Computer Architecture Review

Computer Abstractions and Technology

Jo, Heeseung

Below Your Program

Hardware

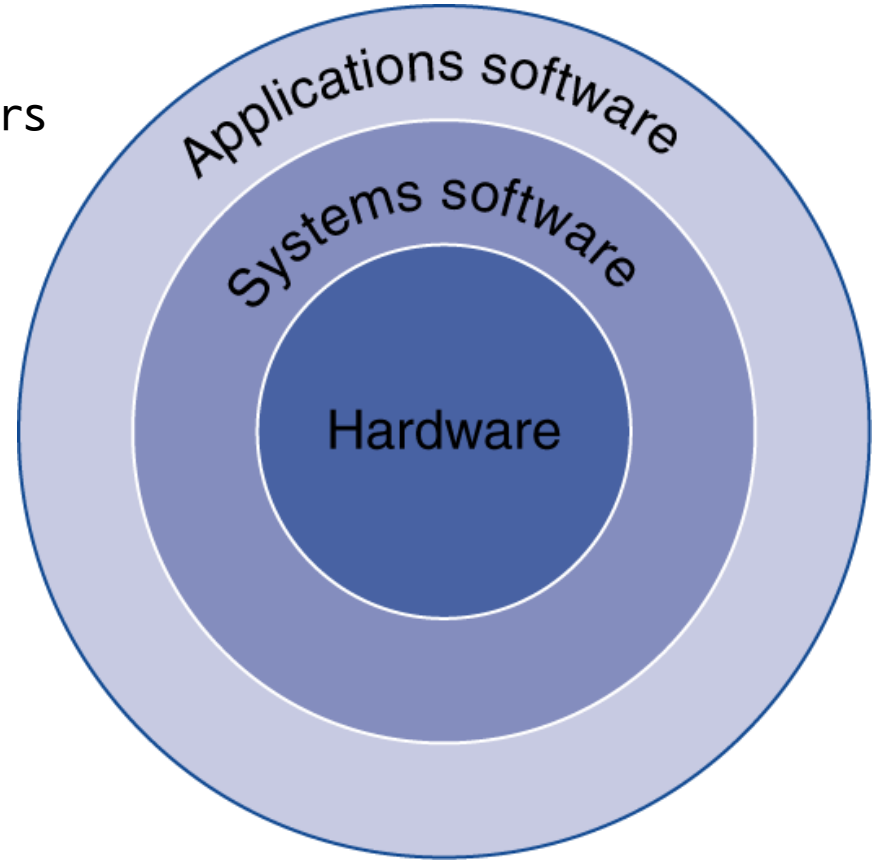
- Processor, memory, I/O controllers

System software

- Compiler: translates HLL code to machine code
- Operating System: service code
  - Handling input/output
  - Managing memory and storage
  - Scheduling tasks & sharing resources

Application software

- Written in high-level language



Levels of Program Code

High-level language

- Level of abstraction closer to problem domain
- Provides for productivity and portability

Assembly language

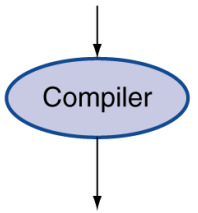
- Textual representation of instructions

Hardware representation

- Binary digits (bits)
- Encoded instructions and data

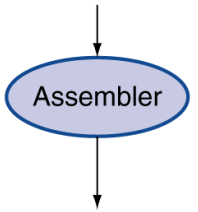
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

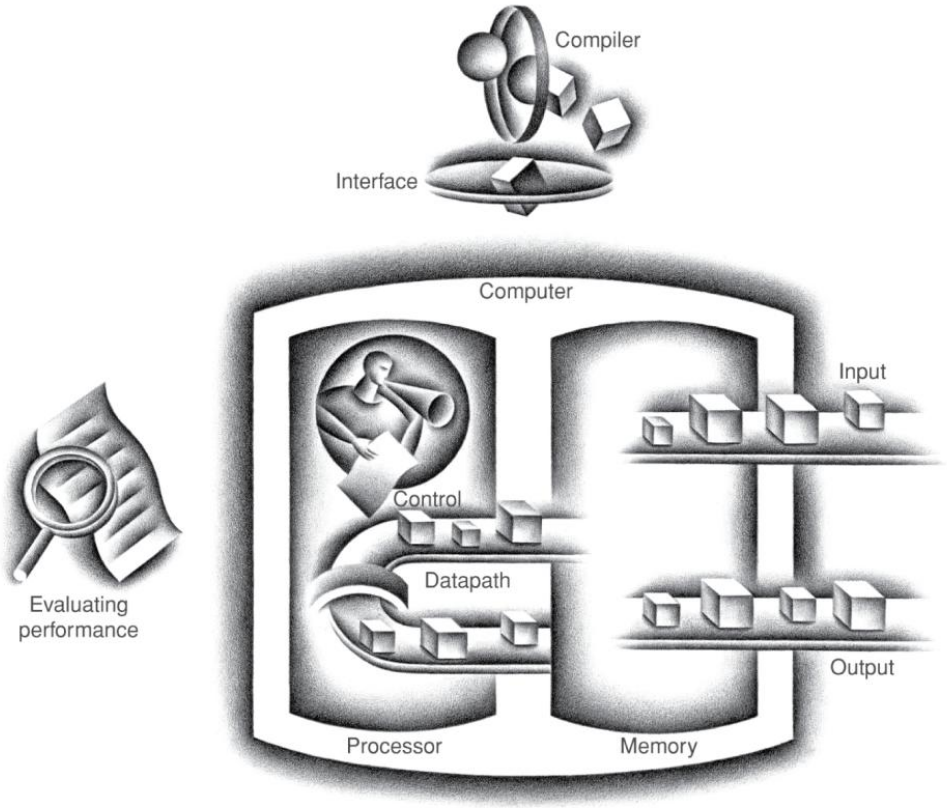
Components of a Computer

Same components for all kinds of computer

- Desktop, server, embedded

Input/output includes

- User-interface devices
  - Display, keyboard, mouse
- Storage devices
  - Hard disk, CD/DVD, flash
- Network adapters
  - For communicating with other computers

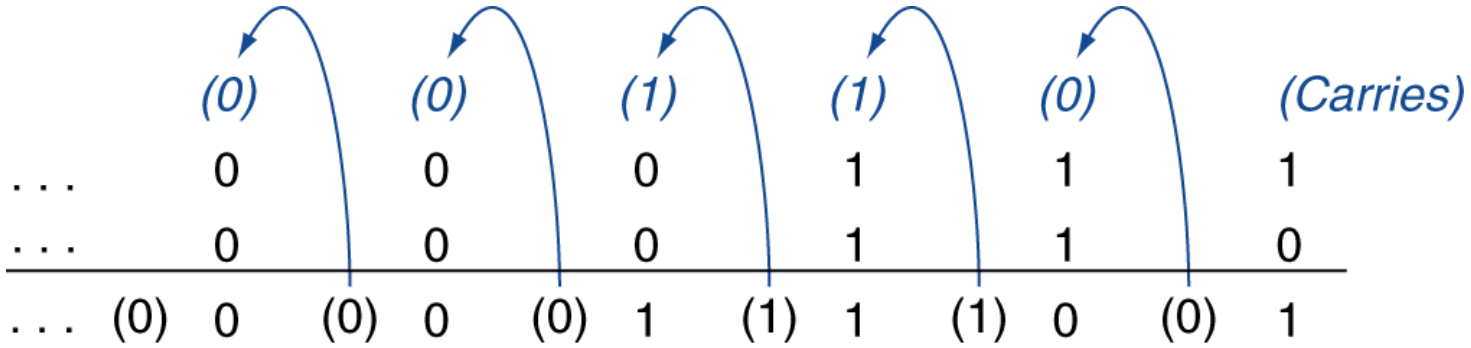


Arithmetic for Computers

Jo, Heeseung

Integer Addition

Example: 7 + 6



Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands
  - Overflow if result sign is 1
- Adding two –ve operands
  - Overflow if result sign is 0

Integer Subtraction

Add negation of second operand

Example: 7 – 6 = 7 + (–6)

```
+7: 0000 0000 … 0000 0111
-6: 1111 1111 … 1111 1010
+1: 0000 0000 … 0000 0001
```

Overflow if result out of range

- Subtracting two +ve or two –ve operands, no overflow
- Subtracting +ve from –ve operand
  - Overflow if result sign is 0
- Subtracting –ve from +ve operand
  - Overflow if result sign is 1

Representation of Negative Numbers

$+N$	Positive Integers (all systems)	$-N$	Negative Integers		
			Sign and Magnitude	2's Complement N^*	1's Complement $\bar{N}$
+0	0000	−0	1000	——	1111
+1	0001	−1	1001	1111	1110
+2	0010	−2	1010	1110	1101
+3	0011	−3	1011	1101	1100
+4	0100	−4	1100	1100	1011
+5	0101	−5	1101	1011	1010
+6	0110	−6	1110	1010	1001
+7	0111	−7	1111	1001	1000
		−8	——	1000	——

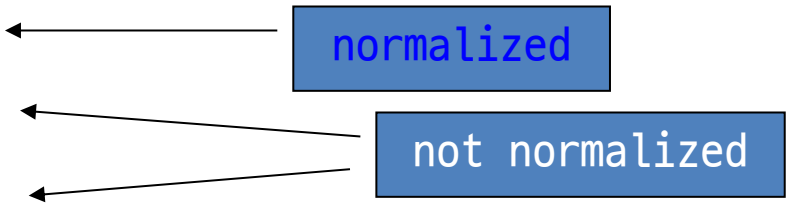
Floating Point

Representation for non-integral numbers

- Including very small and very large numbers

Scientific notation

- -2.34×10^{56}
- $+0.002 \times 10^{-4}$
- $+987.02 \times 10^{9}$



In binary

- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

Types float and double in C

Floating Point Standard

Defined by IEEE Std 754-1985

Developed in response to divergence of representations

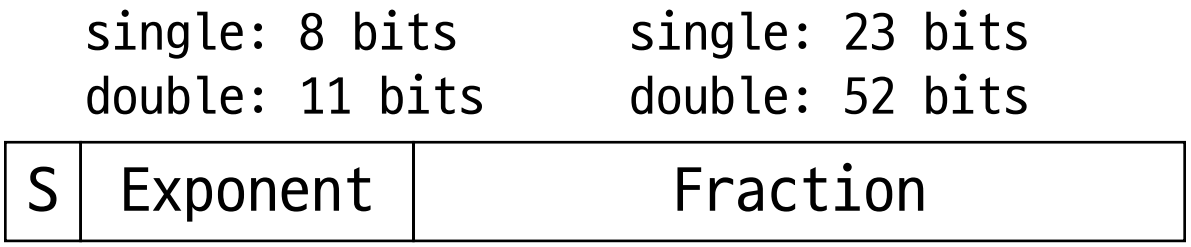
- Portability issues for scientific code

Now almost universally adopted

Two representations

- Single precision (32-bit)
- Double precision (64-bit)

IEEE Floating-Point Format



$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)

Normalize significand: $1.0 \leq |\text{significand}| < 2.0$

- Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
- Significand is Fraction with the "1." restored

Exponent: excess representation: actual exponent + Bias

- Ensures exponent is unsigned
- Single: Bias = 127; Double: Bias = 1023

Floating-Point Example

Represent -0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000\ldots00_2$
- Exponent = -1 + Bias
  - Single: -1 + 127 = 126 = 01111110_2
  - Double: -1 + 1023 = 1022 = 01111111110_2

S	Exponent	Fraction

Single: 10111111010000…00

Double: 10111111111101000…00

Floating-Point Example

What number is represented by the single-precision float

1100000010100000000000000000000

S	Exponent	Fraction

- S = 1
- Fraction = $01000\ldots00_2$
- Exponent = 10000001_2 = 129

$$x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$$
$$= (-1) \times 1.25 \times 2^2$$
$$= -5.0$$

Single-Precision Range

Exponents 00000000 and 11111111 reserved

Smallest value

- Exponent: 00000001
  $\Rightarrow$ actual exponent = 1 – 127 = –126
- Fraction: 000…00 $\Rightarrow$ significand = 1.0
- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Largest value

- exponent: 11111110
  $\Rightarrow$ actual exponent = 254 – 127 = +127
- Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

Exponents 0000…00 and 1111…11 reserved

Smallest value

- Exponent: 00000000001
  ⇒ actual exponent = 1 – 1023 = –1022
- Fraction: 000…00 ⇒ significand = 1.0
- $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

Largest value

- Exponent: 11111111110
  ⇒ actual exponent = 2046 – 1023 = +1023
- Fraction: 111…11 ⇒ significand ≈ 2.0
- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

The Processor

Jo, Heeseung

Instruction Set

The repertoire of instructions of a computer

Different computers have different instruction sets

- But with many aspects in common

Early computers had very simple instruction sets

- Simplified implementation

Many modern computers also have simple instruction sets

Register Operand Example

Add and subtract, three operands

- Two sources and one destination

```
add a, b, c          # a gets b + c
```

C code:

```
f = (g + h) - (i + j);
```

- f, ..., j in $s0, ..., $s4

Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operand Example 1

C code:

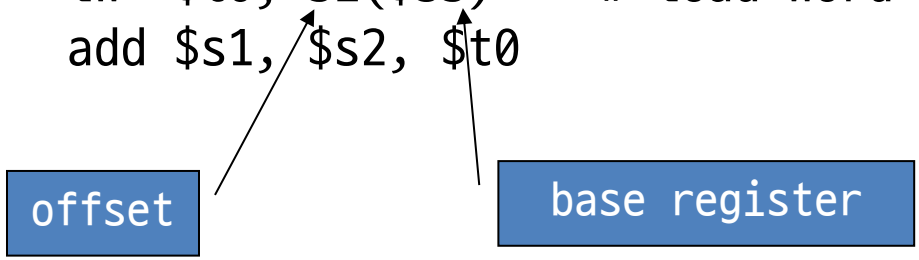
```
g = h + A[8];
```

- g in $s1, h in $s2, base address of A in $s3

Compiled MIPS code:

- Index 8 requires offset of 32
  - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```



Registers vs. Memory

Registers are much much faster to access than memory

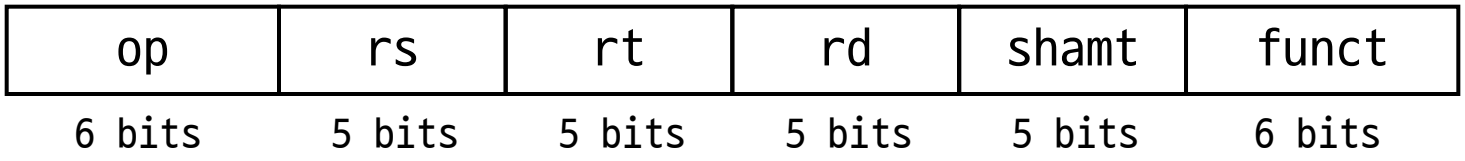
Operating on memory data requires loads and stores

- More instructions to be executed

Compiler must use registers for variables as much as possible

- Only spill to memory for less frequently used variables
- Register optimization is important!

R-format Example



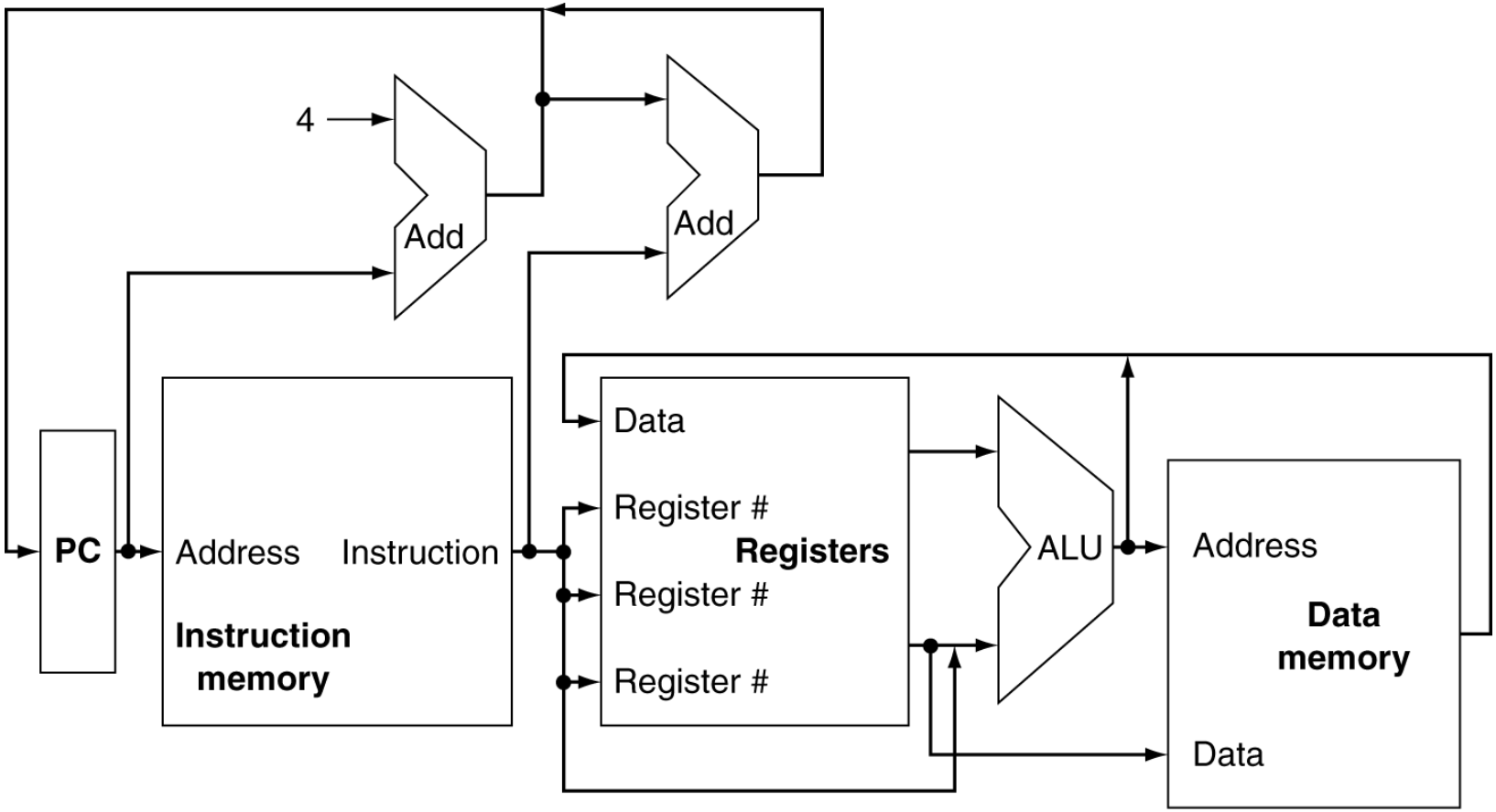

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000100011001001000000000100000_2 = 02324020_{16}$

CPU Overview

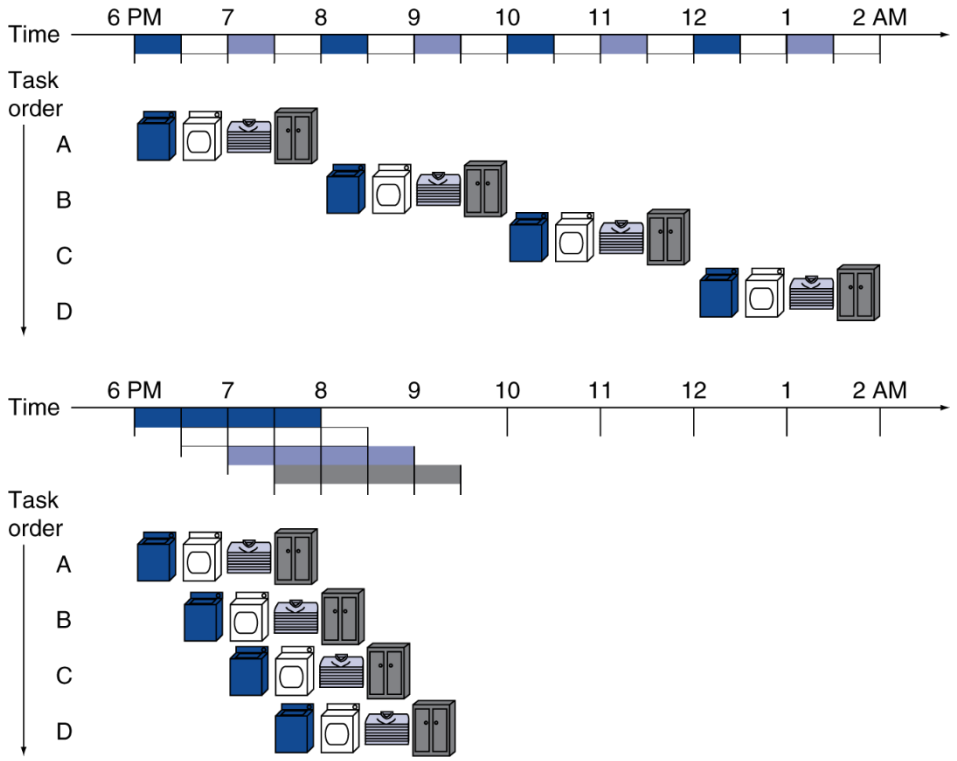
How to make a better CPU?



Pipelining Analogy

Pipelined laundry: overlapping execution

- Parallelism improves performance



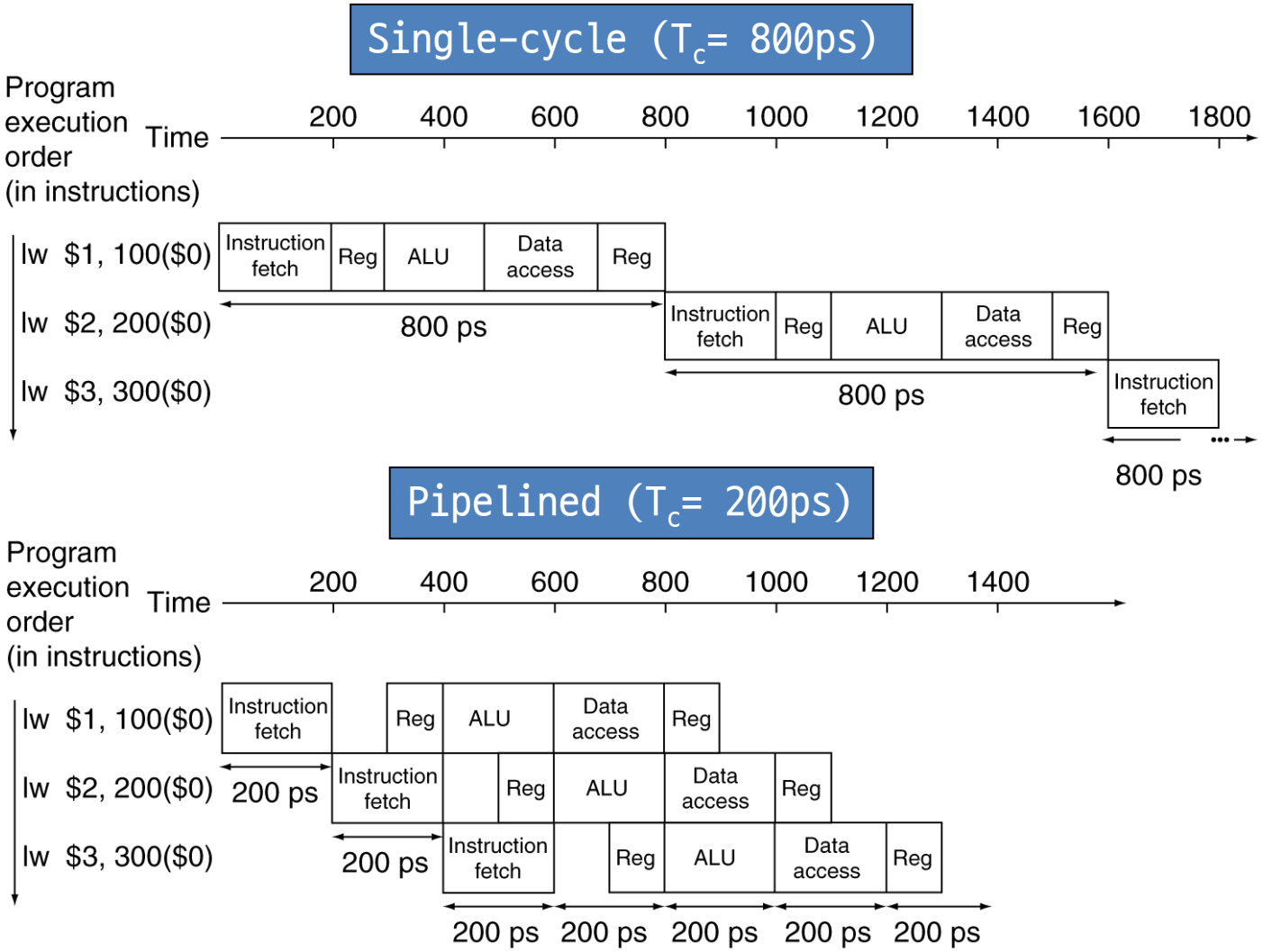
Four loads:

- Speedup
  = 8/3.5 = 2.3

Non-stop:

- Speedup
  ≈ 4
  = number of stages

Pipeline Performance



Hazards

Situations that prevent starting the next instruction in the next cycle

Structure hazards

- A required resource is busy

Data hazard

- Need to wait for previous instruction to complete its data read/write

Control hazard

- Deciding on control action depends on previous instruction

Structure Hazards

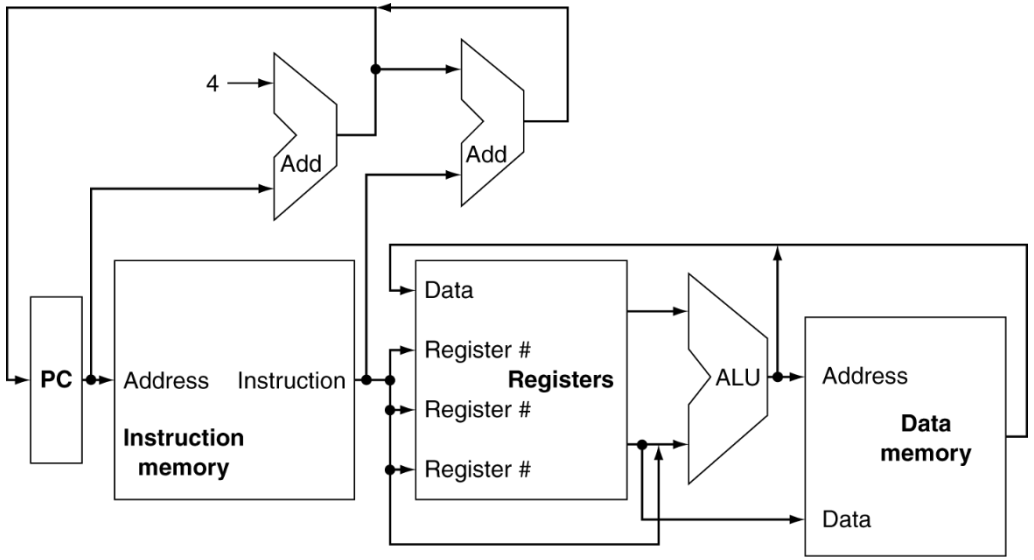
Conflict for use of a resource

If MIPS pipeline uses a single memory (1 inst/data memory)

- Load/store requires data access
- Instruction fetch would have to *stall* for that cycle
  - Would cause a pipeline "bubble"

Hence, pipelined datapaths require separate instruction/data memories

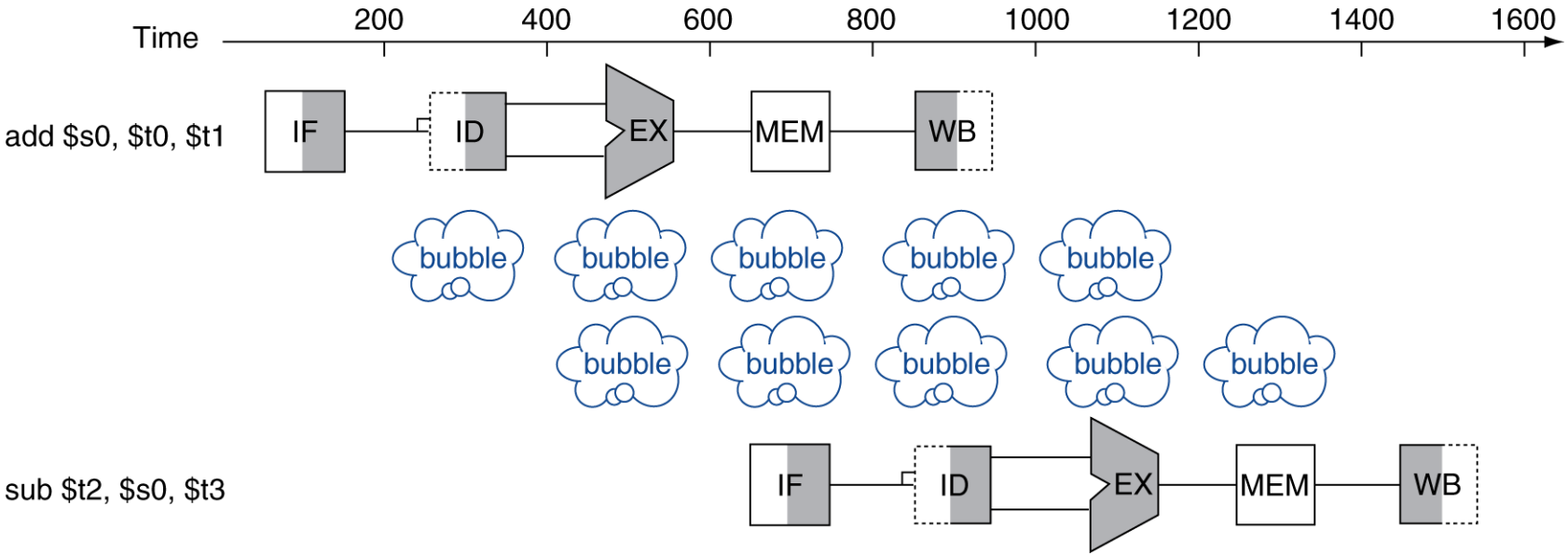
- Or separate instruction/data caches



Data Hazards

An instruction depends on completion of data access by a previous instruction

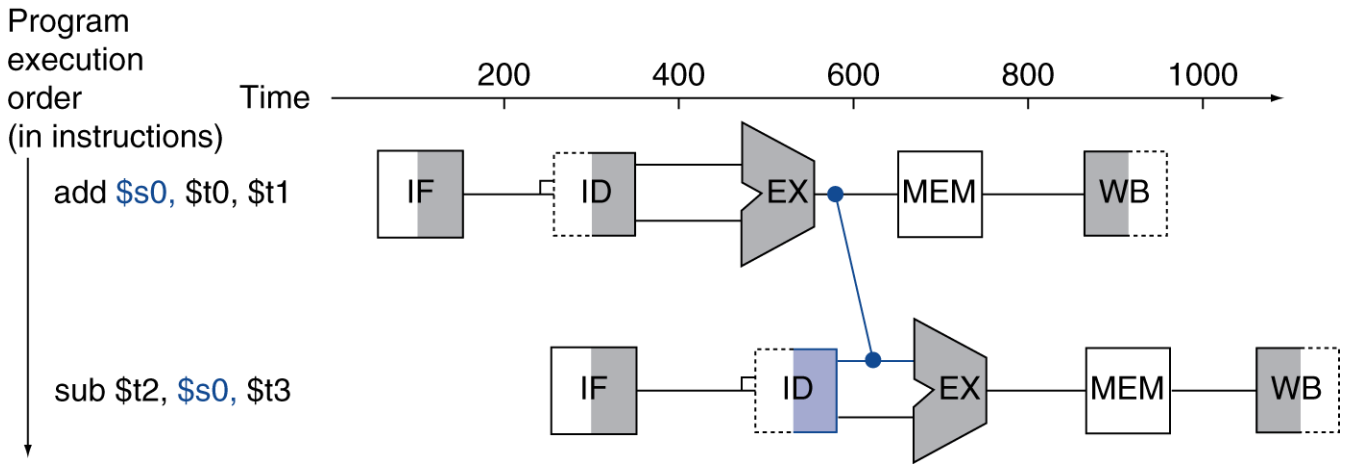
```
add     $s0, $t0, $t1
sub     $t2, $s0, $t3
```



Forwarding (aka Bypassing)

Use result when it is computed

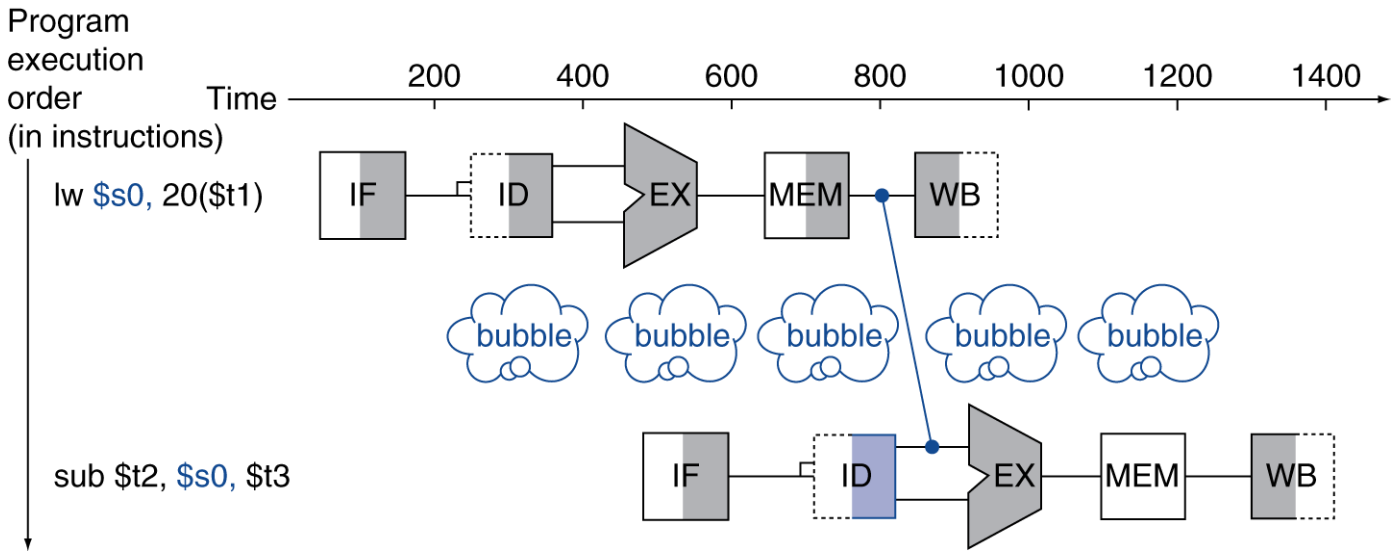
- Don't wait for it to be stored in a register
- Requires extra connections in the datapath



Load-Use Data Hazard

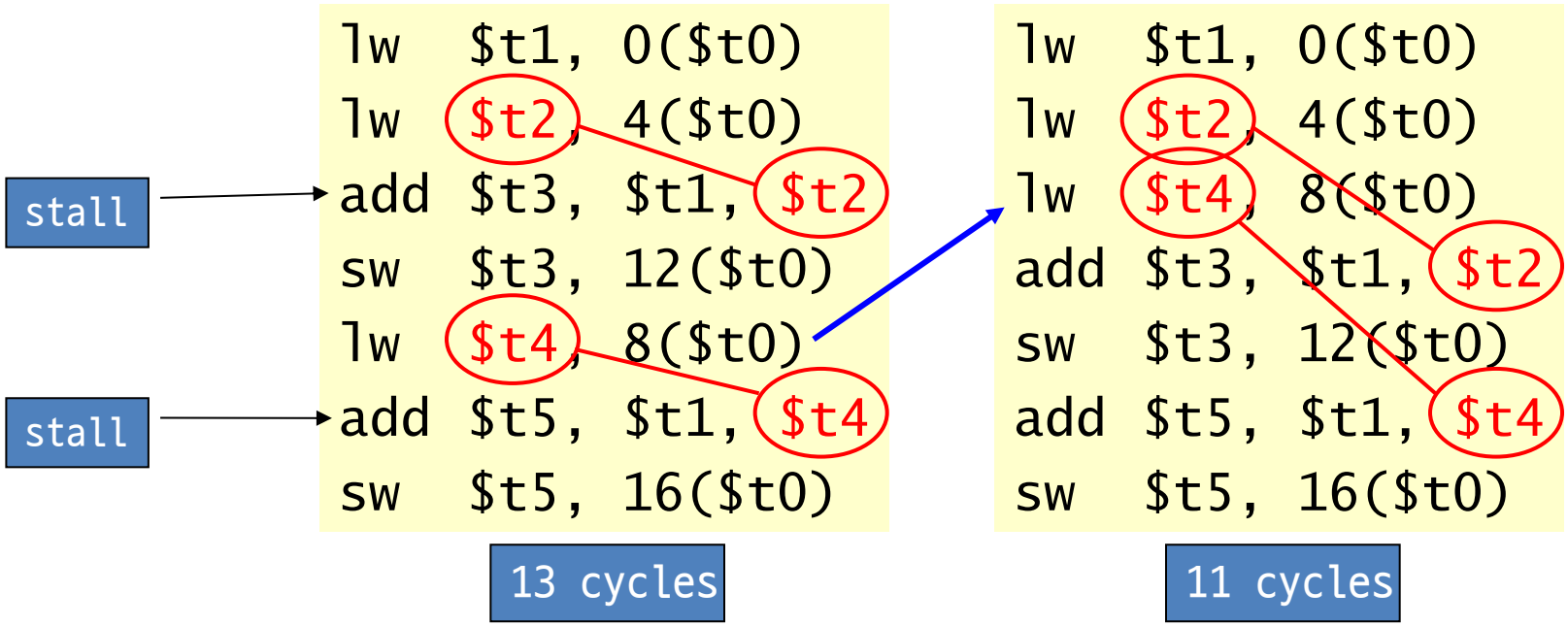
Can't always avoid stalls by forwarding

- If value not computed when needed
- Can't forward backward in time!



Code Scheduling to Avoid Stalls

Reorder code to avoid use of load result in the next instruction



Control Hazards

Branch determines flow of control

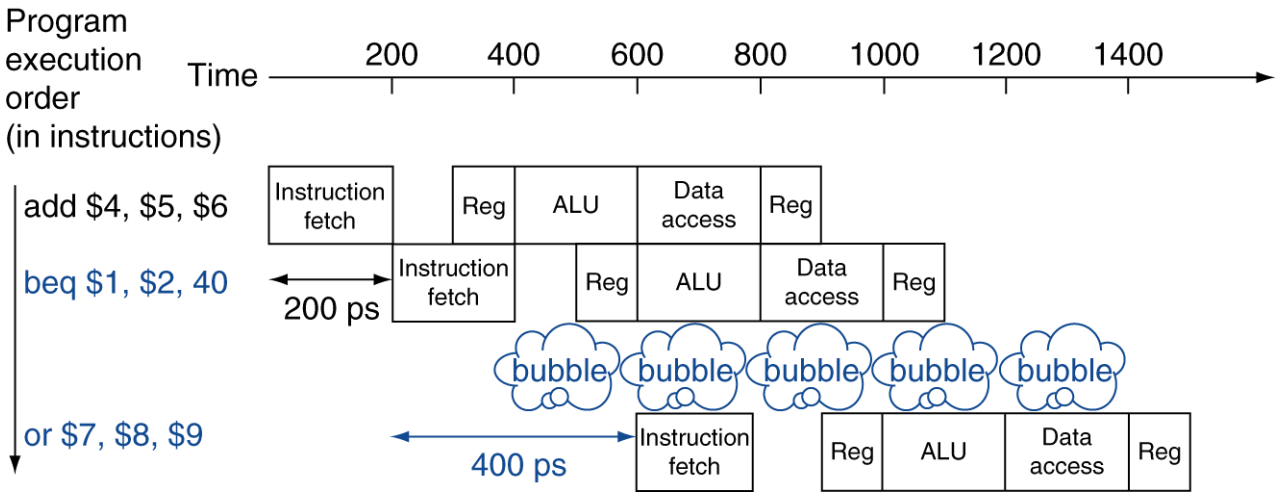
- Fetching next instruction depends on branch outcome
- Pipeline can't always fetch correct instruction
  - Still working on ID stage of branch

In MIPS pipeline

- Need to compare registers and compute target early in the pipeline
- Add hardware to do it in ID stage

Stall on Branch

Wait until branch outcome determined before fetching next instruction



Branch Prediction

Longer pipelines can't readily determine branch outcome early

- Stall penalty becomes unacceptable

Predict outcome of branch

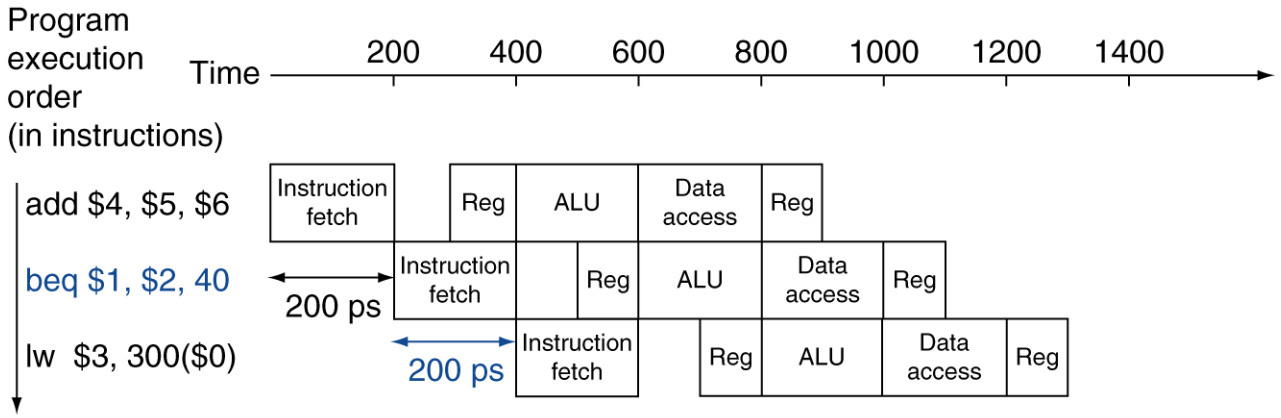
- Only stall if prediction is wrong

In MIPS pipeline

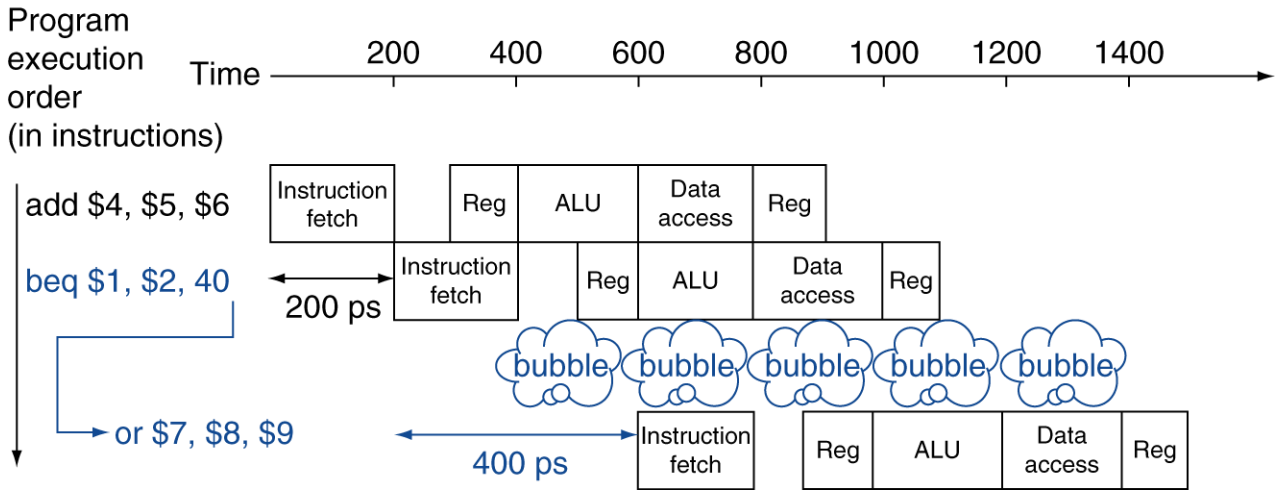
- Can predict branches not taken
- Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

Static branch prediction

- Based on typical branch behavior
- Example: loop and if-statement branches
  - Predict backward branches taken
  - Predict forward branches not taken

Dynamic branch prediction

- Hardware measures actual branch behavior
  - e.g., record recent history of each branch
- Assume future behavior will continue the trend
  - When wrong, stall while re-fetching, and update history

Large and Fast: Exploiting Memory Hierarchy

Jo, Heeseung

Memory Technology

Static RAM (SRAM)

- 0.5ns – 2.5ns, $2000 – $5000 per GB

Dynamic RAM (DRAM)

- 50ns – 70ns, $20 – $75 per GB

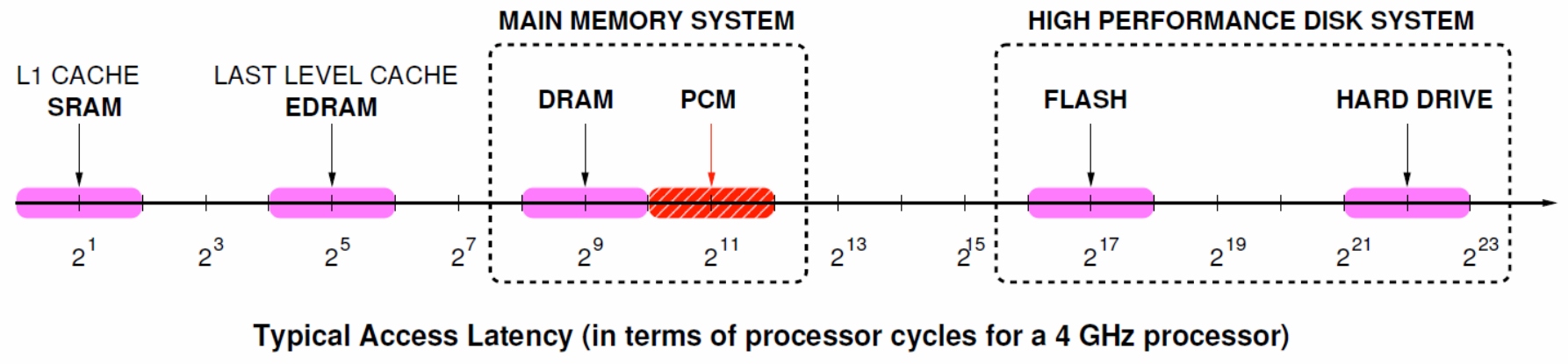
Magnetic disk

- 5ms – 20ms, $0.20 – $2 per GB

Ideal memory

- Access time of SRAM
- Capacity and cost/GB of disk

Registers vs. Memory



Typical Access Latency (in terms of processor cycles for a 4 GHz processor)

Qureshi (IBM Research) et al., Scalable High Performance Main Memory System Using Phase-Change Memory Technology, *ISCA* 2009.

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

Block (aka line): unit of copying

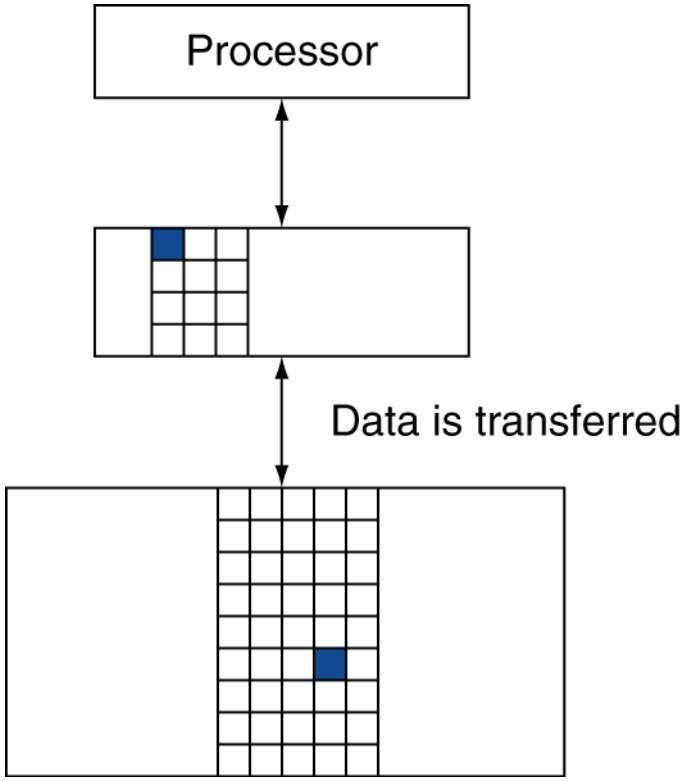
- May be multiple words

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

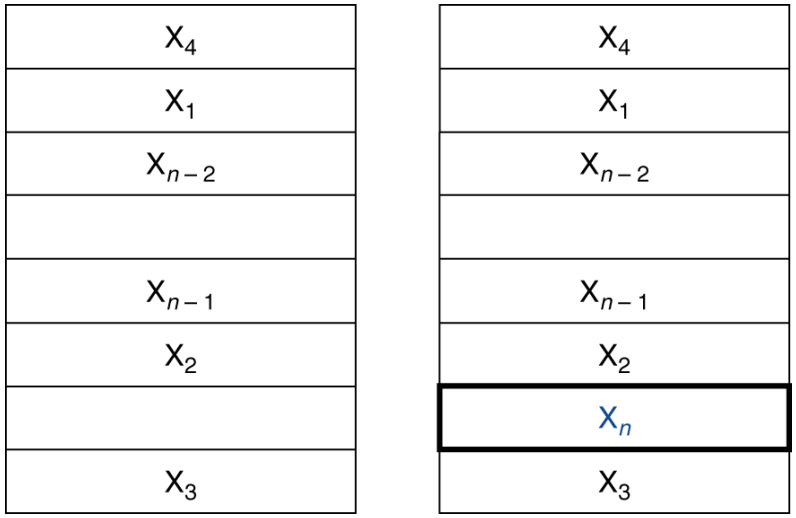


Cache Memory

Cache memory

- The level of the memory hierarchy closest to the CPU

Given accesses X_1, …, 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?

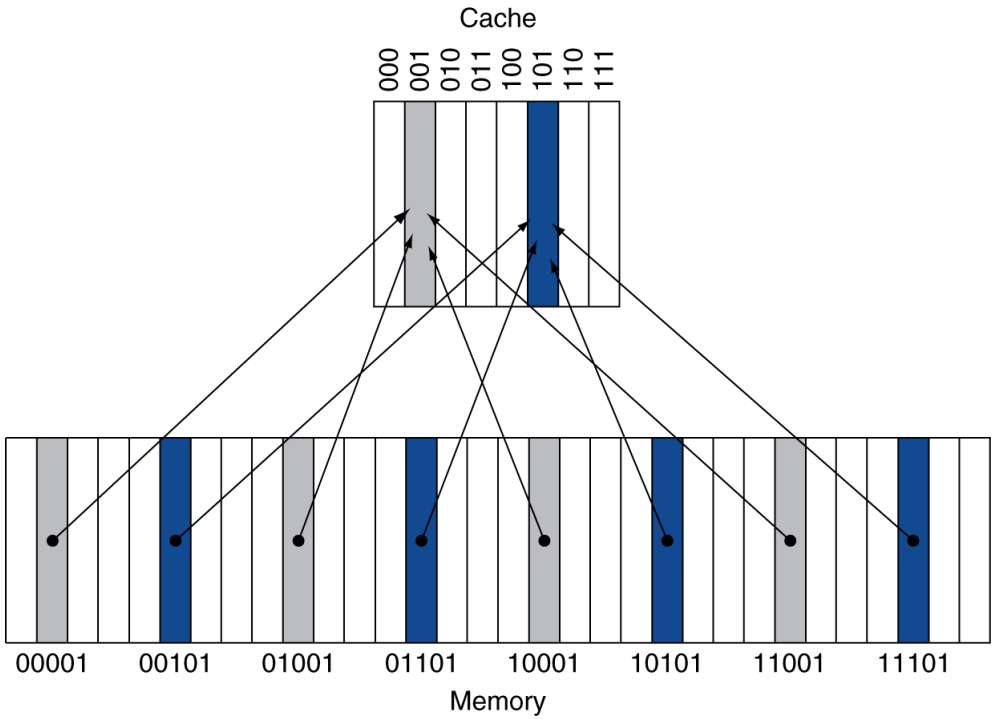
Where do we look?

Direct Mapped Cache

Location determined by address

Direct mapped: only one choice

- (Block address) modulo (#Blocks in cache)



Num. of Blocks is a power of 2

Use low-order address bits

Tags and Valid Bits

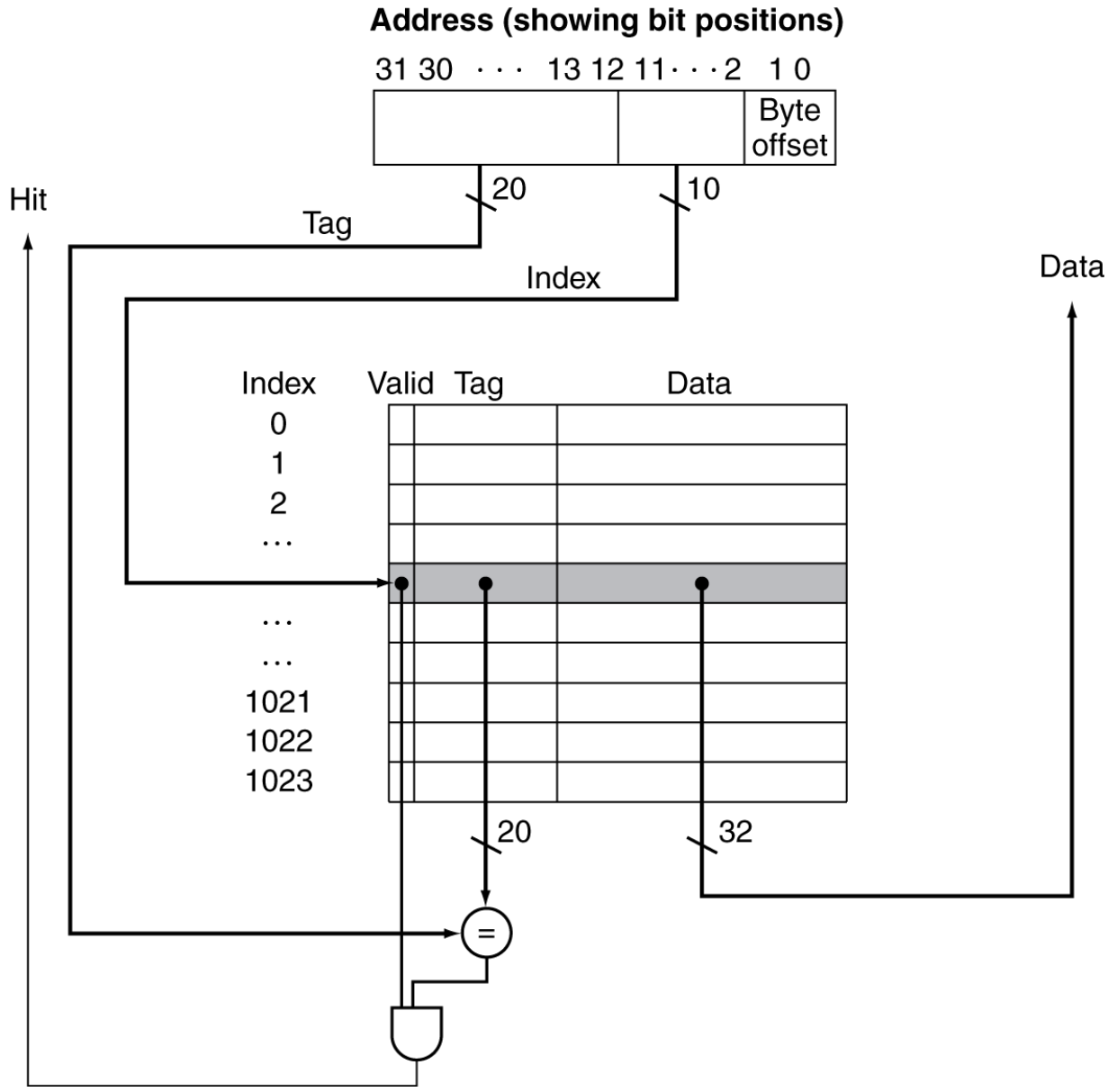
How do we know which particular block is stored in a cache location?

- 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

Address Subdivision



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

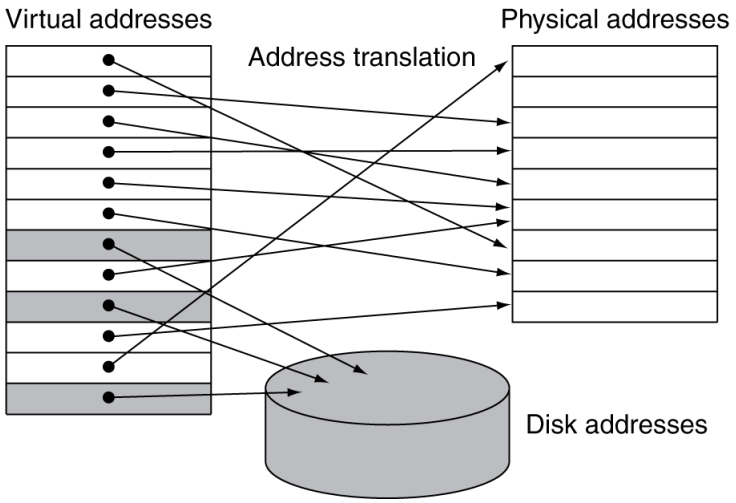
- VM "block" is called a page
- VM translation "miss" is called a page fault

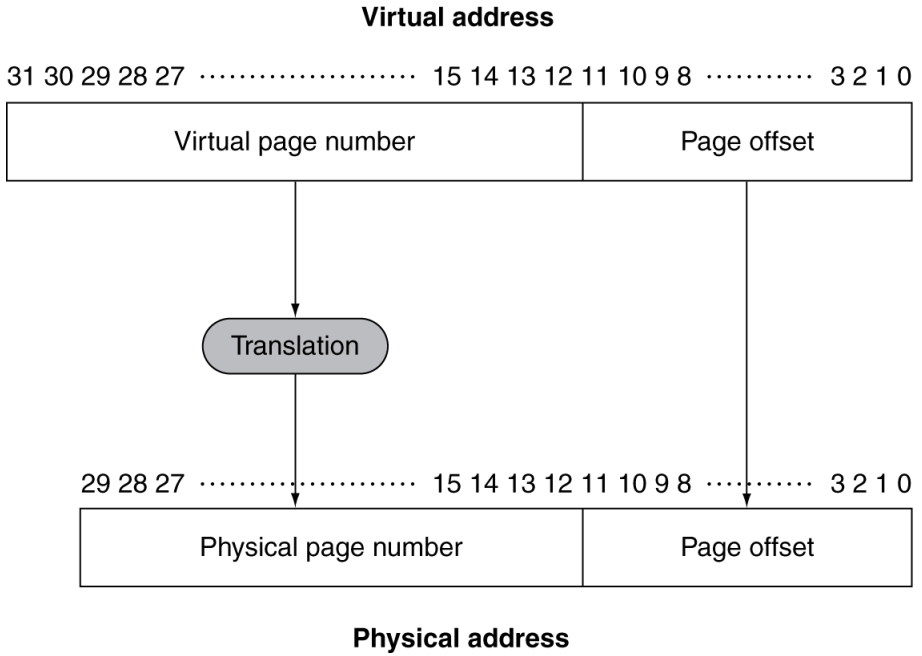
Virtual address <-> Physical address

(App. view) (Managed by kernel)

Address Translation

Fixed-size pages (e.g., 4K)





Page Fault Penalty

Page faults

- Referencing a virtual address in an evicted page

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

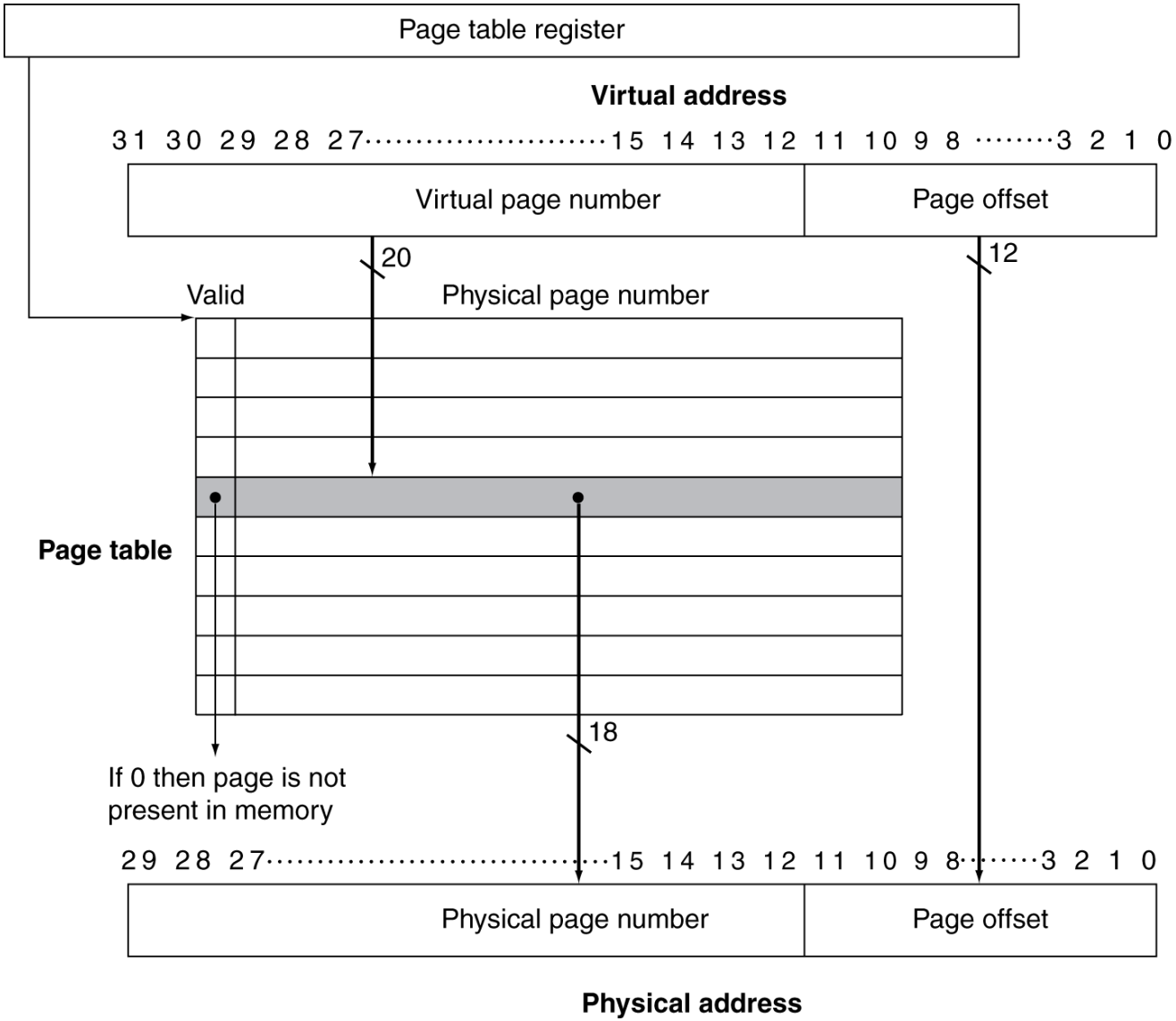
If page is present in memory

- PTE stores the physical page number
- Plus other status bits (referenced, dirty, …)

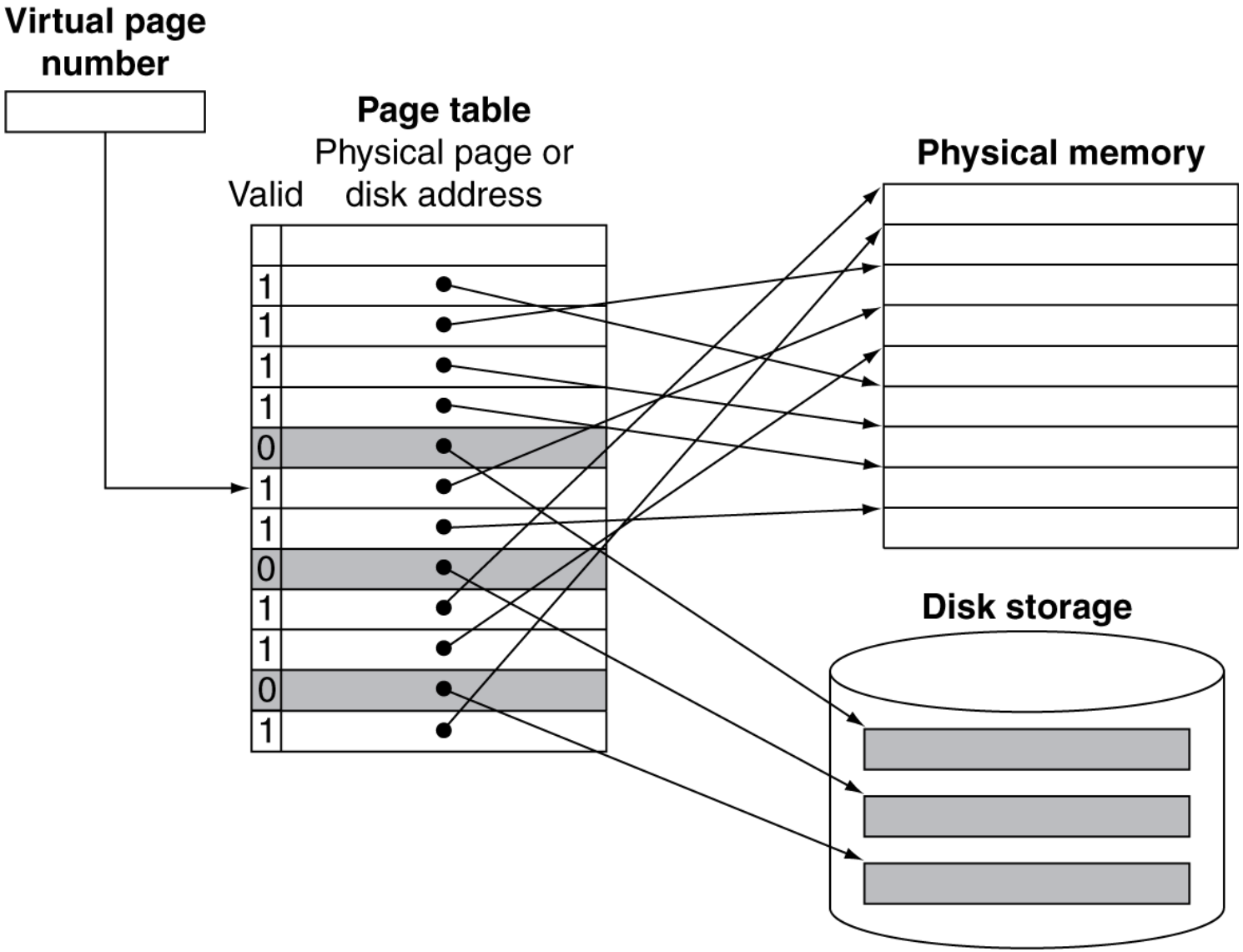
If page is not present

- PTE can refer to location in swap space on disk

Translation Using a Page Table



Mapping Pages to Storage



Fast Translation Using a TLB

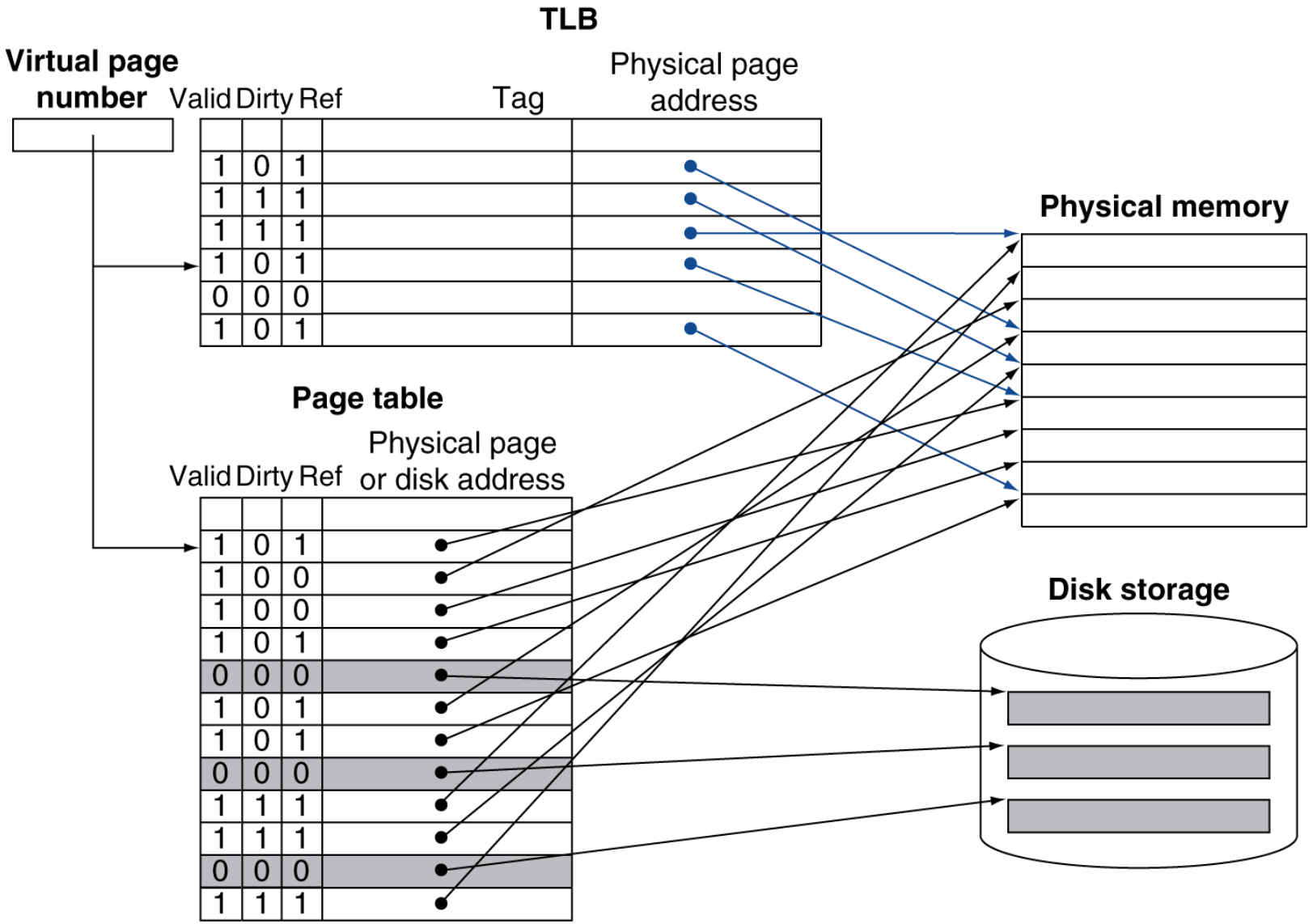
Address translation would appear to require extra memory references

- 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



Storage and Other I/O Topics

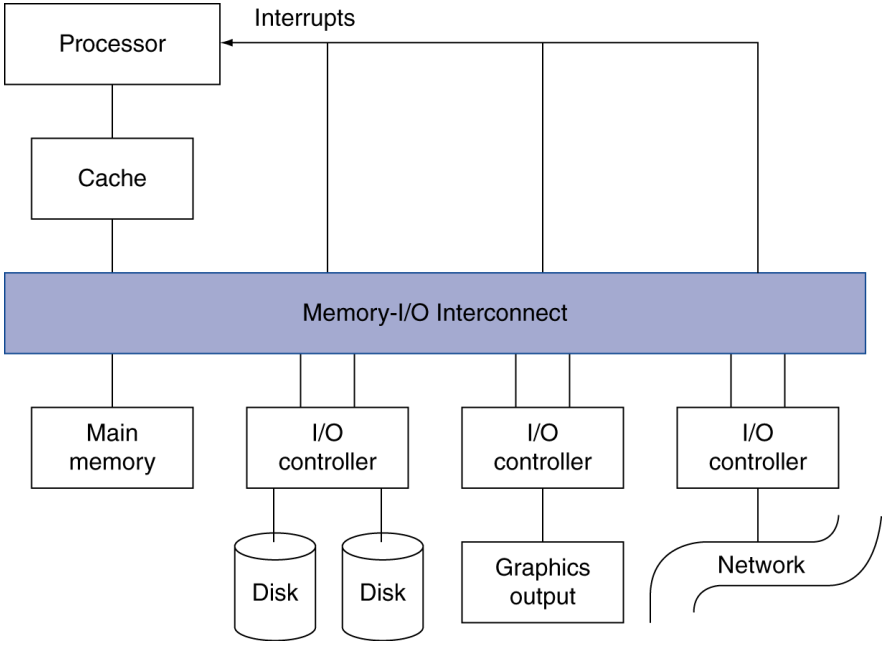
Jo, Heeseung

Introduction

I/O devices can be characterized by

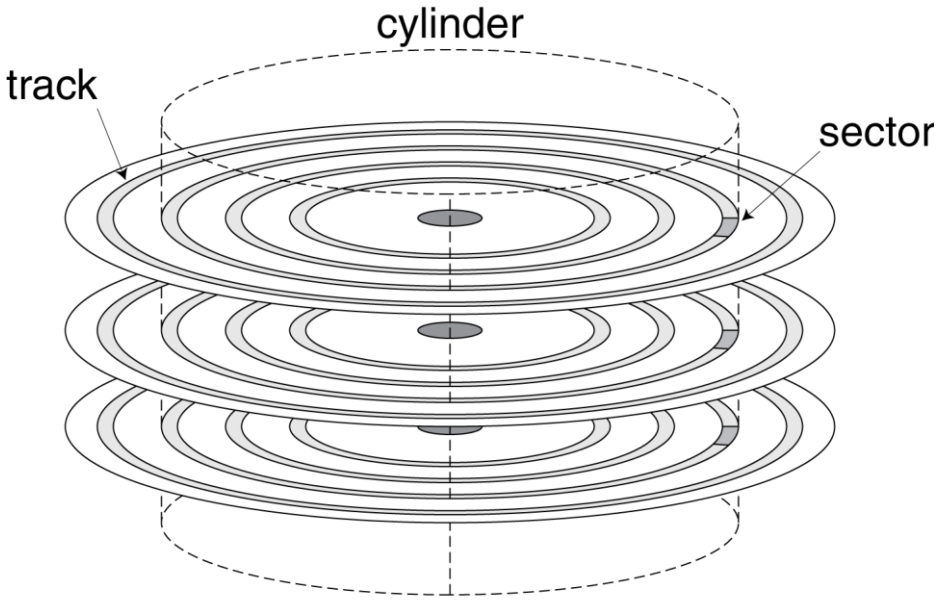
- Behaviour: input, output, storage
- Partner: human or machine
- Data rate: bytes/sec, transfers/sec

I/O bus connections



Disk Storage

Nonvolatile, rotating magnetic storage



Disk Sectors and Access

Each sector records

- Sector ID
- Data (512 bytes, 4096 bytes proposed)
- Error correcting code (ECC)
  - Used to hide defects and recording errors
- 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 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

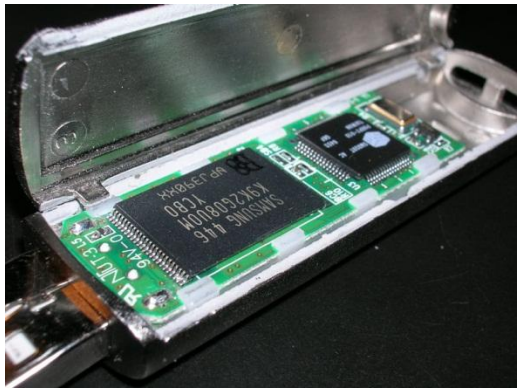
Disk drives include caches

- Prefetch sectors in anticipation of access
- Avoid seek and rotational delay

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

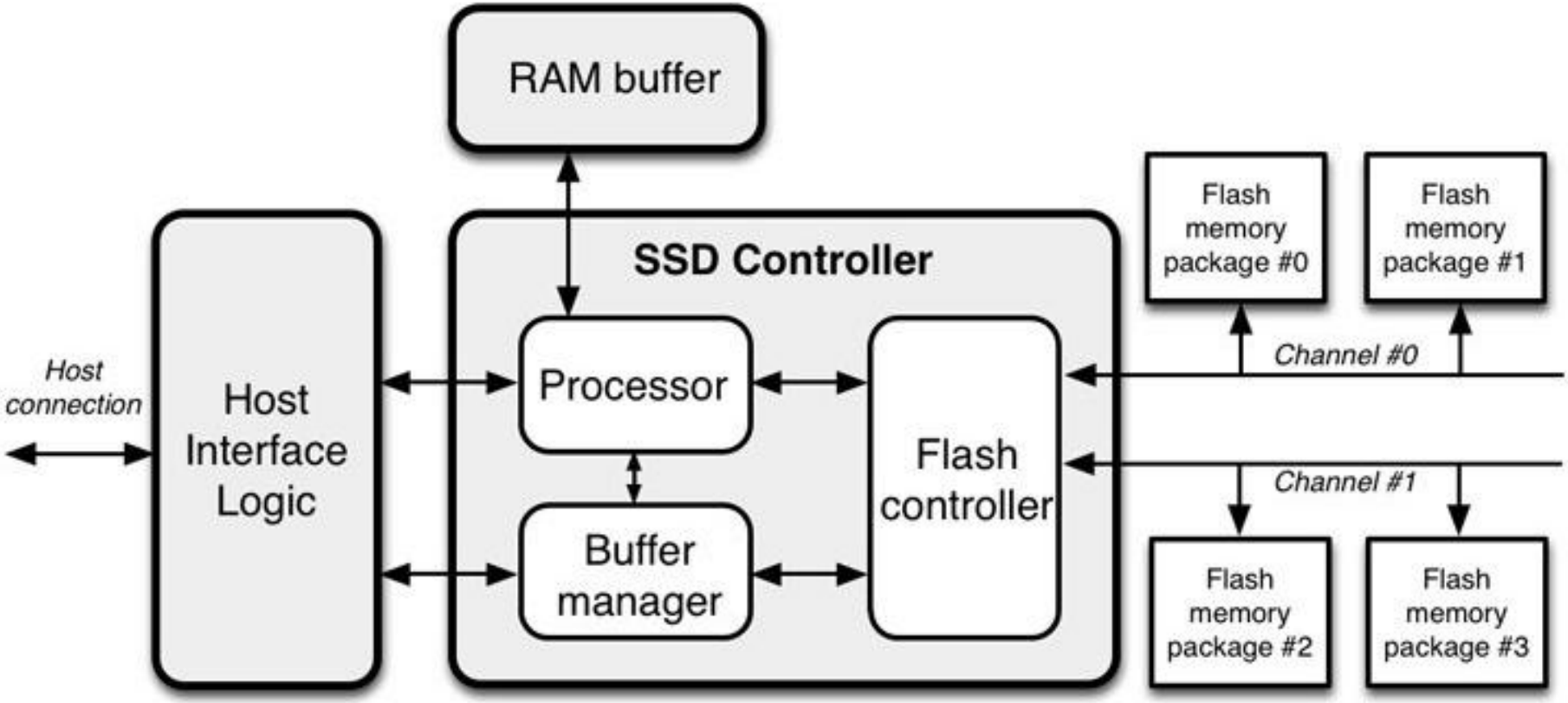
- Denser (bits/area), but block-at-a-time access
- Cheaper per GB
- Used for USB keys, media storage, …

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

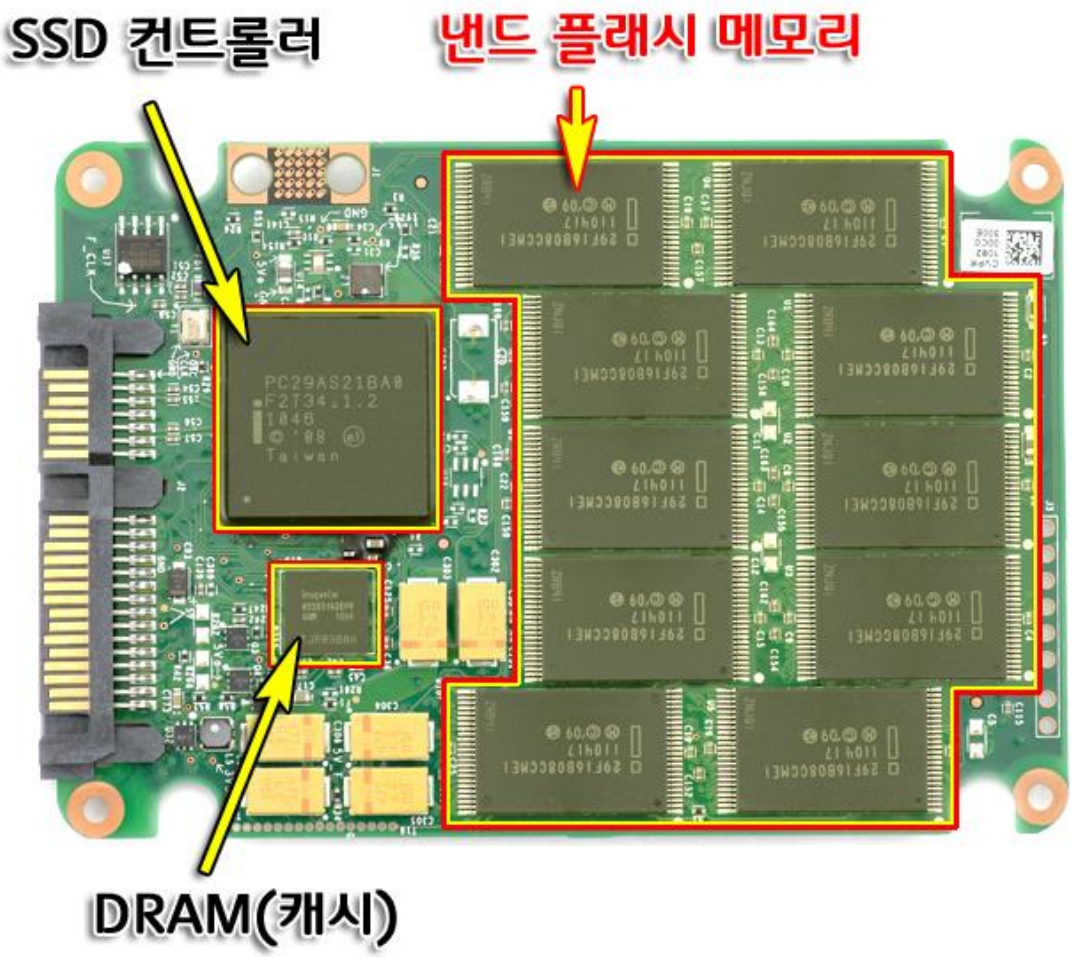
Solid state drive (SSD)

Architecture of SSD

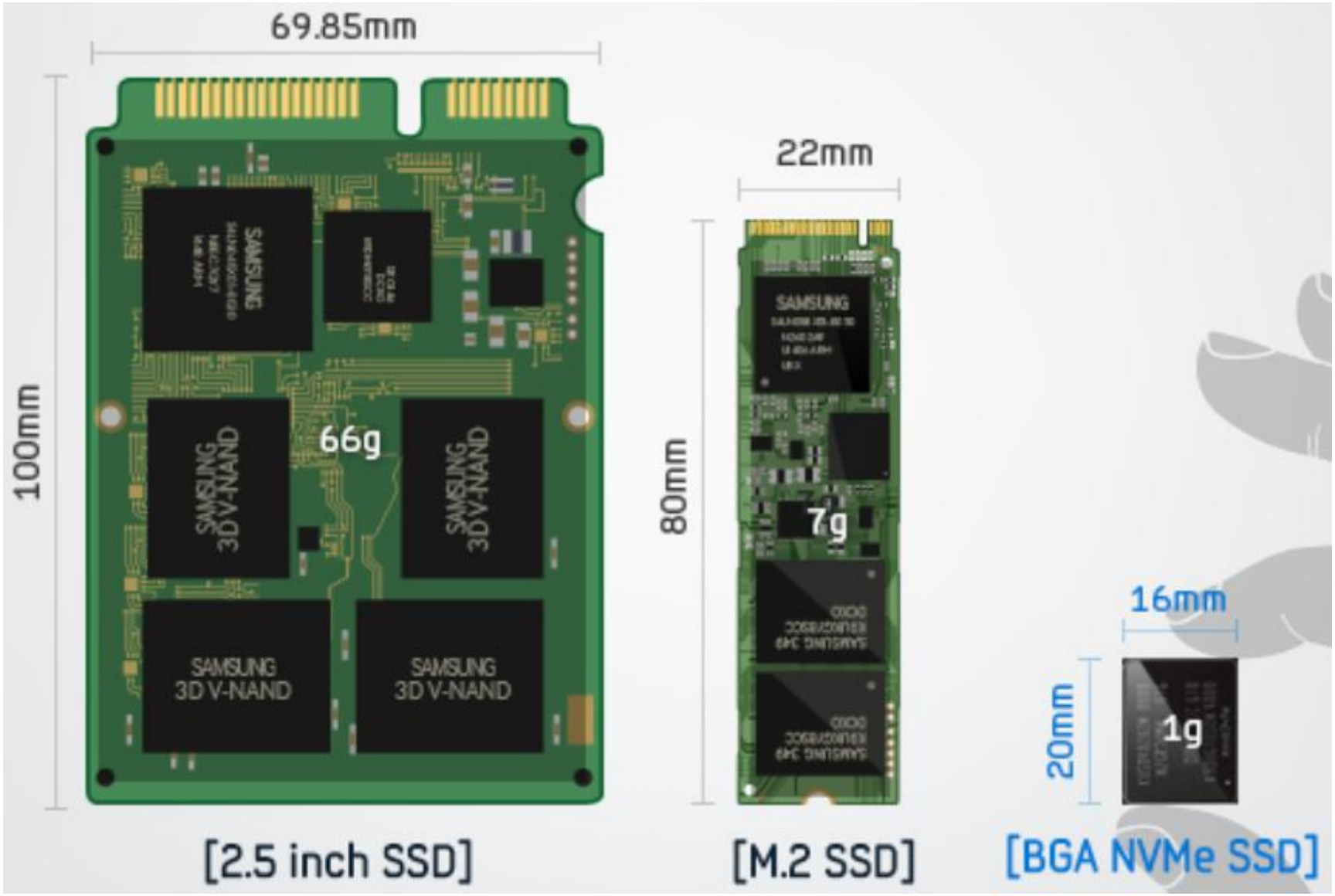


Solid state drive (SSD)

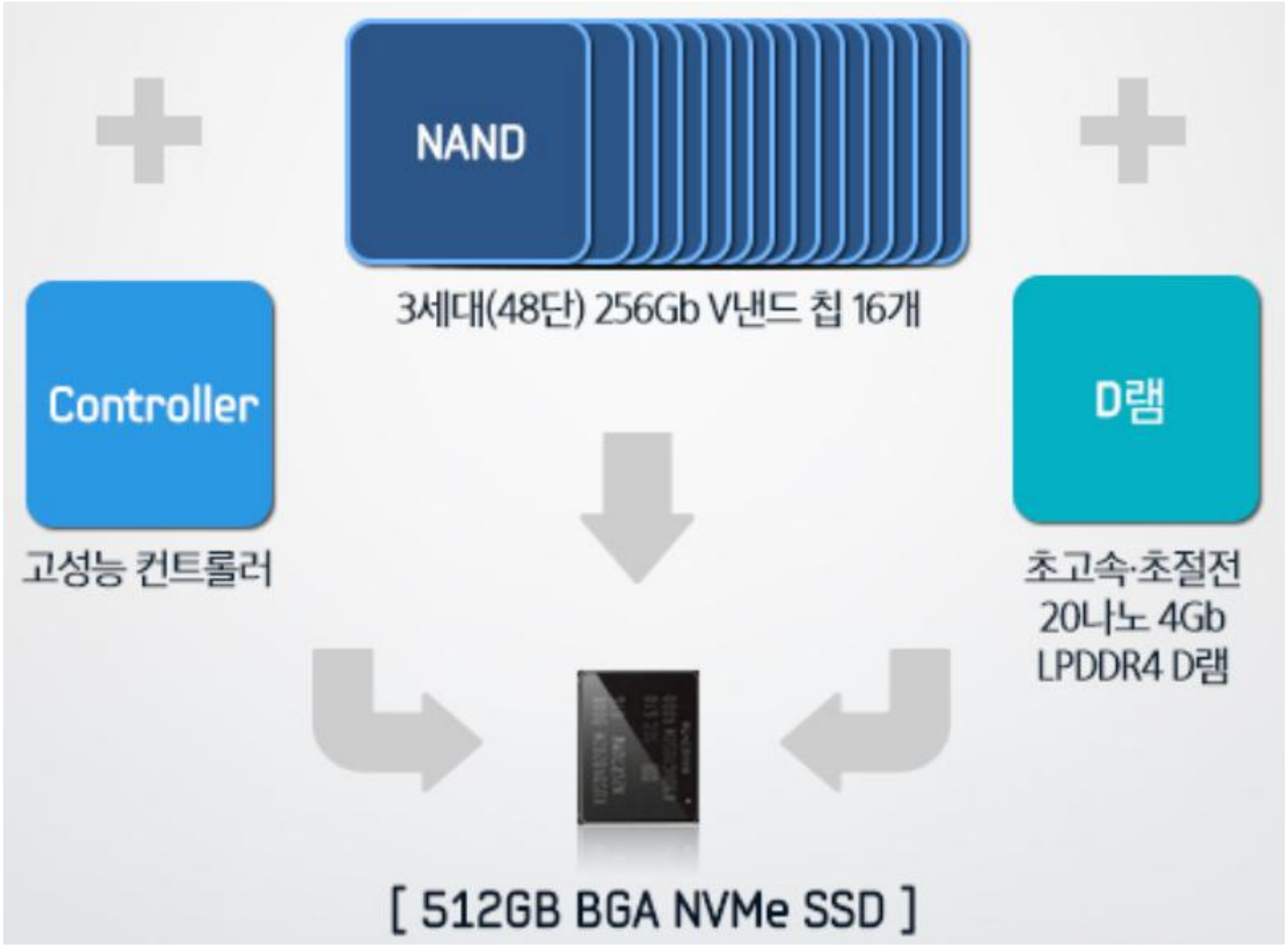
Architecture of SSD



Solid state drive (SSD)

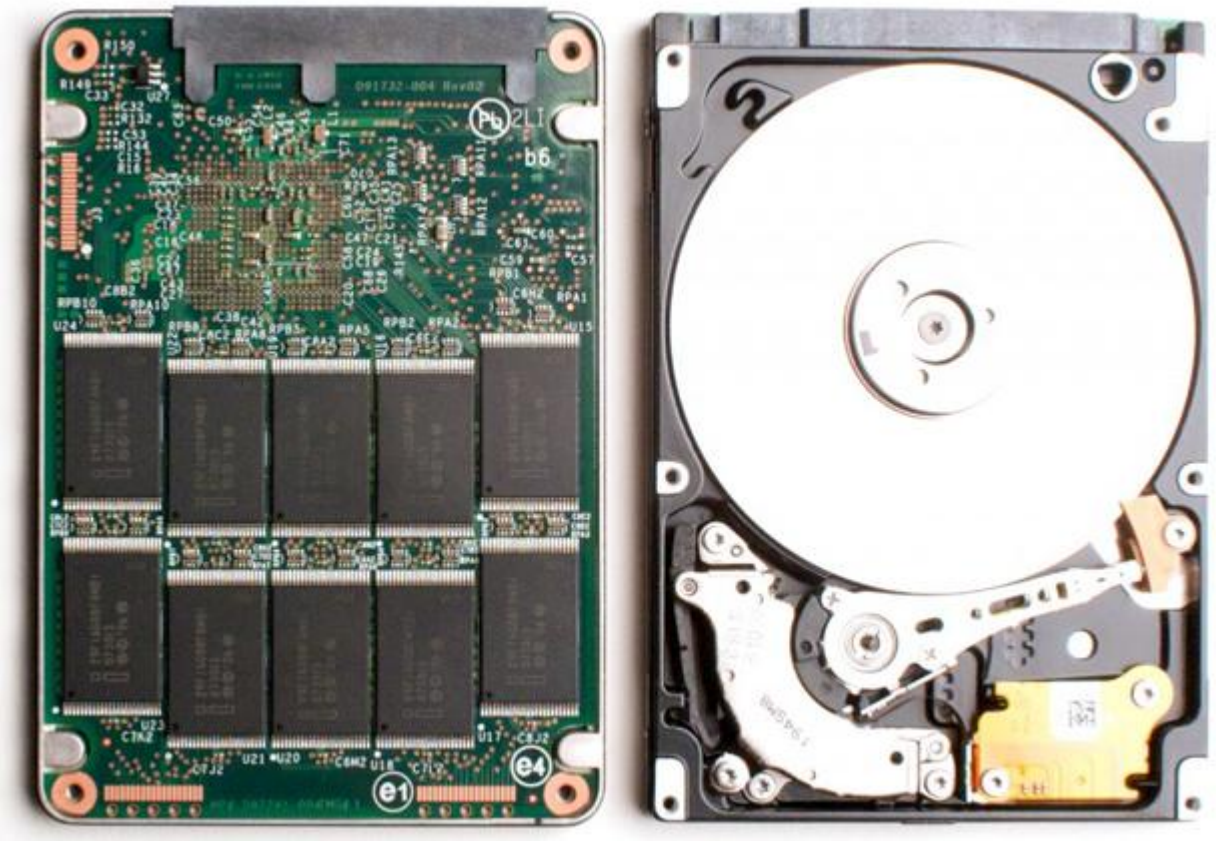


Solid state drive (SSD)



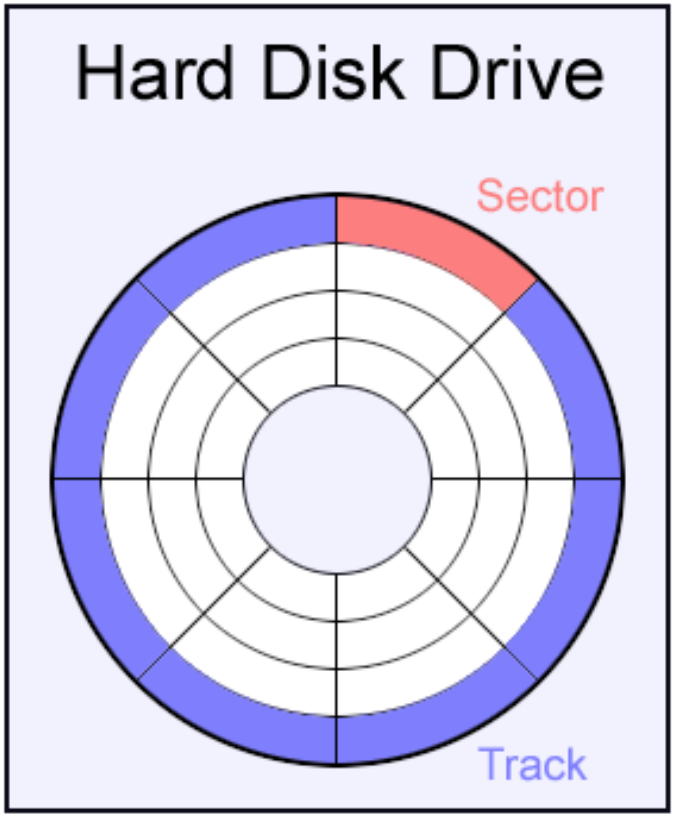
Solid state drive (SSD)

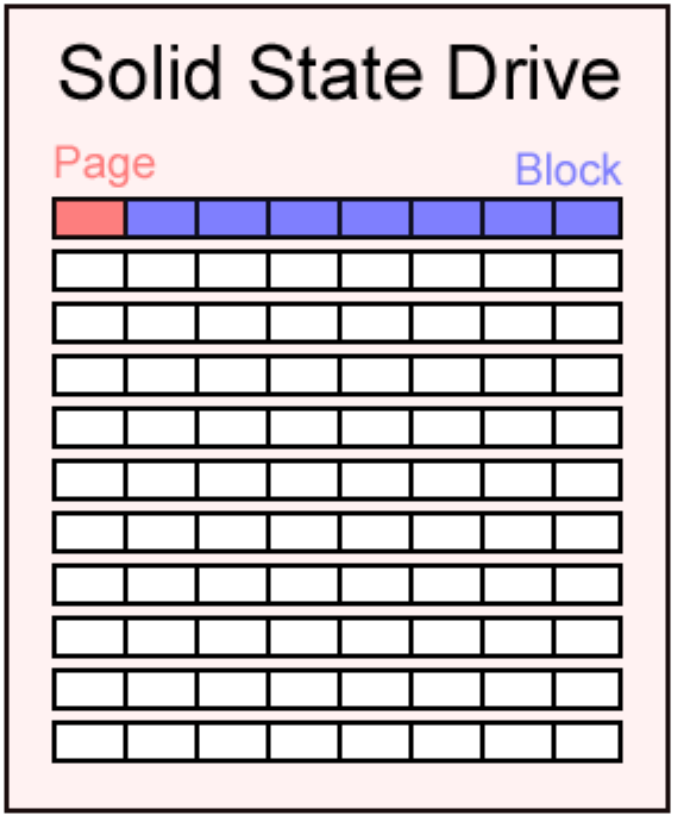
HDD vs. SSD



Solid state drive (SSD)

HDD vs. SSD





Interconnecting Components

Need interconnections between

- CPU, memory, I/O controllers

Bus: shared communication channel

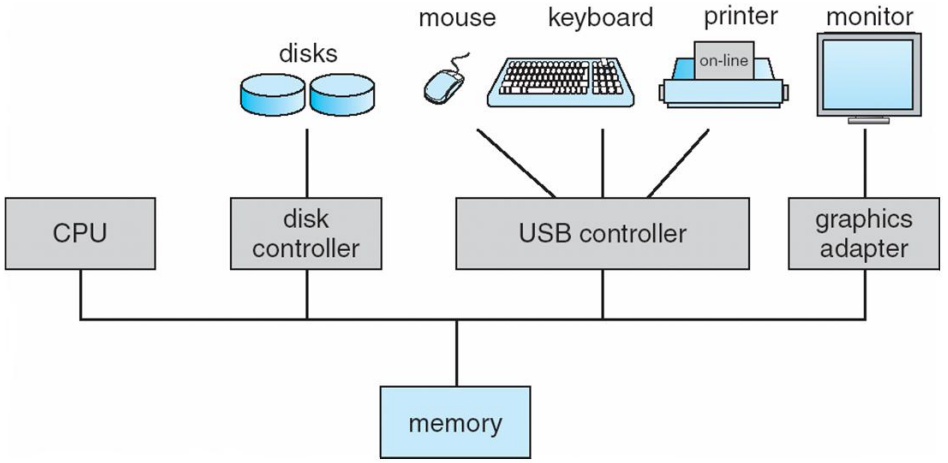
- Parallel set of wires for data and synchronization of data transfer
- Can become a bottleneck

Performance limited by physical factors

- Wire length, number of connections

More recent alternative: high-speed serial connections with switches

- Like networks



Typical x86 PC I/O System