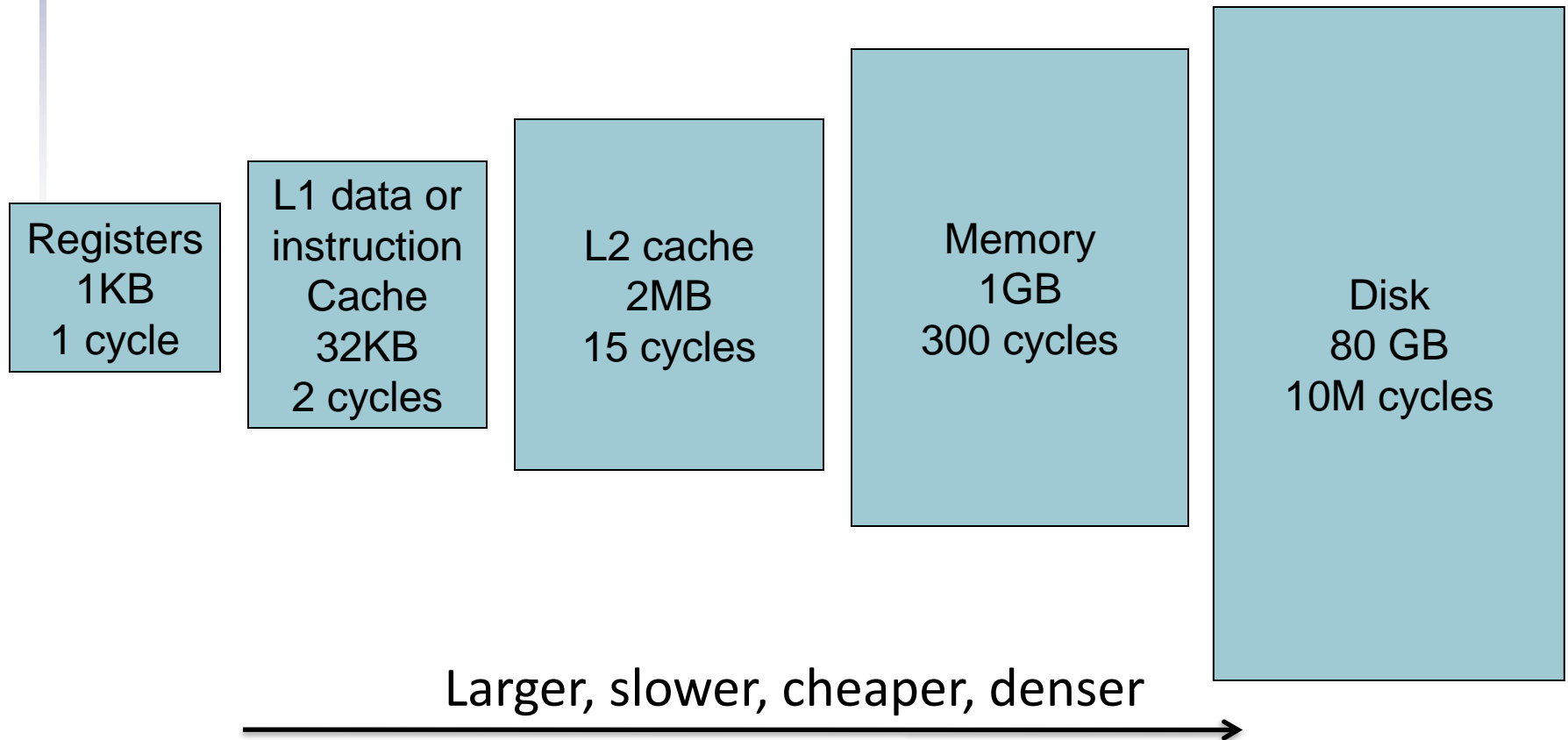


## **Chapter 5**

### **Large and Fast: Exploiting Memory Hierarchy**

# Memory Hierarchy

- Various storage devices in computers:



# Memory Technology

- Access time and price per bit vary widely among different technologies

Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

Data in 2012

- Ideal memory
  - ◆ Access time of Cache
  - ◆ Capacity and cost/GB of disk

# Outline

- Cache (CPU $\leftrightarrow$ memory)
  - ◆ Direct mapped cache
  - ◆ Set associative cache
  - ◆ Multi-level cache
- Virtual memory (memory $\leftrightarrow$ disk)
- Dependable memory
- Real examples

# Cache Hierarchies

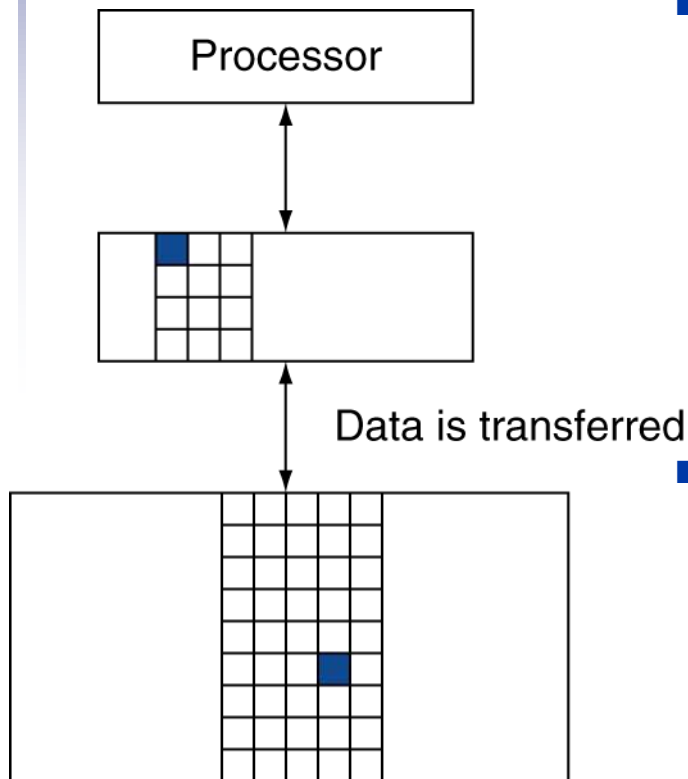
- Data and instructions are stored on DRAM chips
  - ◆ DRAM is a technology that has **high bit density**, but relatively **poor latency**
  - ◆ an access to data in memory can take as many as 300 cycles!
- Hence, some data is stored on the processor in a structure called the **cache**
  - ◆ caches employ SRAM technology, which is **faster**, but has **lower bit density**
- Internet browsers also cache web pages – same concept

# Memory hierarchy

---

- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - ◆ Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - ◆ Cache memory attached to CPU

# Memory Hierarchy Levels



- The memory in upper level is originally empty
- If accessed data is absent
  - ◆ **Miss**: block copied from lower level
    - Time taken: **miss penalty**
    - Miss ratio: misses/accesses
  - ◆ **Block** (also called line): unit of copying
    - May be multiple words
- If accessed data is present in upper level
  - ◆ **Hit**: access satisfied by upper level
    - Hit ratio: hits/accesses =  $1 - \text{miss ratio}$
  - ◆ Then accessed data supplied from upper level

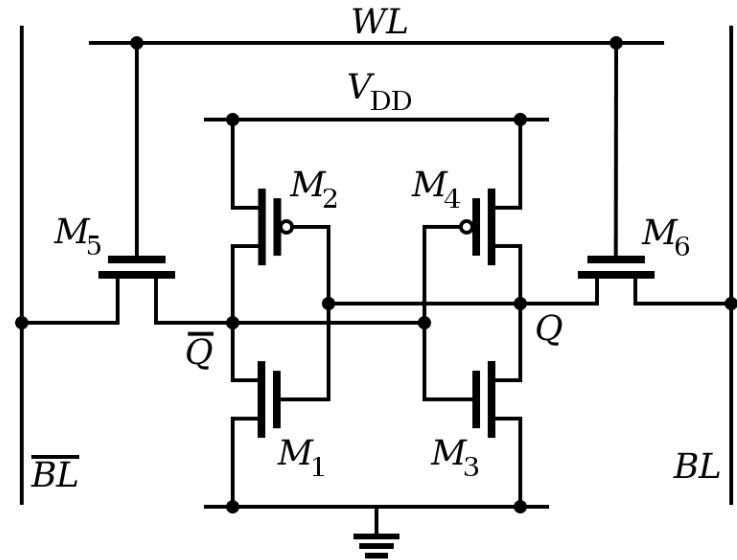
# Locality

- Why do caches work?
  - ◆ **Temporal locality**: if you used some data recently, you will likely use it again
  - ◆ **Spatial locality**: if you used some data recently, you will likely access its neighbors
- No hierarchy:
  - ◆ average access time for data = 300 cycles
- 32KB 1-cycle L1 cache that has a hit rate of 95%:
  - ◆ average access time =  $0.95 \times 1 + 0.05 \times (301) = 16$  cycles



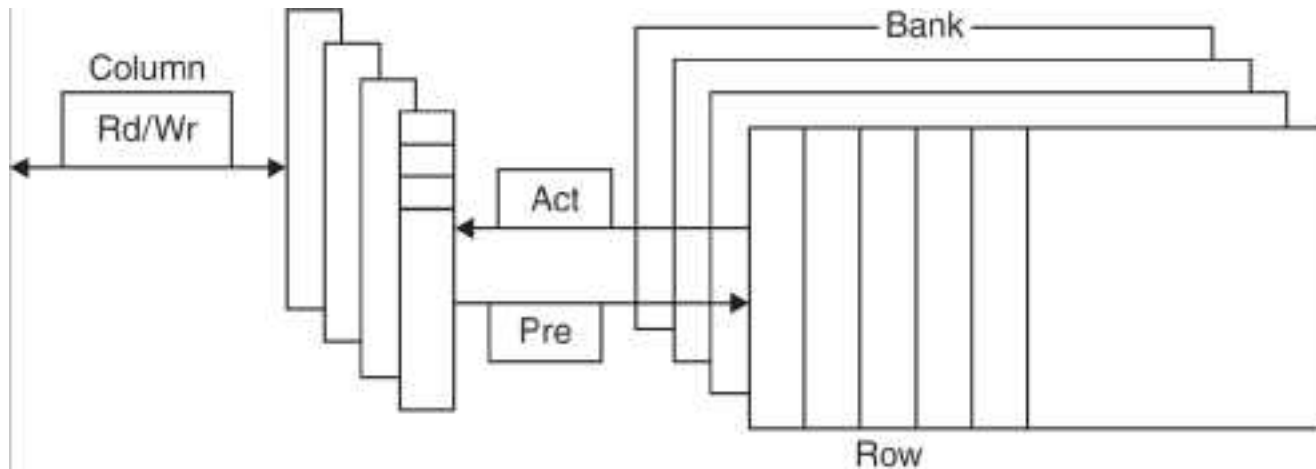
# SRAM Technology

- Static RAM
  - ◆ Memory arrays with a single read/write port
- It's Volatile
  - ◆ The data will be lost when SRAM is not powered
- Compared with DRAM
  - ◆ Don't need to refresh, use 6-8 transistors to install a bit
- Used in CPU cache, integrated onto the processor chip



# DRAM Technology

- Data stored as a charge in a capacitor
  - ◆ Single transistor used to access the charge
  - ◆ Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM “row”



# Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - ◆ DRAM accesses an entire row
  - ◆ Burst mode: supply successive words from a row with reduced latency
- Synchronous DRAM
  - ◆ A clock is added, the memory and processor are synchronized
  - ◆ Allows for consecutive accesses in bursts without needing to send each address
  - ◆ Improves bandwidth
- Double data rate (DDR) DRAM
  - ◆ Transfer on rising and falling clock edges
  - ◆ DDR4-3200 DRAM: 3200M times of transfer per second

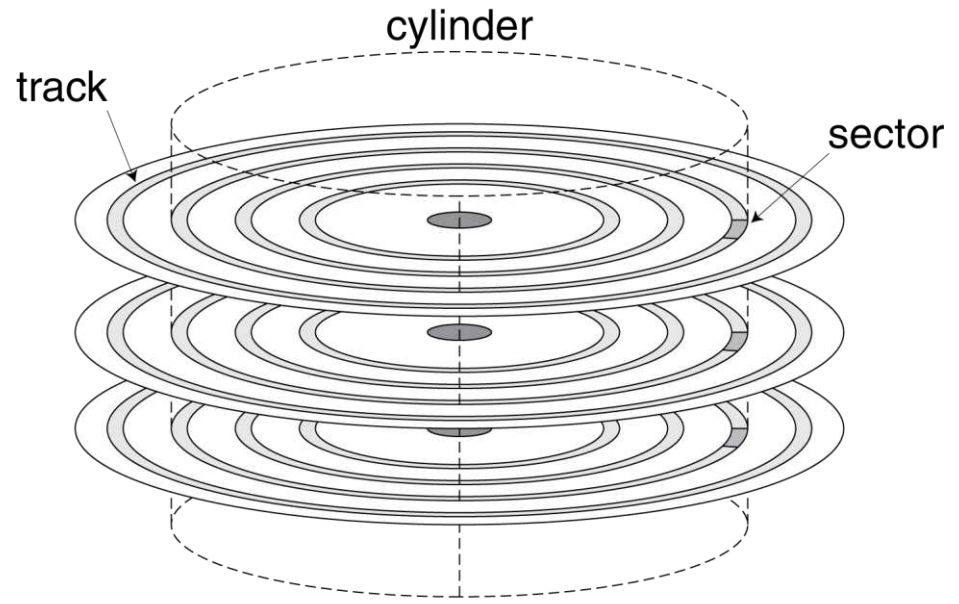
# Flash Storage

- Nonvolatile semiconductor storage
  - ◆ 100× – 1000× faster than disk
  - ◆ Smaller, lower power, more robust
  - ◆ But more \$/GB (between disk and DRAM)
- Flash bits wears out after 1000's of accesses
  - ◆ Not suitable for direct RAM or disk replacement
  - ◆ Wear leveling: remap data to less used blocks



# Disk Storage

- Nonvolatile, rotating magnetic storage



# Disk Sectors and Access

- Each sector records
  - ◆ Sector ID
  - ◆ Data (512 bytes, 4096 bytes proposed)
  - ◆ Error correcting code (ECC)
    - Used to hide defects and recording errors
- Access to a sector involves
  - ◆ Queuing delay if other accesses are pending
  - ◆ Seek: move the heads
  - ◆ Rotational latency
  - ◆ Data transfer
  - ◆ Controller overhead

# Cache Memory

- Cache memory
  - ◆ The level of the memory hierarchy closest to the CPU
- Given accesses  $X_1, \dots, X_{n-1}, X_n$

$X_4$
$X_1$
$X_{n-2}$
$X_{n-1}$
$X_2$
$X_3$

a. Before the reference to  $X_n$ 

$X_4$
$X_1$
$X_{n-2}$
$X_{n-1}$
$X_2$
$X_n$
$X_3$

b. After the reference to  $X_n$ 

- How do we know if the data is present?
- Where do we look?

# Memory Structure

- Address and data
  - ◆ Address is the index, are not stored in memory
    - Address can be in **unit of byte** or **in unit of word**
  - ◆ Only data is stored in memory

address	data	
000	Byte 1	Word1
001	Byte 2	
010	Byte 3	
011	...	
100		
101		
110		
111		

in unit of byte

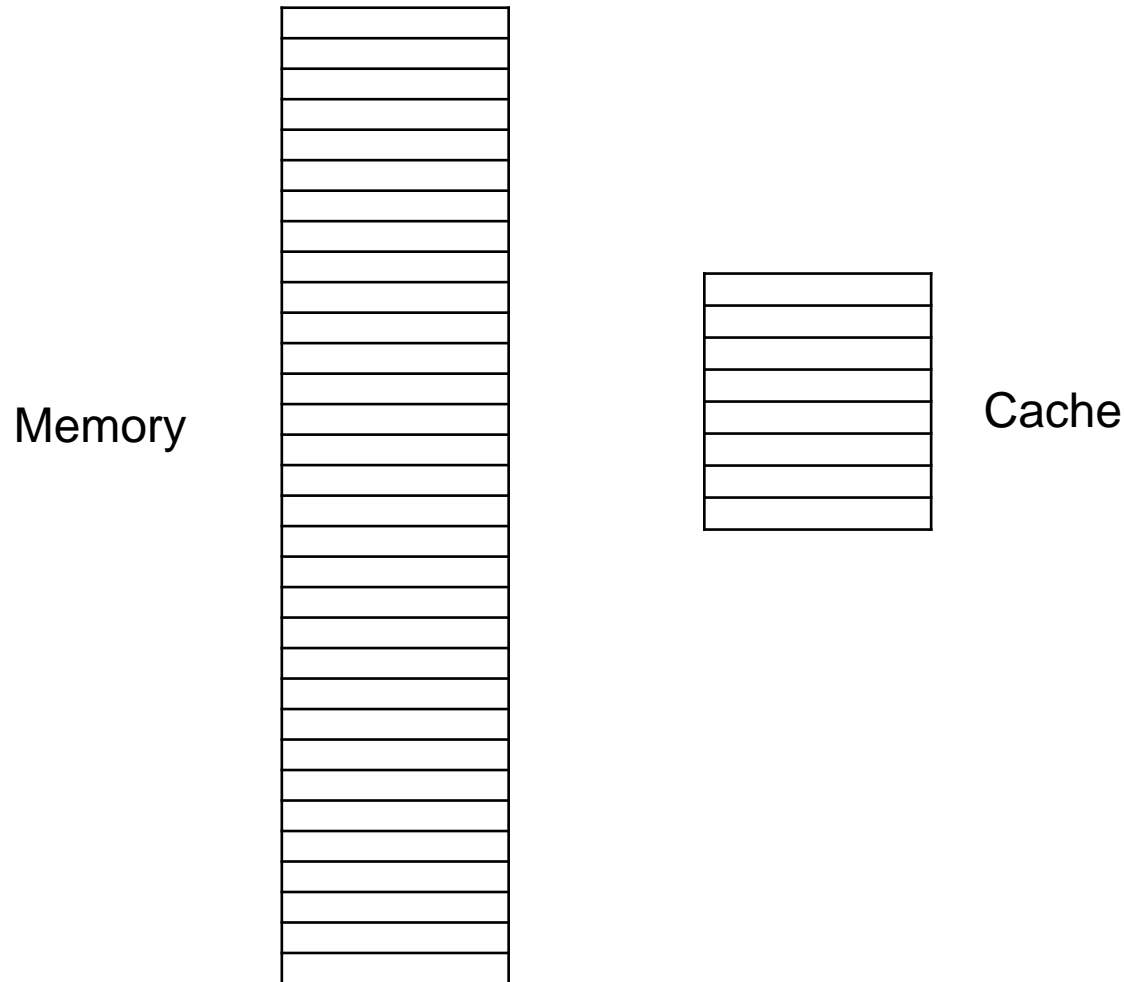
address	data
000	Word1
001	Word2
010	Word3
011	Word4
100	...
101	
110	
111	

in unit of word



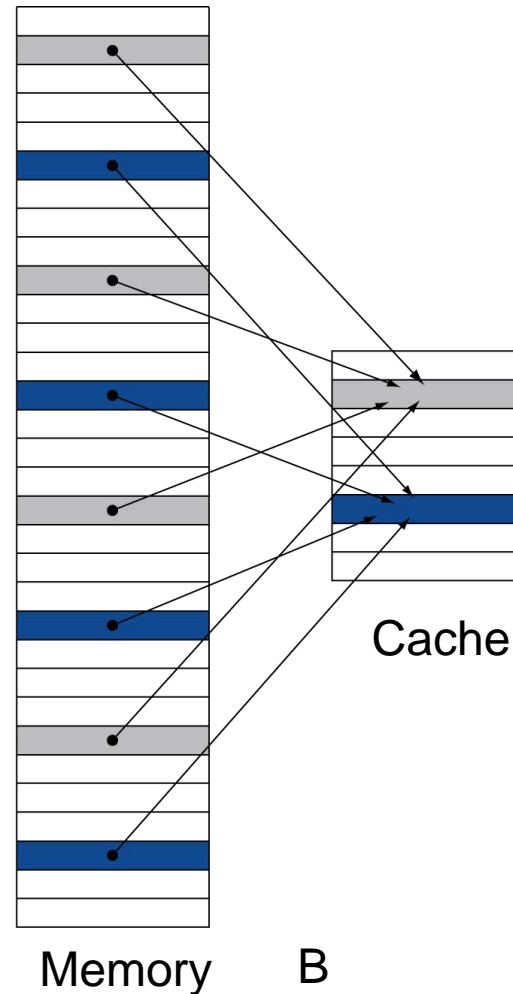
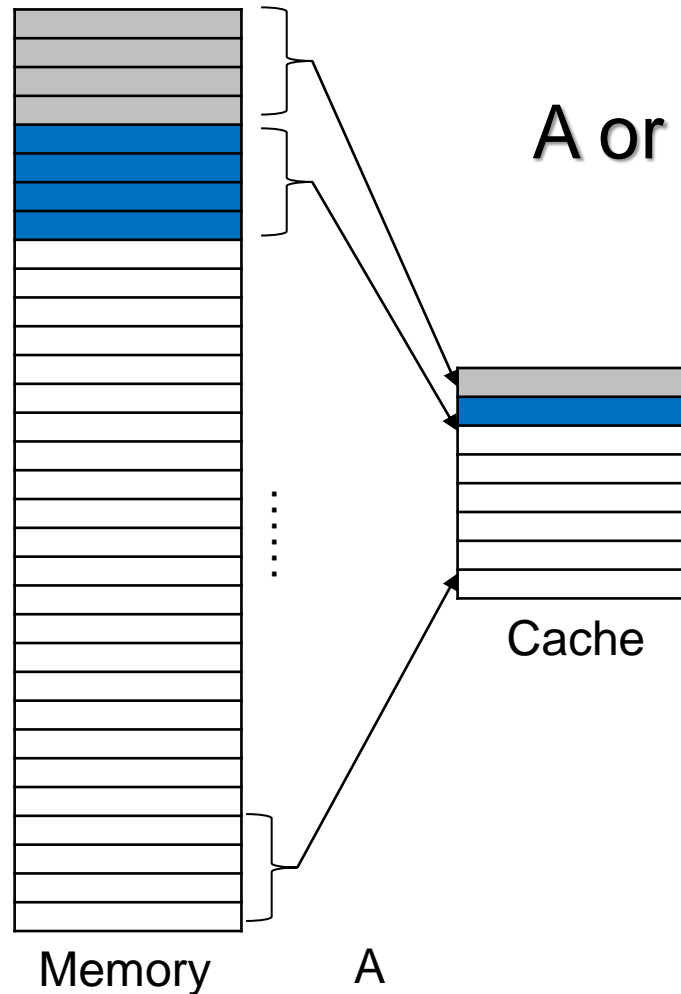
# Direct Mapped Cache

- Memory size: 32 words, cache size: 8 words, block size: 1 word
- The address is in unit of word



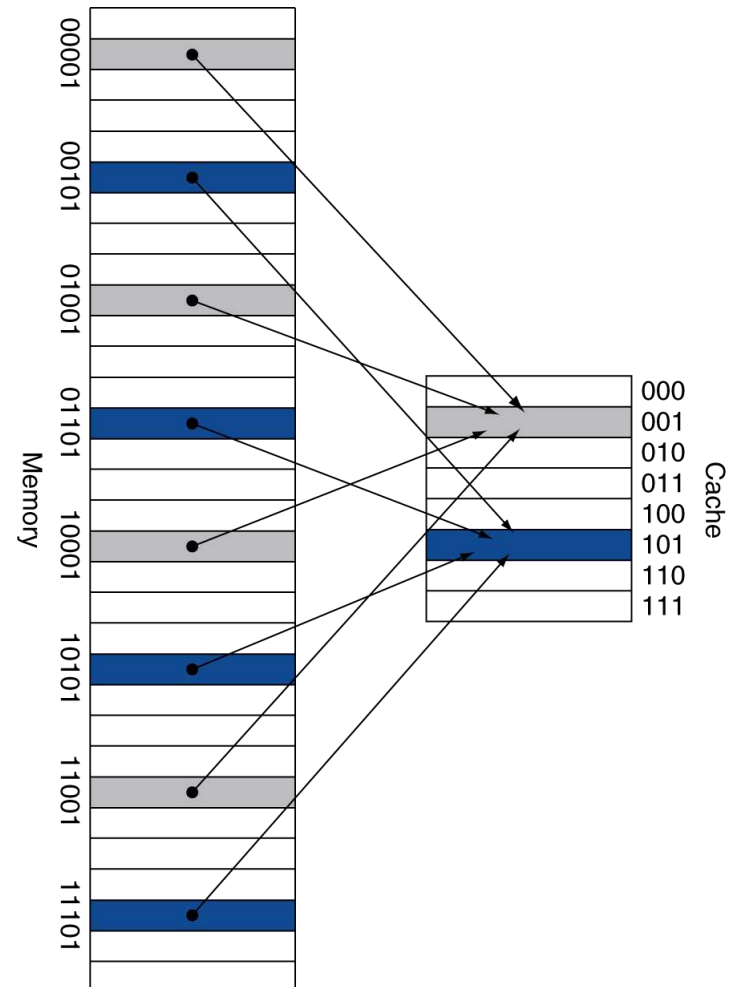
# Direct Mapped Cache

- Memory size: 32 words, cache size: 8 words, block size: 1 word
- The address is in unit of word



# Direct Mapped Cache

- Memory size: 32 words, cache size: 8 words, block size: 1 word
- The address is in unit of word
- Direct mapped cache:
  - ◆ Location determined by address
  - ◆ One data in memory is mapped to only one location in cache
  - ◆ Use low-order address bits or high-order bits?
  - ◆ The lower bits defines the address of the cache
  - ◆ **Index**: which block to select



# Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - ◆ Store block address as well as the data
  - ◆ Actually, only need the high-order bits
  - ◆ Called the **tag**
- What if there is no data in a location?
  - ◆ **Valid bit**: 1 = present, 0 = not present
  - ◆ Initially 0

# Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
<b>110</b>	<b>N</b>		
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
<b>110</b>	<b>Y</b>	<b>10</b>	<b>Mem[10110]</b>
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
<b>110</b>	<b>Y</b>	<b>10</b>	<b>Mem[10110]</b>
111	N		



# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
<b>010</b>	<b>Y</b>	<b>11</b>	<b>Mem[11010]</b>
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
<b>000</b>	<b>N</b>		
001	N		
010	Y	11	Mem[11010]
<b>011</b>	<b>N</b>		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
<b>000</b>	<b>Y</b>	<b>10</b>	<b>Mem[10000]</b>
001	N		
010	Y	11	Mem[11010]
<b>011</b>	<b>Y</b>	<b>00</b>	<b>Mem[00011]</b>
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

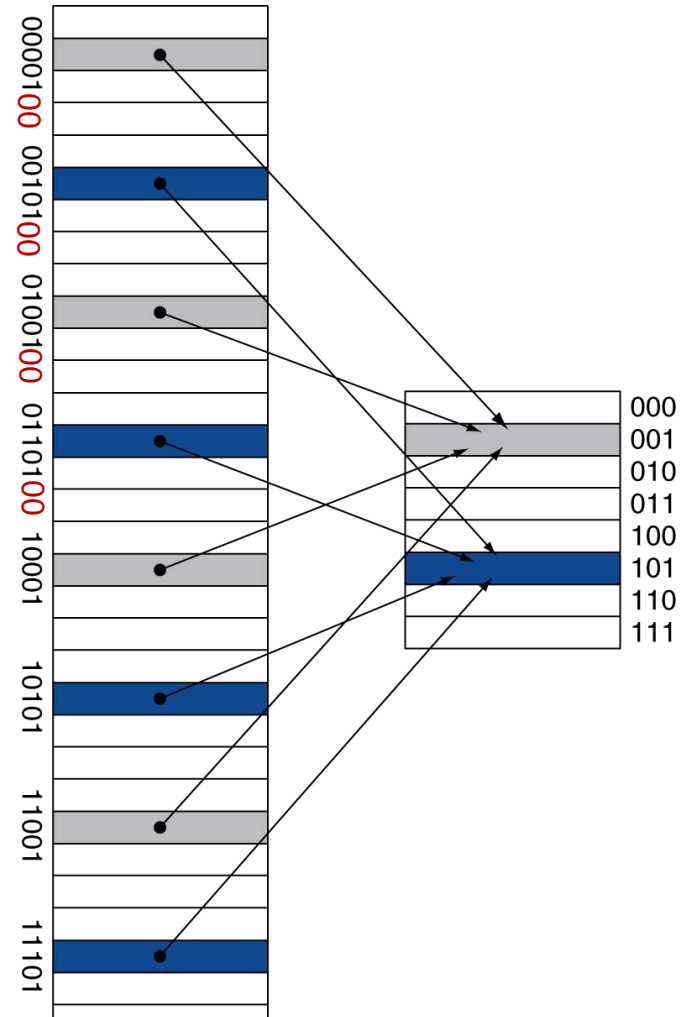
# Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

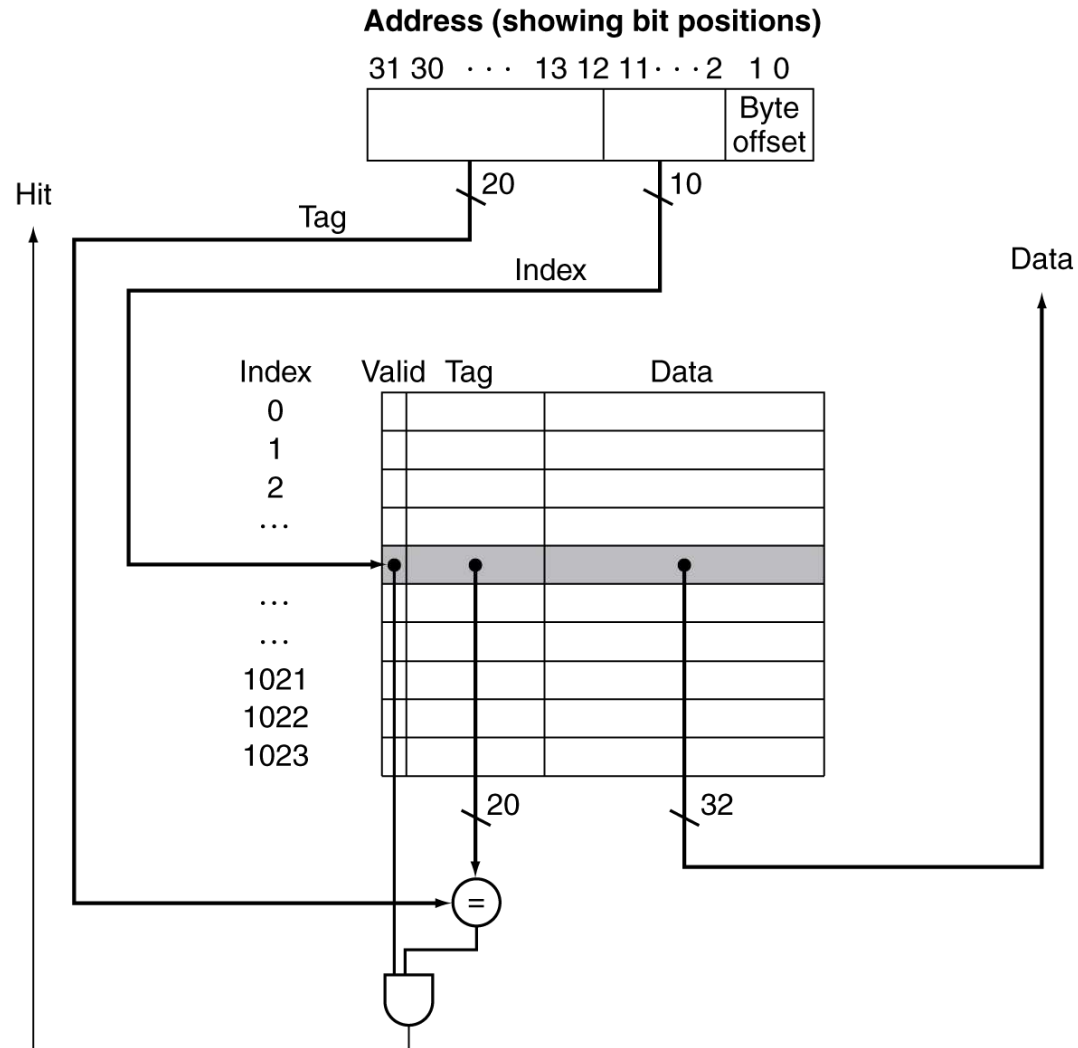
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
<b>010</b>	<b>Y</b>	<b>10</b>	<b>Mem[10010]</b>
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Memory in unit of byte

- How about the memory is in unit of byte? Assume:
- Memory size:
  - ◆ 32 words = 128 bytes
- Cache size:
  - ◆ 8 words = 32 bytes
- Block size:
  - ◆ 1 word = 4 bytes
- How to determine cache index and tag?



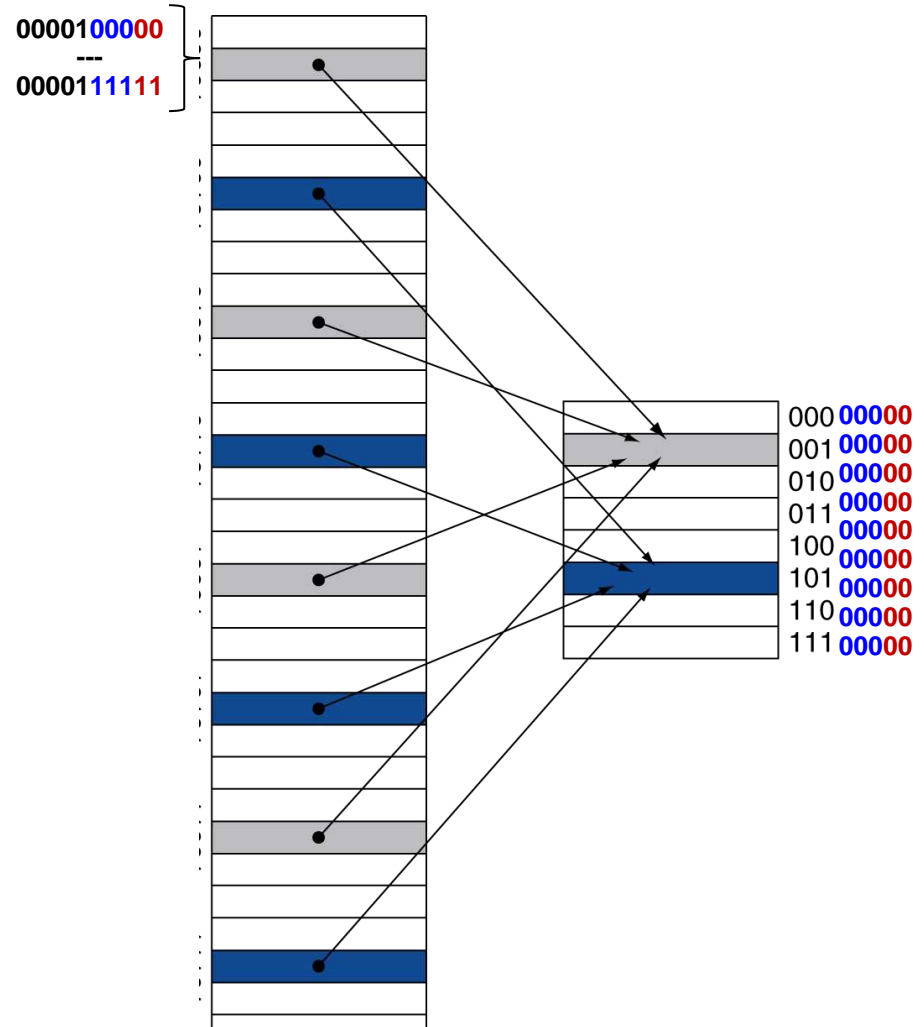
# Address Subdivision





# Larger Block Size

- How about the block size is larger? Assume:
- Memory size:
  - ◆ 256 words = 1024 bytes
- Cache size:
  - ◆ 64 words = 256 bytes
- Block size:
  - ◆ 8 word = 32 bytes
- How to determine cache index and tag?



# Larger Block Size

- Assume:

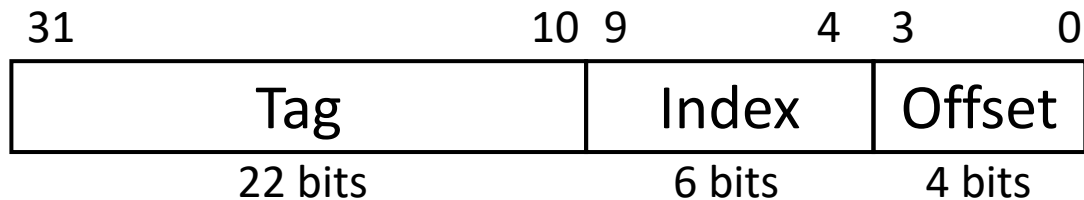
- ◆ 32-bit address
- ◆ Direct mapped cache
- ◆  $2^n$  number of blocks, so  $n$  bit for index
- ◆ Block size:  $2^m$  words, so  $m$  bit for the word within the block

- Calculate:

- ◆ Size of tag field:  $32-(n+m+2)$
- ◆ Size of cache:  $2^n * (\text{block size} + \text{tag size} + \text{valid field size})$   
 $= 2^n * (2^m * 32 + (32 - n - m - 2) + 1)$

# Example: Larger Block Size

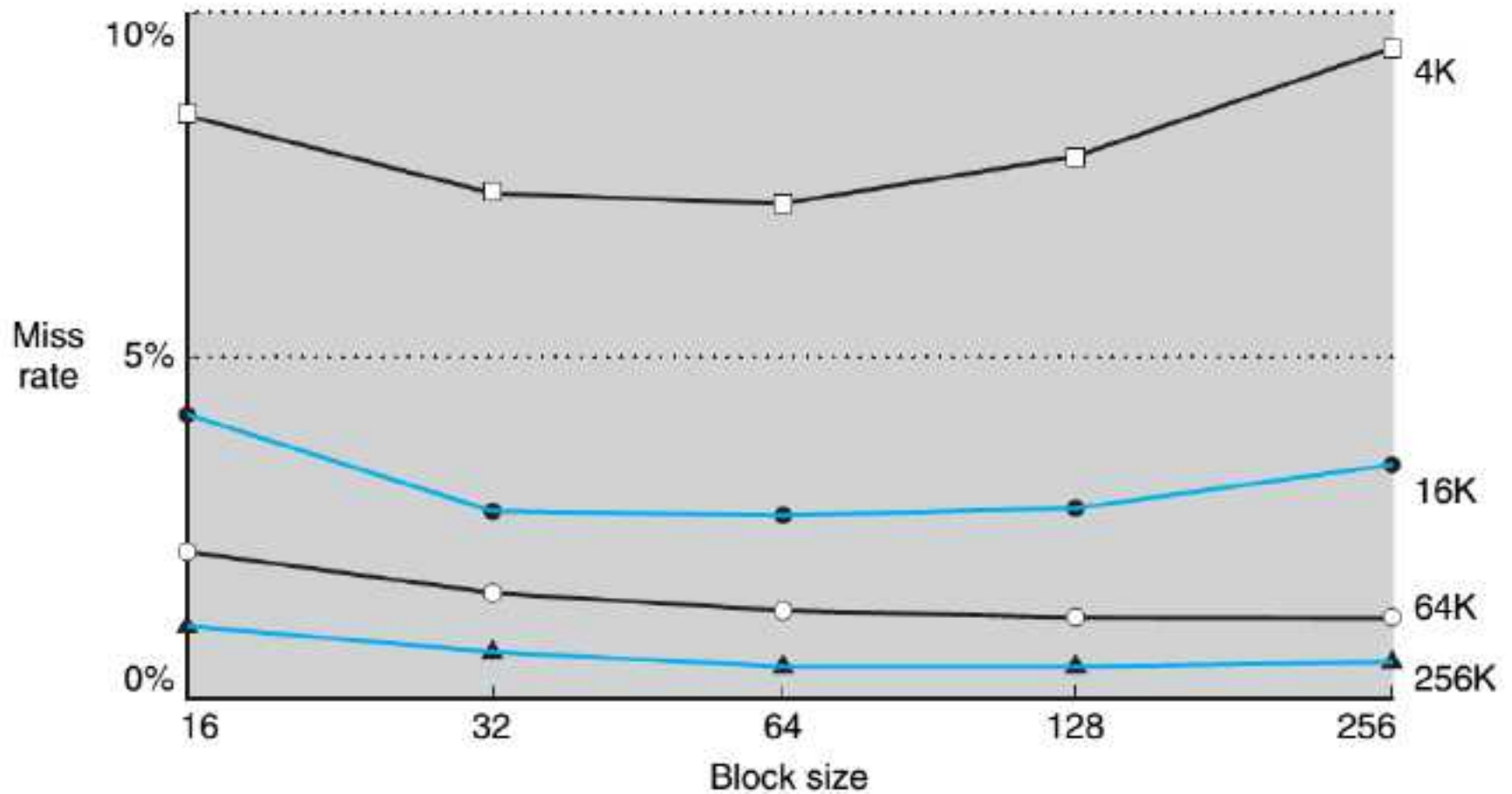
- 64 blocks, 16 bytes/block
  - ◆ To what block number does address 1200 map?
- Block address =  $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



# Block Size Considerations

- Larger blocks should reduce miss rate
  - ◆ Due to spatial locality
- But in a fixed-sized cache
  - ◆ Larger blocks  $\Rightarrow$  fewer of them
    - More competition  $\Rightarrow$  increased miss rate
  - ◆ Larger blocks  $\Rightarrow$  pollution
- Larger miss penalty
  - ◆ Can override benefit of reduced miss rate
  - ◆ Early restart and critical-word-first can help

# Block Size Considerations



# Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - ◆ Read miss vs. write miss
  - ◆ Stall the CPU pipeline
  - ◆ Fetch block from next level of hierarchy
  - ◆ Instruction cache miss
    - Restart instruction fetch
  - ◆ Data cache miss
    - Complete data access

# Write-Through

- On data-write hit, could just update the block in cache
  - ◆ But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - ◆ e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI =  $1 + 0.1 \times 100 = 11$
- 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

- Alternative: On data-write hit, just update the block in cache
  - ◆ Keep track of whether each block is dirty
- When a dirty block is replaced
  - ◆ Write it back to memory
  - ◆ Can use a write buffer to allow replacing block to be read first



# Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - ◆ Allocate on miss: fetch the block
  - ◆ Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - ◆ Usually fetch the block

# Write Policies Summary

- If that memory location is in the cache?
  - ◆ Send it to the cache
  - ◆ Should we also send it to memory right away?  
(write-through policy)
  - ◆ Wait until we kick the block out (write-back policy)
- If it is not in the cache?
  - ◆ Allocate the line (put it in the cache)?  
(write allocate policy)
  - ◆ Write it directly to memory without allocation?  
(no write allocate policy)

# Example: Intrinsity FastMATH

- Embedded MIPS processor
  - ◆ 12-stage pipeline
  - ◆ Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - ◆ Each 16KB:  $256 \text{ blocks} \times 16 \text{ words/block}$
  - ◆ D-cache: write-through or write-back
- SPEC2000 miss rates
  - ◆ I-cache: 0.4%
  - ◆ D-cache: 11.4%
  - ◆ Weighted average: 3.2%

# Example: Intrinsity FastMATH

