THE MEMORY SYSTEM

INTRODUCTION

- Programs and the data they operate on are held in the memory of the computer.
- The execution speed of programs is highly dependent on the speed with which instructions and data can be transferred between the processor and the memory.
- It is also important to have a large memory to facilitate the execution of programs that are large and deal with huge amounts of data.
- Ideally, the memory would be fast, large and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increase in speed and size are achieved at increased cost.
- Much work has gone into developing clever structures that improve the apparent speed and size of the memory, yet keep the cost reasonable.

SOME BASIC CONCEPTS

☞ The addressing scheme of any computer system determines the maximum size of the main memory that a computer can use.

Examples:

1. If number of address bits = 16

   MM Size = 2_{16} = 65,536

   = 64K memory locations

2. If number of address bits = 32

   MM Size = 2_{32} = 42,94,967,296

   = 4G memory locations

3. If number of address bits = 40

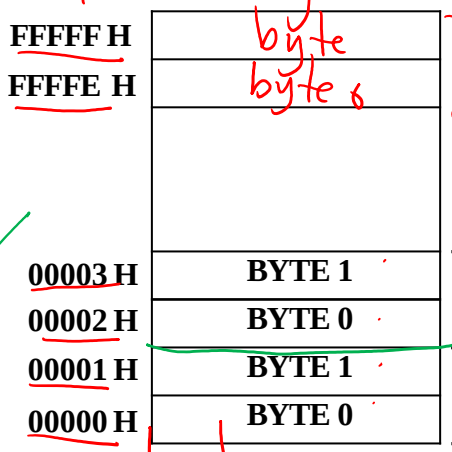
   MM Size = 2_{40} = 1,099,511,627,776

   = 1T memory locations

☞ The MM can usually store and retrieve data in word length quantities. However, the processor can access individual memory bytes as long as each byte has a unique address.

Example:

1. For a 16-bit processor



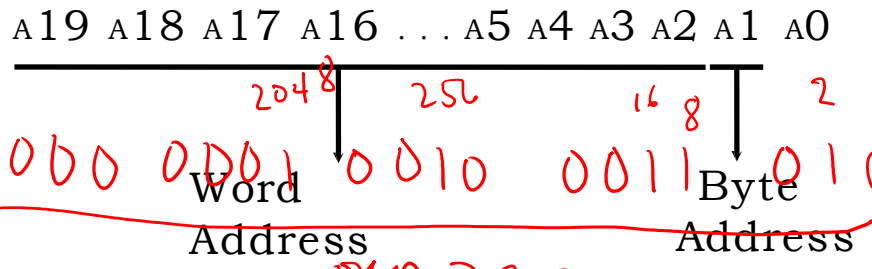
For the first 16-bit word (Word 0):

Low Order Byte Address = 0000...0000
High Order Byte Address = 0000...0001

For the second 16-bit word (Word 1):

Low Order Byte Address = 0000...0010
High Order Byte Address = 0000...0011

Therefore:



3. 0B0B0H = 0000 1011 0000 1011 0000 | 0

16,384 4096 2048 64 8 16

word 22,616 | byte 0

byte 0 of word 22,616 /

1st byte of 22,617 th word

0 - 1st - byte 0

1 - 2nd - byte 1

2. 00003H = 0000 0000 0000 0000 0011 | 1

word 1 | byte1

byte 1 of word 1 or

2nd byte of the 2nd word

for 32-bit CPU

n = | WORD | Byte |

2. For a 32-bit processor

$b_{31} b_{30} \ldots b_3 b_2 \mid b_1 b_0$

n = 30 bits

FFFFF H	3	word # 262,143
FFFFE H	2	
FFFFD	1	
FFFFC	0	
00003 H	BYTE 3	WORD 0
00002 H	BYTE 2	
00001 H	BYTE 1	
00000 H	BYTE 0	

byte # 00 - 1st, 01 - 2nd, 10 - 3rd, 11 - 4th

For the first 32-bit word (Word 0):

1st Byte Address	=	0000...0000
2nd Byte Address	=	0000...0001
3rd Byte Address	=	0000...0010
4th Byte Address	=	0000...0011

Example:

1. 01234H = 0000 0001 0010 0011 0100 | 00

1024 128 8 4 1

Therefore:

word 1165

$A_{19} A_{18} A_{17} A_{16} \ldots A_5 A_4 A_3 A_2 A_1 A_0$

Word Address | Byte Address

byte 0 of word 1165

or

1st byte of 1166th word

2. 034A1H = 0000 0011 0100 1010 0001

2048 1024 256 32 8 1

Word 3,368 | byte 1

byte 1 of word 3,368 =
2nd byte of the 3,369th word

16	9	8	7	6	5	4	3	2	1	0
2^{16}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1024	512	256	128	64	32	16	8	4	2	1

binary places

byte # (2 bit)

byte 0 - 1st = 00
1 - 2nd = 01
2 - 3rd = 10
3 - 4th/last = 11

3. 00002H = 0000 0000 0000 0000 0010

word 0 | byte 2

byte 2 of word 0 =
3rd byte of the 1st word

32-bit CPU

= last byte of word 380

n = 2s bit = 005F3H

0000	0000	0000	0000	0000
0000	0000	0101	1111	0011
0	0	5	F	3

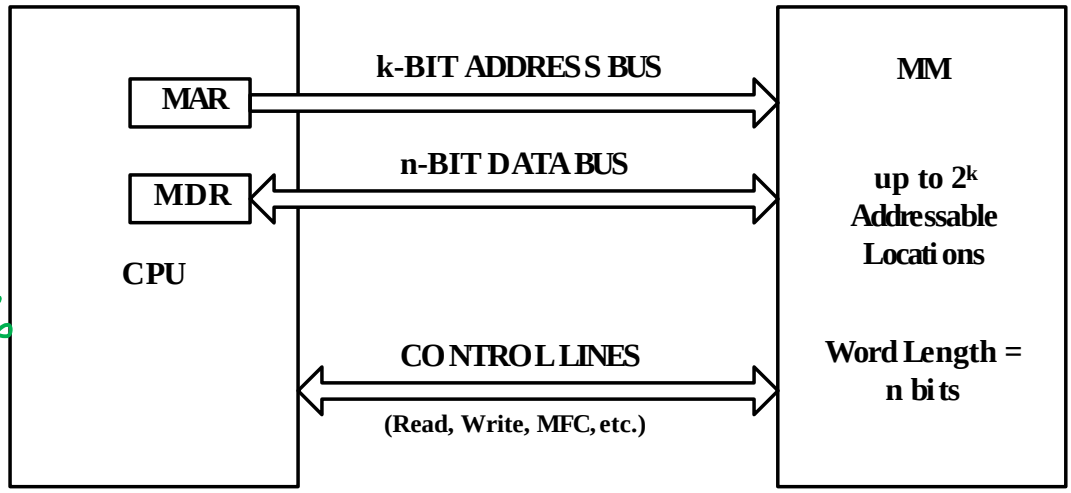
1. 02468H

16-bit

64-bit

b31 b30 ... ... b3 b2 b1 b0

☞ Connection of the main memory to the processor:



MM size =
1MB =
1,048,576
8086
8088
address bus - 20 bit 20 bits

☞ *Memory Access Time* is the time between the initiation of an operation and the completion of that operation.

data bus (extenal)☞ 16-bit 8 bit

Memory Cycle Time is the minimum time delay required between the initiation of two successive memory operations, for example, the time between two successive memory read operations. Cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit.

internal 16-bit 8018 8028 16-bit

- *Random Access Memory* is memory in which the access time of any location is a fixed amount of time that is independent of the location's address Main memory units are of this type. This distinguishesthem from serial, or partly serial, access storage devices such as magnetictapes and disks. Access time on these devices depends on the address or position of data.

- The processor can usually process instruction and data faster than they can be fetched from memory. Thus, MM becomes a bottleneck.

fast huge inexpensive

SPEED, SIZE, AND COST

- Fast memory can be achieved if bipolar static RAM chips are used. But these chips are expensive and have relatively small packing densities. Therefore, it is impractical to build a large memory using these chips.

- The only alternative is to use dynamic RAM chips, which have higher packing densities and are much cheaper. But such memories are significantly slower.

- The solution therefore is to implement main memory using dynamic RAM technology and use bipolar static RAMs in building smaller memory units where speed is of the essence, such as in *cache memories*.

CACHE MEMORY

- The speed of the main memory is very low in comparison with the speed of modern processors. For good performance, the processor cannot spend much of its time waiting to access instructions and data in main memory.

- Therefore, it is important to devise a scheme that reduces the time needed to access the necessary information. Since the speed of the main memory unit is limited by electronic and packaging constraints, the solution must be sought in a different architectural arrangement.

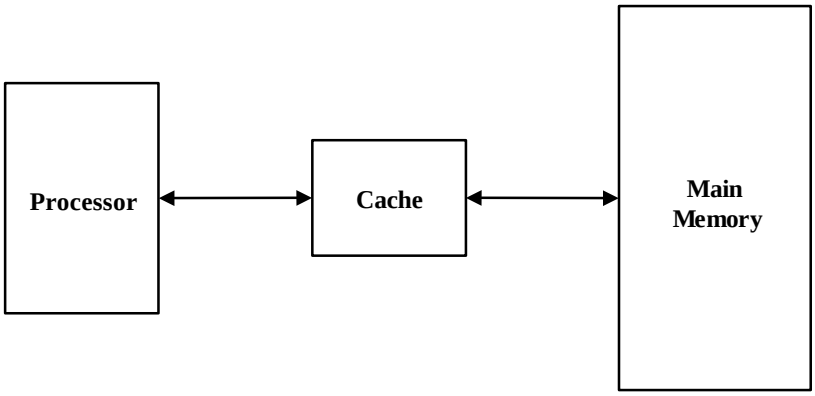
- An efficient solution is to use a fast cache memory which essentially makes the main memory appear to the processor to be faster than it really is.

- The effectiveness of the cache mechanism is based on a property of computer programs called the *locality of reference*. Analysis of programs shows that most of their execution time is spent on routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures or functions that repeatedly call each other.

In other words, locality of reference states that many instructions in a few localized areas of the program are repeatedly executed during some period of time and that the remainder of the program is accessed relatively infrequently.

Locality of reference manifests itself in two ways: *temporal* and *spatial*. Temporal means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions in close proximity to a recently executed instruction are also likely to be executed soon.

☞ It is therefore logical to put the active segments of a program in a fast memory to reduce total execution time. The fast memory is called the *cache memory*.



☞ Operation of Cache Memory:

The temporal aspect of the locality of reference suggests that whenever an information item is needed (instruction or data) is first needed, this item should be brought into the cache where it will hopefully remain until it is needed again.

The spatial aspect suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that reside in adjacent addresses as well. The term *block* is used to refer to a set of contiguous address locations of some size. Main memory is therefore divided into blocks of words. Cache memory is smaller than main memory but it can hold or store a number of these blocks at any given time.

When the processor issues a read request, the system transfers the contents of a block of memory words containing the location specified into the cache one word at a time. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache.

When the cache is full and the program references a memory word that is not in the cache, the cache control hardware must decide which block to remove to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the *replacement algorithm*.

The processor does not need to know explicitly about the existence of the cache. It simply issues Read or Write requests using addresses that refer to the locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.

- If the requested word is in the cache, the Read or Write operation is performed on the appropriate cache location. In this case, a *read hit* or *write hit* is said to have occurred.

- In a Read operation, the main memory is not involved. Only the cache is read.

- In a Write operation, the system can proceed in two ways:

1. In the first technique, called the *write-through protocol*, the cache location and the main memory location are updated simultaneously.

2. The second technique is to update only the cache location to mark it as updated (using an associated flag bit called the *dirty* or *modified* bit). The main memory location of the word is updated later, when the block containing this marked word is to be removed from the cache to make room for a new block. This technique is known as the *write-back* or *copy-back protocol*.

- The write-through protocol is simpler, but it results in unnecessary Write operations in main memory when a given cache word is updated several times during its cache residency.

  The write-back protocol may also result in unnecessary Write operations because when a cache block is written back to the memory all words of the block are written back, even if only a single word has been changed while the block was in cache.

- When the addressed word in a Read operation is not in the cache, a *read miss* occurs. The block of words that contains the requested word is copied from the main memory into the cache. After the entire block is loaded into cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from main memory. This approach, which is called *load- through*, or *early restart*, reduces the processor's waiting period somewhat, but at the expenses of more complex circuitry.

- During a Write operation, if the addressed word is not in the cache, a *write miss* occurs. Then, if the write-through protocol is used, the information is written directly into the main memory. In the case of the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.

Assume: 65,536 bytes

Cache size = 2,048 bytes

block size = 16 bytes / block

$$n = \frac{\log 65,536}{\log 2}$$

$n = 16$ bits

$$\#\text{ of mmBs} = \frac{\text{MM size}}{\text{block size}}$$

$$= \frac{65,536 \text{ bytes}}{16 \text{ bytes/block}}$$

$= 4,096$ (0 - 4,095 mmBs)

$$\#\text{ of cBs} = \frac{\text{cache size}}{\text{block size}}$$

$$= \frac{2,048 \text{ bytes}}{16 \text{ bytes/block}}$$

$= 128$ (0 - 127 cBs)

Assume:

MM size = 65,536 byte

Cache size = 2048 byte

block size = 16 bytes/block

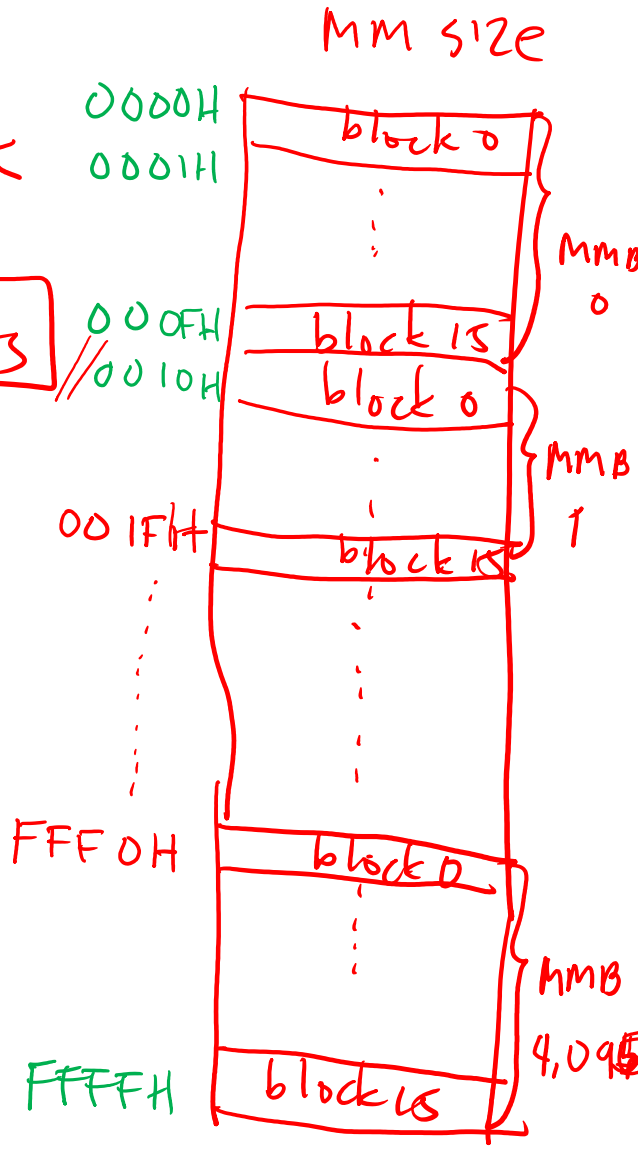
$$n = \frac{\log 65,536}{\log 2} = \boxed{16 \text{ bits}}$$

$$\#\text{ of MMBs} = \frac{\text{MM size}}{\text{block size}} = \frac{65,536 \text{ bytes}}{16 \text{ bytes/block}}$$

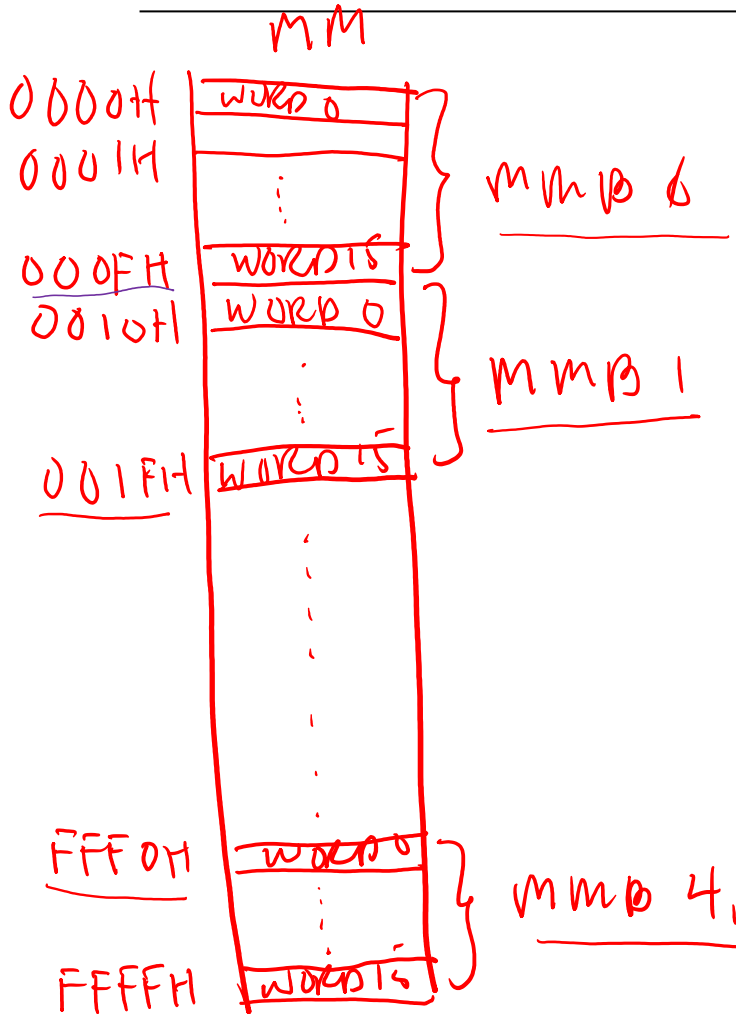
$$\#\text{ of MMBs} = \frac{4,096 \text{ blocks}}{(0 - 4,095)}$$

$$\#\text{ of CBs} = \frac{\text{Cache size}}{\text{block size}} = \frac{2048 \text{ bytes}}{16 \text{ bytes/block}}$$

$$= \frac{128 \text{ blocks}}{(0 - 127)}$$



MMB #	word #



# of mmp bits =

$$= \frac{\log \#\text{ of mmps}}{\log 2}$$

$$= \frac{\log 4,096}{\log 2}$$

$$= 12 \text{ bits}$$

# of block bits = $\frac{\log \text{block size}}{\log 2}$

$$= \frac{\log 16}{\log 2}$$

$$= 4 \text{ bits}$$

# of mm bits = $\frac{\log \text{ of } \# \text{ of mm's}}{\log 2}$

$= \frac{\log 4096}{\log 2}$

$= 12$ bits

# of word bits = $\frac{\log \text{ block size}}{\log 2}$

$= \frac{\log 16}{\log 2}$

$= 4$ bits

EXAMPLE:

MMB # WORD #

1. 001FH = 0000 0000 0001 | 1111
MMB 1 | WORD 15
word 15 of MMB 1

2. 000FH = 0000 0000 0000 | 1111
MMB 0 | WORD 15
word 15 of MMB 0

3. 1234H = 0001 0010 0011 | 0100
MMB 291 | WORD 4
word 4 of MMB 291

n = | MMB # | WORD # |

32-bit CPU

1. 03416H

2. 1ACE1H

3. 2BEA2H

4. 3CA13H

5. 4D1A4H

FFF0H = 1111 1111 1111 | 0000

MMB 4095 | WORD 0

EXAMPLE:

1. 001FH = 0000 0000 0001 | 1111
MMB 1 | word 15

word 15 of MMB 1

2. 000FH = 0000 0000 0000 | 1111
MMB 0 | word 15

3. 1234H = 0001 0010 0011 | 0100
MMB 291 | word 4

2048 1024 256 4 1

1. 03416 = 0000 0011 0100 0001 | 10
word 3,333 | byte 2

byte 2 of word 3,333 or
3rd byte of 3,334th word

2. 1ACE1H = 0001 1010 1100 1110 00 | 01
byte 1 of word 27,449 word 27,448 | byte 1

CACHE MAPPING FUNCTIONS

☞ A *mapping function* specifies where to place a main memory block in the cache.

☞ Case Study:

MM Size = 65,536 words
(16 address bits)

Cache Size = 2,048 words

Block Size = 16 words/block

No. of Cache Blocks = 2,048/16

= 128 blocks
(block 0 to block 127)

No. of MM Blocks = 65,536/16

= 4,096 blocks
(block 0 to block 4,095)

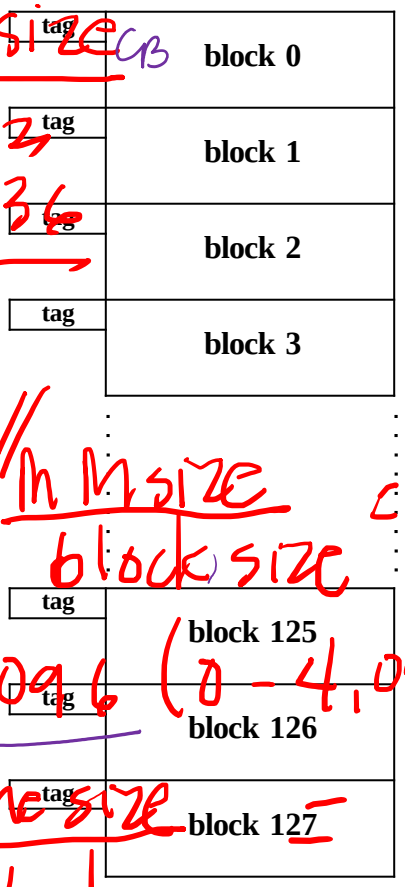
MM size = 65,536 bytes
Cache size = 2,048 bytes
Block size = 16 bytes/block

Mapping Functions

1. *Direct Mapping*

In this technique, block k of the main memory maps onto block k modulo 128 of the cache.

n = | TAG | BLOCK | WORD |



$n = \frac{\log MM\,size_{CB}}{\log_2} = \frac{\log 65536}{\log_2}$

n = 16 bits

# of MMBs = $\frac{MM\,size}{block\,size} = \frac{65,536\ bytes}{16\ bytes/block}$

= 4,096 (0 - 4,095 MMBs)

# of CBs = $\frac{Cache\,size}{block\,size} = \frac{2,048\ bytes}{16\ bytes/block}$

Take note that there will be 32 possible MM blocks that can reside in a particular cache block.

= 128 (0 - 127 CBs)

DIRECT MAPPING

ASSUME: → MMB#

MM SIZE = 65,536 WORDS

CACHE SIZE = 2,048 WORDS

BLOCK SIZE = 16 WORDS / BLOCK

n = 16 bits

# of MMBs = 4,096 (0 - 4,095)

# of CBs = 128 (0-127)

$$\#\text{ of TAG bits} = \frac{\log(\#\text{ of MMBs} / \#\text{ of CBs})}{\log 2}$$

$$= \frac{\log(4,096/128)}{\log 2} = 5 \text{ bits}$$

$$\#\text{ of Block bits} = \frac{\log \#\text{ of CBs}}{\log 2} = \frac{\log 128}{\log 2} = 7 \text{ bits}$$

$$\#\text{ of word bits} = \frac{\log \text{block size}}{\log 2} = \frac{\log 16}{\log 2} = 4 \text{ bits}$$

MMB# = TAG# × # of CBs + Block#

MMB #

1. 1234H = 0001 0 | 010 0011 | 0100 (WORD)

TAG 2 | BLOCK 35 | WORD 4

MMB# = 291

2. DEDEH = 1101 1 | 110 1101 | 1110

2048 1024 512 256 128 64 32 16 8 4 2 1

TAG 27 | BLOCK 109 | WORD 14

MMB# = 3,565

3. 8383H = 1000 0 | 011 1000 | 0011

2,048 32 16 8

TAG 16 | BLOCK 56 | WORD 3

MMB# = 2104

MMB # = TAG # x (# of CBs) + BLOCK #

1. (2 x 128) + 35 = 291
2. (27 x 128) + 109 = 3565
3. (16 x 128) + 56 = 2104

1. 1ACEH = 0001 | 1010 1100 | 1110

TAG 3 | BLOCK 44 | WORD 4

MMB # = 428

2. 2BEAH = 0010 | 1011 1110 | 1010

TAG 5 | BLOCK 62 | WORD 10

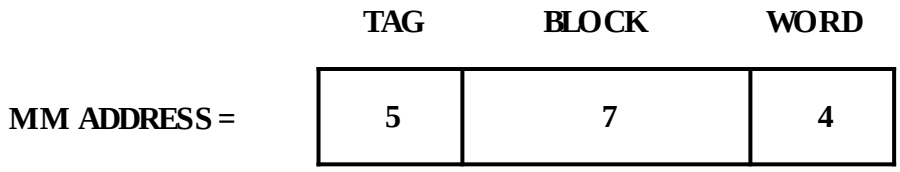
MMB# = 702

---

3. 3CA1H =

4. 4D1AH.

Placement of a block in the cache is determined from the memory address. The memory address can de divided into three fields:



The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit (since there are 2^7 or 128 cache blocks) cache block field determines the cache position in which this block must be stored. The high order 5 bits (since there are 2^5 or 32 possible MM blocks that can reside in a particular cache block) of the memory address of the block are stored in the five *tag bits* associated with its location in the cache.

As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.

The replacement algorithm for the direct mapping technique is trivial. The direct mapping technique is easy to implement but it is not very flexible.

Example:

A 16 MB main memory is to be upgraded by adding a cache memory of 128 KB. Both the main memory and the cache memory are partitioned into blocks of 512 bytes. How many bits make up the tag, block, and word fields of a memory address?

Solution:

No. of MM Blocks = 16 MB/512
= 32,768
No. of Cache Blocks = 128 KB/512
= 256

No. of MM Blocks per Cache Block
= 32,768/256
= 128

No. of Tag Bits = log 128 / log 2
= 7 bits
No. of Block Bits = log 256 / log 2
= 8 bits
No. of Word Bits = log 512 / log 2
= 9 bits

Assume:

TAG | WORD

2. *Associative Mapping*

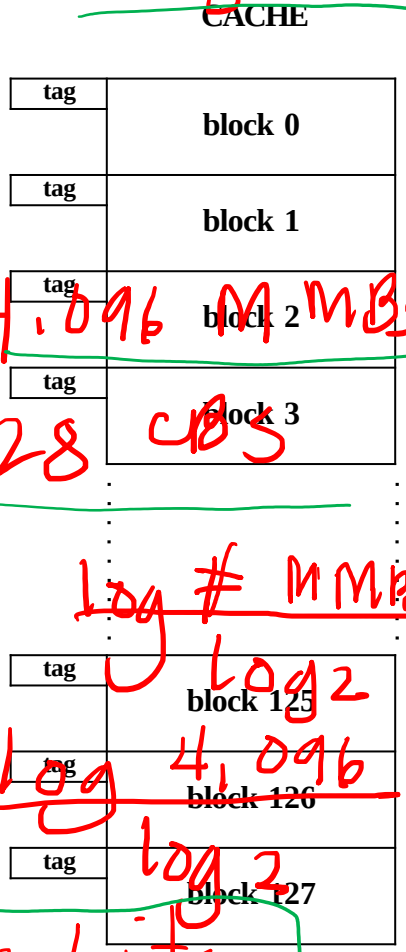
MMsize = 65,536 byte

TAG # = MMB #

Cache size = 2,048 bytes

A main memory block can potentially reside in any cache block position.

block size = 16 byte/block



n = 16 bits

# of MMBs = 4,096 MMBs (0 - 4095)

# of CBs = 128 CBs (0 - 127)

TAG bits = log # MMBs / log 2

= log 4,096 / log 2

= 12 bits

Take note that there will be 4,096 possible main memory blocks that can reside in a particular cache block.

# of word bits = log blocksize / log 2

= log 16 / log 2 = 4 bits

1. 1234H = 0001 0010 0011 | 0100

TAG # = 291 | WORD 4

MMB # = 291

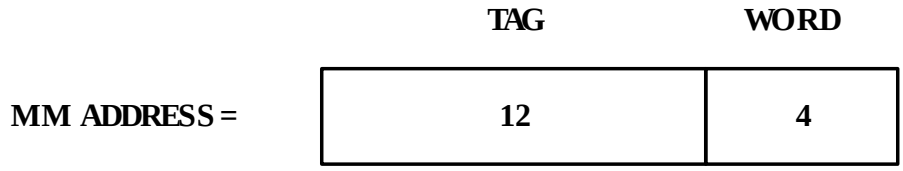
2^{048} 512 128 32 16 8 4

2. ABCD H = 1010 1011 1100 | 1101

TAG 2748 | WORD 13

MMB # 2748

This scheme requires 12 tag bits (because there are 2^{12} or 4,096 possible main memory blocks that can reside in a particular cache block) to identify a main memory block when it is resident in the cache. The system will compare the tag bits of an address received from the processor to the tag bits of each block of the cache to see if the desired block is present.



Since there is complete freedom in block positioning, a wide range of replacement algorithms is possible. However, it might not be practical to make full use of this freedom, because complex replacement algorithms may be difficult to implement.

The cost of an associative cache is higher than the cost of a direct-mapped cache because of the need to search all 128 tag patterns to determine whether a given block is in the cache. A search of this kind is called an *associative search*. For performance reasons, the tags must be searched in parallel.

Example:

A 16 MB main memory is to be upgraded by adding a cache memory of 128 KB. Both the main memory and the cache memory are partitioned into blocks of 512 bytes. How many bits make up the tag and word fields of a memory address?

Solution:

No. of MM Blocks = 16 MB/512

= 32,768

No. of Cache Blocks = 128 KB/512

= 256

No. of Tag Bits = log 32,768 / log 2

= 15 bits

No. of Word Bits = log 512 / log 2

= 9 bits

Assume:

MM size = 65,536 bytes

Cache size = 2048 bytes

Block size = 16 byte/block

n = | TAG | SET | WORD |

3. *Block-Set-Associative Mapping*

Blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any cache block of a specific set.

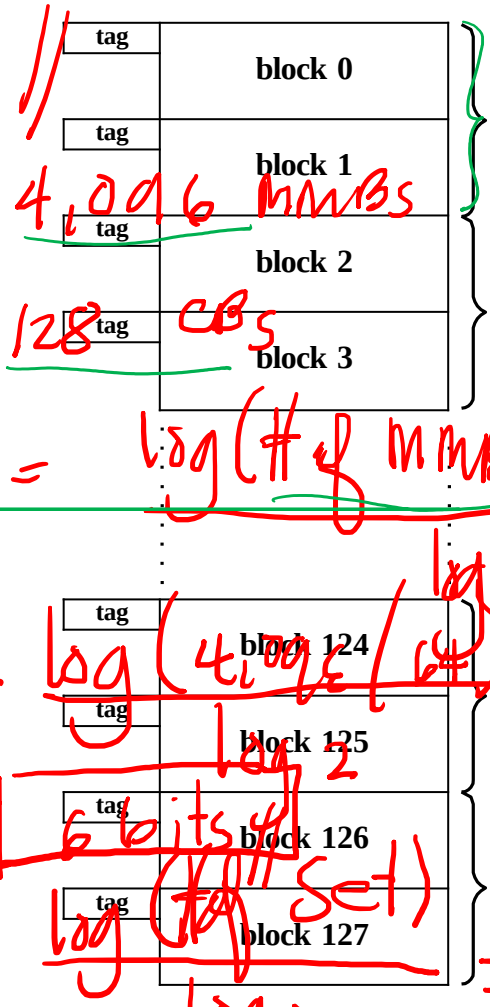
Example: Assume 2 cache blocks per set

# of SET = 128 CBs / 2 CBs/SET

CACHE = 64 SET

n = log 65,536 / log 2

n = 16 bits



# of MMBs = 4,096 MMBs

# of CBs = 128 CBs

# of TAG bits = log(# of MMBs / # of set) / log 2

= log(4,096 / 64) / log 2

= log 64 / log 2 = 6 bits

# of set bit = log(# of set) / log 2 = 6 bits

# of word bit = log 16 / log 2 = 4 bits

$$\#\text{ of TAG bits} = \frac{\log(\#\text{ of MMBs/set})}{\log 2}$$

$$= \frac{\log(4,096/64)}{\log 2}$$

$$= 6 \text{ bits}$$

$$\#\text{ of set bit} = \frac{\log(\#\text{ of set})}{\log 2} = \frac{\log 64}{\log 2}$$

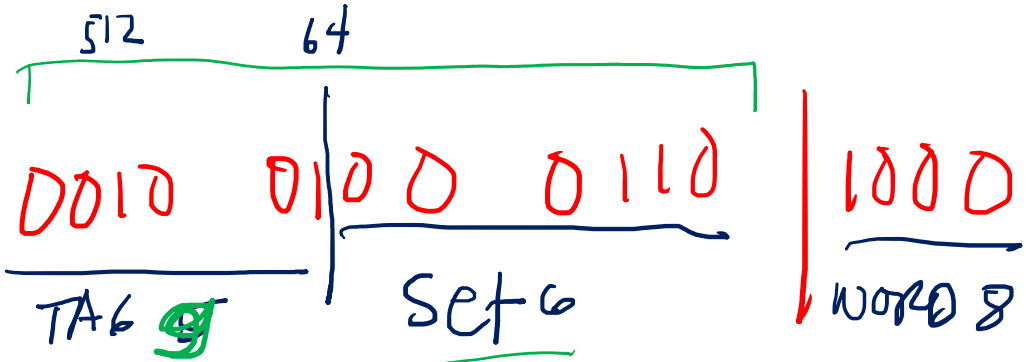
$$= 6 \text{ bits}$$

$$\#\text{ of word bits} = \frac{\log \text{ block size}}{\log 2}$$

$$= \frac{\log 16}{\log 2} = 4 \text{ bits}$$

Example:

1. 2468H =



MMB # = 582

BSAM

MMB# = TAG# × #of set + set #

= 9 × 64 + 6 = 582

(4 × 64) + 35

1. 1234H = 0001 0010 0011 0100

TAG 4 | set 35 | WORD 4

MMB# = 291

2. 1ACEH = 0001 1010 1100 1110

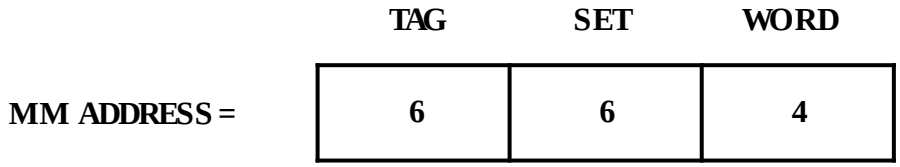
TAG 6 | SET 44 | WORD 14

MMB# = 428

2. (4 × 64) + 35 = 291

3. (6 × 64) + 44 = 428

The contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search.



The 6-bit set field (since there are 2^6 or 64 sets) of the address determines which set of the cache might contain the desired block. The 6-bit tag field (since there are 2^6 or 64 possible main memory blocks that can reside in a particular cache set) of the address must then be associatively compared to the tags of the two blocks of the set if a match occurs signifying block presence.

The extreme condition of having 128 blocks (all cache blocks) per set corresponds to the fully associative technique. The other extreme of having one block per set is the direct-mapping technique.

The block-set-associative mapping technique is the most practical cache mapping technique.

ASSIGNMENT

DM

AM

BSAM

1. 12345H

2. 369BDH

3. 2468AH

4. ABCDEH

5. 41073H

Example:

A 16 MB main memory is to be upgraded by adding a cache memory of 128 KB. Both the main memory and the cache memory are partitioned into blocks of 512 bytes. Assume that there are 8 blocks to a set. How many bits make up the tag, set, and word fields of a memory address?

Solution:

No. of MM Blocks = 16 MB/512
= 32,768

No. of Cache Blocks = 128 KB/512
= 256

No. of Sets = 256/8
= 32

No. of MM Blocks per Set
= 32,768/32
= 1,024

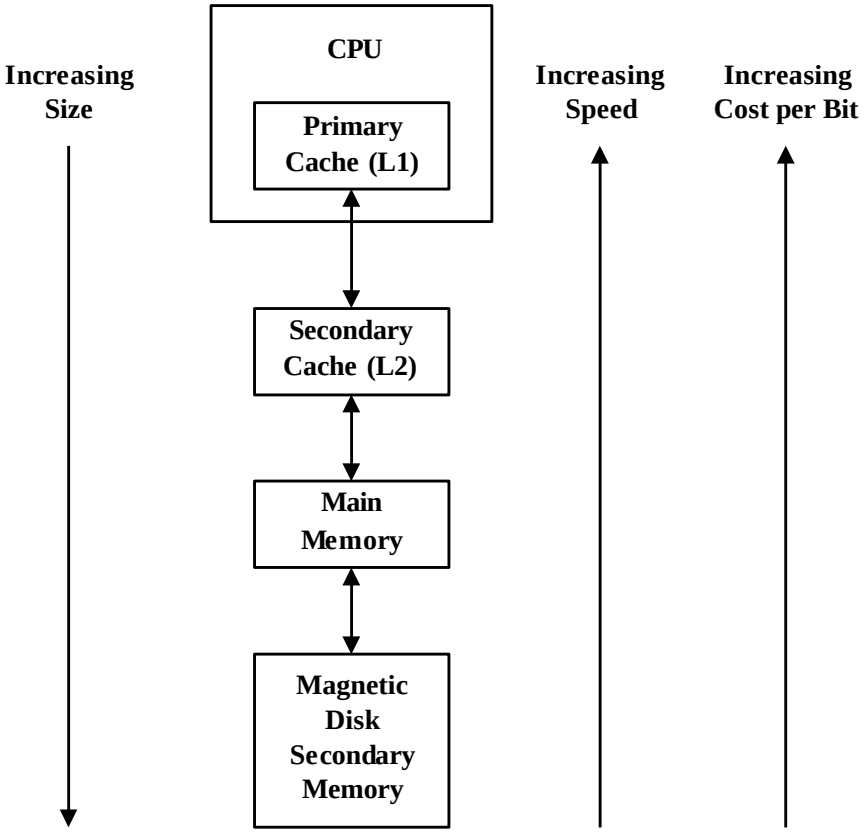
No. of Tag Bits = log 1,024 / log 2
= 10 bits

No. of Set Bits = log 32 / log 2
= 5 bits

No. of Word Bits = log 512 / log 2
= 9 bits

MEMORY HIERARCHY

- The entire memory can be viewed as the hierarchy depicted below:



- The fastest access to data is through the processor registers. If registers are considered to be part of the memory hierarchy, then the processor registers are at the top in terms of speed of access. Of course, the registers provides only a minuscule portion of the required memory.

- At the next level is the cache memory. There are two types of cache memory. A *primary cache* is located on the processor chip. This cache is small, because it competes for space on the CPU chip, which must implement many other functions. The primary cache is referred to as *Level 1* (*L1*) cache.

  A *secondary cache* is placed between the primary cache and the main memory. It is referred to as *Level 2* (*L2*) cache.
  Including a primary cache on the processor chip and using a larger, off-chip, secondary cache is the most common way of designing computers. However, other arrangements can be found in practice. It is possible not to have a cache on the processor chip at all. Also, it is possible to have both L1 and L2 caches on the processor chip.

- The next level in the hierarchy is the main memory. It is much larger but significantly slower than the cache memory. In a typical memory, the access time for the main memory is about ten times longer than the access time for the L1 cache.

- Disk devices provide a huge amount of inexpensive storage (secondary storage). They are very slow compared to the semiconductor devices used to implement main memory.

☞ Pentium 4 Caches

The Pentium 4 processor can have up to three levels of caches.

The L1 cache consist of separate data and instruction caches. The data cache has a capacity of 8 KB, organized in a 4-way set- associative manner. Each cache block has 64 bytes. The write-through policy is used to write to the cache.

Data can be accessed from the data cache in two clock cycles. For a 1.4 GHz Pentium, data can be accessed in less than 1.5 ns. The instruction cache does not hold normal machine instructions. Instead, it holds already decoded versions of the instructions.

The L2 cache is a unified cache with a capacity of 256 KB, organized in an 8-way set-associative manner. Each of its block comprises 128 bytes. The write-back policy is used to write to the cache. The access latency of this cache is seven clock cycles.

Both L1 and L2 caches are implemented on the processor chip. The architecture also allows for inclusion of an on-chip L3 cache. However, this cache is not implemented in the Pentium 4 chips targerted for desktop computers. It is intended for processor chips used in server systems.

PERFORMANCE CONSIDERATIONS

- Two key factors in the commercial success of a computer are performance and cost; the best possible performance at the lowest possible cost is the objective.

- A common measure of success is the *price/performance ratio*.

- Performance depends on how fast machine instructions can be brought into the processor for execution and how fast they can be executed.

- The memory hierarchy results from the quest for the best price/performance ratio. The main purpose of this hierarchy is to create a memory that the processor sees as having a short access time and a large capacity.

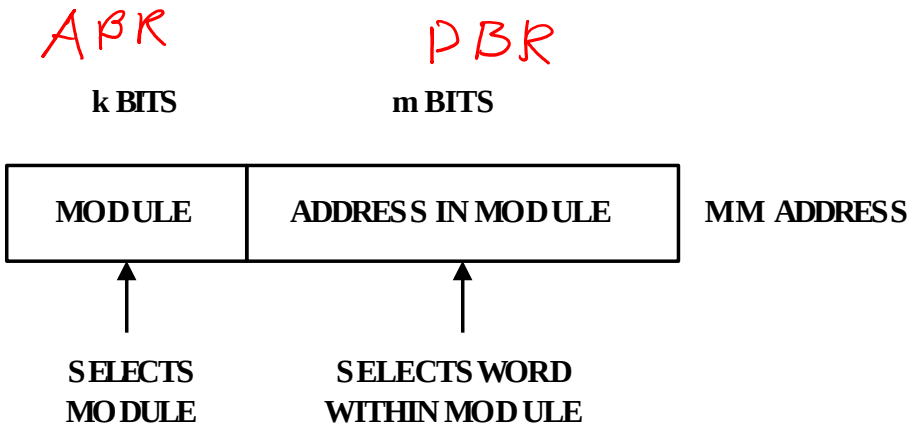
- The speed and efficiency of data transfer between various levels of the hierarchy are also of great significance. It is beneficial if transfers to and from the faster units can be done at a rate equal to that of the faster unit. This is not possible if both the slow and the fast units are accessed in the same manner, but can be achieved when parallelism is used in the organization of the slower unit.

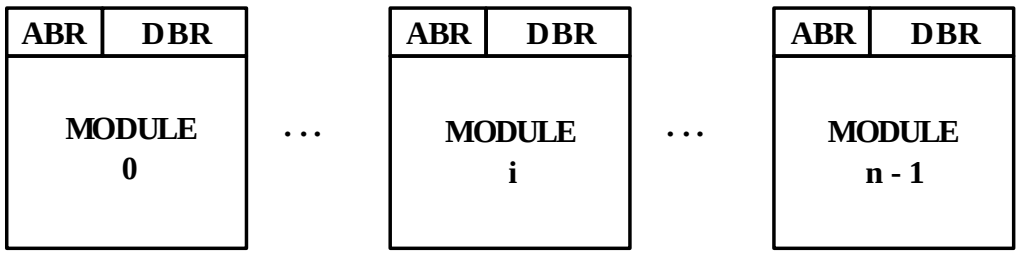
- An effective way to introduce parallelism is to use an *interleaved organization*.

☞ Interleaving

If the main memory of a computer is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same time.

Option 1: Consecutive Words in a Module





Example: MM Size = 1,048,576 locations
(20 address lines or bits)

MMsize = 1M (1,048,576 bytes)

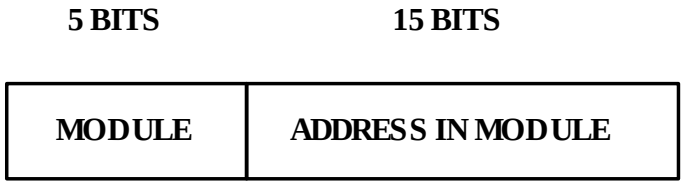
Option 1 divides the main memory into 32 modules of 32,768 locations each.

~~Module~~ size = 32,768 byte/module

The high-order 5 bits selects one of the 32 modules while low-order 15 bits select one of the 32,768 locations in that module.

$n = \frac{\log 1,048,576}{\log 2}$

n = 20 bits //



When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is busy. At the same time, however, devices with direct memory access (DMA) capability may be accessing information in other memory modules.

~~# of Modules~~ = $\frac{MMsize}{Module\ size} = \frac{1,048,576\ byte}{32,768\ byte}$ modu

= 32 modules

$$\#\text{ of address in a Module bits} = \frac{\log \#\text{ of Module size}}{\log 2}$$

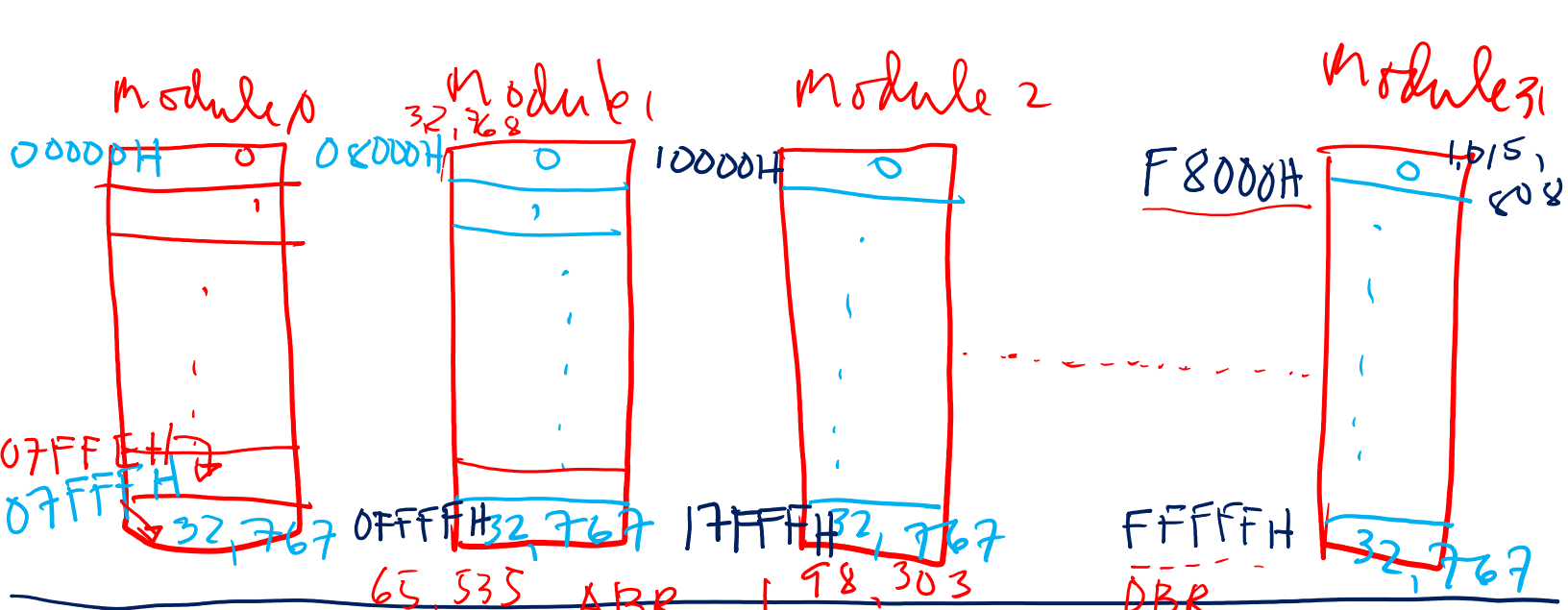
$$= \frac{\log 32,768}{\log 2}$$

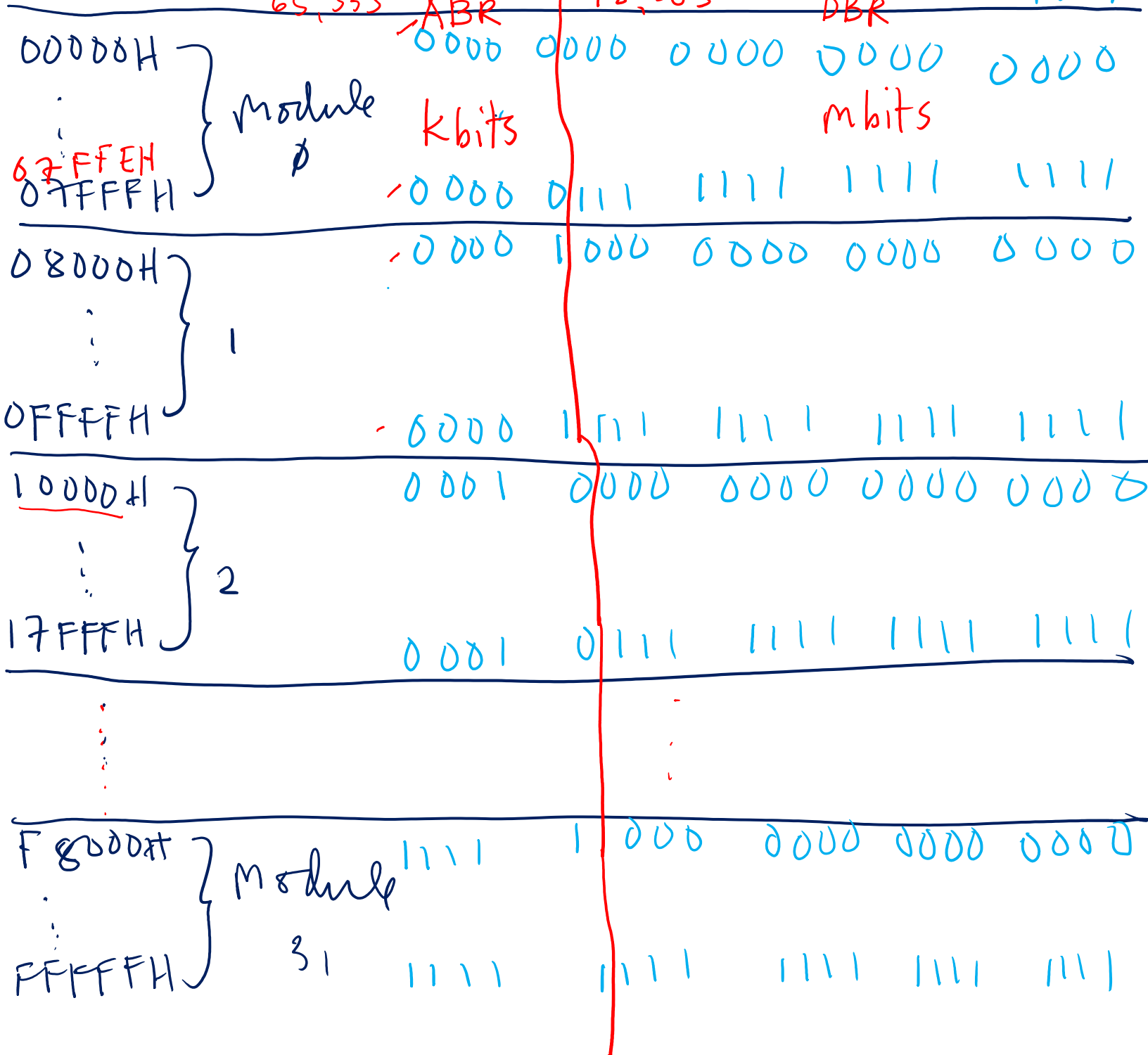
$$= 15 \text{ bits}$$

$$\#\text{ of Module bits} = \frac{\log (\#\text{ of Modules})}{\log 2}$$

$$= \frac{\log 32}{\log 2}$$

$$= 5 \text{ bits}$$


module 0
module 1
module 2
module 31
00000H
08000H
10000H
F8000H
07FFEH
07FFFH
32,767
32,768
65,535
98,303
1,015,808
0FFFFH
17FFFH
FFFFFH


ABR
DBR
kbits
mbits
00000H
07FFEH
07FFFH
module 0
08000H
0FFFFH
1
10000H
17FFFH
2
F8000H
FFFFFH
module 31

Example:

1. 07FFFH

0000 | 0111 1111 1111 1111

Module 0 | address in a module

32,767

2. 10000H =

0001 | 0000 0000 0000 0000

Module 2 | address in a module 0

3. 12345H =

0001 | 0010 0011 0100 0101

Module 2 | 9,029

Example: 2

1. word in a module 3 of module 1

0000	0000	0000	0000	0000
0000	1000	0000	0000	0011
0	8	0	0	3

→ 08003H

2. word in a module 5 of module 16

3. 10883H = 0001 0000 0000 0000 0011

module 2 | words in module 3

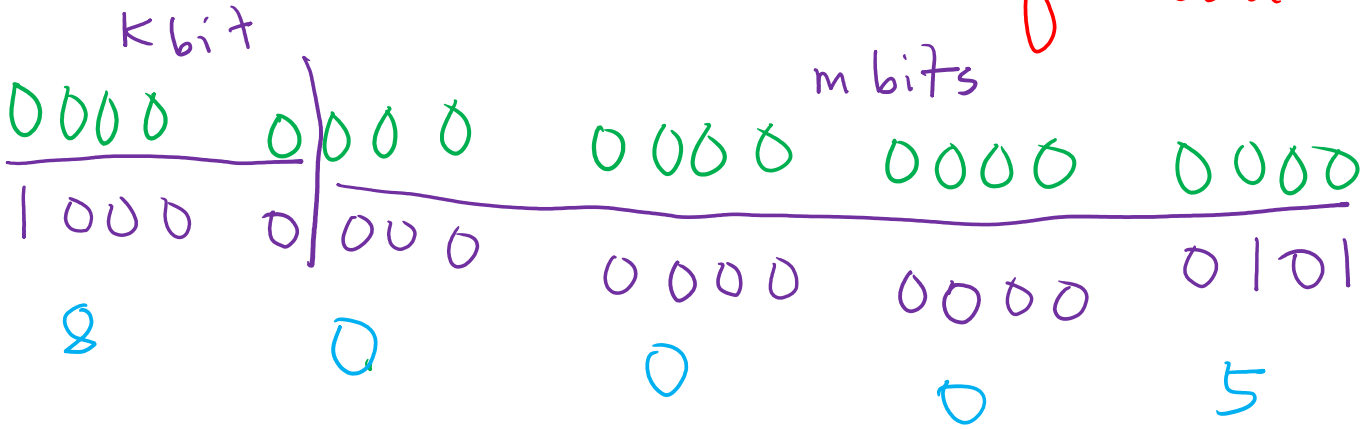
module 9

4. Word in a module 80 of the 10th module

0000	0000	0000	0000	0000
0100	1000	0000	0101	0000
4	8	0	5	0

48050H ✓

2. word in a module 5 of module 16



80005H

last words in a module of module 20

0000	0000	0000	0000	0000
1010	0111	1111	1111	1111
A	7	F	F	F H

A 7FFFH

module 20 ≠ 20th module ≠ 21st module ≠ module 19

Exercise:

1. 1049A H

1024 128 16 8 2

0001 0000 0100 1001 1010

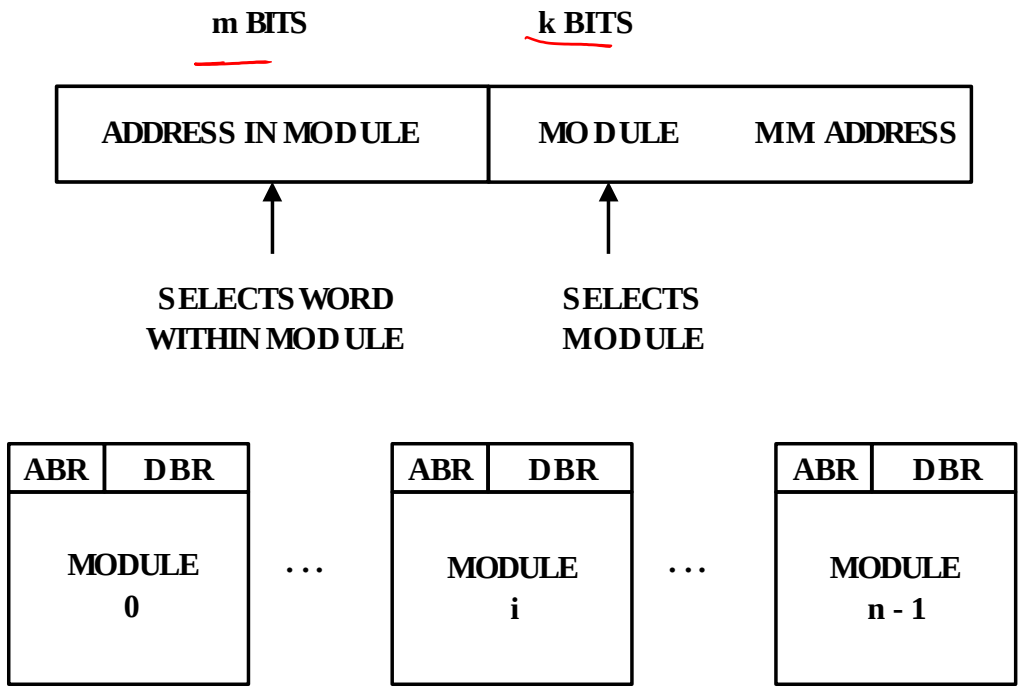
module 2 | words in a module 1178

2. words in a module 300 of the 20th module

0000 0000 0000 0000 0000

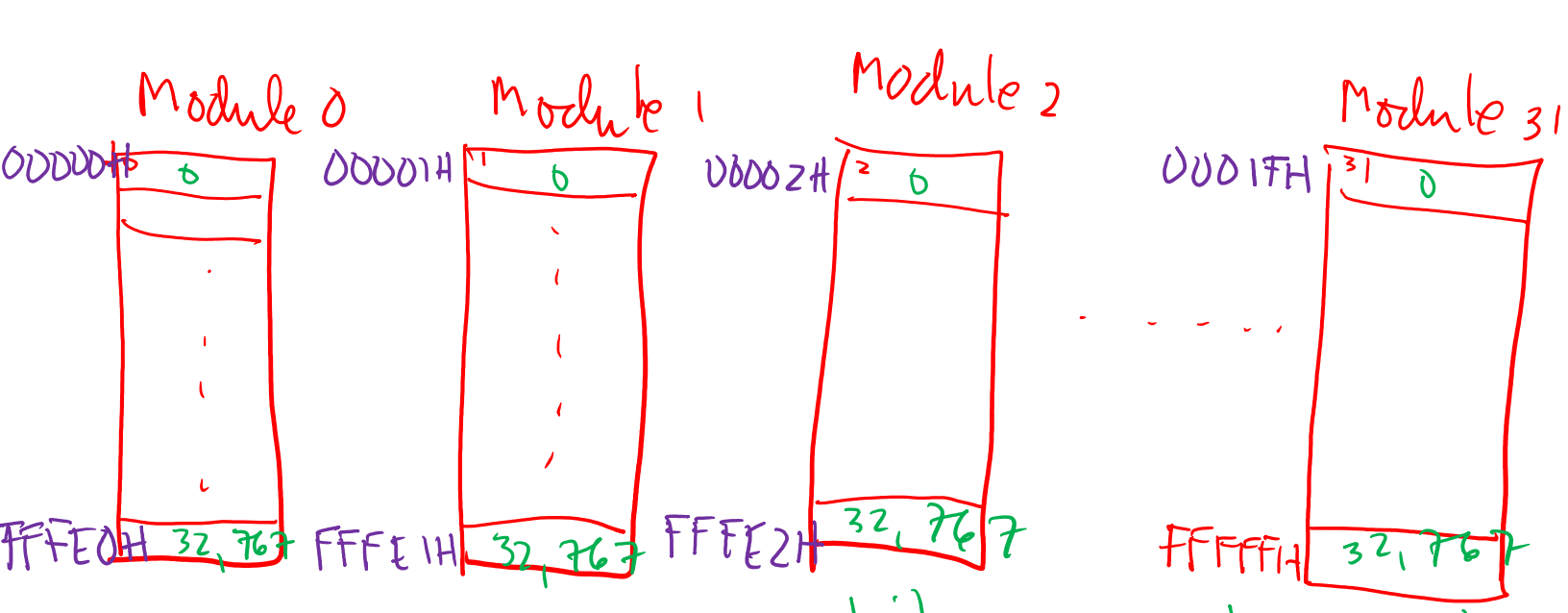
1001 1000 0001 0010 1100 = 98 12CH

Option 2: Consecutive Words in Consecutive Modules or Memory Interleaving



In the second option, the low order bits of the memory address select the a module, and the high order bits select a location within that module. In this way, consecutive addresses are located in successive modules. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at one time. This results in both faster access to a block of data and higher average utilization of the memory system as a whole.

To take advantage of memory interleaving, the system must be capable of initiating a memory access operation while waiting for the completion of a previous memory access.



00000H
⋮
FFFE0H
} 0

0000 0000 0000 0000 0000

1111 1111 1111 1110 0000

00001H
⋮
FFFE1H
} 1

0000 0000 0000 0000 0001

1111 1111 1111 1110 0001

00002H
⋮
FFFE2H
} 2

0000 0000 0000 0000 0010

1111 1111 1111 1110 0010

001FH
⋮
FFFFFH
} 31

0000 0000 0000 0001 1111

1111 1111 1111 1111 1111

Examples:

002014H = 0000 0000 0000 0010 | 0000

words in a module | module

address in a module | 0

1

12345H = 2048 256 16 8 2

0001 0010 0011 0100 | 0101

address in a module | module 5

2330

1 2
512
256
128
4

900

words/address in a module 900 of the last module

0000 0000 0000 0000 0000

0000 0111 0000 1001 1111 = 079FH

256

0 | 512 7 | 128 0 | 4 9 | F

EXERCISE:

231st address in a module = address in a module 230

1. 340F9H =

0011 0100 0000 1111 1001

address in a module | module 25

6663

2. 260th address in a module of the 30th module

0000 0000 0000 0000 0000

0000 0010 0000 0111 1101

020 7DH

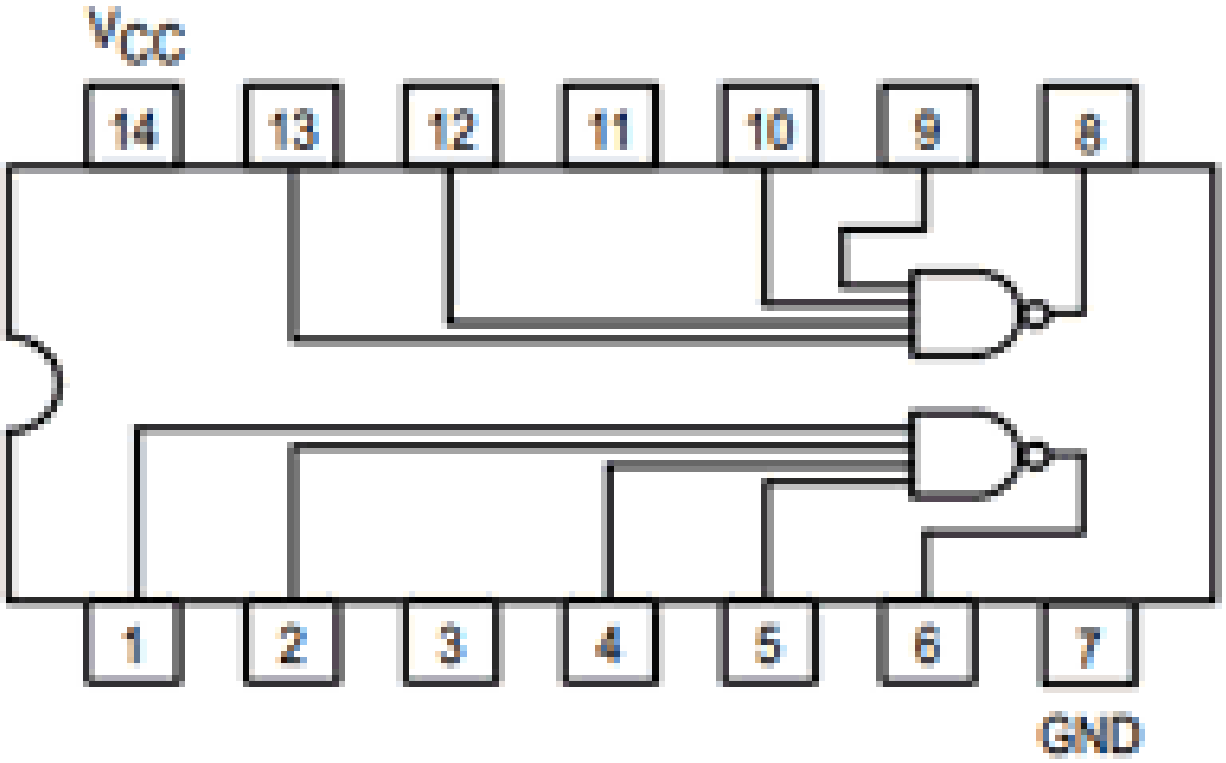
ordinal number

1st module = module 0 ✓

2nd module = module 1 ✓

29th module = module 28 ✓

This is a 4-input NAND gate-74LS20
Pin Configuration



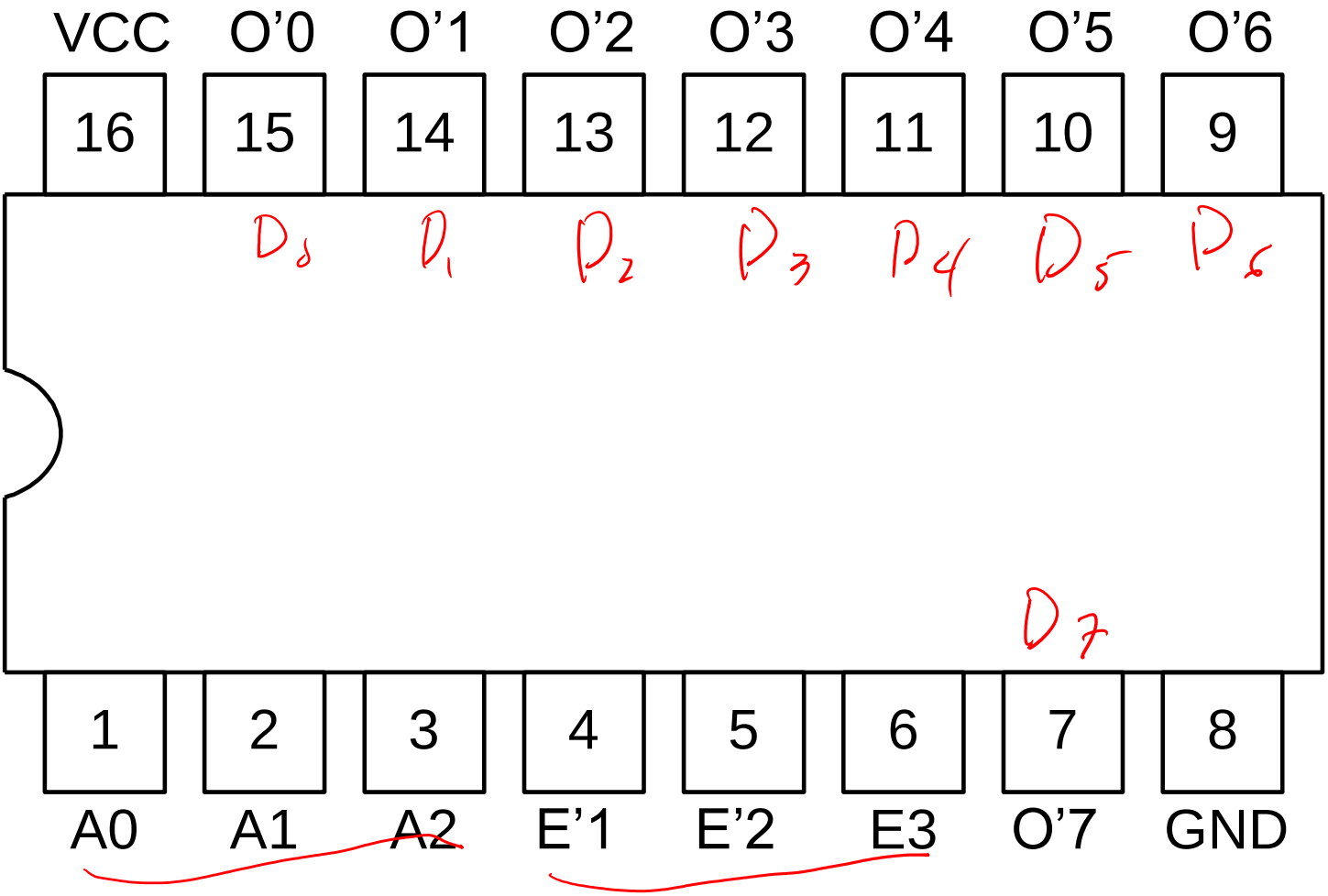
4 2 1

2^2 2^1 2^0

a	b	c	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

n- to- 2^n decoder

74LS138 Pin Configuration



Ungsod Lim
Discaya
Mahaguay
Tañedo
Javero

OPTION 1

OPTION 2

---

1. 3CA13H = 0011 1100 1010 0001 0011
   module 7 | 18963

2. 0A1B2H

3. last word in a module of module 9

Option 2:

3CA13H = 0011 1100 1010 0001 | 0011
(2048, 512, 64, 16; 4096, 1024)
word in a module | module 19

0A1B2H = 0000 1010 0001 1011 | 0010
(81927,760; 256, 128, 16, 2; 32)
module 1 | word in a module 8626

0000 1010 0001 1011 | 0010
(1024, 256, 8, 4, 1)
word in a module 1293 | module 18

last word in a module of module q

OPTION 1:

0000	0000	0000	0000	0000
0100	1111	1111	1111	1111
4	F	F	F	F

4FFFFH

OPTION 2:

0000	0000	0000	0000	0000
1111	1111	1111	1110	1001
F	F	F	E	9

FFFE9H