CS330 - Computer Organization and Assembly Language Programming

Lecture 19

-Pipelining, Memory Hierarchy-

Professor : Mahmut Unan – UAB CS

Agenda

- Pipelining
  - Limitations
  - Hazards
  - Pipeline Stages
  - Prediction
- Memory Hierarchy
  - Storage technologies and trends
  - Locality of reference
  - Caching in the memory hierarchy

Slides were adopted from Dr. Randal E. Bryant

Real-World Pipelines: Car Washes

Sequential

Parallel

Pipelined

- Idea
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any given times, multiple objects being processed

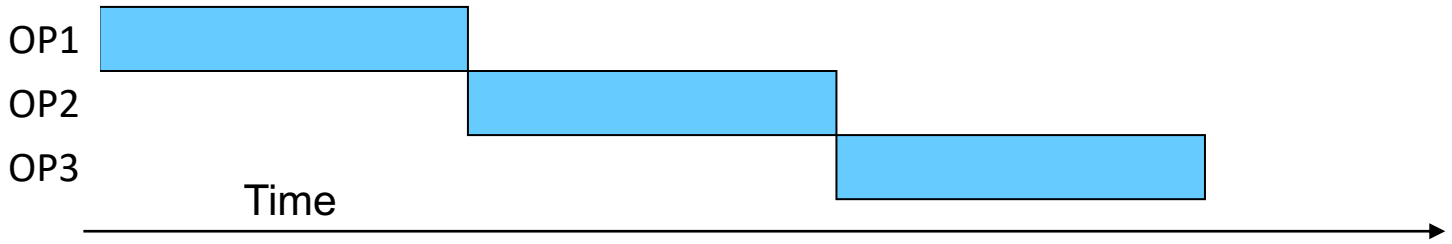
Pipelining

Pipelining is running multiple *stages* of the same *process* in parallel in a way that efficiently uses all the available hardware while respecting the dependencies of each stage upon the previous stages.
Latency is the time to complete a single task.
Throughput is the rate at which tasks complete.

http://15418.courses.cs.cmu.edu/spring2013/article/17

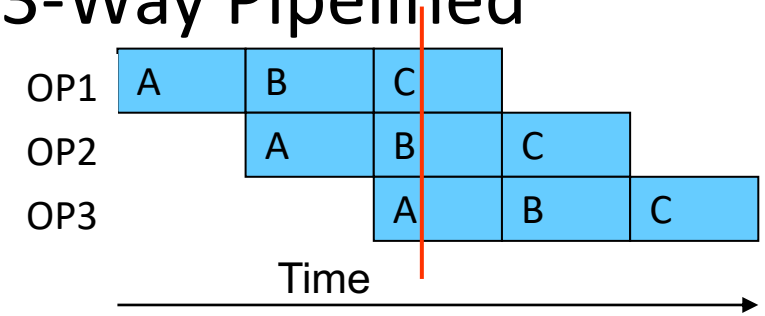
Pipeline Diagrams

- Unpipelined



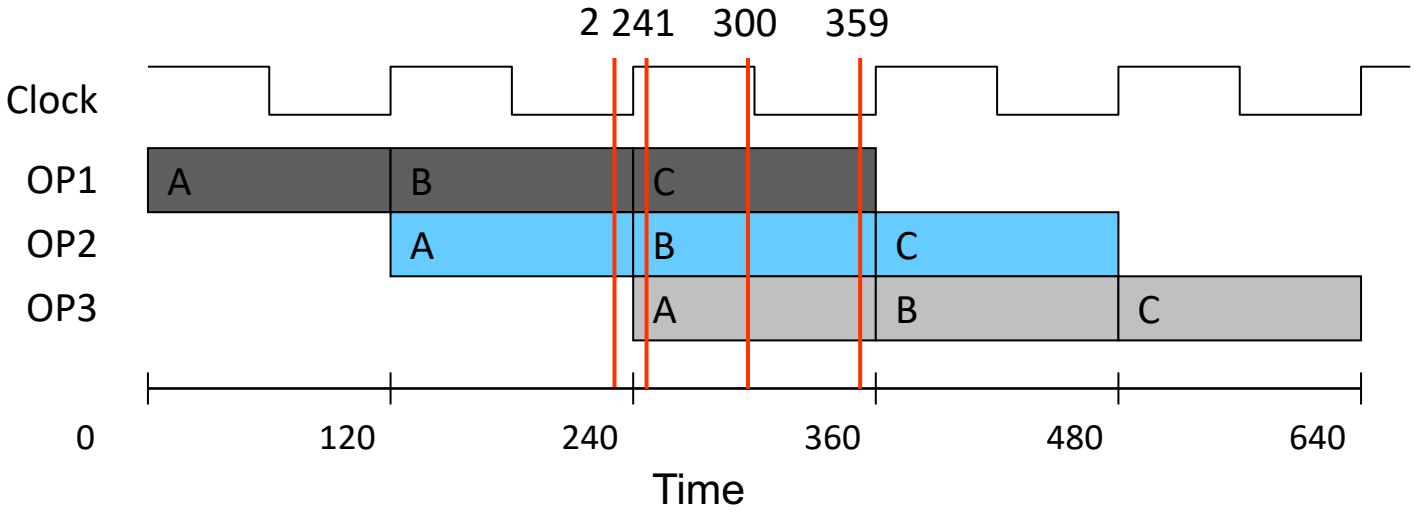
  - Cannot start new operation until previous one completes

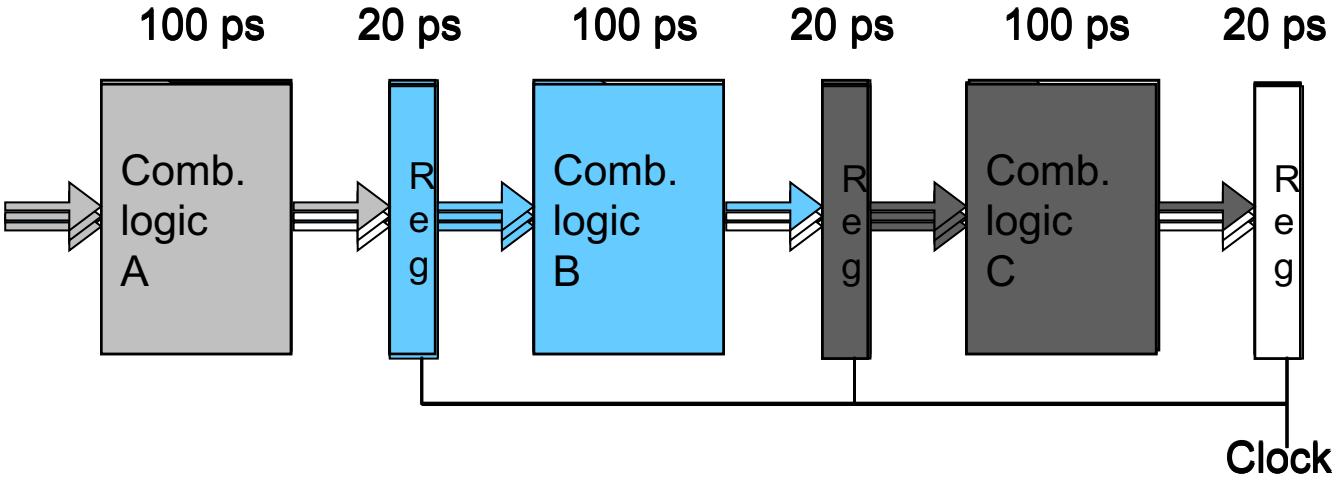
- 3-Way Pipelined



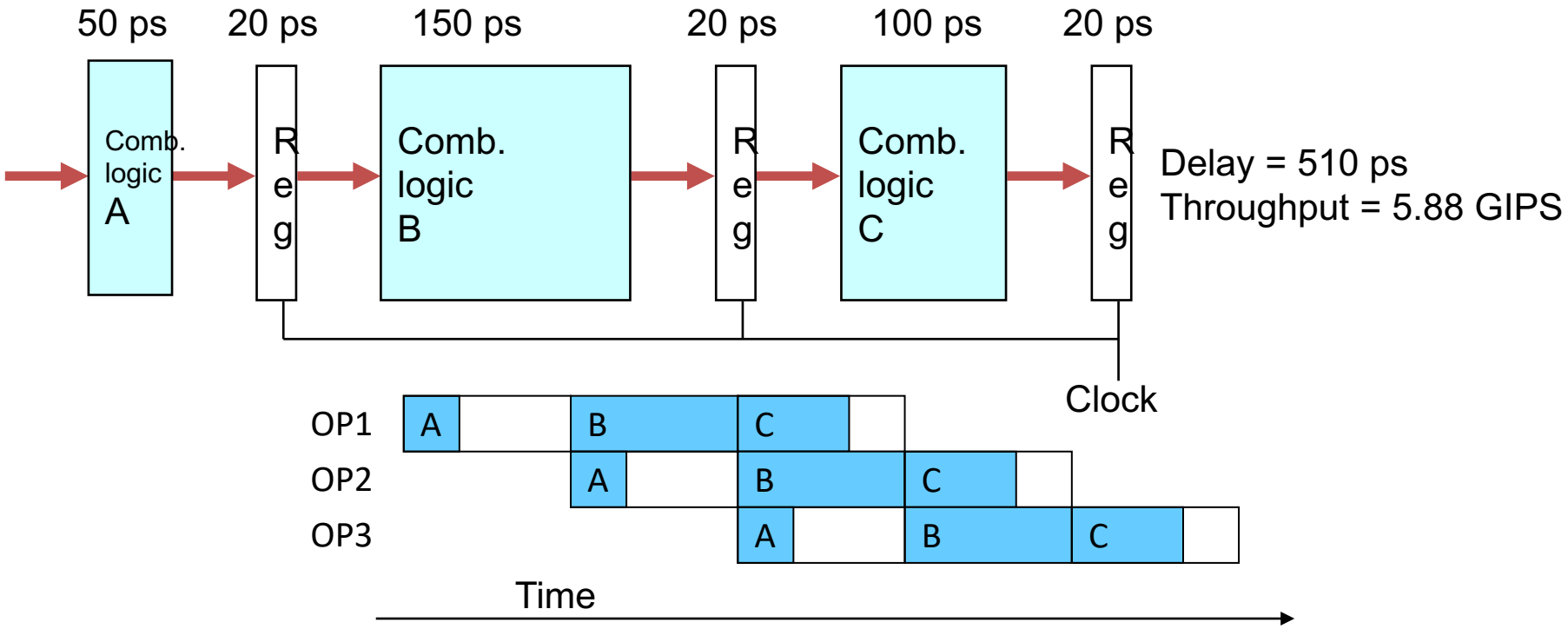
  - Up to 3 operations in process simultaneously

Operating a Pipeline





Limitations: Nonuniform Delays



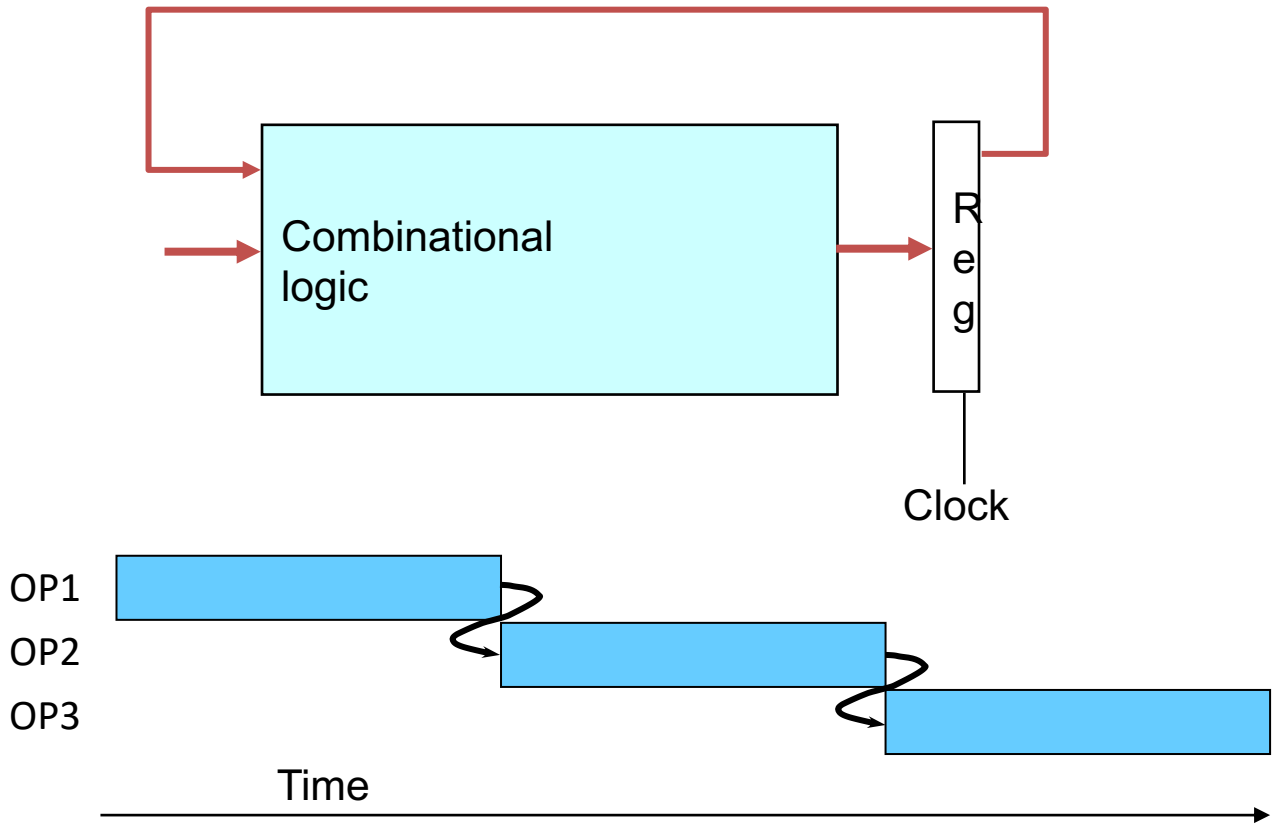
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Hazards

- There are three classes of hazards:
- **Structural Hazards.** They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- **Data Hazards.** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- **Control Hazards.** They arise from the pipelining of branches and other instructions that change the PC.

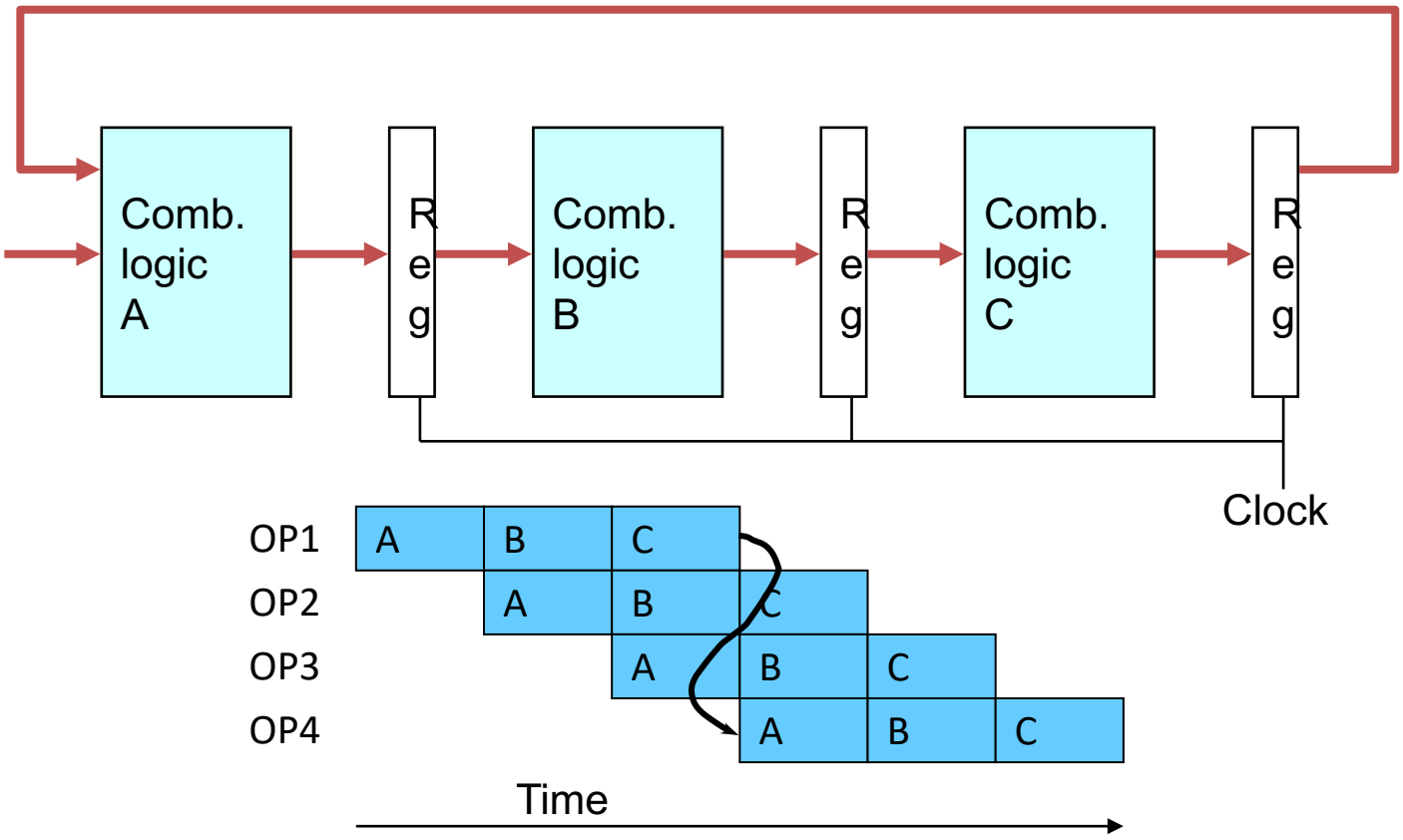
http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/hazards.html

Data Dependencies



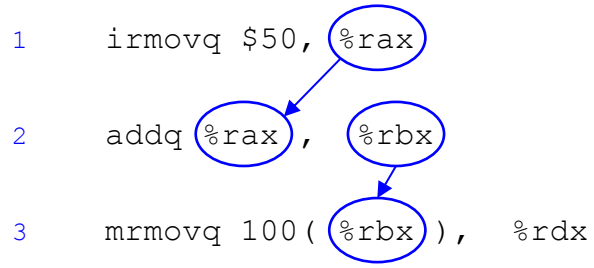
- System
  - Each operation depends on result from preceding one

Data Hazards



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

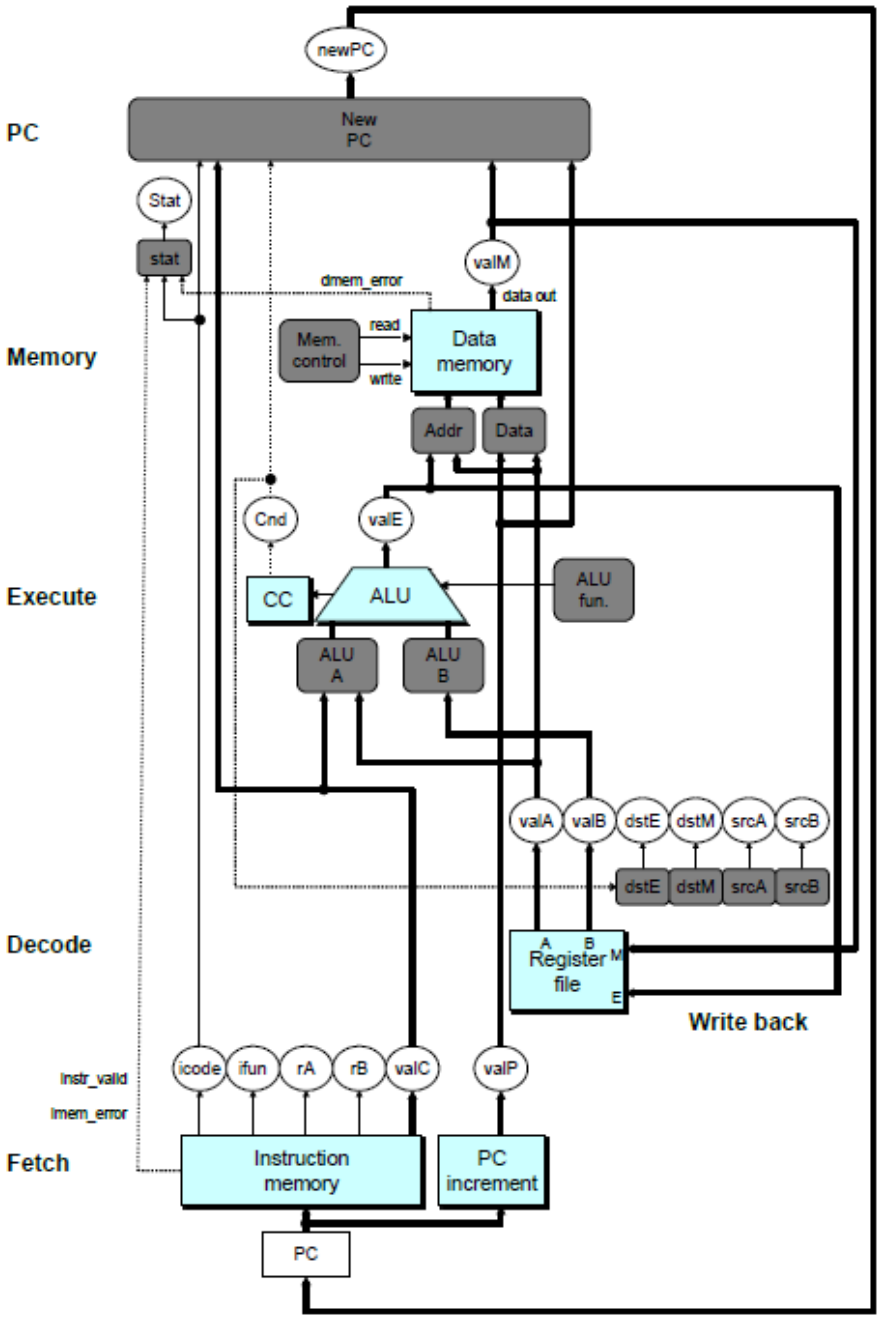
Data Dependencies in Processors



- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

SEQ Hardware

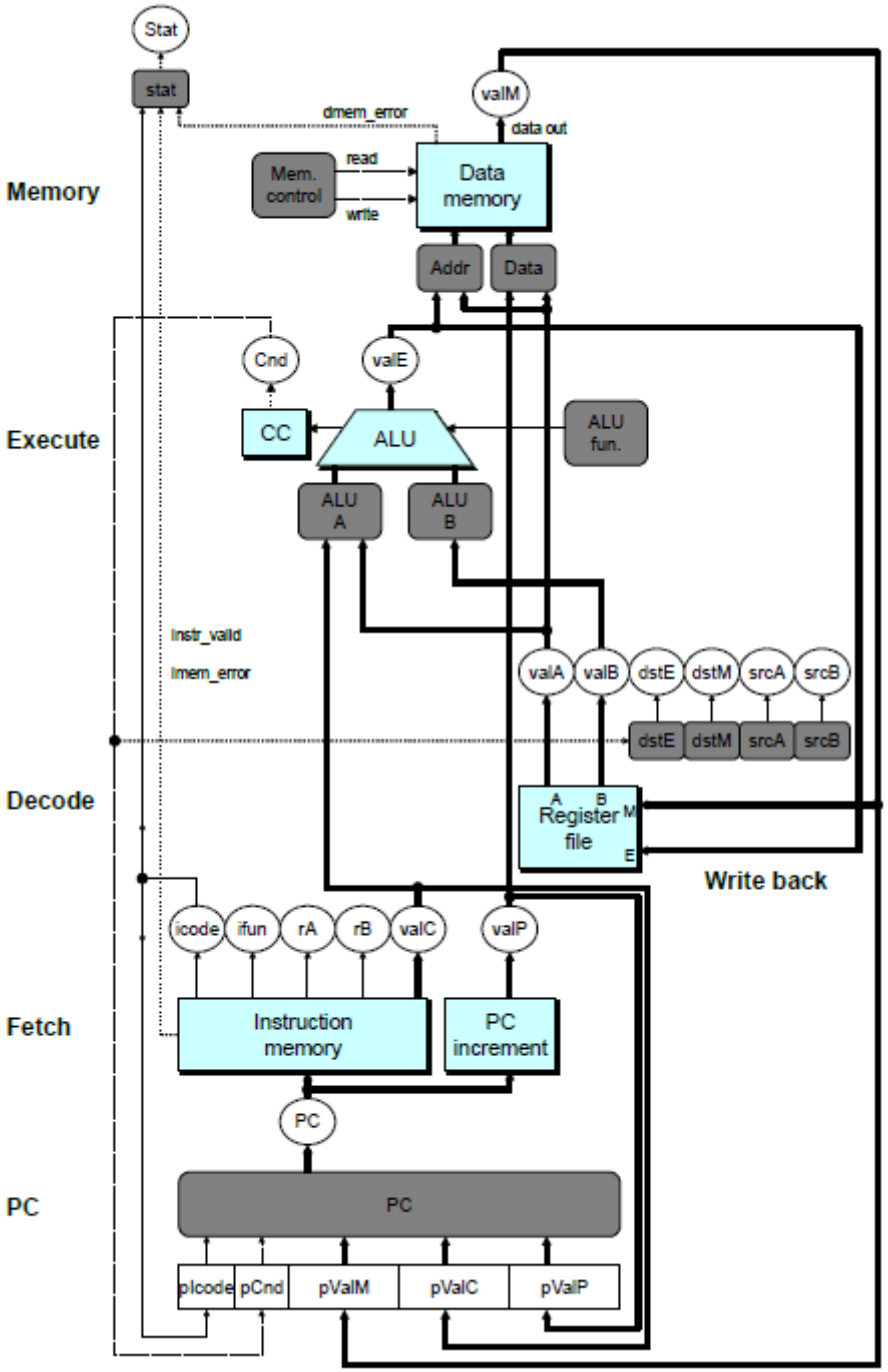
- Stages occur in sequence
- One operation in process at a time



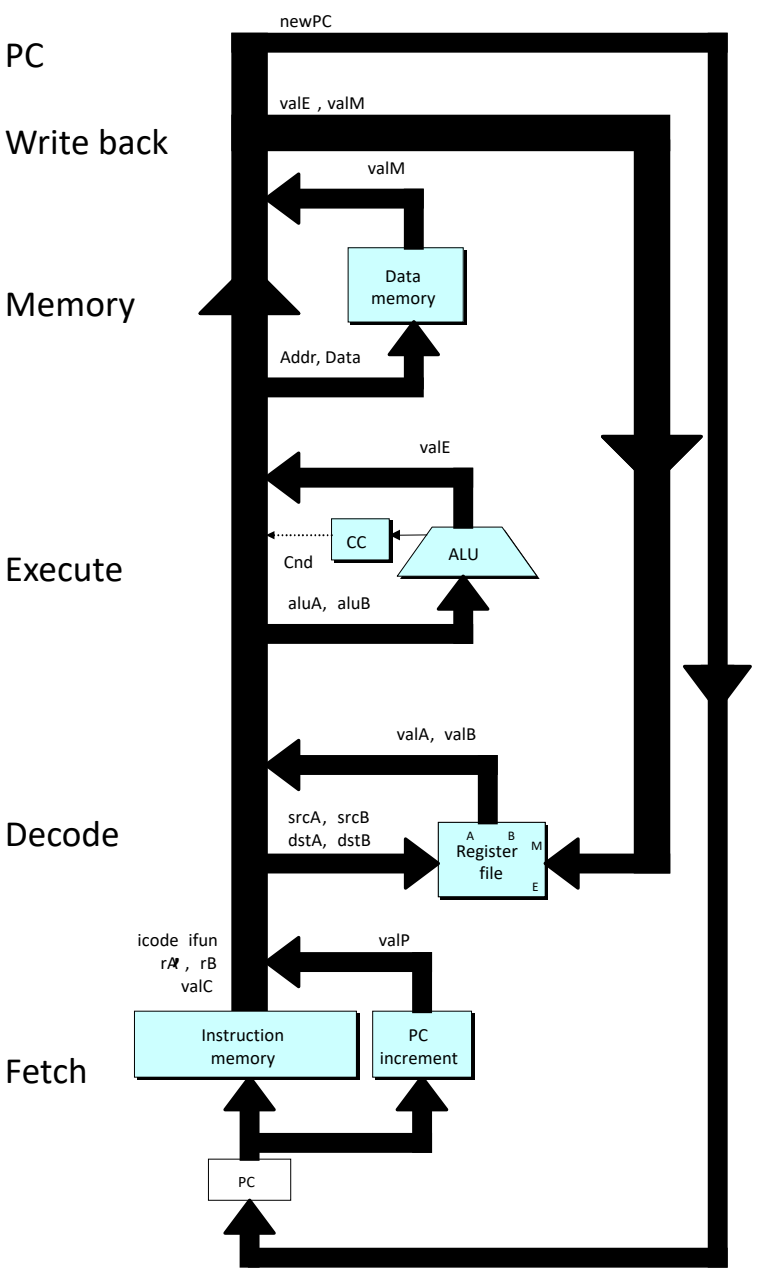
SEQ+ Hardware

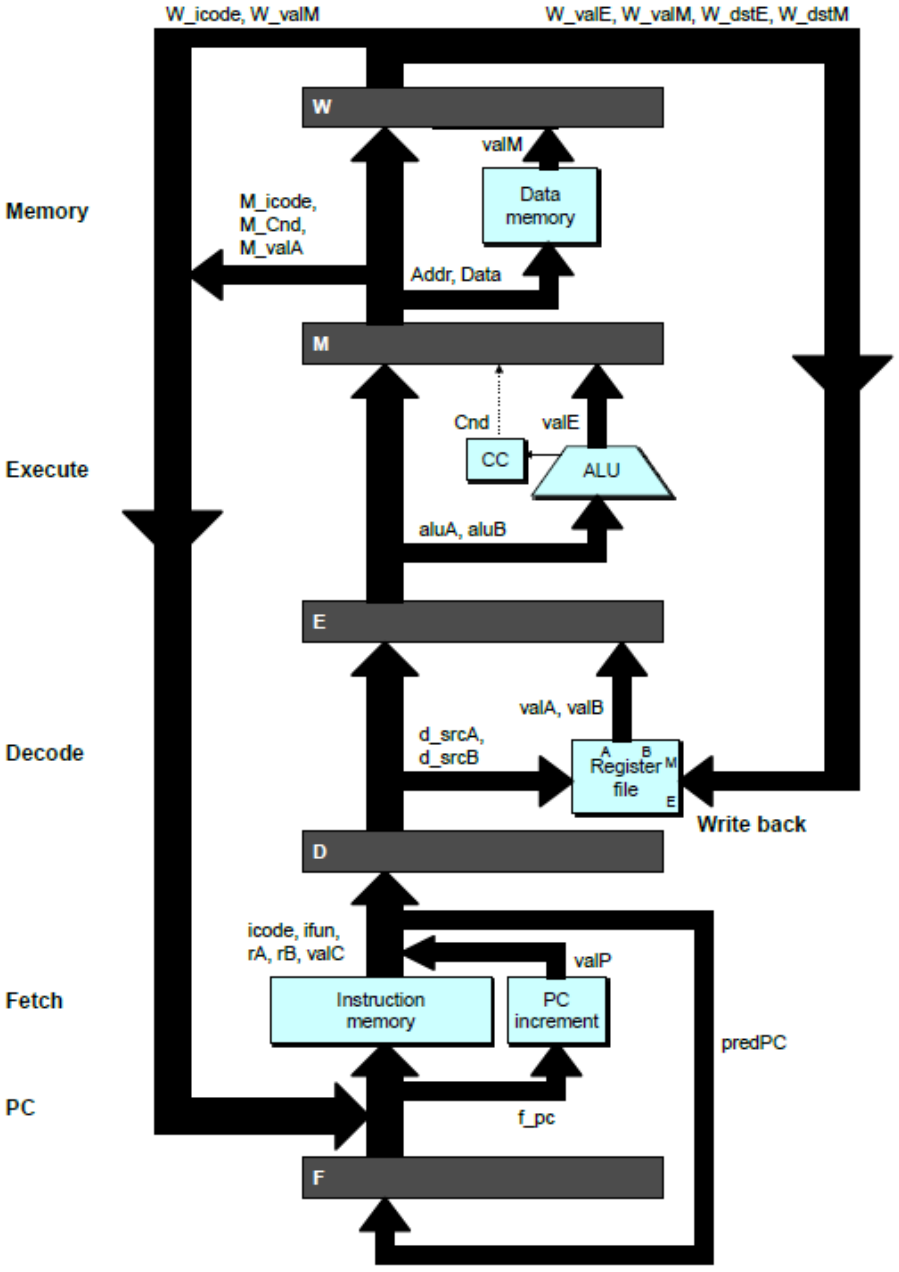
- Still sequential implementation
- Reorder PC stage to put at beginning

- PC Stage
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction
- Processor State
  - PC is no longer stored in register
  - But, can determine PC based on other stored information



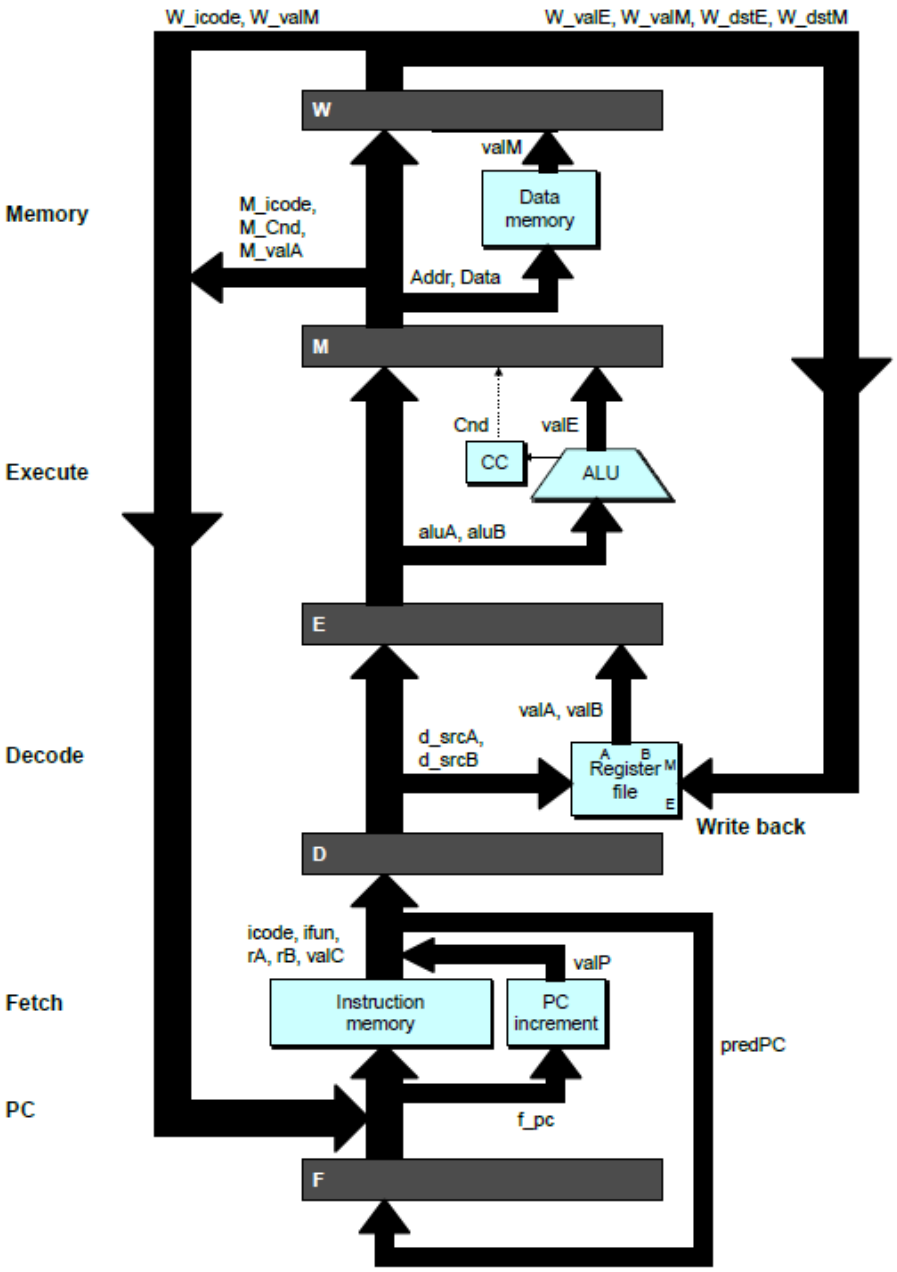
Adding Pipeline Registers



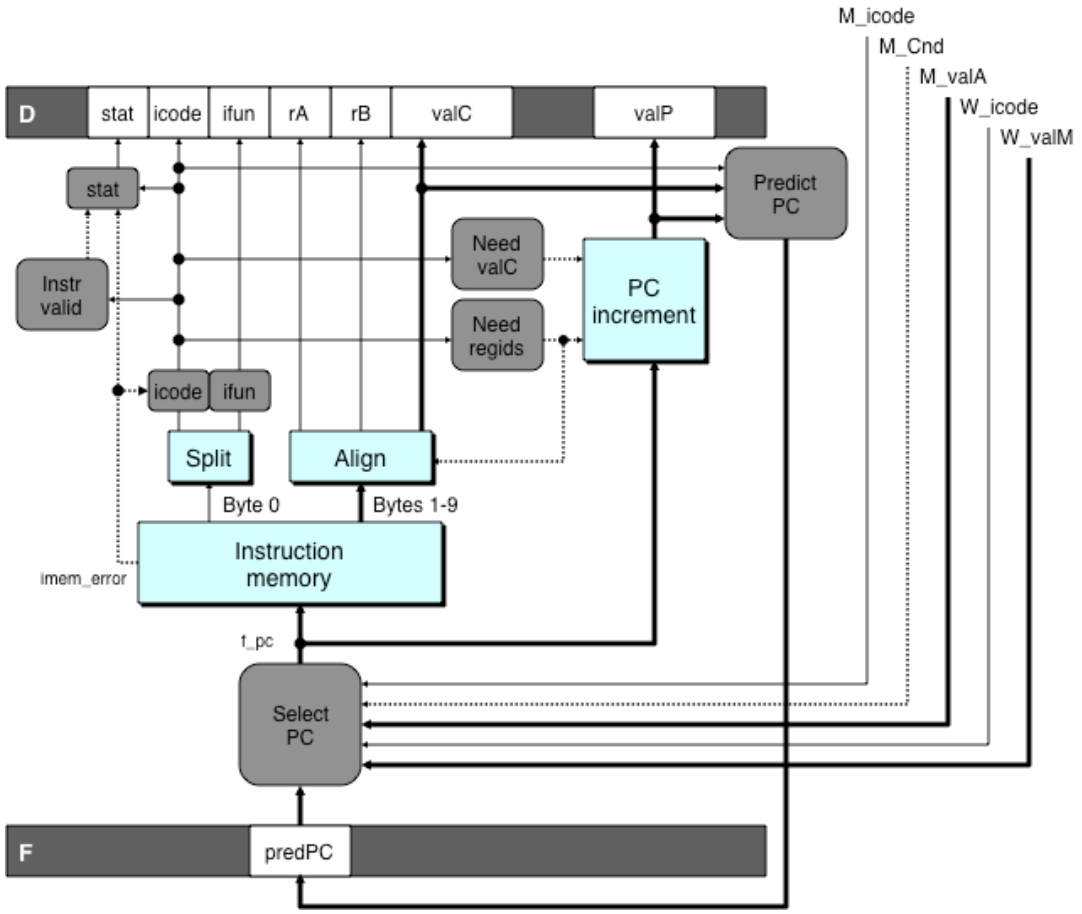


Pipeline Stages

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file



Predicting the PC

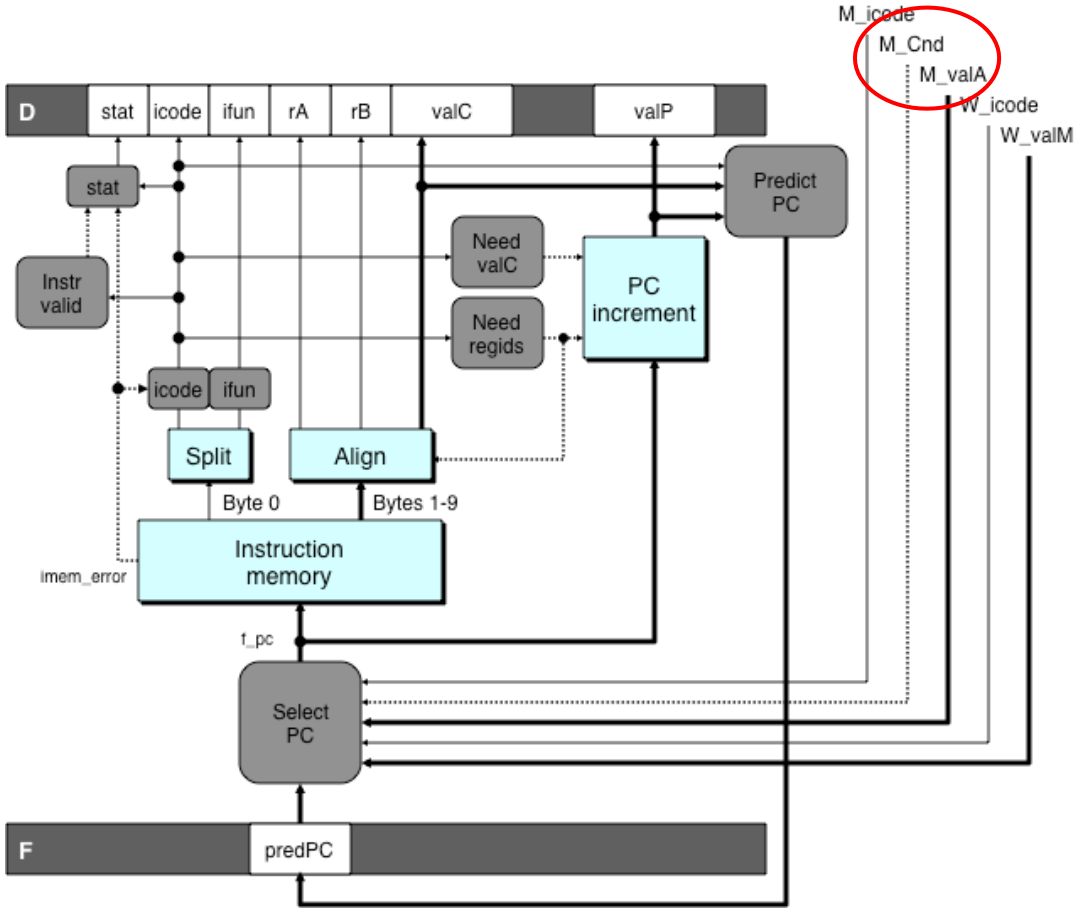


- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

Our Prediction Strategy

- Instructions that Don’t Transfer Control
  - Predict next PC to be valP
  - Always reliable
- Call and Unconditional Jumps
  - Predict next PC to be valC (destination)
  - Always reliable
- Conditional Jumps
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken
    - Typically right 60% of time
- Return Instruction
  - Don’t try to predict

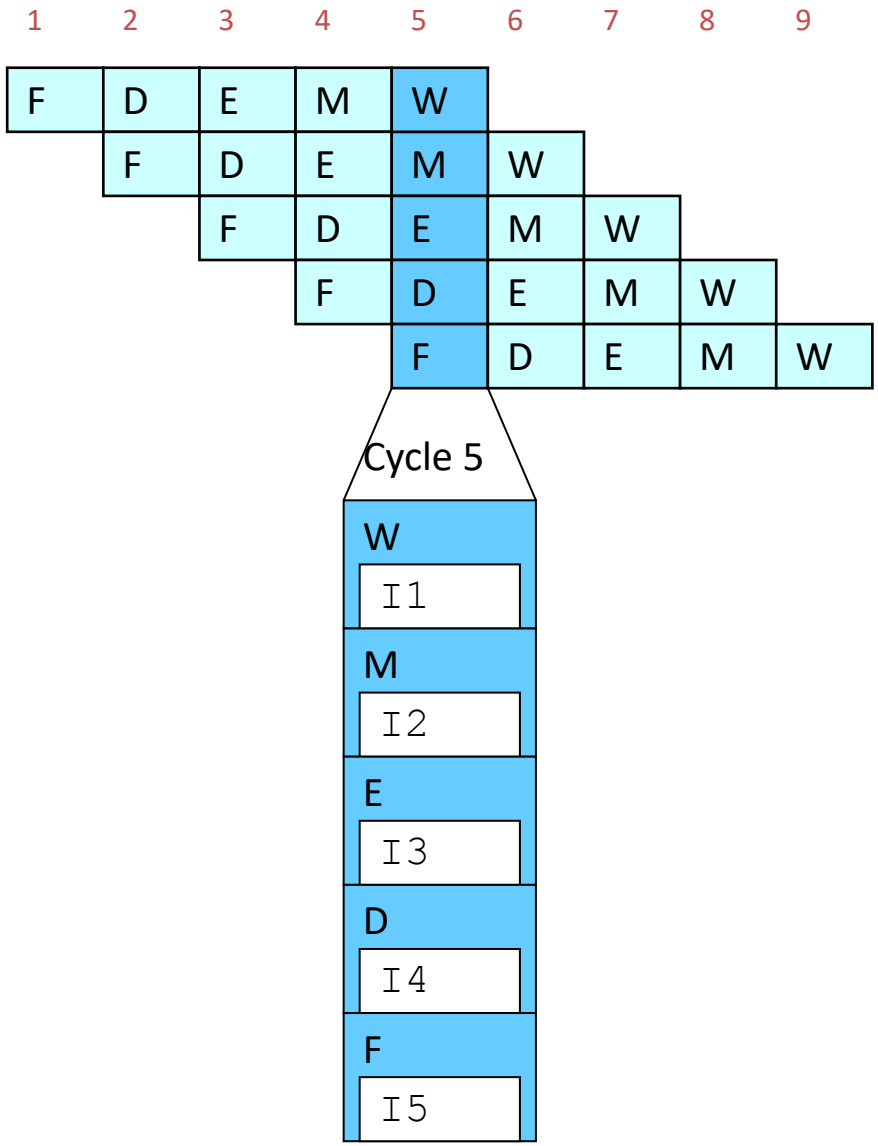
Recovering from PC Misprediction



- Mispredicted Jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from valA (value M_valA)
- Return Instruction
  - Will get return PC when `ret` reaches write-back stage (W_valM)

Pipeline Demonstration

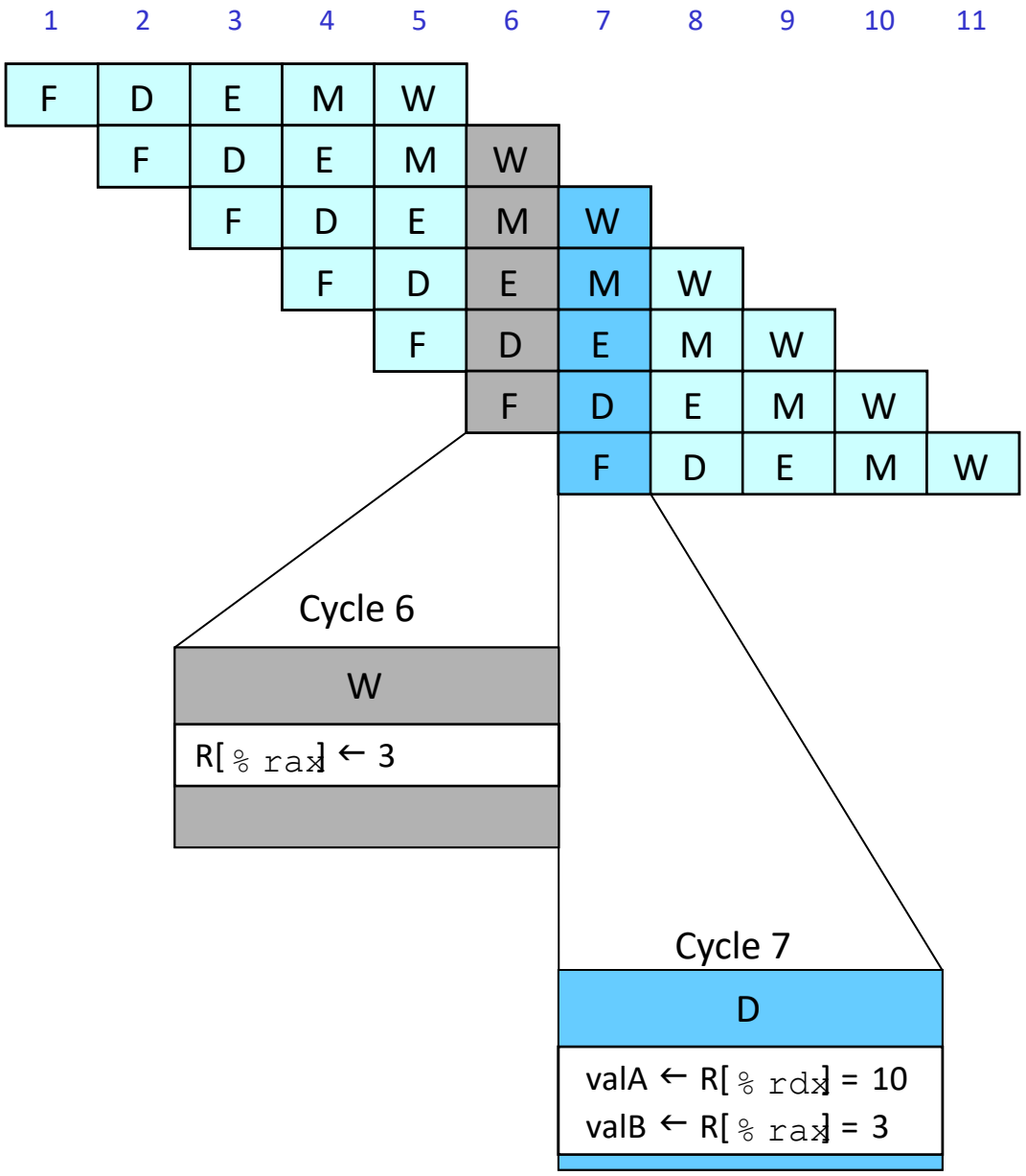
```
irmovq    $1,%rax   #I1
irmovq    $2,%rcx   #I2
irmovq    $3,%rdx   #I3
irmovq    $4,%rbx   #I4
halt                #I5
```



- File: `demo-basic.ys`

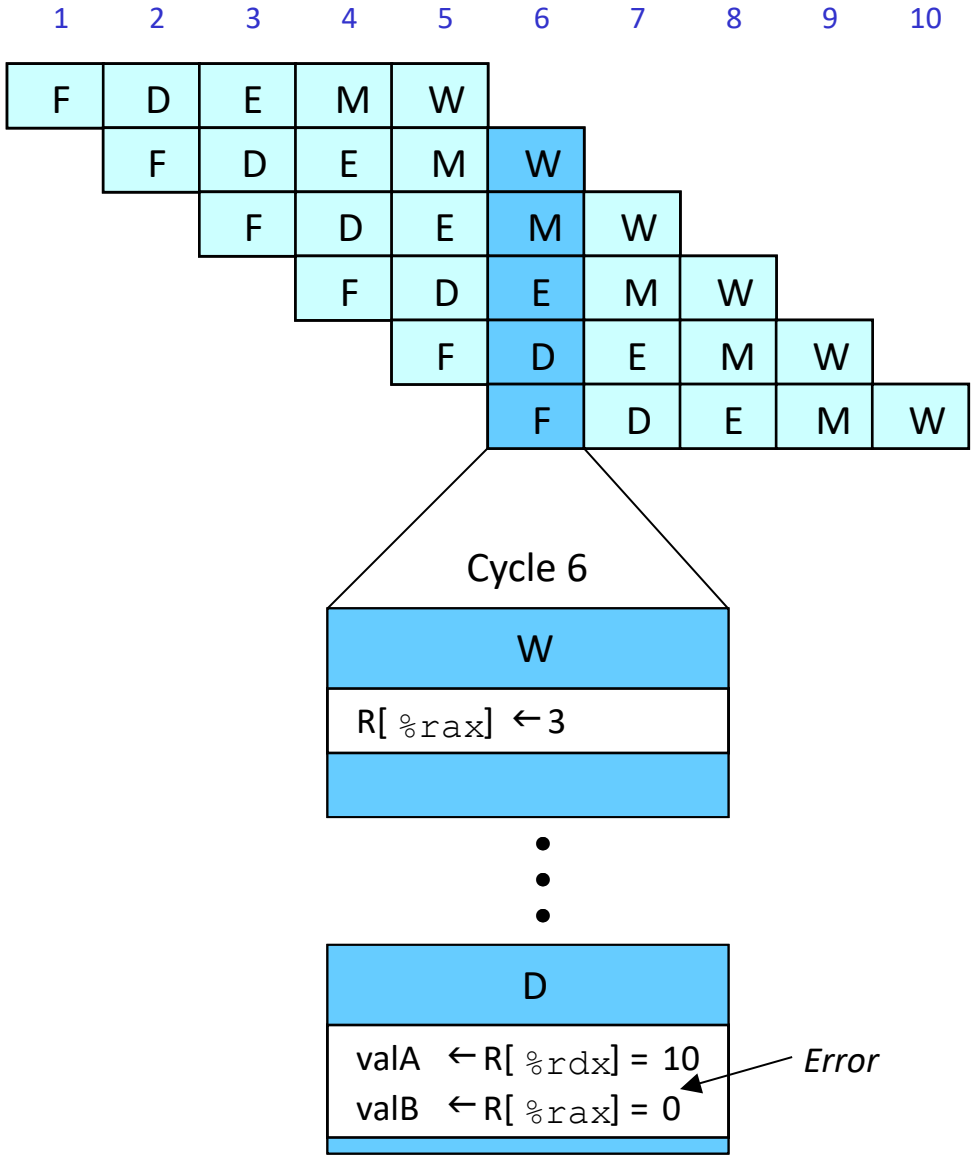
Data Dependencies: 3 Nop's

```
# demo-h3.ys
0x000:  irmovq $10,%rdx
 0x00a: irmovq $3,%rax
 0x014: nop
 0x015: nop
 0x016: nop
 0x017: addq %rdx,%rax
 0x019: halt
```



Data Dependencies: 2 Nop's

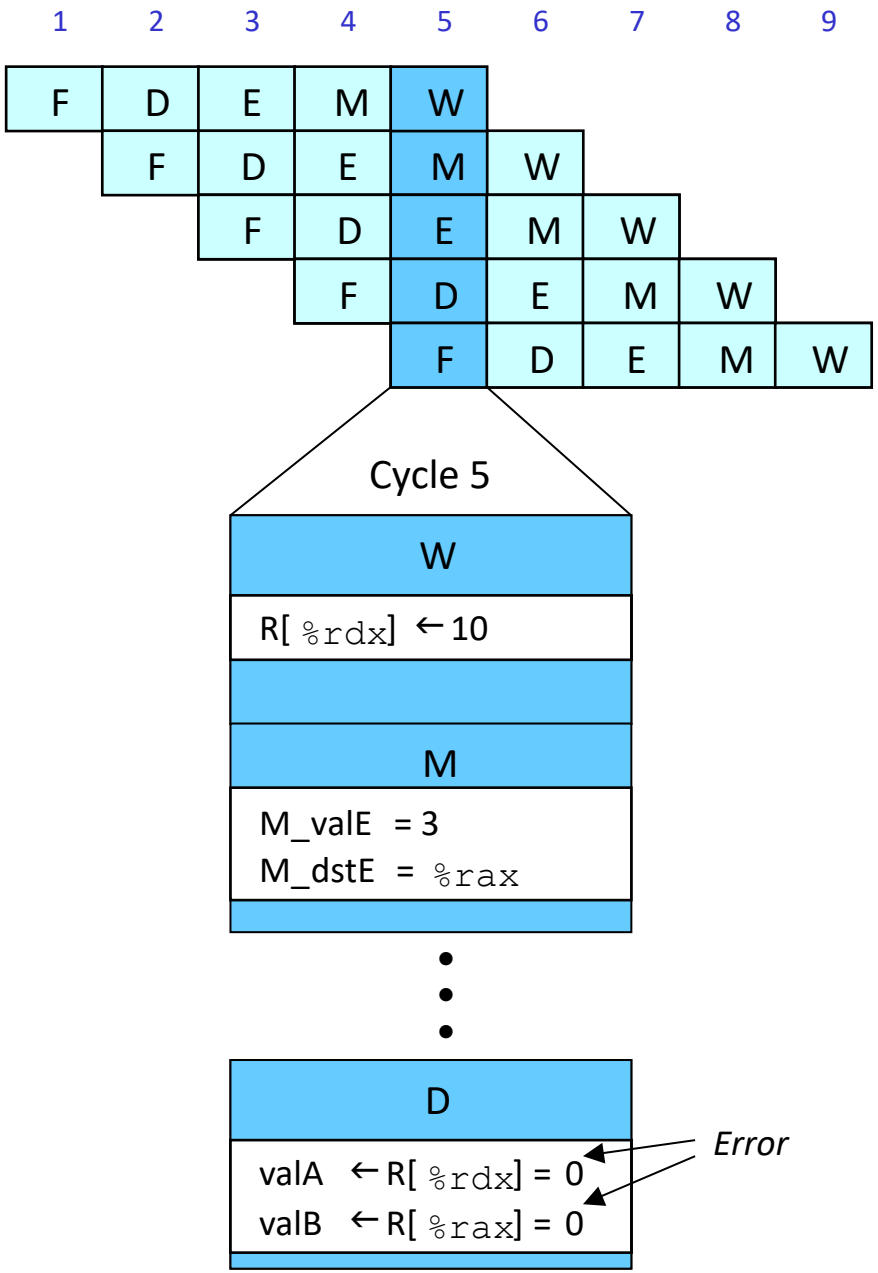
```
# demo-h2.ys
0x000:  irmovq $10,%rdx
 0x00a: irmovq  $3,%rax
 0x014: nop
 0x015: nop
 0x016: addq %rdx,%rax
0x018: halt
```



Data Dependencies: 1 Nop

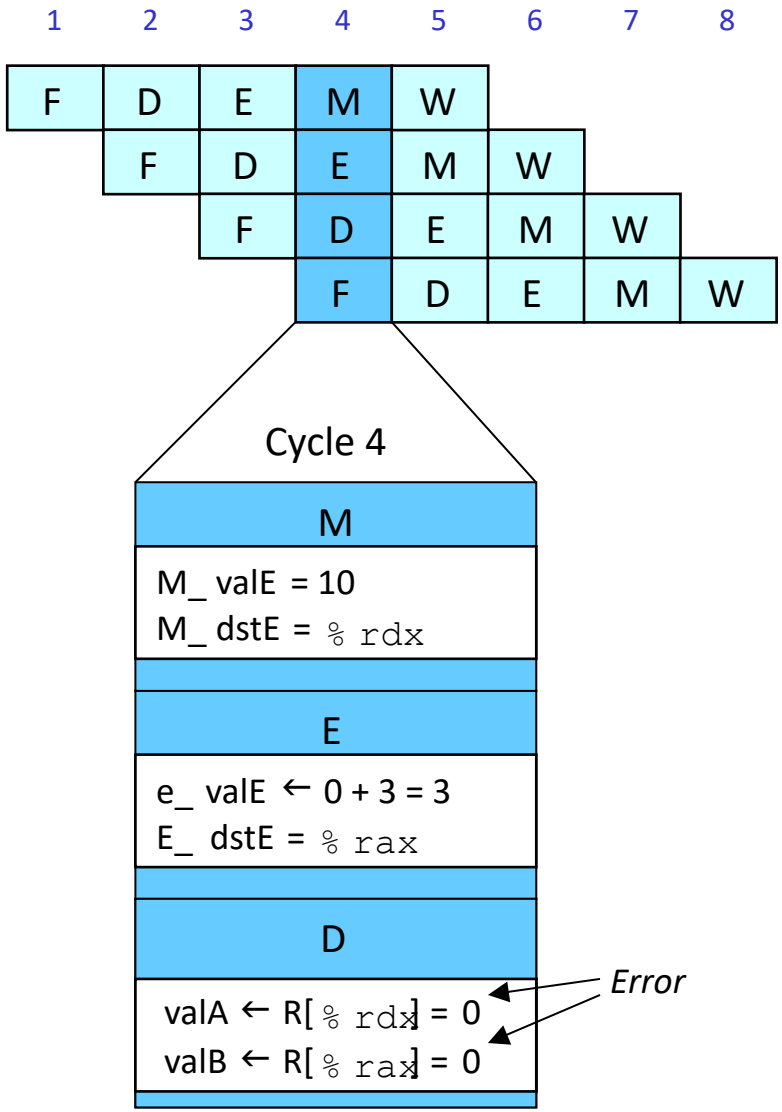
```
# demo-h1.ys

0x000:  irmovq $10,%rdx
 0x00a: irmovq  $3,%rax
 0x014: nop
 0x015: addq %rdx,%rax
0x017: halt
```



Data Dependencies: No Nop

```
# demo-h0.ys
0x000:  irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Random-Access Memory (RAM)

- Key features
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a cell (one bit per cell).
  - Multiple RAM chips form a memory.

- RAM comes in two varieties:
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

- **SRAM**
  - More complex than Dram, require 4 to 6 transistor per bit
  - More expensive (each cell is more complex)
  - Faster than DRAM
  - Used in small – fast memories
  - Stores each bit in bistable memory cell(each cell is implemented with a six-transistor ciruit)
- **DRAM**
  - Simple, Slower, Cheaper
  - Needs to be refreshed constantly
  - Used in main memories and frame buffers associated with the graphic cards
  - Stores each bit as charge on a capacitor

Nonvolatile Memories

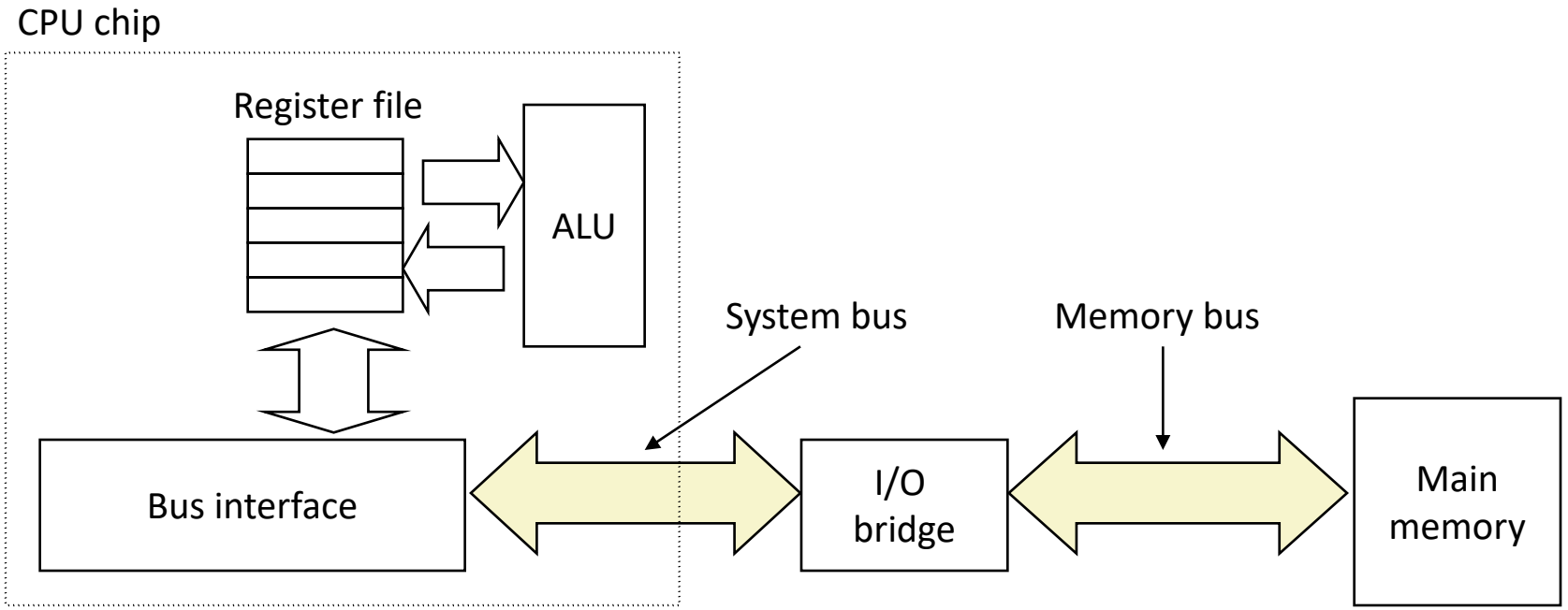
- DRAM and SRAM are **volatile** memories
  - Lose information if powered off.
- **Nonvolatile** memories retain value even if powered off
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Erasable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically erasable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasing

Nonvolatile Memories

- Uses for Nonvolatile Memories
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,…)
  - Disk caches

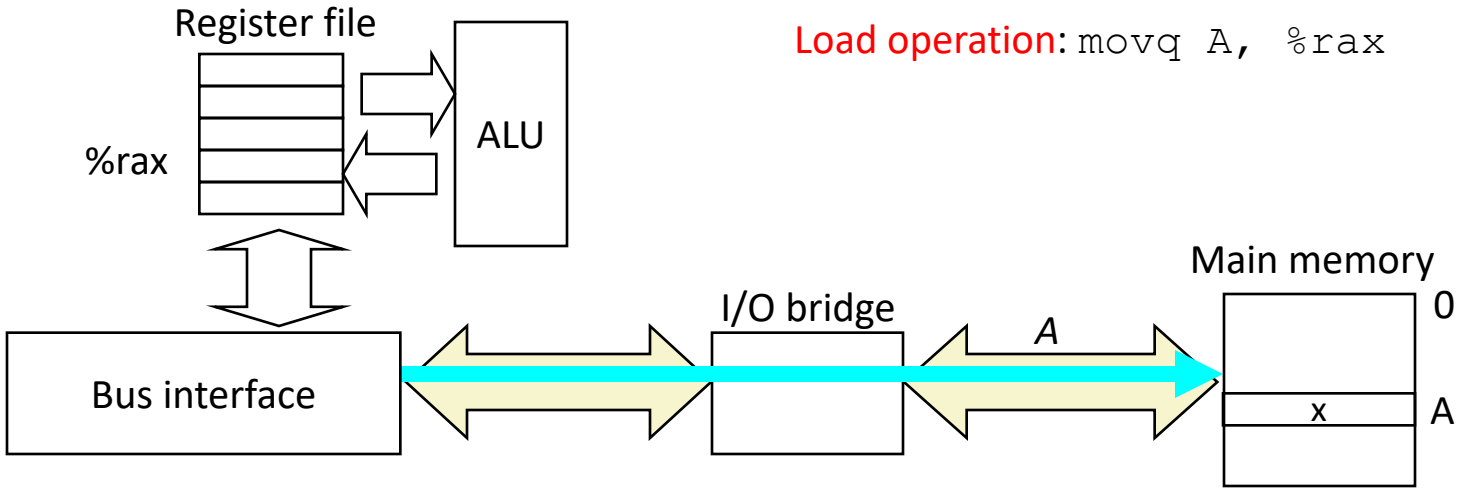
Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



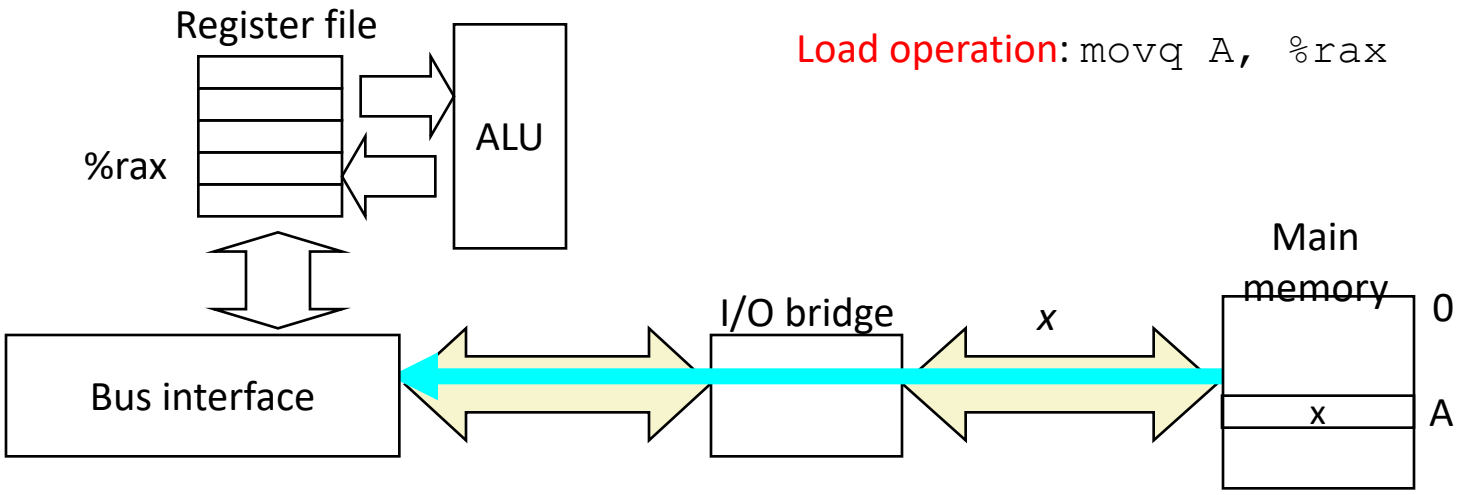
Memory Read Transaction (1)

- CPU places address A on the memory bus.



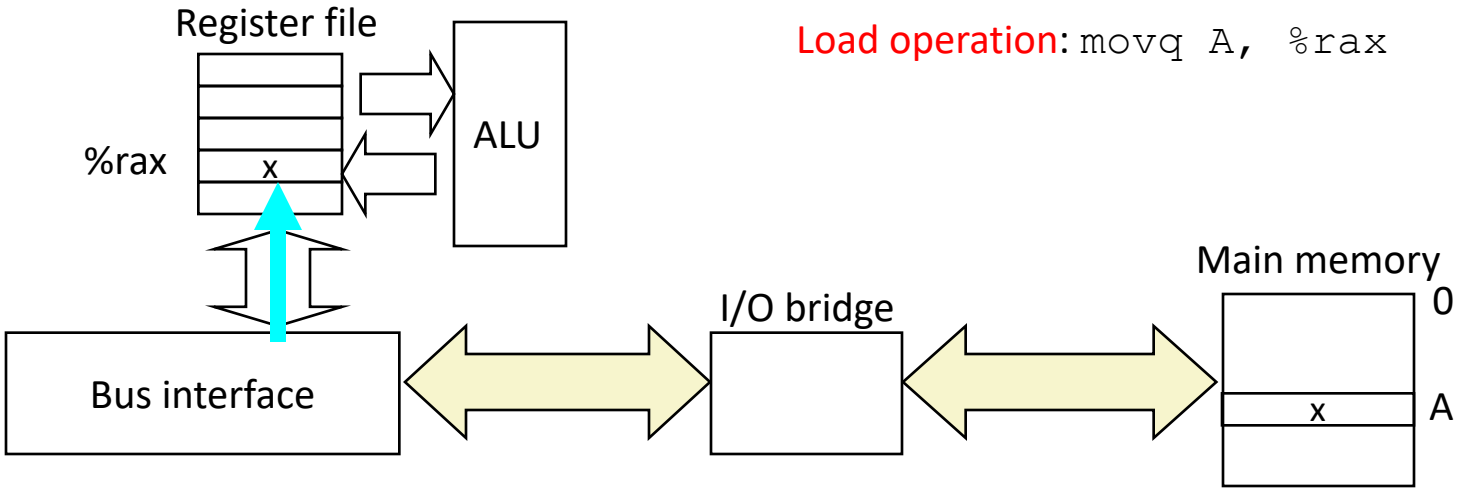
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



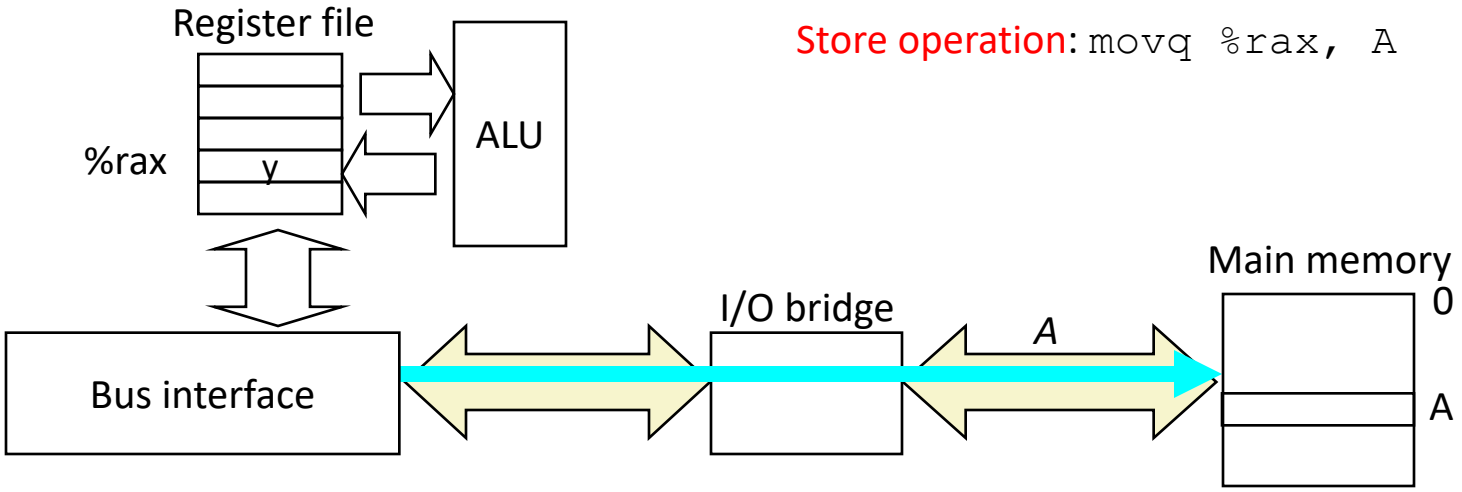
Memory Read Transaction (3)

- CPU read word x from the bus and copies it into register %rax.



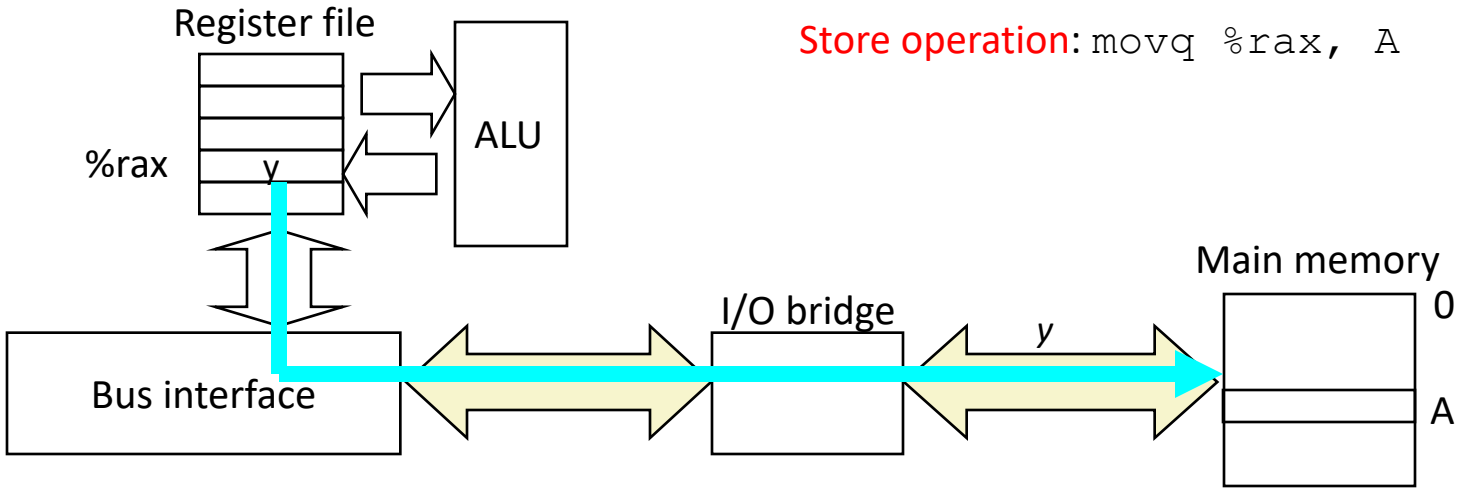
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



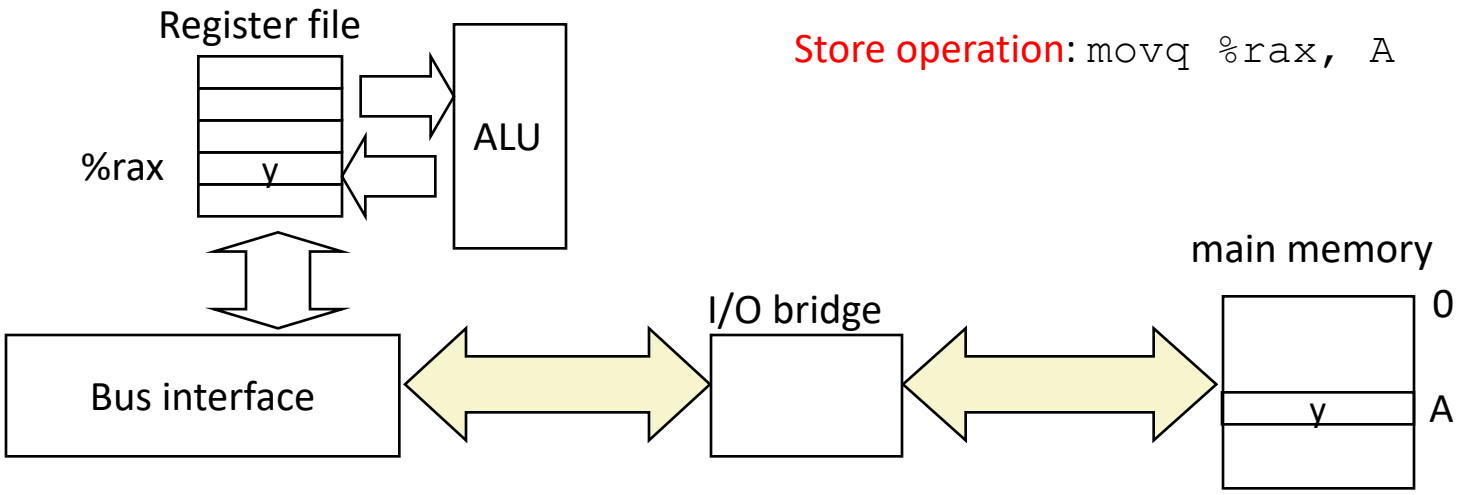
Memory Write Transaction (2)

- CPU places data word y on the bus.



Memory Write Transaction (3)

- Main memory reads data word y from the bus and stores it at address A.



What's Inside A Disk Drive?

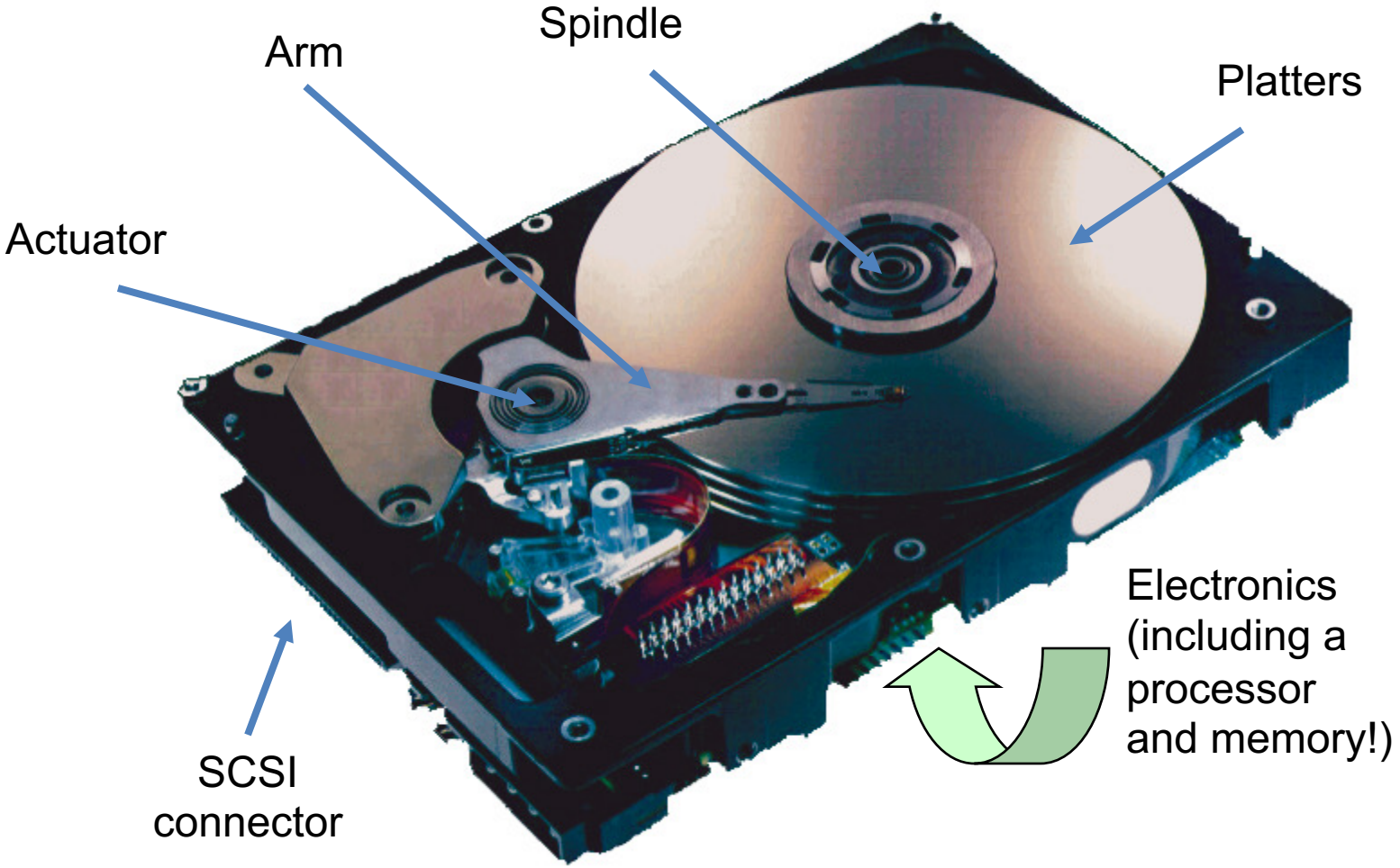
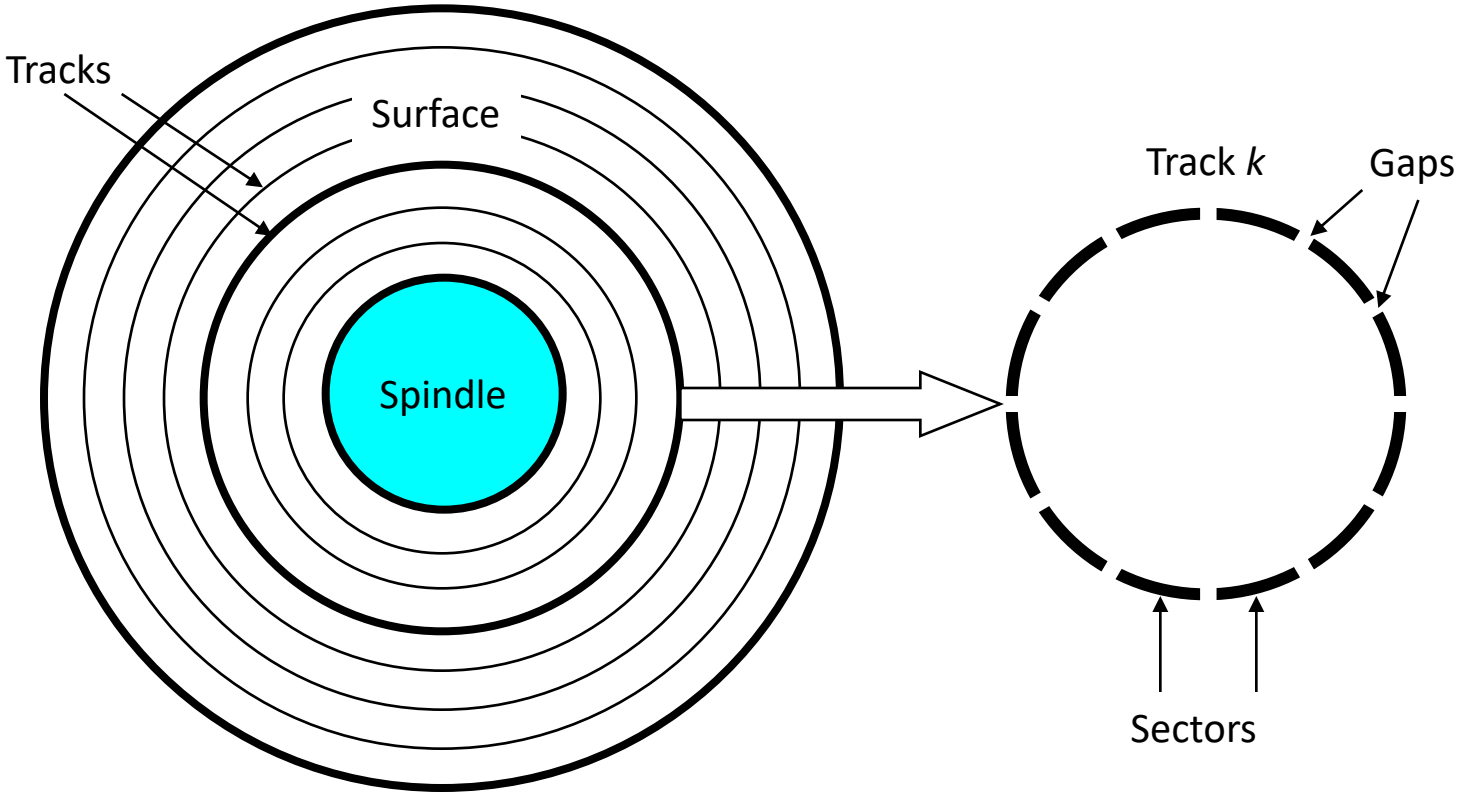


Image courtesy of Seagate Technology

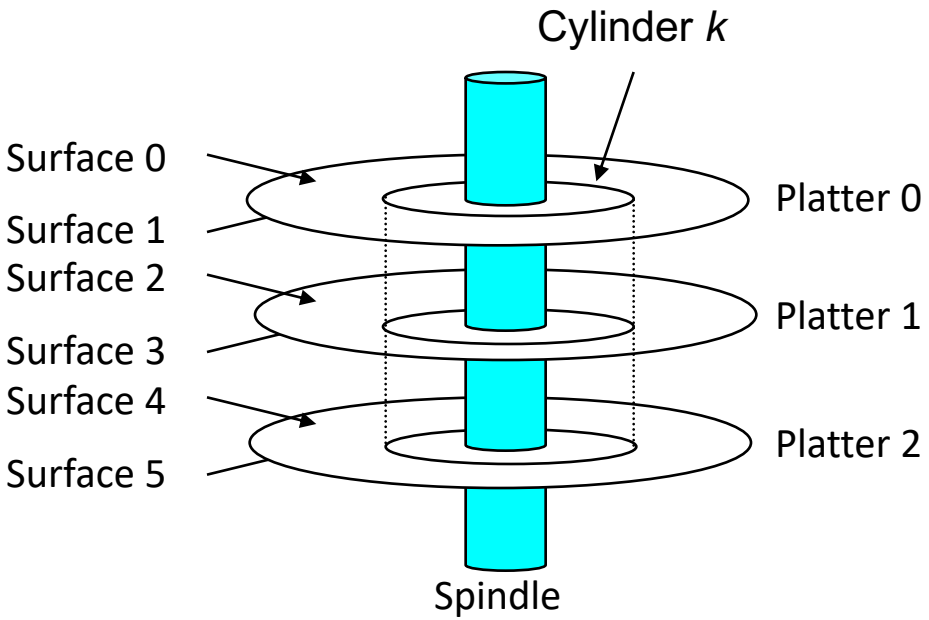
Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.



Disk Geometry (Muliple-Platter View)

- Aligned tracks form a cylinder.

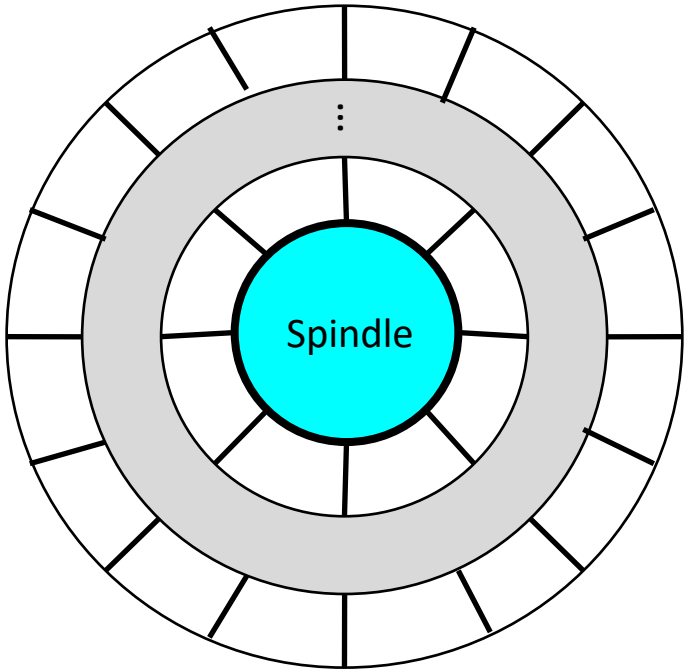


Disk Capacity

- Capacity: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9 Bytes. (actually Terabytes now)
- Capacity is determined by these technology factors:
  - Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - Areal density (bits/in2): product of recording and track density.

Recording zones

- Modern disks partition tracks into disjoint subsets called recording zones
  - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  - Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
  - So we use **average** number of sectors/track when computing capacity.



Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk)

Example:

- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

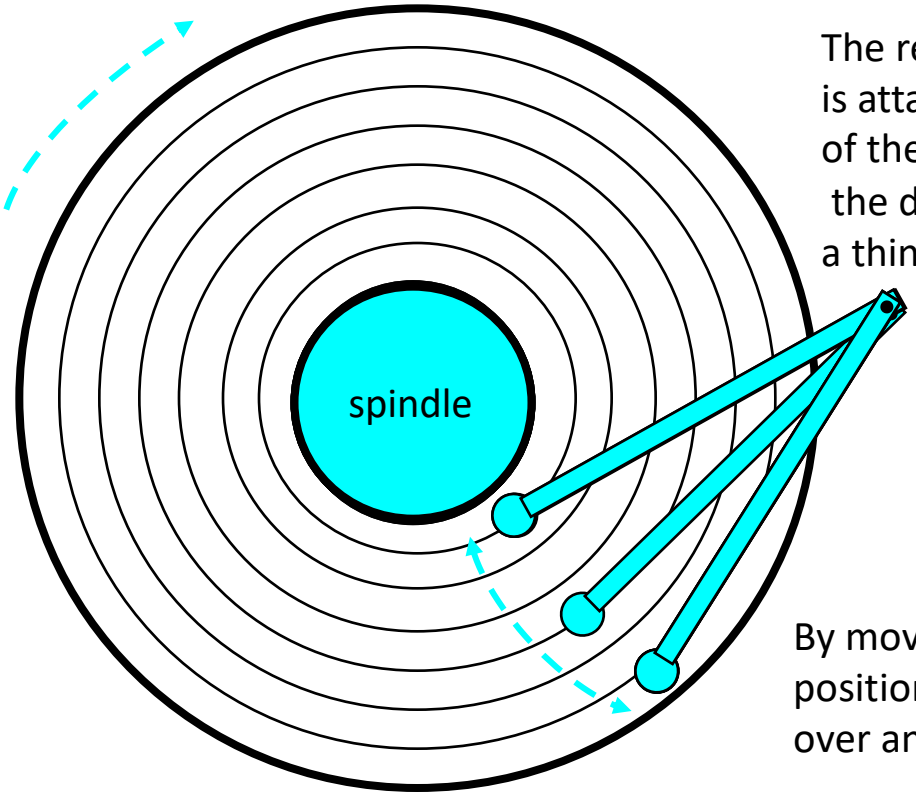
Capacity = 512 x 300 x 20000 x 2 x 5

= 30,720,000,000

= 30.72 GB

Disk Operation (Single-Platter View)

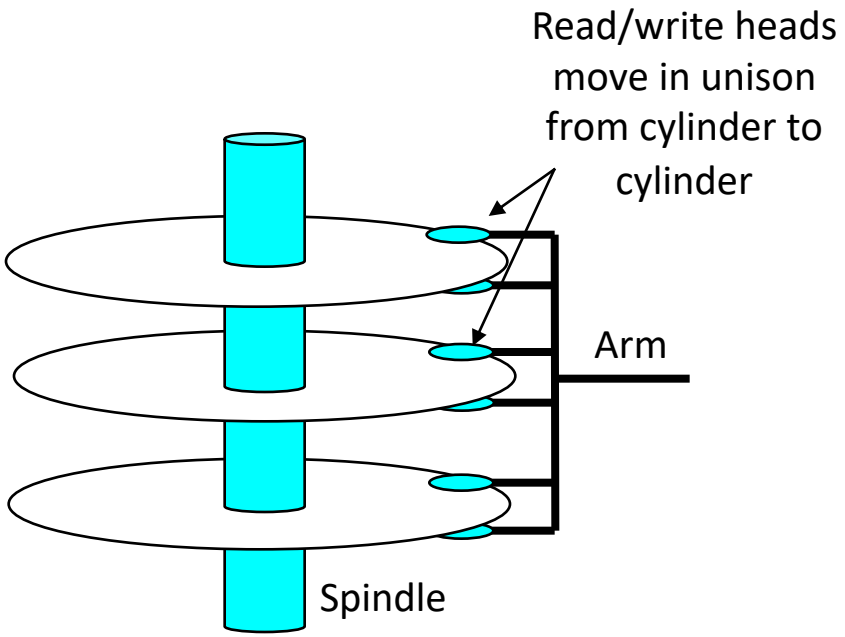
The disk surface spins at a fixed rotational rate



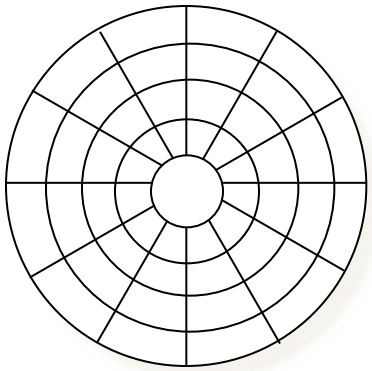
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation (Multi-Platter View)



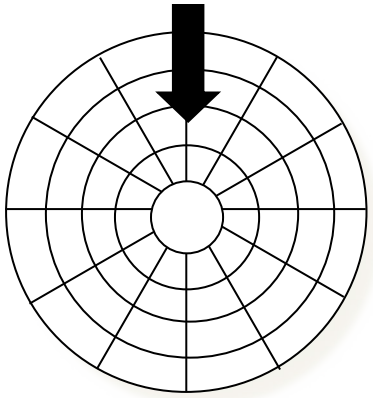
Disk Structure - top view of single platter



Surface organized into tracks

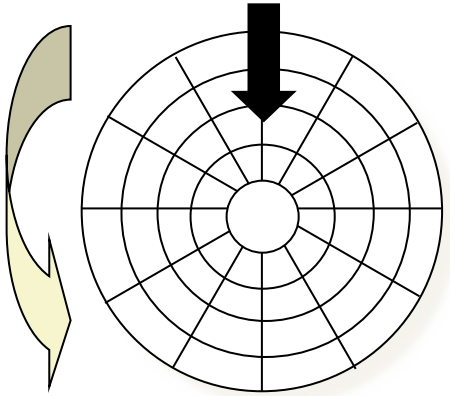
Tracks divided into sectors

Disk Access



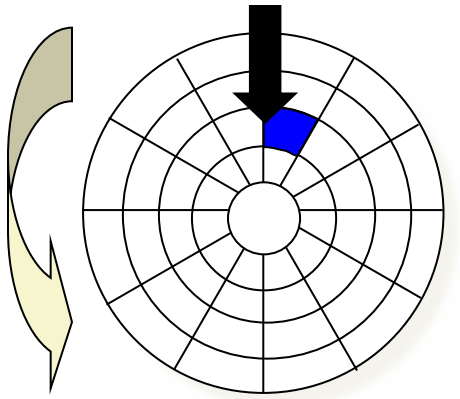
Head in position above a track

Disk Access



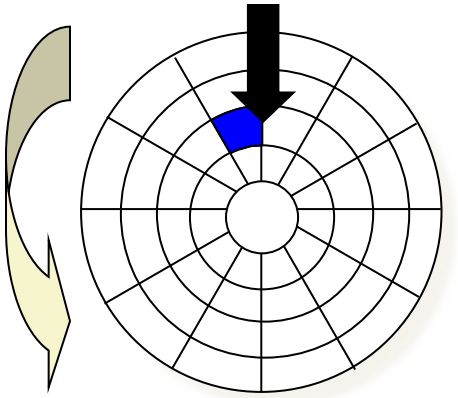
Rotation is counter-clockwise

Disk Access – Read



About to read blue sector

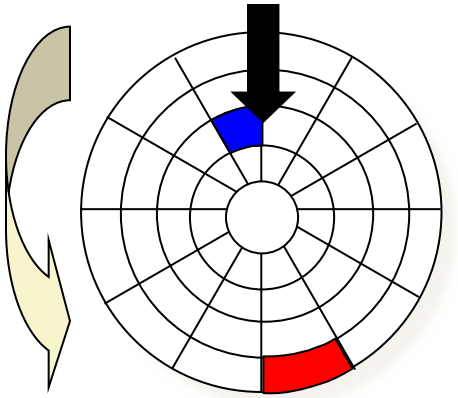
Disk Access – Read



After BLUE read

After reading blue sector

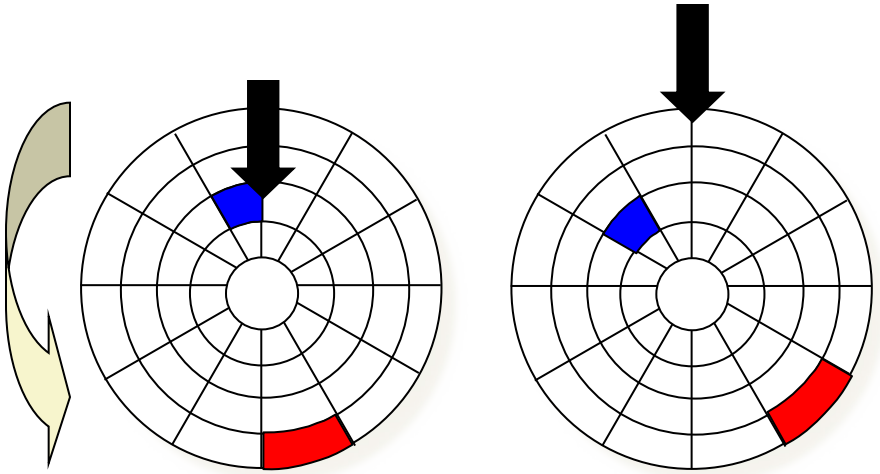
Disk Access – Read



After BLUE read

Red request scheduled next

Disk Access – Seek

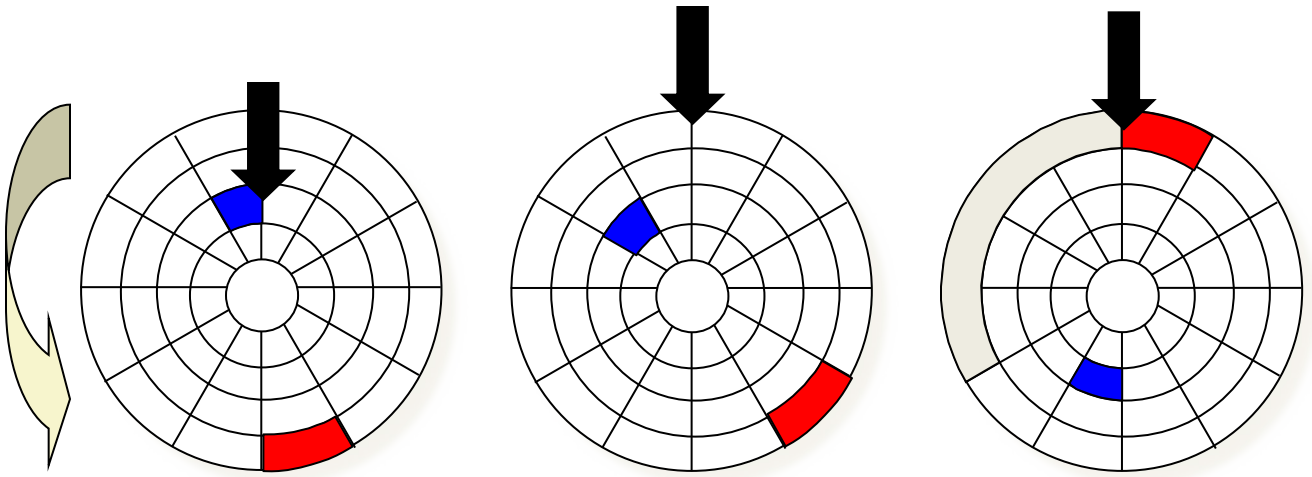


After BLUE read

Seek for RED

Seek to red's track

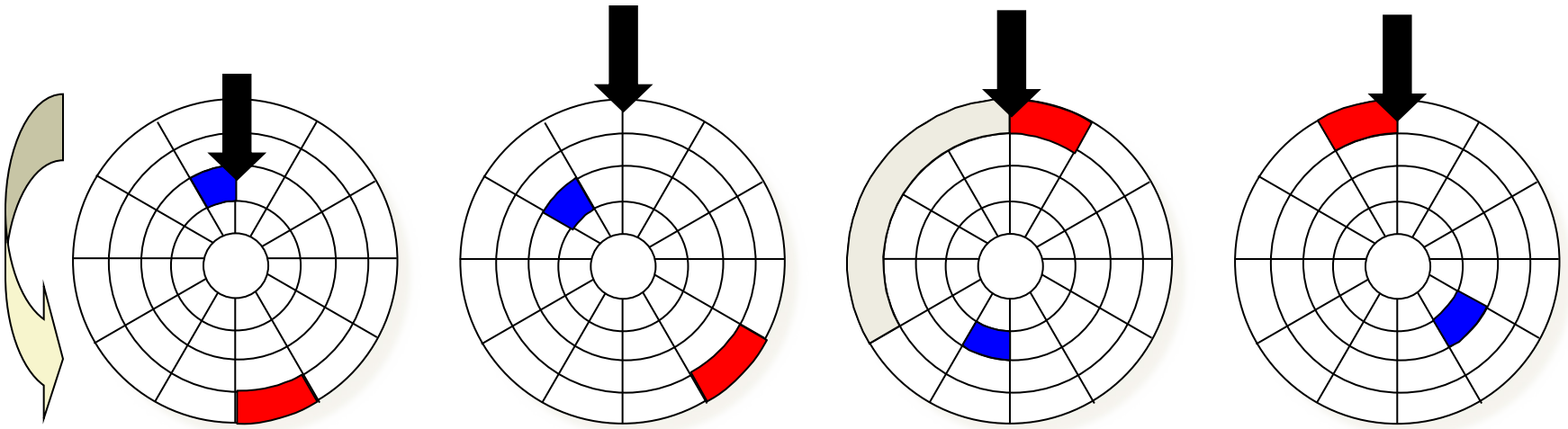
Disk Access – Rotational Latency



After BLUE read Seek for RED Rotational latency

Wait for red sector to rotate around

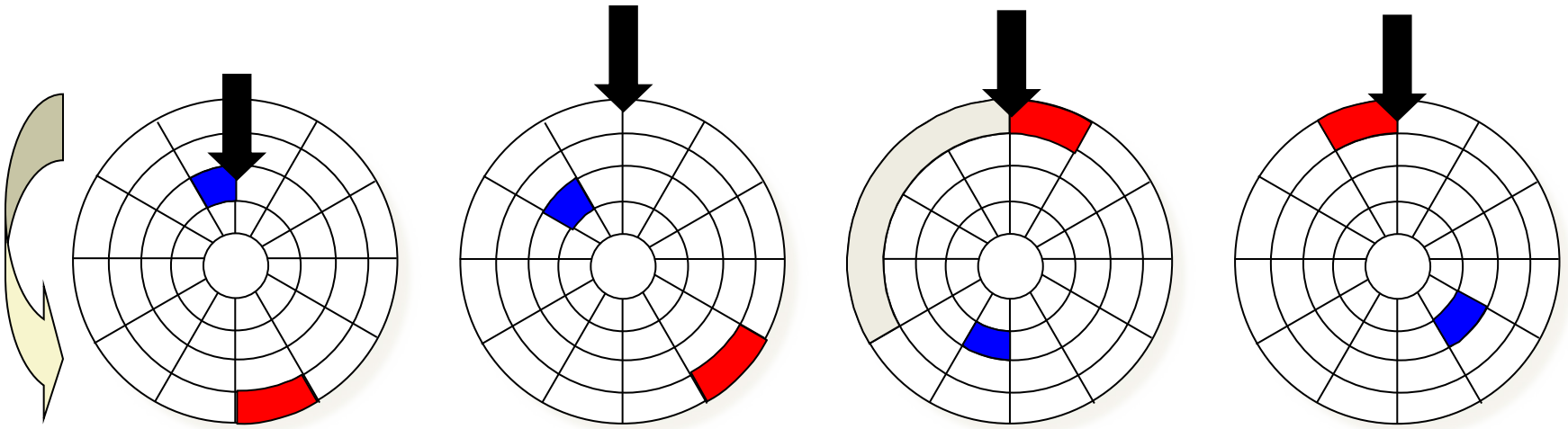
Disk Access – Read



After BLUE read | Seek for RED | Rotational latency | After RED read

Complete read of red

Disk Access – Service Time Components



After BLUE read

Seek for RED

Rotational latency

After RED read

Data transfer

Seek

Rotational latency

Data transfer

Disk Access Time

- **Average time to access some target sector approximated by :**
  - **Taccess = Tavg seek + Tavg rotation + Tavg transfer**
- Seek time (Tavg seek)
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek is 3—9 ms
- Rotational latency (Tavg rotation)
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min
  - Typical Tavg rotation = 7200 RPMs
- Transfer time (Tavg transfer)
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

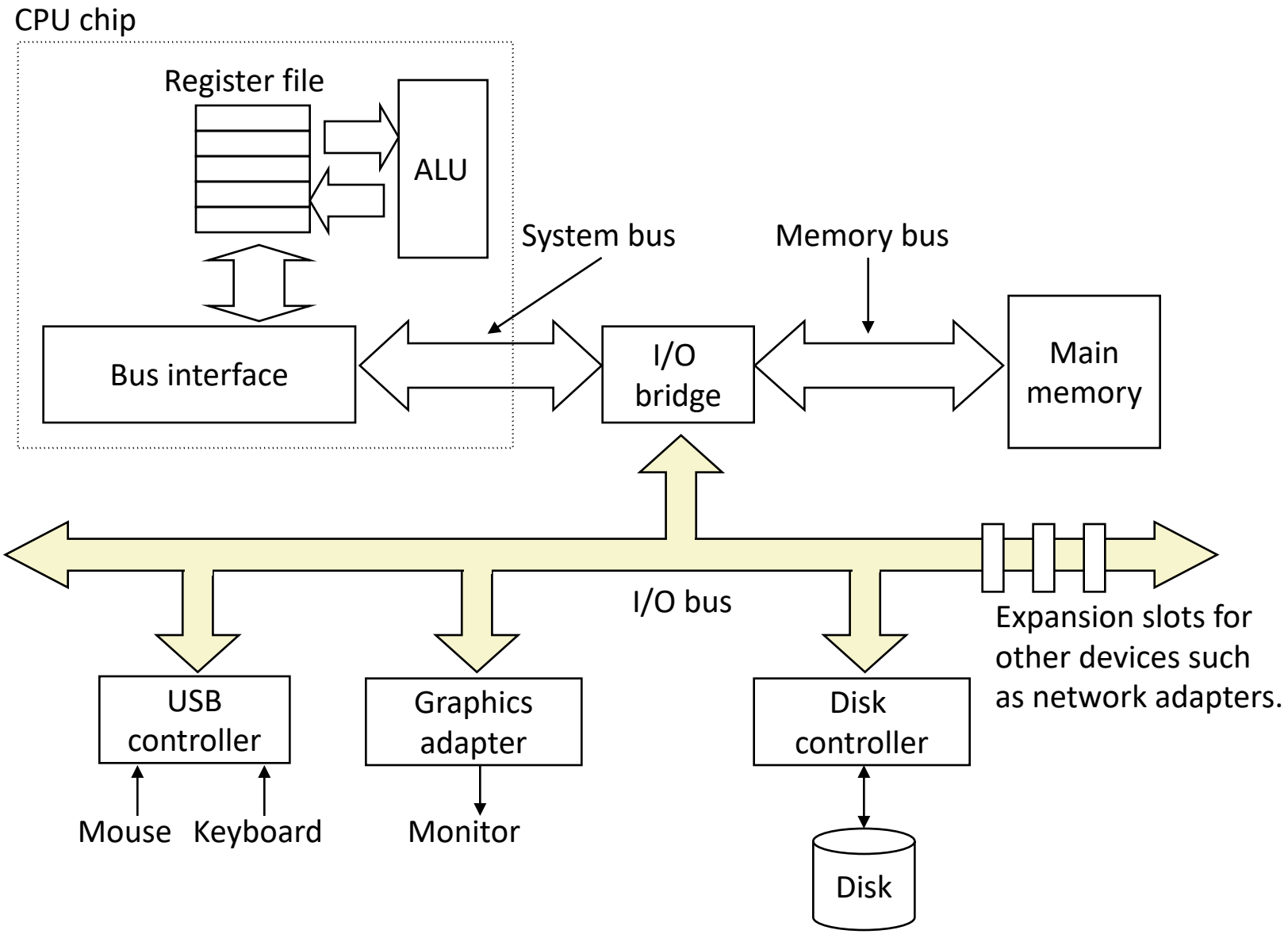
Disk Access Time Example

- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- Derived:
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- **Important points:**
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - **Disk is about 40,000 times slower than SRAM,**
    - **2,500 times slower then DRAM.**

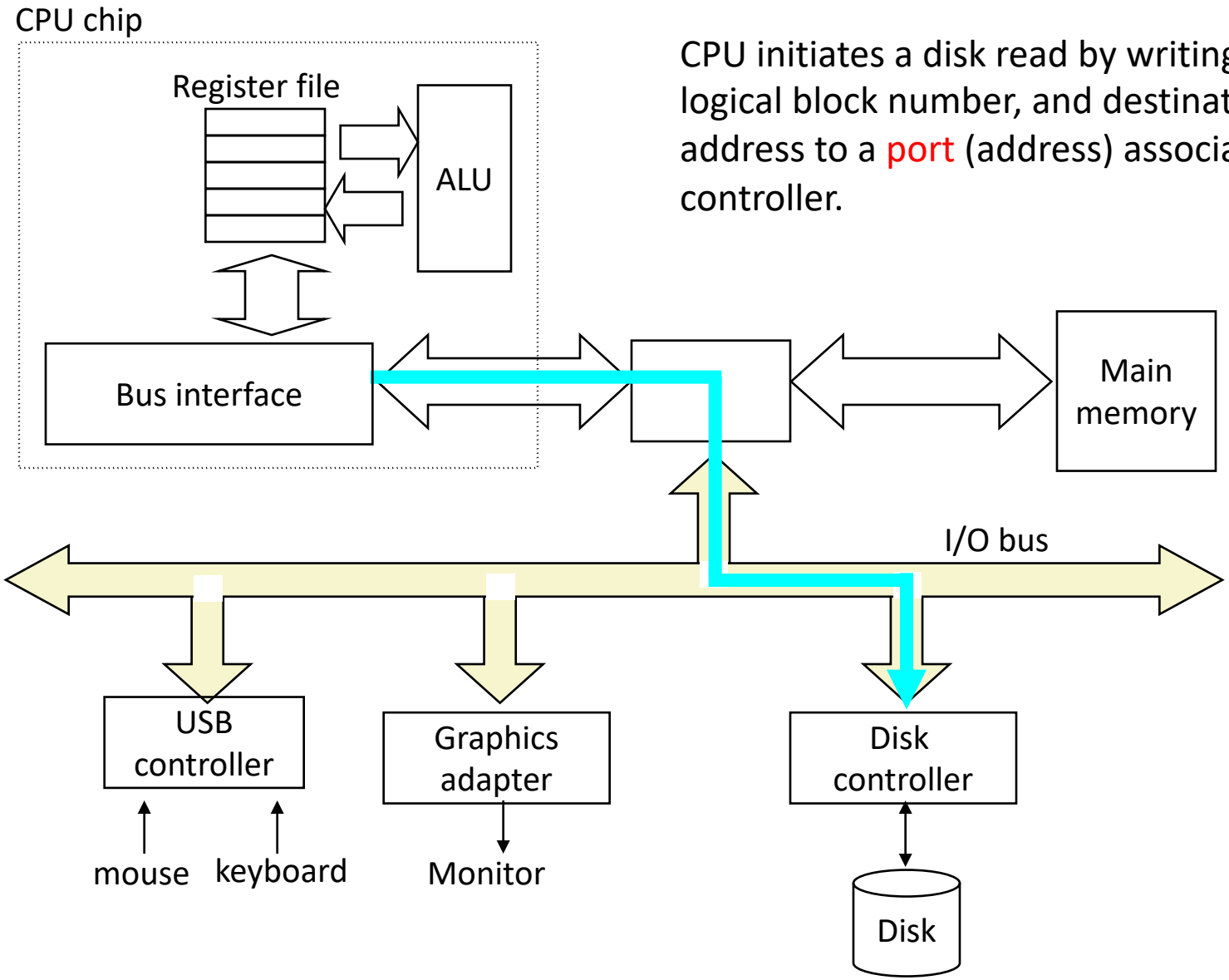
Logical Disk Blocks

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)
- Mapping between logical blocks and actual (physical) sectors
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface,track,sector) triples.
- Allows controller to set aside spare cylinders for each zone.
  - Accounts for the difference in “formatted capacity” and “maximum capacity”.

I/O Bus

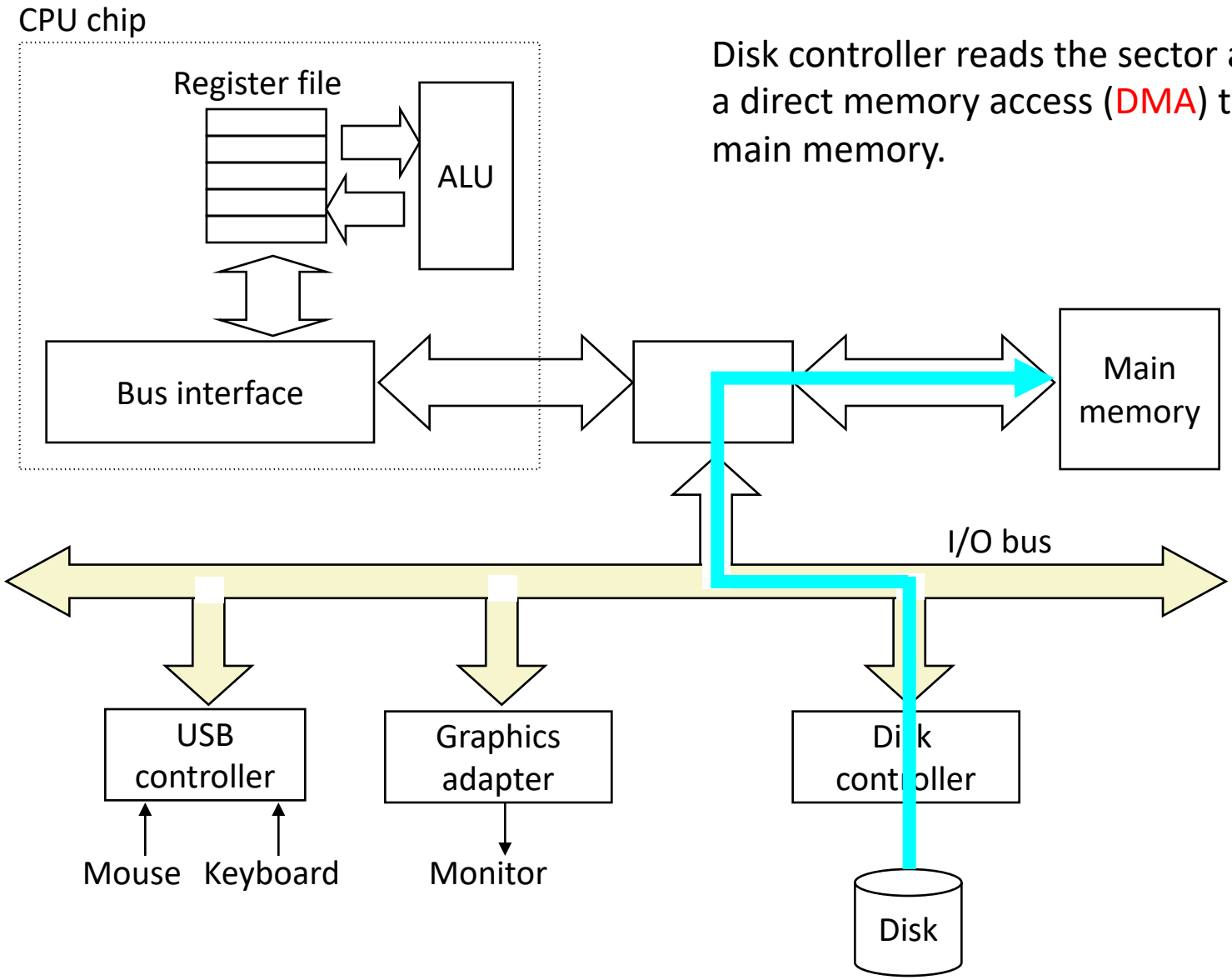

CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Monitor
Disk

Reading a Disk Sector (1)



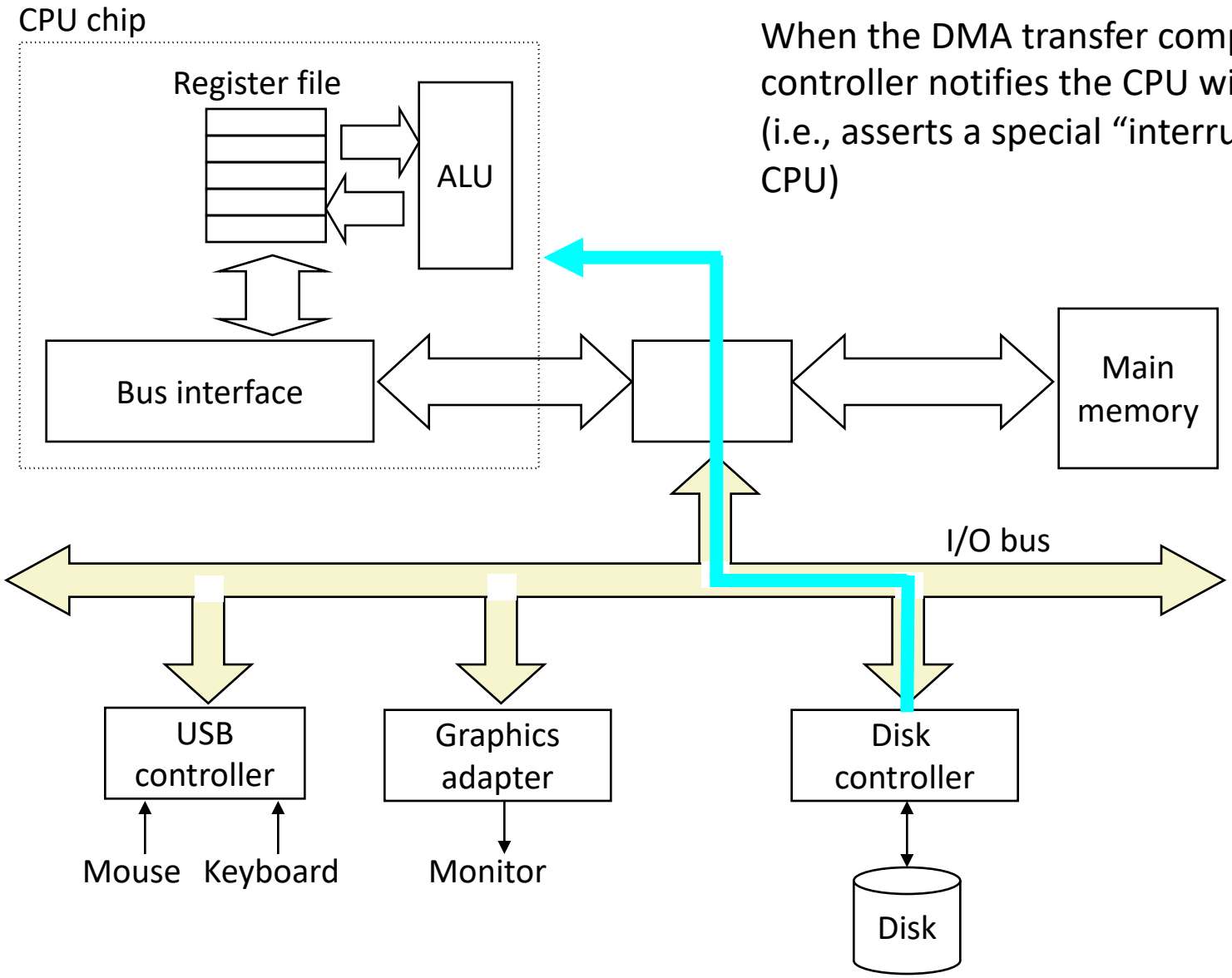
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Reading a Disk Sector (2)



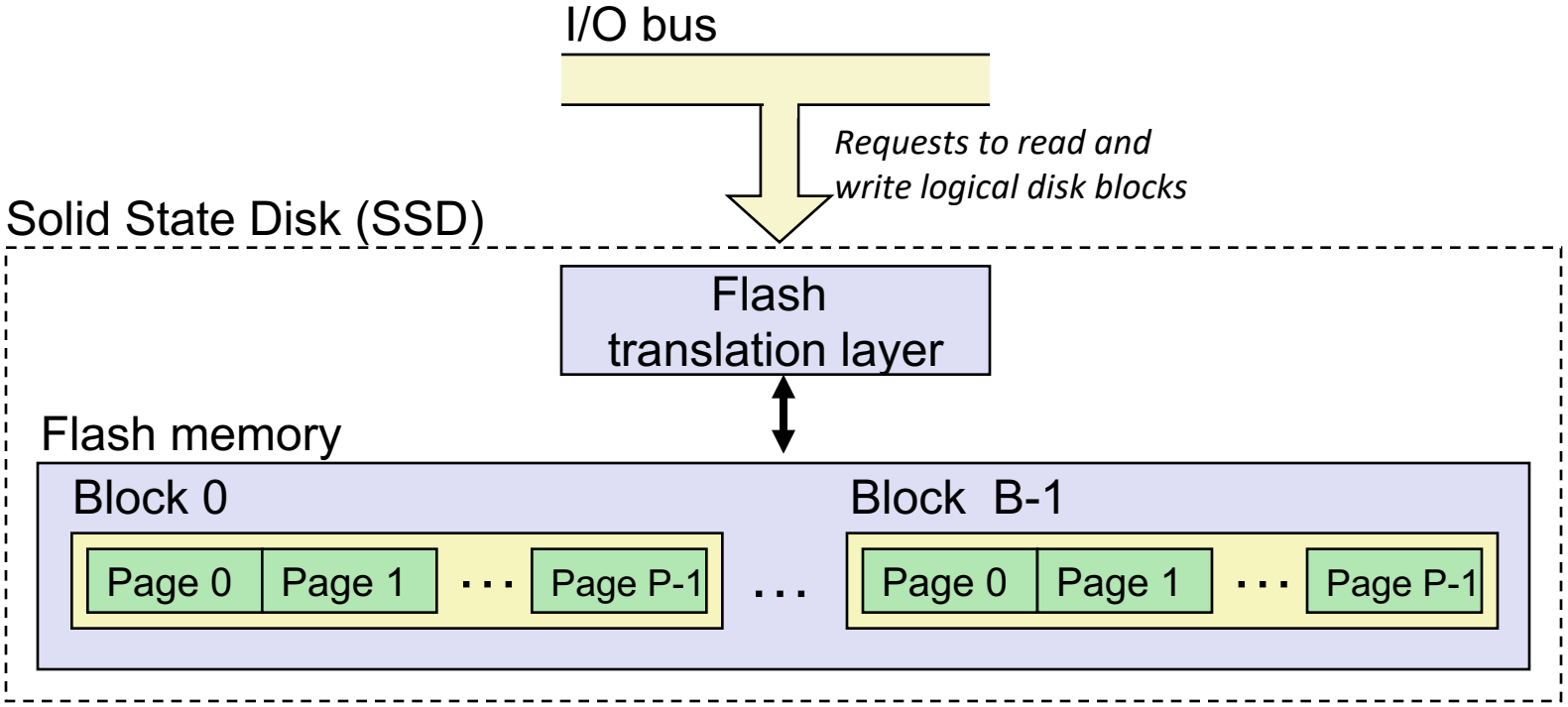
Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)



When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)

Solid State Disks (SSDs)



- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

SSD Performance Characteristics

Sequential read tput	550 MB/s	Sequential write tput	470 MB/s
Random read tput	365 MB/s	Random write tput	303 MB/s
Avg seq read time	50 us	Avg seq write time	60 us

- Sequential access faster than random access
  - Common theme in the memory hierarchy
- Random writes are somewhat slower
  - Erasing a block takes a long time (~1 ms)
  - Modifying a block page requires all other pages to be copied to new block
  - In earlier SSDs, the read/write gap was much larger.

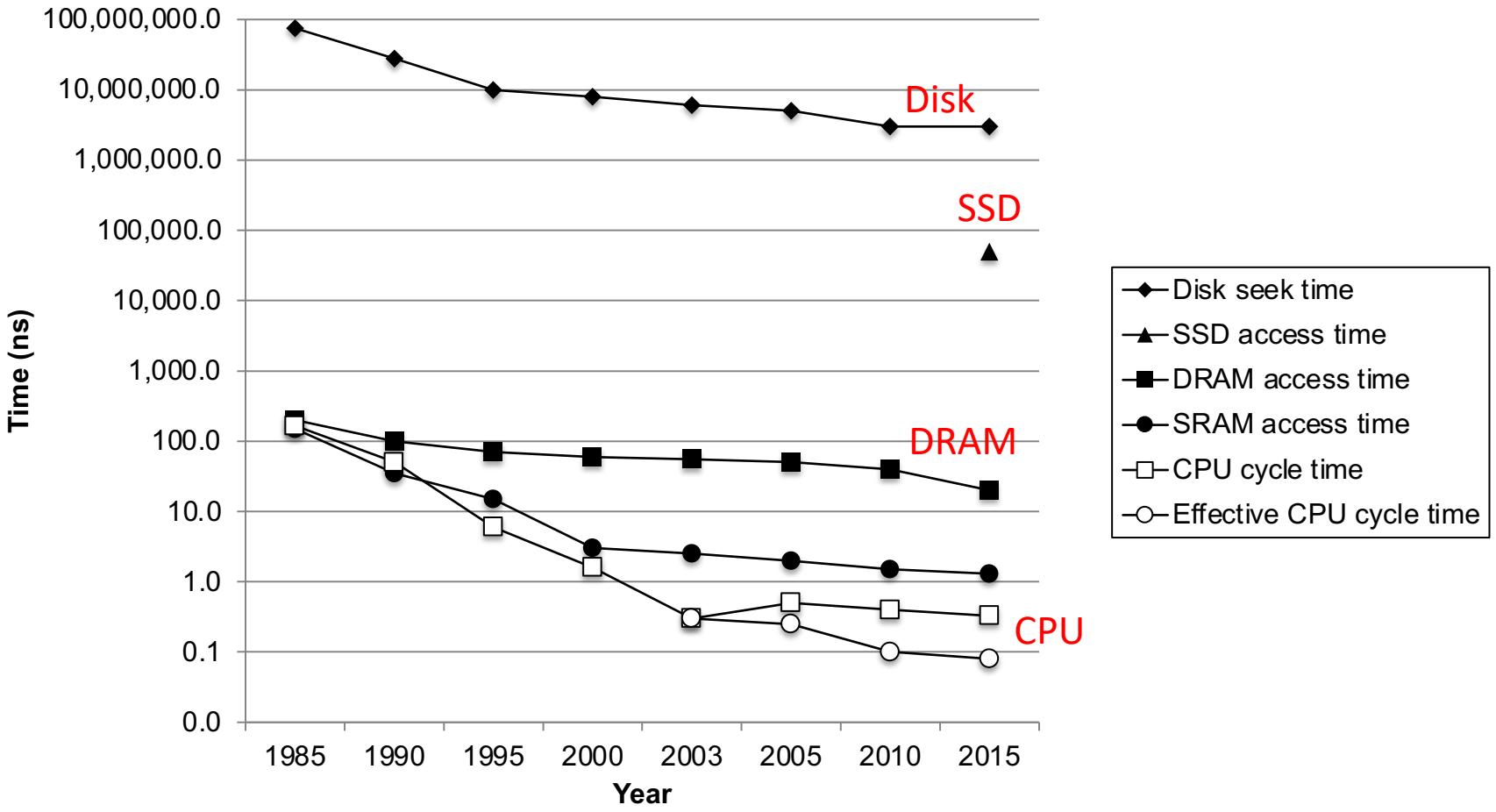
Source: Intel SSD 730 product specification.

SSD Tradeoffs vs Rotating Disks

- Advantages
  - No moving parts → faster, less power, more rugged
- Disadvantages
  - Have the potential to wear out
    - Mitigated by "wear leveling logic" in flash translation layer
    - E.g. Intel SSD 730 guarantees 128 petabyte (128×10^{15} bytes) of writes before they wear out
  - In 2015, about 30 times more expensive per byte
- Applications
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



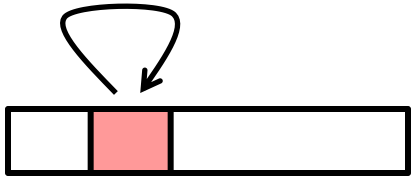
Locality to the Rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

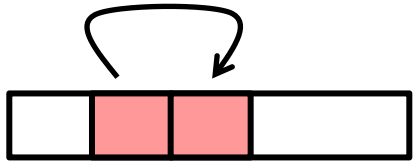
- **Locality of reference**

Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently



- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern). — Spatial locality
  - Reference variable `sum` each iteration. — Temporal locality
- Instruction references
  - Reference instructions in sequence. — Spatial locality
  - Cycle through loop repeatedly. — Temporal locality

Qualitative Estimates of Locality

- Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question: Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```