# Chapter 5

## Large and Fast: Exploiting Memory Hierarchy

# Principle of Locality

- Programs access a small proportion of their address space at any time

- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables

- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

# Taking Advantage of Locality

- 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

Processor

Data is transferred

- **Block (aka 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
- **If accessed data is absent**
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level
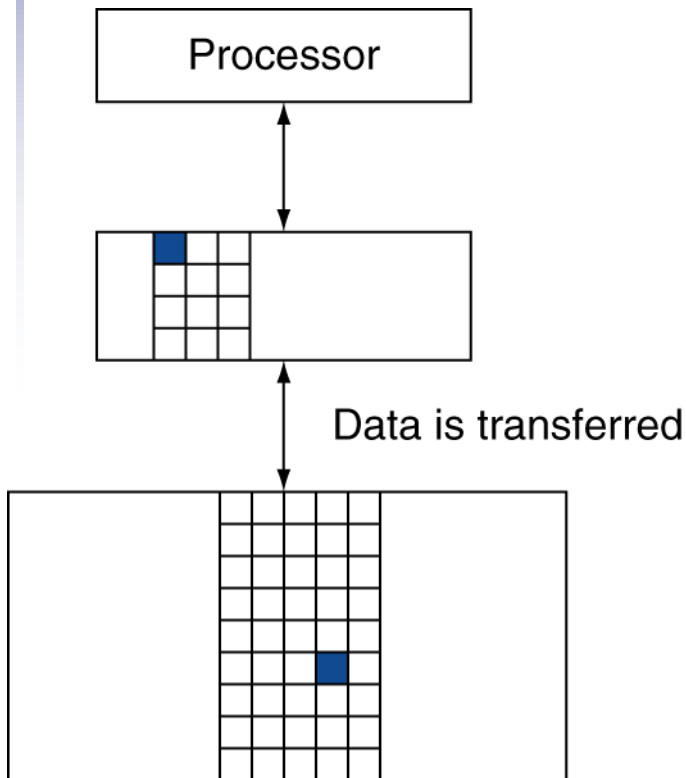
# Memory Technology

- ## Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- ## Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- ## Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB
- ## Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

# 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"
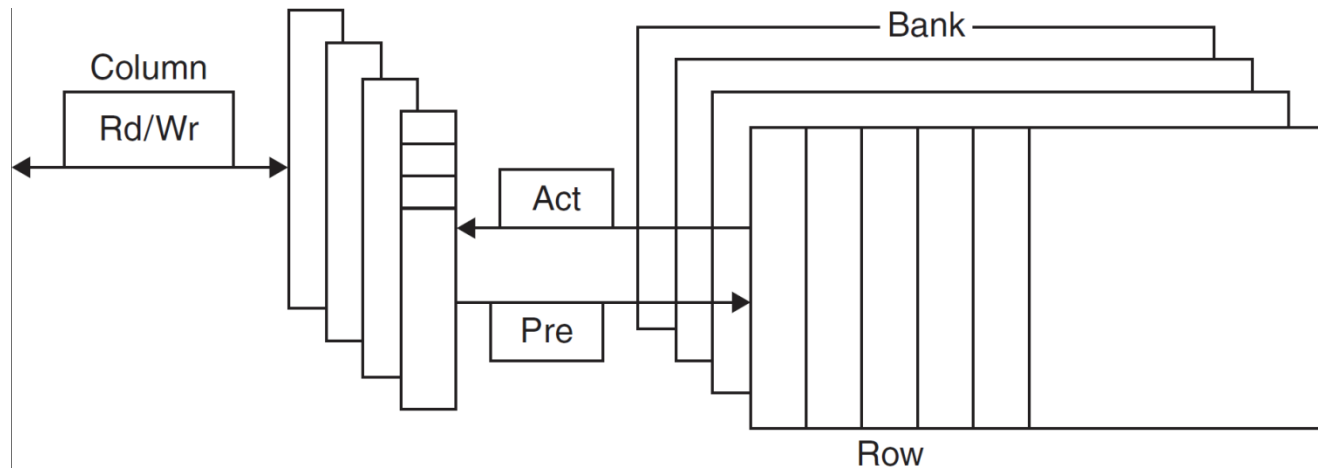
# Advanced DRAM Organization

- 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
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs

# DRAM Generations

| Year | Capacity | $/GB |
|------|----------|-------|
| 1980 | 64Kbit | $1500000 |
| 1983 | 256Kbit | $500000 |
| 1985 | 1Mbit | $200000 |
| 1989 | 4Mbit | $50000 |
| 1992 | 16Mbit | $15000 |
| 1996 | 64Mbit | $10000 |
| 1998 | 128Mbit | $4000 |
| 2000 | 256Mbit | $1000 |
| 2004 | 512Mbit | $250 |
| 2007 | 1Gbit | $50 |
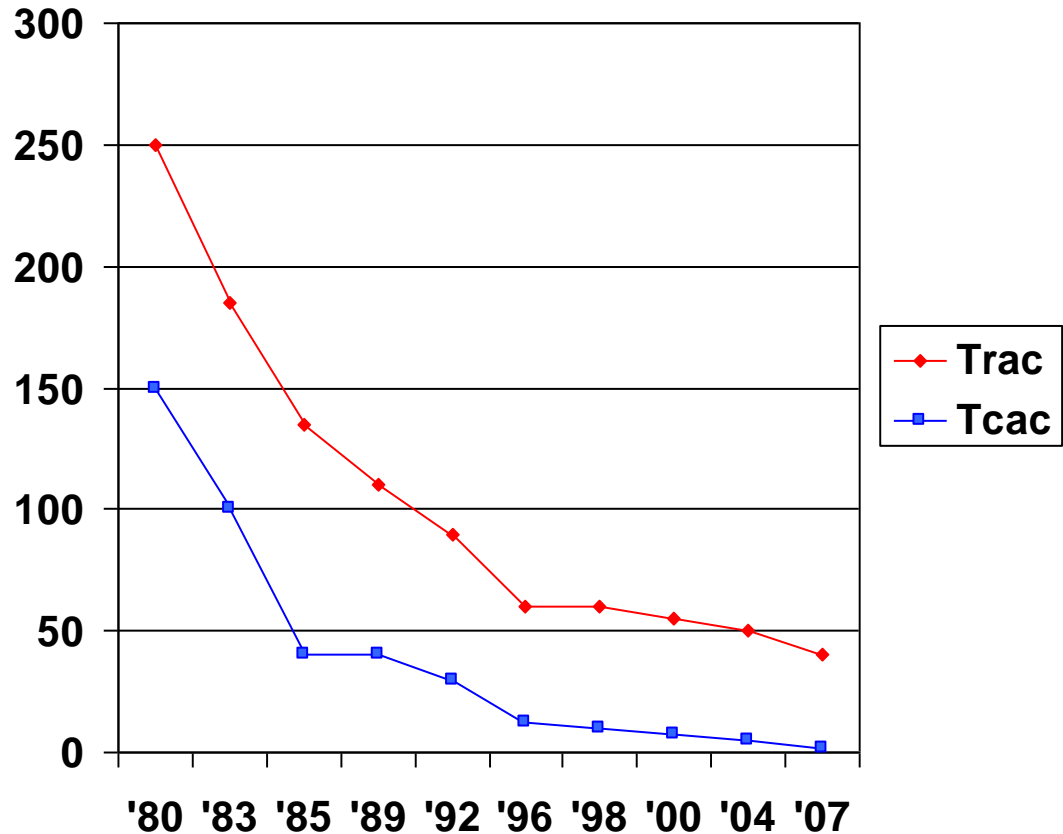
# DRAM Performance Factors

- ## Row buffer
  - Allows several words to be read and refreshed in parallel

- ## Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth

- ## DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth

# Increasing Memory Bandwidth
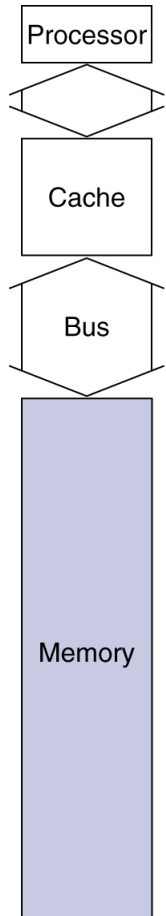


a. One-word-wide memory organization
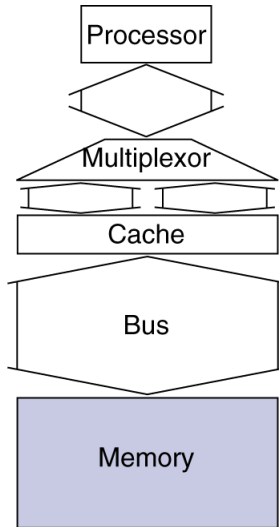
b. Wider memory organization
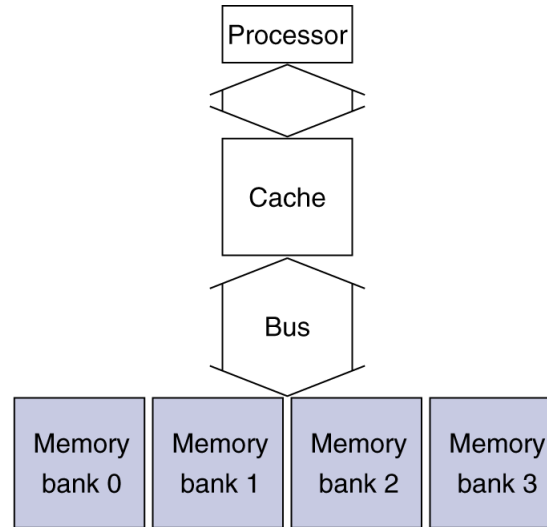
c. Interleaved memory organization

- ## 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- ## 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4 $\times$ 1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

# Flash Storage

- Nonvolatile semiconductor storage
  - $100\times$ – $1000\times$ faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)
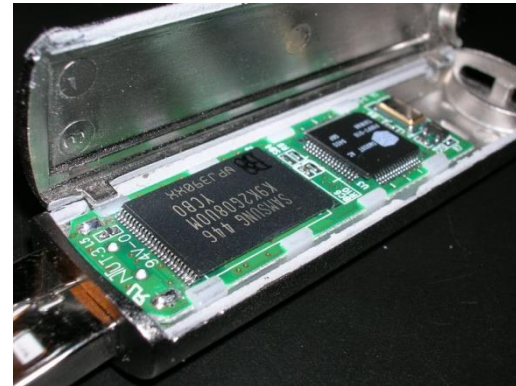
# Flash Types

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems

- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …

- 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
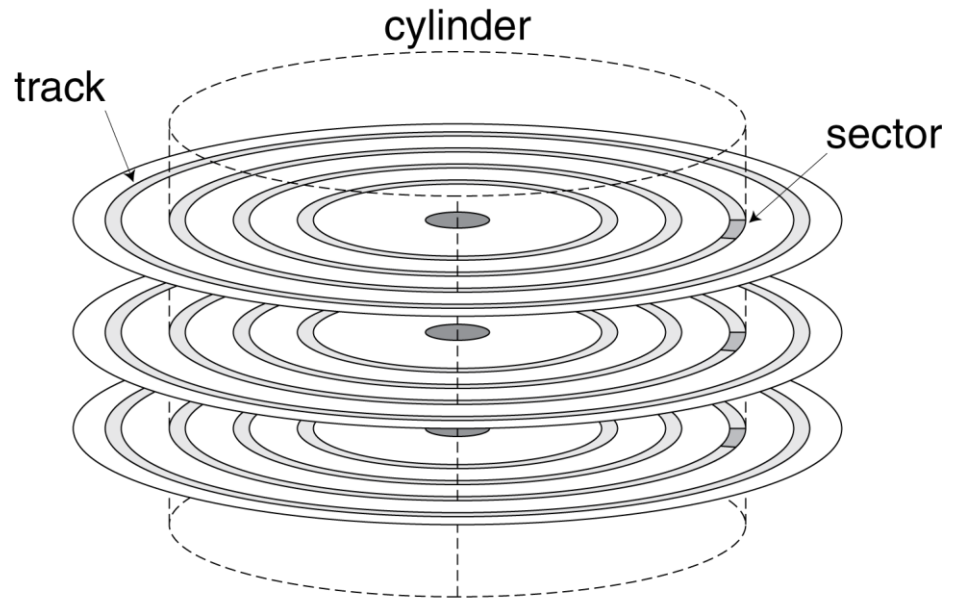
# Disk Storage
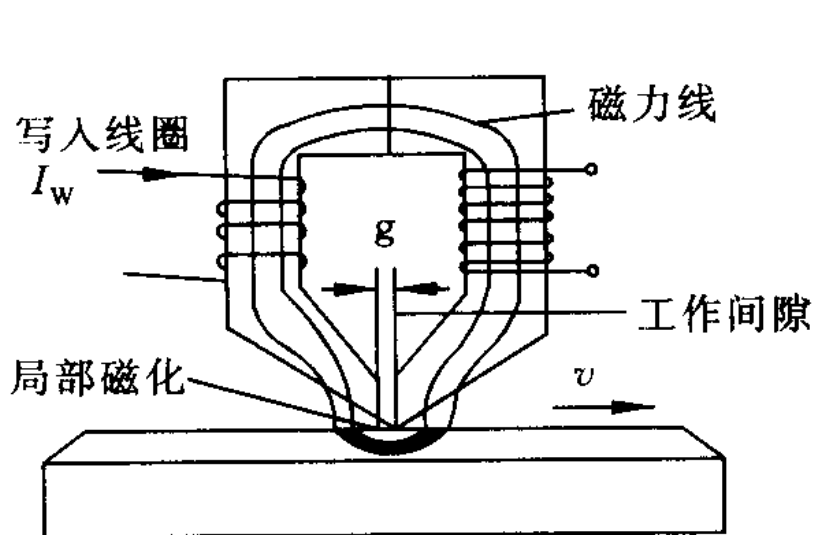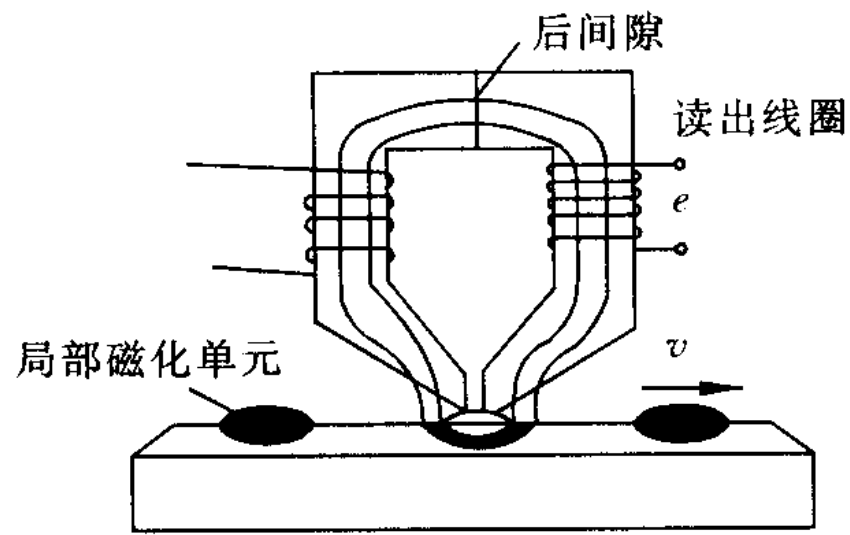


（a）写入 　　　　　　　　　　　（b）读出

# 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
    - Synchronization fields and gaps
- Access to a sector involves
    - Queuing delay if other accesses are pending
    - Seek: move the heads
    - Rotational latency
    - Data transfer
    - Controller overhead

# Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk

- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms

- If actual average seek time is 1ms
  - Average read time = 3.2ms

# Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
  - SCSI, ATA, SATA
- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay

# Cache Memory

- ## Cache memory
  - The level of the memory hierarchy closest to the CPU

- ## Given accesses $X_1, \ldots, X_{n-1}, X_n$

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

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?

# Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

Cache



Memory
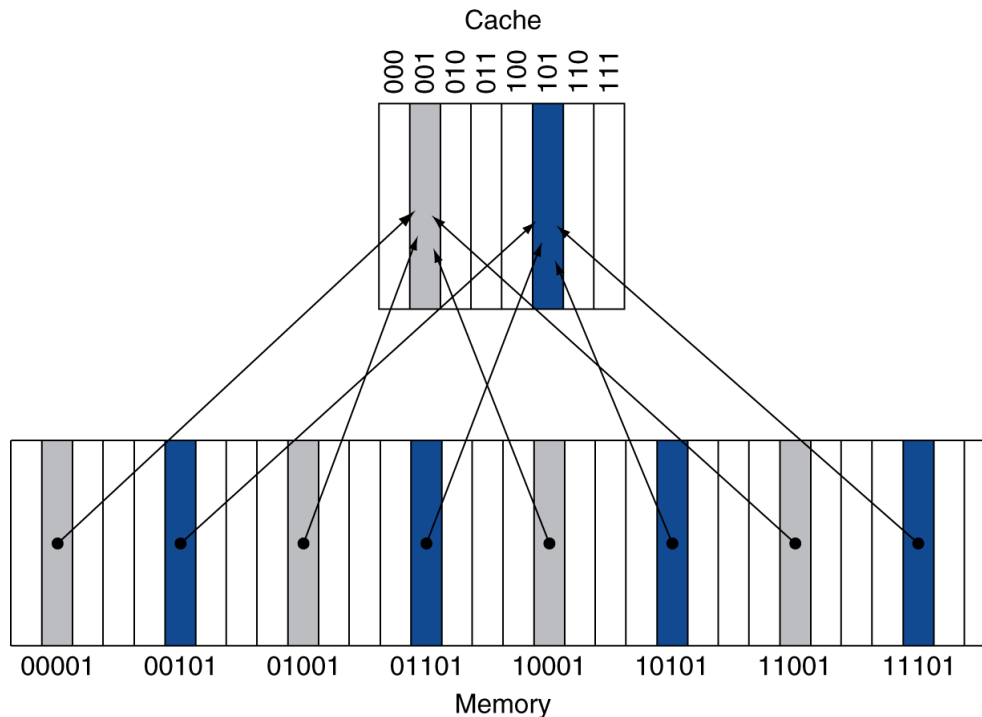
- #Blocks is a power of 2
- Use low-order address bits

# 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 |     |      |
| 110   | Y | 10  | Mem[10110] |
| 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   | 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 |
|-----------|-------------|----------|-------------|
| 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 |
|-------|---|-----|------|
| **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 | | |
| **010** | **Y** | **10** | **Mem[10010]** |
| 011 | Y | 00 | Mem[00011] |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# Address Subdivision
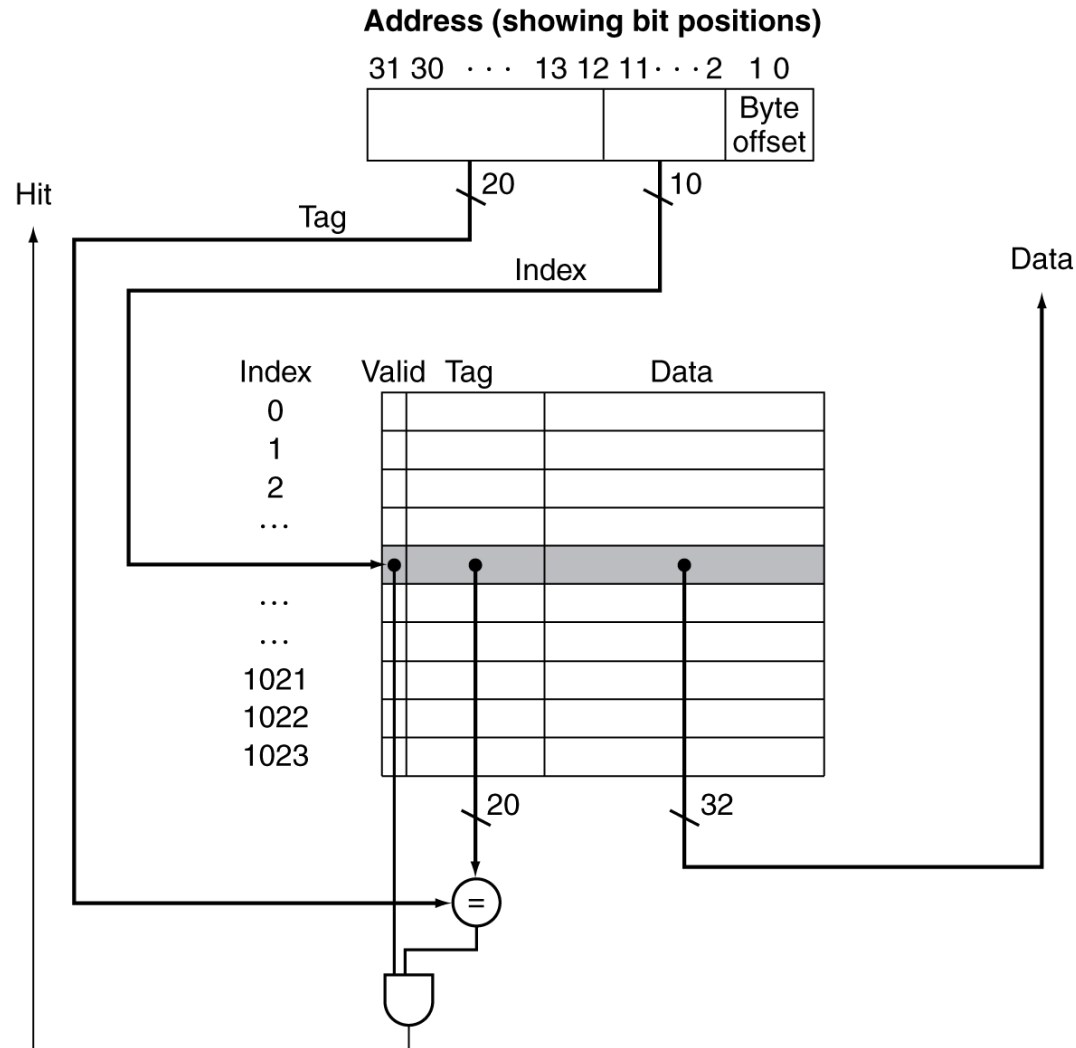
# Example: Larger Block Size

- ## 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

| 31 | 10 9 | 4 3 | 0 |
|:---:|:---:|:---:|:---:|
| Tag | | Index | Offset |
| 22 bits | | 6 bits | 4 bits |
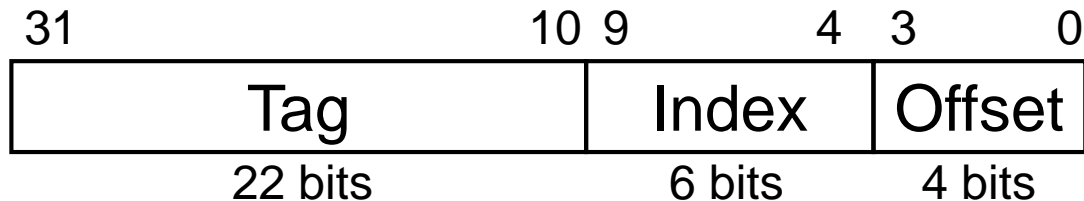
**$1200_{10}$ = 1001011 0000$_2$**

# 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

# Cache Misses

- On cache hit, CPU proceeds normally
- On cache 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

# 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 blocks $\times$ 16 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
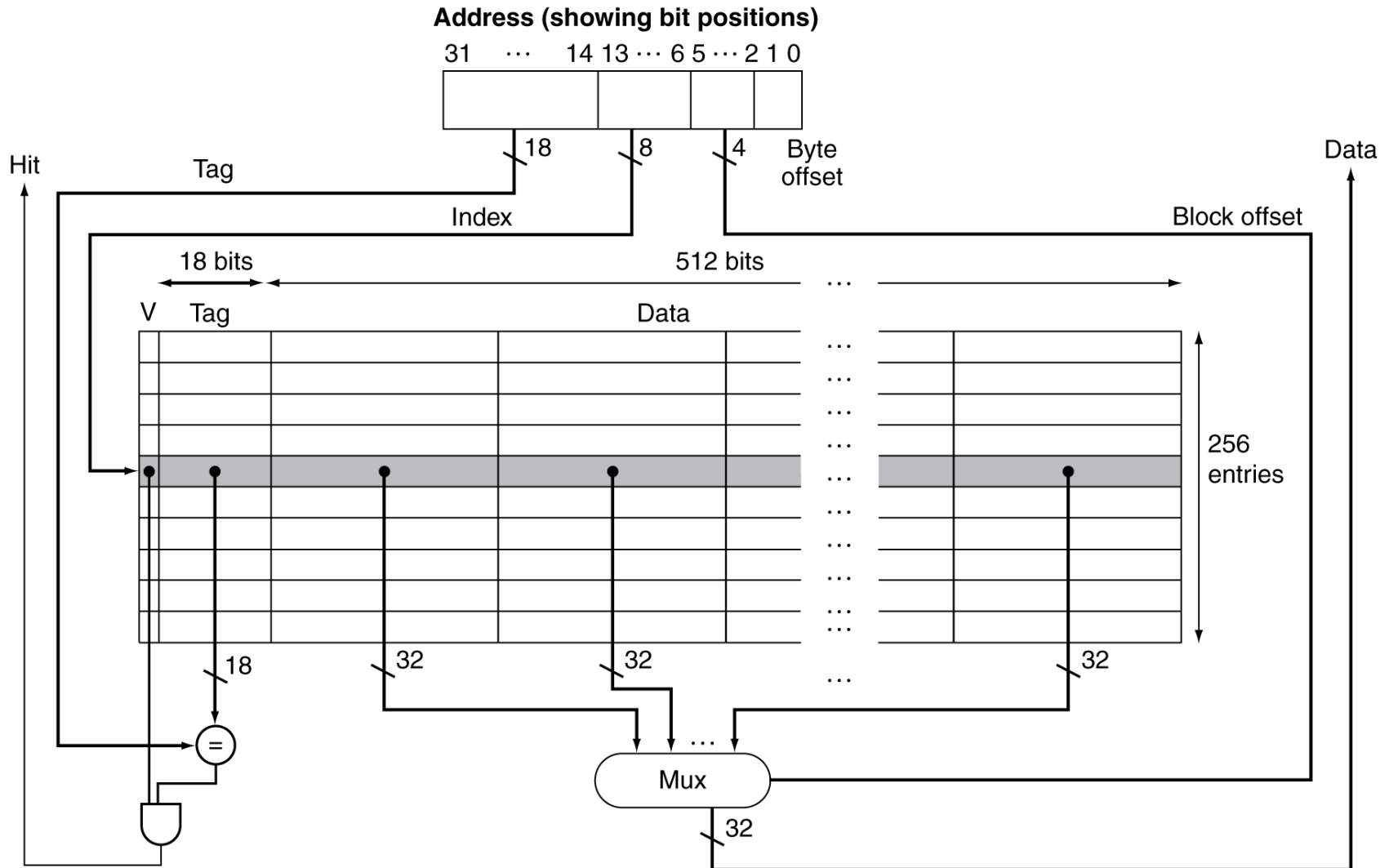
# Main Memory Supporting Caches

- **Use DRAMs for main memory**
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- **Example cache block read**
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- **For 4-word block, 1-word-wide DRAM**
  - Miss penalty = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

# Measuring Cache Performance

- ■ Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- ■ With simplifying assumptions:

$$\text{Memory stall cycles}$$

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

# Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

# Average Access Time

- Hit time is also important for performance

- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate $\times$ Miss penalty

- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 $\times$ 20 = 2ns
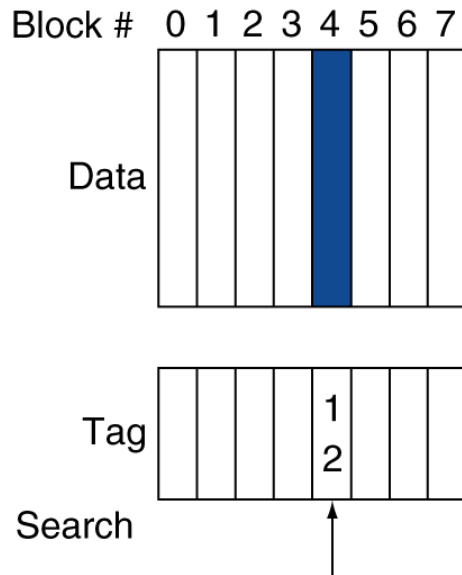    - 2 cycles per instruction

# Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

# Associative Caches

- Fully associative
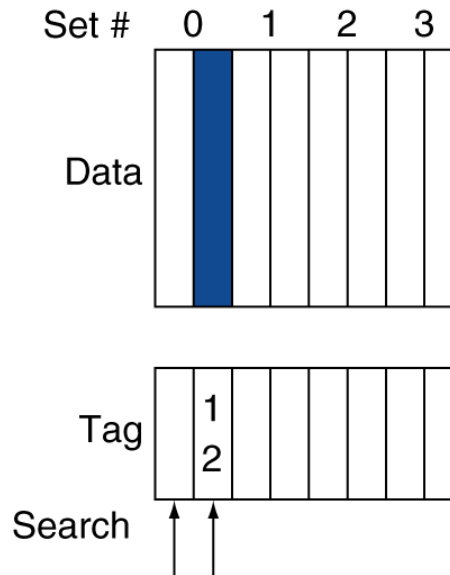    - Allow a given block to go in any cache entry
    - Requires all entries to be searched at once
    - Comparator per entry (expensive)
- *n*-way set associative
    - Each set contains *n* entries
    - Block number determines which set
        - (Block number) modulo (#Sets in cache)
    - Search all entries in a given set at once
    - *n* comparators (less expensive)
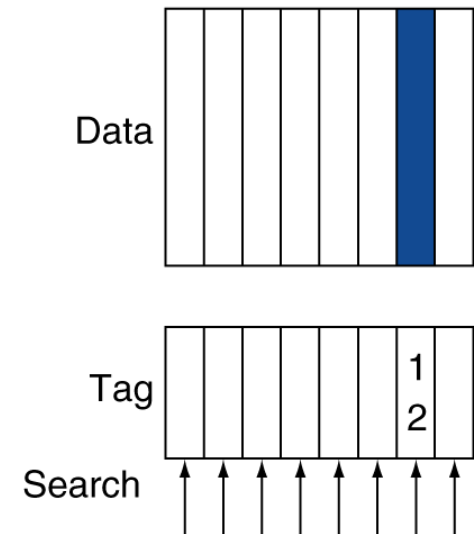
# Associative Cache Example

# Spectrum of Associativity

■ For a cache with 8 blocks

**One-way set associative**
**(direct mapped)**

| Block | Tag | Data |
|-------|-----|------|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

**Two-way set associative**

| Set | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

**Four-way set associative**

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|-----|------|-----|------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |

**Eight-way set associative (fully associative)**

| Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|
| | | | | | | | | | | | | | | | |

# Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 |
| 0 | 0 | miss | Mem[0] | | | |
| 8 | 0 | miss | Mem[8] | | | |
| 0 | 0 | miss | Mem[0] | | | |
| 6 | 2 | miss | Mem[0] | | Mem[6] | |
| 8 | 0 | miss | Mem[8] | | Mem[6] | |

# Associativity Example

- ## 2-way set associative

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | Set 0 | | Set 1 | |
| 0 | 0 | miss | Mem[0] | | | |
| 8 | 0 | miss | Mem[0] | Mem[8] | | |
| 0 | 0 | hit | Mem[0] | Mem[8] | | |
| 6 | 0 | miss | Mem[0] | Mem[6] | | |
| 8 | 0 | miss | Mem[8] | Mem[6] | | |

- ## Fully associative

| Block address | | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| 0 | | miss | Mem[0] | | | |
| 8 | | miss | Mem[0] | Mem[8] | | |
| 0 | | hit | Mem[0] | Mem[8] | | |
| 6 | | miss | Mem[0] | Mem[8] | Mem[6] | |
| 8 | | hit | Mem[0] | Mem[8] | Mem[6] | |

# How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

# Set Associative Cache Organization
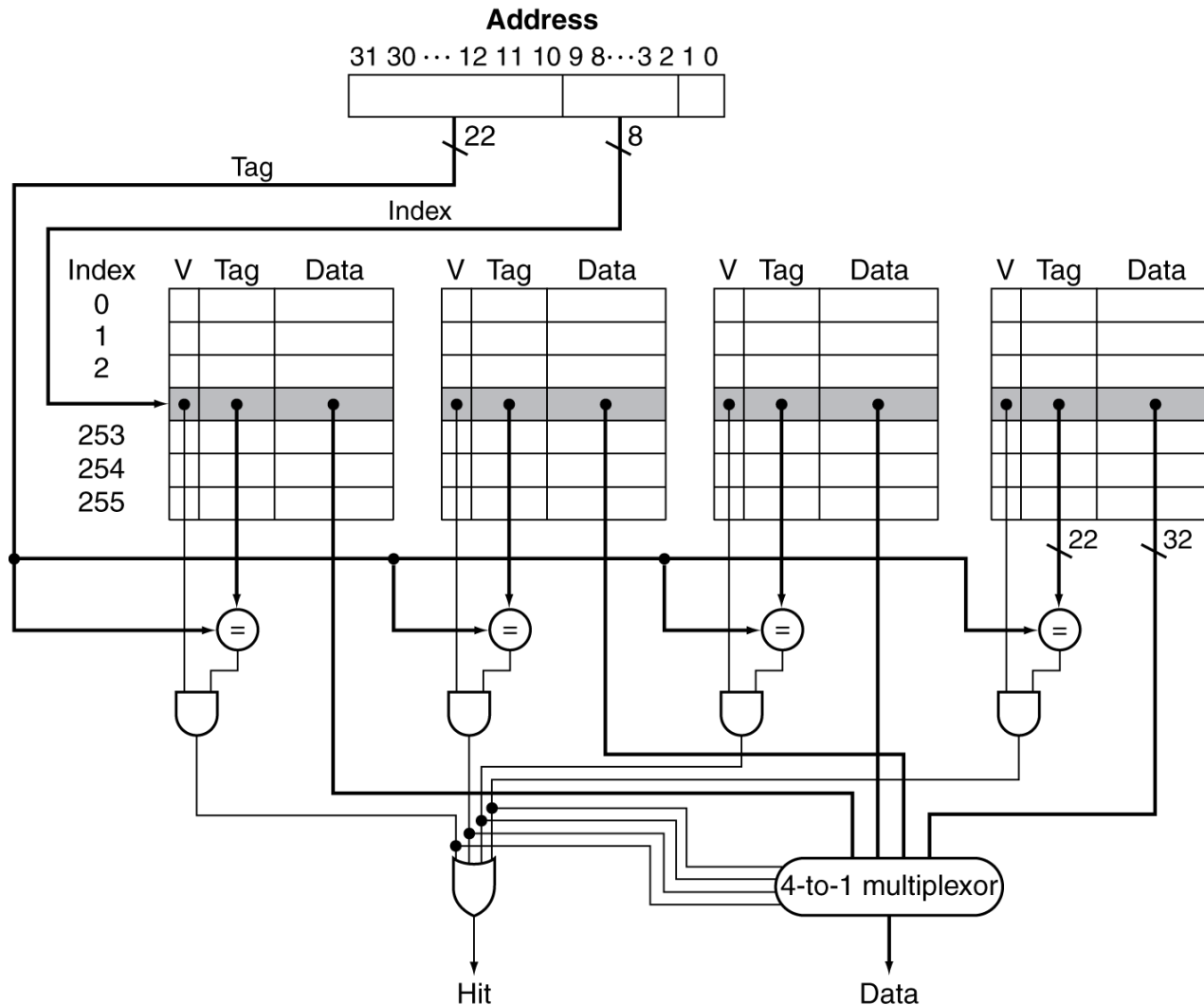
# Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

# Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

# Multilevel Cache Example

- **Given**
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- **With just primary cache**
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 $\times$ 400 = 9

# Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 $\times$ 20 + 0.005 $\times$ 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

# Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

# Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation

# Interactions with Software

- ## Misses depend on memory access patterns
  - ### Algorithm behavior
  - ### Compiler optimization for memory access
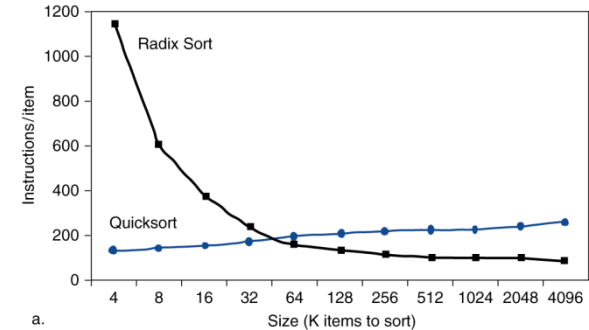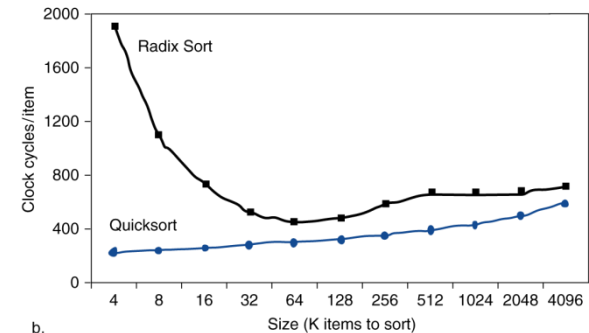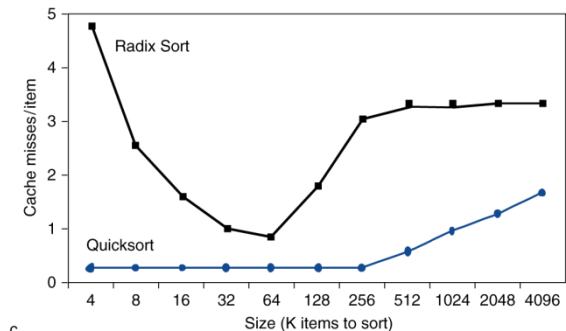
# Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced

- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```

# DGEMM Access Pattern

- C, A, and B arrays



older accesses

new accesses
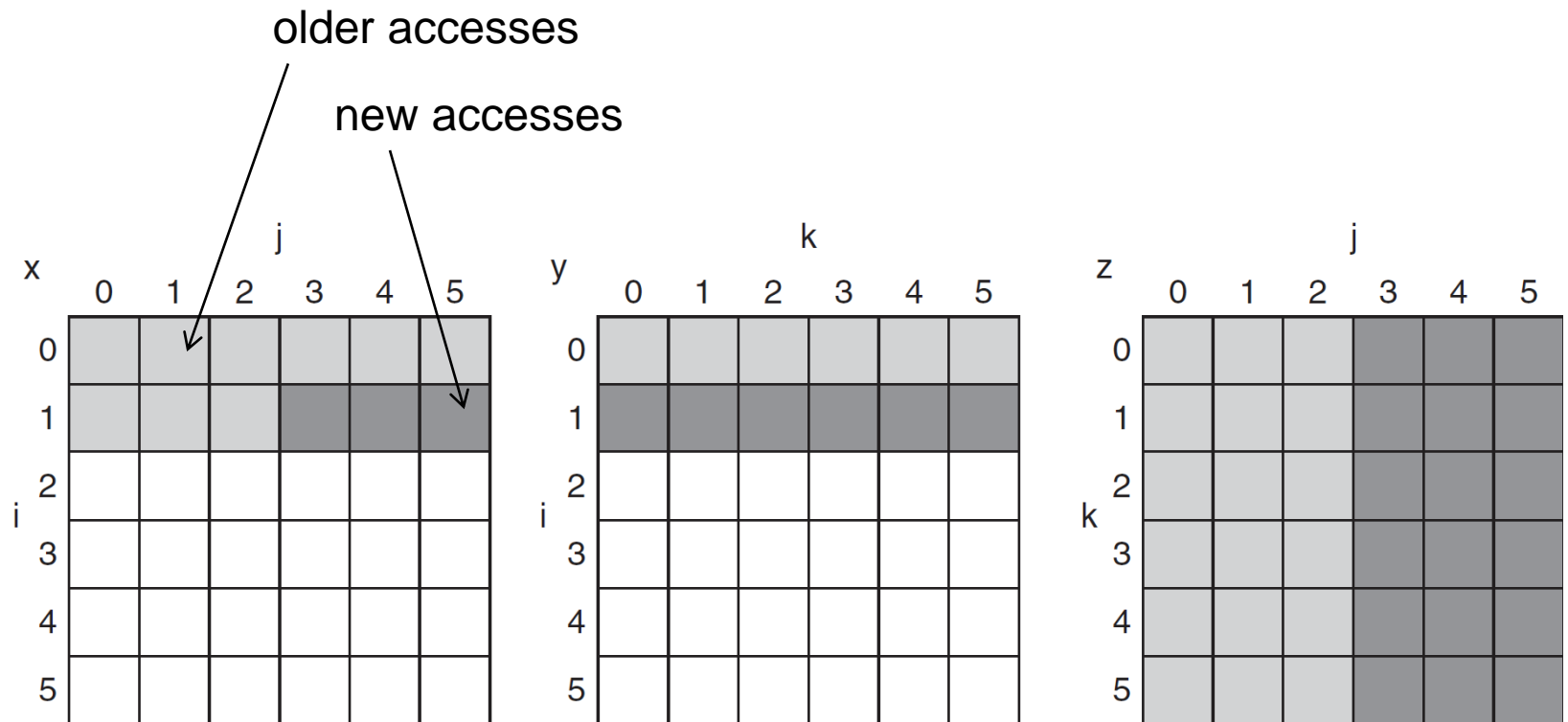
# Cache Blocked DGEMM

```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i+j*n];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11   C[i+j*n] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

# Blocked DGEMM Access Pattern

# Dependability

**Service accomplishment**
Service delivered
as specified

Restoration          Failure
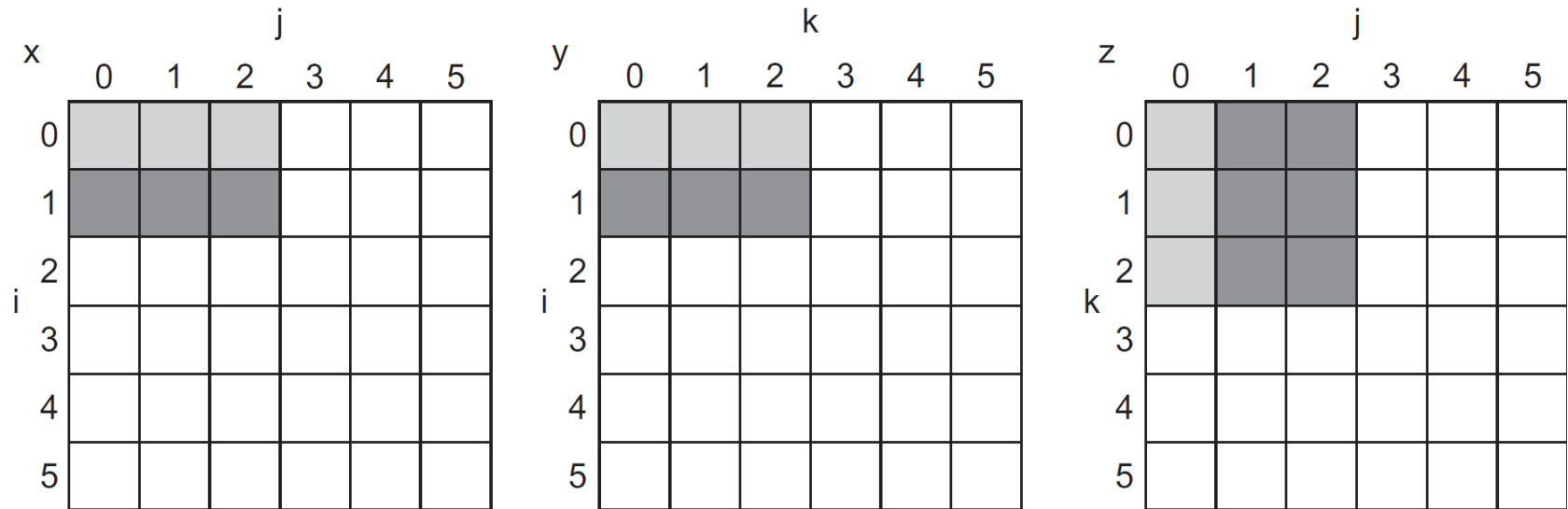
**Service interruption**
Deviation from
specified service
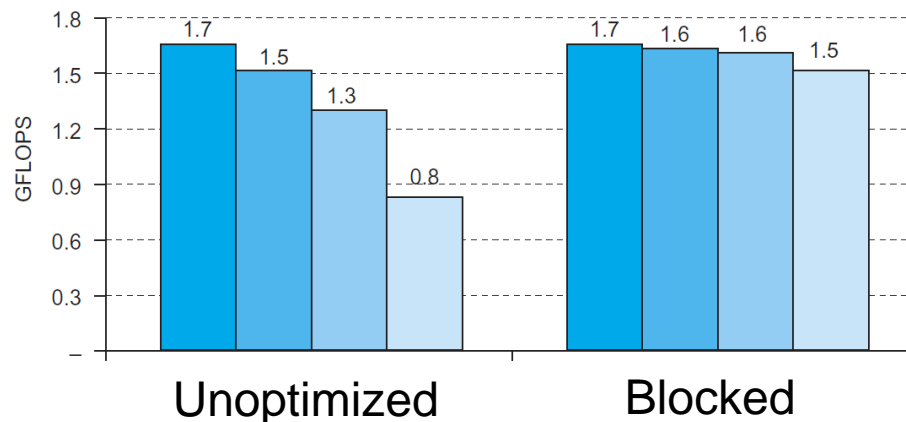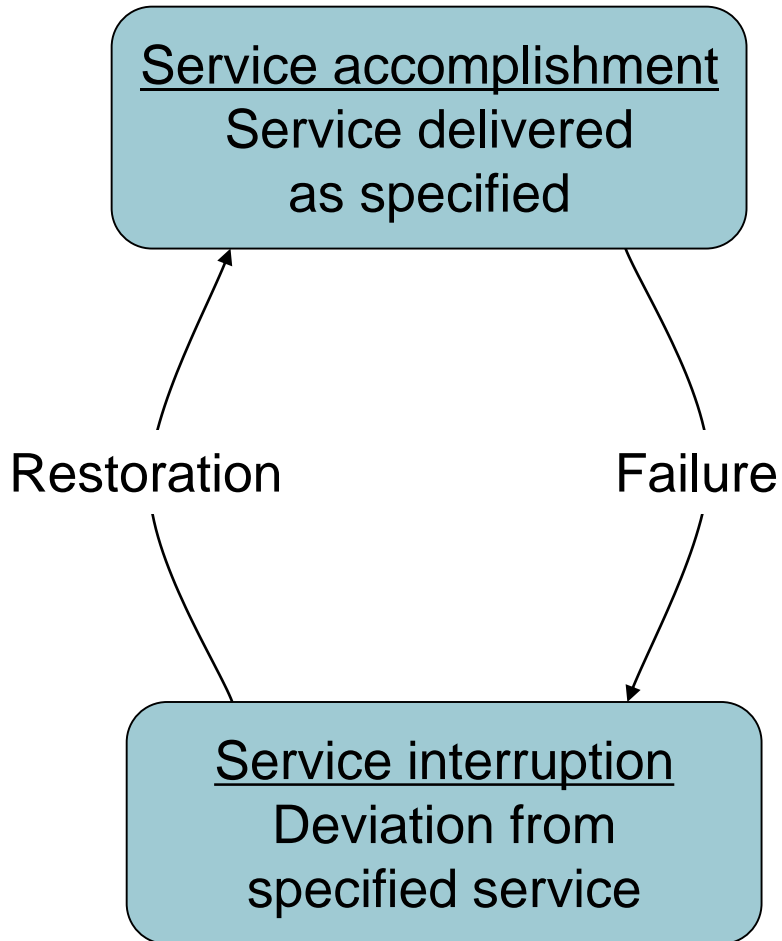
- Fault: failure of a component
  - May or may not lead to system failure

# Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

# The Hamming SEC Code

- **Hamming distance**
  - Number of bits that are different between two bit patterns

- **Minimum distance = 2 provides single bit error detection**
  - E.g. parity code

- **Minimum distance = 3 provides single error correction, 2 bit error detection**

# Encoding SEC

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:

| Bit position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate — p1 | X | | X | | X | | X | | X | | X | |
| Parity bit coverate — p2 | | X | X | | | X | X | | | X | X | |
| Parity bit coverate — p4 | | | | X | X | X | X | | | | | X |
| Parity bit coverate — p8 | | | | | | | | X | X | X | X | X |

# Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

# SEC/DED Code

- Add an additional parity bit for the whole word ($p_n$)

- Make Hamming distance = 4

- Decoding:
  - Let H = SEC parity bits
    - H even, $p_n$ even, no error
    - H odd, $p_n$ odd, correctable single bit error
    - H even, $p_n$ odd, error in $p_n$ bit
    - H odd, $p_n$ even, double error occurred

- Note: ECC DRAM uses SEC/DED with 8 bits protecting each 64 bits

# Virtual Machines

- Host computer emulates guest operating system and machine resources
    - Improved isolation of multiple guests
    - Avoids security and reliability problems
    - Aids sharing of resources
- Virtualization has some performance impact
    - Feasible with modern high-performance computers
- Examples
    - IBM VM/370 (1970s technology!)
    - VMWare
    - Microsoft Virtual PC

# Virtual Machine Monitor

- Maps virtual resources to physical resources
    - Memory, I/O devices, CPUs
- Guest code runs on native machine in user mode
    - Traps to VMM on privileged instructions and access to protected resources
- Guest OS may be different from host OS
- VMM handles real I/O devices
    - Emulates generic virtual I/O devices for guest

# Example: Timer Virtualization

- ## In native machine, on timer interrupt

  - OS suspends current process, handles interrupt, selects and resumes next process

- ## With Virtual Machine Monitor

  - VMM suspends current VM, handles interrupt, selects and resumes next VM

- ## If a VM requires timer interrupts

  - VMM emulates a virtual timer

  - Emulates interrupt for VM when physical timer interrupt occurs

# Instruction Set Support

- **User and System modes**
- **Privileged instructions only available in system mode**
  - Trap to system if executed in user mode
- **All physical resources only accessible using privileged instructions**
  - Including page tables, interrupt controls, I/O registers
- **Renaissance of virtualization support**
  - Current ISAs (e.g., x86) adapting

# Virtual Memory

- Use main memory as a "cache" for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM "block" is called a page
  - VM translation "miss" is called a page fault

# Address Translation

- Fixed-size pages (e.g., 4K)
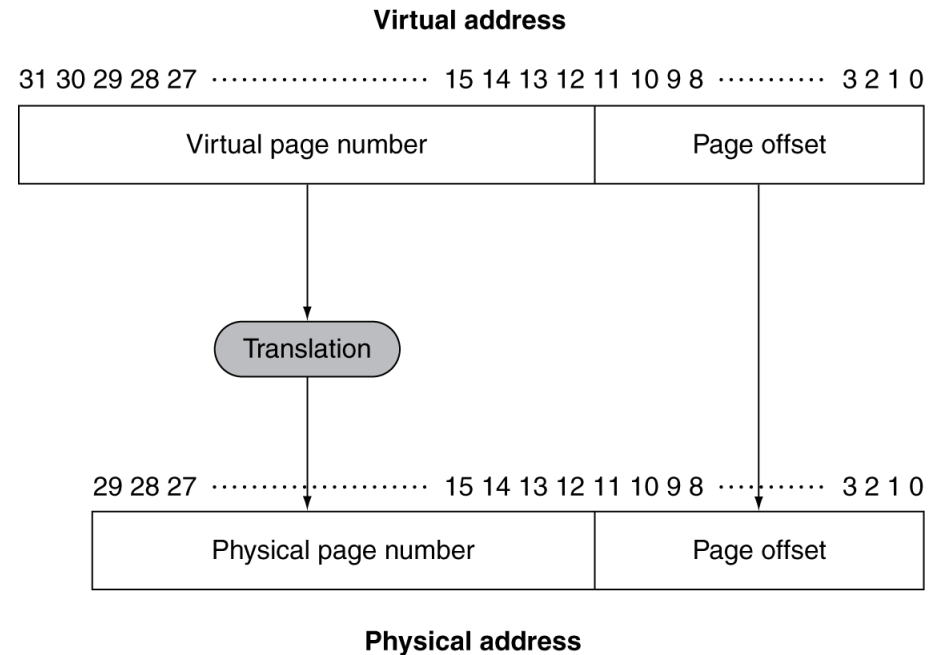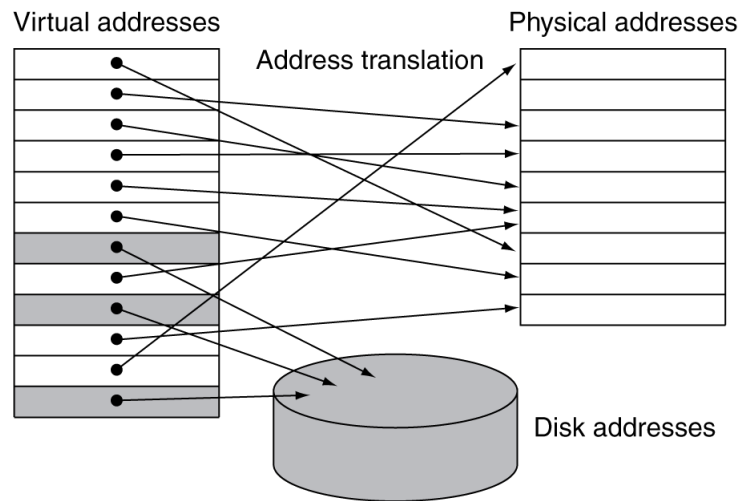
# Page Fault Penalty

- On page fault, the page must be fetched from disk
    - Takes millions of clock cycles
    - Handled by OS code
- Try to minimize page fault rate
    - Fully associative placement
    - Smart replacement algorithms

# Page Tables

- **Stores placement information**
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- **If page is present in memory**
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- **If page is not present**
  - PTE can refer to location in swap space on disk

# Translation Using a Page Table
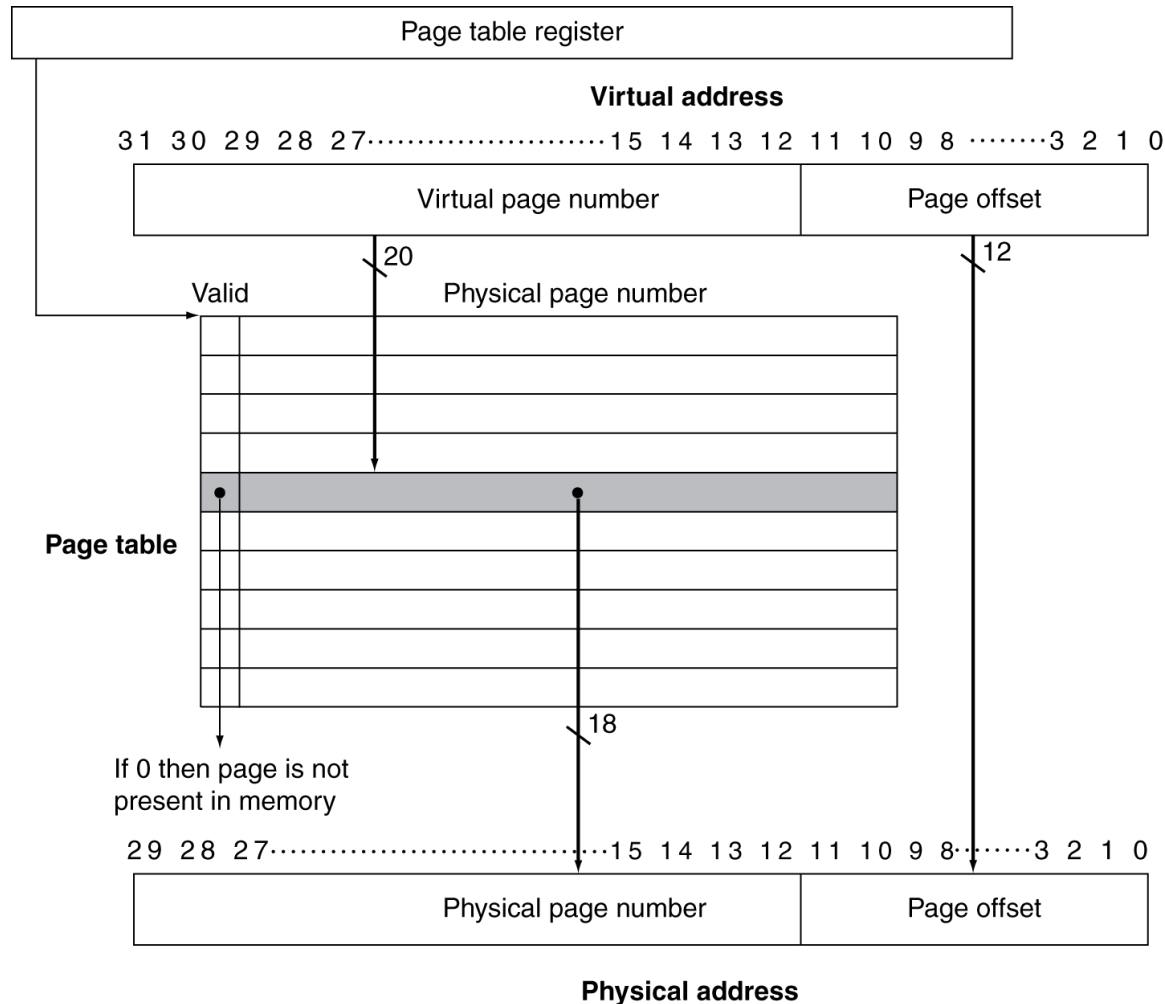
# Mapping Pages to Storage
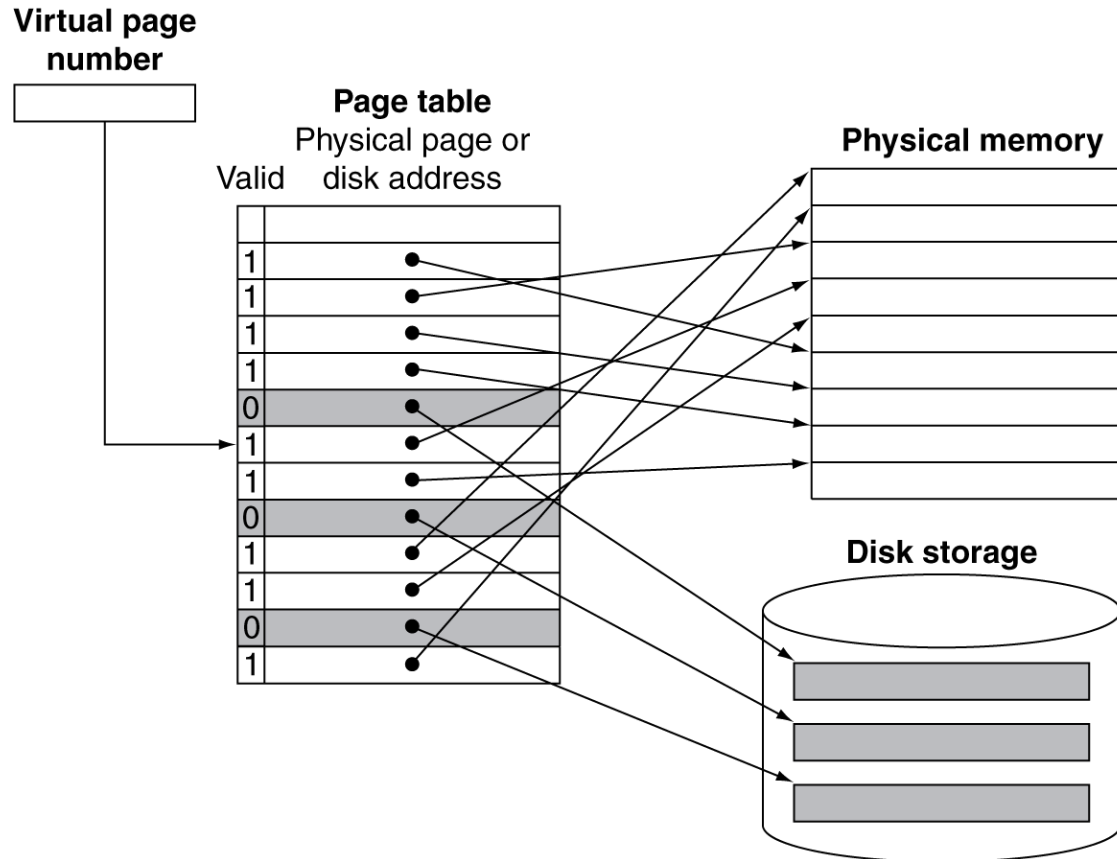
# Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
    - Reference bit (aka use bit) in PTE set to 1 on access to page
    - Periodically cleared to 0 by OS
    - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
    - Block at once, not individual locations
    - Write through is impractical
    - Use write-back
    - Dirty bit in PTE set when page is written

# Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

# Fast Translation Using a TLB
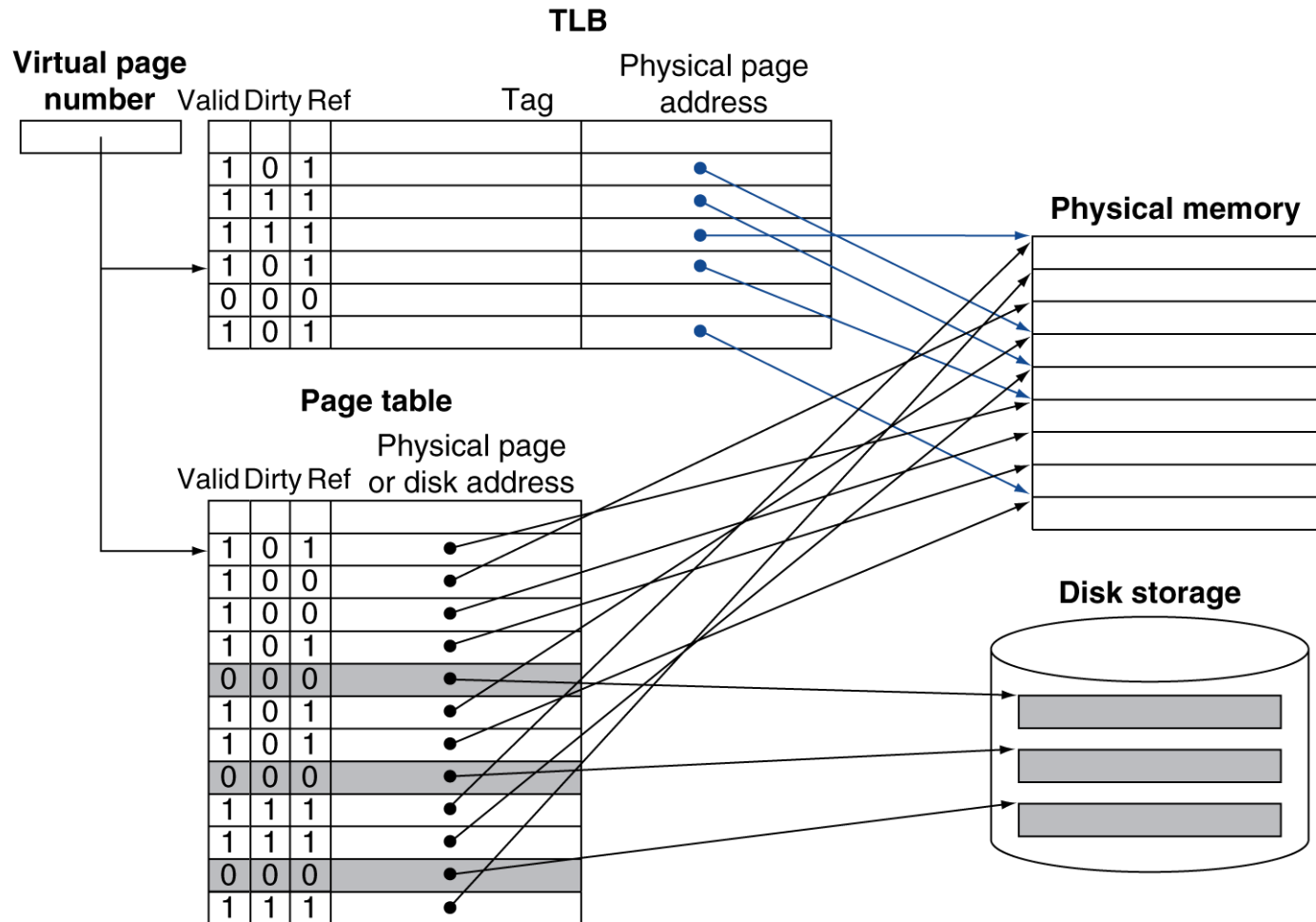
# TLB Misses

- **If page is in memory**
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- **If page is not in memory (page fault)**
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

# TLB Miss Handler

- ## TLB miss indicates

  - Page present, but PTE not in TLB

  - Page not present

- ## Must recognize TLB miss before destination register overwritten

  - Raise exception

- ## Handler copies PTE from memory to TLB

  - Then restarts instruction

  - If page not present, page fault will occur

# Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
    - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
    - Restart from faulting instruction

# TLB and Cache Interaction
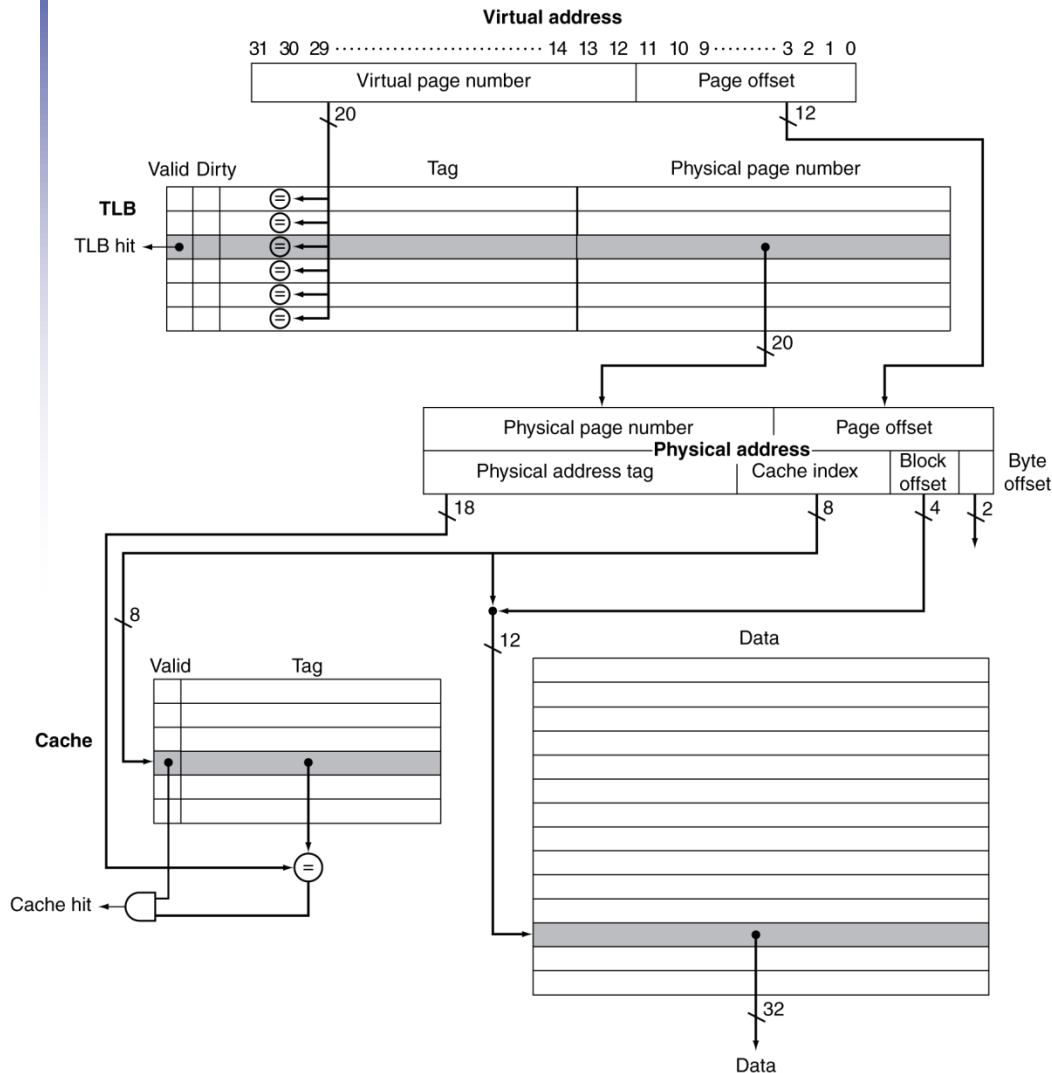


- If cache tag uses physical address
  - Need to translate before cache lookup

- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

# Possible combinations of events in the TLB, virtual memory system, and cache

| TLB | Page table | Cache | Possible? If so, under what circumstance? |
|---|---|---|---|
| Hit | Hit | Miss | Possible, although the page table is never really checked if TLB hits. |
| Miss | Hit | Hit | TLB misses, but entry found in page table; after retry, data is found in cache. |
| Miss | Hit | Miss | TLB misses, but entry found in page table; after retry, data misses in cache. |
| Miss | Miss | Miss | TLB misses and is followed by a page fault; after retry, data must miss in cache. |
| Hit | Miss | Miss | Impossible: cannot have a translation in TLB if page is not present in memory. |
| Hit | Miss | Hit | Impossible: cannot have a translation in TLB if page is not present in memory. |
| Miss | Miss | Hit | Impossible: data cannot be allowed in cache if the page is not in memory. |

# Memory Protection

- Different tasks can share parts of their virtual address spaces
    - But need to protect against errant access
    - Requires OS assistance

- Hardware support for OS protection
    - Privileged supervisor mode (aka kernel mode)
    - Privileged instructions
    - Page tables and other state information only accessible in supervisor mode
    - System call exception (e.g., syscall in MIPS)

# The Memory Hierarchy

**The BIG Picture**

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

# Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

# Finding a Block

| Associativity | Location method | Tag comparisons |
|---|---|---|
| Direct mapped | Index | 1 |
| n-way set associative | Set index, then search entries within the set | n |
| Fully associative | Search all entries | #entries |
| | Full lookup table | 0 |

- **Hardware caches**
  - Reduce comparisons to reduce cost
- **Virtual memory**
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

# Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support

# Write Policy

- ## Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- ## Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- ## Virtual memory
  - Only write-back is feasible, given disk write latency

# Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

# Cache Design Trade-offs

| Design change | Effect on miss rate | Negative performance effect |
|---|---|---|
| Increase cache size | Decrease capacity misses | May increase access time |
| Increase associativity | Decrease conflict misses | May increase access time |
| Increase block size | Decrease compulsory misses | Increases miss penalty. For very large block size, may increase miss rate due to pollution. |

# Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

| 31 | 14 13 | 4 3 | 0 |
|---|---|---|---|
| Tag | Index | Offset | |
| 18 bits | 10 bits | 4 bits | |

# Interface Signals



CPU

Read/Write
Valid
Address        32
Write Data     32
Read Data      32
Ready

Cache

Read/Write
Valid
Address        32
Write Data     128
Read Data      128
Ready

Memory
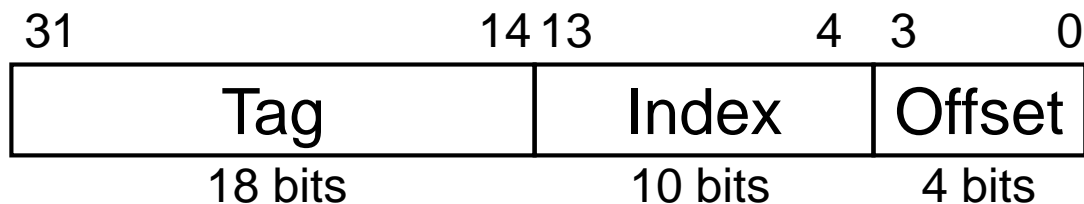
Multiple cycles
per access
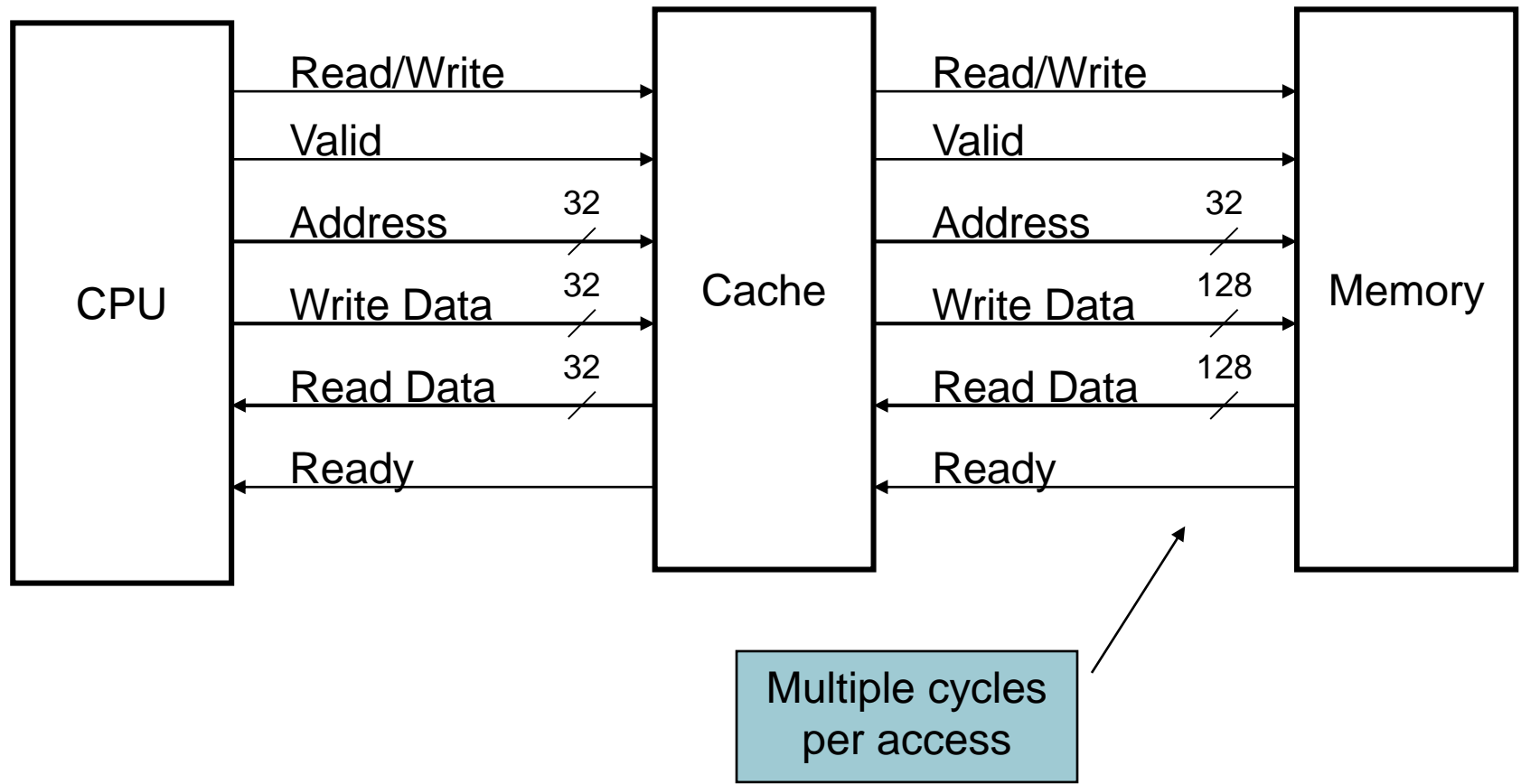
# Finite State Machines

- **Use an FSM to sequence control steps**

- **Set of states, transition on each clock edge**
  - State values are binary encoded
  - Current state stored in a register
  - Next state
    $= f_n$ (current state, current inputs)

- **Control output signals $= f_o$ (current state)**



Combinational control logic

Datapath control outputs

Outputs

Inputs

Next state

Inputs from cache datapath

State register

# Cache Controller FSM



**Idle**

Cache Hit
Mark Cache Ready

Valid CPU request

**Compare Tag**
If Valid && Hit ,
Set Valid, SetTag,
if Write Set Dirty

Could partition into separate states to reduce clock cycle time

Cache Miss and Old Block is Clean

Cache Miss and Old Block is Dirty

Memory Ready

**Allocate**
Read new block from Memory

Memory not Ready

Memory Ready

**Write-Back**
Write Old Block to Memory

Memory not Ready

# Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

| Time step | Event | CPU A's cache | CPU B's cache | Memory |
|:---:|:---|:---:|:---:|:---:|
| 0 | | | | 0 |
| 1 | CPU A reads X | 0 | | 0 |
| 2 | CPU B reads X | 0 | 0 | 0 |
| 3 | CPU A writes 1 to X | 1 | 0 | 1 |

# Coherence Defined

- Informally: Reads return most recently written value

- Formally:
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - $P_1$ writes X; $P_2$ reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - $P_1$ writes X, $P_2$ writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

# Cache Coherence Protocols

- ## Operations performed by caches in multiprocessors to ensure coherence
  - ### Migration of data to local caches
    - Reduces bandwidth for shared memory
  - ### Replication of read-shared data
    - Reduces contention for access
- ## Snooping protocols
  - ### Each cache monitors bus reads/writes
- ## Directory-based protocols
  - ### Caches and memory record sharing status of blocks in a directory

# Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

| CPU activity | Bus activity | CPU A's cache | CPU B's cache | Memory |
|---|---|---|---|---|
|  |  |  |  | 0 |
| CPU A reads X | Cache miss for X | 0 |  | 0 |
| CPU B reads X | Cache miss for X | 0 | 0 | 0 |
| CPU A writes 1 to X | Invalidate for X | 1 |  | 0 |
| CPU B read X | Cache miss for X | 1 | 1 | 1 |

# Memory Consistency

- When are writes seen by other processors
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

# 冗余廉价磁盘阵列

Data disks

Redundant check disks

RAID 0
(No redundancy)
Widely used

RAID 1
(Mirroring)
EMC, HP(Tandem), IBM

RAID 2
(Error detection and
correction code) Unused

RAID 3
(Bit-interleaved parity)
Storage concepts

RAID 4
(Block-interleaving parity)
Network appliance

RAID 5
(Distributed block-
interleaved parity)
Widely used

RAID 6
(P + Q redundancy)
Recently popular
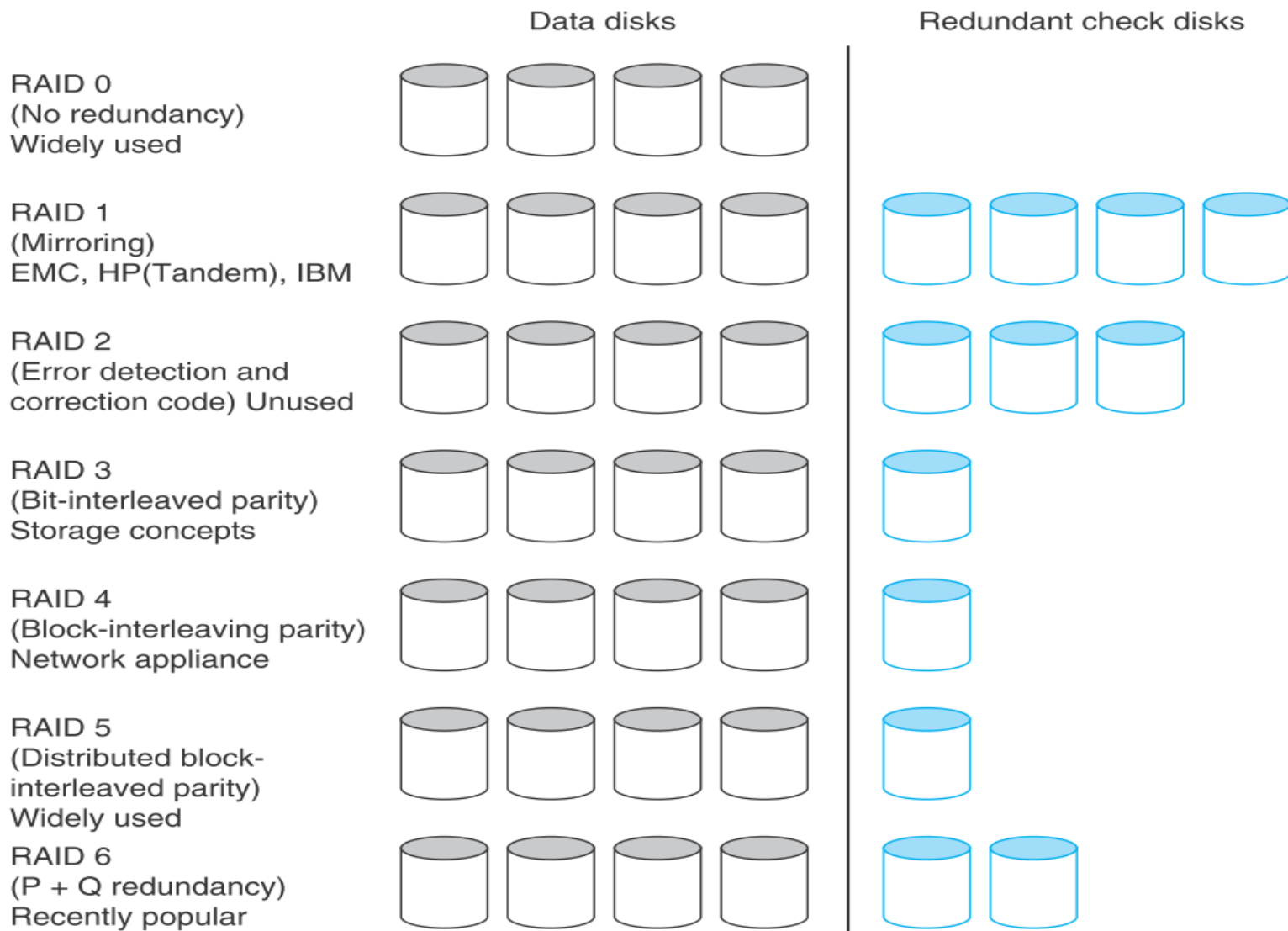
# RAID 0

- 条区（**strip**）
  - 简单地把数据分布于多个磁盘
- 自动强制访问多个磁盘
- 无冗余

**RAID 0**

| Disk 1 | Disk 2 | Disk 3 | Disk 4 | Disk 5 |
|--------|--------|--------|--------|--------|
| A1 | A2 | A3 | A4 | A5 |
| B1 | B2 | B3 | B4 | B5 |
| C1 | C2 | C3 | C4 | C5 |
| D1 | D2 | D3 | D4 | D5 |
| E1 | E2 | E3 | E4 | E5 |
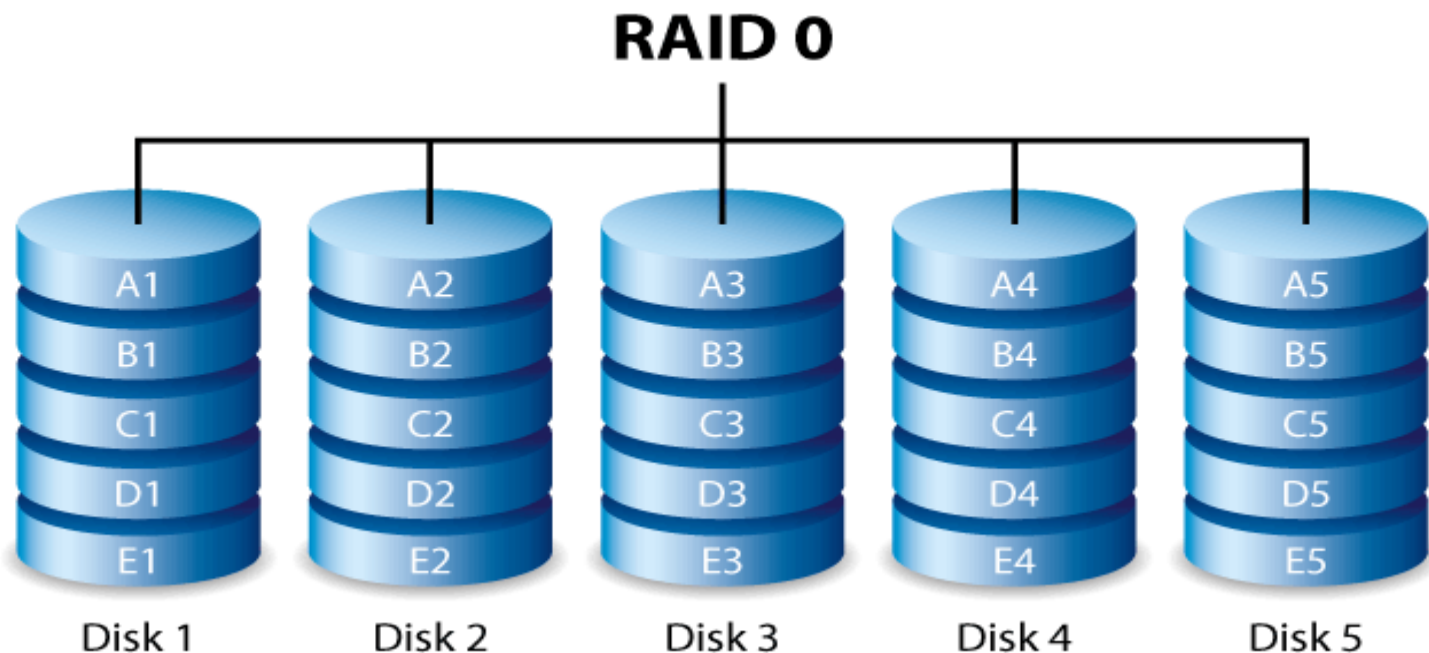
# RAID 1

- 镜像（**Mirroring**）

- 使用的磁盘数量是 **RAID 0**的两倍

- 最贵的**RAID**解决方案



RAID 1

Block 1 | Block 1
Block 2 | Block 2
Block 3 | Block 3
Block 4 | Block 4

DRIVE 1 | DRIVE 2

Mirrored Data to both Drives
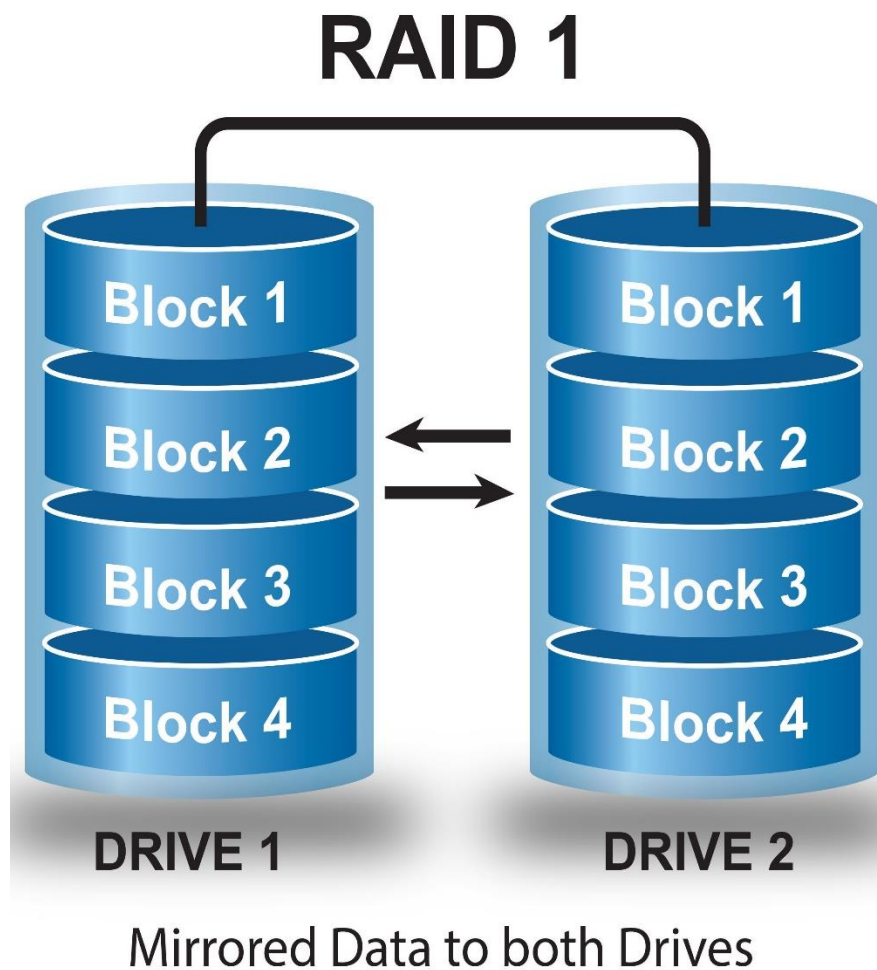
# RAID 2

- 用汉明纠错码来实现冗余
- 条区非常小—1个字节或字
- 读写请求需要所有的磁盘

RAID 2

| Disk 0 | Disk 1 | Disk 2 | Disk 3 | Disk 4 | Disk 5 | Disk 6 |
|--------|--------|--------|--------|--------|--------|--------|
| A1 | A2 | A3 | A4 | $A_{p1}$ | $A_{p2}$ | $A_{p3}$ |
| B1 | B2 | B3 | B4 | $B_{p1}$ | $B_{p2}$ | $B_{p3}$ |
| C1 | C2 | C3 | C4 | $C_{p1}$ | $C_{p2}$ | $C_{p3}$ |
| D1 | D2 | D3 | D4 | $D_{p1}$ | $D_{p2}$ | $D_{p3}$ |
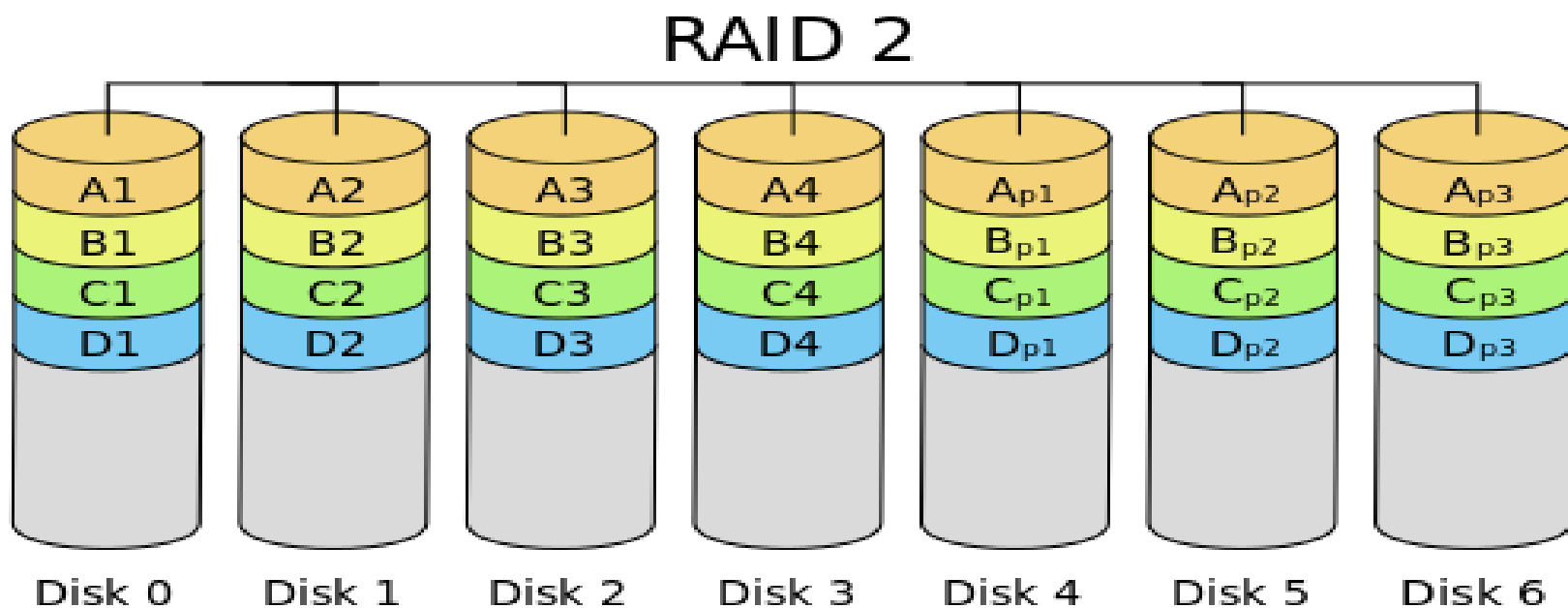
# RAID 3

- 用奇偶位实现冗余
- 条区非常小—1个字节或字
- 读写请求需要所有的磁盘
- 传输率高，**I/O**请求率差
- 易于从故障中恢复（**XOR**）

RAID 3
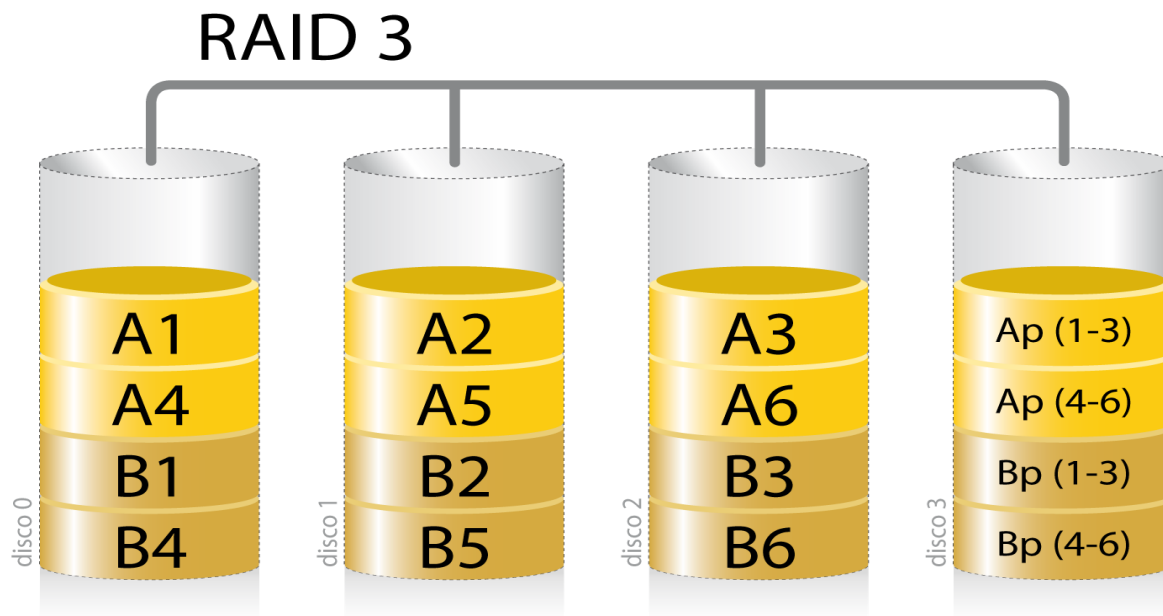
# RAID 4

- 块级奇偶校验
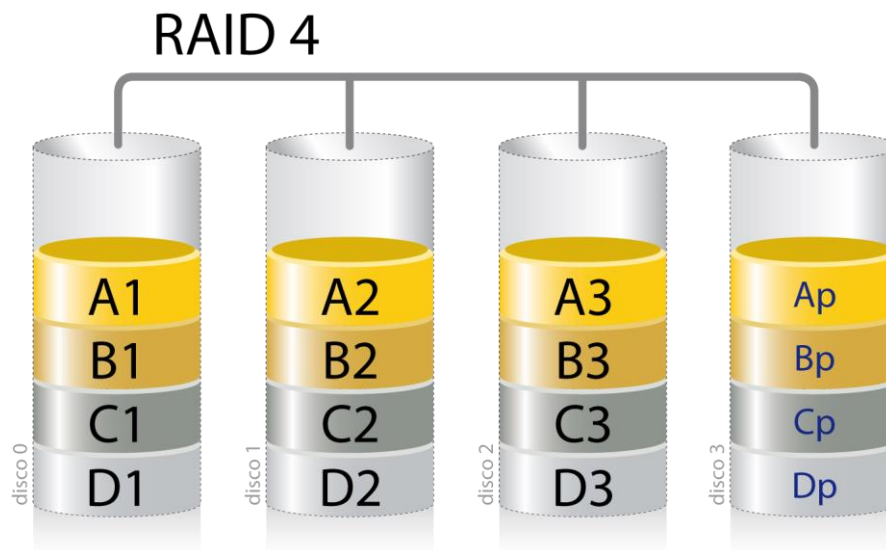- 读操作需要相关磁盘
- 写操作需要相关磁盘和奇偶检验盘
- 易于从故障中恢复（**XOR**）
- 高速**I/O**请求

RAID 4

| disco 0 | disco 1 | disco 2 | disco 3 |
|---------|---------|---------|---------|
| A1 | A2 | A3 | Ap |
| B1 | B2 | B3 | Bp |
| C1 | C2 | C3 | Cp |
| D1 | D2 | D3 | Dp |

# RAID 5

- 分布式块交叉奇偶校验
- 读操作需求相关磁盘
- 写操作需要对特定块的相关磁盘和奇偶校验盘
- 易于从故障中恢复（**XOR**）

**RAID 5**

| Disk 1 | Disk 2 | Disk 3 | Disk 4 | Disk 5 |
|--------|--------|--------|--------|--------|
| A1 | A2 | A3 | A4 | Ap |
| B1 | B2 | B3 | Bp | B4 |
| C1 | C2 | Cp | C3 | C4 |
| D1 | Dp | D2 | D3 | D4 |
| Ep | E1 | E2 | E3 | E4 |
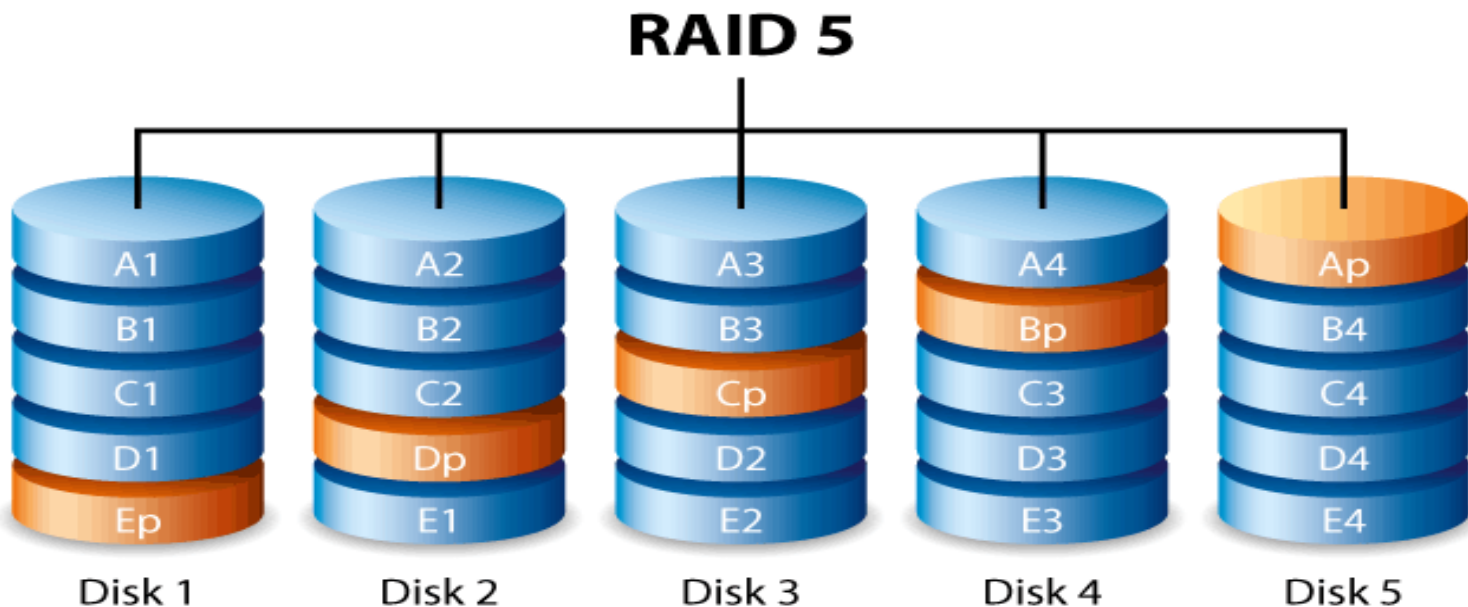
# RAID 6

- 块级条区
- 双奇偶校验**RAID**
  - 每个磁盘有两个奇偶检验块
- **RAID 5**增强版



RAID 6

| A1 | A2 | A3 | Ap | Aq |
| B1 | B2 | Bp | Bq | B3 |
| C1 | Cp | Cq | C3 | C3 |
| Dp | Dq | D2 | D3 | D3 |

Disk 1    Disk 2    Disk 3    Disk 4    Disk 5
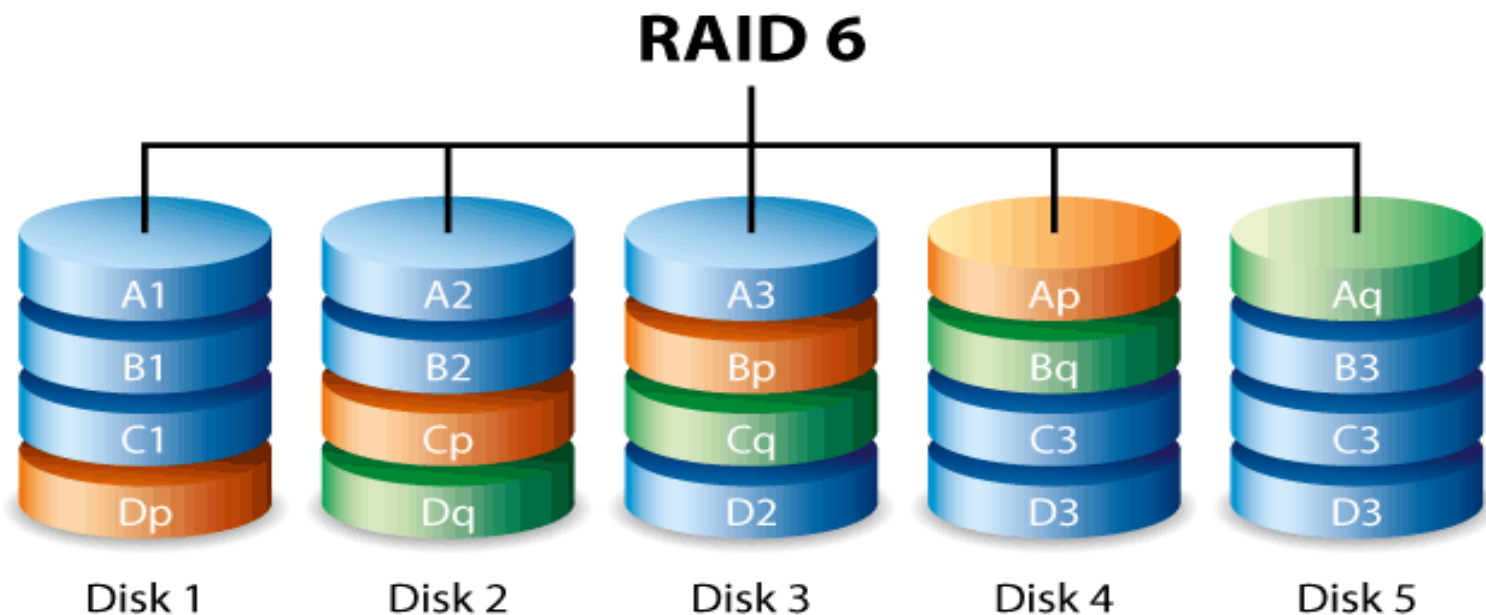
# Multilevel On-Chip Caches

| Characteristic | ARM Cortex-A8 | Intel Nehalem |
|---|---|---|
| L1 cache organization | Split instruction and data caches | Split instruction and data caches |
| L1 cache size | 32 KiB each for instructions/data | 32 KiB each for instructions/data per core |
| L1 cache associativity | 4-way (I), 4-way (D) set associative | 4-way (I), 8-way (D) set associative |
| L1 replacement | Random | Approximated LRU |
| L1 block size | 64 bytes | 64 bytes |
| L1 write policy | Write-back, Write-allocate(?) | Write-back, No-write-allocate |
| L1 hit time (load-use) | 1 clock cycle | 4 clock cycles, pipelined |
| L2 cache organization | Unified (instruction and data) | Unified (instruction and data) per core |
| L2 cache size | 128 KiB to 1 MiB | 256 KiB (0.25 MiB) |
| L2 cache associativity | 8-way set associative | 8-way set associative |
| L2 replacement | Random(?) | Approximated LRU |
| L2 block size | 64 bytes | 64 bytes |
| L2 write policy | Write-back, Write-allocate (?) | Write-back, Write-allocate |
| L2 hit time | 11 clock cycles | 10 clock cycles |
| L3 cache organization | – | Unified (instruction and data) |
| L3 cache size | – | 8 MiB, shared |
| L3 cache associativity | – | 16-way set associative |
| L3 replacement | – | Approximated LRU |
| L3 block size | – | 64 bytes |
| L3 write policy | – | Write-back, Write-allocate |
| L3 hit time | – | 35 clock cycles |

# 2-Level TLB Organization

| Characteristic | ARM Cortex-A8 | Intel Core i7 |
|---|---|---|
| Virtual address | 32 bits | 48 bits |
| Physical address | 32 bits | 44 bits |
| Page size | Variable: 4, 16, 64 KiB, 1, 16 MiB | Variable: 4 KiB, 2/4 MiB |
| TLB organization | 1 TLB for instructions and 1 TLB for data<br><br>Both TLBs are fully associative, with 32 entries, round robin replacement<br><br>TLB misses handled in hardware | 1 TLB for instructions and 1 TLB for data per core<br><br>Both L1 TLBs are four-way set associative, LRU replacement<br><br>L1 I-TLB has 128 entries for small pages, 7 per thread for large pages<br><br>L1 D-TLB has 64 entries for small pages, 32 for large pages<br><br>The L2 TLB is four-way set associative, LRU replacement<br><br>The L2 TLB has 512 entries<br><br>TLB misses handled in hardware |

# Supporting Multiple Issue

- Both have multi-banked caches that allow multiple accesses per cycle assuming no bank conflicts

- Core i7 cache optimizations
  - Return requested word first
  - Non-blocking cache
    - Hit under miss
    - Miss under miss
  - Data prefetching

# DGEMM

- Combine cache blocking and subword parallelism



Legend: 32x32 · 160x160 · 480x480 · 960x960

| | 32x32 | 160x160 | 480x480 | 960x960 |
|---|---|---|---|---|
| Unoptimized | 1.7 | 1.5 | 1.3 | 0.8 |
| AVX | 6.4 | 3.5 | 2.3 | 2.5 |
| AVX + unroll | 14.6 | 6.6 | 4.7 | 5.1 |
| AVX + unroll + blocked | 13.6 | 12.7 | 11.7 | 12.0 |

(GFLOPS)
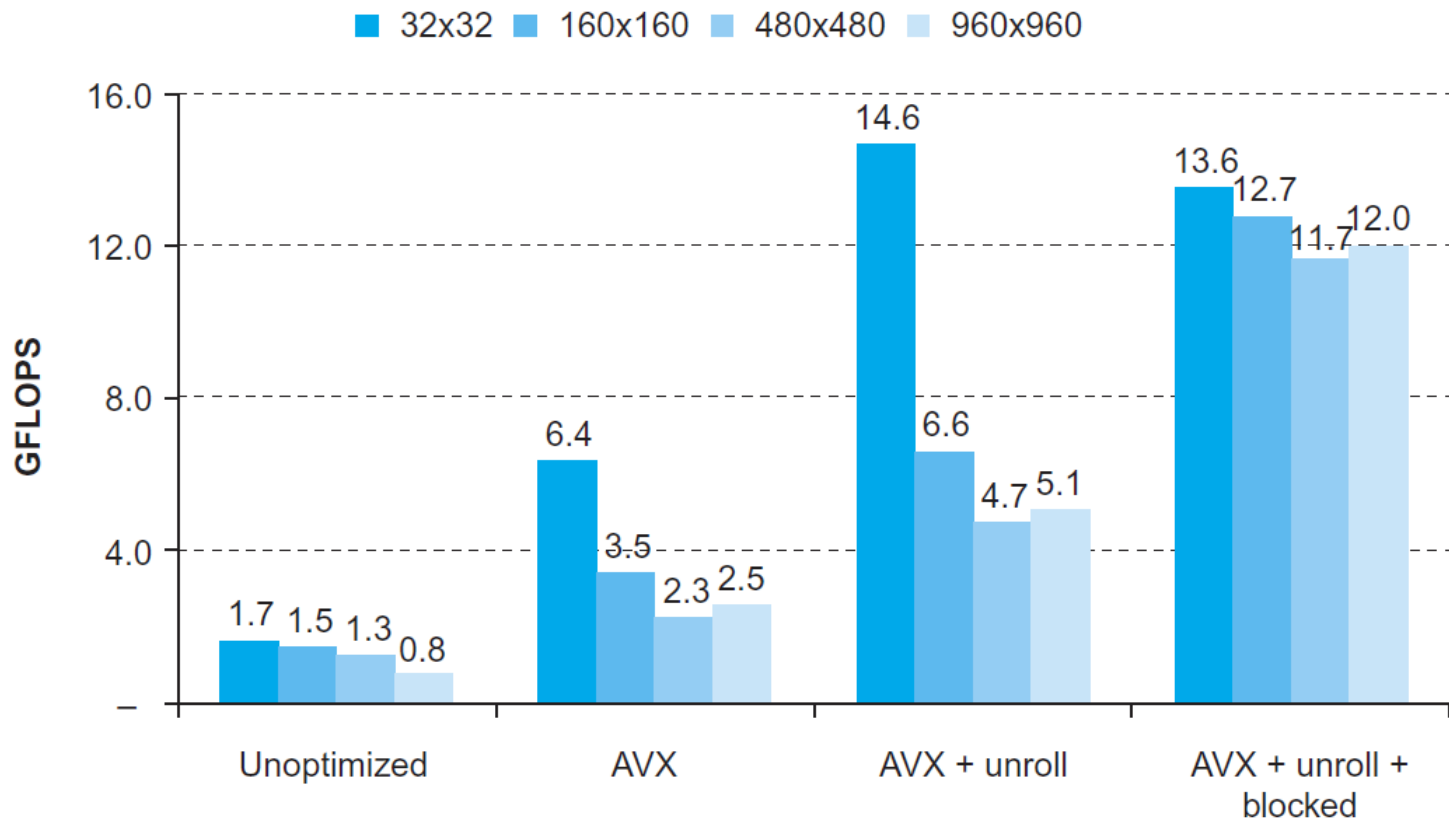
# Pitfalls

- ## Byte vs. word addressing
  - ### Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4

- ## Ignoring memory system effects when writing or generating code
  - ### Example: iterating over rows vs. columns of arrays
  - ### Large strides result in poor locality

# Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores $\Rightarrow$ need to increase associativity
- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

# Pitfalls

- Extending address range using segments
  - E.g., Intel 80286
  - But a segment is not always big enough
  - Makes address arithmetic complicated
- Implementing a VMM on an ISA not designed for virtualization
  - E.g., non-privileged instructions accessing hardware resources
  - Either extend ISA, or require guest OS not to use problematic instructions

# Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- Principle of locality
  - Programs use a small part of their memory space frequently
- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- Memory system design is critical for multiprocessors