UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 20: Memory Hierarchies (Registers, Caches, Main Memory, Storage)

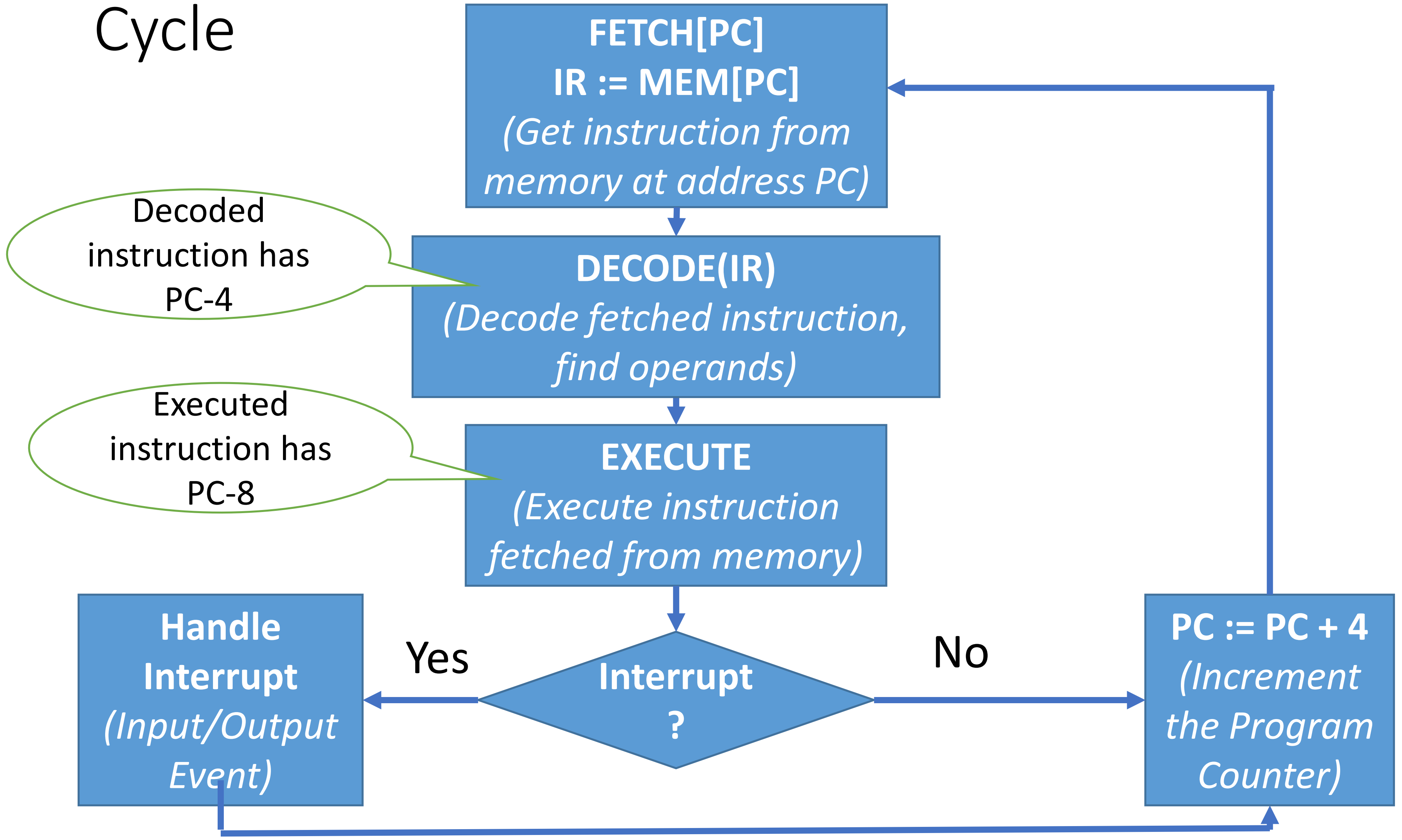
Taylor Johnson

Announcements and Outline

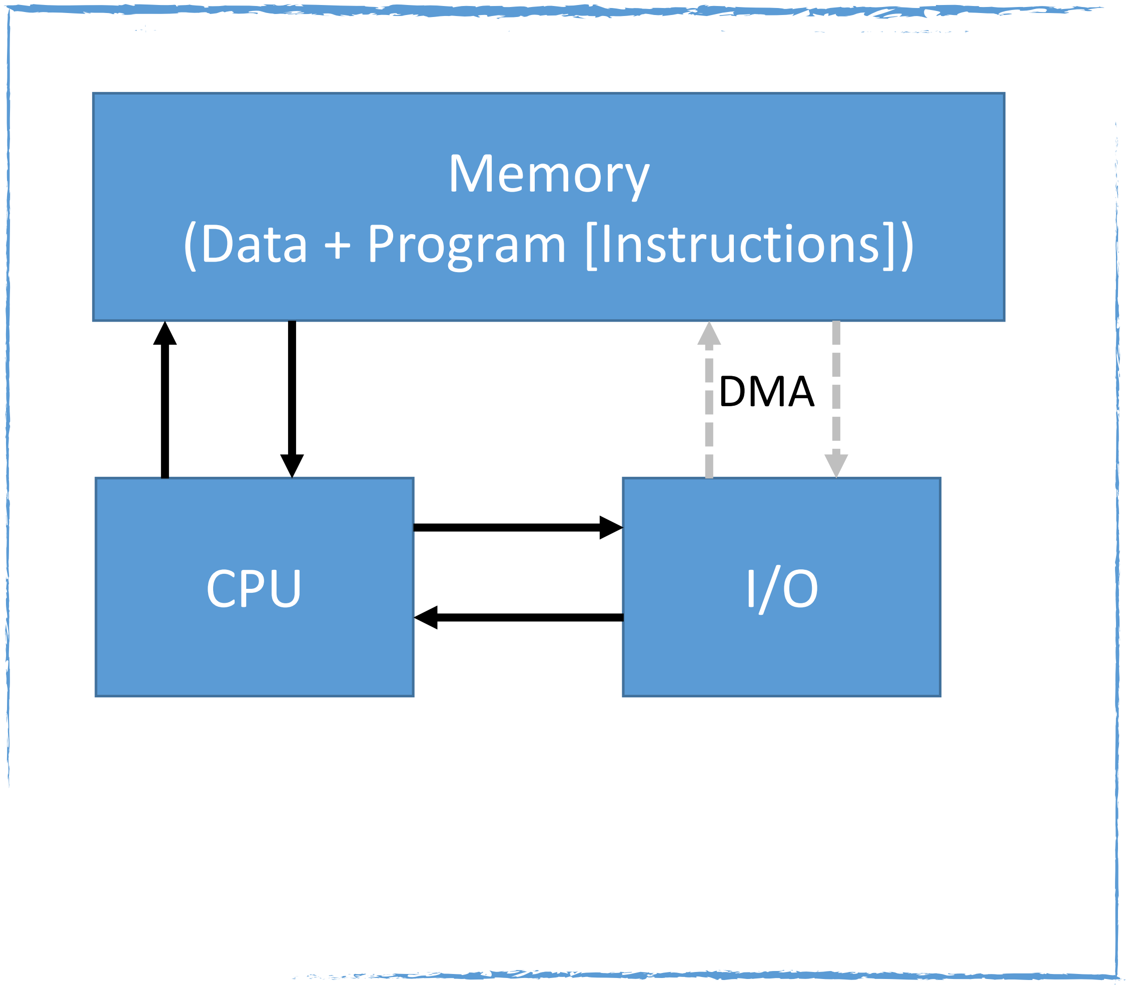
- Programming assignment 1 assigned, due 11/4

- Review
  - Input/output
  - Exceptions and Interrupts
  - Example Debugging UART Interaction with gdb
- Memory Hierarchies
  - Registers
  - Caches
  - Main Memory
  - Storage
  - …

ARM 3-Stage Pipeline Processor Execution Cycle

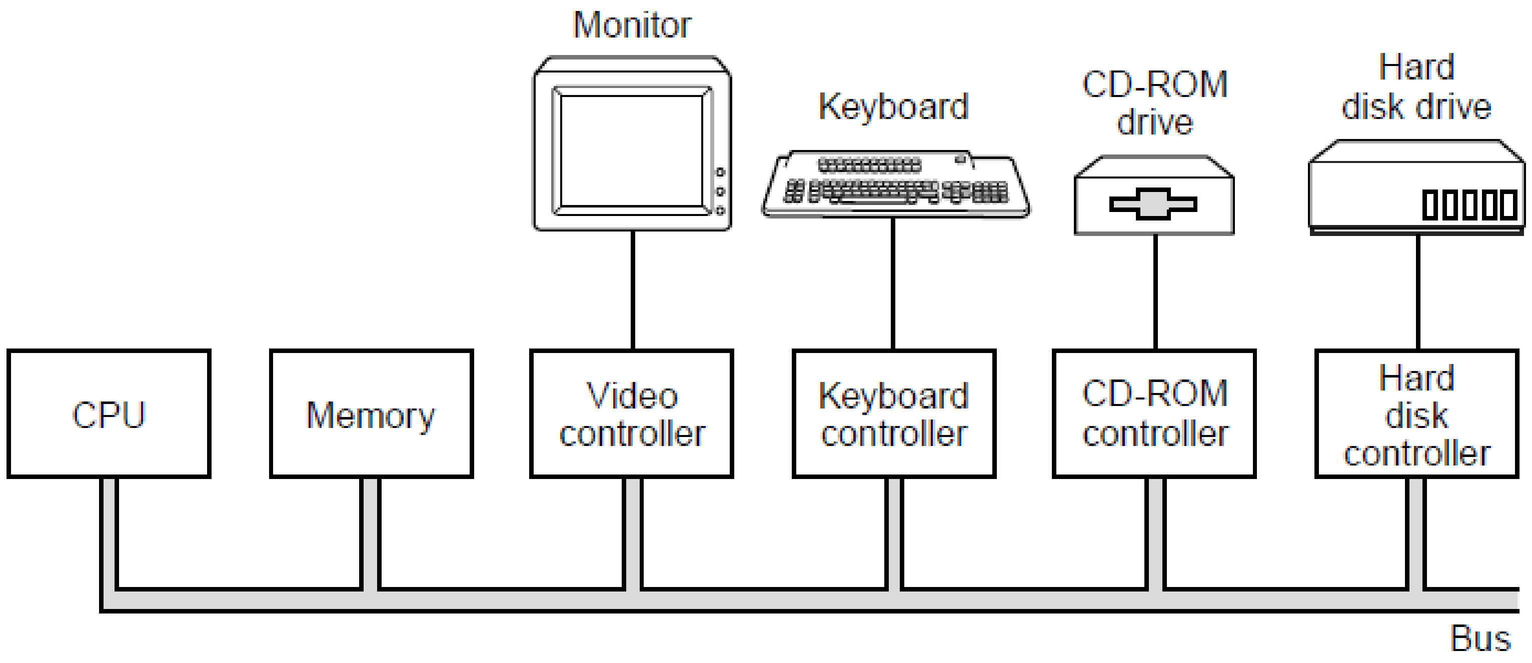

FETCH[PC]
IR := MEM[PC]
(Get instruction from memory at address PC)
DECODE(IR)
(Decode fetched instruction, find operands)
EXECUTE
(Execute instruction fetched from memory)
Interrupt ?
Yes
No
Handle Interrupt (Input/Output Event)
PC := PC + 4 (Increment the Program Counter)
Decoded instruction has PC-4
Executed instruction has PC-8

Von Neumann Architecture



- Both data and program stored in memory
- Allows the computer to be “re-programmed”
- Input/output (I/O) goes through CPU
- I/O part is not representative of modern systems (direct memory access [DMA])
- Memory layout is representative of modern systems

Logical Structure of a Computer



Logical structure of a simple personal computer.

The PCI Bus

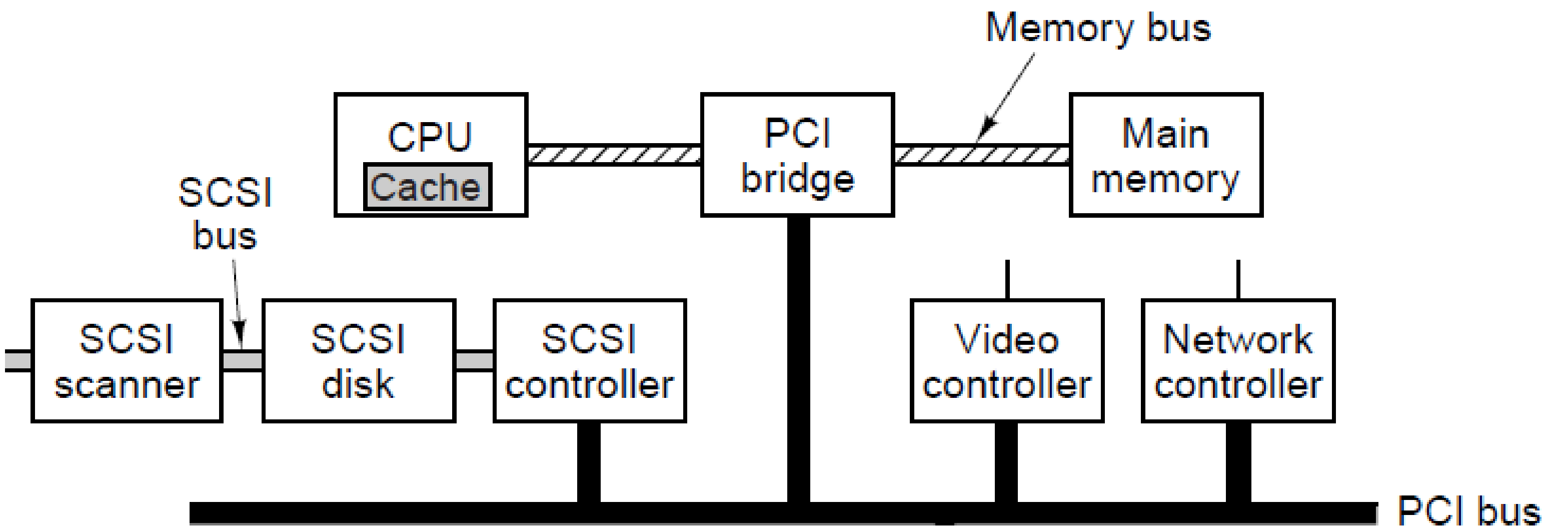


Figure 2-31. A typical PC built around the PCI bus. The SCSI controller is a PCI device.

The PCIe Bus

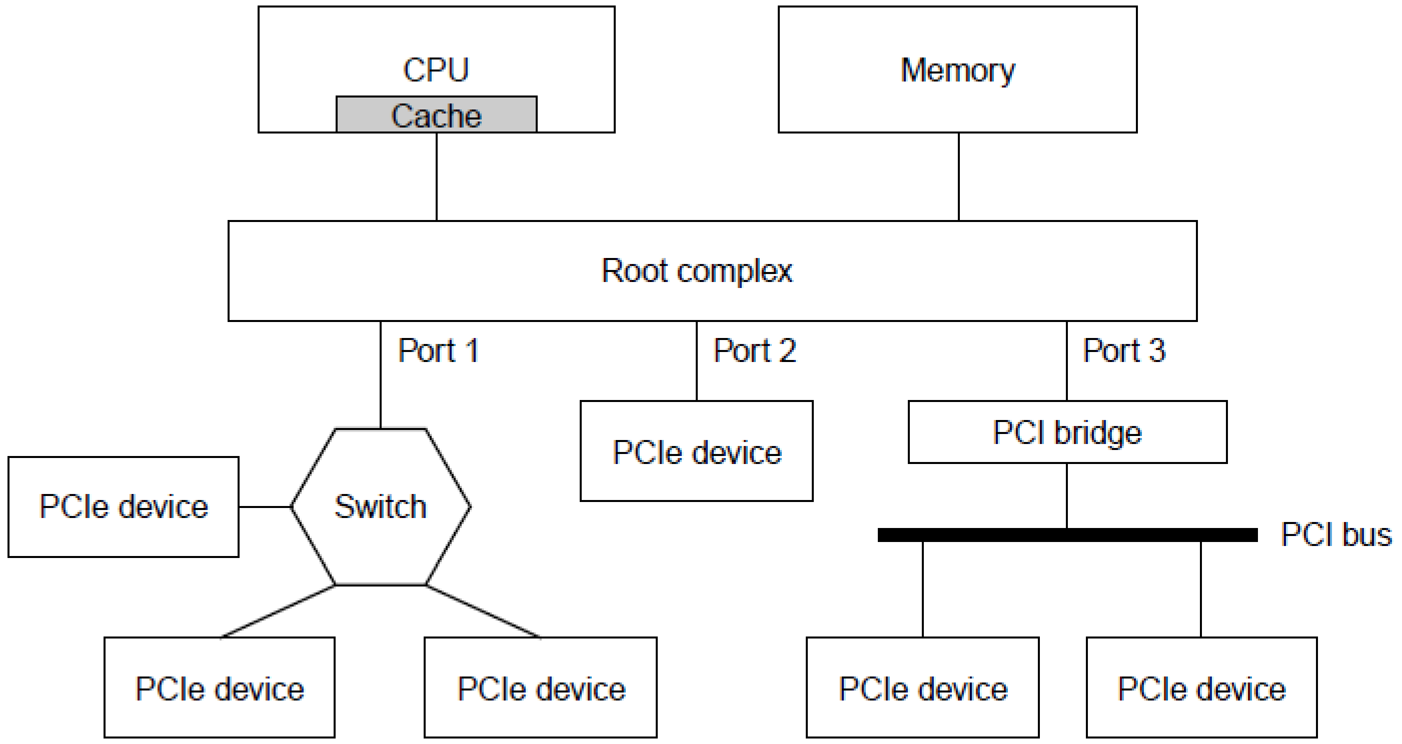


Figure 2-32. Sample architecture of a PCIe system with three PCIe ports.

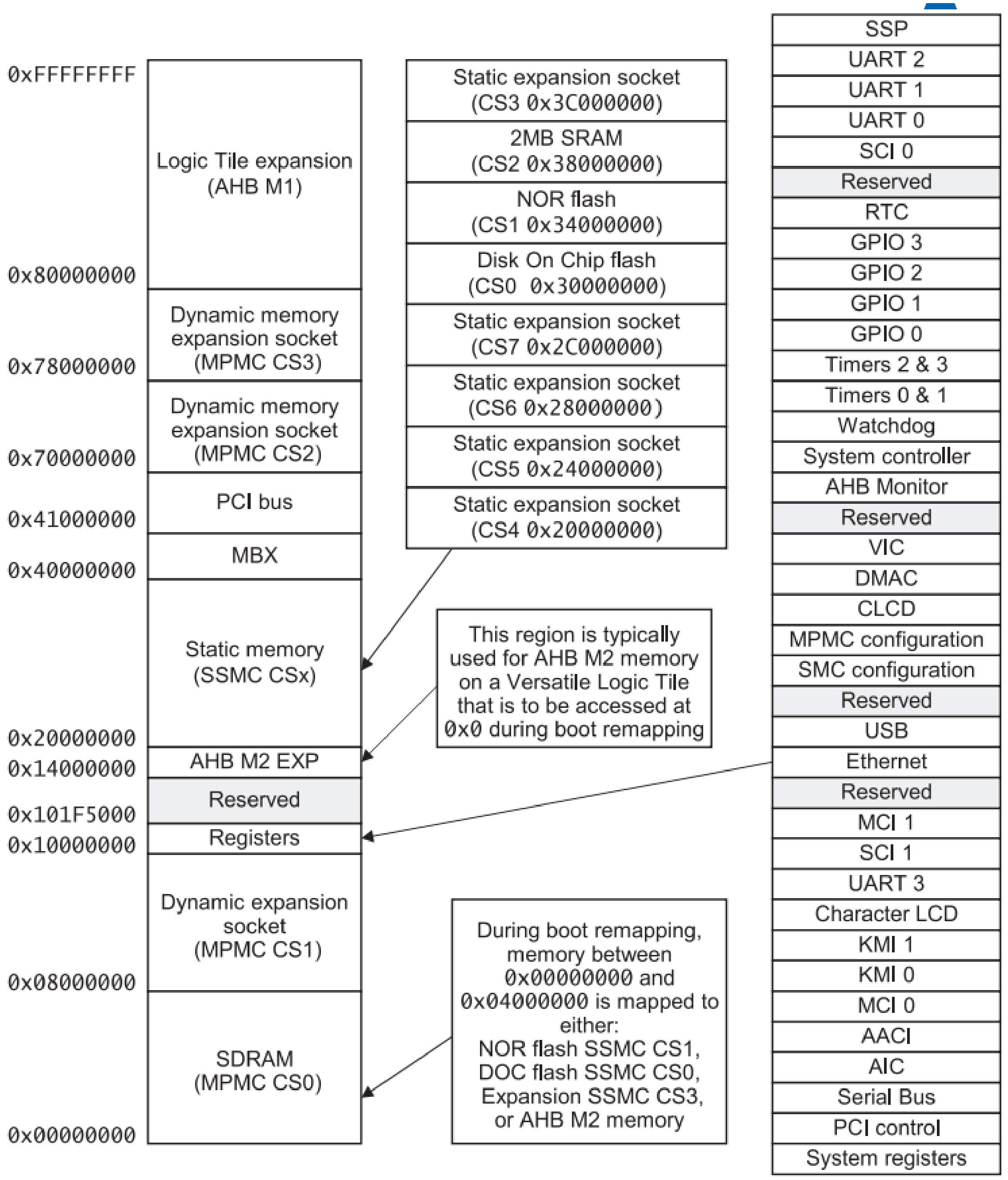
Memory-Mapped I/O

- Some of our original examples displayed output to console by writing to a special memory address

```
.equ    ADDR_UART0, 0x101f1000
ldr         r0,=ADDR_UART0@ r0 := 0x 101f 1000
mov         r2,#’a’                 @ R2 := ‘a’
str         r2,[r0]                 @ MEM[r0] := r2
```

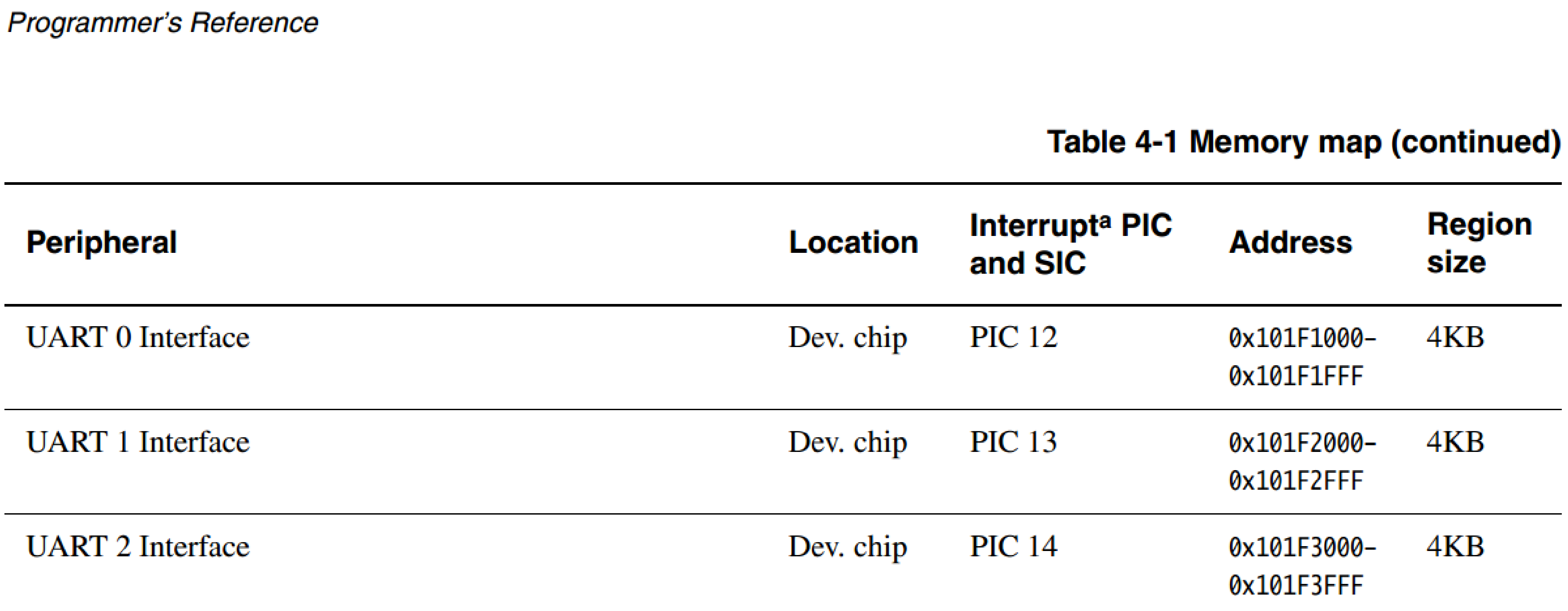
- How does this work? Memory Mapped I/O
  - Registers on peripheral devices (keyboards, monitors, network controllers, etc.) are addressable in same address space as main memory, and their values are mapped (i.e., readable / writeable at certain addresses)
  - How to read input values?
    - Polling vs. interrupts

ARLINGTON


0xFFFFFFFF
0x80000000
0x78000000
0x70000000
0x41000000
0x40000000
0x20000000
0x14000000
0x101F5000
0x10000000
0x08000000
0x00000000
Logic Tile expansion (AHB M1)
Dynamic memory expansion socket (MPMC CS3)
Dynamic memory expansion socket (MPMC CS2)
PCI bus
MBX
Static memory (SSMC CSx)
AHB M2 EXP
Reserved
Registers
Dynamic expansion socket (MPMC CS1)
SDRAM (MPMC CS0)
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either: NOR flash SSMC CS1, DOC flash SSMC CS0, Expansion SSMC CS3, or AHB M2 memory
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers

UNIVERSITY OF TEXAS ARLINGTON

Address from Memory-Map in Manual



Programmer's Reference

Table 4-1 Memory map (continued)

Peripheral	Location	Interrupt[a] PIC and SIC	Address	Region size
UART 0 Interface	Dev. chip	PIC 12	0x101F1000-0x101F1FFF	4KB
UART 1 Interface	Dev. chip	PIC 13	0x101F2000-0x101F2FFF	4KB
UART 2 Interface	Dev. chip	PIC 14	0x101F3000-0x101F3FFF	4KB

http://infocenter.arm.com/help/topic/com.arm.doc.dui0224i/DUI0224I_realview_platform_baseboard_for_arm926ej_s_ug.pdf

UART Register Map

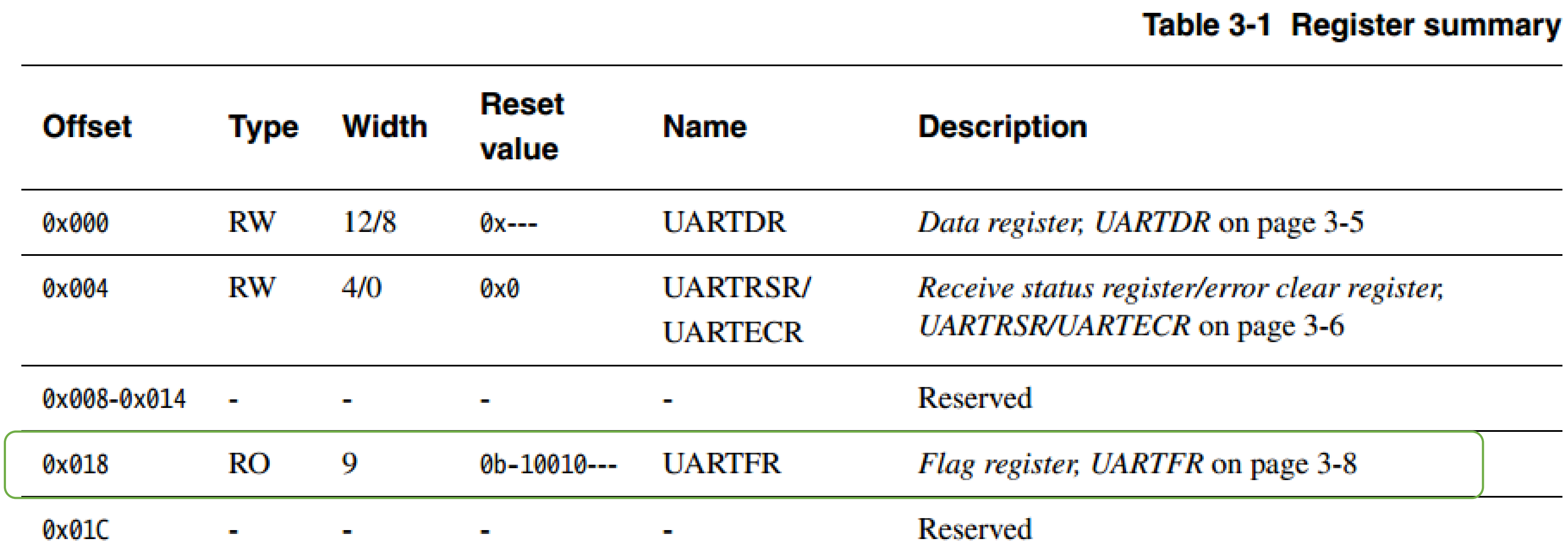
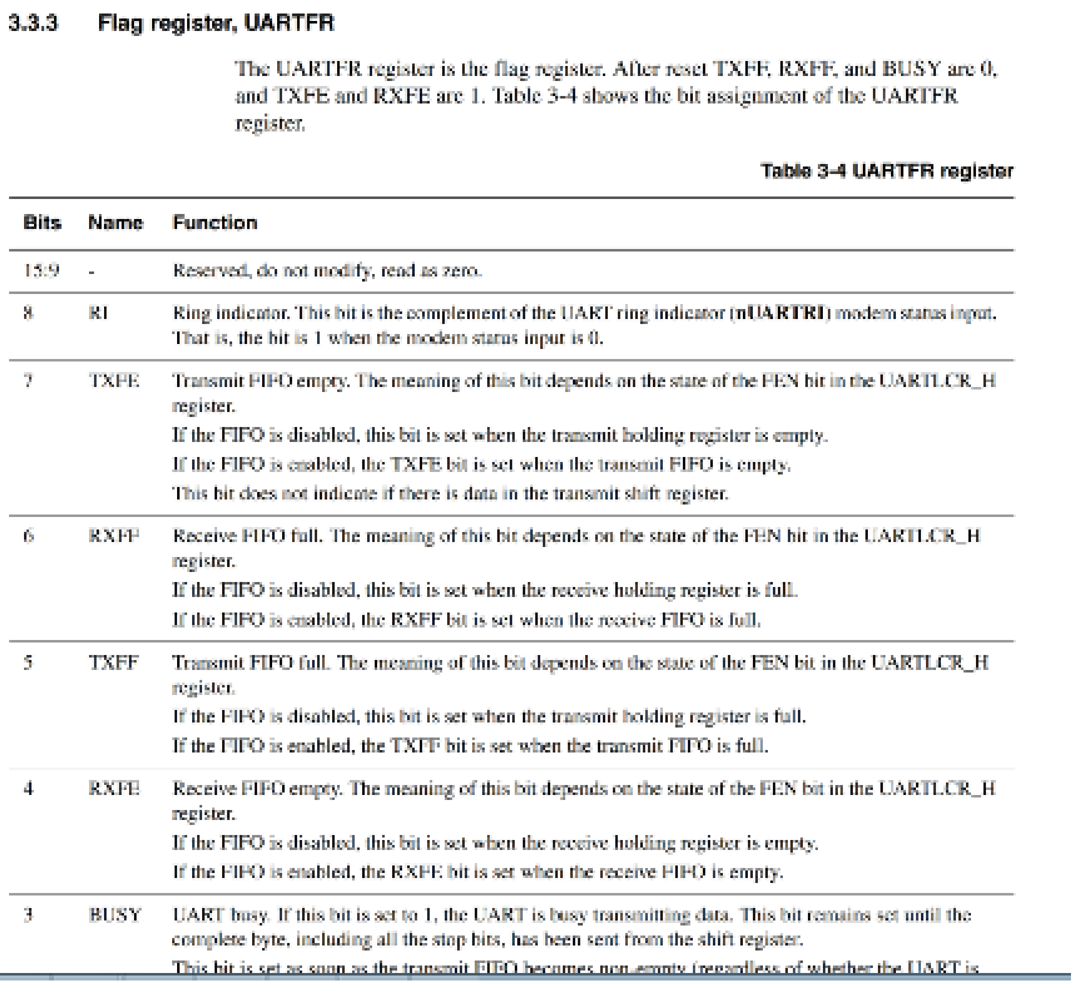


Table 3-1 Register summary

Offset	Type	Width	Reset value	Name	Description
`0x000`	RW	12/8	`0x---`	UARTDR	*Data register, UARTDR* on page 3-5
`0x004`	RW	4/0	`0x0`	UARTRSR/ UARTECR	*Receive status register/error clear register, UARTRSR/UARTECR* on page 3-6
`0x008-0x014`	-	-	-	-	Reserved
`0x018`	RO	9	`0b-10010---`	UARTFR	*Flag register, UARTFR* on page 3-8
`0x01C`	-	-	-	-	Reserved

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf

UART Flag Register Bits



3.3.3 Flag register, UARTFR

The UARTFR register is the flag register. After reset TXFF, RXFF, and BUSY are 0, and TXFE and RXFE are 1. Table 3-4 shows the bit assignment of the UARTFR register.

Table 3-4 UARTFR register

Bits	Name	Function
15:9	-	Reserved, do not modify, read as zero.
8	RI	Ring indicator. This bit is the complement of the UART ring indicator (**nUARTRI**) modem status input. That is, the bit is 1 when the modem status input is 0.
7	TXFE	Transmit FIFO empty. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the transmit holding register is empty. If the FIFO is enabled, the TXFE bit is set when the transmit FIFO is empty. This bit does not indicate if there is data in the transmit shift register.
6	RXFF	Receive FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the receive holding register is full. If the FIFO is enabled, the RXFF bit is set when the receive FIFO is full.
5	TXFF	Transmit FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the transmit holding register is full. If the FIFO is enabled, the TXFF bit is set when the transmit FIFO is full.
4	RXFE	Receive FIFO empty. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the receive holding register is empty. If the FIFO is enabled, the RXFE bit is set when the receive FIFO is empty.
3	BUSY	UART busy. If this bit is set to 1, the UART is busy transmitting data. This bit remains set until the complete byte, including all the stop bits, has been sent from the shift register. This bit is set as soon as the transmit FIFO becomes non-empty (regardless of whether the UART is

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf

Reading Input from UART (***<u>POLLING</u>***)

```
.equ IO_ADDRESS, 0x101f1000   @ uart memory-map address
.equ OFFSET_FR, 0x018         @ flag register offset from uart
.equ RXFE, 0x10               @ receive status bit
.equ TXFF, 0x20               @ transmit status bit

get_char:
      push {r2, r3, r4, lr}   @ preamble
      ldr r4,=IO_ADDRESS      @ r4 := 0x 101f 1000

get_char_wait:
      ldr r2,[r4,#OFFSET_FR]  @ load IO flag register to r2
      and r3,r2,#RXFE         @ mask non receive fifo empty bits
      cmp r3, #0              @ check if r3 == 0
      bne get_char_wait       @ wait if not ready (if r3 != 0)

      ldr r0, [r4]            @ read character
      str r0, [r4]            @ echo character to screen

      pop {r2, r3, r4, lr}    @ wrap up
      bx lr
```

Viewing Memory-Mapped Registers with gdb

```
.equ IO_ADDRESS, 0x101f1000    @ uart memory-map address
.equ OFFSET_FR, 0x018          @ flag register offset from uart
.equ RXFE, 0x10                @ receive status bit
```

```
(gdb) x /16x 0x101f1000                     <- View all registers
0x101f1000:     0x00000000      0x00000000      0x00000000
     0x00000000
0x101f1010:     0x00000000      0x00000000      0x00000090
     0x00000000
0x101f1020:     0x00000000      0x00000000      0x00000000
     0x00000000
0x101f1030:     0x00000300      0x00000012      0x00000000
     0x00000020
(gdb) x /1x 0x101f1000+0x018                <- View Flag Register
0x101f1018:     0x00000090
(gdb) x /1t 0x101f1000+0x018                <- View Flag Register
0x101f1018:     00000000000000000000000010010000
(gdb) x /1t 0x101f1000+0x018     <- Character entered
0x101f1018:     00000000000000000000000011000000
```

Printing Strings

```
@ assumes r0 contains uart data register address
@ r1 should contain first character of string to display
print_string: push  {r1,r2,lr}
str_out: ldrb  r2,[r1]
     cmp  r2,#0x00  @ '\0' = 0x00: null character?
     beq  str_done  @ if yes, quit
     str  r2,[r0]   @ otherwise, write char of string
     add  r1,r1,#1  @ go to next character
     b    str_out   @ repeat
str_done: pop  {r1,r2,lr}
     bx   lr
```

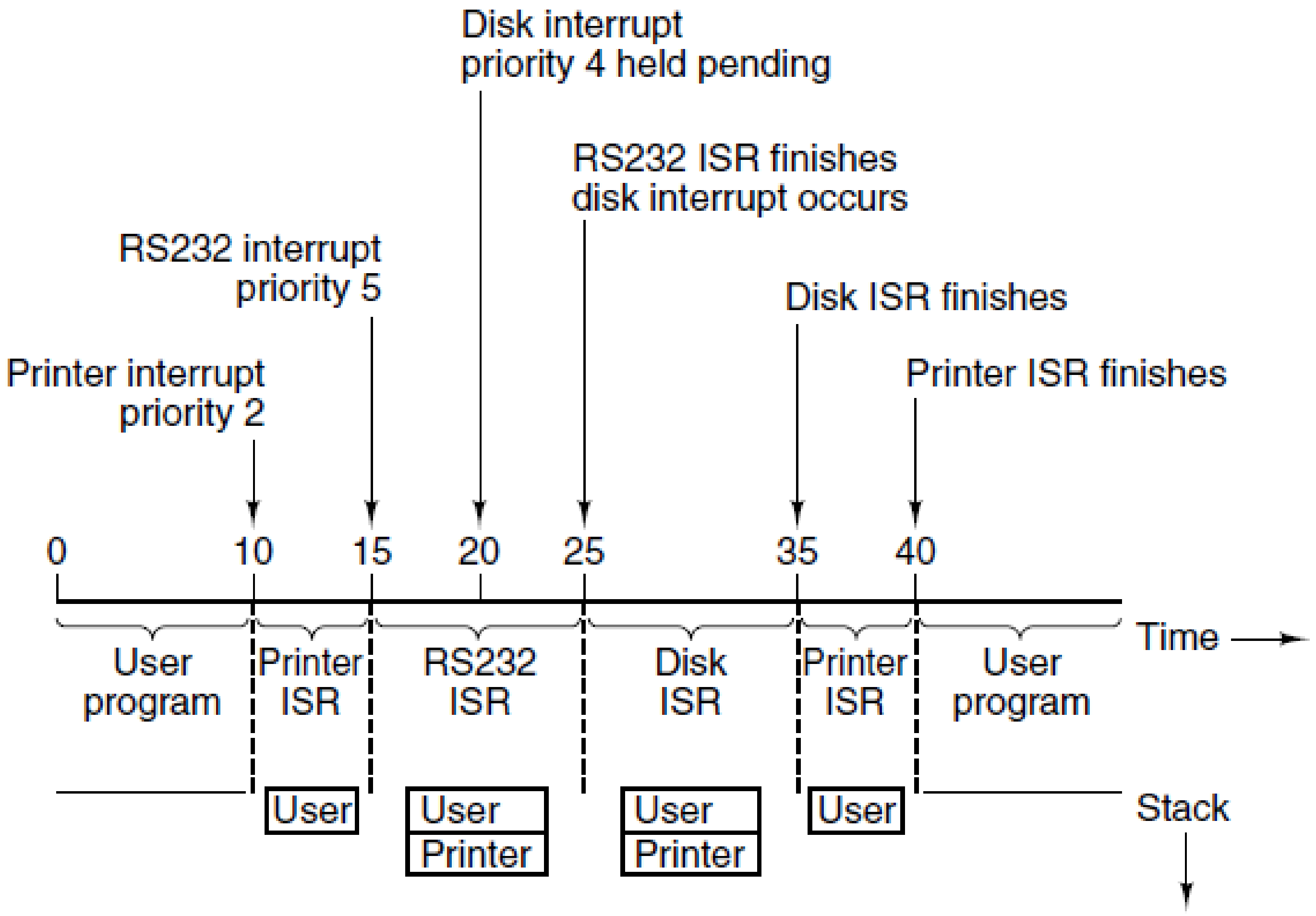
Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception (also called a trap or fault)
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, divide by zero, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Interrupt Service Routines (ISR)

- Coroutine: Procedure that resumes execution at previous point
- Special routines that may interrupt other routines
  - Interrupt: stop execution of procedure A and start execution of procedure B
  - Useful for I/O: suppose we have data coming from the keyboard
  - Priorities: if we have several ISRs, how do we decide which gets executed now?
- Interrupt handler (or ISR): PC gets loaded with address of ISR
- ISR specified in an interrupt vector table

ISR Timeline



Traps / Exceptions / Faults

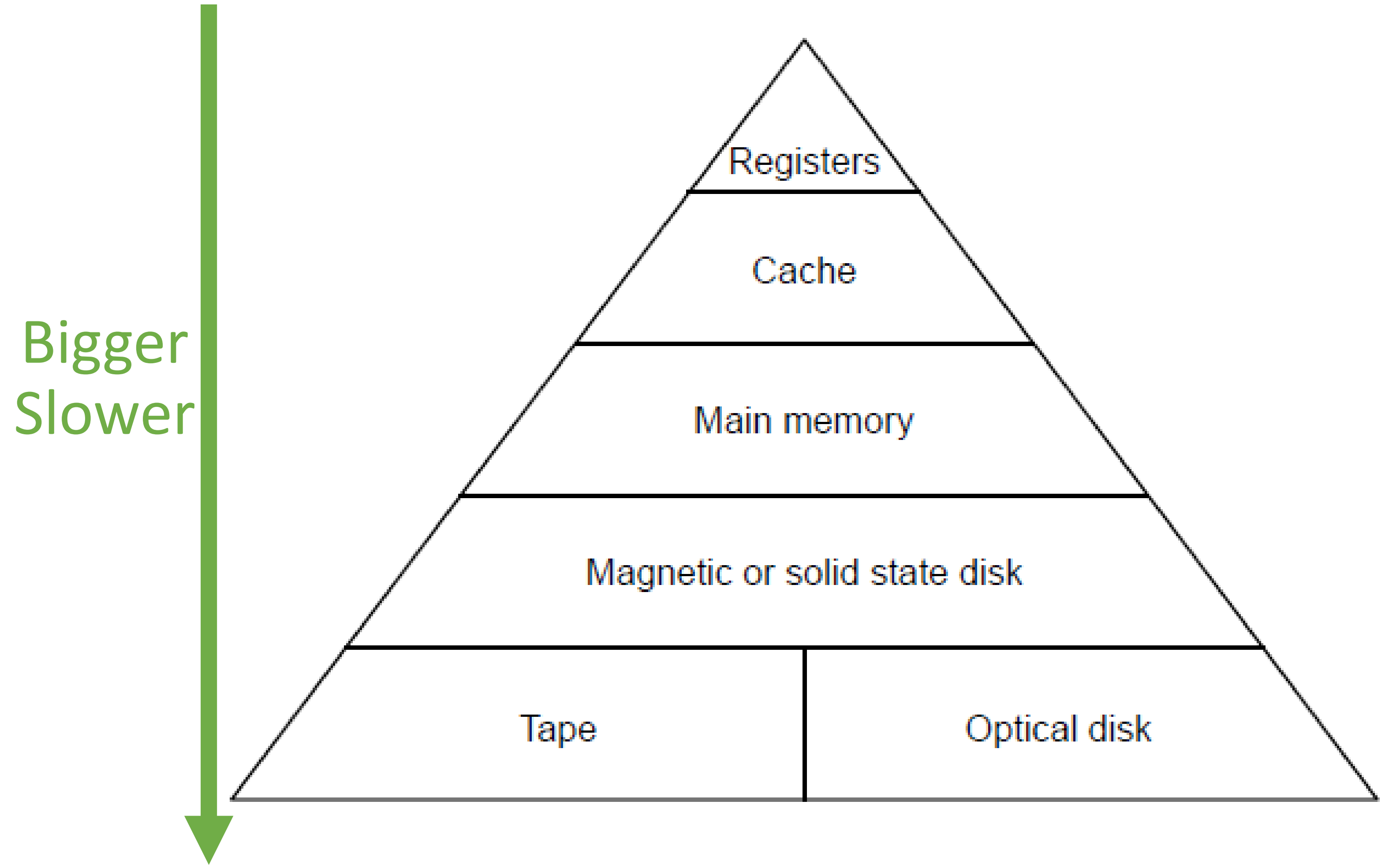
- Automatic procedure call initiated by some exception condition caused by program
- Examples:
  - Overflow (arithmetic, stack)
  - Divide by zero
  - Undefined opcodes
- Trap (or exception or fault) handler: procedure that takes care of exception condition
- If exception detected, PC loaded with exception handler address

Memory Hierarchies

Memory Speed and CPU Execution

- CPUs have always been faster than memories.
- This means that a **load** or **store** instruction, accessing the main memory, is much slower in practice than instructions accessing the ALU.
- How can we deal with this? Suppose we have a **load** instruction:
  - Let the **load** instruction go through the pipeline to fetch the data from memory to a register.
  - If any subsequent instruction tries to use that data before it has arrived, just stall (don't let that instruction proceed to the next pipeline step).

Memory Hierarchy



Levels of the Memory Hierarchy

- Registers: a few tens, accessible at full CPU speed.
  - 1 nanosecond or less.
- Cache, up to a few megabytes.
  - Accessible in a few CPU cycles.
- Main memory: 1GB-16GB for most personal PCs.
  - Access: 10 nanoseconds.
- Solid state drives: 128-512GB.
  - Access: about 100 nanoseconds.
- Magnetic disks (traditional hard drives): 1-4TB.
  - Access: 3-6 milliseconds (millions of nanoseconds).
- Optical disks, tapes: access can take seconds or more.

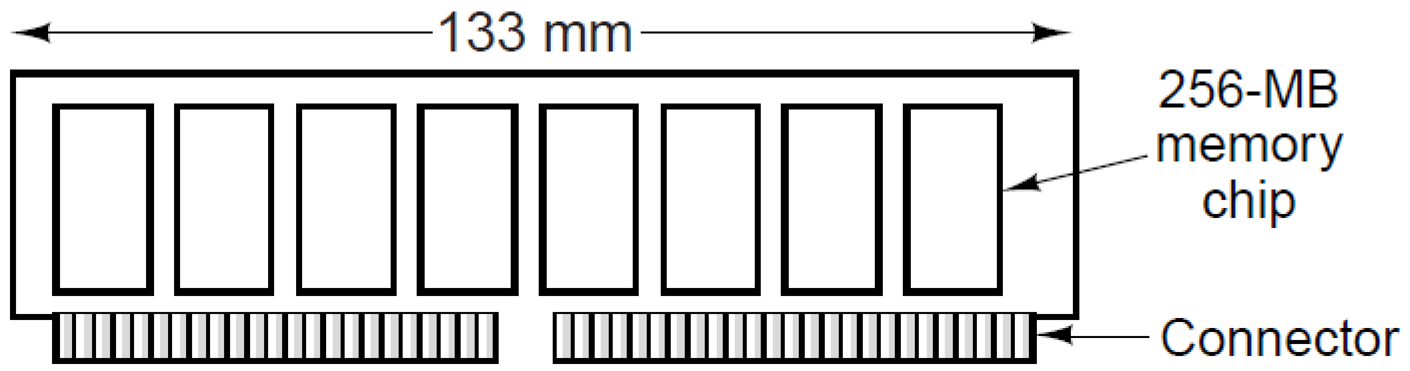
Memory Hierarchies

- We can have superfast expensive memory.
- We can also have slow cheap memory.
- We want lots of superfast cheap memory.
- We can get close to that goal, using a memory hierarchy.
  - At the top, a little bit of superfast expensive memory: CPU registers.
  - Each next level is somewhat slower, and somewhat cheaper.
- Because of the locality principle, the vast majority of memory accesses happen at the lower, faster levels.

Memory Packaging

- Till the early 1990s, memory was installed as single chips.
- Now, typically memory is sold as a group of chips, typically 8 or 16, mounted on the same board.
- This unit may be a SIMM (Single Inline Memory Module).
  - SIMMs can transfer 32 bits per cycle. Rarely used now.
- This unit may also be a DIMM (Dual Inline Memory Module).
  - DIMMS can transfer 64 bits per cycle. Commonly used.
- SO-DIMM (Small Outline DIMM) is used in laptops.
- DIMMs can have a parity bit or error correction.
  - Usually omitted, error rate is one error per 10 years per module.

Memory Packaging and Types



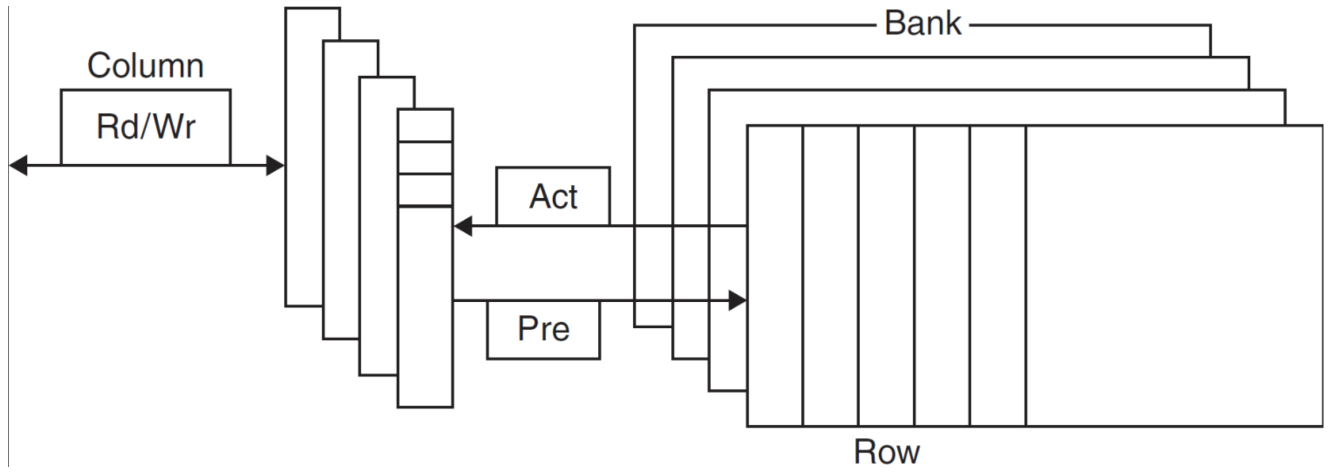
Top view of a DIMM holding 4 GB with eight
chips of 256 MB on each side. The other side looks the same.

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"

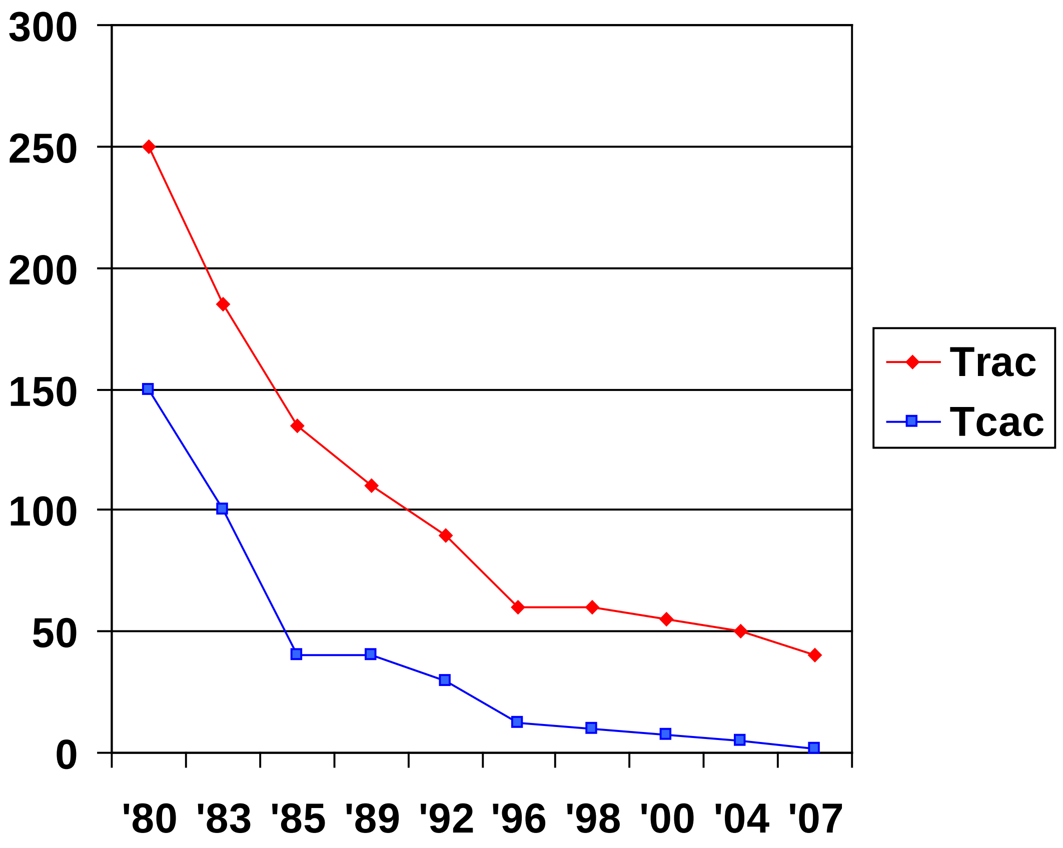


Advanced DRAM Organization

- 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

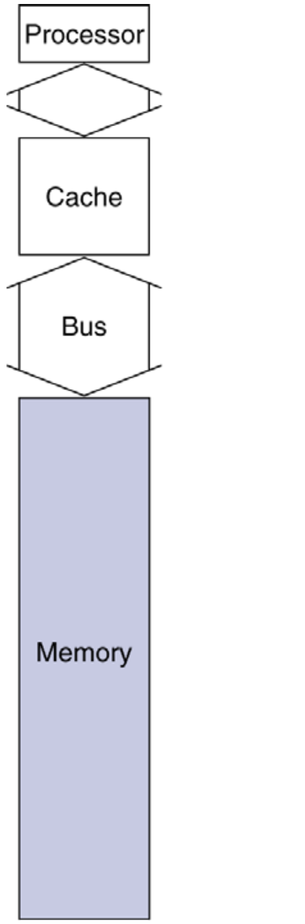
Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



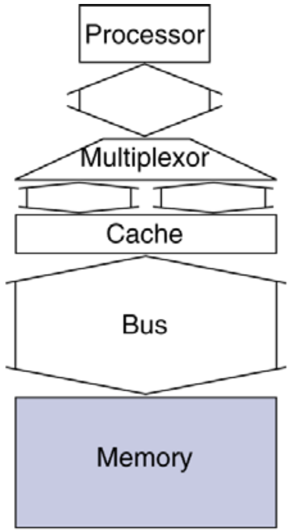
DRAM Performance Factors

- 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

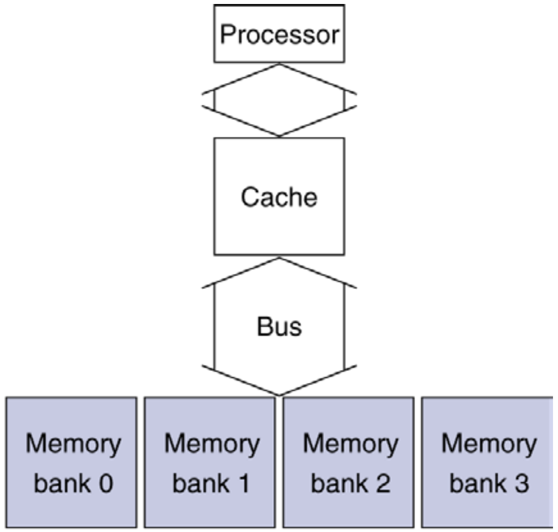
Increasing Memory Bandwidth



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 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 5 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - ?? CPU cycles to execute the 4 instructions not accessing memory.
  - ?? CPU cycles to execute the 1 instruction accessing memory.
- In total, we need ?? CPU cycles per 5 instructions.
  - ??% slower than it would be if memory ran at the same speed as the CPU.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 5 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - 4 CPU cycles to execute the 4 instructions not accessing memory.
  - 5 CPU cycles to execute the 1 instruction accessing memory.
- In total, we need 9 CPU cycles per 5 instructions.
  - 80% slower than it would be if memory ran at the same speed as the CPU.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 50 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - ?? CPU cycles to execute the 4 instructions not accessing memory.
  - ?? CPU cycles to execute the 1 instruction accessing memory.
- In total, we need ?? CPU cycles per 5 instructions.
  - About ?? times slower than it would be if memory ran at the same speed as the CPU.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 50 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - 4 CPU cycles to execute the 4 instructions not accessing memory.
  - 50 CPU cycles to execute the 1 instruction accessing memory.
- In total, we need 54 CPU cycles per 5 instructions.
  - About 11 times slower than it would be if memory ran at the same speed as the CPU.

load and Reordering

- This problem of slow memory access only occurs when subsequent instructions try to use the memory data that **load** is fetching.
  - Unfortunately this is very common.
- Instruction reordering can be used to try to put as many instructions between the **load** instruction and the instruction that tries to use the data fetched by **load**.
  - However, oftentimes that is not very useful. Why?

load and Reordering

- This problem of slow memory access only occurs when subsequent instructions try to use the memory data that **load** is fetching.
  - Unfortunately this is very common.
- Instruction reordering can be used to try to put as many instructions between the **load** instruction and the instruction that tries to use the data fetched by **load**.
  - However, oftentimes that is not very useful. Why?
  - The most common reason why an instruction fetches data from memory is that the next instructions need that data.

load vs. store

- **Store** instructions are not as big a problem as **load** instructions, in terms of hurting performance. Why?

load vs. store

- **Store** instructions are not as big a problem as **load** instructions, in terms of hurting performance. Why?
  - We typically store data back to memory when we do not want to use it anymore.
  - So, subsequent instructions typically do not need to refetch that data from memory.
  - A **store** instruction may need to wait until its data is ready, but that type of wait is much shorter than waiting for **load** to bring data from memory.

Remedy for Slow Memory: The Cache

- Designers of memory systems have to struggle to satisfy two conflicting goals:
  - We want lots of cheap memory.
  - We want memory to be as fast as the CPU.
- To improve performance, it is common to use a hybrid approach:
  - A large amount of slow and cheap memory.
  - A small amount of fast and expensive memory.
- This small, fast memory, is called a cache.

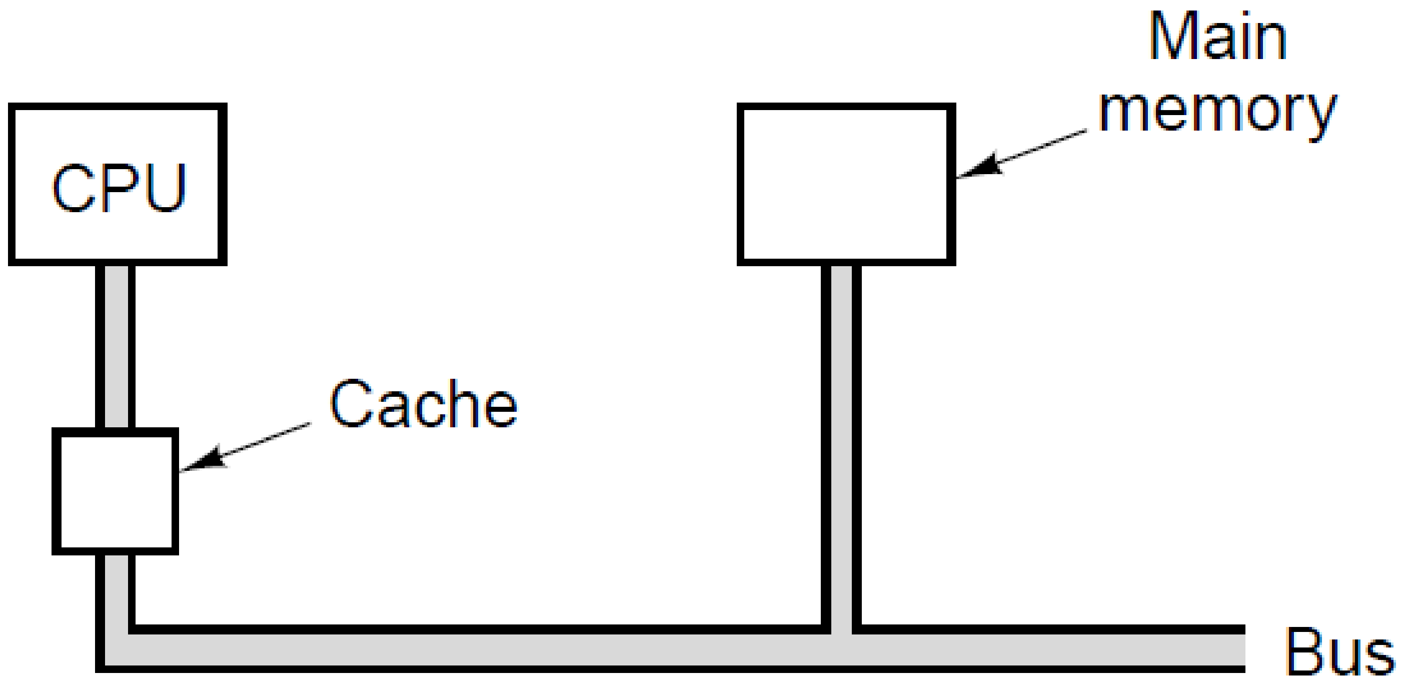
How the Cache Works

- If the CPU needs a memory word, it looks for it in the cache.
- If the word is found in the cache, proceed as normal.
- If the word is not found in the cache, get it from main memory, and store it in the cache.
  - Obviously, every time we store a word in the cache, some other word gets overwritten and is now only available in main memory.
- When will this approach improve performance, when will it hurt performance?

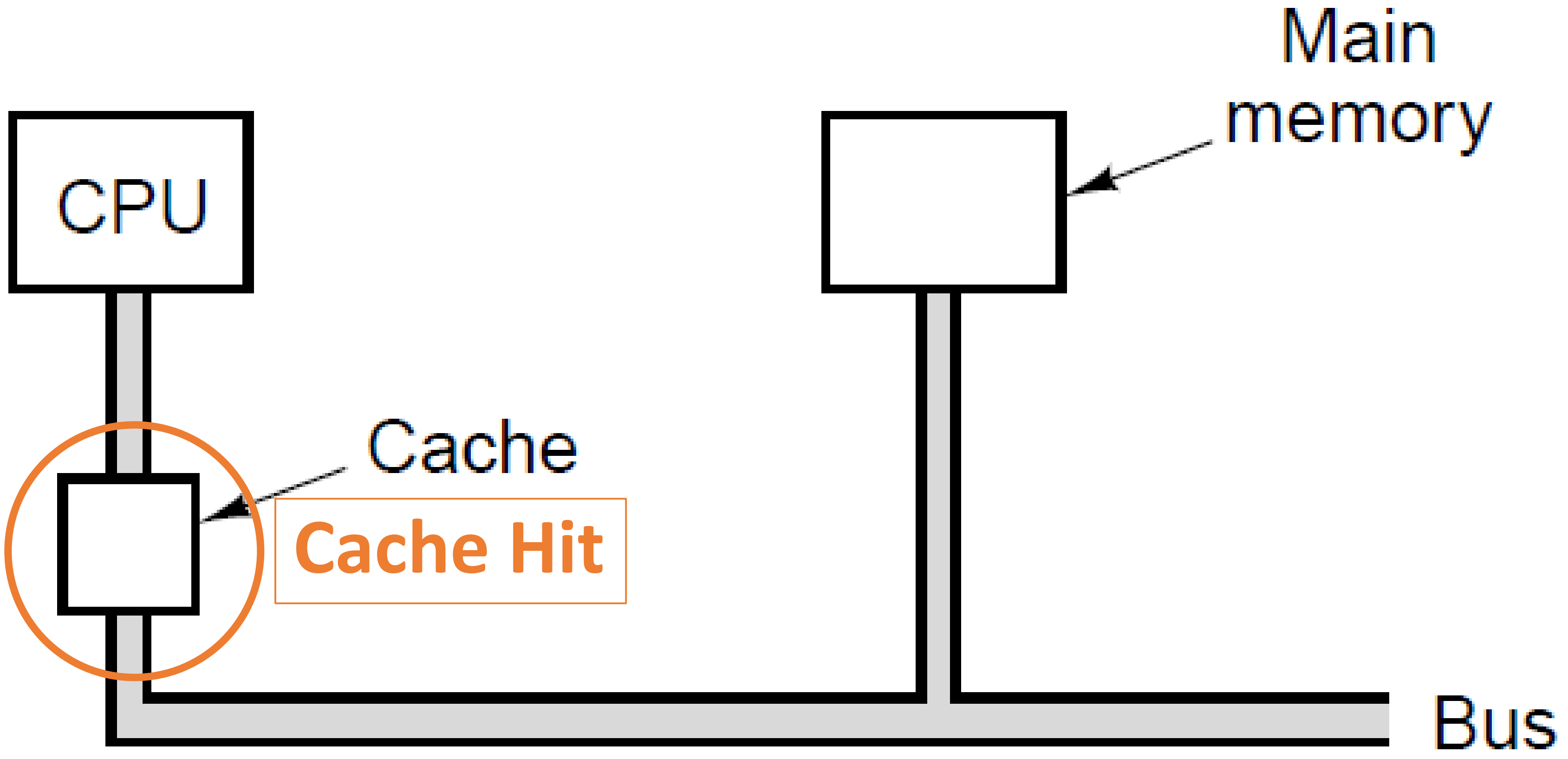
How the Cache Works

- If the CPU needs a memory word, it looks for it in the cache.
- If the word is found in the cache, proceed as normal.
- If the word is not found in the cache, get it from main memory, and store it in the cache.
  - Obviously, every time we store a word in the cache, some other word gets overwritten and is now only available in main memory.
- When will this approach improve performance, when will it hurt performance?
  - It depends on the percentage of times that the word we need is in the cache.

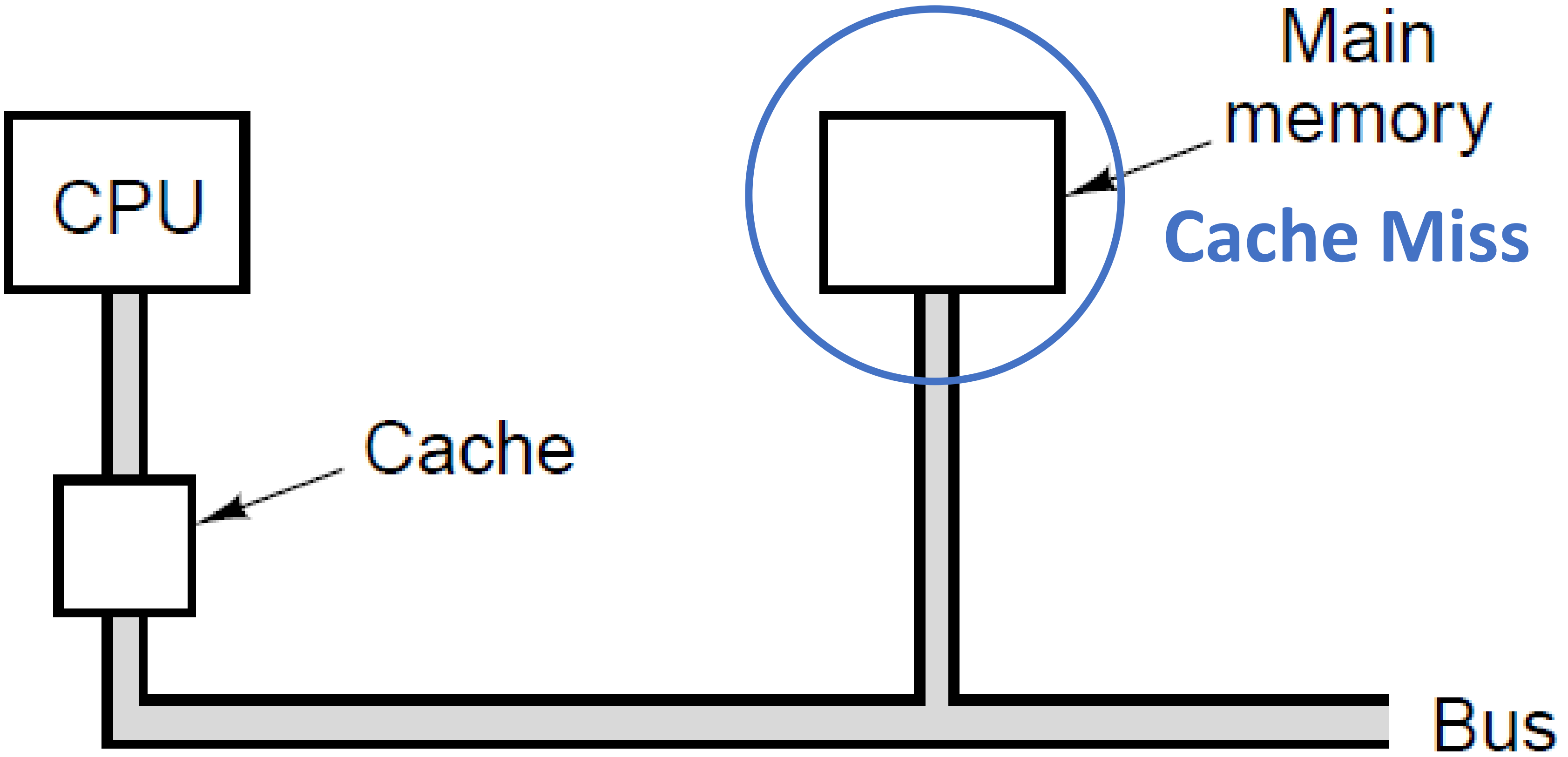
Cache Memory



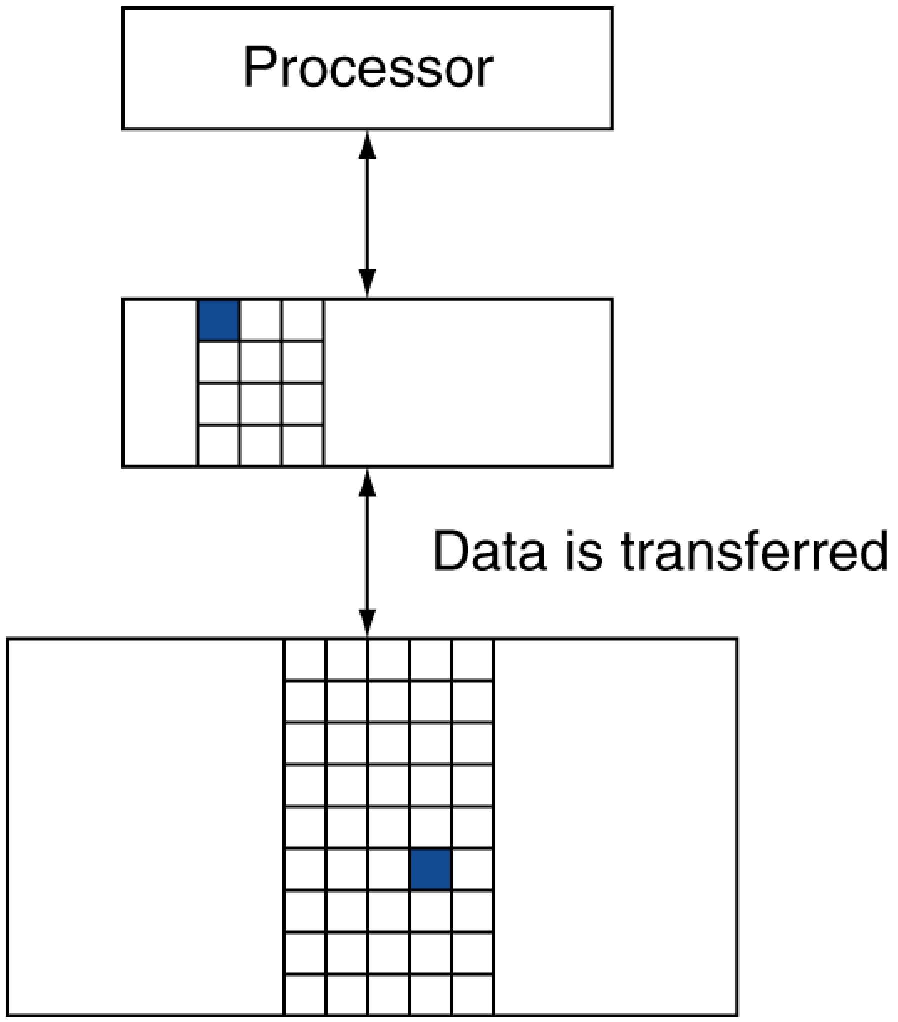
Cache Hit: find necessary data in cache



Cache Miss: have to get necessary data from main memory

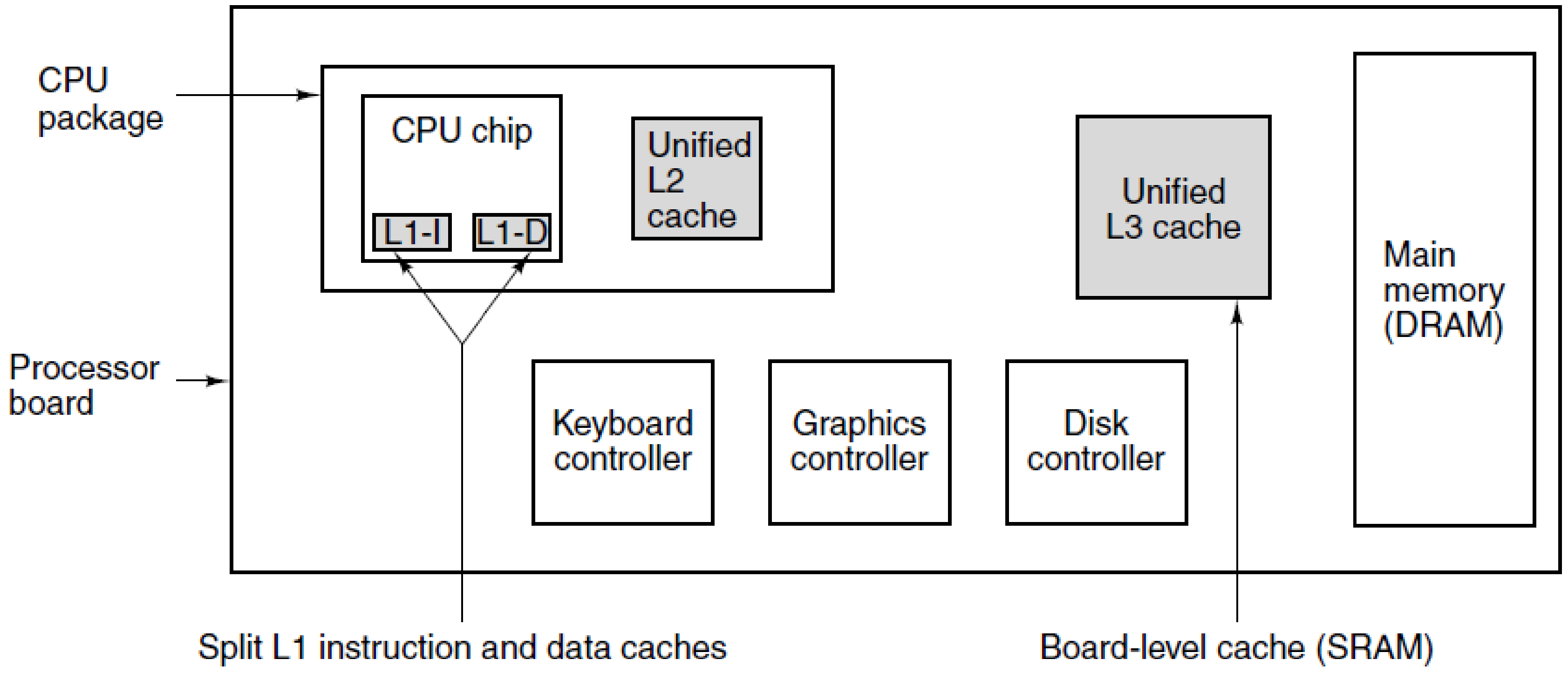


Memory Hierarchy Levels



- 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

More Detailed Cache Organization



Cache Terms

- Cache line: block of cells inside a cache
  - Usually store several words in a line (e.g., store 32 bytes on 32-bit word CPU)
- Cache hit: memory access finds value in cache
  - Antonym: cache miss: have to get it from main memory
- Spatial locality: likely we need data from addresses around one we're requesting (example: array operations)
- Mean access time: C + (1 – H) * M
  - C: cache access time
  - M: main memory access time (usually M >> C, e.g., M > 100 * C)
  - H: hit ratio: probability to find a value in the cache
  - miss ratio: 1 – H
- Time cost of cache miss: c + m memory access time

Cache Design Criteria

- Cache size
  - Bigger cache is more effective, but slower to access and more expensive
- Cache line size
  - Example: 16KB cache divides into
    - 1024 lines of 16 bytes
    - 2048 lines of 8 bytes
    - Etc.
- Cache organization
  - How to keep track of which memory words are in cache?
  - Keep both data and instructions in same cache?

Best and Worst Case

- If almost all words the CPU needs are in the cache, then the average time of accessing memory is close to the time it takes to access the cache.
- If almost all words the CPU needs are NOT in the cache, then the average time of accessing memory is even worse than the time it takes to access main memory
    - Because, before we even access main memory, we need to check the cache.

Quantifying Memory Access Speed

- Let:
  - mean_access_time be the average time it takes for the CPU to access a memory word.
  - C be the average time it takes for the CPU to access a memory word **if that word is currently in the cache**.
  - M be the average time it takes for the CPU to access a word in main memory (i.e., **not in the cache**).
  - H be the **hit ratio**:the fraction of times that the memory word the CPU needs is in the cache.
- mean_access_time = $C + (1 – H) M$
- If H is close to 1:
- If H is close to 0:

Quantifying Memory Access Speed

- Let:
  - mean_access_time be the average time it takes for the CPU to access a memory word.
  - C be the average time it takes for the CPU to access a memory word **if that word is currently in the cache**.
  - M be the average time it takes for the CPU to access a word in main memory (i.e., **not in the cache**).
  - H be the **hit ratio**:the fraction of times that the memory word the CPU needs is in the cache.
- mean_access_time = $C + (1 – H) M$
- If H is close to 1: mean_access_time $\cong$ C.
- If H is close to 0: mean_access_time $\cong C + M$.

Quantifying Memory Access Speed

- mean_access_time = C + (1 – H) M
- If H is close to 1: mean_access_time ≅ C.
  - If the hit ratio is close to 1, then almost all memory accesses are handled by the cache, so the time it takes to access main memory does not affect the average much.
- If H is close to 0: mean_access_time ≅ C + M.
  - If the hit ratio is close to 0, then almost all memory accesses are handled by the main memory. In that case, the CPU:
    - First tries to access the word in the cache, which takes time C.
    - The word is not found in the cache, so the CPU then accesses the word from memory, which takes time M.

The Locality Principle

- In typical programs, memory accesses are not random.
- If we access memory address A, it is likely that the next memory address to be accessed will be close to A.
- More generally, the memory references made in any short time interval tend to use only a small fraction of the total memory.
    - This observation is called the **locality principle**.

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

Using the Locality Principle

- How do we use the locality principle?
- If we need a word and that word is not in the cache:
  - Bring to the cache not only that word, but also several of its neighbors, since they are likely to be accessed next.
- How do we determine which neighbors to load?
  - We divide memories and caches into fixed-sized blocks called **cache lines**.
  - When a cache miss occurs, the entire cache line for that word is loaded into the cache.

Cache Design Optimization

- In designing a cache, several parameters must be determined, oftentimes experimentally.
  - Size of cache: bigger caches lead to better performance, but are more expensive.
  - Size of cache line:
    - 1 word is too small.
    - Setting the cache line to be equal to the cache size is probably too large.
    - Not clear where the optimal value in between is, but simulations can help determine that.

Magnetic Disks

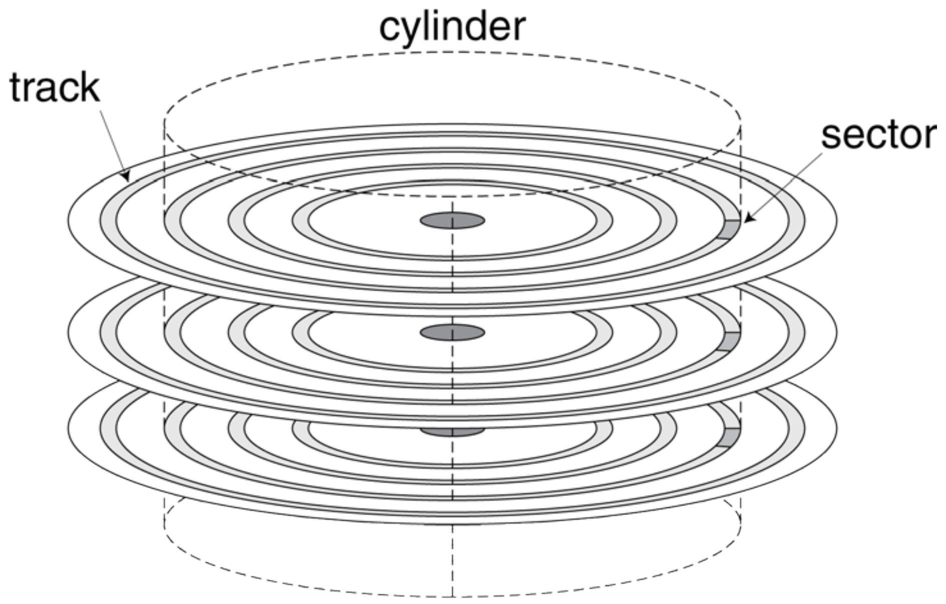
- Consists of one or more platters with magnetizable coating
- Disk head containing induction coil floats just over the surface
- When a positive or negative current passes through head, it magnetizes the surface just beneath the head, aligning the magnetic particles face right or left, depending on the polarity of the drive current
- When head passes over a magnetized area, a positive or negative current is induced in the head, making it possible to read back the previously stored bits
- Track
  - Circular sequence of bits written as disk makes complete rotation
  - Sector: Each track is divided into some sector with fixed length

Classical Hard Drives: Magnetic Disks

- A magnetic disk is a disk, that spins very fast.
  - Typical rotation speed: 5400, 7200, 10800 RPMs.
  - RPMs: rotations per minute.
  - These translate to 90, 120, 180 rotations per second.
- The disk is divided into rings, that are called **tracks**.
- Data is read by the **disk head**.
  - The head is placed at a specific radius from the disk center.
  - That radius corresponds to a specific track.
  - As the disk spins, the head reads data from that track.

Disk Storage

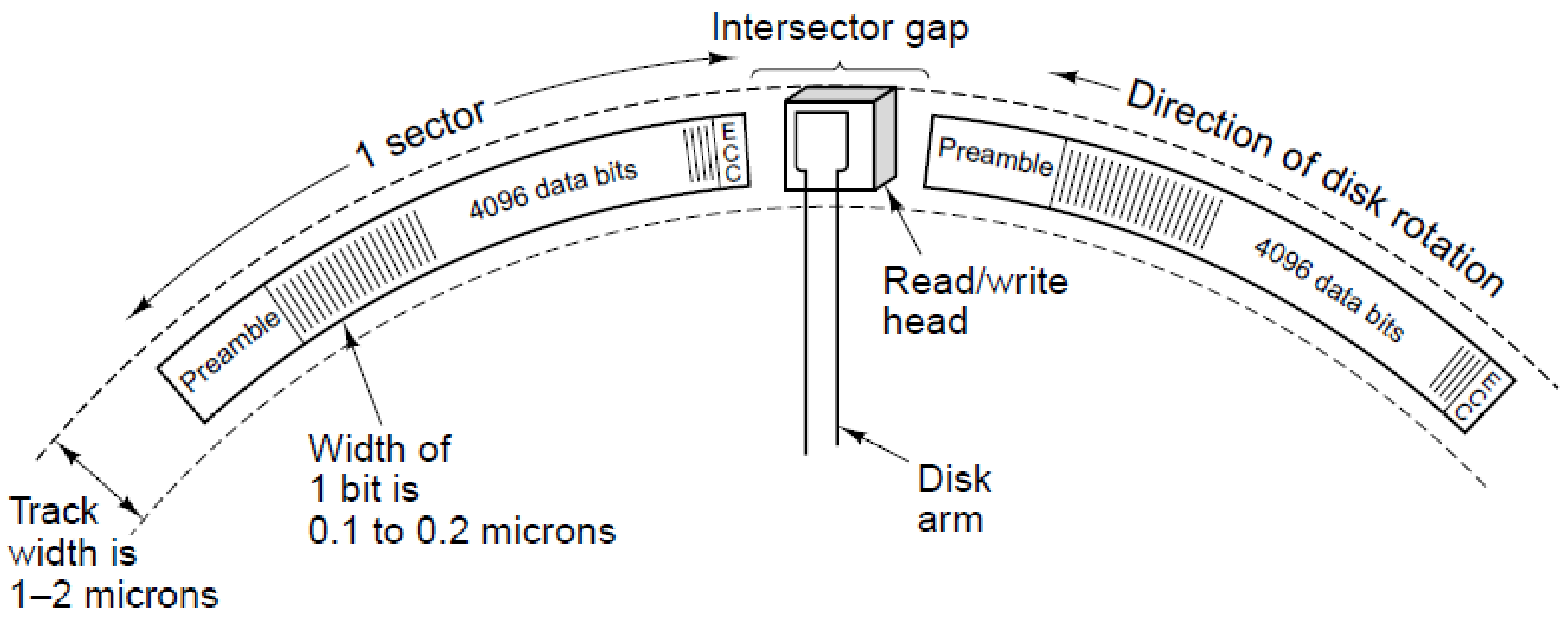
- Nonvolatile, rotating magnetic storage



Disk Tracks and Sectors

- A track can be 0.2μm wide.
  - We can have 50,000 tracks per cm of radius.
  - About 125,000 tracks per inch of radius.
- Each track is divided into fixed-length **sectors**.
  - Typical sector size: 512 bytes.
- Each sector is preceded by a **preamble**. This allows the head to be synchronized before reading or writing.
- In the sector, following the data, there is an error-correcting code.
- Between two sectors there is a small **intersector gap**.

Visualizing a Disk Track



A portion of a disk track. Two sectors are illustrated.

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

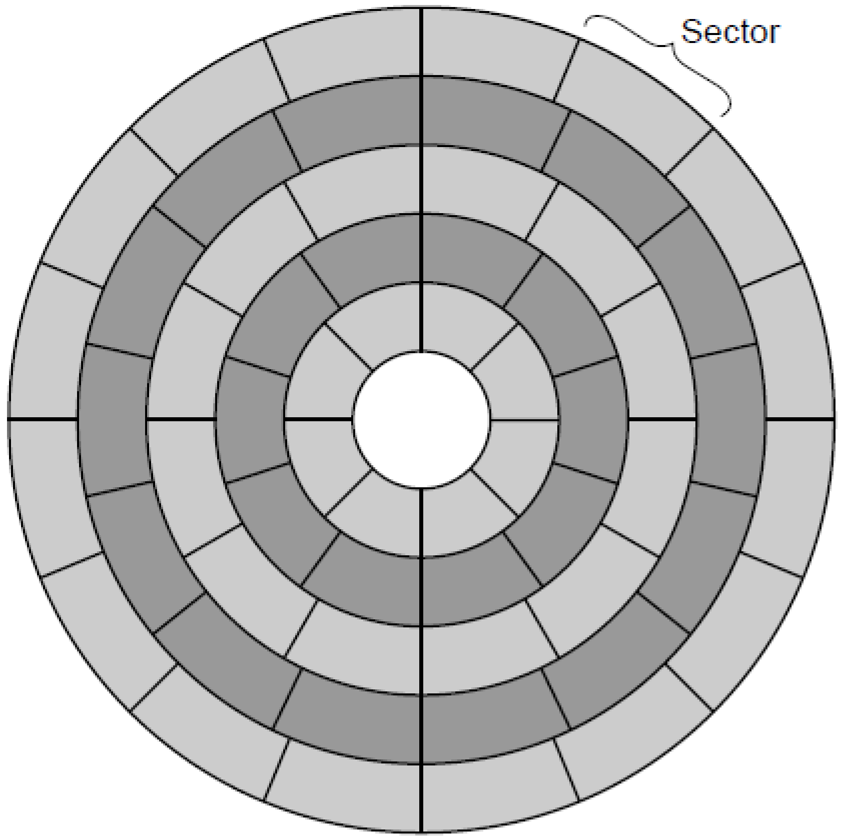
Disk Access Example

- 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

Magnetic Disk Sectors



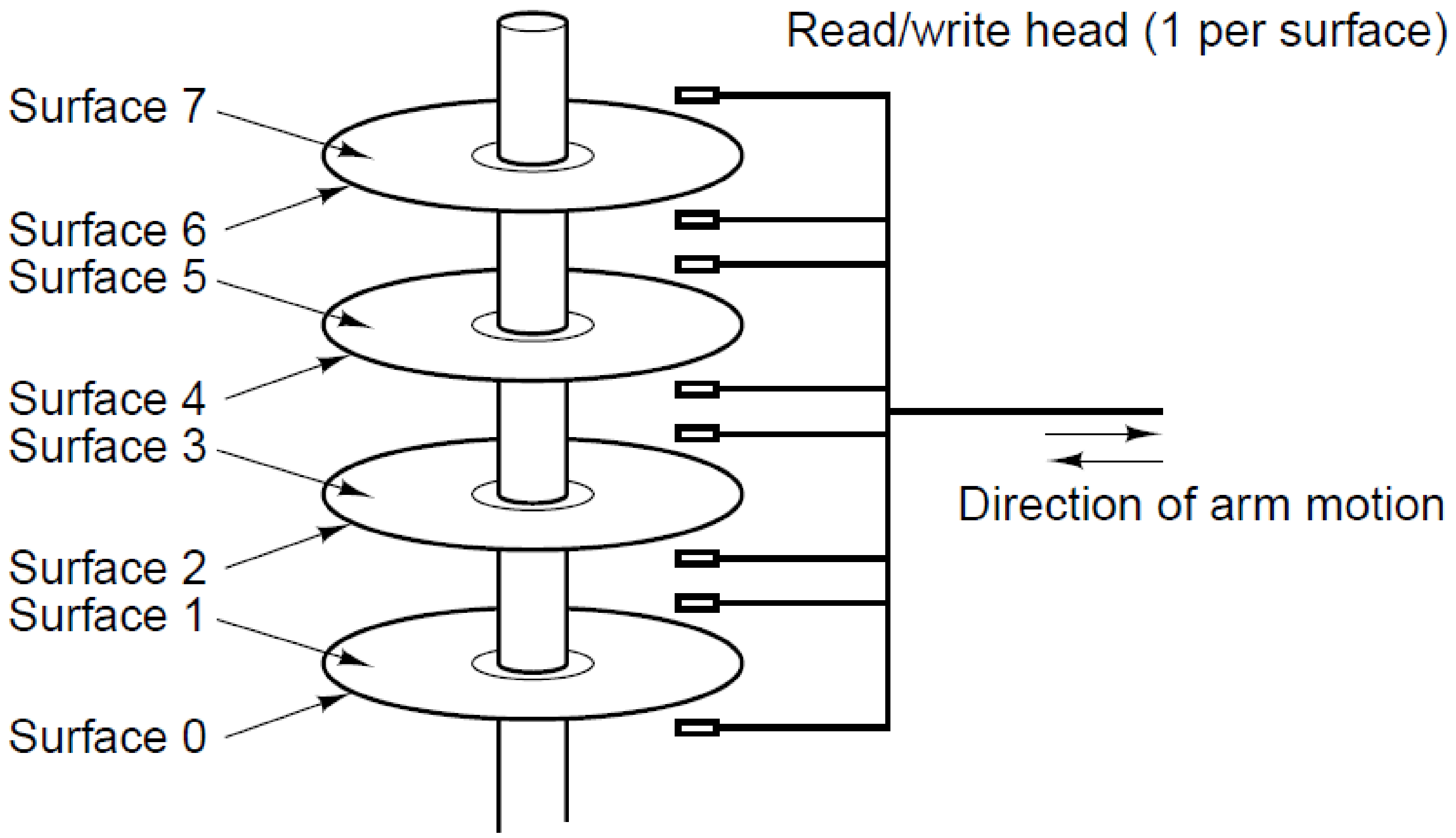
Measuring Disk Capacity

- Disk capacity is often advertized in unformatted state.
- However, **formatting** takes away some of this capacity.
  - Formatting creates preambles, error-correcting codes, and gaps.
- The formatted capacity is typically about 15% lower than unformatted capacity.

Multiple Platters

- A typical hard drive unit contains multiple platters, i.e., multiple actual disks.
- These platters are stacked vertically (see figure).
- Each platter stores information on both surfaces.
- There is a separate arm and head for each surface.

Magnetic Disk Platters



Cylinders

- The set of tracks corresponding to a specific radial position is called a **cylinder**.
- Each track in a cylinder is read by a different head.

Data Access Times

- Suppose we want to get some data from the disk.
- First, the head must be placed on the right track (i.e., at the right radial distance).
  - This is called **seek**.
  - Average seek times are in the 5-10 msec range.
- Then, the head waits for the disk to rotate, so that it gets to the right sector.
  - Given that disks rotate at 5400-10800 RPMs, this incurs an average wait of 3-6 msec. This is called **rotational latency**.
- Then, the data is read. A typical rate for this stage is 150MB/sec.
  - So, a 512-byte sector can be read in ~3.5 μsec.

Measures of Disk Speed

- **Maximum Burst Rate**: the rate (number of bytes per sec) at which the head reads a sector, **once the had has started seeing the first data bit**.
    - This excludes seeks, rotational latencies, going through preambles, error-correcting codes, intersector gaps.
- **Sustained Rate**: the actual average rate of reading data over several seconds, that includes all the above factors (seeks, rotational latencies, etc.).

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?
  - Read random positions, one byte at a time.
  - To read each byte, we must perform a seek, wait for the rotational latency, go through the sector preamble, etc.
- If this whole process takes about 10 msec (which may be a bit optimistic), we can only read ???/sec?

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?
  - Read random positions, one byte at a time.
  - To read each byte, we must perform a seek, wait for the rotational latency, go through the sector preamble, etc.
- If this whole process takes about 10 msec (which may be a bit optimistic), we can only read 100 bytes/sec.
  - More than a million times slower than the maximum burst rate.

Worst Case Speed

- Reading a lot of non-contiguous small chunks of data kills magnetic disk performance.
- When your programs access disks a lot, it is important to understand how disk data are read, to avoid this type of pitfall.

Disk Controller

- The disk controller is a chip that controls the drive.
  - Some controllers contain a full CPU.
- Controller tasks:
  - Execute commands coming from the software, such as:
    - READ
    - WRITE
    - FORMAT (writing all the preambles)
  - Control the arm motion.
  - Detect and correct errors.
  - Buffer multiple sectors.
  - Cache sectors read for potential future use.
  - Remap bad sectors.

IDE and SCSI Drives

- IDE and SCSI drives are the two most common types of hard drives on the market.
- The book goes into a lot of details about each of these types.
- We will skip that in this class.
  - We skip textbook sections 2.3.3 and 2.3.4.
- Just be aware that:
  - IDE drives are cheaper and slower.
    - Newer IDE drives are also called serial ATA or SATA.
  - SCSI drives are more expensive and faster.
- Most inexpensive computers use IDE drives.

RAID

- RAID stands for *Redundant Array of Inexpensive Disks.*
- RAID arrays are simply sets of disks, that are visible as a single unit by the computer.
  - Instead of a single drive accessible via a drive controller, the whole RAID is accessible via a RAID controller.
  - Since a RAID can look as a single drive, software accessing disks does not need to be modified to access a RAID.
- Depending on their type (we will see several types), RAIDs accomplish one (or both) of the following:
  - Speed up performance.
  - Tolerate failures of entire drive units.

Summary

- Memory hierarchy
  - Cache
  - Main memory
  - Disk / storage

RAID for Faster Speed

- Disk performance has not improved as dramatically as CPU performance.
- In the 1970s, average seek times on minicomputer disks were 50-100 msec.
- Now they have improved to 5-10 msec.
- The slow gains in performance have motivated people to look into ways to gain speed via parallel processing.

RAID-0

- RAID level 0: Improves speed via **striping**.
  - When a write request comes in, data is broken into strips.
  - Each strip is written to a different drive, in round-robin fashion.
  - Thus, multiple strips are written in parallel, effectively leading to faster speed, compared to using a single drive.
- Effect: most files are stored in a distributed manner: with different pieces of them stored on each drive of the RAID.
- When reading a file, the different pieces (strips) are read again in parallel, from all drives.

RAID-0 Example

- Suppose we have a RAID-0 system with 8 disks.
- What is the best case scenario, in which performance will be the best, compared to a single disk?

- Compared to a single disk, in the **best** case:
  - The write performance of RAID-0 is: ???
  - The read performance of RAID-0 is: ???
- What is the best case scenario, in which performance will be the best, compared to a single disk?

- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-0 is: ???
  - The read performance of RAID-0 is: ???

RAID-0 Example

UNIVERSITY OF TEXAS ARLINGTON

- Suppose we have a RAID-0 system with 8 disks.
- What is the best case scenario, in which performance will be the best, compared to a single disk?
  - Reading/writing large chunks of data, so striping can be exploited.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-0 is: 8 times faster than a single disk.
  - The read performance of RAID-0 is: 8 times faster than a single disk.
- What is the best case scenario, in which performance will be the best, compared to a single disk?
  - Reading/writing many small, unrelated chunks of data (e.g., a single byte at a time). Then, striping cannot be used.
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-0 is: the same as that of a single disk.
  - The read performance of RAID-0 is: the same as that of a single disk.

RAID-0: Pros and Cons

- RAID-0 works the best for large read/write requests.
- RAID-0 speed deteriorates into that of a single drive if the software asks for data in chunks of one strip (or less) at a time.
- How about reliability? A RAID-0 is **less** reliable, and more prone to failure than that of a single drive.
  - Suppose we have a RAID with four drives.
  - Each drive has a mean time to failure of 20,000 hours.
  - Then, the RAID has a mean time to failure that is ??? hours?

RAID-0: Pros and Cons

- RAID-0 works the best for large read/write requests.
- RAID-0 speed deteriorates into that of a single drive if the software asks for data in chunks of one strip (or less) at a time.
- How about reliability? A RAID-0 is **less** reliable, and more prone to failure than that of a single drive.
  - Suppose we have a RAID with four drives.
  - Each drive has a mean time to failure of 20,000 hours.
  - Then, the RAID has a mean time to failure that is only 5000 hours.
- RAID-0 is not a "true" RAID, no drive is redundant.

RAID-1

- In RAID-1, we need to have an even number of drives.
- For each drive, there is an identical copy.
- When we write data, we write it to both drives.
- When we read data, we read from either of the drives.
- NO STRIPING IS USED.
- Compared to a single disk:
  - The write performance is:
  - The read performance is:
  - Reliability is:

RAID-1

- In RAID-1, we need to have an even number of drives.
- For each drive, there is an identical copy.
- When we write data, we write it to both drives.
- When we read data, we read from either of the drives.
- NO STRIPING IS USED.
- Compared to a single disk:
  - The write performance is: twice as slow.
  - The read performance is: the same.
  - Reliability is: far better, drive failure is not catastrophic.

The Need for RAID-5.

- RAID-0: great for performance, bad for reliability.
  - striping, but no redundant data.
- RAID-1: bad for performance, great for reliability.
  - redundant data, no striping
- RAID-2, RAID-3, RAID-4: have problems of their own.
  - You can read about them in the textbook if you are curious, but they are not very popular.
- RAID-5: great for performance, great for reliability.
  - both redundant data and striping.

RAID-5

- Data is striped for writing.
- If we have N disks, we can process N-1 data strips in parallel.
- For every N-1 data strips, we create an Nth strip, called **parity strip**.
  - The k-th bit in the parity strip ensures that there is an even number of 1-bits in position k in all N strips.
- If any strip fails, its data can be recovered from the other N-1 strips.
- This way, the contents of an entire disk can be recovered.

RAID-5 Example

UNIVERSITY OF TEXAS ARLINGTON

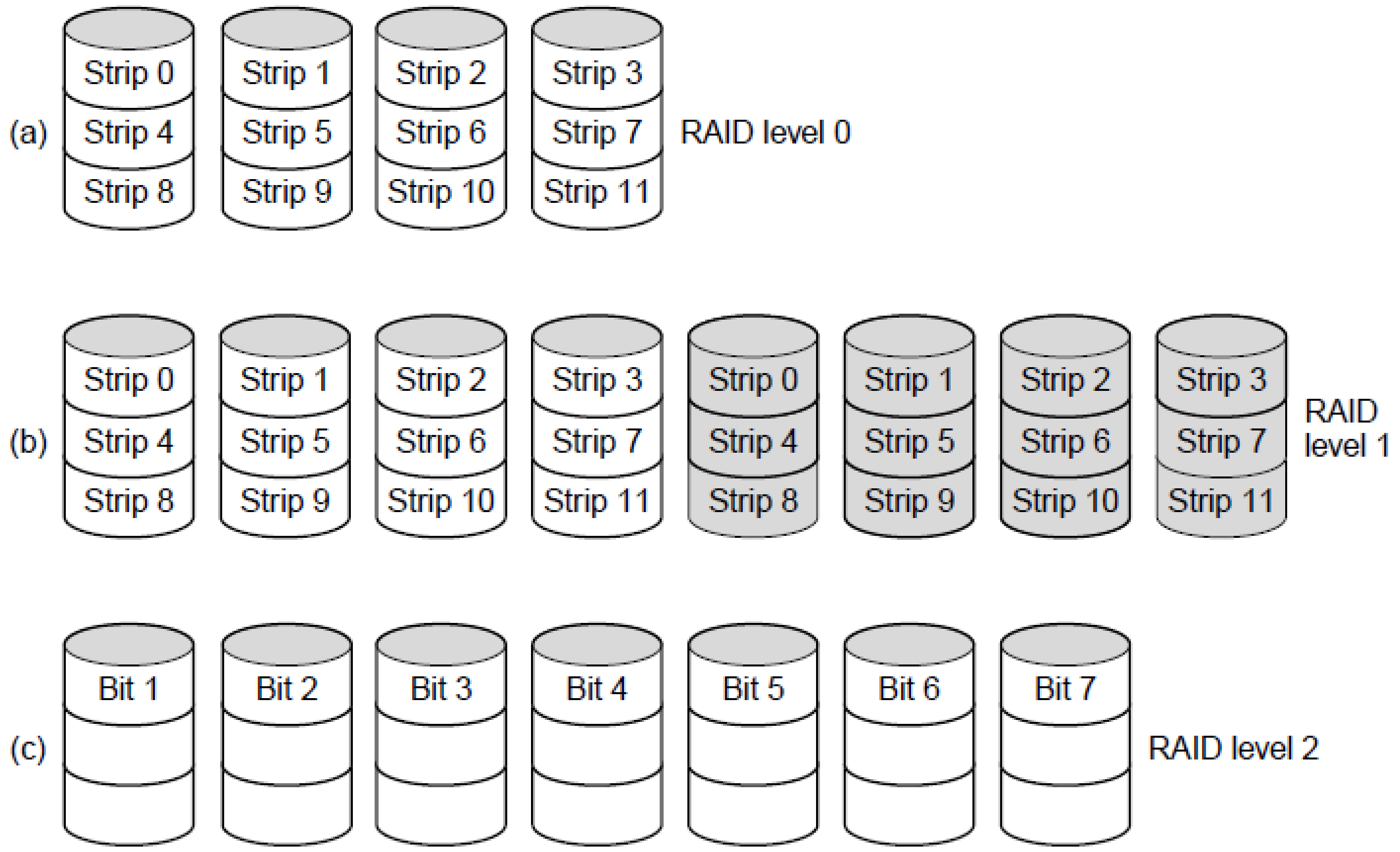
- Suppose we have a RAID-5 system with 8 disks.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-5 is: ???
  - The read performance of RAID-5 is: ???
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-5 is: ???
  - The read performance of RAID-5 is: ???

RAID-5 Example

UNIVERSITY OF TEXAS ARLINGTON

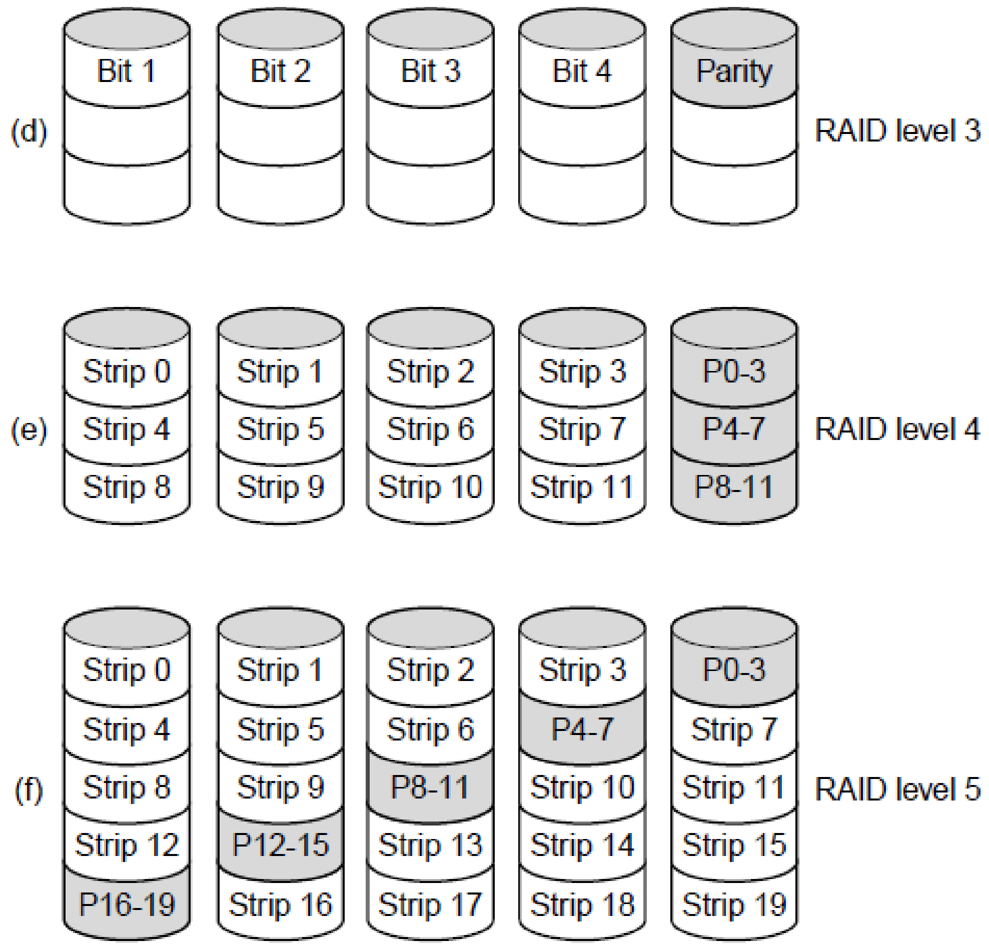
- Suppose we have a RAID-5 system with 8 disks.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-5 is: 7 times faster than a single disk. (writes non-parity data on 7 disks simultaneously).
  - The read performance of RAID-5 is: 7 times faster than a single disk. (reads non-parity data on 7 disks simultaneously).
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-5 is: the same as that of a single disk.
  - The read performance of RAID-5 is: the same as that of a single disk.
  - Why? Because striping is not useful when reading/writing one byte at a time.

RAID-0, RAID-1, RAID-2



RAID levels 0 through 5. Backup and parity drives are shown shaded.

RAID-3, RAID-4, RAID-5

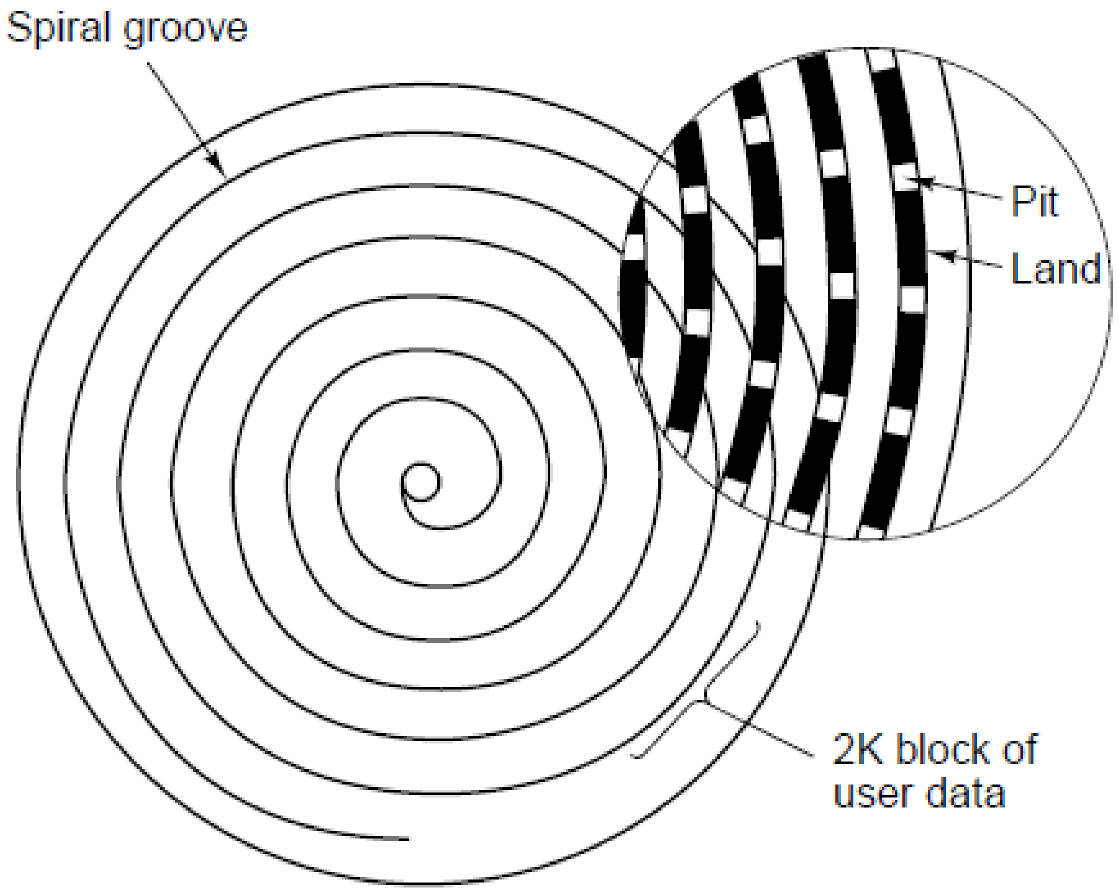


RAID levels 0 through 5. Backup and parity drives are shown shaded.

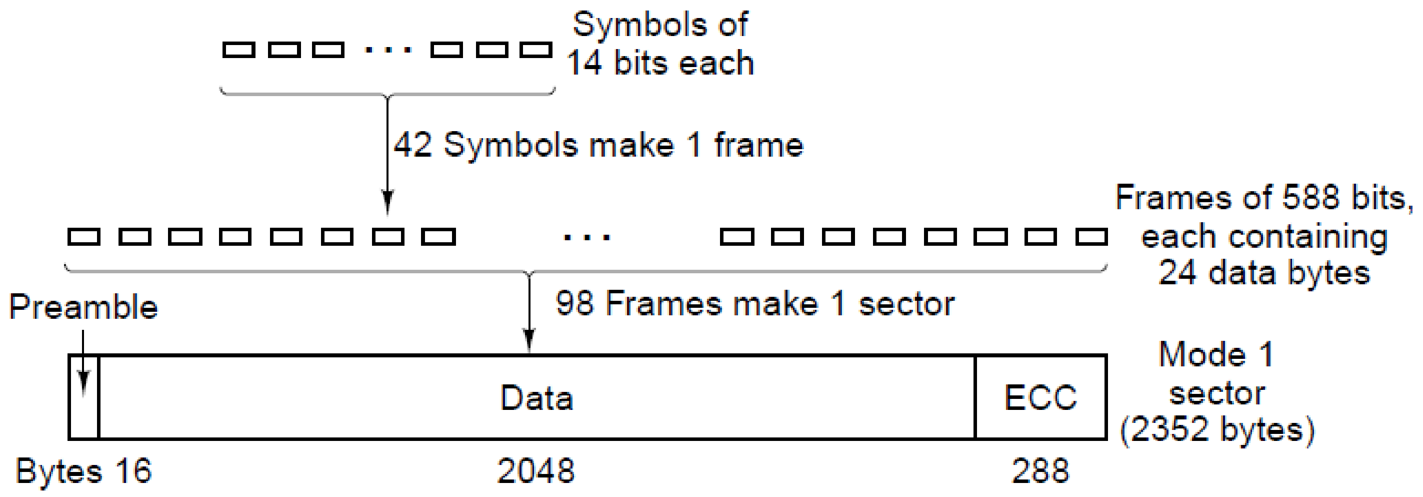
Solid-State Drives

- A solid-state drive (SSD) is NOT a spinning disk. It is just cheap memory.
- Compared to hard drives, SSDs have two to three times faster speeds, and **<u>~100nsec access time.</u>**
- Because SSDs have no mechanical parts, they are well-suited for mobile computers, where motion can interfere with the disk head accessing data.
- Disadvantage #1: price.
  - Magnetic disks: pennies/gigabyte.
  - SSDs: one to three dollars/gigabyte.
- Disadvantage #2: failure rate.
  - A bit can be written about 100,000 times, then it fails.

CDs

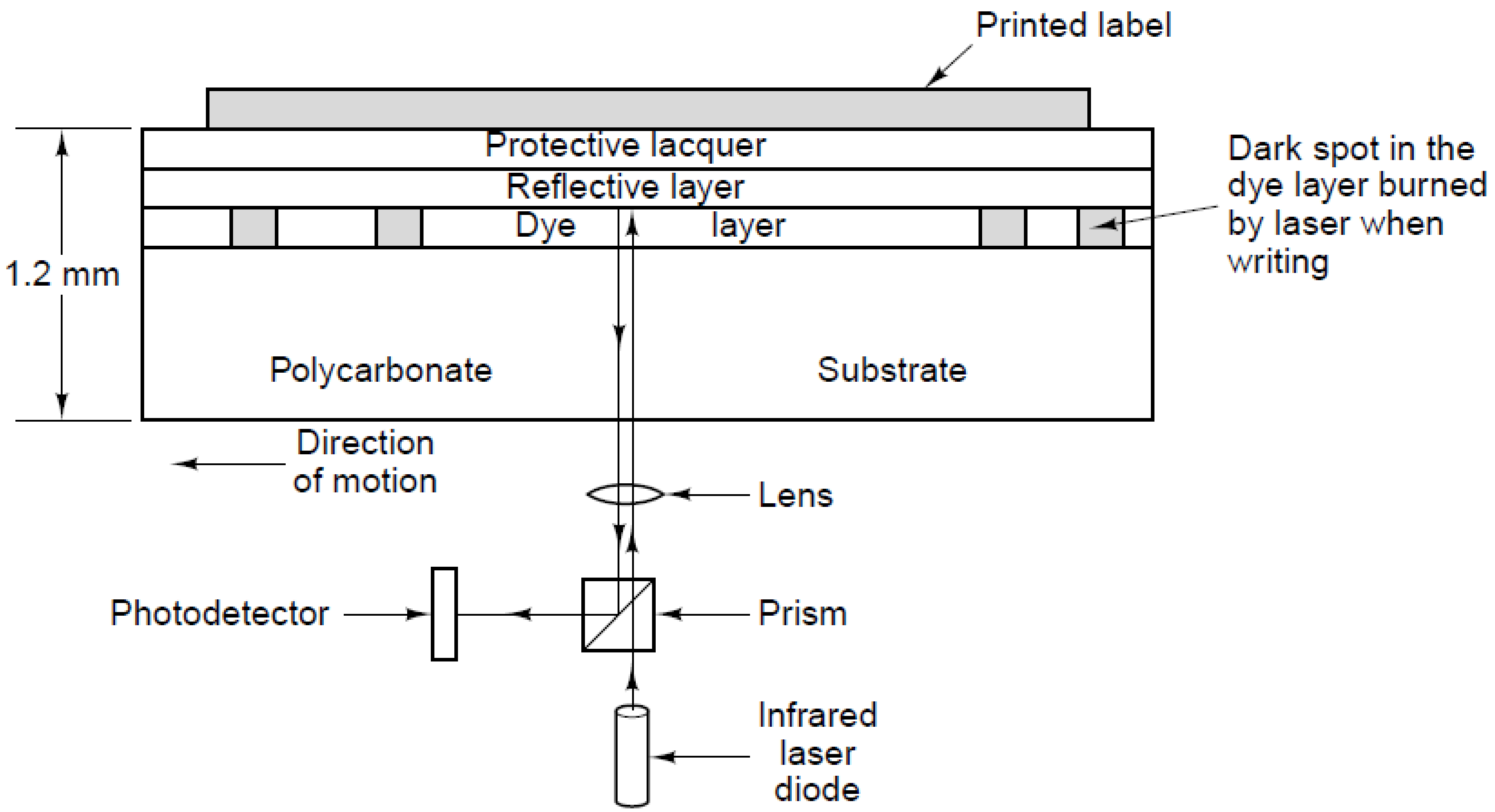


CDs



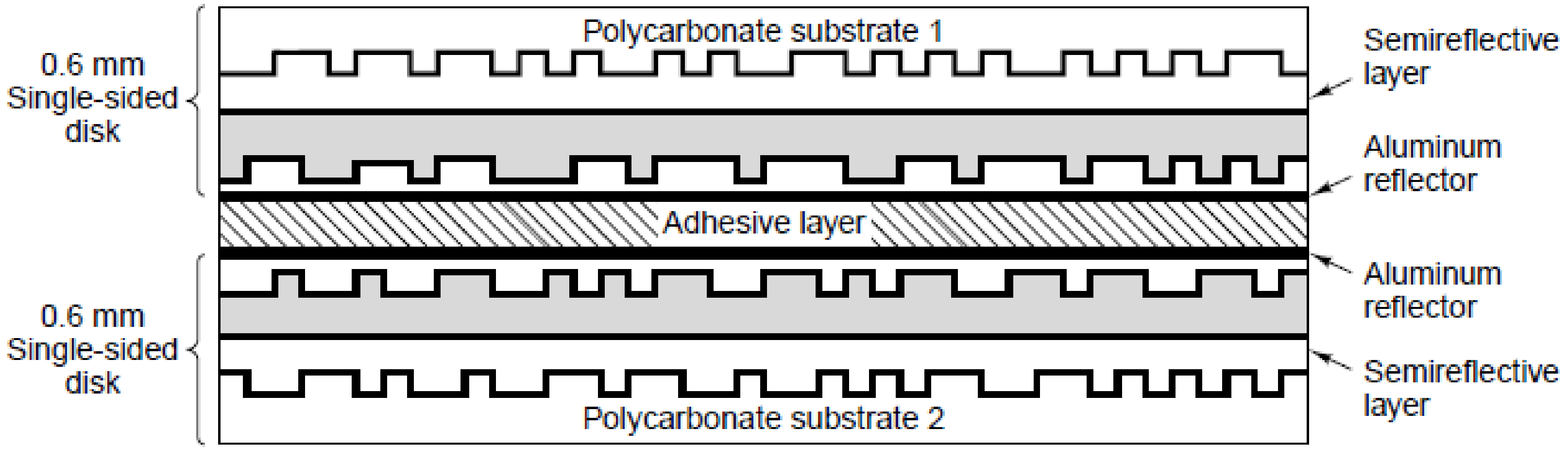
- Mode 1
  - 16 bytes preamble, 2048 bytes data, 288 bytes error-correcting code
  - Single Speed CD-ROM: 75 sectors/sec, so data rate: 75*2048=153,600 bytes/sec
  - 74 minutes audio CD: Capacity: 74*60*153,600=681,984,000 bytes ~=650 MB
- Mode 2
  - 2336 bytes data for a sector, 75*2336=175,200 bytes/sec

CD-R

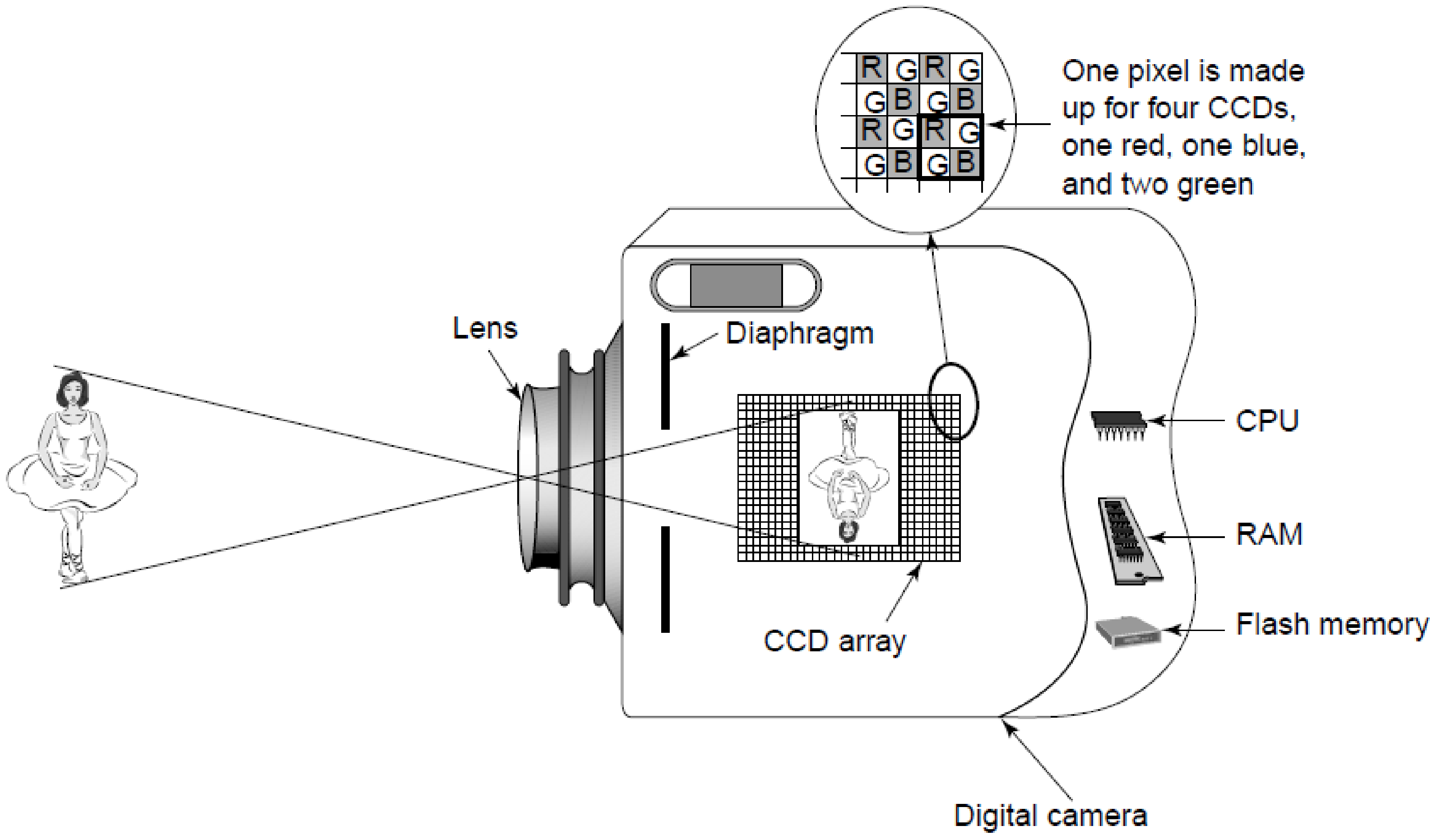

Printed label
Protective lacquer
Reflective layer
Dye layer
Dark spot in the dye layer burned by laser when writing
1.2 mm
Polycarbonate
Substrate
Direction of motion
Lens
Photodetector
Prism
Infrared laser diode

DVDs

- Single-sided, single-layer (4.7 GB)
- Single-sided, dual-layer (8.5 GB)
- Double-sided, single-layer (9.4 GB)
- Double-sided, dual-layer (17 GB)



Storing Images



Optical Disks

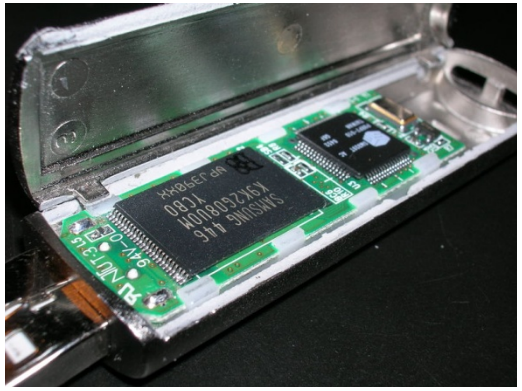
- Disks in this family include:
  - CDs, DVDs, Blu-ray disks.
- The basic technology is similar, but improvements have led to higher capacities and speeds.
- Optical disks are much slower than magnetic drives.
- These disks are a cheap option for write-once purposes.
  - Great for mass distribution of data (software, music, movies).
- CD capacity: 650-700MB.
  - Minimum data rate: 150KB/sec.
- DVD capacity: 4.7GB to 17GB.
  - Minimum data rate: 1.4MB/sec.
- Blu-ray capacity: 25GB-50GB.
  - Minimum data rate: 4.5MB/sec.

Optical Disk Capacities

- CD capacity: 650-700MB.
    - Minimum data rate: 150KB/sec.
- DVD capacity: 4.7GB to 17GB.
    - Minimum data rate: 1.4MB/sec.
    - Single-sided, single-layer: 4.7GB.
    - Single-sided, dual-layer: 8.5GB.
    - Double-sided, single-layer: 9.4GB.
    - Double-sided, dual-layer: 17GB.
- Blu-ray capacity: 25GB-50GB.
    - Minimum data rate: 4.5MB/sec.
    - Single-sided: 25GB.
    - Double-sided: 50GB.

Flash Storage

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

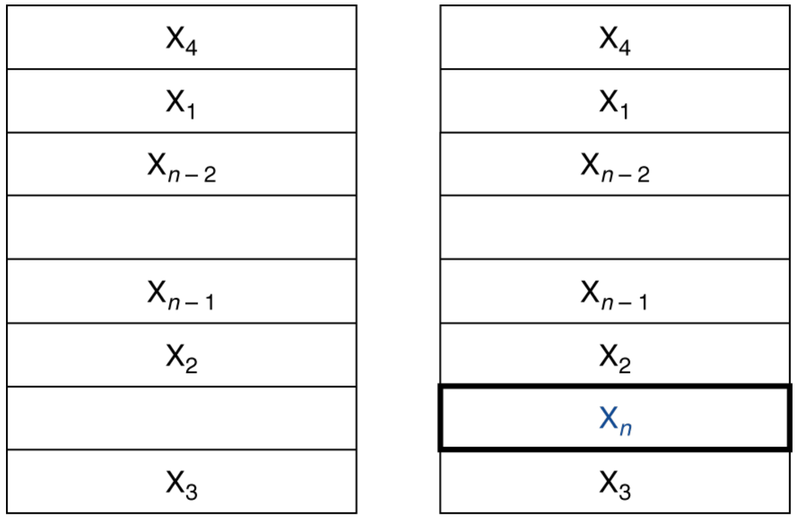


Flash Types

- 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

Cache Memory

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

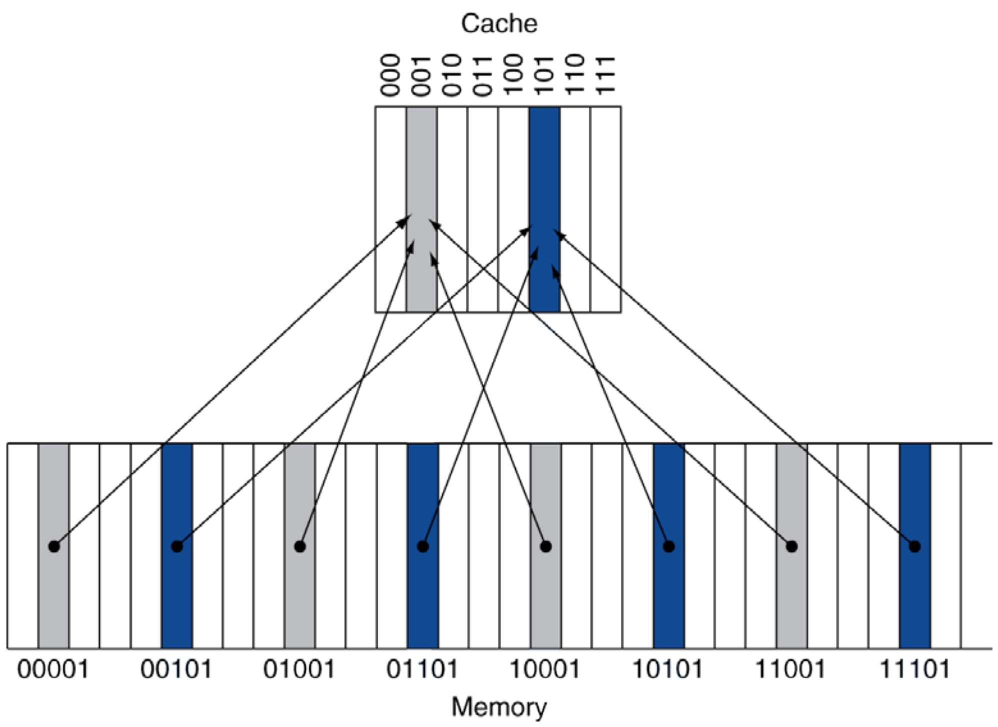


a. Before the reference to X_n b. After the reference to X_n

- 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)



- #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

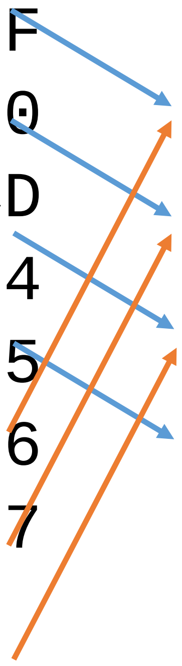
Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

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		

Direct-Mapped Caches

MEMORY

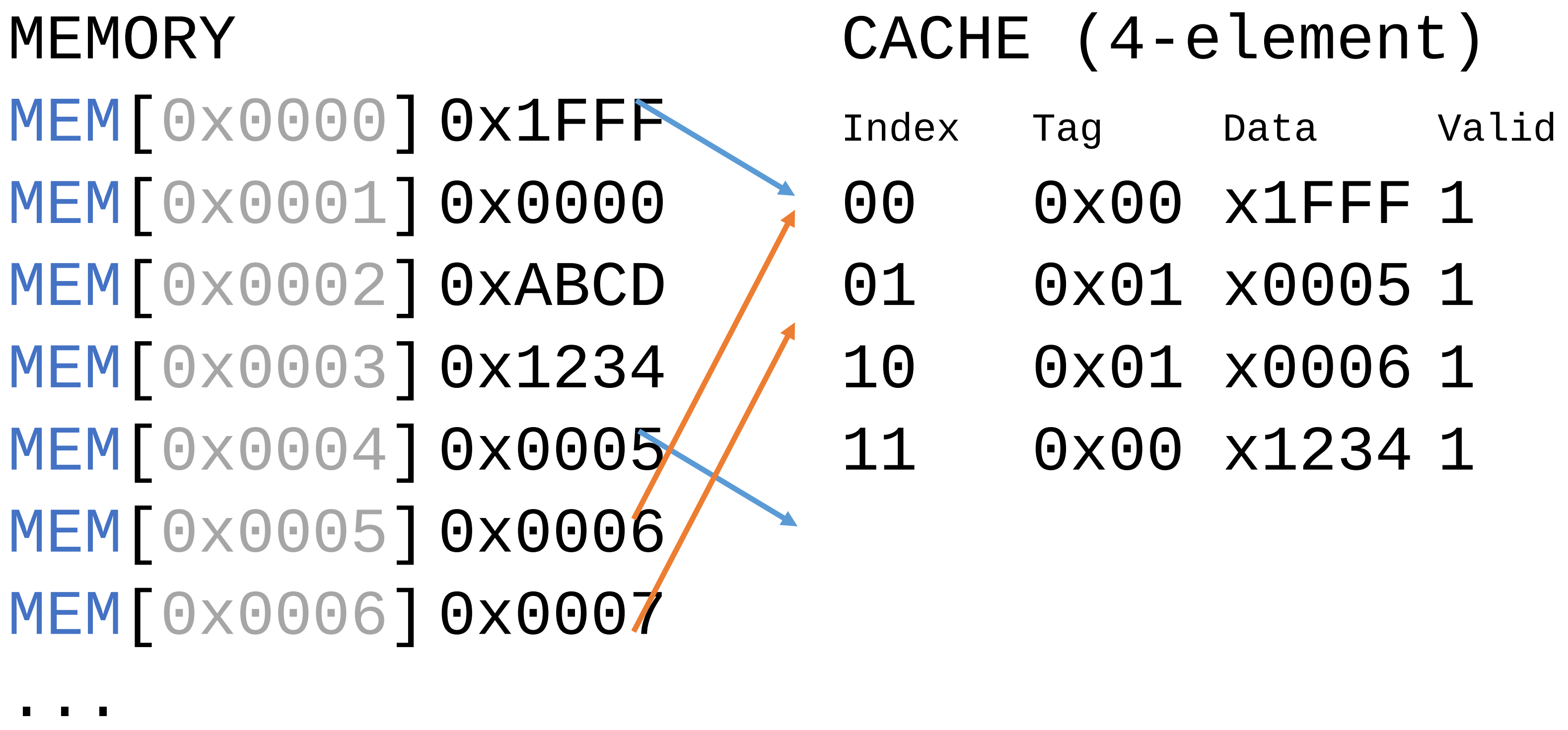
```
MEM[0x0000] 0x1FFF
MEM[0x0001] 0x0000
MEM[0x0002] 0xABCD
MEM[0x0003] 0x1234
MEM[0x0004] 0x0005
MEM[0x0005] 0x0006
MEM[0x0006] 0x0007
...
```



CACHE (4-element)

Index	Tag	Data	Valid
00	0x00	x1FFF	1
01	0x00	x0000	1
10	0x00	xABCD	1
11	0x00	x1234	1

Direct-Mapped Caches



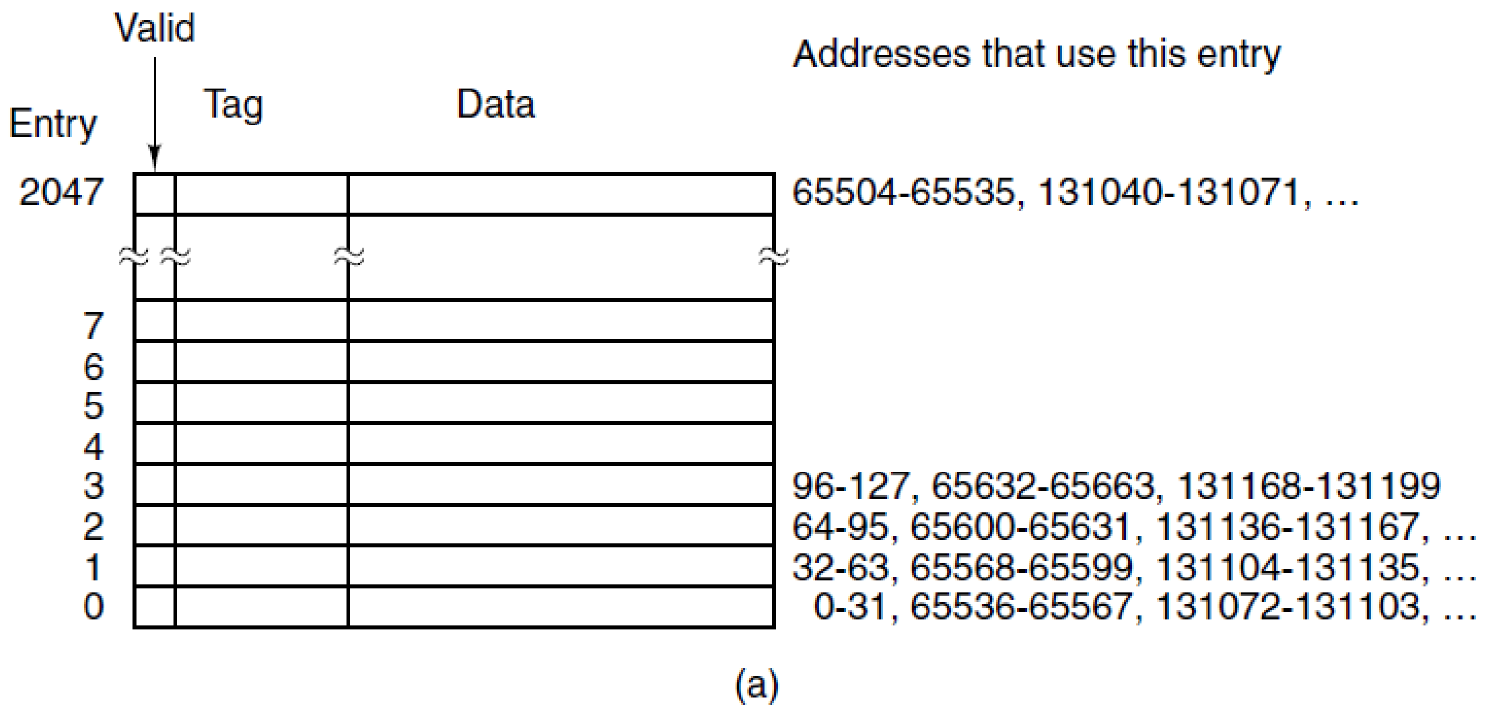
MEMORY

```
MEM[0x0000] 0x1FFF
MEM[0x0001] 0x0000
MEM[0x0002] 0xABCD
MEM[0x0003] 0x1234
MEM[0x0004] 0x0005
MEM[0x0005] 0x0006
MEM[0x0006] 0x0007
...
```

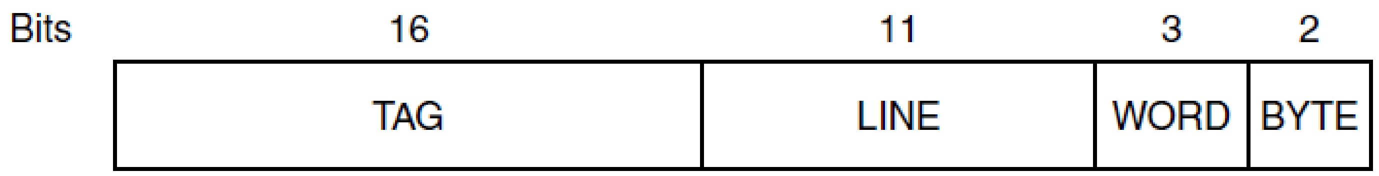
CACHE (4-element)

Index	Tag	Data	Valid
00	0x00	x1FFF	1
01	0x01	x0005	1
10	0x01	x0006	1
11	0x00	x1234	1

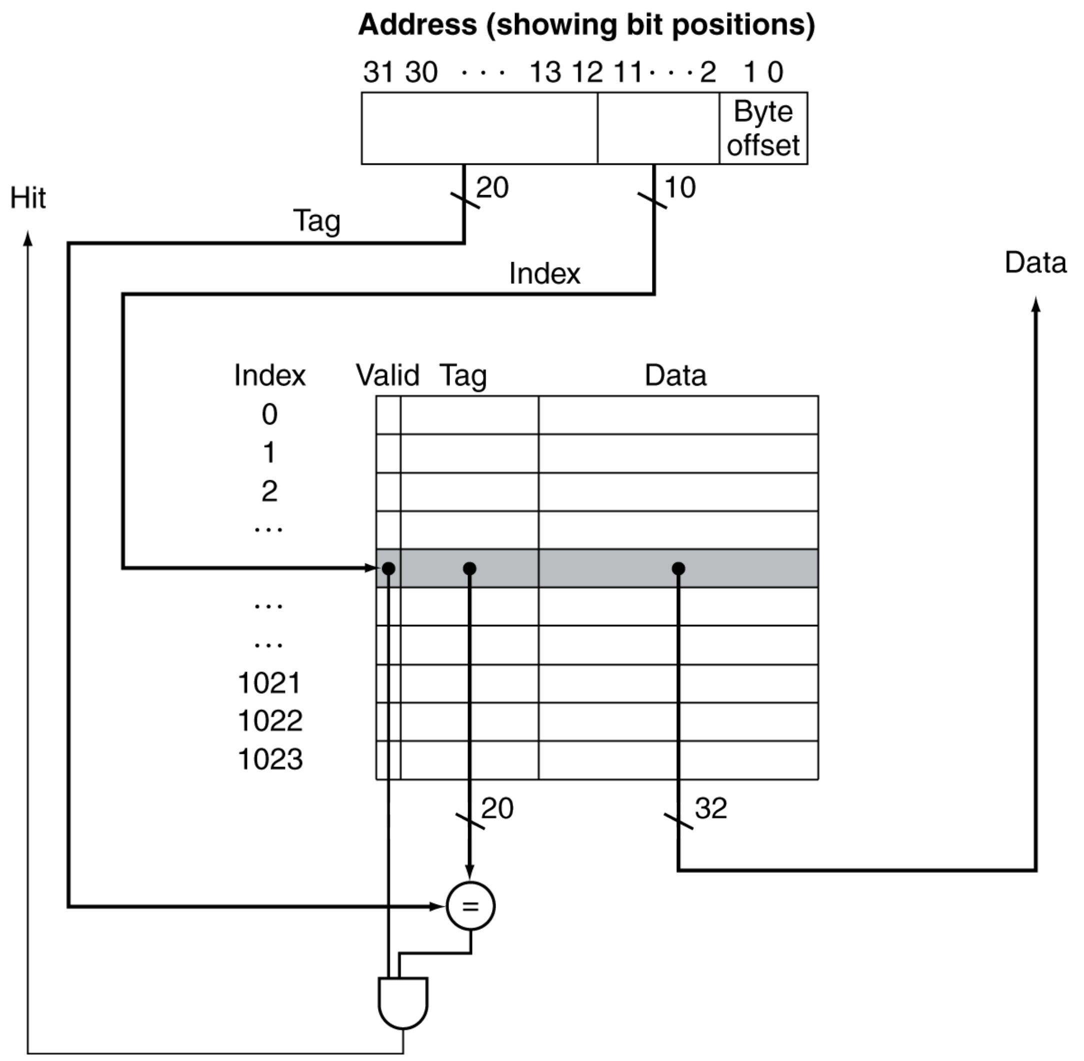
Direct-Mapped Caches



(a)

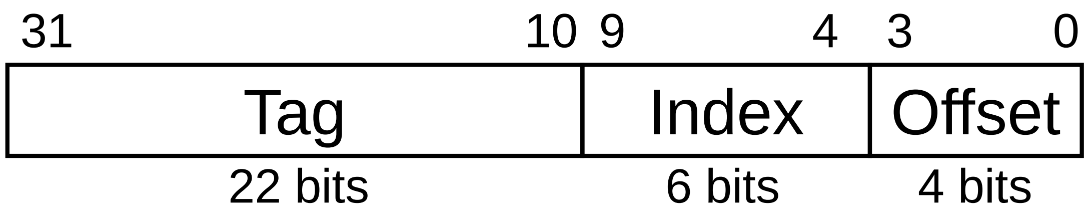


Address Subdivision



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



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 + 0.1×100 = 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