Computer organization and architecture

Lesson 19

Memory systems

Part 1

Computer system performance depends on the memory system as well as the processor microarchitecture.

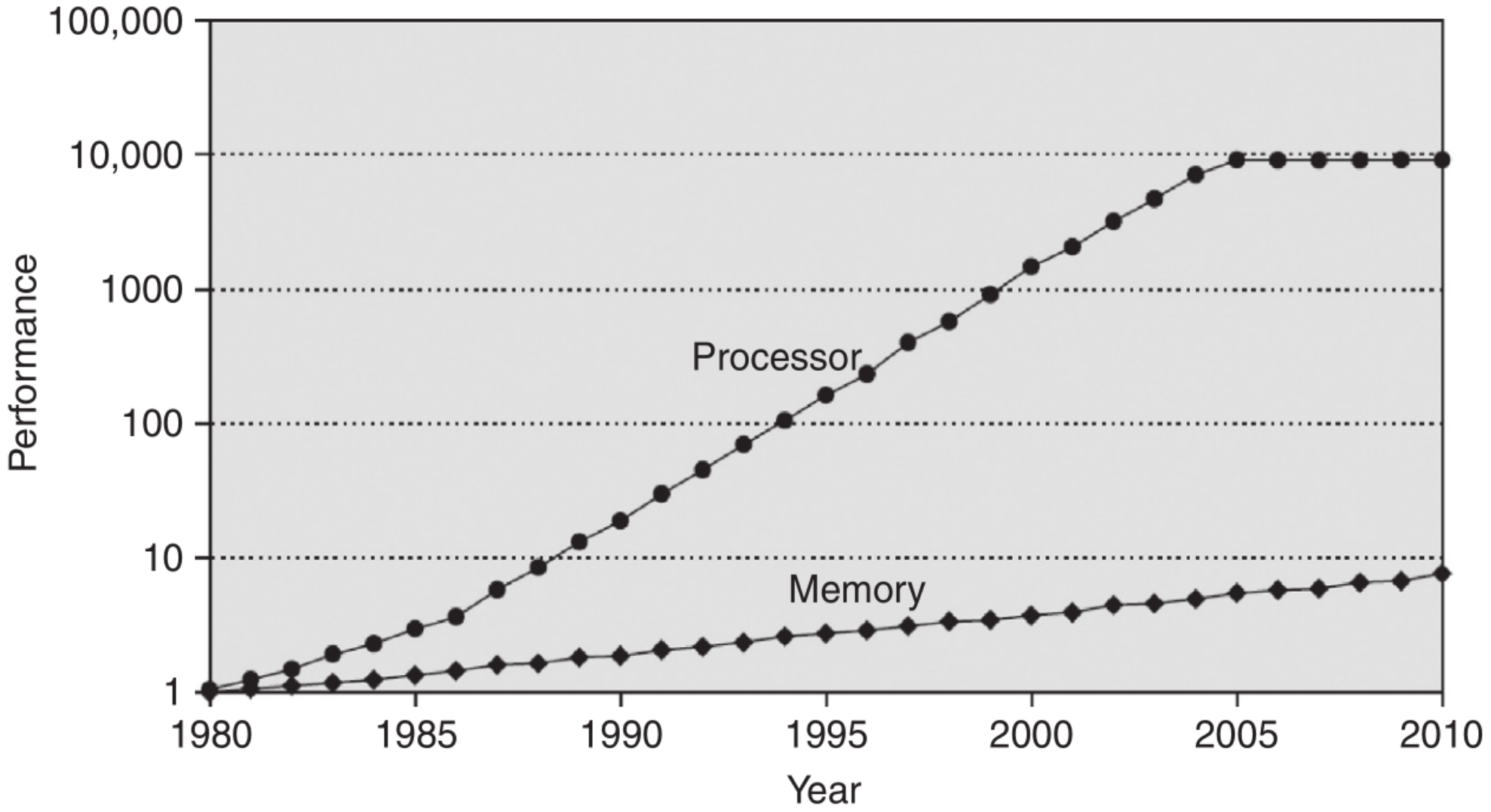
Previously, we assumed an ideal memory system that could be accessed in a single clock cycle.

Early processors were relatively slow, so memory was able to keep up

But processor speed has increased at a faster rate than memory speeds.

DRAM memories are currently 10 to 100 times slower than processors.

Diverging processor and memory performance



Computer memory is generally built from DRAM chips.

The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond).

Computers store the most commonly used instructions and data in a faster but smaller memory, called a **cache**.

The cache is usually built out of SRAM on the same chip as the processor.

The cache speed is comparable to the processor speed, because SRAM is inherently faster than DRAM, and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip.

Locality of reference

Cache is useful because of **locality of reference**

This principle states that

> Memory references by the processor, for both instructions and data, tend to cluster.

Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

It is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is much less than that of the level above.

Spatial and temporal locality

Spatial locality – when the processor accesses a piece of data, it is also likely to access data in nearby memory locations.

This reflects the tendency of a processor

– to access instructions sequentially

– to access data locations sequentially

such as when processing a table of data

Temporal locality – the processor is likely to access a piece of data again soon if it has accessed that data recently.

Example: when an iteration loop is executed, the processor executes the same set of instructions repeatedly.

The cache size is few kilobytes to several megabytes.

If the processor requests data that is available in the cache, it is returned quickly.

This is called a **cache hit**.

Otherwise, the processor retrieves the data from main memory (DRAM).

This is called a **cache miss**.

$$Miss\ Rate = \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate$$

$$Hit\ Rate = \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses} = 1 - Miss\ Rate$$

If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate.

Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses.

In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words.

The third level in the memory hierarchy is the hard drive.

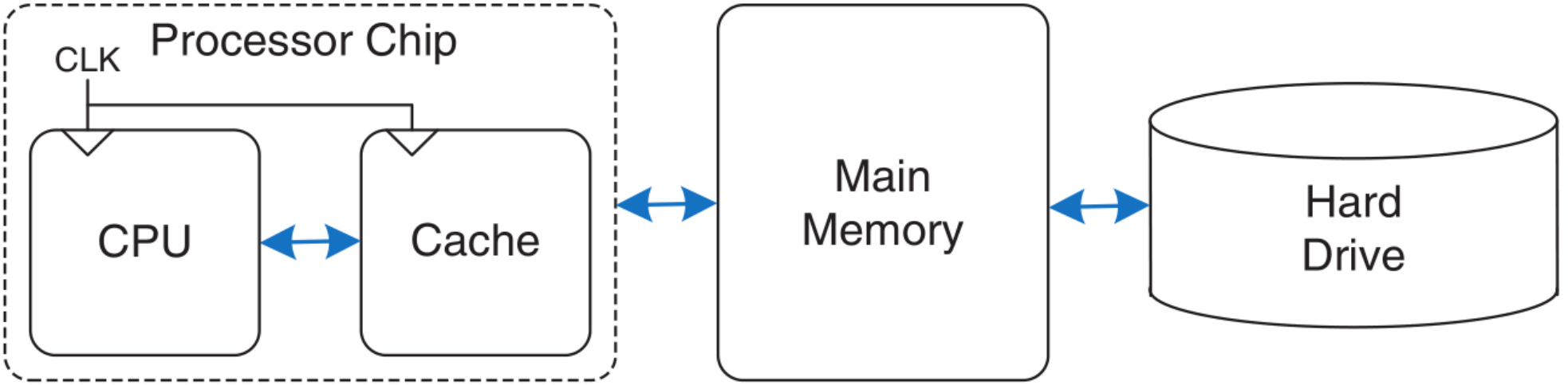
A hard disk drive (HDD), built using magnetic storage.

Solid state drives (SSDs) are built using flash memory.

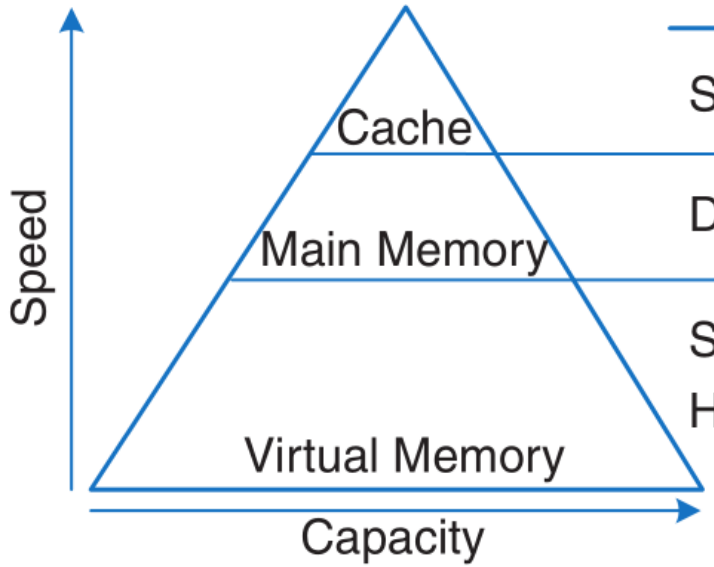
The hard drive provides an illusion of more capacity than actually exists in the main memory.

It is thus called **virtual memory**.

Memory hierarchy



Typical characteristics in 2015



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$5,000	0.5	25+
DRAM	$7	10–50	10
SSD	$0.40	20,000	0.5
HDD	$0.05	5,000,000	0.75

Cache capacity, *C* – the number of data words it can hold

The computer system designer must choose what subset of the main memory is kept in the cache.

The group of words that the cache fetches from memory is called a **cache block** or **cache line**.

The number of words in the cache block is called the **block size**, *b*

A cache of capacity *C* contains *B* = *C/b* blocks.

A cache is organized into *S* **sets**, each of which holds one or more blocks of data.

The relationship between the address of data in main memory and the location of that data in the cache is called the **mapping**.

In a **direct mapped cache**, each set contains exactly one block.

The cache has $S = B$ sets.

In an ***N*-way set associative cache**, each set contains N blocks.

The cache has $S = B/N$ sets.

A **fully associative cache** has only $S = 1$ set.

Consider an ARM memory with 32-bit addresses and 32-bit words.

The memory is byte-addressable.

For simplicity, we analyze caches with an eight-word capacity.

We begin with a one-word block size (b = 1), then generalize later to larger blocks.

Direct mapped cache

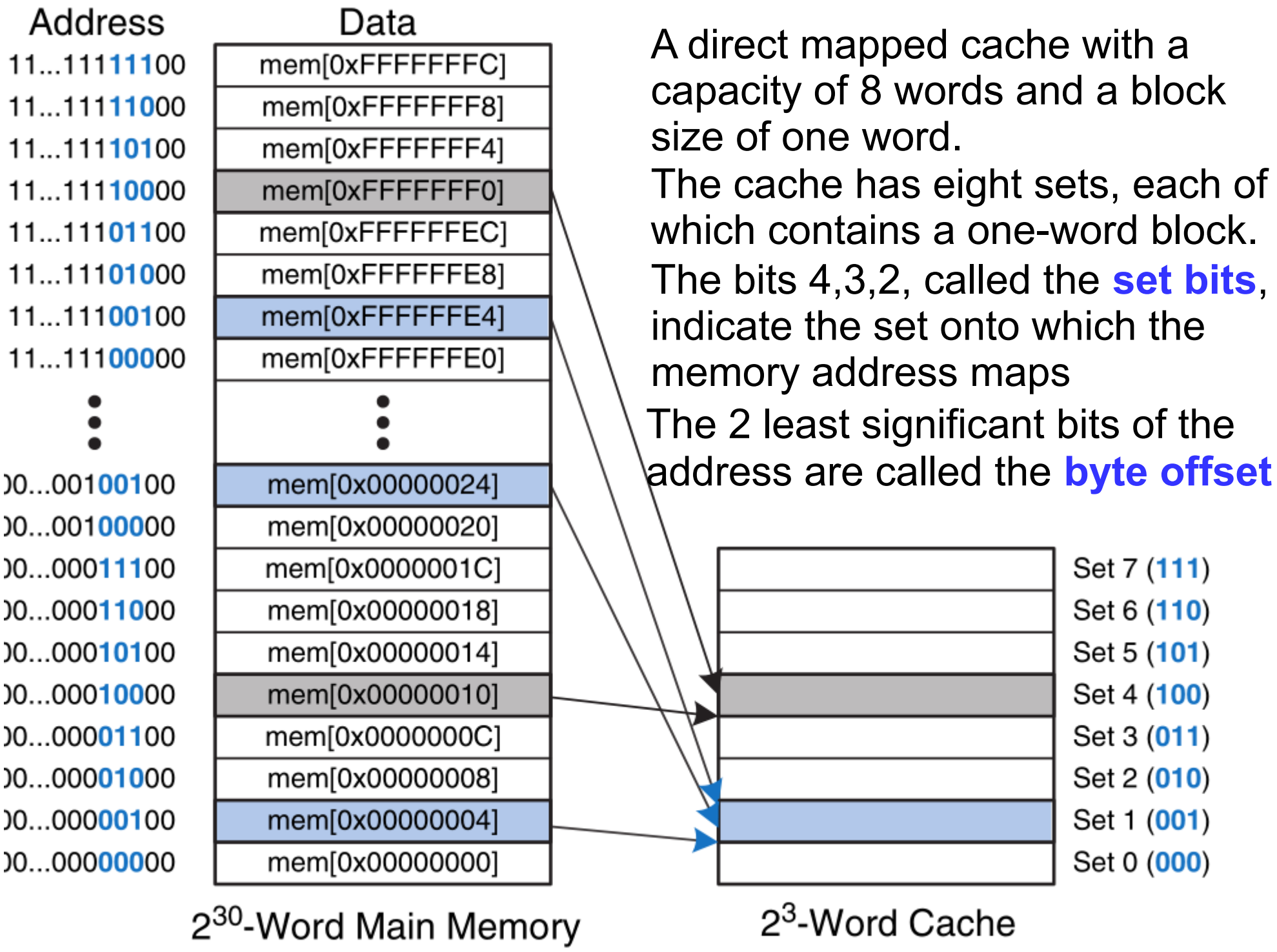
$S = B$ sets

Imagine main memory as being mapped into b-word blocks.

An address in block 0 of main memory maps to set 0 of the cache.

An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B-1$ of main memory maps to block $B-1$ of the cache.

There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

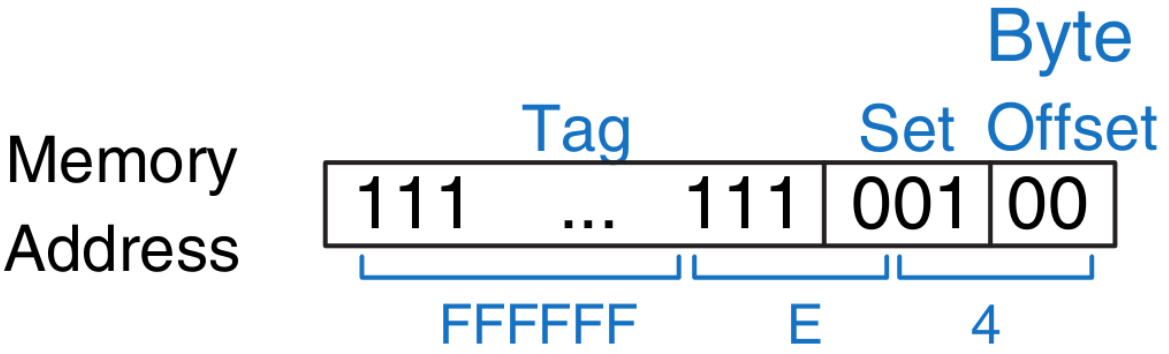

Address
Data
11...111111100
11...111111000
11...111101100
11...111101000
11...111100100
11...111100000
00...001001100
00...001000000
00...000111100
00...000111000
00...000101100
00...000101000
00...000011100
00...000011000
00...000001100
00...000001000
00...000000100
00...000000000
mem[0xFFFFFFFC]
mem[0xFFFFFFF8]
mem[0xFFFFFFF4]
mem[0xFFFFFFF0]
mem[0xFFFFFFEC]
mem[0xFFFFFFE8]
mem[0xFFFFFFE4]
mem[0xFFFFFFE0]
mem[0x00000024]
mem[0x00000020]
mem[0x0000001C]
mem[0x00000018]
mem[0x00000014]
mem[0x00000010]
mem[0x0000000C]
mem[0x00000008]
mem[0x00000004]
mem[0x00000000]
A direct mapped cache with a capacity of 8 words and a block size of one word.
The cache has eight sets, each of which contains a one-word block.
The bits 4,3,2, called the set bits, indicate the set onto which the memory address maps
The 2 least significant bits of the address are called the byte offset
Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)
2^30-Word Main Memory
2^3-Word Cache

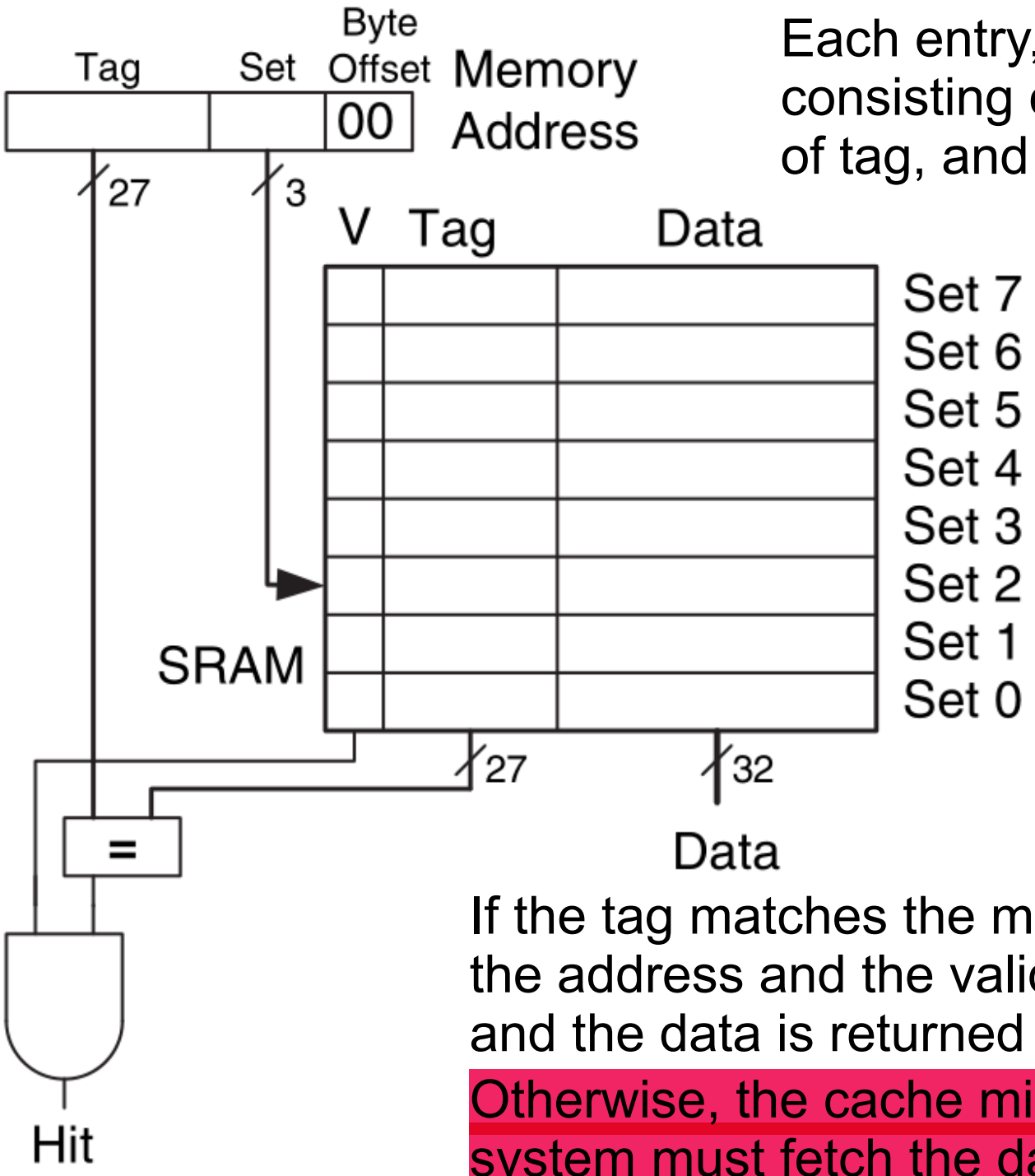
Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set.

The least significant bits of the address specify which set holds the data.

The remaining most significant bits are called the **tag** and indicate which of the many possible addresses is held in that set.

Cache fields for address 0xFFFFFFE4 :





Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit.

The cache uses a valid bit to indicate whether the set holds meaningful data.

The cache is accessed using the 32-bit address

A load instruction reads the specified entry from the cache and checks the tag and valid bits.

If the tag matches the most significant 27 bits of the address and the valid bit is 1, the cache hits and the data is returned to the processor.

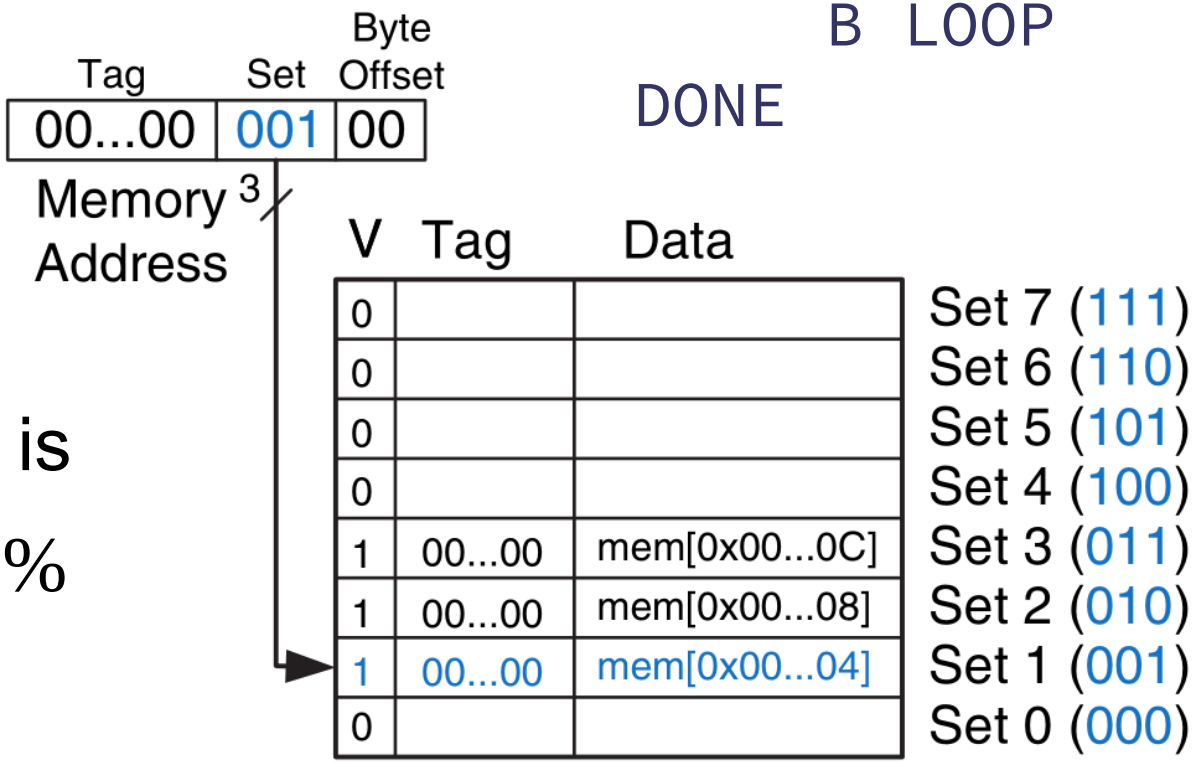
Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example: What is the miss rate?

The first time the loop executes, the cache is empty and the data must be fetched from main memory.

The next four times the loop executes, the data is found in the cache.

```
        MOV R0, #5
        MOV R1, #0
LOOP    CMP R0, #0
        BEQ DONE
        LDR R2, [R1, #4]
        LDR R3, [R1, #12]
        LDR R4, [R1, #8]
        SUB R0, R0, #1
        B LOOP
DONE
```



The miss rate is

$$3/15 = 20\,\%$$

A **conflict** occurs when two recently accessed addresses map to the same cache block.

The most recently accessed address **evicts** the previous one from the block.

Direct mapped caches have only one block in each set, so two addresses that map to the same set always cause a conflict.

```
      MOV R0, #5
      MOV R1, #0
LOOP  CMP R0, #0
      BEQ DONE
      LDR R2, [R1, #0x4]
      LDR R3, [R1, #0x24]
      SUB R0, R0, #1
      B LOOP
DONE
```

Memory addresses 0x4, 0x24 both map to set 1.

The two addresses conflict, and the miss rate is 100%.

Multi-way set associative cache

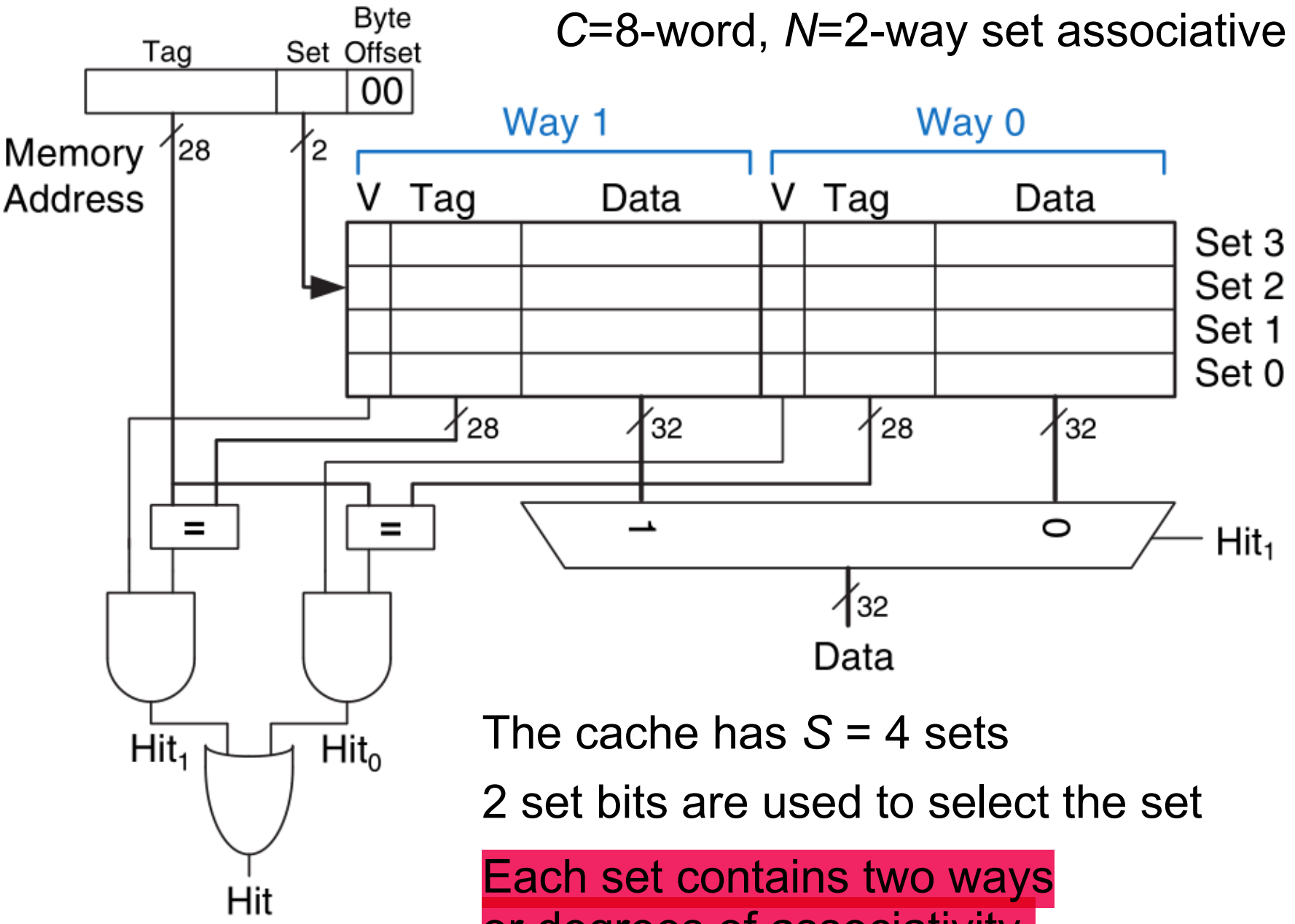
An ***N*-way set associative cache** reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found.

Each memory address maps to a specific set, but it can map to any one of the *N* blocks in the set..

A direct mapped cache is another name for a one-way set associative cache.

N is also called the **degree of associativity** of the cache.

C=8-word, *N*=2-way set associative cache



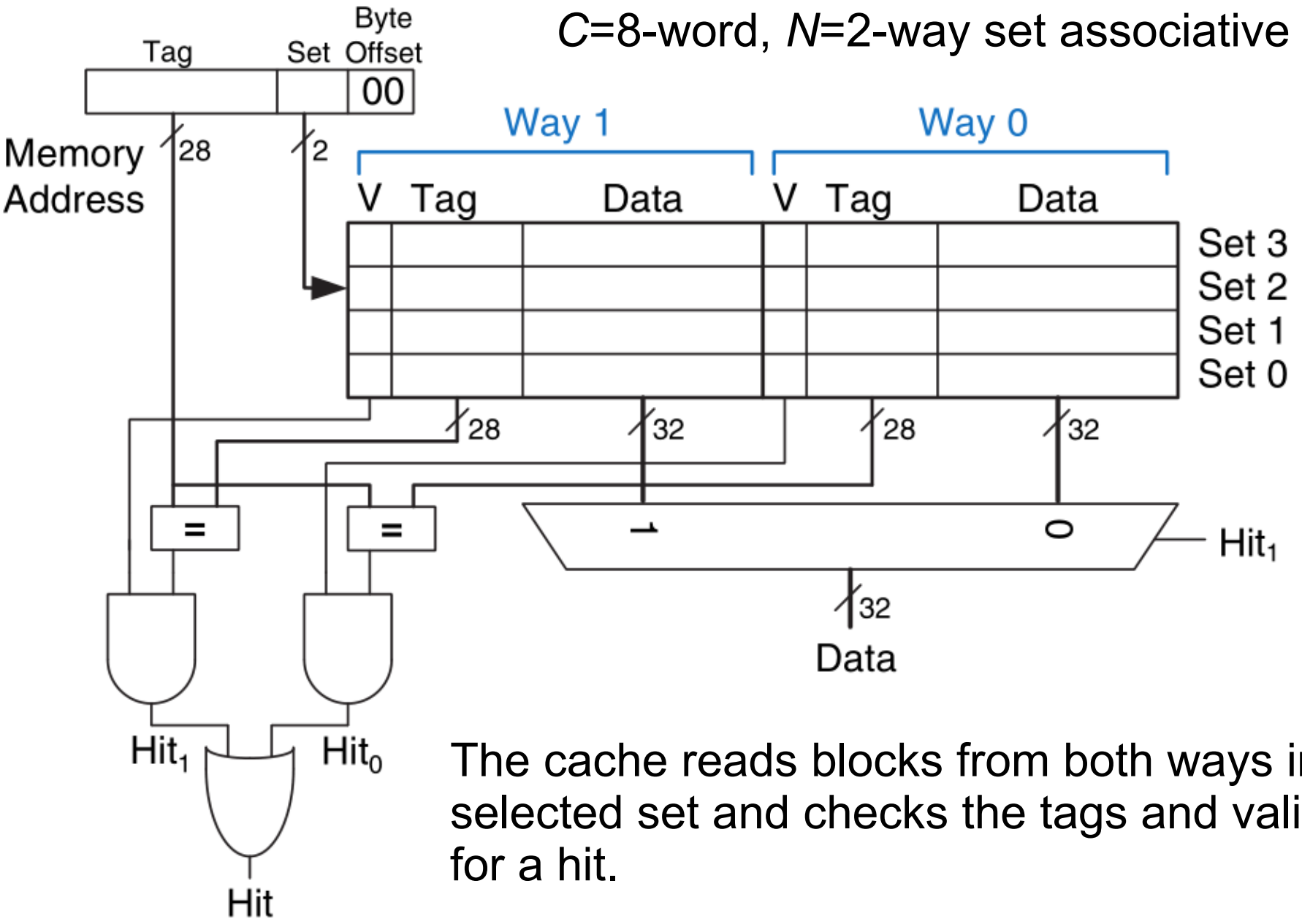
The cache has *S* = 4 sets

2 set bits are used to select the set

Each set contains two ways or degrees of associativity.

Each way consists of a data block and the valid and tag bits.

C=8-word, *N*=2-way set associative cache



The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit.

If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts.

However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators.

They also raise the question of which way to replace when both ways are full.

Most commercial systems use set associative caches.

	Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...00	mem[0x00...24]	1	00...10	mem[0x00...04]	Set 1
0			0			Set 0

```
      MOV R0, #5
      MOV R1, #0
LOOP  CMP R0, #0
      BEQ DONE
      LDR R2, [R1, #0x4]
      LDR R3, [R1, #0x24]
      SUB R0, R0, #1
      B LOOP
DONE
```

Both memory accesses, to addresses 0x4 and 0x24, map to set 1.

The cache has two ways, so it can accommodate data from both addresses.

The miss rate is

$$2/10 = 20\%$$

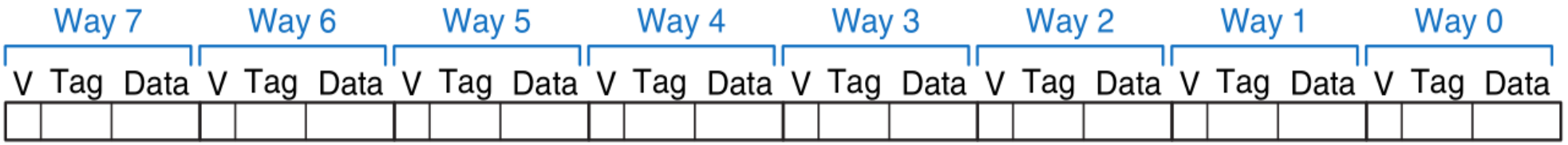
Fully associative cache

– contains a single set with *B* ways, where *B* is the number of blocks.

A memory address can map to a block in any of these ways.

A fully associative cache is another name for a *B*-way set associative cache with one set.

Example: *B* = 8



Upon a data request, 8 tag comparisons (not shown) must be made, because the data could be in any block.

An 8:1 multiplexer chooses the proper data if a hit occurs.

Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons.

They are best suited to relatively small caches because of the large number of comparators.

Block size

With a one-word block size (b = 1), we can take advantage only of temporal locality.

To exploit spatial locality, a cache uses larger blocks (b > 1) to hold several consecutive words.

With a larger block size, when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched.

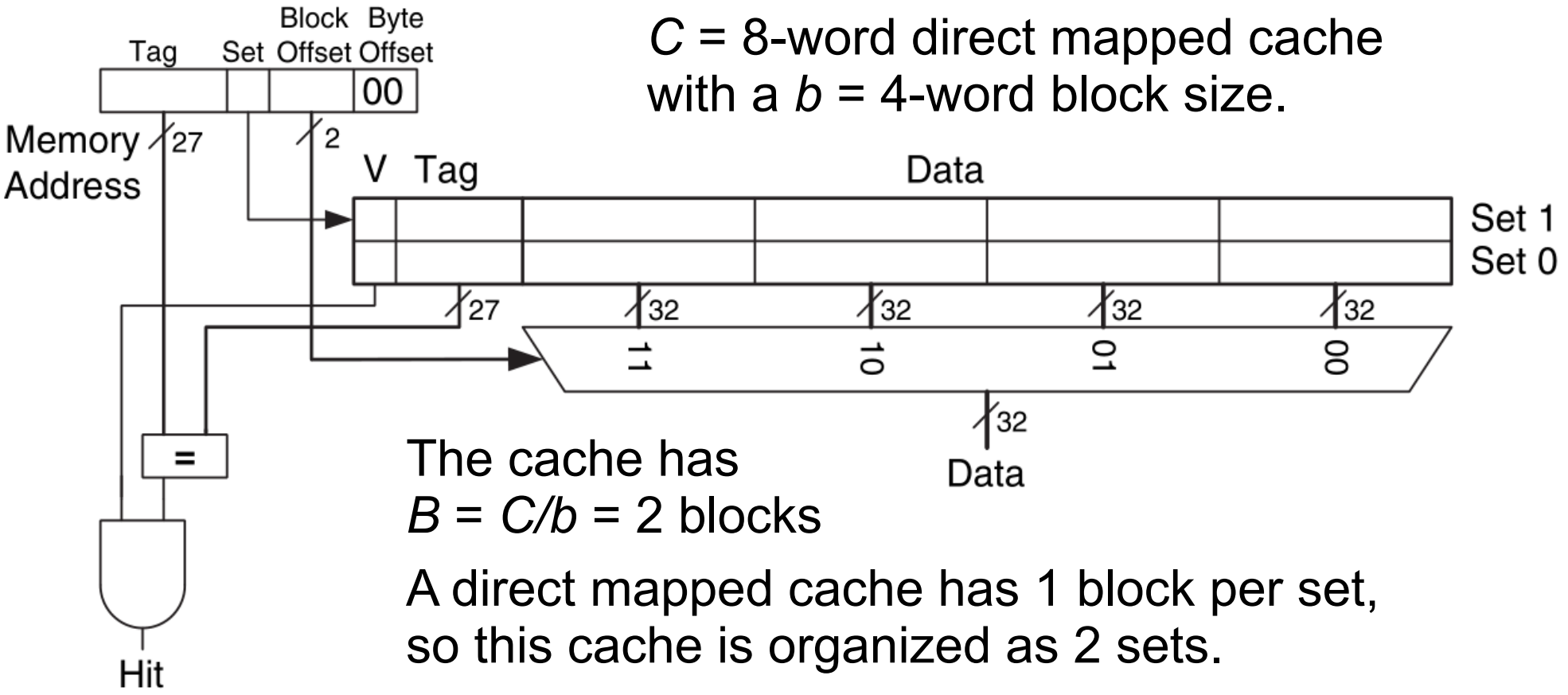
However, a large block size means that a fixed-size cache will have fewer blocks.

This may lead to more conflicts, increasing the miss rate.

It takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory.

The time required to load the missing block into the cache is called the **miss penalty**.

If the adjacent words in the block are not accessed later, the effort of fetching them is wasted.



C = 8-word direct mapped cache with a *b* = 4-word block size.

The cache has
B = *C*/*b* = 2 blocks

A direct mapped cache has 1 block per set, so this cache is organized as 2 sets.

1 bit is used to select the set.

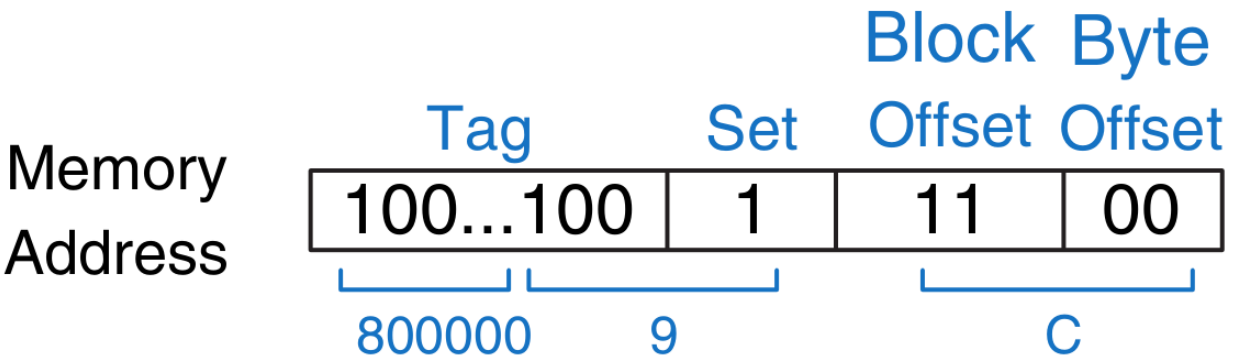
A multiplexer selects the word within the block.

The multiplexer is controlled by the 2 block offset bits of the address.

The most significant 27 address bits form the tag.

Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Example: Cache fields for address 0x8000009C



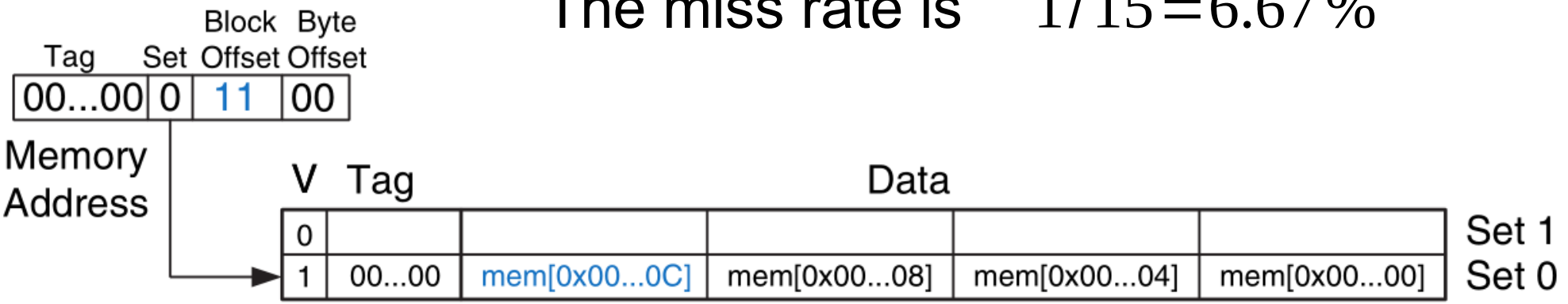
Example: What is the miss rate?

```
        MOV R0, #5
        MOV R1, #0
LOOP    CMP R0, #0
        BEQ DONE
        LDR R2, [R1, #4]
        LDR R3, [R1, #12]
        LDR R4, [R1, #8]
        SUB R0, R0, #1
        B LOOP
DONE
```

On the first loop iteration, the cache misses on the access to memory address 0x4.

All subsequent accesses hit in the cache.

The miss rate is $1/15 = 6.67\%$



LRU replacement

In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full.

The principle of temporal locality suggests that the best choice is to evict the **least recently used** (**LRU**) block, because it is least likely to be used again soon.

In a two-way set associative cache, a use bit, U, indicates which way within a set was least recently used.

Each time one of the ways is used, U is adjusted to indicate the other way.

Example: 8-word 2-way set associative cache, a block size of one word

```
MOV R0, #0
LDR R1, [R0, #4]
LDR R2, [R0, #0x24]
LDR R3, [R0, #0x54]
```

		Way 1			Way 0		
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)

U = 0 – data in way 0 was the least recently used

		Way 1			Way 0		
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)

Pseudo-LRU replacement

For set associative caches with more than two ways, the ways are often divided into two groups and U indicates which group of ways was least recently used.

Upon replacement, the new block replaces a random block within the least recently used group.

Such a policy is called pseudo-LRU

Exercise 19.1

A program has to repeatedly access the following sequence of addresses

0x0, 0x10, 0x20, 0x30, 0x40, ...

Calculate the miss rate for a direct mapped cache with a size (capacity) of 16 words and block size of 4 words and a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Which cache performs better for this sequence?