7. The Memory Hierarchy (1)

The possibility of organizing the memory subsystem of a computer as a hierarchy, with levels, each level having a larger capacity and being slower than the precedent level, was envisioned by the pioneers of digital computers.

The main argument for having a memory hierarchy is economics: a unique, large memory, running at the CPU's speed, would be prohibitive, if possible at all. What makes the memory hierarchy idea work is the principle of locality.

Example 7.1 MEMORY CHIPS AND THEIR CAPACITY:

How many chips are necessary to implement a 4 MBytes memory:
1) using 64 Kbit SRAM;
2) using 1Mbit DRAM;
3) 64 KBytes using 64 Kbit SRAM and the rest using 1Mbit DRAM.

Answer:
The number of chips is computed as:

$$\frac{\text{Memory capacity (expressed in bits)}}{\text{Chip capacity (expressed in bits)}}$$

or as

$$\frac{\text{Memory capacity (expressed in bytes)}}{\text{Chip capacity (expressed in bytes)}}$$

1)n1 = $2^{22}/2^{13}$ = 512 chips.(using the second formula)
2)n2 = $2^{22}/2^{17}$ = 32 chips.
3)n3 = $2^{16}/2^{13}$ + floor((2^{22}-2^{16})/2^{17}) = 8 + 32(SRAM + DRAM)

SRAM circuits are designed to be very fast but they have smaller capacities than DRAM and are more expensive. DRAMs, on the other hand are slower (tens to hundreds of nanoseconds cycle time), but their capacity is very large. Using only SRAM results in a memory that matches very well the CPU's speed, but is very expensive and bulky, not to mention packaging, cooling and other problems. Using DRAMs results in a small dimension memory (only 16 chips) which is cheap but relatively slow: the CPU will have to wait at every memory access until the operation, read or write, is done.

What we would like is a memory that is fast like the SRAM and, at the same time, as cheap and compact as the DRAM version. With a proper organization solution 3 could be the needed compromise; obviously it won't be as fast as the pure SRAM memory, nor so cheap as the DRAM.

7.1 The Principle of Locality

In running a program the memory is accessed for two reasons:

- read instructions;
- read/write data.

Memory is not uniformly accessed; addresses in some region are accessed more often than others, and some addresses are accessed again shortly after the current access. In other words programs tend to favor parts of the address space at any moment of time.

- **temporal locality:** an access to a certain address tend to be repeated shortly thereafter;

- **spatial locality:** an access to a certain address tends to be followed by accesses to nearby addresses.

The numerical expression of this principle is given by the 90/10 rule of thumb: 90% of the running time of a program is spent accessing 10% of the address space of that program. If this is the case, then it is natural to think at a memory hierarchy: map somehow the most used addresses to a fast memory that needs to represent roughly only 10% of the address space, and the program will run for most of the time (90%) using that fast memory. The rest of the memory can be slower because is accessed less, it has to be larger though. This is not that difficult because slower memories are cheaper.

A memory hierarchy has several levels: the uppermost level is the closest to the CPU and it is the fastest (to match the processor's speed) and the smallest; as we go downwards to the bottom of the hierarchy, each level gets slower and larger as the previous one but with a lower price per bit.

As for the data different levels of the hierarchy hold, each level is a subset of the level below it, in that data in one level can be found in the level immediately below it. Memory items (they may represent instructions or data) are brought into the higher level when they are referred for the first time because there is a good chance they will be accessed again soon, and migrate back to the lower level when room must be made for newcomers.

7.2 Finite memory latency and performance

In Chapter 5 we discussed about the ideal CPI of instructions in the instruction set. At that moment we assumed the memory is fast enough to deliver the item being accessed without introducing wait states. If we look at Figure 5.1, we see that in state Q1 the MemoryReady signal is tested to determine if the memory cycle is complete or not; if not, then the Control Unit returns in state Q1, continuing to assert the control lines necessary for a memory access. Every clock cycle in which MemoryReady = No increases the CPI for that instruction with one. The same is true for load or store instructions. The bigger the memory's response time is, the higher the real CPI for that instruction is. Suppose an instruction has an ideal CPI of n (i.e. there is a sequence of n states in the state-diagram, corresponding to this instruction), and k of them are addressing cycles; k=2 for load/store instructions, and k=1 for all other ones in our instruction set. The ideal CPI for this instruction is:

$CPI_{ideal} = n$ (clock cycles per instruction)

If every addressing cycle introduces w waiting clock cycles, we have the real CPI for this instruction:

$CPI_{real} = n + k * w$ (clock cycles per instruction)

Example 7.2 IDEAL AND REAL CPI:

We want to calculate the real CPI for our instruction set; assume that the ideal CPI is 4 (computed with some accepted instruction mix). Which is the real CPI if every memory access introduces one wait cycle? Loads and stores are 25% of the instructions being executed.

Answer: Using the above formulae we have:
$n = 4$
$w = 1$
$k = 1$in $f1=75\%$ of cases
$k = 2$in $f2=25\%$ of cases (the loads and stores)

We get:
$CPI_{real} = 4 + (f1*1 + f2*2)*w$
$CPI_{real} = 4 + (0.75*1 + 0.25*2)*1$
$CPI_{real} = 4 + 1.25 = 5.25$

A machine that had an ideal memory would run faster then the one in our problem by:

$$\frac{CPI_{real}}{CPI_{ideal}} - 1 = \frac{5.25}{4} - 1 = 0.31 = 31\%$$

The above example should make clear what a big difference is between the expectations and the reality. We must have a really fast memory close to the CPU to get full advantage of the CPU's performance. As a final comment, it may happen that the read and write behave differently, in that they require different clock cycles to conclude; the formula giving CPI_{real} will be then slightly modified.

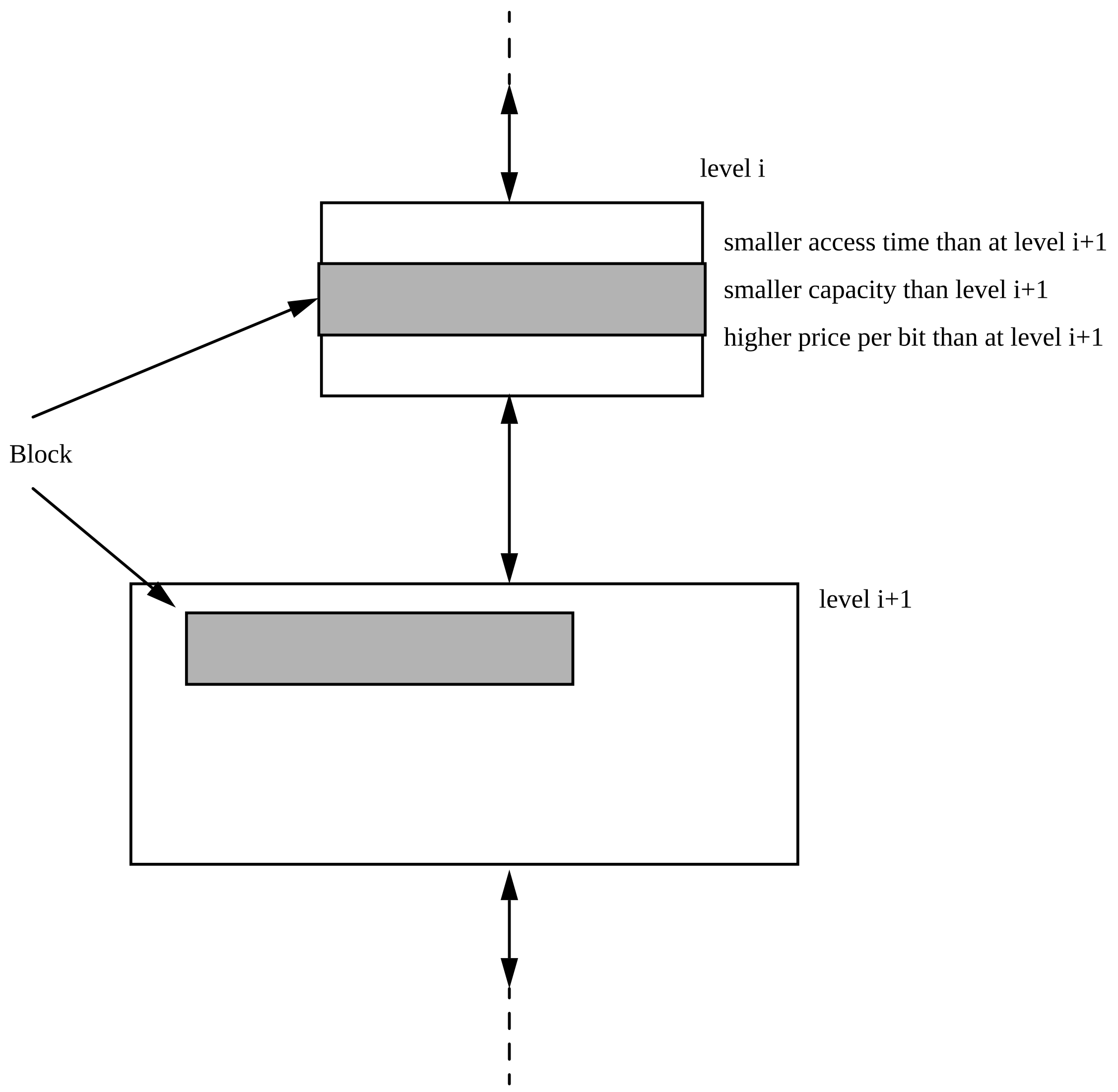
level i
smaller access time than at level i+1
smaller capacity than level i+1
higher price per bit than at level i+1
Block
level i+1

FIGURE 7.1 Two levels in a memory hierarchy. The unit of information that is transferred between levels of hierarchy is called **block.**

7.3 Some Definitions

Information has to migrate between the levels of an hierarchy. The higher a level is in the hierarchy, the smaller its capacity is: it can accommodate only a a small part of the logical address space. Transfers between levels take place in amounts called blocks, as can be seen in Figure 7.1.

A block may be:

- **fixed size**
- variable size.

Fixed size blocks are the most common; in this case the size of the memory is a multiple of the block size.

Note that it is not necessary that blocks between different memory levels all have the same size. It is possible that transfers between level i and i+1 are done with blocks of size b_i, while transfers between level i+1 and i+2 is done with a different block size b_{i+1}. Generally the block size is a power of 2 number of bytes, but there is no rule for this, and, as a matter of fact, deciding the block size is a difficult problem as we shall discuss soon.

The reason for having a memory hierarchy is that we want a memory that behaves like a very fast one and is cheap as a slower one. For this to happen, most of the memory accesses must be found in the upper level of the hierarchy. In this case we say we have a **hit**. Otherwise, if the addressed item is in a lower level of the hierarchy, we have a **miss;** it will take longer until the addressed item gets to the CPU.

The **hit time** is the time it takes to access an item in the upper level of the memory hierarchy; this time includes the time spent to determine if there is a hit or a miss. In the case of a miss there is a **miss penalty** because the item accessed has to be brought from the lower level in memory into the higher level, and then the sought item delivered to the caller (this is usually the CPU). The miss penalty includes two times:

- the **access time** for the first element of a block into the lower level of the hierarchy;
- the **transfer time** for the remaining parts of the block; in the case of a miss a whole block is replaced with a new one from the lower level.

The **hit rate** is the fraction of the memory accesses that hit. The miss rate is the fraction of the memory accesses that miss:

miss rate = 1 - hit rate

The hit rate (or the miss rate if you prefer) does not characterize the memory hierarchy alone; it depends both upon the memory organization and the program being run on the machine. For a given program and machine the hit rate can be experimentally determined as follows: run the program and count how many times the memory is accessed, say this number is N, and how many times accesses are hits, say this number is N_h; then the hit ratio (H) is given by:

$$H = \frac{N_h}{N}$$

The cost of a memory hierarchy can be computed if we know the price per bit C_i and the capacity S_i of every level in the hierarchy. Then the average cost per bit is given by:

$$C = \frac{C_1 * S_1 + C_2 * S_2 + ... + C_n * S_n}{S_1 + S_2 + ... + S_n}$$

7.4 Defining the performance for a memory hierarchy

The goal of the designer is a machine as fast as possible. When it comes to the memory hierarchy, we want an average access time from the memory as small as possible. The average access time can't be smaller than the access time of the memory in the highest level of the hierarchy, t_{A1}.

For a two level memory hierarchy we have:

t_{av} = hit_time + miss_rate * miss_penalty

where t_{av} is the average memory access time. Do not forget that the hit time is basically the access time of the memory at the first level in the hierarchy, t_{A1}, plus the time to detect if it is a hit or a miss.

Example 7.3 HIT TIME AND ACCESS TIME:

The hit time for a two level memory hierarchy is 40ns, the miss penalty is 400ns, and the hit rate is 99%. Which is the average access time for this memory?

Answer:
The miss rate is:

miss_rate = 1 - hit_rate
miss_rate = 1 - 0.99 = 0.01

The average access time is:

t_{av} = 40 + 0.01 * 400 = 44ns

greater by 10% than the hit time.

The hit time as well as the miss time can be expressed as absolute time, as in the example above, or in clock cycles as in the example 7.4

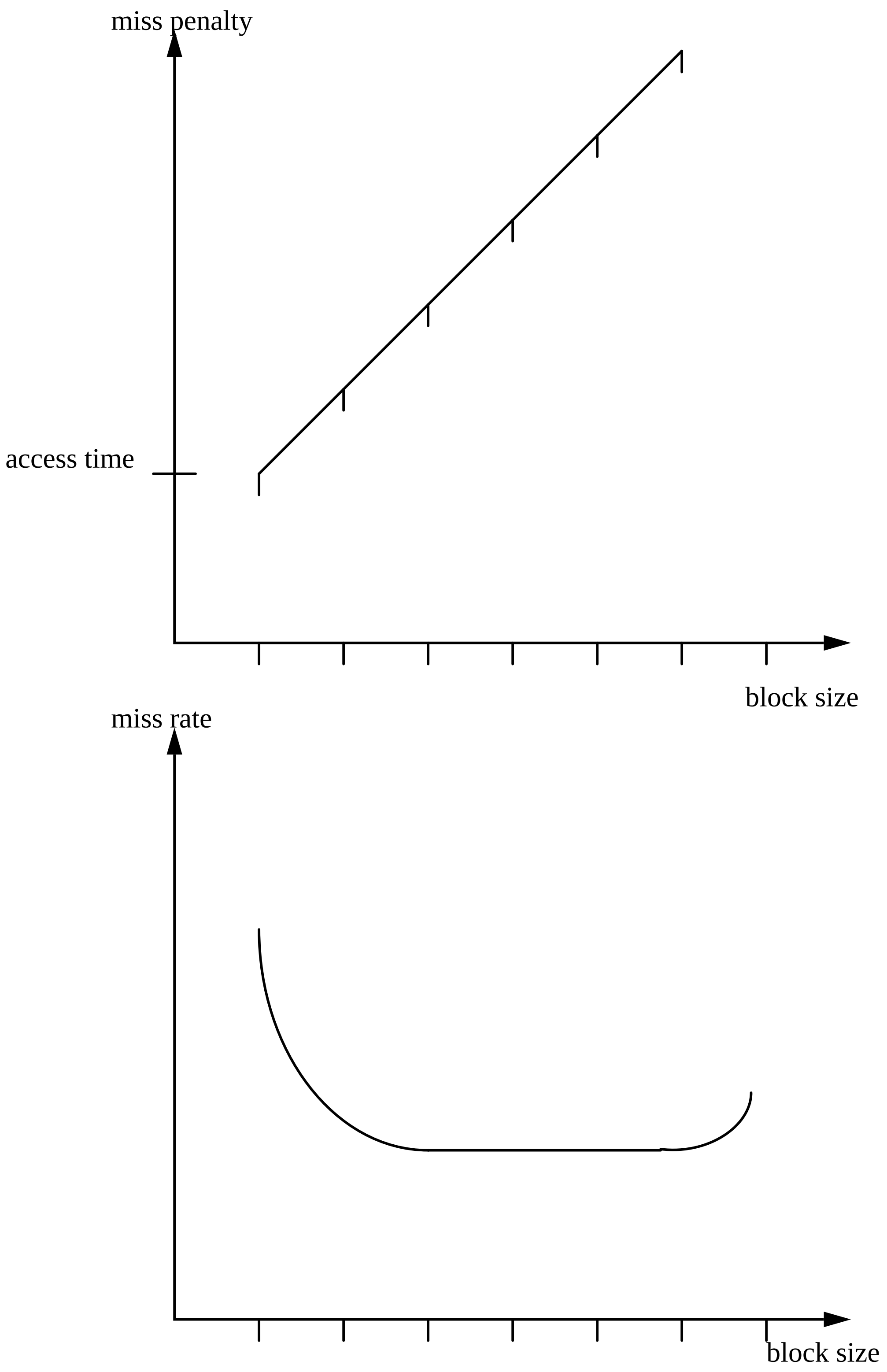

FIGURE 7.2 The relation between block size and miss penalty / miss rate.

Example 7.4 HIT TIME AND ACCESS TIME:

The hit time for a memory is 1 clock cycles, and the miss penalty is 20 clock cycles. What should be the hit rate to have an average access time of 1.5 clock cycles.

Answer:

$$\text{miss\_rate} = \frac{t_{av} - \text{hit\_time}}{\text{miss\_penalty}}$$

$$\text{miss\_rate} = \frac{1.5 - 1}{20} = 0.0025 = 2.5\%$$

hit_rate = 1 - miss_rate = 97.5%

Figure 7.2 presents the general relation between the miss penalty and the block size as well as the general appearance of a relation between the miss rate and the block size, for a given two level memory hierarchy. The minimum value of the miss penalty equals the access time of the memory in the lower level of the memory hierarchy; this happens if there is only one item transferred from the lower level. As the block size increases, the miss penalty increases also, every supplementary item being transferred taking the same amount of time.

On the other hand, the miss penalty decreases for a while when the block size increases, This is due to the spatial locality: a larger block size increases the probability that neighboring items will be found in the upper level of the memory. Above a certain block size, the miss rate starts to increase; as the block size increases the upper level of the hierarchy can accommodate fewer and fewer blocks: when the block being transferred contains more information than needed for the spatial locality properties of the program, it means that time is being spent for useless transfers, and that blocks containing useful information, which could be accessed soon (temporal locality), are replaced in the upper level of the hierarchy.

As the goal of the memory hierarchy is to provide the best access time, the designer must find the minimum of the product:

miss_rate * miss_penalty

7.5 Hardware/Software Support for a Memory Hierarchy

As we have already mentioned, the hit time includes the time necessary to determine if the item being accessed is in the upper level of the memory hierarchy (a hit) or not (a miss). Because this decision must take as little time as possible, it has to be implemented in hardware.

A block transfer occurs at every miss. If the block transfer is short (tens of clock cycles) then it is hardware handled. If the block transfer is large (hundreds to thousands of clock cycles) then it can be software controlled. Which could be the reason for such long lasting transfers? Basically this happens when the difference between memory access times at two levels of memory hierarchy are very large.

Example 7.5 ACCESS TIME AND CLOCK-RATE:

The typical access time for a hard-disk is 10ms. The CPU is running at a 50MHz clock rate. How many clock cycles does the access time represent? How many clock cycles are necessary to transfer a 4KB block at a rate of 10MB/s?

Answer:
The clock cycle is given by:

$$T_{ck}\,[\mathrm{ns}] = \frac{1000}{\mathrm{clock\_rate[Mhz]}}$$

$$T_{ck} = \frac{1000}{50} = 20\mathrm{ns}$$

The number of clock cycles the access time represent is n_A:

$$n_A = \frac{t_A}{T_{ck}} = \frac{10 * 10^6\,[\mathrm{ns}]}{20\,[\mathrm{ns}]} = 500,000 \text{ clock cycles}$$

The transfer time is t_T:

$$t_T\,[\mathrm{s}] = \frac{\mathrm{block\_size\ [Bytes]}}{\mathrm{transfer\_rate[Bytes/s]}}$$

$$t_T = \frac{4 * 10^3}{10 * 10^6} = 4 * 10^{-3}\mathrm{s} = 4\mathrm{ms}$$

The number of clock cycles the transfer represents is n_T:

$$n_T = \frac{t_T\,[\mathrm{ns}]}{T_{ck}\,[\mathrm{ns}]} = \frac{4 * 10^6}{20} = 200,000 \text{ clock cycles}$$

This example clearly shows that a block transfer from the disk can be resolved in software, in the sense that it is the CPU that takes all necessary actions to start the disk accessing process; a few thousand clock cycles are around 1% of the disk access time.

When block transfers are short, up to tens of clock cycles, the CPU waits until the transfer is complete. On the other hand, for long transfers, it would be a waste to let the CPU wait until the transfer is complete; in this case it is more appropriate to switch to another task (process) and works until an interrupt from the accessed devices informs the CPU that the transfer is complete; then the instruction that caused the miss can be restarted (obviously there must be hardware + software support to restart instructions).

7.6 How Does Data Migrate Between the Hierarchy's Levels

At every memory access it must be determined somehow if the access is a hit or miss; as such the question that can be asked is:

- **how is a block identified if it is or not in the upper level of the hierarchy?**

In the case of a miss data has to be brought from a lower level in the hierarchy into a higher level; the question here is:

- where can the block be placed in the upper level?

Bringing a new block into the upper level means that it has to replace some other block here; the question is:

- **which block should be replaced on a miss?**

As we mentioned, a lower level of the hierarchy contains the whole information in its upper level; for this to be true we must know what happens when a write takes place in the upper level:

- **what is the write strategy?**

These questions may be asked for any two neighbor levels of the hierarchy, and they will help us take the proper decisions in design.

Exercises

7.1 Consider a n-level memory hierarchy; the hit-rate H_i for the i-th level is defined as the probability to find the information requested by CPU in level i. The information in memory at level i also appears in the level i+1: hence $H_i < H_{i+1}$. The bottom most level of the hierarchy (disk or tape usually) contain the whole information, such that the hit ratio at this level is 1 (H_n=1). Derive an expression for the average access time t_{av} in this memory.

7.2 Consider the following characteristics of a 3-level memory hierarchy:

Level i	S_i [KB]	C_i [\$/bit]	t_{Ai} [ns]	H_i
1 (cache)	2	0.05	20	0.95
2 (main memory)	2000	0.005	200	0.9999
3 (disk)	200,000	0.00001	5,000,000	1

Which is the average memory access time? You may assume that the block transfer time from level i is roughly t_{Ai}, and the hit time at level i is roughly t_{Ai}.

8. The Memory Hierarchy (2) - The Cache

The uppermost level in the memory hierarchy of any modern computer is the *cache*. It first appeared as the memory level between the CPU and the main memory. It is the fastest part of the memory hierarchy, and the smallest in dimensions.

Many modern computers have more than one cache, it is common to find an instruction cache together with a data cache. and in many systems the caches are hierarchy structured by themselves: most microprocessors in the market today have an internal cache, with a size of a few KBytes, and allow an external cache with a much larger capacity, tens to hundreds of KBytes.

8.1 Some Values

There is a large variety of caches with different parameters. Below are listed some of the parameters for the external cache of a DEC 7000 system which is built around the 21064 ALPHA chip:

Block size (line size)	64 Bytes
Hit time	5 clock cycles
Miss penalty	340 ns
Access time	280 ns
Transfer time	60 ns
Cache size	4 MBytes
CPU clock rate	182 MHz

8.2 Placing a block in the cache

Freedom of placing a block into the cache ranges from absolute, when the block can be placed anywhere in the cache, to zero, when the block has a strictly predefined position.

- a cache is said to be **directly mapped** if every block has a unique, predefined place in the cache;
- if the block can be placed anywhere in the cache the cache is said to be **fully** associative;
- if the block can be placed in a restricted set of places then the cache is called **set associative.** A set is a group of two or more blocks; a block belongs to some predefined set, but inside the set it can be placed anywhere. If a set contains n blocks then the cache is called **n-way set associative.**

Obviously *direct-mapped* and *fully-associative* are particular names for a 1-way set associative and k-way set associative (for a cache with k blocks) respectively.

Transfers between the lower level of the memory and the cache occur in blocks: for this reason we can see the memory address as divided in two fields:

Block-frame address	Block offset

MSB LSB

Example 8.1 MEMORY ADDRESS:

What is the size of the two fields in an address if the address size is 32 bits and the block is 16 Byte wide?

Answer:
Assuming that the memory is byte addressable there are 4 bits necessary to specify the position of the byte in the block. The other 28 bits in the address identify a block in the lower level of the memory hierarchy.

0000 0000 0000 0000 0000 0000 0011	1101
MSB	LSB

The address above refers to block number 3 in the lower level; inside that block the byte number 13 will be accessed.

The usual way to map blocks to positions in the cache is:

- for a direct mapped cache:
  index = (Block-frame address) modulo (number of blocks in the cache);

- for a set associative cache:
  index = (block-frame address) modulo (number of sets in the cache).

For a cache that has a power of two blocks (suppose 2^m blocks), finding the position is a direct mapped cache is trivial: position (index) is indicated by the last (the least significant) $\log_2 m$ bits of the block-frame address.

For a set associative cache that has a power of two sets (suppose 2^k sets), the set where a given block has to be mapped is indicated by the last (the least significant) $\log_2 k$ bits of the block-frame address.

The address can be viewed as having three fields: the block-frame address is split into two fields, the tag and the index, plus the block offset address:

Tag	Index	Block offset
MSB		LSB

In the case of a direct mapped cache the index field specifies the position of the block in the cache. For a set associative cache the index fields specifies in which set the block belongs. As for a fully associative cache this field has zero length.

Example 8.2 POSITION OF BLOCKS:

A CPU has a 7 bit address; the cache has 4 blocks 8 bytes each. The CPU addresses the byte at address 107. Suppose this is a miss and show where will be the corresponding block placed.

Answer:
$(107)_{10} = (1101011)_2$

With an 8 bytes block the least significant three bits of the address (011) are used to indicate the position of a byte within a block.
The most significant four bits ($(1101)_2 = 13_{10}$) represent the block-frame address, i.e. the number of the block in the lower level of the memory.
Because it is a direct mapped cache, the position of block number 13 in the cache is given by:

(Block-frame address) modulo (number of blocks in the cache)
 = 13 mod 4 = 1

Hence the block number 13 in the lower level of the memory hierarchy will be placed in position 1 into the cache. This is precisely the same as using the last

$\log_2 4 = 2$

bits (01), the index, of the block-frame address (1101).

Figure 8.2 is a graphical representation for this example. Figures 8.1 and 8.3 are graphical representations of the same problem we have in example 8.2 but for fully associative and set associative caches respectively.

Because the cache is smaller than the memory level below it, there are several blocks that will map to the same position in the cache; using the Example 8.2 it is easy to see that blocks number 1, 5, 9, 13 will all map to the same position. The question now is: how can we determine if the block in the memory is the one we are looking for, or not?

8.3 Finding a Block in the Cache

Each line in the cache is augmented with a *tag* field that holds the tag field of the address corresponding to that block. When the CPU issues an address, there are, possibly, several blocks in the cache that could contain the desired information. The one will be chosen that has the same tag as that of the address issued by the CPU.

Figure 8.4 presents the same cache we had in figures 8.1 to 8.3, improved with the tag fields. In the case of a fully associative cache all tags in the cache must be checked against the address's tag field; this because in a fully associative cache blocks may be placed anywhere. Because the cache must be very fast, the checking process must be done in parallel, all cache's tags must be compared at the same time with the address tag fields. For a set associative cache there is less work than in a fully associative cache: there is only one set in which the block can be; therefore only the tags of the blocks in that set have to be compared against the address tag field.

If the cache is direct mapped, the block can have only one position in the cache: only the tag of that block is compared with the address tag field.

There must also be a way to indicate the content of a block must be ignored. When the system starts up for instance, there will be some binary configurations in every tag of the cache; they are meaningless at this moment; however some of them could match the tag of an address issued by the processor thus delivering bad data. The solution is a bit for every cache line, which indicates if that line contains valid data. This bit is called the **valid bit** and is initialized to Non-valid (0) when the system starts up.

Figure 8.5 presents a direct mapped cache schematic; a comparator (COMP) is used to check if the Tag field of the CPU address matches the content of the tag field in the cache at address Index. The valid bit at that address must be Valid (1) to have a hit when the tags are the same. The multiplexor (MUX) at the Data outputs is used to select that part of the block we need.

Figure 8.6 presents the status of a four line, direct mapped cache, similar to the one we had in Example 8.2 after a sequence of misses; suppose that after reset (or power-on), the CPU issues the following sequence of reads at addresses (in decimal notation): 78, 79, 80, 77, 109, 27, 81. Hits don't change the state of the cache when only reads are performed; therefore only the state of the cache after misses is presented in Figure 8.6. Below is the binary representation of addresses involved in the process:

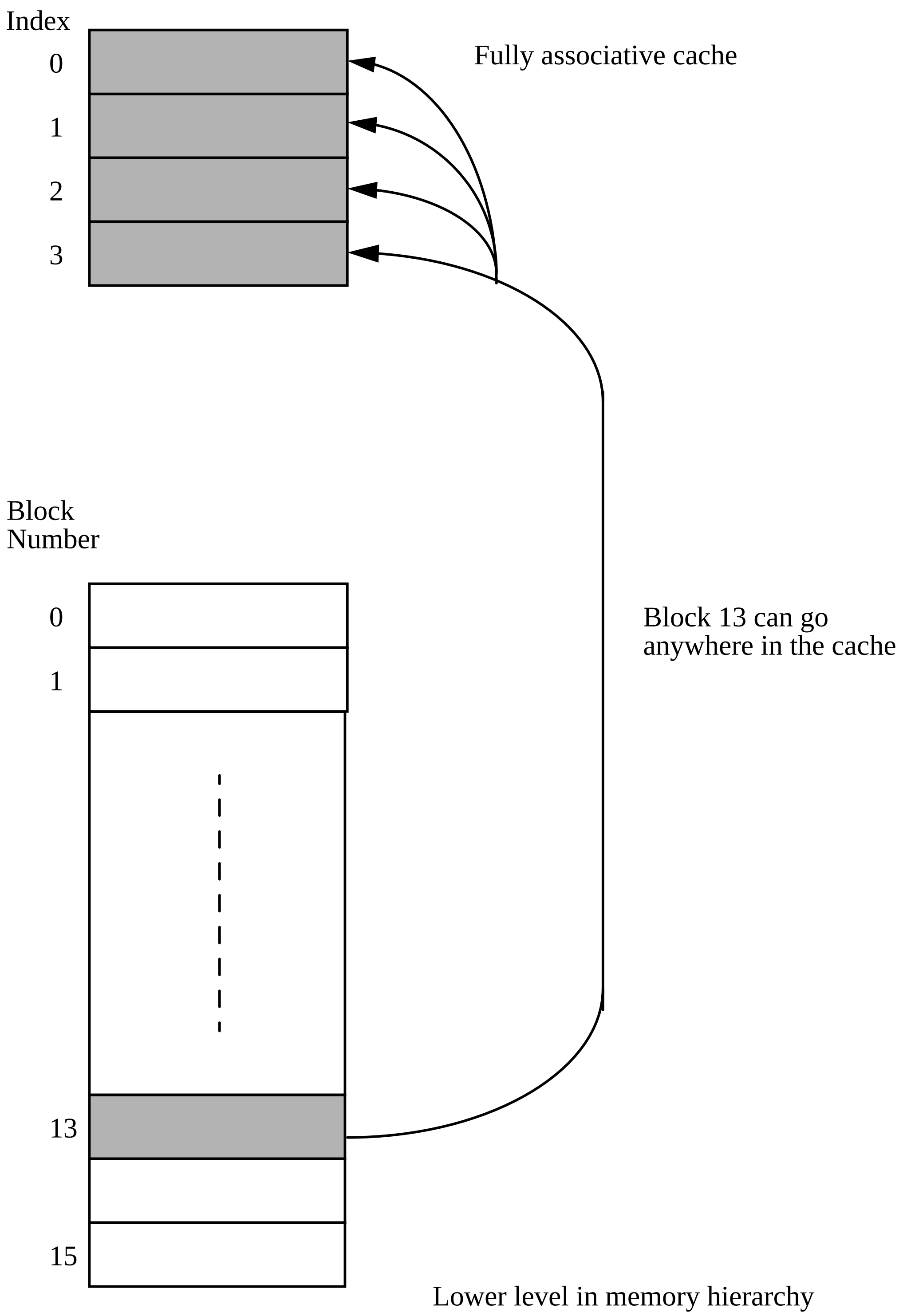

FIGURE 8.1 A fully associative four blocks (lines) cache connected to a 16 blocks.

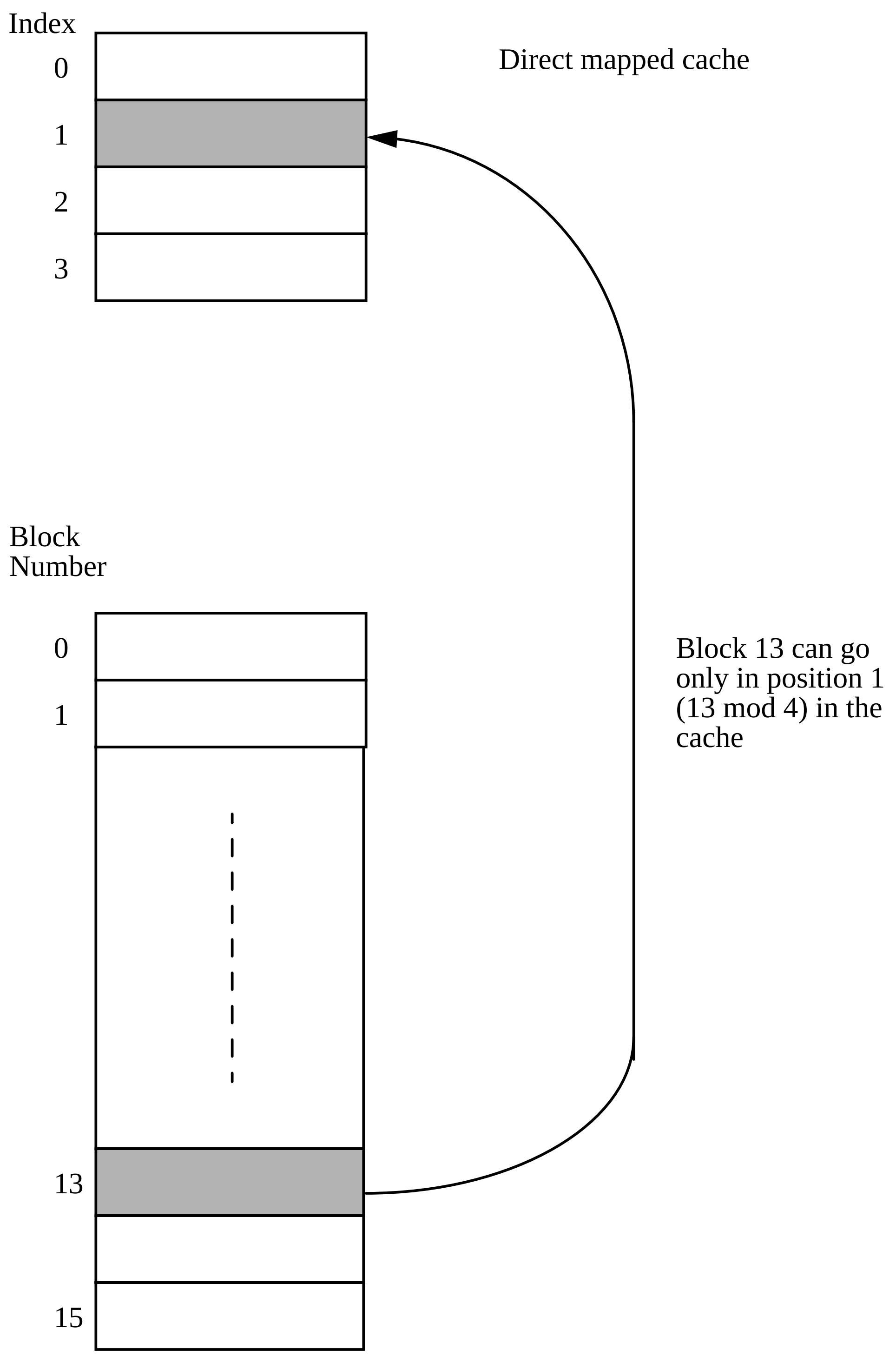

FIGURE 8.2 A Direct mapped, four blocks (lines) cache connected to a 16 blocks memory.

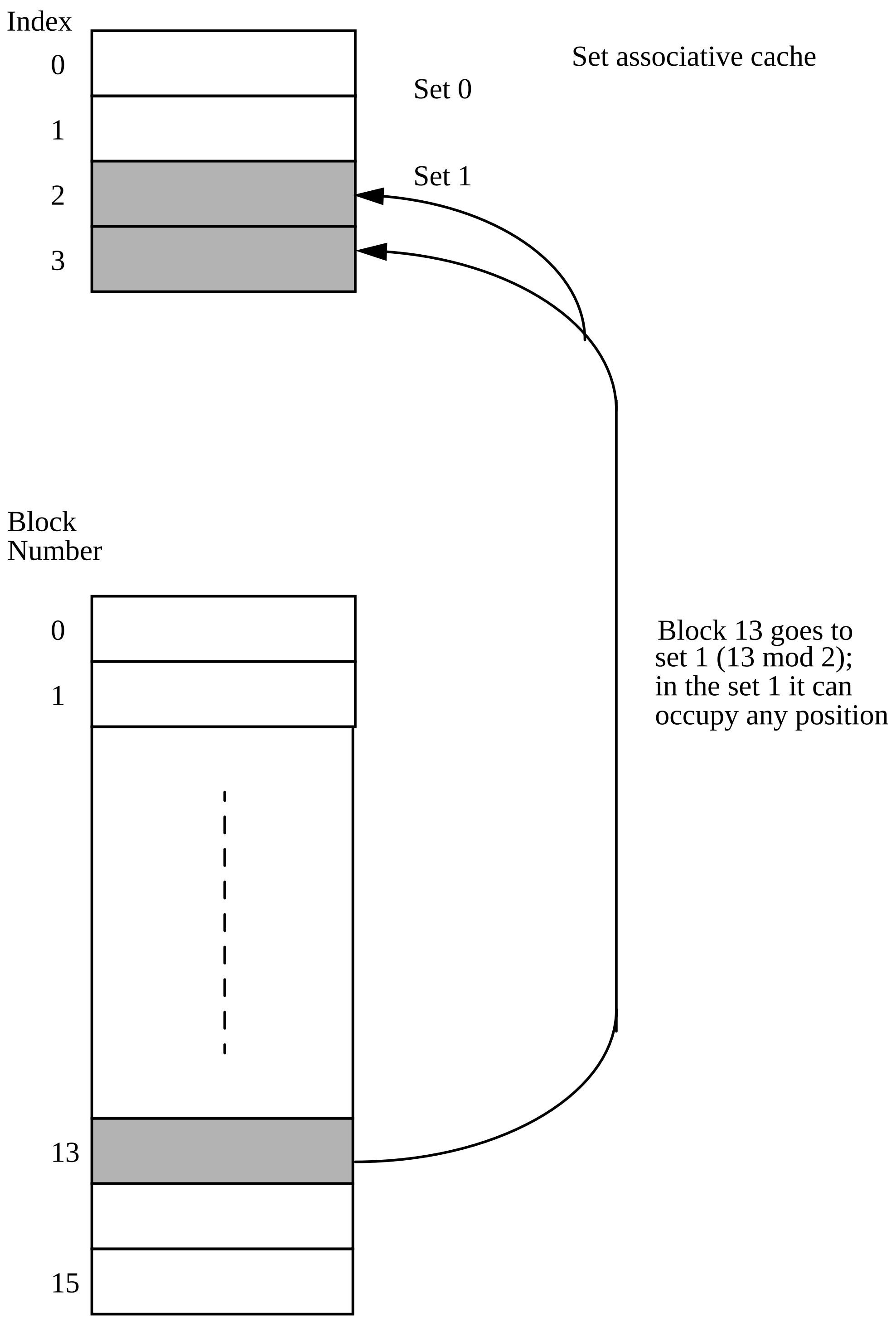
Index
0
1
2
3
Set associative cache
Set 0
Set 1
Block Number
0
1
13
15
Block 13 goes to set 1 (13 mod 2); in the set 1 it can occupy any position

FIGURE 8.3 A 2-way set-associative cache connected to a 16 blocks memory.

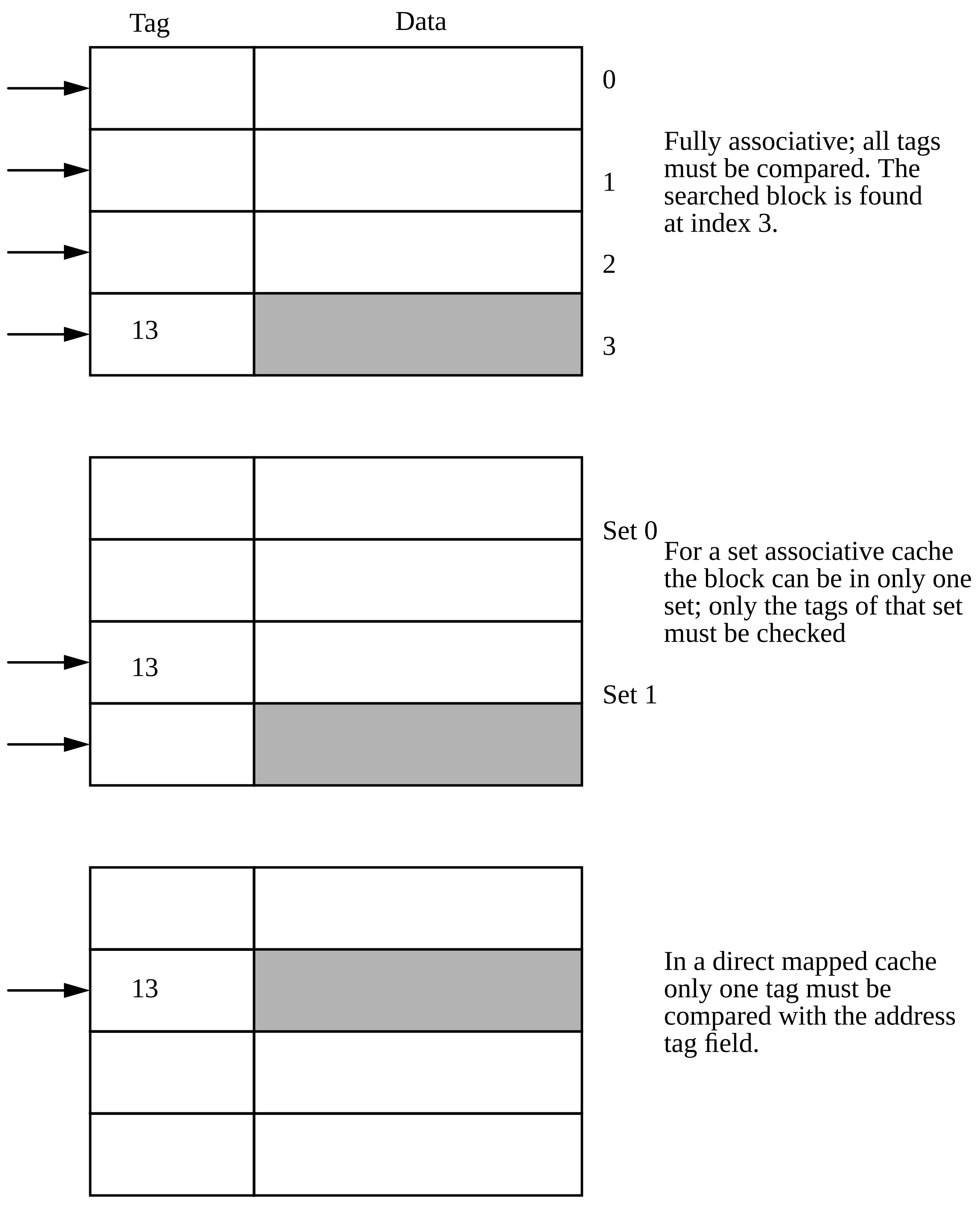
Tag
Data
0
1
2
3
13
Fully associative; all tags must be compared. The searched block is found at index 3.
Set 0
Set 1
13
For a set associative cache the block can be in only one set; only the tags of that set must be checked
13
In a direct mapped cache only one tag must be compared with the address tag field.

FIGURE 8.4 Finding a block in the cache implies comparing the tag field of the actual address with the content of one or more tags in the cache.

Address	Tag	Index	Block offset
78	10	01	110
79	10	01	111
80	10	10	000
77	10	01	101
109	11	01	101
27	00	11	011
81	10	10	001

- Address 78: miss because the valid bit is 0 (Not Valid); a block is brought and placed into the cache in position Index = 01
- Address 79: hit; as Figure 8.6.b points out the content of this memory address is already in the cache
- Address 80: miss because the valid bit at index 10 in the cache is 0 (Not Valid); a block is brought into the cache and placed at this index.
- Address 77: hit, found at index 01 in the cache.
- Address 109: miss; the block being transferred from the lower level of the hierarchy is placed in the cache at index 01, thus replacing the previous block.
- Address 27: miss; block transferred into the cache at index 11.
- Address 81: hit; the item is found in the cache at index 10.

It is a common mistake to neglect the tag field when computing the amount of memory necessary for a cache.

Example 8.3 COMPUTATION OF MEMORY REQUIRED BY A CACHE:

A 16 KB cache is being designed for a 32 bit system. The block size is 16 bytes, and the cache is direct mapped. Which the total amount of memory needed to implement this cache?

Answer:
The cache will have a number of lines equal with:

$$\frac{\text{cache capacity}}{\text{blocksize}} = \frac{16\text{KB}}{16\text{ B}} = 1\text{ KB} = 2^{10}\text{lines}$$

Hence the number of bits in the index field of an address is 10. The tag field in an address is:

32 - 3 - 10 = 19 bits(3 bits are needed as block offset)

Each line in the cache needs a number of bits equal to:

1 + 19 + 16 * 8 = 148 bits

The total amount of memory for the cache is:

line_size * number_of_lines = 148 * 2^{10} = 151.5 Kbit = 18.9 KB roughly

This figure is by 18% larger than the “useful” size of the cache, and is hardly negligible.

8.4 Replacing Policies

Or in other words, answering the question “which block should go out in the case of a cache miss?”. The replacing policy depends upon the type of cache. For a direct mapped cache the decision is very simple: because a block can go in only one place, the block in that position will be replaced. This simplifies the hardware (remember that a cache is hardware managed).

For fully associative and set-associative caches a block may go in several positions (at different indexes), and, as a result, there are different possibilities to choose a block that will be replaced. Note that, due to the high hit rates in the caches (high hit rates are a must for good access times), the decision is painful, with a high probability we will replace blocks that contain useful information.

The most used policies for replacement are:

- **random:** this technique is very simple, one block is selected at random and replaced.
- **LRU** *(Least Recently Used)*: in this approach accesses to the cache are recorded; the block that will be replaced is the one that has been unused (unaccessed) for the longest period of time. This technique is a direct consequence of the temporal locality principle: if blocks tend to be accessed again soon then it seems natural to discard the one that has been of little use in the past.

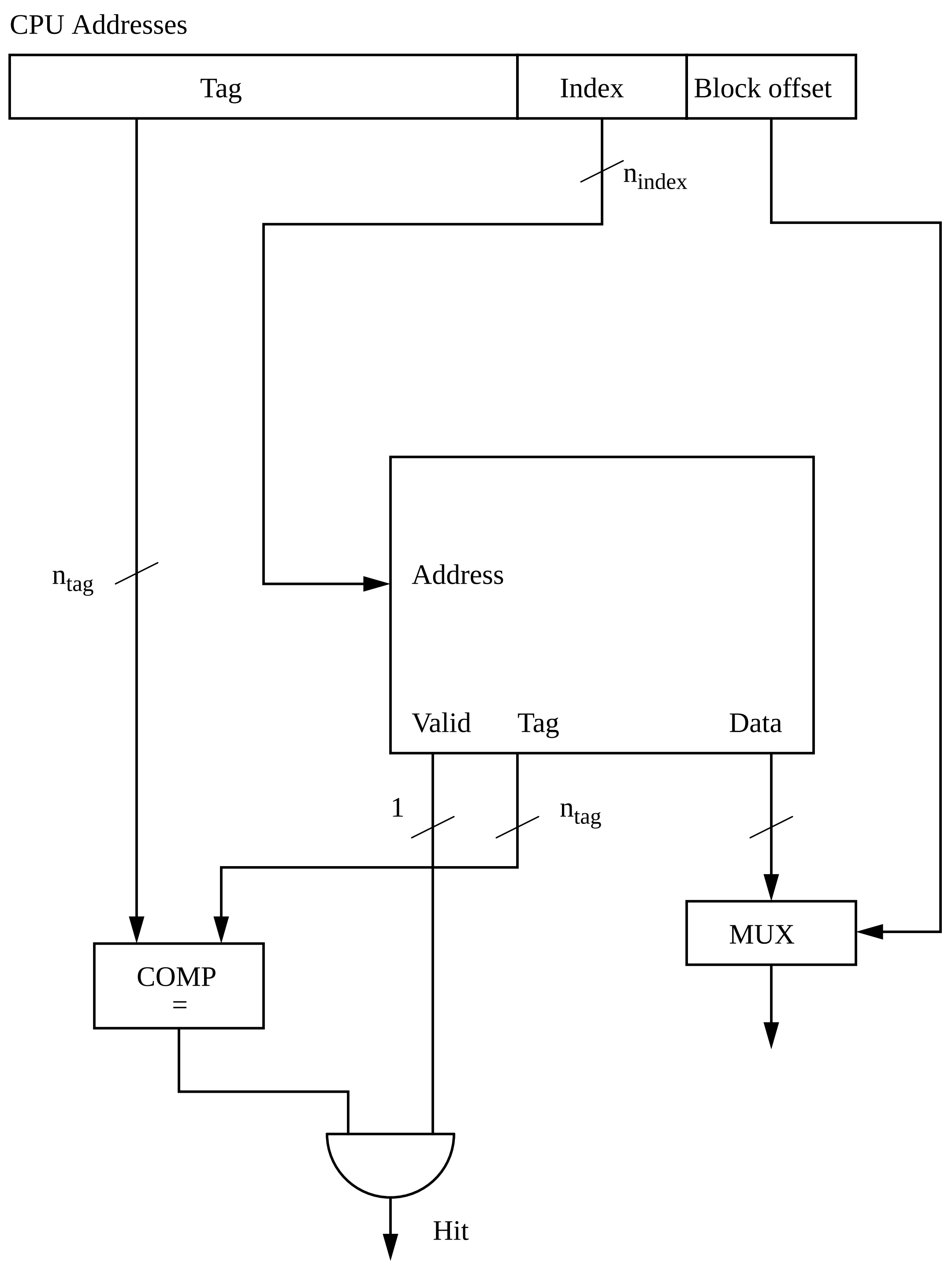

FIGURE 8.5 a direct mapped cache schematic.

Index	V	Tag	Data							
00	0									
01	0									
10	0									
11	0									

a. The initial state of cache after power on. The Tag and Data fields contain some arbitrary binary configurations which are not shown.

Index	V	Tag	Data							
00	0									
01	1	10	M[72]	M[73]	M[74]	M[75]	M[76]	M[77]	$\boxed{M[78]}$	M[79]
10	0									
11	0									

b. After the miss at address 78.

Index	V	Tag	Data							
00	0									
01	1	10	M[72]	M[73]	M[74]	M[75]	M[76]	M[77]	M[78]	M[79]
10	1	10	$\boxed{M[80]}$	M[81]	M[82]	M[83]	M[84]	M[85]	M[86]	M[87]
11	0									

c. After the miss at address 80.

Index	V	Tag	Data							
00	0									
01	1	11	M[104]	M[105]	M[106]	M[107]	M[108]	$\boxed{M[109]}$	M[110]	M[111]
10	1	10	M[80]	M[81]	M[82]	M[83]	M[84]	M[85]	M[86]	M[87]
11	0									

d. After the miss at address 109. The previous block at index 01 has been replaced.

Index	V	Tag	Data							
00	0									
01	1	11	M[104]	M[105]	M[106]	M[107]	M[108]	M[109]	M[110]	M[111]
10	1	10	M[80]	M[81]	M[82]	M[83]	M[84]	M[85]	M[86]	M[87]
11	1	00	M[24]	M[25]	M[26]	$\boxed{M[27]}$	M[28]	M[29]	M[30]	M[31]

e. After the miss at address 27.

FIGURE 8.6 The cache after handling the sequence of addresses: 78 (miss), 79 (hit), 80 (miss), 77 (hit), 109 (miss), 81 (hit).

- **FIFO** *(First In First Out)*: the oldest block in the cache (or in the set for a set associative cache) is selected for replacement. This policy does not take into account the addressing pattern in the past: it may happen the block has been heavily used in the previous addressing cycles, and yet it is chosen for replacement. The FIFO policy is outperformed by the random policy which has, as a plus, the advantage of being easier to implement.

As a matter of fact, almost all cache implementations use either random or LRU for block replacement decision. The LRU policy delivers slightly better performance than random, but it is more difficult to implement: at every access the least recently used block must be determined and marked somehow. For instance, each block could have associated a hardware counter (a software one would be too slow), called *age counter;* when a block is addressed its counter is set to zero, and all other ones are incremented by one. When a block must be replaced, the decision block must find the block with the highest value in its age counter. Obviously the hardware resources are more expensive than for a random policy, and, what is worse, the algorithm is complicated enough to slow down the cache, as compared with a random decision.

Example 8.4 CONTENTS OF A CACHE:

Consider a fully associative four block cache, and the following stream of block-frame addresses: 2, 3, 4, 2, 5, 2, 3, 1, 4, 5, 2, 2, 2, 3. Show the content of the cache in two cases:
a) using a LRU algorithm for replacing blocks;
b) using a FIFO policy.

Answer:

For the LRU replacement policy:

Address:

2	3	4	2	5	2	3	1	4	5	2	2	2	3
2_1	2_2	2_3	2_1	2_2	2_1	2_2	2_3	2_4	5_1	5_2	5_3	5_4	5_5
	3_1	3_2	3_3	3_4	3_5	3_1	3_2	3_3	3_4	2_1	2_1	2_1	2_2
		4_1	4_2	4_3	4_4	4_5	1_1	1_2	1_3	1_4	1_5	1_6	3_1
				5_1	5_2	5_3	5_4	4_1	4_2	4_3	4_4	4_5	4_6
M	M	M		M			M	M	M	M			M

For the FIFO policy:

Address:	2	3	4	2	5	2	3	1	4	5	2	2	2	3
	2*	2*	2*	2*	2*	2*	2*	1	1	1	1	1	1	1
		3	3	3	3	3	3	3*	3*	3*	2	2	2	2
			4	4	4	4	4	4	4	4	4*	4*	4*	3
					5	5	5	5	5	5	5	5	5	5*
	M	M	M		M			M			M			M

For the LRU policy, the subscripts indicate the age of the blocks in the cache. For the FIFO policy a star is used to indicate which is the next block to be replaced. The Ms under the columns of tables indicate the misses.

For the short sequence of block-frame addresses in this example, the FIFO policy yields a smaller number of misses, 7 as compared with 9 for the LRU. However in most cases the LRU strategy proves to be better than FIFO.

8.5 Cache Write Policies

So far we have discussed about how reads are handled in a cache. Writes are more difficult and affect the performance more than reads do. If we take a closer look at the block scheme in Figure 8.5 we realize that, in the case of a read, the two basic operations are performed in parallel: the tag and reading the block are read at the same time. Further, the tags must be compared, and the delay in the comparator (COMP) is slightly higher then the delay through the multiplexor (MUX): if we have a hit then the data is already stable at the cache's outputs; if there a miss there is no harm in reading some improper data from the cache, we simply ignore it.

When we come to writes we realize that the sequence of operations is longer than for a read: the problem is that, for most caches, only a part of the block will be modified; if the block is 16 Bytes wide, and the CPU writes a byte, then only that byte must be changed. This implies a read-modify-write sequence in the cache: read the whole block, modify the needed portion, write the new configuration of the block. Of course the block can not be changed until a hit/miss decision is taken.

There are two options when writing into the cache, depending upon how the information in the lower lever of the hierarchy is updated:

- **write through:** the item is written both into the cache and into the corresponding block in the lower level of the hierarchy; as a result, the blocks in the lower level of the hierarchy contains at every moment the same information as the blocks in the cache;

- **write back:** writes occur only in the cache; the modified block is written into the lower level of the hierarchy only when it has to be replaced.

With the write-back policy there is useless to write back a block (i.e. to write a block into the lower level of the hierarchy) if the block has not been modified while in the cache. To keep track if a block was modified or not, a bit, called the **dirty bit**, is used for every block in the cache; when the block is brought into the cache this bit is set to Not-dirty (0); the first write in that block sets the bit to Dirty (1). When the replacement decision is taken, the control checks if the block is *dirty* or *clean*. If the block is dirty it has to be to the lower level of the memory; otherwise a new block coming from the lower level of the hierarchy can simply overwrite that block in the cache.

For fully or set associative caches, where several bocks may candidate for replacement, it is common to prefer the one which is clean (if any), thus saving the time necessary to transfer a block from the cache to the lower level of the memory.

The two cache write policies have their advantages and disadvantages:

- write through: this is easy to implement, and has the advantage that the memory has the most recent value of data; this property is especially attractive in multiprocessing and I/O. The drawback is that writes going to the lower level in memory are slower. When the CPU has to wait for a write to complete it is said to **write stall**. A simple way to reduce write stalls is to have a **write buffer**. which allows CPU to continue working while the memory is updated; this works fine as long as the rate at which writes occur is lower than the rate at which transfers from the buffer to the memory can be done.

- write back: is more difficult to implement but has the advantage that writes occur at the cache's speed; moreover writes are local to the cache and don't require access to the system bus, unless a dirty block has to be transferred from the cache to the memory. So this write policy uses less memory bandwidth, which is attractive for multiprocessing where several CPUs share the system's resources. Another disadvantage, besides greater hardware complexity, is that read misses may require writes to the memory, in the case a block has to be transferred into the lower level of the hierarchy.

8.6 The Cache Performance

As we discussed very early in this course, the ultimate goal of a designer is to reduce the CPU_{time} for a program. When connected with a memory, we must account both for the execution time of the CPU and for its stalls:

CPU_{time} = (CPU_{exec} + Memory_stalls) * T_{ck}

where both the execution time and stalls are expressed in clock cycles.

Now the natural question we may ask is: do we include the cache access time in the CPU_{exec} or in Memory_stalls? Both ways are possible: it is possible to consider the cache access time in Memory_stalls, simply because the cache is a part of the memory hierarchy. On the other hand, because the cache is supposed to be very fast, we can include the hit time in the CPU execution time as the item sought in the cache will be delivered very quickly, maybe during the same execution cycle. As a matter of fact this is the widely accepted convention.

Memory_stalls will include the stall due to misses, for reads and writes:

Memory_stalls = Mem_accesses_per_program * miss_rate * miss_penalty

We now get for the CPU_{time}:

CPU_{time} = (CPU_{exec} + Mem_accesses_per_program*miss_rate*miss_penalty)*T_{ck}

which can be further modified by factoring the IC (Instruction Count):

CPU_{time} = IC*(CPI_{exec} + Mem_accesses_per_instruction*miss_rate*miss_penalty)*T_{ck}

The above formula can be also written using misses per instruction as:

CPU_{time} = IC*(CPI_{exec} + Misses_per_instruction*miss_penalty)*T_{ck}

Example 8.5 CPU PERFORMANCE WITH CACHE:

The average execution time for instructions in some CPU is 7 (ignoring stalls); the miss penalty is 10 clock cycles, the miss rate is 5%, and there are, on average, 2.5 memory accesses per instruction. What is the CPU performance if the cache is taken into account?

Answer:
CPU_{time} = IC*(CPI_{exec} + Mem_accesses_per_instruction*miss_rate*miss_penalty)*T_{ck}

CPU_{time} (with cache) = $IC*(7 + 2.5*0.05*10)*T_{ck} = IC*8.25*T_{ck}$

The IC and T_{ck} are the same in both cases, with and without cache, so the result of including the cache's behavior is an increase in CPU_{time} by

$$\frac{8.25}{7} - 1 = 17.8\%$$

The following example presents the impact of the cache for a system with a lower CPI (as is the case with pipelined CPUs):

Example 8.6 CPU PERFORMANCE WITH CACHE AND CPI:

The CPI for a CPU is 1.5, there are on the average 1.4 memory accesses per instruction, the miss rate is 5%, and the miss penalty is 10 clock cycles. What is the performance if the cache is considered?

Answer:
$CPU_{time} = IC*(CPI_{exec} + Mem\_accesses\_per\_instruction*miss\_rate*miss\_penalty)*T_{ck}$

CPU_{time} (with cache) = $IC*(1.5 + 1.4*0.05*10)*T_{ck} = IC*2.2*T_{ck}$

This means an increase in CPU_{time} by 46%.

Note that for a machine with lower CPI the impact of the cache is more significant than for a machine with a higher CPI.

The following example shows the impact of the cache on system with different clock rates.

Example 8.7 CPU PERFORMANCE WITH CACHE, CPI AND CLOCK RATES:

The same architecture is implemented using two different technologies, one which allows a clock cycle of 20ns and another one which permits a 10ns clock cycle. Two systems, built around CPUs in the two technologies, use the same type of circuits for their main memories: the miss penalty is, in both cases, 140ns. How does the cache behavior affect the CPU performance? Assume that the ideal CPI is 1.5, the miss rate is 5%, and there are 1.4 memory accesses per instruction on average.

Answer:
$CPU_{time} = IC*(CPI_{exec} + Mem\_accesses\_per\_instruction*miss\_rate*miss\_penalty)*T_{ck}$

For the CPU running with a 20ns clock cycle, the miss penalty is 140/20 = 7 clock cycles, and the performance is given by:

$CPU_{time1} = IC*(1.5 + 1.4*0.05*7)*T_{ck1} = IC*1.99*T_{ck1}$

The effect of the cache, for this machine, is to stretch the execution time by 32%. For the machine running with a 10 ns clock cycle, the miss penalty is 140/10 = 14 clock cycles, and the performance is:

$CPU_{time2} = IC*(1.5 + 1.4*0.05*14)*T_{ck2} = IC*2.48*T_{ck2}$

The cache increases the CPU_{time}, for this machine, by 65%.

Example 8.7 clearly points out that the cache behavior gets more important while CPU are running faster. Neglecting the cache may completely compromise the performance of a CPU. For a given instruction set and a specified program the CPI_{exec} can be measured; the Instruction Count can also be measured, and T_{ck} is known for the given machine. Reducing the CPU_{time} can be achieved by:

- reducing the miss rate: the easy way is to increase the cache size; however there is a serious limitation in doing so for on-chip caches: the space. Most on-chip caches are only a few kilobytes in size.

- reducing the miss penalty: for most cases the access time dominates the miss penalty; while the access time is given by the technology used for memories, and, as a result can not be easily lowered, it is possible to use intermediate levels of cache between the internal cache (on-chip) and main memory.

Here is a short description of internal caches for several popular CPUs:

CPU	Instruction	Data
Intel 80486	8 KB	
Motorola 68040	4 KB	4 KB
Intel PENTIUM	8 KB	8 KB
DEC Alpha	8 KB	8 KB
Sun MicroSPARC	4 KB	2 KB
Sun SuperSPARC	20 KB	16 KB
Hewlett-Packard PA 7100	-	-
MIPS R4000	8 KB	8 KB
MIPS R4400	16 KB	16 KB
PowerPC 601	32 KB	

8.7 Sources for Cache Misses

Misses in a cache can have one of the three following sources:

- **compulsory:** when the program starts running the cache is empty (no block for that program yet);

- **capacity:** if the cache does not contain all the blocks needed for the execution of the program, then some blocks will be replaced and then, later, brought back into the cache;

- **conflict:** this happens in direct mapped and set associative caches if too many blocks map to the same position.

There is little to do against compulsory misses: increasing the block size reduces indeed the number of compulsory misses as the cache will be filled faster; the drawback is that bigger blocks may increase the number of conflict misses as there are fewer blocks in the cache.

Conflict misses seem to be easiest to resolve: a fully associative cache has no conflicts. However full associativity is very expensive in terms of hardware: more hardware tends to slow down the clock, yielding an overall poorer performance.

As for capacity misses, the solution is larger caches, both internal and external. If the cache is too small to fit the requirement of some program, then most of the time will be spent in transferring blocks between the cache and the lower level of the hierarchy; this is called **trashing.** A trashing memory hierarchy has a performance that is close to that of the memory in the lower level, or even poorer due to misses overhead.

8.8 Unified Caches or Instruction/Data Only?

Initial caches were meant to hold both data and instructions. This caches are called **unified** or mixed. It is possible however to have separate caches for instructions and data, as the CPU knows if it is fetching an instruction or loading/storing data. Having separate caches allows the CPU to perform an instruction fetch at the same time with a data read/write, as it happens in pipelined implementations. As the table in section 8.6 shows, most of the today’s architectures have separate caches. Separate caches give the designer the opportunity to separately optimize each cache: they may have different sizes, different organizations, and block sizes. The main observation is that instruction caches have lower miss rates as data caches, for the main reason that instructions expose better spatial locality than data.

Exercises

8.1 Draw a fully associative cache schematic. Which are the hardware resources besides the ones required by a direct mapped cache? You must pick some cache capacity and some block size.

8.2 Redo the design in problem 8.1 but for a 4-way set associative cache. Compare your design with the fully associative cache and the direct mapped cache.

8.3 Design a 16 KB direct mapped cache for a 32 bit address system. The block size is 4 bytes (1 word). Compare the result with the result in Example 8.3.

8.4 Design (gate level) a 4 bit comparator. While most MSI circuits provide three outputs indicating the relation between the A and B inputs (A > B, A = B, A < B), your design must have only one output which gets active (1) when the two inputs are equal.

8.5 Assume you have two machines with the same CPU and same main memory, but different caches:

cache 1: a 16 set, 2-way associative cache, 16 bytes per block, write through;

cache 2: a 32 lines direct mapped cache, 16 bytes per block, write back.

Also assume that a miss takes 10 longer than a hit, for both machines. A word write takes 5 times longer than a hit, for the write through cache; the transfer of a block from the cache to the memory takes 15 times as much as a hit.

a) write a program that makes machine 1 run faster than machine 2 (by as much as possible);

b) write a program that makes machine 2 run faster than machine 1 (by as much as possible).

9. The Memory Hierarchy (3)
Main Memory

Main memory is the name given to the level below the cache(s) in the memory hierarchy. There is a large variety of dimensions, but a smaller one in speed due to the fact that vendors use the same chips to build memory arrays. A main memory may have a few MBytes for a typical Personal Computer, tens to hundreds of MBytes for a workstation, hundreds of MBytes to GBytes for supercomputers. The capacity of main memory has continuously increased over the years, as prices have dramatically dropped. The main memory must satisfy the cache requests as quickly as possible (the main memory should have a low latency), and must provide sufficient bandwidth for I/O devices and for vector units (if it is the case).

The **access time**, is defined as the time between the moment the read command is issued and the moment the requested data is at outputs.
The **cycle time** is defined as the minimum time between successive accesses to memory. The cycle time is usually greater than the access time.

9.1 DRAM/SRAM

To access data in a memory chip with a capacity of NxM bits, one must provide a number of addresses equal to:

$\log_2 N$

N is the number of “words” each chip has; each “word” is M bits wide. As the technology improved, the packaging costs become a real concern, as the number of address lines got greater and greater.

Example 9.1 ADDRESS LINES:

Which is the number of address lines needed for a 4 Mbit memory chip:
a) organized as 4Mx1;
b) organized as 1Mx4?

Answer:
a) 4 M = 2^{22} hence the number of address lines is
$\log_2 2^{22} = 22$

b) 1 M = 2^{20}therefore the number of address lines is
$\log_2 2^{20} = 20$

To keep memory chips cheap, the solution adopted for Dynamic RAM (DRAM) integrated circuits was to multiplex the address, thus reducing the number of pins for addresses to half, and adding two new control lines: **RAS** (Row Address Strobe), which loads into an internal buffer half of the address supplied by the control on the address lines, and **CAS** (Column Address Strobe) which handles the second half of the address.

Example 9.2 NUMBER OF PINS:

How many pins has a 1Mx1 memory chip:
a) in DRAM technology;
b) in SRAM technology?

Answer:
Organization is important because it says how many pins are needed for data input and data output lines; it still does not say everything about the chip, more precisely it does not say if the input and output lines are the same or are separate.

Let's suppose that the chip in this example has one input line and one output line.

a) 1 M = 2^{20}
$n_A = \log_2 2^{20} = 20$ addresses

a)For DRAM the number of address lines (pins) is half of the address size:

10	address lines	
1	RAS	
1	CAS	
1	WE	(Write Enable)
1	Din	(the data input line)
1	Dout	(the data output line)
2	for power supply	

Total 17 pins needed for 1Mx1 DRAM. A real circuit has 18 pins, with one pin unused but devoted to use as an address line in the 4Mx1 chips.

b)For SRAM the number of address lines is the same as the address size:

20	address lines
1	WE
1	Din
1	Dout
2	for power supply

Total 25 pins needed for a 1Mx1 SRAM which fits in a 26 pin chip.

The reason for which SRAM address lines are not multiplexed is **speed;** the package is however more expensive than the package for a DRAM with the same capacity.

Another problem the designer faces, when using DRAM circuits is the refresh: each row in a DRAM circuit has to be accessed within some time interval (say 2 milliseconds), to prevent data from getting lost. This is a consequence of the dynamic technology, where data is stored as electric charge in a small capacitor: due to unavoidable losses in the dielectric of the capacitor, the charge decreases in time; the purpose of refresh is to periodically restore the charge on the capacitors (a charged capacitor stores a 1) thus preserving data.

The refresh requirement implies that the memory will be sometimes unavailable, being busy to do refresh: usually it takes a memory cycle to do a refresh to one row in DRAM. The number of refresh cycles in the critical time interval is a circuit specification, and can be found in the circuit's data sheet. Multiplexed address also mean an involved timing for DRAM memories; the cycle time is always larger than the access time.

SRAM use more transistors per memory-bit to prevent the loss of data. The cycle time for a static memory is very close to the access time, and is in the range of nanoseconds to tens of nanoseconds.

With the today's technology the maximum capacity of DRAMs is 16 times larger than that of SRAMs; the cycle time for a SRAM is, on the other hand, 8 to 16 times shorter than that of a DRAM.

As we saw in chapter 8, the faster a CPU is, the higher the miss penalty is. DRAM capacity has increased, over the last decade, quadrupling every three years. Unfortunately this is not the case with the DRAM performance: the 64 Kbit circuit, introduced in 1980, had a cycle time of 250 ns, while the 4 Mbit circuit introduced in 1989 has a cycle time of 165 ns; the capacity is 64 times larger but the performance is only 51% better (figures for the access time are quite similar).

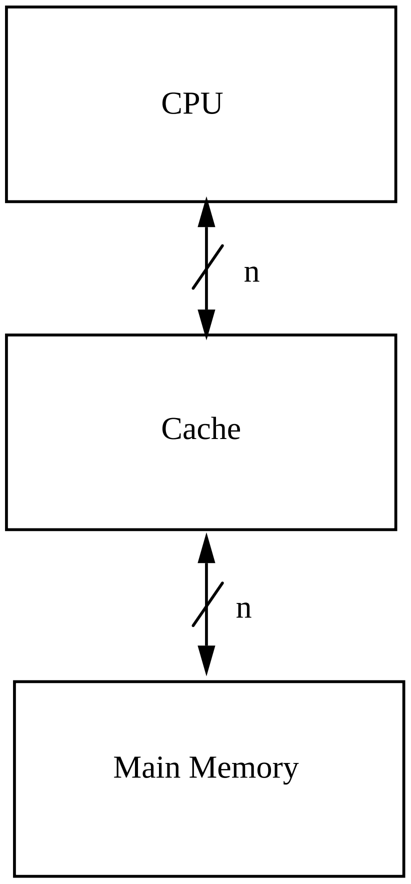

FIGURE 9.1 Connecting the Main memory to the CPU and cache; all buses have the same width.

9.2 Possible Organizations for Main Memory

We shall compare different memory organizations based on the following assumptions:

- all transfers are multiple of word (1 word = 4 bytes);
- 1 clock cycle to send the address;
- 10 clock cycles for the access time;
- 1 clock cycle for a bus transfer of the accessed item.

The simple and cheap approach for memory organization is to have transfers, between all levels of the memory hierarchy, the same size, as depicted in Figure 9.1.

Example 9.3 MEMORY ORGANIZATION:

Compute the miss penalty and the memory bandwidth for a word organized memory system. The cache block size is 8 words (32 bytes).

Answer:
For each word in the block the address must be transmitted (1 clock cycle), a fixed amount of time has to be spent waiting (10 clock cycles), and each word has to be transferred into the cache (1 clock cycle); therefore the miss penalty is:

miss_penalty = 8*(1 + 10 + 1) = 96 clock cycles

The memory bandwidth is:

$$\text{memory\_bandwidth} = \frac{\text{bytes\_transferred}}{\text{clock\_cycles}}$$

$$\text{memory\_bandwidth} = \frac{32}{96} = 0.33 \text{ bytes/clock cycle}$$

There are two parameters that can be modified to obtain a larger memory bandwidth: to increase the number of bytes transferred in the same amount of time, and to decrease the number of clock cycles necessary to complete a block transfer. The two possibilities correspond to two different memory organizations: a wider memory and interleaved memory respectively.

Wider Main Memory

The basic organization of a wider memory is presented in Figure 9.2. The data bus between the cache and main memory is wider than the bus between the cache and the CPU (the size of this one is the size of the

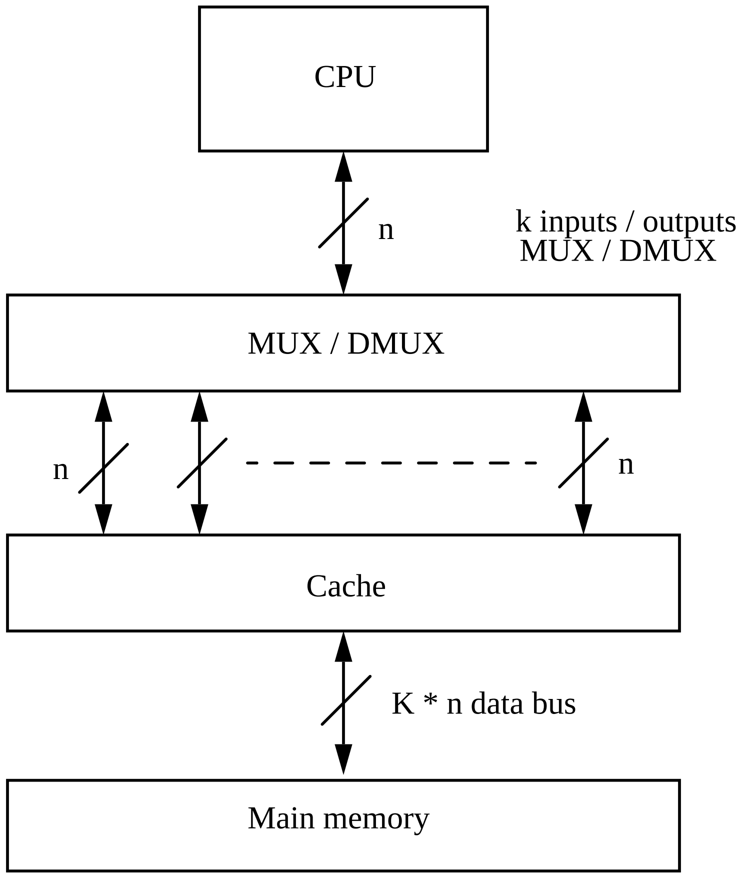

FIGURE 9.2 Wide memory organization. k * n bits are transferred between main memory and the cache, but only n bits between CPU and the cache.

datapath). In this case the address no longer indicates a word in main memory but a block; it is, in other words, the block-frame address that comes to main memory.

Example 9.4 MISS PENALTY AND MEMORY BANDWIDTH:

Calculate the miss penalty and the bus bandwidth for a wide memory organization. The cache line is 8 words wide, and the data bus is also 8 words wide.

Answer:
It takes one clock cycle to transmit the address, 10 clock cycles to access a line in the memory (8 words are accessed at once), and one more clock cycle to transfer the line into the cache; hence, the miss penalty is:

miss_penalty = 1 + 10 + 1 = 12 clock cycles

and the memory bandwidth is:

$$\text{memory\_bandwidth} = \frac{\text{bytes\_transferred}}{\text{clock\_cycles}}$$

$$\text{memory\_bandwidth} = \frac{32}{12} = 2.67 \text{ bytes/clock cycle}$$

The memory bandwidth is 8 times larger than for the word organized memory.

There is a price to be paid for better performance: because the CPU reads only n bits at a time and the cache is n*k bits organized, there must be a multiplexer between the cache and the CPU. Incidentally this is the case with cache represented in Figure 8.5 where we have a multiplexer that selects the proper word from the block (note also that, if the memory is byte addressable the scheme gets more involved, in that the type of item being addressed must be stated).

Another problem with the wide memories is related to the price paid by the customer: if the chips available for expansion have a NxM capacity, then the number of circuits the user must add into the system is a multiple of:

n*k / M

because the user has to add complete lines to the memory for expansion.

Interleaved Memory

The memory can be organized in banks, each bank one word wide, such that transfers between the cache and main memory are word wide, but several words can be read at once and then transferred one after the other to the cache. Figure 9.3 presents a interleaved memory organization.

Having an interleaved memory there is only one address that the CPU has to supply to main memory: this address will access a word in the bank number:

(address) modulo (number of banks)

while the other banks will access words at addresses successive to the address issued by the CPU.

Example 9.5 MISS PENALTY AND MEMORY BANDWIDTH:

Compute the miss penalty and the memory bandwidth for a 4 bank main memory; each bank is one word wide. The cache is 8 words wide.

Answer:
Because the memory has 4 banks, there will be 4 words read in a single burst; this takes one clock cycle to send the address, 10 clock cycles waiting time for the memories to access date, and 4 clock cycles to read the four word coming from the four banks. Since the cache block is 8 words wide this process has to be repeated; therefore, the miss penalty is:

miss_penalty = 2*(1 + 10 + 4*1) = 30 clock cycles

and the memory bandwidth is:

$$\text{memory\_bandwidth} = \frac{\text{bytes\_transferred}}{\text{clock\_cycles}}$$

$$\text{memory\_bandwidth} = \frac{32}{30} = 1.1 \text{ bytes/clock cycle (roughly)}$$

Interleaving gives fairly good performance as compared with a word organized main memory, still having the advantage that it does not require a wider data bus. The manner in which addresses are mapped to banks affect the memory's behavior; mapping word addresses to banks, with banks word wide, is a natural solution for the today's 32 bit machines that access the most frequent words in the memory.

An interleaved memory behaves fine in the case of cache misses due to the fact that words are read sequentially, and transferred one at a time to the cache; it is also attractive for write-back caches, as the words in a block are written sequentially into the memory, with the price of a single access time.

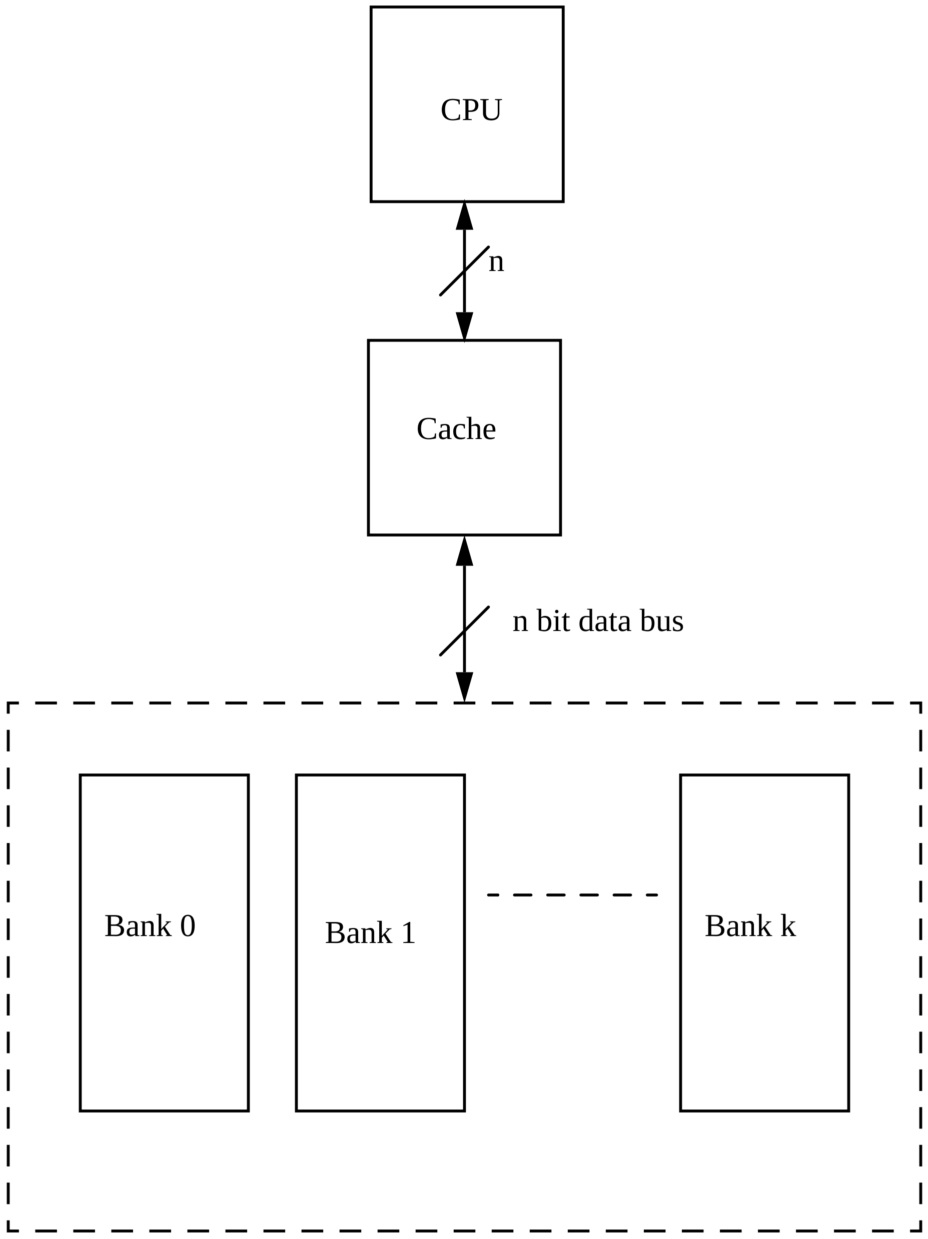

FIGURE 9.3 Interlaced memory organization. CPU, cache and each memory bank have the same width.

There is however the same drawback as for the wide memory organization, the price that the user must pay to increase the system's memory capacity. In this case the user must add the same number of chips in each bank, for a memory upgrade. Moreover, as the memory chips get larger capacities, it is more difficult to organize the chips in banks, as the following example suggests.

Example 9.6 MEMORY CAPACITY AND NUMBER OF CHIPS:

The maximum memory a PC can address is 16 MBytes; you have to design a four bank interleaved memory for this system. Each bank is byte organized. How many chips are necessary, using:
a) 1Mx1 chips;
b) 4Mx1 chips;
c) 16Mx1 chips?

Answer:
The capacity of a bank is:

$$\text{bank\_capacity} = \frac{\text{memory\_capacity}}{\text{number\_of\_banks}}$$

$$\text{bank\_capacity} = \frac{16\text{MB}}{4} = 4\text{MB/bank}$$

a)

$$n_a = (4 \text{ banks}) * \left(\frac{4\text{Mx}8}{1\text{Mx}1}\text{circuits per bank}\right) = 4 * 32 = 128 \text{ circuits}$$

b)

$$n_b = (4 \text{ banks}) * \left(\frac{4\text{Mx}8}{4\text{Mx}1}\text{circuits per bank}\right) = 4 * 8 = 32 \text{ circuits}$$

c)

$$n_a = (4 \text{ banks}) * \left(\frac{4\text{Mx}8}{16\text{Mx}1}\text{circuits per bank}\right) = 4 * 2 = 8 \text{ circuits ??}$$

We are in trouble because we need at least 8 circuits per bank to ensure the proper number of inputs and outputs (using 16Mx1 circuits); using 8 such circuits per bank means a capacity of:

8 * 16Mx1 = 16 MByte/bank

If the system can access only 16 MB, it results that 3/4 of the main memory is inaccessible.

Given the actual conditions in which the CPU performance increases at a faster pace than the memory performance, memory organizations that reduce the cache penalty tend to become common place.

Exercises

9.1 A memory hierarchy is being designed for a system. The following possibilities have to be investigated:
a) cache block size is 1 word, miss rate is 20%
b) cache block size is 4 words, miss rate is 10%
c) cache block size is 8 words, miss rate is 2%
In every case there are 1.4 memory accesses per instruction. Which is the best choice for the main memory? State your assumptions.

9.2 Explore the possibility of using some of the features the new DRAM circuits offer, to improve the memory performance: here are some of the standard DRAM improvements you might want to consider:
a) nibble mode;
b) page mode;
c) static column.

9.3 A new idea being studied is to move the cache closer to the memory, more precisely on the same memory chip die; this is tempting because, in the case of read, a whole row is accessed: for a 1Mx1 DRAM memory a row is 1024 bits wide (supposing the die is square). How do you think could this improve the memory performance? For a good introduction in this, you could consider a series of articles in the IEEE Spectrum, October 1992.

10. Virtual Memory

In a typical memory hierarchy for a compute there are three levels: the cache, the main memory and the external storage, usually the disk. So far we have discussed about the first two levels of the hierarchy. the cache and the main memory; as we saw the cache contains and provides fast access to those parts of the program that are most recently used. The main memory is for the disk, what the cache is for the main memory: the programmer has the impression of a very fast, and very large memory without caring about the details of transfers between the two levels of the hierarchy.

The fundamental assumption of **virtual memory** is that programs do not have to entirely reside in main memory when executed, the same way a program does not have to entirely fit in a cache, in order to run.

The concept of virtual memory has emerged primarily as a way to relieve programmer from the burden of accommodating a large program in a small main memory: in the early days the programmer had to divide the program into pieces and try to identify those parts that were mutually exclusive, i.e. they had not to be present at the same moment in memory; these pieces of code, called *overlays*, had to be loaded in the memory under the user program control, making sure at any given time the program does not try to access an overlay not in memory.

Virtual memory, automatically manages the transfer of pieces of code/

data between the disk and main memory and conversely.

At any moment of time, several programs (processes) are running in a computer; however it is not known at compile time, which will be the other programs with which the compiled program will be running in a computer. Of course we discuss here about systems that allow multiprogramming, which is the case with most of the today's computers: a good image for this is any IBM-PC running Windows, where several applications may be simultaneous active. Because several programs must share, together with the operating system, the same physical memory, we must somehow ensure **protection**, that is we must make sure each process is working only in its own address space, even if they share the same physical memory.

Virtual memory also provides **relocation** of programs; this means that a program can be loaded anywhere in the main memory. Moreover, because each program is allocated a number of blocks in the memory, called pages, the program has not to fit in a single contiguous area of the main memory. Usually virtual memory reduces also the time to start a program, as not all code and data has to be present in the memory to start; after the minimum amount has been brought into main memory, the program may start.

10.1 Some Definitions

The basic concepts of a memory hierarchy apply for the virtual memory (the main memory secondary storage levels of the hierarchy). For historical reasons the names we are using, when discussing about virtual memory, are different:

- **page** or **segment** are the terms used for block;
- **page fault** is the term used for a miss.

The name *page* is used for fixed size blocks, while the name *segment* is used for variable size blocks.

	Minimum	Maximum
Page size	512 Bytes	16 Kbytes
Segment size	1 Byte	64 KB - 4 MB

Until now we have not been concerned about addresses: we did not make any distinction between an addressed item and a physical location in memory. When dealing with virtual memory we must make a clear distinction between the address space of a system (the range of addresses the architecture allows), and the physical memory location in a given hardware configuration:

- the **virtual address** is the address the CPU issues;
- the **physical address** results from translating the virtual address (using a hardware and software combination), and can be used to access the main memory.

The process of translating virtual addresses into physical addresses (also named real addresses) is called *memory mapping*, or *address translation*.

The virtual address can be viewed as having two fields, the virtual page/segment number and the offset inside the page/segment. In the case the system uses a paged virtual memory things are simple, there a fixed size field corresponding to offset and the virtual page number is also of fixed size; with a larger page size the offset field gets larger but, in the mean time fewer pages will fit in the virtual address space, and the page number field gets smaller: the sum of sizes of the two fields is constant, the size of an address issued by the CPU.

Virtual page number	Page offset
MSB	LSB

Obviously the number of virtual pages has not to be the same with the number of pages that fit in the main memory, and, as a matter of fact, usually there much more virtual pages than physical ones.

Example 10.1 VIRTUAL PAGE AND OFFSET:

A 32 bit address system, uses a paged virtual memory; the page size is 2 KBytes. What is the virtual page and the offset in the page for the virtual address 0x00030f40?

Answer:
For a page size of N Bytes the number of bits in the offset field is $\log_2 N$. In the case of a 2 KBytes page there are:

$\log_2 2^{11} = 11$ bits

Therefore the number of bits for the page number is:

$32 - 11 = 21$

which means a total of $2^{21} = 2$ Mpages. The binary representation of the address is:

0000 0000 0000 0011 0000 1	111 0100 0000
31 11	10 0

The given virtual address identifies the virtual page number 0x61 = 97_{10}; the offset inside the page is 0x740 = 1856_{10}.

When the virtual memory is segmented, the two fields of the virtual address have variable length; as a result the CPU must issue two words for a complete address, one which specifies the segment, and the other one specifying the offset inside the page. It is therefore important to know from an early stage of the design if the virtual memory will be paged or segmented as this affects the CPU design. A paged virtual memory is also simpler for the compiler.

Another drawback of segmented virtual memory is related to the replacement problem: when a segment has to be brought into main memory the operating system must find an unused, contiguous area in memory where the segment fits. This is really hard as compared with the paged approach: all pages have the same size, and it is trivial to find space for the page being brought.

The main differences between the cache-main_memory and the main_memory_disk can be resumed as follows:

- caches are hardware managed while the virtual memory is software managed (by the operating system). As the example 7.5 points out, the involvement of the operating system in the replacement decision, adds very little to the huge disk access time.

- the size of the cache is independent of the address size of the architecture, while for the virtual memory it is precisely the size of the address which determines the its size.

- besides its role as the bottom level of the memory hierarchy, the disk also holds the file system of the computer.

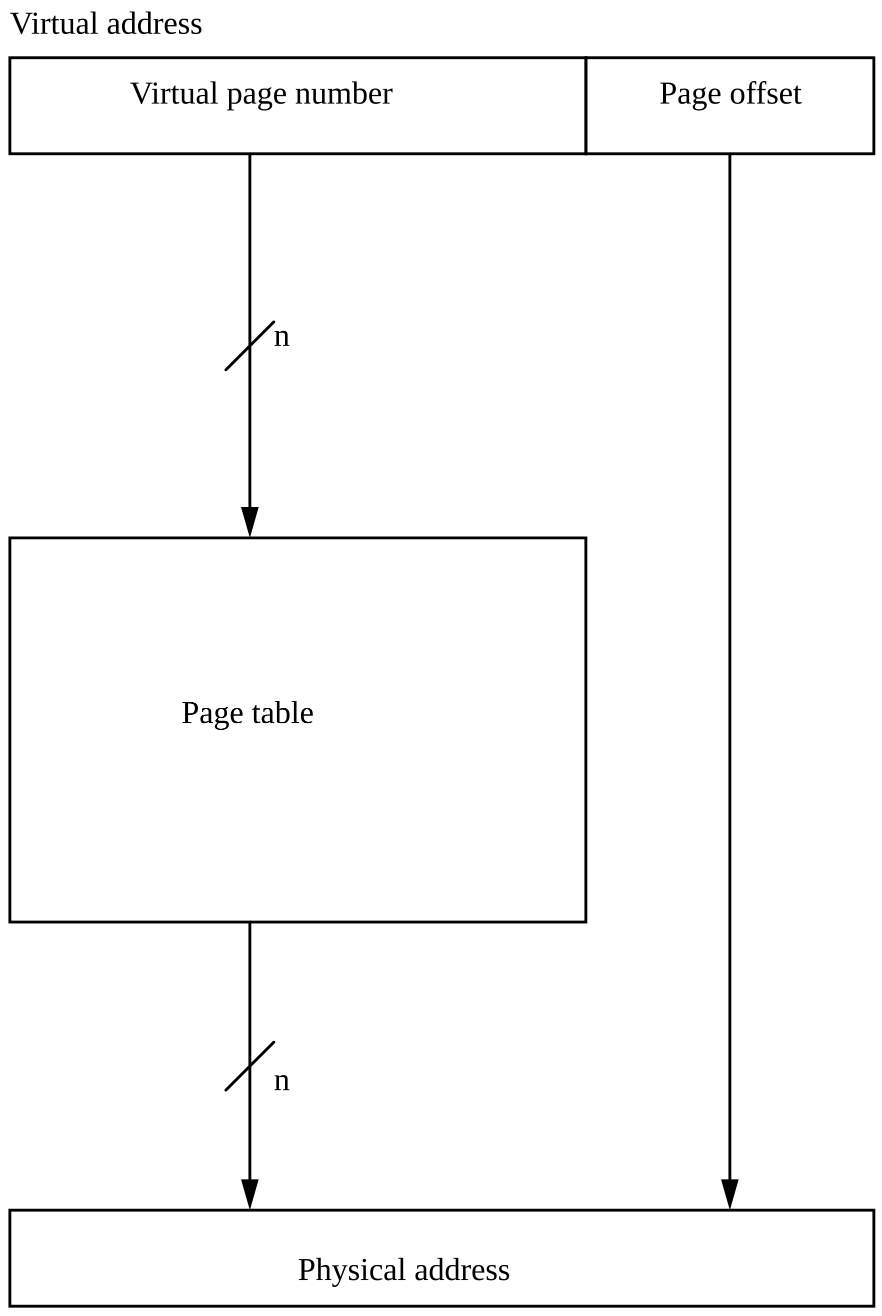

FIGURE 10.1 For a paged virtual memory the page's physical address (at outputs of page table) is concatenated with the page offset to form the physical address to the main memory.

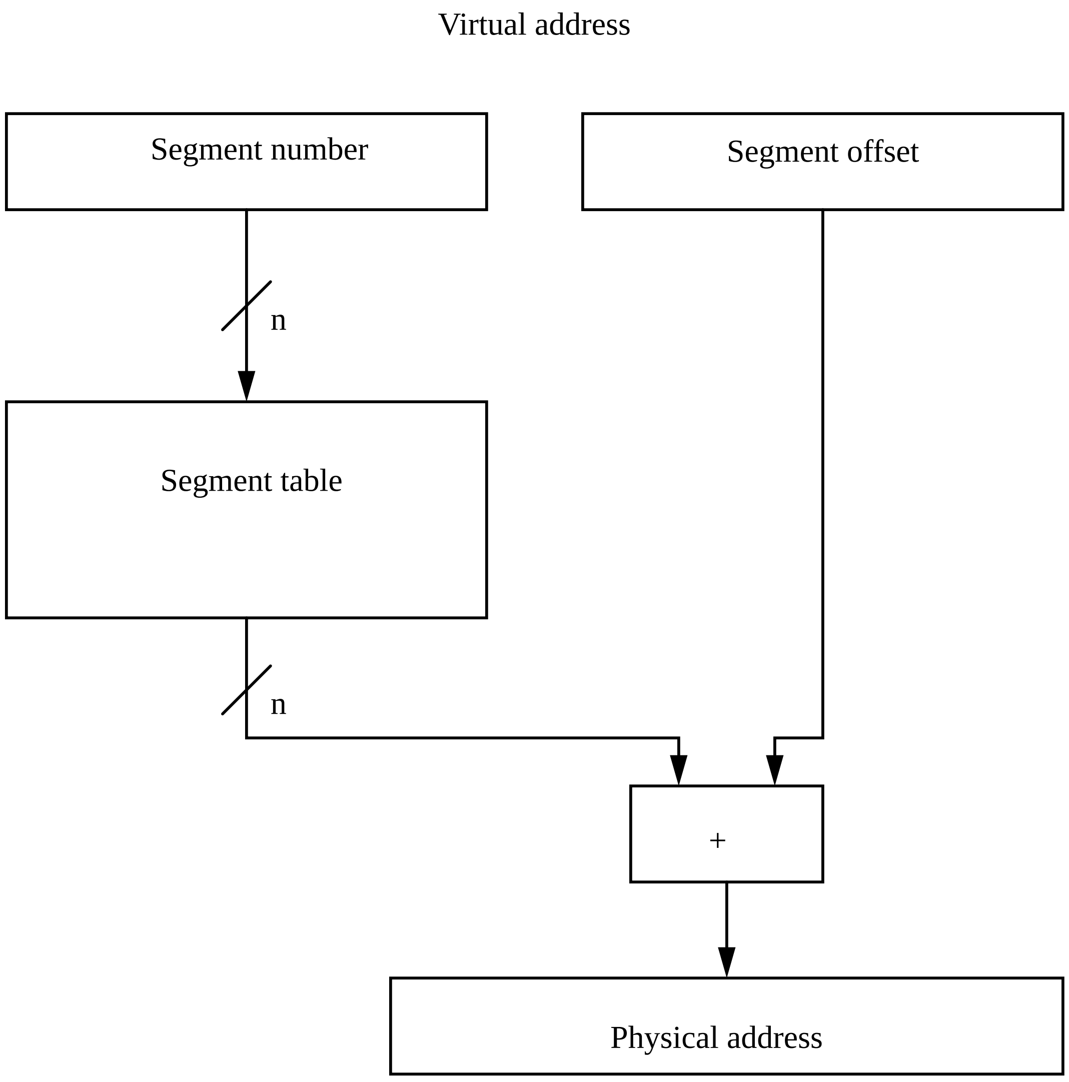

FIGURE 10.2 For a segmented virtual memory the offset is added to the segment's physical address to form the address to the main memory (the physical address).

10.2 How Virtual Memory Works

As with any memory hierarchy, the same basic questions have to be answered.

Where are blocks placed in main memory

Due to the huge miss penalty, the design of the virtual memory focuses in reducing the miss rate. The block placement is a good candidate for optimization. As we saw when discussing about caches, an important source for misses are the conflicts in direct mapped and set associative caches; a fully associative cache eliminates these conflicts. While fully associative caches are very expensive and slow down the clock rate, we can implement a fully associative mapping for the virtual memory. That means any virtual page can be placed anywhere in the physical memory. This is possible because it is in software where misses are handled.

Is the block in the memory or not?

A table ensures the address translation. This table is indexed with the virtual page/segment number to find the start of the physical page/segment in main memory. There is a difference between paging and segmentation:

- paging: the full address is obtained by concatenating the page's physical address and the page offset, as in Figure 10.1

- segmentation: the full address is obtained by adding the segment offset to the segment's physical address, as in Figure 10.2

Each program has its own **page table**, which maps virtual page numbers to physical addresses (a similar discussion can be made about segmentation, but we shall concentrate on paging). This table resides in main memory, and the hardware must provide a register whose content points to the start of the page table: this is the **page table register**, and in our machine can be one of the registers in the general purpose set, restricted to be used only for this purpose.

As it was the case with caches, each entry in the page table contains a bit which indicated if the page corresponding to that entry is in the main memory or not. This bit, the valid bit is set to Valid (1) whenever a page is being brought into the main memory, and to Non-valid (0) whenever a page is replaced in main memory.

Example 10.2 PAGE TABLE SIZE:

A 32 bit address system, has a 4 MBytes main memory. The page size is 1 KByte. What is the size of the page table?

Answer:
The page table is addressed with the page number field in the virtual address. Therefore the number of lines in the page table equals the number of virtual pages. The number of virtual pages is:

$$\text{virtual\_pages} = \frac{\text{address\_space [Bytes]}}{\text{page\_size [Bytes]}}$$

$$\text{virtual\_pages} = \frac{2^{32}}{2^{10}} = 2^{22} = 4 \text{ Mpages}$$

The width of each line in the page table equals the width of the page number field in the virtual address. If the number of virtual pages is 2^{22}, then the line width is 22 bits. It results that the size of the page table is:

page_table_size = number_of_entries * line_size

page_table_size = 2^{22}*22 bits = 11.5 Mbyte!!

As the above example points out the size of the page table can be impressive, in so much that in the example, it does not fit in the memory. However, most of the entries in the page table have the valid bit equal to 0 (Non-valid); only a number of entries equal to the number of pages that fit in the main memory have the valid bit equal to 1; that is only:

$$\frac{\text{main\_memory\_size}}{\text{page\_size}} = \frac{2^{22}}{2^{10}} = 2^{12} = 4096 \text{ entries}$$

have the valid bit equal to 1; this represents only from the total number of entries in the page table.

- One way to reduce the page table size is to apply a hash function to the virtual address, such that the page table must have only so many entries as pages in the main memory, 4096 for the example 10.2. In this case we have an **inverted page table**, The drawback is a more involved access, and some extra hardware.

- Another way to keep a smaller page table is to start with a small page table and a **limit register**; if the virtual page number is larger than the content of the limit register, then the page table must be increased, as the program requires more space. The underlying explanation for this optimization is the principle of locality: the address space won't be uniformly accesses, instead, addresses concentrate in some areas of the total address space.

• Finally, it is possible to keep the page table paged by itself, using the same idea that lags behind the virtual memory. namely that only some parts of it will be needed at any time in the execution of a program.

Even if the system uses an inverted page table, its size is so large that it must be kept in the main memory. This makes every memory access longer: first the page table must be in the memory to get the physical page address, and only then can the data be accessed. The principle of locality works again and helps improving the performance. When a translation from a virtual page to physical one is made, it is probable that it will be done again in the future, due to spatial and temporal locality inside a page. Therefore the idea is to have a cache for translations, which will make most of the translations very fast; this cache, that most new designs provide, is called **translation lookaside buffer** or **TLB**.

What block should be replaced in the case of a miss?

Both because the main purpose is to reduce the miss rate and because the large times involved in the case of a miss allow a software handling, the most used algorithm for page replacing is **Least Recently Used** (**LRU**), which is based on the assumption that replacing the page that has not been used for the longest time in the past will hurt the least the hit rate, in that probably it won't be needed in the near future as it has not been needed in the past.

To help the operating system take a decision, the page table provides a bit, the **reference bit** also called the **use bit**, which indicates if the page was accessed or not. The best candidate for a replacement is a page that has not been referenced in the past; if all pages in the main memory have been referenced, then the best candidate is one page that has been little used in the past and has not been written, thus saving the time for writing it back on the disk.

How are writes handled?

In the case of caches there are two options to manage the writes: write through and write back, each with its own advantages and disadvantages.

For virtual memory there is only one solution: **write back**. It would be disastrous to try writing on disk at every write; remember that the disk access time is in the millisecond range, which means hinders of thousands of clock cycles.

The same improvement that was possible with caches can be used for virtual memory: each entry in the page table has a bit, the dirty bit, which indicates if the content of the table was modified as a result of a write. When a page is brought in main memory, the dirty bit is set to Non-dirty (0); the first write in that page will change the bit to Dirty (1). When it comes to replacing a page, the operating system will prefer to replace a page that has not been written because this saves the time of writing out a page to the disk, again a milliseconds amount of time.

10.3 More About TLB

The TLB is a cache specially designed for page table mappings. Each entry in the TLB has the two basic fields we found in caches: a tag field which is as wide as the virtual page number, a data field which is the size of the physical page number, and several bits that indicate the status of the TLB line or of the page associated with it; these bits usually are a dirty bit which indicates if the corresponding page has been written, a reference bit which indicates if the corresponding page has been accessed, maybe a write protection bit which indicates if writes are allowed in the corresponding page. Note that these bits correspond to the same bits in the page table.

At every memory reference the TLB is accessed with the virtual page number field of the virtual address. If there is a hit, then the physical address is formed using the data field in the TLB for that entry, concatenated with the page offset. If there is a miss in the TLB, the control must determine if there is a real page fault or a simple TLB miss which can be quickly resolved by updating the TLB entry with the proper information from the page table residing in main memory. In the case of a page fault the CPU gets interrupted, and the interrupt handler will be charged to take the necessary steps to solve the fault.

Figure 10.3 presents the logical steps in addressing the memory; the diagram assumes that the cache is addressed using the physical address. The main implication of this scheme is that both accesses, to the TLB and to the cache, are serialized; the time needed to get an item from the memory, in the best case, is the access time in TLB (assume there is a TLB hit), plus the access time in the cache (assume there is a cache hit).

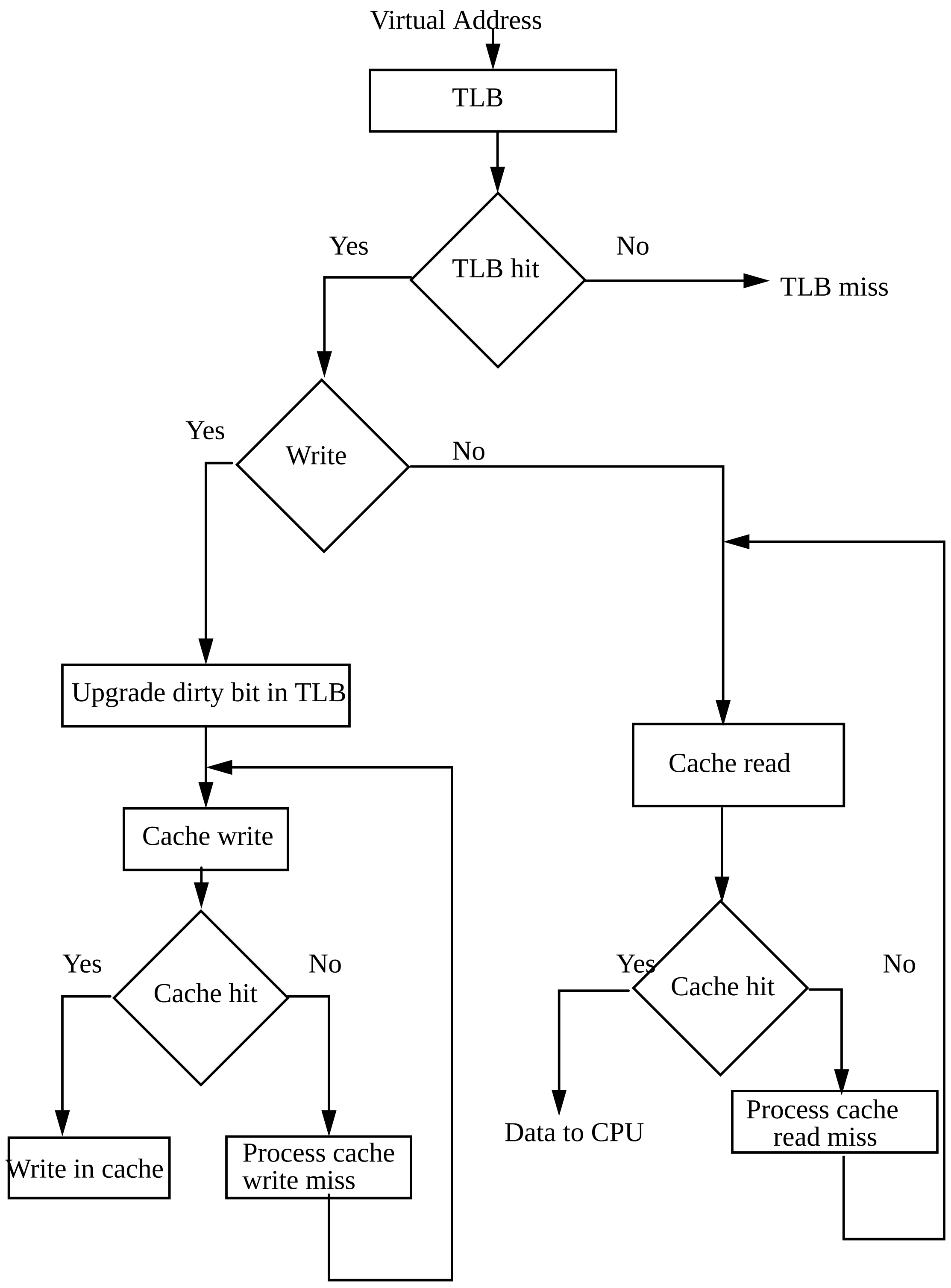

FIGURE 10.3 Steps in read/write a system with virtual memory and cache.

Certainly this is troubling because we would like to get an item from the cache in the same clock cycle the CPU issues the address; with the TLB it is probably impossible to do this, unless the clock cycle is stretched to accommodate the total access time.

Another possibility is the CPU to access the cache using an address that is totally or partial virtual: in this case we have a **virtually addressed cache**. For the virtually addressed caches, there is a benefit, shorter access times, and a big drawback: the possibility of **aliasing**. Aliasing means that an object has more than one name, in this case more than one virtual address for the same object; this can appear when several processes access some common data. In this case an object may be cached in two (or more) locations, each corresponding to a different virtual address. The consequence is that a one process may change data without the other ones being aware of this. Dealing with this problem is a fairly difficult problem.

A final comment about TLB: it is usually implemented as a fully associative cache, thus having a smaller miss rate than other options, with a small number of lines (tens) to reduce the cost of a fully associative cache. For a small number of words even the LRU replacing algorithm becomes tempting and feasible.

10.4 Problems In Selecting a Page Size

There are factors that favor a larger page size and factors that favor a small page size. Here are some. Here are some that favor a large page size:

- the larger the page size is the smaller is the page table, thus saving space in main memory;

- a larger page size makes transfers from/to disk be more efficient, in that given the big access times, it is worth transferring more data once the proper location on the disk has been accessed.

As for factors that favor a small page size, here are some:

- **internal fragmentation**: if the space used by a program is not a multiple of the page size, then there will be wasted space because the unused space in the page(s) can not be used by other programs. If a process has three segments (text, heap, stack), then the wasted space is on the average 1.5 pages. As the page size increases the wasted space gets more relevant

- **wasted bandwidth**: this is related to the internal fragmentation problem. When bringing a page in memory we also transfer the unused space in that page.

- **start time**: with a smaller page many small processes will start faster.

Exercises

10.1 Draw the block schematic of a 32 bit address system, with pages of 4KBytes. The cache has 8192 lines, each 8 words wide. Assume that the cache is word addressable.

10.2 With a TLB and cache there are several possibilities of misses: TLB miss, cache miss, page fault or combinations of these ones. List all possible combinations and indicate in which circumstances they can appear.