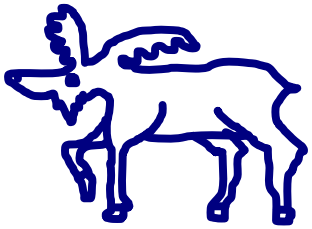
Lecture 21
Direct-Mapped Cache

Byung-gi Kim

School of Computer Science and Engineering

Soongsil University



5. Large and Fast: Exploiting Memory Hierarchy

Direct Mapped Cache

- Each memory location is mapped to exactly one cache location
  - Many lower level blocks must share blocks in the cache
- Cache address (i.e. Index)

  (Block address) *modulo* (Number of blocks in the cache)
- When the number of entries in the cache is 2^N,

  Cache address = low-order N bits of the memory address
- Fields on a cache line
  - Valid bit
  - Data
  - Tag
    - The information required to identify the original location
    - High-order part of memory address except index

Accessing a Cache

- **Example: 8-byte direct mapped cache**

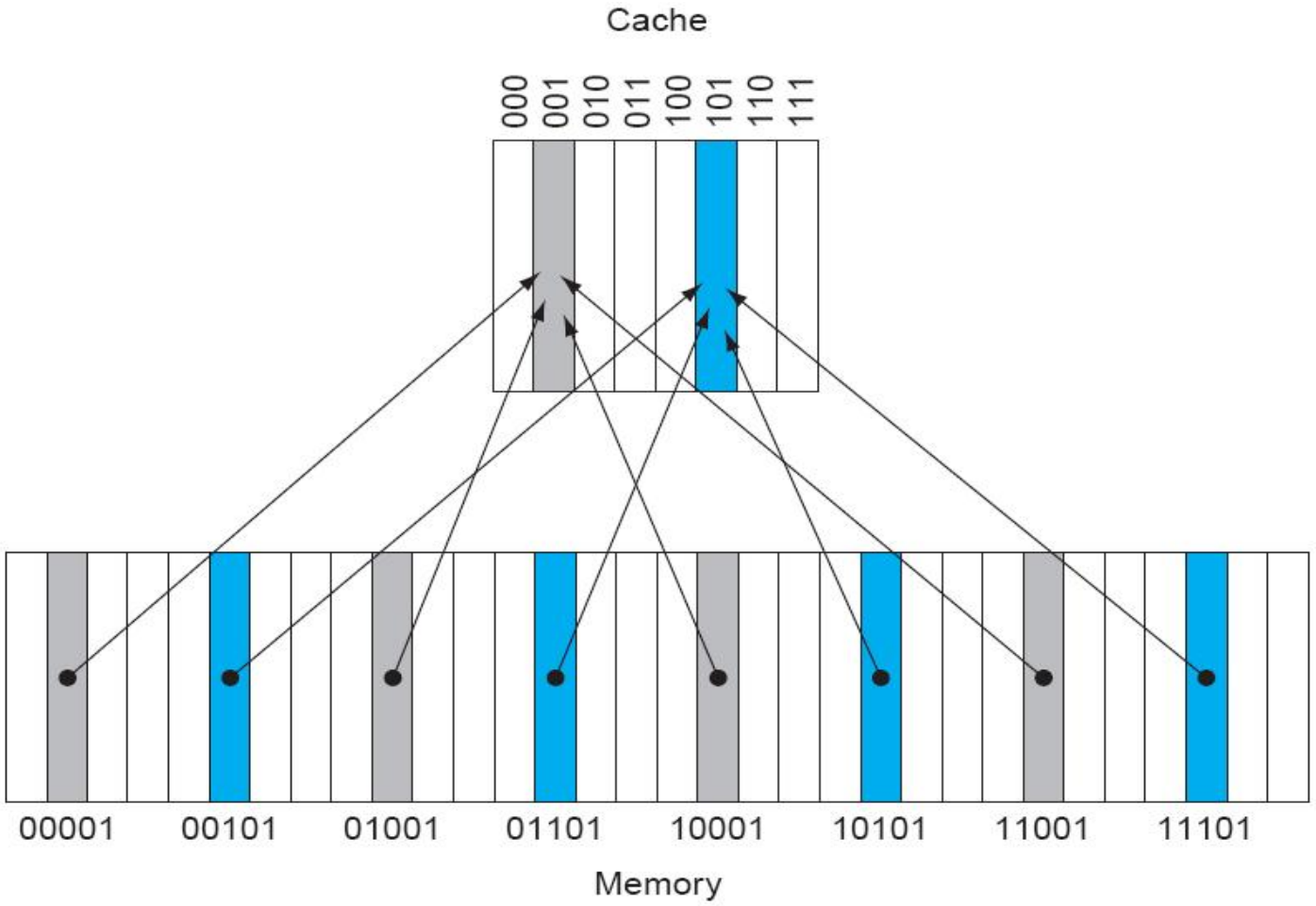


Figure 5.5

States of the Cache

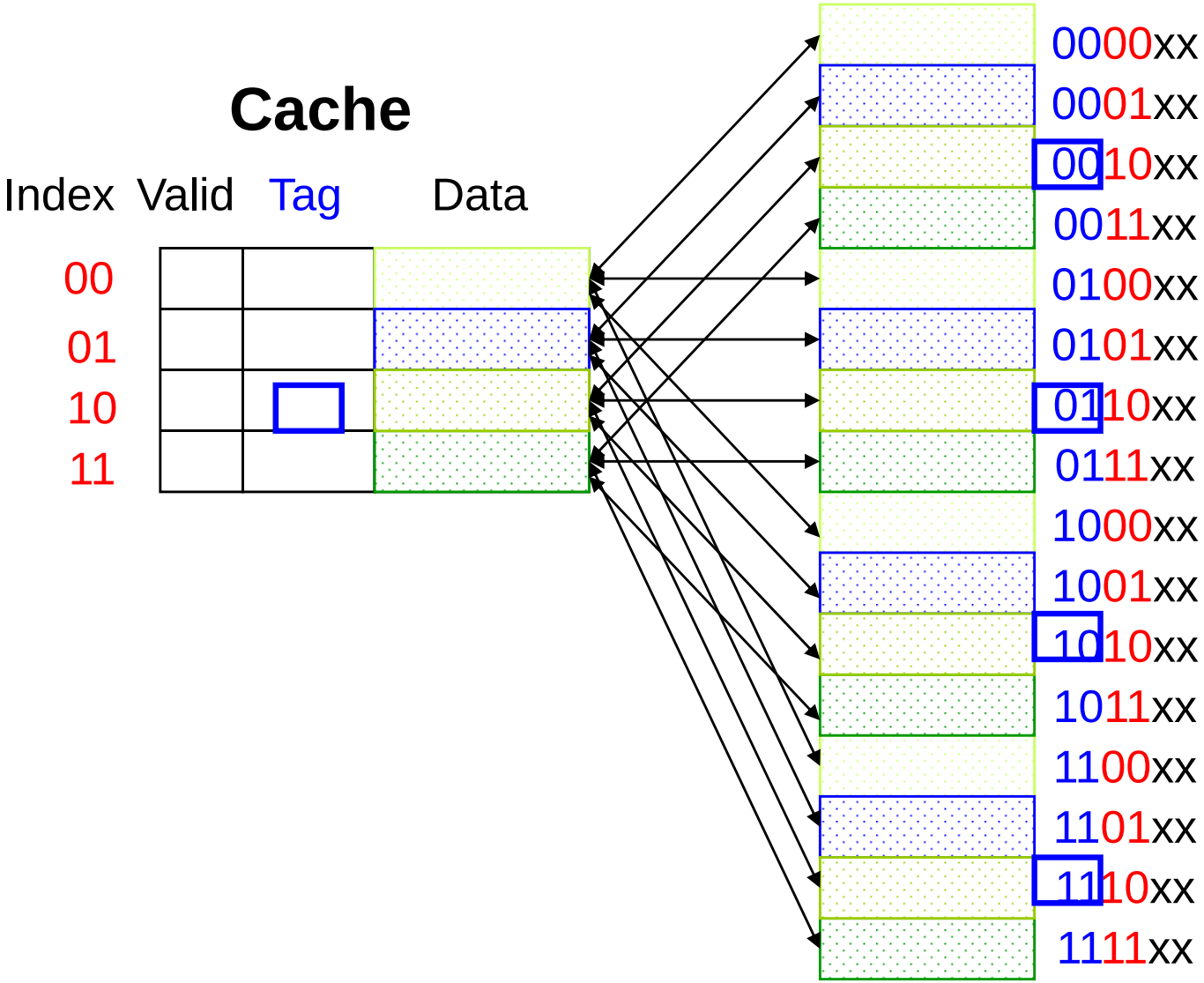
Index	V	Tag	Data
000	Y	10	M[16]
001			
010	Y	11	M[26]
011	Y	00	M[3]
100			
101			
110	Y	10	M[22]
111			

22	10110
26	11010
22	10110
26	11010
16	10000
3	00011
16	10000
18	10010

Example Sequence of References

Decimal addr.	Binary addr.	Hit or miss	Assigned cache block
22	10110	miss	110 = 6
26	11010	miss	010 = 2
22	10110	hit	110 = 6
26	11010	hit	010 = 2
16	10000	miss	000 = 0
3	00011	miss	011 = 3
16	10000	hit	000 = 0
18	10010	miss	010 = 2

4-Block Cache with 4-Byte Blocks



1-word blocks
→ 4-byte blocks
→ 2 low order bits define the byte in the word (offset)

Cache with 1-Word Block

- **32-bit address**
- **2^n words with one-word blocks**
- **Byte offset**

  Least significant 2 bits (not used in cache)
- **Index**

  Next n bits
- **Tag**

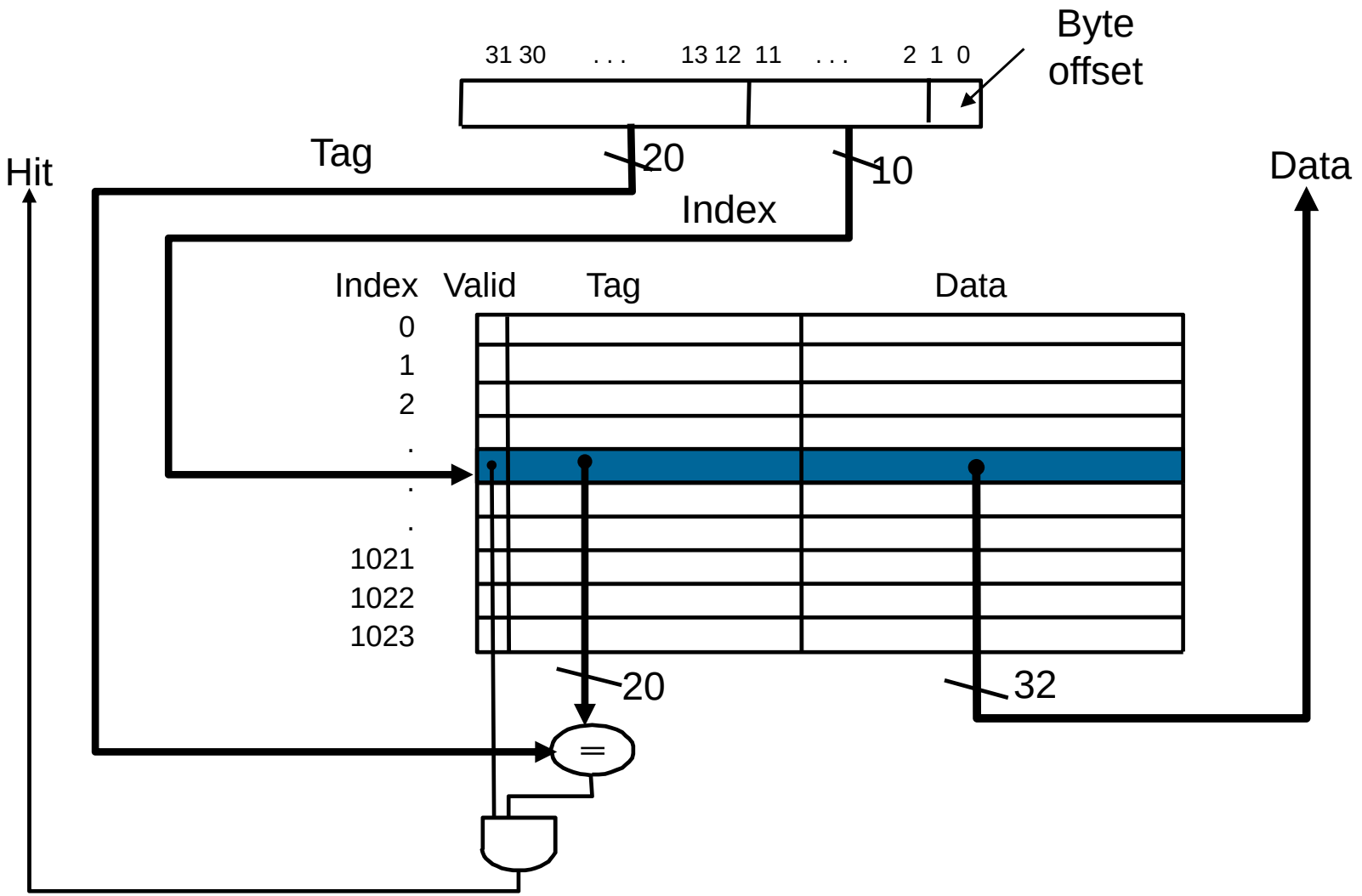
  32 - (n + 2) bits
- **Total number of bits**

  2^n x (valid bit + tag + block size)

  = 2^n x (1 + (32-n-2) + 32)

  = 2^n x (63-n)

Example: 4KB Cache with 1-Word Block



Cache with 2^m-Word Block

- 32-bit address
- 2^n blocks with 2^m-word blocks
- Byte offset : 2 bits
- Block offset : m bits
- Index : n bits
- Tag : 32 - (n + m + 2) bits
- Total number of bits

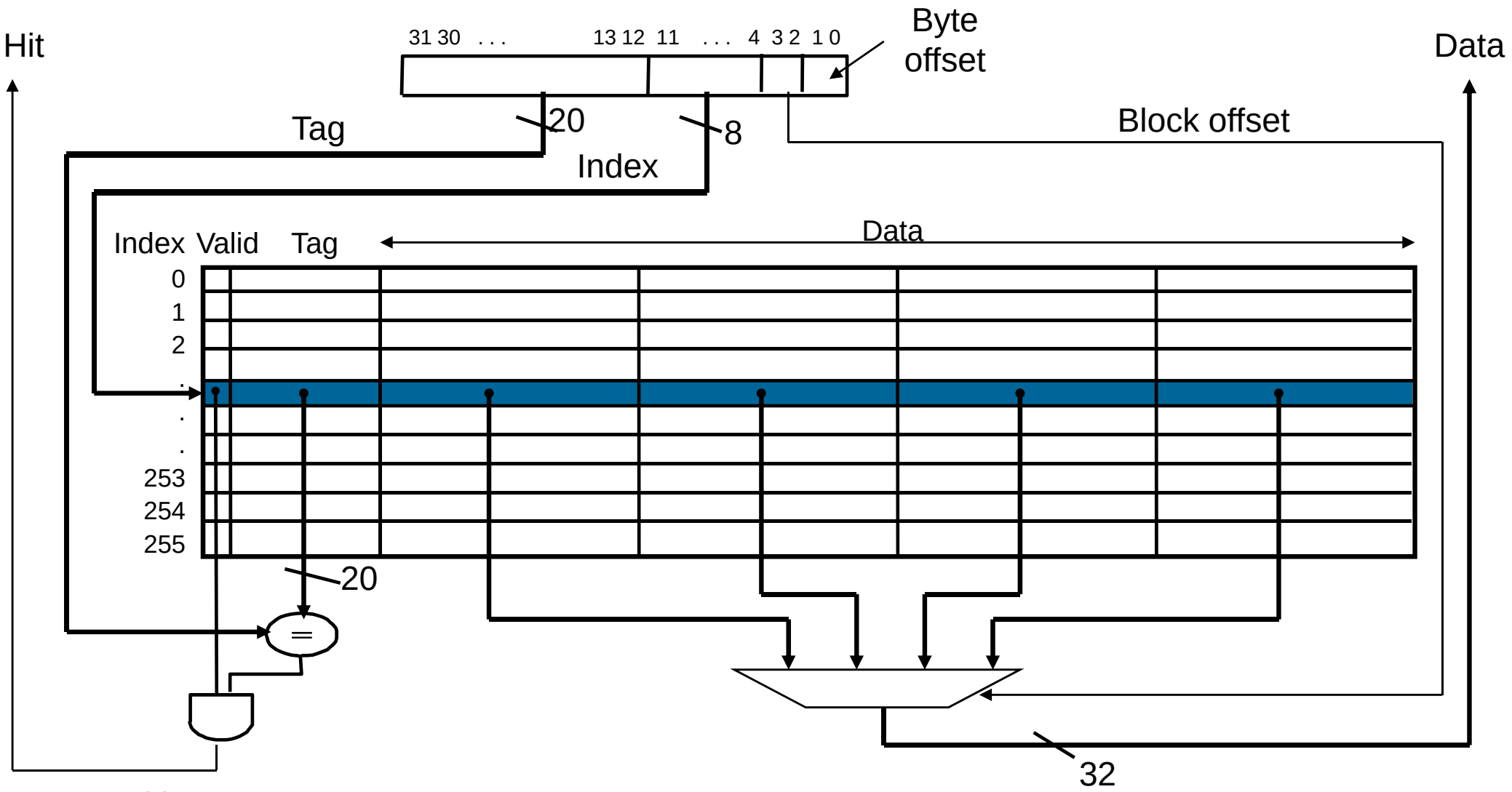
2^n x (block size + tag size + valid field size)

$= 2^n$ x (2^mx32 + (32-n-m-2) + 1)

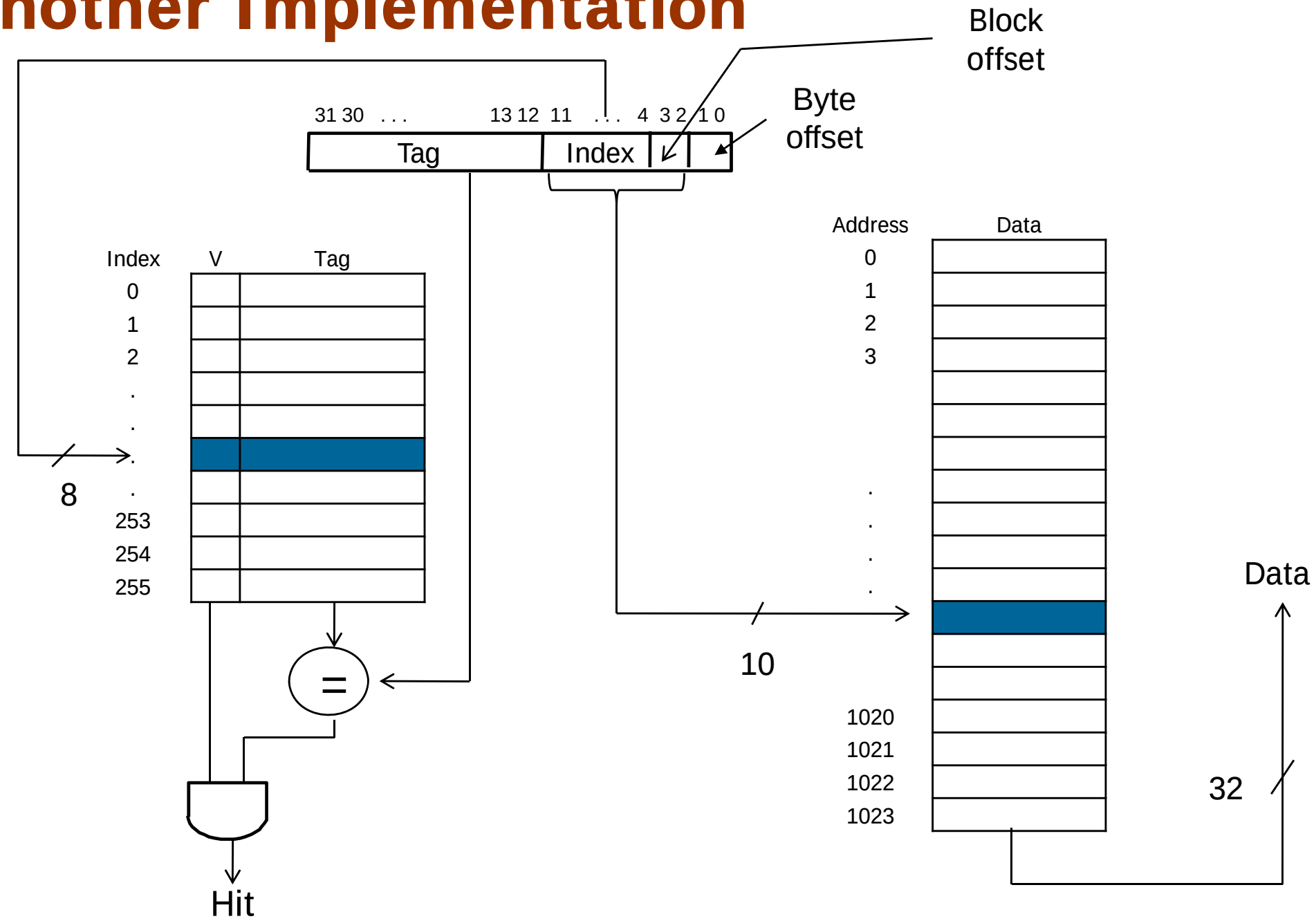
$= 2^n$ x (2^mx32 + 31 – n - m)

Multiword Block Direct Mapped Cache

- 2^2 words/block, cache size = 4 KB (m=2, n=8)



Another Implementation



Direct Mapped Cache (Block Size=1)

- **Reference string: 0 1 2 3 4 3 4 15**

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

~~00~~ (01)	~~Mem(0)~~ (4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
~~00~~ (11)	~~Mem(3)~~ (15)

- 8 requests, 6 misses

Direct Mapped Cache (Block Size=2)

- **Reference string: 0 1 2 3 4 3 4 15**

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

~~00~~ 01	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

11 01	Mem(5)	Mem(4)
~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- 8 requests, 4 misses

Example: Bits in a Cache

- **Calculate total number of bits for the cache.**
  - Direct-mapped
  - 16 KB of data with 4-word blocks
  - 32-bit address

[Answer]

16 KB = 4K words = 2^{12} words

Number of blocks = 2^{12} / 4 = 2^{10}

Block size = 4 words = 4x4 bytes = 4x4x8 bits = 128 bits

Cache size = 2^{10} x (128 + (32-10-2-2) + 1)

= 2^{10} x 147

= 147 Kbits = 18.4 KBytes

Almost 1.15 times as large as data size of the cache.

Miss rate vs. Block size

- **Larger blocks**
  - Increased spatial locality
  - Decreased miss rate
  - But ?

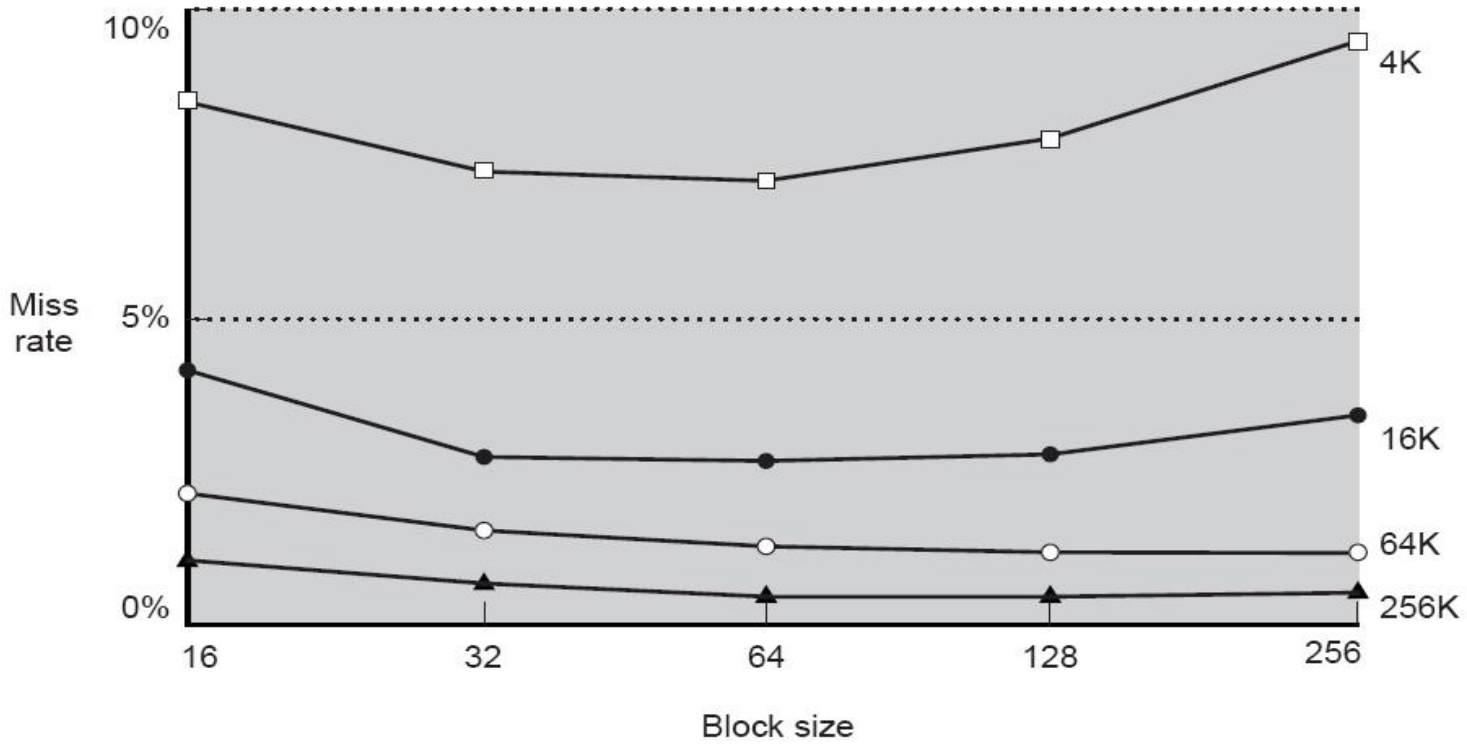


Figure 5.8

Block Size vs. Performance

- **Too large blocks**
  - Small number of blocks in the cache
  - Increased miss rate
  - Increased miss penalty
- **Miss penalty**
  - Miss penalty = block fetch time + cache load time
  - Block fetch time = latency to the first word
    + transfer time for the rest of the block
  - Transfer time $\propto$ block size
  - Increased as the block size increases

Handling Cache Misses

- **Handling instruction cache miss**
  1) Send PC-4 to the memory.
  2) Instruct main memory to perform a read and wait for the memory to complete its access.
  3) Write the cache entry. (Update data and tag fields and turn on the valid bit.)
  4) Restart the instruction execution at the first step.
- **Data cache miss**
  - Stall the processor until the memory responds with the data.

Handling Writes

- **Write-through policy**
  - Always write the data into both the memory and the cache.
  - Keep the main memory and the cache consistent.
- **On write-miss**
  1. Fetch the block containing the word.
  2. Overwrite the word into the cache block.
  3. Also write the word to main memory.
- **On write-hit**
  1. Overwrite the word into the cache block.
  2. Also write the word to main memory.

Problems of Write-through

- **Performance degradation**
  - Every write causes the data to be written to main memory.
  - Slow down the machine considerably.
- **SPEC2000 integer benchmarks**
  - 10% of the instructions are stores.
  - If the CPI without cache miss = 1.0,

    spending 100 extra cycles on every write,

    CPI = 1.0 + 100 x 10% = 11
- **Solution: write buffer**
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- The new value is written only to the cache.
  - Dirty bit or modified bit
- The modified block is written to the main memory when it is replaced.
- Better performance
- More complex to implement

Elaboration

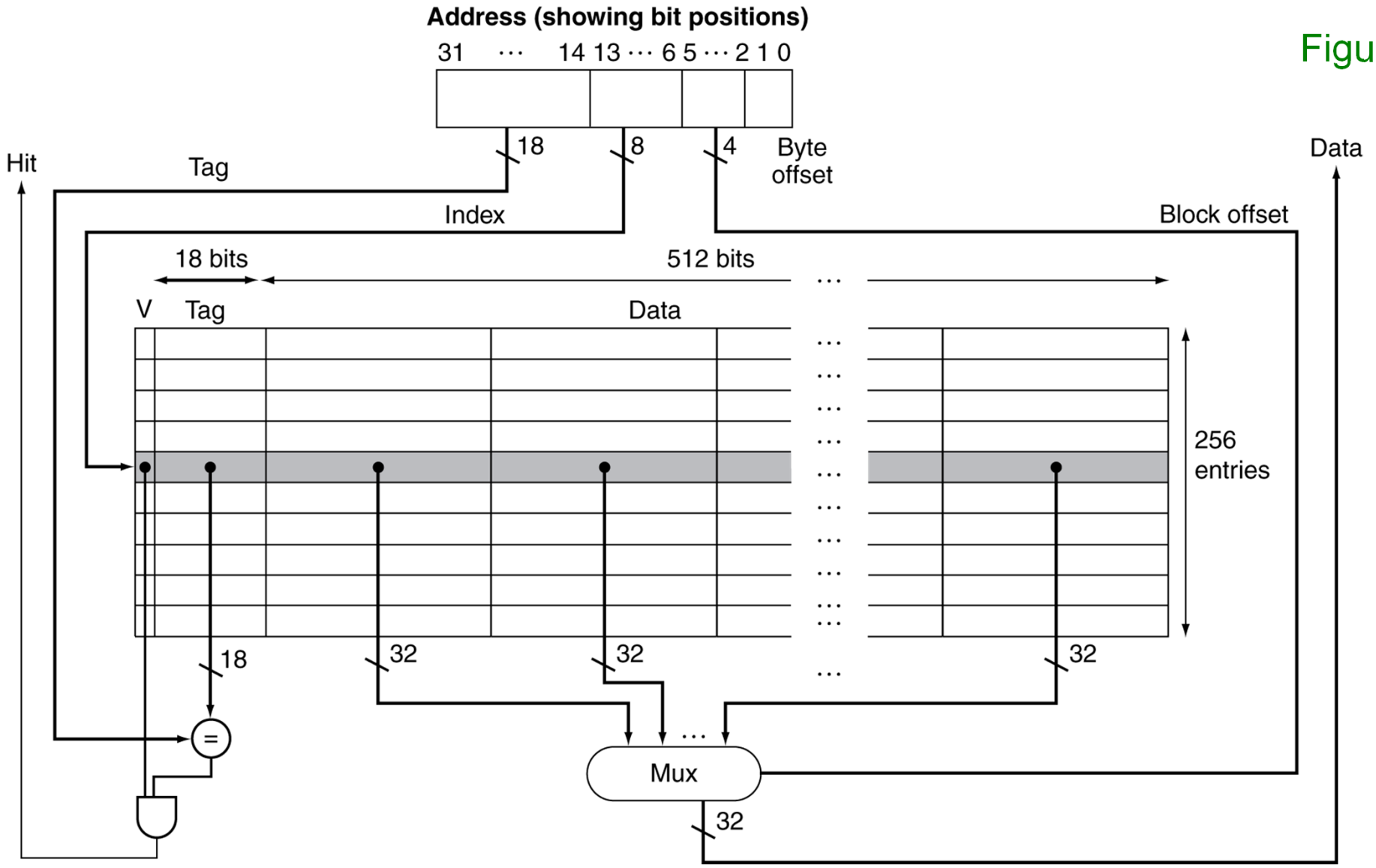
1. **Write allocate**
   - Most write-through caches
2. **No write allocate**

An Example Cache: The Intrinsity FastMATH processor

- **Intrinsity FastMATH**
  - Embedded microprocessor
  - MIPS architecture with a simple cache implementation
  - 12-stage pipeline
  - Split caches
    - Each cache: 16KB with 16-word block and 256 blocks
  - Write policy
    - OS selectable: write-through or write-back
  - Miss rate
    - Instruction: 0.4%
    - Data: 11.4%
    - Effective combined: 3.2%

Cache of Intrinsity FastMATH

Figure 5.9



Elaboration

- **Split caches vs. Combined (or Unified) cache**
  - Combined cache: better hit rate
  - Split cache: larger bandwidth
- **Miss rates for Intrinsity FastMATH**
  - Total cache size: 32 KB
  - Split cache: 3.24%
  - Combined cache: 3.18%
- **High bandwidth is more effective than high hit rate.**
- **Thus, we cannot use miss rate as the sole measure of cache performance.**

Supplement

Example: Prob. 3 of 2011-1 Terminal Exam

- **Direct-mapped cache with block size = 8 bytes**
- **Initial state**

	V	Tag	Data
00	1	0100	?
01	1	0100	?
10	1	0001	?
11	1	1010	?

- **Address**

4 bits	2 bits	3 bits
tag	index	offset

Example: Prob. 3 of 2011-1 Terminal Exam

- **Direct-mapped cache with block size = 8 bytes**
- **Initial state**

	V	Tag	Data
00	1	0100	M[080] ~ M[087]
01	1	0100	M[088] ~ M[08F]
10	1	0001	M[030] ~ M[037]
11	1	1010	M[158] ~ M[15F]

- Block 00: 0100 00 xxx ⇨ 080_{16} ~ 087_{16}
- Block 01: 0100 01 xxx ⇨ 088_{16} ~ $08F_{16}$
- Block 10: 0001 10 xxx ⇨ 030_{16} ~ 037_{16}
- Block 11: 1010 11 xxx ⇨ 158_{16} ~ $15F_{16}$

Reference String (in byte address)

- **083, 007, 034, 15F, 002, 080, 1A8, 038, 001, 1AF**

Address				
Hex	Binary	Tag	Index	
083	0 1000 0011	0100	00	Hit
007	0 0000 0111	0000	00	Miss
034	0 0011 0100	0001	10	Hit
15F	1 0101 1111	1010	11	Hit
002	0 0000 0010	0000	00	Hit
080	0 1000 0000	0100	00	Miss
1A8	1 1010 1000	1101	01	Miss
038	0 0011 1000	0001	11	Miss
001	0 0000 0001	0000	00	Miss
1AF	1 1010 1111	1101	01	Hit

Cache							
Block 00		Block 01		Block 10		Block 11	
Tag	Data	Tag	Data	Tag	Data	Tag	Data
0100	M[080]~	0100	M[088]~	0001	M[030]~	1010	M[158]~
0100							
0000	M[000]~						
				0001			
						1010	
0000							
0100	M[080]~						
		1101	M[1A8]~				
						0001	M[038]~
0000	M[000]~						
		1101					