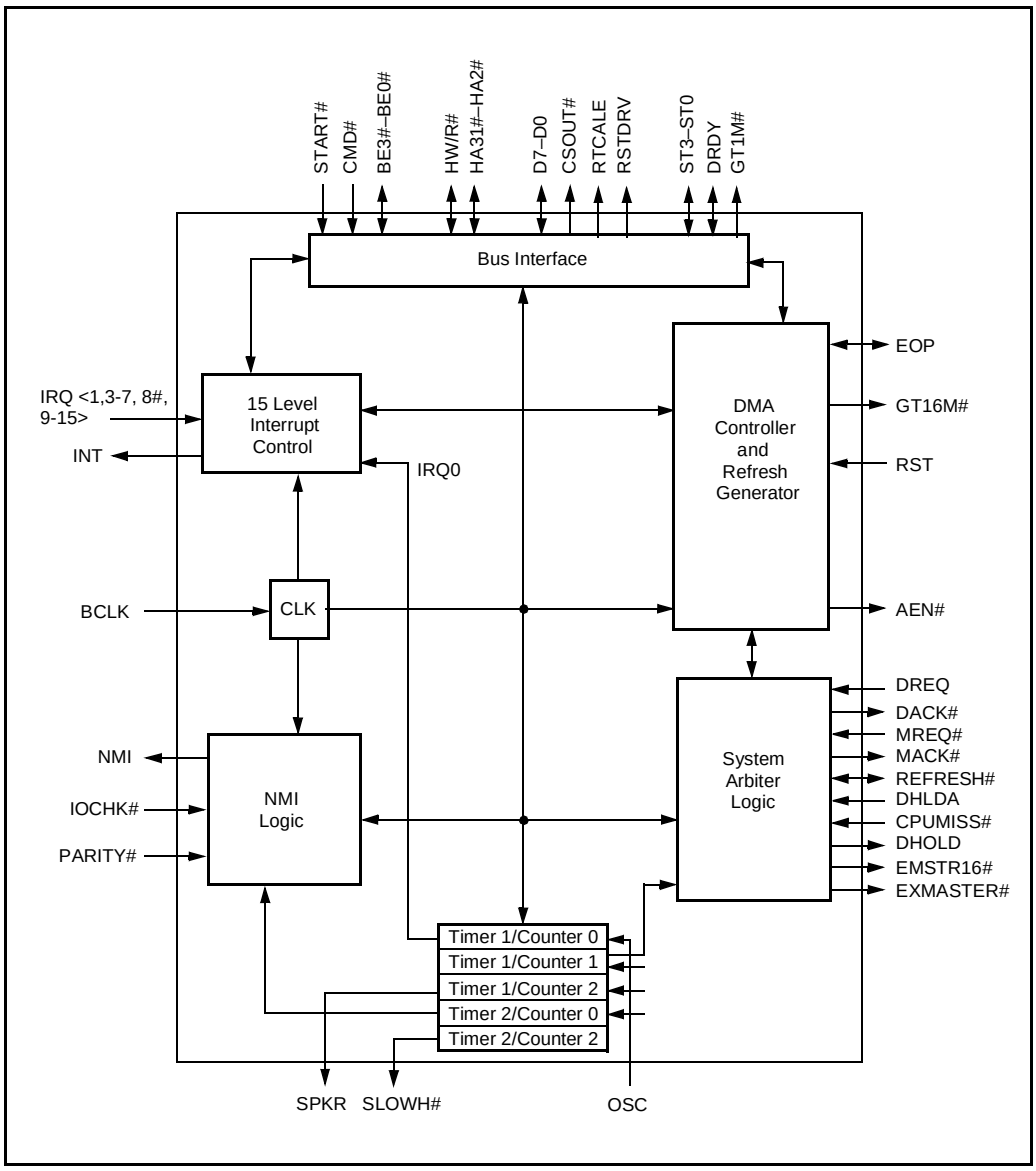


Figure 8-3. Block Diagram of Integrated System Peripheral (ISP)

The EISA bus buffers (EBB) are used to interconnect the host data and address buses to the EISA/ISA data and address bus. The EBB integrates multiple address or data latches and buffers that are typically used in EISA systems, and operates in various modes to support data and address interfaces. It has a 32-bit mode without parity and a 32-bit data mode with parity support

for each of the bytes. It also has an EISA address mode in which the addresses are interfaced with internal latched transceivers. Polarity on the address lines is compatible with the EISA specification, so that, for example, the most significant address byte is inverted.

8.3.3 EBC Host Bus Interface

The EBC resides between a fast host bus and the EISA bus and monitors cycles that are initiated on either bus. When the host initiates a bus master cycle and no response is received from the host slaves, the EBC forwards the cycle to the EISA bus. When an EISA bus master initiates a cycle, then it is always forwarded to the host bus. The EBC provides controls to the EBB device for the address and the data buffers between the two buses. The EBC also inserts delays between back-to-back I/O cycles between the host and the EISA bus.

8.3.3.1 Clock, Control and Status Interface

The host CPU clock (HCLKCPU) runs at the same frequency as the CPU clock. The EBC divides the HCLKCPU appropriately to generate the EISA BCLK signal.

Host address status (HADS1–HADS0) input signals indicate to the EBC that the addresses, byte enables, and cycle type information is valid on the host. These two signals are received by the EBC when there is a master on the host bus and are used to track the host bus cycles. If a host slave does not respond, and if an EISA/ISA slave or ISP is being addressed, then one or more cycles are generated on the EISA bus.

Host byte enables (HBE3#–HBE0#) are bidirectional signals that indicate valid bytes during an operation. They are inputs during host bus master cycles and are outputs during EISA bus master cycles as well as when the ISP is performing DMA or refresh cycles.

Host Byte High Enable (HBHE#) is a bidirectional signal. When asserted, it indicates that the upper byte of the 16-bit host bus is involved in the transfer. It is an input during host bus master cycles when an EISA/ISA slave is being accessed and an output during EISA master cycles or when the ISP is performing DMA or refresh cycles.

Host address bits 1,0 (HA1, HA0) are bidirectional signals that are used in the Intel386™ SX microprocessor systems.

Host next address (HNA#) is an output to the host CPU when it accesses an EISA/ISA slave. HNA# is asserted to indicate that the CPU can put a new address on the host bus.

Host data or control (HD/C#) is a bidirectional signal that differentiates between data and control cycles. It is an input to the EBC during host bus master cycles and is used to decode shutdown and interrupt acknowledge cycles. It is an output from the EBC during EISA/ISA master cycles and when the ISP performs DMA or refresh cycles. The signal is asserted to a high level when it is an output.

Host write or read (HW/R#) is a bidirectional signal that distinguishes between read and write cycles. It is an input to the EBC on host bus master is accessing an EISA/ISA slave, or when the ISP is performing DMA or refresh cycles. It is an output from the EBC on EISA/ISA master cycles.

Host memory or I/O (HMI/O#) is a bidirectional signal that differentiates between memory and I/O cycles. It is an input to the EBC when the host bus master cycles and is an output from the

EBC that is asserted high when the ISP is performing DMA or refresh cycles. It is also used to decode shutdown and interrupt acknowledge cycles.

Host bus ready input (HRDYI#) is an input signal that indicates the termination of a cycle on the host bus.

Host bus ready output (HRDYO#) is an output signal indicating that the EBC has completed a cycle. It is asserted when the host is addressing an EISA/ISA slave and the cycle has completed by appropriate inputs from EXRDY, CHRDY, NOWS#, and DRDY.

Host bus early ready output (HERDYO#) is an early version of the ready output from the EBC for situations in which HRDYO# does not provide enough setup time.

8.3.3.2 Host Local Memory and I/O Interface

Host bus local memory (HLOCMEM#) is an input signal which indicates that a host bus memory slave has decoded the current address as its own without preconditioning the HMI/O# signal. If this signal is asserted on host bus master memory cycles, it prevents an EISA bus cycle from initiating. This signal is used to determine if the memory is being accessed on the host bus during EISA/ISA master memory cycles or during DMA cycles.

Host bus local I/O (HLOCIO#) is an input signal which indicates that a host bus I/O slave has decoded the current address as its own without preconditioning the HMI/O# signal. If this signal is asserted on host bus master I/O cycles, it prevents EISA bus cycle from initiating. This signal is used to determine if the I/O device is being accessed on the host bus during EISA/ISA master I/O cycles.

Host bus stretch (HSTRETCH#) is an input used by host bus slaves during EISA/ISA master cycles to run zero (EISA) wait state cycles. This input can be used during DMA cycles and EISA/ISA bus master cycles to stretch the low period of the BCLK during the CMD# portion of the cycle. BCLK remains low until HSTRETCH# is sampled high. This produces a “stalling” effect of the EISA/ISA master without adding BCLK wait states. If the host memory subsystem is capable of performing EISA cycles without wait states, then the HSTRETCH# can be pulled high and no CPU clock-based logic is required for bus master or DMA cycles.

8.3.3.3 Host Bus Acquisition and Release

Host bus hold request (HHOLD) is an output signal which is asserted by the EBC to indicate a hold request to the host. This occurs when the ISP asserts DHOLD to indicate that an EISA/ISA bus master wants control or that a DMA device requires service.

Host hold acknowledge (HHLDA) is an input signal to the EBC from the bus master to indicate that it has relinquished control.

8.3.3.4 Lock, Snoop, and Address Greater than 16 Mbytes

Host bus lock (HLOCK#) is an input signal which is asserted by the host master when a locked bus cycle is in progress. If the addressed device is on the EISA bus, the signal is propagated to the LOCK# signal on the EISA bus.

Host snoop strobe (HSSTRB#) is an output signal which is driven by the EBC during any write cycle on the host bus. It is asserted during I/O to memory DMA cycles, EISA/ISA bus master write cycles to memory, and CPU write cycles to host memory.

8.3.4 EISA/ISA Bus Interface to the EBC

The EBC translates cycles from EISA masters that can be handled by ISA slaves and translates cycles between ISA masters that can be handled by EISA slaves. It also facilitates transfers between 32-bit and 16-bit EISA devices and 16-bit and 8-bit ISA devices.

Most of the EISA and ISA bus signals connect directly to the EBC or ISP without buffers. The direct connection assumes a worst case load of 300 pF and an IOL of 24 mA, with a worst case clock-to-output propagation delay of 30 ns. Only the AEN8 control signal lacks a direct connection to EISA/ISA. AEN*x* is a slot-specific signal that is decoded and asserted for a specific slot of a particular address. The ISP unit provides a global AEN# that is decoded with the LA bus address bit to generate the AEN*x* signals. This is shown in Table 8-1.

Table 8-1. AENx Decode Table

Address	AEN#	AENx					
		1	2	3	4	5	6
xxxx	1	1	1	1	1	1	1
00xx, 04xx, 08xx, 0Cxx	0	1	1	1	1	1	1
01xx--03xx, 05xx-07xx, 09xx-0Bxx, 0Dxx-0Fxx	0	0	0	0	0	0	0
10xx, 14xx, 18xx, 1Cxx	0	0	1	1	1	1	1
20xx, 24xx, 28xx, 2Cxx	0	1	0	1	1	1	1
30xx, 34xx, 38xx, 3Cxx	0	1	1	0	1	1	1
40xx, 44xx, 48xx, 4Cxx	0	1	1	1	0	1	1
50xx, 54xx, 58xx, 5Cxx	0	1	1	1	1	0	1
60xx, 64xx, 68xx, 6Cxx	0	1	1	1	1	1	0

The following is a brief functional description of the interface signals between the EISA/ ISA bus and the EBC.

8.3.4.1 EBC and EISA Bus Interface Signals

Byte enables (BE3#–BE0#) are bidirectional signals that indicate which bytes are involved in the current cycle. They are outputs during host bus master cycles and are inputs during ISA bus master cycles. They are inputs during EISA bus master cycles and when the ISP is performing DMA or refresh cycles.

Memory or I/O cycle (M/IO#) is an output signal that distinguishes between memory and I/O EISA cycles. It is an output during ISA master cycles and during host bus master-to-EISA/ISA slaves cycles. The signal floats during CPU, DMA, or EISA bus master cycles.

Write or read cycle (W/R#) is a bidirectional signal that is an input during EISA bus master cycles. It is an output of the EBC during host bus master to EISA/ISA slave cycles, and during ISA master cycles.

Start cycle (START#) is a bidirectional input signal to the EBC which starts cycles on the host bus. It is an output from the EBC on host master cycles when no responses are received from the host slaves. It is an output during ISP requests for DMA and refresh cycles. It is also an output during ISA master I/O cycles to 8-bit devices and when the EBC translates a 32-bit or 16-bit EISA bus master into cycles for an EISA/ISA slave with a smaller data bus size.

Command (CMD#) is an output which provides timing control within cycles. It is asserted simultaneously with the negation of START# and remains asserted until the cycle terminates. It is generated by the EBC during any EISA cycle.

Master burst (MSBURST#) is a bidirectional, open-collector output asserted by an EISA master to indicate that it is capable of supporting a burst operation in the next cycle. It is an input during EISA bus master cycles and an output during DMA cycles, when a burst mode DMA has been selected, and when memory is capable of supporting burst operations.

Slave burst (SLBURST#) is a bidirectional open-collector signal that is asserted by EISA slaves to indicate that they can accept burst cycles. It is an input when the ISP requests burst cycles and an output from the EBC when an EISA master is in control. It is asserted if the host memory is accessed and has asserted HSLBURST#.

EISA 32-bit device (EX32#) is a bidirectional, open-collector signal that is asserted by 32-bit EISA slaves to indicate 32-bit bus size. The signal is used to determine matched or unmatched data sizes on masters and slaves. Once the sizes are determined, the EBC assembles and disassembles data and performs multiple EISA or ISA cycles when necessary.

EISA 16-bit device (EX16#) is a bidirectional, open-collector signal that is asserted by 16-bit EISA slaves to indicate 16-bit bus size. The signal is used to determine matched or unmatched data sizes on masters and slaves. Once the sizes are determined, the EBC assembles and disassembles data and performs multiple EISA or ISA cycles when necessary.

EISA ready (EXRDY) is a bidirectional, open-collector signal which indicates that a slave is ready to terminate a cycle. It is an input to the EBC on host master cycles which access EISA or ISA slaves and is propagated to the host as the HRDY# signal. It is also an input for performing DMA or refresh cycles and is propagated as DRDY. It is an output from the EBC when an EISA master is accessing a host bus slave or the ISP. It is an output from the EBC during EISA master cycles to ISA slaves and is derived from CHRDY. It is an output for CPU cycles to ISA slaves for which an EISA cycle has been initiated.

Locked cycle (LOCK#) is an output signal which indicates to EISA slaves that the host CPU is executing a locked cycle. It is asserted by the EBC when the HLOCK# signal is asserted.

8.3.4.2 EBC and ISA Bus Interface Signals

Bus address latch enable (BALE) is an output from the EBC which indicates that a valid address is present on the latched address (LA) bus.

Bus clock (BCLK) is an output signal derived from the host CPU clock (HCLKCPU). The HCLKCPU can be divided by 3, 4, 5, 6 or 8 to give clock frequencies ranging between 8.0 and 8.33 MHz. The high or low time of BCLK can be stretched to synchronize it to four conditions.

16-bit master (MASTER1#16) is an input that indicates that a 16-bit EISA or ISA master has control of the EISA bus. It is sampled twice, at the beginning and at the end of START#. If negated at the first sampling time but asserted at the second sampling time, then it indicates to the EBC that a 32-bit EISA master is translating to 16 bits in order to perform burst operations.

16-bit memory (M16#) is a bidirectional open-collector signal that indicates that the ISA memory is capable of performing 16-bit transfers. It is an output during ISA master cycles in which a host slave or EISA memory slave is accessed. It is an input during host bus master cycles in which the EISA/ISA bus is accessed. It is an input during EISA master cycles.

Standard memory read control strobe (SMRDC#) is an output signal that commands the ISA memory to place data on the data bus. It is asserted during CPU, DMA or EISA/ISA master read cycles to 16-bit or 8-bit ISA memory slaves when the address range is less than one megabyte. It behaves like the MRDC# signal.

Standard memory write control strobe (SMWTC#) is an output signal that commands the ISA memory slave to accept data from the data bus. It is asserted during CPU, DMA or EISA/ISA master write cycles to 16-bit or 8-bit ISA memory slaves when the address range is less than one megabyte.

Channel ready (CHRDY) is a bidirectional, open-collector signal which is used by the ISA slaves to insert wait states. It is an output during ISA master cycles and accesses host bus slaves or EISA slaves.

No wait states (NOWS#) is an input asserted by ISA slaves to request compressed standard wait states, and by EISA bus slaves to request compressed or 1.5 BCLK cycles.

System address bits 1 and 0 (SA1, SA0) are the least significant bits of the latched EISA address bus. They are inputs during ISA bus master cycles and generate appropriate EISA bus or host bus controls. They are outputs during host bus master cycles and access EISA/ISA slaves. Further, they are outputs during EISA master cycles to ISA slaves and during DMA accesses to ISA memory.

System byte high enable (SBHE#) is a bidirectional signal that indicates the validity of the high byte on the EISA bus. It is an input during ISA bus master cycles and an output during host accesses to EISA/ISA slave. Further, it is an output during EISA master cycles to ISA slaves and during DMA accesses to ISA memory.

Refresh (REFRESH#) is an input which indicates that the ISP is performing a refresh cycle. During refresh cycles the EBC generates the MRDC#, CMD# and other host bus signals to refresh the entire system's DRAM memory.

8.3.5 EBC and ISP Interface

The EBC and ISP have a tightly coupled interface, and they interact with the host bus requests, DMA status, EISA bus master size, and other control and status signals described below:

- ISP hold request (DHOLD) is an input from the ISP which is used to request the host bus on behalf of ISA/EISA masters or when a DMA device requests service. DHOLD is used to generate HHOLD.
- ISP ready (DRDY) is a bidirectional signal. It is an input to the EBC when the ISP is in the slave mode. It is an output from the EBC during DMA cycles and refresh cycles.
- Greater than one megabyte (GT1M#) is an input to the EBC that indicates that the current address is above the 00000000h to 000FFFFFh range. If it is not asserted during a host bus master cycle or an EISA/ISA bus master cycle, or during DMA cycles on accessing ISA memory slave, then the EBC generates SMRDC# or SMWTC# signals. The ISP generates the GT1M# signals for all cycles including DMA and non-DMA cycles.
- Host address greater than 16 megabytes (HGT16M#) is an input signal which indicates that the address of the current cycle is greater than 00FFFFFFh. It is driven on DMA cycles, based on the address from the ISP. This signal is used by the EBC during DMA cycles to determine whether to generate the ISA memory command signals, MRDC# and MWTC#. MRDC# and MWTC# are generated during DMA cycles but are inhibited when HGT16M# is active.
- DMA status (ST3–ST0) are bidirectional signals. They are inputs to the EBC during DMA and refresh cycles. They indicate the timing that has been programmed for the current cycle and the size of the I/O device involved in the DMA transfer. They are outputs form the EBC when the ISP is not a bus master. The four signals function as address strobe for the ISP, memory or I/O cycle indicator, the interrupt acknowledge cycle indicator, and the EISA bus master cycle indicator, respectively.
- EISA master (EXMASTER#) is an input signal to the EBC, which indicates that a 16-bit or 32-bit EISA master has control of the EISA bus. It is used with the MASTER16# signal to differentiate between 32-bit EISA masters, 16-bit EISA masters, and 16-bit ISA masters.
- Early indication of 16-bit ISA master (EMSTR16#) is an input signal to the EBC which indicates that a 16-bit master is in control is or about to assume control of the EISA bus.

8.3.6 EBC and EBB Data and Address Buffer Controls

The host data and address buses are connected to the EISA/ISA data and address buses using the EISA bus buffer (EBB). The EBB has internal latches and the outputs can be controlled in either direction. Data from the EISA bus can flow to the host bus on port B and on an individual byte basis on port A. Data can be stored using the provided control signal. Data can also flow from the host bus to the EISA/ISA bus.

The EBB controls byte assembly. Bytes can be transferred as shown in Figure 8-4. The EBC provides signals used to copy the individual bytes. For multiple cycle operations the octal registered transceivers are used to temporarily store the data until an entire word or doubleword is assembled. Following assembly, the word or doubleword is transferred to the destination. Byte assembly logic is used for all bus size mismatches and non-aligned address translations between the host bus, a 32-bit or a 16-bit EISA bus and a 16-bit ISA bus. The EBC generates controls to steer the data buses and to latch the address and data.

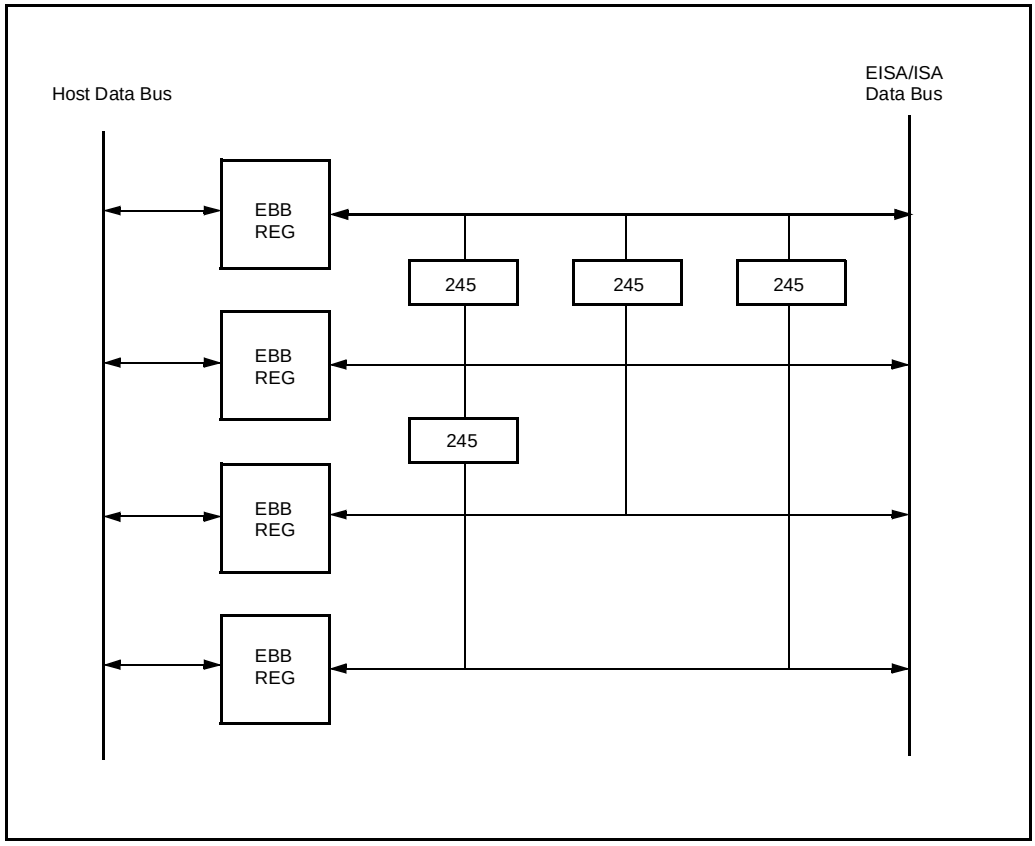


Figure 8-4. EBB Byte Transfer

Copy enable between bytes SDCPYEN(03#–01#, 13#) are output controls that enable the byte copy transceivers between the EISA bus bytes 0, 1, 2, and 3. Data bits 7–0 can be copied between data bits 15-8, 23–16 and 31–24. Data bits 15–8 can be copied between data bits 31–24.

Copy up (SDCPYUP) is an output that controls the direction of the byte copy transceivers to copy the lower bytes to the higher bytes and vice versa.

System (EISA) to host data latch enables (SDHDLE3#–SDHDLE0#) are outputs that control the latching of data from the EISA bus to the host bus.

System (EISA) data output enable (SDOE2#–SDOE0#) are output enables to data buffers on the EISA bus.

Host data to system (EISA) data latch enables (HDSDLE1#–HDSDLE0#) are outputs that control the latching of data from the host data bus to the EISA data bus.

Host data output enables (HDOE1#, HDOE0#) are output enables to the host data bus buffers.

Host address bus to EISA LA bus output enable (HALAOE#) is an output signal which enables the output of the address buffers for host address bus bits 31–2 on to the EISA LA bus bits 31–2. The signal is asserted during CPU, DMA and refresh cycles along with HALAOE#.

Host address latch enable (HALE#) is an output signal which latches the LA address bus on to the host address bus. The latch closes on the trailing edge, and the host address bus is held until the slave terminates the cycle.

EISA LA to EISA SA output enable (LASAOE#) is an output signal which enables the EISA LA bus bits 19-2 on the EISA SA bus. It is asserted during CPU, EISA bus master, DMA and refresh cycles.

EISA LA to host address output enable (LAHAOE#) is an output signal which enables address buffers from the EISA LA bits to the host address bus. It is asserted during EISA/ISA bus master cycles.

LA latch enable (LALE#) is an output signal which latches the host address bus on to the LA address bus. It is useful when the CPU operates in burst mode or when additional address pipelining is required on the host bus.

EISA SA to EISA LA output enable (SALAOE#) can be used to the output of the address buffers from the EISA SA bus bits 16-2 on to the EISA LA bus 16–2. It is asserted during ISA bus master cycles.

SA latch enable (SALE#) is an output signal which latches the LA address bus on to the SA address bus. It can be asserted during EISA master, CPU, regular DMA, and DMA burst cycles.

8.3.6.1 Functions of the ISP

The ISP provides system arbitration, DMA control, interrupt control, and counting by using interval timer/counters.

The system arbiter on the ISP evaluates requests from several sources including DMA channels, EISA devices, refresh requesters, and the host CPU: DREQ is generated by 8-, 16-, or 32-bit devices that require DMA service; MREQ# is generated by 16-bit or 32-bit EISA devices; and CPUMISS# is generated by the host CPU. Refresh requests are generated internally using the timers. Request priority is assigned on different levels, and at each level, devices are given rotating priority. Examples of priorities and assignments are shown in the ISP datasheet. The arbiter determines which requester receives the bus from EISA masters, DMA slaves, refresh requesters and the host CPU.

The on-chip DMA controller is functionally equivalent to two 8237 DMA controllers. Seven independent channels can be programmed. Data widths of 8-, 16-, and 32-bits are supported, as are ISA-compatible, ISA/EISA compatible, type A/type B modes, and EISA type C mode. Single, block, demand, or cascade transfer modes are supported. The DMA controller provides refresh address generation, and buffer chaining.

The ISP provides an ISA-compatible interrupt controller and the functionality of two 8259 interrupt controllers. The ISP can handle fourteen external interrupts and two internal interrupts. The internal interrupts are for internal functions only and not available externally. A non-maskable interrupt can be generated by hardware or software.

The ISP has five interval timers. The counter timers are addressed as if they are contained in two separate 8254 timers.

The ISP operates as a slave device or as a master device. In slave mode, the ISP monitors the address lines and decodes all bus cycles. Here, an EISA master or host bus master can read or write to any of the ISP registers. 16-bit ISA masters can read and write to any of the non-DMA registers and to some of the non-8237/PC AT compatible registers. In the master mode, the ISP becomes the bus master and can perform DMA or refresh cycles.

8.3.6.2 ISP-to-Host Interface

Host addresses HA31–HA2 are 3-stateable address signals which connect to the host bus. HA31–HA20 and HA15–HA2 are bidirectional, whereas HA19–16 are outputs. In master mode all of the address lines are outputs. In slave mode HA15–2 and HA31–2 are inputs. Upon reset these lines are 3-stated and configured as inputs.

Byte enables (BE3#–BE0#) are 3-stateable EISA bus byte enables. In slave mode the BE2#–BE0# are inputs and are used to access ISP internal registers. In master mode BE3#–BE0# are outputs. BE3# is always an output.

Host write/read (HW/R#) is a bidirectional signal which indicates a read or write cycle. It is an input during slave mode and an output during master mode. It is sent to the EBC which propagates the appropriate read/write signals to the EISA bus. Upon reset this signal is 3-stated and configured as an input.

Slow down host CPU (SLOWH#) is an output from CPU slowdown timer 2, which is used to slow down the host CPU.

CPU cache miss (CPUMISS#) is an input signal from the host CPU, or the cache controller subsystem which indicates that a host bus cycle is pending and must contend for the next bus arbitration.

Hold acknowledge (DHLDA) is an input signal which indicates that the system has granted ISP to the host bus.

Interrupt (INT) is an output signal which indicates that an interrupt request is pending and must be serviced. Once asserted, it remains asserted until it receives the first INTA# pulse via the ST2# signal. Upon reset, the state of INT is undefined.

Non-maskable interrupt (NMI) is an output used to force a non-maskable interrupt to the host CPU. Once asserted, it remains asserted until the CPU reads to one of the NMI registers. Upon reset this signal is low.

Parity (Parity#) is an input from the system board which indicates a main memory parity error.

8.3.7 ISP-to-EISA Interface

DMA requests (DMA 7–5, 3–0) are inputs to the ISP, which indicate requests for control of the system bus. They are generated externally by DMA subsystems or by 16-bit masters.

DMA acknowledge (DACK 7–5, 3–0) are outputs from the ISP which indicate that the bus has been granted to the respective requester.

Master requests (MREQ5–MREQ0) are slot-specific inputs to the ISP which are used by EISA masters to request bus access.

Master acknowledges (MACK#5–MACK#0) are outputs from the ISP that acknowledge that the bus has been granted to a requesting EISA master.

Refresh (REFRESH#) is a bidirectional signal. It is an output during refresh cycles and should be used to refresh the entire system memory at once. It is an output only when the ISP DMA is a bus master, while an internal request for a refresh cycle is generated in the ISP. The REFRESH# is an input when an expansion bus adapter acts as a 16-bit ISA bus master.

Start of cycle (START#) is an input which connects to the EISA START# signal. Command (CMD#) is an input that connects directly to the EISA CMD# signal. It is used to 3-state the data buffers following a read cycle.

End of process (EOP) is a bidirectional signal which is directly connected to the TC signal of the ISA/EISA bus. It is used in three modes: as an input in one mode, it is used by DMA slaves to stop DMA transfers; as an input from a slave in a second mode, it is used as a terminal count; as an output in a third mode, it indicates that a chain buffer has expired and that a new chain buffer must be programmed. Interrupt request (IRQ 15–3,1) are interrupt inputs to the ISP.

Byte enables (BE3#–BE0#) are the EISA bus byte enables. BE3-BE1 are bidirectional, and BE0 is output only. In master mode, the ISP drives these lines. In slave mode the BE3-BE1 are inputs to the ISP and are used to access the internal registers. BE0 is remains an output in slave mode.

Ready signal (DRDY) is a bidirectional signal. In slave mode, it is an output which is driven when the ISP detects a slave write to its registers. In master mode, it is an input which indicates to the DMA controller that the current cycle has completed and that the DMA controller must pipeline addresses for DMA burst transfers.

Data (D7-D0) are bidirectional signals that function as outputs when the ISP is in the slave mode. These signals are not used in the master mode. The pins are in output mode when CSOUT# is asserted during an I/O read or interrupt acknowledge cycle.

Slave mode selected (CSOUT#) is an output from the ISP which indicates that it is accessed in the slave mode.

Address enable (AEN#) is an output signal, which indicates whether the host, EISA, or ISA is the current bus master.

I/O check bus error (IOCHK#) is an input from the ISA bus and is used for parity error checks and for other high priority interrupts. It can be programmed to cause a non-maskable interrupt.

8.4 PCI BUS: SYSTEM DESIGN EXAMPLE

8.4.1 Introduction to PCI Architecture

The PCI (Peripheral Component Interconnect) bus is the descendant of the VESA VL bus and is a widely-implemented embedded system solution. The PCI standard was defined by Intel to encourage designers to adopt a common system bus architecture that would accommodate future computing needs. Because the VESA VL standard does not take a sufficient long-term approach, the PCI standard does not support VESA VL. The PCI standard provides the following features.

- 32-bit or 64-bit address buses to accommodate 32-bit and 64-bit CPUs and bus masters
- 32-bit or 64-bit data transfers
- 33 MHz and 66 MHz PCI bus operation speeds
- 132 Mbytes/sec transfer rate for 33 MHz/32-bit implementation, 264 Mbytes/sec for 66 MHz/32-bit or 33 MHz/64-bit implementations, and 524 Mbytes/s for 66 MHz/64-bit implementation.
- All read and write transfers over the PCI bus are burst transfers.

The PCI bus handles 32-bit wide address and data buses in the 32-bit implementation. The PCI specification also provides for 64-bit wide address and data buses (address and data buses by PCI standards are muxed).

All actions on the PCI bus are synchronized using the PCICLK signal. Revision 1.0 of the specification requires that all devices support 16-33 MHz operation. Revision 2.1 requires that all devices support operation down to 0 MHz. Revision 2.2 adds support for 66 MHz implementation, requiring that all devices operate from 0 MHz-66 MHz.

PCI-based computers support a Bus Initiator/Target architecture for intelligent peripherals. All transactions on the PCI bus are in burst mode. The initiator starts by driving an address on the PCI Address/Data bus and by driving the command type onto the PCI Command/Byte Enable bus. Each PCI target latches the address and decodes the start address and command type to determine if it is the addressed device. The device also determines the type of transaction in progress. Upon completion of the address phase, the PCI Address/Data bus is used to transfer data. The target must latch the start address and increment the address to point to the next address for each subsequent data transfer.

PCI systems provide a centralized arbiter that allows efficient bus sharing between multiple PCI bus initiators. Although the PCI specification does not specify the exact method of arbitration (such as fixed and rotational), the 2.1 specification states that the arbiter is required to implement a fairness algorithm to avoid deadlocks. *Fairness* means that each potential bus master must be granted access to the bus independent of other requests. However, this does not mean that all agents are required to have equal access to the bus.

8.4.2 Example PCI System Design

This section describes an example of the PCI architecture implemented in an embedded Intel486 processor system.

This example PCI chipset supports all Intel486 processors and upgrades, including write-back internal (L1) cache and Intel SMM power management. PCI local bus IDE is incorporated for higher performance IDE. A block diagram of a system that uses this type of PCI chip set is shown in Figure 8-5.

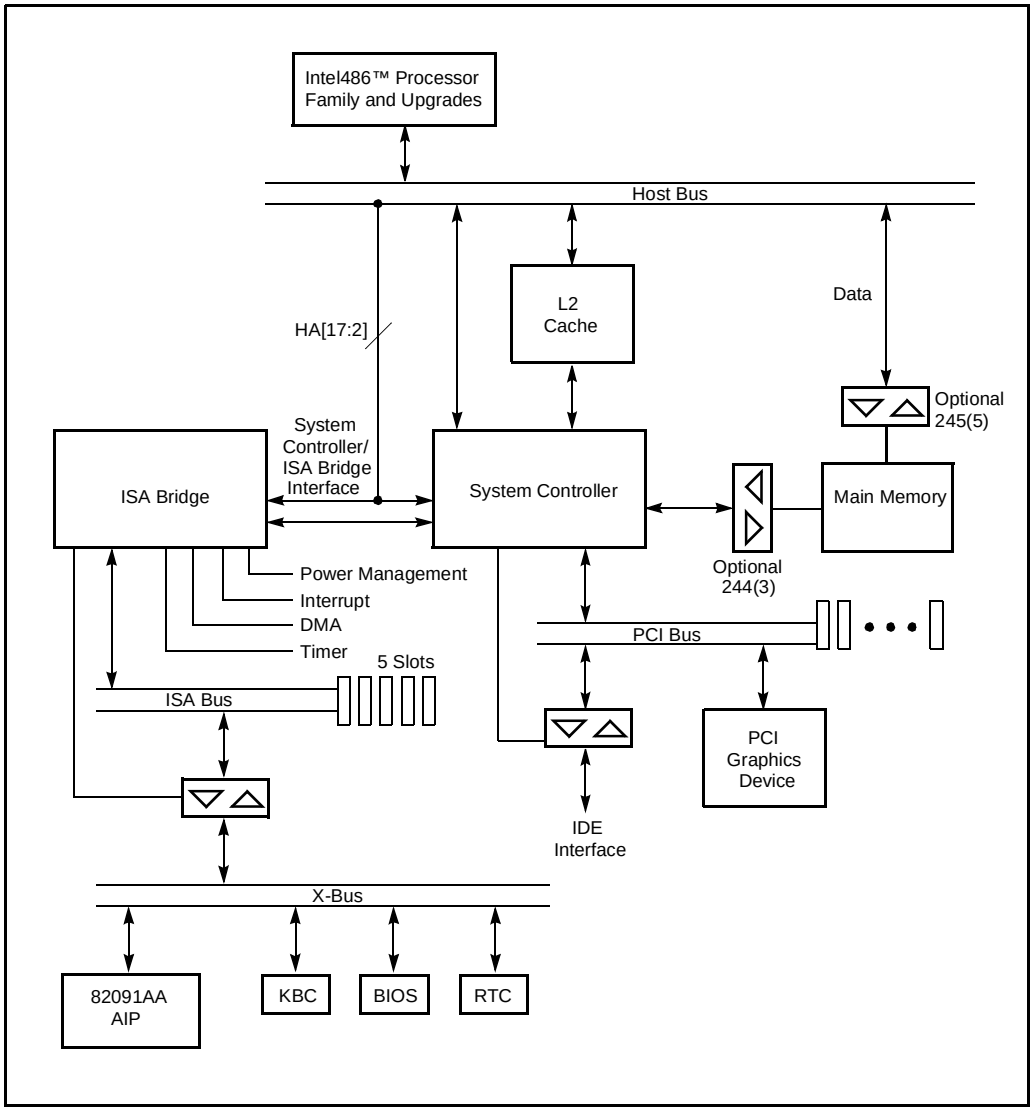


Figure 8-5. Example System Block Diagram

The chipset consists of two components: the system controller and the ISA bridge. The system controller integrates the second-level (L2) cache controller and the DRAM controller. The cache controller supports both write-through and write-back cache policies and cache sizes from 64 Kbytes to 512 Kbytes in an interleaved or non-interleaved configuration. The DRAM controller interfaces main memory to the Host bus and the PCI bus. The system controller supports a two-way interleaved DRAM organization for optimum performance. Up to ten single-sided SIMMs or four double-sided and two single-sided SIMMs provide a maximum of 128 Mbytes of main memory. The system controller provides memory write posting to PCI for enhanced CPU-to-PCI memory write performance. In addition, the system controller provides a high performance PCI local bus IDE interface. Figure 8-6 shows a block diagram of the system controller component of the PCI chip set.

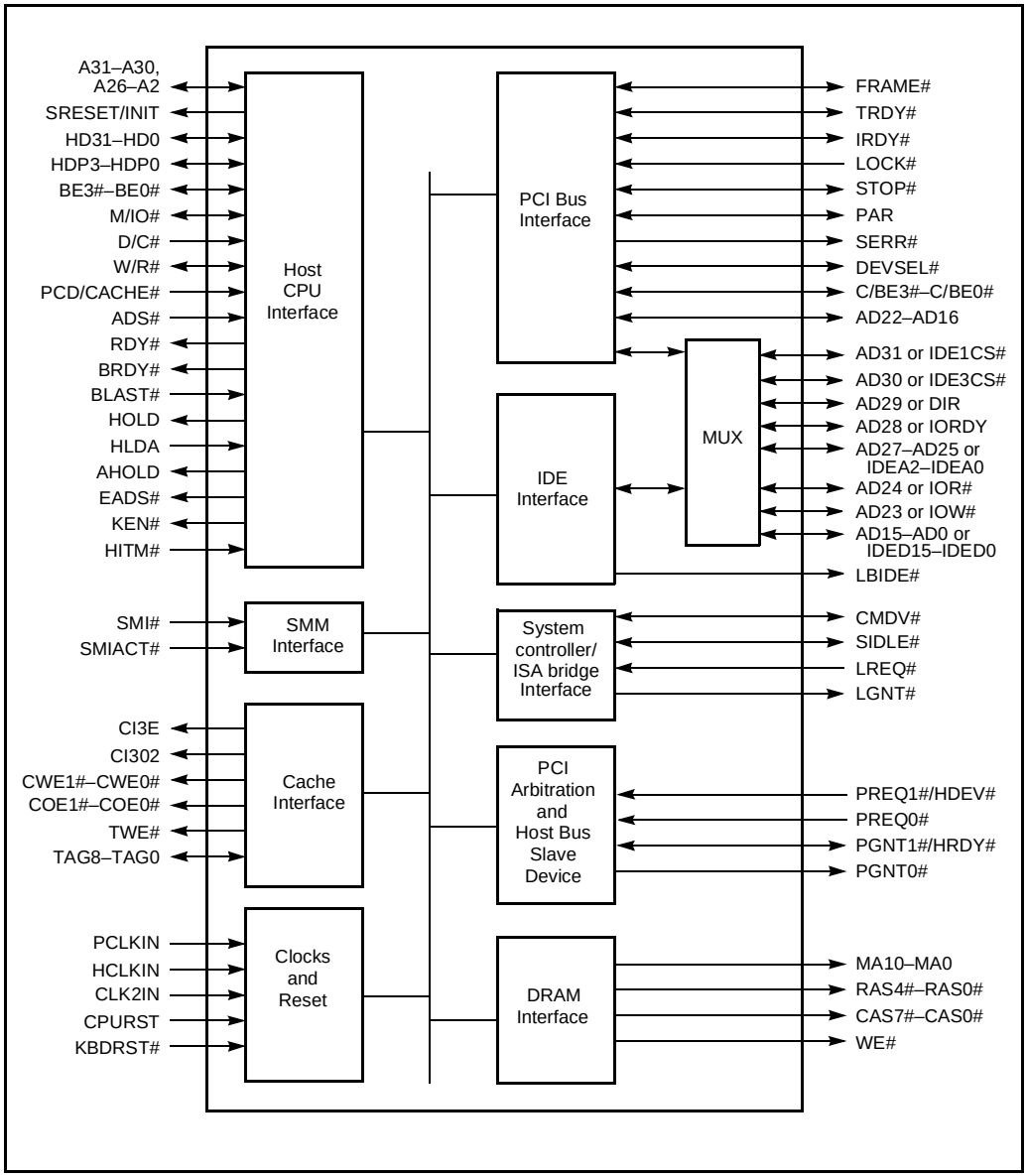


Figure 8-6. System Controller Block Diagram

The ISA bridge links the ISA bus and Host bus, and integrates the common I/O functions found in today's ISA-based systems: a seven channel DMA controller, two 82C59 interrupt controllers, an 8254 timer/counter, Intel SMM power management support, and control logic for NMI generation. The ISA bridge also provides the decode for the external BIOS, real time clock, and key-

board controller. Edge/level interrupts and interrupt steering are supported for PCI plug-and-play compatibility. The ISA bridge integrates the ISA address and data path, reducing TTL and system cost. In addition, the integration of system clock generation logic eliminates the need for external host and PCI clock drivers.

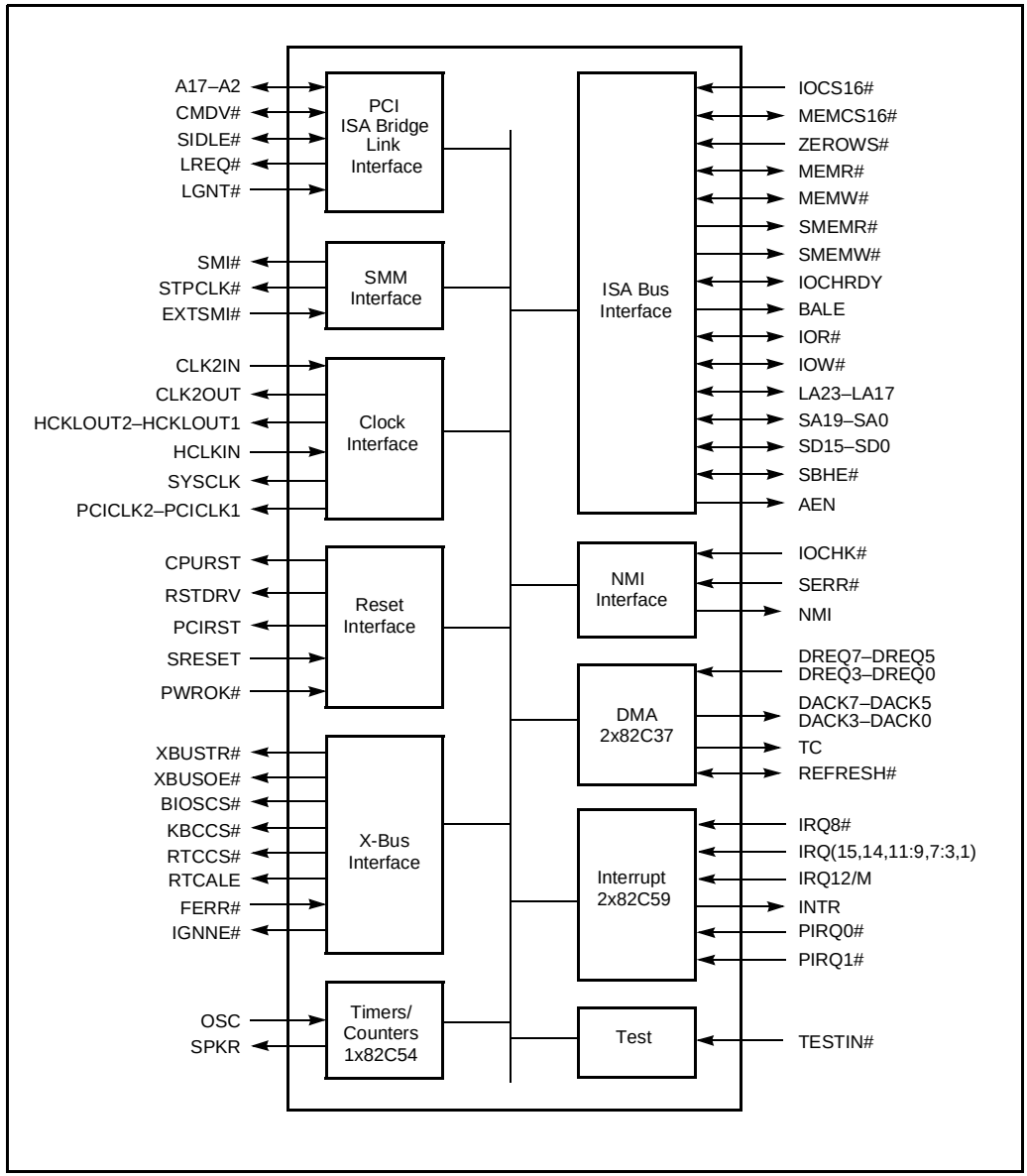


Figure 8-7. ISA Bridge Block Diagram

This PCI chip set interfaces to three system buses: the CPU, PCI, and the ISA buses. The system controller provides positive decode for certain I/O and memory space accesses on the CPU and PCI buses. These decodes include accesses to the PCI Local bus IDE (CPU only), main memory (CPU, ISA, and PCI), and the system controller's I/O Control registers (CPU only). In addition, the system controller subtractively decodes certain CPU/PCI cycles.

The ISA bridge provides the positive decode for certain ISA I/O and memory space accesses. These decodes include accesses to the ISA-compatible registers (for ISA master and DMA initiated cycles), main memory (for ISA and DMA initiated cycles), BIOS, X-Bus, and system events for SMM support. Note that DMA devices and ISA masters cannot access the PCI or CPU buses.

This PCI chip set provides bus arbitration on the Host bus, the PCI bus, and the PCI/ISA interface (to the ISA bus). A device that is the master on any bus is the master of the entire system. (i.e., concurrency of more than one active master is not supported).

When there are no active requests, the CPU owns the system. The system arbitration rotates between the PCI bus, CPU bus, and Link Interface bus (on behalf of DMA and ISA Master devices), with the CPU permitted access every other transition.

8.4.3 Host CPU Interface

This PCI chip set provides a host interface to all of the Intel486 family processors and upgrades.

8.4.3.1 Host Bus Slave Device

The PCI chip set can be configured (via the HOST Device Control register) to support an Intel486 Host bus slave device (for example, a graphics device). Two special signals (HDEV# and HRDY#) as defined by the VL bus specification are used in the interface to the Host bus slave. The system controller can be configured to monitor HDEV# for all memory and I/O ranges that are not positively decoded by the system controller. The system controller can be configured to monitor HRDY# and assert the RDY# input to the CPU, based on HRDY#. The host device may include an I/O range, a memory range, or both I/O and memory ranges. In all cases, these ranges must not be programmed (positively decoded) by the system controller. The host device's memory ranges are non-cacheable.

8.4.3.2 L1 Cache Support

The PCI chip set provides signals that support the CPU's L1 cache. For the S-Series CPUs, these signals are PCD, KEN#, and EADS#. For the D-Series and P24T CPUs, the signals are the KEN#, EADS#, CACHE#, and HITM#. The P24T and the D-Series CPUs include certain signals that are not connected to the PCI chip set. These signals are fixed to 1 or 0, depending on the system configuration.

8.4.3.3 Control and Status Interface

Soft Reset/Initialize, SRESET/INIT, is the soft reset output of the PSSC and should be connected to the SRESET or INIT input to the CPU, depending on the CPU type.

Host Address, A31–A30, A26–A2, are used as inputs to the system controller for CPU-driven cycles. A31–A30, A26–A4 are outputs during Snoop cycles. Note that A29–A27 are not driven by the system controller. These signal lines must be externally driven low either by weak pull-down

resistors or by driving these lines low when HLDA is asserted. A17–A2 are also used for system controller/ISA bridge link interface transfers. These signals are 3-stated after a hard reset.

The Byte Enable signals BE3#–BE0# indicate active bytes during read and write cycles. These signals are 3-stated after a hard reset.

Host Data HD3–HD0 are connected to the host CPU data bus. These signals are inputs after a hard reset.

The Host Data Parity signals, HPD3–HPD0, are bi-directional parity signals for the host data bus. These signals provide parity to the system controller during main memory read cycles. The system controller sends parity information to main memory during non-CPU main memory write cycles. These signals are 3-stated after a hard reset.

Bus Cycle Definition, M/IO#, D/C# and W/R#, are signals that define the Host bus cycle. M/IO# is a bi-directional signal that distinguishes between memory and I/O cycles. D/C# is a bi-directional signal that differentiates between data and control cycles. W/R# is a bi-directional signal that distinguishes between read and write cycles. Note that special cycles are identified by BE3#–BE0# and A4–A2. These signals are 3-stated after a hard reset.

Page Cache Disable/Cache, PCD/CACHE#, is a multiplexed signal pin with two functions, depending on the type of CPU used. The PCD cache input signal, when asserted, indicates the current cycle cannot be cached in the L2 cache during line fill operation. When PCD is asserted, the line is not cached in L1 or L2. The CACHE# signal is active along with the first ADS# until the first RDY# or BRDY#. For line fills, the functionality of the CACHE# signal is identical to that of the PCD signal. During write-back cycles, CACHE# is always asserted at the beginning of the line write-back. The beginning of a write-back cycle is uniquely identified by active ADS#, W/R# and CACHE#. Beginning of the snoop write-back is identified by the ADS#, W/R#, CACHE# and HITM# being active.

The Address Status, ADS#, input indicates that the bus cycle definition signals (M/IO#, D/C#, W/R#), BE3#–BE0#, and A31–A30, A26–A2 are available on their corresponding pins.

Ready, RDY#, indicates that the current non-burst bus cycle is complete. This signal is deasserted after a hard reset.

Burst Ready, BRDY#, performs the same function during a burst cycle that RDY# performs during a non-burst cycle. This signal is deasserted after hard reset.

Burst Last, BLAST#, indicates the end of a burst access for CPU-initiated cycles.

The system controller asserts HOLD to the CPU to request ownership of the Host bus. This signal is deasserted after a hard reset.

Hold Acknowledge, HLDA, must be asserted by the CPU for the system controller to grant a new master on the PCI or ISA buses. When HLDA is deasserted, the CPU is the Host bus master and the system controller is the PCI bus master. When HLDA is deasserted, the system controller is also the master on the system controller/ISA bridge link interface.

Address Hold, AHOLD, output signal forces the CPU to float its address bus in the next clock. The system controller asserts this signal in preparation to perform a system controller/ISA bridge interface transfer, when SRESET needs to be asserted, or upon Deturbo logic requests. This signal is deasserted after a hard reset.

External Address, EADS#, when asserted, indicates that an external address has been driven onto the CPU address lines. This address is used to perform an internal cache snoop cycle. This signal is deasserted after a hard reset.

Cache Enable, KEN#, when asserted, indicates whether the current cycle is cacheable in the CPU L1 cache. This signal is deasserted after a hard reset.

Hit Modified, HITM#, when asserted, indicates that a hit to a modified data cache has occurred during the snoop cycle. A pull-up is used to keep HITM# deasserted when not used.

The system controller has a standard master/slave PCI bus interface. As a PCI device, the system controller can be either a master initiating a PCI bus operation or a target responding to a PCI bus operation. The system controller is a PCI bus master for Host-to-PCI accesses and a target for PCI-to-main memory accesses (or accesses that are forwarded to the ISA bus). The Host can read or write configuration spaces, PCI memory space, and PCI I/O space.

8.4.3.4 PCI Bus Cycles Support

When the host initiates a bus cycle to a PCI device, the system controller becomes a PCI bus master and translates the CPU cycle into the appropriate PCI bus cycle. Post buffers permit the CPU to complete Host-to-PCI writes in zero wait-states.

When a PCI bus master initiates a main memory access, the system controller becomes the target of the PCI bus cycle and responds to the read/write access. As a PCI master, the system controller generates address parity for read and write cycles, and data parity for write cycles. As a target, the system controller generates data parity for read cycles. During PCI-to-main memory accesses, the system controller automatically performs cache snoop operations on the Host bus, if needed, to maintain data consistency.

PCI bus commands indicate to the target the type of transaction desired by the master. These commands are presented on the C/BE3#–C/BE0# signals during the address phase of a transfer. Table 8-2 summarizes the system controller's support of the PCI bus commands.

Table 8-2. Supported PCI Bus Commands

C/BE[3:0]	Command Type	Supported As Target	Supported As Master
0000	Interrupt Acknowledge	No	No
0001	Special Cycle	No	No
0010	I/O Read	Yes	Yes
0011	I/O Write	Yes	Yes
0100	Reserved	D	D
0101	Reserved	D	D
0110	Memory Read	Yes	Yes
0111	Memory Write	Yes	Yes
1000	Reserved	D	D
1001	Reserved	D	D
1010	Configuration Read	No	Yes
1011	Configuration Write	No	Yes
1100	Memory Read Multiple	Yes(1)	No
1101	Dual Address Cycle	No	No
1110	Memory Read Line	Yes(1)	No
1111	Memory Write and Invalidate	Yes(2)	No

NOTES:
1. As a target, the system controller treats this command as a memory read command.
2. As a target, the system controller treats this command as a memory write command.

8.4.3.5 Host to PCI Cycles

Host bus accesses to PCI bus are always in the Host bus address range, as defined by A31–A30, A26–A2 and the four BE lines. The PCI address lines are driven during the address phase. AD29–AD27 lines are driven to the value of A30, during Host accesses to PCI.

The system controller has the ability to burst up to 32 back-to-back CPU memory writes on the PCI bus. This function in controlled by the PCICON register. The system controller is capable of merging 8/16-bit graphic write cycles to the same dword address into the same posted write buffer location (controlled by the PCICON register). The merged data is then driven as a single dword cycle on the PCI bus. Byte merging is performed in the compatible VGA range only.

8.4.3.6 Exclusive Cycles

The system controller, as a PCI master, never performs LOCKed cycles. The CPU does not return active HLDA while it is performing a LOCKed sequence. Also, the CPU is the only active master, as long as HLDA is inactive. Thus, the system controller does not need to drive LOCK to

guarantee the CPU atomic LOCK sequence. Note that this PCI chip set supports a bus locking mechanism (i.e., when a PCI master performs locked accesses, the arbitration is not changed until the locked sequence is completed).

The system controller does not check parity or generate SERR# based on the PCI parity. The system controller only generates SERR# (if enabled via the PCICOM register), when a main memory read results in a parity error. When main memory parity error is detected, the system controller activates SERR#, if enabled, for a single PCICLK.

When a main memory parity error is detected and SERR# generation is enabled, the MMPERR bit in the DS register is set to 1. When SERR# is activated, the SERRS bit in the DS register is set to 1.

8.4.3.7 Status and Control Interface

Address/Data, AD31–AD0, are connected to the PCI multiplexed address/data bus. These signals are also multiplexed with the IDE interface. These signals are driven high after a hard reset.

Bus Command/Byte Enable, C/BE3#–C/BE0#, are multiplexed on the same pins. These signals are driven high after a hard reset.

FRAME# is an output when the system controller is a master on the PCI bus. FRAME# indicates that a PCI cycle has started. This signal is 3-stated after a hard reset.

Target Ready, TRDY#, is an input when system controller is a master on the PCI bus. TRDY# is an output when the system controller acts as a PCI slave. TRDY# indicates that the target device is ready. This signal is 3-stated after a hard reset.

Initiator Ready, IRDY#, is an output when system controller is a PCI master. IRDY# is an input when the system controller is a PCI slave. IRDY# indicates that the initiator of the cycle is ready. This signal is 3-stated after a hard reset.

LOCK# indicates an exclusive bus operation and may require multiple transactions to complete. The system controller supports a bus type of LOCK only. Thus, when a PCI master locks the PCI bus, it owns the system for the duration of the locked transactions.

Stop, STOP#, indicates that the current bus target is requesting the master to stop the current transaction. STOP# is used to disconnect, retry, and abort sequences on the PCI bus. This signal is 3-stated after a hard reset.

Parity, PAR, is driven by the system controller, as a PCI master, during the address and data phases for a write cycle and during the address phase for a read cycle. When the system controller is a PCI slave, parity is driven by the system controller for the data phase of a PCI read cycle. Parity is even across AD31–AD0 and C/BE3#–C/BE0#. PAR lags the corresponding address and data phase by one PCICLK. This signal is asserted after a hard reset.

System Error, SERR#, when driven by the system controller, indicates that either a main memory parity error occurred or the system controller, as a master, received a target abort.

Device Select, DEVSEL#, when asserted, indicates that a PCI slave device has decoded the bus cycle address as the target of the current access. The system controller drives DEVSEL# based on the main memory address range being accessed by a PCI master. As an input, DEVSEL# indicates whether any device on the bus has been selected. This signal is 3-stated after a hard reset.

Request1/Host Device, PREQ1#/HDEV#, is a multiplexed signal that has two functions. PREQ1# is used by the PCI master to gain control of the PCI bus. This signal can be externally cascaded to support multiple PCI masters. The HDEV# function is used when the system controller is programmed to support a Host bus slave device.

Request0, PREQ0#, is used by the PCI master to gain control of the PCI bus. This signal can be externally cascaded to support multiple PCI masters.

Grant1/Host Ready, PGNT1#, is driven by the system controller to grant control of the PCI bus to a PCI master. PGNT1# can be externally cascaded to support multiple PCI masters. The HRDY# function is used when the system controller is programmed to support a Host bus slave device. This signal is driven high during and after a hard reset.

Grant0, PGNT0#, is driven by the system controller to grant control of the PCI bus to a PCI master. PGNT0# can be externally cascaded to support multiple PCI masters. This signal is driven high during and after a hard reset.

8.4.4 System Controller/ISA Bridge Link Interface

The system controller and ISA bridge interface communications include CPU/PCI accesses of the ISA bridge internal registers, CPU/PCI cycles forwarded to the ISA bus, and ISA master or DMA accesses to main memory. The system controller/ISA bridge link interface is a point-to-point communication connection between the system controller and the ISA bridge.

Four sideband signals synchronize data flow and bus ownership: Link Request (LREQ#), Link Grant (LGNT #), Command Valid (CMDV#), and Slave Idle (SIDLE#). LREQ# and LGNT# are used by the ISA bridge to arbitrate for link mastership. Only the ISA bridge drives LREQ# while on the system controller drives LGNT#. CMDV# is driven by the current link master, whereas SIDLE# is driven by the current link slave. Commands, addresses, and data are transferred between the system controller and ISA bridge using the host address bus signals (A17–A2).

8.4.4.1 Status and Control Interface

Command Valid, CMDV#, is asserted by the link master to indicate the beginning of a link transfer. The system controller deasserts this signal after a hard reset. CMDV# is used along with SIDLE# to set the system controller/ISA bridge system clock configuration during a PWROK hard reset. These inputs are strapped to the appropriate levels, sampled while PWROK is inactive, and latched when PWROK goes active.

Slave Idle, SIDLE#, is asserted by the link slave to indicate that it is available for data transfers. The ISA bridge asserts this signal after a hard reset. SIDLE# is used along with CMDV# to set the system controller/ISA bridge system clock configuration during PWROK hard reset. These inputs are strapped to the appropriate levels, sampled while PWROK is inactive, and latched when PWROK goes active.

Link Request, LREQ#, is asserted by the ISA bridge to request a link transfer. This signal is deasserted after a hard reset.

Link Grant, LGNT#, is asserted by the system controller to grant the ISA bridge a link transfer. This signal is deasserted after a hard reset.

Host Address/Link, A17–A2 for system controller/ISA bridge, link transfers of data/commands between the ISA bridge and system controller. These signals are 3-stated after a hard reset.

8.4.5 ISA Interface

The ISA bridge incorporates a fully ISA bus compatible master and slave interface. The ISA bridge directly drives five ISA slots without external data or address buffers. The ISA interface also provides byte swap logic, I/O recovery support, wait-state generation, and SYSCLK generation. The ISA interface supports the following cycle types:

- CPU or PCI master initiated I/O and memory cycles to the ISA bus.
- DMA-compatible cycles between main memory and ISA I/O, and between ISA I/O and ISA memory.
- ISA refresh cycles initiated by either the ISA bridge or an external ISA master.
- ISA master-initiated memory cycles to main memory and ISA master-initiated I/O cycles to the internal ISA bridge registers.

8.4.5.1 I/O Recovery Support

The I/O recovery mechanism in the ISA bridge is used to add additional recovery delay between the CPU or PCI master initiated 8-bit and 16-bit I/O cycles to the ISA bus. The ISA bridge automatically forces a minimum delay of 3.5 SYSCLKs between back-to-back 8- and 16-bit I/O cycles to the ISA bus. This delay is measured from the rising edge of the I/O command (IOR# or IOW#) to the falling edge of the next I/O command. If a delay of greater than 3.5 SYSCLKs is required, the ISA I/O Recovery Timer register can be programmed to increase the delay in increments of SYSCLKs. No additional delay is inserted for back-to-back I/O sub-cycles generated as a result of byte assembly or disassembly.

8.4.5.2 SYSCLK Generation

The ISA bridge generates the ISA system clock (SYSCLK). SYSCLK is a divided down version of HCLKOUT and has a frequency of either 8.00 or 8.33 MHz, depending on the HCLKOUT frequency.

For CPU or PCI initiated cycles to the ISA bus, SYSCLK is stretched to synchronize BALE falling to the rising edge of SYSCLK. During CPU or PCI initiated cycles to the ISA bridge, BALE is normally driven high, synchronized to the rising edge of SYSCLK and then driven low to initiate the cycle on the ISA bus. However, if the cycle is aborted, BALE remains high and is not driven low until the next cycle to the ISA bus.

8.4.5.3 Data Byte Swapping (ISA Master or DMA to ISA Device)

The data swap logic is integrated in the ISA bridge. For slaves that reside on the ISA bus, data swapping is performed if the slave (I/O or memory) and ISA bus master (or DMA) sizes differ and the upper (odd) byte of data is being accessed. The data swapping direction is determined by

the cycle type (read or write). Table 15 shows when data swapping is provided during DMA and ISA master cycles to ISA slaves.

Table 8-3. DMA Data Swap

DMA I/O Device Size	ISA Memory Slave Size	Swap	Comments (I/O) ↔ Memory
8-bit	8-bit	No	SD[7:0]↔SD[7:0]
8-bit	16-bit	No	SD[7:0]↔SD[7:0]
8-bit	16-bit	Yes	SD[7:0]↔SD[15:8]
16-bit	8-bit	No	Not Supported
16-bit	16-bit	No	SD[15:0]↔SD[15:0]

Table 8-4. 16-bit Master to 8-bit Slave Data Swap

SBHE#	SA0	SD[15:8]	SD[7:0]	Comments
0	0	Odd	Even	Word Transfer (data swapping not required)
0	1	Odd	Odd	Byte Swap (1, 2)
1	0		Even	Byte Transfer (data swapping not required)
1	1			Not Allowed

8.4.5.4 Wait-State Generation

The ISA bridge adds wait-states to the following cycles, if IOCHRDY is sampled low (deasserted).

- During Refresh and ISA bridge master cycles (not including DMA) to the ISA bus.
- During DMA-compatible transfers between ISA I/O and ISA memory only.

Wait states are added as long as IOCHRDY remains low.

For ISA master cycles targeted for the ISA bridge's internal registers or main memory, the ISA bridge always extends the cycle by driving IOCHDY low until the transaction is complete.

8.4.5.5 Cycle Shortening

The ISA bridge shortens the following cycles, if ZEROWS# is sampled asserted (low).

- During ISA bridge master cycles (not including DMA) to 8-bit and 16-bit ISA memory.
- During ISA bridge master cycles (not including DMA) to 8-bit ISA I/O only.

For ISA master cycles targeted for the ISA bridge's internal registers or main memory, the ISA bridge does not assert ZEROWS#. When IOCHRDY and ZEROWS# are sampled low at the same time, IOCHRDY takes precedence and wait states are added.

8.4.5.6 Status and Control Interface

Bus Address Latch Enable, BALE, is asserted by the ISA bridge to indicate that the address (SA19–SA0, LA23–LA17) and SBHE# signal lines are valid. This signal is deasserted after a hard reset.

Address Enable, AEN, is asserted during DMA cycles to present I/O slaves from misinterpreting DMA cycles as valid I/O cycles. This signal is also asserted during ISA bridge-initiated refresh cycles. This signal is deasserted after a hard reset.

I/O Channel Ready, IOCHRDY, is deasserted by resources on the ISA bus to indicate that additional time (wait-states) is required to complete the cycle. This signal is normally high on the ISA bus. IOCHRDY is an input when the ISA bridge owns the ISA bus and the CPU or a PCI agent is accessing an ISA slave, or during DMA transfers. IOCHRDY is output when an external ISA bus master owns the ISA bus and is accessing main memory or an ISA bridge register. As an ISA bridge output, IOCHRDY is deasserted from the falling edge of the ISA commands. After data is available for an ISA master read or the ISA bridge latches the data for a write cycle, IOCHRDY is asserted for 70 ns. After 70 ns, the ISA bridge three-states IOCHRDY. The ISA bridge does not drive this signal when an ISA bus master is accessing an ISA bus slave. IOCHRDY is 3-stated upon CPURST.

16-bit I/O Chip Select, ISCS16#, is driven by I/O devices on the ISA bus to indicate that they support 16-bit I/O bus cycles.

I/O Channel Check, IOCHK#, can be driven by any resource on the ISA bus. When asserted, it indicates that a parity or an uncorrectable error has occurred for a device or memory on the ISA bus. If IOCHK# is asserted and NMIs are enabled, an NMI is generated to the CPU.

I/O Read, IOR#, when asserted indicates to an ISA I/O slave device that the slave may drive data on the ISA data bus (SD15–SD0). The I/O slave device must hold the data valid until after IOR# is deasserted. IOR# is an output when the ISA bridge owns the ISA bus. IOR# is an input when an external ISA master owns the ISA bus. This signal is deasserted after a hard reset.

I/O Write, IOW#, asserted indicates to an ISA I/O slave device that the slave may latch data from the ISA data bus (SD15–SD0). IOW# is an output when the ISA bridge owns the ISA bus. IOW# is an input when an external ISA master owns the ISA bus. This signal is deasserted after a hard reset.

Unlatched Address, LA23–LA17, are bi-directional address lines allowing accesses to physical memory on the ISA bus up to 16 Mbytes. LA23–LA17 are outputs when the ISA bridge owns the ISA bus. The LA23–LA17 lines become inputs when an ISA master owns the ISA bus. The LA23–LA17 signals are driven to an unknown state after a hard reset.

System Address bus, SA19–SA0, are outputs when the ISA bridge owns the ISA bus. SA19–SA0 are inputs when an external ISA master owns the ISA bus. Note that SA19–SA17 have the same values as LA19–LA17 for all memory cycles. For I/O accesses only SA15–SA0 are used. SA19–SA0 are driven to an unknown state after a hard reset.

System Byte High Enable, SBHE#, indicates, when asserted, that a byte is being transferred on the upper byte (SD15–SD8) of the data bus. SBHE# is deasserted during refresh cycles. SBHE# is an output when the ISA bridge owns the ISA bus and an input when an external ISA master owns the ISA bus. This signal is at an unknown state after a hard reset.

Memory Chip Select, 16 MEMCS16#, is driven low by ISA slaves that are 16-bit memory devices. MEMCS16# is an input when the ISA bridge owns the ISA bus. MEMCS16# is an output when an ISA bus master owns the ISA bus. The ISA bridge drives this signal low during ISA master to main memory cycles.

Memory Read, MEMR#, is the command to a memory slave that it may drive data onto the ISA data bus. MEMR# is an output when the ISA bridge is a master on the ISA bus and an input when an ISA master, other then the ISA bridge, owns the ISA bus. This signal is also driven by the ISA bridge during refresh cycles. For DMA cycles, the ISA bridge, as a master, asserts MEMR#. This signal is 3-stated after a hard reset.

Memory Write, MEMW#, is the command to a memory slave that it may latch data from the ISA data bus. MEMW# is an output when the ISA bridge owns the ISA bus and an input when an ISA master, other then the ISA bridge, owns the ISA bus. For DMA cycles, the ISA bridge, as a master, asserts MEMW#. This signal is 3-stated after a hard reset.

Standard Memory Read, SMEMR#, is asserted to request an ISA memory slave to drive data onto the data lines. If the access is below the 1 Mbyte range (00000000-000FFFFFh) during DMA compatible, ISA bridge master, or ISA master cycles, the ISA bridge asserts SMEMR#. SMEMR# is a delayed version of MEMR#. This signal is deasserted after a hard reset.

Standard Memory Write, SMEMW#, is asserted to request an ISA memory slave to accept data from the data lines. If the access is below the 1 Mbyte range (00000000-000FFFFFh) during DMA compatible, ISA bridge master, or ISA master cycles, the ISA bridge asserts SMEMW#. SMEMW# is a delayed version of MEMW#. This signal is deasserted after a hard reset.

Zero Wait-States, ZEROWS#, is asserted by an ISA slave after its address and command signals have been decoded to indicate that the current cycle can be shortened. If IOCHRDY is deasserted and ZEROWS# is asserted during the same clock, then ZEROWS# is ignored and wait-sates are added as a function of IOCHRDY (i.e. IOCHRDY has precedence over ZEROWS#).

System Data, SD15–SD8, provide the 16-bit data path for devices residing on the ISA bus. SD15–SD8 correspond to the high order byte and SD7–SD0 correspond to the low order byte. SD15–SD0 are undefined during refresh. These signals are 3-stated after hard reset.

8.4.6 DMA Controller

The DMA circuitry incorporates the functionality of two 82C37 DMA controllers with seven independently programmable channels (Channels 3–0 and Channels 7–5). The DMA supports 8/16-bit devices using ISA-compatible timings and 27-bit addressing as an extension of the ISA-compatible specification. The DMA channels can be programmed for either fixed (default) or rotating priority. The DMA controller also generates ISA refresh cycles. DMA Channel 4 is used to cascade the two controllers and default to cascade mode in the DMA Channel Mode (DCM) register (Figure 10). In addition to accepting requests from DMA slaves, the DMA controller also responds to requests that are initiated by software. Software may initiate a DMA service request by setting any bit in the DMA Channel Request register to a 1. The DMA controller for Channels 3–0 is referred to as “DMA-1” and the controller for Channels 7–4 is referred to as “DMA-2”.

Each DMA channel is hardwired to the compatible settings for DMA device size channels 3–0 are hardwired to 8-bit, count-by-bytes transfers and channels 7–5 are hardwired to 16-bit, count-by-words (address shifted) transfers. The ISA bridge provides the timing control and data size

translation necessary for the DMA transfer between the memory (ISA or main memory) and the ISA bus I/O. ISA-compatible DMA timing is supported. The DMA controller also features refresh address generation and auto-initialization following a DMA termination.

Note that a DMA device (I/O device) is always on the ISA bus, but the memory referenced is located on either an ISA bus device or in main memory. When the ISA bridge is running a DMA cycle, it drives the MEMR# or MEMW# strobes, if the address is less than 16 Mbytes (000000-FFFFFFh). The ISA bridge always generates ISA-compatible DMA memory cycles. The SMEMR# and SMEMW# are generated if the address is less than 1 Mbyte (0000000-00FFFFFh). To avoid aliasing problems when the address is greater than 16 Mbytes (1000000-7FFFFFFh), the MEMR# or MEMW# strobe is not generated.

The channels can be programmed for any of four transfer modes: single, block, demand, or cascade. Each of the three active transfer modes (single, block, and demand), can perform three different types of transfers (read, write, or verify). Note that memory-to-memory transfers are not supported by the ISA bridge. The DMA supports fixed and rotating channel priorities.

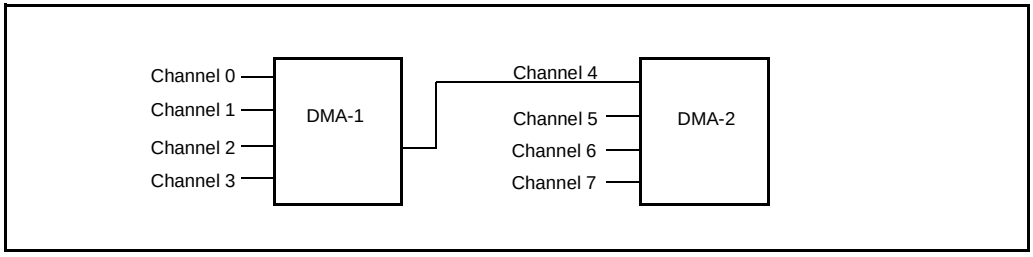


Figure 8-8. Internal DMA Controller

8.4.6.1 DMA Status and Control Interface

DMA Request lines, DREQ3–DREQ0, DREQ7–DREQ5, are used to request DMA service from the ISA bridge's DMA controller or for a 16-bit master to gain control of the ISA expansion bus. The active level (high or low) is programmed via the DMA Command register. All inactive to active edges of DREQ are assumed to be asynchronous. The request must remain active until the appropriate DACK*x*# signal is asserted.

DMA Acknowledge output lines, DACK3#–DACK0#, DACK7#–DACK5#, indicate that a request for DMA service has been granted by the ISA bridge or that a 16-bit master has been granted the bus. The active level (high or low) is programmed via the DMA Command register. These signals are deasserted after a hard reset.

Terminal Count, TC, is asserted by the ISA bridge to DMA slaves as a terminal count indicator. This signal is deasserted after a hard reset.

Refresh, REFRESH#, is an output when asserted indicates when a refresh cycle is in progress. As an output, this signal is driven directly onto the ISA bus. This signal is an output only when the ISA bridge DMA refresh controller is a master on the bus responding to an internally generated request for refresh. As an input, REFRESH# is driven by 16-bit ISA bus masters to initiate refresh cycles. This signal is 3-stated after a hard reset.

9

Performance Considerations

Chapter Contents

CHAPTER 9
PERFORMANCE CONSIDERATIONS

9.1 INTRODUCTION

System performance is a key attribute of any embedded computer system. How quickly a program is run is the common measure of performance. Program performance is a function of many parameters: CPU speed, clock speed, memory latency, memory data transfer rate, memory size, disk access time, disk data transfer rate, video access time, compiler efficiency, operating system efficiency, program algorithms, etc. This chapter focuses on the memory system parameters that affect performance. External caches are also examined as a means of improving memory system performance. Chapters 5 and 6 give specific examples of memory and cache designs.

Memory system design is important. The Intel486™ processor is faster than any practical memory system. It contains a significant amount of logic (e.g., caches, write buffers, prefetcher) to allow the execution logic to keep operating even with slow external memories. The on-chip caches and data bandwidth requirements of the Intel486 processor are different than earlier microprocessors. Memory system design should be approached differently as well. This chapter describes the memory requirements and bus usage characteristics of the Intel486 processor.

9.1.1 Memory Performance Factors

The ideal memory subsystem would operate without wait states. All bus cycles on the Intel486 processor would complete in only two clocks for single access and five clocks for cache fill. This is impractical for almost all applications since they would require huge amounts of 15 ns memory to run at 33 MHz. Practical systems use DRAM of 60-100 ns access times. The Intel486 processor is designed to effectively use DRAM. This chapter examines memory system design using DRAM.

There are many different performance options in the design of the memory subsystem for the Intel486 processor. The CPU clock speed sets the maximum possible performance. Higher is faster, but it then requires faster memories to keep the whole system performance scaling at the frequency rate. The Intel486 processor is designed to allow overall performance to increase up to a point with higher clock speed and constant memory speed.

The most common attribute of memory design is the number of wait states, if any, that are required to read a data item. At 33 MHz, a read operation requires 15 ns SRAM. For slower DRAM or Flash access, at 33 MHz add 30 ns access time for each wait state. Wait states exist in practical memory system design. This chapter examines how they affect Intel486 processor performance.

The Intel486 processor adds a new metric to memory design: read transfer rate. It is important for filling the internal cache of the Intel486 processor. The Intel486 processor can transfer data from memory on every clock for most read transfers. This is twice the rate of individual memory cycles. Memory systems supporting this high speed transfer rate increase performance 10-20% over those without.

A third important attribute is write cycle time. The Intel486 processor write-through cache generates approximately twice as many writes as reads. Write performance is especially important

for 16-bit programs, which generate more writes than 32-bit programs. The cycle time of the write can limit system performance as the total bus usage approaches the maximum allowed.

A common method of improving memory system performance is to add a cache. The Intel486 processor has an on-chip cache (known as L1 cache), which handles most of the read requests. The performance gain is highly dependent on the application—some applications benefit less than 5% with an external cache. Most applications benefit 10-15% in performance, while a few benefit as much as 40%. An external cache is not required for many Intel486 processor applications.

A high-performance Intel486 processor design needs to consider all of these issues in the memory design. The following sections provide more detail on the activity of the Intel486 processor during typical program execution. The memory activity of the CPU needs to be understood to best design the memory subsystem.

9.2 INSTRUCTION EXECUTION PERFORMANCE

The Intel486 processor was designed to execute instructions in fewer clocks than earlier Intel386™ family microprocessors. The reduced clock counts increase performance relative to earlier products. This section reviews how the Intel486 processor accomplishes this and compares it to earlier Intel microprocessors.

The instruction execution rate and internal design is important to understand when designing memory systems. It accounts for the heavy write traffic on the Intel486 processor as compared to earlier microprocessors. It also explains how memory bandwidth and latency affect performance.

9.2.1 Intel486™ Processor Execution Times

The Intel486 processor uses several techniques to execute many frequent instructions in a single clock. The processor has an on-chip code/data cache and a five stage pipelined execution unit. The Intel486 processor decodes many simple instructions directly into hardware actions and uses write buffers to match the execution rate to memory bus speed.

One high-level way to examine the impact of these techniques is to compare the execution time of a typical application. To do so, Intel has measured a set of applications for the frequency of instruction usage. For each instruction we multiply the frequency times the clocks required to execute. The sum of these products then yields the typical number of clocks required to execute an instruction.

Table 9-1 shows such a comparison. The Intel486 processor requires 1.95 clocks for a typical instruction while the Intel386 microprocessor requires 4.919 clocks. This is a 2.5x improvement for integer programs. The floating-point instructions have an even larger improvement, as discussed later. The numbers in Table 9-1 do not include effects of cache misses for the Intel486 processor.

One implication of these numbers is that the Intel486 processor cannot sustain that rate of execution with the cache disabled. The bus bandwidth required for the Intel486 processor with cache disabled would be 2.5 times that of the Intel386 CPU. The Intel486 processor bus has 60% more data bandwidth for reads than the Intel386 CPU, but the same bandwidth for writes. The on-chip cache of the Intel486 processor handles most (90-95%) of the read requests. The external bus

must handle all of the writes. A later section examines bus utilization and on-chip cache hit rates in more detail.

Table 9-1. Typical Instruction Mix and Execution Times for the Intel486™ Processor

Instruction	Percentage Utilization	Intel486™ Processor Clocks	Intel486™ Accumulated Clocks
Move R,M	16.2%	1.16	0.188
Move M,R	6.9%	1	0.069
Push R	6.1%	1	0.061
Move R,R	5.7%	1	0.057
Move R,I	5.5%	1	0.055
JCC taken	4.6%	3.4	0.156
JCC fail	4.5%	1	0.045
ALU2 R,R	4.3%	1	0.043
POP R	4.0%	1.16	0.046
JMP M	2.9%	3.4	0.099
ALU2 R,M	2.9%	2.16	0.063
ALU2 M,I	2.9%	3.16	0.092
Call	2.8%	3.4	0.095
Shift R	2.8%	2	0.056
ALU2 R,I	2.8%	1	0.028
RET	2.7%	5.56	0.028
String	2.6%	3.16	0.150
ALU1 R	1.2%	1	0.082
LDS	1.4%	12	0.020
ALU2 M,R	1.3%	3.16	0.168
ALU1 M	1.2%	3.16	0.041
Push M	1.1%	2.16	0.024
NOP	1.1%	1	0.011
Others	11.7%	2.25	0.263
Average clocks per instruction			1.95

NOTE: All percentages are approximate.

9.2.2 Application Programs Used in Analysis

For the bus utilization and cache statistics presented later, a series of five programs were used. Each was traced to record the address access pattern. These patterns were then used in a cache simulator to measure how many accesses could be handled in the on-chip cache of the Intel486 processor. The cache simulator is an accurate representation of on-chip cache. External bus traffic was also measured to give bus utilization statistics. An external DRAM controller and external cache can also be simulated to measure their effect on program execution.

The programs represent different types of work. Each was run in the UNIX environment. Some are 16-bit DOS applications run under a DOS emulator. Each had 16 million memory references recorded.

9.3 INTERNAL CACHE PERFORMANCE ISSUES

The Intel486 processor is capable of high speed operations, as fast as 1 clock for many common instructions. Since external memory cannot provide data for the CPU every clock, an on-chip cache that can be accessed very quickly is necessary to enhance the overall performance. The cache eases the bandwidth differences between the external bus and the CPU. The size, organization, write policy, miss replacement, and busing of the Intel486 processor on-chip cache were chosen to support a broad range of applications.

9.3.1 On-Chip Cache Organization Issues

The Intel486 processor contains an 8-Kbyte (16-Kbyte on the IntelDX4 processor) on-chip cache. The cache is unified (containing both code and data), and is organized as 4-way set-associative, with four 2-Kbyte (4-Kbyte on the IntelDX4 processor) sets. Each set contains 128 lines (256 lines on the IntelDX4 processor). Cache lines are 16 bytes long. Lines in the cache are either valid or not valid. There is no provision for partially valid lines.

Read requests are generated either by program flow (data request) or an instruction prefetch (code request). The great majority of the time, these requests are usually satisfied by the on-chip cache. However, if a cache miss occurs, an external bus request is generated. For reads to non-cacheable areas of memory, the read is completely normal. If, however, the read request is to a cacheable portion of memory, then the CPU initiates a cache bus line fill. Cache line fills require the execution of additional bus cycles in order to read the remainder of the 16-byte line into the CPU.

Cache line size can impact system performance. If the line size is too large, then the number of blocks that can fit in the cache is reduced. In addition, as the line length is increased, the latency for the external memory system to fill a cache line increases, reducing overall performance.

However, the Intel486 processor bus is optimized for a line size of 16 bytes. Because the Intel486 processor can access four bytes in each bus cycle and the cache lines are 16 bytes long, four bus cycles are necessary to fill a cache line. To reduce latency of reading cache lines, the CPU uses burst cycles. During burst cycles, four bytes of data can be read into the CPU every clock. With the use of burst cycles, a 16-byte cache line can be read into the CPU in as few as five clock cycles. Static column DRAMs can be implemented to support burst cycles to the CPU.

During writes, the main memory update method utilized by the Intel486 processor (except for the IntelDX4 processor) is the write-through policy. All writes from the Intel486 processor initiate

an external bus cycle. In addition, the internal cache is updated when the address written to is contained in the cache. This policy ensures consistency between the on-chip cache and the external memory. The IntelDX4 processor can be configured to update main memory using the write-back policy. During writes, the cache is updated when the address being written to is contained in the cache. The write is not propagated through the system to memory, but is stored and written to memory during a future update.

9.3.2 Performance Effects of the On-Chip Cache

If all program operations use on-chip resources, the fastest possible execution is achieved, as the on-chip registers and cache satisfy all requests. However, on cache read misses or any memory write operation, the external bus has to be accessed, reducing system performance.

A hit rate of approximately 95% is realized from the on-chip cache, depending on the application. The high level of cache hits has three main effects.

1. Performance is improved. The Intel486 processor can access data from its on-chip cache every clock. This high bandwidth allows the execution unit of the Intel486 processor to execute many common instructions in one clock.
2. The system bus utilization decreases. Because a high percentage of reads are satisfied by the cache, the Intel486 processor bus is idle a large percentage of the time. Additional bus masters can reside in the system without bus saturation and the resulting performance degradation.
3. The ratio of writes to reads is increased on the external bus. The number of reads is decreased but the amount of writes remains constant. Therefore, main memory systems should have low latency on write operations.

Internally, two separate 128-bit wide prefetch buffers interface to the L1 cache unit. These can be filled with data fetched from the on-board cache in one clock cycle, or by external memory in as few as four clock cycles. Because the wide prefetch buffers satisfy multiple prefetches, the usual degradation caused by a combined code cache and data cache scheme is avoided.

To optimize performance during cache line fills, a technique called bypassing is used. The first cycle of a cache line fill satisfies the original request. Data read in during the first cycle is sent directly to the requesting unit. Because of this, it is not necessary to wait for the entire cache line to fill before the requested data can be used.

Figure 9-1 shows the on-chip hit rates for prefetch and read operations when running the programs shown in Table 9-2.

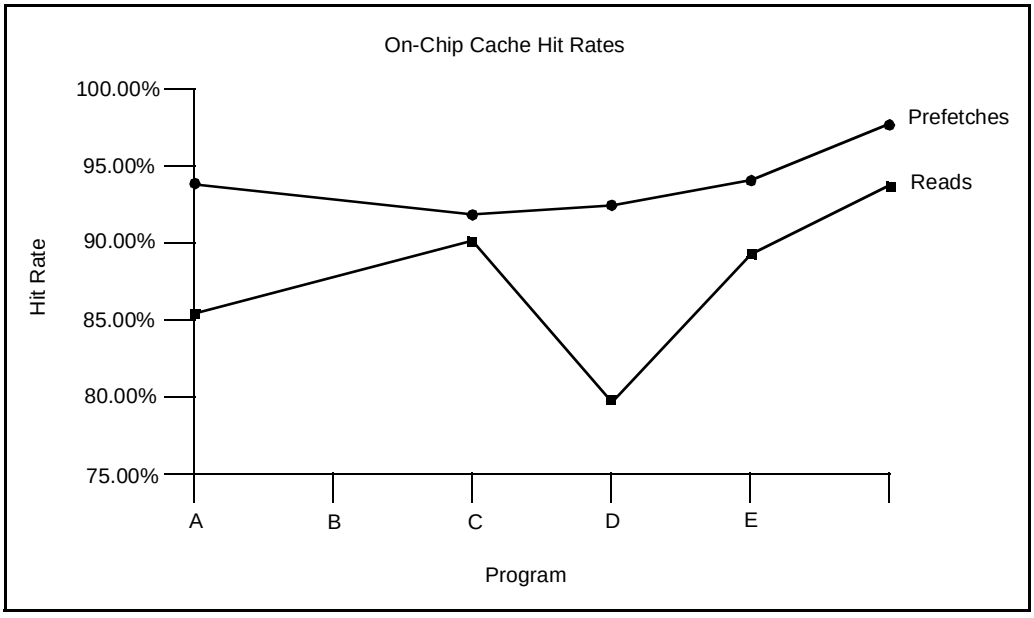


Figure 9-1. Cache Hit Rate for Various Programs

Table 9-2. Programs Used

	Name	Description
A	FRAME	Desktop publishing package
B	PHONGS4	Small benchmark program
C	Sunview	Window manager
D	INVFRAME	Desktop publishing package
E	TPASCAL	Pascal compiler
F	TROFF	Text Formatter

9.3.3 Bus Cycle Mix with and without On-Chip Cache

Microprocessors that lack an on-chip cache must devote a significant portion of execution time to external bus accesses. Code prefetches and data reads must come from the external memory system; subsequently a high percentage of bus accesses are reads. Traditional memory systems are optimized for reads because of this mix of bus cycles.

With the Intel486 processor's on-chip cache, however, the high hit rate reduces the number of external reads. As the on-chip cache implements a write-through policy, the number of writes to the bus is not reduced. As a result, external bus read cycles are now a minor portion of the overall

bus cycles, as shown in Figure 9-3. For best performance, memory systems that use the Intel486 processor should be optimized for write cycles.

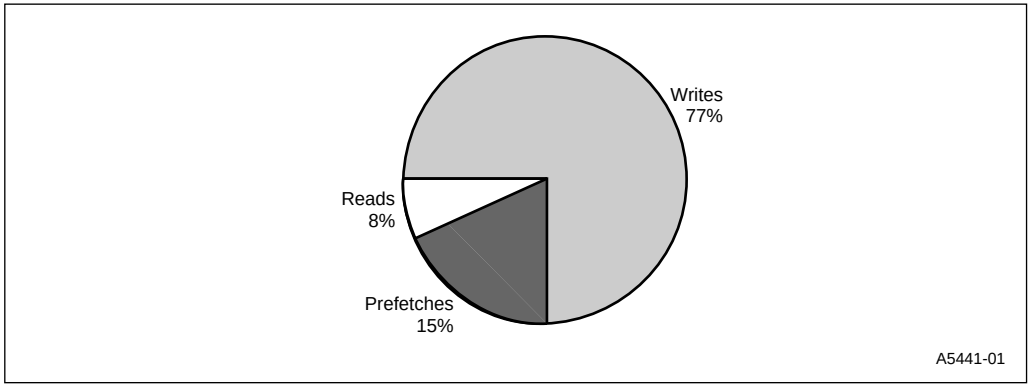


Figure 9-2. Intel486™ Processor Bus Cycle Mix with On-Chip Cache

9.4 ON-CHIP WRITE BUFFERS

As previously discussed, low write latency is more critical for Intel486 processor systems than in previous processors. The Intel486 processor has four write buffers to allow CPU execution without latency for write operations. The buffers can be filled at the rate of one per clock cycle until all four are filled.

When all four write buffers are empty and the bus is idle, a write request propagates to the external bus, bypassing the write buffers directly. If the bus is not available when the write cycle is generated internally, then the write is buffered and propagated as soon as the bus is available. If a cache hit occurs on a write, then the on-chip cache is updated immediately.

Writes are normally executed on the external bus in the same order in which they are received by the write buffers, as in a FIFO. Under certain conditions a memory read can take priority, and the sequence of external bus cycles can be reordered, even though the writes occurred earlier in program execution.

A memory read will only be reordered before all writes under the following conditions. If all writes in the buffers are cache hits and the read is a cache miss, then the read is guaranteed not to conflict with the pending writes. In this case, the bus cycles can be reordered to allow the read operation to occur before the write buffers have been retired.

Intel486 processor performance is enhanced because of the write buffers and bus cycle reordering. The write buffers decouple the internal execution unit from the bus. Program execution can continue without delay of write latency. In addition, reordering allows program execution to continue in some cases even if some write buffers are filled.

9.5 EXTERNAL MEMORY CONSIDERATIONS

9.5.1 Introduction

A well-designed external memory system is needed to optimize Intel486 processor system performance. A system can be designed using different combinations of SRAMs and DRAMs to provide different price/performance levels. SRAMs have faster access times and do not require precharging between accesses or refresh cycles. DRAMs offer higher densities and are less expensive, but they require refresh circuitry, and require the addition of wait states due to the longer access times.

The overall performance of a microprocessor system is directly related to the performance of the memory subsystem. The great majority of bus cycles are used to access memory for instructions and data. As processor speeds increase, so does the demand for higher-speed memories because a high-performance processor that is coupled with a low performance memory offers no better throughput than a low-performance processor.

The cost of using only fast memories in a system may be prohibitive. Yet as slower devices are added to lower the overall cost, the performance penalty of added wait states increases. At frequencies of 25 MHz or more, optimum memory performance can only be achieved by using very fast memory devices.

The cost performance trade-off can be compromised by partitioning functions and using a combination of both fast and slow memories. The most frequently used functions are placed in a faster memory. A common use of faster memory devices is implementation of an external cache, built of fast SRAM devices.

Fast SRAM devices have high enough bandwidth to achieve optimum performance. An external cache (also called L2 cache) can also be used for higher performance. Chapter 6 covers L2 cache concepts.

Regardless of the use of an external cache, the external memory system consists of a combination of EPROM and DRAM devices. EPROM devices tend to have a long access time. Being nonvolatile, EPROMs are used primarily for initialization routines. After initialization EPROMs are accessed infrequently. Thus, system performance is not dependent upon EPROM latency. If a high-level of performance is desired, EPROM contents may be copied to the DRAM memory array. This technique is called shadowing.

Organization of the DRAM memory array is more critical to system performance. DRAM optimization techniques can be used to reduce the average latency of accesses to DRAM devices.

Several of the memory design concepts described in this chapter depend on the principle of locality for high performance. The locality principle basically states that when a program references a particular location in memory, there is a high probability that nearby locations will then also be referenced. Caches and paged memory DRAM design techniques offer high performance because of locality.

9.5.2 Wait States in Burst and Non-Burst Modes

The Intel486 processor can execute non-burst cycles in as little as two clocks. These cycles are called 2-2 cycles, as read and write cycles take two cycles each. The first 2 refers to read cycle time and the second 2 to write cycle time. Accesses to devices that cannot respond by the end of the second clock require the addition of wait states. If a wait state must be added to write cycles, then a 2-3 system is created. The external system generates RDY# and the RDY# signal is sampled at the end of the second clock. If it is asserted (low) at the sample time, it indicates that the external system has placed valid data on the pins for reads, or that the system has accepted the data for writes. Wait states are inserted by driving RDY# inactive (high) at the end of the second clock.

The Intel486 processor non-burst cycles are very similar to non-pipelined Intel386™ DX CPU cycles. In the Intel386 DX processor, the read and write accesses can be as fast as two cycles each. Thus, adding a wait state increases the bus cycle time by 50 percent of the zero wait state bus cycle time. Overall performance does not degrade in direct proportion to the bus cycle increase.

To enhance read performance, the Intel486 processor supports burst cycles. The Intel486 processor bus can burst successive words from memory into the cache every clock. Most memory reads can be performed in bursts as indicated by the BLAST# pin. The Intel486 processor keeps the BLAST# output inactive in the second clock of the cycle, indicating that it is able to perform a burst cycle. The external system indicates that it will initiate a burst cycle by asserting BRDY#. If BRDY# is not asserted at the second clock, wait states are inserted. If a system executes non-burst reads in two clocks, burst reads in one clock, and writes in three clocks, a 2-1-3 system is indicated.

Because of the on-chip cache, the addition of external wait states affects the Intel486 processor's performance less than previous processors. A wait state in a Intel386 DX system incurs a performance degradation of about 20 percent. The Intel486 processor achieves optimum performance through a 2-1-2, zero wait state bus cycle. Adding one wait state in an Intel486 processor system causes a performance degradation of only about 6 percent.

The Intel486 processor can execute an external bus cycle in as little as two clock cycles. For achieving the optimum system performance, memory accesses must also execute in two cycles to eliminate wait states. At higher frequencies, however, it is impractical and cost-prohibitive to implement zero wait states for all memory.

At 25 MHz, a wait state adds 40 ns to the available access time. While an operation with one wait state increases the bus cycle time by 50 percent, system performance does not degrade in direct proportion. The amount of degradation incurred is application-dependent and varies with instruction mix, external cache size, and the number of memory references.

Several DRAM design techniques can reduce wait states and keep system performance at a high level using slower memory devices. These techniques, page mode design and interleaving, and their impact on performance, are discussed in Chapter 5.

9.5.3 Impact of Wait States on Performance

There are many benchmarks used to evaluate the performance of microprocessor systems. Figure 9-3 demonstrates the performance of Intel486 processor systems using different bus cycle implementations. The 100 percent performance level is an Intel486 processor with an external memory that operates a 2-1-2 cycle. The 2-1-2 cycle achieves the highest level of performance while a 5-1-4 cycle achieves the lowest.

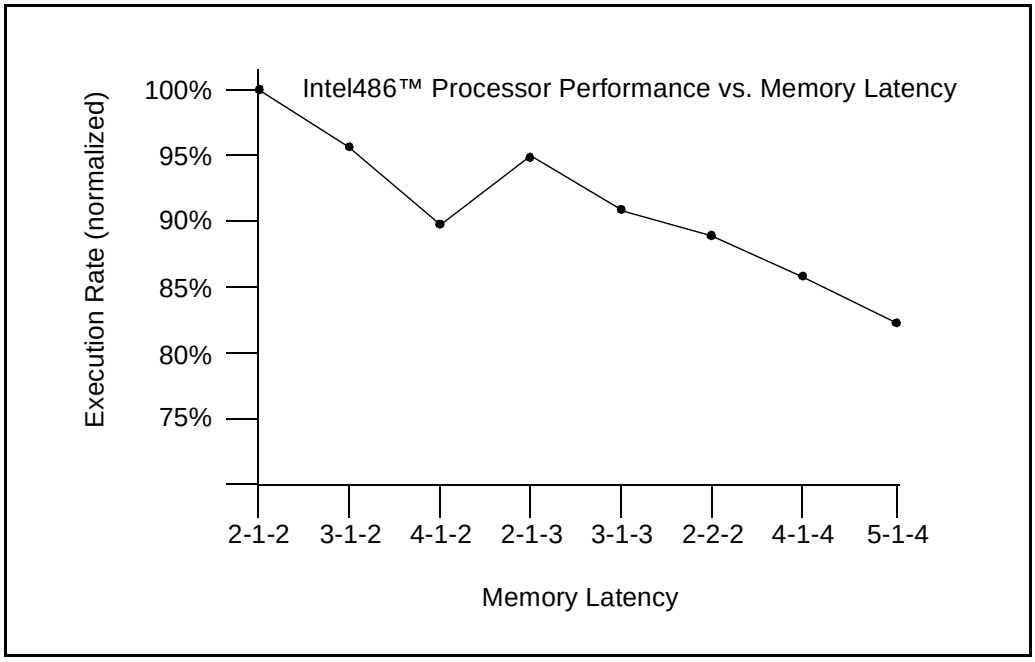


Figure 9-3. Effect of Wait States on Performance

Note that the performance effect of the four on-chip write buffers is apparent. Since more than 75% of external cycles are writes, write latency due to slower external memory should impact overall performance more than read latency. However, the on-chip write buffers reduce the dependence on write latency.

9.5.4 Bus Utilization and Wait States

Figure 9-4 demonstrates external bus utilization versus systems with different wait state configurations. The percentage figures were calculated by dividing the number of bus cycles in which the processor required the bus by the total number of bus cycles. A smaller percentage is better because it indicates that the external bus is accessed less frequently. In the benchmarks used in this demonstration, the percentages varied from 39 percent for a 2-1-2 cycle system to 90 percent for a 5-1-4 cycle system.

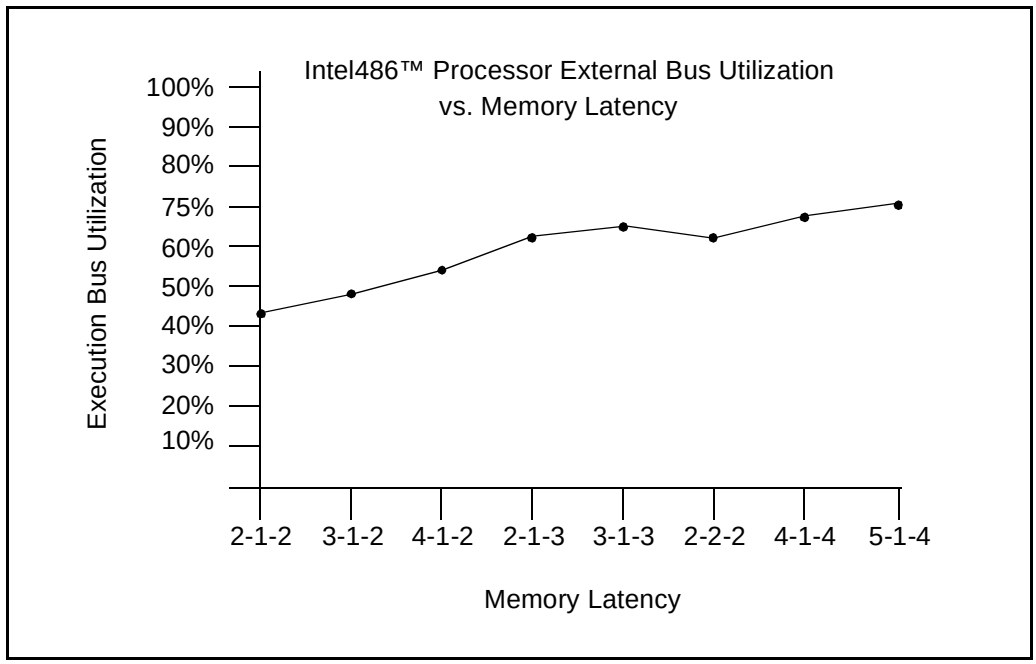


Figure 9-4. Effect of External Bus Utilization versus Wait States

The bus utilization percentage is not critical for single-processor systems. However, when considering multi-processing systems, the amount of time that each CPU needs the bus becomes very important.

9.6 SECOND-LEVEL CACHE PERFORMANCE CONSIDERATIONS

9.6.1 Advantages of a Second-Level Cache

As previously described, approximately 90%-95% of the read cycles generated internally by the Intel486 processor will be satisfied by the processor's on-chip cache. However, the remaining 5%-10% that miss the internal cache will result in external read bus cycles being executed. For best system performance, an external (L2) cache reduces wait states for these read cycles.

This section discusses the use of a L2 cache. Different applications and operating environments experience varying performance benefits from use of an L2 cache. Hit rates for L2 caches depend on the application being executed and the randomness with which the application addresses memory. Systems which make extensive use of multi-tasking should see a very beneficial gain in system performance with use of a L2 cache.

9.6.2 An Example of a Second-Level Cache

The 485Turbocache* Module was a high performance cache designed for the Intel486 processor. This Module provides 64- or 128-Kbytes of cache depth. Multiple 485Turbocache Modules could be cascaded to give 256 Kbyte or 512-Kbyte cache depths. The 485Turbocache Module is organized as a 64- or 128-Kbyte, 2-way set-associative memory. Like the processor, the 485Turbocache Module has a line size of four doublewords. On a cache read operation the address is presented to the 485Turbocache Module, and the tags are compared. If they match, a hit condition has occurred and the data is burst to the Intel486 processor. Data can be sent over in two cycles for the first word, and one cycle for each of the subsequent three doublewords. This implies the fastest read cycle time for cache hits on the 485Turbocache Module. For cache misses, the data is fetched from the main memory, and then sent to both the Intel486 processor and the 485Turbocache Module. On write operations, the 485Turbocache Module operates like the Intel486 processor's cache by updating write hits and not updating write misses. The main memory is updated on all writes, because of the write-through policy.

9.6.3 System Performance with a Second-Level Cache

The performance of an example L2 cache is shown in Figure 9-5. The 1.0 level of performance reflects an Intel486 processor system that operates with 2-1-2 memory accesses. For example, a system which has 4-2-4 cycles for page hits and 7-2-5 cycles for page misses may result in less than 0.6 of optimum (2-1-2) performance with no cache. Adding 256 K of external cache and one level of write buffering to this system increases the performance level to greater than 0.9 optimum performance.

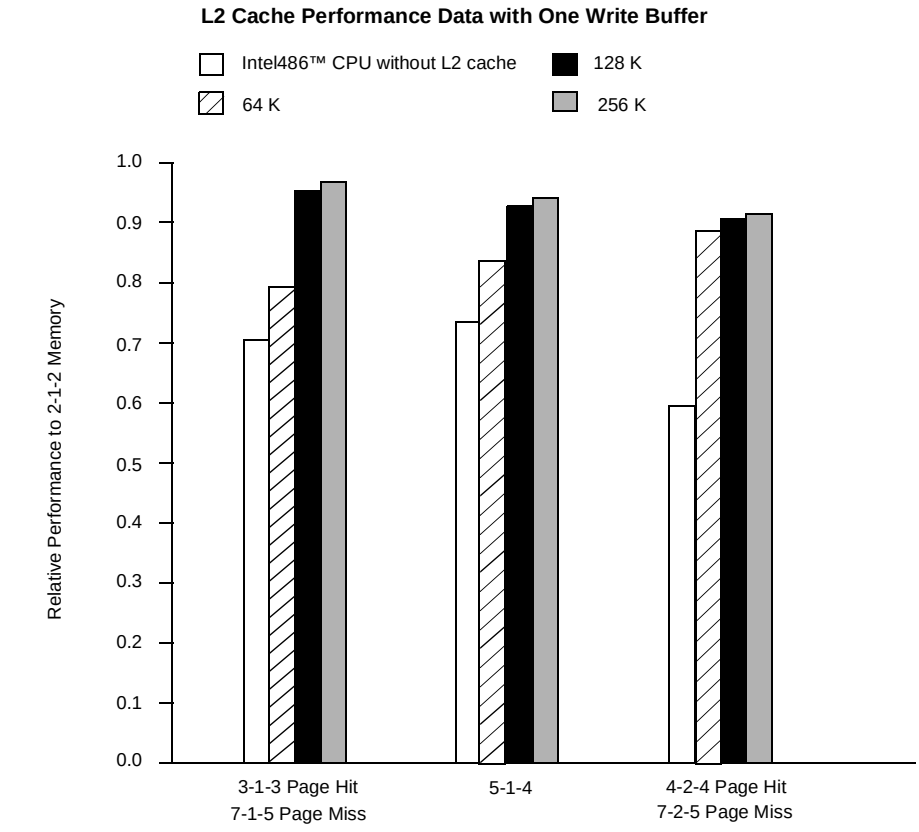


Figure 9-5. L2 Cache Performance Data with One Write Buffer

9.6.4 Impact of Second-Level Cache on Bus Utilization

A second-level cache reduces the number of processor reads to main memory, lowering external system bus utilization. The benefit is more bandwidth available to other bus master devices like DMA or LAN controllers. Systems with multiple CPUs are sensitive to the amount of bus bandwidth used by each CPU. Note that with a write-through cache the minimum bus bandwidth is the number of writes performed.

9.7 DRAM DESIGN TECHNIQUES

An efficient DRAM memory design is needed for a high-performance Intel486 processor system. For some applications, the principle of locality will not be as applicable. A common technique of improving performance with DRAMs uses the commonly seen attribute of locality of reference in programs. This works well with the fast access modes offered by DRAMs that use the same row address. As a result, system performance is more dependent upon DRAM latency.

Normally, a DRAM access is made by first asserting RAS# (Row Address Strobe) to latch the presented row address into the DRAM device. As the DRAM devices have multiplexed address pins, the address must then be externally switched to present the column address. Finally, the CAS# (Column Address Strobe) is asserted to latch the column address and enable the DRAM output buffers. Refer to Chapter 5, "Memory Subsystem Design" for specific details of memory accessing.

The simplest DRAM design offers a fixed number of wait states for each access. As an example, a system could be designed such that all DRAM accesses occur in six clocks. However, many DRAMs offer special modes of operation based on the policy of updating the row address which have higher performance. Some of these modes and their impact on performance are discussed below.

9.8 EXTENDED DATA OUTPUT RAM (EDO RAM)

In Extended Data Output (EDO) RAM designs, a set of gates latch the output data until the CPU reads the data. This is important for high-speed designs because EDO RAM handles sequential reads better than Fast Page Mode (FPM) RAM. Extended Data-Out page mode read accesses are similar to FPM read accesses, except that when CAS is driven high, the data outputs are not disabled, and the data latch is used to guarantee that the valid data is held until CAS goes low again. With EDO RAM, the data latch is controlled during page-mode accesses by CAS. Data is then captured in the latch as a result of CAS going high. A new address can then be applied, and new data accessed, without corrupting the output data from the previous access.

The advantage of an EDO RAM design is that EDO memory has a shorter Page Mode cycle than standard FPM DRAM. Since EDO RAM does not turn the data off by the rising edge of CAS, the data is available longer, enabling the system to read the output data while readying for the next cycle, thus saving one clock cycle for every page access. By eliminating data cycles, EDO memory designs offer an increased peak bandwidth and simplified constraints on access timing, which increase memory performance.

9.8.1 Interleaving

A more complicated DRAM design technique is called interleaving. Interleaving is possible when more than one memory bank is used. Effective implementation of interleaving brings higher performance to a design. Chapter 5, "Memory Subsystem Design" discusses design issues in detail.

Interleaving controls each bank separately. As an access is occurring, the other (non-accessed) banks are being readied for their next access. Interleaving can help provide fast burst accesses for designs. In addition, another use of interleaving is to hide the RAS# precharge time, which is incurred on page misses for paged memory designs. As the number of banks is increased, the

chance for hiding the precharge time is increased. As a result, the performance increases with additional banks.

Figure 9-6 demonstrates the performance differences between an interleaved system supporting one clock bursting and a non-interleaved system in two applications. The performance levels are measured with respect to a zero wait state (2-1-2 bus). Interleaving can improve system performance as much as 15%.

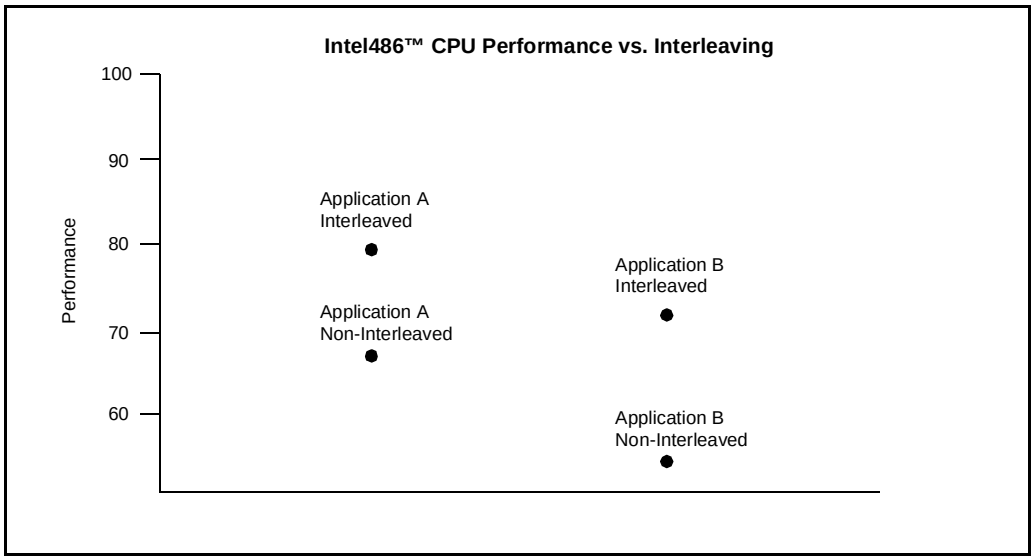


Figure 9-6. Performance in Interleaved and Non-Interleaved Systems

9.8.2 Impact of Performance for Posted Write Cycles

In an Intel486 processor system, the on-board cache reduces the external read cycles so that as much as 77 percent of the external bus cycles are write cycles. In program execution, writes occur in strings of two about 60 to 70% of the time. Writes occur in strings of three 40-50% of the time. The DRAM subsystem must be optimized for write strings; one method is to support posted writes with write buffers. Posting writes means that RDY# is returned to the CPU before the write transaction is completed. This avoids the CPU depending on the write latency time. This is discussed further in Chapter 5, “Memory Subsystem Design.” Figure 9-6 demonstrates the performance in two different applications and shows the improvement gained by using posted writes.

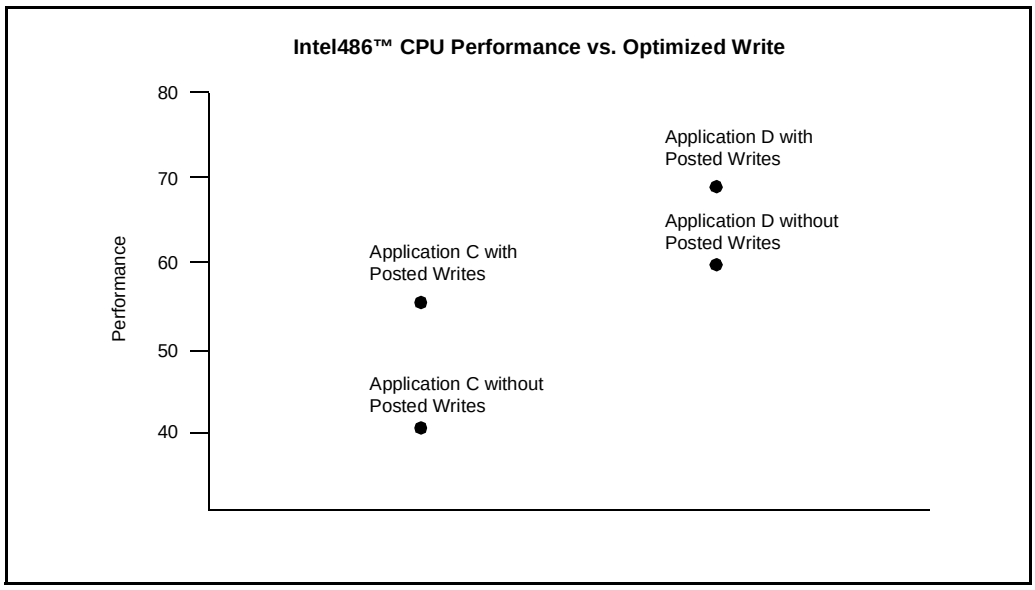


Figure 9-7. Performance in Systems with and without Posted Writes

9.9 FLOATING-POINT PERFORMANCE

9.9.1 Floating-Point Execution Sequences

The floating-point unit on the Intel486 processor contains the logic to execute the floating-point instruction set that is 100% binary compatible to Intel math coprocessors. The floating-point unit operates in parallel with the arithmetic and logic unit, and provides arithmetic functions and transcendental functions. The enhanced floating-point unit provides three to four times the performance of a non-integrated Intel math coprocessor.

An overlap of floating-point instruction execution and non-floating point instruction execution increases the overall throughput.

The floating-point unit can take advantage of pipelined instruction execution. Within the Intel486 processor, the floating-point instructions share the microcode ROM with integer instructions. However, floating-point operations do not utilize the microcode ROM after the operation has been prepared for execution. For example, only the first three clocks of the floating-point add, multiply and divide instructions use the microcode ROM. After the third clock, the floating-point unit completes the operations independently, and the microcode ROM can be utilized by non-floating-point instructions.

Another feature that enhances performance is an efficient on-chip interface. The Intel386 DX CPU and the Intel math coprocessor communicate asynchronously, whereas the Intel486 processor communicates with its on-chip floating-point unit synchronously, allowing higher performance.

The Intel486 processor's on-chip cache dramatically speeds floating-point loads and stores. For the Intel386 processor with a math coprocessor, instructions such as FLD (floating-point load) will take 14-20 clock cycles if any external memory addressing is required. Once operands are on the internal stack, it takes 23 to 31 cycles to execute the floating-point add instruction, depending on the value of the operands. Finally an external memory store can take up to 11-44 cycles.

Because the floating-point unit of the Intel486 processor is integrated, the entire operation executes in fewer cycles. Data from the external memory can be cached. After that it can be accessed by the floating-point unit, and loaded into the stack in three cycles on a cache hit. The floating-point add instruction takes between 8 to 20 cycles depending on the value of the operands. Finally, the store instruction takes 7 clocks.

Because the Intel486 processor provides a higher performance not only for floating point loads and stores, but also for floating-point compute operations, a 3x to 4x performance boost is realized for numerics-intensive routines. A large portion of the performance improvement is attributed to the fact that synchronous floating-point transfers occur on-chip.

9.9.2 Performance of the Floating-Point Unit

To achieve three to four times the floating-point performance of a non-integrated math coprocessor, the Intel486 processor's floating-point circuitry has been enhanced to reduce the number of clock counts needed to execute frequently used instructions. Also, the interface to the processor's registers and buses is much more efficient since all of the interacting units are on the same chip.

Table 9-3 shows the number of clock counts per instruction on the Intel486 processor.

Table 9-3. Floating-Point Instruction Execution

Instruction	Clock Counts Intel486™ Processor
FLD-Load	3
FST-Store	3
FADD/FSUB	8-20
FMUL Floating multiply	16
FDIV Floating divide	73

10

Physical Design and System Debugging

Chapter Contents

CHAPTER 10
PHYSICAL DESIGN AND SYSTEM DEBUGGING

An Intel486™ processor system can easily be implemented using standard interface logic, DRAMs, EPROMs or Flash, and I/O devices. The clock speeds of Intel486 processor family systems require some design guidelines. This chapter outlines the basic design issues, ranging from power and ground issues to achieving the proper thermal environment for the Intel486 processor.

10.1 GENERAL SYSTEM GUIDELINES

The proper operation of any system depends on proper physical layout. The layout issues and design guidelines presented in this chapter are relevant to both higher- and lower-frequency system design implementation.

The improvement of integrated circuit technology has led to an enormous increase in the number of functions that are being implemented on a single chip. Improved technology allows higher clock frequencies. The Intel486 processor, with bus operating frequencies of 25 MHz/33 MHz and corresponding high edge rates and internal clock multiplication, presents a challenge to the conventional interconnection technologies which to date have been adequate for interconnecting less sophisticated devices. This challenge especially applies to system designers who are responsible for providing suitable interconnections at the system level.

The interconnections in a circuit behave like transmission lines which degrade the system's overall speed and distort output waveforms.

In laying out a conventional printed circuit board, there is freedom in defining the length, shape and sequence of interconnections. But with devices such as the Intel486 processor, this task should be carried out with careful planning, evaluation, and testing of the wiring patterns. It is also critical to understand the physical properties of transmission lines because interconnection at high edge rates is analogous to a transmission line.

10.2 POWER DISSIPATION AND DISTRIBUTION

The Intel486 processor uses one-micron or smaller CHMOS IV process technology. The main difference between the previous HMOS microprocessors and the more recent versions is that power dissipation is primarily capacitive, and there is almost no D.C. power dissipation. Because power dissipation is directly proportional to frequency, accommodating high-speed signals on printed circuit boards and through the interconnections is critical. The power dissipation of the Very Large Scale Integration (VLSI) device in operation is expressed by the sum of the power dissipation of the circuit elements, which include internal logic gates, I/O buffers and cache RAMs. It is also a function of the operating conditions.

The worst-case power dissipation of any VLSI device is estimated in the following manner:

- Estimate typical power dissipation for each circuit element:

  P_G: Typical power dissipation for internal logic gates (mW)
  P_{IO}: Typical power dissipation for I/O buffers (mW)
  P_{CRAM}: Typical power dissipation for instruction/data cache RAMs (mW)

- To estimate total typical power dissipation for the device:

  (1) $P_T = P_G + P_{IO} + P_{CRAM}$ (mW),

  where P_T is the total typical power dissipation (mW)

- To estimate the worst case power dissipation:

  (2) $P_d = P_T \times C_V$ (mW),

  where P_d is the worst case power dissipation (mW) and C_V is a multiplier that is dependent upon power supply voltage.

Internal logic power dissipation varies with operating frequency and to some extent with wait states and software. It is directly proportional to supply voltage. Process variations in manufacturing also affect the internal logic power dissipation, although to a lesser extent than with the NMOS processes.

The I/O buffer power dissipation, which accounts for roughly 10 to 25 percent of the overall power dissipation, varies with the frequency and the supply voltage. It is also affected by the capacitive bus loading. The capacitive bus loading for all output pins is specified in the Intel486 processor family datasheets. The Intel486 processor's output valid delays increase if these loadings are exceeded. The addressing pattern of the software can affect I/O buffer power dissipation by changing the effective frequency at the address pins. The frequency variations at the data pins tend to be smaller; a varying data pattern should not cause a significant change in the total power dissipation.

To calculate the total power dissipated by a system board, the following formulas can be used to calculate the maximum statistical power:

$$P_{T1} + P_{T2} + \ldots + (P_{max1} - P_{typical1})^2 + (P_{max2} - P_{typical2})^2 \ldots.$$

where P_{T1} and P_{max1} are the typical and maximum power dissipation of each of the integrated circuits on the board.

10.2.1 Power and Ground Planes

Today's high-speed CMOS logic devices are susceptible to ground noise and the problems this noise creates in digital system design. This noise is a direct result of the fast switching speed and high drive capability of these devices, which are requisites in high-performance systems. Logic designers can use techniques designed to minimize this problem. One technique is to reduce capacitance loading on signal lines and provide optimum power and ground planes.

Power and ground lines have inherent inductance and capacitance, which affect the total impedance of the system. Higher impedances reduce current and therefore offer reduced power consumption, while low impedance (ground plane) minimizes problems such as noise and

electromagnetic interference (EMI). It is very important for a designer to have a controlled impedance design where high speed signals are involved. The formula for impedance is as follows:

$$\text{Impedance} = (L/C)^{1/2}$$

The total characteristic impedance for the power supply can be reduced by adding more lines. The effect of adding more lines to reduce impedance is illustrated in Figure 10-1 which shows that two lines in parallel has half the impedance of a single line. To reduce impedance even further, more lines should be added. To lower the impedance, the number of lines or planes should be increased.

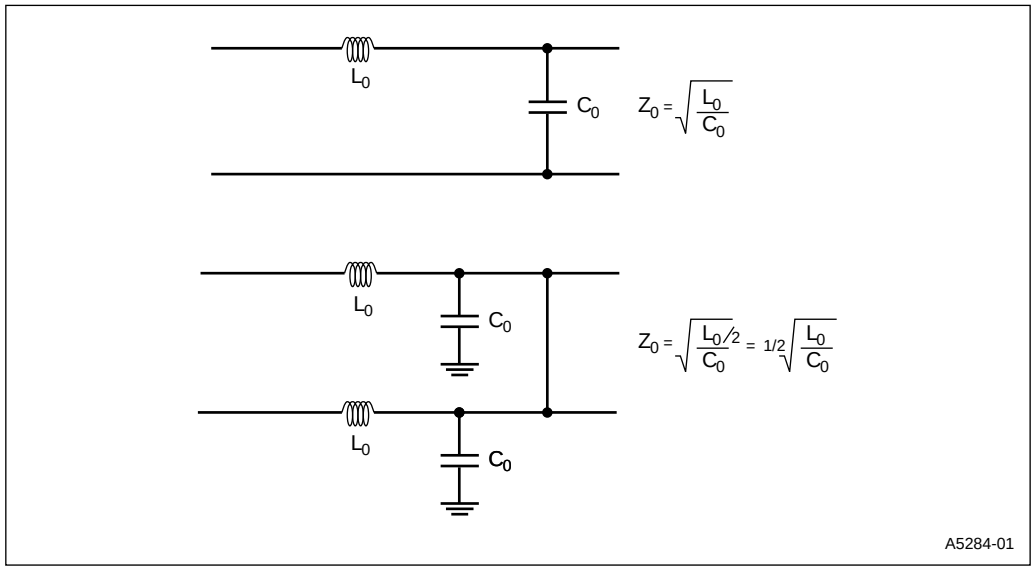


Figure 10-1. Reduction in Impedance

For multi-layer boards, power and ground planes must be used in the Intel486 processor family designs. The ground plane allows best performance at high speeds. It serves two purposes. First it provides a constant characteristic impedance to signal interconnections. Second, it provides a low impedance path for ground currents on the V supply. The advantage of a power plane is to reduce EMI. For example, when adjacent signal lines are switching, EMI may occur. The power plane is used to separate adjacent layers of signal lines, which reduces EMI.

All power and ground pins must be connected to their respective planes. Ideally, the Intel486 processor should be placed at the center of the board to take full advantage of these planes. Although Intel486 processors generally demand less power than conventional devices, the possibility of power surges is increased due to the processor's higher operating frequency and its wide address and data buses. Peak-to-peak noise relative to V should be maintained at no more than 200 mV.

Although power and ground planes are preferable to power and ground traces, double-layer boards present a need for routing of the power and ground traces.

The inductive effect of a printed-circuit board (PCB) trace can be reduced by bypassing. Careful layout procedures should be observed to minimize inductance. Figure 10-2 shows methods for reducing the inductive effects of PCB traces. The power and ground trace layout has a low resistance. This is because the loop area between the integrated circuits (ICs) and the decoupling capacitors is small and the power and ground traces are physically close. This results in lower characteristic impedance, which in turn reduces the line voltage drop.

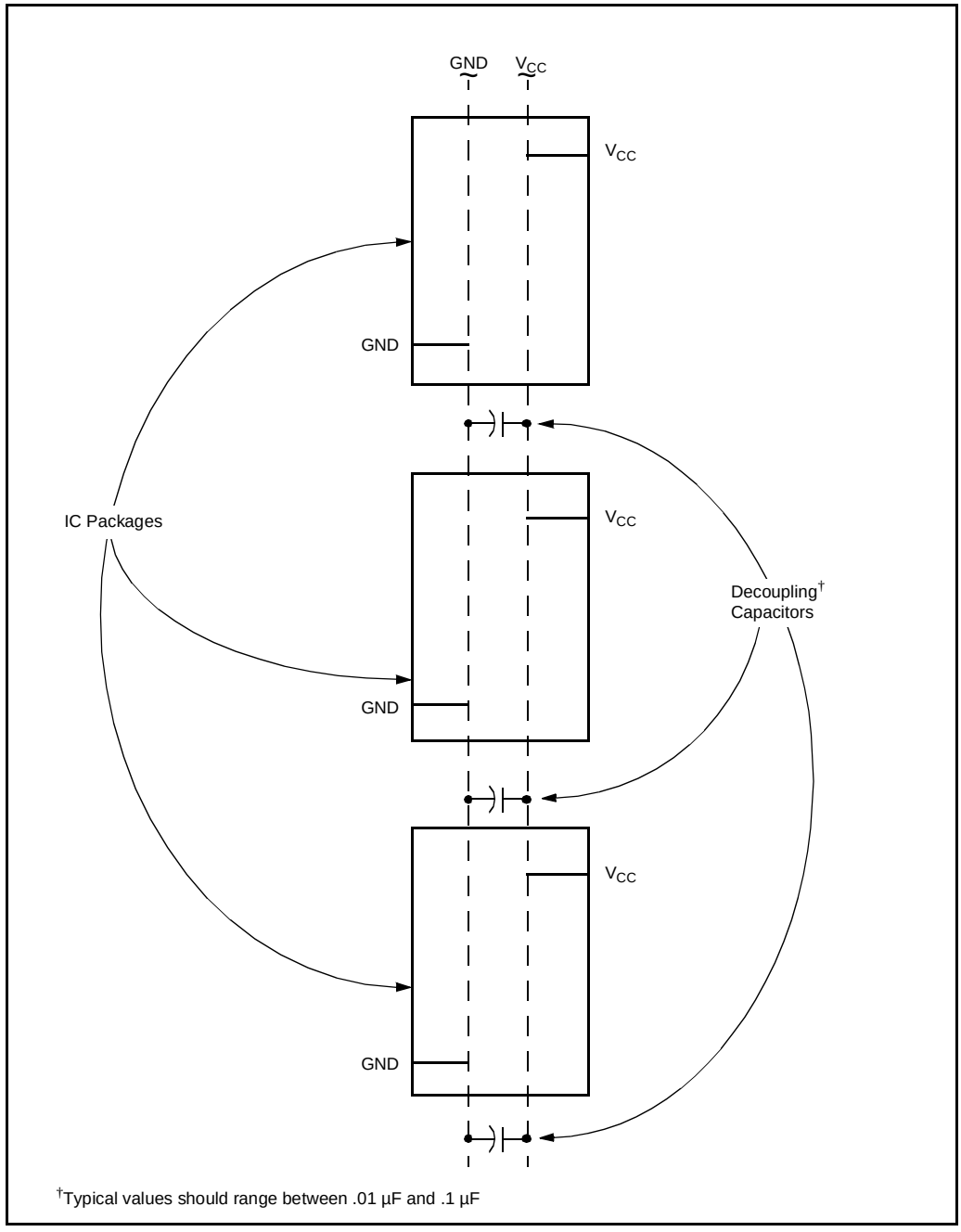


Figure 10-2. Typical Power and Ground Trace Layout for Double-Layer Boards

Another placement technique is called orthogonal arrangement, which requires more area than the previous technique but produces similar results. This arrangement is shown in Figure 10-3. These techniques reduce the electromagnetic interference (EMI), which is discussed in Section 10.3.3.1, "Electromagnetic Interference (EMI)."

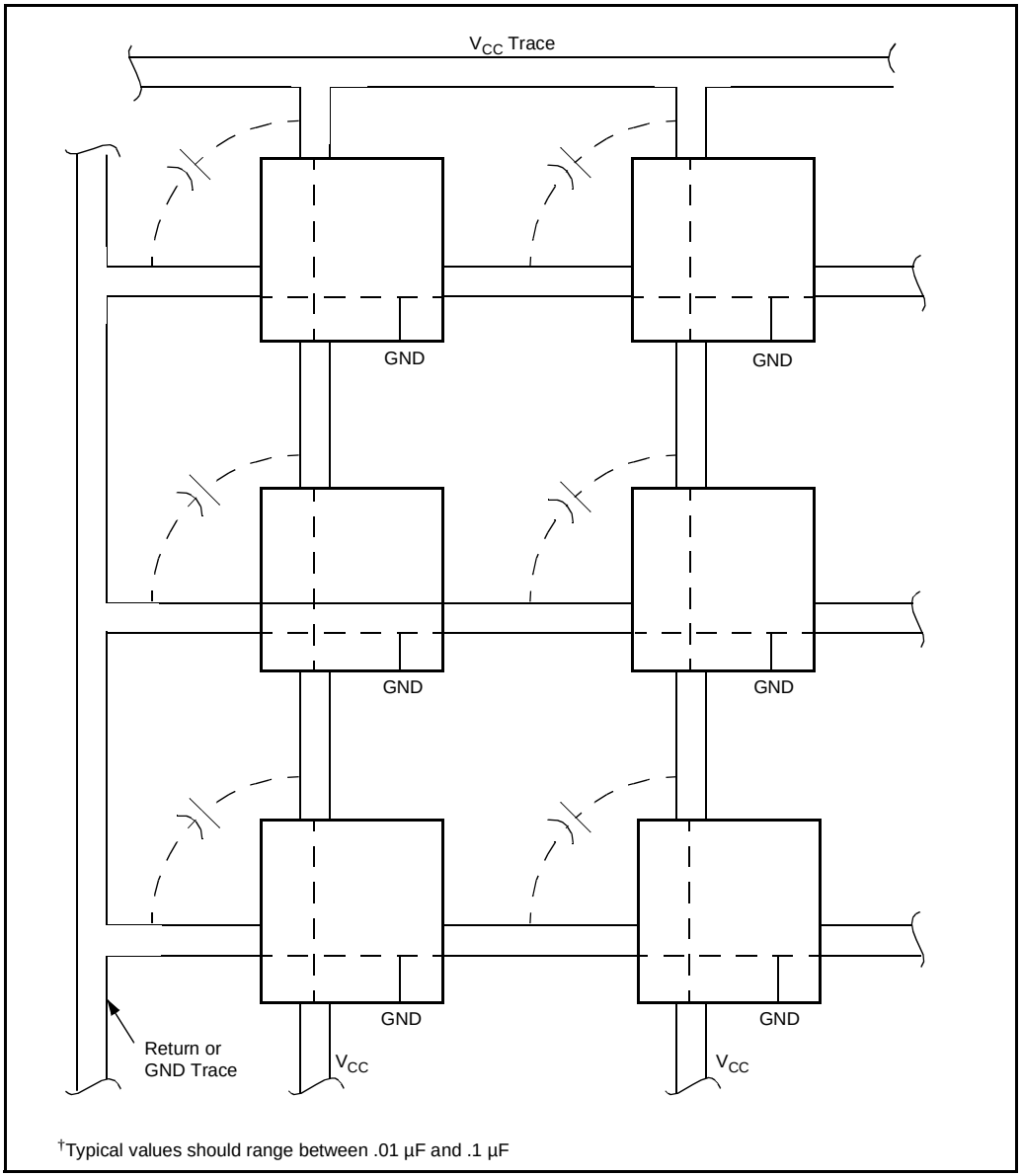


Figure 10-3. Decoupling Capacitors

High-speed CMOS logic families have much higher edge rates than slower logic technologies. The switching speeds and drive capability for high performance also increase noise levels. The switching activity of one device can propagate to other devices through the power supply. For example, in the TTL NAND gate shown in Figure 10-4, both the Q3 and the Q4 transistors are on for a short time while the output is switching. This increased loading causes a negative spike on V_{CC} and a positive spike on V_{SS}.

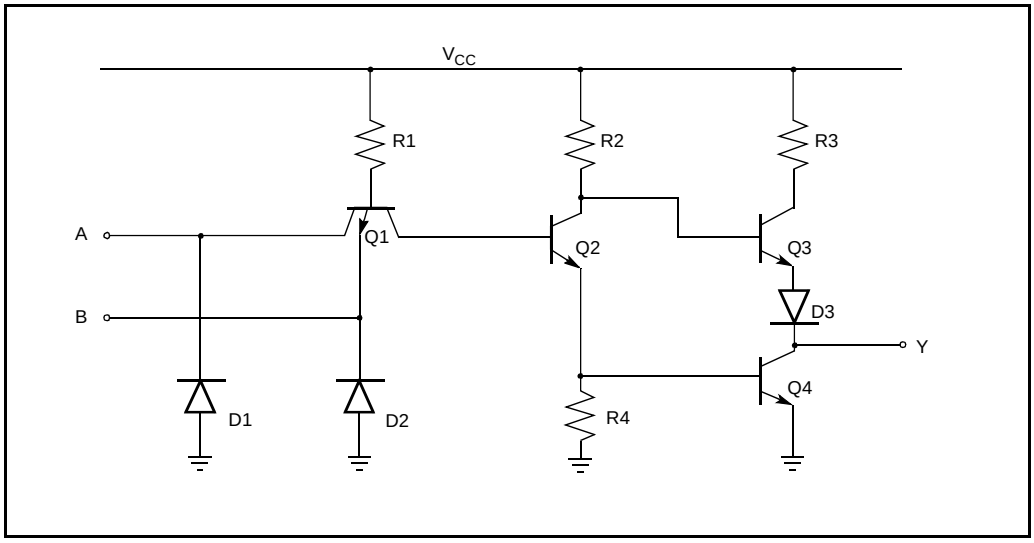


Figure 10-4. Circuit without Decoupling

In synchronous systems where several gates switch simultaneously, the result is a significant amount of noise on the power and ground lines. This noise can be removed by decoupling the power supply. First, it is necessary to match the power supply's impedance to that of the individual components. Any power supply presents a low source impedance to other circuits, whether they are individual components on the same board or other boards in a multi-board system. It is necessary to match the supply's impedance to that of the components in order to lessen the potential for voltage drops that can be caused by IC edge rates, ground- or signal-level shifting, noise induced currents or voltage reflections.

This mismatch can be minimized using suitable high-frequency capacitors for bulk decoupling of major circuitry sections, or for decoupling entire printed circuit boards in multi-board systems. This capacitor is typically placed at the supply's entry point to the board. It should be an aluminum or tantalum-electrolytic type capacitor with a low equivalent series capacitance and low equivalent series inductance. This capacitor's value is typically 10 to 47 µF. Placing several capacitors in parallel provides the lowest effective series resistance (ESR) in the system. Additional 0.1 µF capacitors may be needed if supply noise is still a problem.

Additional decoupling capacitors can be used across the devices between V_{CC} and V_{SS} lines. The voltage spikes that occur due to the switching of gates are reduced since the extra current required

during switching is supplied by the decoupling capacitors. These capacitors should be placed close to their devices, as the inductance of lengthier connection traces reduces their effectiveness.

Most popular logic families require that a capacitor of 0.01 µF to 0.1 µF be placed between every two to five packages, depending on the exact application. For high-speed CMOS logic, a good rule of thumb is to place one of these bypasses between every two ICs, depending on the supply voltage, the operating speed and EMI requirements. The capacitors should be evenly distributed throughout the board to be most effective. In addition, the board should be decoupled from the external supply line with a 10 to 47 µF capacitor. In some cases, it might be helpful to add a 1 µF tantalum capacitor at major supply trace branches, particularly on large PCBs.

Surface mount (chip) capacitors are preferable for decoupling the Intel486 processor because they exhibit lower inductance and require less total board space. They should be connected as shown in Figure 10-5. These capacitors reduce the inductance, which keeps the voltage spikes to a minimum.

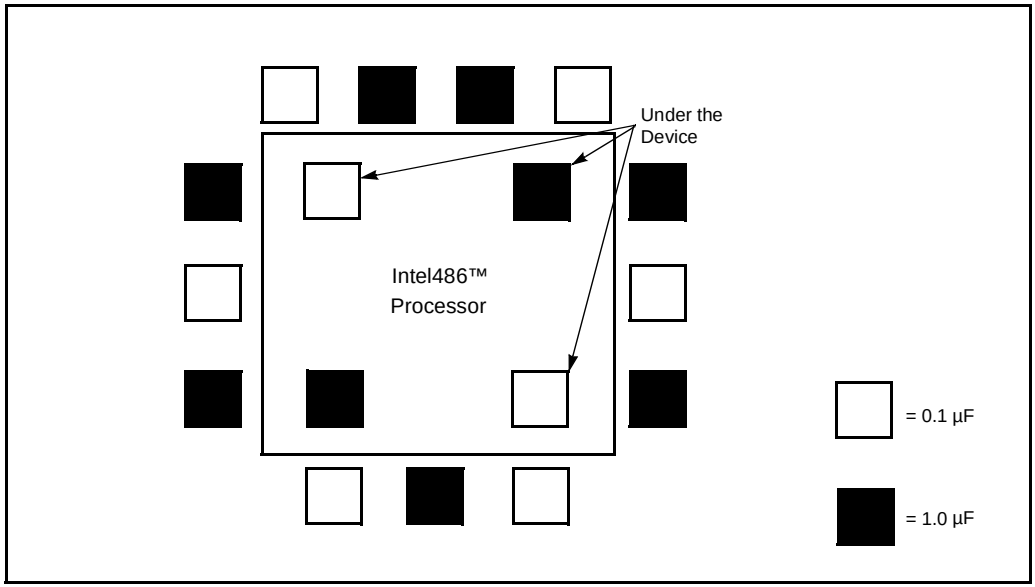


Figure 10-5. Decoupling Chip Capacitors

NOTE

Using Tantalum capacitors allows for smaller capacitance values. Aluminum capacitors in the same applications should be two to five times larger to account for aluminum's higher ESR.

Inductance is also reduced by the parallel inductor relationships of multiple pins. Six leaded capacitors are required to match the effectiveness of one chip capacitor, but because only a limited number can fit around an Intel486 CPU, the configuration shown in Figure 10-6 is recommended.

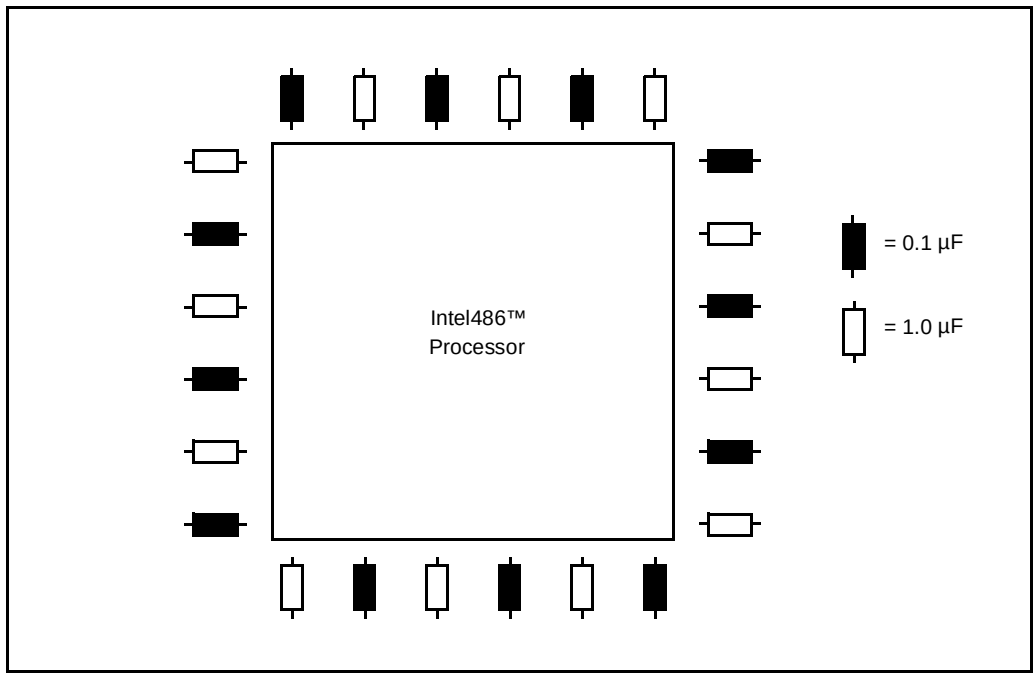


Figure 10-6. Decoupling Leaded Capacitors

10.3 HIGH-FREQUENCY DESIGN CONSIDERATIONS

The overwhelming concern in dealing with high speed technologies is the management of transmission lines. As the edge rates of the signal increase, the physical interconnections between devices behave like transmission lines. Although transmission line theory is straightforward, the difference between ordinary interconnection and transmission line is fairly complex. Transmission lines have distributed elements which are hard to define and designers tend to over-compensate for the effects of these elements.

Efficient Intel486 CPU designs require the identification of the transmission lines over backplane wiring, printed circuit board traces, etc. Once this task is accomplished, the designer's next concern should be to deal with three major problems which are associated with electromagnetic propagation: impedance control, propagation delay, and coupling (electromagnetic interference).

The following sections discuss the negative effects of a transmission line that occur when operating at higher frequencies.

10.3.1 Transmission Line Effects

As a general rule, any interconnection is considered a transmission line when the time required for the signal to travel the length of the interconnection is greater than one-eighth of the signal rise time. The rise time can be either rise time or fall time, whichever is smaller, and it corre-

sponds to the linear ramp amplitude from 0% to 100%. Normally the rise times are specified between 10% to 90% or 20% to 80% amplitude points. The respective values are multiplied by 1.25 or 1.67 to obtain the linear-ramp duration from 0% to 100% amplitude.

For example in a PCB using G-10 and polymide (the two main dielectric systems available for printed circuit boards) signals travel at approximately 5 to 6 inches per nanosecond (ns).

When $T_r/l \times v \geq 8$, the signal path is not a transmission line but it is a lumped element, where:

T_r = rise time 0% - 100%;
V = speed of propagation (5 to 6 inches/sec); and
L = length of interconnection (one-way only).

The calculation is given by:

$T_r/L \times 6 \leq 8$, so

$L \geq (T_r \times 6)/8 \geq (1.25 \times 4 \times 6)/8 \geq 3.75$ inches

This calculation is based on the fact that the maximum rise time of the signals for the Intel486 processor is 4 ns. For L ≥ 3.75 inches, interconnections act as transmission lines.

Every conductor that carries an AC signal and acts as a transmission line has a distributed resistance, an inductance and a capacitance which combine to produce the characteristic impedance (Z). The value of Z depends upon physical attributes such as cross-sectional area, the distance between the conductors and other ground or signal conductors, and the dielectric constant of the material between them. Because the characteristic impedance is reactive, its effect increases with frequency.

10.3.1.1 Transmission Line Types

Although many different types of transmission lines exist, those most commonly used on the printed circuit boards are micro-strip lines, strip lines, printed circuit traces, side-by-side conductors and flat conductors.

10.3.1.2 Micro-Strip Lines

The micro-strip trace consists of a signal plane that is separated from a ground plane by a dielectric as shown in Figure 10-7. G-10 fiberglass epoxy, which is common, has an $e_r = 5$, where:

e_r	is the dielectric constant of the insulation;
w	is the width of signal line (inches);
t	is the thickness of copper (.0015 inches for 1 oz. Cu/.003 inches for 2 oz. Cu);
h	is the height of dielectric for controlled impedance (inches).

The characteristic impedance Z_0, is a function of dielectric constant and the geometry of the board. This is theoretically given by the following formula:

$$Z_0 = [87/\sqrt{(e_r + 1.41)}] \ln (5.98h/.8w + t) \text{ ohms}$$

where e_r is the relative dielectric constant of the board material and h, w, and t are the dimensions of the strip. Knowing the line width, the thickness of Cu and the height of dielectric, the characteristic impedance can be easily calculated.

The propagation delay (t_{pd}) associated with the trace is a function of the dielectric only. This is calculated as follows:

$$t_{pd} = 1.017 \sqrt{(0.475 e_r + 0.67)} \text{ ns/ft}$$

For G-10 fiberglass epoxy boards ($e_r = 5.0$), the propagation delay of micro-strip is calculated to be 1.77 ns/ft.

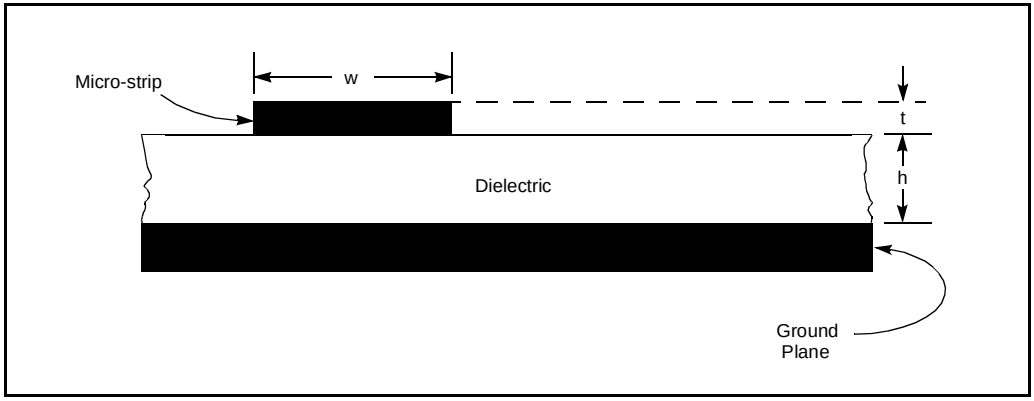


Figure 10-7. Micro-Strip Lines

10.3.1.3 Strip Lines

A strip line is a flat conductor centered in a dielectric medium between two voltage planes. The characteristic impedance is given theoretically by the equation below:

$$Z_0 = [60/\sqrt{e_r}] \ln (5.98b/\pi (0.8w + t)) \text{ ohms},$$

where b = distance between the planes for controlled impedance as shown in Figure 10-8

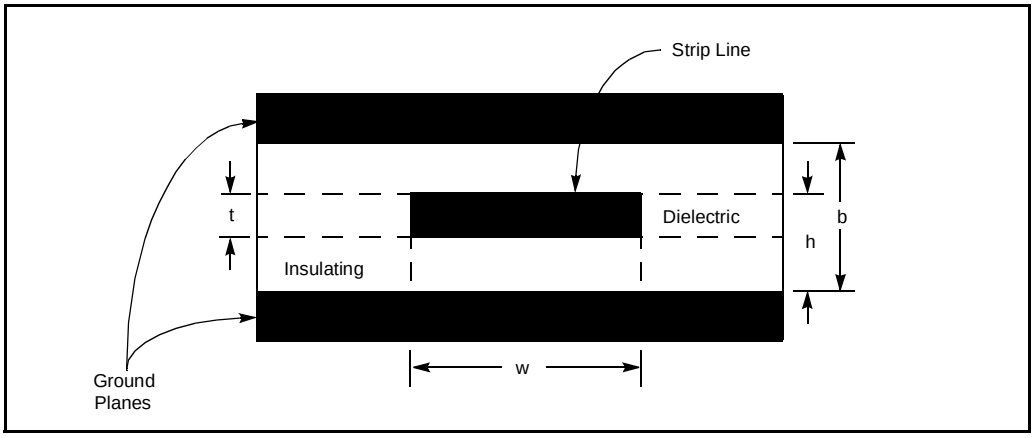


Figure 10-8. Strip Lines

The propagation delay is given by the following formula:

$t_{pd} = 1.017 \sqrt{e_r}$ ns/ft

For G-10 fiberglass epoxy boards ($e_r = 5.0$), the propagation delay of the strip lines is 2.26 ns/ft.

Typical values of the characteristic impedance and propagation delay of these types of lines are as follows:

$Z_0 = 50$ ohms

$t_{pd} = 2$ ns/ft (or 6"/ns)

The three major effects of transmission line phenomenon are impedance mismatch, coupling and skew. The following section discusses them briefly and provide solutions to minimize their effects. For more information on high-frequency design, refer to *High-Speed Digital Design, A Handbook of Black Magic* by Howard W. Johnson and Martin Graham (Publisher: Prentice-Hall Inc.).

10.3.2 Impedance Mismatch

As mentioned earlier, the impedance of a transmission line is a function of the geometry of the line, its distance from the ground plane, and the loads along the line. Any discontinuity in the impedance causes reflections.

Impedance mismatch occurs between the transmission line characteristic impedance and the input or output impedances of the devices that are connected to the line. The result is that the signals are reflected back and forth on the line. These reflections can attenuate or reinforce the signal depending upon the phase relationships. The results of these reflections include overshoot, undershoot, ringing and other undesirable effects.

At lower edge rates, the effects of these reflections are not severe. However at higher edge rates, the rise time of the signal is short with respect to the propagation delay. Thus it can cause problems as shown in Figure 10-9.

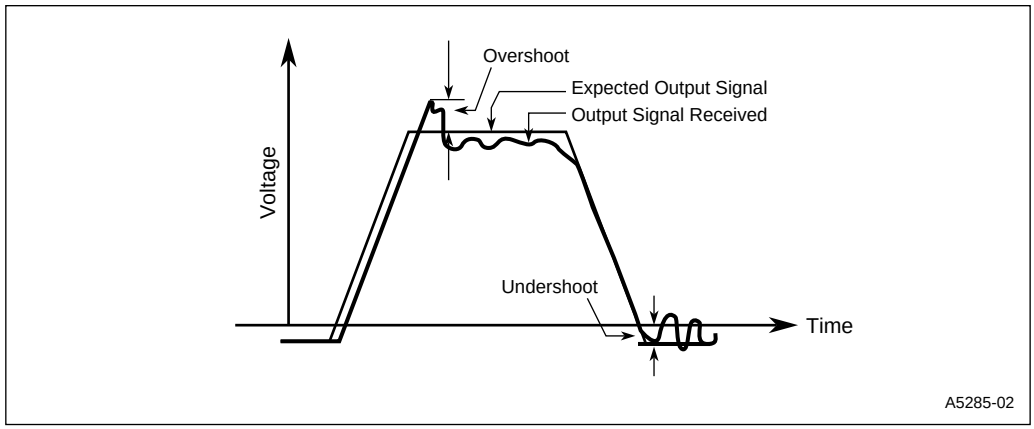


Figure 10-9. Overshoot and Undershoot Effects

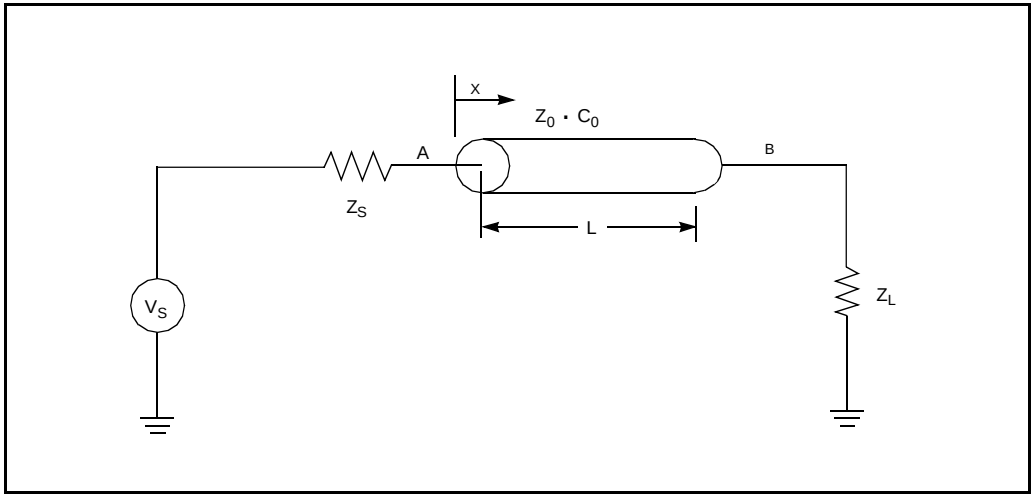


Figure 10-10. Loaded Transmission Line

Overshoot is caused by poor matching, which occurs when the voltage level exceeds the maximum (upper) limit of the output voltage. Undershoot occurs when the level exceeds the minimum (lower) limit. These conditions can cause excess current on the input gates which results in permanent damage to the device.

The amount of reflection voltage can be easily calculated. Figure 10-10 shows a system exhibiting reflections.

The magnitude of a reflection is usually represented in terms of a reflection coefficient. This is illustrated in the following equations:

$t = v_r/v_i$ = Reflected voltage/Incident voltage

$t_L = t_{Load} = (Z_L - Z_0)/ (Z_L + Z_0)$

$t_S = t_{Source} = (Z_S - Z_0)/ (Z_S + Z_0)$

Reflection voltage v_r is given by v_i, the voltage incident at the point of the reflection, and the reflection coefficient.

The model transmission line can now be completed. In Figure 10-10, the voltage seen at point A is given by the following equation:

$$V_A = V_S * Z_0/(Z_0 + Z_S)$$

This voltage V_A enters the transmission line at "A" and appears at "B" delayed by t_{pd}.

$$V_B = V_A(t - x/v)\ H(t - x/v)$$

where x = distance along the transmission line from point "A" and H(t) is the unit step function. The waveform encounters the load Z_L, and this may cause reflection. The reflected wave enters the transmission line at "B" and appears at point "A" after time delay (t_{pd}):

$$V_{r1} = t_L \cdot V_B$$

This phenomenon continues infinitely, but it is negligible after 3 or 4 reflections. Hence:

$$V_{r2} = t_S \cdot V_{r1}$$

Each reflected waveform is treated as a separate source that is independent of the reflection coefficient at that point and the incident waveform. Thus the waveform from any point and on the transmission line and at any given time is as follows:

$$\begin{aligned} V(x,t) = \; & Z_0/(Z_0 + Z_S)\ \{\ [V_S(t-x/v)H(t-x/v)] \\ & t_L\ [V_S(t-(2L-x)/v)]\ [H(t-(2L-x)/v)] \\ & t_L\ t_S\ [V_S(t-(2L+x)/v)]\ [H(t-(2L+x)/v)] \\ & t_1^2\ t_S\ [V_S(t-(4L-x)/v)]\ [H(t-(4L-x)/v)] \\ & t_1^2\ t_S^2\ [V_S(t-(4L+x)/v)]\ [H(t(4L+x)/v)] \\ & +.....\} \end{aligned}$$

Each reflection is added to the total voltage through the unit step function H(t). The above equation can be rewritten as follows:

$V(x,t) = Z_0/(Z_0+Z_S)\ \{\ [V_S(t–t_{pd}x)\ H(t_{pd}–tx)]$
$+\ t_L\ [V_S(t–t_{pd}\ (2L–x))\ H[t–t_{pd}(2L–x))]$
$+\ t_L\ T_S\ [V_S\ (t–t_{pd}(2L+x))H(t–t_{pd}(2L+x))]$
$+.....\}$

This can be further explained by an example.

Let: $V_S = \sin(2\pi\ 10^9\ t)$
$Z_S = 35$ ohms
$Z_L = 20$ ohms
$Z_0 = 50$ ohms

$L = 14$ in
$x = 6$ inches
$t_{pd} = 2$ ns/ft = .17 ns/in

$v = [2$ ns/ft) = .5 ft/ns = 6 in/ns
$t_L = (20 – 50)/(20 + 50) = .43$
$t_S = .18$
at t = .5 ns

$V(x,t) = V(6$ in, .5 ns)
$= 50/(50+35)\{[\sin\ (2\pi 10^9(0.5–0.17\text{ns/in}(6\text{in}))\text{ns})\}$
$+\ (–0.43)\ \{\sin\ (2\pi 10^9(0.5–0.17(6))\text{ns})H(0.5–0.17(6))\}$
$= .59\ \{\sin\ (-1.04\pi) +0\}$ at t = .5 ns

Voltage at A with the transmission line properties accounted for. There is no reflection yet.

$V(x,t) = V(6$ in, 5 ns)
$= [50/50 + 35]\ \{\sin[2\pi\ 10^9\ (5 – (.17)(6)]$
$+\ (–.43)\ \{\sin\ [2\pi\ 10^9\ (5 – .17\ (28 – 6))]\ H\ [5 – .17\ (28 – 6)]\}$
$+\ (–.43)(\ –.18)\ \{\sin\ [2\pi\ 10\ (5 – 17\ (28 + 6))]\ H\ [5 –.17(28 + 6)]\}$
$= .59\ \{\sin\ (–1.04\ \pi) –.43\ \sin\ (2.52\ \pi) + .08\ \sin\ (–1.56\ \pi)\}$

The lattice diagram is a convenient visual tool for calculating the total voltage due to reflections as described in the previous equations. Two vertical lines are drawn to represent points A and B on the horizontal dimension, x. The vertical dimension represents time.

A waveform travels back and forth between points A and B of the transmission line in time, producing the lattice diagram shown in Figure 10-11. The voltage at a given point is the sum of all the individual reflected voltages up to that time. Notice that at each endpoint, two waves are converging, the incident wave and the reflected wave. Therefore, the voltage at the end points A and B at the time of the waveform reflection are calculated by summing both the incident and reflected waves up to and including the point in question.

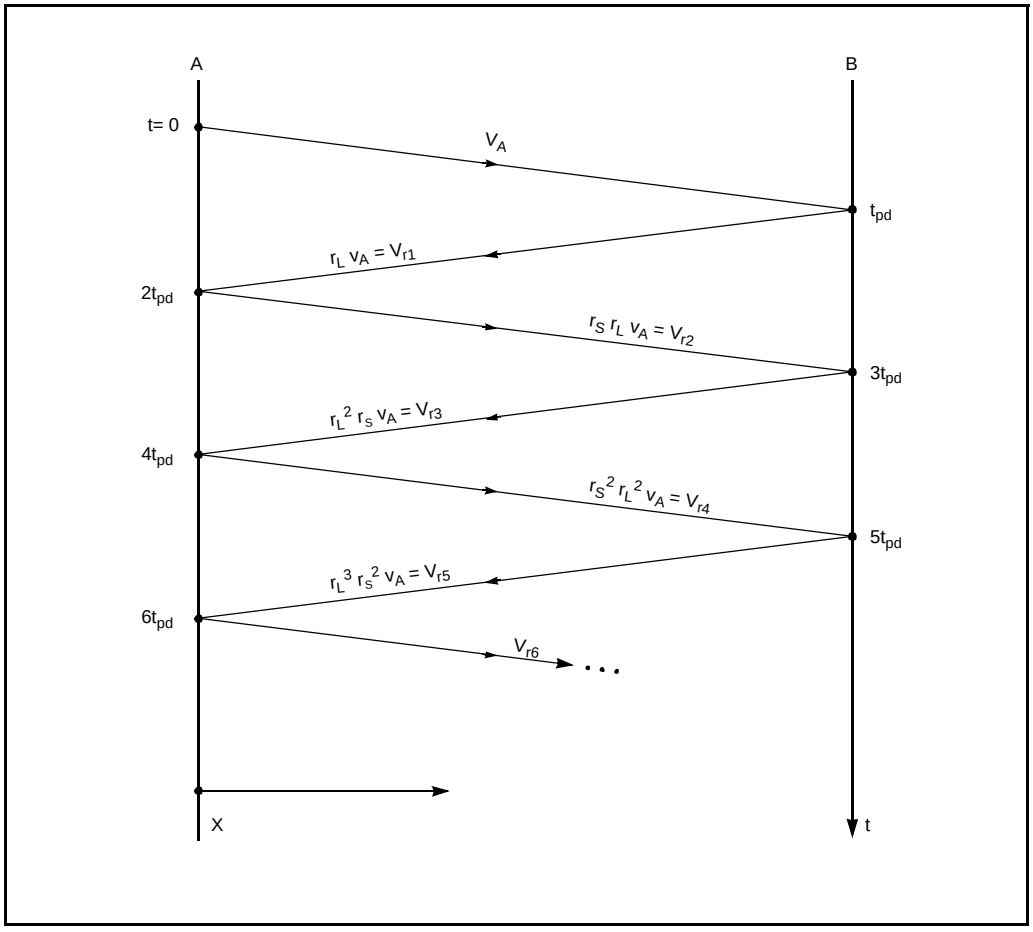


Figure 10-11. Lattice Diagram

As an example, let the simple configuration shown in Figure 10-10 be assumed. Assume the following:

V_S = 3.70 H(t)v
Z_0 = 75 ohms
Z_S = 30 ohms
Z_L = 100 ohms

The appropriate reflection coefficients can be calculated as follows:

source = (30–75)/(30+75)= 0.42857
load = (100–75)/(100+75)= 0.14286

$V_a = V_S \cdot \{75/(75+30)\} = 2.64286$ V
$V_{r1} = 2.64286 \times 0.14286 = 0.37755$ V
$V_{r2} = 0.37755 \times -0.42875 = -0.16181$ V
$V_{r3} = -0.16181 \times 0.14286 = -0.02312$ V
$V_{r4} = -0.02312 \times -0.42857 = 0.00991$ V
$V_{r5} = 0.00991 \times 0.14286 = 0.00142$ V
$V_{r6} = 0.00142 \times -0.42857 = -0.00061$ V
$V_{r7} = -0.00061 \times 0.14286 = -0.00009$ V

Figure 10-12 shows the corresponding lattice diagram.

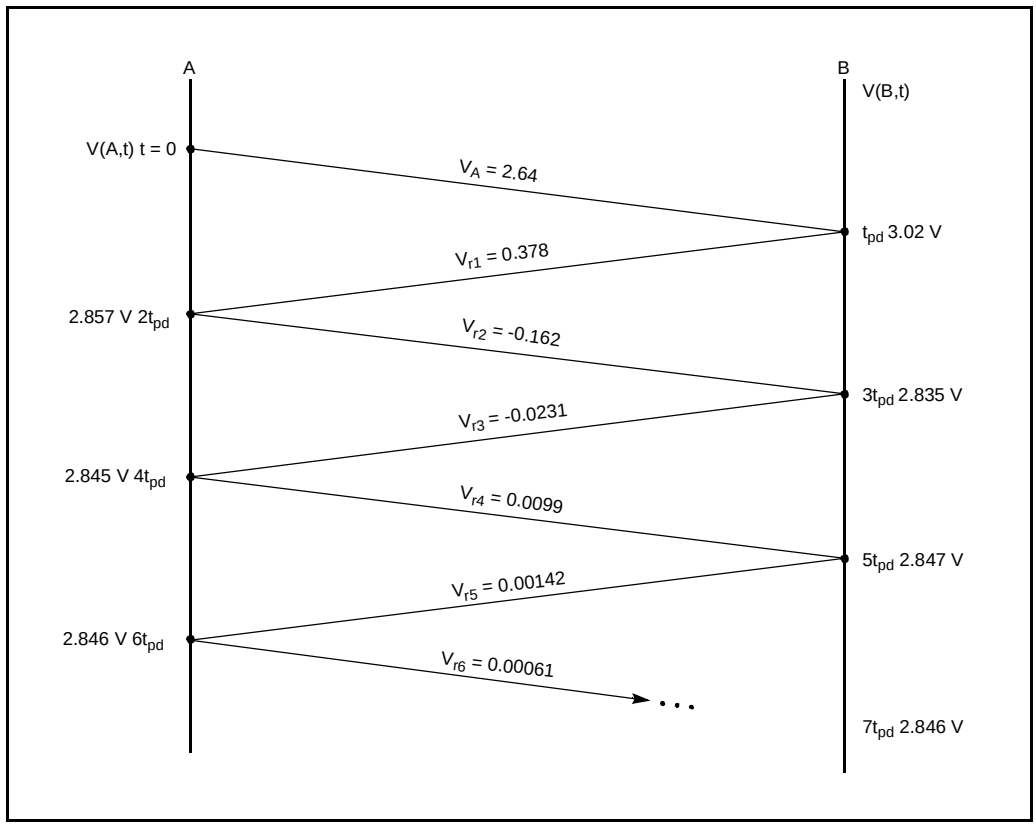


Figure 10-12. Lattice Diagram Example

Impedance discontinuity problems are managed by imposing limits and control during the routing phase of the design. Design rules must be observed to control trace geometry, including specification of the trace width and spacing for each layer. This is very important because it ensures the traces are smooth and constant without sharp turns.

There are several techniques which can be employed to further minimize the effects caused by an impedance mismatch during the layout process:

1. Impedance matching.
2. Daisy chaining.
3. Avoidance of 90° corners.
4. Minimization of the number of vias.

10.3.2.1 Impedance Matching

Impedance matching is the process of matching the impedance of the source or load with that of the trace and it is accomplished with a technique called termination. The reflection, overshoot and undershoot of signals are reduced by terminating the remote end of the transmission line from the source. The terminating impedance combines with the destination input circuitry to produce a load that closely matches the characteristic impedance of the line (board traces have characteristic impedances in the range of 30 ohms to 200 ohms).

The calculation of characteristic impedance was already discussed. Impedance of the printed circuit board backplane connectors have the impedance in the same range as the traces (i.e., 30–200 ohms).

Depending upon the length of the conductors or when using twisted pairs of coaxial cable in place of printed circuit traces, the characteristic impedance of a backplane may change. Backplane impedance is also affected by the number of boards plugged into the backplane.

Need for Termination

The transmission line should be terminated when the t_{pd} exceeds one-third of t_r (risetime). If the $t_{pd} \geq 1/3\ t_r$ (rise time), the line can be left un-terminated, provided the capacitive coupling between the traces does not cause electromagnetic interference.

Termination thus eliminates impedance mismatches, increases noise immunity, suppresses RFI/EMI and helps to ensure that signals reach their destination with minimum distortion. There are five methods for terminating traces on the board:

1. Series
2. Parallel
3. Thevenin
4. AC
5. Active

Terminations usually cost money, because they require additional components and power. In the case of passive terminations, extra drivers are needed to deliver more current to the line. In case of active terminations extra power is needed, which increases the power dissipation of the system.

Series Termination

One way of controlling ringing on longer lines is with the series termination technique also known as damping. This is accomplished by placing a resistor in series with the transmission line

at the driving device end. The receiver has no termination. The value of the impedance looking into the driving device ($R_{driver} + R_{line} = Z_0$) should approximate the impedance of the line as closely as possible. In this circuit the ringing dampens out when the reflection coefficient goes to zero. Figure 10-13 illustrates the series termination.

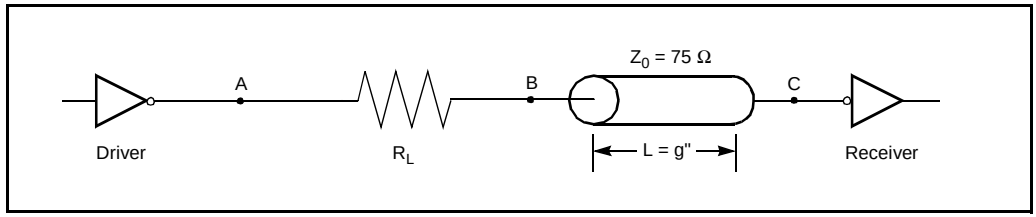


Figure 10-13. Series Termination

One main advantage of series termination is that only logic power dissipation results so that lower overall power is required. There is one penalty, however, in that the distributed loading along the transmission line cannot be used because only half of the voltage waveform is travelling down the line. There is no limit on the number of loads that can be placed at the end of the series terminated connection. However, the drop in voltage across a series terminating resistor limits loading to maximum 10.

Parallel Terminated Lines

Parallel termination is achieved by placing a resistor of an appropriate value between the input of the loading device and the ground as shown in Figure 10-14. To determine an appropriate value, the currents required by all inputs and the leakage currents of the drivers are summed. A resistor should be selected so that its addition to the circuit does not exceed the output capacity of the weakest driver. For the type of termination shown in Figure 10-14, only high logic levels need to be calculated.

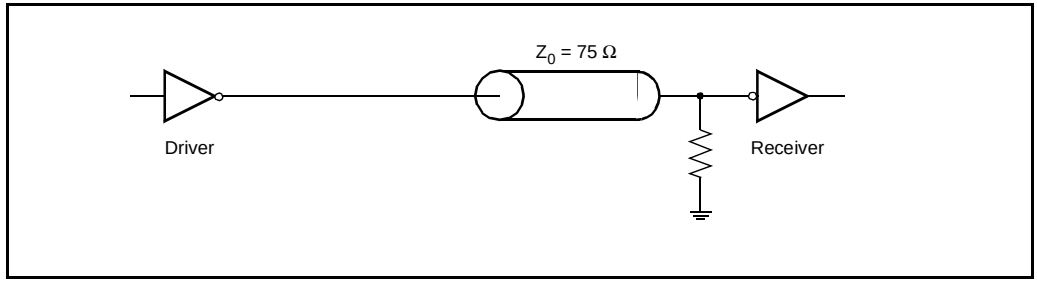


Figure 10-14. Parallel Termination

Since the input impedance of the device is high compared to the characteristic line impedance, the resistor and the line function as a single impedance with a magnitude that is defined by the value of the resistor.

When the resistor matches the line impedance, the reflection coefficient at the load approaches zero, and no reflection occurs. One useful approach is to place the termination as close to the loading device as possible.

Parallel terminated lines are used to achieve optimum circuit performance and to drive distributed loads, which is an important benefit of using parallel terminations.

There are two significant advantages of using the parallel termination. First, it provides an undistributed waveform along the entire line. Second, when a long line is loaded in parallel termination, it does not affect the rise and fall time or the propagation delay of the driving device. Note that parallel termination can also be used with wire wrap and backplane wiring where the characteristic impedance is not exactly defined. If the designer approximates the characteristic impedance, the reflection coefficient is very small. This results in minimum overshoot and ringing. Parallel termination is not recommended for characteristic impedances of less than 100 ohms because of large DC current requirements.

Thevenin's Equivalent Termination

This technique is an extension of parallel termination technique. It consists of connecting one resistor from the line to the ground and another from the line to the V_{CC}. Each resistor has a value of twice the characteristic impedance of the line, so the equivalent resistance matches the line impedance. This scheme is shown in Figure 10-15.

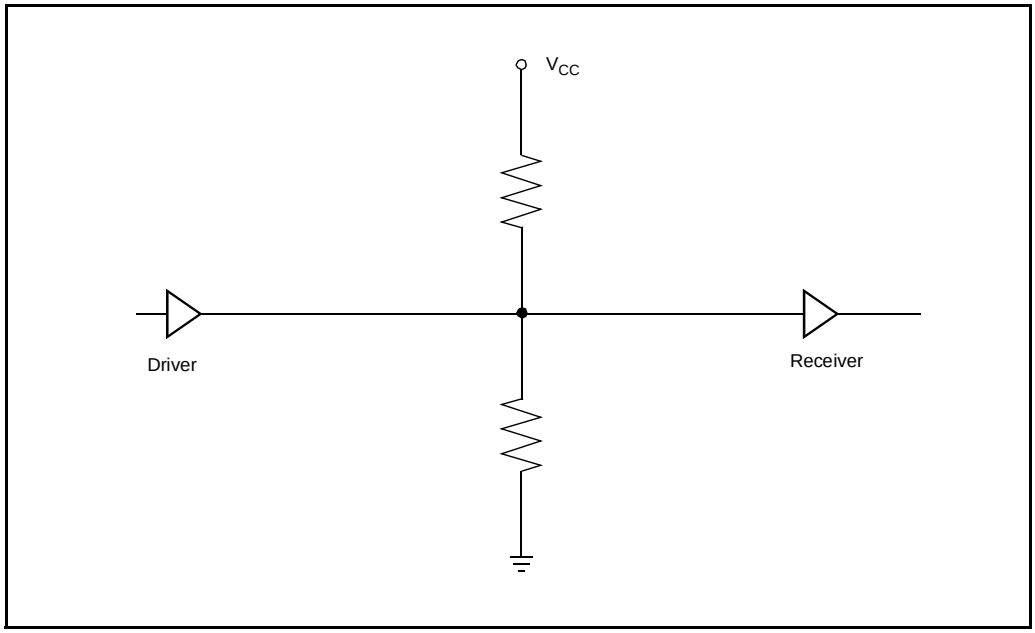


Figure 10-15. Thevenin's Equivalent Circuit

If there were no logic devices present, the line would be placed halfway between the V_{CC} and the V_{SS}. When the logic device is driving the line, a portion of the required current is provided by the

resistors, so the drivers can supply less current than needed in parallel termination. The resistor value can be adjusted to bias the lines towards the V_{CC} or V_{SS}. Ordinarily it is adjusted such that the two are equal, providing balanced performance. The Thevenin's circuit provides good overshoot suppression and noise immunity.

Due to power dissipation, this technique is best suited for bipolar and mix MOS devices and is not suitable for pure CMOS implementations. The reasons for not having Thevenin's equivalent for the pure CMOS system design are as follows:

CMOS circuits have very high impedance to both ground and V_{CC} and their switching threshold is 50% of the supply voltage. Besides dissipating more power, multiple input crossing may occur creating output oscillations.

The main problem is high power dissipation through the termination resistors in relationship to the total power consumption of all of the CMOS devices on the board. Most designers prefer series terminations for CMOS to CMOS connections, because as this does not introduce any additional impedance from the signal to the ground. The main advantage of the series termination technique, apart from its reduced power consumption, is its flexibility. The received signal amplitude can be adjusted to match the switching threshold of the receiver simply by changing the value of the terminating resistor. This is a very useful technique for interconnecting the logic devices with long lines.

AC Termination

AC termination is another technique which can be used for designs which cannot tolerate high power dissipation of parallel termination and delays created by series termination. It consists of a resistor and a capacitor connected in series from the line to the ground. It is similar to the parallel termination technique in functionality except that the capacitor blocks the DC component of the signal and thus reduces the power dissipation. This technique is shown in Figure 10-16.

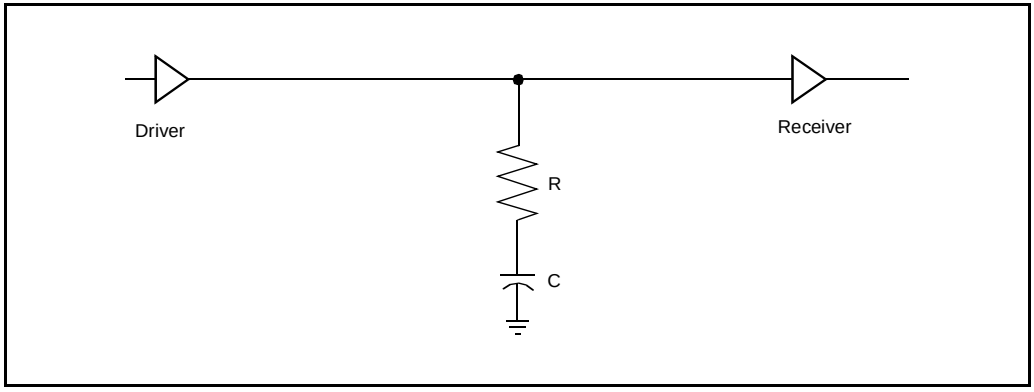


Figure 10-16. AC Termination

The main disadvantage of this technique is that it requires two components. Further the optimum value of the RC time constant of the termination network is not easy to calculate. It usually begins as a resistive value which is slightly larger than the characteristic line impedance. It is critical to

determine the capacitor value. If the value of RC time constant is small, the RC circuit acts as an edge generator and creates overshoot and undershoot. Increasing the capacitor value reduces the overshoot and undershoot, but it increases power consumption. As a rule of thumb, the RC time constant should be greater than twice the delay line. The power dissipation of the AC termination is a function of the frequency.

Active Termination

These terminations consist of resistors that are connected between the inputs and outputs of a buffer driver as shown in Figure 10-17.

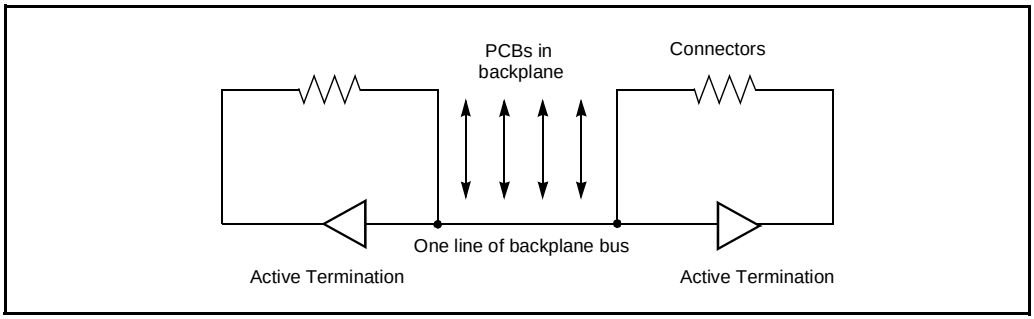


Figure 10-17. Active Termination

The main advantage of this technique is that it can tolerate large impedance variations and this tolerance is valuable when three-state drivers are connected to backplane buses. However, the terminations are costly, and the signals that are produced are not as clean as other terminations. A common solution is to place active terminations at both ends of the bus. This helps to maintain the uniform drive levels along the entire length of the bus, and it reduces EMI and ringing.

Table 10-1 shows the comparisons of different termination techniques.

Table 10-1. Comparison of Various Termination Techniques

Termination	# of Extra Components	R_L Power Consumption		Prop Delay
Series	1	$Z_0 - Z_{OUT}$	Low	Yes
Parallel	1	Z_0	High	No
Thevenin	2	$2Z_0$	High	No
AC†	2	$2Z_0$	Medium	No
Active	1	$2Z_0$	Medium	No

Beyond matching impedances, there are other techniques that can help avoid reflections. These are discussed in the following sections.

Impedance Matching Example

We have already discussed the techniques for calculating characteristic impedances (using transmission line theory) and the termination procedures used to avoid impedance mismatching. This section describes an impedance matching example that utilizes these techniques. Figure 10-18 shows a simple interconnection which acts like a transmission line, as shown by the calculations.

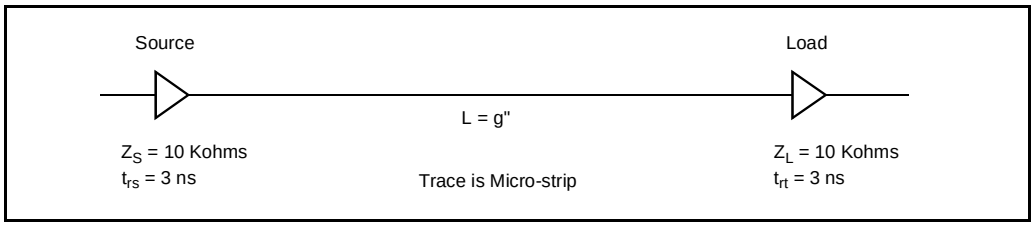


Figure 10-18. Impedance Mismatch Example

In this example the different values are given as follows:

Z_S = source impedance = 10 ohms
t_{rs} = source rise-time = 3 ns (normalized to 0% to 100%)
Z_L = load impedance = 10 Kohms
t_{rl} = load rise-time = 3 ns (normalized to 0% to 100%)
L = length of interconnection = 9"
trace = micro-strip
e = dielectric constant = 5.0
H = .008"
W = .01"
T = .0015" Cu (1 oz. Cu) thickness
v = 6"/ns

The interconnection acts as a transmission line if (as was shown in Section 10.3.1, “Transmission Line Effects”).

$l \geq (tr \times v) / 8 \geq (3 \times 6)/8 \geq 3"$.

The value of l = 9", thus the interconnection acts like a transmission line.

The impedance of the transmission line is calculated as follows:

$Z_0 = 87 / \sqrt{e_r + 1.41} \times \ln(5.98H/(.8W + T))$
$= 34.39 \ln 5.05 = 55.6$ ohms

Because Z_S = 10 ohms, the termination techniques described previously are needed to match the difference of 45.6 ohms. One method is to use a series terminating resistor of 45.6 ohms or use AC termination where r = 55.6 ohms and c = 300 pF. The terminated circuit of Figure 10-18 is shown in Figure 10-19.

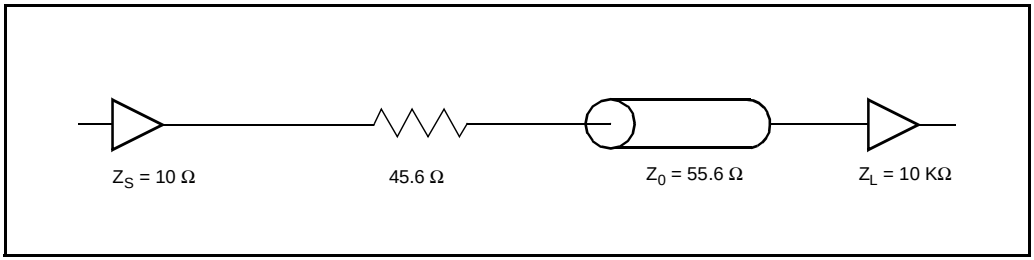


Figure 10-19. Use of Series Termination to Avoid Impedance Mismatch

10.3.2.2 Daisy Chaining

In laying out printed circuit boards, a stub or T-connection is another source of signal reflection. These types of connections act as inductive loads in the signal path. In daisy chaining, a single trace is run from the source, and the loads are distributed along this trace. This is shown in Figure 10-20.

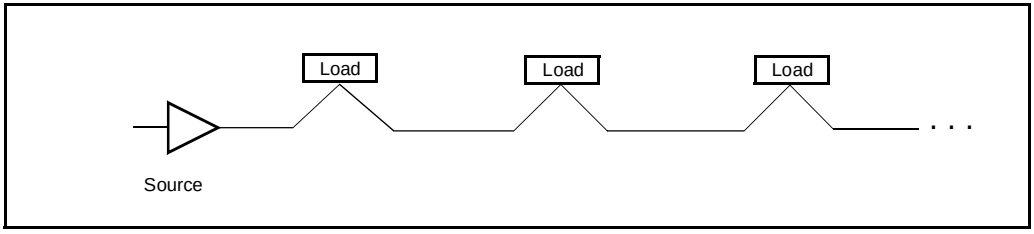


Figure 10-20. "Daisy" Chaining

An alternative to this technique is to run multiple traces from the source to each load. Each trace has unique reflections. These reflections are then transmitted down other traces when they return to the source. In such cases a separate termination is required for each branch. To eliminate these T-connections, high-frequency designs are routed as daisy chains.

Along the chain, each gate provides its own impedance load; thus it is necessary to distribute these loads evenly along the length of the chain. Hence, the impedance along the chain changes in a series of steps and it is easier to match. The overall speed of this line is faster and predictable. Also, all loads should be placed at equal distances (regular intervals).

10.3.2.3 90-Degree Angles

Another major cause of reflections are 90-degree angles in the signal paths, which cause an abrupt change in the signal direction. It promotes signal reflection. For high-frequency layout of designs, avoid 90-degree trace angles and use 45- or 135-degree trace angles as shown in Figure 10-21.

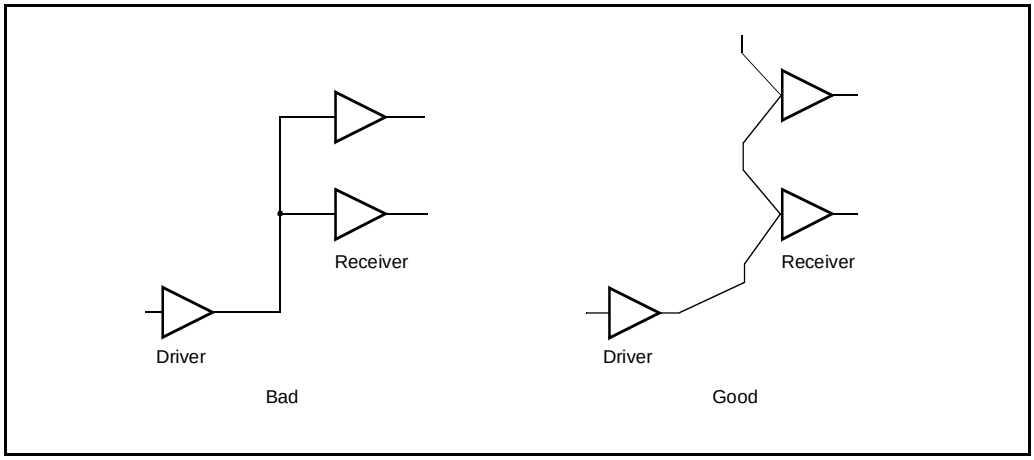


Figure 10-21. Avoiding 90-Degree Angles

10.3.2.4 Vias (Feed-Through Connections)

Another impedance source that degrades high-frequency circuit performance is vias. Expert layout techniques can eliminate vias to avoid reflection sites on PCBs.

10.3.3 Interference

We have discussed reflections in high-frequency design, their causes and techniques to minimize them. The following sections discuss additional issues related to high-frequency design, including interference. In general, interference occurs when electrical activity in one conductor causes transient voltage to appear in another conductor. Two main factors increase the interference in any circuit:

1. Variation of current and voltage in the lines causes frequency interference. This interference increases with the frequency.
2. Coupling occurs when conductors are in close proximity.

Two types of interference are observed in high-frequency circuits:

1. Electromagnetic Interference (EMI)
2. Electrostatic Interference (ESI)

10.3.3.1 Electromagnetic Interference (EMI)

Electromagnetic Interference (EMI) is a problem at high operating frequencies: when operating frequency increases, signal wavelength becomes comparable to the lengths of some of the interconnections on the printed circuit board. EMI is a phenomenon of a signal in one trace which induces another similar signal in an adjacent trace. There are two types of coupling between parallel traces which determine the amount of EMI in a circuit. These are called the inductive coupling and the radiative coupling.

Inductive coupling occurs when a current in one trace produces current in a parallel trace. This current reduces with the distance from the source. Hence, closely spaced wires or traces incur the greatest degree of inductive coupling. Both traces in this case act like normal conductors.

Radiative coupling occurs when two parallel traces act as a dipole antenna which radiates signals that parallel wires can pick up. This results in the corruption of signal that is already present in the trace. The intensity of this type of coupling is directly proportional to the current present in the trace. However, it is inversely proportional to the distance between the radiating source and the receiver.

10.3.3.2 Minimizing Electromagnetic Interference

When laying out a board for an Intel486 processor-based system, several guidelines should be followed to minimize EMI.

One source of EMI is the presence of a common impedance path. Figure 10-22 shows a typical layout which does not have the same earth ground or the signal ground.

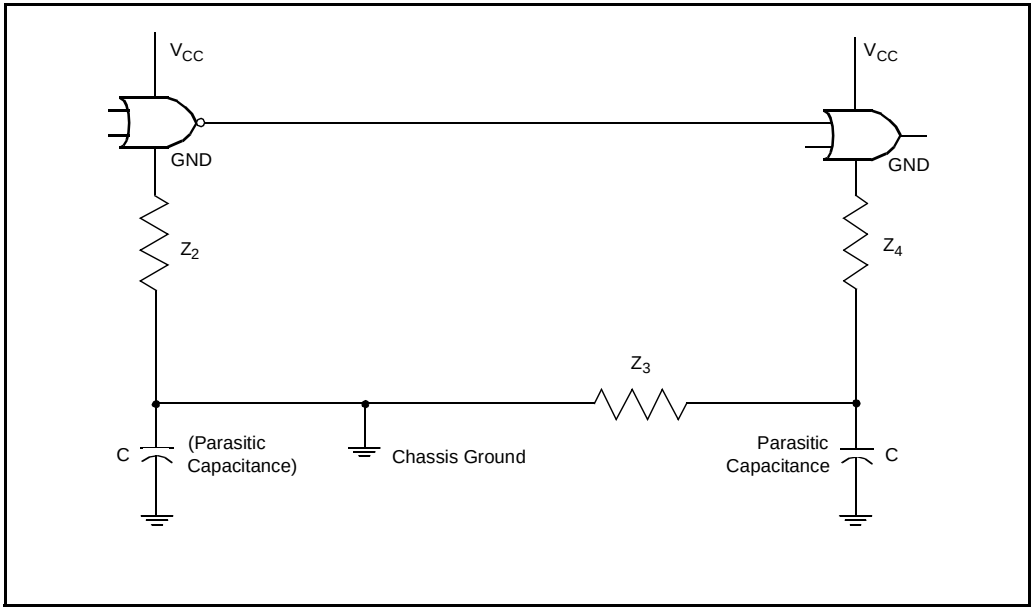


Figure 10-22. Typical Layout

To reduce EMI, it is necessary to minimize the common impedance paths, which are Z_2, Z_3 and Z_4 shown primarily as ground impedances. During current switching, the ground line voltage drops, causing noise emission. By enlarging the ground conductor (which reduces its effective impedance), this noise can be minimized. This technique also provides a secondary advantage in that it forms a shield which reduces the emissions of other circuit traces, particularly in multi-layer circuit boards.

The impedances Z_2 through Z_4 depend upon thickness of copper printed circuit board foil, the circuit switching speeds and the effective lengths of the traces. The current flowing through these common impedance paths radiates more noise as its value increases. The amount of voltage generated by these switching currents and multiplied by the impedance is difficult to predict.

An effective way to reduce EMI is to decouple the power supply by adding bypass capacitors between V_{CC} and Ground. This technique is similar to the general technique discussed earlier. (The goal of the previous technique was to maintain correct logic levels.)

The design of effective coupling and bypass schemes centers on maximizing the charge stored in the circuit bypass loops while minimizing the inductances in these loops. Some other precautions that can minimize the EMI are as follows:

- Running a ground line between two adjacent lines. The lines should be grounded at both ends.
- The address and data busses can be separated by a ground line. This technique may be expensive due to large number of address and data lines.
- Removing closed loop signal paths, which create inductive noise as shown in Figure 10-23.

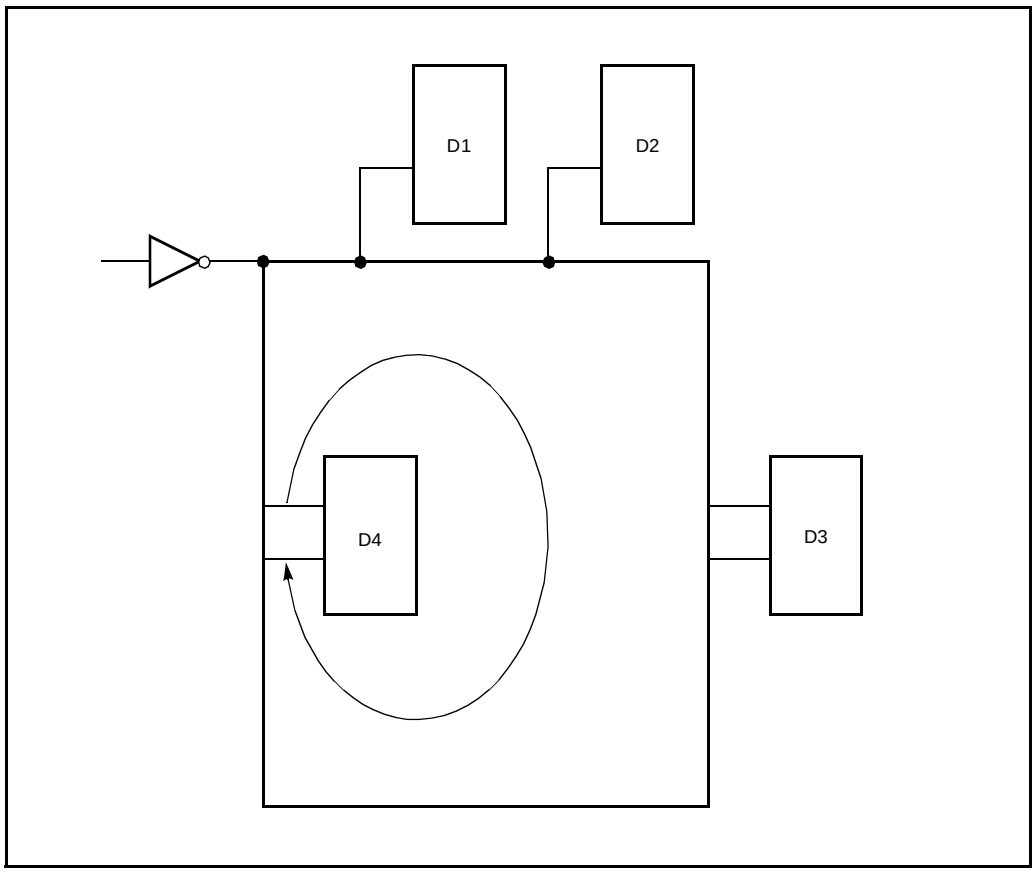


Figure 10-23. Removing Closed Loop Signal Paths

Minimizing EMI involves first examining the circuit's interconnection with its nearest neighbors since parallel and adjacent lines can interact and cause EMI. It is necessary to maximize the distance between adjacent parallel wires.

10.3.3.3 Electrostatic Interference

We have discussed two types of coupling, namely inductive and radiative coupling which are responsible for creating electromagnetic interference. A third, known as capacitive coupling, occurs when two parallel traces are separated by a dielectric and act as a capacitor. According to the standard capacitor equation, the electric field between the two capacitor surfaces varies with the permittivity of the dielectric and with the area of the parallel conductors.

Electrostatic interference (ESI) is caused by this type of coupling. The charge built on one plate of the capacitor induces opposite charge on the other. To minimize the ESI, the following steps should be taken.

- Separate the signal lines so that the effect of capacitive coupling is negated.
- Run a ground line between the two lines to cancel the electrostatic fields.

For high-frequency designs, a rule of thumb is to include ground planes under each signal layer. Ground planes limit the EMI caused by a capacitive coupling between small sections of adjacent layers that are at equipotentials. Additionally, when the width and the thickness of signal lines and their distance from the ground is constant, the effect of capacitive coupling upon impedance remains uniform within approximately ±5 percent across the board. Using fixed impedance does not reduce capacitive coupling, but it does simplify the modeling of propagation delays and coupling effects. In addition, capacitive coupling can cause interference between layers, so the wires should be routed orthogonally on neighboring board layers.

10.3.4 Propagation Delay

The propagation delay of a circuit is a function of the loads on the line, the impedance, and the line segments. The term propagation delay means the signal rise time delay in the entire circuit, including the delay in the transmission line (which is a function of the dielectric constant).

Also, the printed circuit interconnection adds to the propagation delay of every signal on the wire. These interconnections not only decrease the operating speed of the circuits, but also cause reflection, which produces undershoot and overshoot.

When the propagation delays in the circuit are significant, the design must compensate for the signal skew. Signal skew occurs when the wire lengths (and thus the propagation delays) between each source and each corresponding load are unequal.

Another negative aspect of propagation delay is that it causes a generation of race condition. This condition occurs when two signals must reach the same destination within one clock pulse of one another. To avoid race conditions, it is necessary to have the signals travel through the same length traces. But if one route is shorter, the signals arrive at different timings, causing race conditions.

One way to minimize this is by decreasing the length of the interconnections. Overall route lengths are shorter in multi-layer printed circuit boards than in double-layer boards because ground and power traces are not present. In addition to adding ground planes, a routing program can help to shorten the routing paths.

The guidelines discussed thus far are prominent at the higher operating frequencies. Debugging an Intel486 processor-based system at higher frequencies requires careful layout of the physical design. This section also covers latch-up and thermal characteristics which are system design considerations that stem from the device itself.

10.4 LATCH-UP

"Latch-up" is triggered when the voltage limits on the I/O pins are exceeded, causing the internal PN junction to become forward-biased. The following steps ensure the prevention of latch-up.

- Observe the maximum input voltage rating of I/O pins.
- Never apply power to an Intel486 processor pin or to any device connected to it before applying power to the Intel486 processor.
- Use good termination techniques to prevent overshoot and undershoot.
- Ensure proper layout to minimize reflections and to reduce noise on the signals.

10.5 CLOCK CONSIDERATIONS

For best performance, the clock signal (CLK) for the Intel486 CPU must be free of noise and within the specifications listed in the individual Intel486 datasheets. The transmission line effects must also be considered for the clock paths. These paths should be suitably terminated to minimize signal reflections and prevent overshoot and undershoot.

Skew is an effect of unequal transmission line length and matching. This is very important in a synchronous system. Long traces add propagation delay. A longer trace or a load placed further down a trace experiences more delay than a short trace or loads very close to the source. This must be taken into account when doing the worst case timing analysis. In a system where events must occur synchronous to a clock signal, it is important to make sure the signal is available to all inputs a sufficient amount of time prior to the corresponding clock edge. When performing the component placement this is one of the considerations that must be accounted for.

To maintain proper logic levels, all digital signal outputs have a maximum load, they are capable of driving. DC loading is the constant current required by an input in either the high or the low state. It limits the ability of a device driving the bus to maintain proper logic levels. For an Intel486 processor-based system, a careful analysis must be performed to ensure that in a worst case situation no loading limits are exceeded. Even if a bus is loaded slightly beyond its worst case limit, problems may result if a batch of parts whose input loading is close to maximum is encountered. The proper logic level may not be maintained and unreliable operation may result. Marginal loading problems are particularly troublesome, since the effect is often erratic operation and non-repetitive errors that are difficult to track down. For both the high and low logic levels, the sum of the currents required by all the inputs and the leakage currents of all outputs (drivers) on the bus must be added together. This sum must be less than the output capability of the weakest driver. Since the Intel486 processor is a CHMOS device having negligible DC loading, the main contributors to D.C. loading are the TTL devices.

The AC or capacitive loading is caused by the input capacitance of each device and limits the speed at which a device driving a bus signal can change the state from high to low or low to high.

For high-frequency designs, the component and system margins are no longer available to the designer. With less than 1 ns of timing margin, even the small amount of trace capacitance can make a circuit path critical.

A more accurate calculation of capacitive loading can be derived by modeling the device loads and system traces as a series of Transmission Lines Theory. Transmission Line Theory provides

a more accurate picture of system loading in high-frequency systems. In addition, it allows new factors to be considered, such as inductance and the effect of reflections upon the quality of the signal waveform.

10.5.1 Requirements

The Intel486 processor facilitates an easy-to-implement 1x clock interface. An external, TTL-compatible 25/33 MHz clock synchronizes both the internal functional blocks of the microprocessors and the external signals. Most of the Intel486 processor's board logic circuitry also uses this clock.

The clock input requirements for Intel486 processor systems are more stringent than those for many commonly used TTL devices, however. The specifications are -0.3 Volts to 0.8 volts for a logic low and 2.0 volts to V_{CC} plus 0.3 volts for a logic high.

The minimum high and low times are specified as 11 ns at 25 MHz and 5 ns at 33 MHz. The typical clock timings are shown in Figure 10-24.

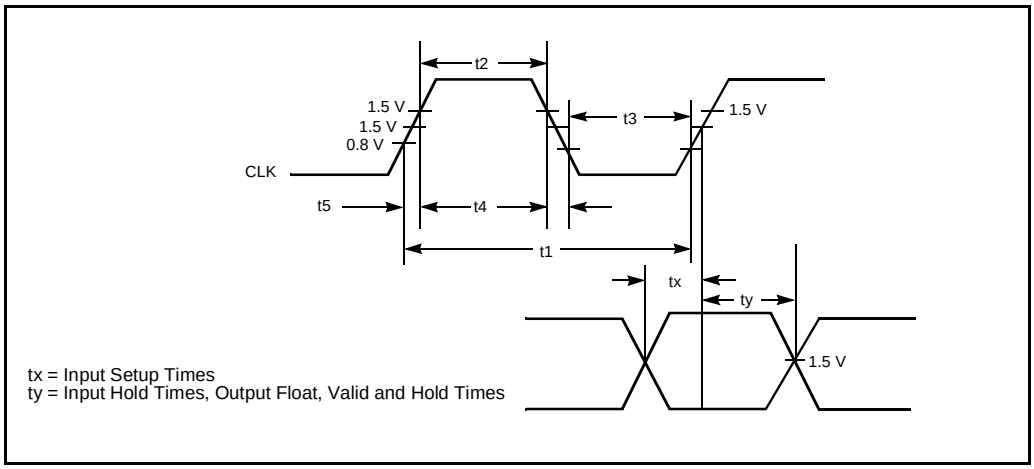


Figure 10-24. Typical Clock Timings

10.5.2 Routing

Achieving the proper clock routing around a 25/33 MHz (or higher) printed circuit board is delicate because problems can arise if certain design guidelines are not followed. For example fast clock edges cause reflections from high impedance terminations. These reflections can cause significant signal degradations in the systems operating at 25/33 MHz clock rates. This section covers some design guidelines for properly laying out the clock lines for efficient Intel486 processor operation.

Since the rise/fall time of the clock signal is typically in the range of 2-4 ns, the reflections at this speed could result in undesirable noise and unacceptable signal degradation. The degree of reflection depends on the impedance of the trace of the clock connection. These reflections can be

optimized by using proper terminations and by keeping the length of the traces as short as possible. The preferred method is to connect all of the loads via a single trace as shown in Figure 10-25, thus avoiding the extra stubs associated with each load. The loads should be as close to one another as possible. Multiple clock sources should be used for distributed loads.

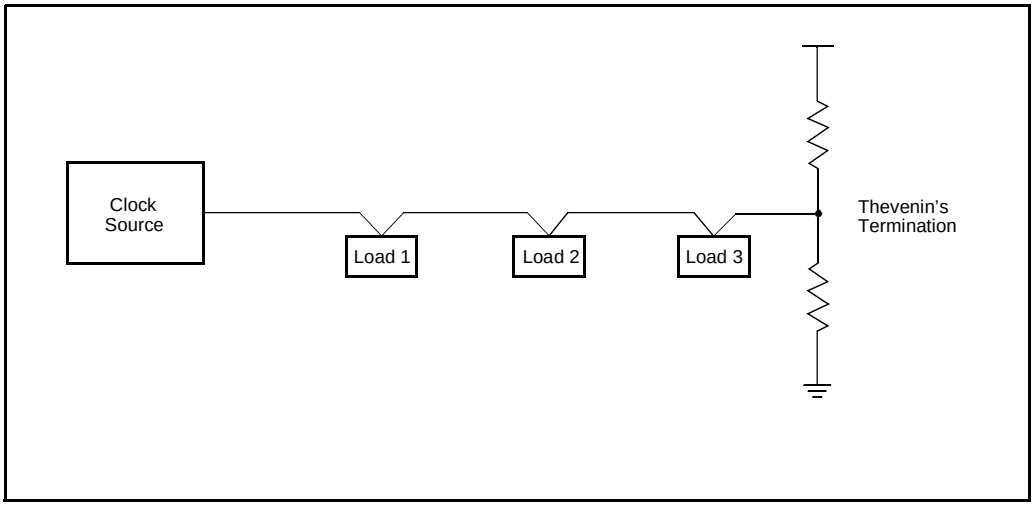


Figure 10-25. Clock Routing

A less desirable method is the star connection layout in which the clock traces branch to the load as closely as possible (Figure 10-26). In this layout, the stubs should be kept as short as possible. The maximum allowable length of the traces depends upon the frequency and the total fanout, but the length of all of the traces in the star connection should be equal. Lengths of less than one inch are recommended.

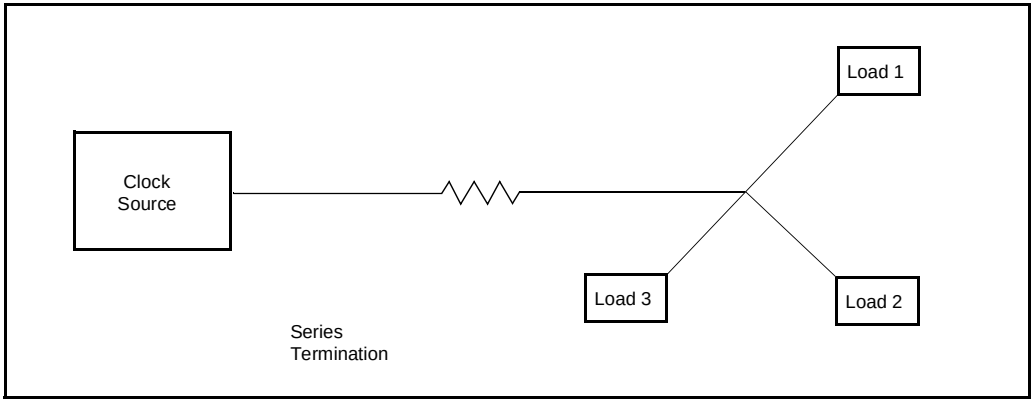


Figure 10-26. Star Connection

10.6 THERMAL CHARACTERISTICS

There are thermal and electrical limitations associated with all the operating electronic devices. In an Intel486 processor-based system, these limitations must be accommodated to achieve proper system performance due to power dissipation concerns.

Generally, thermal and electrical characteristics are interrelated, and actual constraints depend upon the application of a particular device.

To help the user, most of the general information on case temperature (T_C), maximum current and voltage ratings, maximum thermal resistance (θ) at various airflows, and package thermal specifications are given in the individual Intel486 processor datasheets. Despite the wealth of information presented in the datasheet, it is impossible to provide graphs and reference tables to cover all applications. The designer must accurately calculate several factors such as junction temperature (T_j) and total power dissipation (P_d) in particular applications.

This section explains how to perform these calculations, thereby making designing with the Intel486 processor more straightforward.

The thermal specifications for the Intel486 processor are designed to ensure a tolerable temperature at the surface of the Intel486 chip. This temperature, called Junction Temperature (T_j), can be determined from external measurements using the known thermal characteristics of the package.

The following two equations facilitate the calculation of the Junction Temperature (T_j):

Let T_j = junction temperature
T_a = Ambient temperature
T_c = Case temperature
θ_{jc} = Junction to Ambient temperature co-efficient
θ_{ja} = Junction to Ambient temperature co-efficient
P_d = Power Dissipation (worst case $P_d = I_{CC} * V_{CC}$)

Then:

$$T_j = T_a + (\theta * P_d)$$

and

$$T_j = T_c + (\theta * P_d)$$

Given a heat sink with a thermal resistance of θ_{sa} (sink to ambient), and given the thermal resistance from the junction to the case θ_{jc}, then the equation for calculating T_j is as follows:

$$T_j = P_d(\theta_{jc} + \theta_{cs} + \theta_{sa}) + T_a$$

Case temperature calculations offer many advantages over ambient temperature calculations:

- Case temperature is more easily measured compared to ambient temperature because the measurement is localized to a single point (the center of the package).

- The worst case junction temperature (T_j) is lower when calculated with case temperature for two reasons. First, the junction-to-case thermal coefficient (θ_{jc}) is lower than the junction-to-ambient thermal coefficient (θ_{ja}). Therefore, the calculated junction temperature varies less with power dissipation (P_d). Second, the junction-to-case coefficient (θ_{jc}) is not affected by the airflow in the system, whereas the junction-to-ambient coefficient (θ_{ja}) does vary.

Given the case temperature specification, a designer can either set the ambient temperature or use fans to control the case temperature. Finned heatsinks or conductive cooling may also be used in an environment which prohibits the use of fans.

A designer has considerable freedom in designing the heatsink and faces only practical and economic limits. Multiple parallel devices may be helpful in reducing θ_{sa} because if the heat input to the heat sink is dispersed rather than concentrated, the effective thermal impedance is lower.

To approximate the case temperature for varying environments, the two equations discussed earlier should be combined by making the junction temperature the same for both, resulting in the following equation:

$$T_a = T_c - [(\theta_{ja}\ \theta_{jc})\ P_d]$$

Refer to the Intel486 processor datasheets to determine the values of θ_{ja} (per the system's airflow requirement) and the ambient temperature that will yield the desired case temperature. The proper calculations are important in achieving an efficient and reliable Intel486 processor system.

One packaging option for the Intel486 processors is a 168-pin ceramic PGA. The recommended heatsinks for the device are offered in the pin fin design that utilizes air cooling. T_a is greatly improved by adding a heat sink. The heat sink is mounted on the PGA package with a frame and spring. A typical heat sink is shown in Figure 10-27. The dimensions are shown in Figure 10-28.

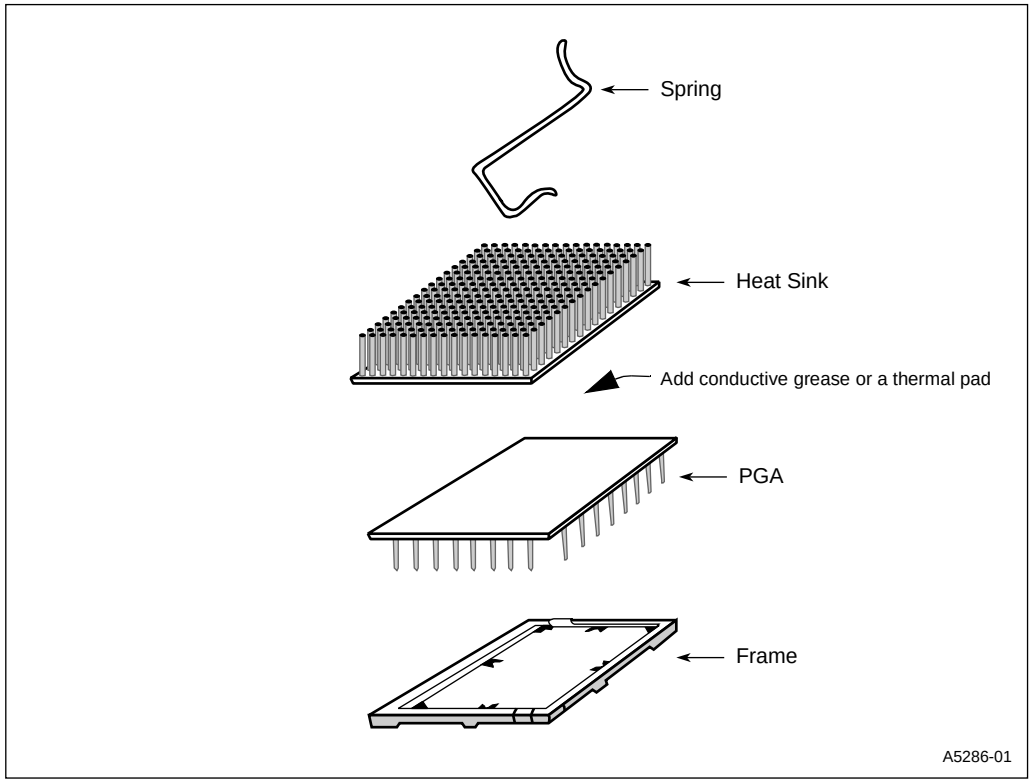


Figure 10-27. Typical Heat Sinks

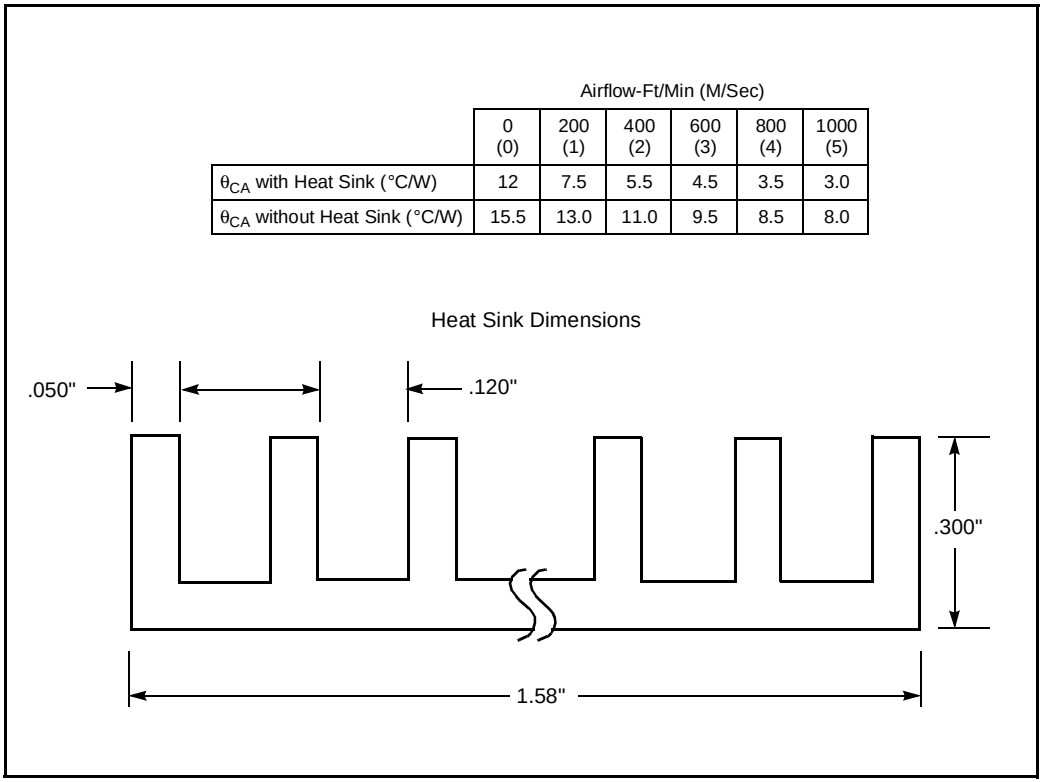


Figure 10-28. Heat Sink Dimensions

10.7 DERATING CURVE AND ITS EFFECTS

A derating curve is a graph that plots the output buffer delay against the capacitive load. The curve is used to analyze a signal delay without necessitating a simulation every time the processor's loading changes. This graph assumes the lumped-sum capacitance model to calculate the total capacitance. The delay in the graph should be added to the specified AC timing value for the device that is driving the load. The derating curve is device-dependent because each device has different output buffers.

A derating curve is generated by tying the chip's output buffers to a range of capacitors. The voltage and resistance values chosen for the output buffers are at the highest specified temperature and are rising (worst case) values. The value of the capacitors centers around the AC timing values for the chip. For 25 MHz and above, this is 50 pF. Since the AC timing specifications are measured for a signal reaching 1.5 V, the output buffer delay is the time that it takes for a signal to rise from 0 V to 1.5 V. A curve is then drawn from the range of time and capacitance values, with 50 pF representing the average, with nominal or zero derating. These curves are valid only for a 25 pF–150 pF load range. Beyond this range the output buffers are not well characterized. The derating curves for the Intel486 processor are shown in Figure 10-29. These curves use the

lumped capacitance model for circuit capacitance measurements and must be modified slightly when doing worst-case calculations that involve transmission line effects. The amount of modifications required can be calculated by performing SPICE simulation or by using other simulation packages.

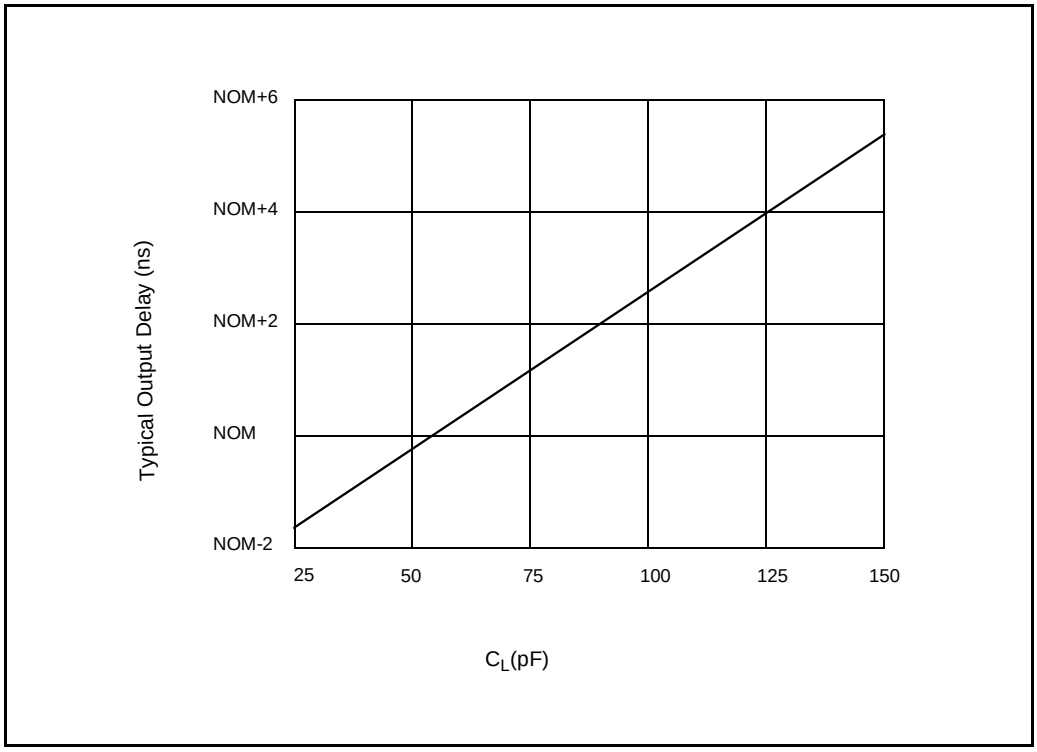


Figure 10-29. Derating Curves for the Intel486™ Processor

10.8 BUILDING AND DEBUGGING THE Intel486™ PROCESSOR-BASED SYSTEM

Although an Intel486 processor-based system designer should plan the entire system, it is necessary to begin building different elements of the core and begin testing them before building the final system. If a printed circuit board layout has to be done, the whole system may be simulated before generating the net list for the layout vendor. It is advisable to work with a preliminary layout to avoid the problems associated with wire wrap boards that operate at high frequencies. A typical Intel486 processor-based system is shown in Figure 10-30.

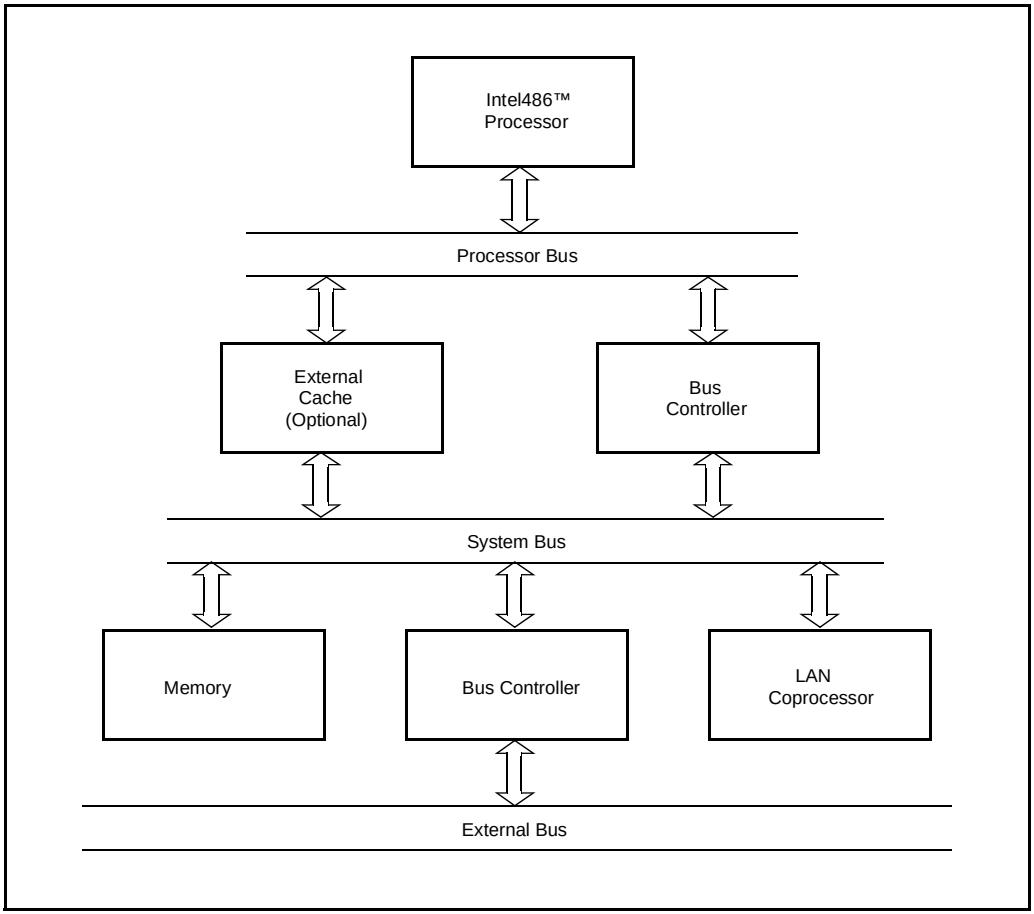


Figure 10-30. Typical Intel486™ Processor-Based System

An optional second-level cache can also be added to the system. The following steps are usually carried out in designing with the Intel486 processor.

1. Clock circuitry should consist of an oscillator and fast buffer. The CLK signal should be clean, without any overshoot or undershoot.
2. The reset circuitry should be designed as shown in Chapter 4, “Bus Operation.” This circuitry is used to generate the RESET # signal for the Intel486 processor. The system should be checked during reset for all of the timings. The clock continues to run during these tests.
3. The INT and HOLD pins should be held low (deasserted). The READY# pin is held high to add additional delays (wait states) to the first cycle. At this instance, the Intel486 processor is reset, and the signals emitted from it are checked for the validity of the state.

The Intel486 processor starts executing instructions at location FFFFFFF0H after reset. The address latch is connected and the address is verified.

4. The PAL implementing the address decoder should be connected to the Intel486 processor.

10.8.1 Debugging Features of the Intel486™ Processor

The Intel486 processor provides several features which simplify the debugging process for the system hardware designer. The device offers three on-chip debugging aids:

- The code execution breakpoint opcode.
- The single-step capability provided by the TF bit in the flag register.
- The code and data breakpoint capability as provided by the debug registers (DR3–DR0, DR6 and DR7).

10.8.2 Breakpoint Instruction

The Intel486 processor provides a breakpoint instruction that can be used by software debuggers. This instruction is a single byte opcode and generates an exception 3 trap when it is executed. In a typical environment a debugger program can place the breakpoint instruction at various points in the program. The single-byte breakpoint opcode is an alias for the two-byte general software interrupt instruction, INT*n* where *n*=3. The only difference between INT 3 and INT n is that INT3 is never IOPL-sensitive but INT*n* is IOPL-sensitive in Protected Mode and Virtual 8086 Mode.

10.8.3 Single-Step Trap

The Intel486 processor supports x86-compatible single-step feature. If the single stepflag bit (bit 8, TF) is set to 1 in the EFLAG register, a single step exception occurs. This exception is auto-vectored to exception 1 and occurs immediately after completion of the next instruction. Typically a debugger sets the TF bit of the EFLAG register on the debugger's stack followed by transfer of the control to the user program. The debugger also loads the flag image (EFLAG) via the IRET instruction. The single-step trap occurs after execution of one instruction of the user program.

Since the exception 1 occurs right after the execution of the instruction as a trap, the CS:EIP pushed onto the debugger's stack points to the next unexecuted instruction of the program which is being debugged, merely by ending with an IRET instruction.

After MOV to SS and POP to SS instructions, the Intel486 processor masks some exceptions, including single-step trap exceptions. Refer to the “Exceptions and Interrupts” chapter in the *Intel486™ Processor Family Programmer’s Reference Manual* for an explanation of this process.

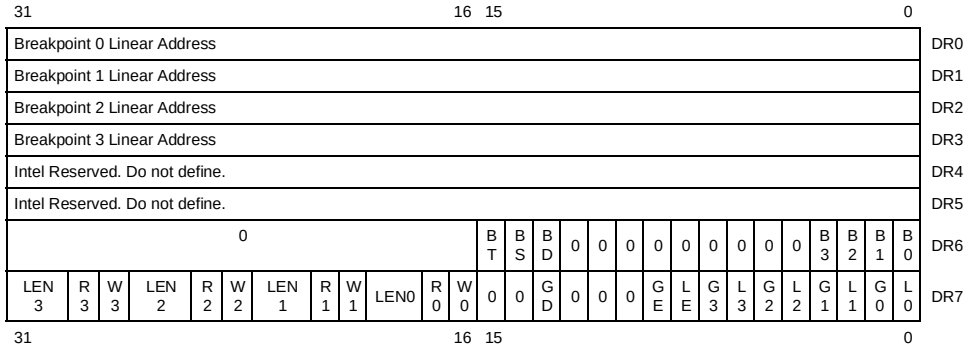
10.8.4 Debug Registers

The Intel486 processor has an advanced debugging feature. It has six debug registers that allow data access breakpoints as well code access breakpoints. Since the breakpoints are indicated by

on-chip registers, an instruction execution breakpoint can be placed in ROM code or in code shared by several tasks. Neither of these is supported by the INT3 breakpoint opcode.

The debug register provides the ability to specify four distinct breakpoint addresses, control options, and read breakpoint status. When the CPU goes through reset, the breakpoints are all in the disabled state. Hence the breakpoints cannot occur unless the debug resisters are programmed.

It is possible to specify up to four breakpoint addresses by writing into debug registers. The debug registers are shown in Figure 10-31. The addresses specified are 32-bit linear addresses. The processor hardware continuously compares the linear breakpoint addresses in DR3–DR0 with the linear addresses generated by executing software. When the paging is disabled then the linear address is equal to the physical address. If the paging is enabled then the linear address is translated to a 32-bit address by the on-chip paging unit. Whether paging is enabled or disabled, the breakpoint register holds linear addresses.



Note: 0 indicates Intel reserved: Do not define.

LENi Encoding	Breakpoint Field Width	Usage of Least Significant Bits in Breakpoint Address Register i, (i = 0–3)
00	1 Byte	All 32 bits used to specify a single-byte breakpoint field.
01	2 Byte	A31–A1 used to specify a two-byte word-aligned breakpoint field. A0 in breakpoint address register is not used.
10	Undefined –Do not use this encoding.	
11	4 Byte	A31–A2 used to specify a four-byte dword-aligned breakpoint field. A0 and A1 in breakpoint address register are not used.

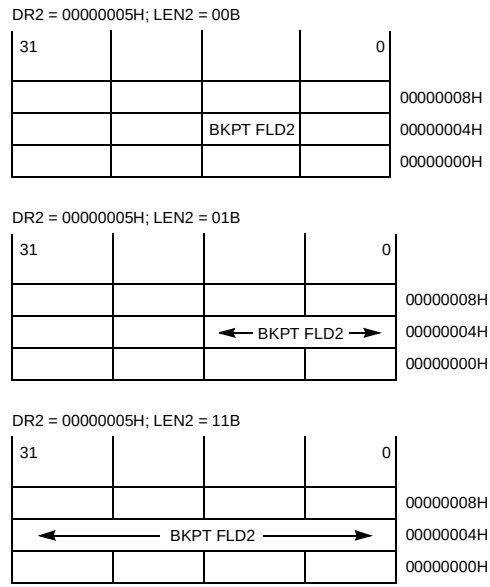


Figure 10-31. Debug Registers

10.8.5 Debug Control Register (DR7)

A debug control register, DR7 shown in Figure 10-31 allows several debug control functions such as enabling the breakpoints and setting up several control options for the breakpoints. There are several fields within the debug control register. These are discussed below:

LENi (breakpoint length specification bits). A 2-bit LEN field exists for each of the four breakpoints. It specifies the length of the associated breakpoint field. It is possible to have three different choices: 1 byte, 2 bytes and 4 bytes. LENi field encoding is shown in Table 10-2.

Table 10-2. LENi Fields

RW Encoding	Usage Causing Breakpoint
00	Instruction execution only
01	Data writes only
10	Undefined—Do not use this encoding
11	Data reads and writes only

The LENi field controls the size of the breakpoint field i by controlling whether all the low order linear address bits in the breakpoint address register are used to detect the breakpoint event. Therefore, all breakpoint fields are aligned: 2-byte breakpoint fields begin on word boundaries, and 4-byte breakpoint fields begin on dword boundaries.

A 2-bit RW field exists for each of the four breakpoints. The 2-bit field specifies the type of usage which must occur in order to activate the associated breakpoint.

RW encoding 00 is used to setup an instruction execution breakpoint. RW encodings 01 or 11 are used to setup write only or read-only or read/write data breakpoints. The data breakpoint can be setup by writing the linear address into DRi. For data breakpoints, RWi can:

= 01 M write only

= 11 M read/write

LENi = 00, 01, 11.

An instruction execution breakpoint can be setup by writing the address of the beginning of the instruction into DRi. RWi must equal 00 and LENi must equal 00 for instruction execution breakpoints. If the instruction beginning at the breakpoint address is about to be executed, the instruction execution breakpoint has occurred, and the breakpoint is enabled, an exception 1 fault occurs before the instruction is executed.

GD (Global Debug Register access detect). The debug registers can only be accessed in real mode or at privilege level 0 in Protected Mode. The GD bit when set provides extra protection against any debug register access even in Real Mode or at privilege level 0 in Protected Mode. This additional protection feature is provided to guarantee that a software debugger can have full control over the debug register resources when required.

The breakpoint mechanism of the Intel486 processors differs from that of the Intel386™ microprocessor. The Intel486 processor always does exact data breakpoint matching regardless of the GE/LE bit settings. Any data breakpoint trap is reported after completion of the instruction that

caused the operand transfer. Reporting is provided by forcing the Intel486 processor execution unit to wait for the completion of data operand transfers before beginning execution of the next instruction.

When the Intel486 processor switches to a new task, the LE bit is cleared. Thus, LE enables fast switching from one task to another task. To avoid having exact data breakpoint match enabled in the new task, the LE bit is cleared by the processor during the task switch. Note that exact data breakpoint match must be re-enabled under software control.

The GE bit supports exact data breakpoint match that is to remain enabled during all tasks executing in the system. The Intel486 processor GE bit is unaffected during a task switch.

NOTE

Note that instruction execution breakpoints are always reported.

G, L (breakpoint enable, global and local). Associated breakpoints are enabled when either G or L are set. When this happens the Intel486 processor detects the i^{th} breakpoint condition, then the exception 1 handler is invoked.

Debug status register. A debug status register, DR6 allows the exception 1 handler to easily determine why it was invoked. Exception 1 handler can be invoked as a result of one of the several events as documented in the Intel486 processor datasheets. This register contains single-bit flags for each of the possible events invoking exception 1. Some of these events are faults while others are traps.

10.8.6 Debugging Overview

Once the Intel486 processor-based system is designed and the printed circuit board is fabricated and stuffed, the next step is to debug the hardware in increments.

The design of a microprocessor-based system can be subdivided into several phases. The design starts with preparation of the system specification followed by conceptual representation in the form of block diagram. The next phase is implementing the design, which consists of the hardware design and the software design occurring in parallel. Hardware debugging usually begins by testing the system with short test programs. Initially the power and ground lines are tested for opens and shorts followed by the testing of the reset function. After the hardware passes these programs, the hardware/software integration phase begins. The test programs are then replaced by the application software and complete system is debugged.

When there are both hardware and software problems, it can be difficult to isolate each. Several types of testing systems are available to assist in this process. The most common type is the in-circuit emulator, which plugs into the microprocessor socket and allows the operation of the system to be controlled and monitored. In-circuit emulators usually include memory that can be used in place of the prototype memory. Another useful test tool is the logic analyzer, which captures the “trace” of the bus activity and displays the sequence of bus cycles that were executed. Most in-circuit emulators also provide this function, which is invaluable for both hardware and software debugging. Test programs can be run from an ICE or a monitor.

The Intel486 processors contain a JTAG (Joint Test Action Group) test-logic unit, which you can use to test the processor and its connections to the system. The JTAG specifications with which this unit complies are documented in *Standard 1149.1-1990, IEEE Standard Test Access Port and Boundary Scan Architecture* and its supplement, *Standard 11.49.1a-1993*. You can also refer to the “Boundary Scan” section of the individual Intel486 processor datasheets for more information on using the JTAG unit.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W