

Architecture of the Pentium Microprocessor

The Pentium CPU is the latest in Intel's family of compatible microprocessors. It integrates 3.1 million transistors in 0.8- μ m BiCMOS technology. We describe the techniques of pipelining, superscalar execution, and branch prediction used in the microprocessor's design.

Donald Alpert

Dror Avnon

Intel Corporation

The Pentium processor is Intel's next generation of compatible microprocessors following the popular i486 CPU family. The design started in early 1989 with the primary goal of maximizing performance while preserving software compatibility within the practical constraints of available technology. The Pentium processor integrates 3.1 million transistors in 0.8- μ m BiCMOS technology and carries the Intel trademark. We describe the architecture and development process employed to achieve this goal.

Technology

The continual advancement of semiconductor technology promotes innovation in microprocessor design. Higher levels of integration, made possible by reduced feature sizes and increased interconnection layers, enable designers to deploy additional hardware resources for more parallel computation and deeper pipelining. Faster device speeds lead to higher clock rates and consequently to requirements for larger and more specialized on-chip memory buffers.

Table 1 (next page) summarizes the technology improvements associated with our three most recent microprocessor generations. The 0.8- μ m BiCMOS technology of the Pentium microprocessor enables 2.5 times the number of transistors and twice the clock frequency of the original i486 CPU, which was implemented in 1.0- μ m CMOS.

Compatibility

Since introduction of the 8086 microprocessor in 1978, the X86 architecture has evolved through several generations of substantial functional enhancements and technology improvements, including the 80286 and i386 CPUs. Each of these CPUs was supported by a corresponding floating-point unit. The i486 CPU,¹ introduced in 1989, integrates the complete functionality of an integer processor, floating-point unit, and cache memory into a single circuit.

The X86 architecture greatly appealed to software developers because of its widespread application as the central processor of IBM-compatible personal computers. The success of the architecture in PCs has in turn made the X86 popular for commercial server applications as well. Figure 1 shows some of the well-known software environments that are hosted on the architecture.

The common software environments allow the X86 architecture to exercise several operating modes. Applications developed for DOS use 16-bit real mode (or virtual 8086 mode) and MS Windows. Early versions of OS/2 use 16-bit protected mode, and applications for other popular environments use 32-bit flat (unsegmented) mode. The Pentium microprocessor employs general techniques for improving performance in all operating modes, as well as certain techniques for improving performance in specific operating

| Table 1. Technology for microprocessor development. | | | | |
|---|------|--|--------------------|-----------------|
| Microprocessor | Year | Technology | No. of transistors | Frequency (MHz) |
| i386 CPU | 1986 | 1.5- μ m CMOS, two-layer metal | 275K | 16 |
| i486 CPU | 1989 | 1.0- μ m CMOS, two-layer metal | 1.2M | 33 |
| Pentium CPU | 1993 | 0.8- μ m BiCMOS, three-layer metal | 3.1M | 66 |

| 16-bit generation | 32-bit generation |
|-------------------|-------------------|
| DOS | Unix SVR4 |
| MS-Windows | SCO |
| OS/2 1.x | OSF/1 |
| | Netware 3.11 |
| | Next Step |
| | 32-bit OS/2 |
| | Solaris |
| | Windows NT |
| | Univel |
| | Taligent |
| 1980s | 1991 199x |

Figure 1. Software environments. (All figures, tables, and photographs published in this article are the property of Intel Corporation.)

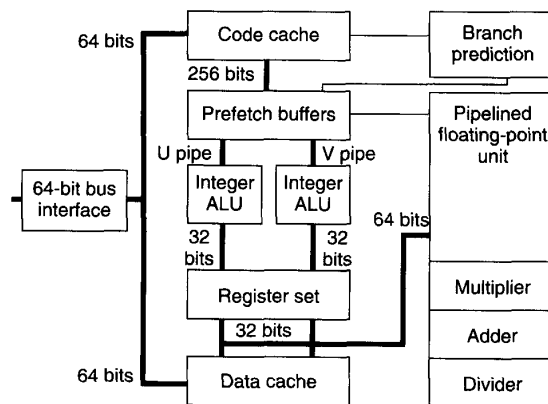


Figure 2. Pentium processor block diagram.

modes. We focus on the 32-bit flat mode here, since this is the most appropriate mode for comparison with the other high-performance microprocessors described at the Hot Chips IV Conference.

The X86 architecture supports the IEEE-754 standard for floating-point arithmetic.² In addition to required operations on single-precision and double-precision formats, the X86 floating-point architecture includes operations on 80-bit, extended-precision format and a set of basic transcendental functions.

Pentium CPU designers found numerous exciting technical challenges in developing a microarchitecture that

maintained compatibility with such a diverse software base. Later in this article we present examples of techniques for supporting self-modifying code and the stack-oriented, floating-point register file.

Performance

A microprocessor's performance is a complex function of many parameters that vary between applications, compilers, and hardware systems. In developing the Pentium microprocessor, the design team addressed these aspects for each of the popular software environments. As a result, Pentium CPU features tuned compilers and cache memory.

We focus on the performance of SPEC benchmarks for both the Pentium microprocessor and i486 CPU in systems with well-tuned compilers and cache memory. More specifically, the Pentium CPU achieves roughly two times the speedup on integer code and up to five times the speedup on floating-point vector code when compared with an i486 CPU of identical clock frequency.

Organization

Figure 2 shows the overall organization of the Pentium microprocessor. The core execution units are two integer pipelines and a floating-point pipeline with dedicated adder, multiplier, and divider. Separate on-chip instruction code and data caches supply the memory demands of the execution units, with a branch target buffer augmenting the instruction cache for dynamic branch prediction. The external interface includes separate address and 64-bit data buses.

Integer pipeline

The Pentium processor's integer pipeline is similar to that of the i486 CPU.³ The pipeline has five stages (see Figure 3) with the following functions:

- *Prefetch*. During the PF stage the CPU prefetches code from the instruction cache and aligns the code to the

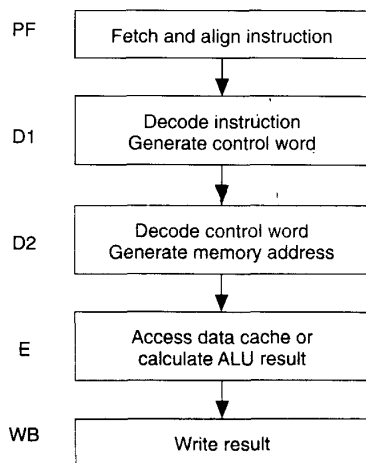


Figure 3. Integer pipeline.

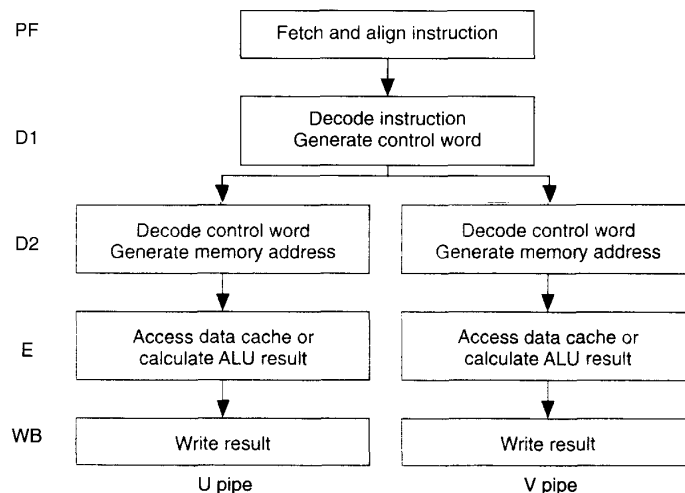


Figure 4. Superscalar execution.

initial byte of the next instruction to be decoded. Because instructions are of variable length, this stage includes buffers to hold both the line containing the instruction being decoded and the next consecutive line.

- **First decode.** In the D1 stage the CPU decodes the instruction to generate a control word. A single control word executes instructions directly; more complex instructions require microcoded control sequencing in D1.
- **Second decode.** In the D2 stage the CPU decodes the control word from D1 for use in the E stage. In addition, the CPU generates addresses for data memory references.
- **Execute.** In the E stage the CPU either accesses the data cache or calculates results in the ALU (arithmetic logic unit), barrel shifter, or other functional units in the data path.
- **Write back.** In the WB stage the CPU updates the registers and flags with the instruction's results. All exceptional conditions must be resolved before an instruction can advance to WB.

Compared to the integer pipeline of the i486 CPU, the Pentium microprocessor integrates additional hardware in several stages to speed instruction execution. For example, the i486 CPU requires two clocks to decode several instruction formats, but the Pentium CPU takes one clock and executes shift and multiply instructions faster. More significantly, the Pentium processor substantially enhances superscalar execution, branch prediction, and cache organization.

Superscalar execution. The Pentium CPU has a superscalar organization that enables two instructions to execute

in parallel. Figure 4 shows that the resources for address generation and ALU functions have been replicated in independent integer pipelines, called U and V. (The pipeline names were selected because U and V were the first two consecutive letters of the alphabet neither of which was the initial of a functional unit in the design partitioning.) In the PF and D1 stages the CPU can fetch and decode two simple instructions in parallel and issue them to the U and V pipelines. Additionally, for complex instructions the CPU in D1 can generate microcode sequences that control both U and V pipelines.

Several techniques are used to resolve dependencies between instructions that might be executed in parallel. Most of the logic is contained in the instruction issue algorithm (see Figure 5) of D1.

```

Decode two consecutive instructions: I1 and I2
If the following are all true
    I1 is a "simple" instruction
    I2 is a "simple" instruction
    I1 is not a jump instruction
    Destination of I1 ≠ source of I2
    Destination of I1 ≠ destination of I2
Then issue I1 to U pipe and I2 to V pipe
Else issue I1 to U pipe
  
```

Figure 5. Instruction issue algorithm.

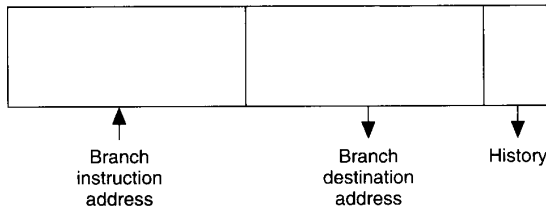


Figure 6. Branch target buffer.

Resource dependencies. A resource dependency occurs when two instructions require a single functional unit or data path. During the D1 stage, the CPU only issues two instructions for parallel execution if both are from a class of “simple” instructions, thereby eliminating most resource dependencies. The instructions must be directly executed, that is, not require microcode sequencing. The instruction being issued to the V pipe can be an ALU operation, memory reference, or jump. The instruction being issued to the U pipe can be from the same categories or from an additional set that uses a functional unit available only in the U pipe, such as the barrel shifter. Although the set of instructions identified as “simple” might seem restrictive, more than 90 percent of instructions executed in the Integer SPEC benchmark suite are simple.

Data dependencies. A data dependency occurs when one instruction writes a result that is read or written by another instruction. Logic in D1 ensures that the source and destination registers of the instruction issued to the V pipe differ from the destination register of the instruction issued to the U pipe. This arrangement eliminates read-after-write (RAW) and write-after-write (WAW) dependencies. Write-after-read (WAR) dependencies need not be checked because reads occur in an earlier stage of the pipelines than writes.

The design includes logic that enables instructions with certain special types of data dependency to be executed in parallel. For example, a conditional branch instruction that tests the flag results can be executed in parallel with a compare instruction that sets the flags.

Control dependencies. A control dependency occurs when the result of one instruction determines whether another instruction will be executed. When a jump instruction is issued to the U pipe, the CPU in D1 never issues an instruction to the V pipe, thereby eliminating control dependencies.

Note that resource dependencies and data dependencies between memory references are not resolved in D1. Dependent memory references can be issued to the two pipelines; we explain their resolution in the description of the data cache.

Branch prediction. The i486 CPU has a simple technique for handling branches. When a branch instruction is executed, the pipeline continues to fetch and decode instructions along the sequential path until the branch reaches the E stage. In E, the CPU fetches the branch destination, and the pipeline resolves whether or not a conditional branch is taken. If the branch is not taken, the CPU discards the fetched destination, and execution proceeds along the sequential path with no delay. If the branch is taken, the fetched destination is used to begin decoding along the target path with two clocks of delay. Taken branches are found to be 15 percent to 20 percent of instructions executed, representing an obvious area for improvement by the Pentium processor.

The Pentium CPU employs a branch target buffer (BTB), which is an associative memory used to improve performance of taken branch instructions (see Figure 6). When a branch instruction is first taken, the CPU allocates an entry in the branch target buffer to associate the branch instruction's address with its destination address and to initialize the history used in the prediction algorithm. As instructions are decoded, the CPU searches the branch target buffer to determine whether it holds an entry for a corresponding branch instruction. When there is a hit, the CPU uses the history to determine whether the branch should be taken. If it should, the microprocessor uses the target address to begin fetching and decoding instructions from the target path. The branch is resolved early in the WB stage, and if the prediction was incorrect, the CPU flushes the pipeline and resumes fetching along the correct path. The CPU updates the dual-ported history in the WB stage. The branch target buffer holds entries for predicting 256 branches in a four-way associative organization.

Using these techniques, the Pentium CPU executes correctly predicted branches with no delay. In addition, conditional branches can be executed in the V pipe paired with a compare or other instruction that sets the flags in the U pipe. Branching executes with full compatibility and no modification to existing software. (We explain aspects of interactions between branch prediction and self-modifying code later.)

Cache organization. The i486 CPU employs a single on-chip cache that is unified for code and data. The single-ported cache is multiplexed on a demand basis between sequential code prefetches of complete lines and data references to individual locations. As just explained, branch targets are prefetched in the E stage, effectively using the same hardware as data memory references. There are potential advantages for such an organization over one that separates code and data.

- 1) For a given size of cache memory, a unified cache has a higher hit rate than separate caches because it balances the total allocation of code and data lines automatically.
- 2) Only one cache needs to be designed.
- 3) Handling self-modifying code can be simpler.

Despite these potential advantages of a unified cache, all of which apply to the i486 CPU, the Pentium microprocessor uses separate code and data caches. The reason is that the superscalar design and branch prediction demand more bandwidth than a unified cache similar to that of the i486 CPU can provide. First, efficient branch prediction requires that the destination of a branch be accessed simultaneously with data references of previous instructions executing in the pipeline. Second, the parallel execution of data memory references requires simultaneous accesses for loads and stores. Third, in the context of the overall Pentium microprocessor design, handling self-modifying code for separate code and data caches is only marginally more complex than for a unified cache.

The instruction cache and data cache are each 8-Kbyte, two-way associative designs with 32-byte lines.

Programs executing on the i486 CPU typically generate more data memory references than when executing on RISC microprocessors. Measurements on Integer SPEC benchmarks show 0.5 to 0.6 data references per instruction for the i486 CPU⁴ and only 0.17 to 0.33 for the Mips processor.⁵ This difference results directly from the limited number (eight) of registers for the X86 architecture, as well as procedure-calling conventions that require passing all parameters in memory. A small data cache is adequate to capture the locality of the additional references. (After all, the additional references have sufficient locality to fit in the register file of the RISC microprocessors.) The Pentium microprocessor implements a data cache that supports dual accesses by the U pipe and V pipe to provide additional bandwidth and simplify compiler instruction scheduling algorithms.

Figure 7 shows that the address path to the translation look-aside buffer and data cache tags is a fully dual-ported structure. The data path, however, is single ported with eight-way interleaving of 32-bit-wide banks. When a bank conflict occurs, the U pipe assumes priority, and the V pipe stalls for a clock cycle. The bank conflict logic also serves to eliminate data dependencies between parallel memory references to a single location. For memory references to double-precision floating-point data, the CPU accesses consecutive banks in parallel, forming a single 64-bit path.

The design team considered a fully dual-ported structure for the data cache, but feasibility studies and performance simulations showed the interleaved structure to be more effective. The dual-ported structure eliminated bank conflicts, but the SRAM cell would have been larger than the cell used in the interleaved scheme, resulting in a smaller cache and lower hit ratio for the allocated area. Additionally, the handling of data dependencies would have been more complex.

With a write-through cache-consistency protocol and 32-bit data bus, the i486DX2 CPU uses buses 80 percent of the time; 85 percent of all bus cycles are writes. (The i486DX2 CPU has a core pipeline that operates at twice the bus clock's

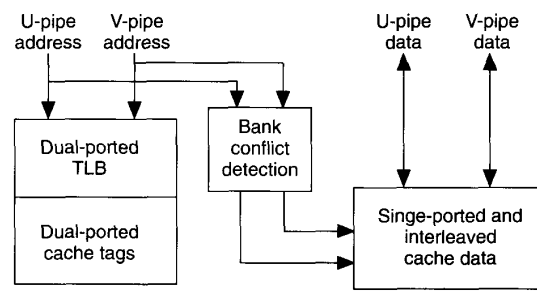


Figure 7. Dual-access data cache.

frequency.) For the Pentium microprocessor, with its higher performance core pipelines and 64-bit data bus, using a write-back protocol for cache consistency was an obvious enhancement. The write-back protocol uses four states: modified, exclusive, shared, and invalid (MESI).

Self-modifying code. One challenging aspect of the Pentium microprocessor's design was supporting self-modifying code compatibly. Compatibility requires that when an instruction is modified followed by execution of a taken branch instruction, subsequent executions of the modified instruction must use the updated value. This is a special form of dependency between data stores and instruction fetches.

The interaction between branch predictions and self-modifying code requires the most attention. The Pentium CPU fetches the target of a taken branch before previous instructions have completed stores, so dedicated logic checks for such conditions in the pipeline and flushes incorrectly fetched instructions when necessary. The CPU thoroughly verifies predicted branches to handle cases in which an instruction entered in the branch target buffer might be modified. The same mechanisms used for consistency with external memory maintain consistency between the code cache and data cache.

Floating-point pipeline

The i486 CPU integrated the floating-point unit (FPU) on chip, thus eliminating overhead of the communication protocol that resulted from using a coprocessor. Bringing the FPU on chip substantially boosted performance in the i486 CPU. Nevertheless, due to limited devices available for the FPU, its microarchitecture was based on a partial multiplier array and a shift-and-add data path controlled by microcode. Floating-point operations could not be pipelined with any other floating-point operations; that is, once a floating-point instruction is invoked, all other floating-point instructions stall until its completion.

The larger transistor budget available for the Pentium microprocessor permits a completely new approach in the design of the floating-point microarchitecture. The aggressive

| | | | | | | | | |
|---------------------|----|----|----|---|----|----|----|----|
| Integer pipe | PF | D1 | D2 | E | WB | | | |
| Floating-point pipe | PF | D1 | D2 | E | X1 | X2 | WF | ER |

Figure 8. Floating-point pipeline.

performance goals for the FPU presented an exciting challenge for the designers, even with more silicon resources available. Furthermore, maintaining full compatibility with previous products and with the IEEE standard for floating-point arithmetic was an uncompromising requirement.

Floating-point pipeline stages. Pentium's floating-point pipeline consists of eight stages. The first two stages are processed by the common (integer pipeline) resources for prefetch and decode. In the third stage the floating-point hardware begins activating logic for instruction execution. All of the first five stages are matched with their counterpart integer pipeline stages for pipeline sequencing and synchronization (see Figure 8).

- *Prefetch.* The PF stage is the same as in the integer pipeline.
- *First decode.* The D1 stage is the same as in the integer pipeline.
- *Second decode.* The D2 stage is the same as in the integer pipeline.
- *Operand fetch.* In this E stage the FPU accesses both the data cache and the floating-point register file to fetch the operands necessary for the operation. When floating-point data is to be written to the data cache, the FPU converts internal data format into the appropriate memory representation. This stage matches the E stage of the integer pipeline.
- *First execute.* In the X1 stage the FPU executes the first steps of the floating-point computation. When floating-point data is read from the data cache, the FPU writes the incoming data into the floating-point register file.
- *Second execute.* In the X2 stage the FPU continues to execute the floating-point computation.
- *Write float.* In the WF stage the FPU completes the execution of the floating-point computation and writes the result into the floating-point register file.
- *Error reporting.* In the ER stage the FPU reports internal special situations that might require additional processing to complete execution and updates the floating-point status word.

The eight-stage pipeline in the FPU allows a single cycle throughput for most of the "basic" floating-point instructions such as floating-point add, subtract, multiply, and compare. This means that a sequence of basic floating-point instructions free from data dependencies would execute at a rate of

one instruction per cycle, assuming instruction cache and data cache hits.

Data dependencies exist between floating-point instructions when a subsequent instruction uses the result of a preceding instruction. Since the actual computation of floating-point results takes place during X1, X2, and WF stages, special paths in the hardware allow other stages to be bypassed and present the result to the subsequent instruction upon generation. Consequently, the latency of the basic floating-point instructions is three cycles.

The X86 floating-point architecture supports single-precision (32-bit), double-precision (64-bit), and extended-precision (80-bit) floating-point operations. We chose to support all computation for the three precisions directly, by extending the data path width to support extended precision. Although this entailed using more devices for the implementation, it greatly simplified the microarchitecture while improving the performance. If smaller data paths were designed, special rerouting of the data within the FPU and several state machines or microcode sequencing would have been required for calculating the higher precision data.

Floating-point instructions execute in the U pipe and generally cannot be paired with any other integer or floating-point instructions (the one exception will be explained later). The design was tuned for instructions that use one 64-bit operand in memory with the other operand residing in the floating-point register file. Thus, these operations may execute at the maximum throughput rate, since a full stage (E stage) in the pipeline is dedicated to operand fetching. Although floating-point instructions use the U pipe during the E stage, the two ports to the data cache (which are used by the U pipe and the V pipe for integer operations) are used to bring 64-bit data to the FPU. Consequently, during intensive floating-point computation programs, the data cache access ports of the U pipe and V pipe operate concurrently with the floating-point computation. This behavior is similar to superscalar load-store RISC designs where load instructions execute in parallel with floating-point operations, and therefore deliver equivalent throughput of floating-point operations per cycle.

Microarchitecture overview. The floating-point unit of the Pentium microprocessor consists of six functional sections (see Figure 9).

The floating-point interface, register file, and control (FIRC) section is the only interface between the FPU and the rest of the CPU. Since the function of floating-point operations is usually self-contained within the floating-point computation core, concentrating all the interface logic in one section helped to create a modular design of the other sections. The FIRC section also contains most of the common floating-point resources: register file, centralized control logic, and safe instruction recognition logic (described later). FIRC can complete execution of instructions that do not need arithmetic compu-

tation. It dispatches the instructions requiring arithmetic computation to the arithmetic sections.

The floating-point exponent section (FEXP) calculates the exponent and the sign results for all the floating-point arithmetic operations. It interfaces with all the other arithmetic sections for all the necessary adjustments between the mantissa and the sign-and-exponent fields in the computation of floating-point results.

The floating-point multiplier section (FMUL) includes a full multiplier array to support single-precision (24-bit mantissa), double-precision (53-bit mantissa), and extended-precision (64-bit mantissa) multiplication and rounding within three cycles. FMUL executes all the floating-point multiplication operations. It is also used for integer multiplication, which is implemented through microcode control.

The floating-point adder section (FADD) executes all the "add" floating-point instructions, such as floating-point add, subtract, and compare. FADD also executes a large set of micro-operations that are used by microcode sequences in the calculation of complex instructions, such as binary coded decimal (BCD) operations, format conversions, and transcendental functions. The FADD section operates during the X1 and X2 stages of the floating-point pipeline and employs several wide adders and shifters to support high-speed arithmetic algorithms while maintaining maximum performance for all data precisions. The CPU achieves a latency of three cycles with a throughput of one cycle for all the operations directly executed by the FADD section for single-precision, double-precision, and extended-precision data.

The floating-point divider (FDIV) section executes the floating-point divide, remainder, and square-root instructions. It operates during the X1 and X2 pipeline stages and calculates two bits of the divide quotient every cycle. The overall instruction latency depends on the precision of the operation. FDIV uses its own sequencer for iterative computation during the X1 stage. The results are fully accurate in accordance with IEEE standard 754 and ready for rounding at the end of the X2 stage.

The floating-point rounder (FRND) section rounds the results delivered from the FADD and FDIV sections. It operates during the WF stage of the floating-point pipeline and delivers a rounded result according to the precision control and the rounding control, which are specified in the floating-point control word.

Safe instruction recognition. Floating-point computation requires longer execution times than integer computation. Pentium's floating-point pipeline uses eight stages, while the integer pipeline uses only five stages. Compatibility requires in-order instruction execution as well as precise exception reporting. To meet these requirements in the Pentium processor, floating-point instructions should not proceed beyond the X1 stage, that is, allow subsequent instructions to proceed beyond the E stage, unless the floating-point instruction is guaranteed to complete without causing an ex-

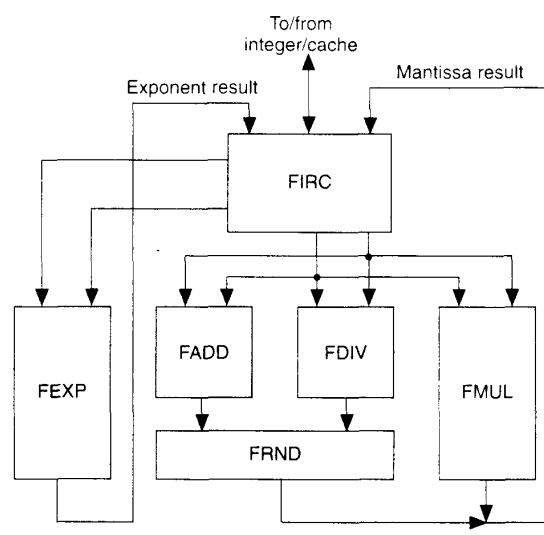


Figure 9. Floating-point unit block diagram.

ception. Otherwise, an instruction may change the state of the CPU, while an earlier floating-point instruction (which has not yet completed) might cause an exception that requires a trap to a software exception handler.

To avoid a substantial performance loss due to stalling instructions until the exception status of a previous floating-point instruction is known, Pentium's floating-point unit employs a mechanism called safe instruction recognition (SIR). This logic determines whether a floating-point instruction is guaranteed to complete without creating an exception and therefore is considered "safe." If an instruction is safe, there is no need to stall the pipeline, and the maximum throughput can be obtained. If, however, the instruction is not safe, the pipeline stalls for three cycles until the unsafe instruction reaches the ER stage and a final determination of the exception status is made.

Six possible exceptions can occur on the Pentium microprocessor's floating-point operations: invalid operation, divide by zero, denormal operand, overflow, underflow, and inexact. The SIR logic needs to determine early in the floating-point pipeline—in the X1 stage—before any computation takes place whether the instruction is guaranteed to be exception free (safe) or not (unsafe). The first three of the six exceptions can be detected without any floating-point calculation. From the latter three exceptions, the inexact exception is usually "masked" by the operating system or the software application (using the precision mask, or PM, bit in the floating-point control word). Otherwise, a trap will occur whenever rounding of the result is necessary. When the pre-

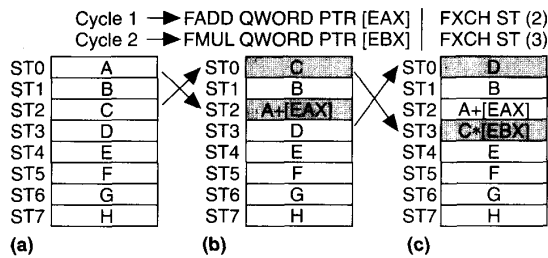


Figure 10. FXCH code example.

cision (inexact) exception is masked, the pipeline delivers the correctly rounded result directly. For overflow and underflow exceptions SIR logic uses an algorithm that monitors the exponent fields of the input operands to conclude the exception status (safe or unsafe).

In the X86 architecture the CPU stores floating-point operands in the floating-point register file with an extended-precision exponent, regardless of the precision control in the floating-point control word. The extended-precision exponent supports much greater range than the double-precision format. Overflow and underflow exceptions caused by converting the data into double-precision or single-precision formats occur only when storing the data into external memory. These characteristics of the X86 floating-point architecture give a unique advantage to the effectiveness of the SIR mechanism in the Pentium CPU, since the SIR algorithm can use the internal (extended-precision) exponent range. Thus, the occurrence of unsafe operations is extremely rare. Our evaluation of the SIR algorithm for the FPU design found no unsafe instructions in simulated execution of the SPEC89 floating-point benchmarks.

Register stack manipulation. The X86 floating-point instruction set uses the register file as a stack of eight registers in which the top of stack (TOS) acts as an accumulator of the results. Therefore, the top of the stack is used for the majority of the instructions as one of the source operands and, usually, as the destination register.

To improve the floating-point pipeline performance by optimizing the use of the floating-point register file, Pentium's FPU can execute the FXCH instruction in parallel with any basic floating-point operation. The FXCH instruction "swaps" the contents of the TOS register with another register in the floating-point register file. All the basic floating-point instructions may be paired with FXCH in the V pipe. The pair execute in parallel, even when data dependency between the two instructions in the pair exists. The use of parallel FXCH redirects the result of a floating-point operation to any selected register in the register file, while bringing a new operand to the top of the stack for immediate use by the next floating-point operation.

The example shown in Figure 10 illustrates the use of parallel FXCH. The code in the example generates the results of two independent floating-point calculations. The floating-point register file contains initial values prior to code execution: register ST0 (TOS) contains the value A, register ST1 contains value B, register ST2 contains value C, and so on. The two operations are

- 1) floating-point addition of value A with the 64-bit floating-point operand addressed by the general register EAX, and
- 2) floating-point multiplication of value C by the 64-bit floating-point operand addressed by the general register EBX.

When the floating-point pipeline is fully loaded and these two operations are part of the code sequence, the parallel FXCH allows the calculation to maintain the maximum throughput of one cycle per operation. Within one cycle the Pentium CPU writes the result of the addition to ST2, while the operand for the next operation moves to the top of the stack. On the next cycle, the processor writes the result of the multiplication to ST3, while the top of the stack contains value D, which may be used for a subsequent operation.

Transcendental instructions. The CPU supports all eight transcendental instructions that are defined in the instruction set through direct execution of microcode sequences. The transcendental instructions are

- | | |
|------------|-----------------------------|
| 1) FSIN | sine, |
| 2) FCOS | cosine, |
| 3) FSINCOS | sine and cosine, |
| 4) FPTAN | tangent, |
| 5) FPATAN | arctangent, |
| 6) F2XM1 | $2^{*}X - 1$, |
| 7) FYL2X | $Y * \text{Log}_2(X)$, and |
| 8) FYL2XP | $1 Y * \text{Log}_2(X+1)$ |

We developed new, table-driven algorithms for the transcendental functions using polynomial approximation techniques. These algorithms substantially improved performance and accuracy over the i486 CPU implementation, which used the more traditional CORDIC algorithms. The approximation tables reside in an on-chip ROM along with the other special constants that are used for floating-point computation.

The performance improvement of the transcendental instructions on the Pentium processor ranges from two to three times over the same instructions on the i486 CPU at the same frequency. The worst-case error for all the transcendental instructions is less than 1 ulp (unit in the last place) when rounding to nearest even and less than 1.5 ulps when rounding in other modes. The functions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

Development process

Developing a highly integrated microprocessor involves collaboration between numerous teams having diverse technical specialties and working under the discipline of well-defined methodologies. A small team of architects and VLSI designers developed the initial concepts of the design. This group conducted feasibility studies of parallel instruction decoding and options for branch prediction techniques. Simultaneously, it evaluated performance by hand for short benchmarks and compiler optimizations. As initial directions were established, additional engineers participated, and subteams focused on the following areas:

- 1) behavioral modeling of the microarchitecture;
- 2) circuit feasibility design for caches, decoding PLAs (programmable logic arrays), floating-point data path, and other critical functions;
- 3) a flexible, trace-driven simulator of instruction timing for performance evaluation;
- 4) a prototype compiler; and
- 5) enhancements to existing instruction-tracing tools.

Throughout the design we refined the Pentium microprocessor using both top-down and bottom-up methods. Top-down refinement was accomplished through comprehensive characterization of executing benchmark work loads on the i486 CPU⁴ and trace-driven experiments concerning alternative machine organizations conducted by architects using the performance simulator.

VLSI design engineers evaluating features critical to the targeted area and frequency refined the design from the bottom up. On two occasions in the design the accumulation of changes from bottom-up refinement caused the need for substantial restructuring of the microprocessor's global chip plan, or "die diets." On those occasions, interdisciplinary teams of specialists collaborated to brainstorm and evaluate ideas that could satisfy the global or local design constraints. In one instance, we found it necessary to refine the set of instructions that could be executed in parallel. Constraints had been assigned to the area and speed of the decoder PLAs. The VLSI designers identified combinations of instruction formats that would feasibly decode in parallel, and the compiler writers determined the optimal selection.

In the end, the measured performance of the Pentium microprocessor in production systems is within 2 percent of that predicted before the design was completed.

The logic validation of the Pentium processor design presented a major challenge to the design team. A comprehensive test base from the validation of previous X86 microprocessors was available. However, the Pentium processor microarchitecture introduced several new fundamental techniques, such as superscalar, write-back cache, and floating-point algorithms, that required a more rigorous veri-

Naming the Pentium processor

In naming the fifth generation of its compatible microprocessor line the Pentium processor, Intel departed from tradition. Pentium breaks a string of CPU products dating back to the late 1970s that used numerics (8086, 286, 386, 486).

"The natural course would be to call this chip the 586," said Andrew S. Grove, president and chief executive officer. "Unfortunately, we cannot trademark those numbers, which means that any company might call any chip a 586, even if it doesn't measure up to the real thing."

Pentium uses the Greek word for five, "pente," as its root to associate with the fifth-generation product and adds "-ium," a common ending from the periodic table of elements. Thus, the Pentium microprocessor is the fifth generation, a key element for future computing.



fication methodology.

We used different validation approaches in pre-silicon testing of the Pentium microprocessor:

- 1) Architecture verification looked at the "black box" functionality from the programmer's point of view. We designed comprehensive tests to cover all possible aspects of the programming model and all the Pentium processor user-visible features.

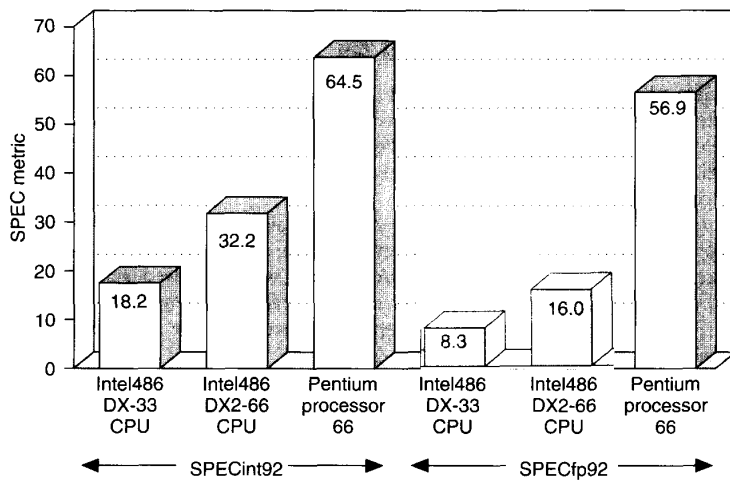


Figure 11. Pentium processor and i486 CPU performance for SPEC benchmarks.

- 2) Design verification checked the internal functionality from the point of view of a logic designer who would understand the behavior of every internal signal. This testing approach is considered a "white box" technique, in which tests are written to exercise all the internal logic and verify its correct behavior.
- 3) Random instruction testing was a valuable tool to cover all those situations that are rarely covered by the more traditional, handwritten tests. Running finely tuned random tests let us verify correct functionality by comparing the results generated by a logic design description of the Pentium processor to the results generated by a software-emulated model.
- 4) A logic-design hardware model (QuickTurn) enabled increased testing coverage capacity by allowing a much larger software base to run on the processor model before the first silicon was available. We ported the logic model of the Pentium processor onto a QuickTurn setup, which was capable of handling the complete design, and tested major operating systems and application programs before finalizing the design.

In addition to the general validation approach, we dedicated a special effort to verify the new algorithms employed by the FPU. We developed a high-level software simulator to evaluate the intricacies of the specific add, multiply, and divide algorithms used in the design. This simulator then evolved into a testing environment, allowing the verification of the FPU logic design model independently from the rest of the Pentium processor. Also, the new algorithms used for the

floating-point transcendental functions required an extensive test strategy that verified the accuracy and monotonicity of the results throughout the development process, comparing the results to a "super accurate" software model. Eventually, when the first silicon of the Pentium processor was available for testing, we used automatic testing techniques to assure the correctness of the transcendental instructions.

Compiler optimizations

The compiler technology developed with the Pentium microprocessor includes machine-independent optimizations common to current high-performance compilers, such as inlining, unrolling, and other loop transformations. In addition, we used techniques specifically developed for the X86 architecture and tuned them for the Pentium processor's microarchitecture.

The X86 architecture has certain characteristics that require specialized optimization techniques different from those for RISC architectures. The architecture supports a variety of instruction formats for equivalent operations. Consequently, it is critical to select instruction formats that are decoded most efficiently by the processor. The X86 register set includes only eight integer and eight floating-point registers. We have found that common global register allocation techniques that assign variables to registers for the entire scope of a procedure are ineffective with such a limited number of registers. Registers must be allocated within a narrower scope and together with instruction scheduling.

The compiler schedules instructions to minimize interlocks and to maximize parallel execution for the Pentium processor's superscalar pipelines. These techniques also benefit performance on the i486 CPU (though to a lesser extent) because the processors' pipeline organizations are similar. The instruction-scheduling techniques have minimal impact on performance for the i386 CPU since that processor uses little pipelining. As explained in the description of the floating-point pipeline, the compiler schedules FXCH instructions to avoid floating-point register-stack dependencies.

THE PENTIUM MICROPROCESSOR employs superscalar integer pipelines, branch prediction, and a highly pipelined FPU to achieve the highest X86 performance levels available elsewhere while preserving binary compatibility with the X86 architecture. Figure 11 summarizes the performance of the Pentium microprocessor and the highest performance i486

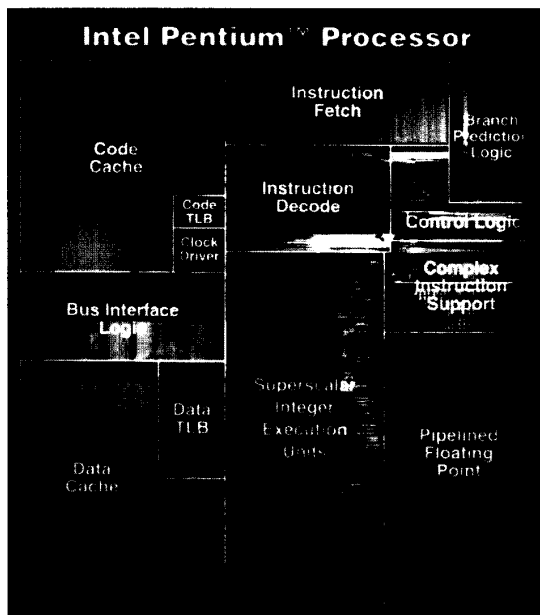



Figure 12. Die photograph.

CPU for the SPEC benchmarks in well-tuned systems. Figure 12 reproduces a photograph of the packaged circuit that integrates 3.1 million transistors. 

Acknowledgments

The individuals who made substantial contributions to the Pentium processor's design are too numerous to list here, so instead we acknowledge groups of contributors. The VLSI design team applied their creativity and determined effort throughout the project. The compiler team developed and implemented novel optimization techniques. Software engineers in several groups developed instruction-tracing and performance simulation tools. Hardware engineers and technicians instrumented measurement and tracing systems. Architects facilitated and integrated efforts of these other teams. The efforts in architecture, optimizing compiler, and performance simulation involved collaboration between teams in Santa Clara and Israel.

References

1. *i486 Processor Programmer's Reference Manual*, Intel Corporation, Santa Clara, Calif., 1990.
2. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif., 1985.

3. John H. Crawford, "The i486 CPU: Executing Instructions in One Clock Cycle," *IEEE Micro*, Vol. 10, No. 1, Feb., 1990, pp. 27-36.
4. Tejpal Chadha and Partha Srinivasan, "The Intel386 CPU Family—Architecture & Performance Analysis," *Digest of Papers Compcon Spring 1992*, CS Press, Feb. 1992, pp. 332-337.
5. Robert F. Cmelik et al., "An Analysis of Mips and Sparc Instruction Set Utilization on the SPEC Benchmarks," *Proc. ASPLOS-IV Conf., Computer Architecture News*, Vol. 19, No. 2, Apr., 1991, pp. 290-302.



Donald Alpert is an architecture manager in Intel Corporation's Microprocessor Division. He holds responsibility for managing the architecture team that developed specifications and modeling and evaluating performance of the Pentium processor. Previously, he held various microprocessor development positions at National Semiconductor Corporation and Zilog.

Alpert received a BS degree from MIT and MS and PhD degrees from Stanford University, all in electrical engineering. He is a member of the IEEE Computer Society and the Association of Computing Machinery.



Dror Avnon is design manager of the floating-point unit of the Pentium processor. He holds responsibility for the micro-architecture, design, performance analysis, and verification for the FPU logic and microcode. He previously held design engineering positions at National Semiconductor Corporation, Computer Consoles, and Elscint.

Avnon received a BSc degree in electronic engineering from Technion-Israel Institute of Technology in Haifa. He is a member of the IEEE Computer Society.

Direct questions to Dror Avnon, Intel Corporation, M/S RN2-27, 2200 Mission College Blvd., Santa Clara, CA 95052; davnon@mipos2.intel.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 159 Medium 160 High 161