1. Introduction

The purpose of this paper is two fold. The first part gives an overview of cache, while the second part explains how the Pentium Processor implements cache.

A simplified model of a cache system will be examined first. The simplified model is expanded to explain: how a cache works, what kind of cycles a cache uses, common architectures, major components, and common organization schemes.

The implementation may differ in actual designs, however the concepts remain the same. This eliminates unnecessary detail and background in hardware design or system dependency. The second part of this paper gives more detail and specifics of how the internal caches work on the Pentium Processor.

2. An Overview of Cache

Before I describe a basic cache model, I need to explain what Cache is. Cache is small high speed memory usually Static RAM (SRAM) that contains the most recently accessed pieces of main memory.

Why is this high speed memory necessary or beneficial? In today's systems , the time it takes to bring an instruction (or piece of data) into the processor is very long when compared to the time to execute the instruction. For example, a typical access time for DRAM is 60ns. A 100 MHz processor can execute most instructions in 1 CLK or 10 ns. Therefore a bottle neck forms at the input to the processor. Cache memory helps by decreasing the time it takes to move information to and from the processor. A typical access time for SRAM is 15 ns. Therefore cache memory allows small portions of main memory to be accessed 3 to 4 times faster than DRAM (main memory).

How can such a small piece of high speed memory improve system performance? The theory that explains this performance is called "Locality of Reference." The concept is that at any given time the processor will be accessing memory in a small or localized region of memory. The cache loads this region allowing the processor to access the memory region faster. How well does this work? In a typical application, the internal 16K-byte cache of a Pentium® processor contains over 90% of the addresses requested by the processor. This means that over 90% of the memory accesses occurs out of the high speed cache.[1]

So now the question, why not replace main memory DRAM with SRAM? The main reason is cost. SRAM is several times more expense than DRAM. Also, SRAM consumes more power and is less dense than DRAM. Now that the reason for cache has been established, let look at a simplified model of a cache system.

---

[1] Cache performance is directly related to the applications that are run in the system. This number is based on running standard desk top applications. It is possible to have lower performance depending on how the software is designed. Performance will not be discussed in the paper.

2.1 Basic Model

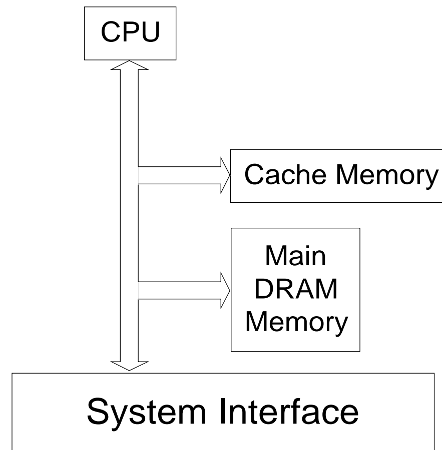

Figure 2-1 Basic Cache Model

Figure 2-1 shows a simplified diagram of a system with cache. In this system, every time the CPU performs a read or write, the cache may intercept the bus transaction, allowing the cache to decrease the response time of the system. Before discussing this cache model, lets define some of the common terms used when talking about cache.

2.1.1 Cache Hits

When the cache contains the information requested, the transaction is said to be a cache hit.

2.1.2 Cache Miss

When the cache does not contain the information requested, the transaction is said to be a cache miss.

2.1.3 Cache Consistency

Since cache is a photo or copy of a small piece main memory, it is important that the cache always reflects what is in main memory. Some common terms used to describe the process of maintaining cache consistency are:

2.1.3.1 Snoop

When a cache is watching the address lines for transaction, this is called a snoop. This function allows the cache to see if any transactions are accessing memory it contains within itself.

2.1.3.2 Snarf

When a cache takes the information from the data lines, the cache is said to have snarfed the data. This function allows the cache to be updated and maintain consistency.

Snoop and snarf are the mechanisms the cache uses to maintain consistency. Two other terms are commonly used to describe the inconsistencies in the cache data, these terms are:

2.1.3.3 Dirty Data

When data is modified within cache but not modified in main memory, the data in the cache is called “dirty data.”

2.1.3.4 Stale Data

When data is modified within main memory but not modified in cache, the data in the cache is called stale data.

Now that we have some names for cache functions lets see how caches are designed and how this effects their function.

2.2 Cache Architecture

Caches have two characteristics , a read architecture and a write policy. The read architecture may be either “Look Aside” or “Look Through.” The write policy may be either “Write-Back” or “Write-Through.” Both types of read architectures may have either type of write policy, depending on the design. Write policies will be described in more detail in the next section. Lets examine the read architecture now.

2.2.1 Look Aside

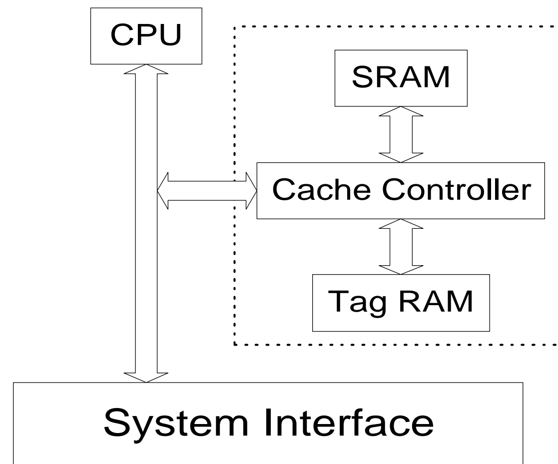

Figure 2-2 Look Aside Cache

Figure 2-2 shows a simple diagram of the “look aside “cache architecture. In this diagram, main memory is located opposite the system interface. The discerning feature of this cache unit is that it sits in parallel with main memory. It is important to notice that both the main memory and the cache see a bus cycle at the same time. Hence the name “look aside.”

2.2.1.1 Look Aside Cache Example

When the processor starts a read cycle, the cache checks to see if that address is a cache hit.

HIT:

If the cache contains the memory location, then the cache will respond to the read cycle and terminate the bus cycle.

MISS:

If the cache does not contain the memory location, then main memory will respond to the processor and terminate the bus cycle. The cache will snarf the data, so next time the processor requests this data it will be a cache hit.

Look aside caches are less complex, which makes them less expensive. This architecture also provides better response to a cache miss since both the DRAM and the cache see the bus cycle at the same time. The draw back is the processor cannot access cache while another bus master is accessing main memory.

2.2.2 Read Architecture: Look Through

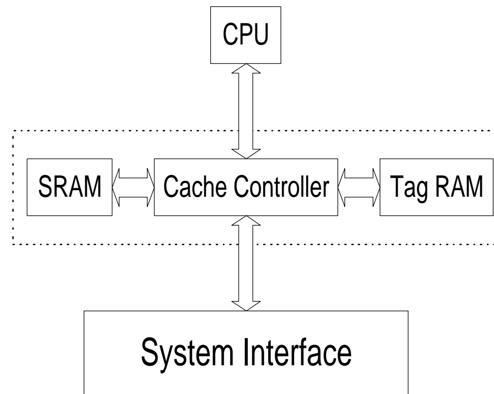

Figure 2-3 Look Through Cache

Figure 2-3 shows a simple diagram of cache architecture. Again, main memory is located opposite the system interface. The discerning feature of this cache unit is that it sits between the processor and main memory. It is important to notice that cache sees the processors bus cycle before allowing it to pass on to the system bus.

2.2.2.1 Look Through Read Cycle Example

When the processor starts a memory access, the cache checks to see if that address is a cache hit.

HIT:

The cache responds to the processor's request without starting an access to main memory.

MISS:

The cache passes the bus cycle onto the system bus. Main memory then responds to the processors request. Cache snarfs the data so that next time the processor requests this data, it will be a cache hit.

This architecture allows the processor to run out of cache while another bus master is accessing main memory, since the processor is isolated from the rest of the system. However, this cache architecture is more complex because it must be able to control accesses to the rest of the system. The increase in complexity increases the cost. Another down side is that memory accesses on cache misses are slower because main memory is not accessed until after the cache is checked. This is not an issue if the cache has a high hit rate and their are other bus masters.

2.2.3 Write Policy:

A write policy determines how the cache deals with a write cycle. The two common write policies are Write-Back and Write-Through.

In Write-Back policy, the cache acts like a buffer. That is, when the processor starts a write cycle the cache receives the data and terminates the cycle. The cache then writes the data back to main memory when the system bus is available. This method provides the greatest performance by allowing the processor to continue its tasks while main memory is updated at a later time. However, controlling writes to main memory increase the cache's complexity and cost.

The second method is the Write-Through policy. As the name implies, the processor writes through the cache to main memory. The cache may update its contents, however the write cycle does not end until the data is stored into main memory. This method is less complex and

therefore less expensive to implement. The performance with a Write-Through policy is lower since the processor must wait for main memory to accept the data.

2.3 Cache Components

The cache sub-system can be divided into three functional blocks: SRAM, Tag RAM, and the Cache Controller. In actual designs, these blocks may be implemented by multiple chips or all may be combined into a single chip.

2.3.1 SRAM

Static Random Access Memory (SRAM) is the memory block which holds the data. The size of the SRAM determines the size of the cache.

2.3.2 Tag RAM

Tag RAM (TRAM) is a small piece of SRAM that stores the addresses of the data that is stored in the SRAM.

2.3.3 Cache Controller

The cache controller is the brains behind the cache. Its responsibilities include: performing the snoops and snarfs, updating the SRAM and TRAM and implementing the write policy. The cache controller is also responsible for determining if memory request is cacheable[2] and if a request is a cache hit or miss.

2.4 Cache Organization

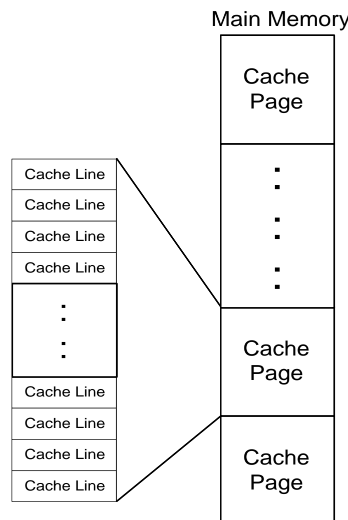

Figure 2-4 Cache Page

In order to fully understand how caches can be organized, two terms need to be defined. These terms are *cache page* and *cache line*. Lets start by defining a cache page. Main memory is divided into equal pieces called *cache pages*[3]. The size of a page is dependent on the size of the cache and how the cache is organized. A cache page is broken into smaller pieces, each

[2] It is not desirable to have all memory cacheable. What regions of main memory determined to be non-cacheable are dependent on the design. For example, in a PC platform the video region of main memory is not cacheable.

[3] A cache page is not associated with a memory page in page mode. The word page has several different meaning when referring to a PC architecture.

called a *cache line*. The size of a cache line is determined by both the processor and the cache design. Figure 2-4 shows how main memory can be broken into cache pages and how each cache page is divided into cache lines. We will discuss cache organizations and how to determine the size of a cache page in the following sections.

2.4.1 Fully-Associative

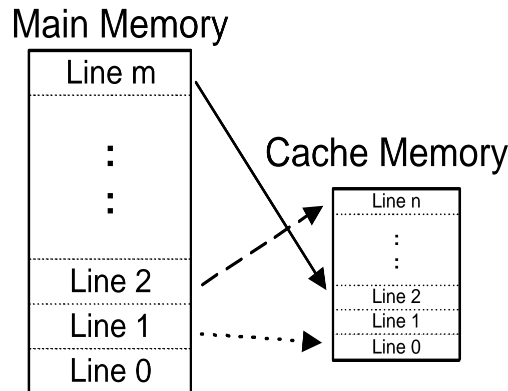

Figure 2-5 Fully-Associative Cache

The first cache organization to be discussed is Fully-Associative cache. Figure 2-5 shows a diagram of a Fully Associative cache. This organizational scheme allows any line in main memory to be stored at any location in the cache. Fully-Associative cache does not use cache pages, only lines. Main memory and cache memory are both divided into lines of equal size. For example Figure 2-5 shows that Line 1 of main memory is stored in Line 0 of cache. However this is not the only possibility, Line 1 could have been stored anywhere within the cache. Any cache line may store any memory line, hence the name, Fully Associative.

A Fully Associative scheme provides the best performance because any memory location can be stored at any cache location. The disadvantage is the complexity of implementing this scheme. The complexity comes from having to determine if the requested data is present in cache. In order to meet the timing requirements, the current address must be compared with all the addresses present in the TRAM. This requires a very large number of comparators that increase the complexity and cost of implementing large caches. Therefore, this type of cache is usually only used for small caches, typically less than 4K.

2.4.2 Direct Map

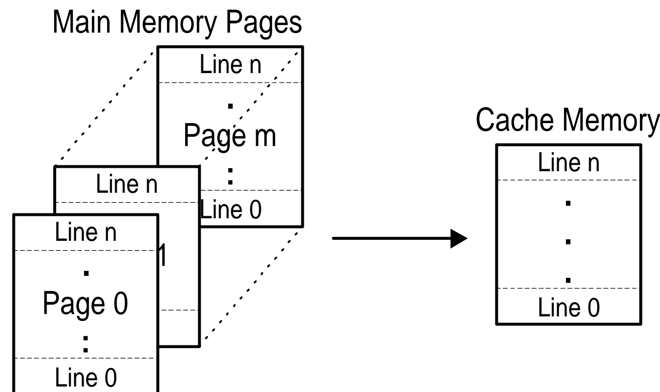

Figure 2-6 Direct Mapped

Direct Mapped cache is also referred to as 1-Way set associative cache. Figure 2-6 shows a diagram of a direct map scheme. In this scheme, main memory is divided into cache pages. The size of each page is equal to the size of the cache. Unlike the fully associative cache, the direct map cache may only store a specific line of memory within the same line of cache. For example, Line 0 of any page in memory must be stored in Line 0 of cache memory. Therefore if Line 0 of Page 0 is stored within the cache and Line 0 of page 1 is requested, then Line 0 of Page 0 will be replaced with Line 0 of Page 1. This scheme directly maps a memory line into an equivalent cache line, hence the name Direct Mapped cache.

A Direct Mapped cache scheme is the least complex of all three caching schemes. Direct Mapped cache only requires that the current requested address be compared with only one cache address. Since this implementation is less complex, it is far less expensive than the other caching schemes. The disadvantage is that Direct Mapped cache is far less flexible making the performance much lower, especially when jumping between cache pages.

2.4.3 Set Associative

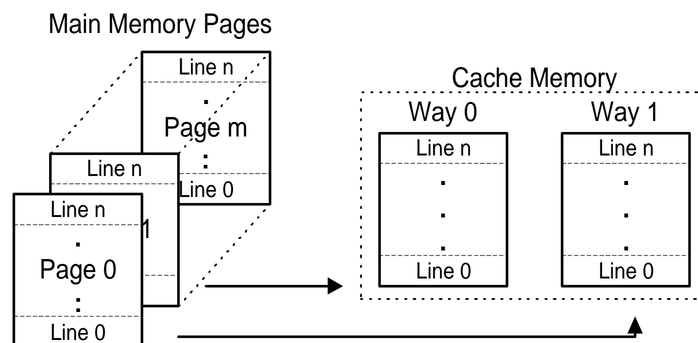

Figure 2-7 2-Way Set Associative

A Set-Associative cache scheme is a combination of Fully-Associative and Direct Mapped caching schemes. A set-associate scheme works by dividing the cache SRAM into equal sections (2 or 4 sections typically) called cache ways. The cache page size is equal to the size of the cache way. Each cache way is treated like a small direct mapped cache. To make the explanation clearer, lets look at a specific example. Figure 2-7 shows a diagram of a 2-Way Set-

Associate cache scheme. In this scheme, two lines of memory may be stored at any time. This helps to reduce the number of times the cache line data is written-over?

This scheme is less complex than a Fully-Associative cache because the number of comparitors is equal to the number of cache ways. A 2-Way Set-Associate cache only requires two comparitors making this scheme less expensive than a fully-associative scheme.

3. The Pentium(R) Processors Cache

This section examines internal cache on the Pentium(R) processor. The purpose of this section is to describe the cache scheme that the Pentium(R) processor uses and to provide an overview of how the Pentium(R) processor maintains cache consistency within a system.

The above section broke cache into neat little categories. However, in actual implementations, cache is often a series of combinations of all the above mentioned categories. The concepts are the same, only the boundaries are different.

Pentium(R) processor cache is implemented differently than the systems shown in the previous examples. The first difference is the cache system is internal to the processor, i.e. integrated into the part Therefore, no external hardware is needed to take advantage of this cache[4] - helping to reduce the overall cost of the system. Another advantage is the speed of memory request responses. For example, a 100MHz Pentium(R) processor has an external bus speed of 66MHz. All external cache must operate at a maximum speed of 66mhz. However, an internal cache operates at 100MHz. Not only does the internal cache respond faster, it also has a wider data interface. An external interface is only 64-bits wide while the internal interface between the cache and processor prefetch buffer is 256-bits wide. Therefore, a huge increase in performance is possible by integrating the cache into the CPU.

A third difference is that the cache is divided into two separate pieces to improve performance - a data cache and a code cache, each at 8K.. This division allows both code and data to readily cross page boundaries without having to overwrite one another.

[4] In order for the Pentium Processor to know if an address is cacheable, KEN# must be asserted. For more detailed information regarding the KEN# signal or other Pentium Processor signals, please refer to the Pentium Processor Family Developer's Manual Volume 1:Pentium® Processors, order number 241428. (way cool!)

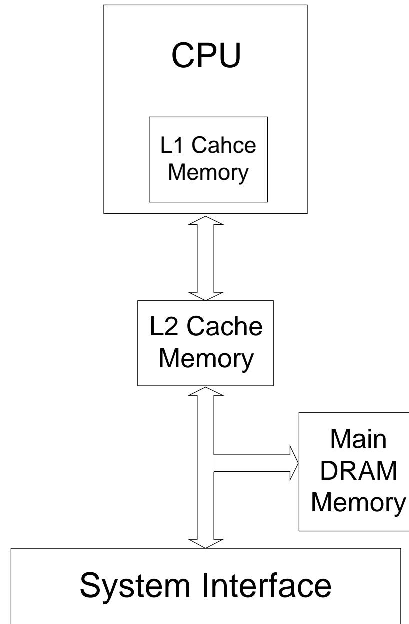

Figure 3-1 Pentium® Processor with L2 cache

When developing a system with a Pentium(R) processor, it is common to add an external cache. External cache is the second cache in a Pentium(R) processor system, therefore it is called a Level 2 (or L2) cache. The internal processor cache is referred to as a Level 1 (or L1) cache. The names L1 and L2 do not depend on where the cache is physically located,(i.e., internal or external). Rather, it depends on what is first accessed by the processor(i.e. L1 cache is accessed before L2 whenever a memory request is generated). Figure 3-1 shows how L1 and L2 caches relate to each other in a Pentium(R) processor system.

3.1 Cache Organization

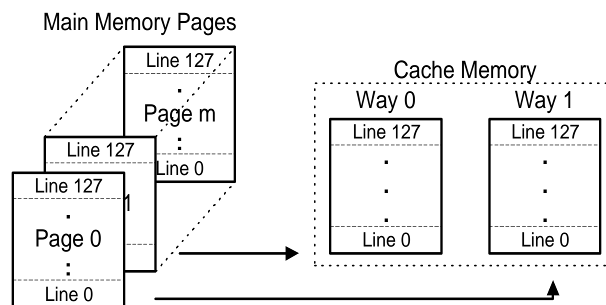

Figure 3-2 Internal Pentium® Processor Cache Scheme

Both caches are 2-way set-associative in structure. The cache line size is 32 bytes, or 256 bits. A cache line is filled by a burst of four reads on the processor's 64-bit data bus. Each cache way contains 128 cache lines. The cache page size is 4K, or 128 lines. Figure 3-2 shows a diagram of the 2-way set-associate scheme with the line numbers filled in.

3.2 Operating Modes

Unlike the cache systems discussed in section 2 (An Overview of Cache), the write policy on the Pentium(R) processor allows the software to control how the cache will function. The bits that control the cache are the CD (cache disable) and NW (not write-through) bits. As the name

suggests, the CD bit allows the user to disable the Pentium(R) processors internal cache. When CD = 1, the cache is disabled, CD = 0 cache is enabled. The NW bit allows the cache to be either write-through (NW = 0) or write-back (NW = 1).

3.3 Cache Consistency

The Pentium(R) processor maintains cache consistency with the MESI[5] protocol. MESI is used to allow the cache to decide if a memory entry should be updated or invalidated. With the Pentium(R) processor, two functions are performed to allow its internal cache to stay consistent, Snoop Cycles and Cache Flushing.

The Pentium(R) processor snoops during memory transactions on the system bus[6]. That is, when another bus master performs a write, the Pentium(R) processor snoops the address. If the Pentium(R) processor contains the data, the processor will schedule a write-back.

Cache flushing is the mechanism by which the Pentium(R) processor clears its cache. A cache flush may result from actions in either hardware or software. During a cache flush, the Pentium(R) processor writes back all modified (or dirty) data. It then invalidates its cache,(i.e., makes all cache lines unavailable). After the Pentium(R) processor finishes its write-backs, it then generates a special bus cycle called the Flush Acknowledge Cycle. This signal allows lower level caches, e.g. L2 caches, to flush their contents as well.

4. Conclusion

Caches are implemented in a variety of ways, though the basic concepts of cache are the same. As shown above in the overview of cache, most systems have the major pieces - SRAM, TRAM, and a controller. However, you can see the details in the Pentium(R) Processors implementation cross several of the initially defined boundaries. It is important to understand that the Pentium(R) Processor uses only one method to implement cache. Their are many other valid way to do the same things, but, they all lead to the same place. Cache is simply a high speed piece of memory that stores a snapshot of main memory which enables the processor higher performance.

---

[5] MESI stands for the four states a cache line can be: Modified (M), Exclusive (E), Shared (S), and Invalid (I). For more detailed information about MESI, please refer to the Developer's Manual

[6] This discussion assumes only Pentium® Processor internal cache and no external cache. The concept still applies if an L2 cache is present.