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:

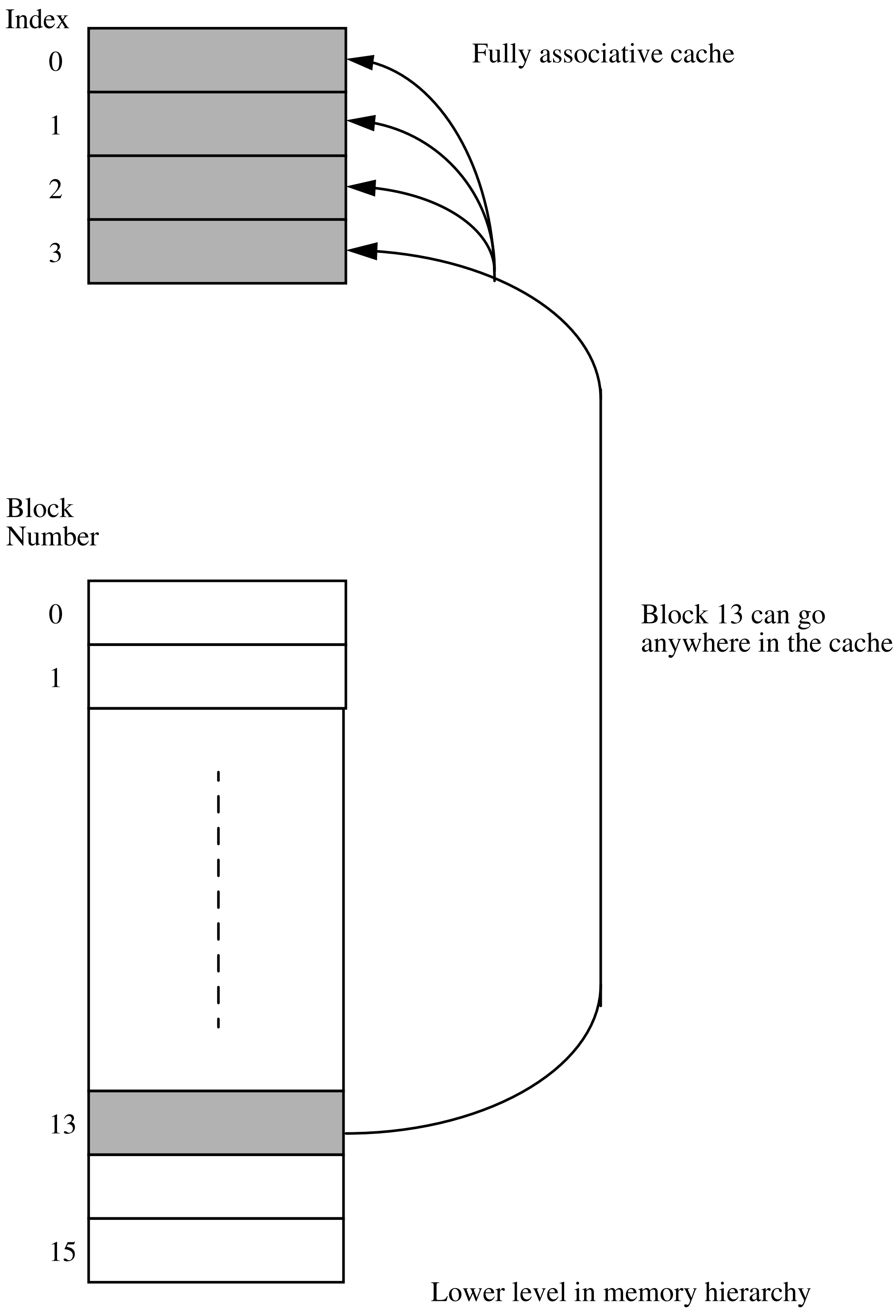

FIGURE 8.1 A fully associative four blocks (lines) cache connected to a 16 blocks.

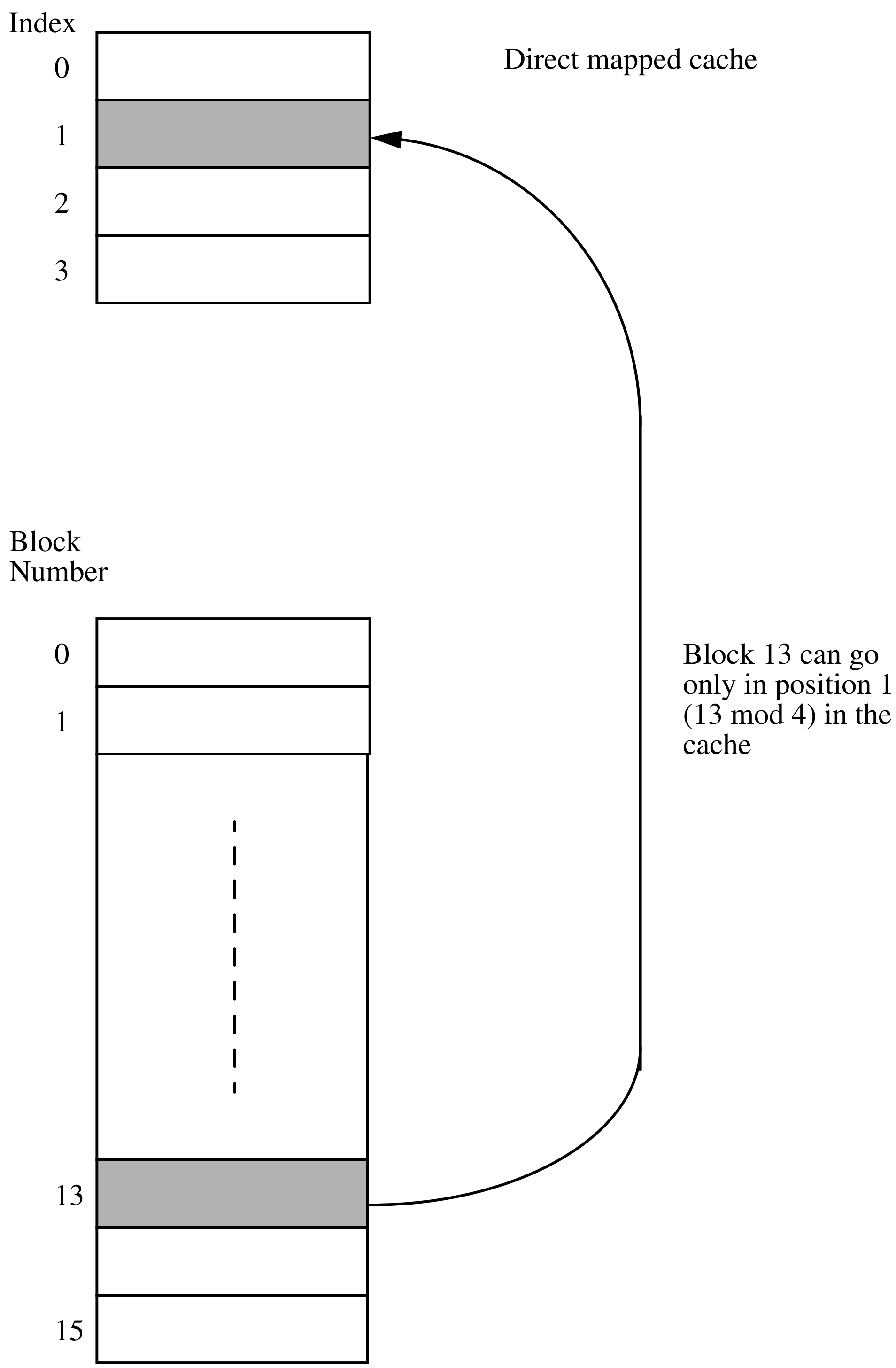

FIGURE 8.2 A Direct mapped, four blocks (lines) cache connected to a 16 blocks memory.

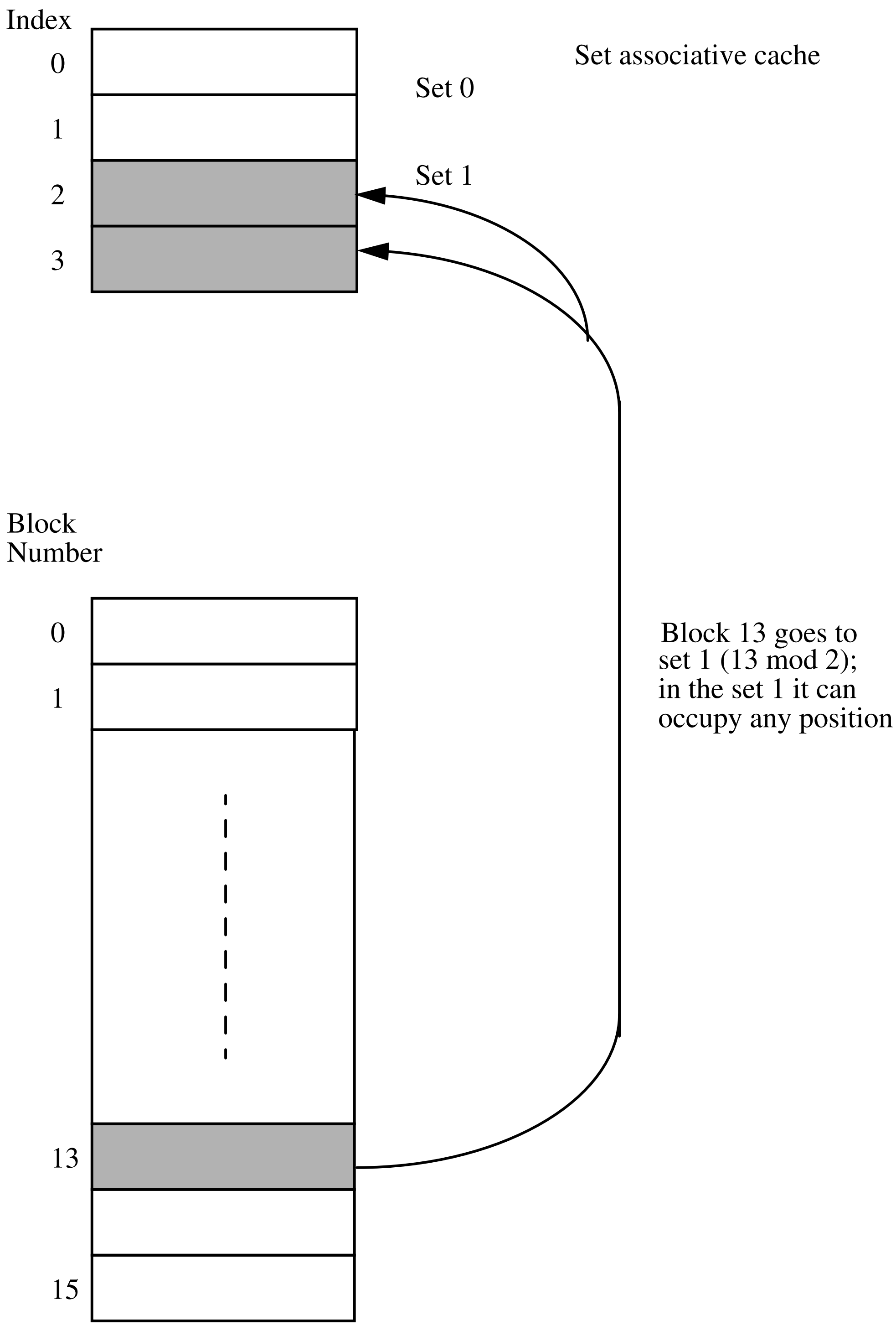

FIGURE 8.3 A 2-way set-associative cache connected to a 16 blocks memory.

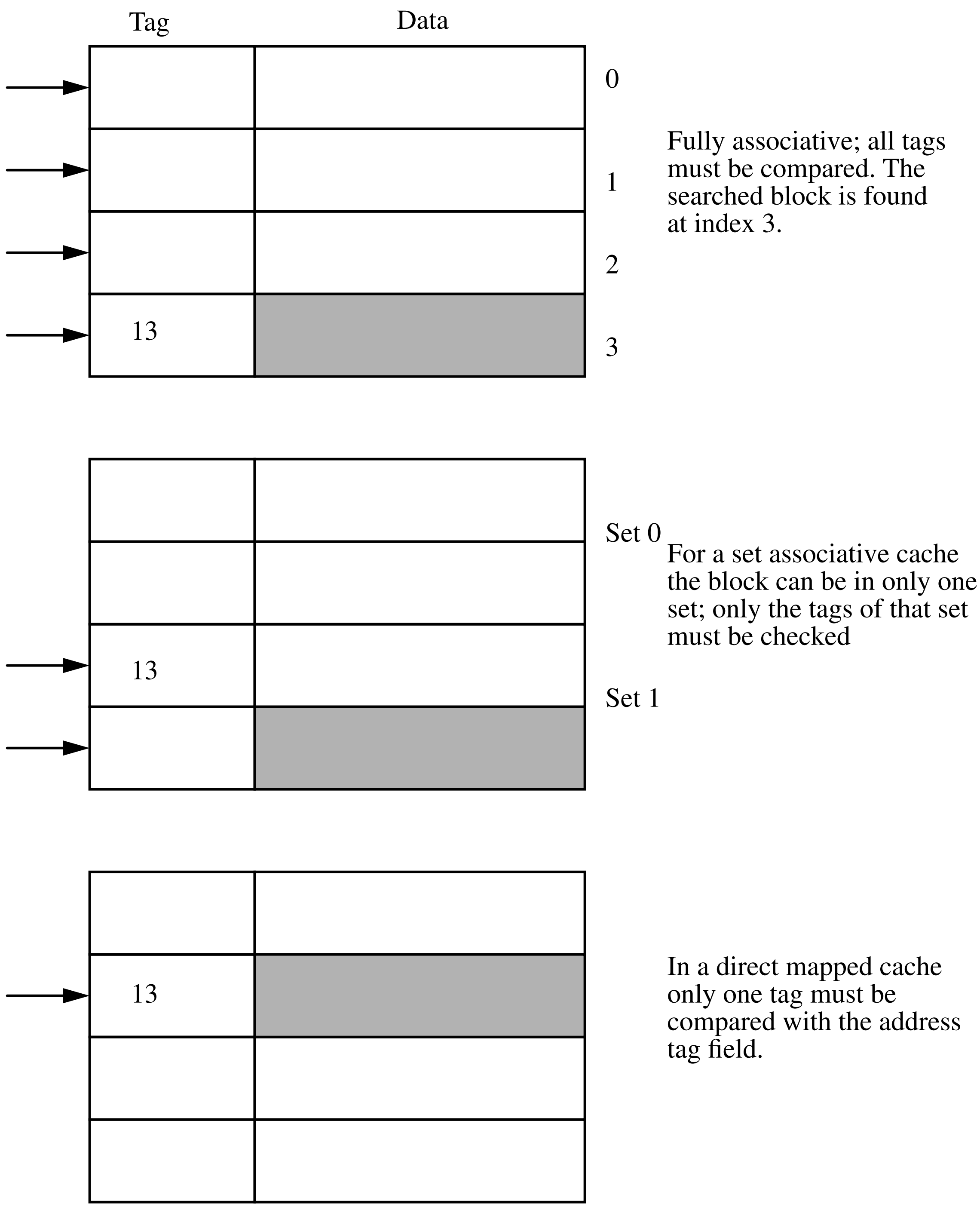

FIGURE 8.4 Finding a block in the cache implies comparing the tag field of the actual address with the content of one or more tags in the cache.

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.

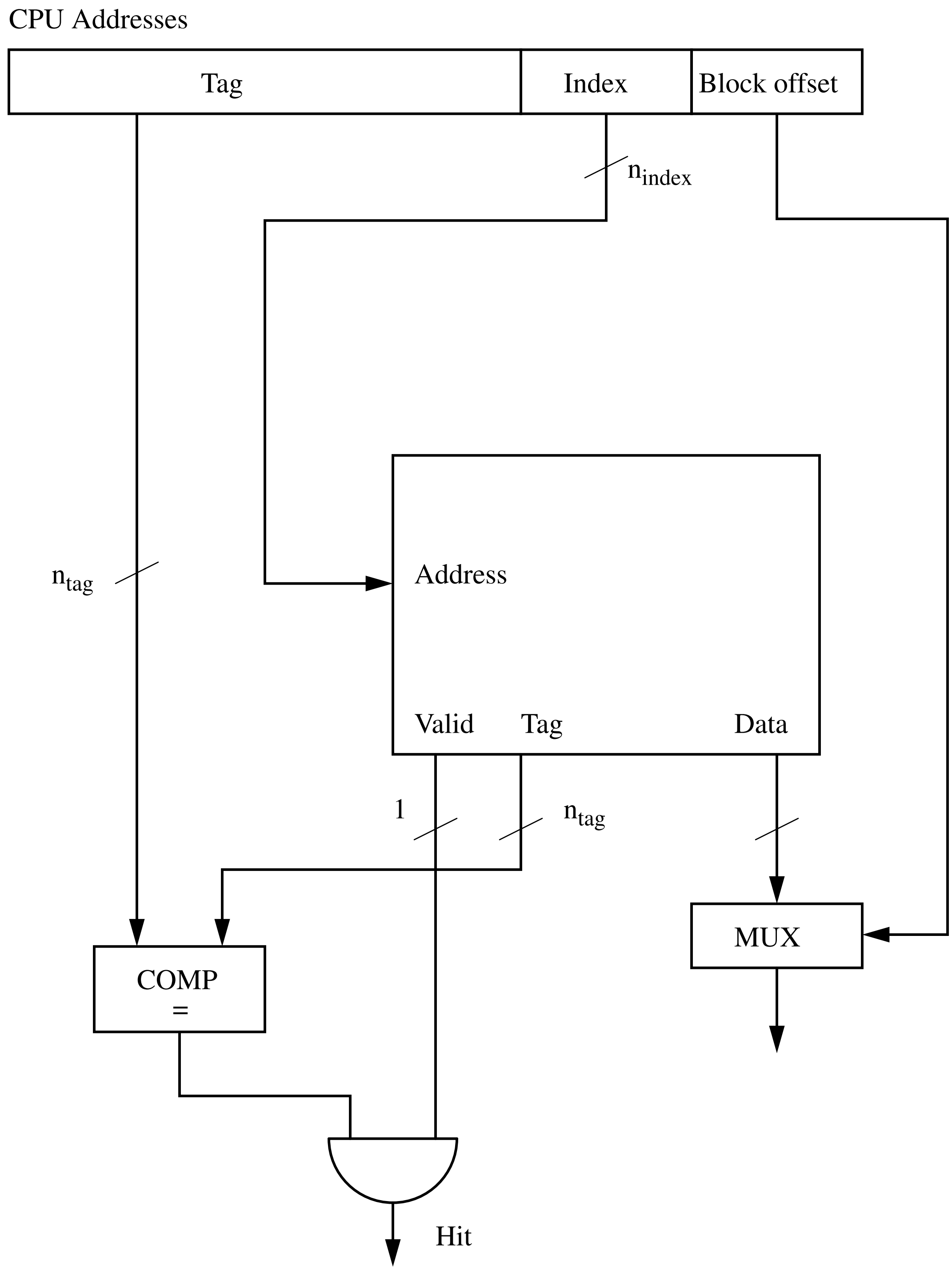

FIGURE 8.5 a direct mapped cache schematic.

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]	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	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]	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]	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:

<table>
<tr><th>CPU</th><th>Instruction</th><th>Data</th></tr>
<tr><td>Intel 80486</td><td colspan="2">8 KB</td></tr>
<tr><td>Motorola 68040</td><td>4 KB</td><td>4 KB</td></tr>
<tr><td>Intel PENTIUM</td><td>8 KB</td><td>8 KB</td></tr>
<tr><td>DEC Alpha</td><td>8 KB</td><td>8 KB</td></tr>
<tr><td>Sun MicroSPARC</td><td>4 KB</td><td>2 KB</td></tr>
<tr><td>Sun SuperSPARC</td><td>20 KB</td><td>16 KB</td></tr>
<tr><td>Hewlett-Packard PA 7100</td><td>-</td><td>-</td></tr>
<tr><td>MIPS R4000</td><td>8 KB</td><td>8 KB</td></tr>
<tr><td>MIPS R4400</td><td>16 KB</td><td>16 KB</td></tr>
<tr><td>PowerPC 601</td><td colspan="2">32 KB</td></tr>
</table>

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).