

PC Processor Microarchitecture

A Concise Review of the Techniques Used in Modern PC Processors

by Keith Diefendorff

Having commandeered nearly all the performance-enhancing techniques used by their mainframe and supercomputer predecessors, the microprocessors in today's PCs employ a dizzying assemblage of microarchitectural features to achieve extraordinary levels of parallelism and speed. Enabled by astronomical transistor budgets, modern PC processors are **superscalar**, deeply pipelined, **out of order**, and they even execute instructions speculatively. In this article, we review the basic techniques used in these processors as well as the tricks they employ to circumvent the two most challenging performance obstacles: memory latency and branches.

Two Paths to Performance

The task normally assigned to chip architects is to design the highest-performance processor possible within a set of cost, power, and size constraints established by market requirements. Within these constraints, application performance is usually the best measure of success, although, sadly, the market often mistakes clock frequency for performance.

Two main avenues are open to designers trying to improve performance: making operations faster or executing more of them in parallel. Operations can be made faster in several ways. More advanced semiconductor processes make transistors switch faster and signals propagate faster. Using more transistors can reduce **execution-unit latency** (e.g., **full vs. partial multiplier arrays**). Aggressive design methods can minimize the levels of logic needed to implement a given function (e.g., **custom vs. standard-cell design**) or to increase circuit speed (e.g., **dynamic vs. static circuits**).

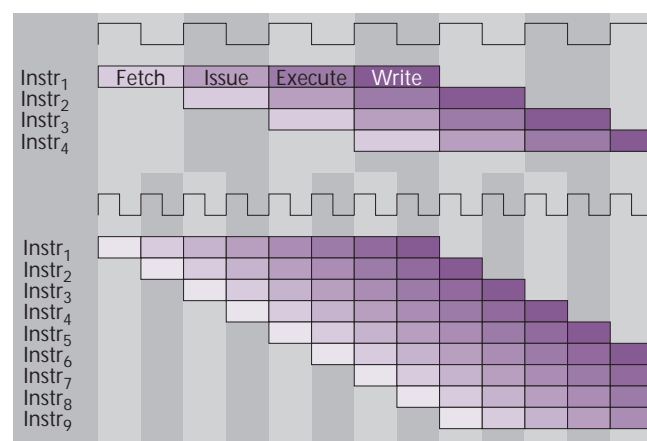


Figure 1. Pipelines overlap the execution of instructions in time. Lengthening the pipeline increases the number of instructions executed in a given time period. Longer pipelines, however, suffer from a higher percentage of stalls (not shown).

For parallelism, today's PC processors rely on **pipelining** and **superscalar** techniques to exploit **instruction-level parallelism** (ILP). Pipelined processors overlap instructions in time on common execution resources. Superscalar processors overlap instructions in space on separate resources. Both techniques are used in combination.

Unfortunately, performance gains from parallelism often fail to meet expectations. Although a four-stage pipeline, for example, overlaps the execution of four instructions, as Figure 1 shows, it falls far short of a 4× performance boost. The problem is pipeline stalls. Stalls arise from **data hazards** (data dependencies), **control hazards** (changes in program flow), and **structural hazards** (hardware resource conflicts), all of which sap pipeline efficiency.

Lengthening the pipeline, or **superpipelining**, divides instruction execution into more stages, each with a shorter **cycle time**; it does not, in general, shorten the execution time of instructions. In fact, it may increase execution time because stages rarely divide evenly and the frequency is set by the longest stage. In addition, longer pipelines experience a higher percentage of stall cycles from hazards, thereby increasing the average **cycles per instruction** (CPI). **Superscalar** techniques suffer from similar inefficiencies.

The throughput gains from a longer pipeline, however, usually outweigh the CPI loss, so performance improves. But lengthening the pipeline has limits. As stages shrink, **clock skew** and **latch overheads** (setup and hold times) consume a larger fraction of the cycle, leaving less usable time for logic.

The challenge is to make the pipeline short enough for good efficiency but not so short that ILP and frequency are left lying on the table, i.e., an **underpipelined** condition. Today's PC processors use pipelines of 5 to 12 stages. When making this decision, designers must keep in mind that frequency is often more important in the market than performance.

Prophetic Hardware for Long Pipelines

Branch prediction and **speculative execution** are techniques used to reduce pipeline stalls on control hazards. In a pipelined processor, conditional branches are often encountered before the data that will determine branch direction is ready. Because instructions are fetched ahead of execution, correctly predicting unresolved branches allows the instruction fetcher to keep the **instruction queue** filled with instructions that have a high probability of being used.

Some processors take the next step, actually executing instructions speculatively past unresolved conditional branches. This technique avoids the control-hazard stall altogether when the branch goes in the predicted direction. On **mispredictions**, however, the pipeline must be flushed,

instruction fetch redirected, and the pipeline refilled. Statistically, prediction and speculation dramatically reduce stalls. How dramatically depends on prediction accuracy.

Branch predictors range in sophistication from simple **static predictors** (compiler or heuristic driven), which achieve 65–85% accuracy, to complex **dynamic predictors** that can achieve 98% accuracy or more. Since one in five instructions is typically a conditional branch, high accuracy is essential, especially for machines with long pipelines and, therefore, with large **mispredict penalties**. As a result, most modern processors employ dynamic predictors.

The Past Predicts the Future

The simplest dynamic predictor is the **branch history table** (BHT), a small cache indexed by the address of the branch being predicted. Simple BHTs record **one-bit histories** of the direction each branch took the last time it executed. More sophisticated BHTs use **two-bit histories**, which **add hysteresis** to improve prediction accuracy on loop branches. Even more sophisticated schemes use **two-level predictors** with longer per-branch histories that index into **pattern tables** containing two-bit predictors (see MPR 3/27/95, p. 17).

A simplified version of the two-level predictor uses a single **global-history register** of recent branch directions to index into the BHT. The **GShare** enhancement (see MPR 11/17/97, p. 22) adds per-branch sensitivity by hashing a few bits of the branch address with the **global-history register**, as Figure 2 shows. The **agrees-mode** enhancement encodes the prediction as agreement or disagreement with a static prediction, thereby avoiding excessive mispredictions when multiple active branches map to the same BHT entry. In architectures with no static-prediction opcode bits, such as the x86, the static prediction must be based on branch heuristics (e.g., backward: predict taken).

Some processors predict the target instruction stream as well as the direction. Target predictions are made with a **branch target address cache** (BTAC), which caches the address to which control was transferred the last time the branch was taken. BTACs are sometimes combined with the BHT into a **branch target buffer** (BTB). Instead of a BTAC, some processors use a **branch target instruction cache** (BTIC), which caches the first few instructions down the target path so the pipeline can be primed without an inline fetch cycle. Many processors also include a special-purpose **return-address stack** to predict the return addresses of subroutines.

Rearranging Instructions Boosts Throughput

Pipeline stalls arising from data and structural hazards can sometimes be avoided by judiciously rearranging instruction execution. Stalls on data hazards, for example, can be avoided by arranging instructions such that they do not depend on the results of preceding instructions that may still be in execution. The extent to which this is possible, without violating the program's **data-flow graph**, establishes an upper limit on the ILP the processor can exploit.

Although compilers can statically reschedule instructions, they are hampered by incomplete knowledge of **run-time** information. **Load-use penalties**, for example, are resistant to static rescheduling because their length is generally unpredictable at **compile time**. It is simply impossible to find enough independent instructions to cover the worst-case number of **load-delay slots** in every load.

Static rescheduling is also constrained by **register namespace** and by ambiguous dependencies between memory instructions. A large register namespace is required for good **register allocation**, for freedom in rearranging instructions, and for loop unrolling. Register limitations are especially severe in x86 processors, which have only eight general-purpose registers. **In-order processors**—which issue, execute, complete, and retire instructions in strict **program order**—must rely entirely on static rescheduling and can suffer a large number of pipeline stalls.

Therefore, most current PC processors implement dynamic instruction rescheduling to some degree. The simplest **out-of-order processors** issue instructions in order but allow them to execute and complete out of order. Processors of this type use **register scoreboarding** to interlock the pipeline, stalling instruction issue when an instruction's operands aren't ready. Such processors can achieve somewhat more parallelism than in-order processors by permitting instructions to execute in parallel through execution units with different or **variable latencies**.

Even simple out-of-order processors require complex hardware to reorder results before the corresponding instructions are **retired** (removed from the machine). Although strict result ordering is not needed from a data-flow perspective, it is required to maintain **precise exceptions** (the appearance of in-order execution following an interrupt) and to recover from mispredicted speculative execution.

The most common reordering method is the **reorder buffer** (ROB), which buffers results until they can be written to the register file in program order. Accessing operands from the reorder buffer, which is needed for reasonable performance, requires an associative lookup to locate the most recent version of the operand.

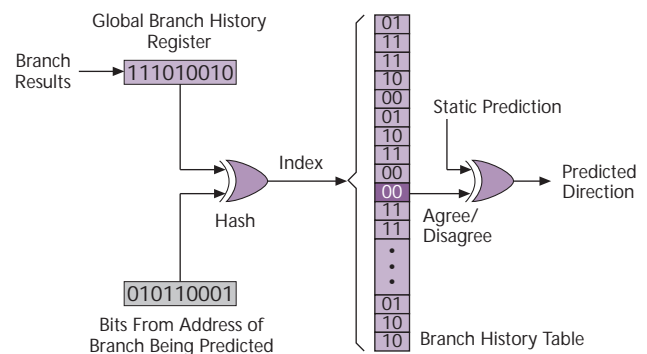


Figure 2. The GShare algorithm with agrees-mode encoding is used by several PC processors to dynamically predict branches.

Other types of reordering hardware are also used. **History buffers**, for example, record **source-operand history** so the processor can backtrack to a **precise architectural state**. **Future files** maintain the current state and the architectural state in separate register files, allowing the processor to be checkpointed back to a precise state.

Complexity Rears Its Ugly Head

More-aggressive out-of-order processors can also avoid stalling **instruction dispatch** on data hazards. This is accomplished by setting instructions aside in **reservation stations**, where they can wait for operands while dispatch proceeds. As operands become available, instructions are **issued** to execution units, possibly out of program order. (Note: although some architects reverse the use of the terms *dispatch* and *issue*, we find the convention used here is the most common.) **Distributed reservation stations** are dedicated to individual execution units, as Figure 3 shows, while **centralized reservation stations** serve multiple units.

Although out-of-order processing averts many stalls, it suffers from new ones created by **false dependencies**—artifacts of using register names out of order. Stalls on these dependencies—called **write-after-read hazards** (WAR or antidependencies) and **write-after-write hazards** (WAW or output dependencies)—can be avoided by **register renaming**, a process whereby each new instruction is assigned a unique destination register from a large pool of physical registers. The mapping of physical registers to architectural registers is maintained in the ROB, usually by some type of **register alias table** (RAT), allowing register operands to be accessed by their architectural name. Physical registers are released and returned to the pool as instructions are retired.

In out-of-order processors, the capacity of the reservation stations, reorder buffer, and rename tables limits the number of instructions that can be **in flight**. Processors that keep many instructions in flight generally exploit more parallelism and have higher execution-unit utilization; thus they achieve higher instruction throughput (**instructions per clock** or IPC) for a given amount of hardware.

The instructions that can be considered for issue to execution units each cycle is called the **window**. The complexity

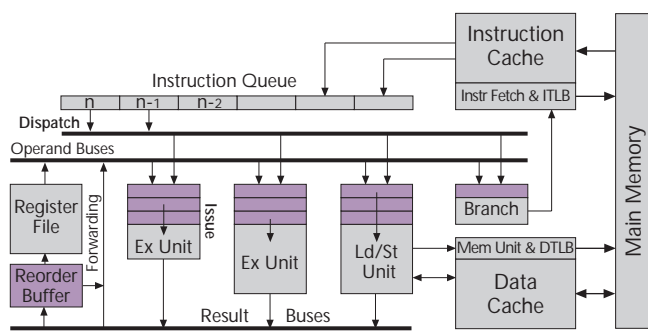


Figure 3. A generic two-issue out-of-order superscalar processor with reorder buffer and distributed reservation stations (purple) to dynamically reschedule instructions.

of the data-dependence analysis required to place instructions into execution is roughly proportional to the square of the maximum number of instructions in the window. (The window should not be confused with the superscalar **dispatch width**, which is the number of instructions that can be decoded and dispatched to reservation stations each cycle, or the **issue width**, which is the number of instructions that can be issued to execution units each cycle.)

Memory: the Grand Challenge

Although dynamic instruction scheduling provides some degree of latency tolerance, it is impractical to build reordering hardware deep enough to cover very many long-latency operations. There are several operations that tend to have long latency, such as multiply, divide, and most floating-point operations, but memory accesses present the greatest challenge to the CPU designer.

The latency of an access to PC memory can be 20–30× the CPU cycle time, even in the best case that the access hits an **open page** (or bank) in DRAM. Accesses that require opening a page, or closing one and opening another, can take much longer. Access time can be even longer if the processor must contend with another device for access to the memory; **queuing theory** suggests that the **average service time** of a memory request increases rapidly with memory **utilization**.

There are plenty of opportunities to experience memory delays; roughly every third instruction in a program is a memory operation. Clearly, steps must be taken to reduce access time, else even the fanciest instruction-level-parallel hardware will go for naught. Even with heroic efforts, however, memory access time remains the biggest performance obstacle.

With today's **SDRAMs (synchronous DRAMs)**, it isn't terribly difficult to build high-bandwidth memory systems. Unfortunately, processor performance tends to be more sensitive to memory latency than to memory bandwidth, and low memory latency is far harder to achieve. The two primary weapons in the fight against memory latency are caching and reordering.

Caches are small, fast buffers managed by hardware to exploit **locality of reference** as a means of reducing memory latency. Caches can exploit both **temporal locality** (the propensity to access the same data repeatedly within a short time period) and **spatial locality** (the propensity to access memory addresses that are close together).

When the processor first accesses a memory location, the surrounding block of memory is brought into a **cache line** and **tagged** with the address of the block. Thereafter, each time the processor accesses memory, the cache tags are checked for a match. On a match (**cache hit**), data is supplied to the processor from the corresponding cache line. If no match is found (a **cache miss**), the data is brought in from memory, supplied to the processor, and deposited into the cache. To the extent that locality is present, there will be more hits than misses, and the **average memory access time** (T_{avg}) will be reduced.

Most PC processors utilize separate **instruction caches** (I-caches) and **data caches** (D-caches). These caches are usually accessed and tagged with **physical memory addresses**. Processors translate **logical addresses** (program addresses) to physical addresses via lookup in a **translation-lookaside buffer** (TLB). A TLB caches recent translations of virtual-page numbers to physical-page numbers, as well as **memory-protection** attributes such as **write protect**. Usually there is an **instruction TLB** (I-TLB) and a **data TLB** (D-TLB); these are sometimes backed by a larger **unified TLB**. TLB misses typically invoke a microcode routine that searches for a translation in the OS-maintained **virtual-memory** mapping tables. If none is found, a **page-fault** interrupt transfers control to the OS to resolve the problem.

If You Want It Fast, Keep It Close

Caches suffer from three types of misses: **compulsory misses**, which occur on the initial access to a memory location; **capacity misses**, which occur when the cache can't hold all the data being accessed; and **conflict misses**, which occur when multiple memory blocks map to the same cache line. Ignoring the program's **memory-access patterns**—which can have a large effect on cache behavior—the physical characteristics of the cache determine its **miss ratio**. Important characteristics include size, **associativity**, line length, replacement policy, **write policy**, and **allocation policy**.

Compulsory misses can be reduced by increasing the **line size** to take more advantage of spatial locality. Increasing the line size, however, reduces the number of blocks in the cache, thereby increasing capacity misses and conflict misses. Line lengths of 32 and 64 bytes provide a good balance for small caches and are the most commonly used sizes.

Ideally, stalls from compulsory misses could be avoided if the compiler could **boost** loads so memory access could get started ahead of the time the program will actually need the data. Unfortunately, a load cannot generally be boosted out of its **basic block**, because the block may not be executed and the load could fault. Furthermore, it is frequently impossible for a compiler to **disambiguate memory address**, forcing it to be overly conservative. As a result, loads cannot usually be boosted far enough to cover much memory latency.

An increasingly popular solution to the problem of compulsory misses is **nonbinding prefetch** instructions. These instructions are simply hints to the hardware, suggesting it should try to fetch a memory block into the cache. Because prefetch instructions don't modify machine state (registers) and are nonfaulting, they can be placed arbitrarily far ahead of a load, allowing time for a compulsory miss to be serviced, so the load sees a cache hit.

It Was Here Just a Moment Ago

Capacity misses are mainly a function of cache size. Large caches have lower miss ratios than small caches; as a general rule of thumb, miss ratio improves proportionally to the square root of a cache-size increase: e.g., a 4× larger cache has

roughly half the miss ratio. This rule suggests rapidly diminishing returns on cache size.

Moreover, access time increases with cache size, thanks to physics. This fact sets up a tradeoff between a small, fast cache and a larger, slower cache. To reduce **thrashing**, caches should be larger than the **working set** of the program. Because working sets are sometimes too large for caches small enough to have one- or two-cycle access times, many processors use a **cache hierarchy**. Two-level caches, for example, comprise a small, fast **level-one cache** (L1) backed by a larger, slower **level-two cache** (L2), as Figure 4 shows.

Early two-level caches consisted of **on-chip** L1s, with the external L2 connected to the system or **frontside bus** (FSB). FSBs are not ideal cache interfaces, however. Designed as **shared multidrop buses** for DRAM, I/O, and multiprocessor (MP) traffic, FSBs are usually slow. A 500-MHz processor requires an average 2.5 CPU cycles to synchronize each memory request to a slow 100-MHz bus, adding to L2 access time. This slow speed also throttles the **burst transfers** used to fill cache lines. To minimize these effects, processors burst data **critical word first**, and **forward** it immediately to the processor so the pipeline can be restarted posthaste.

Multilevel Caches Move on Chip

To speed L2 accesses, many processors have adopted **dual-bus architectures**, placing the L2 on a dedicated **backside bus** (BSB). Because a BSB connects exclusively to the cache, it can be optimized for SRAM transfers and can operate at the full CPU clock rate. Since SRAMs capable of operating at full CPU speeds are expensive, however, most PC processors operate the BSB at half the CPU clock rate. Still, the BSB makes a much faster L2 interface than an FSB. Some processors have taken the additional step of moving the **L2-cache tags** onto the processor die to speed hit/miss detection and to allow higher **set-associativity**.

With the advent of 0.25-micron processes, PC processor vendors began bringing the BSB and the L2 on chip. The alternative of increasing the size of the L1s is still favored by some designers, but the two-level approach will become more popular as on-chip cache size grows. The trend toward on-chip L2s will accelerate with 0.18-micron processes, and external L2s may disappear completely by the 0.13-micron generation.

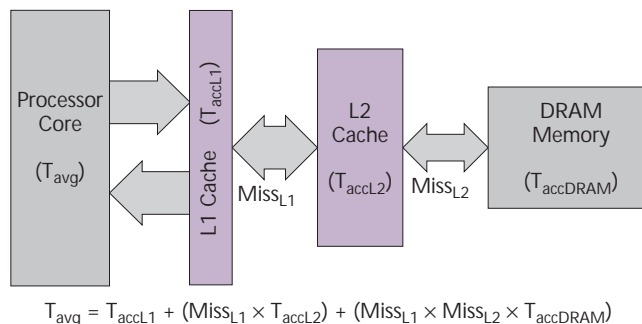


Figure 4. Two-level cache hierarchies are designed to reduce the average memory access time (T_{avg}) seen by the processor.

Although on-chip L2s are typically smaller than external L2s, they can also be faster. On chip, the BSB can be very wide and operate at the full CPU clock rate. In addition, the L2 can have higher **set-associativity, multiple banks, multiple ports**, and other features that are impractical to build off chip with commodity SRAMs. These attributes can increase speed and hit ratios dramatically, offsetting the smaller size. On most PC applications, a full-speed 256K on-chip L2 outperforms a half-speed external 512K L2.

Associativity Avoids Conflicts

Conflict misses can be reduced by **associativity**. In a nonassociative or **direct-mapped cache**, each memory block maps to one, and only one, cache line. But because multiple blocks map to each cache line, accesses to different memory addresses can conflict. In a **fully associative cache**, on the other hand, any memory block can be stored in any cache line, eliminating conflicts. Fully associative caches, however, are expensive and slow, so they are usually approximated by ***n*-way set-associative caches**.

As a rule of thumb, a two-way set-associative cache has a miss rate similar to a direct-mapped cache twice the size. Miss-rate improvement, however, diminishes rapidly with increasing associativity. For all practical purposes, an eight-way set-associative cache is just as effective as a fully associative cache. A **least-recently used** (LRU) replacement algorithm is the one most often used to decide into which **way** a new line should be allocated (even though LRU is known to be suboptimal in many cases).

Cache performance is also affected by a cache's write policy. The simplest policy is **write through**, wherein every store writes data to main memory, updating the cache only if it hits. This policy, however, leads to slow writes and to excessive write traffic to memory. As a result, most PC processors use **write-back** caches (sometimes called **copy-back** or **store-in** caches). Write-back caches write to the cache, not to memory, on a hit. Thus, a cache line can collect multiple store hits, writing to memory only once when the line is replaced. Most write-back caches use a **write-allocate** policy, which allocates a new line in the cache on a write miss.

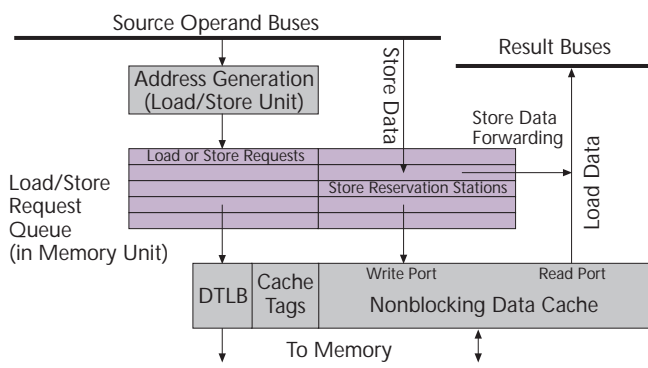


Figure 5. Some processors allow loads to bypass stores that are waiting on data. With a nonblocking cache, load hits can access the cache while a previous miss is waiting on data from memory.

Who Has What?

In systems with caches, there is the nasty problem of **cache coherency**. If, for example, a processor has a memory block in its cache, and an I/O device writes to an address in that block, then the data in the processor's cache becomes **stale**. If the processor has modified the cache line, the data written by the I/O device will be overwritten and lost permanently when the cache line is eventually written back to memory.

Avoiding these situations with software is difficult and error prone. With multiple processors, the problem becomes even more complicated, and software solutions become intractable. Although not a problem for PCs today, multiprocessors will one day become attractive. Thus, to simplify I/O software and enable multiprocessing in the future, PC processors all enforce cache coherence via hardware.

The most popular scheme is the four-state **coherence protocol** called **MESI** (modified, exclusive, shared, invalid). In this scheme, MESI status bits are maintained with each cache line. The first time a processor writes to a shared line in its cache, it broadcasts a **write-invalidate coherence transaction** to other devices. Any device with a copy of the line in its cache invalidates its copy, which, if modified, requires writing the line to memory before allowing the processor to take **ownership** of it. As an optimization, some processors allow lines to be allocated into a fifth, owned state (**MOESI**) to improve the efficiency of accessing shared data in **symmetric multiprocessor** (SMP) systems.

In this coherence scheme, every processor or device with a cache **snoops** (watches) every memory transaction issued by every device in the **coherence domain**. If a snooped transaction hits on a line that is held exclusively, its status is changed to shared. If a snooped transaction hits a modified line, the offending transaction is held off until the dirty line can be written to memory. Alternatively, the processor with the hit can **intervene** to produce the modified data, allowing other processors and memory to **snarf** it off the bus.

Memory Instructions Are Different

Loads and stores have two distinct phases: **address generation** and memory access. Through the address-generation phase, processors treat loads and stores just like other instructions. To deal with the unique characteristics of memory, however, processors generally decouple the memory-access phase by issuing memory requests along with their resolved addresses to a queue or a buffer in the **memory unit**, as Figure 5 shows.

One important function performed by the memory unit is **load/store reordering**. Store instructions are often issued before their store data is ready. If all memory transactions were forced to access memory in program order, subsequent loads would be blocked, unnecessarily stalling the pipeline. To alleviate the blockage, some memory units provide **store reservation stations**, where stores can wait on data while subsequent loads access memory. To ensure correct program operation, the memory unit must perform

dynamic memory-address disambiguation, disallowing any load from passing a store to the same address.

As a further optimization, some processors support **store-data forwarding**, which allows store data that has arrived in the reservation stations to be forwarded directly to a subsequent load, thereby eliminating the load's cache access altogether. This feature is especially important on x86 processors, which, because of the dearth of registers, frequently store data to memory and read it back soon thereafter.

Another important memory-unit function is **hit-under-miss** processing. Without this feature, a cache miss will unnecessarily block subsequent independent loads, stalling the pipeline. Implementation of this feature requires a **non-blocking or lock-up-free cache**. Processors that allow multiple outstanding misses can also support **miss-under-miss** processing.

Although store instructions are less urgent than loads from a data-flow perspective, they can still cause stalls. In fact, stores are a large problem in out-of-order processors, because writes cannot be **committed** to memory before it is known that all previous instructions will retire without exception. As a result, store misses can constipate the **ROB**, creating a structural hazard when it fills.

The problem is exacerbated by architectures whose **memory consistency model requires sequential consistency** (all memory operations must be performed in program order, as observed by another processor). Thus, most processors implement some form of **relaxed ordering**. The x86 architecture specifies **processor consistency**, which relaxes load-store order but requires strict store ordering. PowerPC improves store throughput further with its **weak consistency** model, which also allows stores (to different addresses) to be performed out of order.

Even weak ordering, however, doesn't eliminate the ROB hazard entirely. Thus, processors often add other features to improve store throughput. **Write buffers**, for example, capture writes that miss the cache, allowing the store to be committed and released from the ROB before the write to memory actually occurs. A **write-combining** write buffer collects multiple writes to adjacent addresses so they can be transferred to memory as a single transaction, improving write bandwidth. This latter feature is important for byte writes to **noncachable memory**, like graphics frame buffers.

To DRAM Via the System Bus

Misses in the cache hierarchy are resolved by accessing main memory over the processor's system bus. For filling cache lines, an important bus characteristic is its **peak bandwidth**—the maximum rate at which data can be transferred across the bus. Today's PC system buses carry 64 bits (8 bytes) of data clocked at 100 MHz, yielding a peak bandwidth of 800 MBytes/s. Transmission-line problems with **shared multidrop buses**, however, preclude operation much above 133 MHz. Next-generation buses, however, will use **point-to-point signaling with source-synchronous clocking**, allowing bus clocks to

reach 400 MHz (a peak bandwidth of 3.2 GBytes/s on a 64-bit bus) or more.

Because of the long initial access time of DRAMs, the **sustained bandwidth** of a bus would be much less than its peak bandwidth if not for **address pipelining**. Address pipelining allows addresses to overlap data transfers so the memory controller can start the next access as soon as possible. This feature is normally implemented on a **demultiplexed bus** (one with separate address and data paths). All popular PC buses support address pipelining, although some support more **outstanding transactions** than others.

Some advanced buses also implement **split transactions**, which avoid locking up the bus and stalling the processor on accesses to long-latency I/O devices. Even more advanced buses implement **out-of-order transactions**. In these buses, request and reply transactions are distinct; requests are tagged, so out-of-order replies can be matched with the right request. This feature is essential for maintaining high throughput in multiprocessor systems with a variety of memory modules and I/O devices.

The Instruction Set Still Matters

Theoretically, processors based on **reduced instruction set computer** (RISC) architectures are easier to parallelize and easier to clock at high frequencies than those based on **complex instruction set computer** (CISC) architectures. Due to the enormous resources behind x86 processors (CISCs) and the astronomically high transistor budgets available to designers, however, many of CISC's disadvantages have been overcome. But not all flaws or omissions in an **instruction-set architecture** (ISA) are easy to overcome.

One important attribute is the number of registers; more registers can reduce memory traffic and can support more parallelism with less hardware complexity. Related to this is the operand format. **Two-operand destructive formats** ($R_D \leftarrow R_D \text{ op } R_S$) are less register efficient than **three-operand nondestructive formats** ($R_D \leftarrow R_{S1} \text{ op } R_{S2}$).

Also important is the instruction encoding. Although complex, variable-length instructions are **code-space** efficient—and thus I-cache and instruction-bandwidth efficient—they are harder to decode than simple, fixed-length instructions. High-frequency x86 processors use enormous decoders and several pipeline stages just for **instruction decode**. RISC processors don't need these extra stages.

To minimize decode time in the pipeline, some processors employ a **predecoded instruction cache**. These processors partially decode the instruction stream on the way into the I-cache, storing predecoded information, typically 1 to 3 bits per instruction byte, in the cache along with the associated instructions. This technique is especially effective for locating the beginning of variable-length x86 instructions.

Other examples of ISA features that can affect performance include **static branch prediction**, nonbinding prefetch instructions, **flat-register addressing** (as opposed to the stack-based addressing of the x87 floating-point unit), and

floating-point multiply-add. This FMADD feature chains a multiply and an add together with the latency of a multiply, improving the performance of inner products and other important numerical algorithms. **Fusing** the multiply-add with a single rounding enables even greater speed.

A relatively new feature in ISAs is **single-instruction, multiple-data** processing (SIMD). Its popularity is being driven by the increasing demand for digital-signal processing (DSP) and multimedia processing in PC applications. DSP and multimedia algorithms are rich in **data-level parallelism**. To take advantage of this parallelism, SIMD instructions operate on packed fixed-length **vector operands**, as Figure 6 shows, rather than on the single-element operands of conventional **scalar instructions**. Although instruction-level parallel techniques can also exploit data-level parallelism, SIMD units exploit it to a higher degree and with less complexity.

Most SIMD ISA extensions support special features for DSP algorithms that are not found in traditional general-purpose processor ISAs. **Saturation arithmetic**, for example, **clamps overflowed or underflowed elements** at their maximum or minimum values—an important feature for processing packed short integers with limited dynamic range.

Process Technology: the Silver Bullet

Unfortunately, all of the techniques reviewed so far come at a cost. Mostly they increase complexity, which has negative effects on frequency, die size, and power. Complexity increases nonlinearly with pipeline length and with issue width, and it is exacerbated by the cascade of tricks required to minimize stalls. Complexity adds logic gates, slowing cycle time or adding pipeline stages. It adds transistors, lengthening signal paths and increasing die size and cost. It also increases design time, which, since processor performance progresses at about 60% per year, costs 4% in relative performance every month—a nontrivial amount.

IC process is different. Everything improves: performance, frequency, die size, and power. No known microarchitectural technique comes close to the massive improvements made by a single process generation.

One **process generation**—defined as a 30% linear **shrink** of feature sizes—halves **die size** (or doubles the transistor budget for the same die size). A smaller die size dramatically lowers **manufacturing cost** because it both increases the **gross die per wafer** and improves **yield** (see MPR 8/2/93, p. 12). Yield improves exponentially with decreasing die size because statistically fewer die are lost to **defects**.

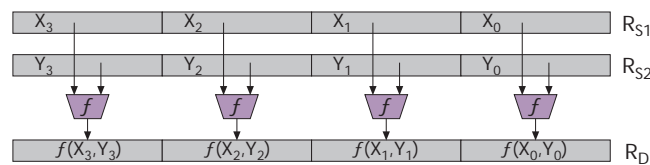


Figure 6. SIMD instructions perform the same operation on all the elements of short vectors stored in registers.

Defect density is usually somewhat higher at the introduction of a new process, but it improves quickly with experience and volume.

One process generation also improves the **intrinsic speed** (CV/I) of transistors by 30–50%. As if these gains weren't enough, each generation is typically accompanied by about a 25% reduction in voltage, which, since power is a quadratic function of voltage ($P = CV^2f$), cuts power consumption in half. The only critical parameter that doesn't naturally improve with process shrinks is **interconnect delay** (RC delay). Manufacturers are combating this recalcitrant term by lowering **capacitance** (C) with **low dielectric-constant** (low- k) insulators, and by lowering **resistance** (R) with thicker metal layers or by moving from aluminum to **copper metallization** (see MPR 8/4/97, p. 14).

Currently, most PC microprocessors are built on a 0.22- to 0.25-micron process with ≈ 0.18 -micron **gate lengths** (L_{gate}), five layers of aluminum interconnect, and operating voltages from 1.8 to 2.5 V. Logic densities are roughly 60,000 transistors/mm², and SRAM cells are about 10 μm^2 . Next-generation 0.18-micron processes—which will begin volume production during 2H99 at most companies—will have an L_{gate} of ≈ 0.14 microns, six layers of aluminum or copper interconnect, operating voltages of 1.5 V or less, logic densities of 120,000 transistors/mm², and SRAM cells smaller than 5 μm^2 (see MPR 9/14/98, p. 1; MPR 1/25/99, p. 22).

The **package** is also a factor in processor performance, cost, and size (see MPR 9/13/93, p. 12). Chips are either **wire bonded**, which is the cheaper method, or **flip-chip mounted** with solder bumps, which is the electrically superior method, onto a package **substrate**. **Plastic** substrates are the least expensive, but cannot handle as much power as **ceramic** substrates. **Pin-grid-array** (PGA) packages are used where socketability is required, but surface-mount **ball-grid-arrays** (BGAs) are smaller (important for notebooks), cheaper, and electrically superior. For all package types, package costs and test costs are a function of the number of pins (or balls).

Organic BGA packages appear to be the way of the future. They offer a **low- k dielectric substrate** and a copper **lead frame** for superior electrical characteristics. They also have low cost and low thermal resistance, since a heat sink can be directly attached to the silicon die.

Disagreement Over Microarchitecture Abounds

Aside from a state-of-the-art semiconductor process, which is the minimum ante to play in the PC processor business, there is considerable disagreement over which is the best collection of microarchitectural features for a PC processor. Some designers, for example, prefer complex wide-issue out-of-order microarchitectures; others believe that simple, fast, in-order pipelines with large caches are better. This and other differences of opinion are evident in current and upcoming PC processors. ■

In the next installment of this article, we look at how specific PC processors use the techniques reviewed in this article.