The Memory Hierarchy

Review of Basics

Clocks

A clock is a continuously running signal that alternates between two values at a fixed frequency. A **clock cycle** is the period of the wave form that the clock generates, i.e., the length of a clock cycle is the amount of time from the start of a high value to the start of the next high value, so it is measured in **time units**. The **frequency** is the inverse of the period; it is the number of cycles per unit of time. Frequency is usually measured in **Hertz**. One Hertz, abbreviated ***Hz***, equals one cycle per second. For example, 100 Hz means 100 cycles per second. When it comes to frequencies, the prefixes kilo, mega, giga, and so on are powers of ten, not two, so one megaHertz is one million Hertz, or 1,000,000 cycles per second. The length of a clock cycle whose frequency is 10 megaHz is $1/10^7 = 10^{-7}$ seconds. Figure 1 illustrates the clock cycle.

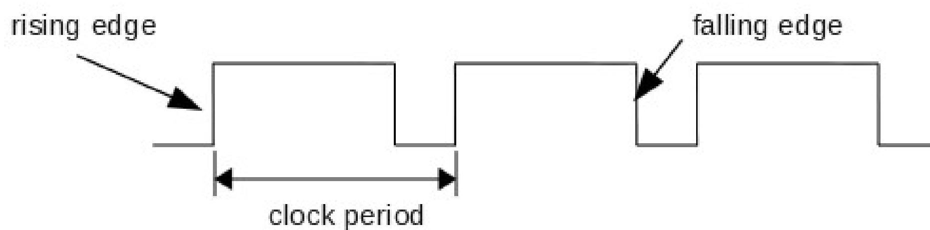


Figure 1: Rising and falling edges of the clock cycle.

In actuality, the switch between high and low states does not take zero time; there is a small amount of time needed to make these transitions, as shown in Figure 2, which is a more accurate depiction of a clock signal's wave form..

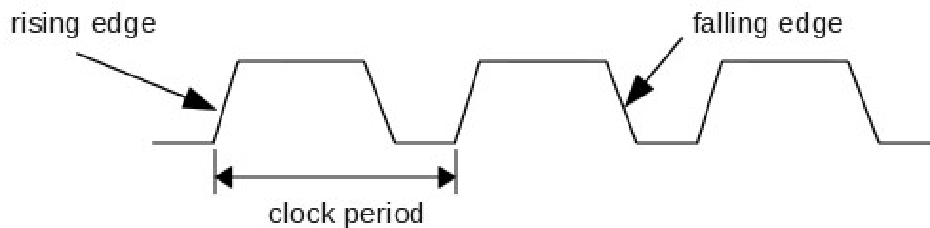


Figure 2: More accurate representation of rising and falling clock edges

Clocks are used to synchronize changes of state in sequential logic circuits. A sequential logic circuit is a combinational circuit with one or more elements that retain state (e.g.., flip-flops – see below.) The state of a sequential logic circuit can be changed either when the clock line is in a high state or when the clock line changes state. If the state changes when the clock line changes state, it is called an **edge-triggered** circuit. If it changes when the clock line is in a high state, it is called a **level triggered** circuit. Edge triggering is efficient because it confines the state element's changes to such a small window of time around an edge transition, that it can be considered to be instantaneous.

Storage Elements: Latches and Flip-flops

The storage elements usually found in modern computers are **flip-flops** and **latches**. The difference between flip-flops and latches is that, in a clocked latch, the state is changed whenever the appropriate inputs change and the clock is asserted, whereas in a flip-flop the state is changed only on a clock edge.

The simplest storage element is an unclocked **S-R latch (set-reset latch)** built from 2 *NOR* gates.

- Used as a building block for more complex memory elements such as D-latches and flip-flops.
- Does not require a clock signal for state update
- The outputs Q and Q' represent the stored state and its complement, respectively.
- If R is asserted, Q will be deasserted (reset), and if S is asserted, Q will be asserted (set).

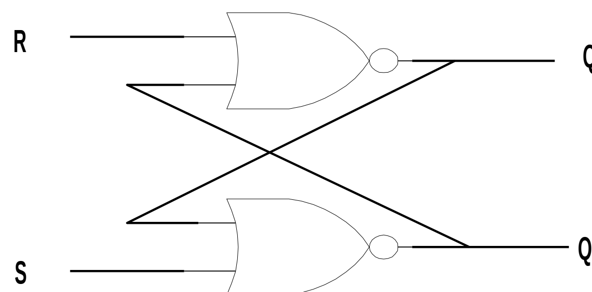


Figure 3 S-R Latch

- If both R and S are de-asserted, the state of the latch is whatever it was before these inputs are de-asserted.

One problem with a simple S-R latch is that it has an indeterminate state, when both the S and R input are set. This is overcome in a **D-latch**, shown in Figure 4. In a D-latch there are 2 inputs: a data signal D and a clock signal, and 2 outputs: Q, the internal state, and Q', its complement. The D input is wired directly to the S input, and D complement is wired to the R input. In addition, an external clock signal C is wired into the latch, making it a **clocked latch**. In a clocked latch, the state changes when the input changes while the clock is asserted (level-triggered).

When the clock C is asserted, Q and Q' are the values of the input and its complement and the latch is OPEN. When the clock is de-asserted, Q and Q' are the values that were stored when it was last open, and it is CLOSED.

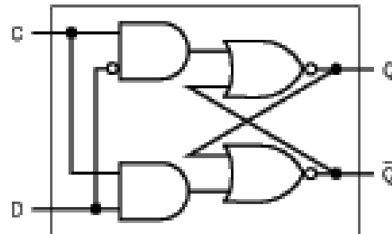


Figure 4: Clocked D latch

In a **flip-flop**, the state changes on a clock edge. A D flip-flop has a data input D and a clock input C. When the clock edge rises or falls, the flip-flop outputs D on Q and D' on Q'. A flip-flop requires that the D value be valid for a period of time before and after the clock edge. The minimum amount of time for which it must be valid before the clock edge is called the **setup time**, and the minimum amount of time for which it must be valid after the clock edge is called the **hold time**. Figure 5 illustrates this. A flip-flop can use either the rising or the falling clock edge to trigger the change in the output. Regardless of which it uses, the input signal must be valid for the sum of the setup hold times.

The same is true for clocked latches; a clocked latch also has setup and hold time requirements, and these are defined similarly, except that because a latch is level-triggered and the setup and hold times are defined in terms of when the clock input is in the high, or enabled, state.

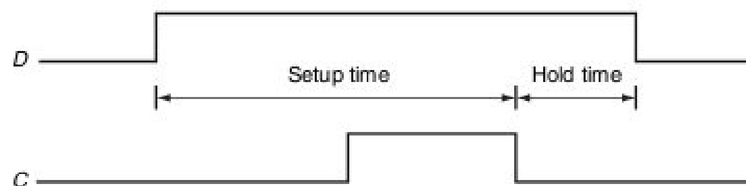


Figure 5: Setup and hold times for a D-flip-flop with a falling-edge trigger.

Flip-flops are used to build registers. Because the textbook that we use in the course uses edge-triggered methodology, it always uses flip-flops for state elements, and so will we.

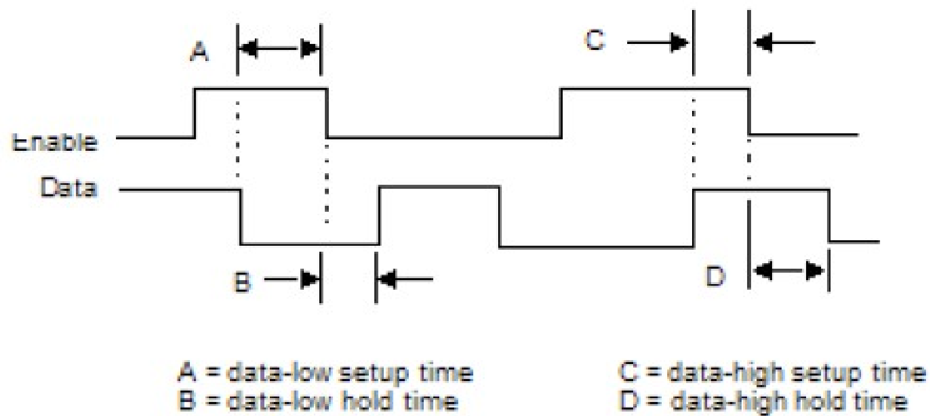


Figure 6: Setup and hold times for a clocked D-latch (from notes by Puneet Gupta)

Decoders and Multiplexers

A **decoder** is a logic circuit that has an *n*-bit input and 2^n outputs. For each input combination exactly one output line is asserted. A **multiplexer** is a logic circuit that has *n* data inputs and $log_2\, n$ selector inputs. The selector input lines select exactly one of the input lines, which is output. If multiplexer were built purely out of 1-input and 2-input logic gates, the number needed would increase exponentially as a function of the number of selector inputs. There are more efficient methods of building multiplexers, but in general, they cannot be made too large because of fan-in and pin limitations.

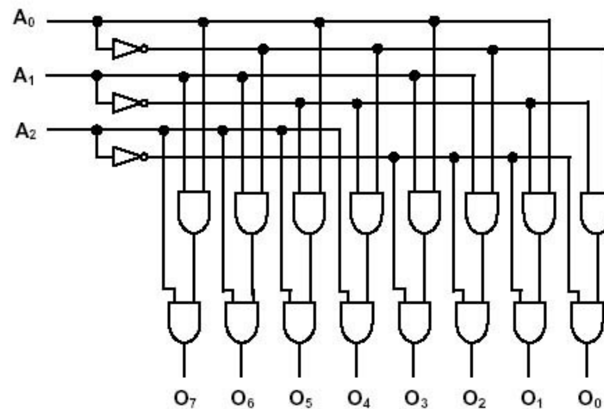


Figure 7: A 1-of-8 decoder

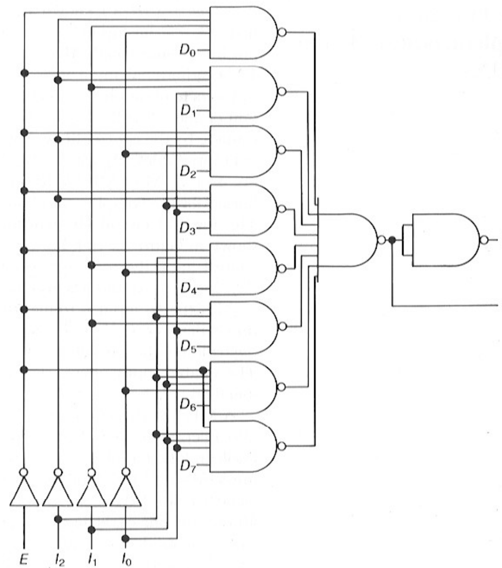


Figure 8: A 4-to-1 multiplexer

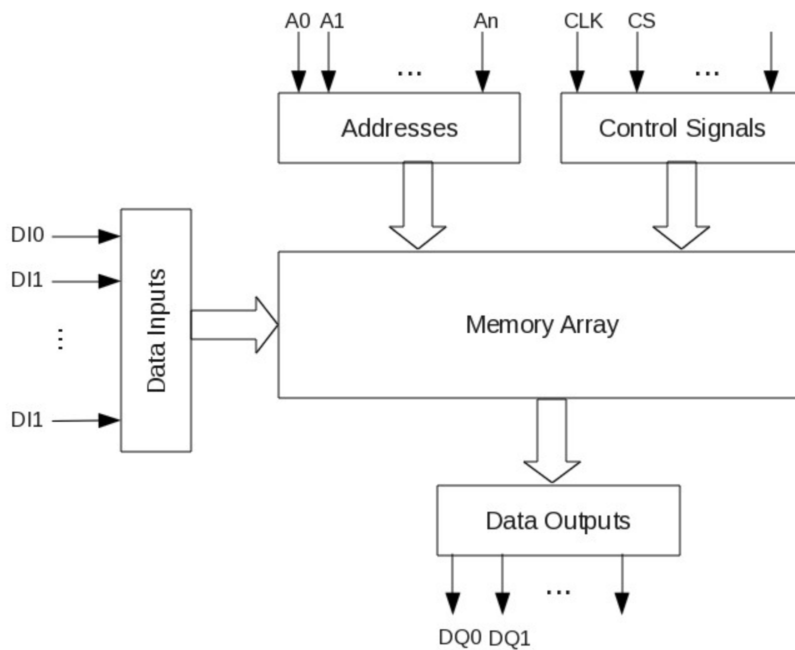


Figure 9: Block diagram of a synchronous SRAM

Registers

A register is an array of flip-flops.

Register Files

A **register file** is a set of registers that can be indexed by a register number, either for reading or for writing. To read a register, the register number is input to the register file, and the read signal is activated. The register number is used as the *select* switch in an output side **multiplexer**. The multiplexer is used to select the out line of the chosen register. Because many machine instructions have two register operands, register files are often designed to accept two register numbers and have two output lines.

Register files are limited in how many registers they can contain because multiplexers become impractical as they get large.

Writing to a register is more complex. The inputs include

- the register number,
- the data to write, and
- a write signal.

The register number is the input to a decoder. Exactly one output of the line of the decoder is asserted, the one corresponding to the register number given as input. The C input to every register is formed from the AND of the write signal and the decoder output. Exactly one register will have its C line asserted, and the data to write will be placed on the D line of every register. There are timing constraints of course -- the data and signals have to be asserted long enough for the write to take place properly.

RAM

RAM is short for **random access memory.** There are two basic types of RAM in common use today, **static** and **dynamic** random access memory, abbreviated **SRAM** and **DRAM** respectively. Both are physically organized as an array of memory cells. Both use a decoder whose input is a set of address lines to select one or more memory cells for the memory operation. Both can store data as long as power is applied. However, even when powered, a DRAM loses data if it is not periodically refreshed, whereas in a SRAM the data can be stored without any kind of extra processing or refreshing. As a result, SRAMs are less complex than DRAMs, and because of this we study them before DRAMs, but only after we have defined how the performance of a memory system in general will be measured.

The performance of a memory is measured in two ways: by its **access time** and **cycle time**. The access time is the time required to select a word and read it. The cycle time is the time required to complete a write operation. These will be explained in more detail below.

SRAM

SRAM is an acronym for **Static Random Access Memory**. The basic architecture of SRAM includes one or more rectangular arrays of memory cells with support circuitry to decode addresses and implement the required read and write operations. Additional support circuitry for

special features such as burst operation or pipelined reads may be present on the memory chip. Figure 9 contains a block diagram of a SRAM.

The inputs to SRAM include:

- address line (log of height bits) also called a **word line**
- chip select signal
- output enable signal
- write enable signal
- data input signal (*w* bits, where *w* = width of the SRAM)

The output is an output line of *w* bits, where *w* is the width of the SRAM.

The individual memory cells in an SRAM are built out of circuits that resemble D-flip-flops. ***A single cell typically requires from four to eight transistors***, depending upon the design (four-transistor cells are not as stable as eight-transistor cells.) The core of the cell is a pair of inverting gates, which store the state of the cell. Figure 10 illustrates how the cross-coupled inverting gates store state. The figure shows that, in addition to the inverting gates, the cell uses a pair of transistors to connect it to the **word** and **bit lines** of the SRAM. The word line is the address line for that particular cell. The bit line is the line for the particular bit of that address. Conceptually the word lines are the rows of the SRAM, and the bit lines are the columns. Each unique word address corresponds to a single row.

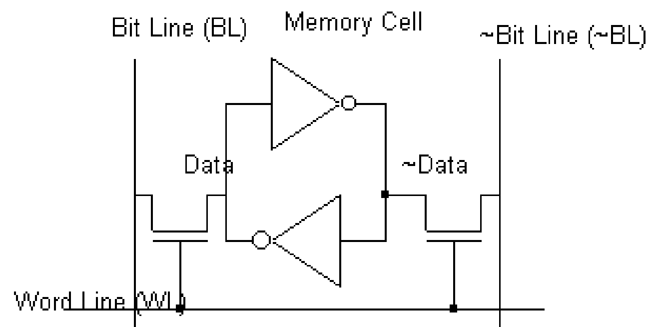


Figure 10: An SRAM cell represented by a pair of inverting gates.

Figure 11 is a circuit diagram for a six-transistor memory cell.

SRAMs may be **synchronous** or **asynchronous**. Asynchronous SRAMs respond to changes at the device's address pins by generating a signal that drives the internal circuitry to perform a read or write as requested. They are not clocked, and are limited in their performance. Synchronous SRAMs (SSRAMs) are faster. They are driven by one or more external clock signals that control the SRAM operations, which allows them to synchronize with the fastest processors. Asynchronous SRAMs come in two flavors, fast and slow. The fast variety has access times under 25 ns, whereas the slow ones have access times greater than 45 ns. In

contrast, SSRAMs can have access times *under 1 ns*. As of this writing, there are many different types of SSRAM, including:

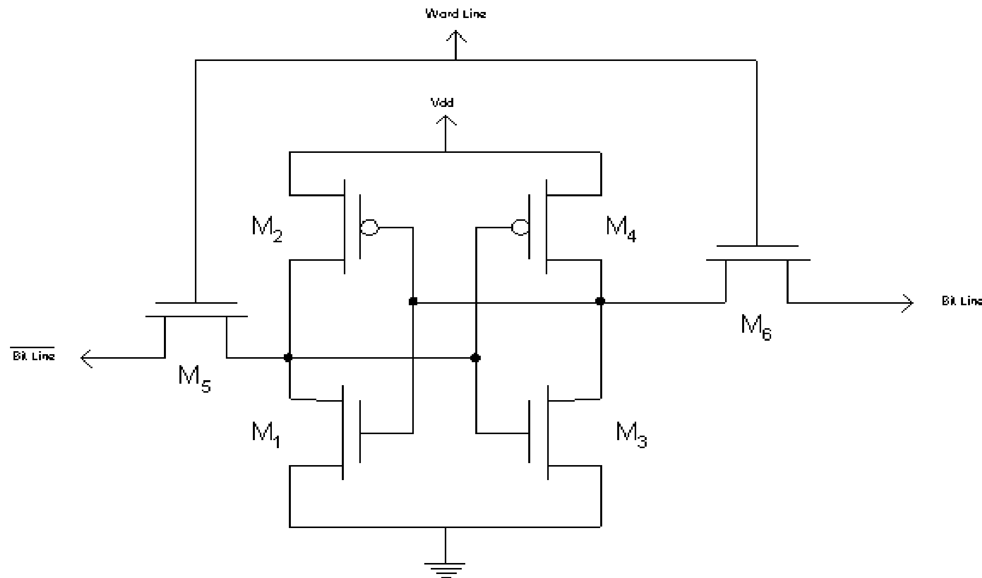


Figure 11: A six-transistor SRAM cell.

- Single Data Rate SRAM
  - Pipelined vs. Flowthrough SRAMs
  - Burst SRAMs
  - Network SRAMs - NoBL™/ZBT™ SRAMs
- Double Data Rate SRAMs
  – Standard DDR SRAMs
  – QDR™ SRAMs
- NetRAM™

Storage capacities for a single SRAM chip have reached 72 Mbits. Synchronous SRAM tends to have greater storage capacity than asynchronous SRAM. SRAM chips are specified by their height *h* and width *w*, e.g., a 256K x 1 SRAM has height 256K and width 1. This means it has 256K addresses, each 1 bit wide. Common shapes are x1, x4, x8, x16, x18, and x36. Figure 12 shows a 4 by 2 SRAM module, with a single 2-to-4 decoder.

Decoding is usually two-dimensional for better performance. For example, if the height is 1M (2^{20}), a single decoder would be 20 x 2^{20} and require 2^{20} 20-input AND-gates. If instead, we break the address into a 10-bit upper and a 10-bit lower part, then we can use two 10 x 1024 decoders, requiring only 2048 AND-gates instead of more than one million.

Another design for smaller memories uses a combination of a decoder and multiplexers. Figure 14 illustrates this idea. The high order bits are input to a decoder, which selects a row that is enabled in all of the SRAM modules. The low-order bits are the input signal to a series of multiplexers, each of which selects a single bit from each array.

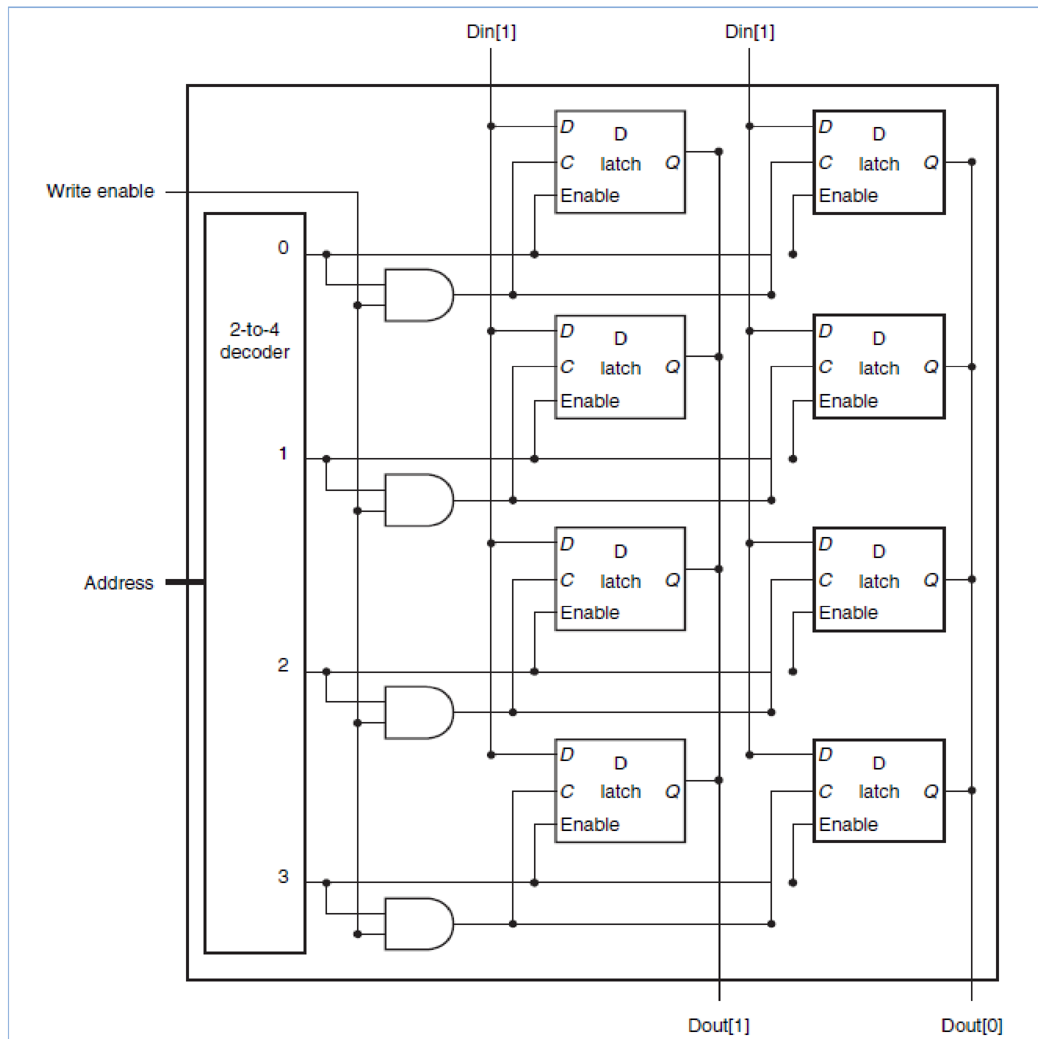


Figure 12: A 4 by 2 SRAM module

It is impractical to use a multiplexer exclusively to select the output of an SRAM cell. It would be on the order of 64K-to-1 or worse. The technology to build SRAMs uses a combination of decoders and multiplexers, and is based on tri-state buffers.

Tri-state buffers can be connected to form an efficient multiplexer. A tri-state buffer is a buffer with a data input, an output-enable input, and a single output. If the output-enable is asserted, the value of the data input is placed on the output line. If the output-enable is de-asserted, the output line is in a high impedance state. This means, in essence, that the output line is neither low nor high voltage, but is neutral voltage, and that another tri-state buffer can put a value on the line.

In practice many SRAMs are built with three-state buffers incorporated into the flip-flops themselves and these share an output line.

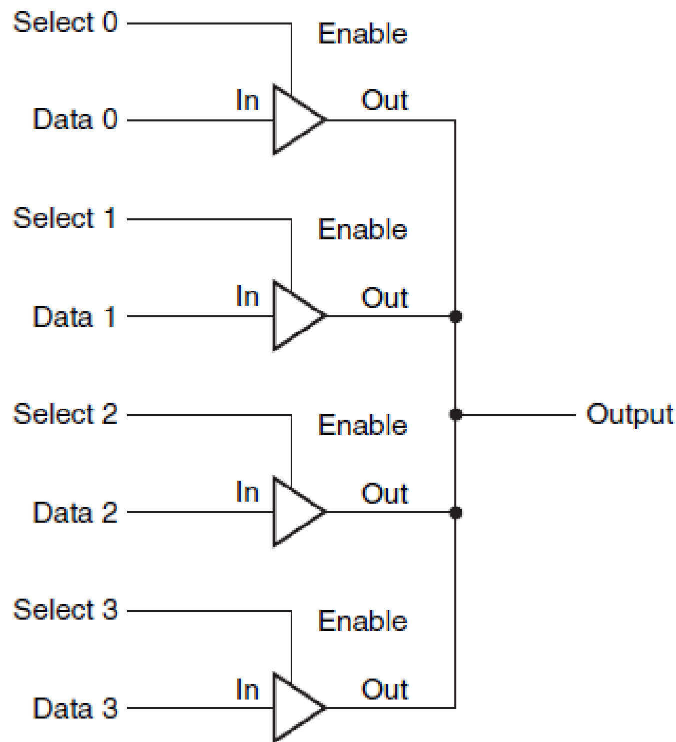


Figure 13: Four tri-state buffers forming a 4 to 1 multiplexor

A 4M by 8 SRAM can be organized into 4096 rows with 1024 x 8 bits per row. Therefore, eight 4K x 1024 SRAM chips can be used as shown in Figure 14.

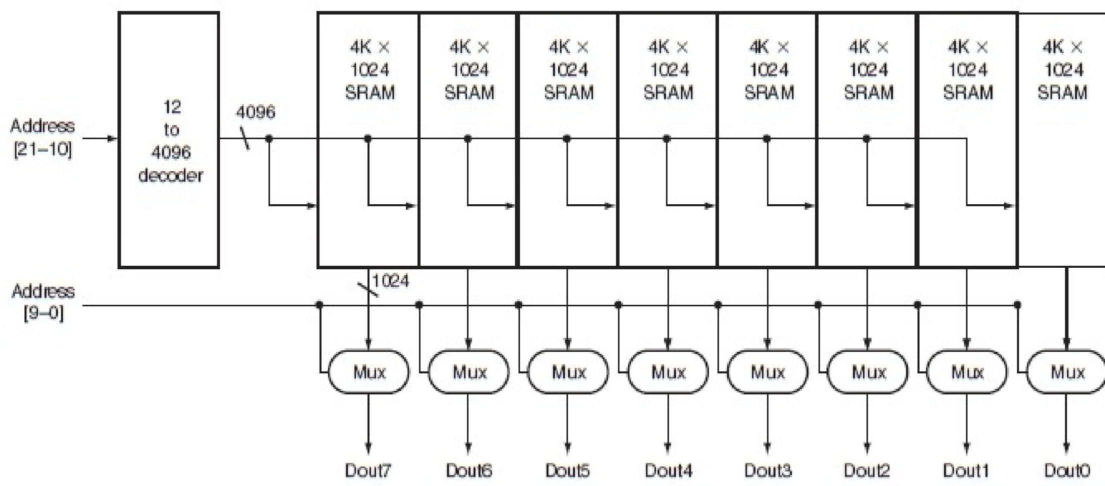


Figure 14: Eight 4096 by 1024 SRAM chips forming a 4MB memory

DRAM

A **DRAM** (Dynamic Random Access Memory) stores a cell's state in a capacitor rather than in a set of inverting gates, and it changes the state with a transistor. See Figure 15. This uses less than one-fourth to one-sixth of the space used by a SRAM with equal capacity, since each cell uses a single transistor and a single capacitor. However, because the capacitor cannot hold the state indefinitely, it must be refreshed periodically, hence the term "**dynamic**". Refreshing can be

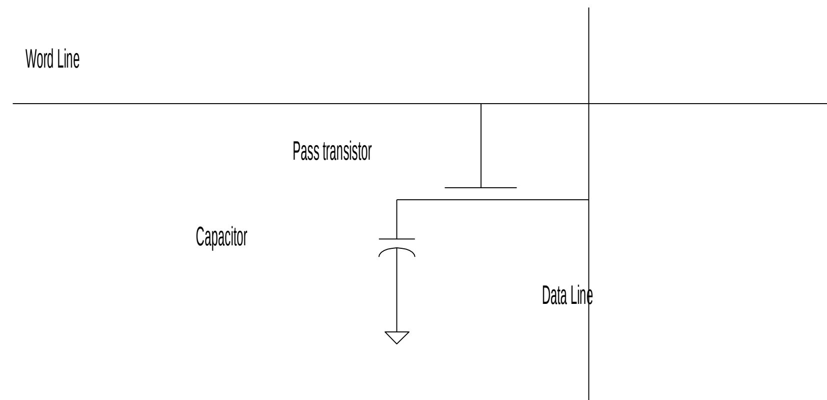


Figure 15: A DRAM cell

done by a separate controller, and can use 1% to 2% of the active memory cycles.

To write data into a DRAM cell, a voltage signal is placed on the data line and a signal is applied to the address line. This switches on the transistor, which allows a high voltage to charge the capacitor. If there is a 0 signal, the capacitor receives no charge. To read the cell the address line is activated, and any charge in the capacitor is transferred onto the data line. This is a **destructive read**. Therefore, the data read out must be amplified and written back to the cell. This rewrite is often combined with the periodic refresh cycle that is necessary in DRAM chips.

DRAM cells store a very small charge, saving on power consumption. The read out requires that the data line be charged to a voltage about halfway between the voltages representing 0 and 1. The small change in voltage is detected on the data line.

DRAMs use a two level decoder. The address is split into a row and a column, and the row is sent followed by the column. The row number is usually formed from the high order bits, and the column, from the low order bits. The address path is only wide enough for one of these.

The row address is placed on the address line and the **Row Access Strobe** (RAS) is sent to the DRAM to indicate that the address is a row address. The row decoder decodes it and a single row is activated. All of the bits in that row are sent to the column latches. The column address is placed on the address line and the **Column Access Strobe** (CAS) is activated. The bits of the individual columns are chosen by the output side multiplexor and placed on the Data Out line.

The two-level access and the more complex circuitry make DRAMs about 5 to 10 times slower than SRAMs, although their cost is much less. DRAMs are used for main memories; SRAMs for caches[1].

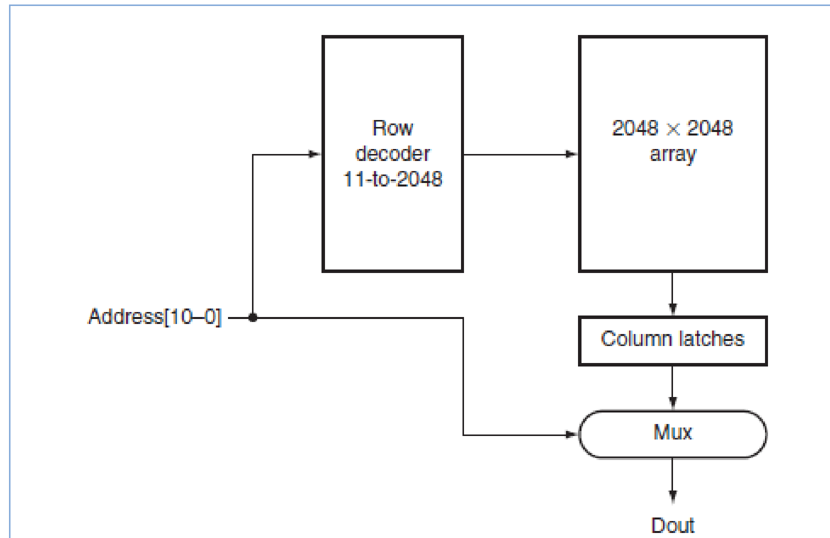


Figure 16: DRAM module

Synchronous DRAM

DRAM was originally an asynchronous type of RAM. **Synchronous DRAM** (SDRAM), like synchronous SRAM, uses an external clock signal to respond to its input signals. This makes it possible to synchronize with the bus and therefore, to improve its performance. Basically, it allows for an internal pipelined type of operation: after an initial setup, a sequence of addresses can be accessed partly in parallel.

In asynchronous DRAM, if a sequence of consecutive rows needs to be transferred to or from the memory, each address is decoded separately, one after the other. In SDRAM, a single address and a burst length are supplied to the memory. Additional circuitry within the SDRAM allows one row to be latched while the next row is accessed. The external clock signal is used to coordinate the transfer of successive rows. This obviates the need to decode multiple addresses, speeding up the transfer.

Memory Hierachy

Main Objectives of a Memory Hierarchy:

1. To minimize execution time of executing programs
2. To maximize the throughput of the computer
3. To minimize response time

subject to the constraint that high-speed memory is limited in size.

---

[1] SRAM is also used in devices such as cellphones and cameras as primary memory.

Two Principles of Locality

Temporal Locality: If an item is referenced, it will tend to be referenced again in the near future.

Spatial Locality: If an item is referenced, items whose addresses are close will tend to be referenced soon.

Empirical and theoretical justification:

- programs tend to stay in loops, so instructions and their data are repeatedly accessed (temporal)
- instructions tend to be executed sequentially (spatial)
- data tends to be accessed sequentially, as in array accesses (spatial)

The ***Memory Hierarchy*** is based on the idea that the faster the memory, the more costly to build and therefore the smaller in capacity, and conversely, the larger the memory, the slower to access and less costly. In 2014, the memory hierarchy consists of registers at the very top, followed by up to three levels of cache, then primary memory (built out of DRAM), then nonvolatile memory such as a magnetic disk or solid-state (semiconductor) disk.

Analogy: books in the library.

Technologies used to build parts of the memory hierarchy:

Technology	**Access Time**	**2012 $ per Gbyte**
SRAM semiconductor memory	0.5 - 2.5 ns	$500 - $1000
DRAM semiconductor memory	50 -70 ns	$10 - $20
Flash semiconductor memory	5000 – 50000 ns	$0.75 - $1.00
Magnetic Disk	5-20 million ns	$0.05 – $0.10

Memory Hierarchy Rules:

Regardless of how many levels are in the hierarchy, data is copied between adjacent levels of the hierarchy only. We focus on just two levels at a time.

A **block** is the smallest unit of information that can be present in a two-level hierarchy (called a 2-hierarchy). In other words, regardless of which two levels are the subject, the smallest transferable unit of the two levels is called a block.

A **hit** occurs when data requested by the processor is present in the upper level of this 2-hierarchy. Otherwise it is a **miss**.

The **hit ratio** is the fraction of memory accesses resulting in a hit.

Hit time is the time to access the upper level, including the time to decide if the access is a hit or a miss. (Time to look through books on the desk)

The **miss penalty** is the time required to replace a block in the upper level with the corresponding block in the lower level, plus the time to deliver this block to the processor. (time to search library shelves, find book, and bring it to the desk.)

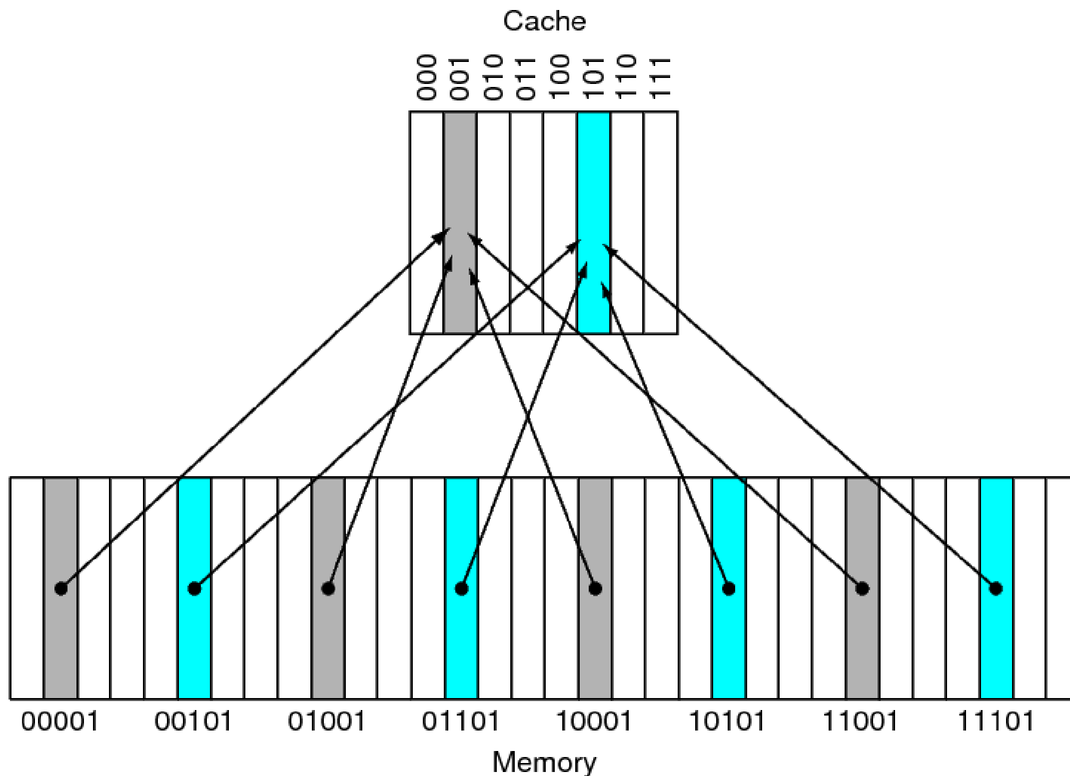
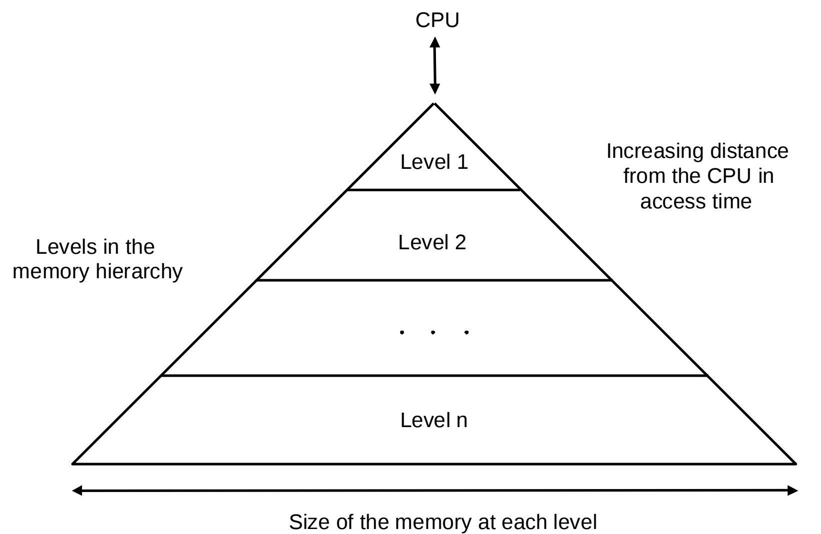


Figure 17. A direct-mapped cache with 8 blocks for a lower level with 32 blocks. (from Computer Organization and Design, 4th edition. Patterson and Hennessey)



Performance is affected by the hit ratio, the time that it takes to access the upper level, and the miss penalty. The miss penalty dominates these times.

A trade-off to keep in mind: *as the size of the upper level increases, although the hit ratio increases, the hit time also increases*.

CACHE BASICS

Cache: a safe place to hide things.

Cache has a specific and a general meaning.

Cache can mean the specific level of the hierarchy between the CPU and main memory.

Cache can also mean more generally, any store interposed between two levels of a memory hierarchy to take advantage of locality of access.

A Simple Cache

A **direct mapped cache** is one in which each word in memory is mapped to a unique cache location.

(There will be many words that map to the same location, but no word maps to two different locations. Thus, this is a **many-to-one** mapping.)

Simplest direct mapping is

cache_address = block_address % number_of_cache_blocks_in_cache

Example

Suppose the cache has 8 blocks, each containing one word (assume that words are the smallest addressable memory units.)

Then the low-order 3 bits of the block address are the index into the cache, because there are $8=2^3$ blocks. All memory words with the same low order 3 bits in their map to the same cache block:

0, 8, 16, 24, → 0

1, 9, 17, 25, → 1

2, 10,18,26, → 2

…

7, 15, 23, 31, → 7

A **tag** is attached to each cache block – the tag contains the upper portion of the block address, i.e., the portion that was not used to choose the cache location for the block.

tag = block_address / number_of_cache_blocks_in_cache

A **valid bit** is needed to tell whether cache data is meaningful or not, since the cache block might not have actual data in it yet.

Each cache entry contains:

- a valid bit
- tag bits
- data bits

Example

(This uses Figure 17 to illustrate how the cache is used and the logic needed to determine whether a hit occurred and how to retrieve data in case of a hit.)

Assume the CPU requests words at addresses 10110, 11010, 10110, 11010, 10000, 00011, 10000, 10010, and 01010 and that the cache described above is used. It has eight one-word blocks and word *w* is placed in cache block *w % 8*.

The sequence of accesses, together with whether they hit or miss and their locations in the cache are as follows:

```
10110 => 110  (miss)     new block
10010 => 010  (miss)     new block
10110 => 110  (hit)      loaded in step 1
11010 => 010  (miss)     replaced in step 3
10000 => 000  (miss)     new block
00011 => 011  (miss)     new block
10000 => 000  (hit)      loaded in step 5
00011 => 011  (hit)      loaded in step 6
10000 => 000  (hit)      loaded in step 5
10010 => 010  (miss)     block replaced in step 4
01010 => 010  (miss)     block replaced in step 10
```

These last two misses result in replacement because there already was a different word in the cache block to which the word was mapped.

Direct mapping takes advantage of temporal locality to a limited extent: when a reference is made to an address r, r replaces some block already in the same cache location. Temporal locality increases the probability that r will be referenced again soon. Spatial locality weighs in favor of r's being accessed over some distant location that might replace it.

Calculating Cache Size

Suppose a cache has 2^n words, where 1 word = 4 bytes. Suppose memory addresses are 32 bits.

1. The low order 2 bits of an address specify the byte offset within a word.
2. The next n bits specify the cache index.
3. Therefore, the tag field must be 32-(n+2) bits long, because there are this many bits required to specify the unique memory word.
4. The data field is 1 word = 32 bits long
5. The valid bit = 1 bit
6. Therefore, we need (32 +(32 - n - 2) + 1) = 63 – n bits per cache block.
7. Since there are 2^n cache blocks, the total bits in the cache is 2^n * (63 - n)

Spatial Locality and Multiword Cache Blocks

To take advantage of spatial locality, one should store adjacent memory words together in the same cache block. This takes advantage of the high likelihood that adjacent words will be referenced in the near future. This concept leads to the idea of **multi-word cache blocks**.

A **multi-word cache block** is is a cache block that consists of more than a single word. For example, it might be two words, or four, or eight words long. The number of words is usually a power of two.

Mapping an address to a multiword cache block

When a cache has multi-word blocks, the memory address is decomposed into 4 fields:

tag, block address, block offset, byte offset

The tag is used in the same way as in a cache with single word cache blocks.

The block address is the address of the block. It is computed in one of two ways:

byte address/ number of bytes per block or

word address / number of words per block

The block address is used in the same way as before, to access the index of the block in the cache, i.e., the row of the cache containing that block, so the row of the cache is obtained by using the block address % number of blocks in the cache.

The block offset is the position of the word relative to the block. It can also be computed in one of two ways:

(*byte address % number of bytes per block)/ bytes per word*, or

word address % number of words per block

The block offset is used by the multiplexer when selecting the input lines whose data should be put on the output line.

Example

Suppose a cache has 128 blocks and a block size of 4 words = 16 bytes. Suppose we want to map a memory address into the cache whose byte address is 3400. Its block address is floor(3400/16) = 212, so the cache block has index 212%128 = 84. The block offset is (3400%16)/4 = 8/4=2 and the byte offset is 3400 % 4 = 0, i.e., The byte is in the second word in the cache block, in position 0 of that word.

Figure 18 is a schematic diagram of a cache that uses direct mapping with 4-word blocks. The cache has 4096 blocks, for a total of 16*4K bytes, or 64 KB.

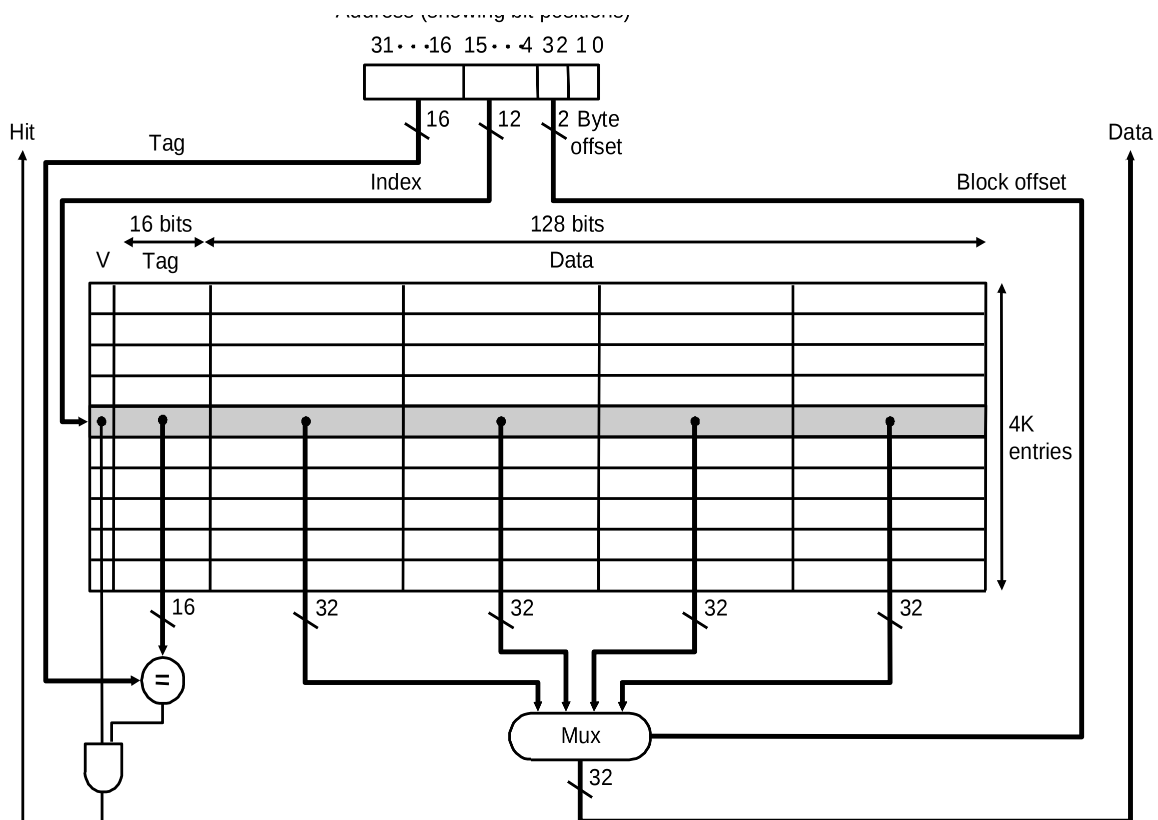


Figure 18: Direct mapped cache with 4-word blocks

Performance Issues with Multiword Blocks

Using multiword cache blocks

- reduces the number of misses and the number of data transfers, and
- reduces cache size, because adjacent words share the same index and tag fields

when the block size is not "too large" as explained below. A good way to see the drop in miss rate is to imagine what happens as a program executes instructions in consecutive words , e.g. at byte addresses 0, 4, 8,12,16,20, 24,…. If 4-word blocks are used, the reference to 0 loads 0, 4, 8, and 12. Then the reference to 16 loads 16, 20, 24, and 28, and so on.

As the number of words per block increases, for a fixed size cache,

1. the number of tag bits decreases,
2. the miss rate decreases because of spatial locality
3. the number of cache blocks in the cache decreases, increasing competition for the blocks, and increasing the rate at which blocks are replaced, and increasing the miss rate,
4. the cost of a miss increases, because although memory latency does not change, block transfer time increases due to increased block size.

These factors are contradictory. Studies show that miss rate is minimized when block size is around 16 to 64 bytes; larger block sizes have the opposite effect.

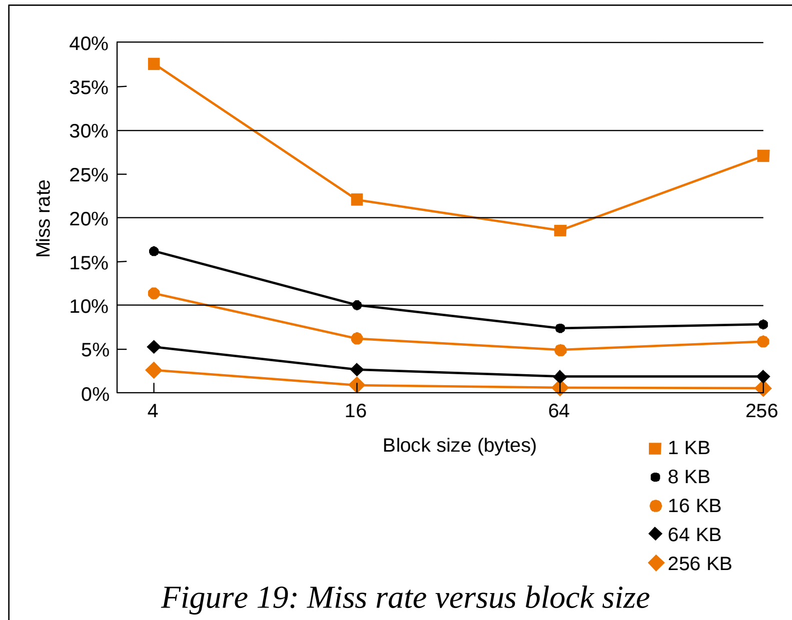


Figure 19: Miss rate versus block size

Handling Cache Misses

What modifications must be made to the processor to handle using a cache?

Handling hits is easy.

Handling misses is harder. There are different kinds of misses:

- **instruction miss**: the instruction is not in cache
- **source data miss**: one of the operands is not in the cache
- **target data miss**: one of the locations to write to is not in the cache

Regardless of the kind of miss, the general approach is to **stall the CPU** while the data or instruction is loaded into cache, and then repeat the cycle that caused the miss. Specifically, the actions include: freezing register contents, fetching missed data or instruction, and restarting at the cycle that caused the miss.

A separate controller handles fetching the data into the cache from memory.

Unlike a pipeline stall, the entire machine is frozen in a cache-miss stall, waiting for the cache to be ready.

Handling Instruction Misses

If the instruction to be loaded into the Instruction register is not in the cache, then the cache must be loaded and the instruction restarted. Since the program counter (PC) is incremented (by 4) before the miss is discovered, the first step is to decrement the PC.

1. Send the PC causing the miss (which is the current PC - 4) to the memory.
2. Instruct memory to perform the read and wait for it to complete read.
3. Write the instruction into the cache entry, putting the read data into the data portion, the upper bits of the address into the tag field, and setting the valid bit to true.
4. Restart the instruction at the first step, which re-fetches the instruction.

Handling Data Misses

This is almost the same as an instruction miss. If an operand is missed,

1. Send the missed address to the memory.
2. Instruct the memory to perform the read and wait for it to complete the read.
3. Write the cache entry the same way as above.
4. Continue the instruction from the point at which the miss occurred.

An alternative to this simple approach is to use a **stall-on-use** policy: the CPU does not stall on a data miss until the data is actually needed as an operand; usually this occurs very soon thereafter and so there is not much benefit.

Handling Cache Writes

When the processor would ordinarily need to write data to memory, in the presence of a cache, the data is written instead to the cache. If the block to which the processor is trying to write is already in the cache, it is called a **write hit**. If the block is not in the cache, it is a **write miss**.

Write Miss in a Single-Word Block Cache

If the block into which a data word must be written is not in the cache, there is no reason to read it first from memory into the cache, since it will be overwritten by the new value, so a write miss is handled by simply writing the data into the cache and updating the tag and valid bits.

Summarizing, the steps are:

1. Index the cache using the index bits of the address.
2. Write the tag bits of the address into the tag, write the data word into the data bits, and set the valid bit.
3. Write the data word to main memory using the entire address (write-through).

The harder problem is when the cache has multi-word blocks. Then the block *does* have to be fetched first, because otherwise the block will become corrupted -- the word to be written is not a part of the block presently in the cache, but of some other block that is not in the cache. This will be explained below.

In either case, writes are more complex than reads, as we now describe.

After a write to the cache, the cache and main memory have different values for the same block. This is called **cache inconsistency**. There are two strategies for preventing cache inconsistency from causing data corruption: **write-through** and **write-back**.

A **write-through** strategy is one in which data is always written to memory and the cache at the same time.

A **write-back** strategy is one in which data is written to the cache block only, and the modified cache block is written to memory only when it is replaced in the cache.

The write-through strategy would not improve performance for writes over a system without a cache because it incurs the cost of a memory access for each write.

Example.

If 13% of instructions are writes, and 10 cycles are needed to process a write to main memory, and the **CPI**[2] without cache misses is 1.2, then the effective CPI would be 1.2 + 10*0.13 = 1.2 + 1.3 = 2.5, a halving of the speed.

To reduce this cost, a **write buffer** is used. Data is written to a write buffer without having to wait for memory to access the location and transfer the data. The data is transferred while the processor continues.

If the processor tries to write to the buffer and the buffer is full, the processor is stalled. Stalls can occur due to **write bursts** from the processor, even if the overall rate of writes is smaller than that which the memory can handle. Usually, multiple buffers are used to reduce the chance of a stall. In the DECStation 3100, a 4-word buffer is used.

Combined Data and Instruction Caches Versus Separate Caches

To compute the effective cache miss rate for a split cache, you need to know the fraction of cache accesses to the instruction and data caches. If $0 <= p <= 1$ is the fraction of accesses to the instruction cache, and the miss rate at the instruction cache is m_I and the miss rate at the data cache is m_D, then the effective miss rate m is

$$m = p*m_I + (1-p)*m_D$$

The advantages of separate instruction and data caches are:

- Twice the cache bandwidth, since instruction and data caches can be checked simultaneously
- Simplified architecture for instruction cache, since it is read-only.

Disadvantages are:

- Lower hit rate than combined cache

[2] Average clock cycles per instruction in a particular program.

Example.

The program `gcc` was run on the DECStation 3100 with its separate caches, and a machine with a combined cache of the same size as the total, 128 KB. The results:

Split cache effective miss rate: 5.4%

Combined cache miss rate: 4.8%

The miss rate for the combined cache is only slightly better, not enough to outweigh the doubled bandwidth of the split cache.

Handling Writes in Multi-word Blocks

Writes are handled differently in multi-word blocks. The processor can only write one word at a time. Suppose that cache holds 4-word blocks, and that memory addresses X and Y both map to cache address C and that at the current time, cache block C contains the memory block starting at Y. Now suppose that the processor issues a write request to X. If it were to write to block C, then C would contain one word of X and 3 words of Y.

Example

Suppose a cache has 256 blocks, each with 4 words, for a total of 1024 words. Suppose that the first block of the cache is filled, i.e., line 0 has 4 words from word addresses 0, 1, 2, and 3 in memory. The processor now issues a write to word 1024 in memory. The block to which this word belongs is block 1024/4 = 256. Block 256 is supposed to be placed into row 256 % 256 = 0. This is therefore a write miss to word 0 of block 256. The entire block has to be replaced otherwise we would write word 1024 into the the block whose other words would be from locations 1, 2, and 3.

To solve this problem when using a write-through cache, the processor writes the data to the cache block simultaneously as the tag field is checked. If tags match, the write access was a hit and the write is written through to memory. If not, it is a write miss. In this case, the block containing X is fetched from memory into the cache and the word is rewritten into the cache (and written through to memory). This would not work with a write-back cache, because the write to X could replace a word that was not yet written back to memory in the block starting at Y.

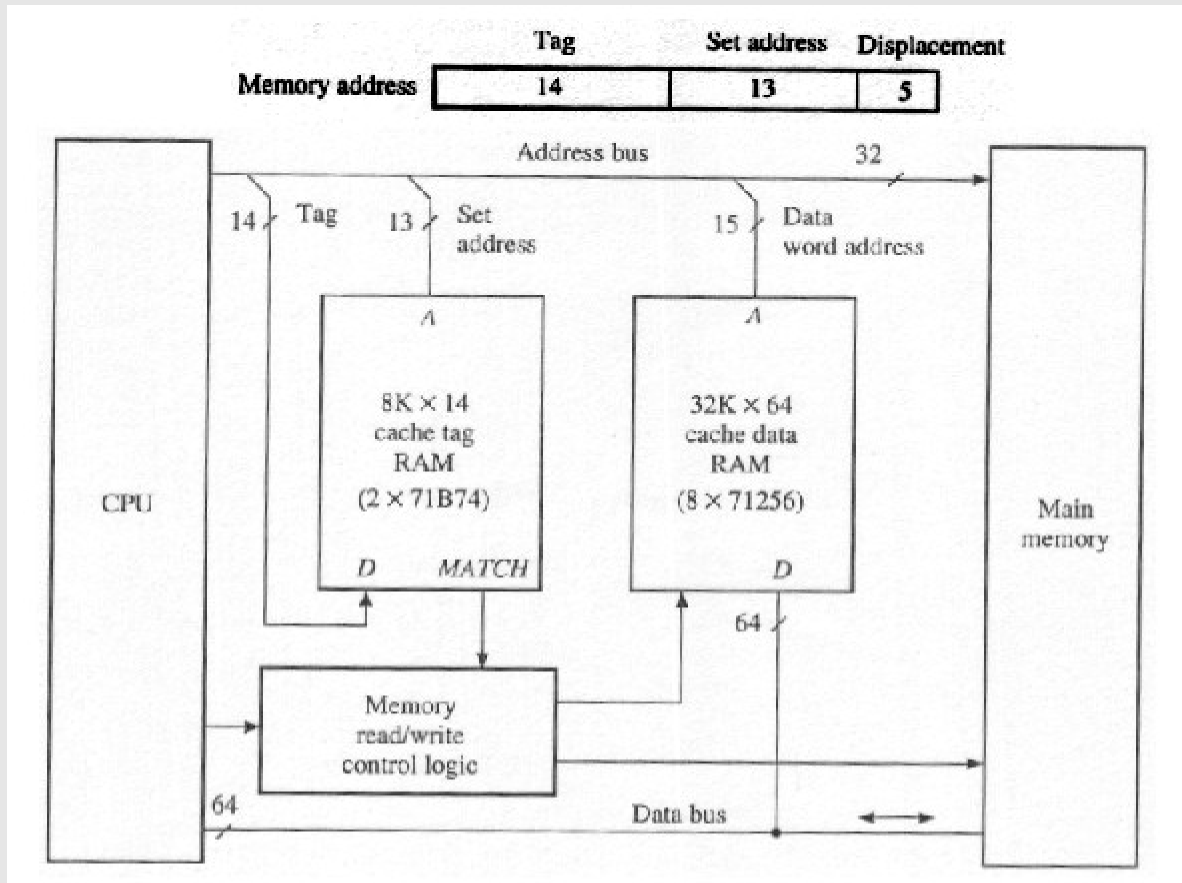
Notice that when blocks contain multiple words, a write miss requires reading from memory, whereas when blocks contain one word, a write miss does not. This is independent of whether write-through or write-back is used.

A Real Multiword Cache from Integrated Device Technology, 1994. (optional)

Below is an illustration of the design of a 256KB direct-mapped cache with 32 byte blocks. On this machine, words are 64-bits, or 8 bytes. Therefore, each cache block consists of 4 eight-byte words. The memory is connected to the CPU and the cache data RAM via a 64 bit bidirectional bus. The CPU uses a 32 bit address line to address memory. Since the cache has 256KB in 32 byte blocks, it has 8K = 2^{13} blocks, hence the 13-bit index (called a set address in the diagram.) Since there are 32 bytes per block, the byte offset requires 5 bits. Since the memory bandwidth is 64 bits = 8 bytes, which is ¼ of a cache block, a cache block access requires a 13 bit block

address plus 2 bits to select which 8 byte word of the block is requested. Therefore, there is a 15-bit path to the cache data RAM.

Notice that the output of the tag RAM is a 1-bit MATCH, which is sent to the memory control.



(from *Computer Architecture and Organization*, J.P. Hayes, McGraw-Hill, 1998)

Designing Memory to Support Caches

The goal is to reduce the miss penalty. Memory access time is

number of clock cycles (S) to send the address +

number of clock cycles (A) to access the DRAM +

number of clock cycles (T) to transfer a word of data

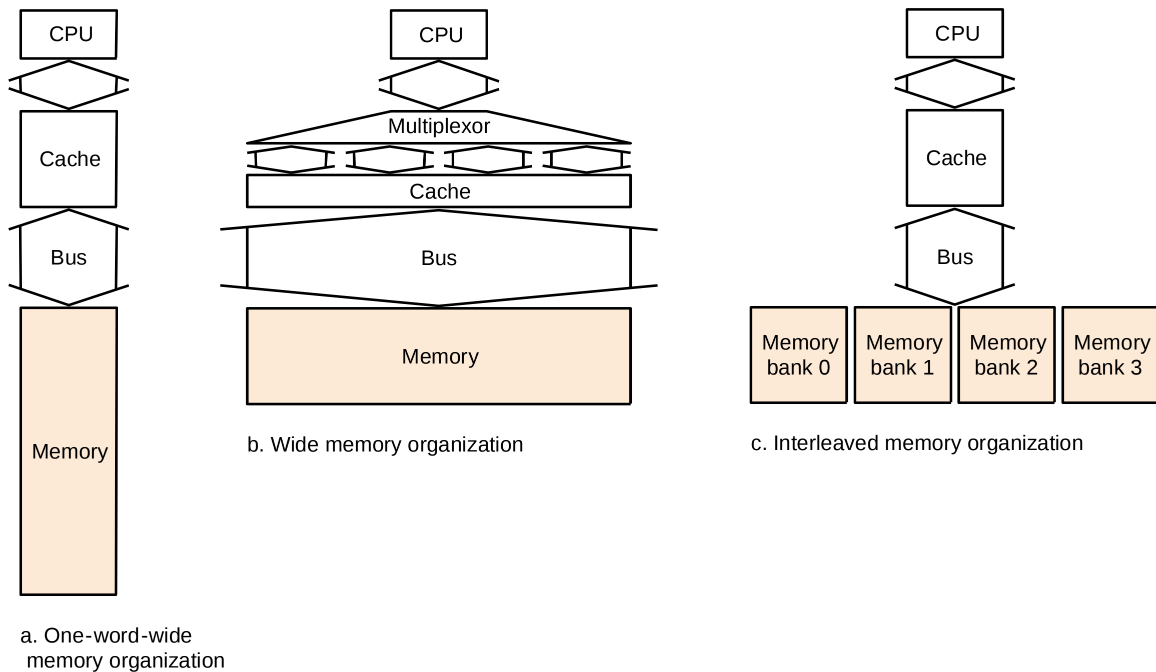


Figure 20: Three different memory organizations

Example.

If S = 1, A = 15, T = 1, then the miss penalty for a 4-word cache block would be

1 + 4*15 + 4*1 = 65 clock cycles.

Since there are 16 bytes in a 4-word block, this is an average of 16/65 = 0.25 bytes per clock cycle.

Memory Organizations

One word wide memory. Sequentially accessed words.

Multi-word wide memory with a multi-word wide bus and cache.

Words are accessed and transferred in parallel, reducing access and transfer times by a factor equal to the width. For example, with S, A, and T as above, the miss penalty for a 4-word block with a 2-word wide bus and memory is

1 + 2*15 + 2*1 = 33 clocks (bandwidth = 16/33 = 0.48 bytes per clock)

and for a 4-word wide bus and memory would be

1 + 15 + 1 = 17 clocks (bandwidth = 16/17 = 0.94 bytes/clock)

The dominant costs are the expense of the bus and CPU control logic and the increased time to multiplex the cache.

Interleaved multi-word wide memory with a one-word wide bus.

Memory banks are **interleaved**. A single access can access all banks, but transfers must be sequential. An address and read or write request can be sent to all banks simultaneously, and

each accessed simultaneously, but the transfers take place one at a time. For example, the miss penalty for a four-word block would be

1 + 15 + 4 *1 = 20 clocks (bandwidth = 16/20 = 0.80 bytes/clock)

Measuring and Improving Cache Performance

Measuring Cache Performance

The first step in improving cache performance is to know exactly what it is you are trying to improve. Another way to put this is that you need to decide on a measure by which you can compare the performance of two caches (or perhaps a cache and the supposedly improved version of this cache.) Ultimately, the gauge of a cache's performance is the amount of time by which it improves the running time of programs executed in the CPU.

For any single program, we can measure this performance by comparing the running time of the program on a CPU with this cache against the running time of the same program on the same CPU but with a "perfect" cache. A perfect cache would always have the referenced block; the processor would never have to access main memory. The running time with a perfect cache would be the total time of the instructions executed in the CPU, without any added cycles due to memory stalls. To make this precise,

Actual CPU Time = (CPU cycles + memory_stall_cycles) amount_of_time_in_one_cycle*

We can assume that cache misses dominate the cost of stalls for simplicity. Then

Memory_stall_cycles = read_stall_cycles + write_stall_cycles

where

*read_stall_cycles = (reads per program) * read miss rate * read miss penalty*
*write_stall_cycles = (writes per program) * write miss rate * write miss penalty + write buffer stalls*

The latter formula is true for a write-through scheme with multiword blocks. In a write-back scheme, because write stalls can still incur a write cost at block replacement time, there is still a penalty due to writes.

Usually the write miss penalty is about the same as a read miss penalty in a write-through scheme, so they can be combined:

*Memory stall cycles = (total memory accesses) * miss rate * miss penalty*

or, equivalently,

*Memory stall cycles = total instructions * (misses per instruction) * miss penalty*

Example

The gcc (gnu C compiler) was subjected to experiments and it was determined that it had, on a particular architecture, an instruction miss rate of 2% and a data miss rate of 4%, and that 36% of all instructions were either loads or stores. If the CPI of a machine with a perfect cache is 2.0 and the miss penalty is 40 clock cycles, how much faster is the machine with the perfect cache?

Solution

Since 36% of instructions make a data access, the memory stall cycles due to data misses is

*data miss cycles = I * 0.36 * 0.04 * 40 clock cycles = 0.576 I clock cycles*

and those due to instruction misses is

*instruction miss cycles = I * 0.02 * 40 clock cycles = 0.80 I clock cycles*

so that the total cycles required for memory stalls is

0.80 + 0.576 = 1.376 I

The CPI with stalls is 2.0 + 1.376 = 3.376, and the ratio

(CPU Time with imperfect cache)/(CPU Time with perfect cache)

*= (I * CPI with imperfect cache) / (I * CPI perfect cache)*

= 3.376/ 2.0 = 1.688

so that the perfect cache is 1.688 times faster.

Improvements to the processor may speed up performance in general, but the impact of memory stalls will reduce the effective speedup. To illustrate this, we can consider two different ways to speed up running time without changing the cache or its design.

Effects of changes to architecture on performance

1. Increasing Processor Speed

Suppose that we speed up the processor without changing the system clock, by a factor of 2, so that the CPI using a perfect cache is 1.0 instead of 2.0 (i.e., it uses half the time as before.) Then the CPI with stalls is

$CPI_{stalls} = 1.0 + 1.376 = 2.376$

so

$CPI_{stalls} / CPI_{perfect} = 2.376/1.0 = 2.376$

as opposed to 1.69 from above. In other words, if this cache were used on a faster processor, it would have an even greater negative impact on overall performance than if it were used on a slower one.

2. Increasing Clock Rate

Suppose that we double the clock speed without changing the processor speed. A change in the clock speed will not affect the response time of memory; the miss penalty will take the same time but will take twice as many cycles, in this case 80 cycles (since it was 40 before.) Therefore, the CPI with stalls is derived as follows:

*instruction miss rate = I * 0.02 * 80 clock cycles = 1.6 I*

*data miss rate = I * 0.36 * 0.04 * 80 clock cycles = 1.152 I*

miss cycles per instruction = 1.6 + 1.152 = 2.752 I

$CPI_{stalls} = 2.0 + 2.752 = 4.752$

and

$CPI_{stalls} / CPI_{perfect} = 4.752/2.0 = 2.376$

If we compare the performance before and after the change, we have to take into account the fact that a clock cycle is smaller after the change, so the units of comparison are different. We doubled the clock speed without speeding up the processor. Therefore, the effective CPI is double what it was before, since each instruction uses twice as many cycles as it did before we doubled the clock speed (i.e., each clock is half the time it was before, so we have to double the CPI to compare it to the CPI with stalls after the change.) If the effective CPI with stalls before was 3.376, then after the change, with this smaller clock cycle, it is 6.752. If we compare the performance before and after the change, we get

CPI_{stalls} *before change* / CPI_{stalls} *after change* $= 6.752 / 4.752 = 1.42$

In other words, although we doubled the clock speed, the actual speed-up is only 1.42, or 42% faster.

These two examples show that there is a limit to the amount of speedup obtained by changing the processor alone. The key to real gain is to reduce the combined cost of high miss rates and miss penalties.

Average Memory Access Time (AMAT)

AMAT = hit time + miss rate * miss penalty

Example

A processor has a 1 ns clock cycle, a miss penalty of 20 clocks, and a miss rate of 0.05 (per instruction.) Cache access time is 1 clock cycle. Assuming read and write penalties are the same,

AMAT = hit time + miss rate * miss penalty

= 1 clock cycle + (0.05 * 20) clock cycles

= 2 clock cycles = 2 ns.

Reducing Miss Rate by Flexible Placement of Blocks

In direct mapped caches, each block can go in exactly one place in the cache. By allowing blocks to be placed into more than one location, we can improve performance.

Set associativity

In an **n-way associative cache**, there are n different cache blocks into which a memory block may be placed. The "n" is the number of equivalent choices. If some of these blocks are free at the time the block is stored into the cache, one of them is chosen using some **placement strategy**. If none are free, a **replacement strategy** is used to decide which of the n blocks to replace. The most common replacement strategies are random replacement and

least-recently-used (LRU) replacement. Each memory block maps to a unique set, but not a unique element within the set.

Notice from the illustrations that as n increases, the number of rows, or sets, decreases, and that, if the cache has 2^k *total blocks, it has* 2^k */n sets.*

Direct-mapped caches are just 1-way set-associative caches – there is just one place to put the block. A direct-mapped cache with 2^k blocks has 2^k sets.

Fully associative caches are the special case in which the entire cache is a single set and all cache blocks are in this one set. The set has 2^k blocks and $n = 2^k$.

Set-associative caches are caches in which $1 < n < 2^k$. for a cache with capacity 2^k. They have a number of sets greater than 1 and less than the number of cache blocks. The mapping for a set-associative cache is

set index = block number % (number of sets in the cache)

Figure 21 illustrates how an 8-block cache can be configured as direct-mapped, 2-way associative, 4-way associative, or fully associative.

The schematic diagram for an n-way set associative cache with n = 4 is illustrated in Figure 22.

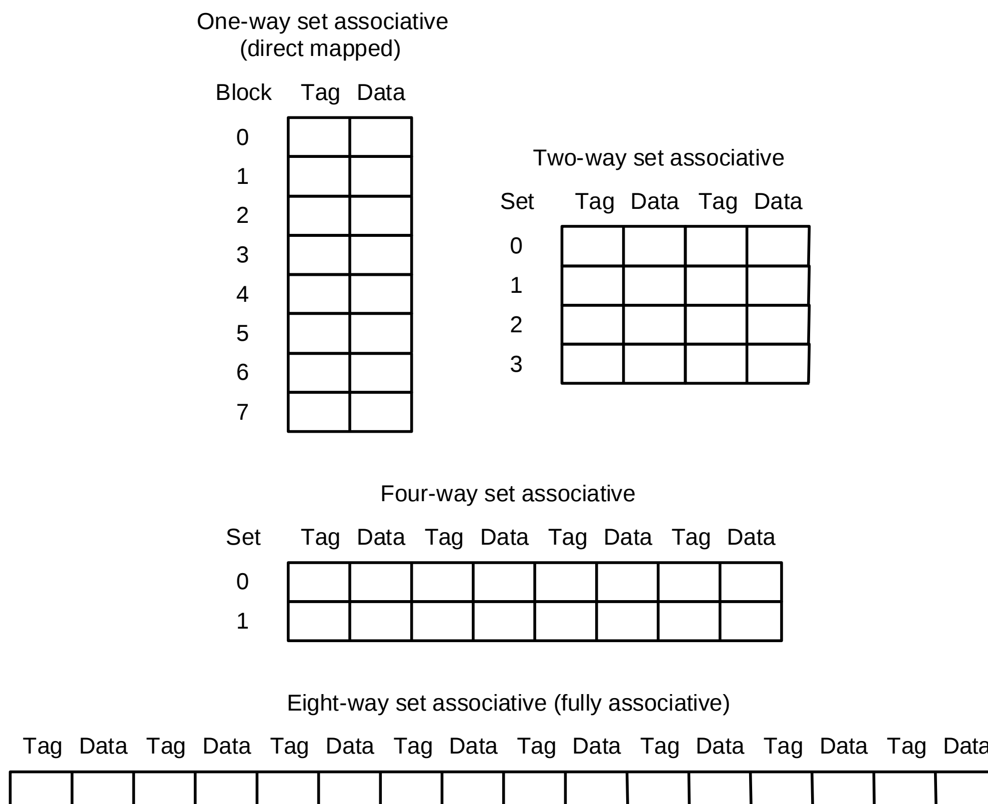


Figure 21: Different degrees of set associative caches

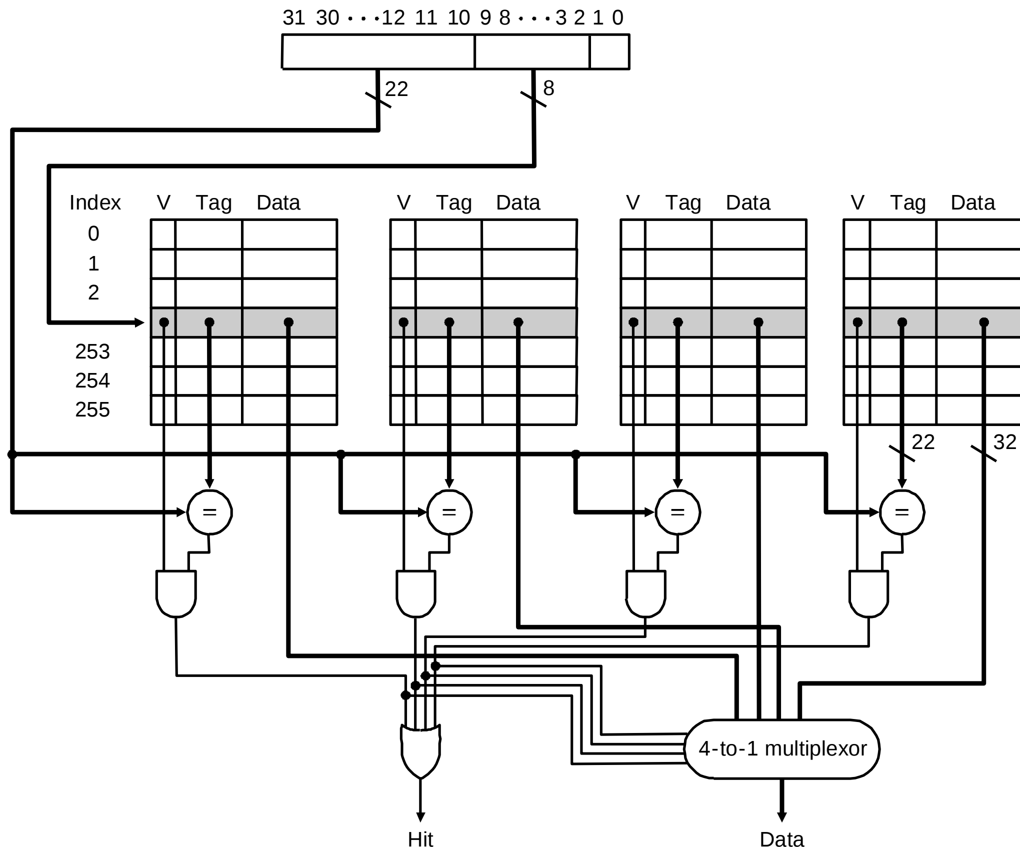


Figure 22: Four -way set associative cache circuitry

In this cache, the index bits select a set. If the memory block is in the set, it is in exactly one block of the set. The tag bits are input to each tag in the row. If the block is present and the data is valid, exactly one of the AND-gates will output a 1, and the set of outputs of the AND gates is used as the input to a multiplexor with a decoded select input, to select which column will be placed on the data out line of the cache. If no block matches, the HIT output is false and the data out line will be invalid. The multiplexor can be eliminated if the comparator outputs an Output enable signal that can drive the data from the matching comparator onto the output lines.

Examples

Assume we have three caches with 4 one-word blocks. One is direct-mapped, the other, 2-way associative, and the last, fully associative. Assume the following sequence of block addresses is requested by the processor: 0, 8, 0, 6, 8.

Direct-Mapped Cache

The mapping of block addresses to cache blocks is given by

set index = block number % 4

Block Address	Cache Block
0	(0 % 4) = 0
6	(6 % 4) = 2
8	(8 % 4) = 0

Hence the sequence 0, 8, 0, 6, 8 maps to 0, 0, 0, 2, 0. The chart below shows the cache after each reference. **Bold** indicates the newly placed block. Each reference caused a miss, five misses in all.

Block Address	Hit or Miss	Contents of Cache Block After Reference			
		0	1	2	3
0	miss	**Mem[0]**			
8	miss	**Mem[8]**			
0	miss	**Mem[0]**			
6	miss	Mem[0]		**Mem[6]**	
8	miss	**Mem[8]**		Mem[6]	

Two-way associative using LRU

There are just two sets, so the mapping is *set index = block number % 2*

Block Address	Cache Set
0	(0 % 2) = 0
6	(6 % 2) = 0
8	(8 % 2) = 0

The cache contents after each reference, using LRU for replacement, and using the smallest numbered available block when more than one is free. There are four misses.

Block Address	Hit or Miss	Contents of Cache Set After Reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	**Mem[0]**			
8	miss	Mem[0]	**Mem[8]**		
0	hit	**Mem[0]**	Mem[8]		
6	miss	Mem[0]	**Mem[6]**		
8	miss	**Mem[8]**	Mem[6]		

Fully associative using LRU

There is one set, so all blocks map to the single set 0.

The cache contents after each reference, using LRU for replacement, and using the smallest numbered available block when more than one is free.

Block Address	Hit or Miss	Contents of Cache Block After Reference			
		set 0	set 0	set 0	set 0
0	miss	**Mem[0]**			
8	miss	Mem[0]	**Mem[8]**		
0	hit	Mem[0]	Mem[8]		
6	miss	Mem[0]	Mem[8]	**Mem[6]**	
8	hit	Mem[0]	Mem[8]	Mem[6]	

There are three misses, which is the least possible, since there are three distinct blacks referenced.

Reduction in Miss Rate

As the associativity increases, the miss rate tends to diminish. Increasing associativity means that blocks can be placed with more flexibility; at the extreme, in a fully associative cache, as long as there is a free block, a block can be placed, and if no blocks are free, the LRU replacement policy will tend to replace the block with the least likelihood of being referenced in the near future, if the programs exhibit a large amount of temporal locality.

Locating a Block in the Cache

In direct mapping, the number of distinct indices is equal to the number of cache entries.

In an n-way set associative cache, the number of distinct entries is equal to this number divided by n.

In a fully associative cache, there is just one index, 0.

The index is

Index of set in cache = Block number % number of sets in the cache

Each doubling of associativity decreases the number of bits in the index by 1 and increases the number of bits in the tag by 1. In a fully associative cache, there is no index; the entire address (excluding the block offset) is compared against the tag field of every block.

This is done in parallel for efficiency. That is why it is called associative. The cells of an associative cache are associative memory cells, which have a match output. An associative memory cell has an exclusive-nor gate that outputs 1 if the data in the cell matches the data on the input line, and 0 if it does not.

Cost of Cache: Number of Comparators and Tag Bits.

The increase in associativity increases the number of comparators and tag bits.

Assume 32 bit addresses, 2^n blocks in the cache, and 4 bytes per block. For a 2^m way associative cache, with 0 <= m <= n, how many tag bits are there and how many comparators?

Ans. There are $2^n / 2^m = 2^{(n-m)}$ distinct sets in the cache. Each set has 2^m members, each of which has its own tag field. So there are $2^{(n-m)} * 2^m = 2^n$ tag fields. The number of bits in each tag field is $32 - (n - m) - 2$, so there are

$$2^n * (30 + m - n) \text{ tag bits}$$

The number of comparators is always 2^m.

Block Replacement Policy

LRU or an approximation to it is the most common. It is hard to implement an exact LRU algorithm, since it would require time stamping each block. Various approximations to LRU have been implemented; we will see how they work later in the chapter.

Multilevel Caches

The larger the cache the smaller the miss rate, but the larger the hit time and cost. The idea of multilevel caches is to take advantage of the smaller hit time and cost of a small cache and the smaller miss rate of the larger one. Using a two level cache makes a significant difference in performance. It reduces the miss penalty without increasing the hit time.

A small primary cache is usually integrated into the processor. A secondary cache, much larger in size, used to be a separate chip on the motherboard but is also now on the same chip as the processor. The larger cache has a smaller miss rate than the primary, and its hit time is smaller than a memory access. Accesses that miss the primary cache but are caught by the secondary cache have a much smaller miss penalty than if the secondary cache were not present.

Example.

Assume

the base CPI = 1.0.

clock rate is 4 GHz.

main memory access time of 100 ns including all miss handling

miss rate per instruction at primary cache = 2%

Assume we add a secondary cache with a 5 ns access time for a hit or a miss, and which is large enough to reduce the miss rate from the processor to main memory to 0.5%. In other words, only 0.5% of memory references miss both caches. What is the increase in speed?

The clock cycle is 1/(4 GHz) = 0.25 ns.

The miss penalty is 100 ns/ (0.25 ns per cycle) = 400 cycles.

The CPI with a 1 level cache alone is

1.0 + 0.02*400 = 1.0 + 8.0 = 9.0 clock cycles

The miss penalty for primary cache accesses that hit the secondary cache is

5 ns / (0.25 ns per cycle) = 20 clock cycles.

The CPI with a 2-level cache is therefore

CPIperfect + cost of misses to primary caught by secondary + misses to both

= 1.0 + (0.02- 0.005)*20 + 0.005*(20 + 400)

= 1.0 + 0.015*20 + 0.005*420

= 1.0 + 0.3 + 2.1

= 3.4 cycles.

The speed increase gained by adding a secondary cache is 9.0/3.4 = 2.6 or 260%.

The secondary cache often uses larger block sizes and higher associativity than the primary cache, to reduce the miss rate. The primary cache has to be small to keep the hit time down and use a smaller block size to increase the number of total blocks available, to decrease miss rates.

Virtual Memory

If you have already had an operating systems course, then you should know what is meant by virtual memory. If so, your idea about virtual memory is about to be put into a different perspective by your understanding about caches and the different kinds of cache designs.

So far we have been looking at two particular levels of the memory hierarchy, the lower level being primary memory and the upper level, the processor cache. Now we drop down the hierarchy one level and let the lower level be the secondary storage device and the upper level be the primary memory.

Viewed this way, memory can be treated like a cache for the secondary storage device. Think of a program as generating secondary storage addresses, some of which have been copied into primary memory. All of the program's storage is on the secondary storage device, and some portion of it is copied into memory. From this observation it follows that the address space of a program does not need to be limited to the size of the physical memory any more than it is limited by the size of the processor cache. Instead, the process's address space can be as large as the secondary storage device's capacity, and the blocks that it currently uses will be memory-resident. The term "*virtual memory*" refers to this method of treating memory. The terms "*virtual address*" and "*logical address*" are used interchangeably.

In summary, in a virtual memory system,

- Logical memory is different from physical memory. The **logical address space** of a process is the set of logical addresses it is allowed to reference. The **physical address space** is the set of physical addresses in memory that are allocated to the process at a given time.
- Logical and physical address spaces are divided into uniform-size blocks called **pages**. Copies of logical pages are always kept on the secondary storage device. When they are accessed, they are brought into physical memory. The page-sized units of physical

memory (what would be called cache blocks in a cache) in which logical pages are stored are called **frames**.

- Physical memory is treated like a fully set-associative cache for secondary storage. Any logical page of a process can be placed in any physical page in its allotment of memory. But physical memory is not physically an associative memory, so if a logical page might be in any frame, the only way to be able to find it would be to store tags with each address and search through all tags, or to use a look-up table. The latter strategy is how the pages are located.
- The sum of the memory used by all active processes can be larger than the amount of physical memory.
- A single program can have a logical address space larger than physical memory. Not all of the logical address space of a program has to be in physical memory while the program is executing; only those pages currently in use must be in physical memory.
- Programs are easily ***relocated*** between different parts of physical memory because the method of placing logical pages into physical memory allows them to be placed anywhere while at the same time allowing the processor to access them easily.
- The translation between logical and physical addresses provides a natural means of protecting programs and data from other processes.

Paging and Caching: An Analogy

The concepts are the same, but the terminology is different.

Caching	Paging
Memory block address	Virtual memory address
Cache block location + block offset	Physical address
Memory block	Page
Cache miss	Page fault

Address Translation

Address translation is the mapping of logical to physical addresses. A page is a block of memory. To make this work efficiently, page size must be a power of 2. A logical address is separated into a high-order set of bits called the page number and a low-order set of bits called the page offset:

*Logical address = page number*page size + page offset*

The page number is shipped to a page translation mechanism that uses it to generate a physical memory address for the start of that page provided that it is in memory. If it is not in memory, this is the equivalent of a cache miss, but it is called a ***page fault***. The physical page address is called the ***physical page number***. It forms the high order bits of the physical address as shown in the figure below.

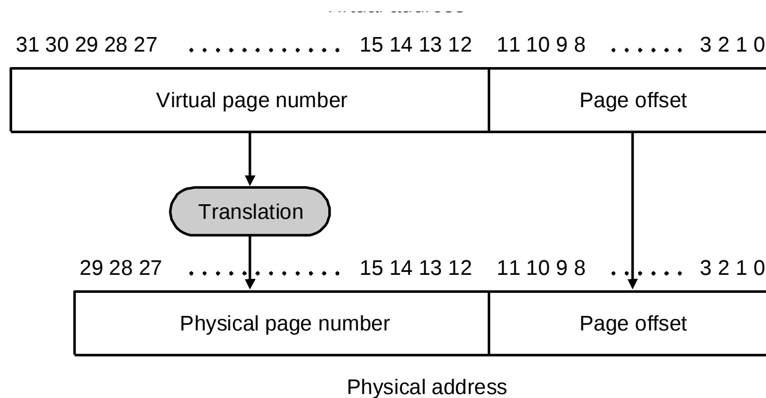


Figure 23: Conceptual translation

In the example above, the logical pages are 4096 (2^{12}) bytes each and the physical memory is 1 GB = 2^{30} bytes. The translation is a combination of hardware and software, as will become clear shortly.

Design Choices to Improve Performance

1. Page faults are very costly (disk is very slow); reduce cost of a fault by making pages large (4KB to 64KB) to take advantage of spatial locality. But just as cache blocks that are too large reduce performance, large page sizes can become inefficient and costly.
2. Reduce the fault rate by placing translations in fully associative tables to maximize flexibility of placement of page translations.
3. Handle faults in software because the software overhead is small relative to disk access time, and software logic for replacement can be more easily implemented.
4. Use write-back for writes; write-through is much too costly.

Page Placement and Faults: Finding Pages

A page table is a table that maps all virtual pages to physical page numbers. There is a unique entry for every virtual page. The index of the table is simply the virtual page number, and the contents of that table entry include the physical page number.

Because each virtual page number is the index into a unique entry in the page table, the entry does not need a tag field to distinguish between different virtual pages – the page number is the complete index.

Each process has the same set of virtual addresses, called a virtual address space. Typically virtual addresses are 32 bits long in a 32 bit architecture, making it possible to address 4GB of virtual memory. Because all processes reference the same set of virtual addresses, each must have its own unique page table. The page table for a process is part of the process's state and is located within the process's logical address space.

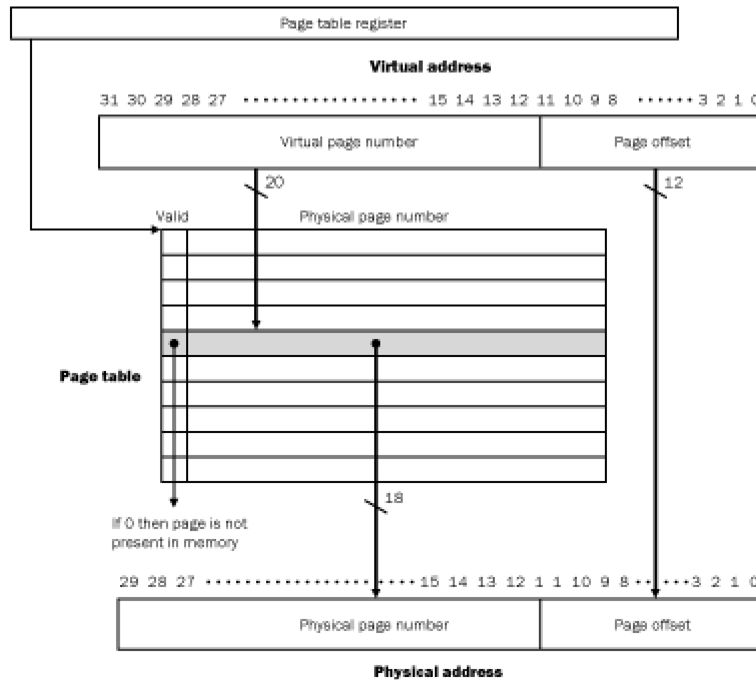


Figure 24: Page translation using the page table (alone)

Write-Back Policy

In a virtual memory system, physical memory is always treated like a write-back cache, never write-through. Writing to a page makes the page ***dirty***. Until a dirty page is replaced, it is not written to disk; only on replacement does the write-back to disk take place. In contrast, if a page is replaced whose dirty bit is not set, it does not need to be written back. It is therefore better to replace clean pages than dirty pages, to avoid the extra disk access. The page tables must keep a dirty bit in each page to keep track of which pages are clean or dirty. The page table also contains the page's address on secondary storage; sometimes a second table is used for this purpose.

LRU Page Replacement

Least Recently Used (LRU) page replacement chooses for replacement the page that has not been used for the longest time.

Pure LRU is too expensive in hardware and software. Approximations to it use hardware reference bits, set by the operating system when a page is referenced and cleared by OS periodically. Each page has a reference bit as well as a dirty bit. Some approximations use counters in the page table. The counters are cleared by a page reference and incremented on a regular basis by a clock. The page with the highest counter has not been referenced for the longest time.

Page Table Implementations

Size of page table.

With a 32 bit address, 4KB pages, 4 bytes per page table entry, there are

$2^{32} / 2^{12} = 2^{20}$ pages

The page table would have 4 bytes/entry $* 2^{20}$ entries = 2^{22} bytes = 4MB

Each active process would require a 4MB page table! Page tables would eat up memory.

Solutions

1. Only use as much memory for the page table as its current size by keeping a bounds register for the page table and allocating more memory for it if the page number is larger than the bounds register.
2. Like 1, but allow the page table to grow in either direction, i.e., two separate page tables that grow in opposite directions. (Required when compiler generates stack and heap based storage areas.) Not useful when address space is used sparsely.
3. Use inverted page tables. An inverted page table maps virtual page numbers to physical pages using a hash function. In other words, in an inverted page table, there is not an entry for each virtual page number. Instead, the table is a collection of pairs of the form (virtual page number, physical page number), and a hash function is applied to the virtual page number to find the pair, to look-up the physical page number.
4. Use multilevel page tables. Multilevel page tables are a tree-structured collection of page tables. With multilevel page tables, the virtual page number is partitioned into separate table indices. For example, the upper 5 bits might be the index into the root table, then the next 5 bits, the index into the second-level page table, the next 5 bits, an index into the third-level page table, and so on. The highest level is checked first. Only if the valid bit is set, is the lower level checked. Only those page tables actually used for translations exist. This scheme allows sparsely used address spaces to be stored more efficiently.
5. Paged page tables. Page tables can be placed into the paged memory. Each process has a page table that keeps track of where the page table's pages are stored. This per-process page table is kept in the operating system's address space, which is often non-paged. The complete page table for each process is kept in secondary storage.

In practice, modern systems tend to use the fourth choice, multilevel page tables, with in the most extreme case, four levels. For example, in the Intel 32-bit architecture, multilevel page tables are used, and the number of levels depends upon the specific architecture.

- With 32-bit paging using 4 KB pages, the high order 20 bits are divided equally into two 10-bit index values. Bits 31:22 identify the first page table entry and bits 21:12 identify the second page table, which stores the page frame number. Bits 11:0 of the linear address are the page offset within the 4-KB page frame.
- With PAE paging and 4 KB pages, the first index is 2 bits (31 and 30) and points to the highest level page table. Bits 29:21 identify a second level page table entry and bits 20:12 identify a third, which contains the page frame number.

- In the IA-32e architecture, addresses are extended to 48 bits. With 4 KB pages, each page table has 512 entries and translation uses 9 bits at a time from the 48-bit linear address. Bits 47:39 identify the highest page table entry, bits 38:30 identify the next, bits 29:21, the third, and bits 20:12 identify the fourth, which stores the page frame number.

Translation Lookaside Buffer (TLB)

Without any other hardware to support it, a virtual memory system will slow down execution by a factor of at least, because every memory access will require two or more accesses: one for each level of page table access (with multilevel page tables this can be severe) to retrieve the page table entry and one to do the actual load or store operation. In short, each absolute address computation would require a number of accesses corresponding to the page table depth. Furthermore, these accesses cannot be performed in parallel since they depend on the previous table lookup's result.

So designers came up with the idea of caching the translations of the virtual addresses into physical addresses. The cache in which the translations are stored is called a ***Translation Lookaside Buffer*** (TLB). Since the page offset part of the virtual address does not play any part in the computation of the physical frame address, only the rest of the virtual address is used as the tag for this cache (the TLB). The tag field therefore contains the bits of the virtual page number, and the data field contains the frame number of this page, if present. The valid bit indicates whether the page translation is meaningful. The dirty and reference bits indicate whether the page is dirty or recently used.

Depending on the page size this means hundreds or thousands of instructions or data objects share the same tag and therefore same tag. For example, if pages are 4 KB, then every word address in a single page has the same tag, which means that 2^10 distinct one-word structures have the same tag.

The TLB caches the most recently used portions of the page table. It is usually a small cache because it has to be extremely fast. Modern CPUs provide multi-level TLBs, just like the other caches; the higher-level caches are larger and slower. The small size of the level 1 TLB is often made up for by making it fully associative, with an LRU replacement policy. If it is larger, it might be set associative. The level 1 TLB is often separated into an instruction TLB (iTLB) and a data TLB (dTLB). Higher-level TLBs are usually unified, as is the case with the L2 and L3 caches.

In a system with a TLB, the steps followed on a reference are:

1. Look up page number in TLB.
2. If a hit, turn on reference bit (and also dirty bit if a write) and form physical address.
3. If a miss, go to page table with page number and get translation.
4. If the translation is there, load it into TLB and try again.
5. Otherwise, signal the OS to retrieve the page from disk, wait for it to return it, and load it into the page table and then load the TLB and try again.

6. If an entry needs to be replaced, its reference and dirty bits have to be written back to the page table. (They are the only thing that changes in the data field). The write always takes place at replacement time.

Typical TLB parameters:

size:	32- 4096 entries
block size:	1 – 2 page table entries
hit time:	0.5 – 1 clock cycle
miss penalty:	10 – 30 clock cycles
miss rate:	0.01% - 1%

Integrating the TLB, Cache, and Virtual Memory

When a cache and TLB are both present, the question is, should the TLB or the cache be accessed first? If the cache is *physically indexed and physically tagged*, it means that the cache is referenced using physical addresses, not virtual addresses. This means that the virtual addresses must be translated before the cache is accessed. In this case the TLB access precedes the cache access, as shown in Figure 25.

The alternative is to access the cache with virtual addresses, in which case it is called a *virtually addressed cache*. Because both the tags and the index bits come from a virtual address, the cache is also said to be *virtually indexed* and *virtually tagged*. In this case, the TLB is not accessed first. The processor sends the virtual address directly to the cache. The cache is indexed with the virtual address and the tags are virtual addresses. The data is the actual data stored at that virtual address. On a cache hit, the data in that virtual address can be delivered to the processor, but on a cache miss, the virtual memory system must be used to locate the physical page in which the missed address resides.

A virtually addressed cache does not work if multiple processes may be accessing the cache, because different processes have the same virtual address space, and ***aliasing*** may result. To illustrate, suppose we have two processes, P and Q. P has a virtual address 5062 that maps to physical address 1020. Q has a virtual address 5062 that maps to physical address 3280. There is just one entry in the cache for virtual address 5062. Suppose P runs and writes that location. Now Q runs and reads 5062. Q will not get the data from 3280, but from 1020. When caches are virtually indexed and tagged, this problem must be overcome.

One method is a compromise in which the tag is translated to a physical address but the index remains virtual. This is called a *virtually indexed and physically tagged* cache. If the index bits are entirely contained within the page offset, then they are actually physical addresses, since the page offset is not virtual. If the index bits extend past the highest-order page offset bits, then the index is in fact partly a virtual address. If the index bits are entirely contained within the page offset, then the page offset is used to index the cache and the TLB is used to translate the page number to a physical address, which is given to the cache as well. The aliasing problem disappears because, in effect, the cache acts like a physically indexed cache.

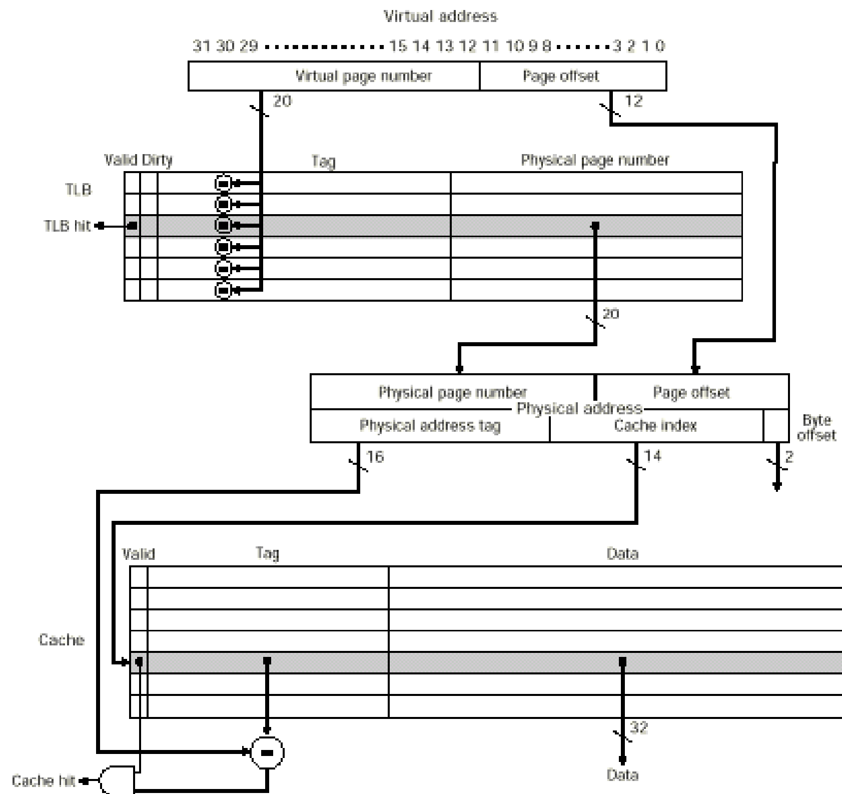


Figure 25: Integrating the TLB with a physically addressed cache

An Example: The MIPS R2000 TLB

Features:

1. Physically indexed and tagged
2. 4 KB page size
3. 32-bit address space
4. TLB has 64 entries and is fully associative
5. It is shared between instructions and data, has 64 bits per entry, consisting of a 20 bit tag and a 20 bit physical page number, and valid and dirty bits.

Handling Page Faults and TLB Misses

Unlike cache misses, which are handled entirely in hardware, page faults are handled partly by software and partly in hardware. Which does which steps is system-dependent. The general steps are the topic of this section. We assume a physically tagged and indexed cache.

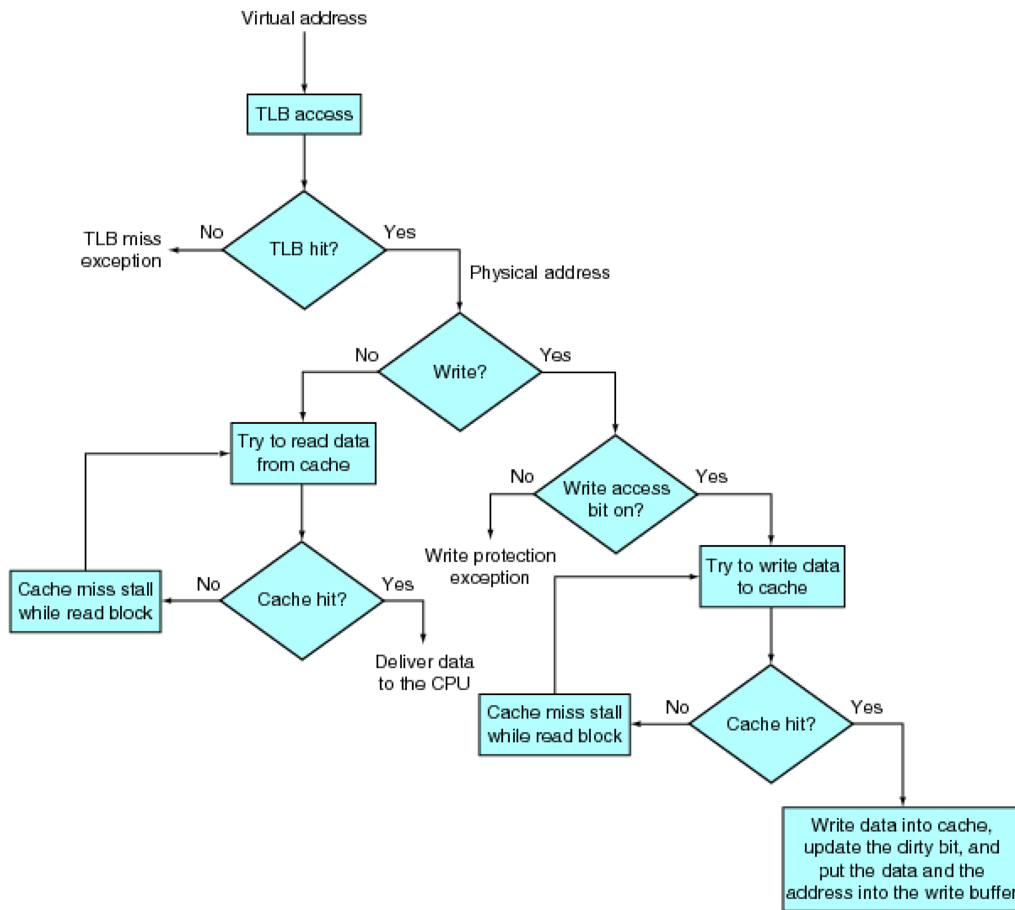


Figure 26: Algorithm for page translation using physically addressed cache

When the processor references a page:
if it is present in the TLB,

the physical page number is copied from the TLB into the physical address register and the control bits are updated as necessary;

else

the page table base register contents are accessed and used with the page number to locate the page table entry in physical memory;
memory is accessed and the page table entry is read into the MMU;
if the valid bit is set (the page is in memory),

an entry is added to the TLB for this logical page/physical page pair, replacing an existing entry if necessary.

else

(a page fault has occurred) page is not in memory, the hardware circuits in the MMU generate an exception to the CPU to indicate the fault. The exception causes the running process to be preempted and its current program counter (PC) to be saved (usually into an exception PC register). and the operating system to run. The operating system finds the page on disk and chooses a frame into which to place it. If it must, it replaces an existing page, writing it to disk first if it is dirty. The operating

system updates the page table to contain the correct translation, setting the valid bit, and the entry is copied into the TLB. The reference to the page is re-executed.

The physical address is sent to the cache.

If it is a caches miss

the address is given to memory, which fetches the block and delivers it to the cache controller.

The contents of the cache are delivered to the processor.

The table below summarizes what combinations of events are possible in a combined virtaul memory system with a physically-tagged and indexed cache.

TLB	Page Table	Cache	Possible?
Hit	Hit	Hit	Yes, but the page table is not checked on a TLB hit.
Hit	Hit	Miss	Yes, but the page table is not checked on a TLB Hit
Hit	Miss	Hit	No, even if the page table were checked, every translation in the TLB must be in the page table.
Hit	Miss	Miss	No, for the same reason as above.
Miss	Hit	Hit	Yes -- not in TLB, found in page table, and data is in cache.
Miss	Hit	Miss	Yes -- not in TLB, found in page table, but the data is not in the cache.
Miss	Miss	Hit	No, because this implies data is in cache but not in memory.
Miss	Miss	Miss	Yes -- page fault and then a cache miss.

Virtual Memory and Protection

All modern computers have multiprogramming operating systems, which means that they allow multiple programs to be in memory at the same time, and they timeshare the processor among them. When multiple programs are memory resident, this leads to problems that do not exist when only one is in memory at a time. One problem is that programs could access each others' data and/or code. Therefore, the operating system must prevent different programs from accessing or modifying each others', or the operating system's, in-memory data and code. Without some type of hardware support, this is infeasible.

At the very least, the hardware must provide three capabilities:

1. The processor must have two different modes of operation:
   - supervisor or privileged or kernel mode, and
   - user or unprivileged mode
2. Instructions that can be executed only in privileged mode, and portions of the processor state that can be read but not modified in unprivileged mode, and only modified in

privileged mode. For example, the bit that shows which mode it is in should be readable but not writeable in unprivileged mode.

3. An ability to switch between the two modes. Usually this is accomplished with special traps, or exceptions, that transfer control to a specific address in the operating system's address space in which the change of context is carried out.

In addition, in a virtual memory system, it must ensure that the page tables themselves are not modifiable by any user process, i.e., they must be placed in the kernel's (operating system's) address space.

Each running process has its own page table. To use memory efficiently, some processes are allowed to share physical pages. For example, frequently run software, such as the C runtime libraries, are shared among processes. To make this possible, the page table entries of different processes map to the same physical pages. But to prevent processes from modifying such shared pages, there have to be bits that indicate that such logical pages are not writable. This property of being non-writable is process-specific, as some processes may have the ability to modify a physical page but not others. Therefore this bit belongs in the page table, not to the frame or physical page. The bit that controls whether or not a page can be modified is typically called something like the read-only bit, or the write-bit.

In general, each page should have the following bits[3]:

`PAGE_PRESENT`	Page is resident in memory and not swapped out (valid bit)
`PAGE_RW`	Set if the page may be written to
`PAGE_USER`	Set if the page is accessible from user space (to protect system pages)
`PAGE_DIRTY`	Set if the page is written to
`PAGE_ACCESSED`	Set if the page is accessed (reference bit)

These bits are part of a page table entry and must be supported by the hardware. In other words, the processor and TLB should have operations to access these bits easily. It is not necessary that the bits be easily accessible when they are in the page table but not in the TLB, because on a TLB miss, the page table entry is first brought into the TLB before the entry is accessed.

Multiprocessor Cache Coherency

In a shared memory multiprocessor with separate caches in each processor, the problem is that the separate caches can have copies of the same memory blocks, and if the protocol does not address this issue, the copies will end up having different values for the blocks. This will happen whenever two different processors update their copies with different values and they each use a write-back policy. Because they use write-back neither sees the change in the block made by the other.

Example

Suppose a machine has two processors named A and B have write-through caches. Suppose that each has loaded a copy of a memory address X into its respective cache and that before the load, X had the value 1. Then they each have a 1 in the cache block for X. Now processor A writes the value 2 into X, which it finds in its own cache. Because it is write-through, a 2 is written to the

3 These names are used in Linux implementations.

memory copy of X. Now processor B's copy of X is old, and B will be using an outdated value of X.

When two caches have different values, they are said to be ***incoherent***. ***Cache coherence*** refers to the state of the system in which all caches have up to date copies of the memory blocks. A more precise definition is as follows.

Definition. A multiprocessor memory system is ***cache coherent***, if for any processors P and Q,

1. A read by P of location X that follows a write by P to X, with no writes to X by another processor between the write and the read by P, always returns the value written to X by P.
2. A read by P of location X that follows a write by Q to X returns the written value by Q if the read and write are sufficiently separated in time, and no other writes to X occur between the two accesses.
3. Writes by P and Q to the same location are serialized. This means that if P writes a value c to location X and Q writes a value d to X, then all processors will see either c followed by d or all processors will see d followed by c, but it is not possible for some processors to see c followed by d and others to see d followed by c.

The first condition is just the ordinary order-preserving property that single-processor machines expect to see -- if a process writes a value to memory and then reads that location, it should see what it wrote.

With respect to the preceding example, when A writes a 2 to X, then B should see that 2 in its cache before it attempts to access X, provided that enough time has passed. If A and B both tried to write to X at the same time, one's value would be written to memory first and then the other's, but there is no specific order that must be followed.

Coherence Enforcing Methods and Cache Snooping

The caches in a multiprocessor provide two different mechanisms for handling shared data in order to make cache coherence possible:

Data Migration. A block of data is moved into a local cache so that it can be used locally in that cache by its processor. Migration reduces the cost of accessing shared data items across a communication channel such as a bus or network.

Data Replication. A shared data item that is read (and not written) it is copied into each cache that reads it. This reduces memory accesses and contention.

The hardware has to provide this functionality to make any cache coherence method feasible. A set of rules that the hardware uses to provide cache coherence is call a ***cache coherence protocol***. In order for such a protocol to work, the state of each data block has to be tracked, and this must be facilitated by the hardware. What kinds of information are in the state depends on the protocol.

The most common cache coherence protocol is ***cache snooping***. Each cache snoops on the bus activity. See Figure 27. Each cache has a special tag memory and controller called the ***snoop tag***. The snoop tag is essentially a copy of the tag memory of the cache and logic circuits that listen to all bus activity to determine whether memory addresses placed on the bus match any valid blocks that it has, without interfering with cache operation..

Briefly, the way they work is that, on a read miss, all caches check to see if they have a copy of the requested data. If they do, they supply the data to the cache that missed (data migrates). On a write to any cache, all caches check to see if they have a copy of the written block, and if they do, they either invalidate it or update their copy.

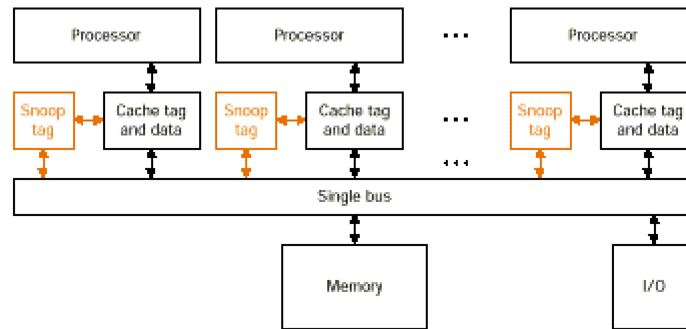


Figure 27: Snooping cache schematic diagram

Cache Snooping Protocols

There are two common cache snooping protocols: ***write-invalidate*** and ***write-update***.

Write-invalidate

In this method, a writing process guarantees that it has exclusive access to the data item it is about to write. The writing processor causes all copies in other caches to be invalidated before changing its local copy by sending out an invalidate signal to all other caches telling them to invalidate their copies of the item. It then writes its local data. Until another processor requests the same block, it remains valid and no further invalidation is required. This is a multiple-reader,single-writer synchronization problem. It is analogous to a write-back cache in a single processor system in that it has a valid copy but memory does not.

Advantage: reduces bus activity because only first write invalidates; subsequent writes do not require invalidate signals.

Write-update

The writing processor broadcasts the new data over the bus and all copies are then updated. This is also called ***write broadcast.*** In this protocol, the data itself is sent over the bus to the other caches. It is analogous to a write-through cache in a single processor system.

Advantage: makes data available immediately, reducing cache latency seen at processors.

Commercial caches often use write-invalidate.

Role of Block Size

The larger the block, the more unnecessary data is transferred or invalidated. If one word is changed in an 8-word block, the entire block is invalidated or updated. This reduces bandwidth unnecessarily. Large block sizes can create an effect called ***false sharing***. As blocks get larger and larger, they can contain different data items that are being used by different processors. At that point the data items are part of the same bock and so the block now looks like a shared block, even though the different processors are not sharing the same data item. Smaller block sizes reduce the probability of false-sharing.

An Example of a Cache Coherency Protocol (optional)

A cache coherency protocol can be completely described by the states that a cache block can be in, and the effects upon those states caused by all possible events of interest, either inside the processor to which the cache belongs, or "on the bus", meaning events caused by some other processor. Events on the local processor are things like reads to the cache block or writes to it. Events on the bus are things like writes to a copy of that block by another processor, or invalidate signals for that block sent on the bus by a processor. Therefore, we can describe protocols as finite state diagrams, in which the states are those of an arbitrary cache block. There are 3-state and 4-state protocols.

A Basic 3-State Protocol

The textbook describes the simplest 3-state protocol. In this protocol there are three states: (1) ***read-only*** or ***shared***, (2) ***read/write***, ***exclusive***, or ***dirty***, and ***invalid***. They have the following meanings.

Read-only: It is a clean block that can be ***shared***. I.e., it has only been read.

Read/Write: The block is dirty because the processor wrote to it; it is now in a state in which it cannot be shared by another processor. Also called ***dirty*** or ***exclusive***, as noted above.

Invalid: The block has been invalidated by a cache belonging to another processor.

Explanations of the Transitions

If the processor has a read-miss for a cache block, no matter what the current state of the block (in this cache), it needs to get the block from memory or from another cache if possible. If the caches are write-back caches, one may have a more recent copy than memory, since it might have been written to but not yet written back to memory. If such a dirty copy exists, the cache that owns the dirty copy aborts the memory read operation placed on the bus. The processor owning this cache must force the cache to write its dirty block back to memory before the read takes place. If the cache that had the read miss has a dirty block that needs to be replaced to load this one, that write-back must take place before the load. All caches must therefore monitor the bus for read misses. The read miss ultimately causes the block to become read-only.

If the processor writes data to a block that is currently shared or read-only, it must send an invalidate signal to other caches. The data had been clean in it, so it is safe to overwrite it, but now the block is now dirty.

If the block was in a read-only state but there was a write miss for that block (multiword blocks), it is the same as a write-hit. The cache must get the block from memory or another cache, so it sends an invalidate signal (and hopes a cache will send the block). Eventually it gets the block and writes to it, so it becomes a dirty block.

If a processor writes to a block that is already dirty, nothing has to be done, because the processor itself made the block dirty and it is free to change its contents.

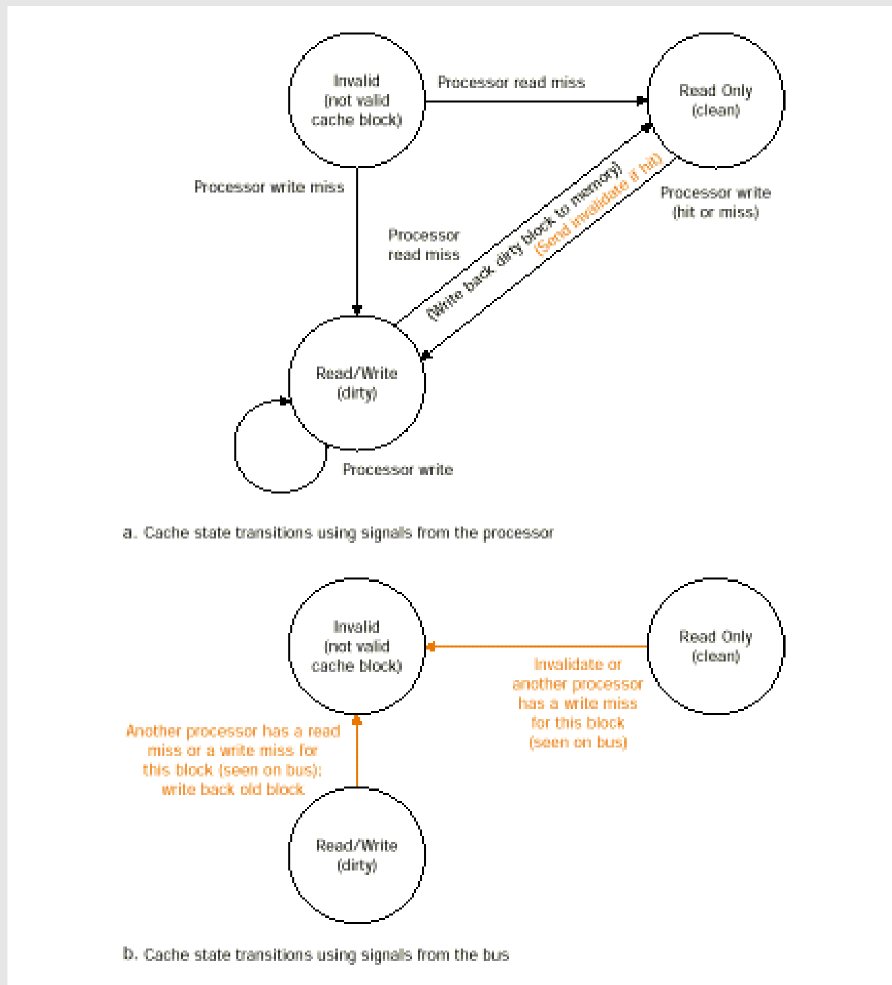


Figure 28: Finite state automata of snooping cache protocol

If a block is invalid, an attempt to write to it is a write miss. In this case, it must get the block from memory or the other caches (depending on whether they support this) and set it to dirty.

These transitions can be summarized by the pair of state transition diagrams in Figure 28. The first FSA defines the transitions caused by processor events; the second defines the transitions caused by bus events. I have also included a second FSA instead of the FSA in Figure 28b, because I believe that the one in Figure 28b is incorrect. The problem has to do with read misses seen on the bus when a block is in the dirty state. If a block is dirty and some other processor has a read miss for that block, then that processor will obtain that block from memory or some other cache. If the block is dirty, it means that the local processor wrote to the block but no other processor has changed a copy of it since that write. Therefore, this is the most recent copy of the block and it is this copy that must be given to the cache that had the miss. When this happens, all caches will have a copy of this block and it will therefore be clean again.

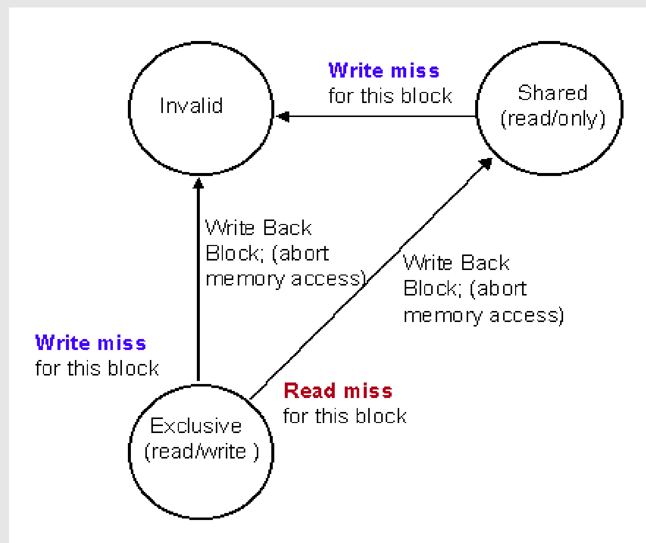


Figure 29: An alternative FSA for the bus events

Appendix

Real Stuff (from the textbook)

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
Virtual address	48 bits	48 bits
Physical address	44 bits	48 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement The L2 TLB is four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB has 512 entries TLB misses handled in hardware	1 L1 TLB for instructions and 1 L1 TLB for data per core Both L1 TLBs fully associative, LRU replacement 1 L2 TLB for instructions and 1 L2 TLB for data per core Both L2 TLBs are four-way set associative, round-robin Both L1 TLBs have 48 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

FIGURE 5.38 Address translation and TLB hardware for the Intel Nehalem and AMD Opteron X4. The word size sets the maximum size of the virtual address, but a processor need not use all bits. Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present. Nehalem supports two hardware-supported threads per core (see Section 7.5 in Chapter 7). Copyright © 2009 Elsevier, Inc. All rights reserved.

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?) clock cycles

FIGURE 5.39 First-level, second-level, and third-level caches in the Intel Nehalem and AMD Opteron X4 2356 (Barcelona). Copyright © 2009 Elsevier, Inc. All rights reserved.