Advanced Processor Architecture
Summer 1997

Prepared By : Tarak Modi
Advisor : Dr. Wells

Introduction

The first IBM PC was based on the Intel 8088 microprocessor running at 4.77 Mhz. The 8088 took 4 clock cycles to run a zero wait state bus cycle. Over the years, the x86 family of processors increased in speed (about 50 times), however, the access time of DRAM memory has not decreased at the same rate. This results in a performance bottleneck; one or more states are added in every access to memory. This has a severe degrading effect on the processor speed.

Instruction Prefetcher

In an effort to decouple the processor's execution unit from the lagging memory, Intel incorporated a prefetcher and a prefetch queue into each processor. The prefetcher works on the assumption that the prefetcher will want the next sequential instruction. While the processor is executing one instruction, the processor also utilizes the idle bus time to fetch the next instruction from the memory, before the execution unit actually requests it. Thus the memory access time is hidden by the execution time of the previous instruction. This prefetched instruction is placed in the prefetch queue. Since most program execution is sequential in nature the hit rate on the prefetch queue is pretty high. However, if the current instruction was a jump instruction, and the jump was actually taken, the prefetcher gambled and lost. The queue is flushed, and the execution unit is stalled while the actual next instruction is fetched.

Processor	Prefetech Queue Length
8088	4
8086	6
80286	6
80386	16
80486	32
Pentium	two 64-byte

When the processor speeds reached the 20-25 Mhz vicinity, reasonably priced DRAM memory could no longer be accessed with zero wait bus cycles. Memory access time was therfore a problem. There were two possible solutions :

- Use SRAM instead of DRAM.
- Implement a cache system.

SRAM solution

This is not economically viable for the following reasons :

- SRAM is about ten times more expensive than DRAM.
- SRAM chips are physically larger, thus requiring more real-estate for the same amount of memory.
- SRAM require more power than DRAM.
- SRAMs generates more power than DRAM.

The Cache Solution

External Cache
System designers achieved a performance-cost tradeoff by implementing an external cache consisting of a relatively small amount of SRAM, and a cache controller. The bulk of the rest of the memory was relatively inexpensive DRAM. The main advantage was that it reduced many memory accesses to zero wait states by trying to keep a copy of the most frequently requested information in the SRAM cache, and attempting to maximize the hit ratio. The disadvantage is that the memory acces is still upper bound by the bus speed.

A Unified Internal Code/Data Cache
It incorporates the advantage of the external cache coupled with the fact that every time a memory request is fulfilled by the internal cache, one less bus cycle results. In addition, the access of data is much faster since the only delay would be to look up the data/code and to deliver it to the internal requester. This also frees up the bus for other bus masters.

This concept has been utilized on the x486 processor. The disadvantage of using a unified data and code cache is that it can lead to internal contention for access to the cache. At any given instant of time, the execution unit may be executing a move instruction to read data from memory into a register. This read request is submitted to the internal unified cache. If, at the same instant, the code prefetcher issues a memory read request to the cach for the code, the execution unit's request is serviced first. This leads to a situtation called instruction pipeline starvation. The pentium avoids this by including a separate code and data cache.

By including separate on-chip code and data caches, and also an external cache to handle misses by the internal cache, the pentium processor system minimizes the throttling effect of the relatively slow DRAM memory. Now the chocking point is the pipeline. To address this problem, the pentium processor incorporates dual instruction pipelines. A check is performed on each pair of instructions to determine their pairability. If deemed pairable, the two instructions are fed to the dual pipelines simultaneoulsy, and are executed concurrently. This is referred to as superscaler architecture.

Branch prediction

Prior to the Pentium processors, the instruction prefetchers were simple in nature. Their only goal was to keep the prefetch instruction queue full, and worked on the assumption that the next instruction to be executed would be the next memory location sequentially. However, if the execution unit executed a branch instruction, and the branch was taken, the pipeline and the prefetch queue would have to be flushed. The next instruction from the branch target location would have to be fetched, and the execution unit would be stalled.

The pentium processor alleviated this choke-point by adding branch prediction logic, enabling the prefetcher to make more intelligent decisions. The processor has two prefetch queues, but only one of them is used at a time. When a branch instruction enters the pipeline, the prediction logic predicts if the branch will be taken, and depending on the prediction will continue to fill up the current queu with sequential instructions or will swith queues and start fetching instructions from the branch target address. Now when the branch instruction is actually executed and the prediction was correct then everyhting is fine. If the prediction predicted the branch taken, and it was not actually taken, the pipeline and the active queue are flushed. It swithces back to the original queue which still has the correct instructions.

Technical Innovations

A number of innovative product features contribute to the Pentium processor's unique combination of high performance, compatibility, data integrity and upgradability. These include:

Superscalar Architecture

The Pentium(R) processor's superscalar architecture enables the processor to achieve new levels of performance by executing more than one instruction per clock cycle. The term "superscalar" refers to a microprocessor architecture that contains more than one execution unit. These execution units-or pipelines-are where the chip processes the data and instructions that are fed to it by the rest of the system. The Pentium processor's superscalar implementation represents a natural progression from previous generations of processors in the 32-bit Intel architecture. The Intel486(TM) processor, for example, is able to execute many of its instructions in one clock cycle, while previous generations of Intel microprocessors require multiple clock cycles to execute a single instruction.

This ability to execute multiple instructions per clock cycle is due to the fact that the Pentium processor's two pipelines can execute two instructions simultaneously. As with the Intel486 processor's single pipeline, the Pentium processor's dual pipelines execute integer instructions in five stages: prefetch, decode 1, decode 2, execute and writeback. This permits several instructions to be in various stages of execution, thus increasing processing performance.

The Pentium processor also uses hardwired instructions to replace many of the microcoded instructions used in previous microprocessor generations. Hardwired instructions are simple and commonly used, and can be executed by the processor's hardware without requiring microcode. This improves performance without affecting compatibility. In the case of more complex instructions, the Pentium processor's enhanced microcode further boosts performance by employing both dual integer pipelines to execute instructions.

Separate Code and Data Caches

Another significant advancement is the Pentium(R) processor's innovative on-chip cache implementation. On-chip caches increase performance by acting as temporary storage places for commonly-used instructions and data, replacing the need to go off-chip to the system's main memory to fetch information. The Intel Pentium processor incorporates separate on-chip code and data caches. This increases performance because bus conflicts are reduced (with a single cache, conflicts can occur between instruction pre-fetches and data accesses) and the caches are available more often when they are needed.

The Pentium processor's code and data caches each contain 8 Kbytes of information, and both are organized as two-way set associative caches—meaning that they save time by searching only pre-specified 32-byte segments rather than the entire cache. This performance-enhancing feature is in turn supplemented by the Pentium processor's 64-bit data bus, which ensures that the dual caches and superscalar execution pipelines are continually supplied with data.

The Pentium processor's data cache uses two other important techniques: "writeback" caching and an algorithm called the MESI (Modified, Exclusive, Shared, Invalid) protocol. The writeback method transfers data to the cache without going out to main memory (data is written to main memory only when it is removed from the cache). In contrast, previous-generation "write-through" cache implementations transfer data to the external memory each time the processor writes data to the cache. The writeback technique increases performance by reducing bus utilization and preventing needless bottlenecks in the system.

To ensure that data in the cache and in main memory are consistent, the data cache implements the MESI protocol. By obeying the rules of the protocol during reads/writes, the Pentium processor can maintain

cache consistency and circumvent problems that might be caused by multiple processors using the same data.

Branch Prediction

Branch prediction is an advanced computing technique that boosts performance by keeping the execution pipelines full. This is accomplished by predetermining the most likely set of instructions to be executed. The Pentium processor is the first PC-compatible microprocessor to use branch prediction, which until now has traditionally been associated with the mainframe computers.

For a better understanding of this concept, consider a typical application program. After each pass through a software loop, the program performs a conditional test to determine whether to return to the beginning of the loop or to exit and continue on to the next execution step. These two choices, or paths, are called branches. Branch prediction forecasts which branch the software will require, based on the assumption that the previous branch that was taken will be used again. The Pentium processor makes branch predictions using a Branch Target Buffer (BTB). This software-transparent innovation eliminates the need for recompiling code, thus increasing overall speed and application software performance.

To efficiently predict branches, the Pentium processor uses two prefetch buffers. One buffer prefetches code in a linear fashion (for the next execution step) while the other prefetches instructions based on addresses in the Branch Target Buffer (to jump to the beginning of the loop). As a result, the needed code is always prefetched before it is required for execution.

The Pentium processor's prediction algorithm can not only forecast simple branch choices, but also support more complex branch prediction-for example, within nested loops. This is accomplished by storing multiple branch addresses in the Branch Prediction Buffer. The BTB's design allows 256 addresses to be recorded, and thus the prediction algorithm can forecast up to 256 branches.

High-Performance Floating-Point Unit

The Pentium processor family takes math computational ability to the next performance level by using an enhanced on-chip floating-point unit. Through instruction scheduling and overlapped (pipelined) execution, the floating-point unit is capable of executing two floating-point instructions in a single clock. Incorporated into the unit is a sophisticated eight-stage pipelining. The first four stages are the same as that of the integer pipelines while the final four stages consist of a two-stage Floating Point Execute, rounding and writing of the result to the register file, and Error Reporting. In addition, common floating-point functions—such as add, multiply and divide—are hardwired for faster execution.

Enhanced 64-Bit Data Bus

The data bus is the highway that carries information between the processor and the memory subsystem. Because of its external 64-bit data bus, the Pentium processor can transfer data to and from memory at rates up to 528 Mbytes/second, a more than five-fold increase over the peak transfer rate of the Intel486 (TM) DX2-66MHz microprocessor (105 Mbytes/second). This wider data bus facilitates high-speed processing by maintaining the flow of instructions and data to the processor's superscalar execution unit.

In addition to having a wider data bus, the Pentium processor implements bus cycle pipelining to increase bus bandwidth. Bus cycle pipelining allows a second cycle to start before the first one is completed. This gives the memory subsystem more time to decode the address, which allows slower and less-expensive memory components to be used resulting in a lower overall system cost. Burst reads and writes, parity on address and data, and a simple cycle identification all contribute to providing better bandwidth and improved system reliability.

The Pentium processor also has two write buffers, one corresponding to each pipeline, to enhance the performance of consecutive writes to memory. Write buffers improve performance by allowing the

processor to proceed with the next pair of instructions, even though one of the current instructions needs to write to memory while the bus is busy.

Data Integrity Features

Protecting important data and ensuring its integrity has become increasingly important as mission-critical applications continue to proliferate. To ensure the Pentium processor's reliability, Intel ran millions of simulations and tests. In addition, designers integrated two advanced features traditionally associated with mainframe-class designs internal error detection and functional redundancy testing—to help preserve data integrity in today's evolving PC-based networks.

Internal error detection places parity bits on the internal code and data caches, translation look aside buffers, microcode, and branch target buffer. This feature helps detect errors in a manner that remains transparent to both the user and the system.

SL Enhanced Power Management Features

The Pentium processor family incorporates SL technology features for superior power-management capabilities. These features operate at two levels: the microprocessor and the system. Power management at the processor level involves putting the processor into low power state during non-processor intensive tasks (such as word processing), or into a very low-power state when the computer is not in use ("sleep" mode). At the system level, Intel's SL technology uses system management mode (SMM) to control the way power is used by the computer (including peripherals). This mode provides intelligent system management that allow the microprocessor to slow down, suspend, or completely shut down various system components so as to maximize energy savings. All members of the Pentium processor family include SMM.

Multiprocessor Support

The Pentium processor is ideal for the increasing wave of multiprocessing systems. Multiprocessing applications that combine two or more Pentium processors are well served by the chip's advanced architecture, separate on-chip code and data caches, chip sets for controlling external caches, and sophisticated data integrity features.

As previously discussed, the Pentium processor family uses the MESI protocol to maintain cache consistency among several processors. The Pentium processor also ensures that instructions are seen by the system in the order that they were programmed. This strong ordering helps software designed to run on a single-processor system to work correctly in a multiprocessing environment.

The Pentium processor family also includes two new multiprocessor (MP) features: a multiprocessor interrupt controller on-chip and the dual processor mode. The processor's on-chip MP interrupt controller can support up to 60 processors. The dual processor mode enables two processors to share a single second-level cache, allowing the development of low-cost shared-cache multiprocessor systems for workstations and low-end servers.

Performance Monitoring

Performance monitoring is a feature of the Pentium processor that enables system designers and application developers to optimize their hardware and software products by identifying potential code bottlenecks. Designers can observe and count clocks for internal processor events that affect the performance of data reads and writes, cache hits and misses, interrupts, and bus utilization. This allows them to measure the effect that their code has on both the Pentium processor architecture and their product, and to fine-tune their application or system for optimal performance. The benefit to end users is better value and higher performance, due to the greater synergy between the Pentium processor, its host system, and application software.

Memory Page Size Feature

The Pentium processor offers the option of supporting either the traditional memory page size of 4 Kbytes, or a larger 4-Mbyte page. This feature-which is transparent to the application software—was provided to reduce the frequency of page swapping in complex graphics applications, frame buffers, and operating system kernels, where the increased page size allows users to map large, previously unwieldy objects. The larger page enables an increased page hit rate, which results in higher performance.

Upgradability

As with all new implementations of the Intel 32-bit microprocessor architecture, the Pentium processor has been designed for easy upgradability using Intel's upgrade technology. This innovation protects user investments by adding performance that helps to maintain the productivity levels of Intel processor-based systems over their entire lifespans.

Upgrade technology makes it possible for users to take advantage of more advanced processor technology in their existing systems. Intel will offer a future OverDrive(R) Processor for Pentium Processors. Users should contact system manufacturers for specific systems which support upgradability.

Advances in Manufacturing Process Technology

The use of sub-micron technology allows designers to develop smaller transistors and to fit more transistors on a smaller chip. Increasing the "transistor density" means that electrons have less distance to travel to complete circuits. This enables higher clock rates and results in higher performing processors.

The Pentium processor 75, 100, and 120 MHz are implemented in 3.3 volt, 0.6 micron BiCMOS technology with 3.3 million transistors each. The Pentium processor 120, 133, 150,166, and 200 MHz are the world's first volume microprocessors to be produced on a 0.35 micron process. They are implemented on 3.3 volt, BiCMOS technology with 3.3 million transistors. The 0.35 micron process enables an active die area which is approximately 50% smaller than that from the 0.6 micron process and enables much faster clock rates.

Desktop Processors

Core Freq. (MHz)	iCOMP(R) 2.0 Index	Voltage (Volts)	Bus Freq. (MHz)	# Transistors (Million)	Process (Micron)	Metal Layers
200	142	3.3	66	3.3	0.35	4
166	127	3.3	66	3.3	0.35	4
150	114	3.3	60	3.3	0.35	4
133	111	3.3	66	3.3	0.35	4
120	100	3.3	60	3.3	0.35/0.6	4
100	90	3.3	66	3.3	0.6	4
75	67	3.3	50	3.3	0.6	4

The increase in the number of transistors has made it possible to integrate components that were previously external to the processor, such as math coprocessors, caches and multiprocessor interrupt controllers, and place them on-board the chip. Placing components on-board decreases the time required to access them and increases performance dramatically. Another way to decrease the distance between components and at the same time increase the speed at which they communicate is to provide multiple layers of metal for interconnection among the transistors on the processor. Intel's 0.6 and 0.35 micron BiCMOS processes utilize four layers of metal interconnection.

Pentium Cache Structure : An Overview

As mentioned earlier, implementing a cache system is a feasible way of achieving a higher percentage of zero wait bus cycles when accessing memory.

Cache Working Principle

The working of the cache system is based on the principle of locality of reference. The locality of reference is of two basic types :

· **Temporal Locality**

Since programs run in loops, the same instructions may be fetched again on a frequent and continuing basis. This means that most programs have a tendency to use the most frequently accessed information again.

· **Spatial Locality**

Programs and the data they access tend to reside in consecutive memory locations. This means that programs are more likely to need code or data that are close to locations already accessed.

Cache Performance

The performance of a cache system is measured in terms of the cache hits. Hit rate is a term used to indicate the performance of a cache system and is defined by the following formula

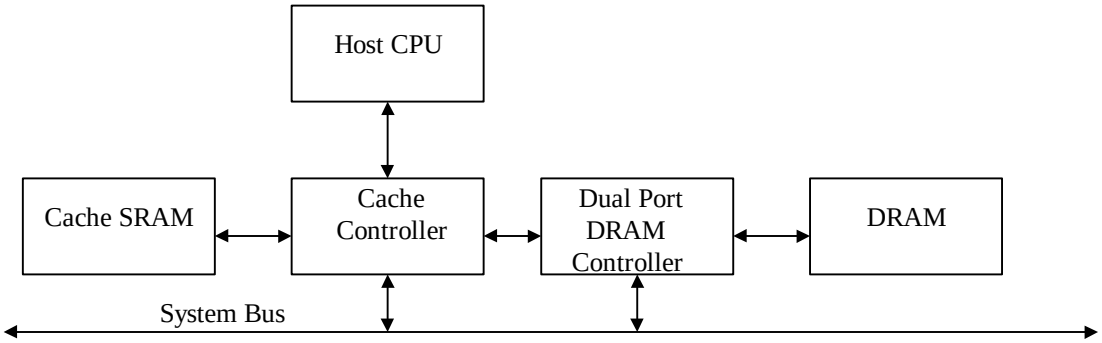
Hit Rate % = (Cache Hits / Total Memory Requests) * 100

The performance of a cache system is very dependable on the program the processor is executing. Thus a carefully designed program can greatly enhance the cache performance.

Cache Architecture

Two basic types of architectures exist

· **Look Through Cache**



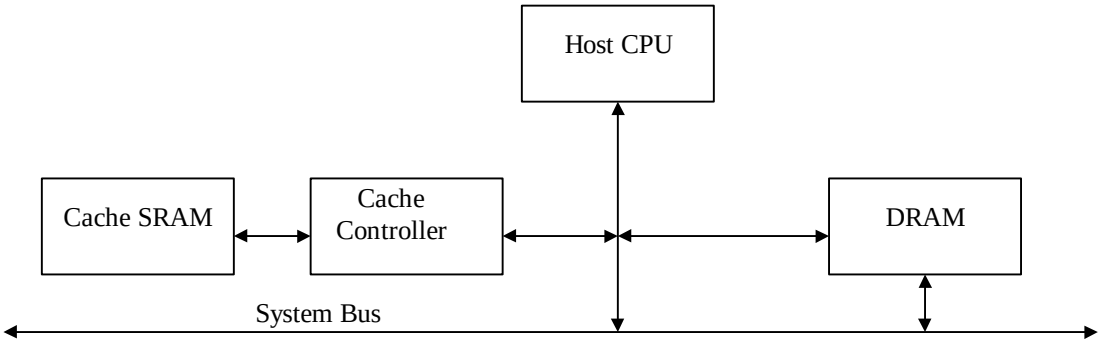
The performance of systems implementing this type of cache is typically higher than systems implementing the look aside cache system. All memory read and write requests from the processor pass through the cache controller. If the information is found, a zero wait condition occurs. In case of a write request, even if the information is not found within the cache, the cache controller might trick the processor into thinking that the write occured in zero wait states. The cache controller buffers the write

information, and performs the actual write later. This design also has the advantage of allowing multiple bus masters at the same time. This is called concurrent bus operations. To summarize :

Advantages
- Reduces system and memory bus utilization
- Allows Bus Concurrency
- Allows write operations in zero wait states

Disadvantages
- All processor memory requests pass through the cache controller, and in case of misses a delay called the lookup penalty is added.
- The design is more complex and costly.

- **Look Aside Cache**



This design does not isolate the processor bus from the system and the memory bus. When the processor initiates a bus cycle, all devices in the system immediately detect the bus cycle address. The cache controller monitors each memory access request to determine if it has a copy of the requested information. If there is a hit, the controller terminates the bus cycle in zero wait states and also instructs the memory subsystem to ignore the request. To summarize :

Advantages
- Improved response for cache miss cycles
- Simplicity of design
- Lower cost to implement

Disadvantages
- System bus utilization is not reduced
- Concurrent bus operation is not possible

Cache Coherency

In order for everything to work properly, all bus masters must be guaranteed to receive the latest copy of the requested information.

Whenever a write hit occurs, the cache memory is updated. At this point, the contents of the cache memory are more current. This cache line is referred to as dirty or modified. The corresponding memory line must be updated. If this is not done, another bus master could receive stale data. Three write policies are used to prevent this consistancy problem :

· **Write Through**
Many look through designs pass each write operation initiated by the processor to the memory. This implementation is very effective but results in poor performance because every write must access slow memory.

· **Buffered Write Through**
A variant of the above design policy; it has the added advantage of providing zero wait state write operations for both hits and misses. The cache controller for all processor initiated writes tricks the processor into thinking that the information was written into memory in zero wait states. The cache actually stores the entire write operation, thus enabling it to complete the memory write later without impacting the processor performance. If however, two back to back writes occur, the second write would have wait states inserted, to allow the cache controller to finish the first write. Another bus master is not allowed to use the bus until the write-through is completed.

· **Write Back**
This design updates memory only when necessary. Three conditions would lead to an update :

1. Another bus master initiates a read access to stale memory data.
2. Another bus master initiates a write access to stale memory data.
3. A cache line that contains modified data is about to be overwritten by a newly acquired line from memory

When a cache line is updated, it is marked as dirty. The cache must monitor (or snoop) all reads and writes to memory from all other bus masters to catch accesses to stale memory data. This design is difficult to implement because some amount of intelligenct is built into it on when to write back dirty data.

Bus Snooping/Snarfing

Bus snooping is a technique used by cache subsystems to monitor accesses performed by other bus masters. If the line being accessed by the other bus master is also a cache hit, it is called a snoop hit.

- Consider a memory write request by another bus master. Now the data in the cache will be stale. Cache subsystems handle this situation in different ways depending on the design

1. Invalidate the line in cache, so that the next processor request for that line causes a miss.
2. Snarf the data from the system bus while its being written to the memory by the other bus master, and update the corresponding cache line with the captured data.

Consider a memory read request by another bus master. Depending on the write policy, the data in the memory may be stale. Typically, only cache designs using a write back policy must snoop the system bus for read operations. If the corresponding cache line is marked dirty, the cache controller forces the requesting bus master to backoff until the cache controller can write the dirty line to the memory line. Another solution is that the cache controller itself could supply the necessary data, and instruct the memory not to supply the data.

Cache Implementation in the Pentium

The Pentium processor improves the performance over the cache implementation in the 486 in several ways. First of all, the Pentium has separate data and code caches. The line size for each of these caches is 32 bytes requiring a burst of four consecutive transfers to fill the cache (data bus width is 64 bits).

The data cache(L1) in the processor can be implemented with a write through or a write back policy on a line-by-line basis. The code cache is read only, so it does not require a write policy. Maximum performance is achieved using an external L2 cache.

Case 1 : Write through L1 cache with a write through L2 cache
Assume that both L1 and L2 have copies of the line being written to the memory by another bus master. L2 cache snoops the address, and detects a snoop hit. After the write operation the iformation in the corresponding cache line in L2 will be stale, and hence it will be invalidated. The next access to that line in L2 by L1 will cause a miss. Since L2 had a snoop hit, L1 might also have a copy of the data. Therefore the snoop address is passed onto L1 and if there is a snoop hit, the corresponding line in L1 is invalidated. The non-cacheble address (NCA) logic monitors each address that the processor outputs to determine whether the location is cacheble.

Case 2 : Write back L1 cache with a write back L2 cache
The write-back policy adds to the complexity of the design

- **Memory Reads initiated by the Pentium Processor**

When either the prefetcher or one of the execution units requests information from an internal cache, the request is immediately fulfilled if a cache hit occurs. If the information is not in the internal cache, the processor must perform a cache line fill. The L2 cache determines the memory read request and interrogates its directory to determine if it has a copy. If there is a hit, L2 notifies the processor that the address is cacheble, and the cache line fill is fulfilled in a burst of four consecutive 64-bit transfers from L2. But, if L2 had a cache miss, then the request is passed on to the memory, and NCA evaluates if the address is cacheble. If the address is not cacheble the bus cycle is not converted into a cache line fill, and a single transfer bus cycle is run.

- **Memory Writes initiated by the Pentium Processor**

The internal data cache checks the target address of the memory write to determine if a copy exists. In the event of a miss, the Pentium processor initiates a single bus transfer cycle to the target location(s). This is received by L2, which in turn interrogates its directory. If there is a miss, and L2 does not have an allocate-on-write capability, the memory write cycle is passed on to the memory. If L2 supports allocate-on-write it will also perform a cache line fill.

Assume that a write hit occurred in L1, and the data was modified. Since L1 has a copy of this data and was modified, the copy of this data in L2 and the memory will be stale. Some time later, another bus master accesses the data. The L2 cache will snoop the address, and since it is unaware that the data has been changed in L1, will believe that the other bus master is getting clean data. Oops.

Write-Once Policy
The Pentium processor internal data cache implements write-back with a write-once policy to ensure that the above discussed problem. Iniitially, assume that L1, L2 and the memory have the same copy of the data. The data in L1 is marked as shared to force a write-through on the first L1 write hit. Once again assume that a write hit occurred in L1, and the data was modified. Since this data is marked shared, a write-through to L2 will occur, and the data in L2 will be modified and be marked dirty. Now the copy of the data in L1 is marked as exclusive (instead of shared), and subsequent write hits on this data are not written-through on L2, although the data is marked as modified (from exclusive). Some time later, another bus master accesses the data. The L2 cache snoops the address, and a cache hit occurs. The copy of data in L2 is marked dirty, and hence L2 causes the other bus controller to backoff. L2 also passes the snoop address to L1 where another cache hit occurs. In our case, the copy of line in L1 is marked modified and hence L1 initiates a burst memory wite-back cycle. Once the write-back is completed L2 releases the backoff on the other bus master. Now the data in L1, L2, and the memory match once again, and hence the data in L2 is marked clean, and that in L1 is marked shared.

- **Memory Reads initiated by the Other Bus Masters**

Since L1 data cache is write-back cache, the L2 cache must ensure that proper cache coherency is maintained. The pentium processor can not snoop the system bus directly. The above discussed write-once policy ensures that all other bus masters receive the latest copy of data.

- **Memory Writes initiated by the Other Bus Masters**

Assume that another bus master is performing a memory write, L2 is snooping the bus, and it experiences a snoop hit. Assume that the line is also marked modified. This means that the data in memory is stale, and a write-back must be performed. L2 causes the other bus controller to backoff and passes the snoop address to L1 where another cache hit occurs. The action taken by L2 depends on the results of the L1 snoop. If L1 does not have a copy of the data, then L2 initiates a memory write transfer. If L1 has the data marked in exclusive or shared state, it will invalidate the line. L2 will then initiate a write-back transfer. If L1 has the data marked as modified, L2 waits for the Pentium to initiate a write-back transfer.

Once the write-back is completed L2 releases the backoff on the other bus master. The copy of the data in both L1 and L2 is invalidated, since the other bus master modified it in memory.

Pentium Bus Cycles

The Pentium processor runs several types of bus cycles, These may be divided into three general catagories

- Burst Bus Cycle
- Single Transfer Bus Cycle
- Special Cycles
- Interrupt Acknowledge Bus Cycle

Burst Bus Cycles

The Pentium processor runs burst bus cycles only when an entire cache line (32 bytes) is to be transferred to or from the Pentium processor. More specifically, the only burst bus cycles occur during a cache line-fill or cache line write-back operations. Note that the burst bus cycle can be run only when the processor asserts the CACHE# signal.

A cache line fill occurs when an access to memory results in a data cache or instruction cache miss. When a miss occurs, a line fill request is sent from the cache to the bus unit, which in turn initiates the burst bus cycle. A cache line fill occurs when the KEN# signal is sampled asserted by the processor. If the KEN# is deasserted, then a single read transfer is performed.

Write-back burst bus cycles occur when the processor must write an entire cache line back to the memory, which may occur due to three reasons :
1. An external snoop hit to a modified line in the internal data cache.
2. An internal snoop hit to a modified line in the internal data cache.
3. A modified line in the internal data cache must be replaced by a newly acquired line from the memory.

Single Transfer Bus Cycles (Non-Burst)

These bus cycles are run by the Pentium processor when accessing non-cacheble address space. Non-cacheble address space includes all I/O locations, and other memory locations deemed non-cacheble. Such locations include :

- All memory locations when the internal caches are disabled.
- Locations programmed as non-cacheble via the Non-Cacheble Address (NCA) logic. The processor asserts the CACHE# output, informing external logic that it wishes to perform a cache line-fill, but NCA returns a deasserted KEN#. When the processor samples the deasserted KEN#, it ends the bus cycle after a single transfer.
- Locations within a page of memory, page table or page table directory that have been designated non-cacheble by the operating system.

Single transfer cycles cna be categorized into four basic types

1. Non-Cacheble Memory Reads
2. Non-Cacheble Memory Writes
3. I/O Reads
4. I/o Writes

Special Cycles

During special cycles, the address and data bus are floated except durin the branch trace message cycle. When external logic detects a special cycle is in progress, then the byte enables (BE5#:BE0#) are decoded to determine which special cycle is being run. Special cycles include:

Special Cycle Type	BE5#:BE0#
Shutdown	111110
Flush	111101
Halt	111011
Write-Back	110111
Flush Acknowledge	101111
Branch Trace Message	011111

Interrupt Acknowledge Bus Cycle

The Pentium processor performs two back-to-back interrupt acknowledge bus cycles when an interrupt request (INTR) is recognized. The processor uses these cycles to communicate with the interrupt controller(s). It is the responsibilty of the designer to insert wait states into the interrupt acknowledge bus cycles to meet setup and hold requirements of the interrupt controllers.

The Bus Cycle State Machine

The Pentium processor's bus unit has six bus states as follows :

Ti	Bus Idle State	No bus cycles are being run
T1	Address Time	The first clock cycle of a bus when no other cycles are outstanding. The address and bus cycle definition are driven during T1 and ADS# is asserted. No bus cycle pipelining is occuring.
T2	Data Time	The second and subsequent clocks in a bus cycle. If a read bus cycle is being run, data is latched when BRDY# is sampled asserted at the end of T2. No other bus cycles are currently running.
T12	Address Time (2nd cycle pipelined) and Data Time (1st cycle in progress)	T12 represents a pipelined state in which the address phase of a newly a pipelined cycle and the data time for the cycle already in progress occur simultaneoulsy. In short the processor is still in the T2 state for the current cycle and has entered the T1 state for the next cycle.
T2P	Data Time (1st cycle) and Data Time (2nd cycle pipelined)	Both cycles are in the T2 state. BRDY# is sampled for the first bus cycle initiated by the processor. When this cycle completes, the state transitions to T2, indicating only one outstanding cycle.
TD	Dead State	This state indicates that there is one outstanding bus cycle, and BRDY# is not being sampled because the data bus transceivers need time to turn around between consecutive reads and writes and vice versa. This state occurs only during pipelined transfers.

INTRODUCTION TO THE MESI MODEL

In multi-processing environments where several cache subsystems exist, maintaing cache coherency becomes even more challenging. More than one cache may contain a copy of information from agiven memory location. There must be a method which guarantess that access to any memory location will provide the latest information. The MESI (**M**odified **E**xclusive **S**hared **I**nvalid) cache consistancy model provides a method of tracking the various states of a cache line to ensure consistancy across all possible soureces. A brief description of the four states follows :

· **Modified (M)**
The line in the cache has been updated due to a write hit in the cache. This alerts the cache to snoop the system bus and to write the line back when a snoop hit to that line is detected.

· **Exclusive (E)**
This state indicates that this cache knows that no other cache has this line, and therefore it is exclusive to this cache.

· **Shared (S)**
This state indicates that this line may be present in several other caches, and each cache has the same copy of the information (clean copy).

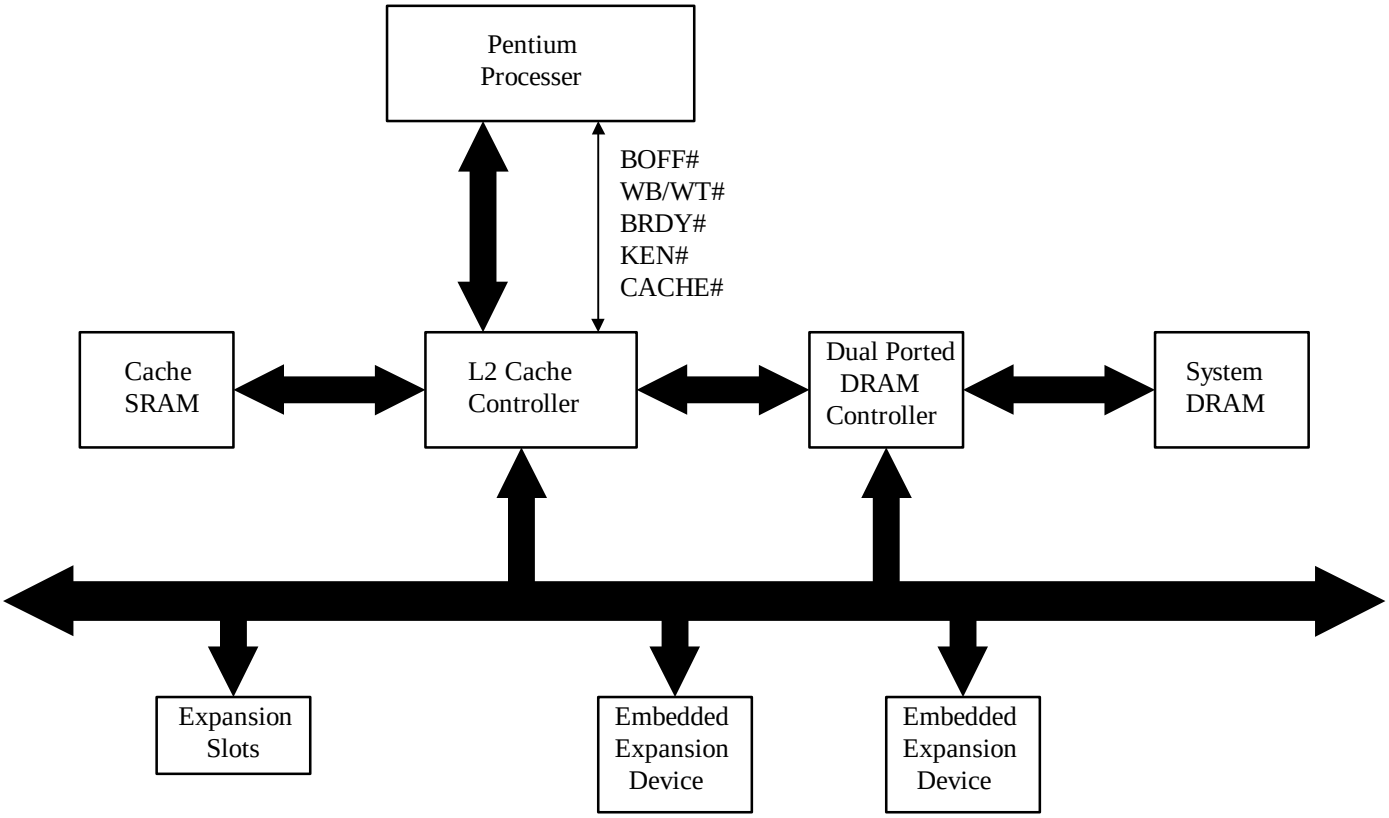
· **Invalid (I)**
This state indicates that the line is not preset in the cache, and will therefore cause a read/write miss.

Every line in cache is assigned one of these states. Transistions from one state to another may be caused by a local processor read/write, or by a bus snoop. The MESI model was created primarily to suppoert multi-processor systems, but is also used by the pentium for single processor systems

Single Processor MESI Implementation

A cache line may go through a number of MESI state transitions after it is first read into the L2 cache, and the processors L1 data cahce. The Pentium processor includes signals that allow external logic to tell it when to snoop the buses, as well as signals to tell the processor which state the cache line should be placed int after each bus cycle or snoop. These signals allow the pentium data cache to function as a write-back cache and still maintain cache coherency.

During reset, the MESI state for allcache lines in L1 and L2 caches are forced to the I state. This causes all initial accesses to L1 and L2 caches to be misses. The L1 code cache only uses a subset of the MESI model because it is a read only cache, and hence, for example, the M state would never be used. In this paper, we cover the MESI model when applied to the L1 data cache since it would be a superset of that applied to the L1 code cache. Also, the L2 cache is assumed to write-back, look through, that implements a back off when it detects a snoop hit. The L2 cache also supports the write-once policy.



Initial Read from System Memory
When one of the execution units request information from the L1 data cache (from now on L1), since all cache lines are reset to the I state, a read miss will occur. Similarly, a read miss occurs in L2, and the transfer is made from system memory. The state of the cache line in L2 is set to E, since no other processor-L2 combination exists (single processor system). The state of the line in L1 is set to S because L2 drives the WB/WT# signal low. This action enforces the write-once policy.

First Write to the L1 Cache Line
When the first write hit occurs in L1, the cache line is updated, and since the state of the line is S, L2 is notified that the cache line in L1 is being modified. L2 in turn interrogates its directory, and finds a copy of it in the E state. L2 updates the cache line and sets the state to M. L2 now asserts WB/WT# indicating to L1 to transition the cache line's state from S to E. Susequent write to this cache line in L1 will now be handled using the write-back policy.

Another Bus Master Access to the same Memory Location

· **Read Access**
L2 snoops the address driven by the other bus master, and detects a snoop hit. The cache line is in the M state. This means that if the other bus master were allowed to complete its read access, it would get stale data. To prevent this the other bus master is backed off by L2, preventing it from completing the bus cycle. L2 also transfers the snoop address to L1. This is necessary since L2 can not be certain that it has the latest copy of the information. While all this is going on, the L1 cache might be in the process of performing a cache line fill or accessing a decive on its local bus. The L2 cache therefore first, asserts the AHOLD signal forcing the Pentium to float its address bus, then passes the snoop address onto the processors local bus, and finally asserts the EADS# causing the L1 cache to snoop the address. The line is

found in the E state and transitions it to the S state because it knows that another bus master is accessing the line and may place it in its local cache. L1 also aserts the HIT# signal, and desserts the HITM# signal. L2 samples these signals and realizes that it does in fact have the latest copy. It noe performs a write-back to the memory. Back off is now released, and the other bus master is allowed to complete its read access. The L2 cache line state transitions from M to S.

· **Write Access**

L2 snoops the address driven by the other bus master, and detects a snoop hit. The cache line is in the M state. If the other bus master was allowed to complete its write to the memory location, it would be updating a stale line. Also, since the cache line in L2 is in the M state, some portion of the line has been modified but not written back to memory. To prevent this the other bus master is backed off by L2, preventing it from completing the bus cycle. L2 also transfers the snoop address to L1. This is necessary since L2 can not be certain that it has the latest copy of the information. While all this is going on, the L1 cache might be in the process of performing a cache line fill or accessing a decive on its local bus. The L2 cache therefore first, asserts the AHOLD signal forcing the Pentium to float its address bus, then passes the snoop address onto the processors local bus, and finally asserts the EADS# causing the L1 cache to snoop the address. The line is found in the E state and transitions it to the I state since L2 has asserted the INV signal. L1 also aserts the HIT# signal, and desserts the HITM# signal. L2 samples these signals and realizes that it does in fact have the latest copy. It now performs a write-back to the memory. Back off is now released, and the other bus master is allowed to complete its write access. The L2 cache line state transitions from M to I.

Second and Subsequent write to the Internal Cache

Assume that the line in L1 is in the E state, and that in L2 is in the M state. A second internal write hit causes the state of the line in L1 to transition from E to M. On all subsequent writes the state of the line in L1 stays M. Note that no write through to L2 occurs during the second and subsequent writes, and that at this point L2 is not aware of any changes to the line in L1.

Another Bus Master Access to the same Memory Location

· **Read Access**

L2 snoops the address driven by the other bus master, and detects a snoop hit. The cache line is in the M state. This means that if the other bus master were allowed to complete its read access, it would get stale data. To prevent this the other bus master is backed off by L2, preventing it from completing the bus cycle. L2 also transfers the snoop address to L1. This is necessary since L2 can not be certain that it has the latest copy of the information. While all this is going on, the L1 cache might be in the process of performing a cache line fill or accessing a decive on its local bus. The L2 cache therefore first, asserts the AHOLD signal forcing the Pentium to float its address bus, then passes the snoop address onto the processors local bus, and finally asserts the EADS# causing the L1 cache to snoop the address. The line is found in the M state and transitions it to the S state because it knows that another bus master is accessing the line and may place it in its local cache. L1 also aserts the HIT# and the HITM# signal. L2 samples these signals and realizes that it does not have the latest copy. L1 now performs a write-back to the memory. At this point L1, L2 and the memory, all hav the same line. Back off is now released, and the other bus master is allowed to complete its read access. The L2 cache line state transitions from M to S.

· **Write Access**

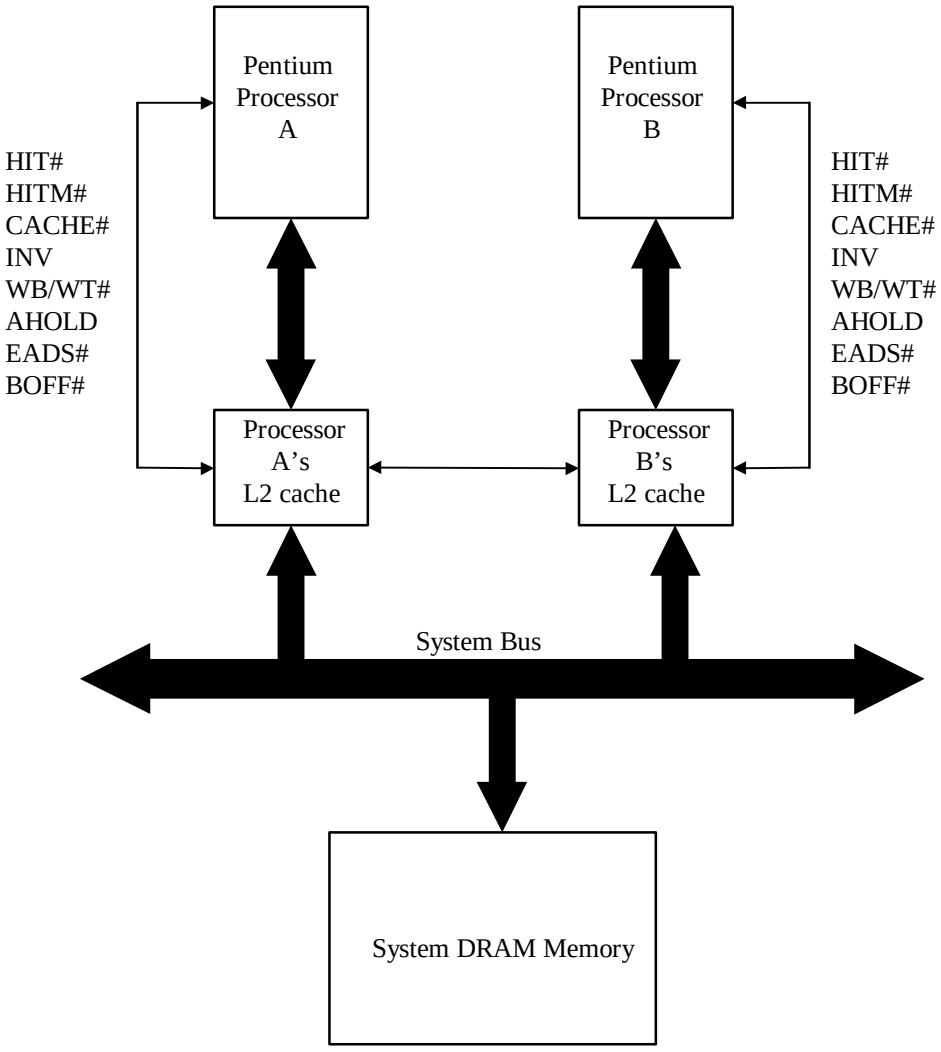
L2 snoops the address driven by the other bus master, and detects a snoop hit. The cache line is in the M state. If the other bus master was allowed to complete its write to the memory location, it would be updating a stale line. Also, since the cache line in L2 is in the M state, some portion of the line has been modified but not written back to memory. To prevent this the other bus master is backed off by L2, preventing it from completing the bus cycle. L2 also transfers the snoop address to L1. This is necessary since L2 can not be certain that it has the latest copy of the information. While all this is going on, the L1 cache might be in the process of performing a cache line fill or accessing a decive on its local bus. The L2

cache therefore first, asserts the AHOLD signal forcing the Pentium to float its address bus, then passes the snoop address onto the processors local bus, and finally asserts the EADS# causing the L1 cache to snoop the address. The line is found in the M state and transitions it to the I state since L2 has asserted the INV signal. L1 also aserts the HIT# and the HITM# signal. L2 samples these signals and realizes that it does not have the latest copy. L1 now performs a write-back to the memory. Back off is now released, and the other bus master is allowed to complete its write access. The L2 cache line state transitions from M to I.

Multi-Processor MESI Implementation

We will consider the case of a tightly coupled multi-processor system, consisting of two pentium processors each with its own L1 and L2 caches. As in the single processor system both L1 and L2 caches are look through employing a write-back policy. In this way each processor spends more time accessing its cache subsystem, rather than accessing the common DRAM memory. This not only increases efficiency, and speed, but also reduces the system bus load. The term processor complex in the following discussion is used to identify a processor and its local L2 cache system.

In a multiprocessor system more than one processor complex might have acopy of the same line in its cache(s). The MESI model ensures consistancy between the multiple copies.



CASE 1 : Read by Processor B from a Line Present in Processor A's Cache

The action taken by each processor complex depends on :
1. The current state of its cache line
2. The type of transfer in progress
3. The state of the snoop related signals generated by the other processor complex.

Consider the case where processor A has just completed a read from a memory location for the first time. The actions taken by the pentium processor and its L2 cache would be the same as in the single processor case. The state of the cache line in processors A's L1 cache is S and that of L2 is E. Now suppose processor B initiates a read to the same memory location, and both L1 and L2 have a miss. This results in a bus cycle to system memory to perform a cache line fill. Processor A's L2 cache detects this read bus cycle and snoops the address. Processor A's L2 cache has the line in the E state, and hence it experiences a snoop hit. Since another bus master is reading the same line from the memory, the state of that cache changes from E to S. Since the state of the line was E, no snoop cycle need to be sent to processor A. Also, processor A's L2 cache asserts the CHIT# signal thus notifying processor B's L2 cache that the memory location being accessed is also present in another cache system. After completing the line fill, processor B's line state is set to S because it sampled the CHIT# signal from processor A's cache.

Now consider the case where after initially reading the line from memory processor A updates the cache line. As in the single processor case the write-through (due to the write-once) occurs, and the state of the line in processor A's L1 cache transitions to E, and that of L2 to M. Once again, some time later processor B initiates a read to the same memory line and a memory read bus cycle is initiated and detected by processor A's L2 cache. A snoop hit occurs and the state of the line is M. Processor B's L2 cache is backed off to prevent it from receiving stale data. This is done by asserting the CHIT# and CHITM# signals. Processor B's L2 cache detects these asserted signals and in turn asserts the BOFF# signal to force processor B to suspend the current bus cycle. In the mean time processor A's L2 cache must determine if it has the latest copy of the cache line. The L2 cache asserts the AHOLD signal, passes the address on the local processor bus, and then asserts the EADS# signal. INV is deasserted because the other bus master is performing a read and there is no reason to invalidate the data in L1 in case of a snoop hit. A snoop hit does occur which is indicated by asserting the HIT# signal. The state of the line is E and hence L2 does have the latest copy. This is indicated by deasserting the HITM# signal. The state of the line in L1 is transitioned to S. The L2 cache will now perform a write back, and changes the line's state from M to S. It also deasserts CHIT# and CHITM#, thus releasing the back off on processor B's L2 cache which in turn deasserts the BOFF# signal. Once again processor B initiates the read bus cycle. Processor A's L2 detects a snoop hit and asserts CHIT#. Processor B's L2 cache rsponds by setting the state of this line to S. It also drives WB/WT# low so that processor B's L1 cache set the state of the line to S. Thus, both L1 and L2 caches of both processor complex have identical copies of the memory location in the S state. A write to this line in either processor complex will be written-through, and detected by the other complex's L2 cache.

Finally, consider the case where after initially reading the line from memory processor A updates the cache line. As in the single processor case the write-through (due to the write-once) occurs, and the state of the line in processor A's L1 cache transitions to E, and that of L2 to M. Processor A updates the cache line one more time changing the state of this line in L1 from E to M. This time however, no write-through occurs. Once again, some time later processor B initiates a read to the same memory line and a memory read bus cycle is initiated and detected by processor A's L2 cache. A snoop hit occurs and the state of the line is M. Processor B's L2 cache is backed off to prevent it from receiving stale data. This is done by asserting the CHIT# and CHITM# signals. Processor B's L2 cache detects these asserted signals and in turn asserts the BOFF# signal to force processor B to suspend the current bus cycle. In the mean time processor A's L2 cache must determine if it has the latest copy of the cache line. The L2 cache asserts the AHOLD signal, passes the address on the local processor bus, and then asserts the EADS# signal. INV is

deasserted because the other bus master is performing a read and there is no reason to invalidate the data in L1 in case of a snoop hit. A snoop hit does occur which is indicated by asserting the HIT# signal. The state of the line is M and hence L2 does not have the latest copy. This is indicated by asserting the HITM# signal. The state of the line in L1 is transitioned to S. Processor A now performs a write-back cycle. The L2 cache also updates its line, and changes the line's state from M to S. It also deasserts CHIT# and CHITM#, thus releasing the back off on processor B's L2 cache which in turn deasserts the BOFF# signal. Once again processor B initiates the read bus cycle. Processor A's L2 detects a snoop hit and asserts CHIT#. Processor B's L2 cache rsponds by setting the state of this line to S. It also drives WB/WT# low so that processor B's L1 cache set the state of the line to S. Thus, both L1 and L2 caches of both processor complex have identical copies of the memory location in the S state. A write to this line in either processor complex will be written-through, and detected by the other complex's L2 cache.

CASE 2 : Write by Processor B from a Line Present in Processor A's Cache

The action taken by each processor complex depends on :
1. The current state of the target cache line
2. The type of transfer in progress
3. The state of the snoop related signals generated by the processor complex A.

Consider the case where processor A has just completed a read from a memory location for the first time. The actions taken by the pentium processor and its L2 cache would be the same as in the single processor case. The state of the cache line in processors A's L1 cache is S and that of L2 is E. Now suppose processor B initiates a write to the same memory location, and both L1 and L2 have a miss. This results in a bus cycle to system memory to perform a cache line fill. Processor A's L2 cache detects this write bus cycle and snoops the address. Processor A's L2 cache has the line in the E state, and hence it experiences a snoop hit. Since another bus master is writing the same line to the memory, the state of that cache changes from E to I. Processor A's L2 cache also assumes that processor A's L1 cache also has a copy of the same line, and it therefore asserts AHOLD, passes the snoop address on the processor's local bus, asserts EADS#, and also asserts INV indicating that in case of a snoop hit the line is to be set to state I. A snoop hit does occur in L1, and HIT# is asserted which is ignored by L2 since all necessary action has already been taken. Processor B completes the write operation to the memory.

Now consider the case where after initially reading the line from memory processor A updates the cache line. As in the single processor case the write-through (due to the write-once) occurs, and the state of the line in processor A's L1 cache transitions to E, and that of L2 to M. Once again, some time later processor B initiates a write to the same memory line and a memory write bus cycle is initiated and detected by processor A's L2 cache. A snoop hit occurs and the state of the line is M. Processor B's L2 cache is backed off . This is done by asserting the CHIT# and CHITM# signals. Processor B's L2 cache detects these asserted signals and in turn asserts the BOFF# signal to force processor B to suspend the current bus cycle. In the mean time processor A's L2 cache must determine if it has the latest copy of the cache line. The L2 cache asserts the AHOLD signal, passes the address on the local processor bus, and then asserts the EADS# signal. INV is asserted because the other bus master is performing a read and there is no reason to invalidate the data in L1 in case of a snoop hit. A snoop hit does occur which is indicated by asserting the HIT# signal. The state of the line is E and hence L2 does have the latest copy. This is indicated by deasserting the HITM# signal. The state of the line in L1 is transitioned to I. The L2 cache will now perform a write back, and changes the line's state from M to I. It also deasserts CHIT# and CHITM#, thus releasing the back off on processor B's L2 cache which in turn deasserts the BOFF# signal. Once again processor B initiates the write bus cycle. This time processor A's L2 detects a snoop miss, and the write cycle is completed.

Next, consider the case where after initially reading the line from memory processor A updates the cache line. As in the single processor case the write-through (due to the write-once) occurs, and the state of the

line in processor A's L1 cache transitions to E, and that of L2 to M. Processor A updates the cache line one more time changing the state of this line in L1 from E to M. This time however, no write-through occurs. Once again, some time later processor B initiates a write to the same memory line and a memory write bus cycle is initiated and detected by processor A's L2 cache. A snoop hit occurs and the state of the line is M. Processor B's L2 cache is backed off. This is done by asserting the CHIT# and CHITM# signals. Processor B's L2 cache detects these asserted signals and in turn asserts the BOFF# signal to force processor B to suspend the current bus cycle. In the mean time processor A's L2 cache must determine if it has the latest copy of the cache line. The L2 cache asserts the AHOLD signal, passes the address on the local processor bus, and then asserts the EADS# signal. INV is asserted because the other bus master is performing a write and hence the data in L1 would be invalid in case of a snoop hit. A snoop hit does occur which is indicated by asserting the HIT# signal. The state of the line is M and hence L2 does not have the latest copy. This is indicated by asserting the HITM# signal. The state of the line in L1 is transitioned to S. Processor A now performs a write-back cycle. The L2 cache also updates its line, and changes the line's state from M to I. It also deasserts CHIT# and CHITM#, thus releasing the back off on processor B's L2 cache which in turn deasserts the BOFF# signal. Once again processor B initiates the write bus cycle. This time processor A's L2 detects a snoop miss, and the write cycle completes normally.

Finally, consider the case where both processor complexes contain identical copies of the same line of information. This means that all instances have the state set to S. Assume that processor complex B performs a write to its copy in L1. Since the state of the line is S, a write-through is initiated. Processor B's L2 cache also experiences a write hit and updates the line. It also initiates memory bus cycle, marks its copy as E, and informs L1 to keep its copy in the S state by driving WB/WT# low. In the mean time processor complex A's L2 cache detects a snoop hit on a shared line. It invalidates the line and also generates a snoop to processor A instructing processor A to invalidate the line in case of a snoop hit.

Much of the enhanced performance of the Pentium processor results from its dedicated code cache, the dedicated data cache, the dual integer pipelines, and the pipelined floating-point unit. Another feature that provides significant performance gains is the branch prediction mechanism.

THE CODE CACHE

The Pentium processor incorporates an 8KB, two way set associative code cache dedicated to feeding instructions to the prefetcher. Both the code and the data caches are disabled when the processor exits the RESET stage. The POST programmer must enable the caches by clearing the cache disable (CD) and not write-through (NW) bits to zero. When the internal caches are first enabled, the internal code cache contains no valid information, and the next prefetch results in a miss. This causes the cache to issue a cache line-fill request to the bus. If the locations being requested are cacheable, 32 bytes of code are transferred (burst cycle), otherwise a single transfer occurs. The first quadword is also routed to the prefetcher, which is what was requested by the prefetcher.

The cache banks are referred to as way zero and way one. This cache is a read-only cache, and hence implements a subset of the MESI model (namely S and I). Each cache line is 32 bytes wide, allowing 32 bytes to be delivered to the prefetch queue during a single prefetch. The cache directories are triple ported allowing three simultaneous directory accesses. Two of the ports support the split line accessibility. The 20-bit tag field within each directory identifies each page in memory that the cache line was copied from. A single state bit identifies the state of the line (S or I). Each tag also has a parity bit to detect errors. During a prefetch, the address bits A31:A12 of the prefetch address identify one of one million 4KB pages within the 4GB address space, address bits A11:A5 are used to identify the line where the target line resides in the cache or memory, and are used as the index into the cache directory. The lower 5 bits (A4:A0) of the prefetch address identify a byte within the line and is not used during the lookup.

The code cache considers the 4GB memory space to be divided into 1,048,576 pages, each of 4KB size. In addition each page is divided into 128 lines, each of which is 32 bytes in length. Assume that a line was fetched from line eight in memory page 20. The code cache uses the line number eight to index into entry eight of its twin directories and takes one of the following actions:

- If either of the directory entries is marked invalid, the target page address is stored in that directory address as the tag address. The state bit is set to shared. The line fetched is stored in line eight of the code cache way associated with this directory. The LRU bit associated with the pair of directory entries is set to reflect that entry eight in the opposite directory is now the least recently used of the pair.
- If both entries are marked valid, the code cache will overwrite the entry currently pointed to by the LRU bit. This is referred to as cache line replacement. The LRU bit associated with the pair of directory entries is now set to the opposite directory.

Internal Snooping

The code cache in addition to performing the regular external snooping, performs internal snooping. When the data cache initiates a read or a write operation, the code cache snoops the address, and if a snoop hit occurs, invalidates the entry in the code cache. Note that the entry is invalidated even for read operations.

Split-Line Access

In a CISC processor, instructions are of variable length. In the Pentium processor the smallest instruction is one byte while the maximum legal length is 15 bytes. A code cache miss always results in a 32-byte cache line fill, if its a cacheable address. Multi-byte instructions may straddle two sequential lines stored in the code cache. When the prefetcher determines that the instruction is straddled across two lines, it would have to perform two sequential cache accesses, which would hamper performance. For this reason the Pentium processor incorporates a split line access which allows the upper half of one line and the lower half of the next line to be accessed in one cycle. When a split line access is made the bytes must be

rotated so that they are in proper order. In order for the split access to work efficiently, instruction boundaries within the cache line need to be defined. When an instruction is decoded the first time the length of the instruction is fed back to the cache. Each code cache entry marks instruction boundaries within the line so that if necessary split line accesses can be performed.

THE PREFETCHER

During normal operation with the caches enabled, the prefetcher receives a line of code from the bus unit, which it places in the active prefetch queue. The prefetcher has two prefetch queues, out of which only one is active at a given time. Each queue can hold two line (64 bytes) of information. The prefetcher continues to fetch instructions sequentially and place them in the active queue, until the branch prediction logic predicts that a branch will be taken. The branch prediction logic also provides the predicted branch target address to the prefetcher. The prefetcher now starts fetching instructions, sequentially, from this new address and switches the active queue. If the branch taken was correctly predicted, then the instructions following the jump instruction in the instruction pipelines are the correct instructions, and the execution unit does not stall. However if the prediction was wrong, and the branch was not taken, the instructions behind the jump instruction in the pipelines the active prefetch queue must be flushed. The prefetcher also switches back to the original queue, since that queue already has the correct instructions. The execution unit stalls briefly (three to four clock cycles). If, however, the branch prediction logic did not predict a branch that was taken, then the instructions in the pipelines behind the jump must be flushed, the active queue switched, and the new instructions fetched from the branch target address.

THE INTEGER INSTRUCTION PIPELINES

Decode One, Or the D1 Stage

In this stage the two instruction pipelines, each receive instructions simultaneously from the active prefetch queue. The D1 queue performs two checks :

- The two instructions are checked for pairability. If they are not pairable, the instruction in the V pipeline's D1 stage is removed, and the prefetcher will shift it into the U pipeline's D1 stage when the current instruction in the U pipeline's D1 stage moves into the D2 stage.
- If either instruction is a branch instruction, the branch prediction logic makes a prediction on whether the branch will be taken or not when the branch reaches the execution stage. If the branch is predicted taken the logic will tell the prefetcher to switch queues and start fetching new instructions from the branch target address.

When paired, the two instructions in both the U and V pipelines enter and leave each stage in the pipeline in unison. If the instruction in one pipeline stalls in one stage, the instruction in the other pipeline is not sent to the next stage until the other one is ready.

Rules For Instruction Pairing

Not all instructions are pairable. If two consecutive instructions are not pairable, then both of them must execute serially in the U pipeline. Instructions are pairable if and only if the following criteria are met:

1. Instructions must be simple. Single instructions are hardwired to execute quickly in a single clock cycle in the Pentium processor. Some instructions requiring two or three clock cycles are also simple instructions. Examples of some simple instructions are : nop; mov reg, reg/mem/imm; mov mem, reg/imm; inc reg/mem.

2. Instructions must not have register contention. Special hardware allowing some exceptions are added. Register contention occurs when two instructions attempt to access the same registers during parallel execution. Two types of register contention exist :

· **Explicit Register Contention**
This occurs when two instructions specify access to the same register

e.g. 1. Consider the two sequential consecutive instructions mov bx, 8c and mov ax, bx.
In this example we have a register write followed by a register read. If these two instructions are executed in parallel then the first instruction would load bx with 0x8c, while the second one would read the old value of bx into ax (instead of 0x8c).

e.g. 2. Consider the instructions mov ax, bx and mov bx, [bp]
In this example no register contention occurs. The first instruction executes in the U pipeline and reads from bx during the execute stage and writes to ax during the write-back stage. The second instruction reads the contents of memory pointed to by [bp] during the execution stage and writes it to bx during the write-back stage.

· **Implicit Register Contention**
This occurs when two instructions imply reference to the same register. The same read/write dependencies apply as in the explicit case. Exceptions are flags references (cmp followed by jcc) and stack pointer references (pushes and pops).

3. When entering the pipeline for the first time, an instruction is determined to be one byte in length. In this case the prefetcher can easily determine the boundary between this instruction and the next. When a multi-byte instruction is executed for the first time, the instruction pipeline provides feedback to the code cache indicating the instruction's length. The code cache stores this boundary information in the cache directory entry corresponding to the line instruction was originally fetched from.

4. When an instruction entering the pipeline has been fetched from the code cache before, the code cache also delivers its length, allowing the prefetcher to easily determine the boundary between this instruction and the next.

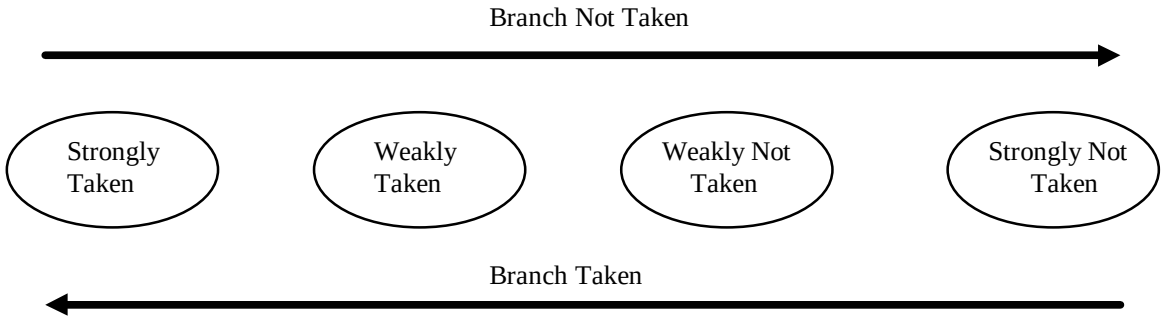
Instruction Branch Prediction
The Pentium processor has branch prediction logic, allowing it to avoid pipeline stalls if it correctly predicts whether or not the branch will be taken when the branch instruction is executed. However, if the branch prediction is not correct a penalty of three cycles is incurred if the branch was executed in the U pipeline, and four cycles if it was executed in the V pipeline.

The prediction mechanism is implemented using a four way, set associative cache with 256 entries. This is referred to as the Branch Target Buffer (BTB). The directory entry for each line contains the following information :
- A valid bit that indicates if the entry is in use.
- History bits that track how often the branch has been taken each time it has entered the pipeline before.
- The source memory address that the instruction was fetched from.

If the directory entry is valid, the target address of the branch is stored in the corresponding data entry of the branch target buffer.

The BTB is a look-aside cache that sits off the side of the D1 stages of the two pipelines and monitors for branch instructions. The first time the instruction enters either pipeline, there is a BTB miss. It therefore predicts that the branch will not be taken, even if it is an unconditional branch. When the instruction reaches the execution stage, the branch is either taken or not taken. When the branch is taken for the first time, the execution unit provides feed back to the branch prediction logic. The branch target address is recorded and a directory entry is made in the BTB containing the source memory address, and the history bits are set to strongly taken. The history bits can be one of the following :

Strongly Taken	When an entry is first made the history bits are initialized to this state. In addition if a branch marked weakly taken is taken again the history bits are upgraded to strongly taken. Also, if a branch marked strongly taken is not taken it is downgraded to weakly taken.
Weakly Taken	If a branch marked weakly taken is taken again the history bits are upgraded to Strongly taken. If a branch marked weakly taken is not taken it is downgraded to weakly not taken.
Weakly Not Taken	If a branch marked weakly not taken is taken again the history bits are upgraded to weakly taken. If a branch marked weakly not taken is not taken it is downgraded to strongly not taken.
Strongly Not Taken	If a branch marked strongly not taken is taken again the history bits are upgraded to weakly not taken.



Decode Two, Or the D2 Stage

During the D2 stage, addresses are calculated for operands that reside in memory. The D2 stage performs the same basic function as the segment unit in the i486, but it has been enhanced for higher speed. The Pentium processor can handle instructions with displacement or immediate values, or instructions using base and index addressing in a single clock cycle. The processor also performs segmentation protection checks which is required in the protected mode.

The Execution Stage

The execution stage is comprised of the ALU. The U pipeline's ALU incorporates a barrel shifter, while the V pipeline's ALU does not. Therefore, the V pipeline can not handle all instructions. Access to the data cache can be performed by the U and V pipeline simultaneously. Note that both instructions enter the execution stage in both pipelines at the same time. If the instruction in the V pipeline stalls, the instruction in the U pipeline is allowed to proceed to the write-back stage. However, if the instruction in the U pipeline stalls, the instruction in the V pipeline must wait. No additional instructions are transferred to the execution stage in either pipeline until both instructions are sent to the write-back stage.

The Write-Back Stage

Target registers, including the EFLAGS register are updated (if necessary) during this stage.

THE FLOATING-POINT PIPELINE

The floating-point unit (FPU) in the Pentium has been significantly improved over the i486 floating-point unit. The new floating-point unit is heavily pipelined, permitting several instructions to execute simultaneously. Most floating-point instructions are issued singly to the U pipeline and can not be paired with integer instructions. However some floating-point instructions may be paired.

The floating-point pipeline consists of eight stages. The first four are shared with the integer pipeline The pipeline stages are as follows:

Stage	Description
Prefetch	Identical to the Integer Prefetch
Instruction Decode 1 (D1)	Identical to the Integer Instruction Decode 1 (D1)
Instruction Decode 1 (D1)	Identical to the Integer Instruction Decode 2 (D2)
Execution	Register read, memory read, memory write as required
FP Execution 1	Information from the register or memory is written to a FP register. Data is converted to FP format before loading into the FPU
FP Execution 2	Floating-point operation is performed within the FPU
Write FP Result	The FPU results are rounded and written to the target floating-point register
Error Reporting	If an error is detected, an error reporting stage is entered where the error is reported and the FPU status word is updated

FPU Pipelining Restrictions
1. FDIV can not executed simultaneously with any other instructions.
2. Two consecutive FMUL instructions can not execute simultaneously.
3. One FMUL can be executed simultaneously with one or two FADD instructions
4. Three FADD instructions can be executed simultaneously.

THE INTERNAL DATA CACHE

All instructions requiring access to the memory are routed to the processor's internal data cache to determine if a copy of the target data is already on the processor. Most Pentium processor systems employ a level two (L2) cache subsystem to increase the performance when a miss occurs in the internal data cache (L1). The Pentium processor has an 8KB, two way set associative data cache, so that like the code cache, each cache way is 4KB in size. This means that the total 4GB address space is viewed as 1048576 pages, each of which is 4KB in size. Each page in memory is viewed by the cache controller to be organized like the cache ways. This means that each page in memory is seen to have 128 lines with each line 32 bytes in length. The L1 cache is interleaved on four byte boundaries (double word) to permit simultaneous double word accesses from both the U and V pipelines. As long as the two do not access the same bank, teo double words can be read from the data cache in the same clock cycle. Each memory address sent to the internal cache controller is used to determine the page (A31:A12), line (A11:A5) and double word (A4:A2) that contain the target memory address (Note: A1:A0 are unused).

When a line of information is read from a page of external memory, the cache controller stores the line in one of the two cache memory ways. Within the selected way, it stores the line in the same line number as it is in the memory page. This line number also acts as a lookup index for the directory. Each directory entry has a tag field used to record the page number of the memory page that the line of information came from. When a line of information is read from the memory, the cache controller indexes into the directory using the line number. It then examines the state bits (2). If the state for this line is I, no further checking is necessary and it proceeds to way one. If the line is valid, the cache controller compares the page portion of the memory address with the tag field of the directory entry. If it matches, it means that the cache has a copy of the information. If the line is invalid in both cache ways or the page addresses do not match, it results in a cache miss. After the line of information is fetched, it must be stored in the cache and its state must be set according to the MESI model. If the line is being used in both cache directories, the current setting of the LRU is used to determine which cache line to overwrite. The LRU bit for this line must now be updated to point to the opposite cache.

Anatomy of A Read Hit
Cosider the following two instructions being executed

```
mov AX, [1056]
mov BX, [108C]
```

Also assume that the processor is in protected mode, the data segment starts at memory location 002A0000h, and paging is turned off. The first instruction in the U pipeline tells the processor to read two bytes of infromation from memory locations 002A1056h and 002A1057h and place them in the AX register. The second instruction in the V pipeline tells the processor to read two bytes of infromation from memory locations 002A108Ch and 002A108Dh and place them in the BX register.

At the moment of access, the following conditions are assumed in these entries

Directory Status	Entry 2	Entry 4	Entry 7C
LRU bit	1	0	0
Status Bits Directory 0	S	S	M
Status Bits Directory 1	S	E	S
Tag Field Directory 0	002A1	002A1	00F5
Tag Field Directory 1	00FB6	0385C	00F49

Both the U and V pipe addresses lie in page 2A1h in memory. The U address is in line two double word five of page 2A1h and the V address is in line four double word three of the same page. Thus the same bank will not be accessed for both locations, they can be accessed simultaneously. When the data cache controller evaluates the state bits for the U pipeline access, the line 2 entry in directory zero shows that the data is shared. Since it is not invalid, it compares the page portion of the address with the tag field in entry two and finds a match. Hence a read hit is detected. Since the address bits A4:A2 have a five, the bank select activates bank 5 via the bank select logic and the target locations are accessed and sent back to the U pipe for a register write back. Simultaneously, the data cache controller also evaluates the state bits for the V pipeline access. Once again a hit is detected in cache way zero. Since the bank being accessed is three, it is activated, target locations accessed and the information set back to the V pipeline for a register write back.

The LRU bit for entry two is already one so no change is required, but for entry four, the LRU bit must be changed to reflect the latest access to way zero. Hence it is changed to one.

If at the same time another bus master is accessing a memory location which causes a snoop hit in the L2 cache. The L2 cache causes the other bus master to back off and passes the snoop address to the Pentium processor. Since the data cache directory is tripple ported, a snoop lookup can occur during U and V lookups.

If both instructions in the U and V pipes try to access the same banks within the data cache, a bank conflict occurs, because the banks are single ported. In such instances the U pipe has priority over the V pipe instruction. A one clock penalty on the V pipe instruction is incurred.

Anatomy of A Read Miss

Cosider the following two instructions being executed

```
mov EAX, [0054]
mov EBX, [008C]
```

Also assume that the processor is in protected mode, the data segment starts at memory location 00000000h, and paging is turned off. The first instruction in the U pipeline tells the processor to read two bytes of information from memory locations 00000054h and 00000057h and place them in the EAX

register. The second instruction in the V pipeline tells the processor to read two bytes of infromation from memory locations 0000008Ch and 0000008Fh and place them in the EBX register.

At the moment of access, the following conditions are assumed in these entries

Directory Status	Entry 2	Entry 4	Entry 7C
LRU bit	1	1	0
Status Bits Directory 0	S	S	I
Status Bits Directory 1	S	E	I
Tag Field Directory 0	002A1	002A1	00F5
Tag Field Directory 1	00FB6	0385C	00F49

In this case, it is obvious that a cache miss will occur for both the U and V pipeline instruction reads. When a miss occurs the internal data cache issues a cache line fill request to the bus unit which attempts to perform the line fill. In this case two cache line fill requests are issued. The U pipe read occurs first followed by the V pipe read. Let us consider a typical scenario where the U pipe request is found in the L2 cache memory, but the V pipe request is not found in L2.

Lets start with the U pipe access. The bus cycle begins when the processor outputs the address and the bus cycle definition. ADS# is asserted indicating that the current address and bus cycle definition on the bus are valid. When driving the address, the bus unit strips off the least three significant bits of the address, and converts them into appropriate byte enable signals. The address A31:A3 inicates 00000050h in which locations 54h to 57h reside. Byte enable lines BE7#:BE4# further specift that only the upper four bytes are requested by the U pipe instruction. The processor also sets PCD low, indicating that the address resides in a cacheable page. Note that when paging is turned off, the state of PCD is determined by the state of the CD bit. In this case the cache is enabled, and hence CD is set low. L2 receives the request and performs a cache lookup, and detects a hit. L2 asserts the KEN# signal. Normally cache line fills are fulfilled by burst bus cycles. The processor indicates that by asserting the CACHE# signal. However, until it samples an asserted KEN# signal, it does not know if the address is cacheable. The processor samples the KEN# signal when BRDY# is sampled asserted at the end of T2 time. The processor captures the first quadword (50-57h) and proceeds with the rest of the transfer. The cache line is placed in line two of the data cache. The LRU bit for entry two indicates that line two of cache way one has been least recently used, and must be replaced. In this case valid data is overwritten. The LRU bit is updated to zero, to indicate that line two of way zero is now the least recently used. The line in L2 is assumed to have been in E state, and hence it drove the WB/WT# signal low. This causes the state of the line in L1 to be set to S.

Now lets consider the V pipe access. This results in a miss in L2, and L2 transfers the bus cycle to the memory buses. Now, the NCA logic examines the address to see if it is cacheable. We assume that it is. The NCA will therefore assert KEN# and pass it to the L2 cache. When the memory is ready to transfer data it issues BRDY# to the L2 cache. The L2 cache in turn issues BRDY#, KEN# and WB/WT# signals to the processor. The line read from memory must be placed in line four of the data cache. Once again the LRU bit is used to determine which cache way to use (since line four in both ways has valid data). Since the replaced line state is S, no write-back to L2 and memory is required. Hence the replacement is said to be clean.

Anatomy of a Write Hit

When an instruction is executed that requires a write to memory, the internal data cache interrogates its internal directory to see if it has a copy of the information. If the information is found it is updated, Depending on the state of the line different situations arise:

- Write to a line in the S state

The processor updates the line in the internal cache, and performs a write-through to external memory. The processor does not perform burst cycles when a write-through occurs. Burst cycles occur only during write-back cycles

- Write to a line in the E state

The location is updated but no bus cycles are generated. The state will however change from E to M.

- Write to a line in the M state

The location is updated. No bus cycles occur and the state does not change.

Anatomy of a Write Miss

When a write miss in the internal data cache occurs, a write bus cycle is intiated to write the target information in the memory. A single non-burst write transfer occurs each time a write miss occurs. When backs to back memory write operations occur, write buffers are used to increase performance. In addition to this write buffer, there are three additional write buffers dedicated to write-back functions:

- The External Snoop Write Buffer

Used to store write-backs caused by an external snoop hit to a modified line

- The Internal Snoop Write Buffer

Used to store write-backs caused by an internal snoop hit to a modified line during a code cache miss

- Line Replacement Write Buffer

Used to store write-backs caused by the replacement of a least recently used line that is modified.

Write cycles are driven to the bus in the following priority order:

- Contents of the external snoop buffer
- Contents of the internal snoop buffer
- Contents of the replacement write buffer
- Contents of write buffers