The i486 CPU: Executing Instructions in One Clock Cycle

A cache integrated into the instruction pipeline lets this 386-compatible processor achieve minicomputer performance levels.

John H. Crawford

Intel Corp.

Intel began the i486 processor development program shortly after it introduced the 386 processor in 1985. From initial concept, the chip design team worked with the following CPU goals:

1) ensure binary compatibility with the 386 microprocessor and the 387 math coprocessor,
2) increase performance by two to three times over a 386/387 processor system at the same clock rate, and
3) extend the IBM PC standard architecture of the 386 CPU with features suitable for minicomputers.

Compatibility. We recognized the need to be compatible with the 386 Family Architecture.[1-3] As Table 1 shows, a large base of installed software exists to run on a processor that can execute this instruction set. This need

Table 1.
The 386 Family Architecture software base.

	Operation system	
DOS	Unix/386	OS/2
10,000+ applications	3,000 applications	Multitasking
150+ 32-bit applications	Full 32-bit operating system	Graphical interface
Windows graphical interface	System V, Release 4.0	Multithreaded applications
Windows/386 multitasking	Multiprocessor Unix	LAN+ database

0272-1732/90/0200-0027 $01.00 © 1990 IEEE

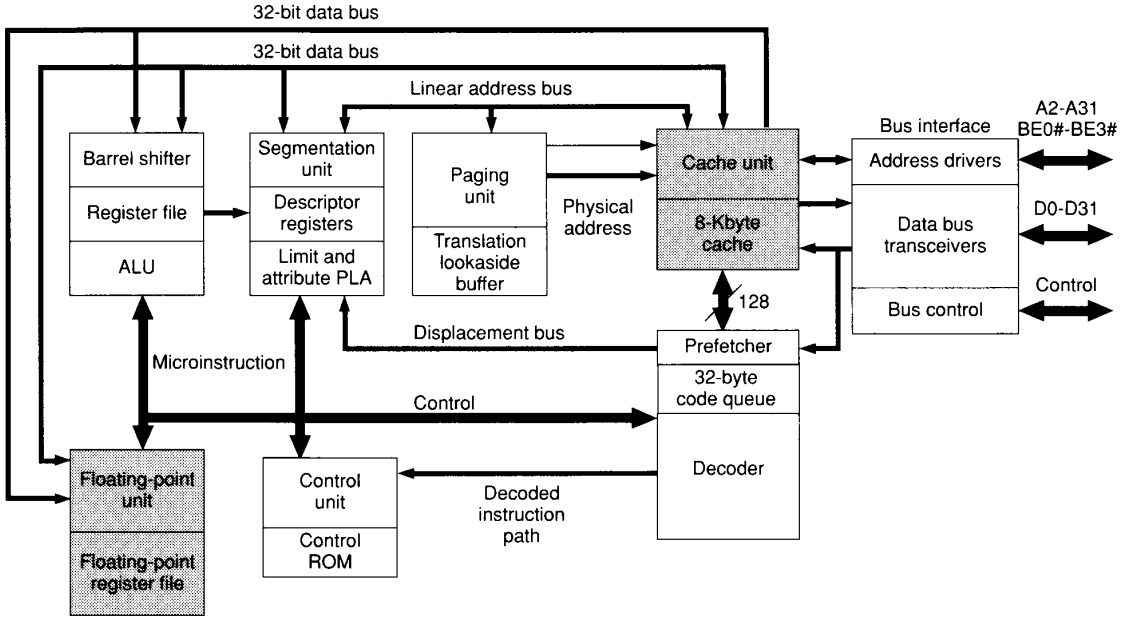

Figure 1. Block diagram of the i486 processor.

to be compatible with an existing software base influenced many of the design decisions during the development of the i486 processor. Where appropriate, I note these influences throughout this article.

In contrast to the clear choice for software compatibility, we saw no strong need for hardware compatibility, that is, to provide the same hardware interface (bus) used in the 386 CPU. Indeed, to achieve higher performance at the same clock rate, and to provide higher clock rate selections over time, the i486 CPU bus had to differ significantly from the 386 processor bus. Though I don't discuss the bus here, Intel's data book describes it in detail,[4] and Grochowski and Shoemaker's article[5] discusses our bus design decisions.

Performance. We used a 1.5-micrometer process to fabricate the 386 CPU, which contains approximately 275,000 transistors. However, we developed the i486 CPU for a 1-μm CMOS process that provided more than two times the number of transistors, or 1.2 million. The process also provided a dense static RAM cell suitable for an on-chip memory cache. The significant increase in the transistor budget provided the raw material for performance improvement.

Figure 1 shows the block diagram of the i486 processor. The shaded blocks indicate functions that are partitioned onto separate chips in a system based on the 386 CPU but which appear on chip in the i486 CPU. These functions are

- the FPU (floating-point unit), which boosts performance of floating-point operations; and
- the 8-Kbyte, unified cache memory.

Although the remaining blocks look similar to the corresponding blocks in the 386 CPU, note that the underlying logic is quite different. We revamped the prefetch, instruction decode, and control logic to optimally integrate the cache into the instruction execution pipeline. This critical improvement allowed the i486 CPU to execute the most frequent, simplest instructions in one clock cycle.

Note also that, like the 386 processor, the i486 CPU includes all the hardware necessary for a complete memory management and protection mechanism. This mechanism provides four protection levels using a combination of segmentation and paging techniques. An on-chip paging cache, or TLB (translation lookaside buffer), of 32 entries supports paging operations.

Let's look at the cache and the integer pipeline. To be brief, I won't discuss the floating-point pipeline here.

On-chip cache

Although the performance goals required, and the transistor budget supported, an 8-Kbyte, on-chip memory cache, we needed to explore a large design space to obtain the best cache design for the i486 processor. The

need to be compatible with the existing software focused the design to a cache that would be invisible to software. In particular, the hardware had to guarantee consistency of the cache with main memory without software intervention.

It may not be obvious that a lowly PC needs any cache consistency support. However, even the original IBM PC (1981) includes a DMA (direct memory access) processor that can generate writes to memory by "stealing" the memory bus from the CPU. Also, we knew the 486 CPU had to execute old versions of operating systems (DOS, OS/2 version1.1, Unix) that did not/could not include any "cache flush" instructions.

We achieved cache-consistent hardware support with a buffered write-through consistency protocol combined with "snooping" (or address detection) hardware. We also added a pin to allow external address-decoding hardware to identify noncacheable memory areas. As an option for new versions of operating systems that use paging, we included new attribute bits in the page table to identify noncacheable memory at a page granularity.

Buffered write through. A buffered write-through technique keeps main memory consistent with the contents of the on-chip cache by passing all store operations (memory writes) to the main memory. All memory writes initiated by the CPU move directly to main memory, a flow that is smoothed by four buffers. This process ensures that the cache and memory remain consistent despite writes to memory by the processor.

While writing to main memory typically takes several clock cycles, the CPU executes a store instruction in one cycle. Without buffers, a store following another store would have to wait for the write operation to complete to memory before it could execute. In a system with a well-tuned memory system that completes a write cycle in two or three clock cycles, the four write buffers permit up to six consecutive writes to occur before introducing a stall. The first write will complete before the write buffers fill up, permitting the second write to be pulled from the write buffers before the sixth consecutive write occurs. This feature is particularly important for procedure calls that typically contain a sequence of store operations to push parameters to the stack, push a return address for the call, and save working registers by pushing them onto the stack.

Consistency in cache and main memory also requires correct handling of writes to memory that are initiated by bus masters other than the processor (DMA or a second processor). The snooping hardware on the CPU handles this aspect of consistency. When a bus master other than the i486 processor performs a memory write, the system hardware provides the address at the pins of the processor. It also activates a control pin to indicate that the address at the pins should be invalidated if present in the on-chip cache. Upon activation of this control pin, the CPU will sample its address pins and invalidate the line mapping that address, if present in the cache. If the address is not found in the cache, the state of the cache does not change. In this fashion, the system hardware can force the processor to remove from the cache any areas of memory that are written by other bus masters.

Cache parameters. We fixed the cache size at 8 Kbytes based on the limits of the 1-μm process used for the i486 CPU. We selected the other cache parameters to maximize the effectiveness of the cache, given this fixed size.

We implemented the cache using a four-way, set-associative organization[6] with LRU, or least recently used, replacement technique. To interleave cache and TLB access, we needed at least a two-way, set-associative cache to achieve an 8K size, since we had a 4-Kbyte page size. A four-way, associative design would provide a 10-percent lower miss rate[7] than a two-way cache yet require minimal additional die area. An eight-way design, however, would provide little improvement.

Based on performance simulations of the cache with address traces captured from a 386 CPU,[5] we used a 16-byte line size. We felt it necessary to perform simulations specific to the i486 CPU since we had found little published information on the overall performance of a cache.[7,8] In addition, we realized the proper optimization is very dependent on specific implementation details. These simulations indicated that a larger line size (32 bytes) would actually slightly decrease the performance of the CPU, even though it would provide a higher hit rate in the cache, a phenomenon also reported in the last two references.[7,8]

The cache, bus, and CPU core minimize the impact of a cache miss on performance. Thus, we achieve the maximum performance possible given a fixed-size cache, and the design supports the use of a second-level cache external to the CPU for maximum performance configurations. Our main considerations were to

1) *Use a burst cycle for cache misses to retrieve a cache line from memory with the minimum bus occupancy.* The burst cycle requires the normal latency to access the first 32-bit piece of a line, but the remaining three pieces can be retrieved at the rate of one per clock cycle. Once the first piece is addressed, the line can be read in with a "burst" of data cycles.

2) *Structure the bus and cache so that burst cycles first access the data causing the miss.* This feature requires that a burst cycle be capable of starting on any one of the four 32-bit pieces of our 16-byte cache line.

3) *Structure the CPU and cache so that the CPU can proceed with the next instruction(s).* This feature requires the CPU to proceed once the data that caused the miss has been retrieved, while the cache and bus complete the line fill in the background.

4) *Use a line buffer for cache fills.* Here, we wanted to take only one cache cycle for cache fills, rather than

Table 2.
Miss rates in the i486 CPU memory cache.*

Program	Miss rates: 4-byte prefetches	16-byte prefetches	Read	Total
1-2-3	—	0.00	0.02	0.01
Auto CAD	—	0.02	0.01	0.02
Windows 386	—	0.05	0.11	0.08
Frame	0.05	0.14	0.06	0.10
Sunview	0.04	0.10	0.08	0.09
Falcgph	0.02	0.06	0.09	0.07
Falcsnd	0.04	0.11	0.07	0.09
Invframe	0.07	0.20	0.08	0.14
Tpascal	0.04	0.11	0.05	0.08
Troff	0.02	0.06	0.02	0.04
Geometric mean	0.04	0.09	0.07	0.08

*1-2-3 and AutoCAD not included in mean.

four. By doing so, we free up cache cycles for instructions following a miss.

Cache performance. Trace-driven simulations provided a confirmation of our choice of cache parameters. Table 2 lists the miss rates for traces from several single-user multitasking environments, resulting in an overall 8-percent miss rate. This table includes data for 4-byte prefetches to correlate with other published data, as well as for 16-byte prefetches that are more useful for estimating the performance of the cache integrated into the CPU. The 8-percent overall miss rate correlates closely to the 8.4-percent "design target miss rate" cited by Hill and Smith.[7,10]

Cache bandwidth. Because the instruction pipeline can execute most instructions, including those that reference memory, in one clock cycle, the cache on the i486 processor must support a peak rate of two memory references per clock cycle. The references use 1 to 8 bytes to load/store a data operand for the current instruction, and 1 to 15 bytes to fetch an instruction further ahead in the instruction stream.

Two approaches provide a good solution to the need to perform two memory accesses in one cycle. One approach splits the cache into an instruction cache and a data cache, and provides each cache with a private bus to connect to the processor core. The other approach uses one cache to store both instructions and data. It also uses an instruction prefetch buffer with a wide bus to the cache to minimize the contention at the cache between instruction and data accesses.

Many microprocessor implementations use the split-cache approach.[11-14] This approach is particularly attractive for implementations that have the caches external to the CPU, since pin limitations dictate that narrow buses connect to the cache. It is particularly effective when processor pins can be shared by time-multiplexing the signals for the instruction and data buses.[14] In most RISC processors, the cache ports can be 32 bits wide, since all instructions and the majority of data references (except for programs making heavy use of double-precision floating-point data) are 32 bits wide. This fixed partitioning guarantees that the peak rate of one instruction fetch and one memory access can occur in one clock cycle with no contention.

The unified-cache approach using a wide bus in conjunction with prefetch buffers provides an alternative that is particularly attractive for the i486 CPU. The 386 Family Architecture requires a prefetch buffer to support fast alignment and decoding of variable-length instructions. It is easy to support a wide path from the on-chip cache into this prefetch buffer with on-chip wiring. In fact, the chip plan places the prefetch buffers close to the cache array to minimize the wiring required. This plan provides a very high bandwidth path for fetching instructions, thus minimizing the contention of instruction and operand access to the unified cache. A 16-byte bus gave us a bandwidth of 400 Mbytes/s (at 25 MHz) to fetch instructions. Since the i486 CPU requires approximately 50 Mbytes/s of bandwidth on the average, we minimize the contention between instruction and operand accesses to the cache.

The simplicity and performance of the unified-cache approach also influenced our decision. A unified cache simplified the design in several ways. For example, we had to design only one cache! Also, the prefetch buffers needed for this scheme were required anyway to support high-speed decoding of the 386 Family Architecture instructions. Third, no special logic is required to handle self-modifying code.

The unified cache improves performance in ways that compensate for the small amount of contention that remains between instruction and operand access to the cache. 1) The cache accommodates an 8-byte operand access in one clock cycle. This feature improves the performance of instructions referencing double-precision floating-point data and instructions that load segment descriptors. Both require an 8-byte operand access. 2) The hit rate of the unified cache is higher than separate caches of the same total size.[7,9,10]

The improved hit rate results from the lack of fixed partitioning between instructions and data. Consequently, the partitioning can adjust itself based on the instantaneous locality of code versus operand references. The charts in Figure 2, which use a "stacked-line" graph to show the contents of the cache over time, illustrate this phenomenon. In these charts, the number

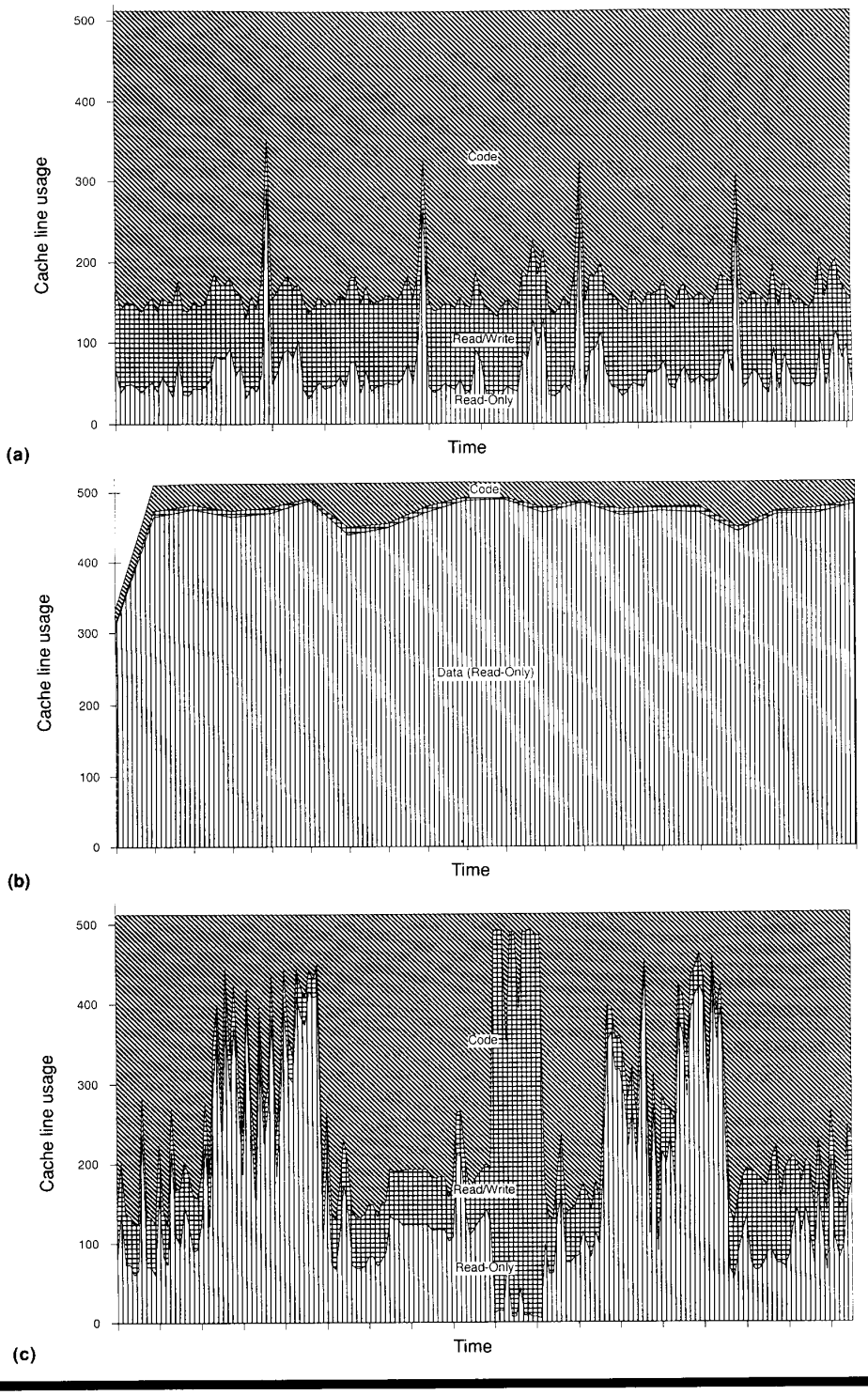

Figure 2. Cache partitioning vs. time: more code than data (a), more data than code (b), and high variability (c).

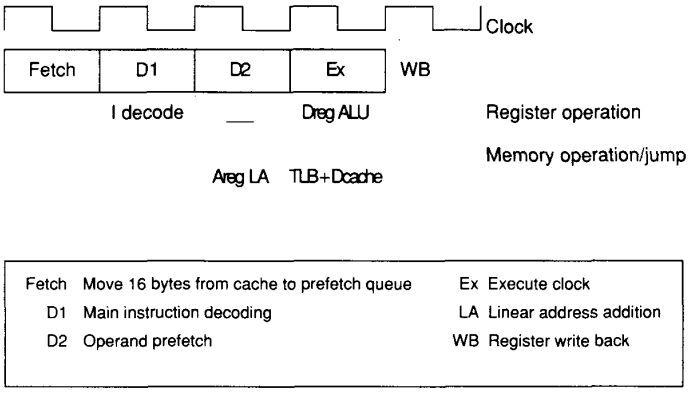

Figure 3. The 486 CPU pipeline.

of lines (out of a total of 512) containing read-only data appear on the bottom, the number of lines containing read/write data in the middle, and the number of lines containing instructions on the top. A cache simulator processing several large address traces taken from a variety of applications produced the charts. Each graph represents 16 million memory references, or about 1 second of activity on a 25-MHz i486 CPU.

Figure 2a shows an application that devotes about 25 percent of the cache to data lines, and 75 percent to instructions. Apparently this application has less locality in instruction accesses than data accesses. In contrast, Figure 2b shows an application that devotes almost all of the cache to data and only a few lines to instructions. Apparently this application is in a tight code loop. Figure 2c shows an application that makes dramatically different demands on the cache during the 1-second sampling window. These examples visually confirm the "self-adjusting" phenomenon that permits a unified cache to exhibit a higher hit rate than split caches of the same total size.

Instruction pipeline

We designed the instruction pipeline to execute instructions at a sustained rate of one per clock cycle. A key to achieving this rate was integrating the cache into the pipeline.

Another key to achieving this rate was pipelining the instruction decoder into two stages to provide a sustained throughput of one instruction per cycle. An overall design concern centered on the fact that existing programs had to perform well. That is, we could not rely on a compiler to cover any inefficiencies in the pipeline.

Pipeline overview. Figure 3 illustrates the five stages in the execution pipeline. In this figure, the horizontal axis represents time.

The Fetch stage extracts the instruction from the cache. D1 represents the main instruction decoding stage, while in the D2 stage secondary instruction decoding and memory address computation take place. In the Ex stage the instruction begins to execute, and in the WB stage results are written to the register file.

Fetch stage. Since the CPU fetches an entire cache line from the cache, most instructions don't require this stage. On the average, the CPU fetches about five instructions during each 16-byte cache access. We include this stage as part of the "average" instruction handling because it is always required at the target of a branch, and of course an instruction must be fetched before it can be decoded or executed.

D1 stage. This first instruction decoding stage processes up to 3 instruction bytes. It also decodes the actions to occur during the D2 stage for computing memory addresses. The D1 stage determines the length of the instruction and directs the instruction aligner/prefetch queue to step to the next instruction, possibly stepping over data to be used in the D2 stage.

Decoding of instruction prefixes proceeds 1 byte at a time, with each prefix occupying a single D1 stage in addition to the main D1 stage for decoding the opcode. Instructions having a 2-byte opcode, those containing hex 0F in the first byte, require two clock cycles in the D1 stage (as if 0F were a prefix byte). All other instructions use just one D1 cycle. About 6 percent of the instructions in a "typical" DOS program require the extra D1 cycle, while the average Unix program proportion is less than 3 percent.

D2 stage. This stage completes the decoding of instructions and also computes memory addresses. It decodes a 1- to 4-byte memory displacement field, or a 1- to 4-byte immediate constant per clock cycle. If both a memory displacement and an immediate constant are present in the instruction, decoding takes two D2 cycles.

The D2 stage also computes memory addresses in parallel with decoding displacements or immediate constants, as directed by the D1 stage. It will decode a displacement (if present) and add it with a base register (if present) in the first D2 cycle, and a scaled index register (if present) in the second cycle. If no index register exists, address computation completes in one D2 stage. Only about 5 percent of the instructions require two D2 stages.

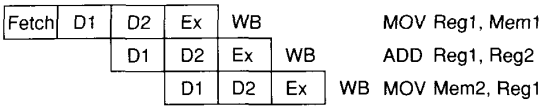

Figure 4. No data load delay in the pipeline.

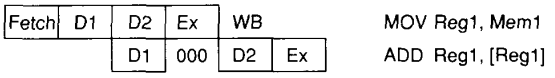

Figure 5. A pointer load delay.

Ex stage. A microprogram in the microcode ROM controls the actions of the main execution stage of the i486 CPU; hardwired logic controls all other stages.

Instructions that reference memory (including jump instructions) access the cache in this stage. Instructions that load data from memory use one Ex clock cycle, upon a cache hit. Cache lookup proceeds in parallel with the TLB lookup phase.

Instructions that perform arithmetic in the ALU using values from on-chip registers or immediate constants enter the Ex stage to read data from registers, compute the necessary function, and latch the results.

WB stage. This stage updates the register file either with 1) data read from the cache or main memory or 2) a result from the ALU output latch. We included the necessary bypass paths to avoid stalling the pipeline should the WB stage of one instruction write to a register that is used in the Ex stage of the next instruction. These stages occur at the same time for the two successive instructions.

No data load delay. Most RISC machines pipeline memory access to an off-chip cache. As a result, the data becomes available from a load instruction too late to be used by the next instruction, even if a cache hit occurs. Instead, the data generally becomes available after a delay of one[12-14] or two[11] clock cycles. This phenomenon, known as a load delay, means that for maximum performance these processors require the compiler to reorder instructions so that unrelated instructions can be inserted between a memory load and an instruction that uses the new data.

We had to take care to avoid a load delay in the i486 CPU, since most of the software to run on the processor exists as "shrink-wrapped" disks developed long before the CPU was available. So, we could not rely on a compiler to make up for any load delays. Instead, we wanted to ensure that data loaded by one instruction could be used in the next instruction. The CPU pipeline supports this requirement, as illustrated in the three-instruction sequence shown in Figure 4.

The first instruction in the figure (the MOV Reg1,Mem1 load instruction) moves data from the memory location given by Mem1 into register Reg1. The second instruction (ADD Reg1,Reg2) adds the contents of register Reg2 to this new value in register Reg1 and leaves the result in Reg1.

The load instruction accesses the cache in its Ex clock cycle. With a cache hit, the data becomes available at the end of the Ex cycle. The CPU writes the data into the register during the WB cycle, which occurs at the same time as the Ex cycle of the add instruction needing this new data. Because the CPU contains the necessary bypass paths, it can write this data to the register file and forward it to the ALU at the same time, avoiding a load delay.

The third (MOV Mem2,Reg1) instruction also has a potential load delay. The value computed in the previous instruction (and stored into Reg1) must be available in the Ex stage of this store instruction so that it can be written into the cache. Again, the data moves along the necessary bypass path from the WB stage of the previous instruction.

The on-chip cache and computation of memory addresses in the D2 pipeline stage eliminated the load delay. Having the cache on chip allows a full cache operation to complete in one clock cycle while minimizing the latency that must be covered by pipelining. By arranging the pipeline so that this one cycle of cache lookup occurs in the Ex stage of the pipeline, we eliminated the load delay. To arrange the pipeline this way required that memory addresses be computed before the Ex stage; the CPU computes them in the D2 stage. As this D2 stage was needed anyway for pipelining the instruction decoding, we eliminated the load delay without adding extra stages to the pipeline. Processors with off-chip caches require at least 1.5 cycles for a cache access and typically don't require our extra decoding stage. As a result, they require a load delay of one or two cycles.

Pointer load delay. Although the CPU does not have a load delay for data operations, it does require a load delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register, and that register is then used as a base register in the next instruction, the CPU will stall for one cycle. The delay occurs because of a pipeline collision between the WB stage of the load and the D2 stage of the instruction using the base register. These stages collide because the CPU pipeline computes memory addresses during the D2 stage, requiring that the base register be accessed one clock cycle earlier than if it were used as data, as in the previous example.

Figure 5 illustrates the pointer load delay with a two-instruction sequence that loads a value from memory into a register. It then uses the register as the base

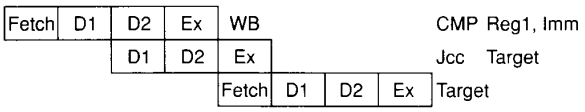

Figure 6. Jump instruction timing.

Table 3.
Clock cycle counts for basic instructions.

Instruction type	Counts 386	486	Sparc	88100
Load	4	1	2	1-3
Store	2	1	3	1
ALU	2	1	1	1
Jump taken/	9	3	1	2
not taken	3	1	2	1
Call	9	3	3	1

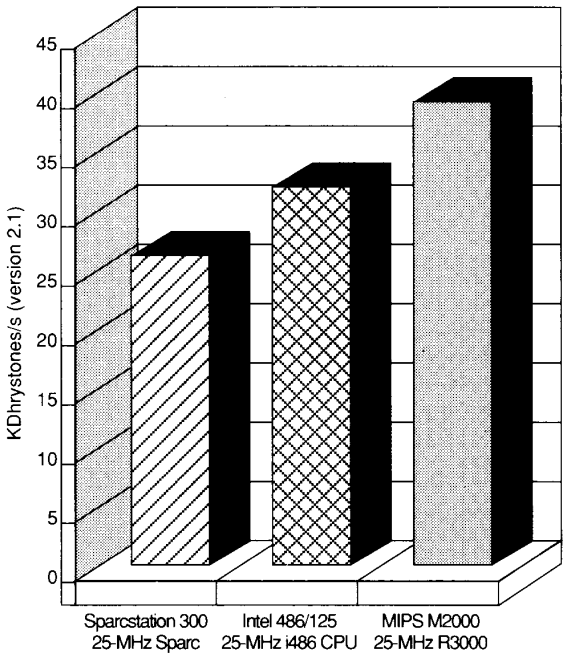

Figure 7. Selected processor performance results from Dhrystone version 2.1 tests.

register in the next instruction, which also is a load instruction. In this case, the CPU accesses the cache in the Ex stage of the first instruction and stores the value retrieved (assuming a hit) in the register during the WB stage. However, the next instruction needs this register in its D2 stage, requiring the CPU to stall for one cycle. When the D2 stage lines up with the WB stage of the previous instruction, the WB-to-D2 bypass path can be used to provide the data for the memory address computation.

Jump instruction timing. Figure 6 illustrates the timing of a jump instruction, assuming that the jump is taken—that is, assuming that the CPU evaluates the jump condition as true. The first instruction in the sequence is a compare instruction, which sets the condition code to condition the jump. The compare operation occurs in the Ex stage, and the CPU writes it to the condition flags in the WB stage. A bypass path makes the condition flags available in the Ex stage of the next instruction, which is a conditional jump. During the Ex stage of the jump, the CPU evaluates the condition to determine if the jump should be taken.

In parallel, the CPU runs a speculative fetch cycle to the target of the jump during the Ex stage of the jump instruction. If the CPU determines a false jump condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded). In this case (a false jump condition), the jump executes in just one Ex cycle.

If the CPU evaluates the condition as true, the fetch to the target address refills the pipeline. As shown in Figure 6, this means that the Ex stage of the jump instruction corresponds to the Fetch stage of the target. By lining up the pipeline stages of the two instructions, we see that the target instruction reaches the Ex stage three clock cycles after the jump.

In summary, conditional jumps take one clock cycle for a jump that is not taken, and three cycles for a taken jump. Unconditional jump and call instructions also take three cycles.

Performance summary. Table 3 summarizes the clock cycle counts for the most frequently executed instructions for the 386, i486 CPU, Sun Microsystems Sparc,[12] and Motorola 88100[11] processors. Note that the i486 CPU achieves a reduction in cycle counts of between two and four times over the 386 CPU. The i486 CPU does a little better than the RISC processors for data-manipulation instructions but executes a little slower on branches.

Figure 7 shows the relative performance of these processors on the popular Dhrystone benchmark version 2.1. An Intel System 486/125 (Multibus II, 8-Mbyte memory, no external cache, Unix System V release 3.2) running the benchmark as compiled by the MetaWare C compiler, beta release 2.2c, produced the

i486 CPU numbers. We took the Sun Sparcstation 300 and Mips M2000 numbers from published benchmark reports.[15, 16]

The fastest available (at the time of this writing) computer systems containing the various processors produced the benchmark results. Several vendors, including Intel, have announced future availability of faster versions of all of the processors in this chart. However, we had to restrict the benchmark exercise to complete systems available for running the benchmarks.

Figure 8 shows the relative performance of the i486 CPU to several RISC processors on the C programs in the SPEC benchmark suite.[17] This suite contains 10 programs intended to measure the performance of a workstation running "typical" engineering workstation activities. Four of these programs are written in C and emphasize integer performance. They therefore provide a good measure of the effectiveness of the integer performance techniques discussed in this article. The remaining six programs in the SPEC suite are floating-point-intensive Fortran applications not discussed here.

A Compaq Deskpro 486/25, Model 650 (25-MHz i486 CPU, 128-Kbyte cache, 16-Mbyte RAM) running ISC Unix V release 3.2 and using the MetaWare C compiler release 2.2c produced the numbers for the i486 CPU. (Compaq's SPEC license number is 136.) These results—obtained on preproduction hardware and a beta release of the MetaWare compiler—are preliminary and may not reflect the actual results received when the SPEC benchmark runs on a production-level system. By the time this article appears in print, vendors expect production quantities of the system and compiler to be available.

The numbers for the Sun Sparcstation 330 and Mips M2000 systems appeared in the first issue of the *SPEC Newsletter*.[17] Note that all three of the systems ran at a 25-MHz clock rate.

These benchmark results confirm the performance numbers implied by the instruction clock cycle counts listed in Table 3. They show the performance level of the i486 CPU to be competitive with current RISC-based systems.

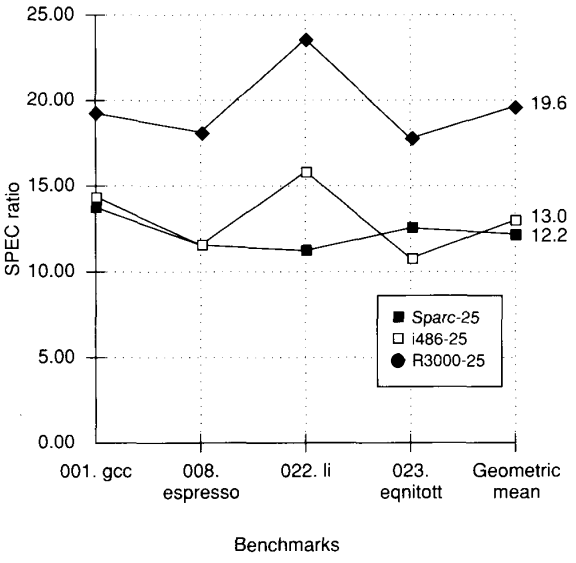

Figure 8. Performance results from C programs in the SPEC benchmark suite.

Although the implementation of the i486 CPU resembles in many ways several popular RISC implementations, certain key decisions strongly affected our final design. Our need to be compatible with the PC standard instruction set and our highly integrated implementation drove these decisions. They are:

1) Hardware had to maintain cache consistency, with no operating system intervention.

2) We chose a unified cache (code and data combined) rather than having separate caches for code and data.

3) Integrating the cache on chip allowed us to avoid a data load delay, an important design factor.

The i486 CPU provides a performance level similar to today's RISC processors yet retains compatibility with a large base of existing software. We accomplished this level of performance along with compatibility by carefully integrating a cache into the instruction pipeline to achieve a sustained execution rate of one instruction per clock cycle.

Acknowledgments

The i486 CPU demanded a very large effort, involving many groups of people throughout Intel Corporation. For the implementation choices just described, I would especially like to recognize Beatrice Fu, Pat Gelsinger, Ed Grochowski, and Ken Shoemaker from the chip design team. I thank Compaq Computer Corp. and MetaWare Inc. for their timely assistance in producing the SPEC benchmark numbers.

References

1. J. Crawford and P. Gelsinger, *Programming the 80386*, Sybex Books, Alameda, Calif., 1987.
2. Intel Corp., *80386 Programmer's Reference Manual*, Santa Clara, Calif., 1986.
3. J. Crawford, "Architecture of the Intel 80386," *Proc. ICCD 86*, Oct. 1986, pp. 155-160.
4. Intel Corporation, *i486 Microprocessor* (data book), 1989.
5. E. Grochowski and K. Shoemaker, "Issues in the Implementation of the 486 Cache and Bus," *Proc. ICCD 89*, IEEE Computer Society, Los Alamitos, Calif., Oct. 1989, pp. 193-198.
6. A.J. Smith, "Cache Memories," *ACM Computing Surveys*, New York, Vol. 14, No. 3, Sept. 1982, pp. 473-530.
7. M. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," Tech. Report 823, Computer Science Dept., University of Wisconsin, Madison, Feb. 1989.
8. S. Przybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, May 1988, pp. 290-298.
9. A.J. Smith, "Line (Block) Size Choice for CPU Cache Memories," *IEEE Trans. Computers*, Vol. C-36, No. 9, Sept. 1987, pp. 1063-1075.
10. A.J. Smith, "Cache Evaluation and the Impact of Workload Choice," *Proc. 12th Ann. Symp. Computer Architecture*, June 1985, pp. 64-73.
11. *MC88100 User's Manual*, Motorola Inc., Phoenix, Ariz., 1988.
12. *MB86900 High Performance 32-Bit RISC Processor Data Sheet*, Fujitsu Microelectronics Inc., Santa Clara, Calif., July 1987.
13. G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N. J., 1987.
14. *PaceMips R3000 32-Bit, 25MHz RISC CPU with Integrated Memory Management Unit* (data sheet), Performance Semiconductor, Sunnyvale, Calif., 1989.
15. *Performance Brief*, Mips Computer Systems, Inc., June 1989.
16. *DEC Benchmark Report*, Digital Equipment Corp., Maynard, Mass., July 11, 1989.
17. *SPEC Newsletter*, "Benchmark Results," Vol. 1, Issue 1, Waterside Associates, Freemont, Calif., Fall 1989.

John H. Crawford, the chief architect of the 386 and i486 microprocessors, manages the 386 Family Architecture for Intel Corp. in Santa Clara, California. He has been involved with the 386 from its inception in 1982 and previously worked in compiler development. His research interests include high-performance CPU design, multiprocessor systems, computer architecture, microprogram verification, and compiler technology.

Crawford received a BS degree from Brown University and an MS from the University of North Carolina, Chapel Hill, both in computer science. Author of several papers on compiler technology and microprocessor architecture, he also coauthored *Programming the 80386*. He is a member of the IEEE and serves on the Editorial Board of *IEEE Micro*.

Readers may direct questions concerning this article to the author at Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051.