CSCI 210: Computer Architecture
Lecture 28: Pipelining

Stephen Checkoway

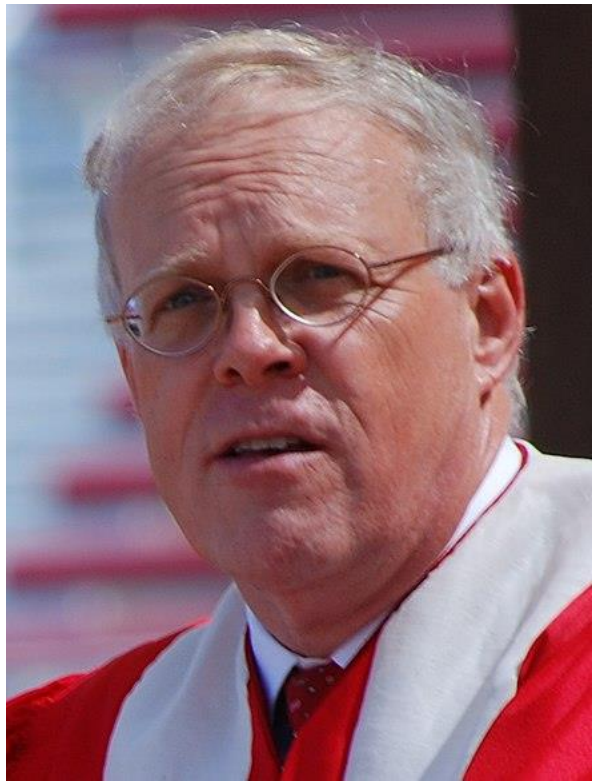
Slides from Cynthia Taylor

CS History: The IBM 7030 Stretch



- First pipelined computer
  - Three stages: Fetch-Decode-Execute
- Fastest computer in the world from 1961 until 1964
- Much slower than IBM anticipated
  - Dropped price from $13.5 million to $7.5 million
  - PC World named it one of the biggest IT project management failures in history
- Many ideas from the Stretch, including pipelining, were used in the very successful IBM System/360

CS History: MIPS



- “Microprocessor without Interlocked Pipelined Stages”
- Started as a class project in graduate class taught by John Hennessy at Stanford in 1981
- In 1984 Hennessy formed MIPS Computer Systems to sell it commercially
- Emphasized short clock cycle and use of pipelining

Instruction Critical Paths

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type						
ld						
sd						
beq						
jal						

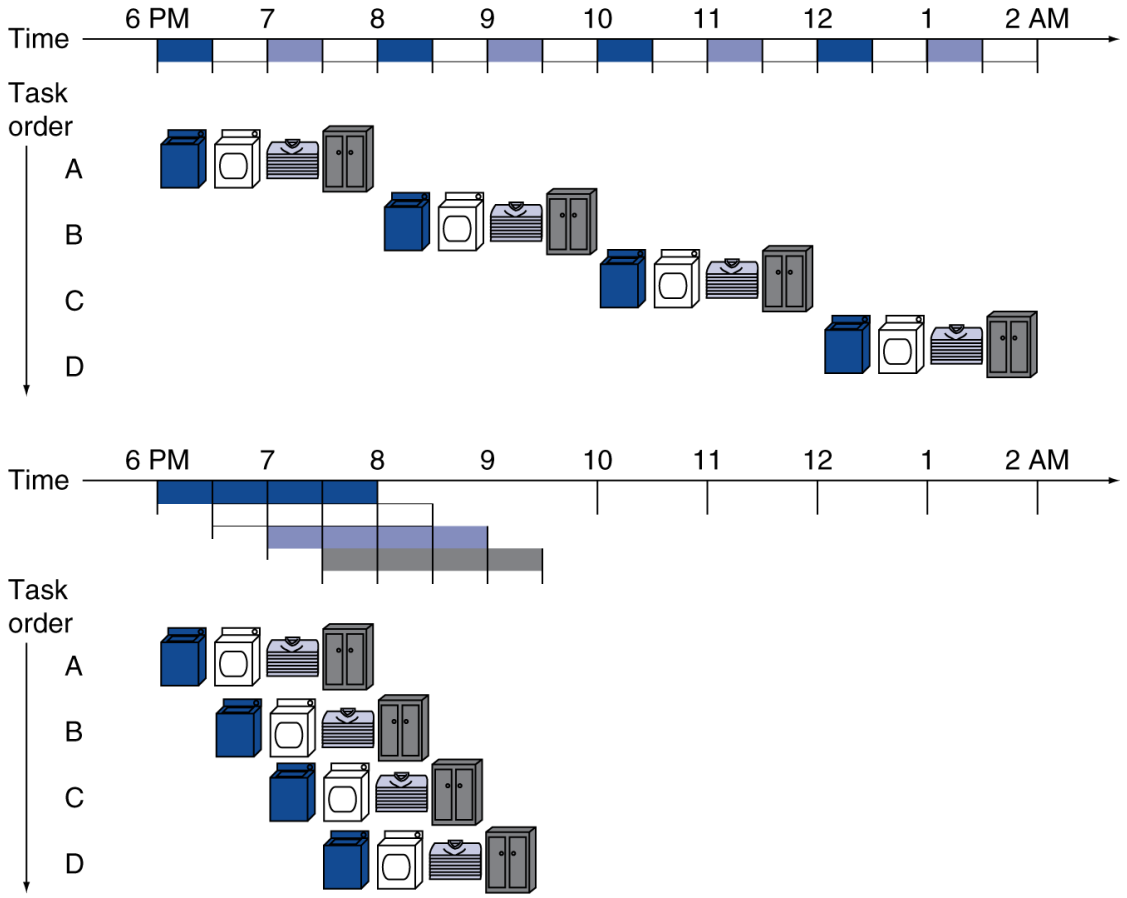
Times are just examples and not representative of real-world latencies

Performance Issues

- Longest delay determines clock period
  - Critical path: ld instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary clock period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

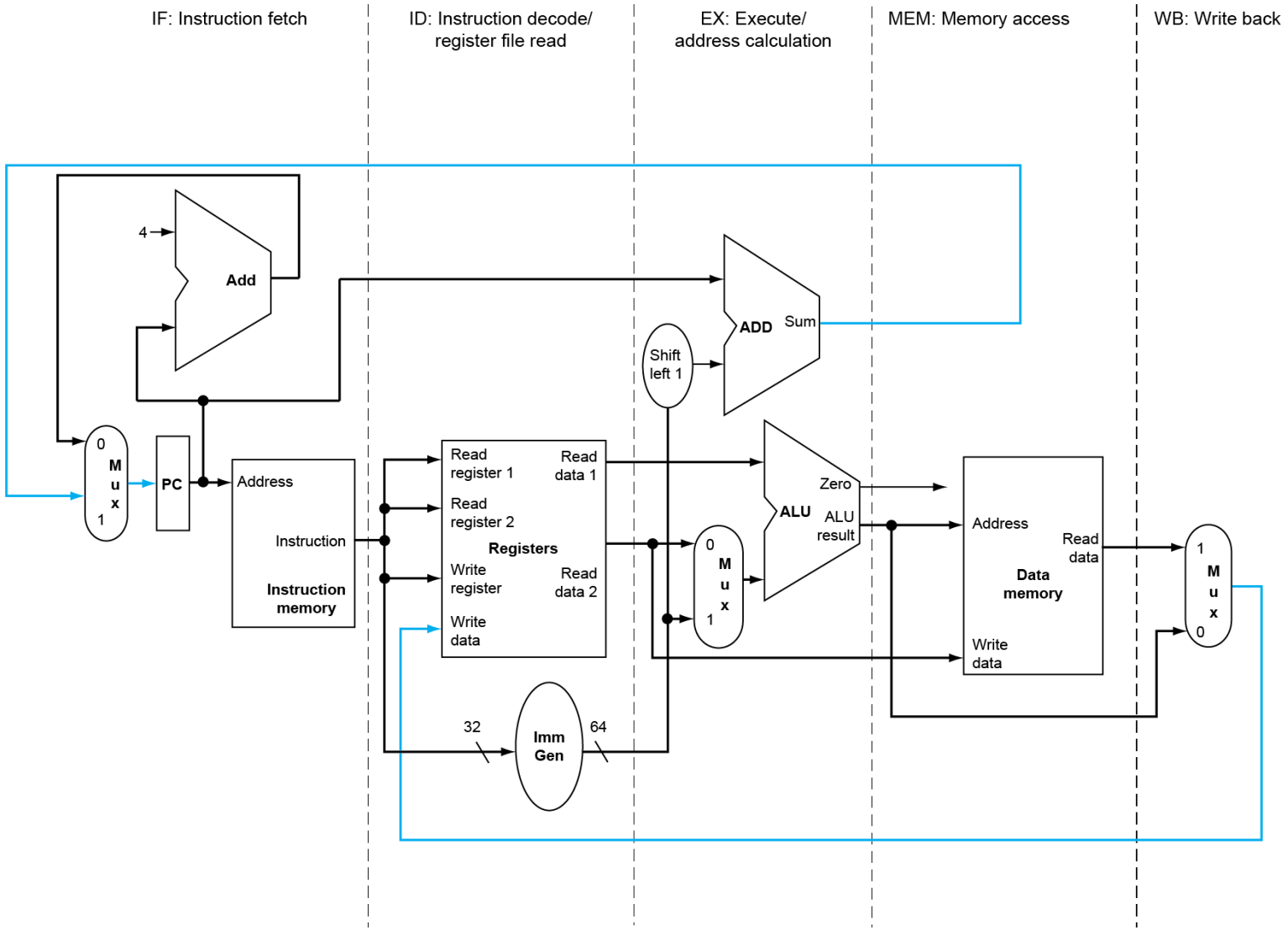


Classic 5-stage Pipeline

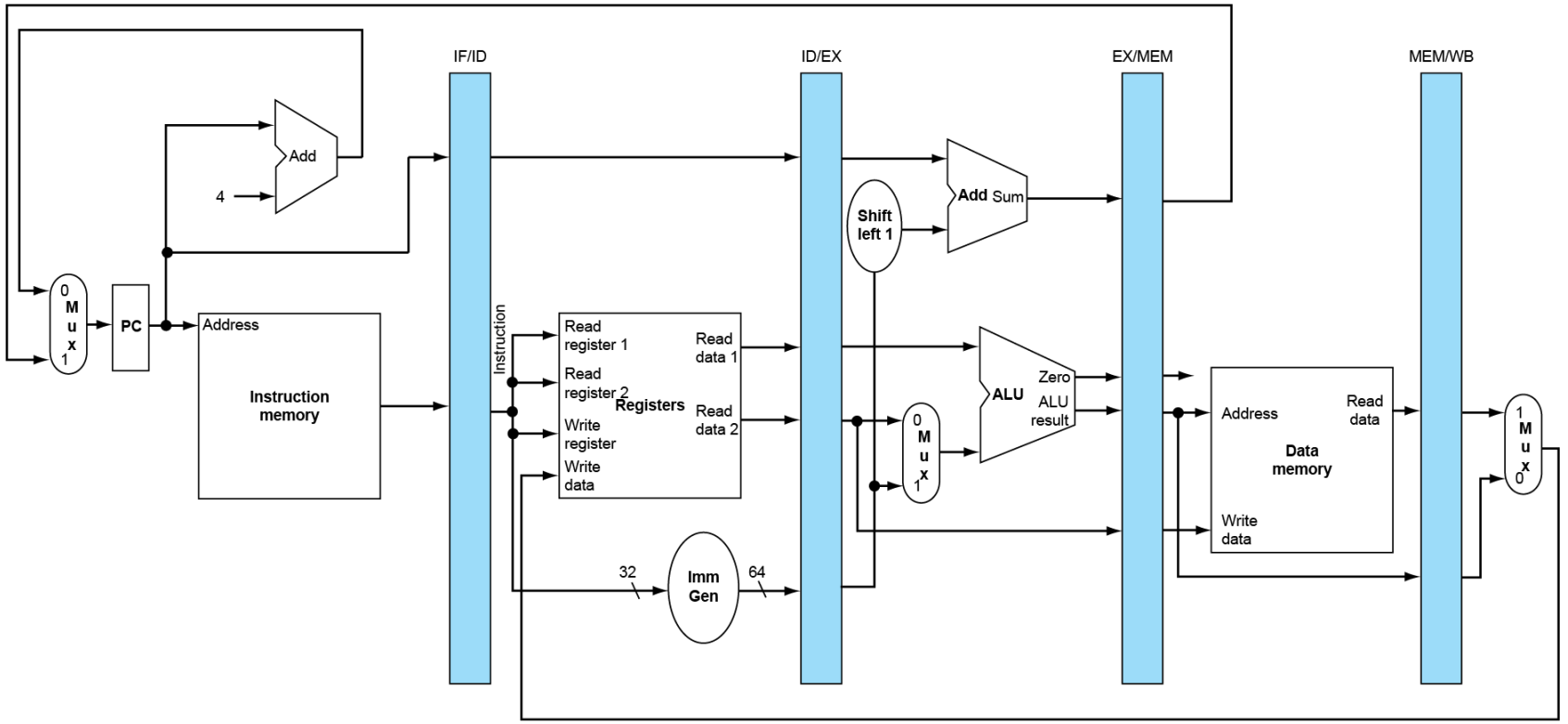
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

- Move from one instruction per clock cycle, to one stage per clock cycle (with shorter cycle period!)

The 5 stages

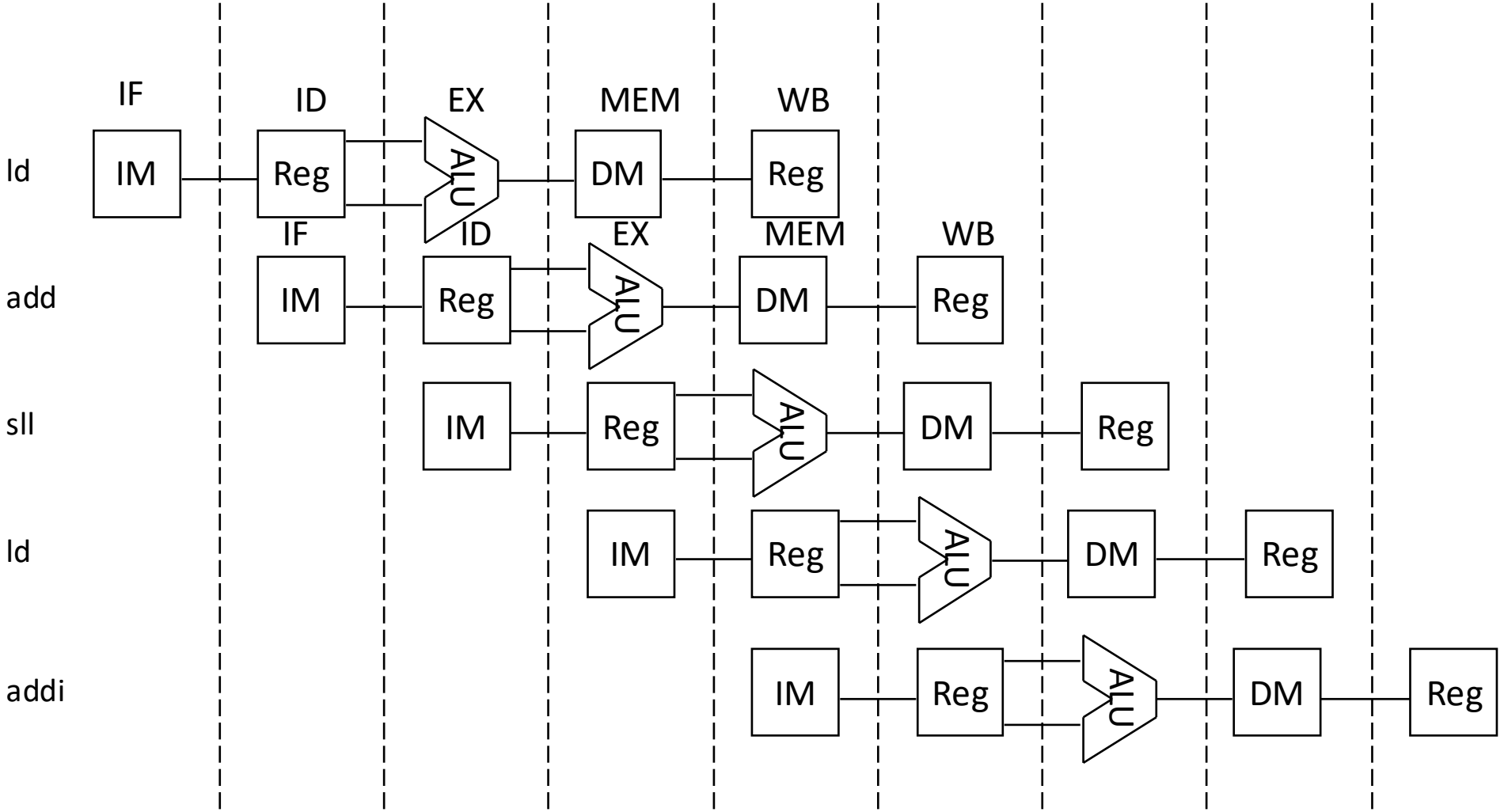

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
4
Add
ADD
Sum
Shift left 1
0
M
u
x
1
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
0
M
u
x
1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0
32
Imm Gen
64

Pipeline Registers

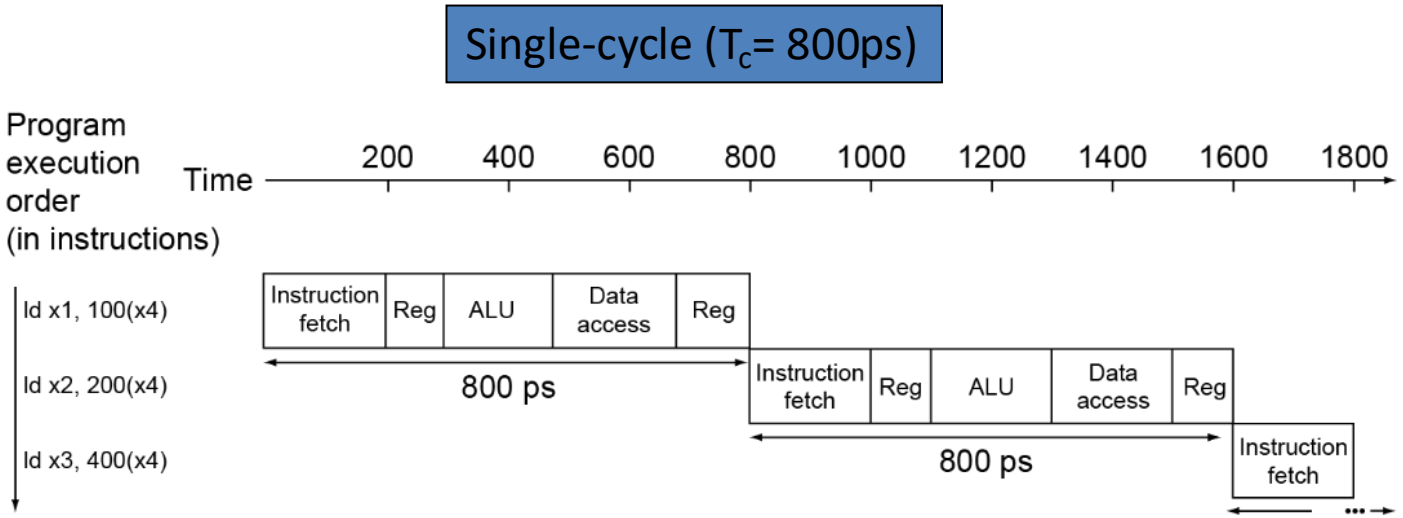


IF: Instruction Fetch ID: Instruction Decode EX: Execute MEM: Memory WB: Writeback

Execution in a Pipelined Datapath



Pipeline Performance



If we pipeline by running different stages at the same time (i.e., running instruction fetch for the next instruction during the Reg stage of the first instruction), running two instructions will take us

A. 900 ps

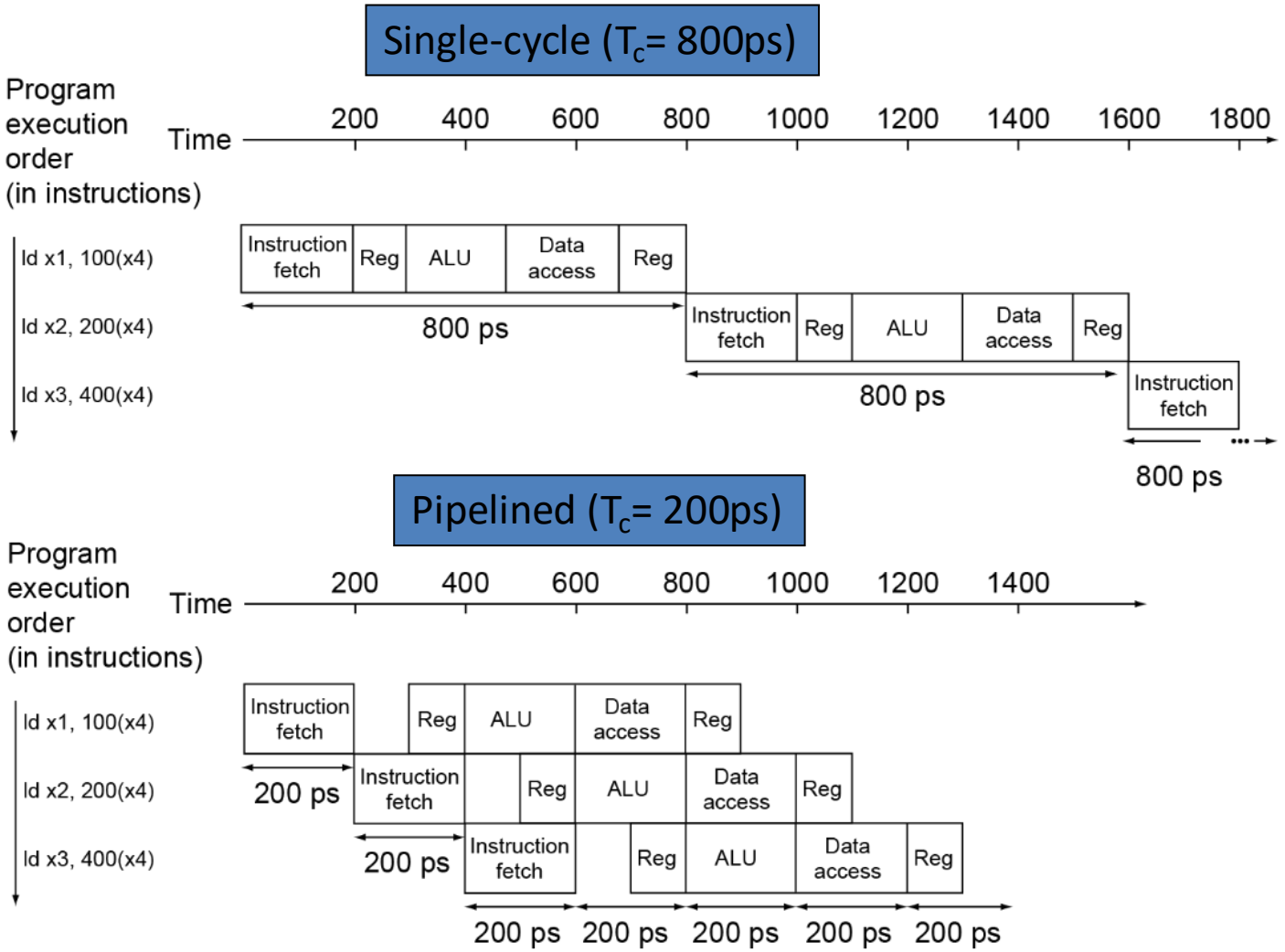
B. 1000 ps

C. 1200 ps

D. 1600 ps

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- HINT: How long will the clock cycle be in the pipelined version?

Pipeline Performance



Fill in the blanks: In our pipelined data path, latency (ns per instruction) _____, but throughput (instructions per second) _____

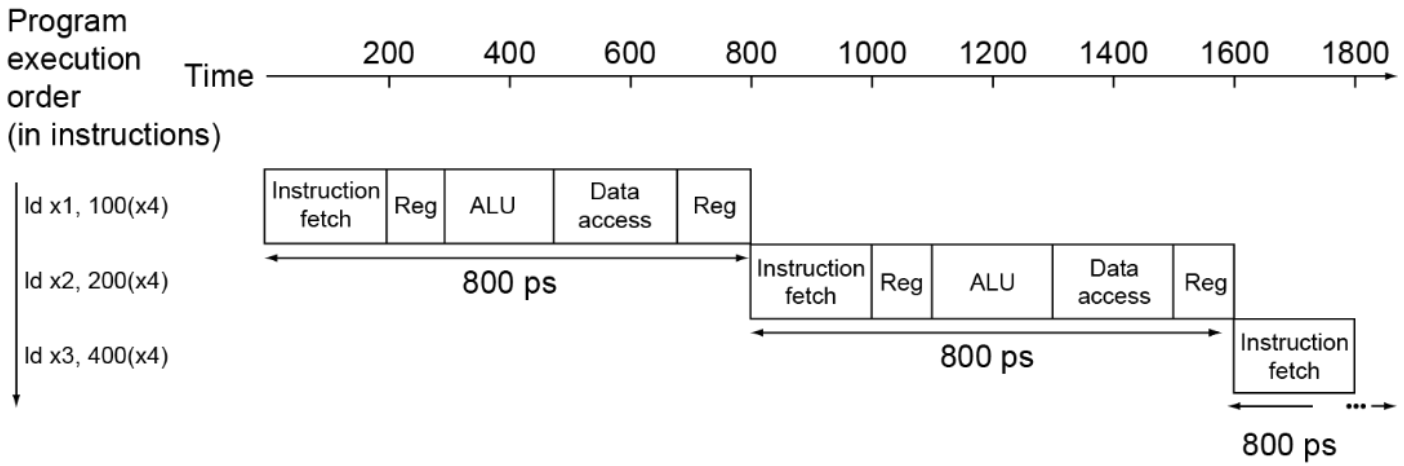
A. Improves; gets worse

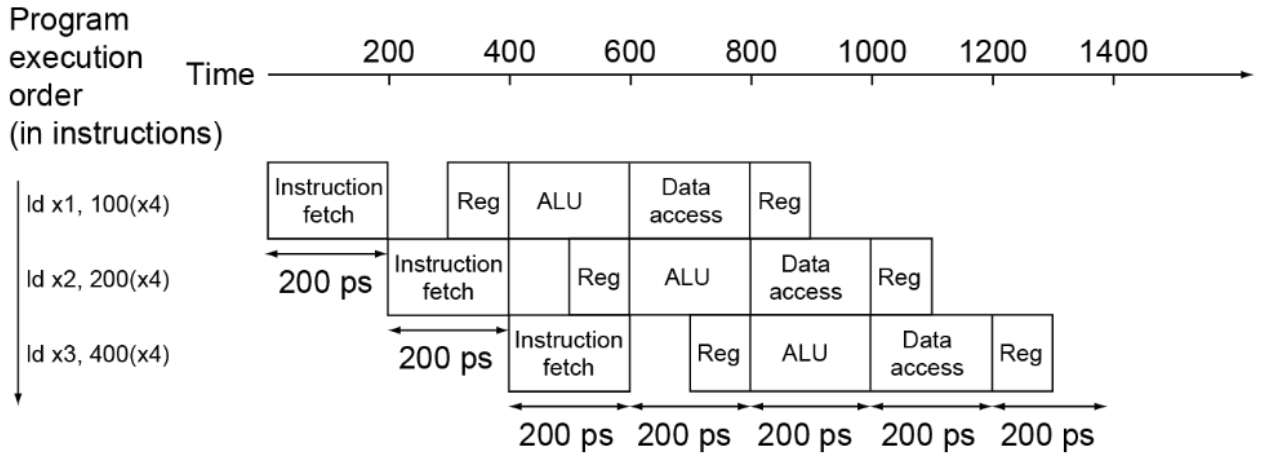
B. Improves; stays the same

C. Gets worse; improves

D. Stays the same; improves

E. None of the above





Maximum Pipeline Speedup

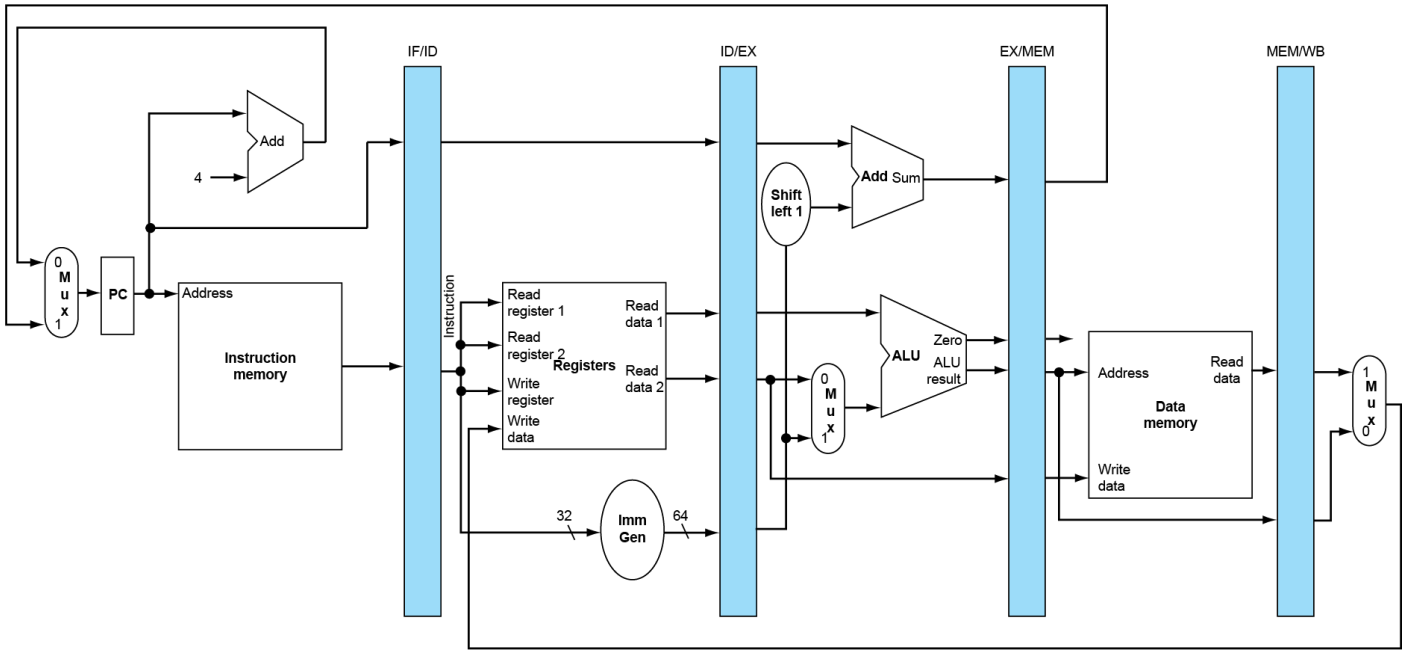
- If all stages are balanced
  - i.e., all take the same time

  $$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

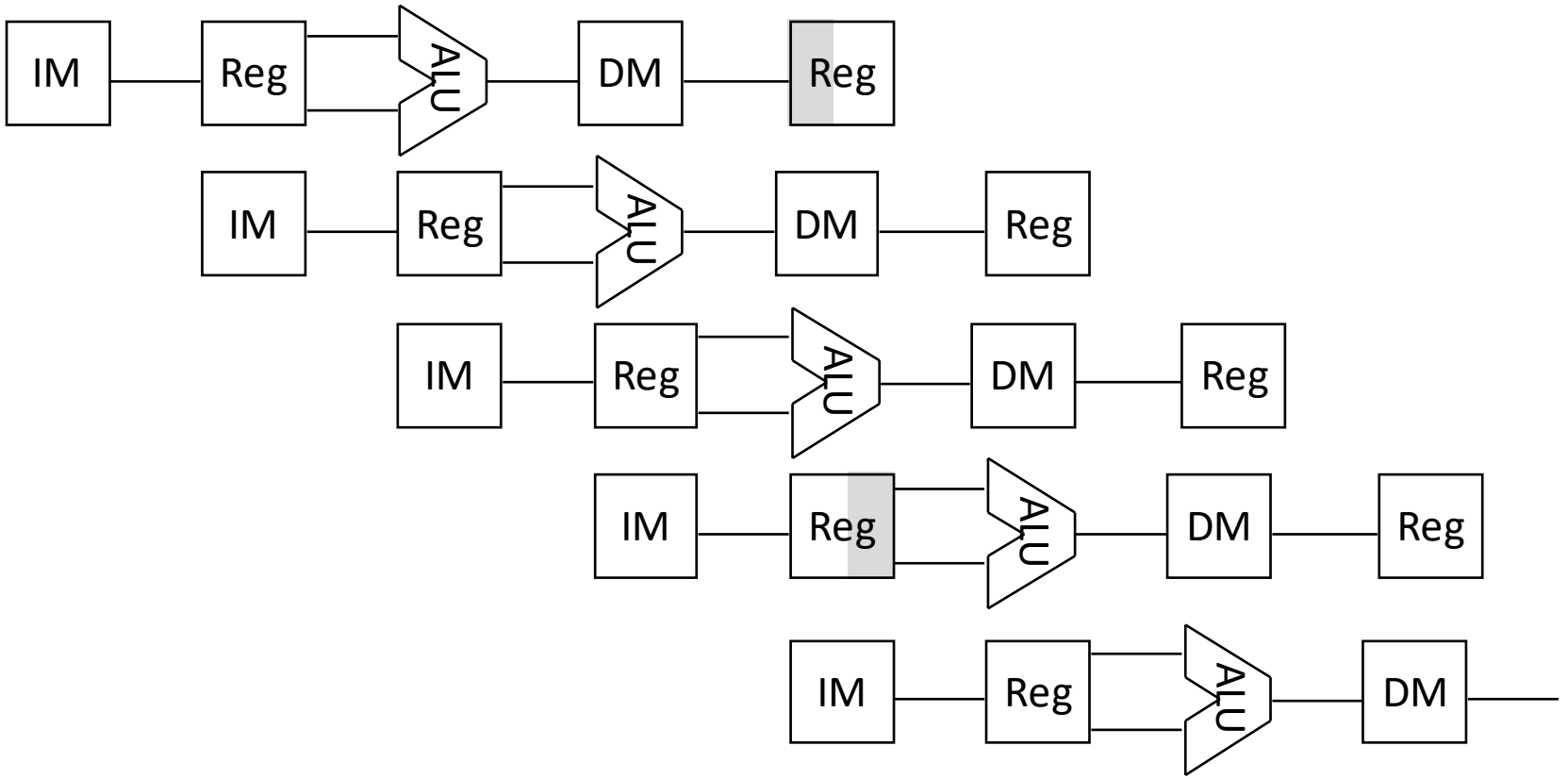
- Speedup = $\text{time}_{\text{pipelined}} / \text{time}_{\text{nonpipelined}}$ = number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch in one cycle
    - cf. x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one cycle
  - Load/store addressing
    - Can calculate address in 3rd stage (EX), access memory in 4th stage (MEM)



```
sub x2, x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sd  x15, 32(x2)
```



What just happened here which is problematic (BEST ANSWER)?

A. The register file is trying to read and write the same register
B. The ALU and data memory are both active in the same cycle
C. A value is used before it is produced
D. Both A and B
E. Both A and C

Hazards

Situations that (if not handled) prevent starting the next instruction in the next cycle

- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

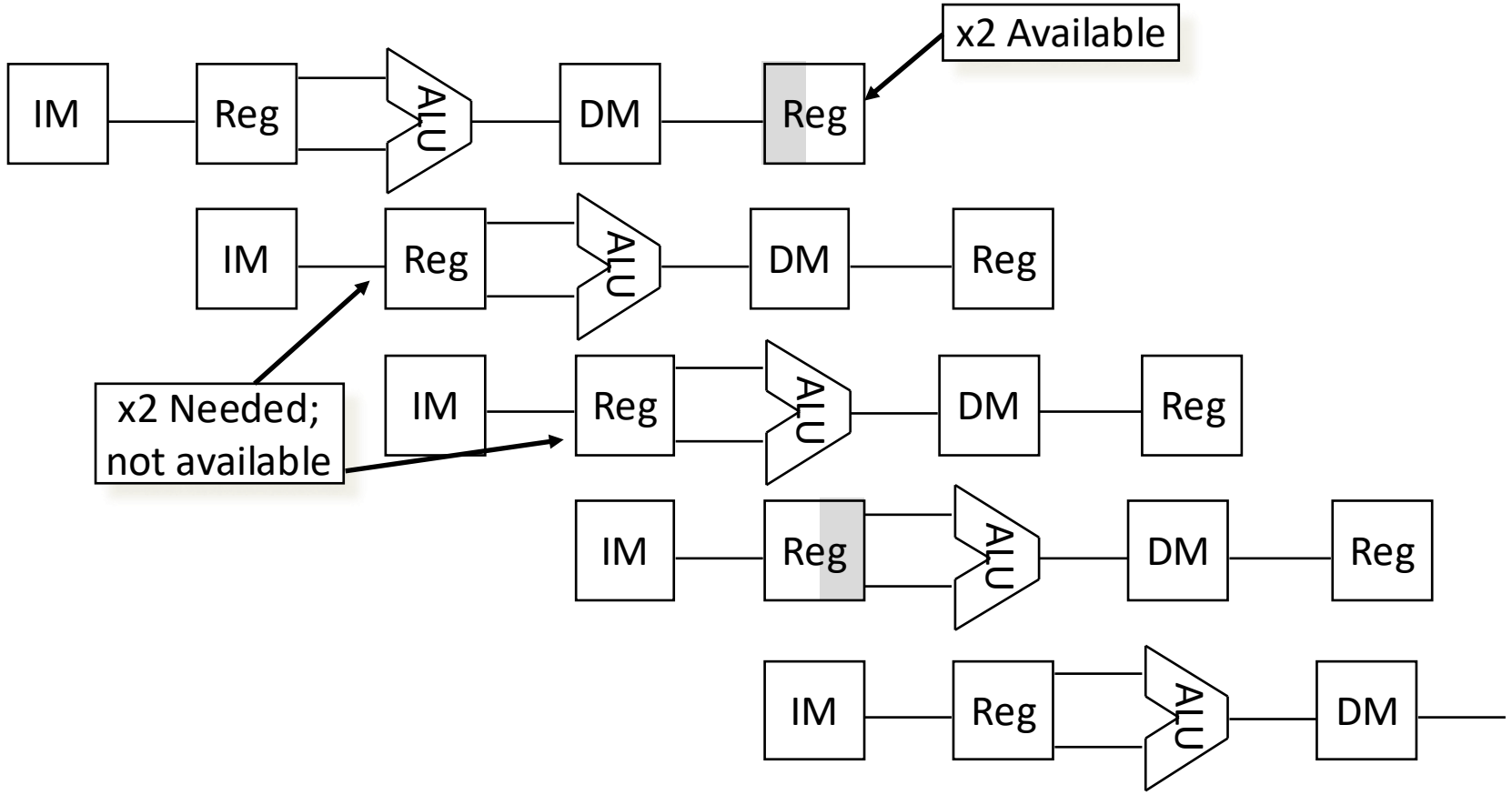
Structure Hazards

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined data paths require separate instruction/data memories (or “caches” which we’ll talk about later in great detail)

Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

```
sub x2, x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sd  x15, 32(x2)
```



We could solve data hazards by

A. Reordering instructions

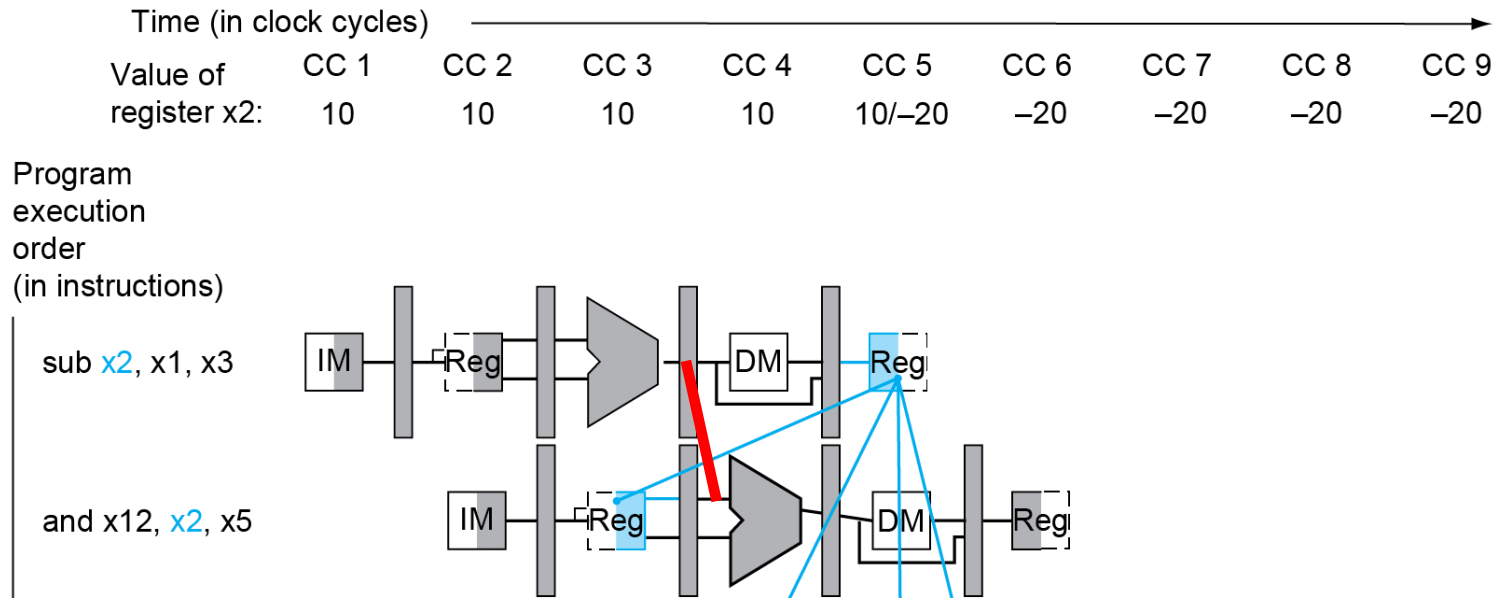
B. Not running the second instruction until the data is ready

C. Sending the calculated value straight from the ALU to the next instruction, skipping the registers

D. More than one of the above

Forwarding (a.k.a. Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the data path (and extra control signals)



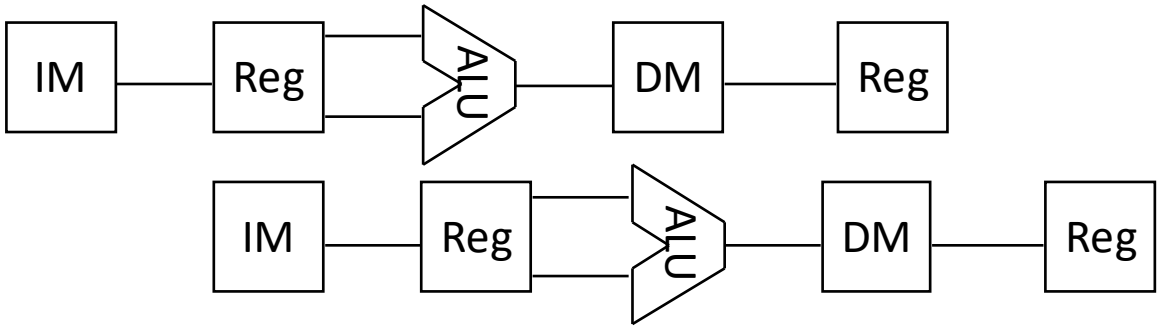
Would forwarding work if our instructions were

```
ld  x2, 20(x1)
and x4, x2, x5
```

A. Yes

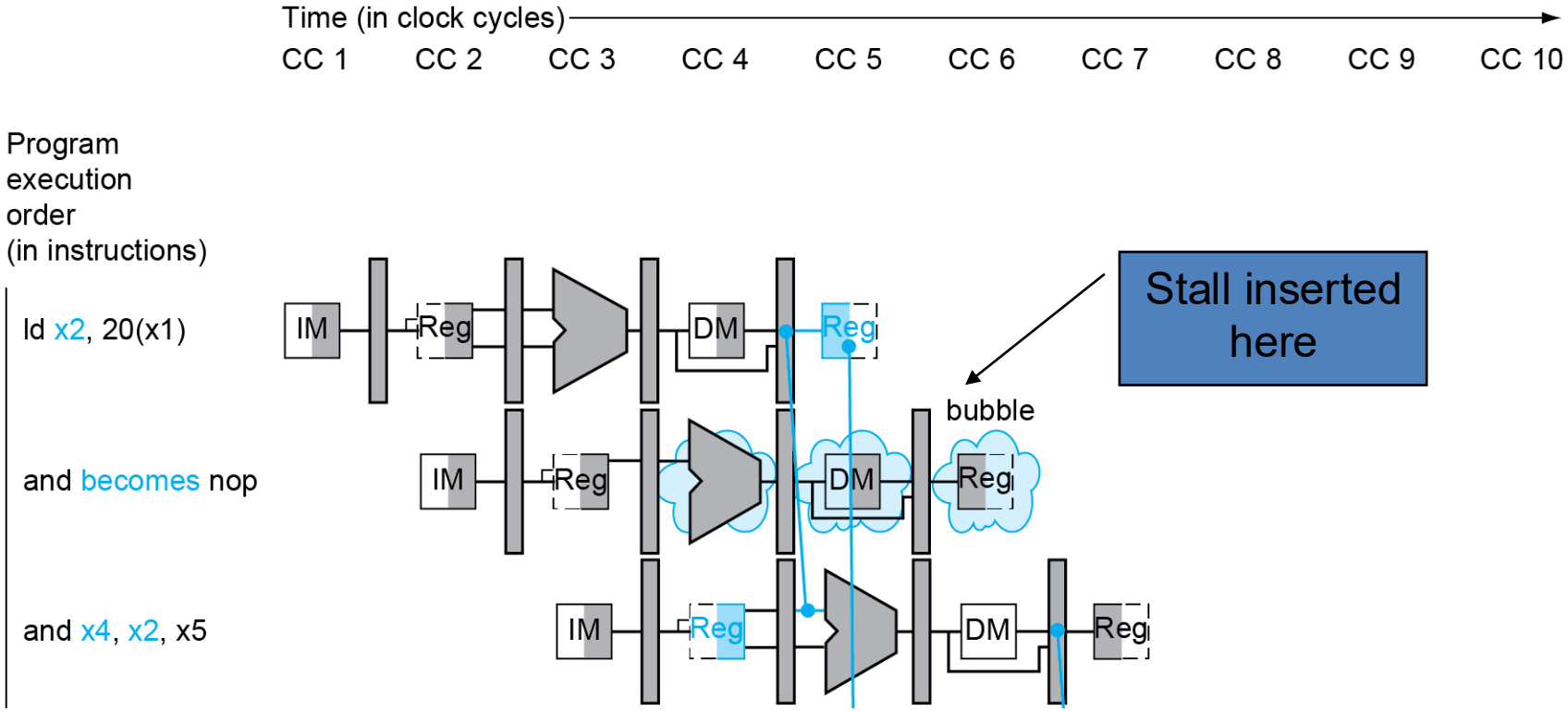
B. No

C. Depends on the value loaded



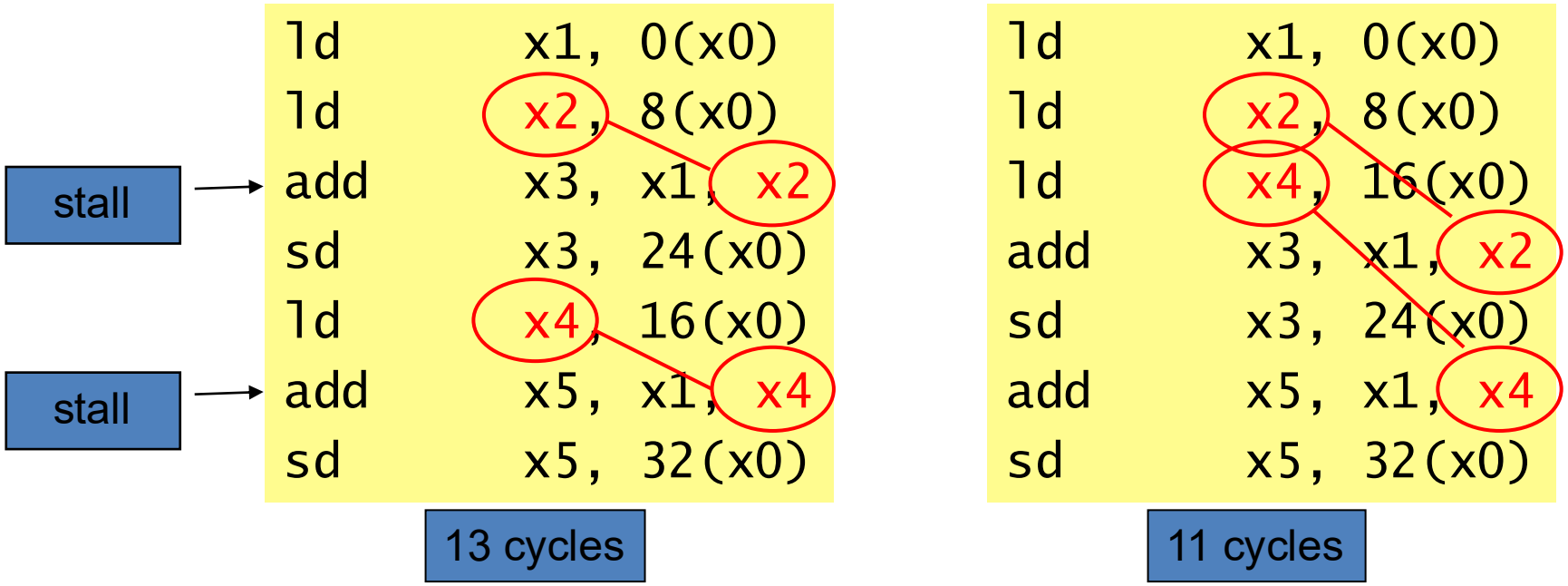
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- Assembly code for `A = B + E;`
  `C = B + F;`



Questions on Data Hazards?

- Basic problem: Need a value before it's written to a register
- Solutions: Forwarding, stalling, reordering instructions (code scheduling)

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

Control Hazards in RISC-V

- In the RISC-V 5 stage pipeline
  - Branch outcome and branch target aren't determined until the EX stage
  - By this point, the following two instructions are in the pipeline (the instruction after the branch is in the ID stage, the instruction after that is in the IF stage)
- Partial solution
  - Compare registers and compute target earlier in the pipeline
  - Add circuitry to do it in ID stage

In the single cycle implementation, we used the ALU to subtract the two register values and used its zero output to decide if they are equal. If we move the comparison to the ID stage, do we need a full ALU to compare the register values for equality?

A. Yes, we need a full ALU
B. No, we can use a subtracter
C. No, we can use something simpler than a subtracter [what?]
D. It depends

Branches

- MIPS only has `beq` and `bne` branch instructions
  - MIPS only needs to XOR the two register values to determine equality
- Some processors (including RISC-V) can branch if one register is less than (or greater than or equal to) another register
  - Since the output only needs to be 0 or 1, a “digital comparator” can be used in place of a subtractor
  - A digital comparator takes two n-bit numbers as input and outputs 1 if the first is less than the second

Aside about conditional branching

- MIPS and RISC-V branch based on the comparison of two registers
- x86 has a set of 1-bit flags that are updated after many (but not all) arithmetic and logical operations; `cmp` and `test` *only* update flags
  - flags include zero flag, sign flag, and carry flag
  - Conditional jump instructions will branch based on the value of the flags (e.g., `jnz label` branches to label if the zero flag is not 0 (jump if not zero)
- PowerPC branches based on the value in "condition registers"
  - Similar to the x86 flags, but multiple sets of them
  - Multiple forms of each arithmetic instructions, some of which update condition register 0 (e.g., `add. r3, r4, r5` updates `cr0` with the result of the addition; `add r3, r4, r5` does not; the period is key)

More about conditional computation

- ARM uses status codes which are essentially flags in the `cpsr` register
  - Some instructions have an s–suffixed form that update these flags (e.g., `adds r2, r3, r4`)
  - Like x86, `cmp` updates the status codes
  - 32-bit ARM also has "predicated" instructions that only do anything if the specified status codes are set
- Some architectures also have weird other conditional branches like x86's `jcxz label` which branches if register `ecx` is 0 or PPC's `bc` which can be used to decrement a special `ctr` register and then branch if it is 0 (and even more weird things!)
- Some architectures (like x86) have conditional move instructions like `cmovc rax, rbx` (move `rbx` into `rax` if the carry flag is set); this can avoid branches similar to ARM's predicated instructions

Move the branch instruction implementation as early as we possibly can

- We can’t compute the branch outcome earlier than the ID stage because we need to read register values

- We still have a control hazard because we don’t know what the next instruction to execute is!

To solve control hazards, we could

A. Rearrange instructions

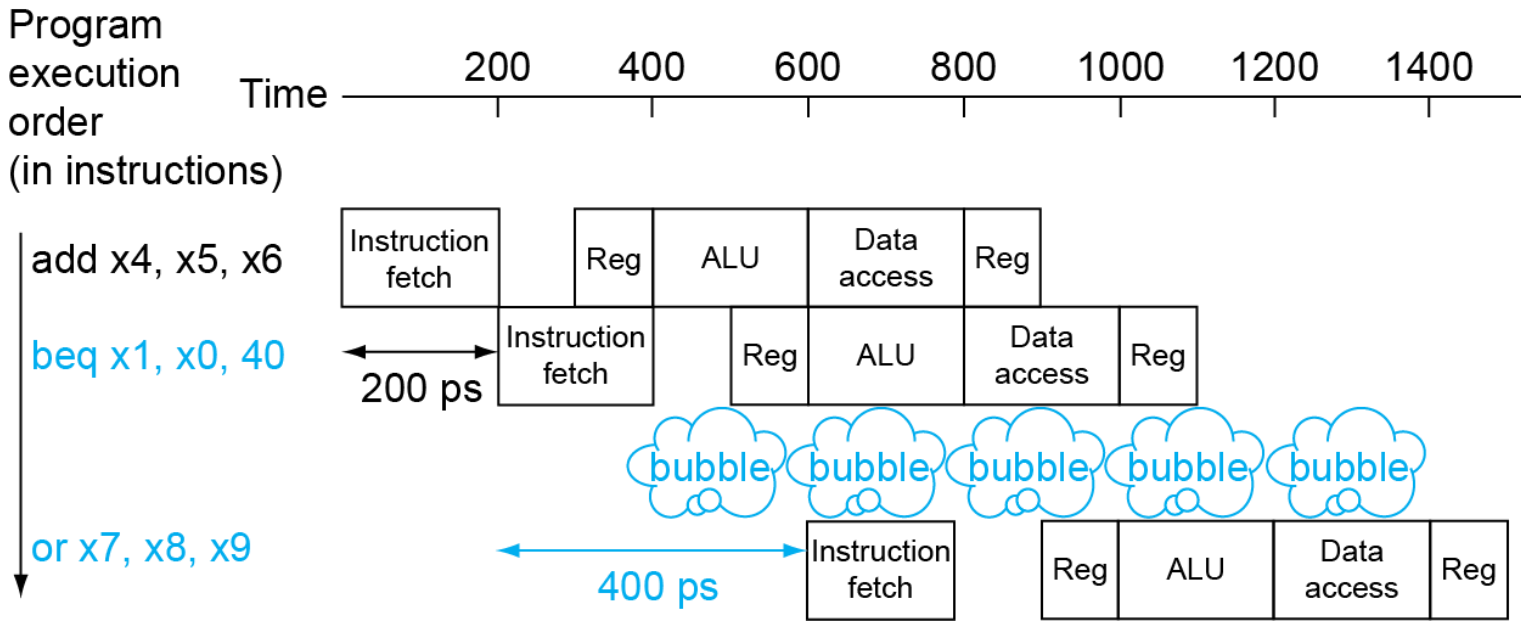
B. Guess what the branch will do and if we guess wrong, discard the incorrect instruction and fetch the correct one

C. Stall until we know the branch outcome

D. More than one of the above [which?]

Stall on Branch

Wait until branch outcome determined before fetching next instruction



Problem

- Branches are about 15% of instructions
- If all instructions except branches have a CPI of 1, and branches have a CPI of 2, we have an average CPI of 1.15
- Even worse if we can’t resolve the branch until later

Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - If prediction is wrong, then "flush" the pipeline
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction (more later)

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Questions on Control Hazards?

Reading

- Next lecture: Pipelined Data path
  - Section 4.7