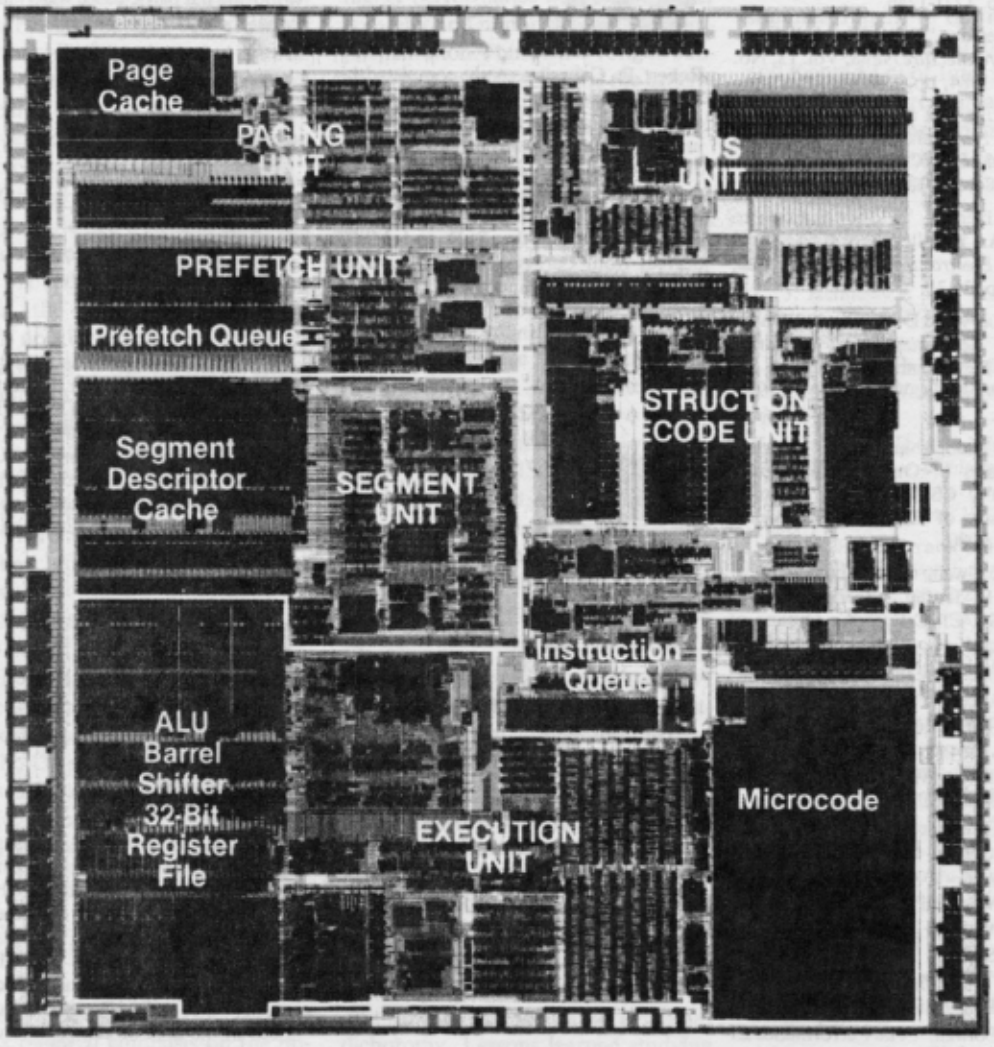
Page Cache
PREFETCH UNIT
Prefetch Queue
Segment Descriptor Cache
SEGMENT UNIT
Instruction Queue
ALU
Barrel Shifter
32-Bit Register File
EXECUTION UNIT
Microcode

Architecture of the Intel 80386

John Crawford

Intel Corporation, SC4-59
2525 Walsh Ave.
Santa Clara, Ca. 95051

ABSTRACT

Intel's 80386 32-bit microprocessor[1,2,3] is the newest member of the 86 Family of microprocessors. It features a full 32-bit architecture implemented in a 1.5μ CMOS technology[6], supports an average instruction rate of 3 to 4 million instructions per second, and is fully software compatible with the 8086, 8088, 80186, 80188, and 80286 microprocessors. Memory management is fully supported with on-chip paging underneath segmentation.

Introduction

The 386 offers a full 32-bit architecture to the programmer, featuring 32-bit registers, 32-bit address formation, and a full set of 32-bit instructions. At the same time, the 386 is object code compatible with the 16 bit members of Intel's 86 family of processors. A complete segmented/paged Memory Management and Protection mechanism supports segments up to 4 Gigabytes in size, and uses a standard two-level paging mechanism underneath segmentation to support physical memory management of these large segments.

Segmentation and Paging are fully supported on-chip with a high-speed address translation pipeline which caches segmentation and paging mapping information on-chip. Address translation is fully pipelined with other CPU operations so that a memory Load instruction, including segment and page address translation, requires just 4 clock cycles (250 ns at 16 Mhz) and a Store instruction requires just 2 clock cycles (125 ns at 16 Mhz).

Basic Architecture

General Registers

Eight 32-bit general registers shown in figure 1 are named EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI, and provide high-speed storage for integer data and addresses. The low order 16 bits of these registers contain the 16-bit 8086/80286 registers AX, BX, CX, DX, SP, BP, SI, and DI. Eight 8-bit registers AH, AL, BH, BL, CH, CL, DH, and DL provide direct access to the upper and lower 8 bits of the AX, BX, CX, and DX

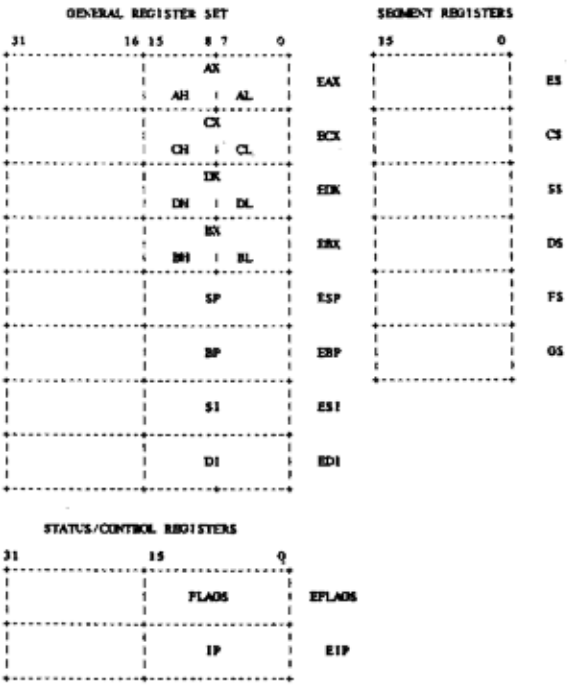

Figure 1 - General Registers

registers. When accessed as 8 or 16-bit registers, the other bits of the 32-bit registers are undisturbed.

Two additional 32-bit registers, also shown in figure 1, support processor control. The 32-bit EIP register is the *Instruction Pointer* register, and points to the next instruction the processor is to execute. The 32-bit EFLAGS register provides a number of status and control bits. Status bits are set after most arithmetic operations to indicate carry, overflow, sign, and zero result. Control bits are provided to mask interrupts, provide single-step execution, and control a number of execution modes. The lower 16 bits of EIP and EFLAGS contain the 8086/80286 IP and FLAGS registers.

Three additional sets of registers provide specialized functions for processor control. One set of registers

controls the operation of the memory management hardware and provides base addresses for the memory mapping tables. A second set of registers provides access to the hardware to simplify production testing.[4] A third set of registers support effective software debugging. Four address registers and two status/control registers provide the ability to set up to four code or data breakpoints at arbitrary addresses, and to break program execution when a breakpoint is encountered.

Memory Addressing

Segmentation. In order to manage its large virtual memory space, the 386 uses a segmented, or two-dimensional, addressing mechanism. Segments divide main memory into multiple linear address spaces, which correspond to the logical units viewed by the programmer. Within a segment, data is addressed by giving a simple byte offset. Because they divide memory into multiple linear address spaces, segments greatly simplify the relocation, sharing, and protection of multiple logical units. Up to 16K (2^{14}) segments can be defined in each task, and each segment can be up to 4 Gigabytes (2^{32}) in size, so the virtual address space is 64 Tera-bytes (2^{46}) per task.

Segment Regsisters. Due to the use of segmentation, main memory addresses consist of two parts: a segment part and and offset within that segment. 6 Segment registers, shown in figure 1, are provided to hold the segment parts of addresses. In order to address data within a segment, a 16-bit selector which identifies that segment must be loaded into one of these 6 segment registers. 14 bits of the selector provide an index into a protected segment descriptor table where the processor reads the base address, limit, and access attributes of the segment, as described in a later section. Two of the segment registers are dedicated to holding selectors for the current code segment (CS) and stack segment (SS). The remaining four segment registers are available to allow up to 4 data segments to be referenced at any point in time.

Addressing Modes. 32-bit offsets within segments are generated by adding together up to 3 components: a 32-bit base register, a 32-bit index register scaled by 1,2,4, or 8, and an 8 or 32-bit displacement. Any of the 8 general registers can supply the base or index parts of a memory address. Table 2 summarizes the 32-bit address mode choices available on the 80386. For compatibility with previous processors, the full set of 16-bit address modes are also supported on the 80386, but are not illustrated here.

Immediate Operands. The simplest way to supply an instruction operand is to include it directly within the instruction. These *immediate* operands can be 8, 16, or 32-bits in size. Full size constants of 8, 16, or 32 bits

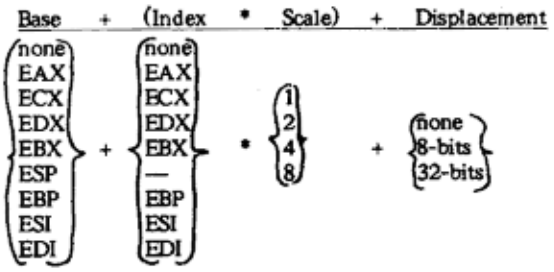

Table 2 - 80386 Addressing Modes

can be supplied for instructions which operate on byte, word, or double-word data. A 16 or 32-bit immediate operand can be given as a sign-extended 8-bit immediate in order to conserve code space in the frequent case of a small number of significant digits in the immediate operand.

Data Types

On the 386, as well as all other members of the 86 family of processors, main memory is byte addressable, so each 8-bit byte in memory has an address. If more than 8 bits are required to represent the values in a data type, multiple sequential bytes are used, with the low order bytes stored at lower addresses, and with the address of the datum given by the address of the low order byte. As with the other 86 family members, a *word* is 16-bits wide, and a *double-word, or dword*, is 32 bits wide. Floating point numbers are stored in 32, 64, and 80 bit formats, which occupy 4, 8, and 10 consecutive bytes, respectively.

Data Type and Instruction Summary	
8, 16, *32-bit* Integer	Add, Subtract, Multiply, Divide Add w/Carry, Sub w/Borrow Increment, Decrement, Negate
8, 16, *32-bit* Ordinal	And, Or, Xor, Not, Move, Push, Pop, Exchange *Move with Sign/Zero Extend* Shift, Rotate, *Double Shift* Compare, Test
packed BCD unpacked BCD	Add, Subtract, Multiply, Divide
16, 32, 64-bit Integer 32, 64, 80-bit Real 80-bit BCD	Add, Subtract, Multiply, Divide Compare, Remainder, Round Move, Exchange, Convert, Scale log, exponential, square root tangent, *sine*, *cosine*, arctan
BYTE string WORD string *DWORD string* (1..4G bytes length)	Move, Compare Fill, Scan, Translate
bit array (1.4G bits)	*Test, Set, Clear, Complement*

Table 3 - Data Types and Instruction Summary

Table 3 summarizes the data types directly supported by the 80386 and its companion floating point coprocessor, and lists the most important instructions supplied for each type. New data types and instructions are printed in italics.

Instruction Forms

The 80386 provides a rich set of instructions which can be broken into data manipulation instructions (e.g. add, move), and control transfer instructions (e.g. jump, call). Instructions can have no operands, one operand, or two operands, where the operands can be in a processor register, main memory, or directly in the instruction as an immediate operand. Representative data manipulation instructions are listed in table 3, with new instructions in italics.

Data Manipulation Instructions. One operand instructions can take their operands either from memory or from a register. These instructions generally use this single operand both as a source and a destination. Two operand instructions are available in the forms listed in table 4. Note the presence of instruction forms with memory destinations. These operate directly on data located in memory without the need to first load the data into a register.

Two Operand Instruction Forms	
Destination	Source
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Table 4 - Two Operand Instruction Forms

A powerful set of efficient string instructions is provided for operating on strings with BYTE, WORD, or DWORD elements. A string move instruction operates at the maximum bus bandwidth for rapidly moving blocks of data in memory. Two strings can be compared. A string can be filled with a fixed value, can be scanned for the first occurance of a given value, or can be translated using a character translation table. A set of string moves is provided to transfer data rapidly between memory and I/O space to support fast device access such as to hard disks or network controllers.

Control Transfer Instructions. A set of conditional jumps conditions changes in program flow on the settings of the status bits in the EFLAGS register (e.g. carry, sign, greater/less than, equal). Two sets of unconditional Call, Jump, and Return instructions are provided to transfer either within a segment, or between two segments. Intra-segment transfers change only the contents of the EIP register. Inter-segment transfers change both the EIP register and the CS segment register to begin execution in a different segment.

The CALL and RETURN instructions use the program stack contained in the segment addressed by the SS register, and whose top is marked by the ESP register. Parameter passing and local variable allocation on this program stack are supported directly in the instruction set. Parameters can be placed on the stack before executing a CALL instruction with PUSH instructions, and can be accessed within the called procedure at a small displacement from the ESP or EBP registers. ESP can be used for simple languages which do not require the maintenence of a subprogram display. EBP is useful as a pointer to the activation record for the current subroutine in languages which require the ability to address local variables of outer procedures, and to maintain the necessary static and dynamic links. Two instructions, ENTER and LEAVE, support procedure entry and exit. ENTER will build the display for a new procedure and allocate space for local variables on the stack. LEAVE will deallocate the display and local variables just before returning. The RETURN instruction can adjust the ESP register to remove parameters pushed before the matching CALL by subtracting from ESP after the return pointer is popped off the stack.

Instruction Encoding. In order to support binary compatibility with the previous 16-bit members of the 86 family, the 80386 instruction encoding includes all of the 8 and 16-bit instructions from the 80286. The 32-bit instructions were added by using the same instruction set encoding, but simply interpreting the 16-bit instructions as 32-bit instructions by use of an operand size indication. The operand size can be set to 16 or 32 for all the instructions in a code segment with an attribute bit in the code segment descriptor. Or, the operand size can be set for a single instruction by prepending an instruction prefix byte to that instruction. The code segment attribute provides an efficient method to type entire code segments as one size or the other. The prefix provides the flexibility to operate on 16-bit data in a 32 bit code segment, and vice-versa.

A similar sizing mechanism is used to select between 32-bit addressing and 16-bit addressing. The code segment attribute indicates the default address size for an entire code segment, and an instruction prefix indicates that a different address size should be used for a single instruction.

A number of new instructions were added to the 386 and 387 in addition to the 32-bit extensions described above. Representative operations are listed in table 3 in italics.

Memory management and Protection

The memory management mechanism combines both segmentation and paging for a flexible, complete mapping and protection mechanism.[5] Segmentation is the top, logical level of the memory management model. It supports the definition of protected regions of memory that correspond directly to constructs used by the pro-

grammer (e.g. code procedures, data structures, stacks). Four levels of protection are provided with the segment model: at a given time the processor can be executing at one of four privilege levels from 0 (most privileged) to 3 (least privileged). If executing at privilege level *n* the processor can only access segments at level *n* or levels of lesser privilege (numerically greater levels).

Paging underneath segmentation provides an efficient mechanism for the management of physical memory, both in the processor's main memory and the paging disk on virtual memory systems.

Figure 5 illustrates the two stages of address translation. First a two part virtual address is translated by segmentation to a 32-bit linear address, that is then passed through the page translation mechanism to obtain the physical address. The page translation step can be disabled by setting a processor control bit. In this case, the address put out by the segment address translation process is the physical address. This provides support for systems that do not need paging, and also provides compatiblity with the 80286, which did not support paging.

The segmented protection model is a superset of that provided for the 80286, in order to support binary compatibility even at the OS level. The model was extended to support 32-bit segment base addresses and 4 gigabyte segment sizes for the 80386, but the concepts and mechanisms were carried forward from the 80286.

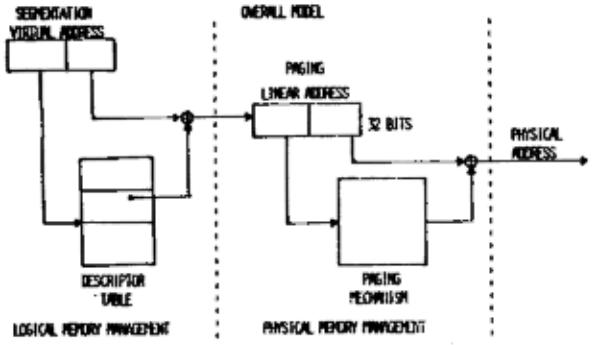

Figure 5 - Overall Address Translation

Segmentation

Memory is addressed with a two part address, a segment part and an offset within a segment. Segments are identified by user programs by use of a 16-bit *selector* which contains a 14-bit index into protected *descriptor* tables maintained by OS software. The descriptor associated with a selector contains the base address, size, and access attributes for the segment. To address data within a given segment, the base address is added to the offset part of the two part address to obtain a 32-bit *linear* address that is the output of the segment address relocation process. A fault is reported

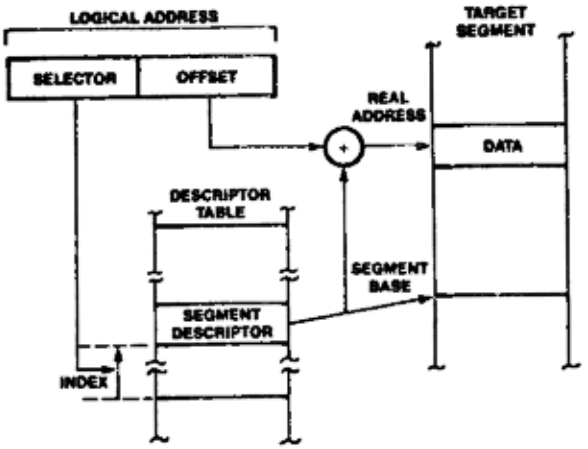

Figure 6 - Segment Address Translation

if the offset is larger than the segment size, or if the type of the access is not permitted by the access attributes in the descriptor. Rather than access the descriptor table for every memory access, descriptors are "cached" into shadow registers every time a selector is loaded into a segment register. Once cached, all references to the segment are relocated and validated by the processor before accessing physical memory. The segment address translation process is illustrated in fig. 6.

Paging

A standard 2 level page table is used, with 4K pages, as illustrated in figure 7. A processor register points to the base of the first level table. Table entries at both levels are 4 bytes wide, and each table contains 1K entries, so the page tables themselves exactly fit into 4K pages to simplify allocation and swapping of page tables.

The 32-bit address from the segment translation process is divided into 3 parts for page mapping. The upper 10 bits select an entry in the first level table which points to a second level table. The middle 10 bits select an entry in this second level table which contains the upper 20 bits of the physical address of the desired

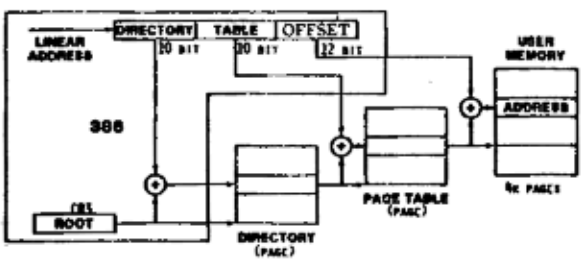

Figure 7 - Page Address Translation

page. The lower 12 bits of the input address, which do not participate in the page translation process, are concatenated with these upper 20 bits to form the output physical address.

This page translation process requires two memory accesses to the page table map for every memory access. A page translation cache, sometimes called a translation look-aside buffer (TLB), is used to cache the 32 most recent virtual to physical address translations to avoid referencing the memory-resident page tables.

Full support for virtual memory is provided, including full restartability of all instructions, and provision of page usage statistics with Dirty and Accessed bits per page. Instruction restart is supported to ease the burden on operating system software in recovering from page faults. After a missing page is retrieved from the disk, the faulting program is simply restarted by returning to the instruction which caused the page fault.

Protection

The segmented memory model provides protection between tasks, and between user programs and the operating system. Each task can have its own address space, supported by its own set of segment and page tables, to provide protection between tasks. To efficiently support an address space per task, two segment descriptor tables are used. One, the Global Descriptor Table (GDT), is shared by all tasks in the system, and generally holds descriptors for OS code and data. The second table, the Local Descriptor Table (LDT), is unique to each task in the system.

Within a task, four privilege levels are defined to partition the segments defined by a task between system access and user access. As illustrated in figure 8, the processor executes at one of four privilege levels and has access to segments at that level or higher (less privileged) levels. Attempts to access segments at lower (more privileged) levels cause protection traps. This permits user and system segments to reside in a common address space, and still protects system segments from unrestricted user access.

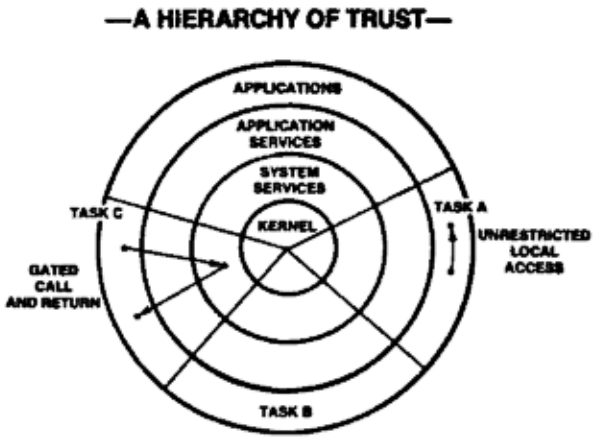

Figure 8 - 4 Rings of Privilege

Changes in privilege levels are controlled by Gates. Gates are special segment descriptor types that indirect access to a fixed entry point in another segment. This provides a restricted transfer of control from higher levels (less privileged) to specific entry points in lower levels using the standard inter-segment CALL instruction. To avoid protection holes inherent in the use of a single stack, each privilege level has its own program stack. During level transitions through gates, the progam stack of the new level is made the active stack by reloading the SS and ESP registers. Parameter passing is supported during the gate transition by the ability to copy data from the caller's stack to the callee's stack. Through the use of gates, the user program, while completely restricted from access to system level segments, can be allowed to call OS service routines directly with the same inter-level CALL, and the same parameter passing conventions used to transfer control to other user-level routines.

Tasks

Another dimension of the Operating System support incorporated into the 80386 is the direct support of the task concept, to provide efficient context switching in multi-tasking environments. A special segment type, a Task State Segment (TSS), is provided to store the machine state for a dormant process. This segment contains the general registers, EFLAGS, EIP, segment registers, and pointers to the Local Descriptor Table and Page Table for the task.

The 80386 will perform a complete task switch if an inter-segment CALL or JUMP instruction, or an interrupt indicates a transfer to a TSS. A task switch operation involves storing the current processor state in the current TSS, making the new TSS the current TSS, and then loading the processor state from this new TSS. A task switch initiated by a CALL instruction or interrupt will also store the selector for the old TSS into a link field in the new TSS to permit the old task to be resumed upon "return" from the new task.

I/O Space

A 64K I/O address space, totally separate from the main memory space, is provided for a clean interface to device registers in peripheral contollers. A set of instructions provide data movement between the I/O space and the AL, AX, or EAX registers. String instructions provide a high-bandwidth transfer between I/O space and a block of main memory.

This I/O space is protected from arbitrary access through two mechanisms. The 2-bit IOPL field in the EFLAGS register defines the highest privilege level for which unrestricted I/O access is permitted. A variable length I/O permission bitmap located in the current

TSS contains a bit for every I/O address. A program executing at a higher (less privileged) level than IOPL will consult this I/O permission bitmap if it attempts to execute an I/O instruction. If the bitmap indicates that the task has permission to access the given I/O address, the I/O instruction will execute normally. Otherwise, the I/O instruction is aborted, and a protection fault is generated.

Virtual 8086 Mode

A special processor mode, named *Virtual 8086 Mode*, was added to support multitasking of 8086 (8088, 80186, 80188) tasks within the segmented/paged protected environment. This mode allows a 386 OS to provide a complete virtual 8086 machine to execute even "dirty" PC applications, each with its own copy of an 8086 operating system. Within this environment, segment registers are loaded as in the 8086, and relocate 16-bit addresses within the 1 Megabyte space supported by the 8086. This 1 megabyte address space is mapped by paging to allow protection and swapping of virtual 8086 programs. I/O instructions use the I/O permission bitmap described above to give "dirty" applications direct access to a restricted set of I/O devices, to support fast I/O interfacing. Interrupts and exceptions which occur when the processor is executing in Virtual 8086 mode cause a mode switch back to protected mode where the interrupt is handled by the protected mode OS. This OS must emulate certain privileged instructions that may be executed by the Virtual 8086 program, and can choose to handle interrupts itself, or reflect them back to the Virtual 8086 program.

Interrupts

The 80386 supports a vectored interrupt mechanism to signal asynchronous external events and to report error or exceptional conditions that occur as part of instruction execution. Interrupts are vectored through a 256 entry Interrupt Descriptor Table. Interrupts can be handled by interrupt procedures within the current task, or can cause a task switch to a new task. Entries in the IDT are gates which identify the entry points of interrupt handling procedures, or identify the TSS for an interrupt handling task.

Conclusion

The 80386 combines a full 32-bit architecture with full object code compatibility with the previous 16-bit members of the 86 family. A flexible and powerful memory management model combining both segmentation and paging is fully supported with on-chip hardware to minimize cost and maximize performance. Virtual 8086 mode, combined with the I/O permission bitmap, provides an environment for efficient execution of even "dirty" PC programs within the protected, paged, multi-tasked environment supported by the 80386. Thanks to these key architectural features, the 80386 provides access to a vast amount of industry standard software, while at the same time delivering state of the art performance in a 32-bit CPU.

References

[1] "80386 Programmer's Reference Manual", Intel Corp., Santa Clara, CA, 1986.

[2] "80386 Hardware Reference Manual", Intel Corp., Santa Clara, CA, 1986.

[3] "80386 System Software Writer's Guide" Intel Corp., Santa Clara, CA, 1986.

[4] P. Gelsinger, "Built-in Self Test for the 80386", Proceedings, ICCD Conference, Oct. 1986.

[5] P. J. Denning, "Virtual Memory", *Computing Surveys, Vol. 2, No. 3*, pp. 153-189.

[6] J. Prak, "High Performance Technology, Circuits, and Packaging for the 80386", Proceedings, ICCD Conference, Oct. 1986.