ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2024

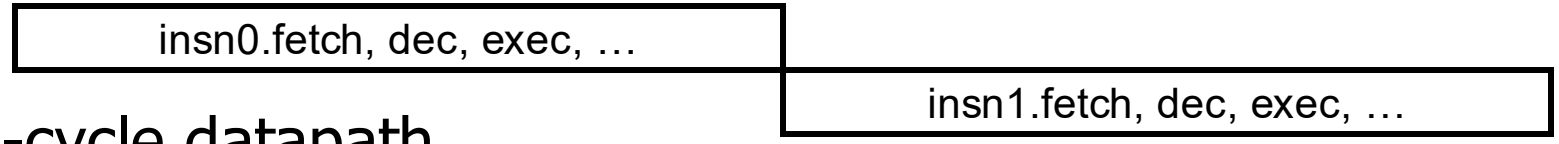
Pipelining

Rabih Younes
Duke University

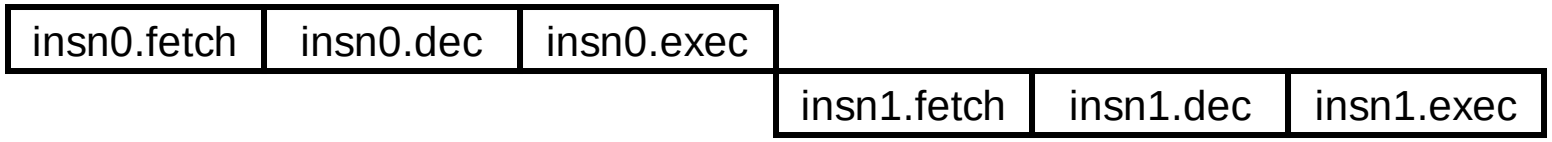
Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

Clock Period/Frequency and CPI/IPC

- Throughput = IPC x Frequency = 1/(CPI x Period)
- Single-cycle datapath
  - Low CPI: 1
  - Long clock period: to accommodate slowest insn

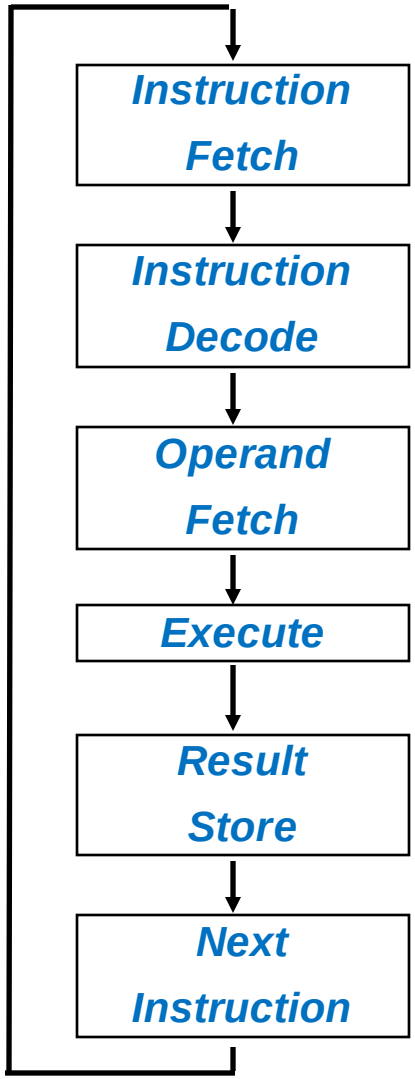


- Multi-cycle datapath
  - Short clock period (high frequency)
  - High CPI (low IPC)



- Can we have both low CPI and short clock period?
  - No good way to make a single insn go faster
  - Insn latency doesn't matter anyway… insn throughput matters
  - → Key: exploit inter-insn parallelism

Remember The von Neumann Model?

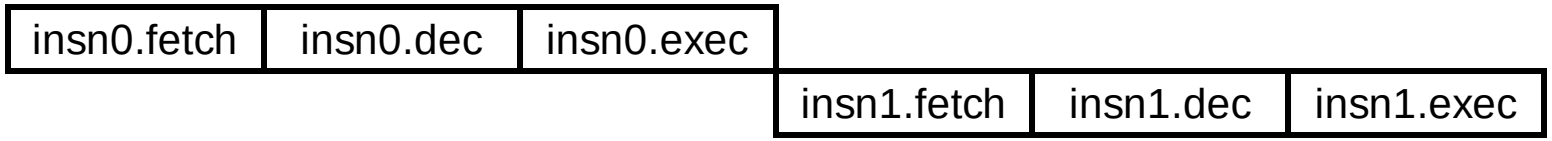


- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the numbers)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

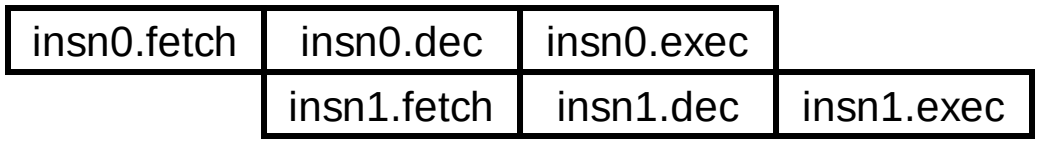
We'll call this the "**VN loop**"

Pipelining

- In multi-cycle design:

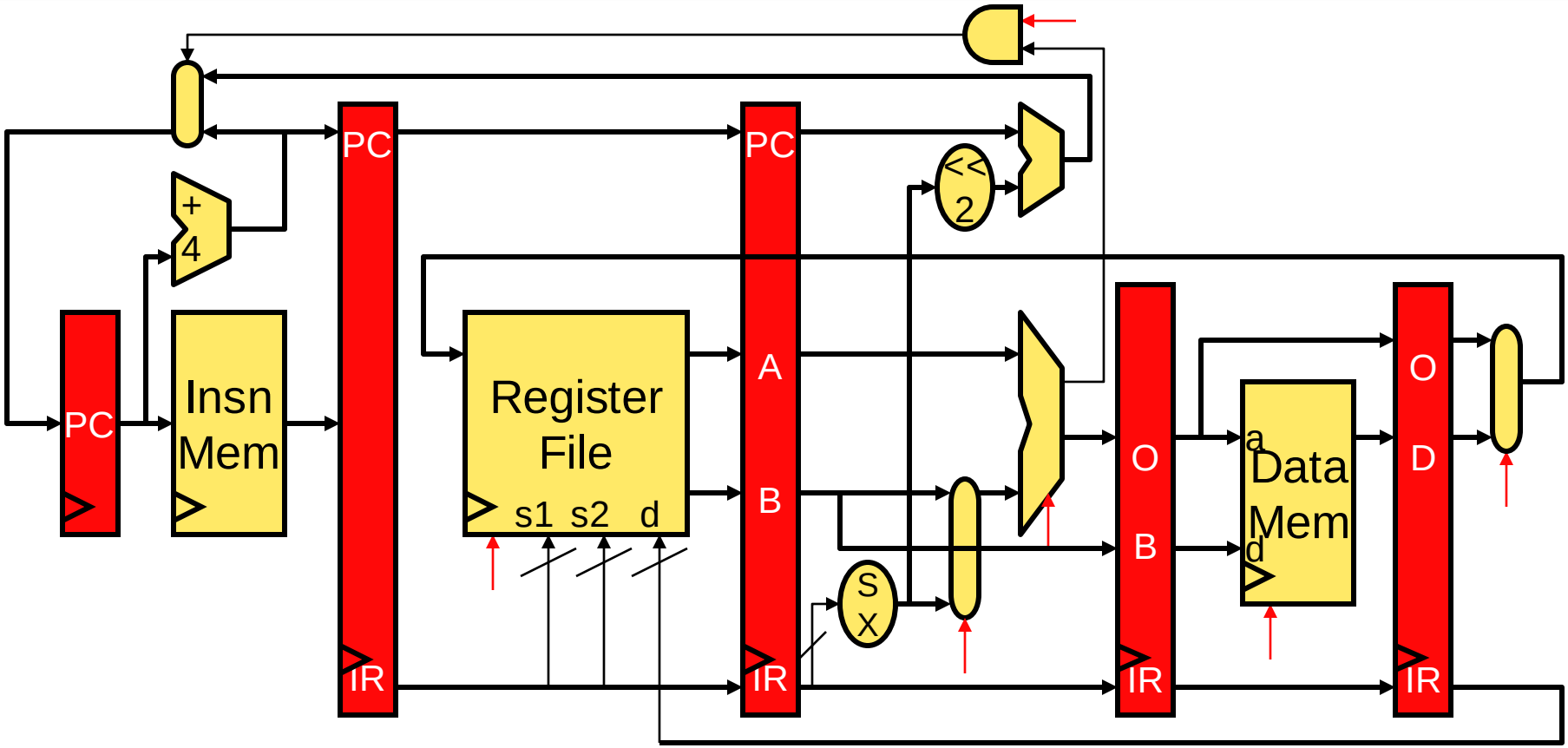


- In a pipelined design:



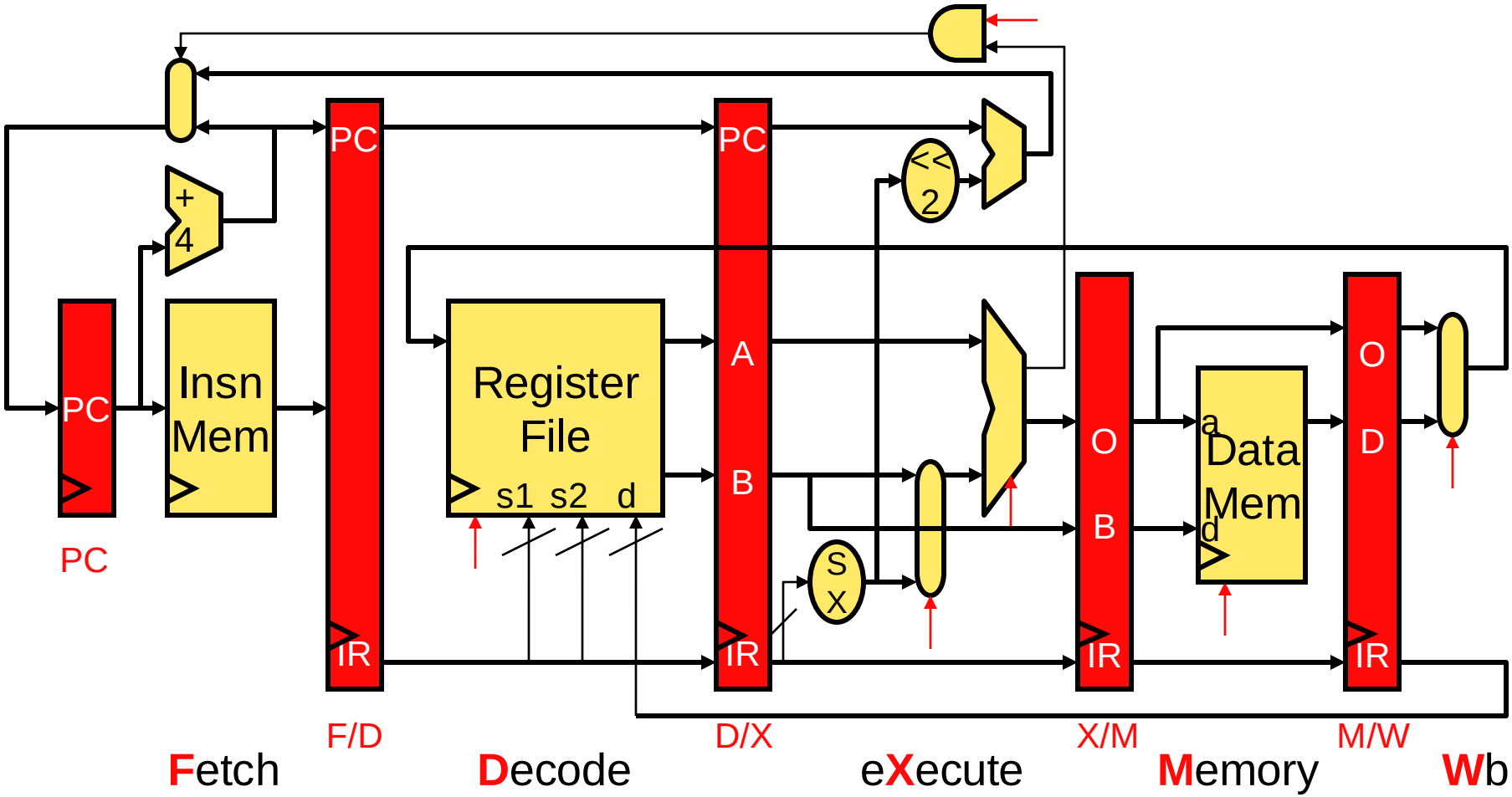
  - **Improves insn throughput rather than insn latency**
  - **Exploits parallelism at insn-stage level to do so**
  - **Individual insns take same number of stages**
  - \+ **But insns enter and leave at a much faster rate**
  - Breaks "atomic" VN loop, but maintains the illusion
- *Analogy: automotive assembly line*
- Challenges?

5-Stage-Pipelined MIPS Datapath



- Temporary values (PC, IR, A, B, O, D) re-latched (saved) every stage
  - Notice, PC not latched after ALU stage

Pipeline Terminology

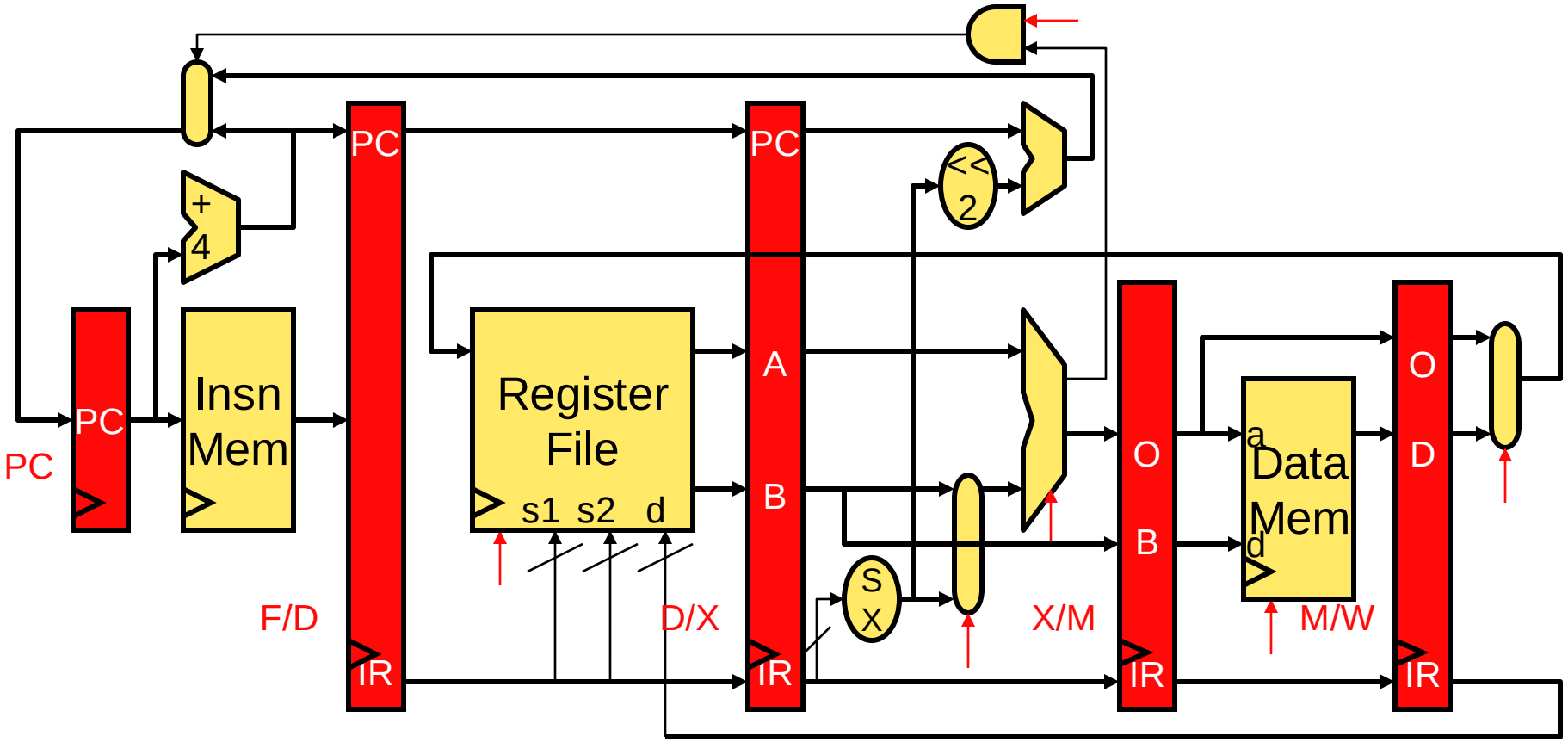


- Stages: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
- Latches (pipeline registers): **PC**, **F/D**, **D/X**, **X/M**, **M/W**

Some More Terminology

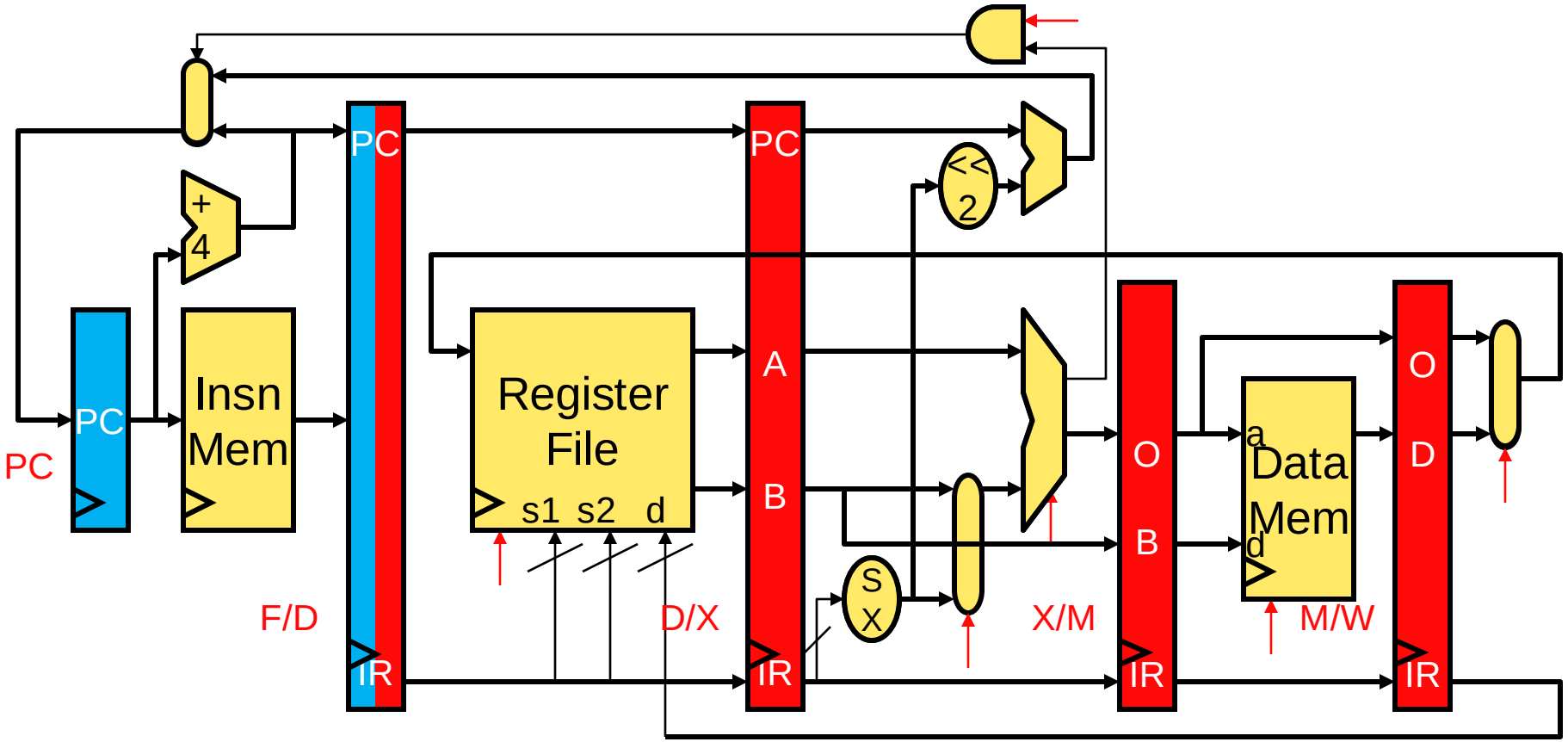
- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: "superscalar" (check ECE 552)
- **In-order pipeline**: insns enter execute stage in VN order
  - Alternative: "out-of-order" (check ECE 552)
- **Pipeline depth**: number of pipeline stages
  - Nothing magical about 5
  - Trend has been to have deeper pipelines
- → our MIPS pipeline is a scalar in-order 5-stage pipeline

Pipeline Example: Cycle 1



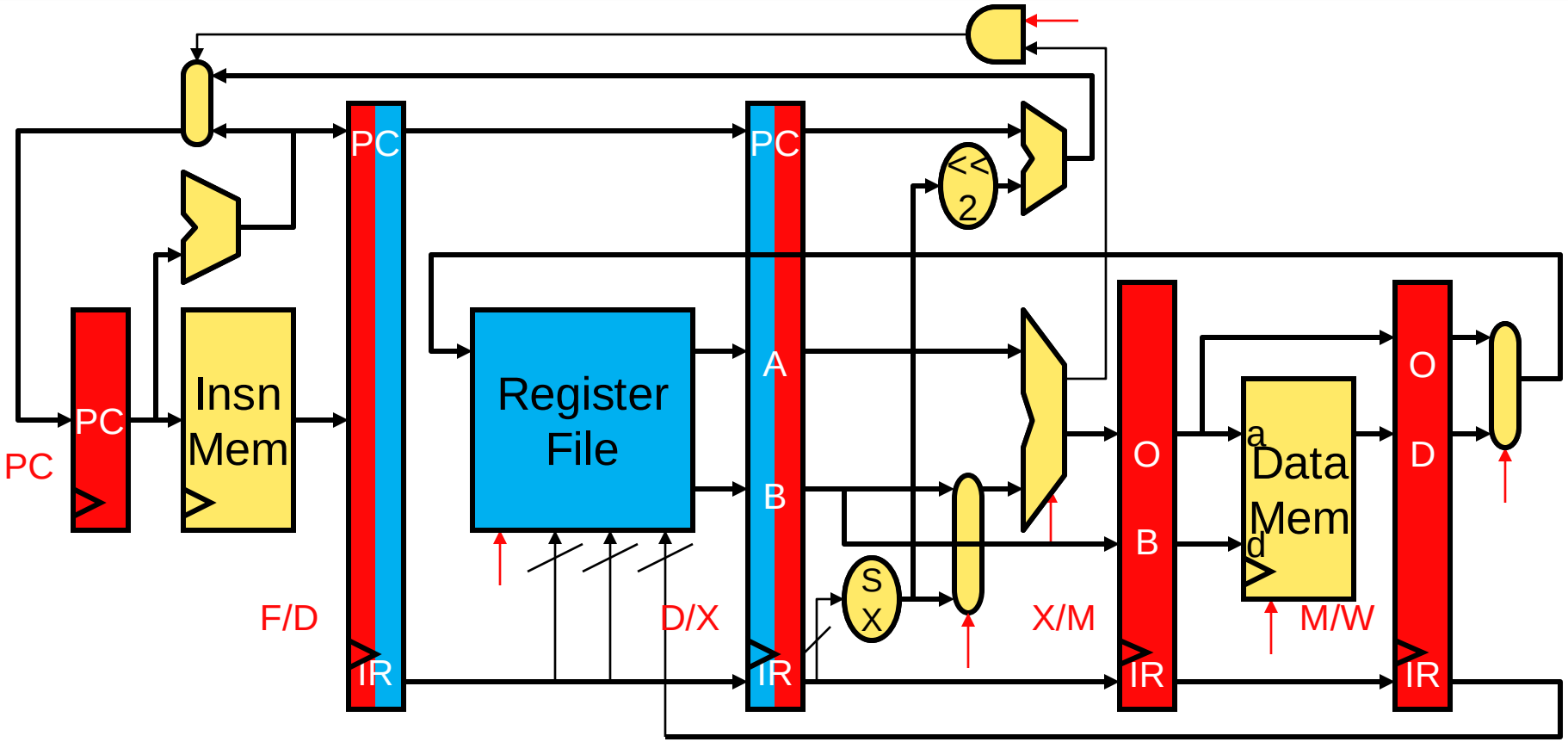

```
add $3,$2,$1
```

Pipeline Example: Cycle 2



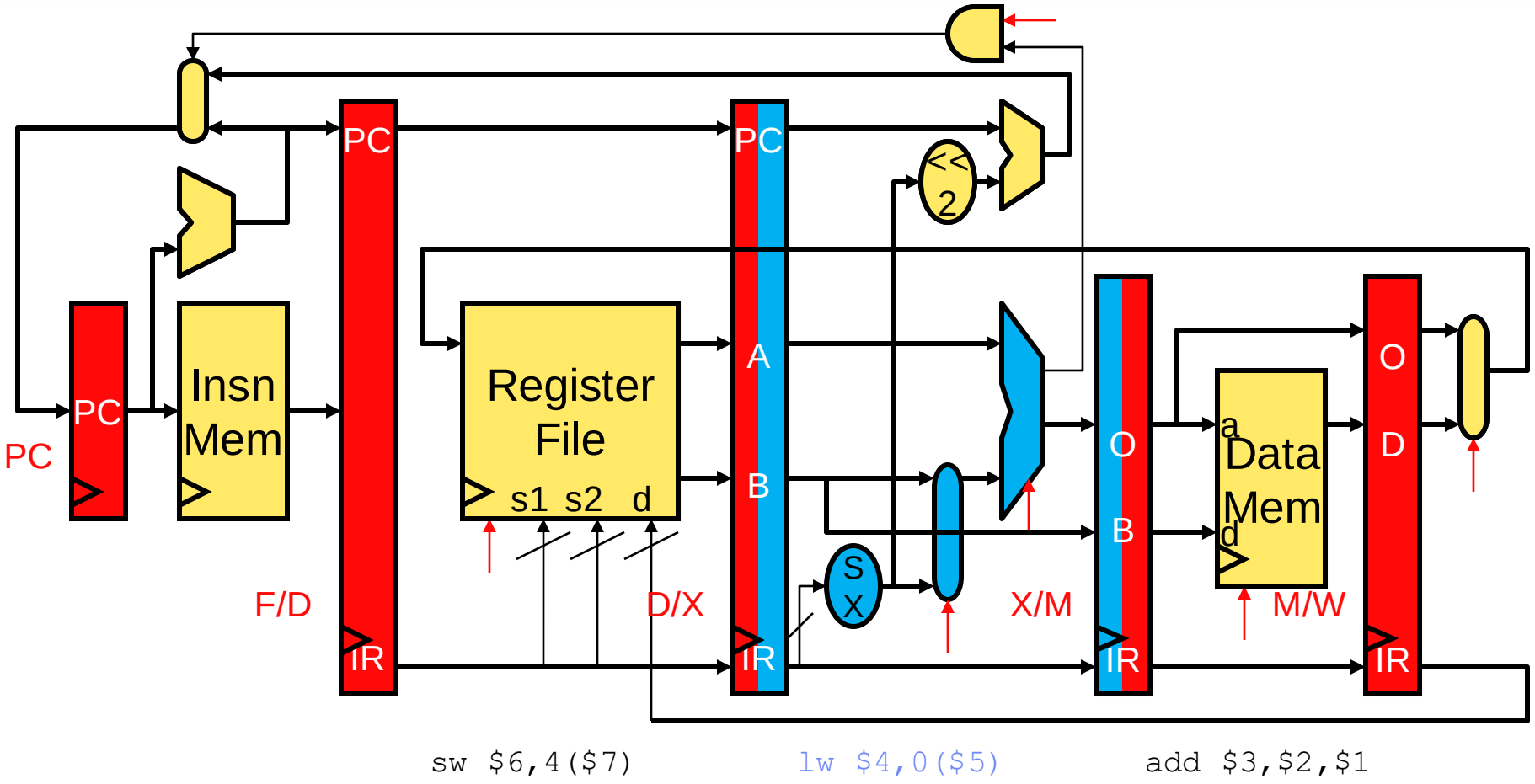
`lw $4,0($5)` `add $3,$2,$1`

Pipeline Example: Cycle 3

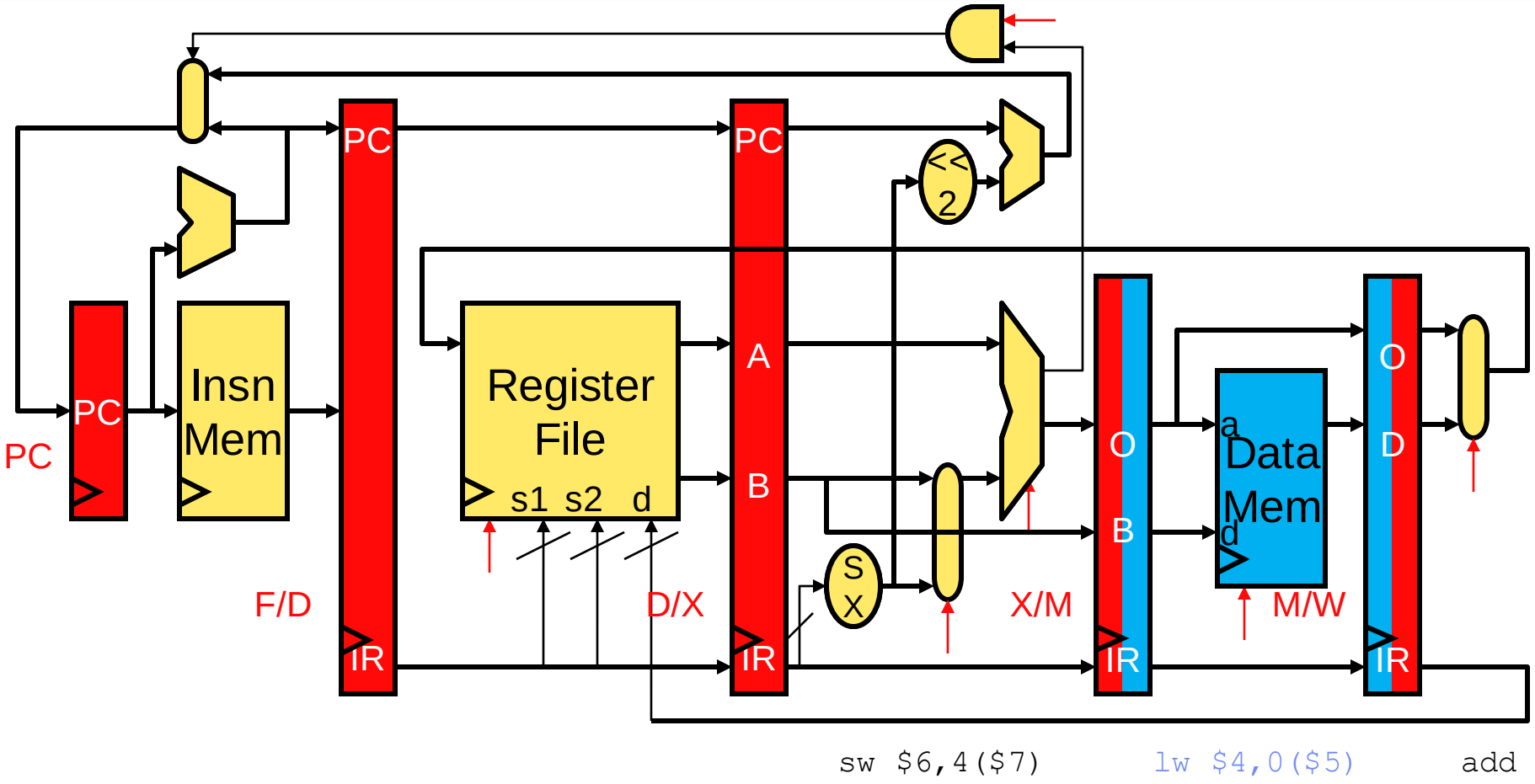



```
sw $6,4($7)        lw $4,0($5)        add $3,$2,$1
```

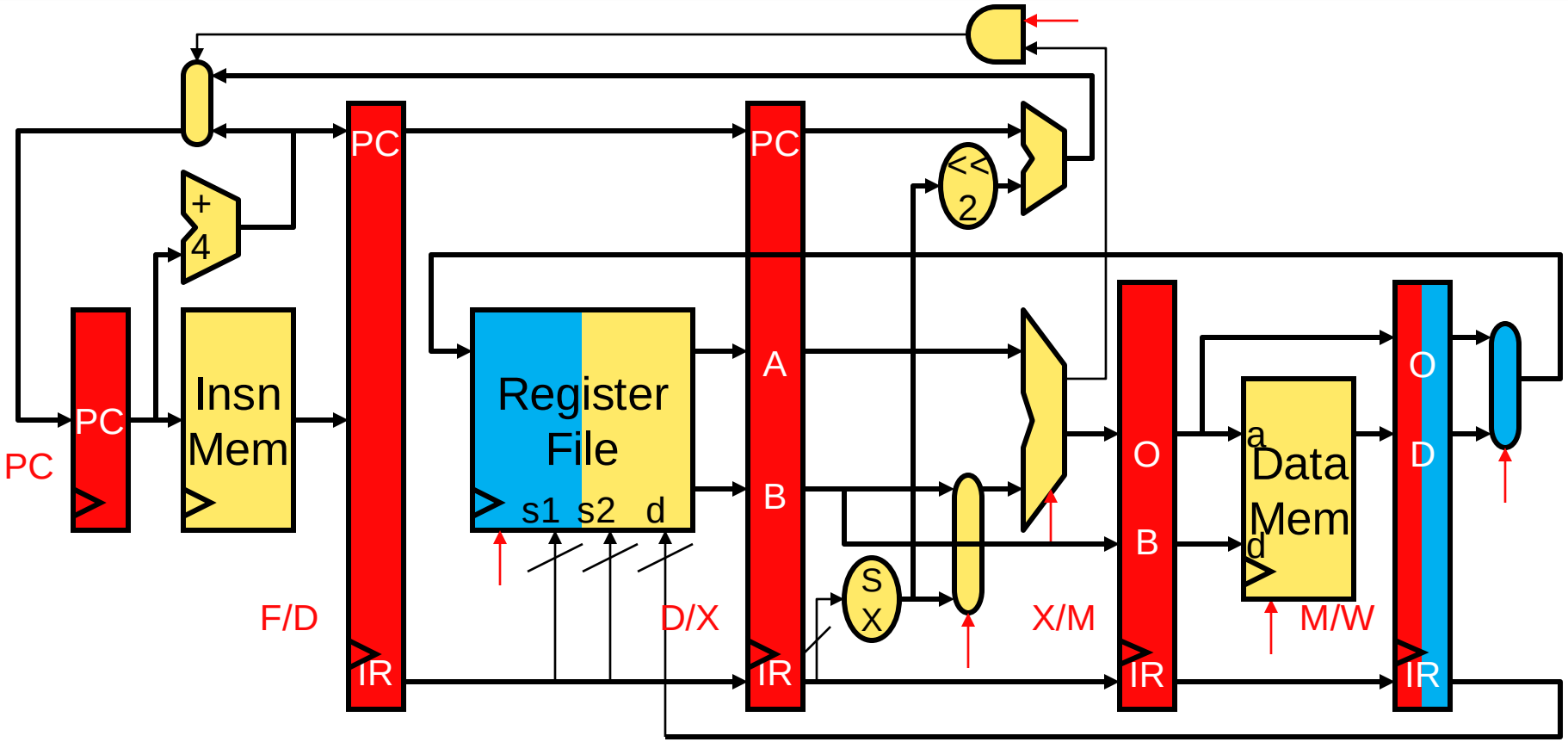
Pipeline Example: Cycle 4



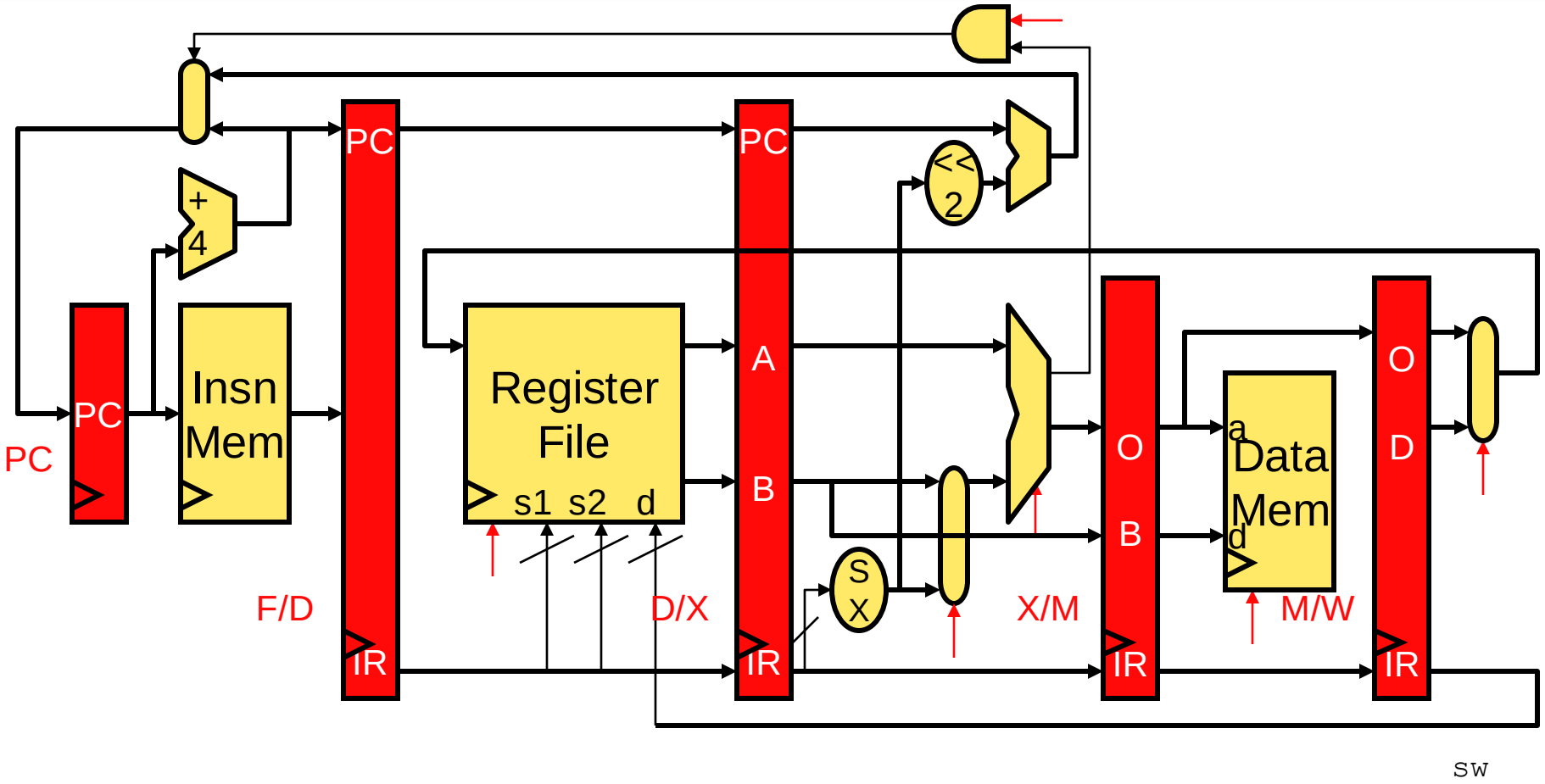
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

- Shorthand for what we just saw

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

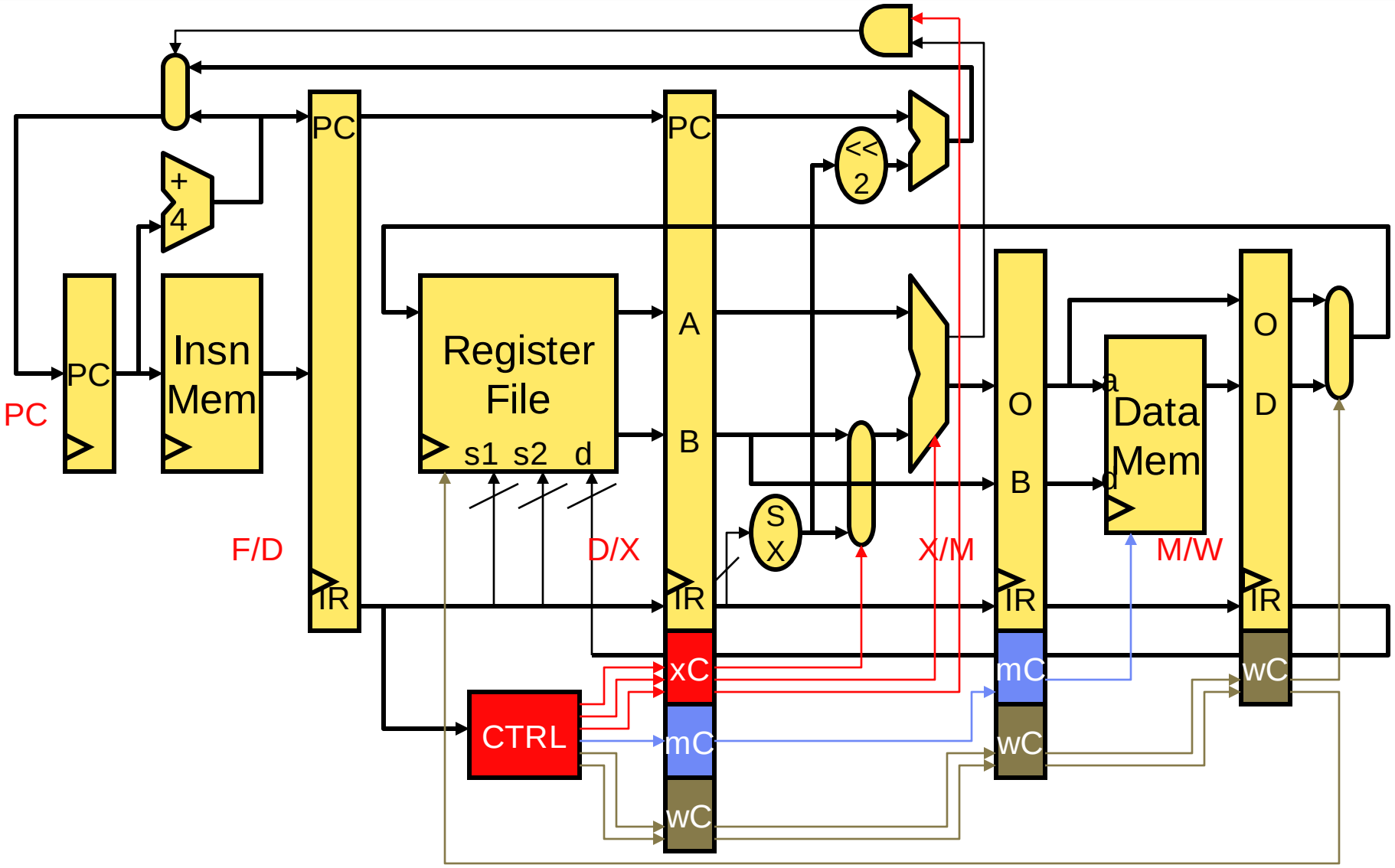
- Columns: clock cycle number
- Rows: insn
- Cells: stage
- Convention: **X** means **`lw $4,0($5)`** finishes execute stage and writes into X/M latch at the end of cycle 4

What About Pipelined Control?

- Should it be like single-cycle control?
  - But individual insn signals must be staged
- How many different control units do we need?
  - One for each insn in pipeline?

- → Solution: use simple single-cycle control, but pipeline it
  - Single controller
  - Key idea: pass the control signals with the insn through the pipeline

Pipelined Control



Pipeline Performance Example

- Single-cycle
  - Clock period = **50ns**, CPI = 1
  - Performance = **50ns/insn**

From previous lecture

- Multi-cycle
  - Branch: 20% (3 cycles), load: 20% (5 cycles), other: 60% (4 cycles)
  - Clock period = **12ns**, CPI = (0.2*3+0.2*5+0.6*4) = 4
    - Remember: latching overhead makes it 12, not 10
  - Performance = **48ns/insn**
    - *this will be even worse when accounting for a clock period that accommodates for the slowest stage

- Pipelined
  - Clock period = **12ns**
  - CPI = **1.5** (on average insn completes every 1.5 cycles)
  - Performance = **18ns/insn**

Some Questions…

- **Why is pipeline clock period > (delay through single-cycle dp / # stages)?**
  - Registers add delay
  - Pipeline stages have different delays → clock period should accommodate for the slowest stage
  - Note that both factors have implications on the ideal number of pipeline stages

Some Questions…

- **Why is pipeline CPI > 1?**
  - CPI for scalar in-order pipeline is 1 **+ stall penalties**
  - Stalls are used to resolve hazards
    - **Hazard**: condition that jeopardizes the VN illusion
    - **Stall**: artificial pipeline delay introduced to restore VN illusion (NOP instructions)

pronounced "no-op"

- Calculating pipeline CPI
  - **Frequency of stall * stall cycles**
  - Penalties add (stalls generally don't overlap in in-order pipelines)
  - 1 + stall-freq$_1$*stall-cyc$_1$ + stall-freq$_2$*stall-cyc$_2$ + …

- Correctness/performance/MCCF
  - Long penalties are OK if they happen rarely
    - e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for ideal number of pipeline stages

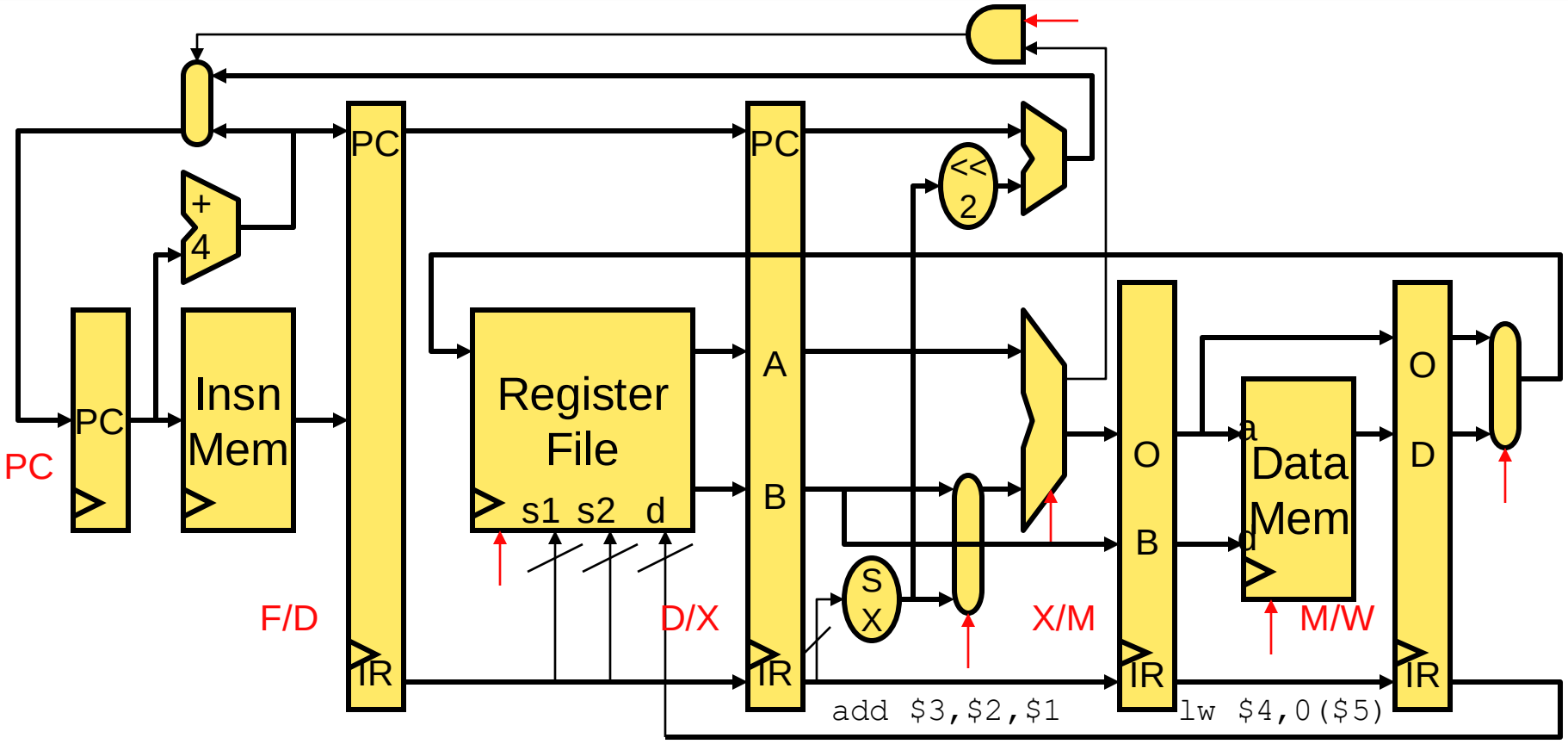
Dependences and Hazards

- **Dependence**: relationship between two insns
  - **2 types:**
    - **Data**: two insns use same storage location
    - **Control**: one insn affects whether another executes at all
  - Not a bad thing. Programs would be boring without them.
  - Not a problem in single-/multi-cycle designs
  - But we must account for it in a pipeline

- **Hazard**: dependence leading to a wrong outcome/state
  - Leads to stalling the pipeline → reduce performance
  - **3 types of hazards:**
    - **Structural**: due to datapath restrictions
    - **Data**: due to specific data dependences
    - **Control**: due to specific control dependences

Structural Hazards

Why Does Every Insn Take 5 Cycles?



- Could/should we allow **`add`** to skip M and go to W? No
  - It wouldn't help: peak fetch is still only 1 insn per cycle
  - → **Structural hazard**

Structural Hazards

- **Structural hazards**
  - Two insns trying to use same circuit at same time
- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once for at most one cycle
  - If we want to accommodate for a different behavior, we should somehow duplicate hardware

Data Hazards

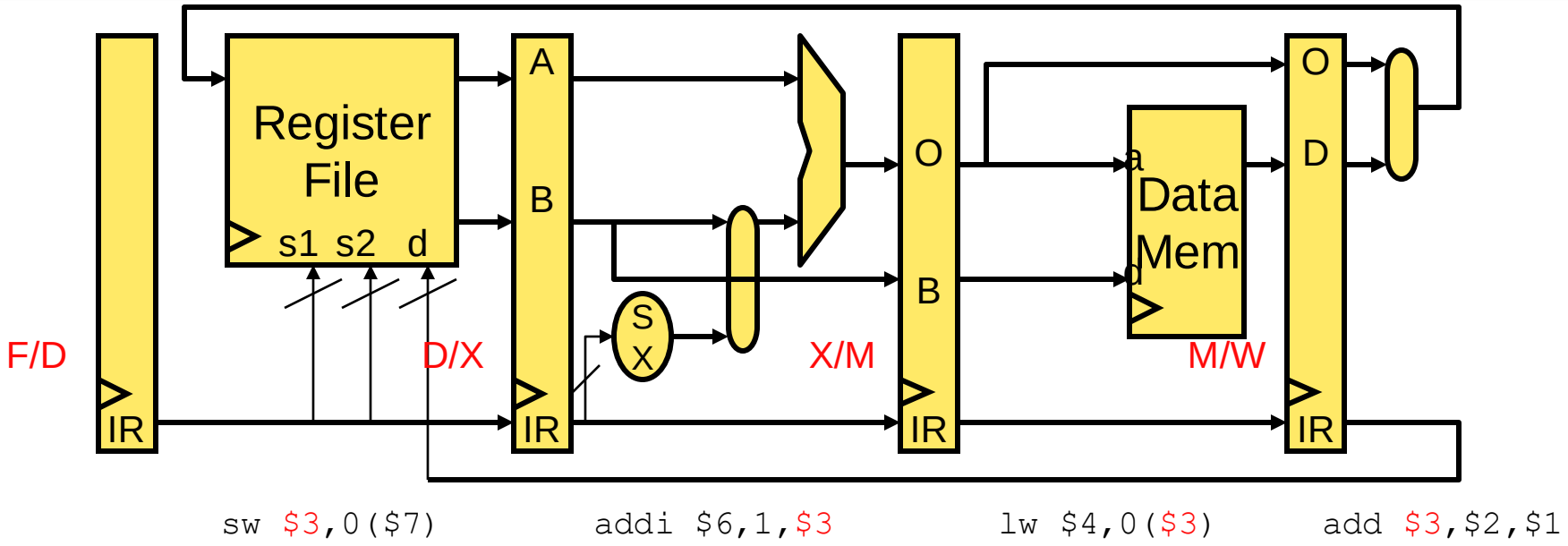
Data Hazards

- *Let's forget about branches and the control for a little bit…*

- Programs have **data dependences**
  - They pass values via registers and memory
  - 4 types of data dependencies
    - RAR (Read After Read)
    - RAW (Read After Write)
    - WAR (Write After Read)
    - WAW (Write After Write)

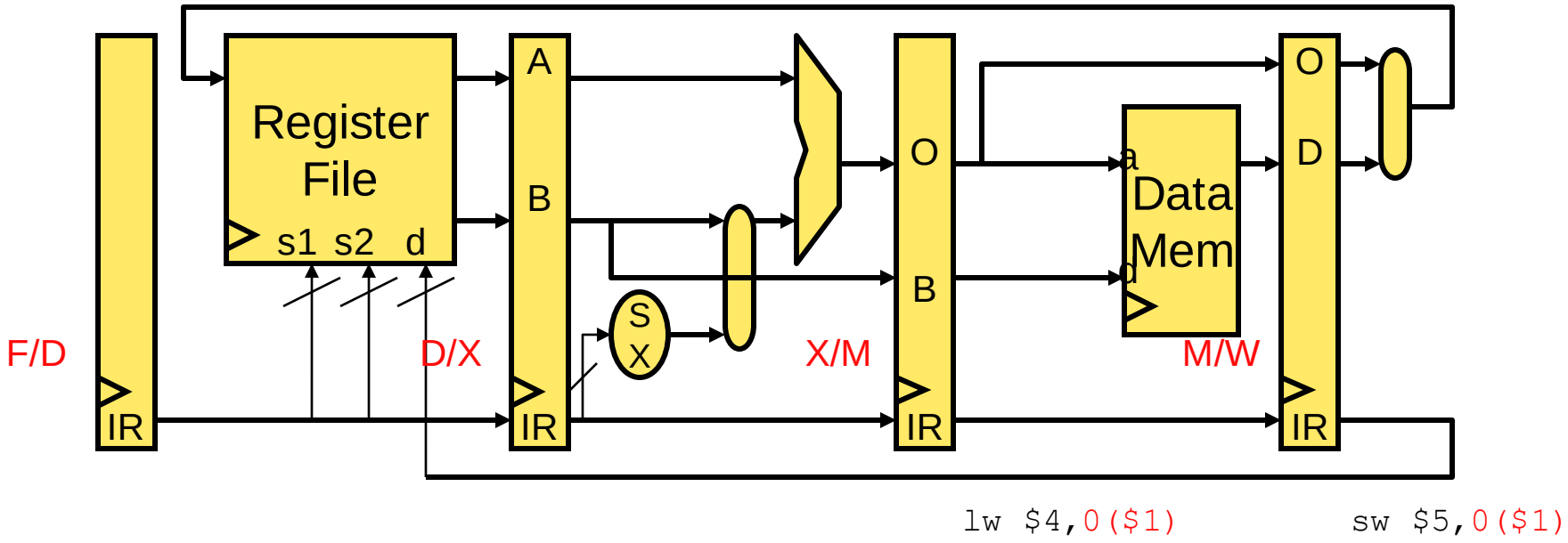
- Do all data dependences lead to hazards?
  - We should just check for RAW-related hazards in this case

Data Hazards



- Would this "program" execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - **`add`** is writing its result into **`$3`** in current cycle
  - **`lw`** read **`$3`** 2 cycles ago → got wrong value
  - **`addi`** read **`$3`** 1 cycle ago → got wrong value
  - **`sw`** is reading **`$3`** this cycle → OK (if regfile writes first, reads second)

Memory & RegFile Data Hazards



- What about data hazards through memory?
  - No
  - **`lw`** following **`sw`** to same address in next cycle, gets right value
  - Why? DMem read & write take place in the same stage
- Data hazards through registers?
  - Yes (previous slide)
  - Occur because regfile write is 3 stages after regfile read

Fixing RegFile Data Hazards

- Make a new rule, which is:
  - We can only read register value 3 cycles after writing it

- One way to enforce this: make sure programs don't do it
  - Compiler puts two independent insns between write & read insn pair (if they aren't there already)
    - Independent means: "do not interfere with the reg in question"
    - Compiler moves around existing insns to do this
      - Called **code scheduling**
    - If none can be found, insert **NOPs** between data-dependent insns
    - This is called **software interlocks**

Software Interlock Example

```
1.  sub $3,$2,$1
2.  lw $4,0($3)
3.  sw $7,0($3)
4.  add $6,$2,$8
5.  addi $3,$5,4
```

Problem(s)?

- Insns 1 and 2 need to be separated by 2 insns
- Can any of the last 3 insns be scheduled between the first two?
  - **`sw $7,0($3)`**? No, because it creates hazard with **`sub $3,$2,$1`**
  - **`add $6,$2,$8`**? Yes
  - **`addi $3,$5,4`**? Yes
    - This one isn't that straight forward
    - When in the pipeline, **`lw`** and **`sw`** read their "old" **`$3`** value before **`addi`** writes its "new" **`$3`** value in the W stage, so the code's behavior is still the same as originally intended

```
1.  sub $3,$2,$1
2.  add $6,$2,$8
3.  addi $3,$5,4
4.  lw $4,0($3)
5.  sw $7,0($3)
```

Software Interlock Performance

- Using same values of earlier example:
  - 20% of insns require insertion of 1 **`nop`**
  - 5% of insns require insertion of 2 **`nops`**

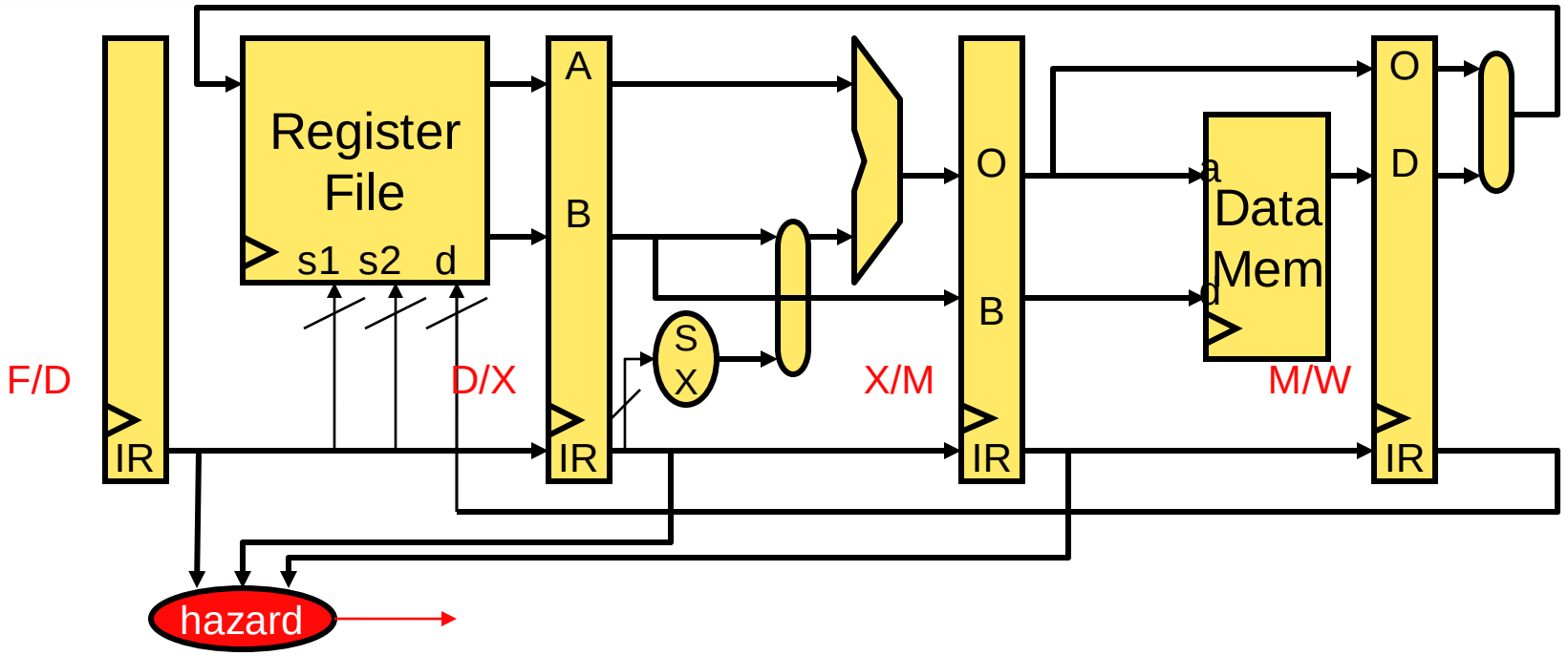
  - CPI is still 1 technically
  - But now there are more insns
  - #insns = 1 + 0.20*1 + 0.05*2 = **1.3**
  - **30% more insns (30% slowdown) due to data hazards**

Hardware Interlocks

- Problem with software interlocks?
  - Not compatible or scalable
  - Where does **3** in "read register 3 cycles after writing" come from?
    - From the structure (depth) of pipeline
  - What if the next MIPS version uses a 7-stage pipeline?
    - Programs compiled assuming 5-stage pipeline will break

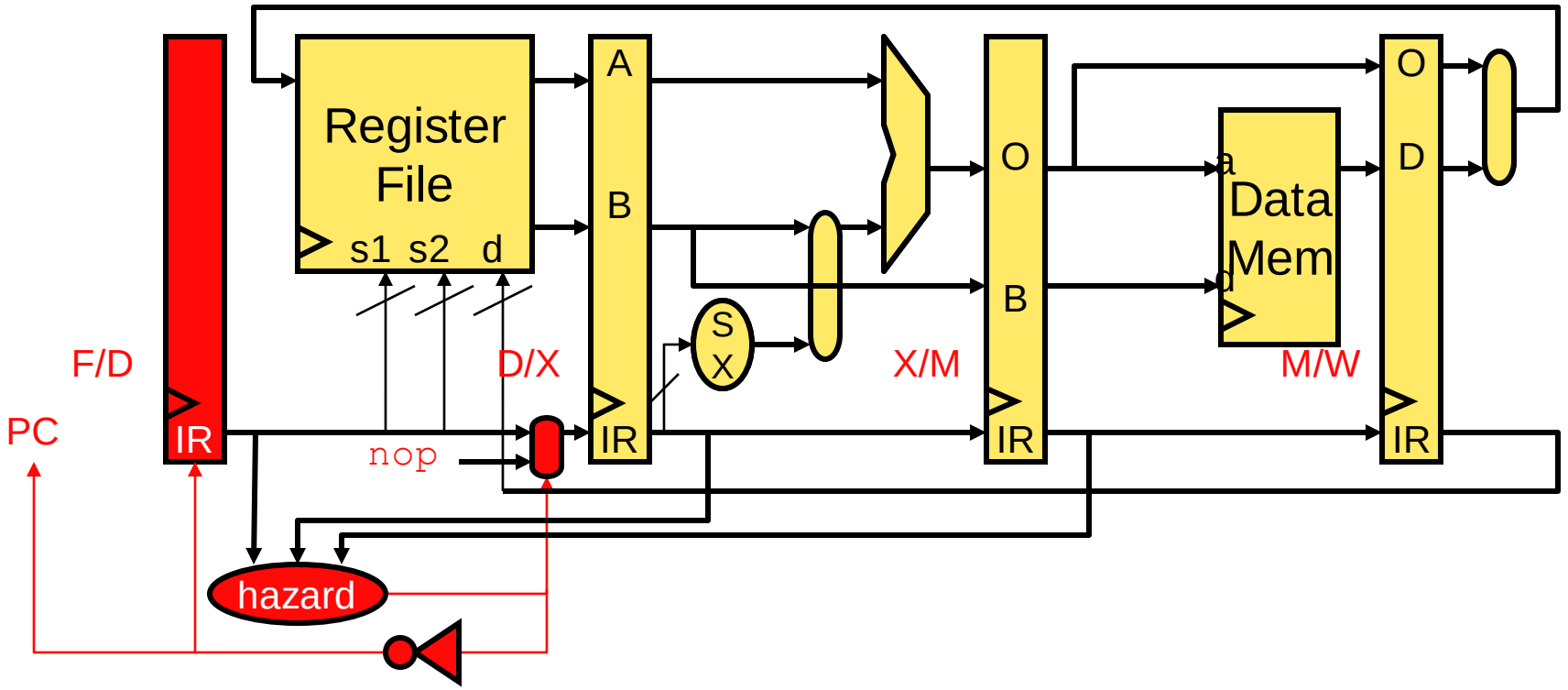
- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
  - Two aspects to this:
    1. Detecting hazards
    2. Fixing hazards

Detecting Data Hazards



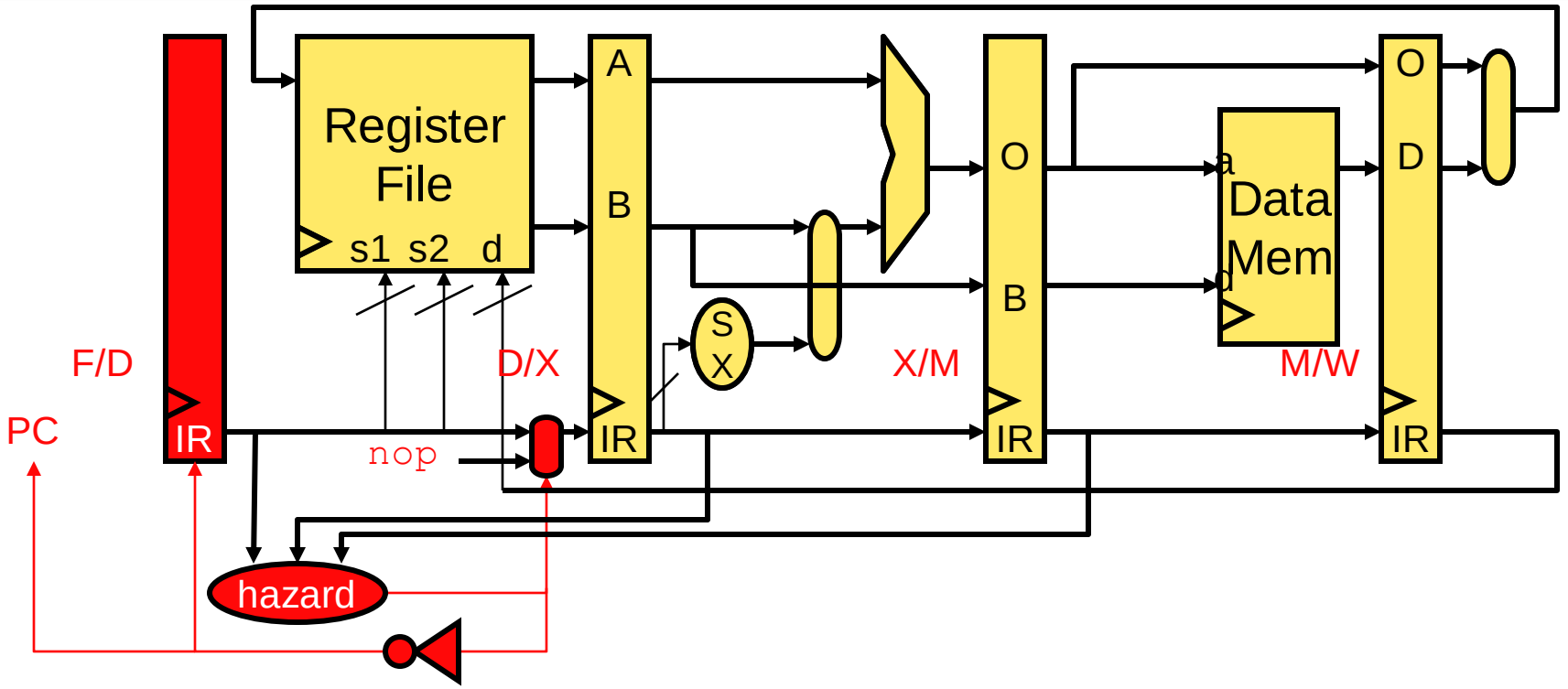
- Compare F/D insn input register names with output register names of older insns in pipeline
  - Hazard =
    - (F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
    - (F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

Fixing Data Hazards



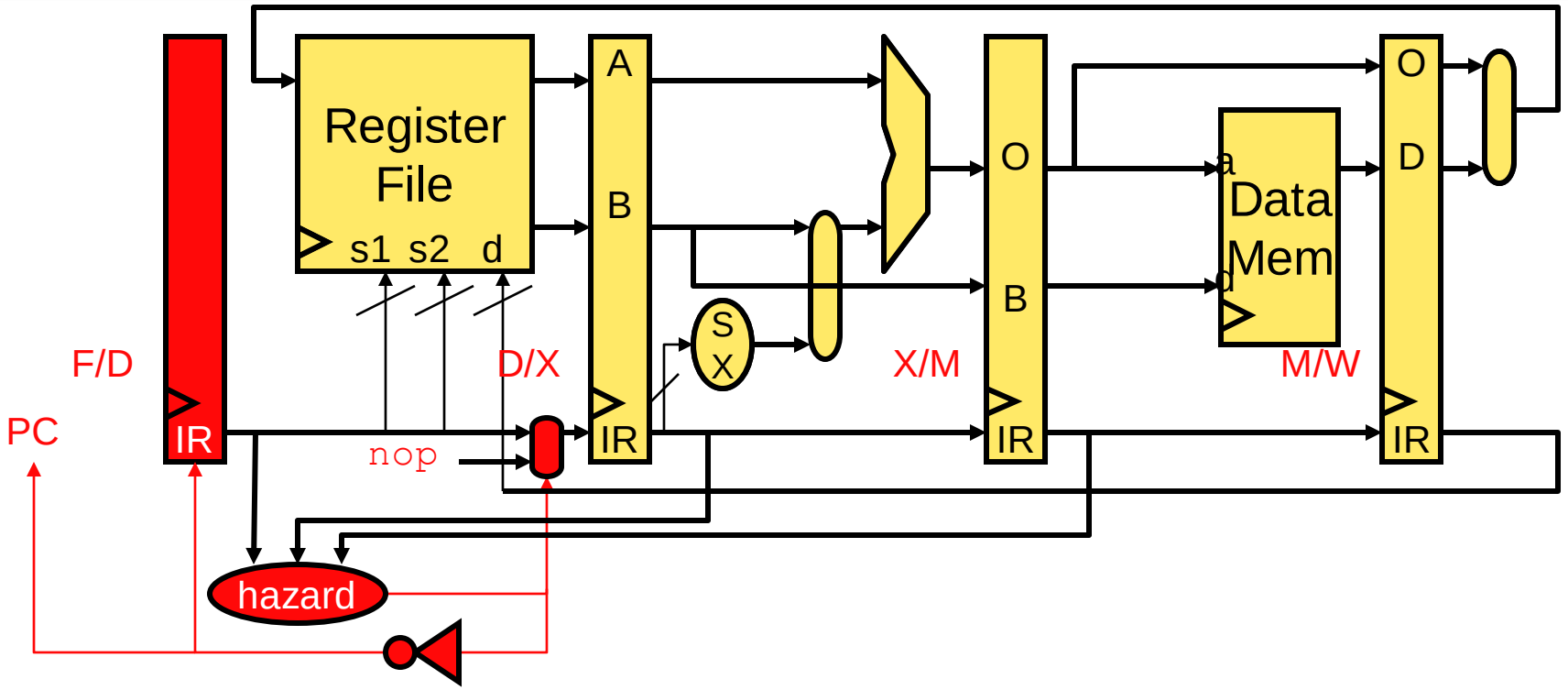
- Prevent F/D insn from reading (advancing) this cycle
  - Write **`nop`** into D/X.IR (effectively, insert **`nop`** in hardware)
    - *Also, reset (clear) the datapath control signals*
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Hardware Interlock Example: cycle 1



(**F/D.IR.RS1 == D/X.IR.RD**) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)
= **1**

Hardware Interlock Example: cycle 2

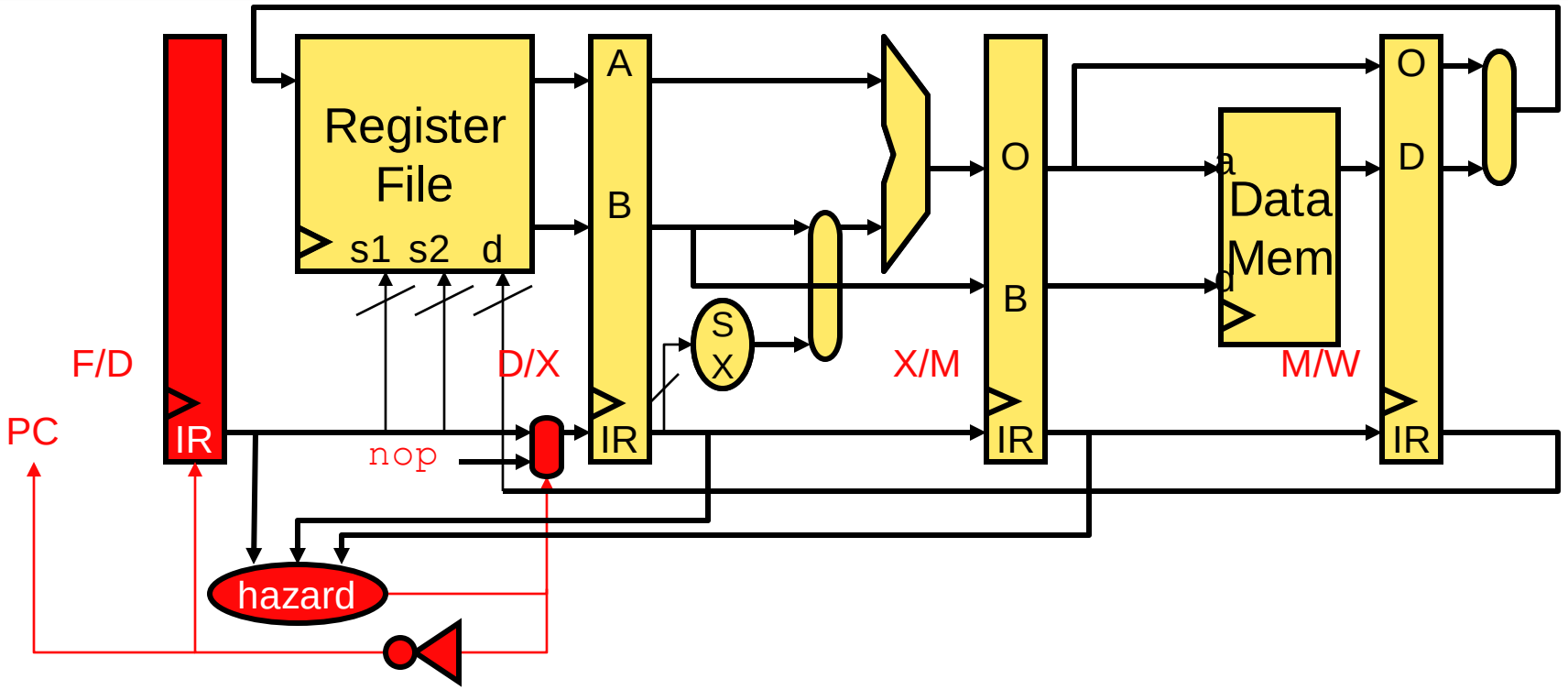


(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||

(**F/D.IR.RS1 == X/M.IR.RD**) || (F/D.IR.RS2 == X/M.IR.RD)

= **1**

Hardware Interlock Example: cycle 3




```
lw $4,0($3)        nop        nop        add $3,$2,$1
```

(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)
= **0**

Pipeline Control Terminology

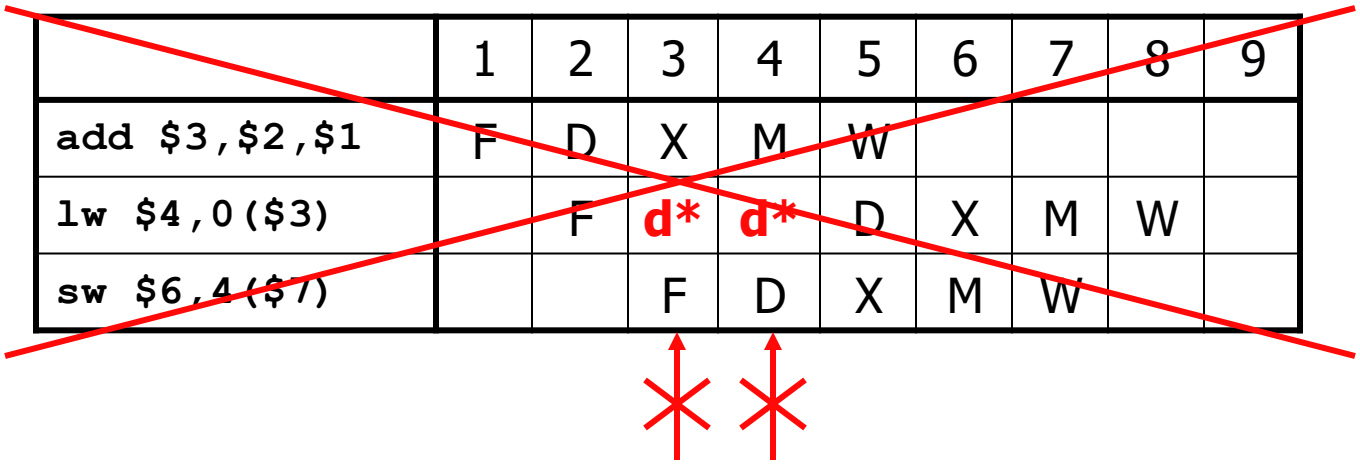
- Hardware interlock maneuver (inserting a NOP) is called **stall** or **bubble**
- Mechanism is called **stall logic**
- Part of a more general **pipeline control** mechanism
  - Controls advancement of insns through pipeline
- Distinguished from **pipelined datapath control**
  - Controls datapath at each stage
  - "Pipeline control" controls advancement of "datapath control"

Pipeline Diagram with Data Hazards

- Data hazard stall indicated with **d***
  - Stall propagates to younger insns

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	**d***	**d***	D	X	M	W	
`sw $6,4($7)`			F	F	F	D	X	M	W

  - This is not OK (why?)



Pipeline Diagram with Data Hazards

- Can also do:

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	D	D	X	M	W	
`sw $6,4($7)`			F	F	F	D	X	M	W

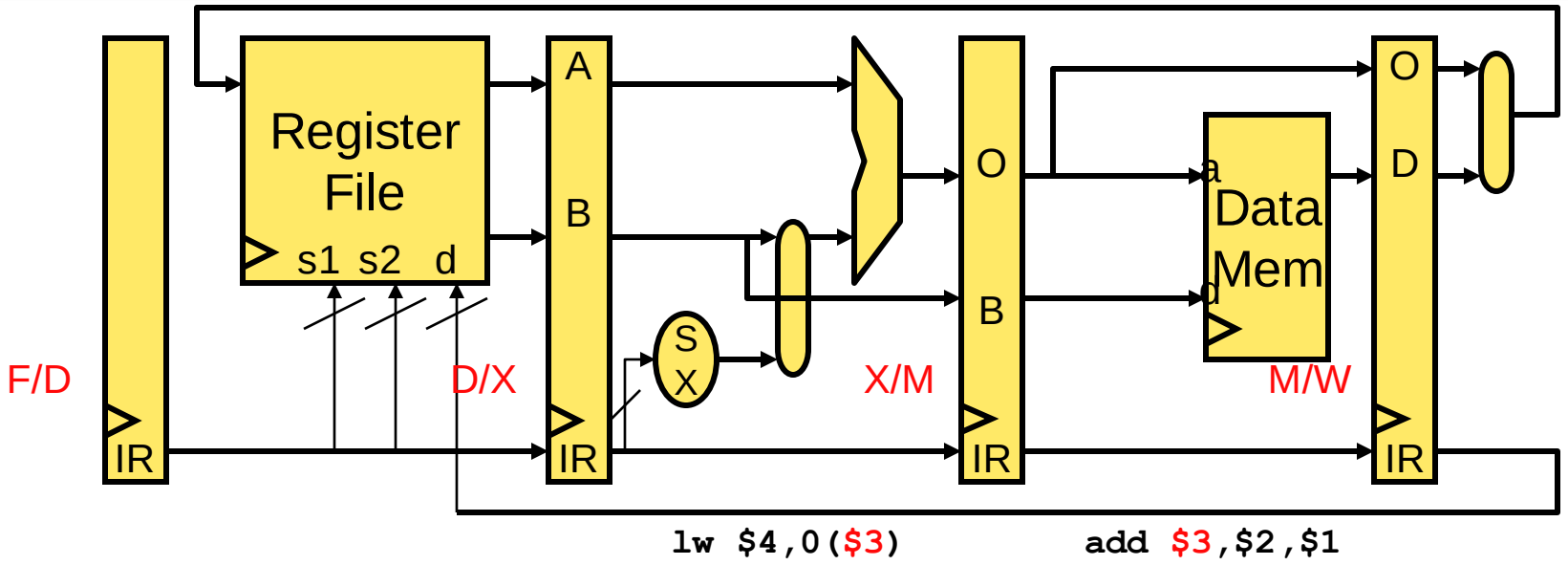
Hardware Interlock Performance

- Hardware interlocks: same as software interlocks
  - 20% of insns require 1 cycle stall (i.e., insertion of 1 **`nop`**)
  - 5% of insns require 2 cycle stall (i.e., insertion of 2 **`nops`**)

  - CPI = 1 + 0.20*1 + 0.05*2 = **1.3**
  - So, either CPI stays at 1 and #insns increases 30% (software)
  - Or, #insns stays at 1 (relative) and CPI increases 30% (hardware)
  - Same difference

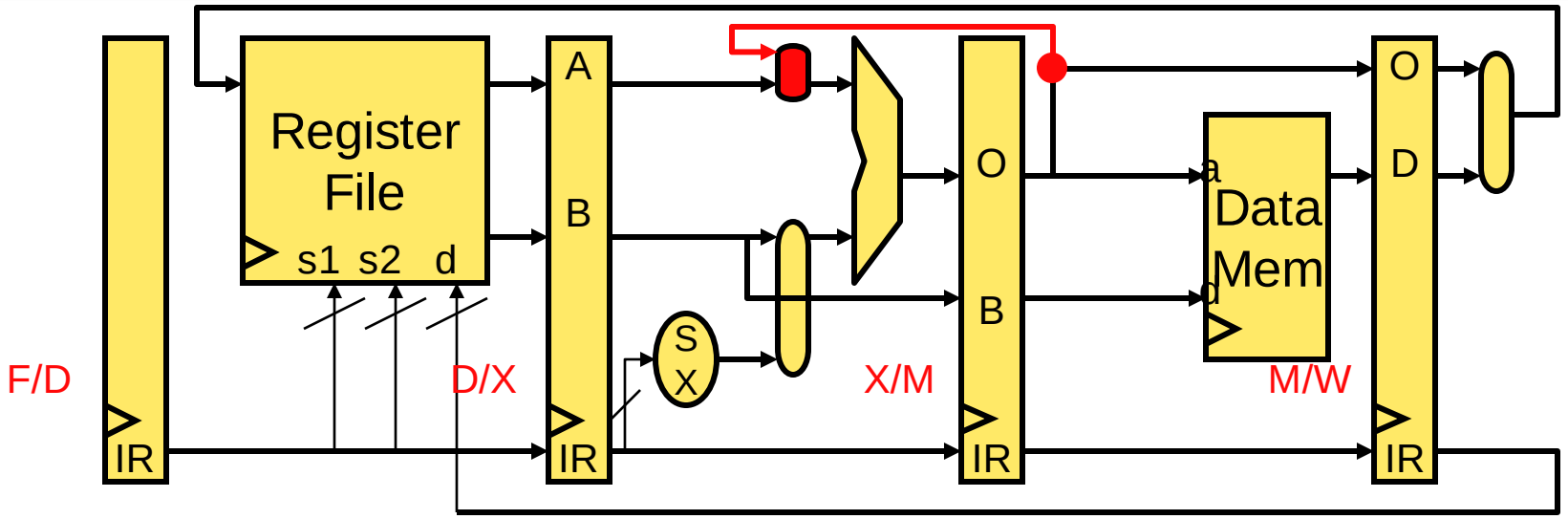
- We can do better!

Observe



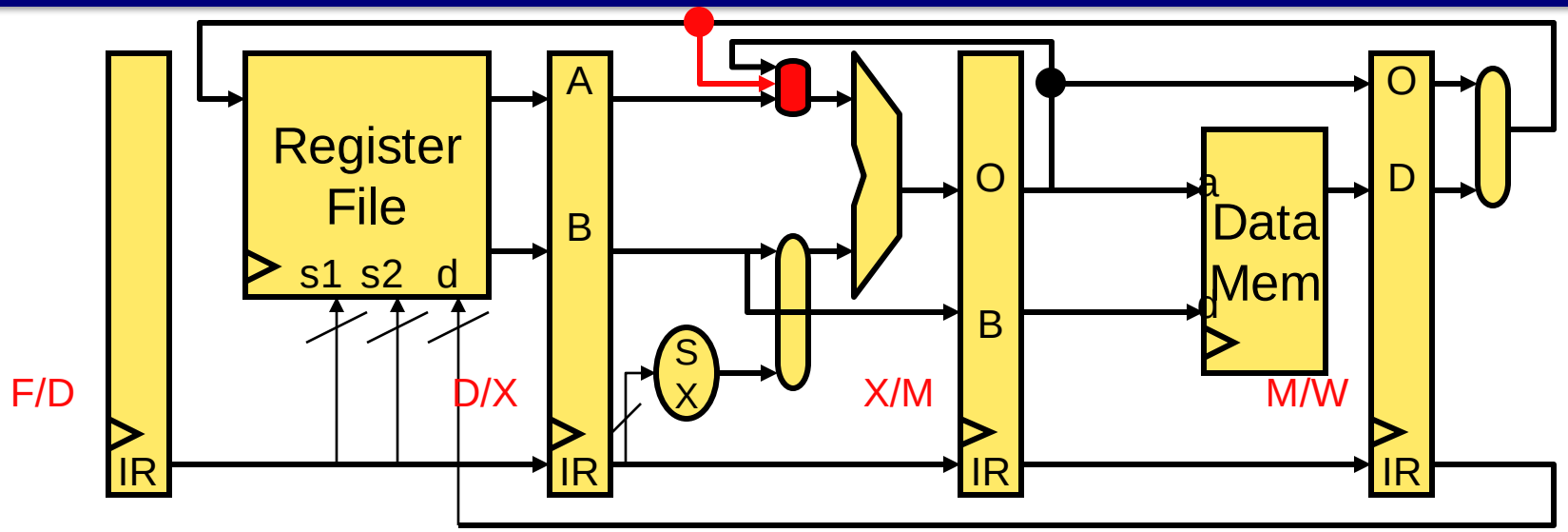
- This situation seems broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But fundamentally, everything is still OK
  - `lw $4,0($3)` hasn't actually used `$3` yet (nothing written yet)
  - `add $3,$2,$1` has already computed `$3`

Bypassing



- **Bypassing**
  - Reading a value from an intermediate (microarchitectural) source
  - Not waiting until it is available from primary source (RegFile)
  - Here, we are bypassing the register file
  - Also called **forwarding**

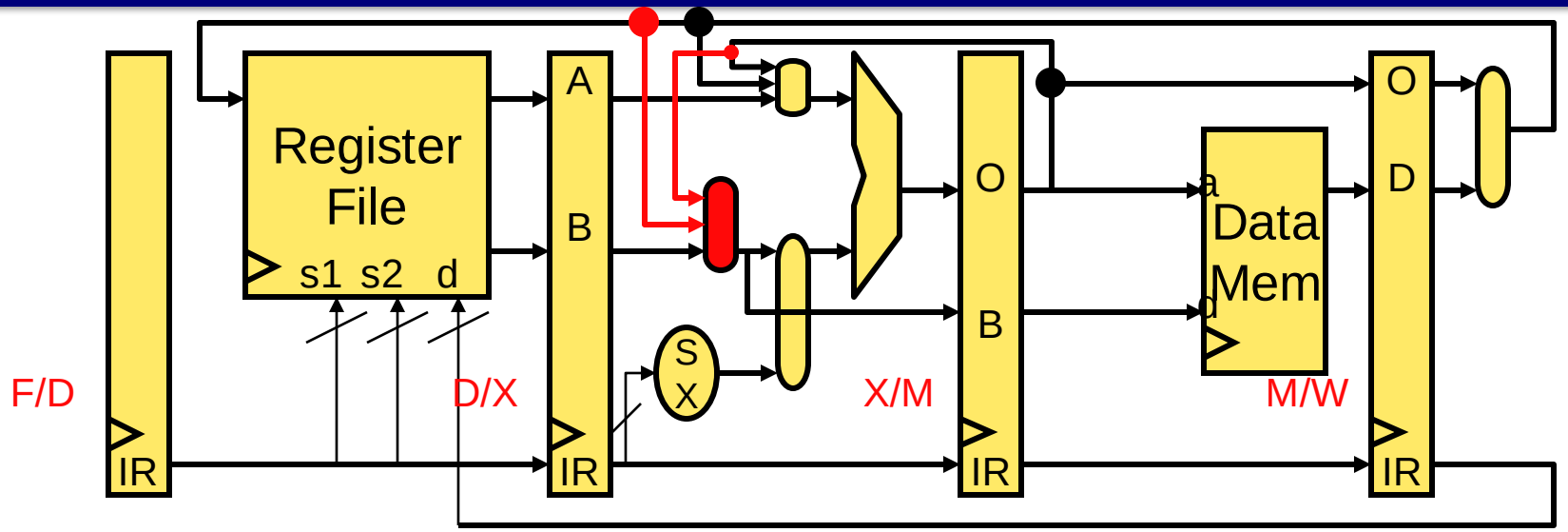
WX Bypassing



`lw $4,0($3)` `add $3,$2,$1`

- What about this combination?
  - Add another bypass path and MUX input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

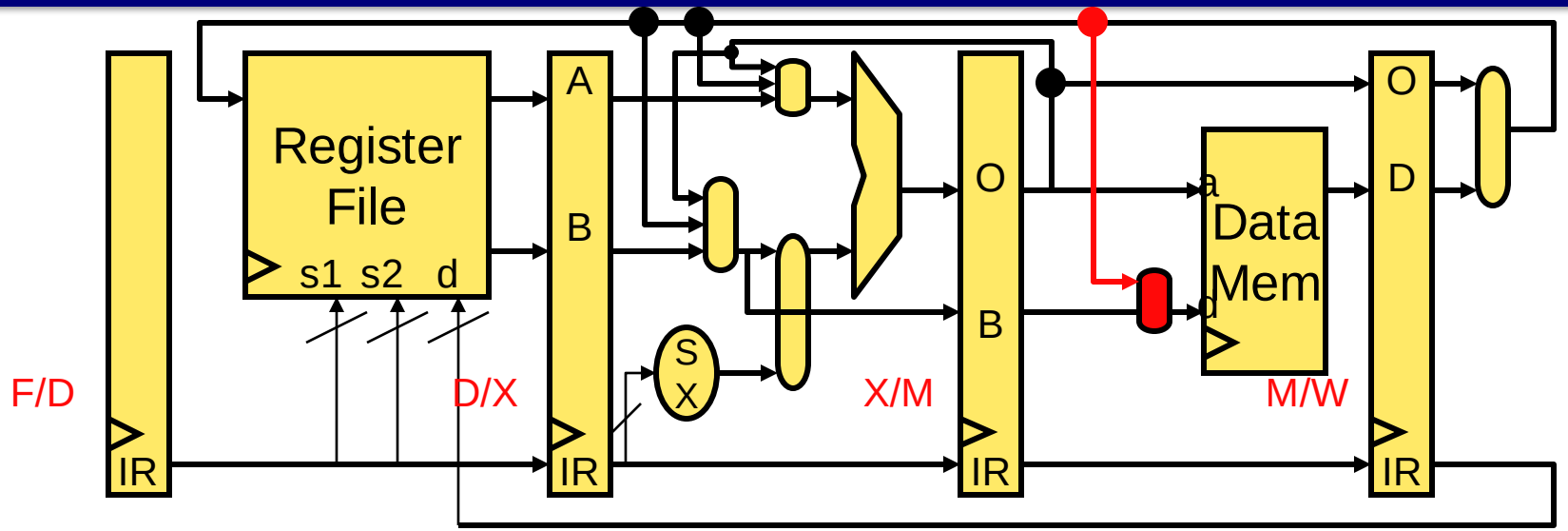
ALUinB Bypassing



`add $4,$2,$3` `add $3,$2,$1`

- Can also bypass to ALU input B

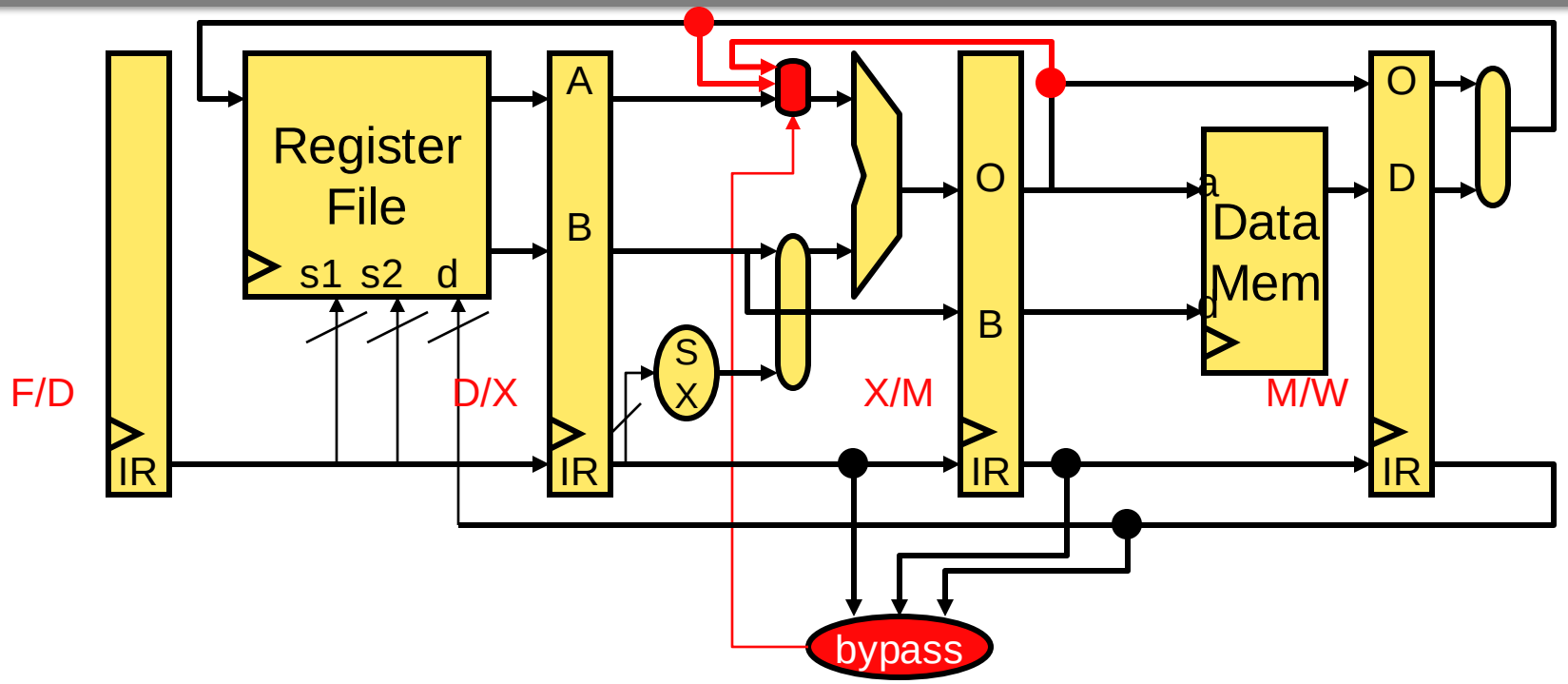
WM Bypassing?




```
sw $3,0($4)        lw $3,0($2)
```

- Does WM bypassing make sense?
  - To the data input?
    - Yes
  - What about to the address input?
    - No. Address is **computed** at X stage (reg value + immediate)

Bypass Logic



- Each mux has its own control
- E.g., for ALUinA mux:

  (D/X.IR.RS1 == X/M.IR.RD) → mux select = 0

  (D/X.IR.RS1 == M/W.IR.RD) → mux select = 1

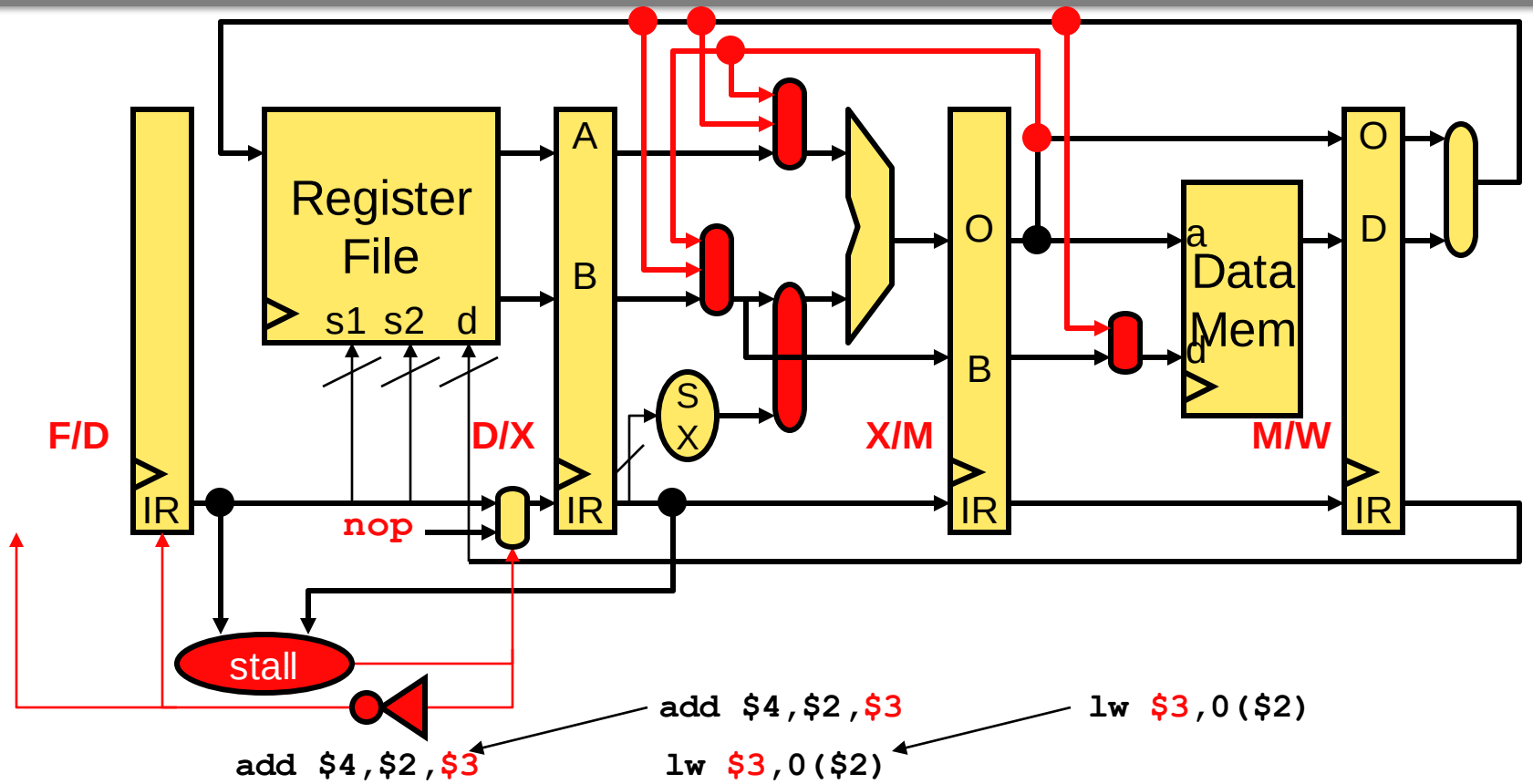
  Else → mux select = 2

Bypass and Stall Logic

- Two separate things
  - Stall logic controls pipeline registers
  - Bypass logic controls muxes
- But complementary
  - For a given data hazard: if we can't bypass → must stall

- Slide #46 shows **full bypassing**: all possible bypasses
  - Is stall logic still necessary?
    - Yes

Yes, Load Output to ALU Input

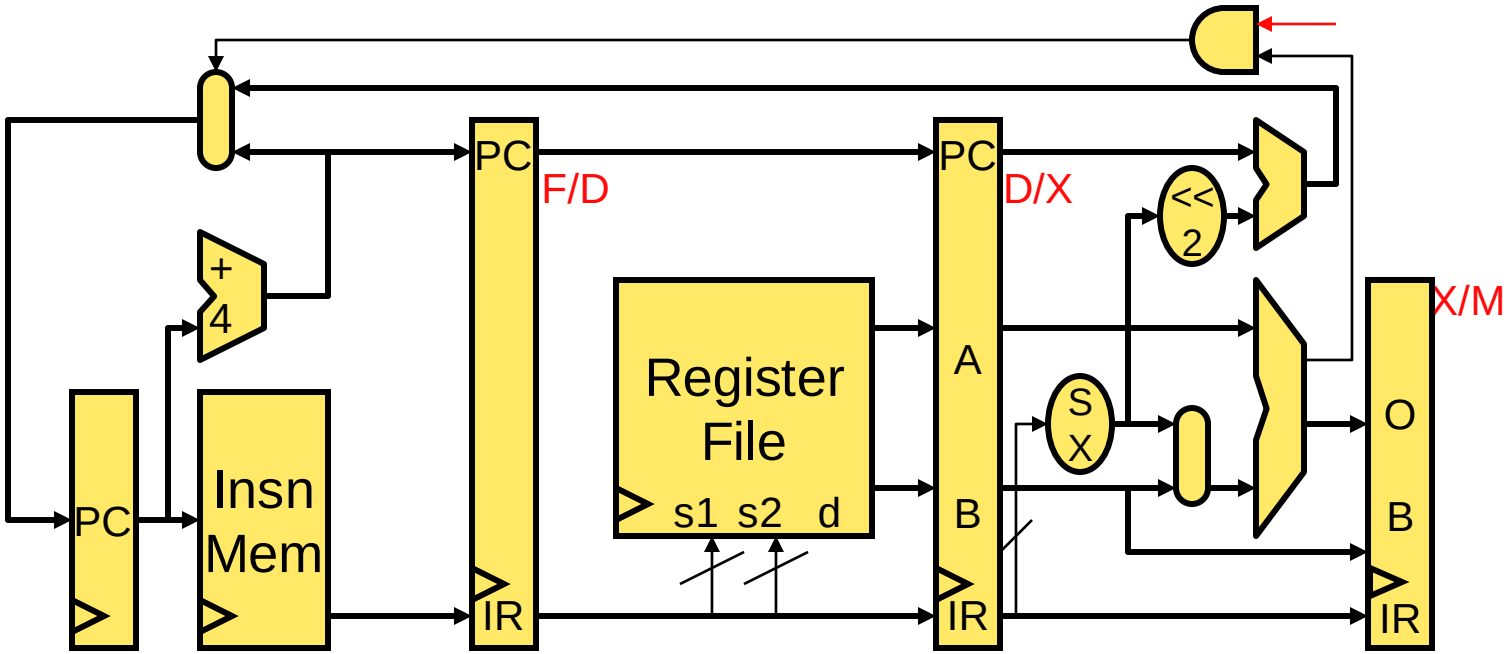



```
Stall = (D/X.IR.OP==LOAD) && (
          (F/D.IR.RS1==D/X.IR.RD) ||
          ((F/D.IR.RS2==D/X.IR.RD) && (F/D.IR.OP!=STORE))
        )
```

Intuition: "Stall if it's a load where rs1 is a data hazard for the next instruction, or where rs2 is a data hazard in a *non-store* next instruction". This is because rs2 is safe in a store instruction, because it doesn't use the X stage, and can be M/W bypassed.

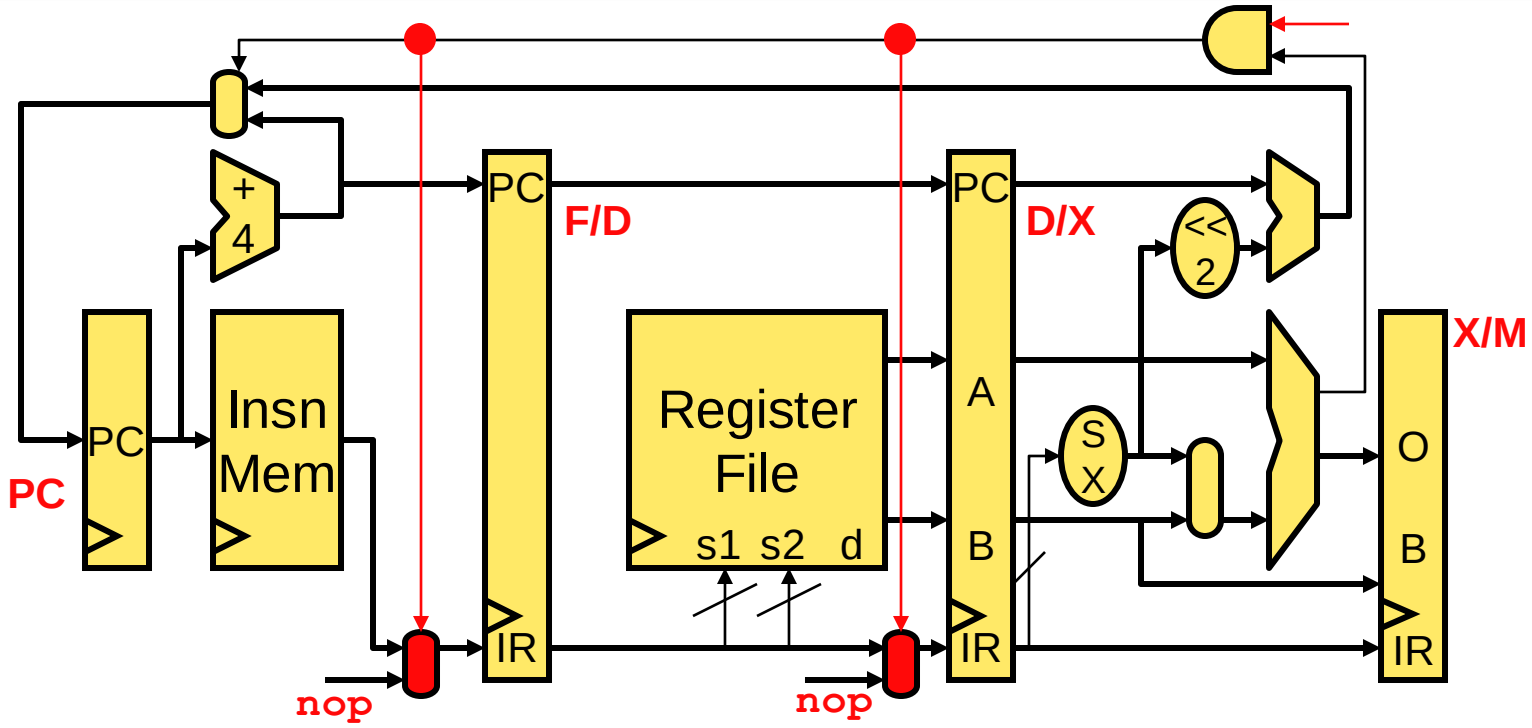
Control Hazards

Control Hazards



- **Control hazards**
  - Must fetch post-branch insns before branch outcome is known
    - Since we don't know whether or not we're branching until the insn reaches the **X stage** (ALU)
  - Default: assume "branch **not taken**" (at fetch, can't tell it's a branch)

Branch Recovery



- **Branch recovery**: what to do when branch **is** taken
  - **Flush** insns currently in F/D and D/X (they're wrong)
    - Replace with **NOPs**

    + Haven't yet written to RegFile or DMem

Branch Recovery Pipeline Diagram

	1	2	3	4	5	6	7	8	9
`addi $3,$0,1`	F	D	X	M	W				
`bnez $3,targ`		F	D	**X**	M	W			
~~`sw $6,4($7)`~~			~~F~~	~~D~~					
~~`addi $8,$7,1`~~				~~F~~					
`targ: sw $6,4($7)`					**F**	D	X	M	W

- Control hazards sometimes indicated with **c***
  - Penalty for taken branch is 2 cycles

	1	2	3	4	5	6	7	8	9
`addi $3,$0,1`	F	D	X	M	W				
`bnez $3,targ`		F	D	X	M	W			
`sw $6,4($7)`			**c***	**c***	F	D	X	M	W

Branch Performance

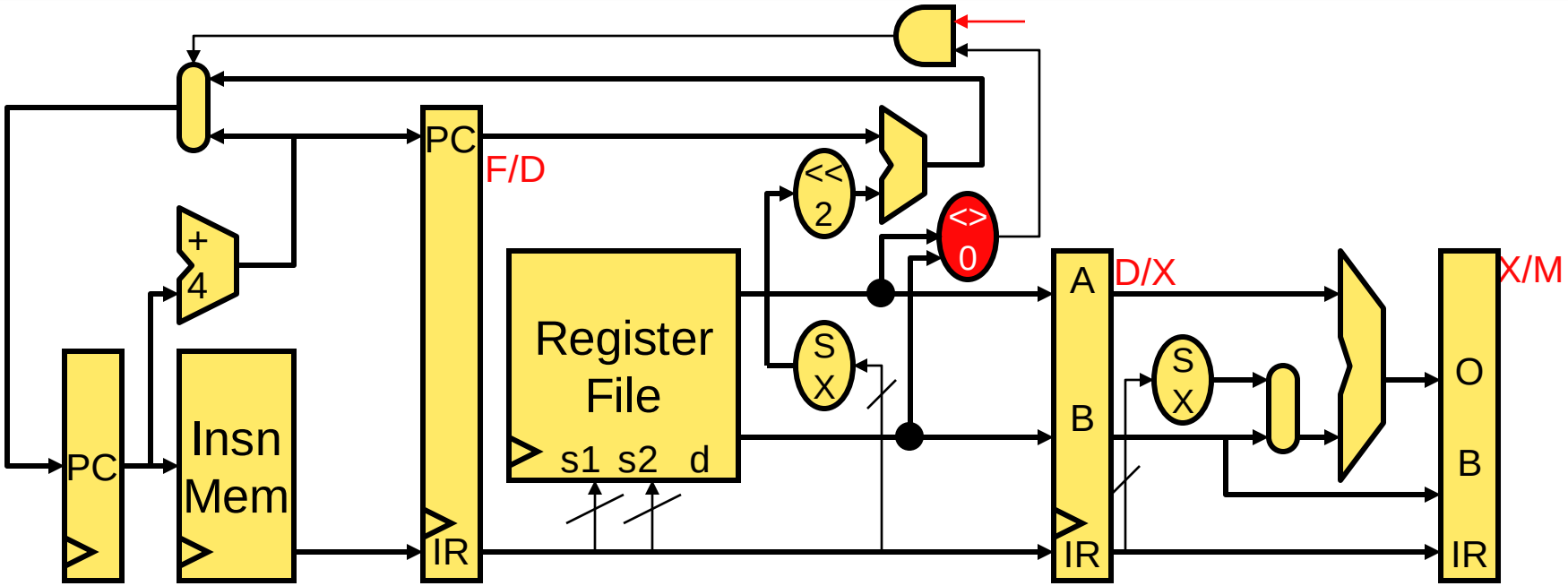
- Again, measuring effect on CPI (clock period is fixed)

- Approximate calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - **75% of branches are taken**
    - **How come?**

- CPI if no branches = 1
- CPI with branches = 1 + 0.20*0.75*2 = 1.3
  - **Branches cause 30% slowdown**
  - How do we reduce this penalty?

Can We Perform Better w.r.t. Control Hazards?

Fast Branch



- **Fast branch**: can decide at D instead of X
  - Duplicate comparison logic only, not the whole ALU
  - \+ New taken branch penalty is now **1** stall instead of 2
  - – Additional insns (`slt`) for more complex tests, must bypass to D too
  - 25% of branches have complex tests that require extra insn
  - CPI = 1 + 0.20*0.75*1(branch) + 0.20*0.25*1(extra insn) = 1.2

Speculative Execution

- Speculation: "risky transactions on chance of profit"

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    + Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    – Must abort/flush/squash incorrect insns
    – Must undo incorrect changes (recover pre-speculation state)
  - The "game": **[$\%_{correct}$ * gain] – [(1–$\%_{correct}$) * penalty]**

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?

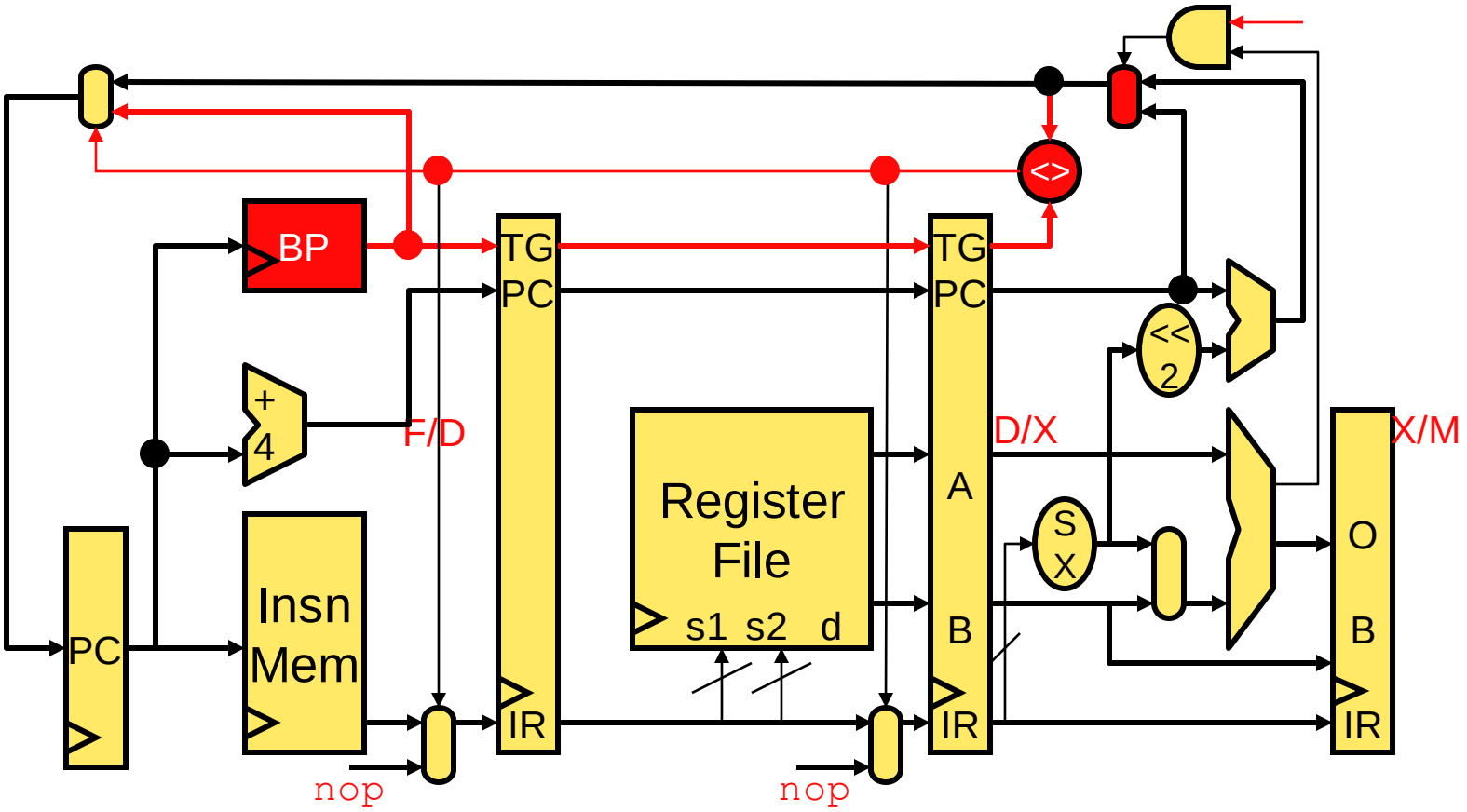
Control Speculation Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is PC+4
  - Can actively guess other targets: **dynamic branch prediction**

- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated insns
    - Hopefully haven't modified permanent state (Regfile, DMem)
    + Happens naturally in in-order 5-stage pipeline

- "Game" for in-order 5 stage pipeline
  - $\%_{correct}$ = ?
  - Gain = 2 cycles
  + Penalty = 0 cycles → **mis-speculation no worse than stalling**

Dynamic Branch Prediction



- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction** (notice new recovery circuit)

Branch Prediction: Short Summary

- Key principle of micro-architecture:
  - Programs do the same thing over and over (why?)
  - Exploit for performance:
    - Learn what a program did before
    - Guess that it will do the same thing again
- Inside a branch predictor: the short version
  - Use some of the PC bits as an **index** to a separate RAM
  - This RAM contains (a) branch destination and (b) whether we predict the branch will be taken
  - RAM is updated with results of past executions of branches
  - Algorithm for predictions can be simple ("assume it's same as last time"), or get quite fancy

Branch Prediction Performance

- Same parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken
- Dynamic branch prediction
  - Assume branches predicted with 75% accuracy (so 25% are penalized)
  - CPI = 1 + 0.20*0.25*2 = **1.1**
- Branch (esp. direction) prediction was a hot research topic
  - Accuracies now 90-95%

Pipelining And Exceptions

- Remember exceptions?
  - Pipelining makes them nasty
  - 5 instructions in pipeline at once
  - Exception happens, how do you know which instruction caused it?
    - Exceptions propagate along pipeline in latches
  - Two exceptions happen, how do you know which one to take first?
    - One belonging to oldest insn
  - When handling exception, have to flush younger insns
    - Piggy-back on branch mis-prediction machinery to do this
- → take ECE 552

Pipeline Depth

- No magic about 5 stages, trend had been to deeper pipelines
  - 486: 5 stages (50+ gate delays / clock)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (~10 gate delays / clock) **"super-pipelining"**
  - Core1/2: 14 stages

- Increasing **pipeline depth**
  - \+ Increases clock frequency (reduces period)
  - – But decreases IPC (increases CPI)
  - Branch mis-prediction penalty becomes longer
  - Non-bypassed data hazard stalls become longer
  - At some point, CPI losses offset clock gains, question is when?
    - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
    - What was the point?
      - People buy frequency, not frequency*IPC (throughput)

Real pipelines…

- Real pipelines fancier than what we have seen
  - Superscalar: multiple instructions in a stage at once
  - Out-of-order: re-order instructions to reduce stalls
  - SMT: execute multiple threads at once on processor
    - Side by side, sharing pipeline resources
  - Multi-core: multiple pipelines on chip
    - Cache coherence: No stale data

ECE 550D
Fundamentals of Computer Systems and Engineering

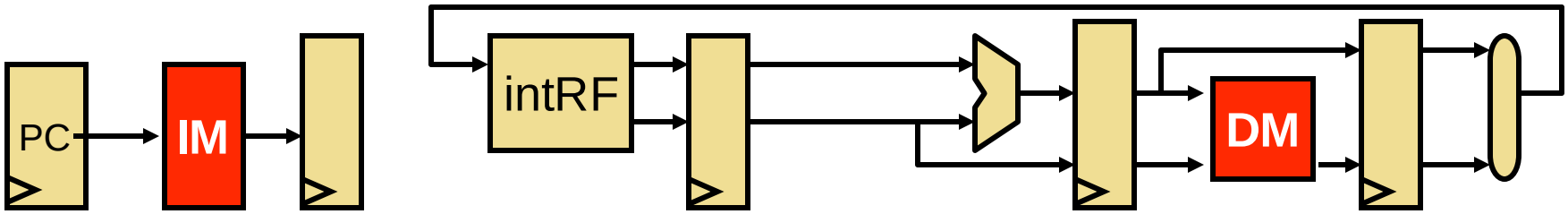
Fall 2024

Memory Hierarchy

Rabih Younes
Duke University

Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

How Do We Build Insn/Data Memory?

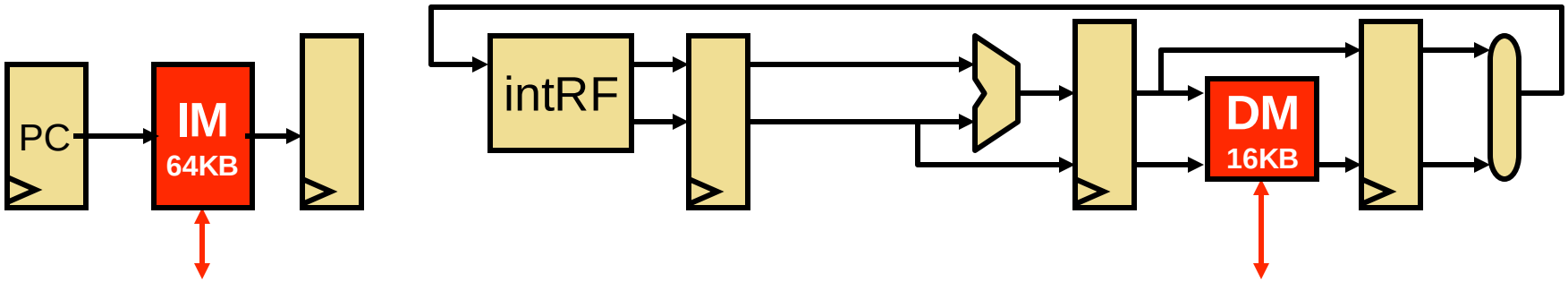


- Fast memories are usually designed using **SRAM** technology
- Register file? Just a multi-ported SRAM
  - 32 x 32-bit int registers = 2^5 * 2^5 b = 1Kb = 128B
  - Multiple ports make it bigger and slower but still OK

Port = address+data bus pair

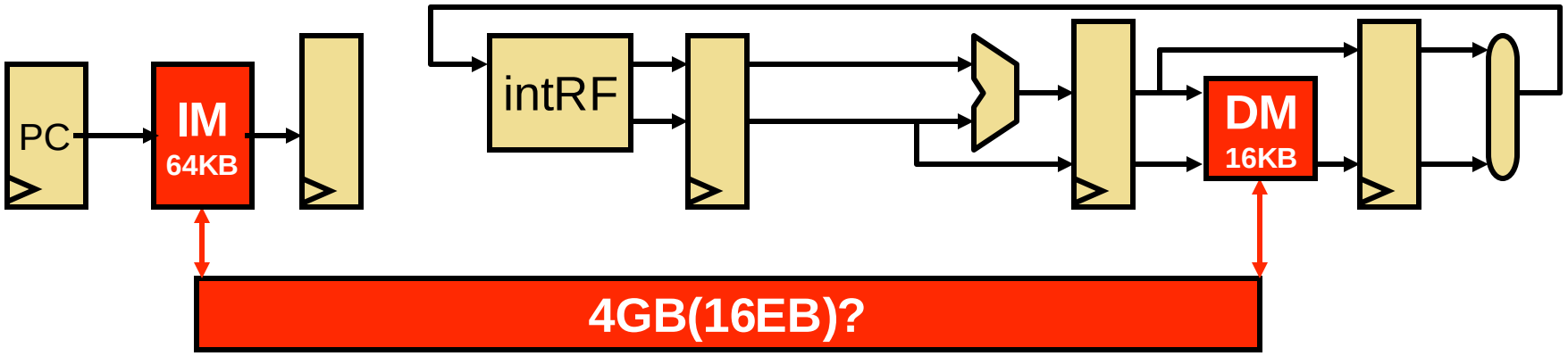
- Insn/data memory? Just a single-ported SRAM?
  - In 32-bit systems, it's up to 2^{32}B = **4GB!**
  - It would be huge, expensive, and pointlessly slow
  - And we can't build something that big on-chip anyway
  - Most ISAs now are 64-bit, so memory is 2^{64}B = **16EB!** (EB: exabyte)
    - Kilo < Mega < Giga < Tera < Peta < Exa

So, What Do We Do? Actually…



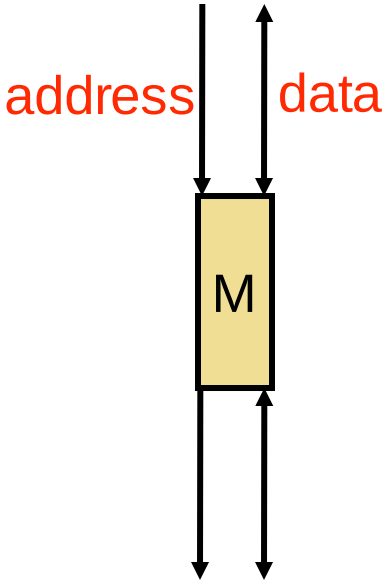
- "Primary" insn/data memory are single-ported SRAMs…
  - "primary" = "in the datapath"
  - Key 1: they contain only a dynamic subset of "memory"
    - Subset is small enough to fit in a reasonable SRAM
  - Key 2: missing chunks fetched on demand (transparent to program)
    - From somewhere else… (next slide)
  - Program has the **illusion** that all 4GB (or 16EB) of memory is physically there
    - Just like it has the illusion that all insns execute atomically

But…



- If requested insn/data is not found in primary memory
  - Doesn't the place it comes from have to be a 4GB (or 16EB) SRAM?
  - And won't it be huge, expensive, and slow?
  - And can we build it?
  - *We'll answer these questions in this and the following lecture (Virtual Memory)*

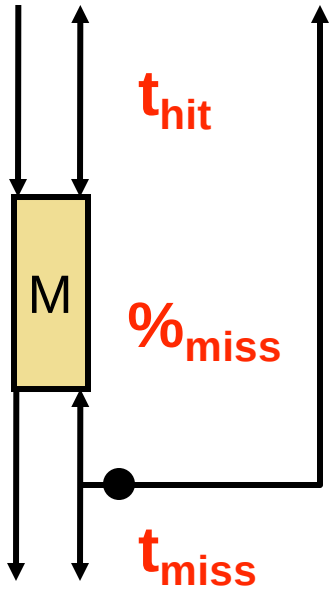
Memory Overview



- Functionality
  - "Like a big array…"
  - N-bit address bus (on N-bit machine)
  - Data bus: typically read & write on same bus
  - Can have multiple ports: address+data bus pairs

- Access time:
  - Access latency ~ #bits * #ports^2

Memory Performance Equation



- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get data from another component
    - *No notion of "miss" in the register file*
  - **Fill**: action of placing data in M

  - **$\%_{miss}$** (miss-rate): #misses / #accesses
  - **t_{hit}**: time to read data from (or write data to) M
  - **t_{miss}**: time to read data into M (get data to M)

- Performance metric: average access time

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

Why not: $t_{avg} = \%_{hit} * t_{hit} + \%_{miss} * t_{miss}$?

Memory Hierarchy

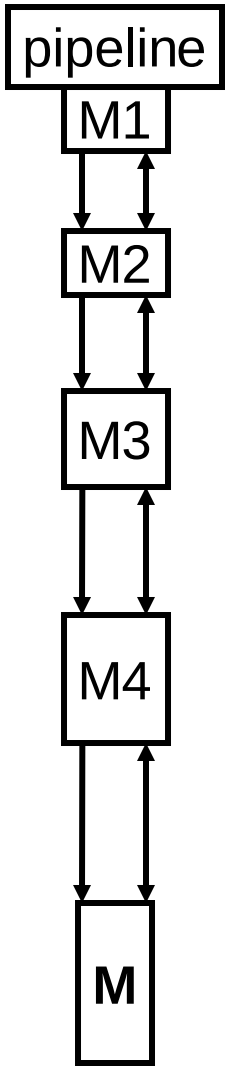
$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

- **Problem:** hard to get low t_{hit} and $\%_{miss}$ in one structure
  - **Large structures** have low $\%_{miss}$ but high t_{hit}
  - **Small structures** have low t_{hit} but high $\%_{miss}$

- **Solution:** use a **hierarchy** of memory structures

> *"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."*
>
> *Burks, Goldstine, VonNeumann*
>
> *"Preliminary discussion of the logical design of an electronic computing instrument"*
>
> *IAS memo,* ***1946***

Abstract Memory Hierarchy

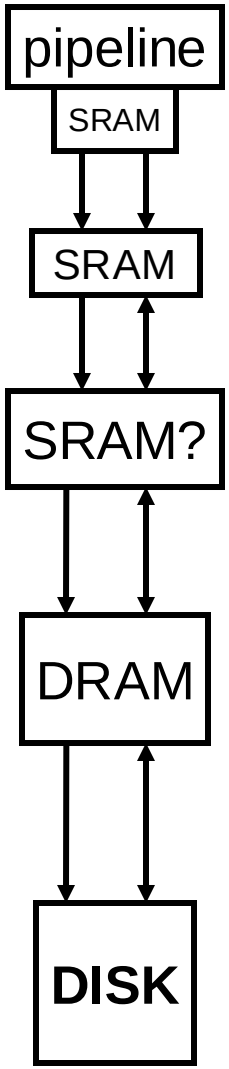


- Hierarchy of memory components
  - Upper levels: small → low t_{hit}, high $\%_{miss}$
  - Going down: larger → higher t_{hit}, lower $\%_{miss}$
- Connected by buses
- Make average access time close to M1's
  - How?
    - Most frequently accessed data in M1
    - M1+next most frequently accessed in M2
    - …
    - …
    - Automatically move data up/down the hierarchy as needed

Why Memory Hierarchy Works

- **10/90 rule (of thumb)**
  - 10% of static insns/data account for 90% of accessed insns/data
    - Insns: inner loops
    - Data: frequently used globals, inner loop stack variables
- **Temporal locality**
  - Recently-accessed insns/data are likely to be accessed again soon
    - Insns: inner loops (next iteration)
    - Data: inner loop local variables, globals
  - Hierarchy can be **"reactive"**: move things up when accessed
- **Spatial locality**
  - Insns/data near recently-accessed insns/data are likely accessed soon
    - Insns: sequential execution
    - Data: elements in array, fields in struct, variables in stack frame
  - Hierarchy is **"proactive"**: move things up speculatively

Exploiting Heterogeneous Technologies

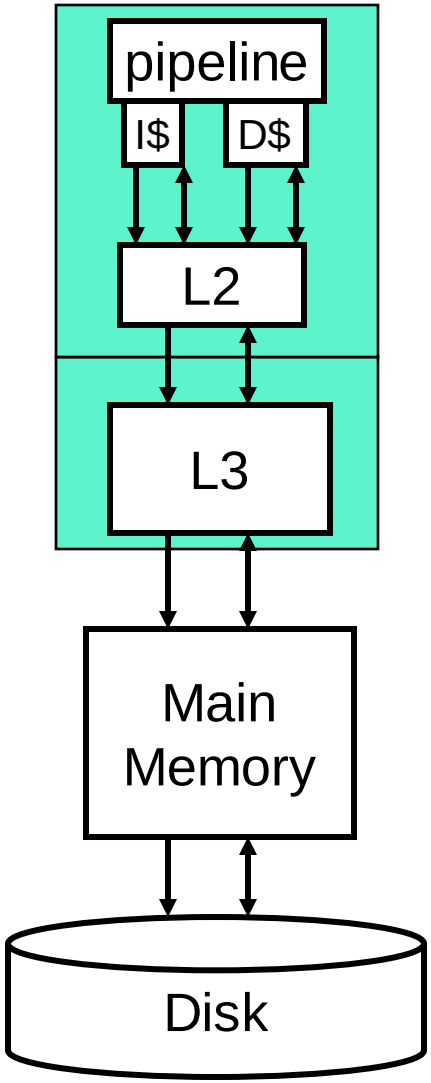


- Apparent problem:
  - Lower-level components must be huge
  - Huge SRAMs are difficult to build and expensive
- Solution: don't use SRAM for lower levels
  - Use cheaper, denser storage technologies
  - Will be slower than SRAM, but that's OK
    - Won't be accessed very frequently
    - We have no choice anyway
  - Upper levels: SRAM → expensive (per Byte), fast
  - Going down: DRAM+Disk → cheaper (per B), slower

Memory Technology Overview

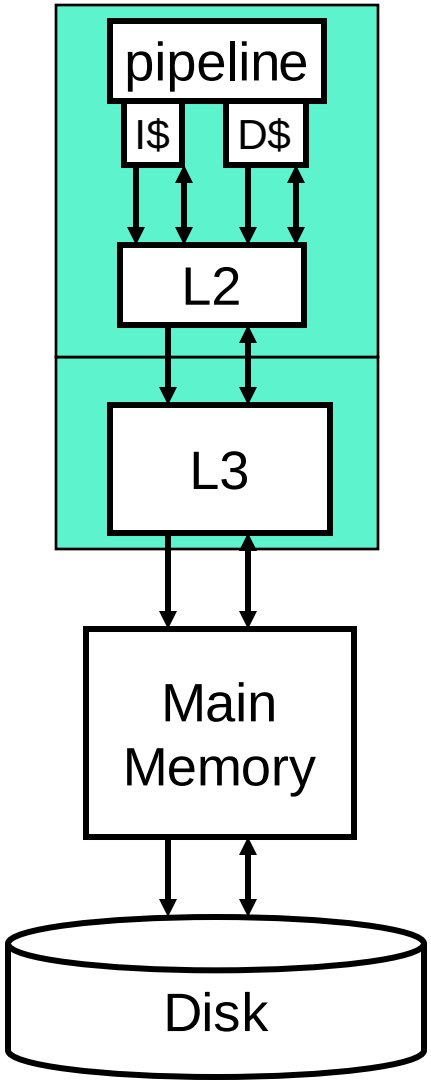
- **Latency**
  - SRAM: <1 to 5ns (on chip)
  - DRAM: ~100ns — 100x or more slower than SRAM
  - Disk: 10,000,000ns or 10ms — 100,000x slower than DRAM
  - Flash: ~200ns — 2x slower than DRAM (read, much slower writes)
- **Bandwidth**
  - SRAM: 10-100GB/sec
  - DRAM: ~1GB/sec — 10x less than SRAM
  - Disk: 100MB/sec (0.1 GB/sec) — sequential access only
  - Flash: about same as DRAM for read (much less for writes)
- **Cost**: what can ~$50 buy?
  - SRAM: 32MB
  - DRAM: 4,000MB (4GB) — 125x cheaper than SRAM
  - Disk: 1,000,000MB (1TB) — 250x cheaper than DRAM
  - Flash: 256,000 MB (256GB) with latest USB tech — 64x cheaper than DRAM

(Traditional) Concrete Memory Hierarchy



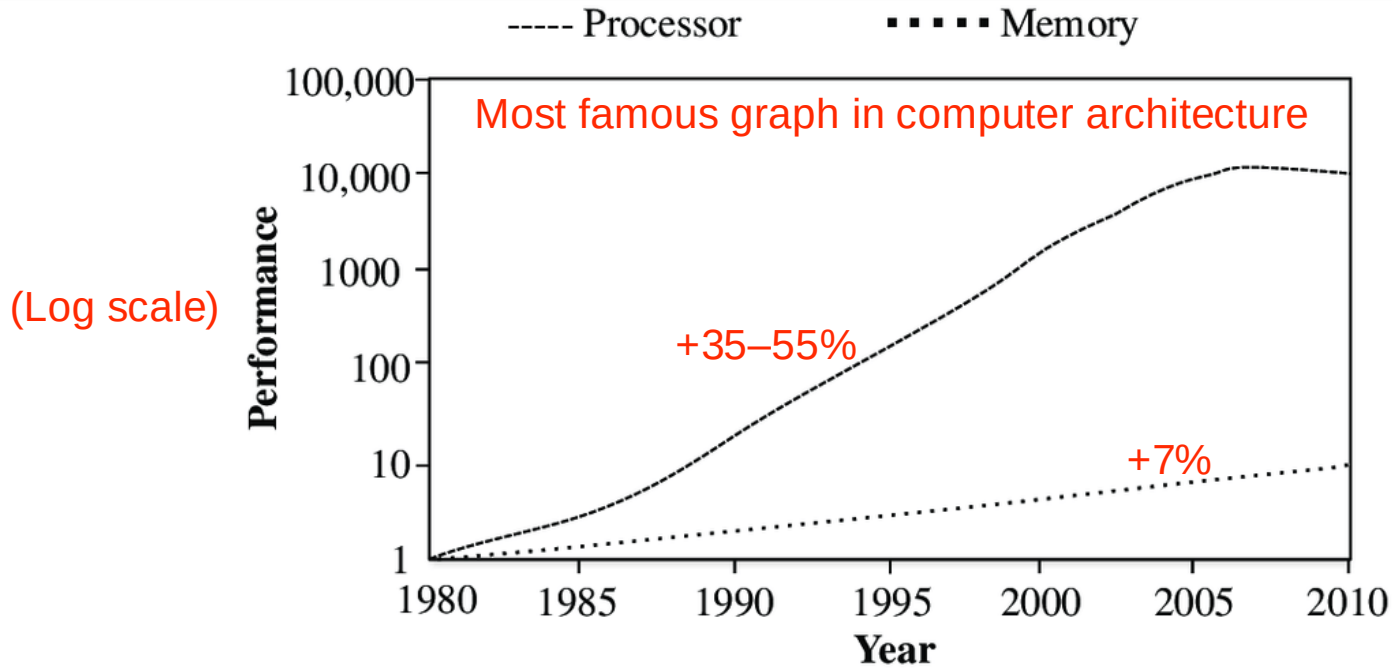
- 1st level: **I$**, **D$** (insn/data caches)
  - $ = cache
- 2nd level: **L2** (cache)
  - On-chip, certainly on-package (with CPU)
  - Made of SRAM
- 3rd level: **L3** (cache)
  - Same as L2, may be off-chip
- **Main memory**
  - Off-chip
  - Made of DRAM
- **Disk**
  - Electrical (SSD) or electrical-mechanical (HDD)

Virtual Memory Teaser



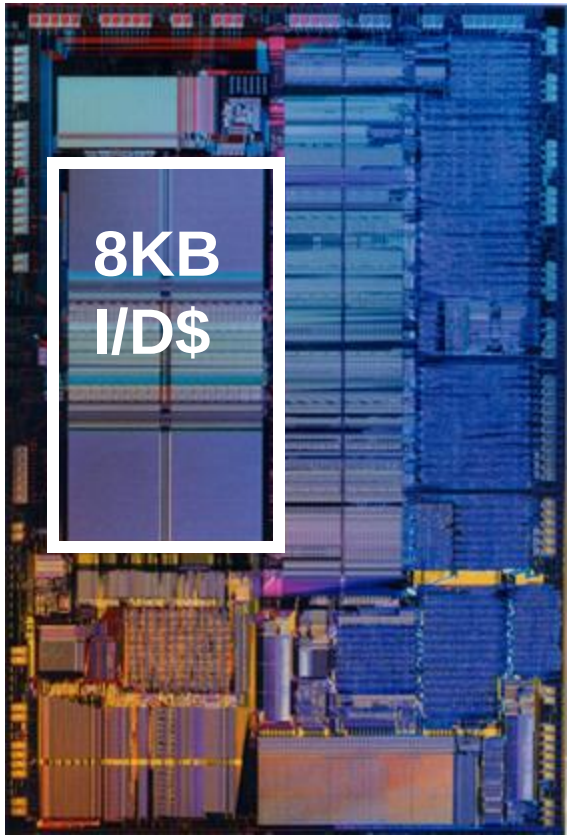
- For 32-bit ISA:
  - 4GB main memory and disk are doable
- What about 64-bit ISA?
  - 16EB main memory is right out
  - Even 16EB disk is extremely difficult
- →We need virtual memory
  - We'll study it next lecture
- Let's start with caches…

Why Are There 2-3 Levels of Cache?

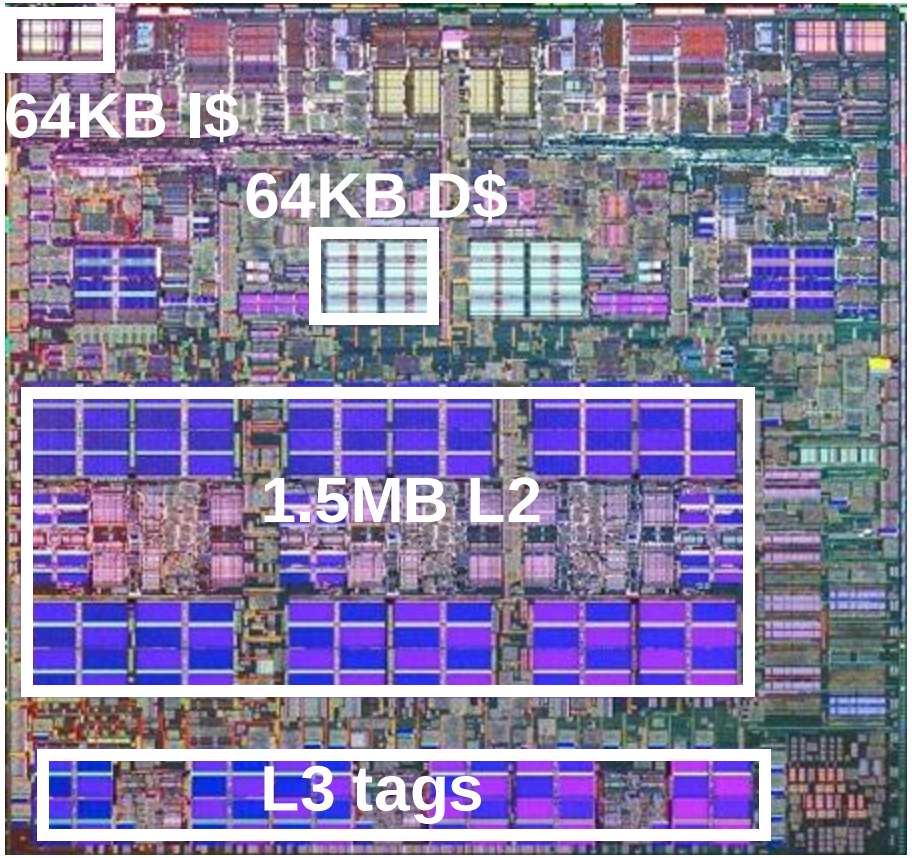


- **"Memory Wall"**: memory 100X slower than primary caches
  - Multiple levels of cache needed to bridge the difference
- Also, **"Disk Wall?"**: disk is 100,000X slower than memory
  - Why aren't there 56 levels of main memory to bridge that difference?
  - Doesn't matter: program can't keep itself busy for 10M cycles
  - So slow, may as well swap out and run another program

Evolution of Cache Hierarchies



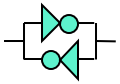
Intel 486, 1989



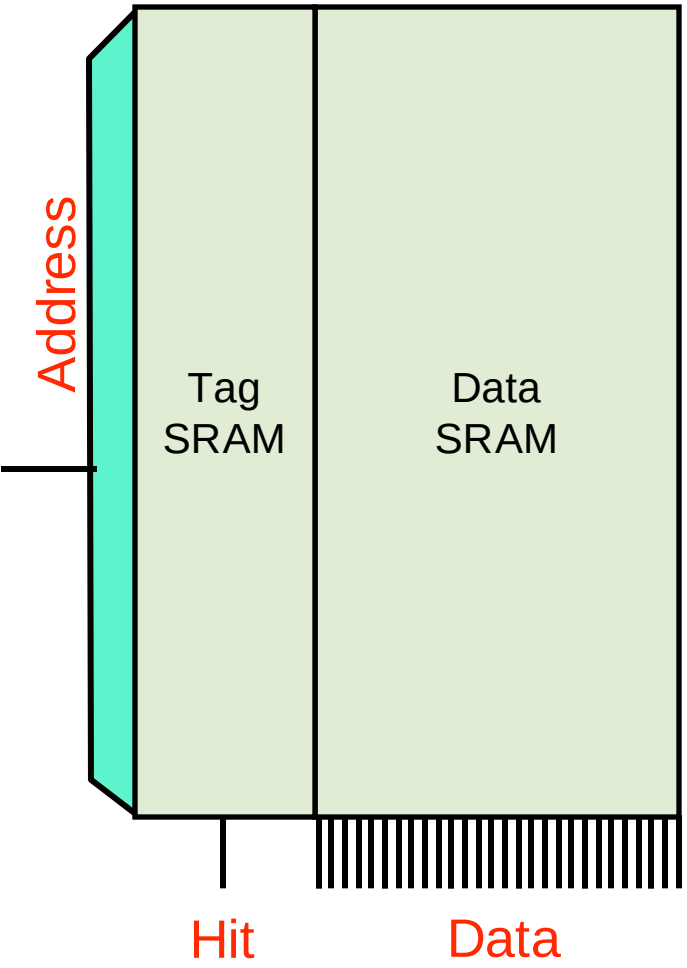
IBM Power5 (dual core), 2004

Processor chips today: 30–70% cache (by area)

RAM and SRAM

- Reality: large storage arrays implemented in "analog" way
  - Not as flip-flops (FFs) + giant muxes
- **RAM (random access memory)**
  - Ports implemented as shared buses called wordlines/bitlines
- **SRAM: static RAM**
  - Static = bit maintains its value indefinitely, as long as power is on
  - Bits implemented as cross-coupled inverters (CCIs) 
  + 2 gates, 4 transistors per bit
  - All processor storage arrays: regfile, caches, branch predictor, etc.
- Other forms of RAM: Dynamic RAM (DRAM), Flash

SRAMS → Caches

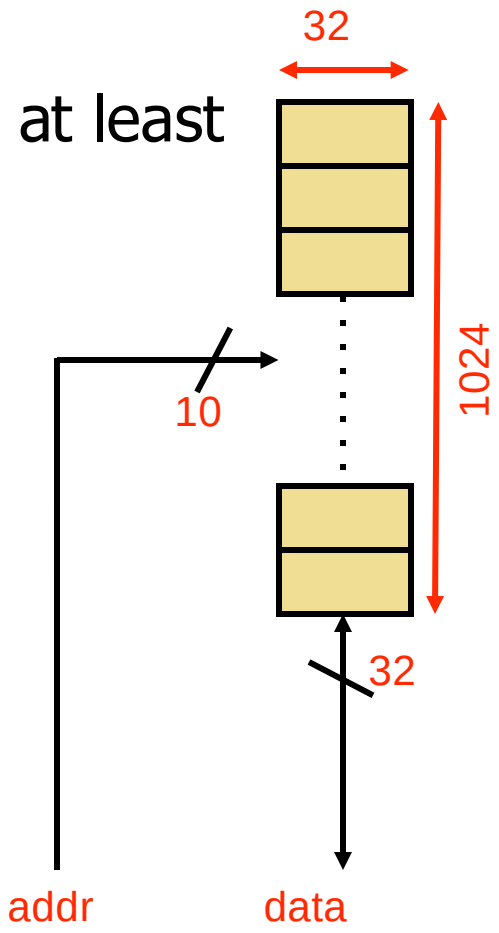


- SRAMs are used to make caches
  - Hold a subset of memory
- Reading:
  - Input: Address to read (32 or 64 bits)
  - Output:
    - Hit? 1-bit: was it there?
    - Data: if there, requested value

Cache Design Basics

- **Block**: basic unit of data in cache
  - Data will move in blocks between memory hierarchy units

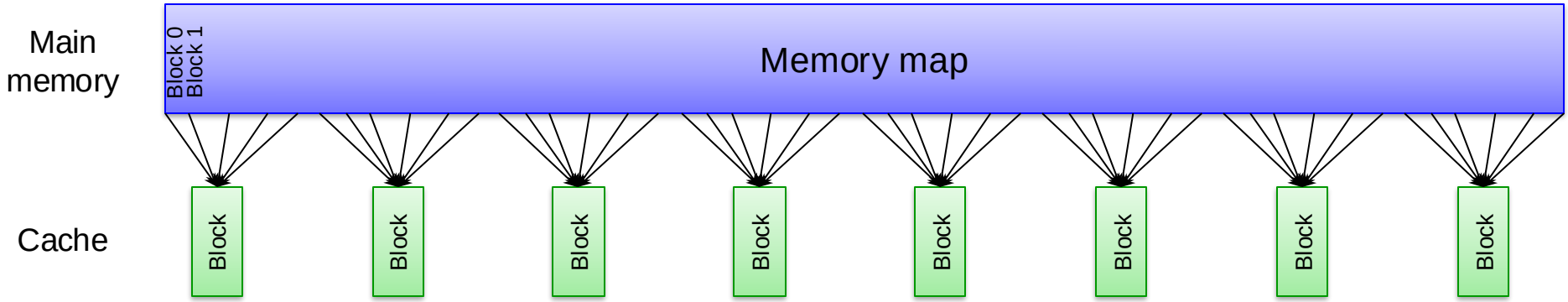
- For a 32-bit systems, blocks be able to hold at least 32 bits (i.e., 4B or 1 word)
  - E.g., a 4KB cache could be designed as follows:
    - 1K (1024) blocks x 4B per block
    - →10-bit address input
      - To index all 1K blocks
    - 32-bit data input/output



Mapping: Which Bits to Use for Indexing Blocks?

- **Not** the lowest $\log_2$(block_size) bits: those tell us which byte in the block we're looking for (since memory is byte-addressable).
- Of the remaining bits, do we pick the lowest ones or the highest ones?

- If we pick **highest** bits for index:

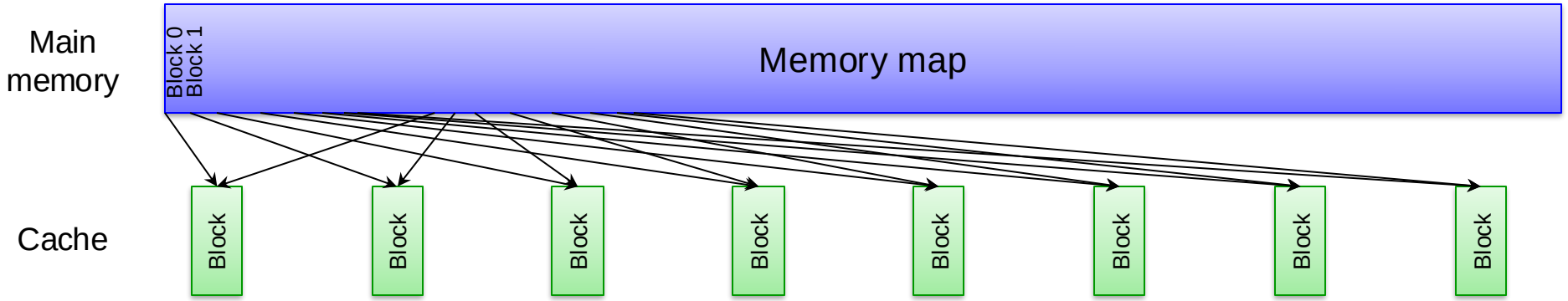


- Neighbor blocks in memory are likely to collide in cache (fight over the same cache block)
- → Opposite of what we want – this penalizes spatial locality
- **→ Bad!**

Mapping: Which Bits to Use for Index?

- **Not** the lowest $\log_2$(block_size) bits: those tell us which byte in the block we're looking for (since memory is byte-addressable).
- Of the remaining bits, do we pick the lowest ones or the highest ones?

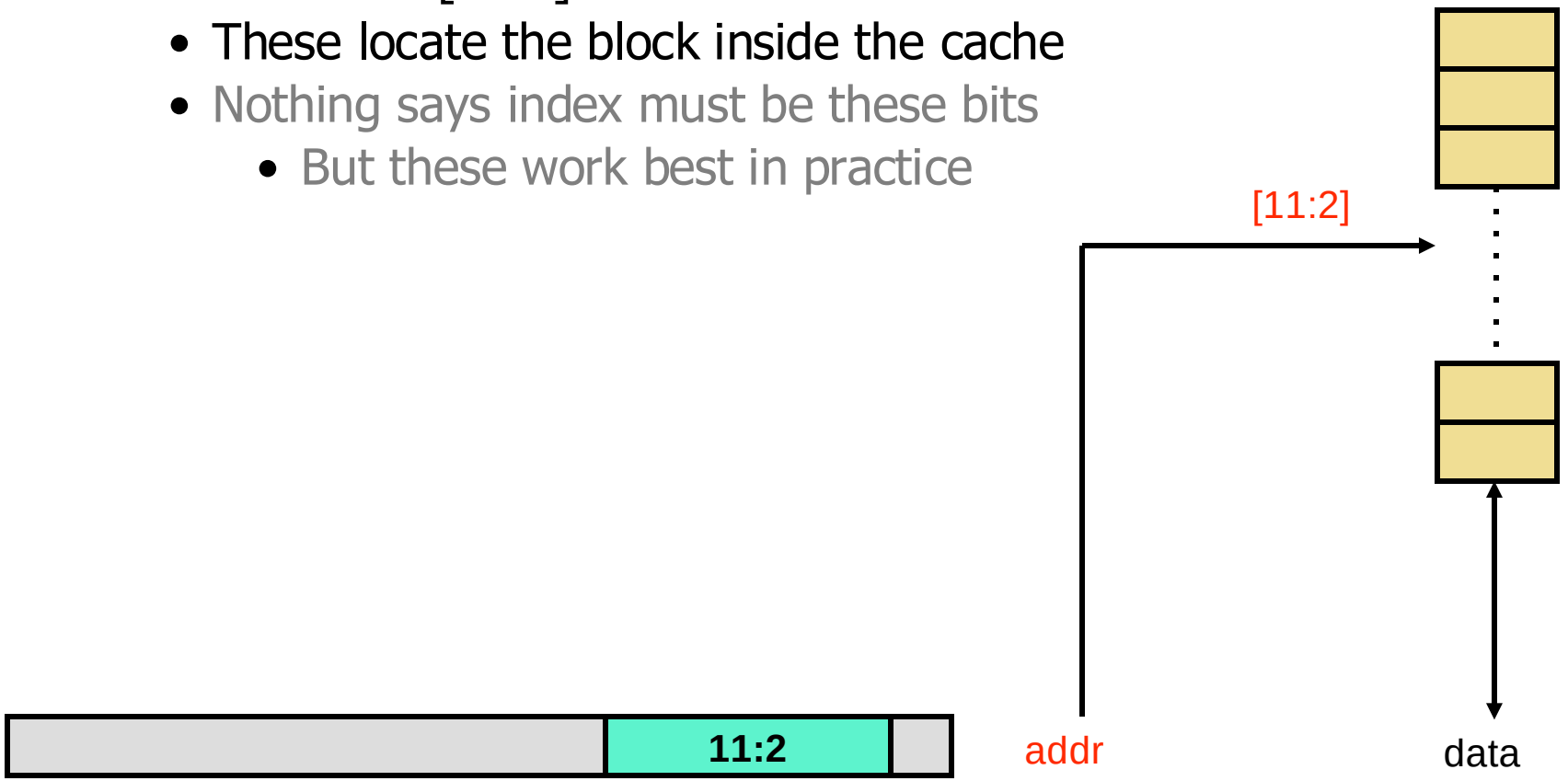
- If we pick **lowest** bits for index:



- Neighboring blocks in memory get neighboring blocks in cache
- → Spatial locality leads to broad use of cache capacity
- **→ Good! → that's what we'll do!**

Mapping a Memory Address in a Cache

- → Based on our previous example (4KB cache with 4B blocks):
  - 2 LSBs [1:0] are called the **offset bits**
    - They locate the designated byte within a block
  - Next 10 LSBs [11:2] are called the **index bits**
    - These locate the block inside the cache
    - Nothing says index must be these bits
      - But these work best in practice

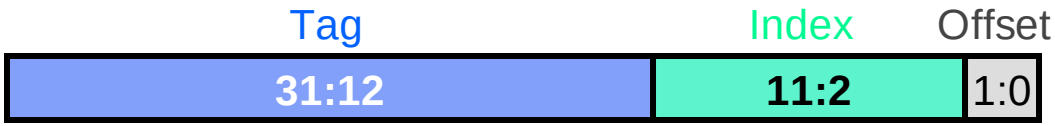


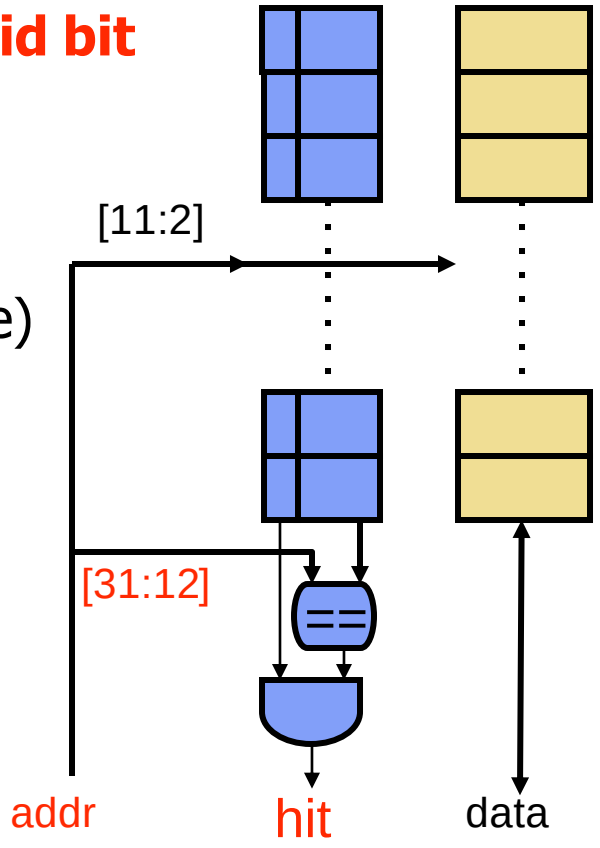
Hit or Miss?

- How do we know if this cache block hold the data we need?
  - 2^{20} (20 = 32-10-2) different addresses map to one particular block
- Build a separate and parallel **tag array**
  - 1K (for each block) x 21-bit SRAM
  - 21 bits = 20-bit **tag** (see next slide) + 1 **valid bit**
- Lookup algorithm:
  - Split address into 3 parts:
    - Offset: least-significant $\log_2$(block-size)
    - Index: next $\log_2$(number-of-blocks/cache)
    - Tag: remaining bits
  - Read cache block tag indicated by index bits
  - If (tag==our tag && valid bit==1)

    ? Hit

    : Miss



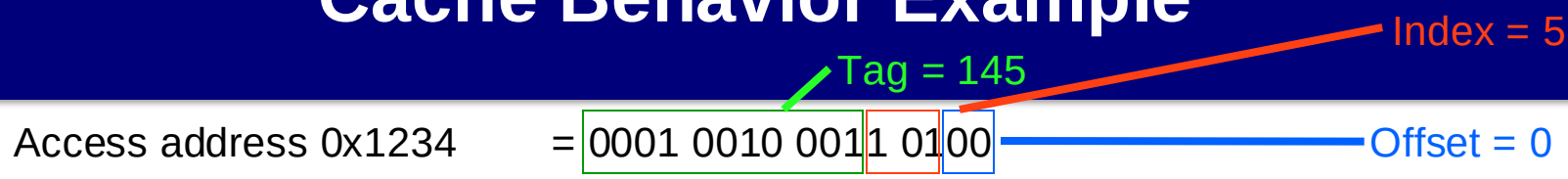


Cache Behavior Example

Cache starts empty (valid = 0), 8 blocks (or sets) in cache, 4B blocks, 16-bit address

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	0	000	00 00 00 00
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Cache Behavior Example

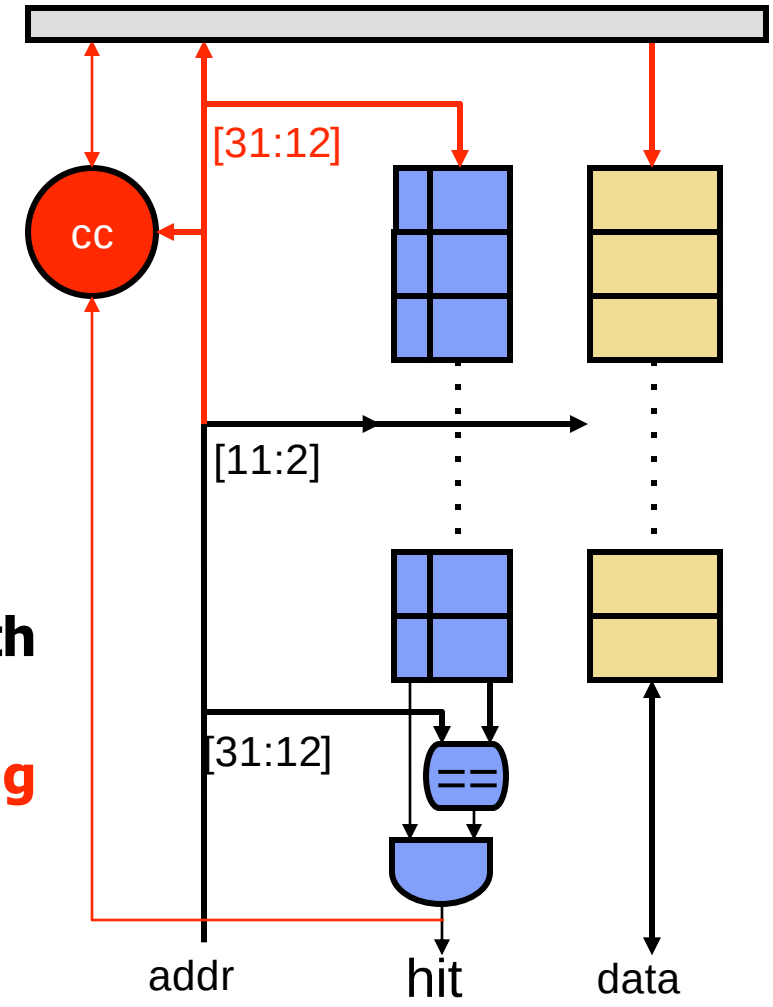


Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	0	000	00 00 00 00
6	0	000	00 00 00 00
7	0	000	00 00 00 00

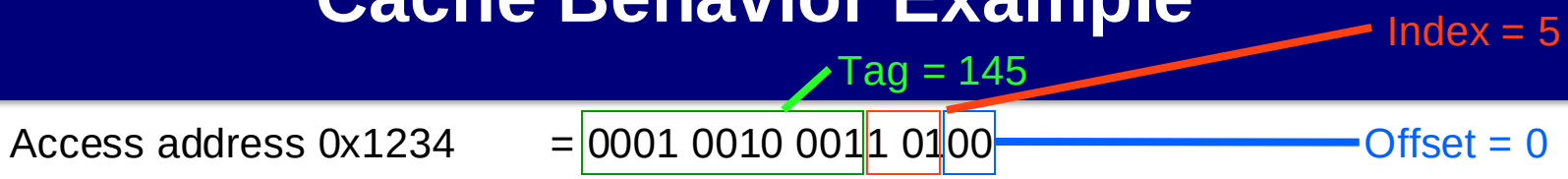
Valid bit == 0 → miss

Handling a Cache Miss

- What if requested word isn't in the cache?
  - How does it get in there?
- **Cache controller**: an FSM
  - Remembers miss address
  - Asks next level of memory
  - Waits for response
  - Writes data/tag into proper locations
  - All of this happens on the **fill path**
  - Sometimes called **backside**
  - **Works in parallel to everything else!**



Cache Behavior Example



Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	145	0F 1E 39 EC
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Valid && tag match → hit

```
lb: 00 00 00 EC
lh: 00 00 39 EC
lw: 0F 1E 39 EC
```

Cache Behavior Example

Tag = 145

Index = 5

Access address 0x1236 = 0001 0010 0011 0110

Offset = 2

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	145	0F 1E 39 EC
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Valid && tag match → hit

```
lb: 00 00 00 1E
lh: 00 00 0F 1E
lw: (unaligned)
```

Cache Behavior Example

Access address 0x1238 = 0001 0010 0011 1000

Tag = 145

Index = 6

Offset = 0

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	145	0F 1E 39 EC
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Not valid → miss

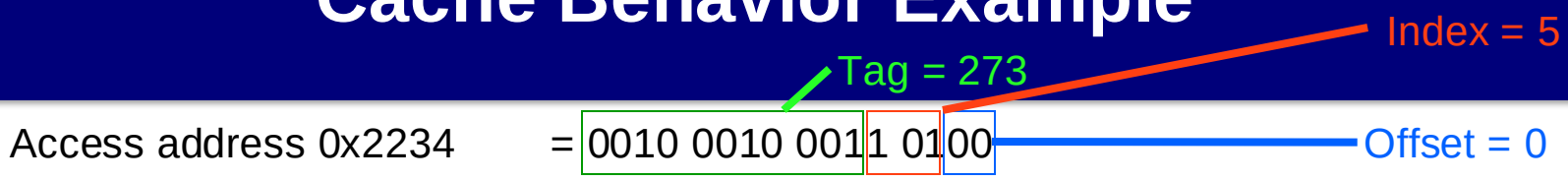
Cache Behavior Example

Access address 0x1238 = 0001 0010 0011 1000

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	145	0F 1E 39 EC
6	**1**	**145**	00 00 00 00
7	0	000	00 00 00 00

Make request to next level...
Wait for it…

Cache Behavior Example



Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	145	0F 1E 39 EC
6	1	145	3C 99 11 12
7	0	000	00 00 00 00

Valid, but tags don't match → miss

Cache Behavior Example

Access address 0x2234 = 0010 0010 0011 0100

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	**273**	0F 1E 39 EC
6	1	145	3C 99 11 12
7	0	000	00 00 00 00

Make request to next level...
Wait for it…

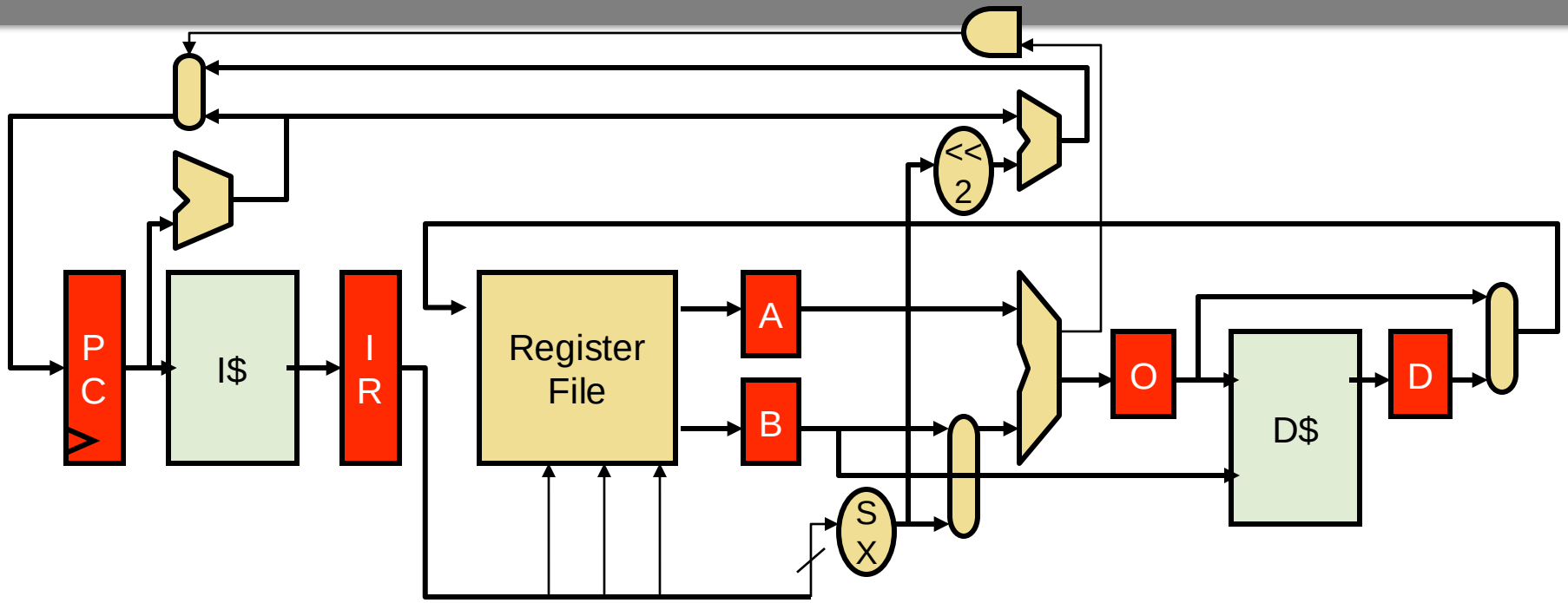
Cache Behavior Example

Access address 0x2234 = 0010 0010 0011 0100

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	273	01 CF D0 87
6	1	145	3C 99 11 12
7	0	000	00 00 00 00

Note that now, 0x1234's data is gone
Replaced by 0x2234's data

Cache Misses and CPI



- I$ and D$ misses stall datapath (multi-cycle or pipeline)
  - Increase CPI
  - Cache hits built into "base" CPI
    - E.g., Loads = 5 cycles in multi-cycle includes t_{hit}
    - Some loads may take more cycles...
      - **Need to know latency of "average" load (t_{avg})**

Measuring Cache Performance

- Ultimate metric is t_{avg}
  - Cache capacity roughly determines t_{hit}
  - Lower-level memory structures determine t_{miss}
  - Measuring $\%_{miss}$:
    - Hardware performance counters (since Pentium)
    - Performance Simulator
    - Paper simulation (like we did earlier)
      - Only works for small caches
      - Small number of requests (would not do for 1M accesses)

Cache Miss Paper Simulation

- 8B cache, 2B blocks

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C				invalid	0	0	1	
E								
8								
3								
8								
0								
8								
4								
6								

- Tag, index, offset?

Cache Miss Paper Simulation

- 8B cache, 2B blocks → 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C				invalid	0	0	1	
E								
8								
3								
8								
0								
8								
4								
6								

- Offset: 1 bit. Index: 2 bits
- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks → 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E				invalid	0	1	1	
8								
3								
8								
0								
8								
4								
6								

- Offset: 1 bit. Index: 2 bits
- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks → 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8				invalid	0	1	1	
3								
8								
0								
8								
4								
6								

- Offset: 1 bit. Index: 2 bits
- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks → 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3				1	0	1	1	
8								
0								
8								
4								
6								

- Offset: 1 bit. Index: 2 bits
- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks → 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8	1	0	0	1	0	1	1	Hit
0	0	0	0	1	0	1	1	Miss
8	1	0	0	0	0	1	1	Miss
4	0	2	0	1	0	1	1	Miss
6	0	3	0	1	0	0	1	Miss

- Offset: 1 bit. Index: 2 bits
- What happens for each request?

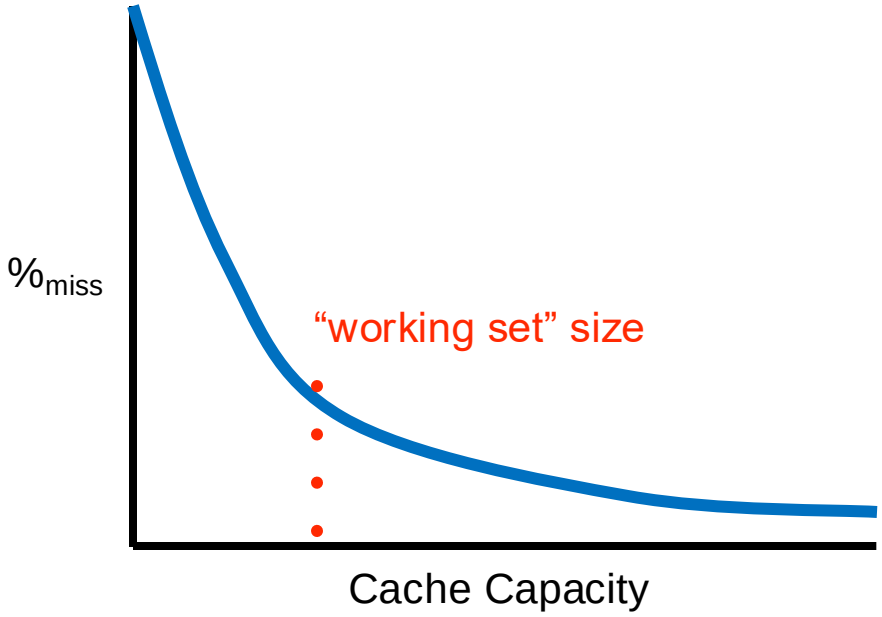
Cache Miss Paper Simulation

- %miss: 6 / 9 = 66%
  - Not good...
  - How could we improve it?

Result
Miss
Hit
Miss
Hit
Hit
Miss
Miss
Miss
Miss

Capacity and Performance

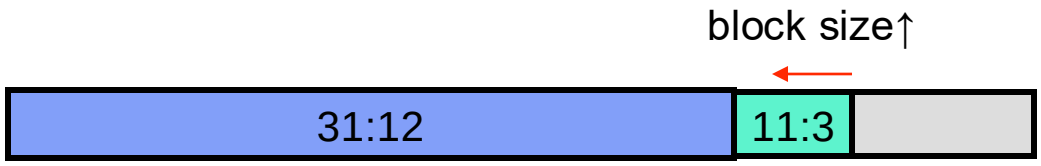
- Simplest way to reduce $\%_{miss}$: increase cache capacity
  - + Miss rate decreases monotonically
    - Due to optimizing for **"working set"**
      - Working set: insns/data program is actively using
  - – t_{hit} increases
  - t_{avg}?

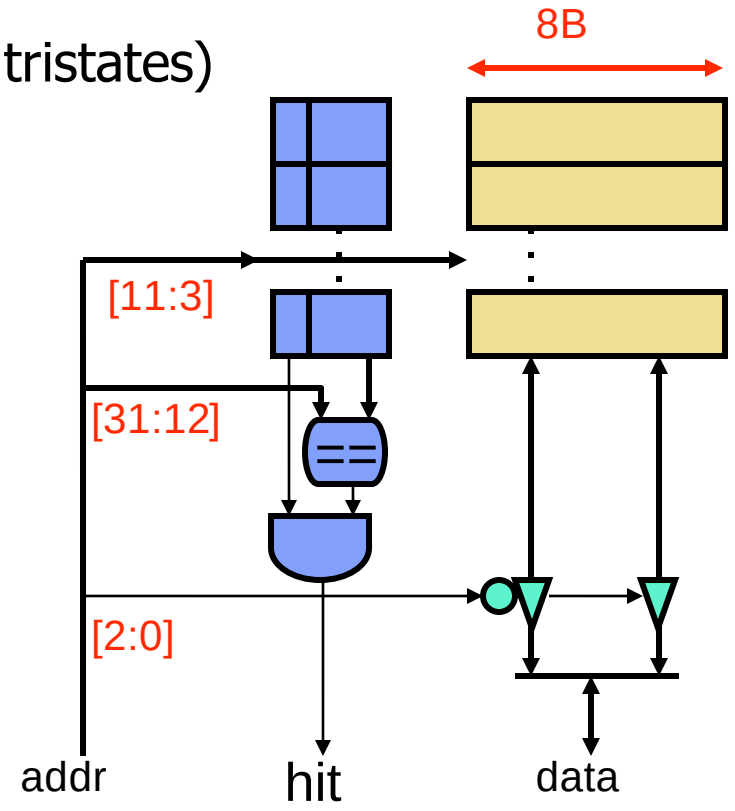


- → Next:
  - Given a fixed capacity, manipulate $\%_{miss}$ by changing **organization**

Block Size and Performance

- One possible re-organization: increase block size
  - + Exploit spatial locality
  - – Increase conflicts
    - Note that cache size in not increasing; only block size. So, we'd have less blocks in the cache.
  - – Increases t_{hit}: need word select mux (or tristates)
    - By a little; not too bad
  - + Reduce tag overhead
    - tag overhead = tag size / data size





Block Size and Tag Overhead

- 4KB cache with 1024 4B blocks?

- 4KB cache with 512 8B blocks?

- A realistic example: 64KB cache with 64B blocks?

Block Size and Tag Overhead

- 4KB cache with 1024 4B blocks?
  - 4B blocks → 2-bit offset, 1024 frames → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag / 32-bit block = **63% overhead**

- 4KB cache with 512 8B blocks?
  - 8B blocks → 3-bit offset, 512 frames → 9-bit index
  - 32-bit address – 3-bit offset – 9-bit index = 20-bit tag
  - 20-bit tag / 64-bit block = **32% overhead**

- A realistic example: 64KB cache with 64B blocks?
  - 16-bit tag / 512-bit block = **~ 3% overhead**

Cache Miss Paper Simulation (4B instead 2B blocks)

- 8B cache, 4B blocks → 2 sets

Address	Tag	Index	Offset	Set 0	Set 1	Result
C				invalid	0	Miss
E						Hit
8						Miss
3						**Miss**
8						**Miss**
0						Miss
8						Miss
4						Miss
6						**Hit**

- What happens for each request?

Cache Miss Paper Simulation (4B instead 2B blocks)

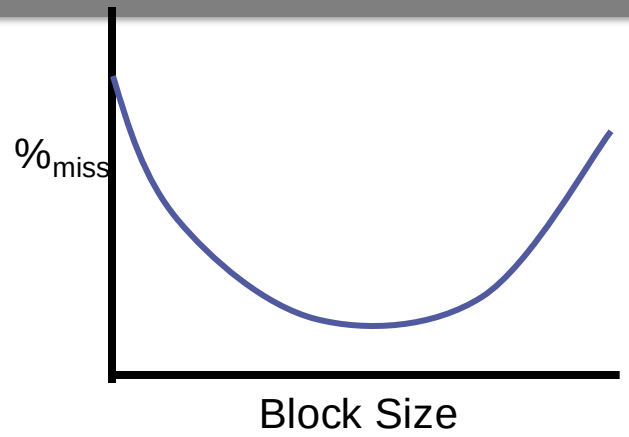
- 8B cache, 4B blocks → 2 sets

Address	Tag	Index	Offset	Set 0	Set 1	Result
C	1	1	0	invalid	0	Miss
E	1	1	2	invalid	1	Hit
8	1	0	0	invalid	1	Miss
3	0	0	3	1	1	**Miss**
8	1	0	0	0	1	**Miss**
0	0	0	0	1	1	Miss
8	1	0	0	0	1	Miss
4	0	1	0	1	1	Miss
6	0	1	2	1	0	**Hit**

- Offset: 2 bits. Index: 1 bit
- 8,3: new conflicts (fewer sets)
- 4,6: spatial locality

Block Size and Miss Rate Reduction

+ **Spatial prefetching**
  - For blocks with adjacent addresses
  - Turns miss/miss pairs into miss/hit pairs
  - Example: `4, 6`



– **Conflicts**
  - For blocks with non-adjacent addresses (but in adjacent blocks)
  - Turns hits into misses by disallowing simultaneous residence
  - Example: `8, 3`

- Both effects are always present to some degree
  - Spatial prefetching dominates initially (until 64–128B)
  - Conflicts dominate afterwards
  - Optimal block size is 32–256B (varies across programs)
  - Typical: 64B

Block Size and Miss Penalty

- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - "Critical Word First / Early Restart (CRF/ER)"
  - Requested word fetched first, pipeline restarts immediately
  - Remaining words in block transferred/filled in the background
- t_{miss}'es of a cluster of misses will suffer
  - Reads/transfers/fills of two misses cannot be overlapped
  - Latencies start to pile up
  - This is technically a bandwidth problem (more later)

Cache Miss Paper Simulation

- 8B cache, 4B blocks → 2 sets

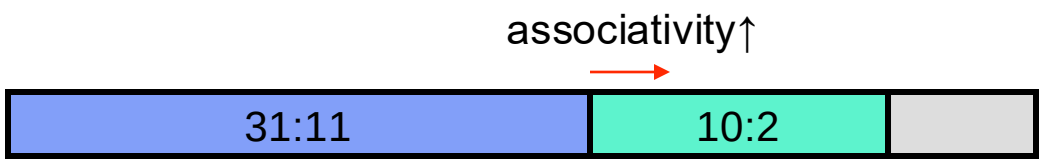
Address	Tag	Index	Offset	Set 0	Set 1	Result
C	1	1	0	invalid	0	Miss
E	1	1	2	invalid	1	Hit
8	1	0	0	invalid	1	Miss
3	0	0	3	1	1	**Miss**
8	1	0	0	0	1	**Miss**
0	0	0	0	1	1	Miss
8	1	0	0	0	1	Miss
4	0	1	0	1	1	Miss
6	0	1	2	1	0	**Hit**

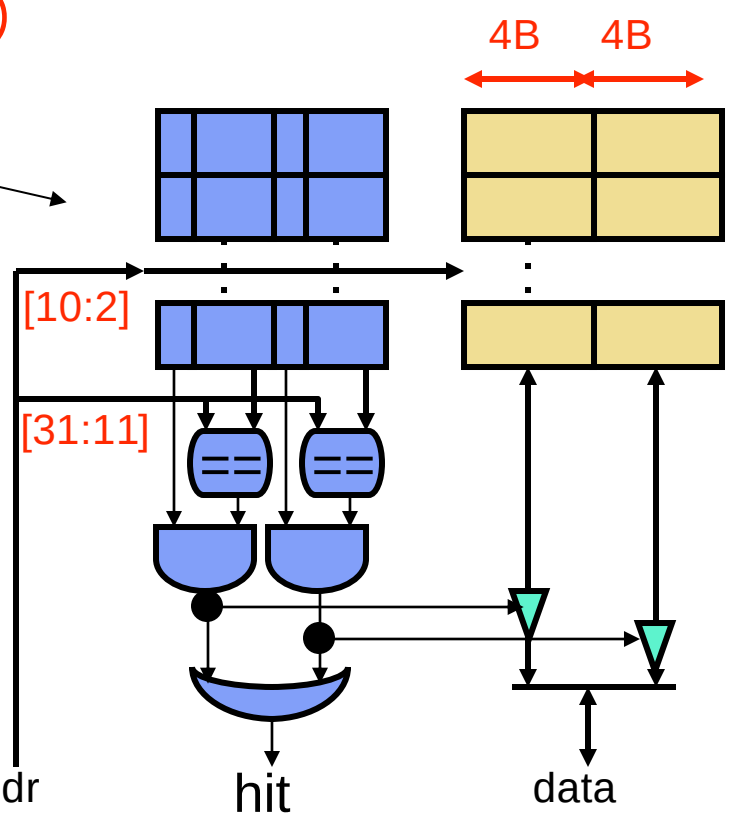
- 8 (1000) and 0 (0000): same set for any $ < 16B
- Can we do anything about this?

Associativity

- New organizational dimension: **Associativity**
  - Block can reside in one of few frames
  - Frame groups called sets
  - Each frame in set called a way
  - If 1-way → cache is direct-mapped (DM)
  - If 1-set → cache is fully-associative (FA)
  - This is 2-way set-associative (SA)

- Lookup algorithm:
  - Use index bits to find set
  - Read data/tags in all frames **in parallel**
  - **Any** (match && valid bit) ? Hit : Miss



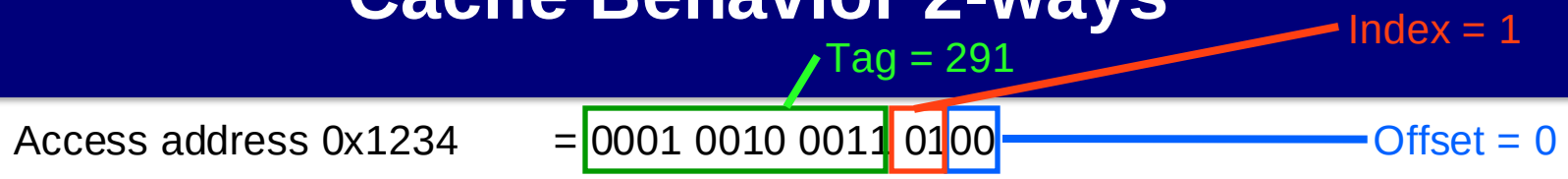


Cache Behavior in 2-ways

Cache: 4 sets, 2 ways, 4B blocks

Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	0	000	00 00 00 00	0	000	00 00 00 00
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

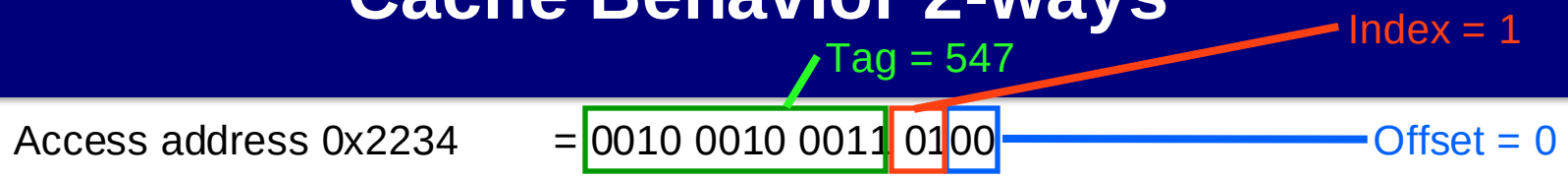
Cache Behavior 2-ways



Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	**1**	**291**	00 00 00 00	0	000	00 00 00 00
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

Miss. Request from next level. Wait...

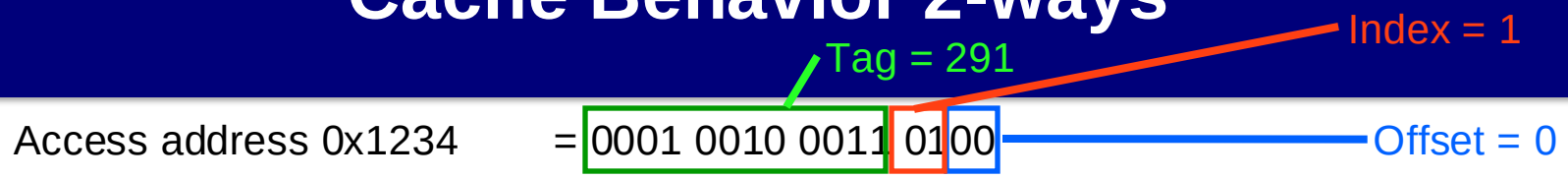
Cache Behavior 2-ways



Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	1	291	0F 1E 39 EC	**1**	**547**	00 00 00 00
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

Miss. Request from next level. Wait...

Cache Behavior 2-ways



Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	1	291	0F 1E 39 EC	1	547	01 CF D0 87
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

Hit in Way 0

Cache Miss Paper Simulation

- 8B cache, 2B blocks, 2 ways

				Set 0		Set 1		
Address	Tag	Index	Offset	Way0	Way1	Way0	Way1	Result
C				invalid	0	0	1	
E								
8								
3								
8								
0								
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

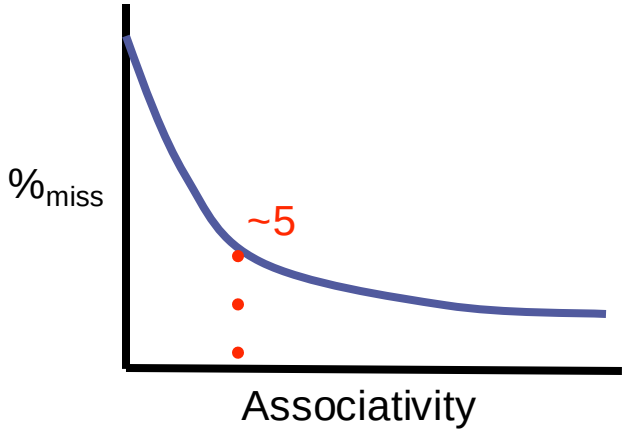
- 8B cache, 2B blocks, 2 ways → 2 sets (=8/2/2)

				Set 0		Set 1		
Address	Tag	Index	Offset	Way0	Way1	Way0	Way1	Result
C	3	0	0	invalid	0	0	1	Miss
E	3	1	0	3	0	0	1	**Miss**
8	2	0	0	3	0	0	3	Miss
3	0	1	1	3	2	0	3	Hit
8	2	0	0	3	2	0	3	Hit
0	0	0	0	3	2	0	3	Miss
8	2	0	0	0	2	0	3	**Hit**
4	1	0	0	0	2	0	3	Miss
6	1	1	0	0	1	0	3	Miss

- Offset: 1 bit. Index: 1 bit

Associativity And Performance

- The associativity game
  - + Higher associative caches have lower $\%_{miss}$
    - Less chance of conflict between addresses belonging to same set
  - – t_{hit} increases
    - But not much for low associativities (2,3,4,5)
  - t_{avg}?



- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
- 5-way set-associativity? No problem

Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?

- Belady's (oracle): block that will be used furthest in future
- Random
- FIFO (first-in first-out)
- LRU (least recently used)
  - Fits with temporal locality, LRU = least likely to be used in future
- **NMRU (not most recently used)**
  - An easier-to-implement approximation of LRU
  - Equivalent to LRU for 2-way SA caches

Cache Structure Math Summary

- Please don't memorize this! This should all be straight forward.

- Given capacity, block_size, ways (associativity), and address_size.
- Cache parameters:
  - num_frames = capacity / block_size
  - num_sets = num_frames / ways = capacity / block_size / ways
- Address bit fields:
  - num_offset_bits = $\log_2$(block_size)
  - num_index_bits = $\log_2$(num_sets)
  - tag_bits = address_size – num_offset_bits – num_index_bits
- Numeric way to get offset/index/tag from address:
  - block_offset = addr % block_size
  - index = (addr / block_size) % sets
  - tag = addr / (sets*block_size)

ABC

- ↗ **Associativity:**
  - \+ Decreases conflict misses
  - – Increases t_{hit}

*What are all these **blue** words? We'll see...*

- ↗ **Block size:**
  - – Increases conflict misses
  - \+ Decreases compulsory misses
  - ± Increases or decreases capacity misses
  - • Little effect on t_{hit}, may increase t_{miss}

- ↗ **Capacity:**
  - \+ Decreases capacity misses
  - – Increases t_{hit}

- How much exactly they help depends...

Different Problems → Different Solutions

- Suppose we have 16B, direct-mapped cache with 4B blocks
  - 4 sets
  - Examine some access patterns and think about what would help
  - Misses are in red
- Access pattern A:
  - As is: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26
  - 8B blocks? 0, 2, **4**, 6, 8, 10, **12**, 14, 16, 18, **20**, 22, 24, 26
  - 2-way assoc? 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26
- Access pattern B:
  - As is: 0, 128, 1, 129, 2, 130, 3, 131, 4, 132, 5, 133, 6
  - 8B blocks? 0, 128, 1, 129, 2, 130, 3, 131, 4, 132, 5, 133, 6
  - 2-way assoc? 0, 128, **1, 129, 2, 130, 3, 131,** 4, 132, **5, 133, 6**
- Access pattern C (All 3: as is, 8B blocks, and 2-way assoc):
  - 0,20,40,60,48,36,24,12,1,21,41,61,49,37,25,13,2,22,42,62,50,38,…

Analyzing Misses: 3C Model

- Divide cache misses into categories based on cause
  - **Compulsory**: block size is too small (i.e., address not seen before)
  - **Capacity**: capacity is too small
  - **Conflict**: associativity is too low

Different Problems → Different Solutions

- Access pattern A: Compulsory misses
  - 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26
  - For misses: have not accessed that block
    - Size/associativity won't help (never had it)
    - Larger block → include more data in one block → more hits

- Recognizing compulsory misses:
  - Never seen the **block** before

Different Problems → Different Solutions

- Access pattern B: Conflict misses
  - 0, 128, 1, 129, 2, 130, 3, 131, 4, 132, 5, 133, 6
  - 0 and 128 map to same set (set 0): kick each other out ("conflict")
  - Larger block? No help
  - Larger cache? Only helps if MUCH larger (256 B instead of 16B)
  - Higher associativity? Fixes problem
    - Can have both 0 and 128 in set 0 at same time (different ways)

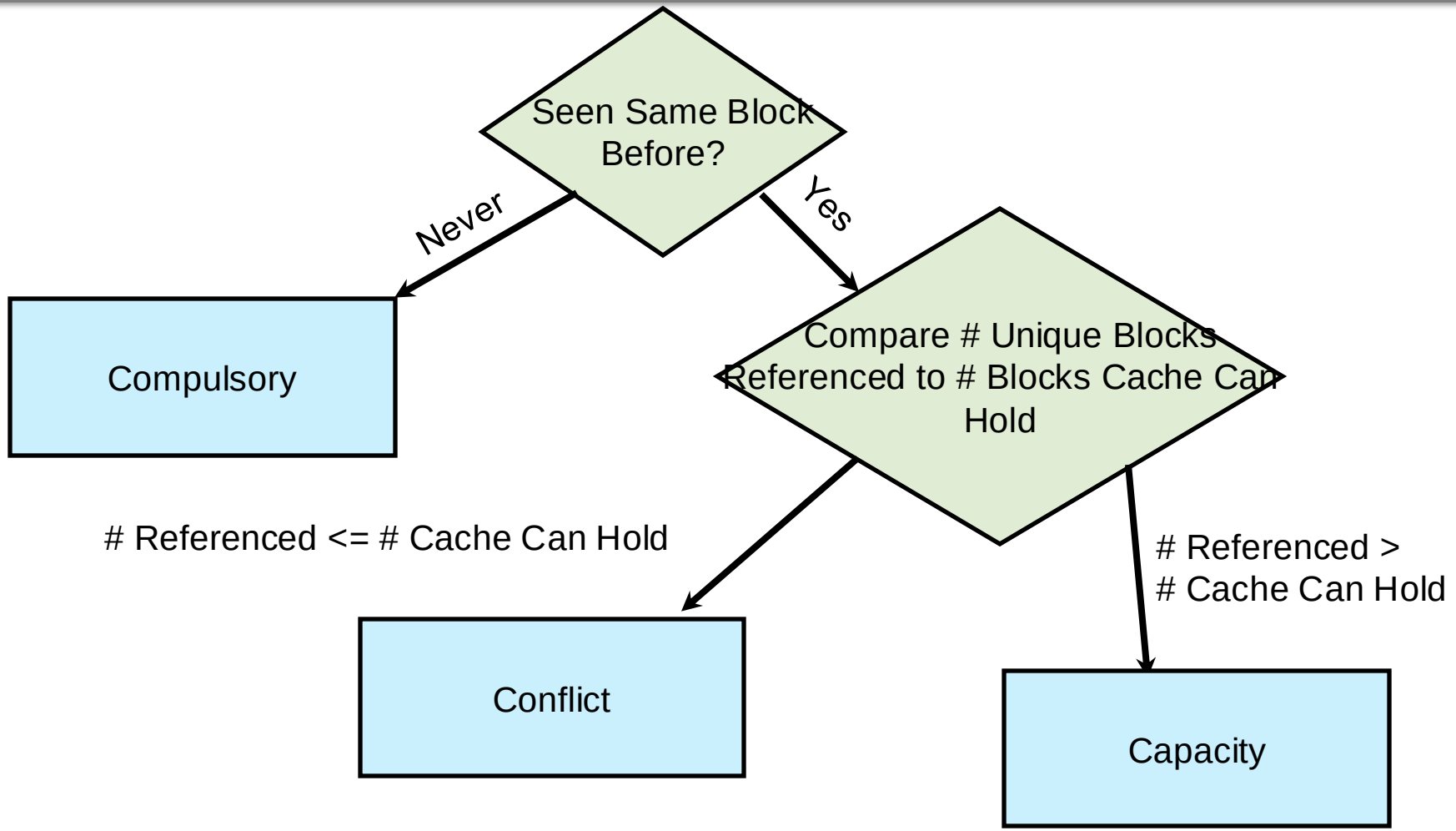
- Recognizing conflict misses:
  - Count unique blocks between last access and miss (inclusive)
  - Number of unique blocks <= number of blocks in cache? Conflict
    - Enough space to hold them all...
    - Just must be having set conflict

Different Problems → Different Solutions

- Access pattern C: Capacity Misses
  - 0,20,40,60,48,36,24,12,1,21,41,61,49,37,25,13,2,22,42,62,50,38,…
  - Larger block size? No help
    - Even 16B block (entire cache) won't help
  - Associativity? No help… even at full assoc
    - After 0, 20, 40, 60: kick out 0 for 48
    - Kick out 20 for 36
    - Kick out 40 for 24…
  - Solution: make cache larger
    - Doubling cache size turns most misses into hits
      - A few compulsory misses remain
  - 0,20,40,60,48,36,24,12,1,21,41,61,49,37,25,13,2,22,42,62,50,38,…

- Recognizing Capacity Misses
  - Count unique blocks between last access and miss (inclusive)
  - Number of unique blocks > number of blocks in cache? Capacity
    - Just can't hold them all

Miss Categorization Flow Chart



Two Optimizations

- **Victim buffer**: for capacity/conflict misses
  - Technically, reduces t_{miss} for these misses, doesn't eliminate them

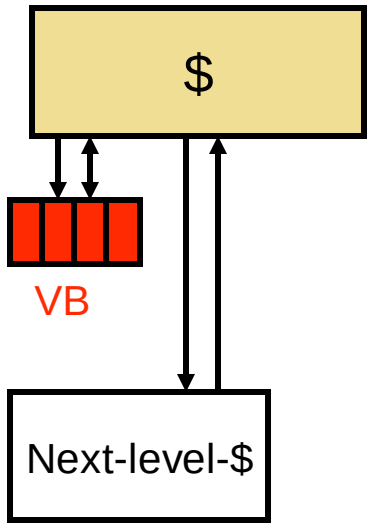
- **Prefetching**: for compulsory misses

Victim Buffer

- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed

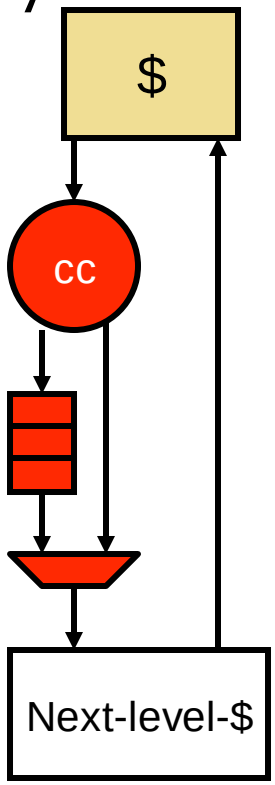
Fully Associative

- **Victim buffer (VB)**: small FA cache (e.g., 4 entries)
  - Small, so very fast
  - Blocks kicked out of cache are placed in VB
  - On miss, check VB: hit ? Place block back in cache
  - 4 extra ways, shared among **all sets**
    + Only a few sets will need it at any given time
  - On cache fill path: reduces t_{miss}, no impact on t_{hit}
  + Very effective in practice



Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - In software: insert prefetch (non-binding load) insns into code
  - In hardware: cache controller (cc) generates prefetch addresses
- Keys: anticipate upcoming miss addresses accurately
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: don't evict useful data
  - Prioritize misses over prefetches
- Simple algorithm: **next block prefetching**
  - Miss address **X** → prefetch address **X+block_size**
  - Works for insns: sequential execution
    - *What about non-sequential execution?*
  - Works for data: arrays
    - *What about other data-structures?*
  - Address prediction is an actively-researched area

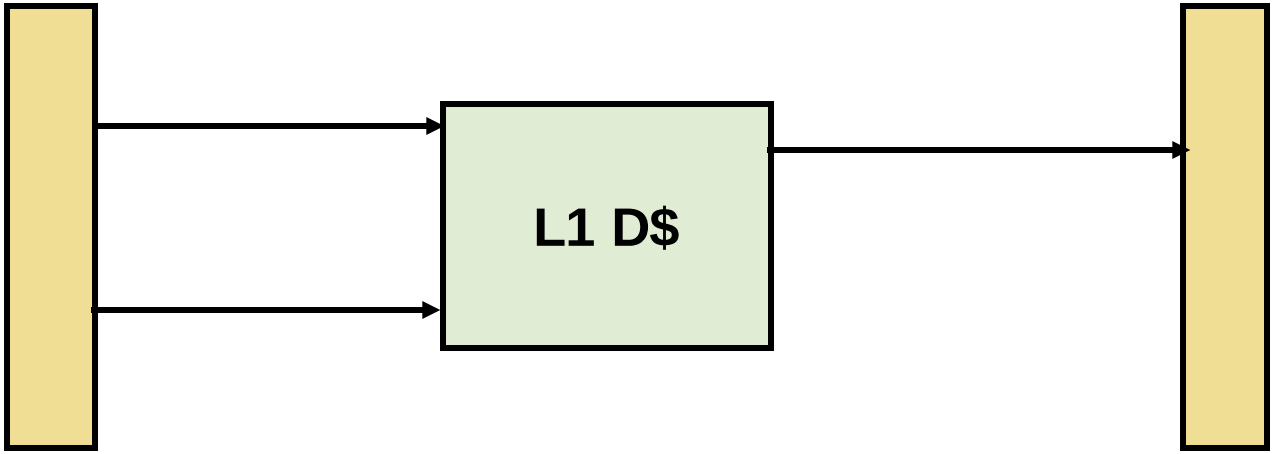


Write Issues

- So far, we have looked at reading from cache
  - Insn fetches, loads
- What about writing into cache?
  - Specifically, writing to D$

- Several new issues:
  - Must read tags first **before** writing data
    - Cannot be in parallel
  - Cache may have **dirty** data
    - I.e., data which has been updated in this cache, but not lower levels
    - Must be written back to lower level before eviction

Recall Data Memory Stage of Datapath



- So far, we have just assumed D$ in Memory Stage...
  - Actually, it's a bit more complex for a couple reasons...

Problem with Writing #1: Store Misses

- Load instruction misses D$:
  - Have to stall datapath
  - Need missing data to complete instruction

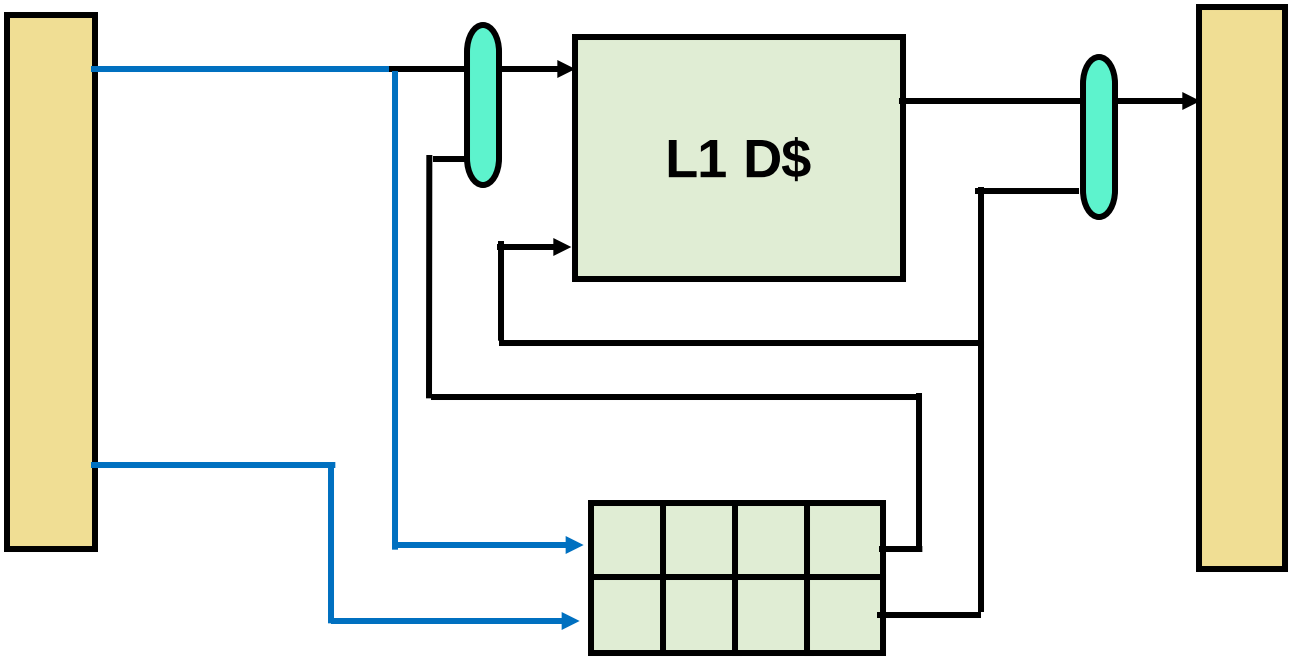
- Store instruction misses D$:
  - Stall?
  - Would really like not to
  - Store is writing the data
  - Need rest of **block** because we cannot have part of a block
    - Generally, do not support "these bytes are valid, those are not"
  - How to avoid this?

Problem with Writing #2: Serial Tag/Data Access

- Load can read tags/data in parallel
  - Read both SRAMs
  - Compare Tags → Select proper way (if any)

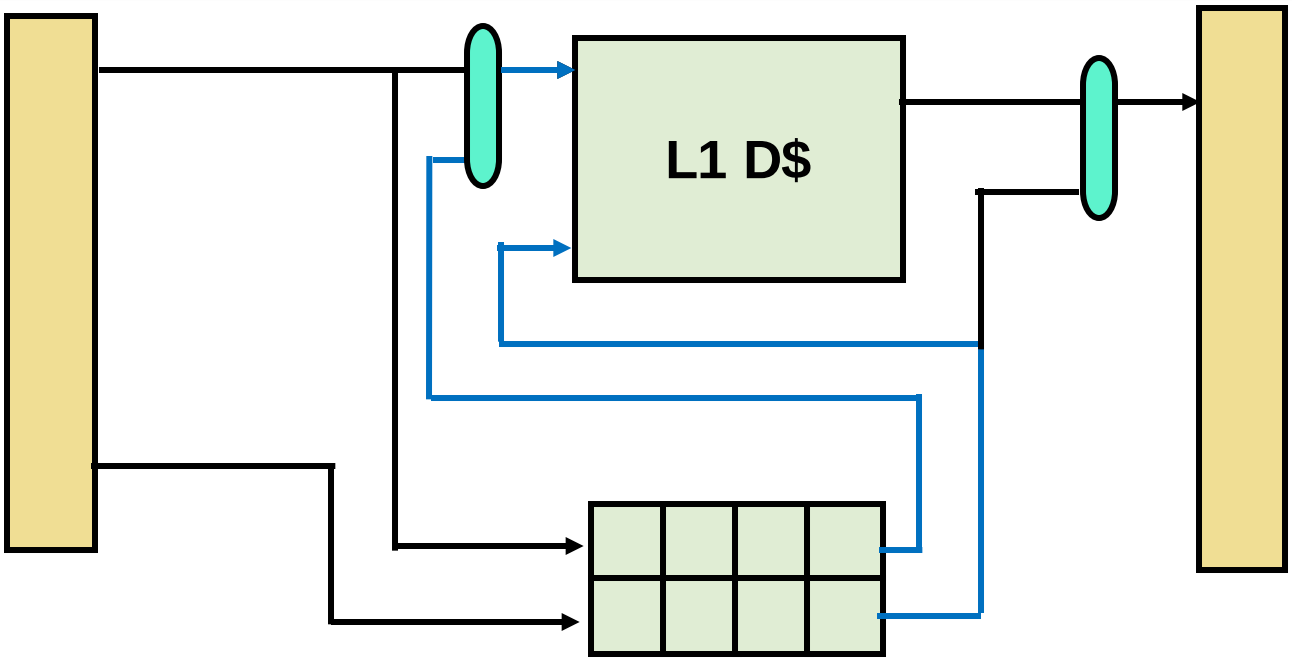
- Stores cannot write tags/data in parallel
  - Read tags/write data array at same time??
    - How to know which way?
    - Or even if its a hit?
    - Incorrect → overwrote data from somewhere else…
- Multi-cycle datapath:
  - Stores take an extra cycle? → higher CPI
- Pipelined datapath:
  - Tags in one stage, Data in the next?
  - Works for stores, but loads serialize tags/data → higher CPI

Store Buffer



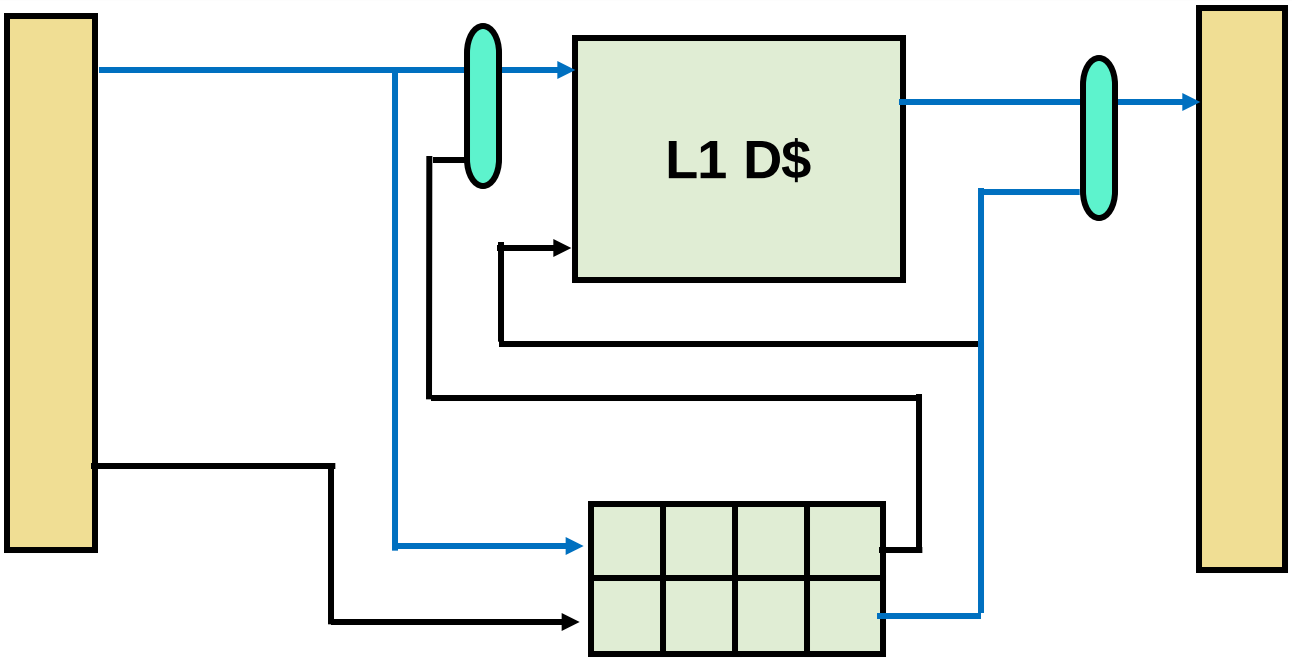
- Stores write into a **store buffer**
  - Holds address, size, and data of stores

Store Buffer



- Stores write into a **store buffer**
  - Holds address, size, and data of stores
  - Store data written from store buffer into cache
    - Miss? Data stays in buffer until hit

Store Buffer



- Loads search store buffer for matching store
  - Match? **Forward** data from the store buffer
  - No match: Use data from D$

- Addresses are a CAM: allow search for match

Store Buffer

- How does this resolve our issues?

- Problem with Writing #1: Store misses
  - Stores write to store buffer and are done
  - **FSM** writes stores into D$ from store buffer
  - Misses stall store buffer → D$ write (but not pipeline)
  - Pipeline will stall on full store buffer
- Problem with Writing #2: Tags → Data
  - **FSM** that writes stores to D$ can check tags… then write data
  - **Decoupled from datapath's normal execution**
    - Can happen whenever loads are not using the D$

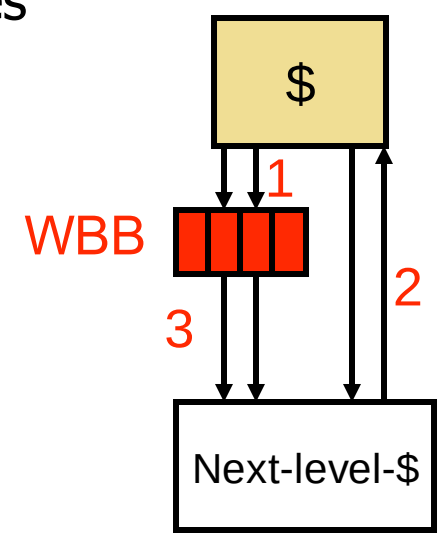
Write Propagation

- When to propagate new value to lower-level memory?

- **Write-thru**: immediately
  - Requires additional bus bandwidth
  - Not common
- **Write-back**: when block is replaced
  - Blocks may be **dirty** now (have modified data)
  - Add a "**dirty bit**" (in tag array)
    - Bit set (dirty bit = 1) by a store to the block
    - On fill (when we need to replace the dirty block with a new one):
      - If dirty bit==1→propagate change down the memory hierarchy
      - Clear dirty bit (dirty bit = 0)

Write Back: Dirty Misses

- Writeback caches may have **dirty misses**:
  - Victim block (one to be replaced) is dirty
    - Must first writeback to next level
    - Then request data for miss
    - Slower :(
  - Solution:
    - Add a buffer on back side of cache: **writeback buffer** (WBB)
      - Small full associative buffer, holds a few lines
    - Request miss data immediately
    - Put dirty line in WBB
    - Writeback later



What This Means to the Programmer

- If you're writing code, you want good performance.
- The cache is **crucial** to getting good performance.
- The effect of the cache is influenced by the **order of memory accesses**.

CONCLUSION:

The programmer can change the order of memory accesses to improve performance!

Cache Performance Matters!

- A **HUGE** component of software performance is how it interacts with cache
- Example:

  Assume that x[i][j] is stored next to x[i][j+1] in memory ("row major order").

  Which will have fewer cache misses?

```
for (k = 0; k < 100; k++)
   for (j = 0; j < 100; j++)
      for (i = 0; i < 5000; i++)
                x[i][j] = 2 * x[i][j];
```

A

```
for (k = 0; k < 100; k++)
   for (i = 0; i < 5000; i++)
      for (j = 0; j < 100; j++)
                x[i][j] = 2 * x[i][j];
```

B

Cache Performance Matters!

- A **HUGE** component of software performance is how it interacts with cache
- Example:

  Assume that x[i][j] is stored next to x[i][j+1] in memory ("row major order").

  Which will have fewer cache misses?

```
for (k = 0; k < 100; k++)
   for (j = 0; j < 100; j++)
      for (i = 0; i < 5000; i++)
                x[i][j] = 2 * x[i][j];
```

A

```
for (k = 0; k < 100; k++)
   for (i = 0; i < 5000; i++)
      for (j = 0; j < 100; j++)
                x[i][j] = 2 * x[i][j];
```

B

Blocking (Tiling) Example

```
/* Before */
 for(i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      for (k = 0; k < SIZE; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Two Inner Loops:
  - Read all NxN elements of z[ ] (N = SIZE)
  - Read N elements of 1 row of y[ ] repeatedly
  - Write N elements of 1 row of x[ ]
- Capacity Misses a function of N & Cache Size:
  - 3 NxN => no capacity misses; otherwise ...
- Idea: compute on BxB submatrix that fits

Blocking (Tiling) Example

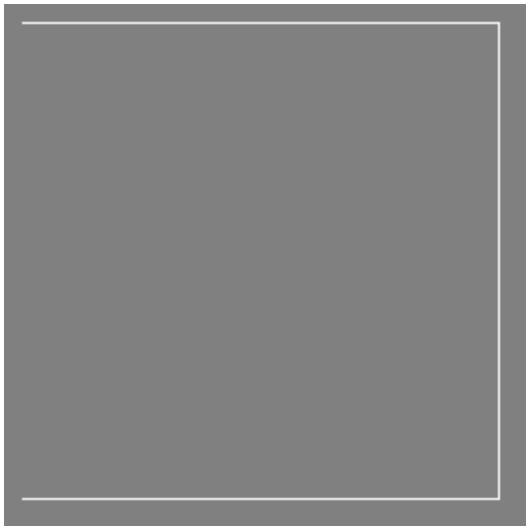
```
/* After */
 for(ii = 0; ii < SIZE; ii += B)
    for (jj = 0; jj < SIZE; jj += B)
      for (kk = 0; kk < SIZE; kk +=B)
        for(i = ii; i < MIN(ii+B-1,SIZE); i++)
          for (j = jj; j < MIN(jj+B-1,SIZE); j++)
            for (k = kk; k < MIN(kk+B-1,SIZE); k++)
               c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

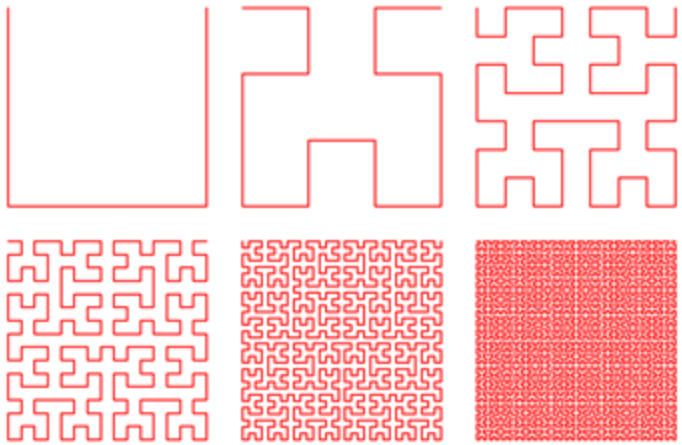
- Capacity Misses decrease

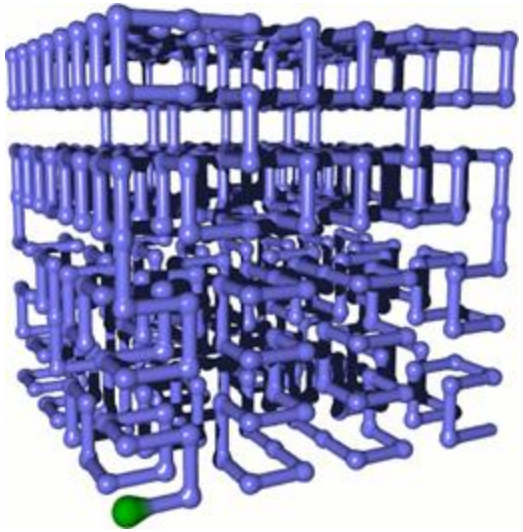
  $2N^3 + N^2$ to $2N^3/B + N^2$
- B called *Blocking Factor (Also called Tile Size)*

Hilbert Curves: A Fancy Trick for Matrix Locality

- Turn a 1D value into an n-dimensional "walk" of a cube space (like a 2D or 3D matrix) in a manner that maximizes locality
- Extra overhead to compute curve path, but computation takes no memory, and cache misses are very expensive, so it may be worth it
- (Actual algorithm for these curves is simple and easy to find)







Memory Bus

- **Memory bus**: connects CPU package with main memory
  - Has its own clock
    - Typically slower than CPU internal clock: 100–500MHz vs. 3GHz
    - SDRAM operates on this clock
  - Is often itself internally pipelined
    - Clock implies bandwidth: 100MHz → start new transfer every 10ns
    - Clock doesn't imply latency: 100MHz !→ transfer takes 10ns
  - Bandwidth is more important: determines peak performance

Memory Latency and Bandwidth

- Nominal **clock frequency** applies to CPU and caches

- Careful when doing calculations
  - Clock frequency increases **don't reduce memory or bus latency**
    - May make misses come out faster
    - At some point memory bandwidth may become a **bottleneck**
    - Further increases in clock speed won't help at all

Clock Frequency Example

- Baseline setup
  - Processor clock: 1GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores: 4, Loads: 4 + t_{avgL1}
  - L1 D$: t_{hit} = 1 cycle, 10% miss
  - L2$: t_{hit} = 20 cycles, 5% miss
  - Memory: 200 cycles

Computation

t_{avgL2} = 20 + 0.05 * 200 = 30

t_{avgL1} = 1 + 0.10 * 30 = 4

Average load latency = 4 + 4 = 8

CPI = 0.2 * 8 + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = 4.6

The clock rate is 1GHz, or 1e9 cycles/second.
The CPI is 4.6 cycles/instruction.

Performance = (1e9 cycles/second) / (4.6 cycles/instruction)
= 217,391,304 instructions/second
= 217 MIPS

Clock Frequency Example

- Baseline setup
  - Processor clock: **2**GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores: 4, Loads: 4 + t_{avgL1}
  - L1 D$: t_{hit} = 1 cycle, 10% miss
  - L2$: t_{hit} = 20 cycles, 5% miss
  - Memory: **4**00 cycles

Computation

t_{avgL2} = 20 + 0.05 * **400** = **40**

t_{avgL1} = 1 + 0.10 * **40** = **5**

Average load latency = 4 + **5** = **9**

CPI = 0.2 * **9** + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = **4.8**

The clock rate is **2**GHz, or **2**e9 cycles/second.
The CPI is **4.8** cycles/instruction.

Performance = (**2e9** cycles/second) / (**4.8** cycles/instruction)
= **416,666,666** instructions/second
= **417** MIPS (91% speedup, for 100% freq increase)

Clock Frequency Example

- Baseline setup
  - Processor clock: **4**GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores: 4, Loads: 4 + t_{avgL1}
  - L1 D$: t_{hit} = 1 cycle, 10% miss
  - L2$: t_{hit} = 20 cycles, 5% miss
  - Memory: **8**00 cycles

Computation

t_{avgL2} = 20 + 0.05 * **800** = **60**

t_{avgL1} = 1 + 0.10 * **60** = **7**

Average load latency = 4 + **7** = **11**

CPI = 0.2 * **11** + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = **5.2**

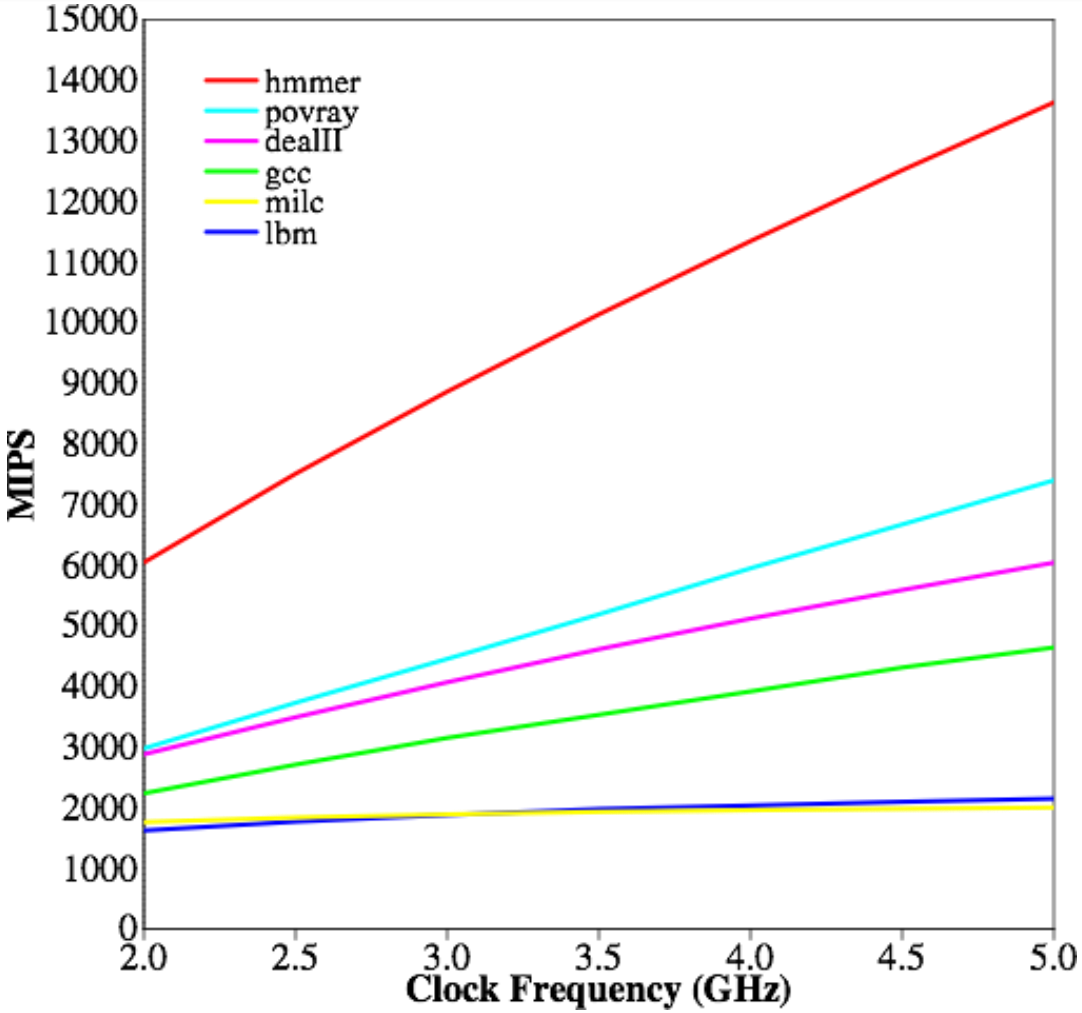
The clock rate is **4**GHz, or **4**e9 cycles/second.
The CPI is **5.2** cycles/instruction.

Performance = (**4e9** cycles/second) / (**5.2** cycles/instruction)
= **769,230,769** instructions/second
= **769** MIPS (84% speedup, for 100% freq increase)

Actually, a Bit Worse..

- Only looked at D$ miss impact
  - Ignored store misses: assumed storebuffer can keep up
- Also have I$ misses
- At some point, become bandwidth constrained
  - Effectively makes t_{miss} go up (think of a traffic jam)
  - Also makes things we ignored matter
    - Storebuffer may not be able to keep up as well → store stalls
  - Data we previously prefetched may not arrive in time
    - Effectively makes %miss go up

Clock Frequency and Real Programs



Detailed Simulation Results
- Includes all caches, bandwidth,...
- Has L3 on separate clock
- Real programs
- 2.0 Ghz → 5.0 Ghz (150% increase)

hmmer:
- Very low %miss
- Good performance for clock
- **125%** speedup

lbm, milc:
- Very high %miss
- Not much performance gained
- lbm: **32%**
- milc: **14%**

ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2024

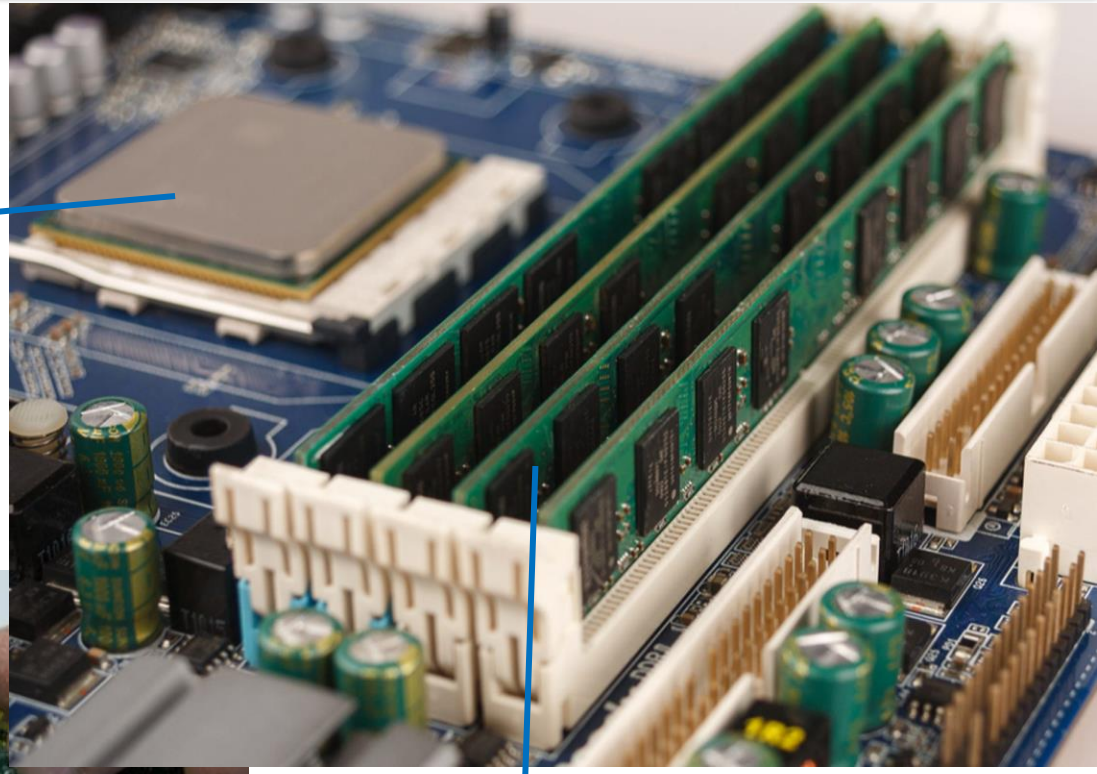
Virtual Memory

Rabih Younes

Duke University

Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

Main Memory



Typically referred to as “RAM”
(uses DRAM technology)

Problems With Our Current Approach to Memory?

- Reasonable (main) memory: 4GB—64GB?
  - In 32-bit systems:
    - Program can address 2^32 bytes = **4GB**/program
  - In 64-bit systems:
    - Program can address 2^64 bytes = **16EB**/program!
- What if we're running many programs, not just 1?
  - Impossible using what we know for now

- → We need an approach called: **virtual memory**
  - Gives every program **the illusion** of having access to the entire address space
  - Hardware and OS (operating system) move things around behind the scenes
- How?
  - Good rule to know: when we have a functionality problem
    → we can usually solve it by adding a level of indirection

Virtual Memory

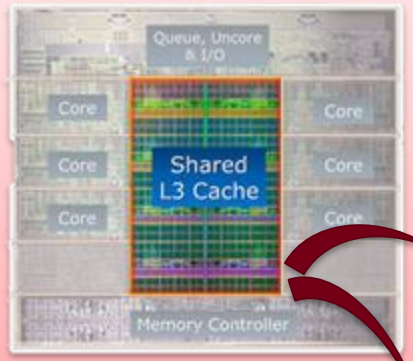
- Predates "caches" (by a little)

- Original motivation: **compatibility**
  - Ability to run the same program on machines with different main memory sizes
  - Prior to virtual memory, programmers needed to explicitly account for memory size

- **Virtual memory:**
  - Treats memory like a cache for disks (or other secondary storage)
    - Disks should be able to contain all our data
  - Contents of memory would be a dynamic subset of program's address space
  - Dynamic content management of memory is transparent to program
    - Caching mechanism makes it appear as if memory is 2^N bytes regardless of how much memory there actually is

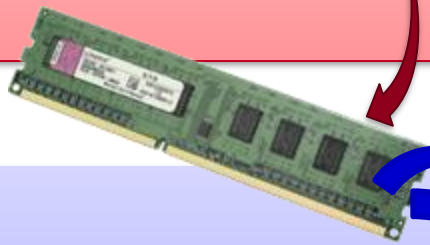
Caching vs. Virtual Memory

CACHING



Copy **block** if **popular**

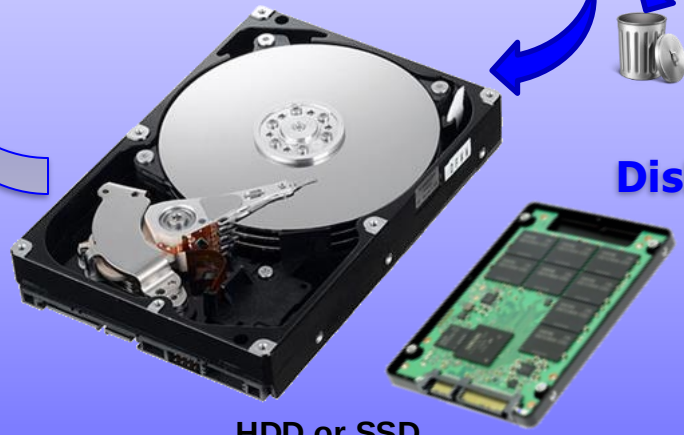
Cache
- Faster
- More expensive
- Lower capacity



RAM

VIRTUAL MEMORY

Load **page** if **needed**



Disk
- Slower
- Cheaper
- Higher capacity

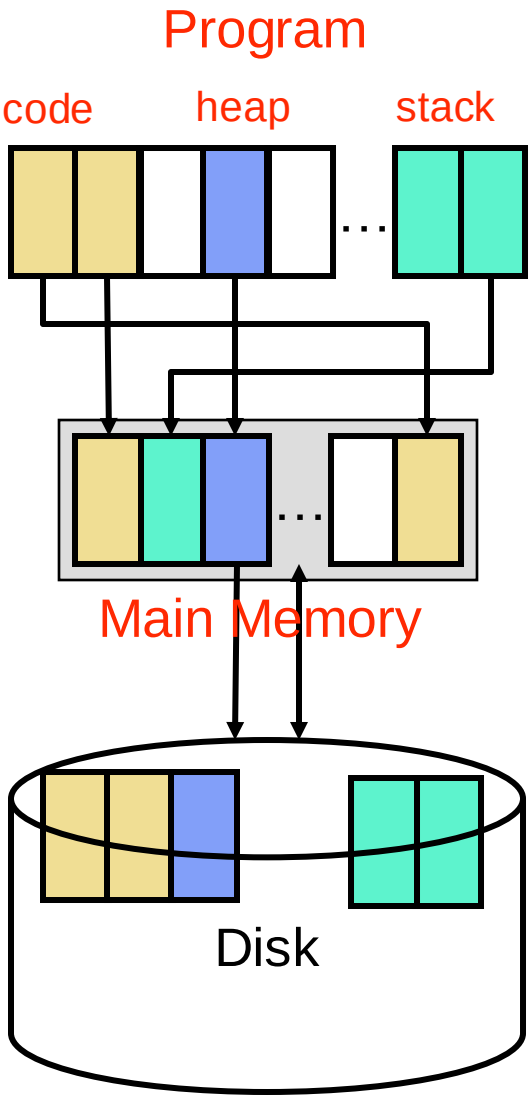
HDD or SSD

Pages

- What is swapped between disk and memory?
  - A **page** (vs. **block** in caching)
    - Using a process called "demand paging" (or "paging")

- **Page:** A small chunk (~4KB) of memory with its own record in the memory management hardware

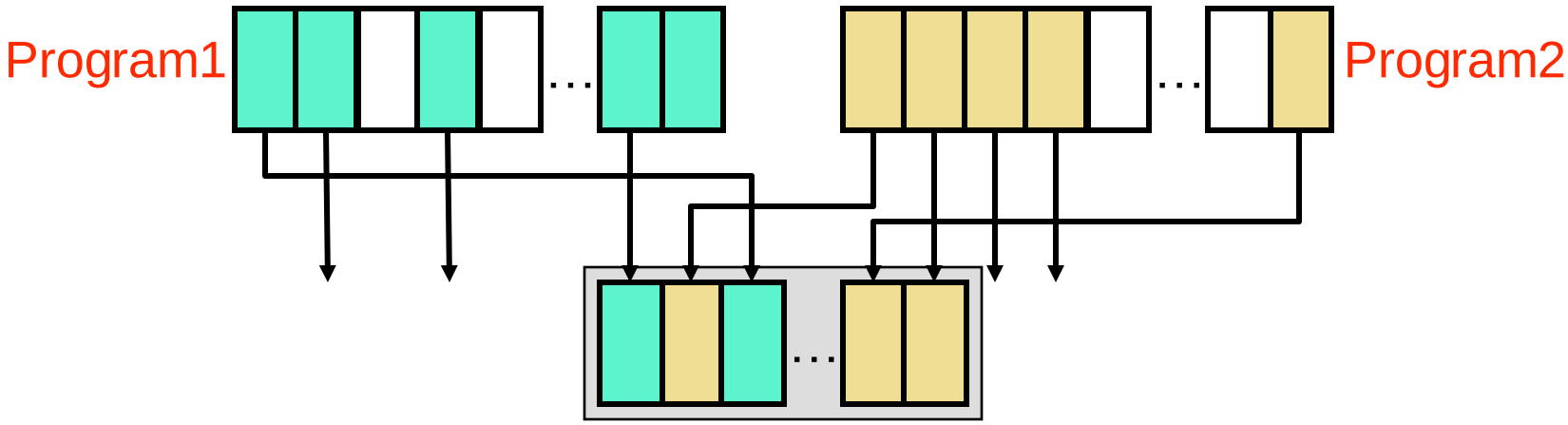
How Virtual Memory Solves Our Problem(s)



- Programs use **virtual addresses (VA)**
  - 0 to 2^N–1
  - N is the machine/system size (bus width)
    - E.g., Pentium4 is 32-bit, Core i9 is 64-bit
- Memory uses **physical addresses (PA)**
  - 0 to 2^M–1 (M<N, especially if N=64)
  - 2^M is most physical memory machine supports
- VA to PA translation at page granularity
  - → **VP to PP translation**
    (Virtual Page to Physical Page)

Other Uses of Virtual Memory

- Virtual memory is quite useful for 1 program, but is also very useful for **multiprogramming** (more than 1 program)
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - "System" maps VPs from different processes to different PPs
    + Prevents processes from reading/writing each other's memory



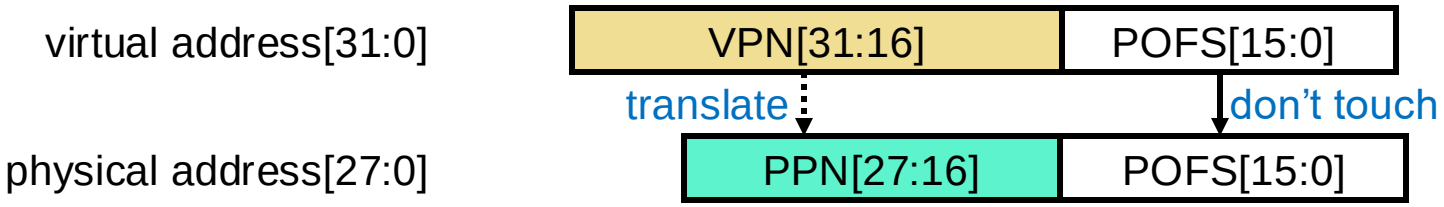
Even More Uses of Virtual Memory

- Inter-process communication
  - Map VPs in different processes to same PPs
- Direct memory access I/O
  - Think of I/O device as another process
  - Will talk more about I/O in the future
- Protection
  - Piggy-back mechanism to implement page-level protection
  - Map VP to PP … and RWX protection bits
  - Attempt to execute data, or attempt to write insn/read-only data?
    - Exception → OS terminates program

Address Translation (VA→PA or VP→PP)

Address Translation

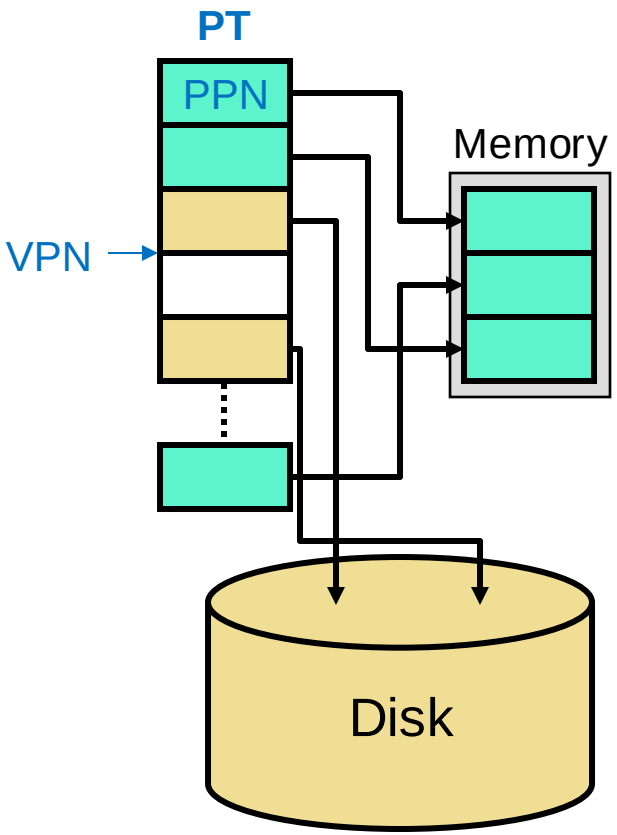
- VA→PA mapping is called **address translation**
  - Split VA into virtual page number (VPN) and page offset (POFS)
  - Translate VPN into physical page number (PPN)
  - POFS is not translated
    - Why? Because it takes us to the desired byte in a page, regardless of where that page is residing
  - VA→PA = [VPN, POFS]→[PPN, POFS]



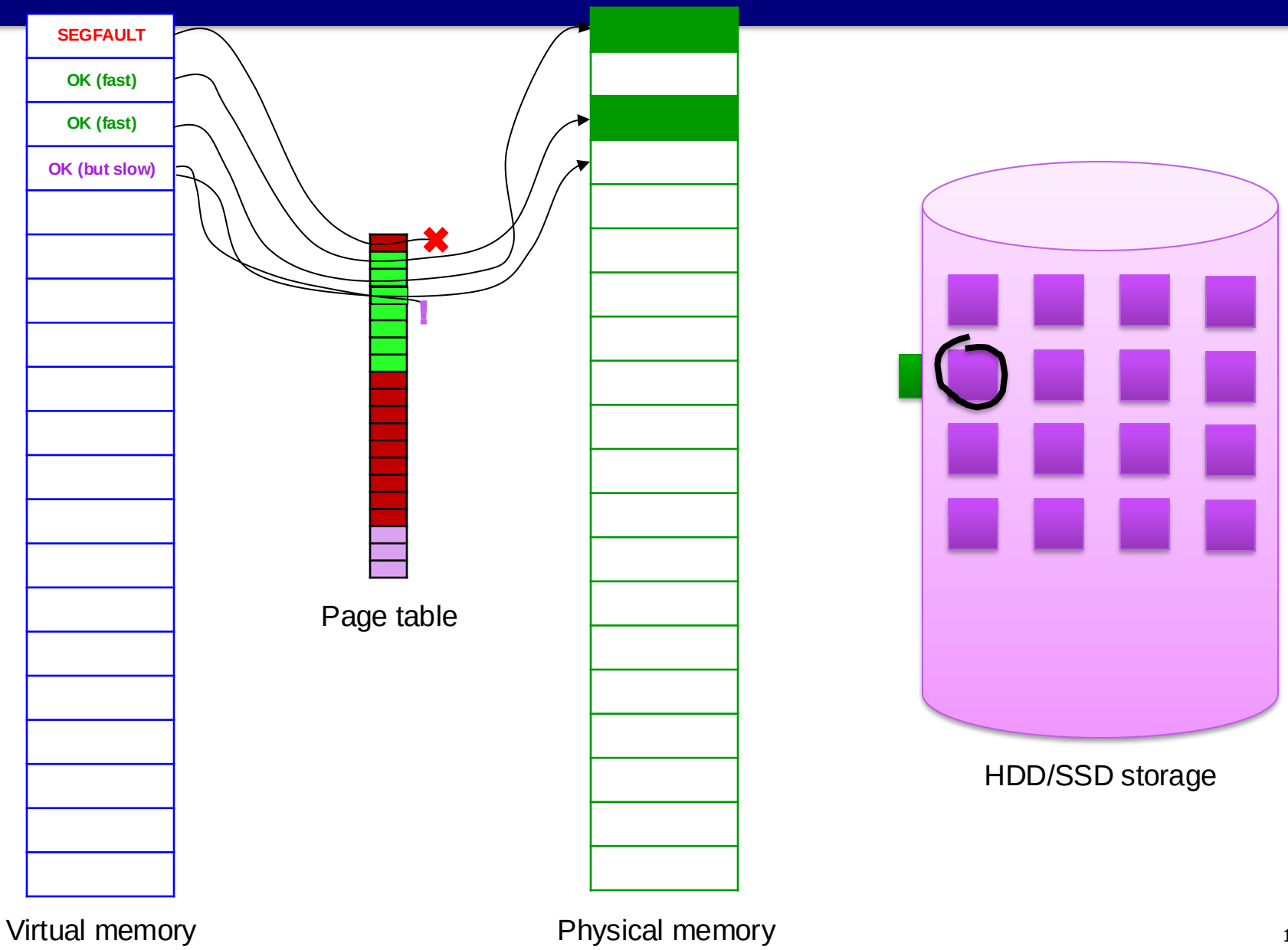
- In the example above:
  - 64KB pages? → 16-bit (= $\log_2 64K = \log_2 2^{16}$) POFS
  - 32-bit machine? → 32-bit VA → 16-bit VPN (= 32b VA – 16b POFS)
  - Maximum 256MB memory? → 28-bit PA → 12-bit PPN (= 28b – 16b)

Mechanics of Address Translation

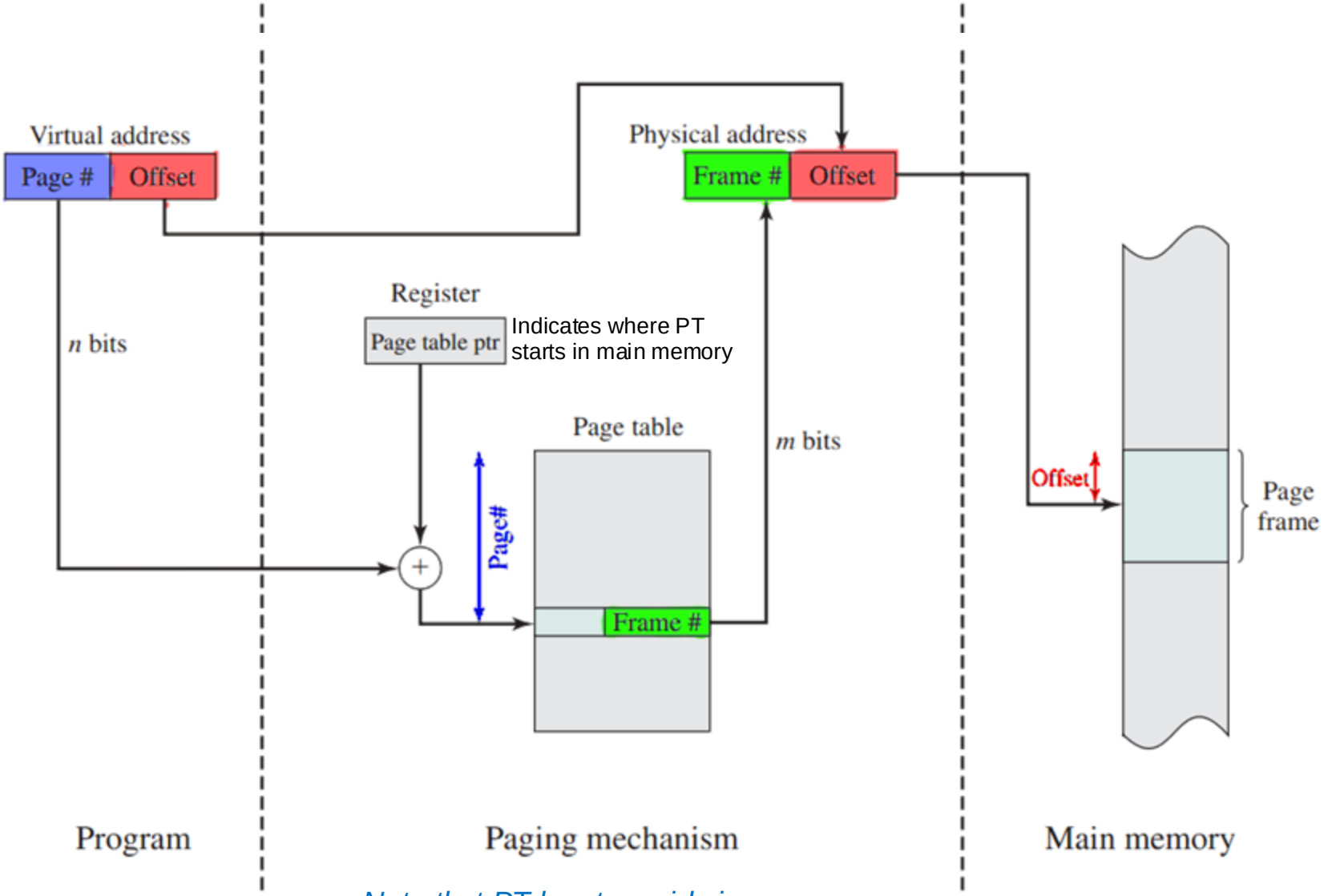
- Each process is allocated a **page table (PT)**
  - PT maps VPs to PPs or to disk addresses
    - VP entries are empty if page is never referenced
  - Translation is called **table lookup**
    - PT here is a lookup table (LUT)



High-Level Operation



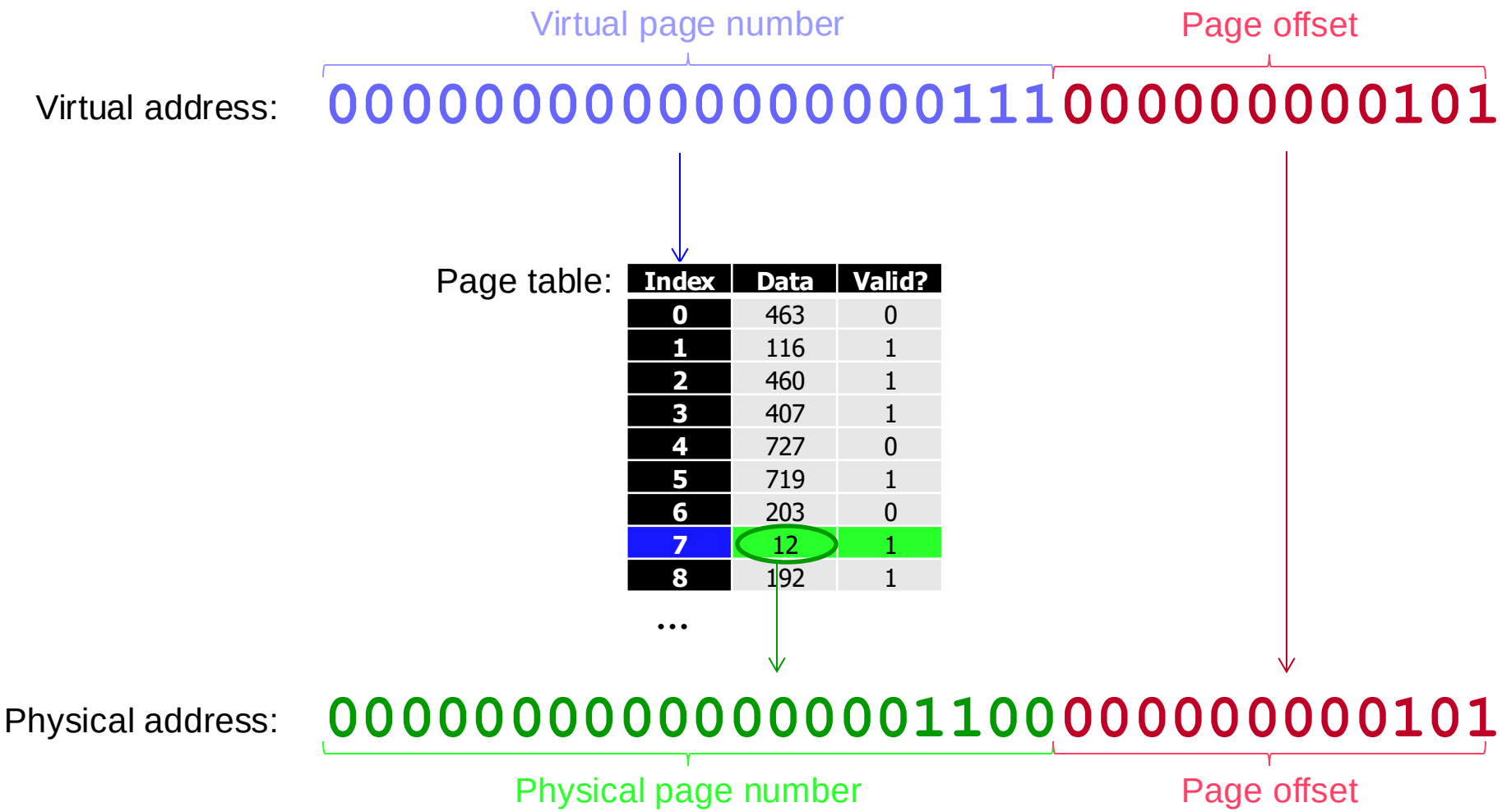
Address Translation



Note that PT has to reside in memory

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Address Translation



Structure of the Page Table

Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages

- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine with 8B PTEs?

Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages
- Solution
  - 32-bit machine → 32-bit VA → 2^{32}B virtual memory (VM) = 4GB VM
  - 4GB/VM / 4KB/page = 1M pages/VM → 1M PTEs/program's PT
  - 1M PTEs/PT * 4B/PTE → 4MB/PT (or "4MB PTs")

- How big would the page table be with 64KB pages? 256KB PTs
- How big would it be for a 64-bit machine with 8B PTEs? 32PB PTs!

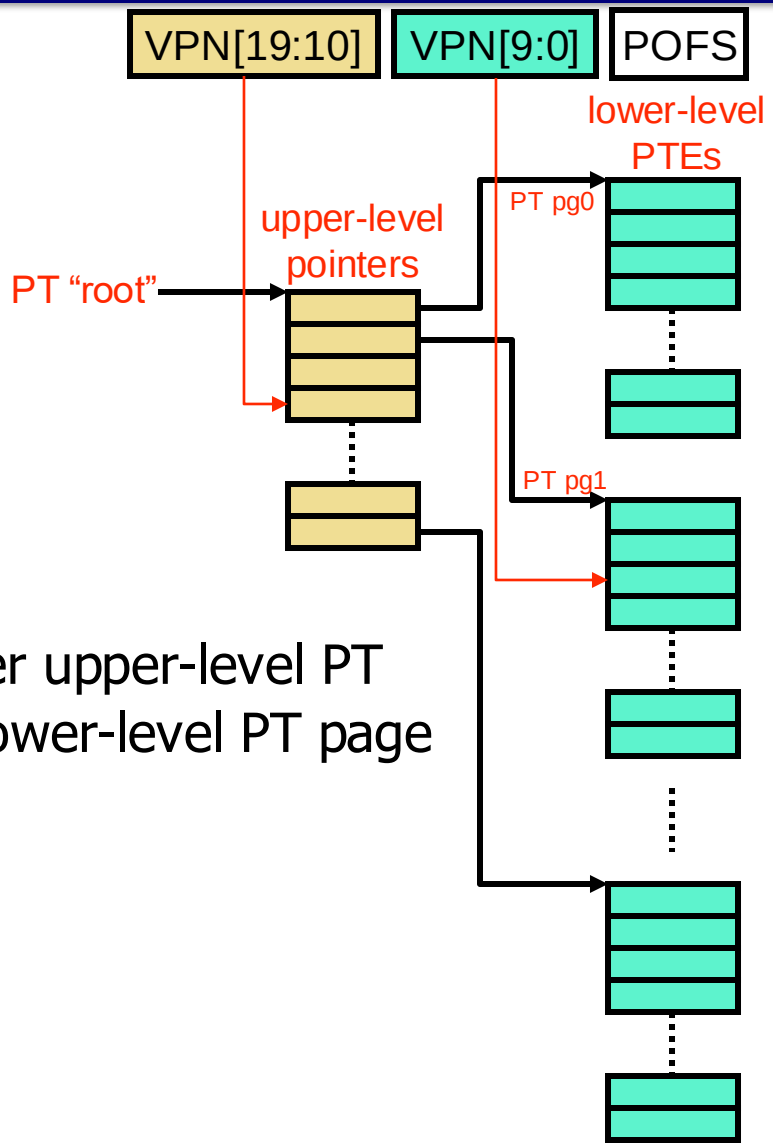
- **PTs can get enormous!**
  - **Can we make them smaller?**

Multi-Level Page Table

- One way to make PTs smaller: **multi-level page tables**
  - Hierarchy of page tables
  - Lowest-level table hold PTEs
  - Upper-level tables hold pointers to pages in lower-level tables
  - Different parts of VPN used to index different levels
  - *Another (better) way is using Inverted Page Tables (see ECE 650)*

- Example: two-level PT for the 32-bit machine on previous slide
  - Compute number of pages needed to address pages in lower-level PT (the one holding PTEs)
    - 4MB/PT / 4KB/pages = 1K pages/PT
  - Compute size of upper-level PT
    - 1K lower-level PT pages → 1K pointers needed in upper-level PT
    - 1K pointers * 4B/pointer = 4KB/upper-level PT

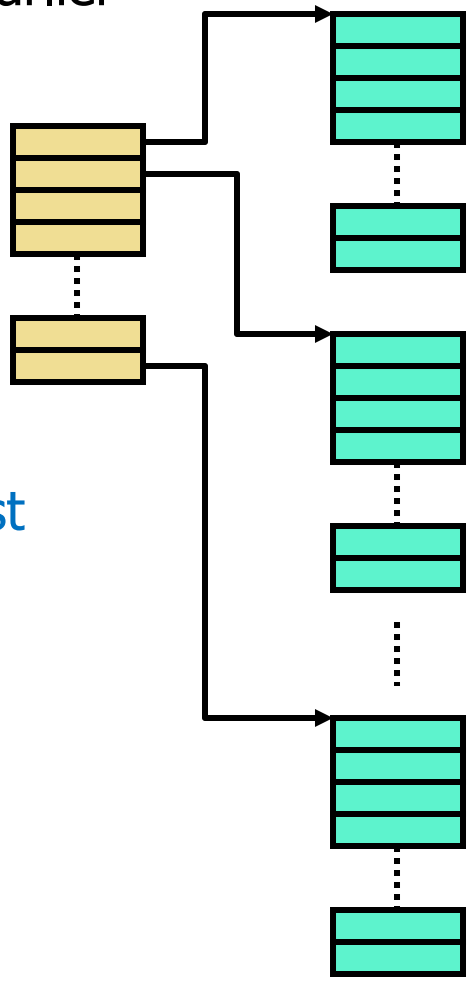
Multi-Level Page Table



- 20-bit VPN:
  - Upper 10 bits index 2^{10} pointers per upper-level PT
  - Lower 10 bits index 2^{10} PTEs per lower-level PT page

Multi-Level Page Table

- Have we saved any space?
  - Isn't total size of lower-level PT the same as earlier (i.e., 4MB)?
  - Not exactly…
- Lower-level PT can be offloaded to disk!
  - Its pages are paged to memory as needed
- Large virtual address regions unused
  - Corresponding lower-level pages need not exist
  - Corresponding upper-level pointers are null



Multi-Level Page Table

- What about a 64-bit machine (with 4KB pages)?
  - Compute number of pages needed to address pages in lower-level PT (the one holding PTEs)
    - 2^{55} B/PT / 2^{12} B/pages = 2^{43} pages/PT
  - Compute size of upper-level PT
    - 2^{43} lower-level PT pages → 2^{43} pointers needed in upper-level PT
    - 2^{43} pointers * 8B/pointer = 2^{46} B/upper-level PT
    - → Still impossible!
    - → Add more levels of indirection! (i.e., more indirect PTs)
      - Adding an additional level of PTs: 2^{37} B/upper-level PT
      - Adding an additional level of PTs: 2^{28} B/upper-level PT
      - …
    - Note that every additional level costs an additional memory access!

- Try changing the page size and see what happens

Address Translation Mechanics

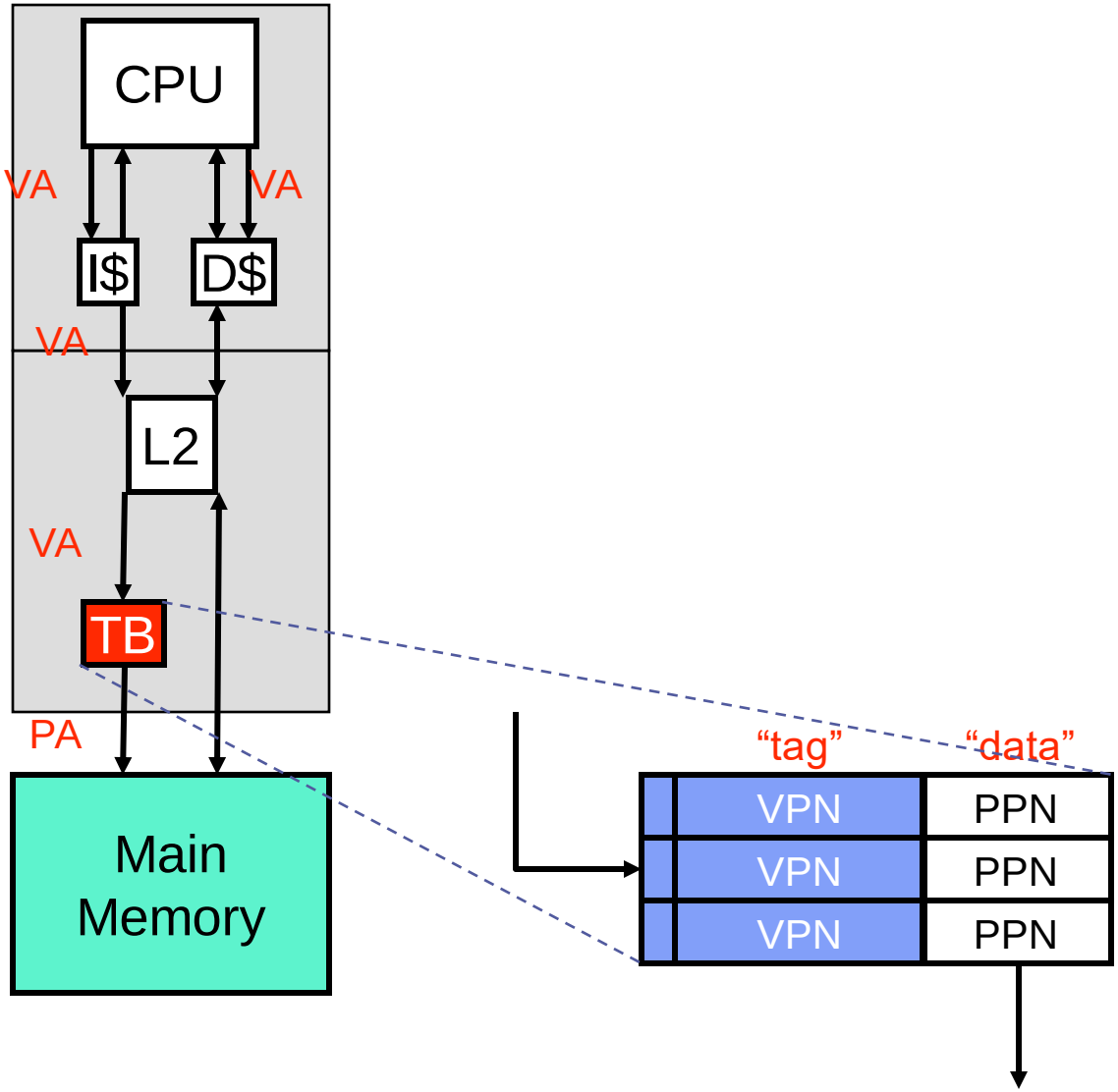
- The six questions:
  - What? Address translation
  - Why? Compatibility, multi-programming, protection
  - How? Page table
  - Where does page table reside? In memory (top-level PT at least)
  - **Who performs it? Operating system (OS)**
    - Cannot be handled by each process alone because we wouldn't be able to manage different processes using the same main memory
    - → We have to let something else handle it that knows about all processes → The OS
  - **When?**

Caching Address Translation

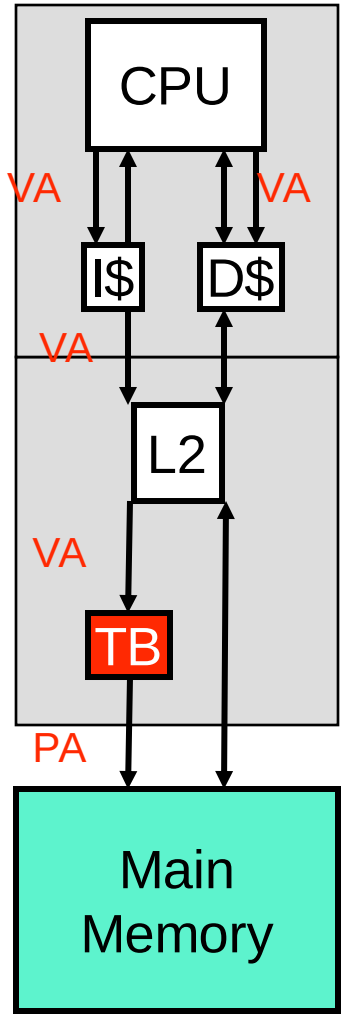
There's Still a Problem…

- OS handling address translation at every L3 miss is too slow!
- We need to accelerate the translation of VA to PA
- Answer: **translation buffer**
  - A special **cache** for page tables
  - A typical example of the way we solve performance problems in computing by adding caches
    - **Performance problem → Add cache!**

Translation Buffer

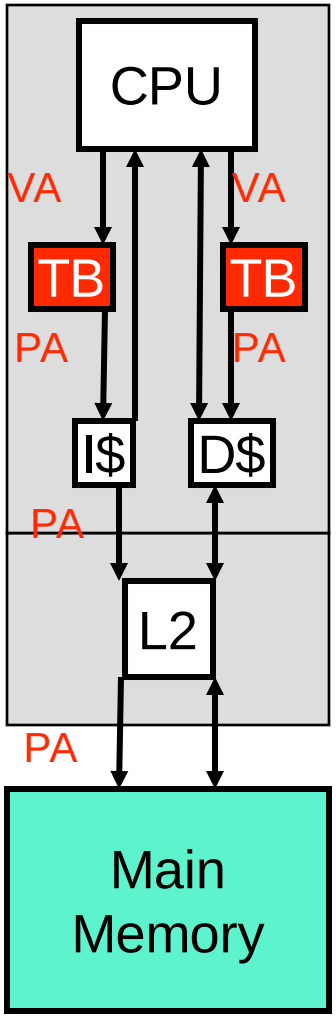


Virtual Caches



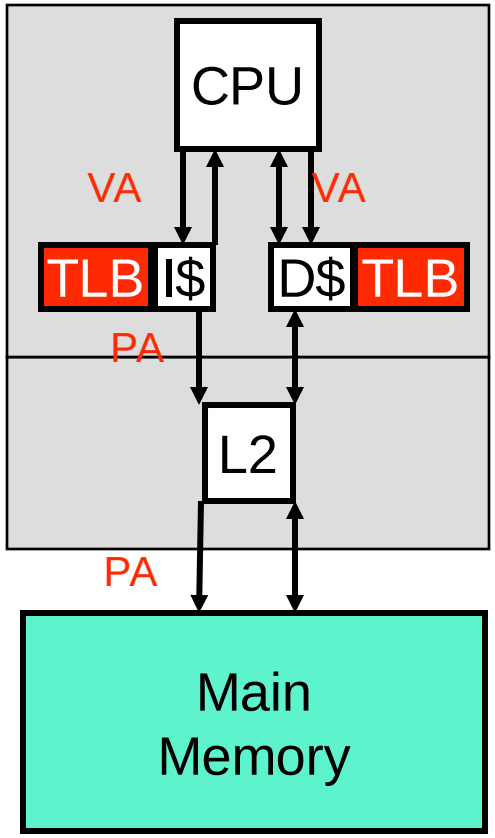
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - \+ Fast: avoids translation latency in common case
- What to do on **process switches**?
  - Flush caches? Slow
  - Add process IDs to cache tags?
- Does inter-process communication work?
  - **Aliasing**: multiple VAs map to same PA
    - How are multiple cache copies kept in sync?
    - Also, a problem for I/O (later in the course)
  - Disallow caching of shared memory? Slow

Physical Caches



- Alternatively: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  + No need to flush caches on process switches
    - Processes do not share PAs
  + Cached inter-process communication works
    - Single copy indexed by PA
  – Slow: adds 1 cycle to t_{hit}

Virtual-Physical Caches



- Compromise: **virtual-physical caches**
  - Indexed by VAs
  - Tagged by PAs
  - Cache access and address translation in parallel
  - \+ No context-switching/aliasing problems
  - \+ Fast: no additional t_{hit} cycles
- → A TB that acts in parallel with a cache:
  - The **Translation Lookaside Buffer (TLB)**
  - A Common organization in processors today

Other Performance Issues

The Table of Time

Event	Picoseconds	≈	Hardware/target	Source
Average instruction time*	30	30 ps	Intel Core i7 4770k (Haswell), 3.9GHz	https://en.wikipedia.org/wiki/Instructions_per_second
Time for light to traverse CPU core (~13mm)	44	40 ps	Intel Core i7 4770k (Haswell), 3.9GHz	http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5
Clock cycle (3.9GHz)	256	300 ps	Intel Core i7 4770k (Haswell), 3.9GHz	Math
Memory read: L1 hit	1,212	1 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L2 hit	3,636	4 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L3 hit	8,439	8 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: DRAM	64,485	60 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Process context switch or system call	3,000,000	3 us	Intel E5-2620 (Sandy Bridge), 2GHz	http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
Storage sequential read, 4kB (SSD)**	7,233,796	7 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage sequential read, 4kB (HDD)**	65,104,167	70 us	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (SSD)	100,000,000	100 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (HDD)	10,000,000,000	10 ms	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Internet latency, Raleigh home to NCSU (3 mi)	21,000,000,000	20 ms	courses.ncsu.edu	Ping
Internet latency, Raleigh home to Chicago ISP (639 mi)	48,000,000,000	50 ms	dls.net	Ping
Internet latency, Raleigh home to Luxembourg ISP (4182 mi)	108,000,000,000	100 ms	eurodns.com	Ping
Time for light to travel to the moon (average)	1,348,333,333,333	1 s	The moon	http://www.wolframalpha.com/input/?i=distance+to+the+moon

* Based on Dhrystone, single core only, average time per instruction
** Based on sequential throughput, average time per block

Thrashing

- If we frequently have to go to disk, the memory latency becomes like disk latency (1000x worse!)

- Need the vast majority of memory accesses to be in memory.

- Remember when our **working set** didn't fit in **cache**?
- It's like that, but now our **working set** isn't fitting into **RAM**!

- This is called **thrashing**
  - When a process is busy swapping pages between disk and memory
  - Makes CPU appear under-utilized, as most time is spent waiting on disk

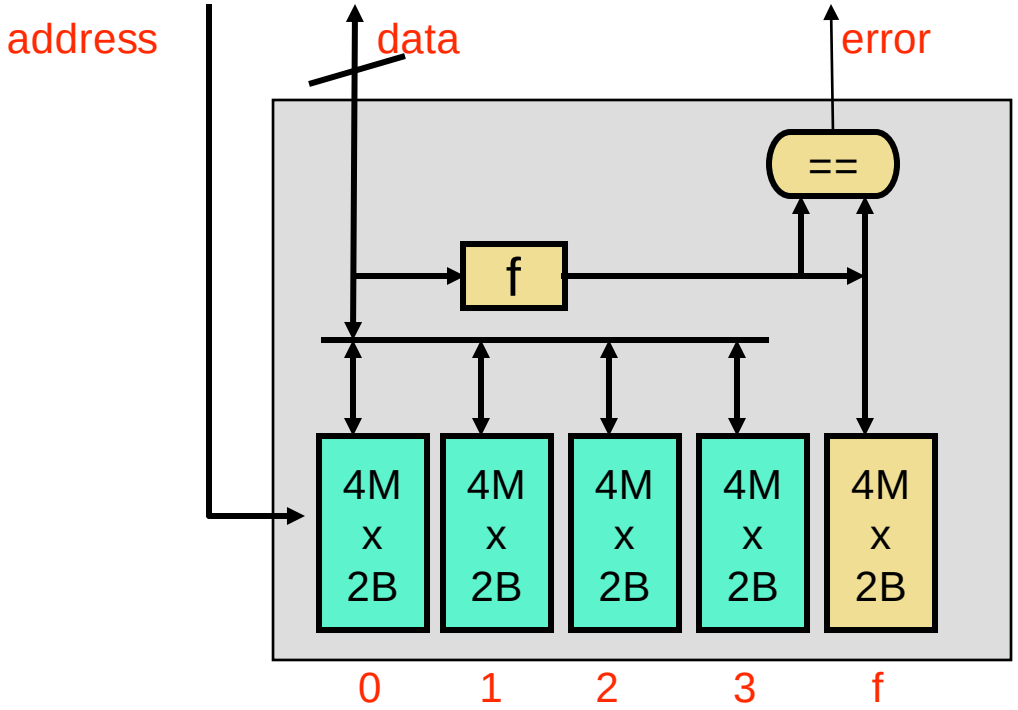
Error Correction in DRAM

Error Detection and Correction

- One last thing about DRAM technology: **errors**
  - DRAM fails at a higher rate than SRAM or CPU logic
    - Capacitor wear
    - Bit flips from energetic α-particle strikes
    - Many more bits
  - Modern DRAM systems have built-in error detection/correction

- **Solution: redundancy**
  - Main DRAM chips store data, additional chips store f(data)
    - #bits |f(data)| < #bits |data|
  - On read: re-compute f(data), compare to stored f(data)
    - Different ? Error…
  - Option I (**detect error**): kill program when error detected
  - Option II (**correct error**): enough information to fix error? fix and go on

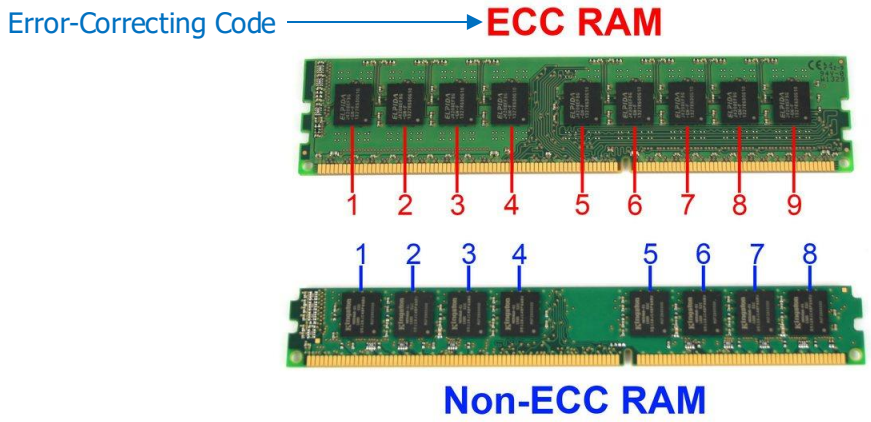
Error Detection and Correction



- Error detection/correction schemes are distinguished by:
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Error Detection Example: Parity

- **Parity**: simplest error detection scheme
  - f($data_{N-1...0}$) = XOR($data_{N-1}$, …, $data_1$, $data_0$)
  + Can detects a single bit flip (this is usually enough)
    - Will miss two simultaneous bit flips
    - But the odds of that happening are usually very low (depends on many factors technology)
  – Cannot correct error: no way to tell which bit flipped
- Many other schemes exist for detecting/correcting errors
  - Take ECE 554 (Fault Tolerant Computing) for more info
  - Also, some of this in ECE 650



ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2024

Interrupts, Exceptions, and System Calls

Rabih Younes
Duke University

Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

External Events

- While running software on a CPU, we might need to accommodate for "external" events, which **may** come at un-expected times, e.g.:
  - Data arrives on network
  - Disk complete read request
  - Fixed interval timer
  - Etc.

- How can we deal with such events?

How to Find Out?

- Suppose we expect an outside event
  - E.g., read request from disk
  - It will get back to us later with data (e.g., after 10 million clock cycles)

- How do we (CPU) know when it's done?
  - Option 1: **Polling**
    - Ask it (disk) periodically
      - "Is data ready yet?" No… "Is data ready yet?" No…
    - Downside: can be inefficient (processor busy asking)
  - Option 2: **Interrupts**
    - CPU switches to another useful task
    - External device (disk) signals to processor when it needs attention (when data is ready)
      - Uses wires in **hardware** for signaling, not code running on CPU

Interrupts

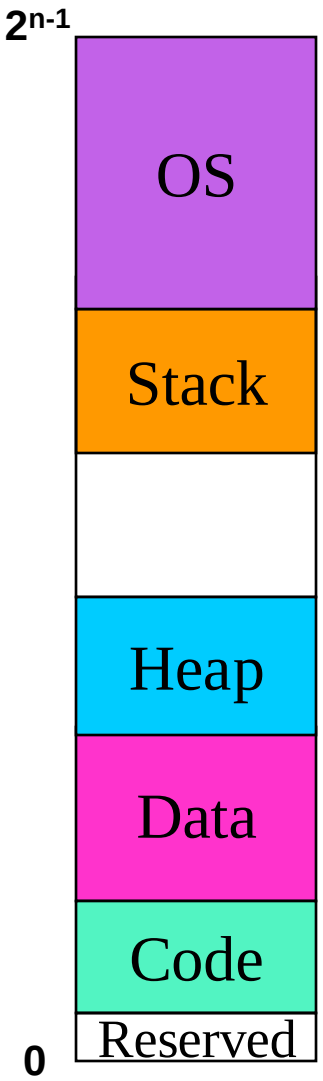
- Step 1: External device raises an interrupt (in hardware)
  - "Hey, processor! I need your attention!"
  - Different interrupt numbers specify which interrupt it is
  - Multiple interrupts at once?
    - **Interrupt controller prioritizes** which one goes to processor
- Step 2: CPU transfers control to OS **interrupt handler**
  - Stops what it's doing (drain pipeline: stall front end until empty)
  - Jumps into interrupt handler (and saves current PC)
  - Switches into **privileged (OS) mode**
- Step 3: OS runs interrupt handler
  - Software routine to do whatever needs to be done
- Step 4: OS returns from interrupt
  - Jumps back to application code, leaving privileged mode

Interrupt Handlers

- How does processor know where to jump?
  - OS sets up the **interrupt vector** during system startup
    - Array of pointers to jump to for each interrupt routine
    - LUT indexed by interrupt number
  - What if….
    - Another interrupt happens while handling the first one?
    - Or an interrupt happens during interrupt vector setup?
    - → OS takes care of it
      - Can enable/disable and prioritize interrupts

Speaking of OS Code… Where Is It?

- Where does OS code reside?
  - In memory….
  - But doesn't application think it has all of memory to itself?
  - Well, sort of…
    - It doesn't think anything exists past the top of the stack…
    - So, the OS "lives" there
    - Same physical pages mapped into **all processes'** virtual address spaces
    - Privileged bit in page table prevents access by "normal" code
      - Mapping only "valid" when in privileged mode



Interrupt Example: Timer Interrupt

- Common interrupt: timer interrupt
  - "Ticks" at fixed interval
  - Gives OS a chance to change currently running program
  - …and keep track of the current time and other things
- This is what lets your computer run multiple programs
  - The OS switches between them quickly
  - Enabled by timer interrupt giving control to OS
- Multitasking also saves time by letting another process run while the current one is waiting, thus using CPU cycles more efficiently (more in ECE 650)

Exceptions

- Interrupts come from external (to the processor) events
  - Asynchronous — don't really "belong to" any current instruction

- Exceptions: unusual circumstances for an instruction running on the processor
  - Belong to one particular instruction
  - Examples:
    - Page fault: load or store missing translation
    - Divide by 0
    - Illegal instruction
      - Bits do not encode any valid instruction of the ISA
      - Or, privileged instruction from user code

Interrupts vs. Exceptions

- Exceptions:
  - Processor must (typically) tell OS which instruction caused it
  - OS may want to restart from same instruction
    - Example: page fault for valid address (on disk)
  - Or OS may kill program:
    - Segmentation fault: (or other fatal signal)
    - Aside: OS sends "signals" to program to kill them
      - Segfault = SIGSEGV
      - Programs can "catch" signals and not die…
      - But not in this class…
- Interrupts: no particular instruction
  - But OS will always restart program after last complete insn
- Both require a **precise state: insns either done, or not**
  - Division between "done and not done" in program order
  - Can't be half done

Precise State

- Instructions either done or not: sounds obvious right?
  - Problem: "half done" instructions: pipeline splits into stages
    - Need to ensure **no** state change (reg or mem) if not done
  - Also: need "clean" division
    - Instructions before exception: all done
    - Instructions after (and including) exception: no effect

- For interrupts:
  - Must be precise, but division can be anywhere.

Handling Exceptions

- Exceptions handled just like interrupts
  - Some ISAs just give them interrupt numbers
  - Others have separate numbering for exceptions

System Calls: Exceptions on Purpose

- Programs need OS to do things for them
  - Read/write IO (input/output) devices, including printing, disks, network, …
  - Tell "real" time of day
  - Spawn new processes / execute other programs
  - …
  - Any interaction with the "outside world" of the program

- Programs make a system call (`syscall`) to do this
  - Special instruction which **traps** into OS
  - Basically, just causes exception — specifically for this purpose
  - OS gets control (in privileged mode), and does what program asked
    - Knows what program wants by arguments set in specific registers
    - OS code may deny request and not do it… then returns an error

System Calls: Kind of slow

- Bothering OS for stuff: kind of slow
  - Empty pipeline…
  - Transfer control/change privilege
  - Have OS figure out what you want…
  - Then do it…
  - Then drain pipeline again
  - Then jump back into program

- For long tasks, overhead to enter/leave is amortized
  - Reading disk (very slow)
- For short tasks, overhead is very high
  - Get current time of day

Avoiding Slowness

- Userspace (not OS) libraries help avoid by buffering
  - Example: malloc
  - Malloc does not ask OS for more memory on every call
  - Instead, malloc asks OS for large chunks of memory
  - Then manages those chunks itself (in user space)
  - Pedantic annoyance: malloc is not a system call!

Vsyscalls: a slick trick

- Linux has a slick trick: vsyscalls
  - Don’t actually make a system call!
  - Example: get current time of day
    - Just needs to read an int (time in seconds)
  - OS maps vsyscall page into all processses
    - Read/execute only
    - All processes map to same physical page
  - OS writes current time to fixed location on this page
    - On each timer interrupt
  - gettimeofday “system call” actually not a system call
    - Just library function which jumps onto vsyscall page
    - Code there reads time and returns it

ECE 550D
Fundamentals of Computer Systems and Engineering

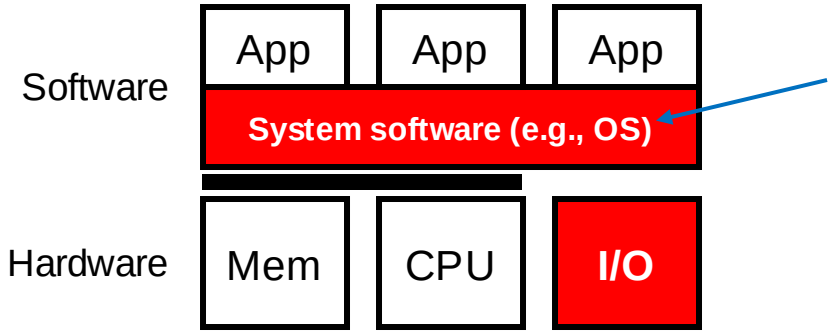
Fall 2024

Introduction to the Operating System (OS)

Rabih Younes
Duke University

Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

Operating Systems



- OS is not always needed
- We use it when we need some of its services:
  - Memory management (virtual memory)
  - Process management
  - File management
  - Input/output management
  - Security and Privacy
  - …

- In this lecture, we'll focus on:
  - File management (can also read this)
  - Process (instance of a program) management
  - Bootstrapping (how the system starts up)

File Systems

Filesystems

- Files exist outside the memory
  - On disk
- How do we store files/directories on the disk?
- Essential concept: disks store data in chunks that we can reference, called **blocks**
- A **Filesystem** imposes structure on that data
  - Directories that contain files
  - Files that contain data
    - …and meta-data: access time, ownership, permissions,…

Filesystems

- Popular filesystems: NTFS, fat32, ext2, ext3, ext4, …

- Filesystem are made of **blocks**
  - Fixed size allocations of space (e.g., 4KB)
  - Can hold file data or filesystem information
- Blocks are organized into **block groups**

- Block Group locations in table after superblock
  - Array specifying where block groups start
- Superblock: describes key info about filesystem
  - One per filesystem
  - But replicated (to avoid single point of failure)
  - At fixed locations

Block Groups

- Block Group Descriptor Table
  - Array telling where each block group starts
  - One or more blocks (superblock says how many)
  - Location in memory: after superblock

- Block groups
  - Many blocks with good spatial locality (e.g., same cylinder in HDD)
  - Use one block to track free data blocks
  - Another block to trace free **inode** blocks
  - Main point: spatial locality — try to allocate blocks within same group

How to Represent a File in a UNIX Filesystem?

- **Inode** (index node) is a data structure in Unix-style filesystems that describe the filesystem's objects

- Each inode contains information about a **file**
  - Owner
  - Permissions
  - Access time
  - **Where data blocks are located** (for this file)
  - Number of blocks used
  - …
  - All meta-data about a file **except its name**

- Inodes have a **fixed size** of 256 bytes

Inode: Where to Find File Data?

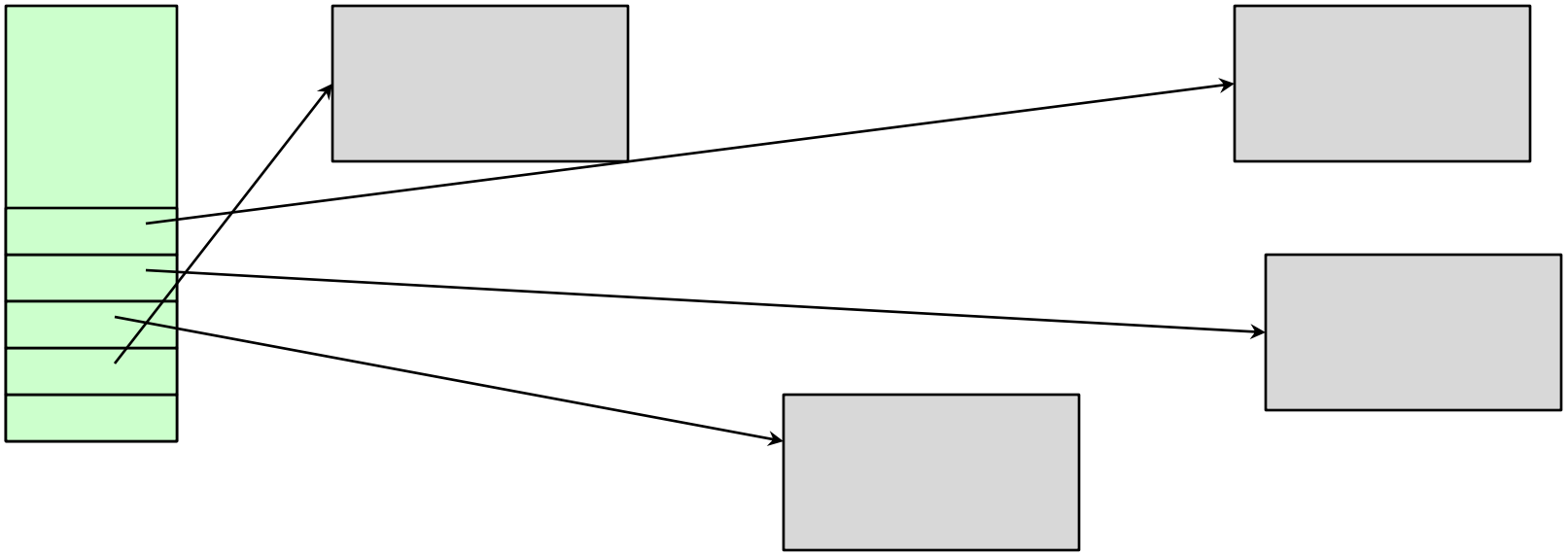
- How do inodes specify where the data blocks reside?
  - Pointers (e.g., block numbers/IDs) to the data

- Solution 1: Direct pointers in inodes
  - Pros?
  - Cons?

Inode: Where to Find File Data?

- How do inodes specify where the data blocks reside?
  - Pointers (e.g., block numbers/IDs) to the data

- Solution 1: Direct pointers in inodes
  - Pros: Fast (read inode, read data)
  - Cons: Small limit on file size (~16 pointers * 4KB/block = 64KB files)



Inode: Where to Find File Data?

- How do inodes specify where the data blocks reside?
  - Pointers (e.g., block numbers/IDs) to the data
- Solution 1: Direct pointers in inodes
  - Pros: Fast (read inode, read data)
  - Cons: Small limit on file size (~16 pointers * 4KB/block = 64KB files)
- "I can't store large files" = functionality problem
  - Solution?

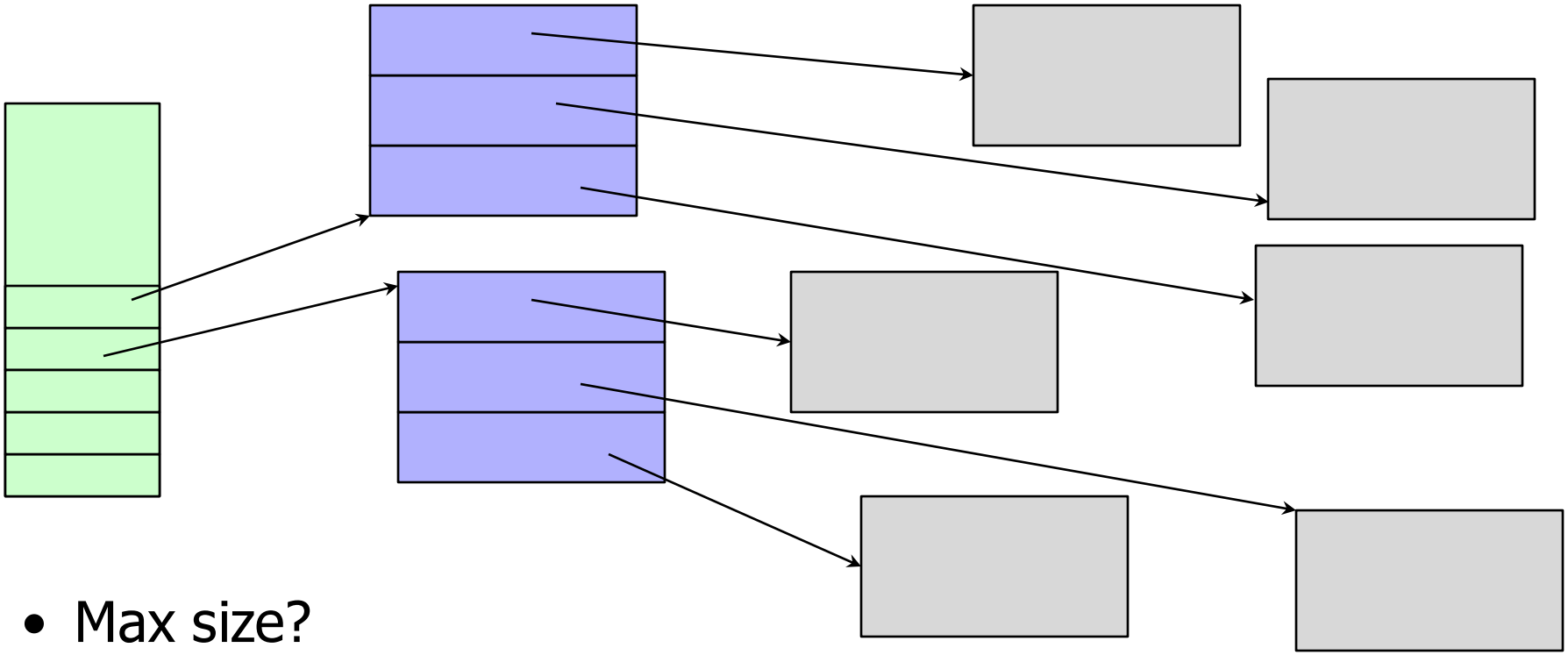
Inode: Where to Find File Data?

- How do inodes specify where the data blocks reside?
  - Pointers (e.g., block numbers/IDs) to the data

- Solution 1: Direct pointers in inodes
  - Pros: Fast (read inode, read data)
  - Cons: Small limit on file size (~16 pointers * 4KB/block = **64KB** files)

- "I can't store large files" = **functionality problem**
  - Solution? **Add a level of indirection**
    - Inode has pointers to blocks containing pointers to data
      - Max file size?

Inode: Indirection



- Max size?
  - 16 pointers, each to a 4KB block
  - 1K (4KB block / 4B pointers for 32-bit systems) pointers per block
  - Each referencing a 4KB block of data
  - →16 * 1K * 4KB = **64MB** files
  - Ok… better… but we still need to store/access bigger files

Inode: More Indirection

- 2 levels of indirection:
  - ~16 ptrs in inode * 1K 1st level * 1K 2nd level * 4KB = ~**64 GB**
  - Better, but we still might need more?

- 3 levels of indirection?
  - **64 TB**: probably big enough….
  - But kind of **slow**? Now need 5 disk reads to get the data?
    - (Inode, 1st lvl, 2nd lvl, 3rd lvl, Data)
  - Might be willing to pay this price if using a 100+GB file… but what about a tiny little file?

Real Inodes: A Mix of Approaches

- Real inodes mix approaches for best of both worlds
  - 12 direct pointers (first 48KB of data)
  - 1 indirect pointer (next 4MB of data)
  - 1 doubly indirect pointer (next 4GB of data)
  - 1 triply indirect pointer (next 4TB of data)

- Example of "**make the common case fast**"
  - Small files are accessed very fast
  - Slow access for really large files
    - This rarely happens
    - Can also cache indirect block tables to speed things up

Stepping Back a Level

- Inodes have meta-info on files
  - Including how to find its data
  - Not including file names
    - File names exist in **directories**

- How do we find files?
  - We organize them into **directories**
    - E.g.: `/Users/rabihyounes/Desktop/`
  - How do we store directories?
    - They are just **files** too!

UNIX File Types

- UNIX has multiple file types
  - All have inodes
  - File type is in the inode

  - **Regular files**: what you think of for files (contain data)
  - **Directories**: contain a list of {name, inode #} pairs
  - **FIFOs**: aka named pipes
    - Allow two processes to communicate via a queue
  - **Symlinks**: a symbolic link to another file
    - Contains the path to the other file
    - But accessing it takes you to the other file
  - **Devices (char/block)**: interface to hardware devices
  - **Sockets**: for inter-process communication
    - Similar to FIFOs, but different

ECE 650

Directories

- Directories contain {name, inode #} pairs
  - Iterate through them looking for the name you want
    - Then go to inode #
  - Want a sub-directory? Works the same as other files
  - Also, 2 special names: '.' and '..'
    - . = current directory (maps back to own inode #)
    - .. = parent directory (maps back to parent inode #)
    - They are created automatically in every directory and can't be deleted

- Some types of filesystems support more scalable directory lookup

Filesystem vs Swap Space

- Filesystem is for files
  - But disk also used for virtual memory ("swap space")
- Different **partitions** of the disk used for each
  - May also have multiple file systems on multiple partitions
  - File systems are **mounted** at some path, then look identical to normal directories to user
- **Swap space**: managed differently
  - Temporary (no need to remember layout across reboot)
  - Fixed-size: always operate on a page at a time
  - Kernel can just track what is free/in use, where each page is
  - More about this in ECE 650

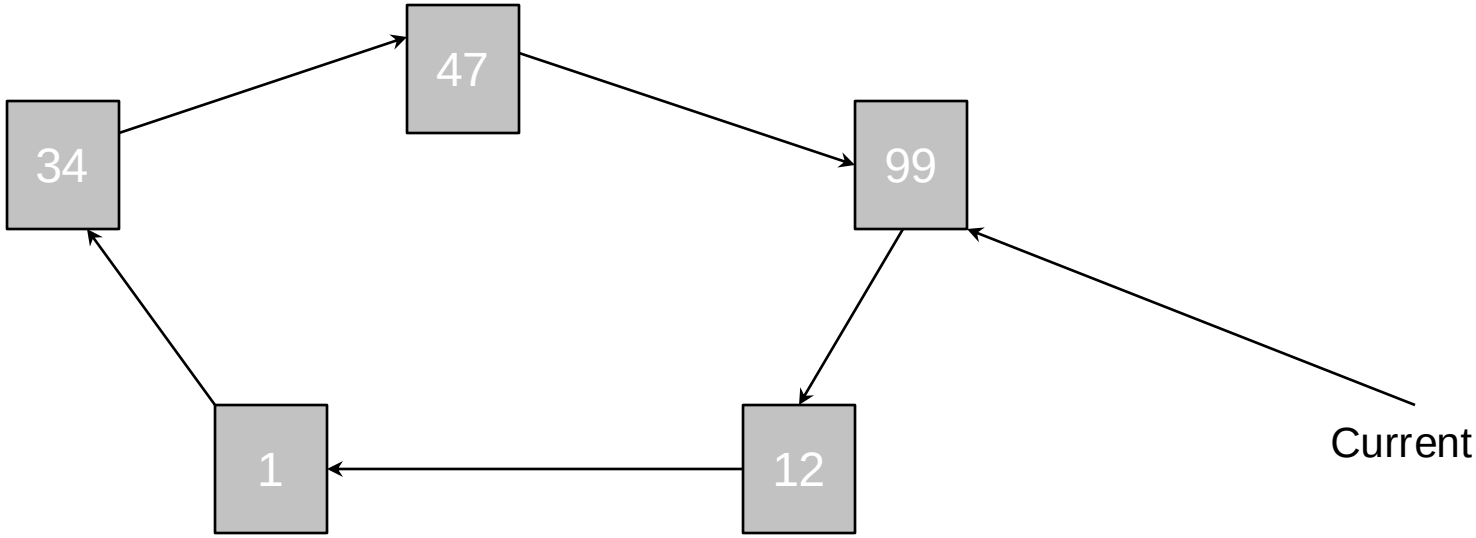
Processes

Processes

- A process is a running **instance of a program**, e.g.:
  - Program: Terminal
  - May run N (e.g., 4) Terminal processes at once

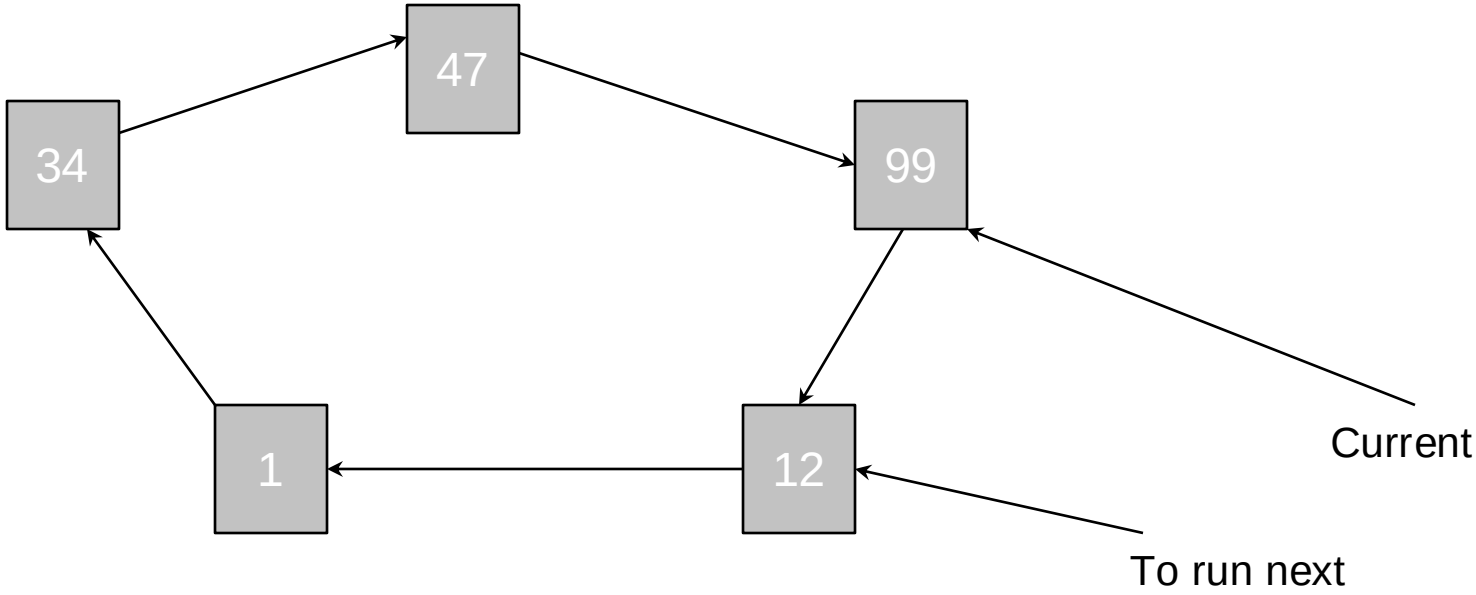
- Each process has a process id (**pid**)
  - A number which uniquely (at the time) identifies the process

Process Scheduling



- OS maintains a scheduler queue data structure
  - Basic: circular queue, round robin
  - Fancier: priority-based scheduling, fancy algorithms, …
- OS remembers which process is currently running

Process Scheduling

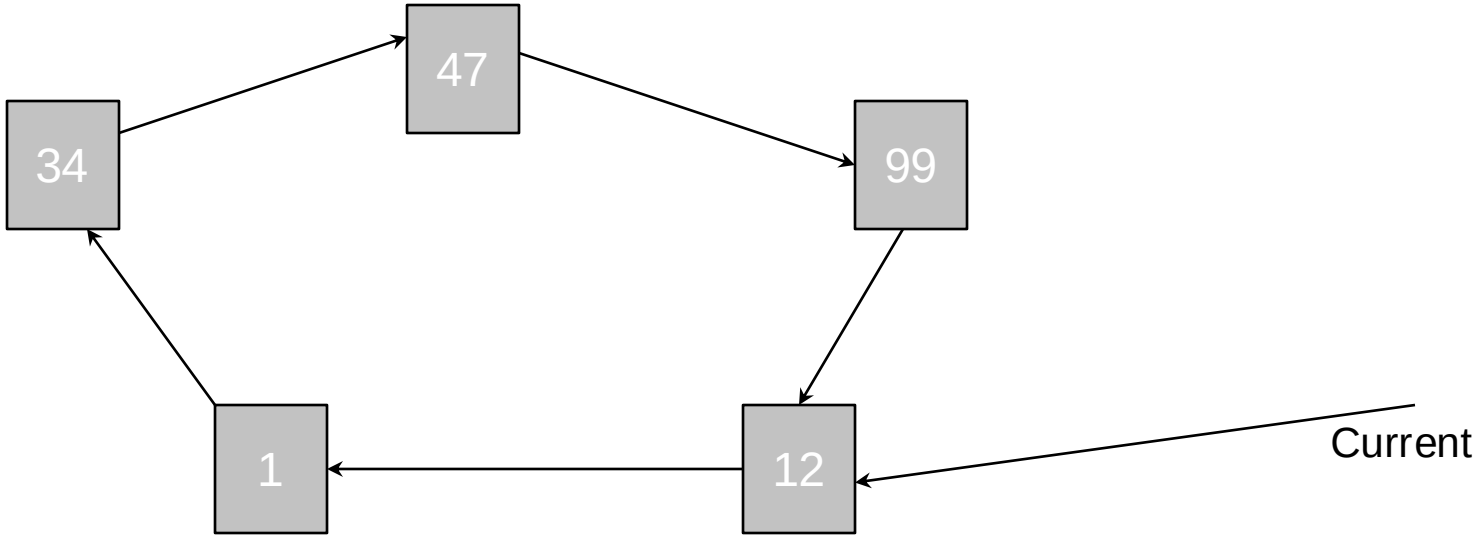


- Timer interrupt drives scheduling
  - Interrupt occurs: scheduler figures out what to run next
  - Some processes may not be runnable right now
    - E.g., waiting for disk

Context Switching

- To change currently running process, OS performs what is called a **context switch**
  - Save all registers into OS's per-process data structure
    - Elements of scheduler list are large structs
  - Change processor's page table root to point to PT of new process
  - Load registers for new process
  - Return from interrupt
    - Leave privileged mode
    - Jump back to saved PC for the process that will now run

Process Scheduling



- Now new process runs until interrupt or exception
  - Note: OS only entered by interrupts, exceptions, or syscalls
- If no process is runnable, kernel has an "idle task"
  - Tells processor to go to sleep until next interrupt

Process Creation

- Processes come from duplicating existing processes
  - `fork()`: make an exact copy of this process, and let it run
  - Forms a parent-child relationship between old and new process
    - We can tell which process is running this same code by the return value of `fork()`
      - `fork()` returns 0 → this is the child's process' code
      - `fork()` returns >0 → parent (return value = child's pid)
  - No guarantees which of them the scheduler runs first
    - Or they can both run at same time if CPU has multiple cores

What About Running New Programs?

- If we just duplicate existing processes, how to run anything else?

- `fork()` can be followed by **`exec()`**
  - `exec()` takes the filename for program binary from disk
  - Loads that program into the current process' memory
    - Destroying anything currently in it
    - Resetting stack and heap pointers
  - Sets PC to be the starting PC of the program (stored in the binary)
  - …and never returns (except on error)
    - Instructions after it are never executed
    - But a process can be `kill()`-ed

Fork-then-Exec…wasteful?

- Fork: make duplicate copy of process
- Exec: overwrite with newly loaded program

- Seems wasteful to make a copy of everything
  - Then throw it away?

- Imagine: Big complicated application (2GB memory)
  - Wants to run external command (often)
    - `fork()`: copy 2GB memory
    - `exec()`: discard copy to load new program

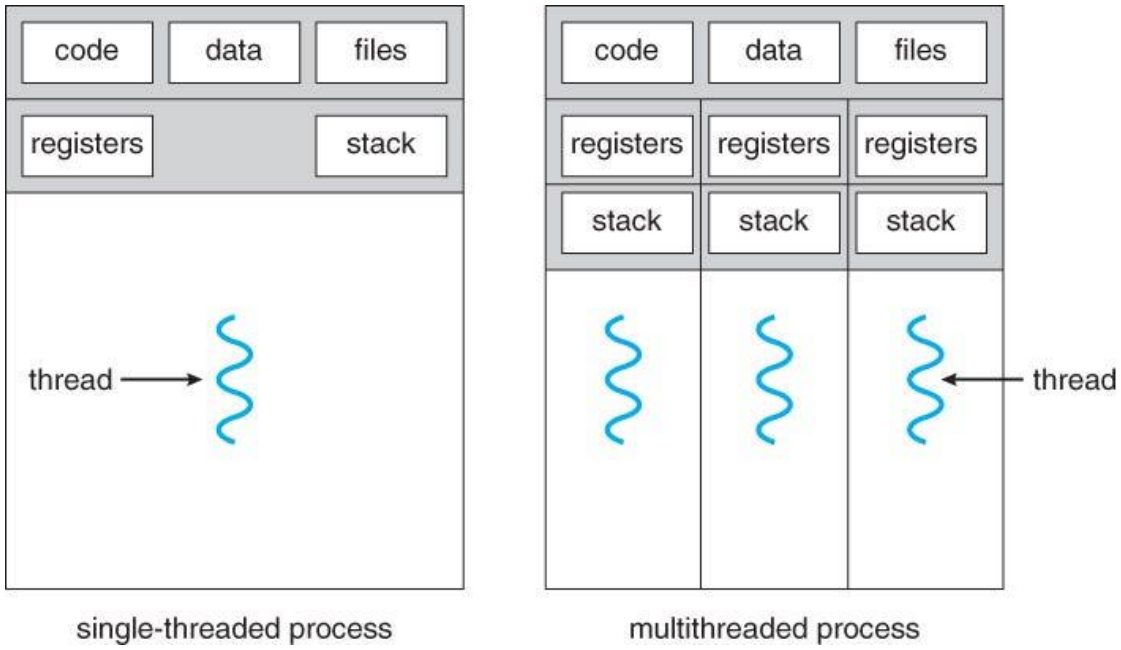
Copy-on-Write: page table magic

- Virtual memory hackery to the rescue
  - Instead of copying all of memory, **just copy page tables**
    - Two programs now have PTs pointing at the same physical pages
  - Now, mark each page read-only
    - Writes will cause page-faults
  - Kernel remembers it did this, and copies the page on a write
    - Then marks it writeable, and resumes the process

  - Exec? Only copy page tables!
  - No exec? Copy page tables up front, then copy pages as written

Multiple Threads

- A process may also have multiple **threads**
  - Threads execute concurrently, but **share virtual address space**



- Correct programming with threads requires **synchronization**
  - Locks
  - Barriers
- More in ECE 650

What About The First Process?

- If each process comes from `fork()`ing, how do we get the first process?
- For that matter, how do page tables get set up?
- And… how does the system start in general?

Booting

Booting the System

- Booting is architecture specific: we'll talk about x86_64
  - Processor initializes in 16-bit real mode
    - Virtual memory is off (real mode = use real addresses)
      - Real address is another word for physical address
    - Execute BIOS (low-level firmware) startup code
      - Splash screen/startup/press DEL to enter setup
    - BIOS reads Master Boot Record (sector 0) of hard disk
      - Loads contents into memory and jumps into it
      - This code is tiny (440 bytes)
      - First stage of **bootloader**
    - This (tiny) code loads more data (code) from disk
      - Then loads stage 2 bootloader
      - Asks BIOS to do disk IO for it

Booting continued

- Stage 2 bootloader
  - May present menu, ask for options, etc.
  - Then loads kernel — requires reading filesystem
  - Then jumps to kernel entry point

- Now the actual OS kernel is in control
  - Still in 16-bit real mode
  - Sets up page tables
  - Sets up interrupt vector
  - Sets up a few other x86-specific things
  - Enters “protected mode” (switches to 64 bit with virtual memory)
  - Creates idle task and spawns init (pid 1, from /bin/init)

Init

- Init: First "normal" program
  - OS loads /bin/init as pid 1
  - Init reads configuration file (in /etc)
  - Spawns other programs (e.g., /bin/login, sshd, etc.)
    - Done with normal fork()/exec()
  - Periodically reaps orphaned processes

Much more to it

- Could spend whole semester on OSes
  - Barely scratched surface with an overview
  - If this were an OS class, we would
    - Write kernel modules
    - Modify the linux source
    - Make our own filesystem (?)
    - Fiddle with the scheduler
    - Go into much more detail on all these topics
    - Cover a bunch of other topics
- Take ECE 650 for more about OS

ECE 550D
Fundamentals of Computer Systems and Engineering

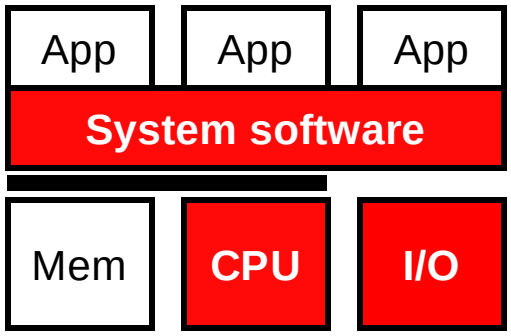
Fall 2024

Input/Output (IO) Devices

Rabih Younes
Duke University

Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

IO: Interacting With the Outside World



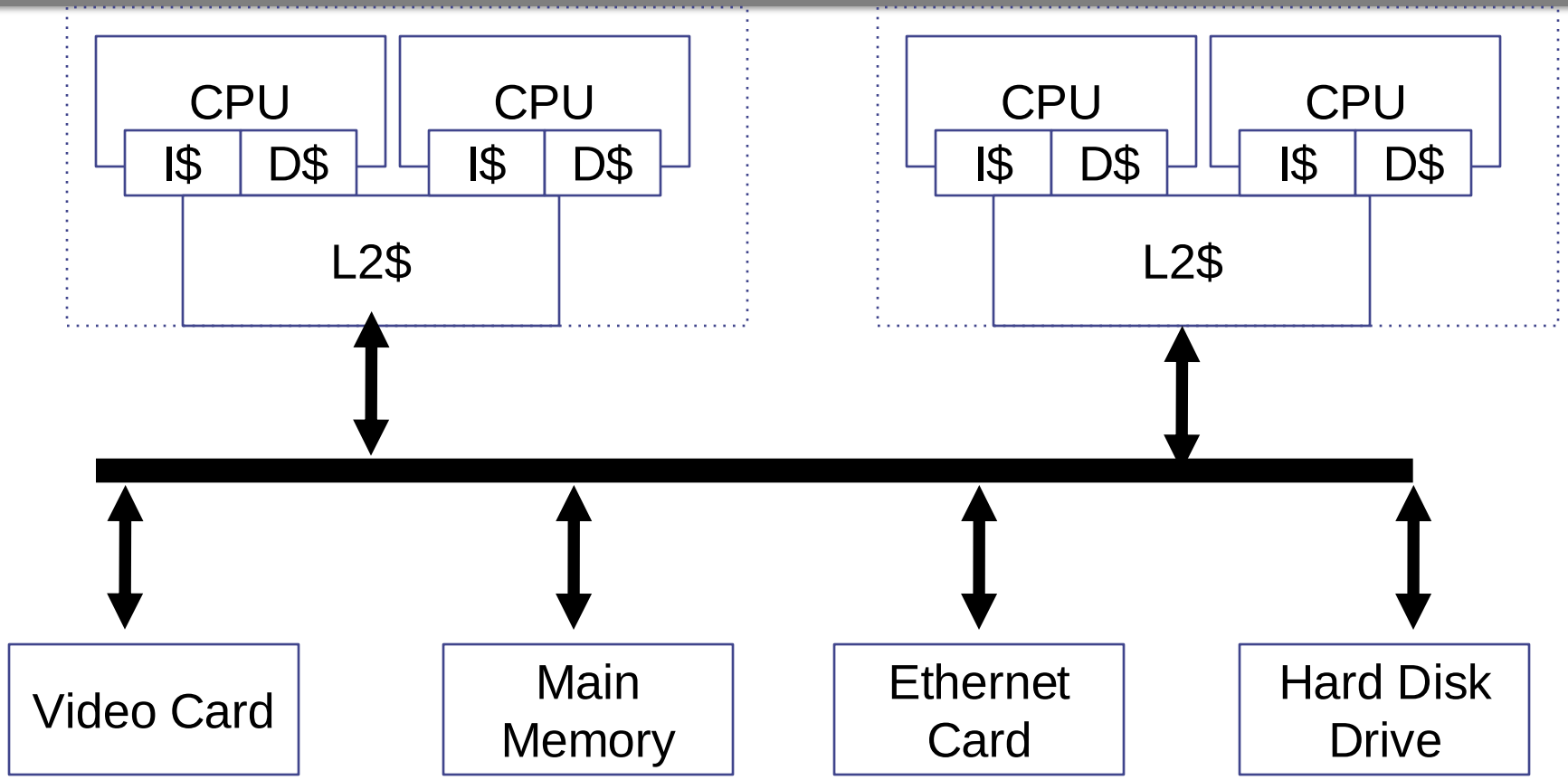
- Input and Output Devices
  - Video
  - Disk
  - Keyboard
  - Sound
  - …

Communication with IO Devices

- Processor needs to get info to/from IO devices

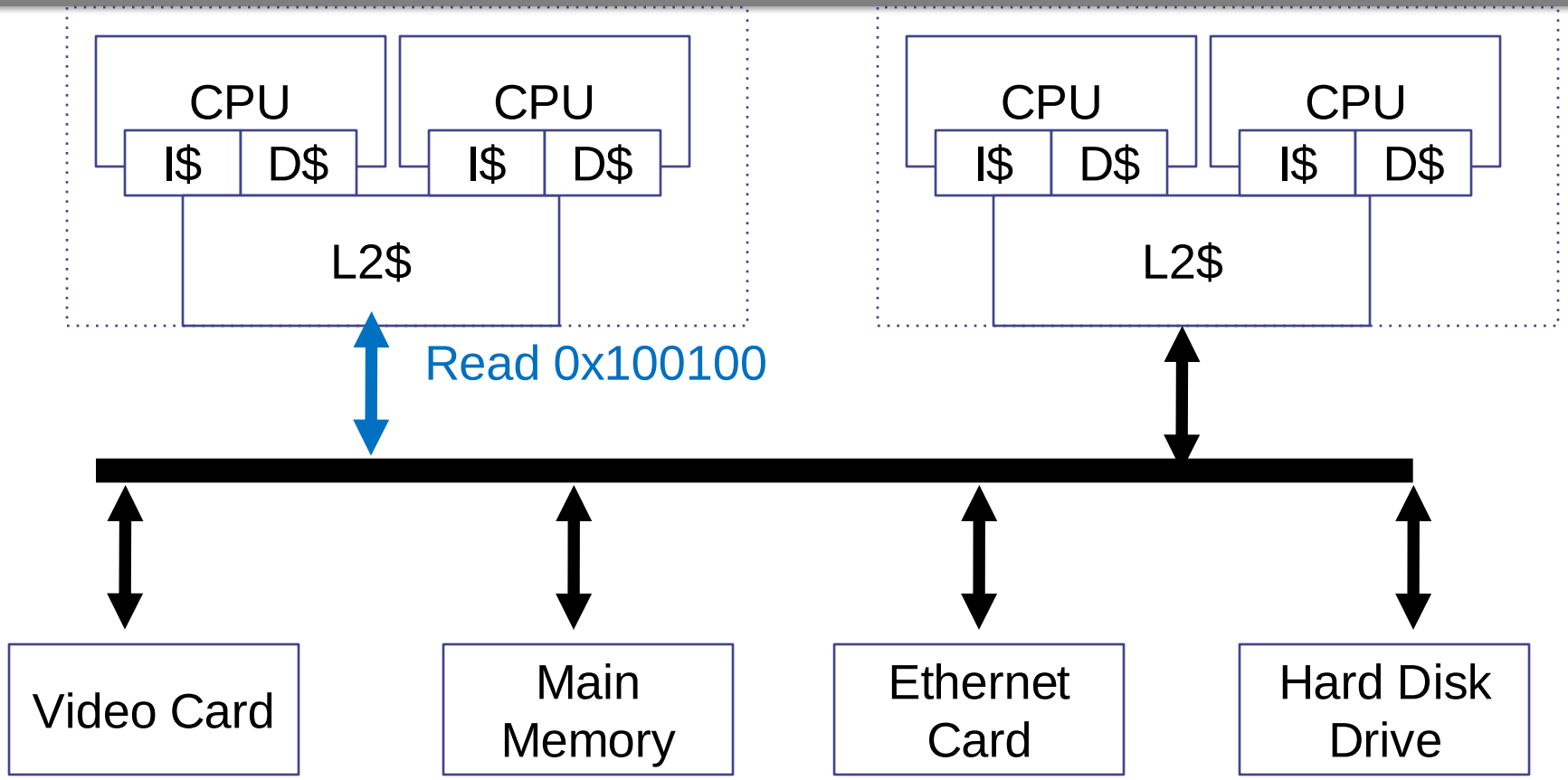
- Two ways to achieve it:
  - In/out ISA instructions
    - Read/write value to "IO port"
    - Devices have specific port numbers
  - Memory mapped
    - Regions of physical addresses not actually in DRAM, but mapped to IO device
      - Stores to mapped addresses send info to device
      - Reads from mapped addresses get info from device

Memory Mapping Example



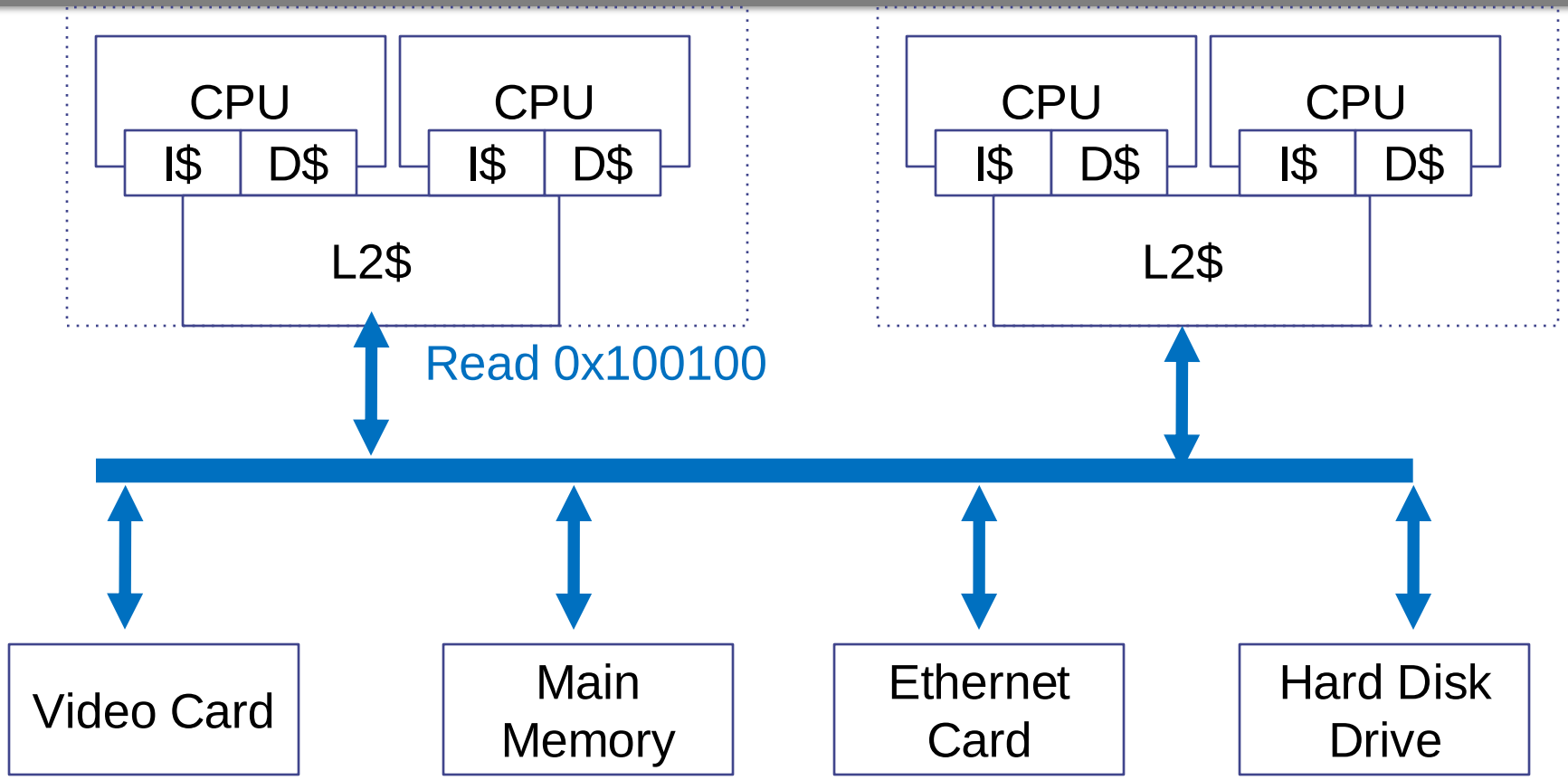
- 2 "socket" system (each with 2 cores)
- Real systems: more IO devices, maybe more cores, …

Memory Mapping Example



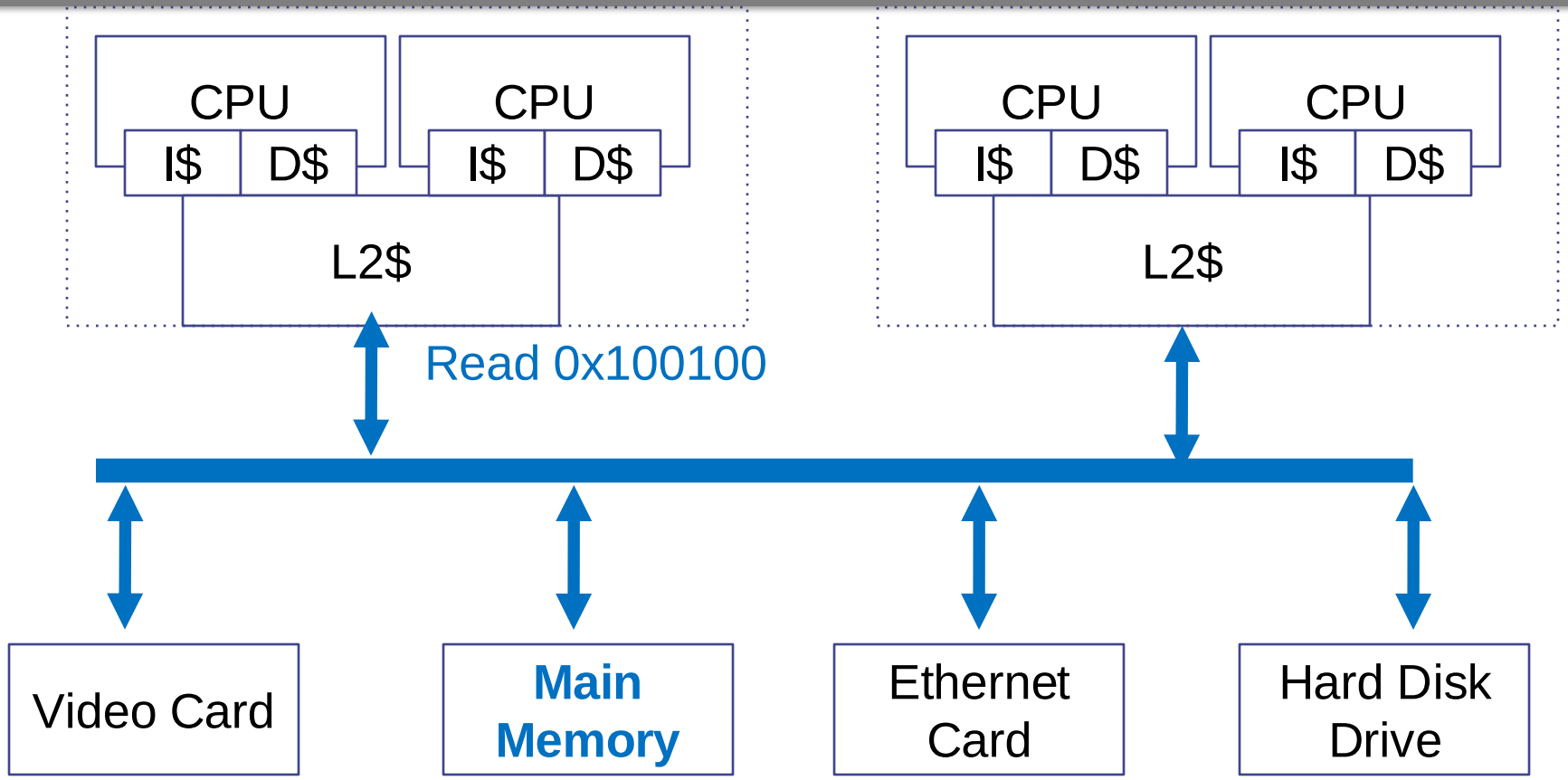
- Chip 0 requests read of 0x100100

Memory Mapping Example



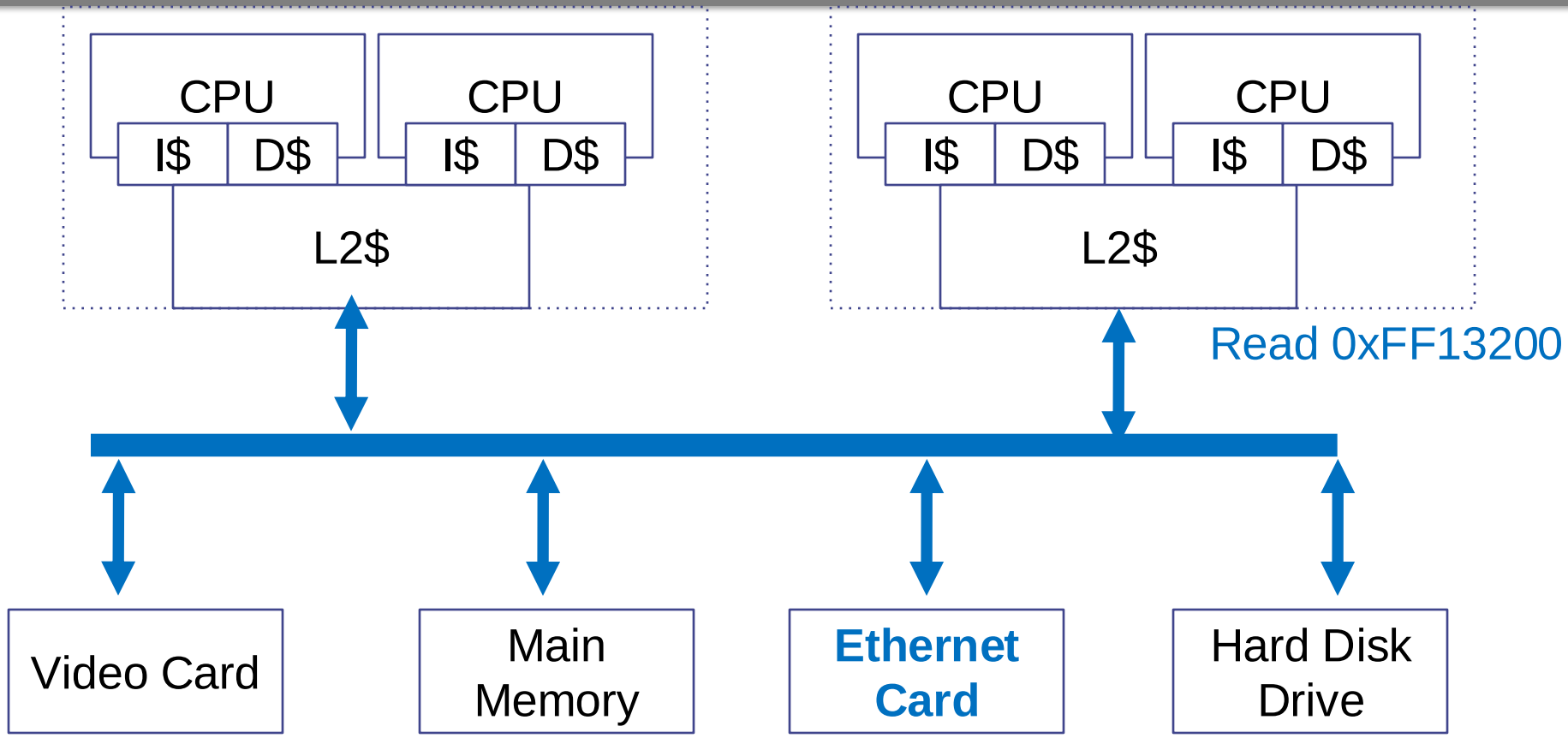
- Chip 0 requests read of 0x100100
- Request goes to all devices

Memory Mapping Example



- Chip 0 requests read of 0x100100
- Request goes to all devices, which check address ranges

Memory Mapping Example



- Other address ranges may be for a particular device

A Simple "IO device" Example

- Read (physical) address 0xFFFF1000 for "ready"
  - Tells about the status of IO device
- If ready, read address 0xFFFF1004 for data value

```
read_dev:
li $t0, 0xFFFF1000
loop:
   lw $t1, 0($t0)
   beqz $t1, loop
   lw $v0, 4($t0)
   jr $ra
```

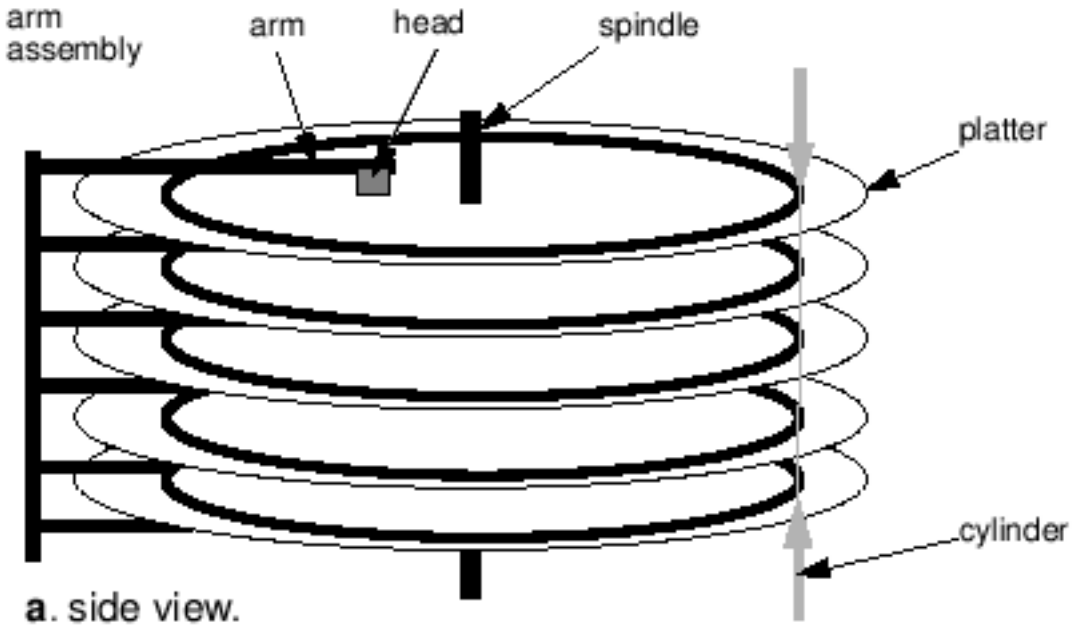

Interrupt handler function for reading ready data from device after its corresponding interrupt occurs

A handful of questions…

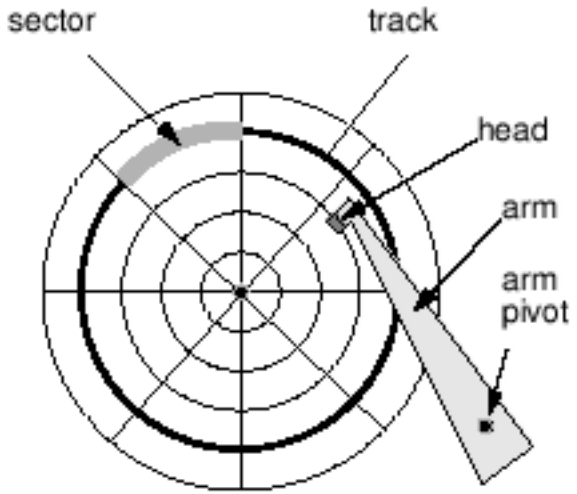
- How do we use physical addresses?
  - Programs only know about virtual addresses, right?
  - Only OS accesses IO devices:
    - OS knows about physical addresses, and can use them

- What about caches?
  - Won't the first lw bring the current value of 0xFFFF1000 into the cache?
  - And then subsequent requests just hit the cache?
  - **Pages have attributes, including cacheability**
    - IO mapped pages marked non-cacheable
      - Remember, many IO devices "generate" data in a specific order and for specific purposes over time, independently from the CPU (main difference between IO memory and main memory)
    - This also prevent speculative loads (e.g., out-of-order streaming)

Important IO Device: Hard Disc Drive (HDD)



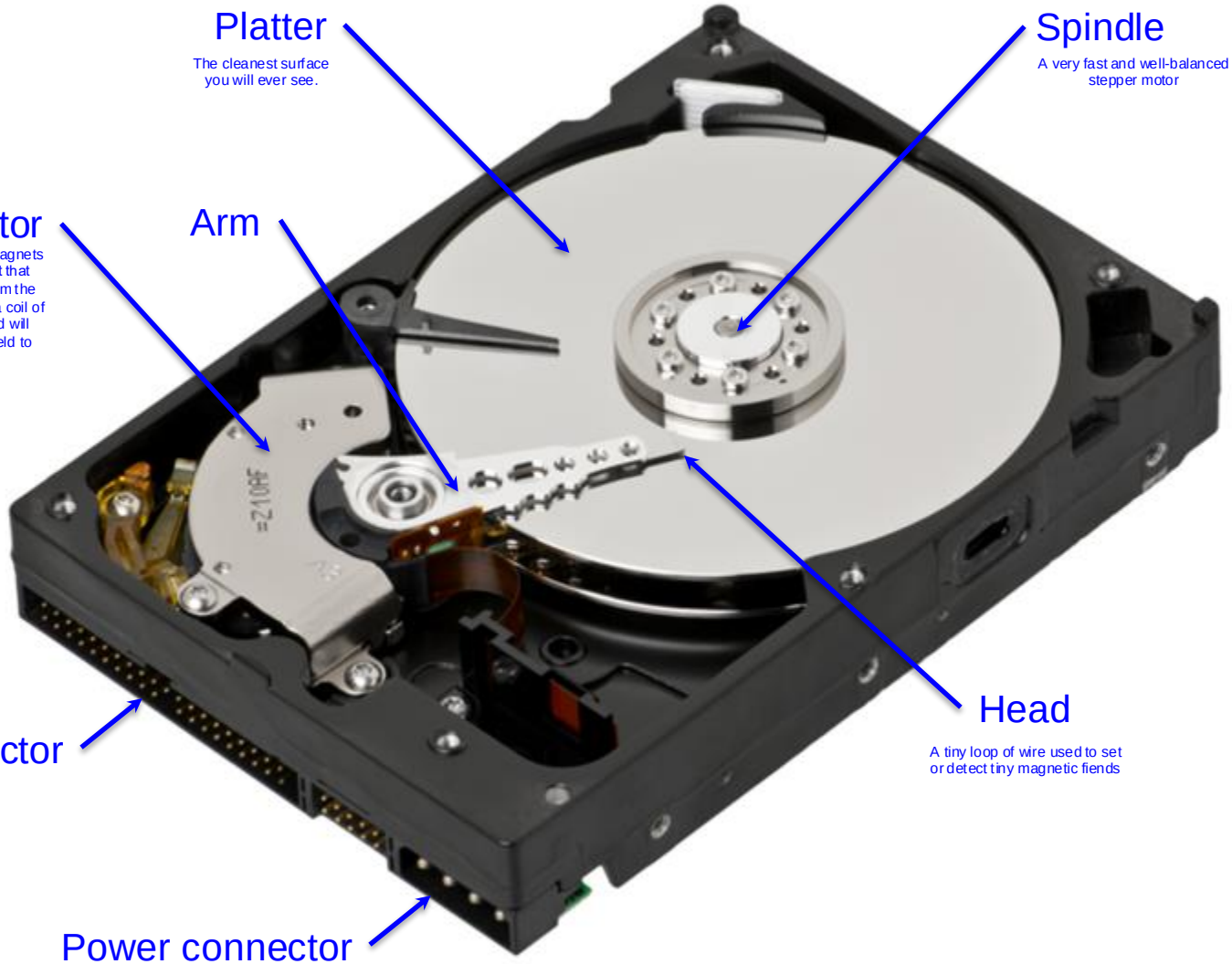
a. side view.



b. top view.

- Disks are circular platters of spinning metal
- Multiple tracks (concentric rings)
- Each track divided into sectors
- Modern disks: addressed by "logical block"

HDD Internals



Platter
The cleanest surface you will ever see.

Spindle
A very fast and well-balanced stepper motor

Actuator
Two extremely powerful magnets with a "mumetal" bracket that shields magnetic field from the rest of the drive. Inside is a coil of wire that when energized will swing in the magnetic field to move the arm.

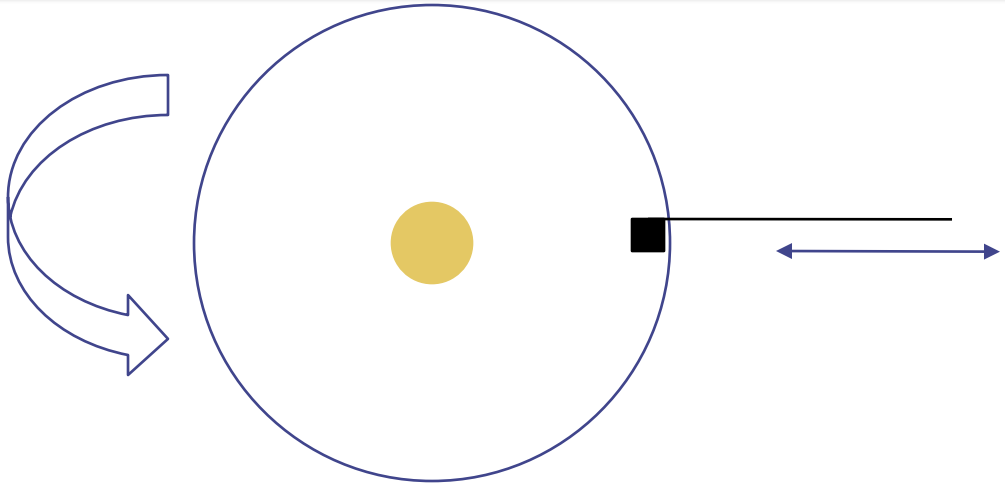
Arm

IO connector

Head
A tiny loop of wire used to set or detect tiny magnetic fiends

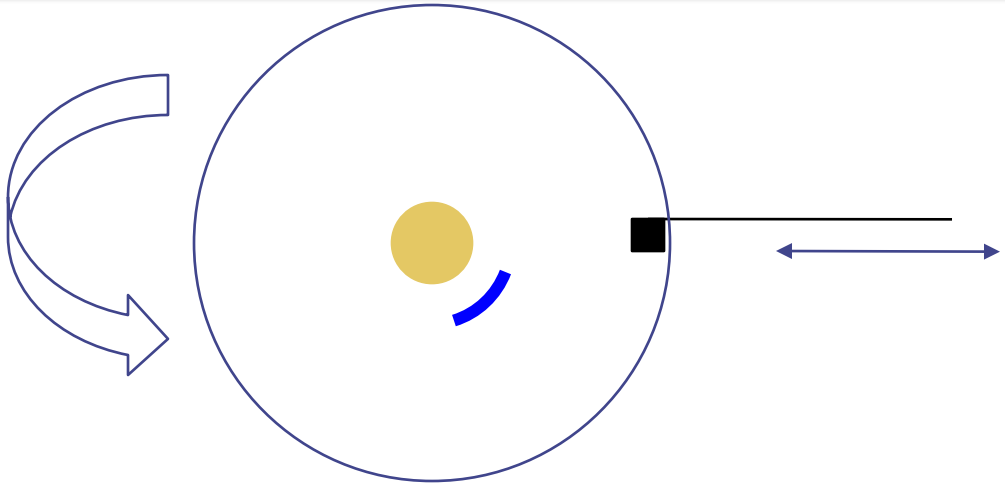
Power connector

HDDs



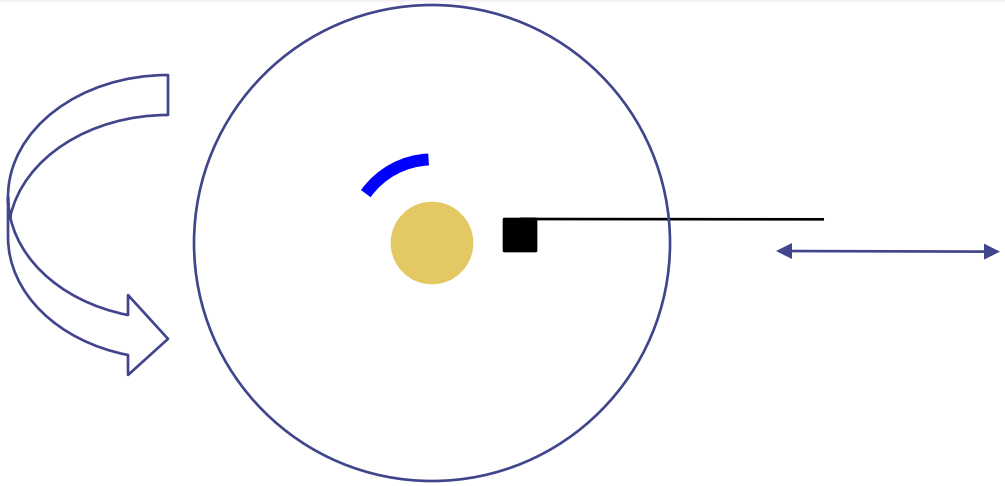
- Read/written by "head"
  - Moves across tracks **("seek")**
  - After seek completes, wait for proper sector to rotate under head
  - Reads or writes magnetic medium by sensing/changing magnetic state (this takes time as the desired data 'spins under' the head)

HDDs



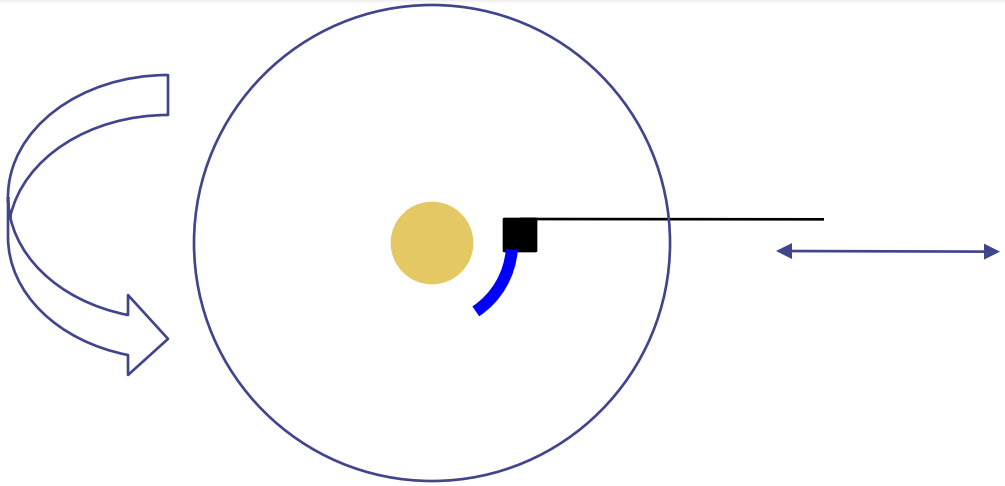
- Want to read data on blue curve
  - First step: seek—move head over right track
    - Takes time (Tseek), while disk keeps spinning

HDDs



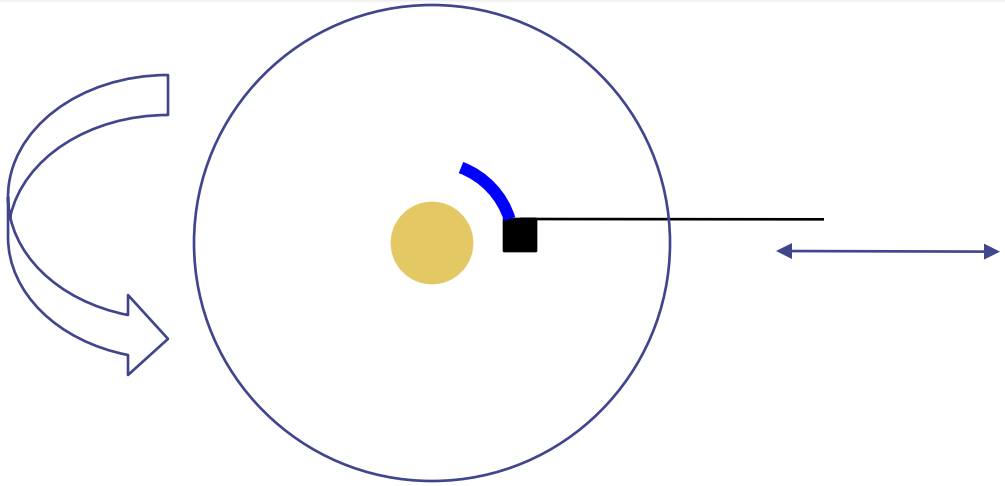
- Want to read data on blue curve
  - First step: seek—move head over right track
    - Takes time (Tseek), while disk keeps spinning
    - Now head over right track… but data needs to move under head
  - Second step: wait (Trotate)

HDDs



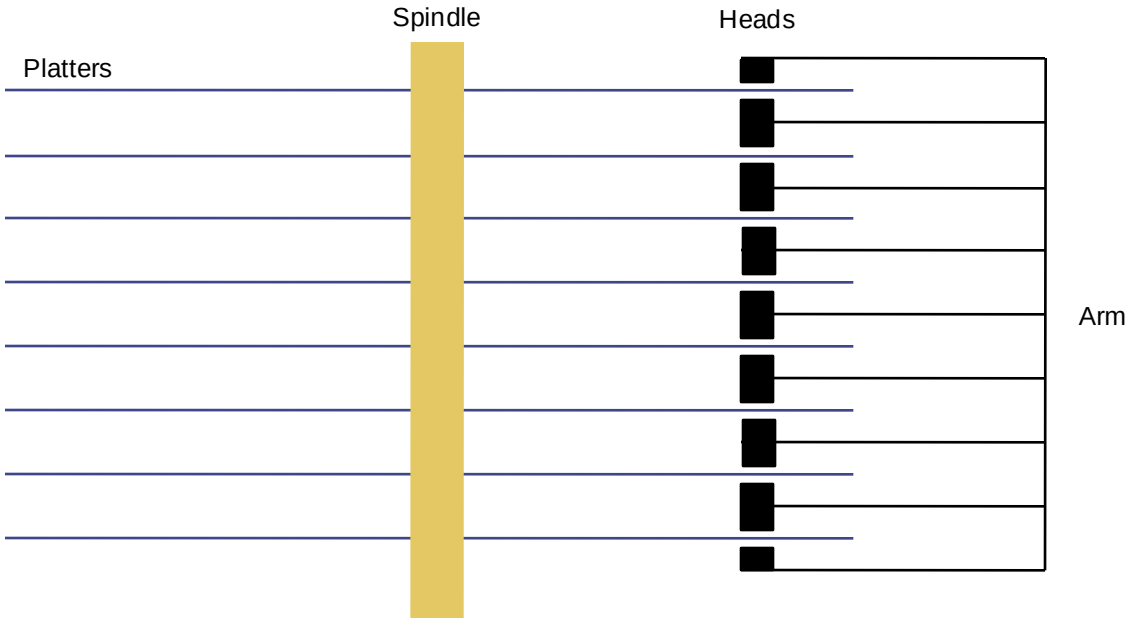
- Want to read data on blue curve
  - First step: seek—move head over right track
    - Takes time (Tseek), while disk keeps spinning
    - Now head over right track… but data needs to move under head
  - Second step: wait (Trotate)
  - Third step: as data comes under head, start reading

HDDs



- Want to read data on blue curve (imagine circular arc)
  - First step: seek—move head over right track
    - Takes time (Tseek), while disk keeps spinning
    - Now head over right track… but data needs to move under head
  - Second step: wait (Trotate)
  - Third step: as data comes under head, start reading
    - Takes time for data to pass under read head (Tread)

HDDs: from the side



- Multiple platters, each with a head above and below
  - Two-sided surface
  - Heads all stay together ("cylinder")
  - Heads not actually touching platters: just **very** close

A Few Things About HDD Performance

- Tseek:
  - Depends on how fast heads can move
  - And how far they have to go
    - OS may try to schedule IO requests to minimize Tseek
- Trotate:
  - Depends largely on how fast disk spins (RPM)
  - Also, how far around the data must spin, but usually assume avg
    - OS cannot keep track of position, nor schedule for better
- Tread:
  - Depends on RPM + how much data to read

HDD Performance

- Suppose on average
  - Tseek = 10 ms
  - Trotate = 3.0 ms
  - Tread = 5 µsec / 512B sector
- What is the average time to read one 512-byte sector?
  - 10 ms + 3 ms + 0.005 ms = 13.005 ms
- What would be the bitrate (throughput[b/s]) for reading 1 sector?
  - 512 B / 13.005 ms → ~40KB/sec
- What would be the bitrate for reading 1MB of (contiguous) data?
  - 1MB / 512B = 2K OR 2048 sectors
  - 10 + 3 + 0.005 * 2048 = 23.24 ms → ~43MB/sec

HDD Performance

- … ~40KB/sec
- … ~43MB/sec

- … Larger contiguous improve performance (throughput)
  - **Amortize** Tseek + Trotate (key to good disk performance)

HDD Performance

- Hard disks have caches (spatial locality)
- OS will also buffer disk in memory
  - Ask to read 16 bytes from a file?
  - OS reads multiple KB, buffers in memory
- "Defragmenting":
  - Improve locality by putting blocks for same files near each other

What about SSDs?

- Solid state drive (SSD)
  - Storage drives with no mechanical component
  - Internal storage similar to a logic-gate-based memory, but persistent (non-volatile)!
  - SSD Controller implements Flash Translation Layer (FTL)
    - Emulates a hard disk
    - Exposes logical blocks to the upper level components
    - Performs additional functionality

Source: wikipedia

SSDs summarized

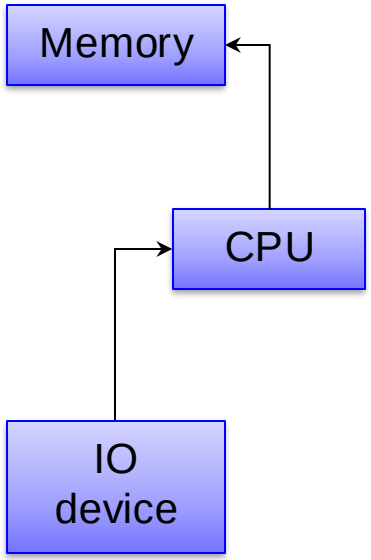
- Tradeoffs of SSDs:
  - + No expensive seek, uniform access latency
  - – Due to physics, can WRITE small data blocks (~4KB) but can only ERASE big data blocks (~1MB, also slow).
    - Complicated controller logic does tons of hidden tricks to make it seem like a regular hard drive while hiding all the weirdness
  - – More expensive per GB capacity
  - + Less expensive per unit of IO performance (mainly, throughput)

- There's more to it, but that will do, for now...

Transferring Data From Disk to Memory

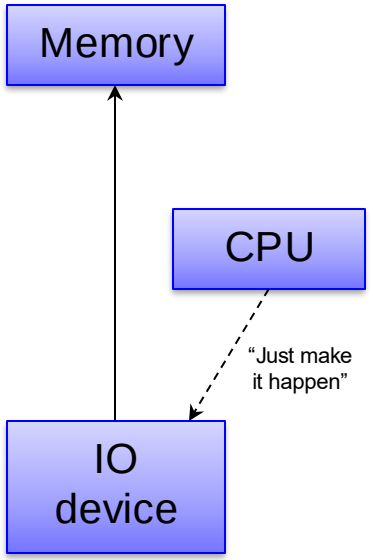
- OS asks disk to read data
  - Disk read takes a long time (15 ms → millions of cycles)
  - Does OS poll disk for 15M cycles looking for data?
  - No — disk interrupts OS when data is ready

- How is data copied? Option 1:
  - Disk has data, needs it transferred to memory
  - OS does "memcpy"-like routine:
    - Read disk memory mapped IO
    - Write appropriate location in main memory
    - Repeat
      - For many KB to a few MB

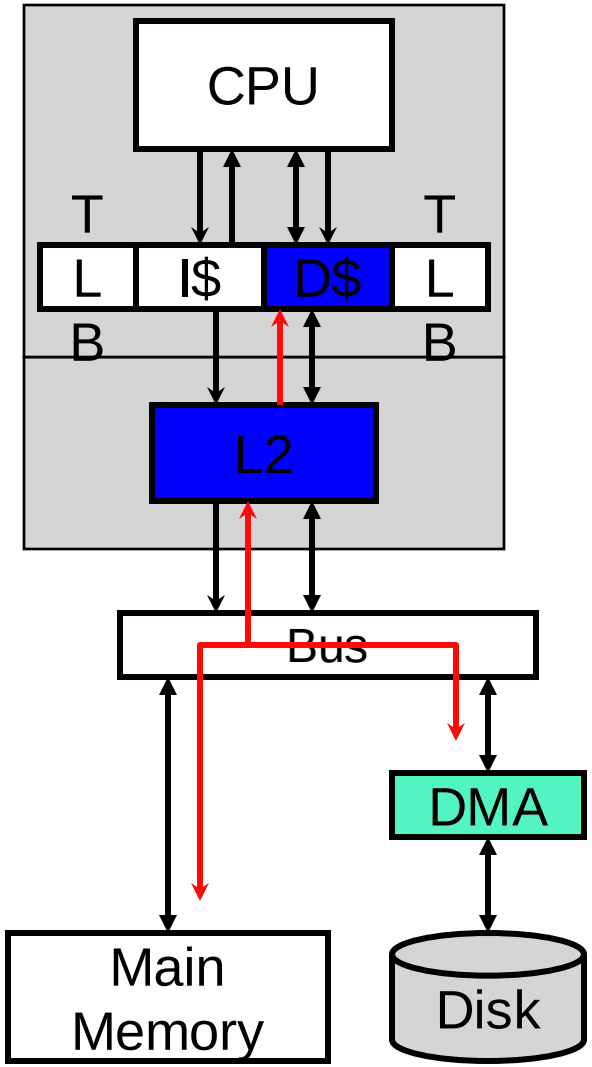


DMA: Direct Memory Access

- Option 2: DMA
  - When OS requests disk read, set up DMA
    - "DMA, please read this data from the disk, and put it in memory for me"
  - DMA controller handles "memcpy"
  - Frees up CPU to do useful things!
  - → **Indirection!**



Cache Coherence



- Caches introduce a **coherence problem** for DMA
- Simple solution:
  Disallow caching of I/O data
- Fancy solution:
  D$ and L2 **"snoop"** bus traffic
  - Observe transactions
  - Check if written addresses are resident
  - **Self-invalidate** those blocks

What About Disk Reliability?

- Hard disks fail relatively easily
  - Spinning piece of metal
  - With head hovering <1mm from platter
- Hard drive failures: major pain..

Reliability

- **Solution to a functionality problem?**
  - **→ Add a level of indirection**
- **Solution to a performance problem?**
  - **→ Add a cache/buffer**
- **Solution to a reliability problem?**
  - **→ Add redundancy!**
    - For error checking and correction
      - Simplest correction: keep 2 copies of the data
      - More efficient solution (for disk data reliability): use **RAID** systems

RAID

- Redundant Array of Inexpensive Disks (RAID)
  - Keep 2 hard-drives with identical copies of the data
  - One fails? Replace it, copy the other drive to it, resume
    - Can work from other drive while waiting for replacement
  - Performance?
    - Writes to both drives in parallel (no cost)
    - Reads from **either** drive
      - Improve performance: twice the bandwidth
  - Downside?
    - Cost: need to buy 2x as many disks for 1x the space
  - But still pretty popular
  - Very easy to set up

RAID

- Mirroring data (prev slide) is called: "RAID 1"

- Tons of other RAID configurations:
  - Use parity schemes that reduce overhead for num disks > 2
    - Still give reliability and good performance
  - Good to know they exist

Other Devices?

- Wide variety of IO devices
  - Most basically work the same way from high level
  - Reality: each device has its own protocol
    - Requires device **driver**: Software module that handles protocol details of specific device
      - Which memory locations to read/write, etc.
    - Having these drivers is an example of?

Abstraction!