



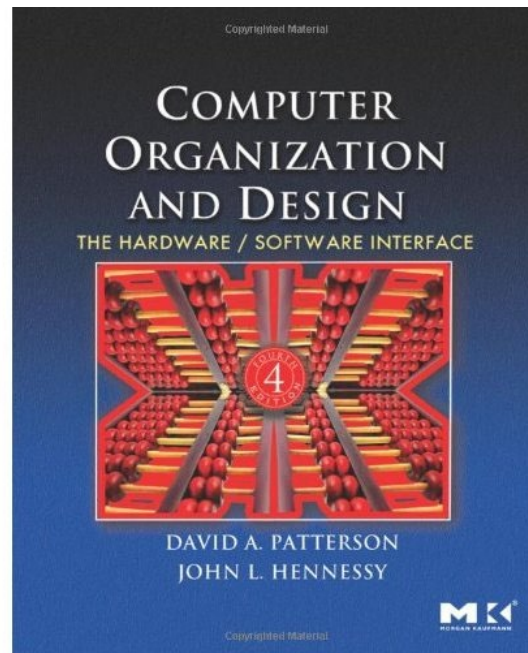
Informatik der Systeme – Chapter 6: The Processor

Prof. Dr. Michael Menth

<https://kn.inf.uni-tuebingen.de>

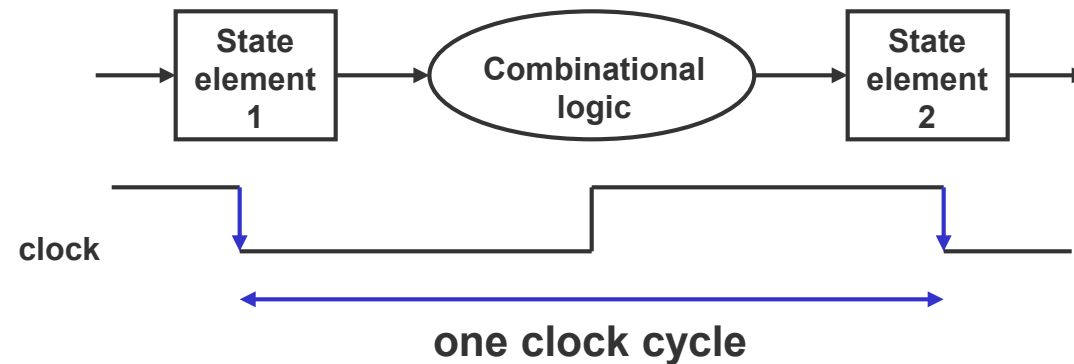


- ▶ This set of slides is an adaptation of Prof. Mary Jane Irwin's lecture notes, <http://www.cse.psu.edu/research/mdl/mji/>
- ▶ Adapted from Patterson & Hennessy: "Computer Organization and Design", 4th Edition, © 2008, MK
- ▶ German translation: Patterson & Hennessy: "Rechnerorganisation und Rechnerentwurf"





- ▶ The **clocking methodology** defines when data in a state element is valid and stable relative to the clock
 - State elements – a memory element such as a register
 - Edge-triggered – all state changes occur on a clock edge
- ▶ Typical execution
 - Read contents of state elements $\Rightarrow$ send values through combinational logic $\Rightarrow$ write results to one or more state elements

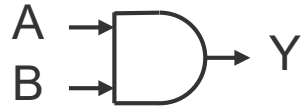


- ▶ Assumes state elements are written on every clock cycle; if not, need explicit write control signal
 - Write occurs only when both the write control is asserted and the clock edge occurs



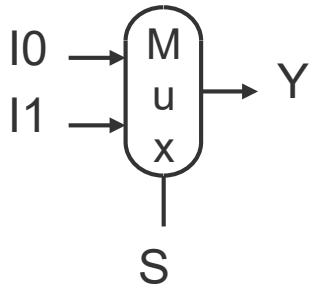
► AND-gate

- $Y = A \& B$



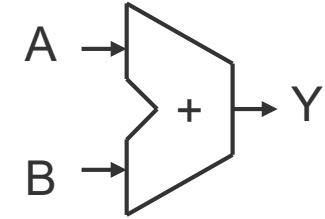
► Multiplexer

- $Y = S ? I1 : I0$



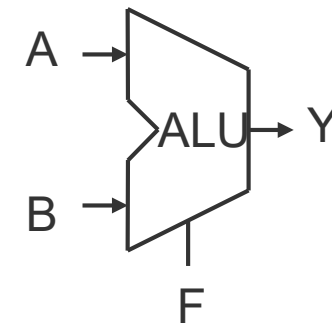
► Adder

- $Y = A + B$



► Arithmetic/Logic Unit (ALU)

- $Y = F(A, B)$





► Datapath

- Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...

► We will build a MIPS datapath incrementally

- Refining the overview design



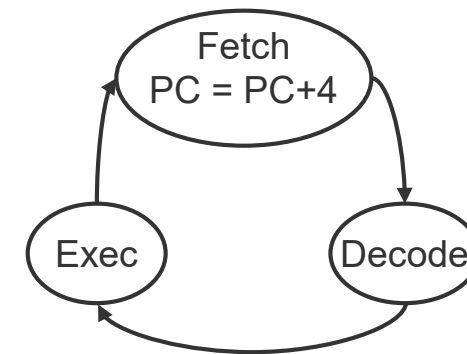
► Our implementation of the MIPS is simplified

- Memory-reference instructions: `lw`, `sw`
- Arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt`
- Control flow instructions: `beq`, `j`

► Generic implementation

- Use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
- Decode the instruction (and read registers)
- Execute the instruction

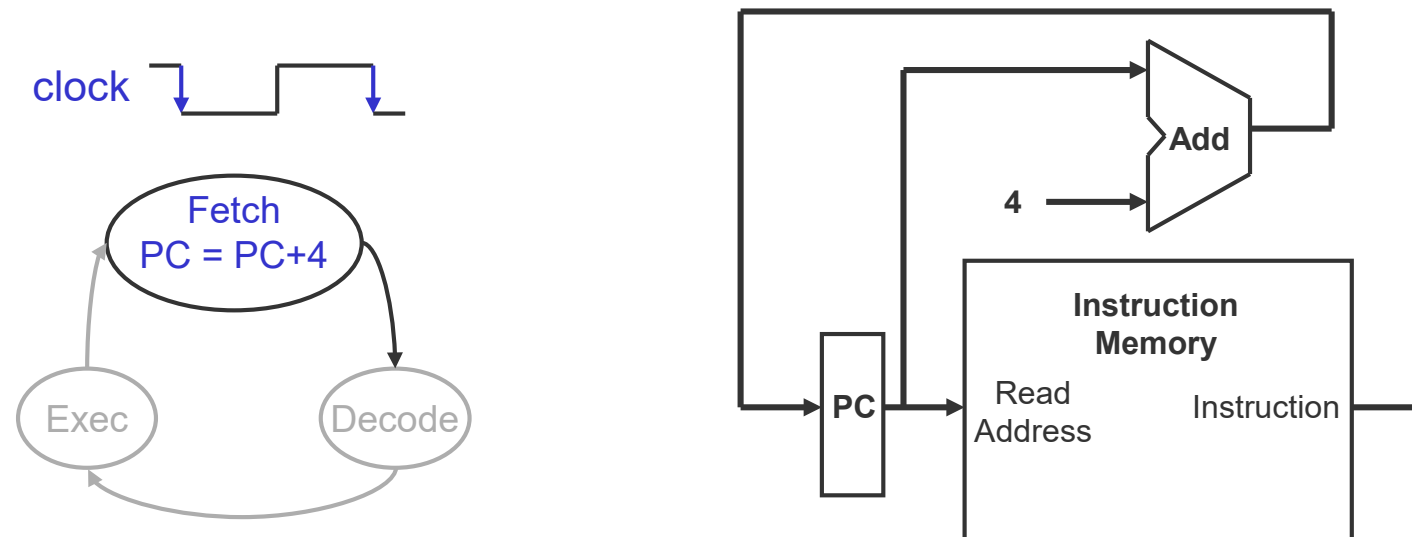
► All instructions (except `j`) use the ALU after reading the registers





► Fetching instructions involves

- Reading the instruction from the Instruction Memory
- Updating the PC value to be the address of the next (sequential) instruction

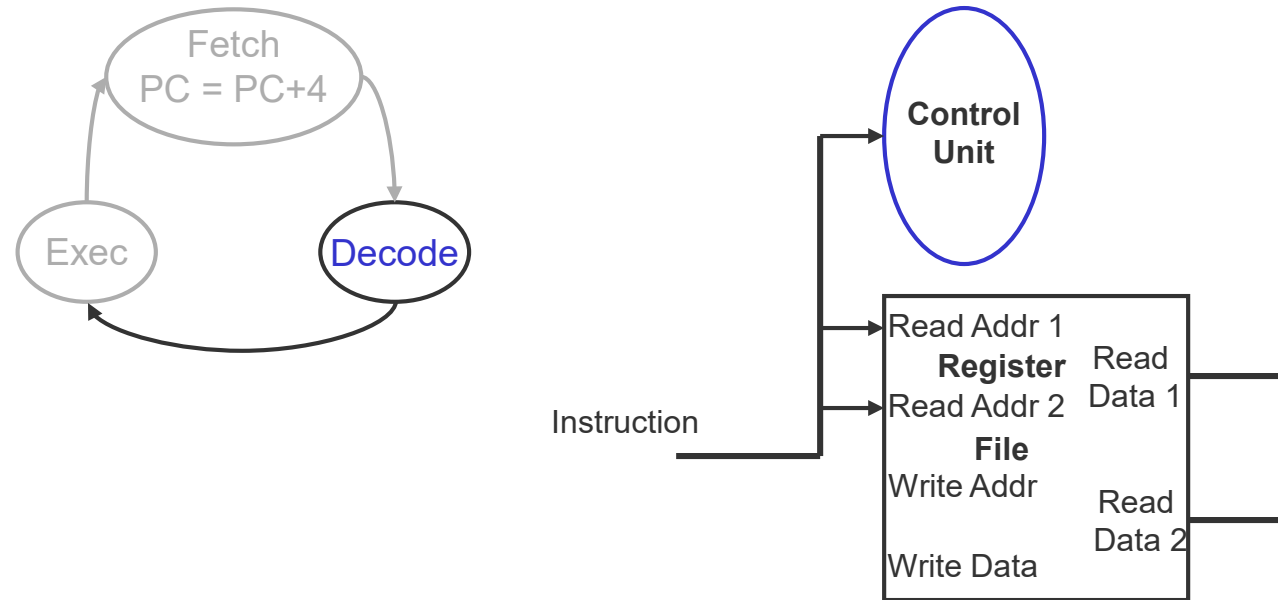


- PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
- Reading from the Instruction Memory doesn't need an explicit read control signal



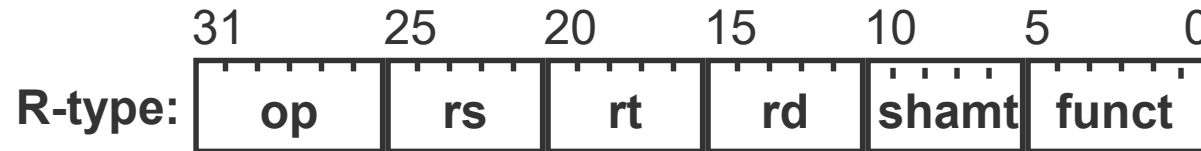
► Decoding instructions involve

- Sending the fetched instruction's opcode and function field bits to the control unit and
- Reading two values from the Register File
 - Register File addresses are contained in the instruction

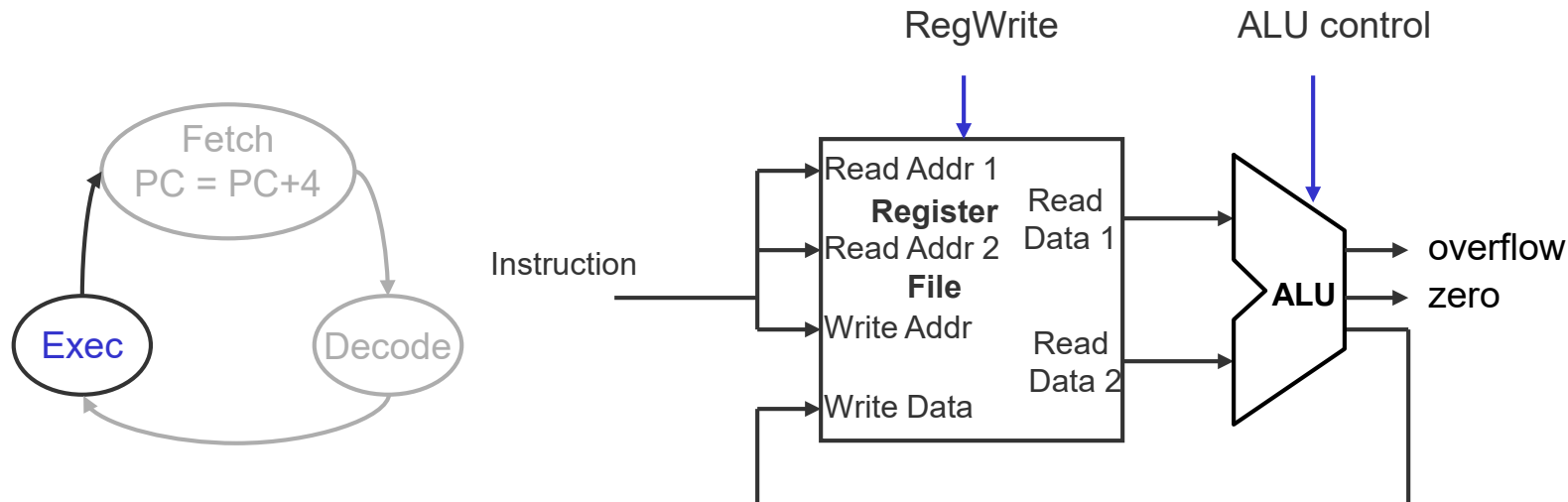




► R format operations (**add**, **sub**, **slt**, **and**, **or**)



- Perform operation (**op** and **funct**) on values in **rs** and **rt**
- Store the result back into the Register File (into location **rd**)

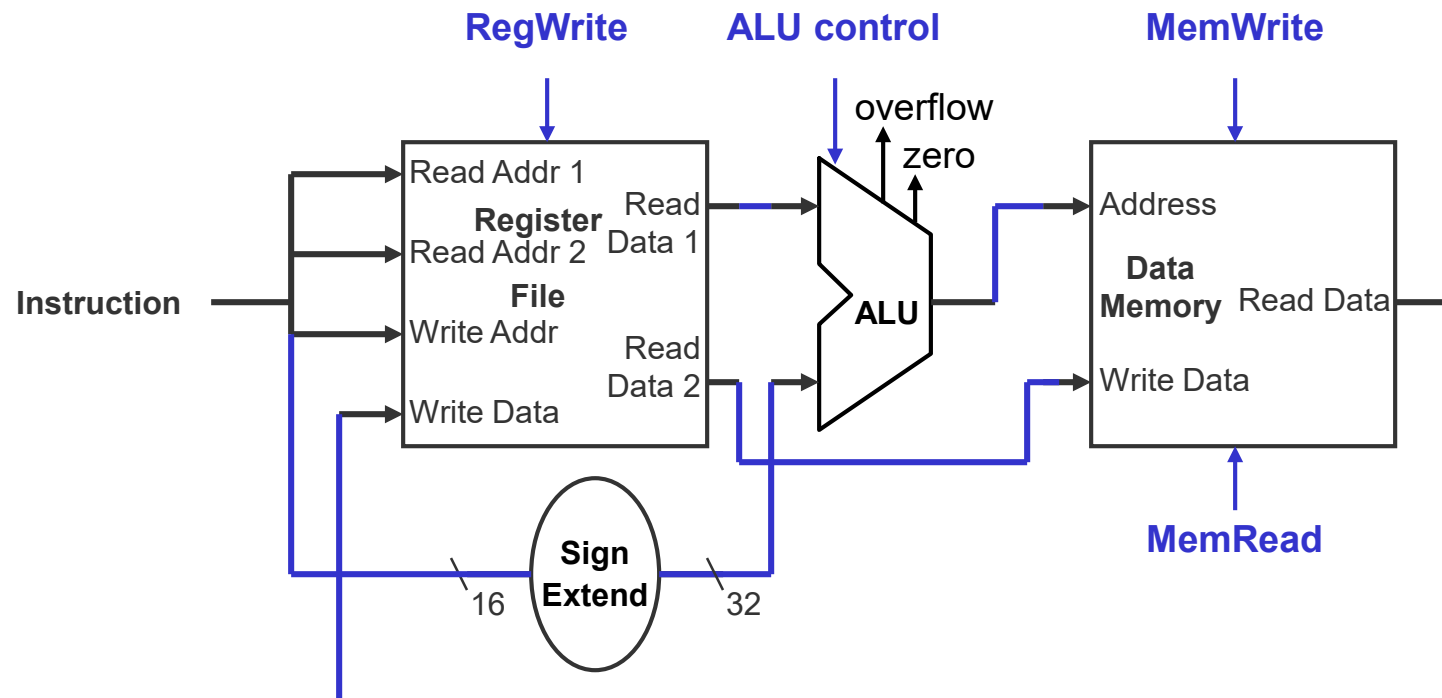


- Note that Register File is not written every cycle (e.g. **sw**), so we need an explicit write control signal for the Register File



► Load and store operations involve

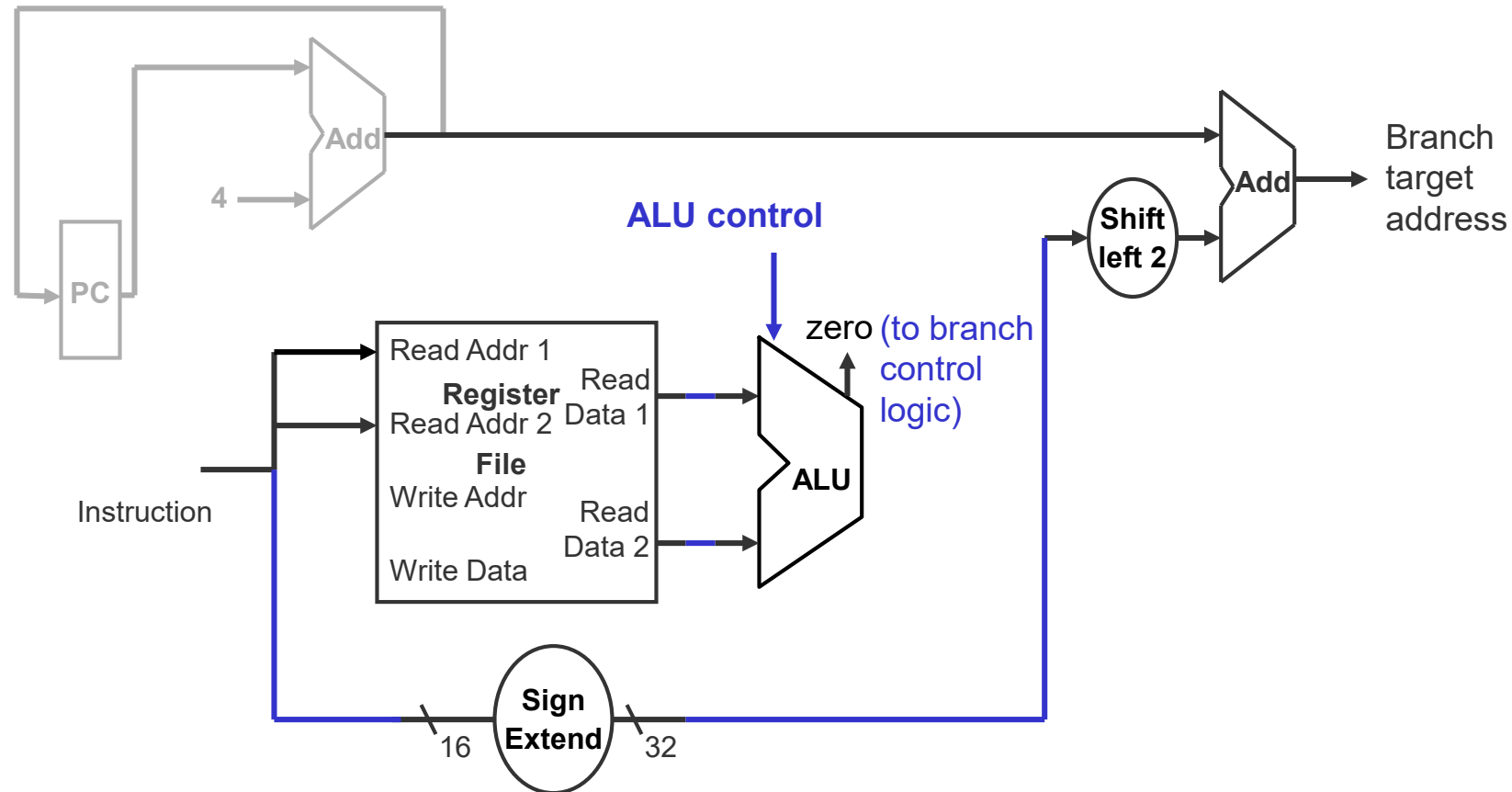
- Compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
- **Store** value (read from the Register File during decode) written to the Data Memory
- **Load** value, read from the Data Memory, written to the Register File





► Branch operations involves

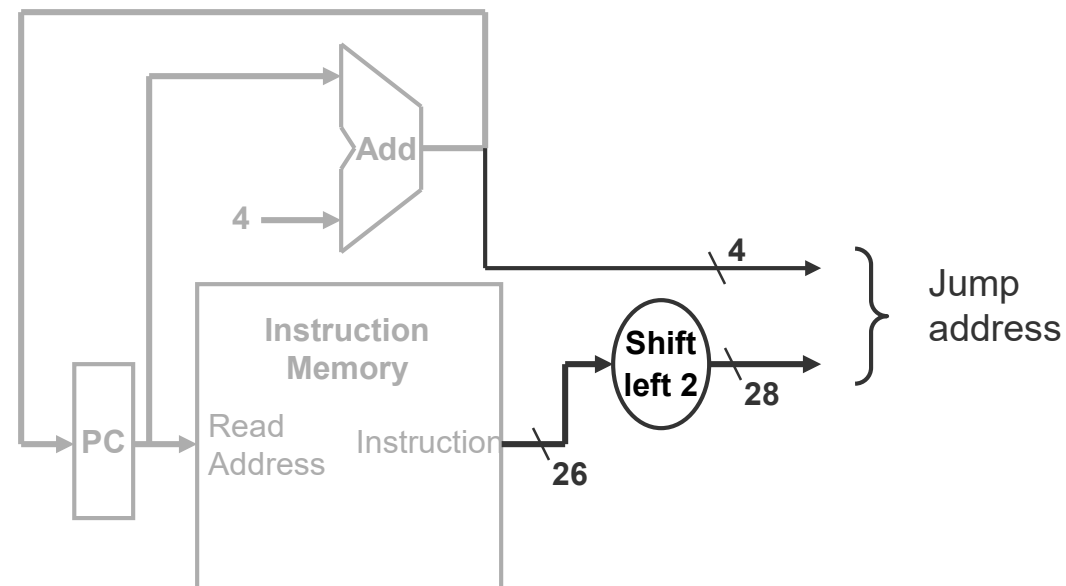
- Compare the operands read from the Register File during decode for equality (**zero** ALU output)
- Compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr





► Jump operation involves

- Replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits

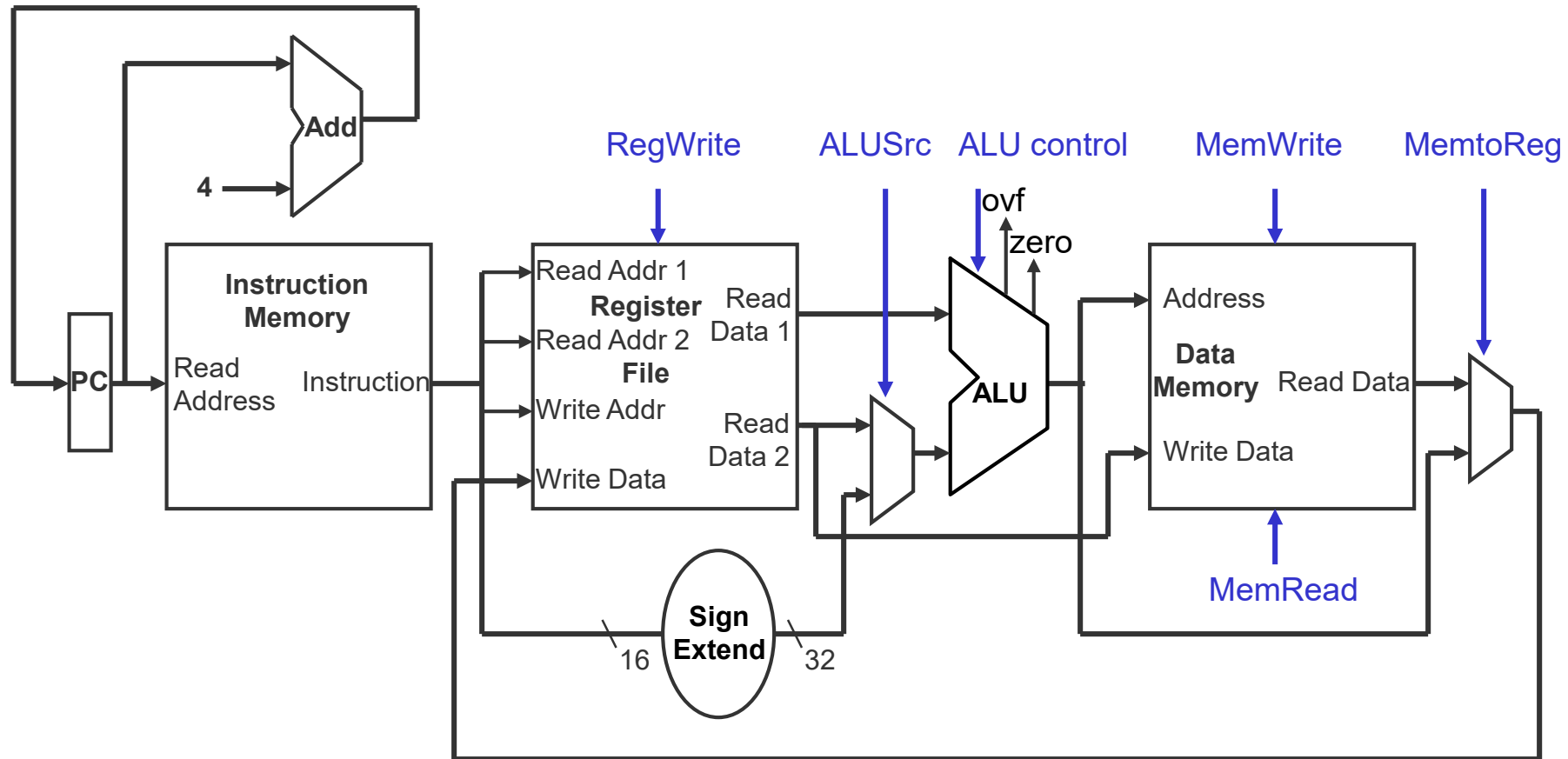




- ▶ Assemble the datapath segments and add control lines and multiplexers as needed
- ▶ **Single cycle** design – fetch, decode and execute each instruction in **one** clock cycle
 - No datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
 - **Multiplexers** needed at the input of shared elements with control lines to do the selection
 - Write signals to control writing to the Register File and Data Memory
- ▶ Cycle time is determined by length of the longest path



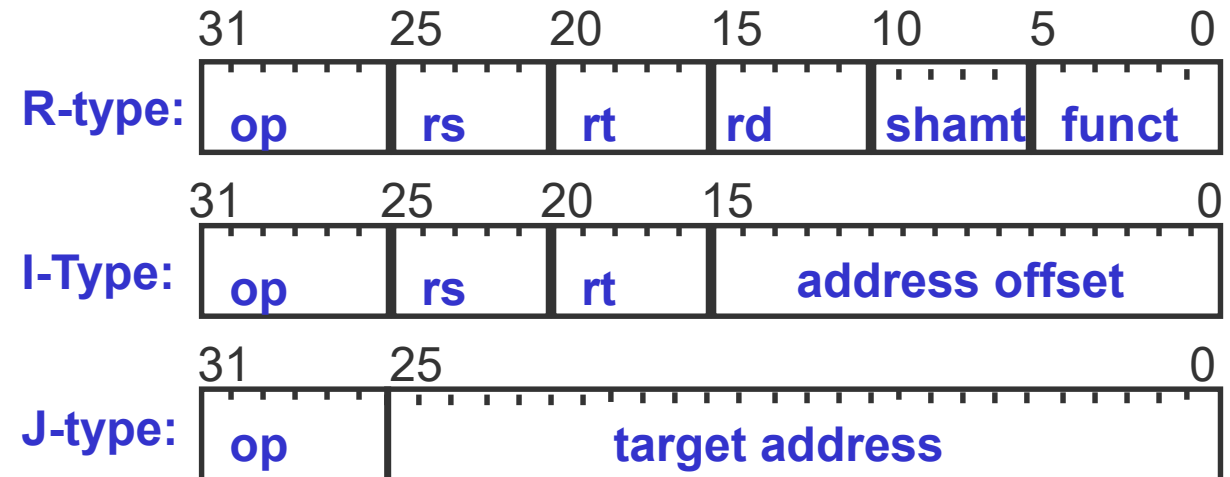
Fetch, R, and Memory Access Portions



- ▶ Selecting the operations to perform (ALU, Register File and Memory read/write)
- ▶ Controlling the flow of data (multiplexer inputs)

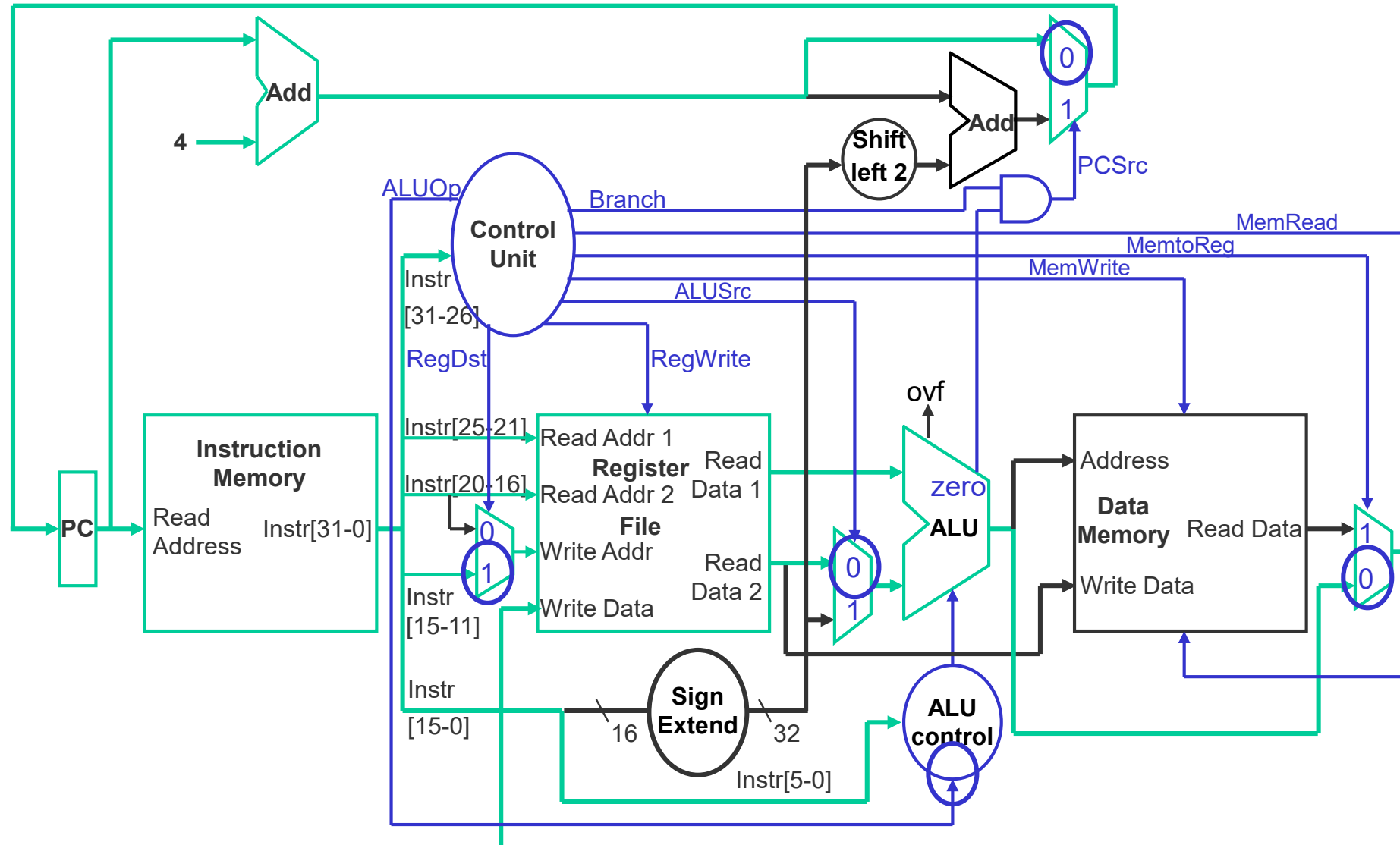
▶ Observations

- op field **always** in bits 31-26
- Addr of registers to be read are **always** specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
- Addr of register to be written is in one of **two** places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
- Offset for beq, lw, and sw **always** in bits 15-0



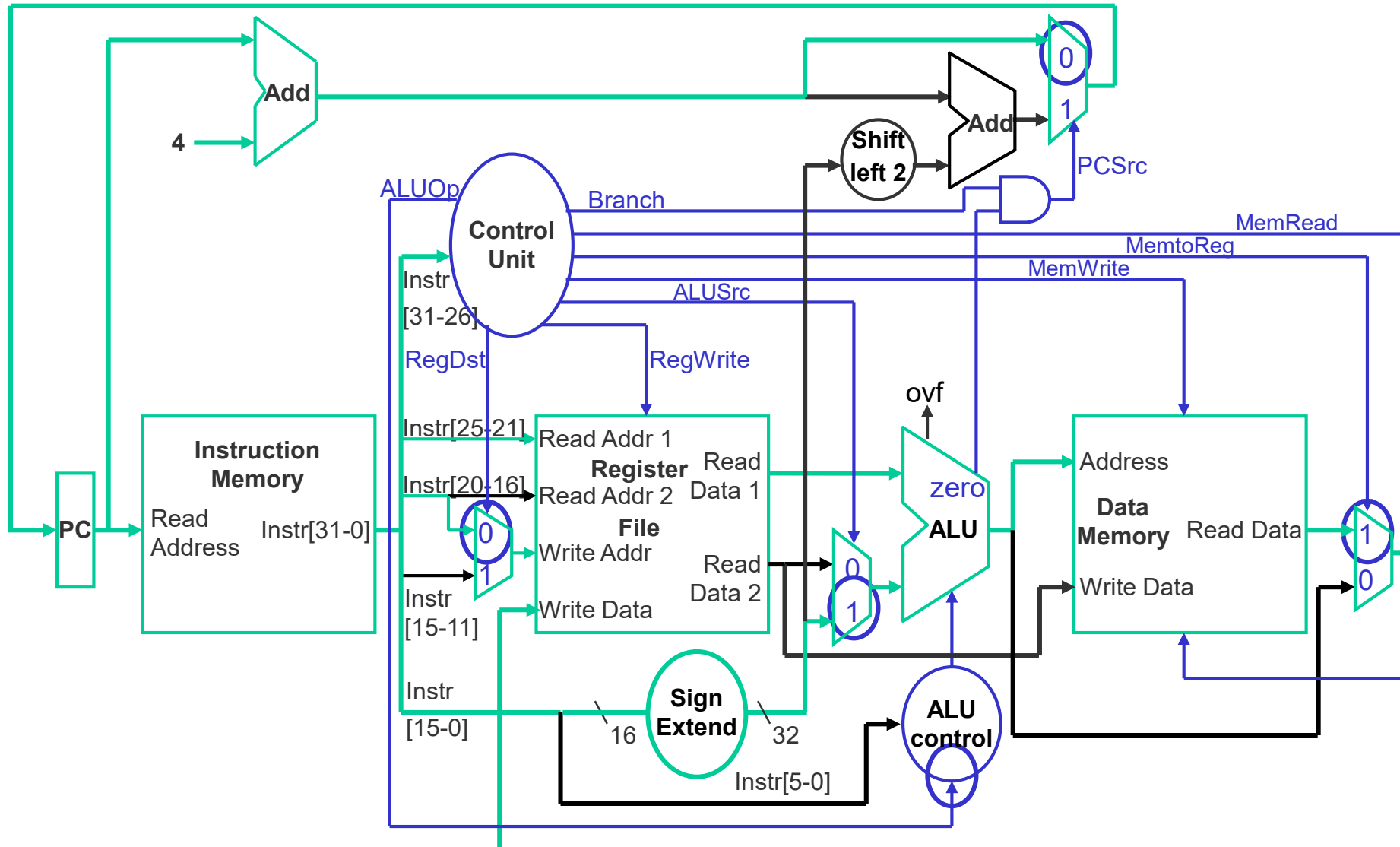


R-type Instruction Data/Control Flow



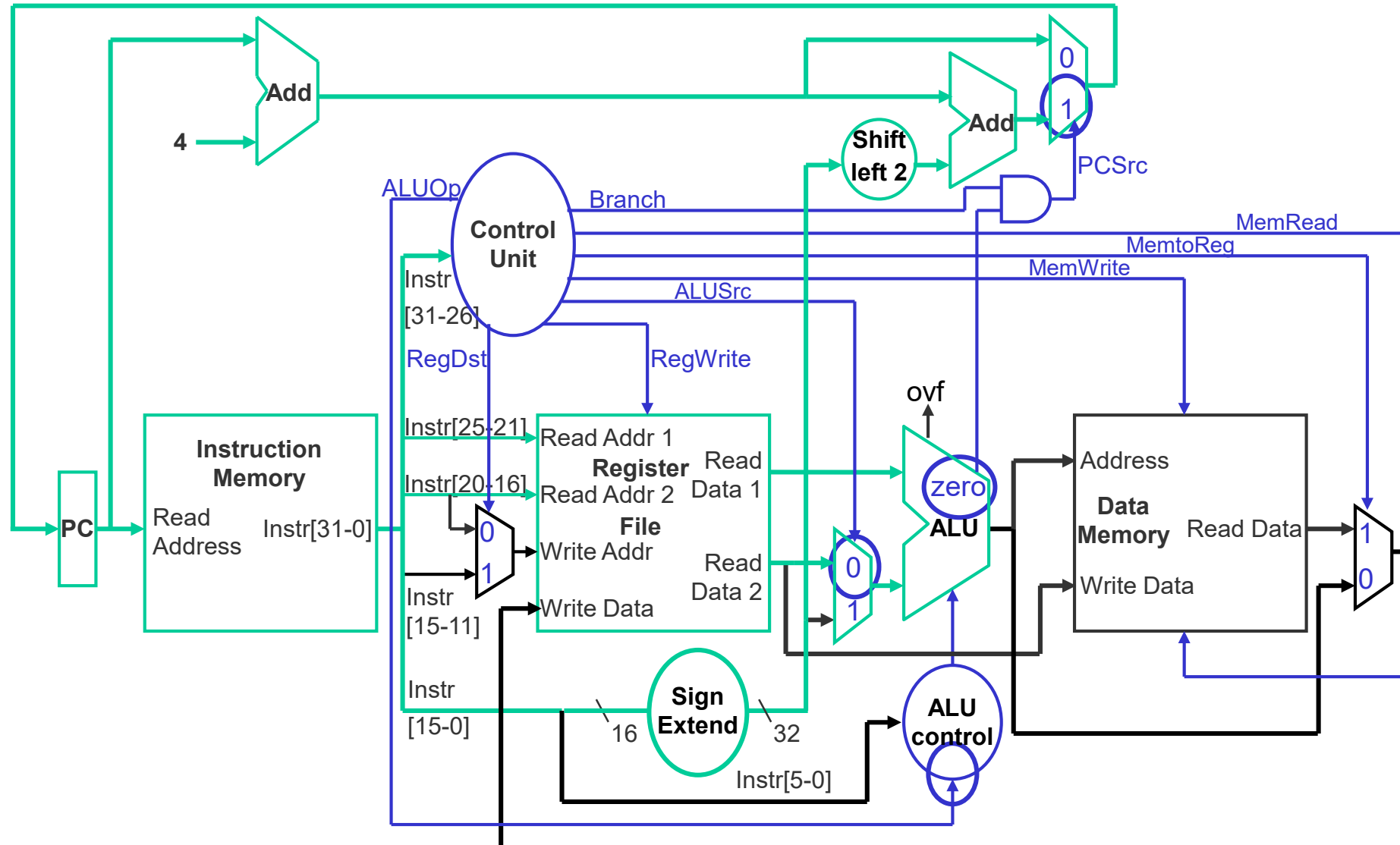


Load Word Instruction Data/Control Flow



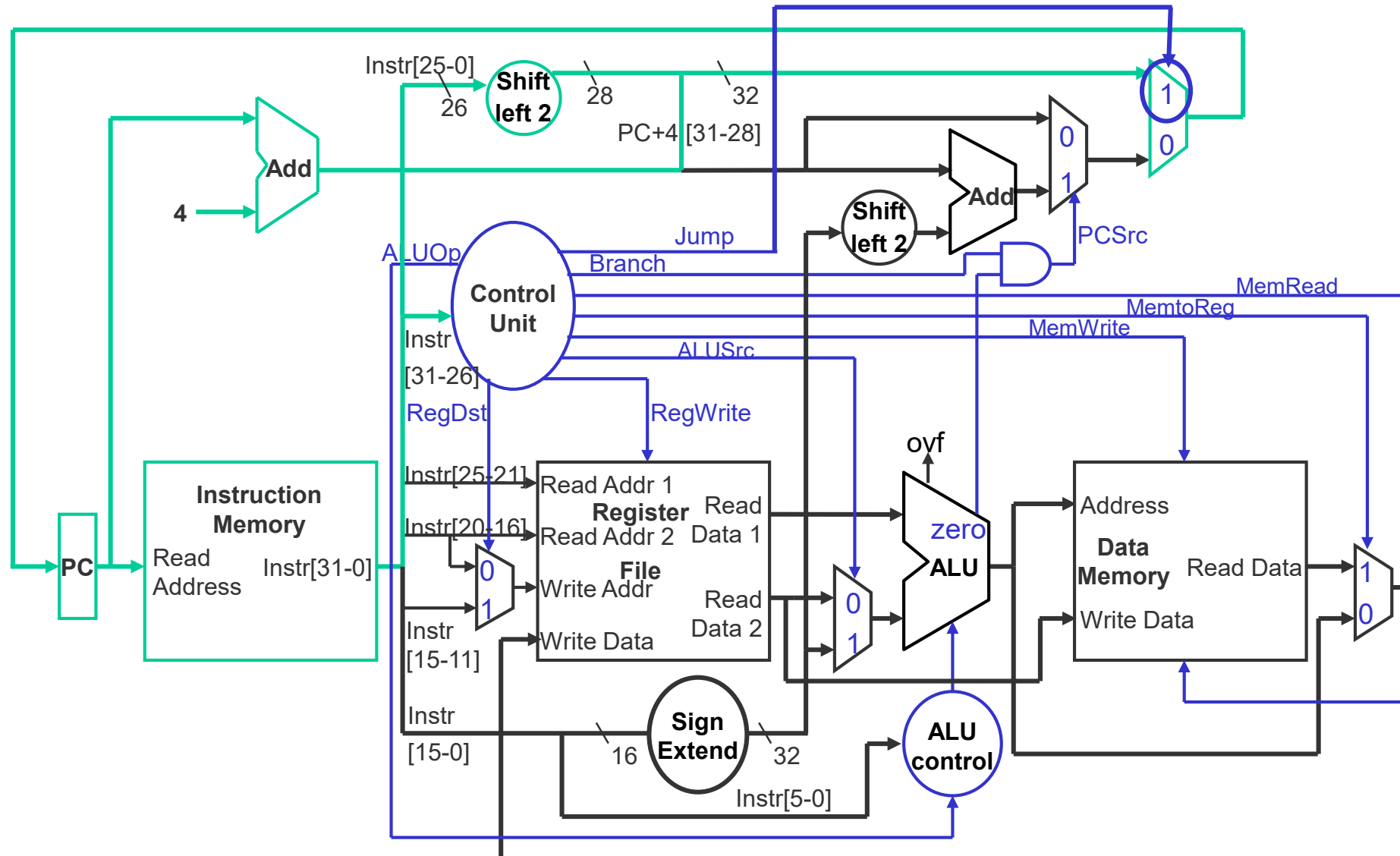


Branch Instruction Data/Control Flow





Adding the Jump Operation





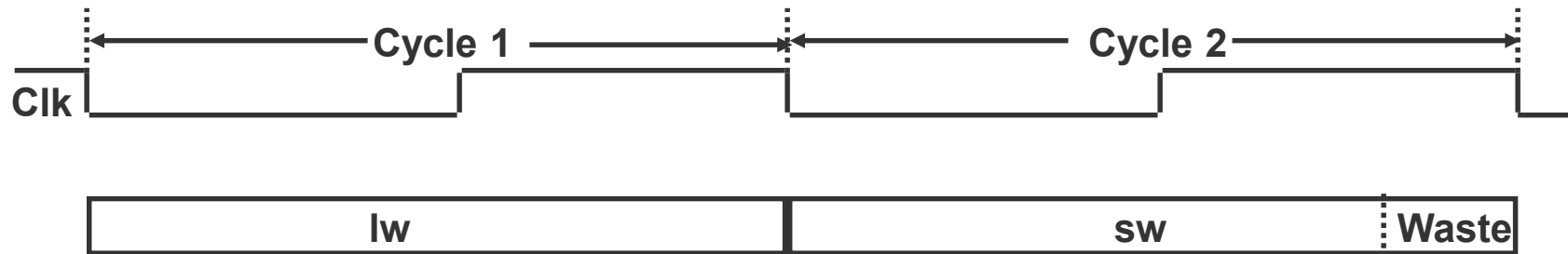
- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
- Instruction and Data Memory (200 ps)
 - ALU and adders (200 ps)
 - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200



Single Cycle Disadvantages & Advantages

- ▶ Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instruction
 - Especially problematic for more complex instructions like floating point multiply



- ▶ But is simple and easy to understand

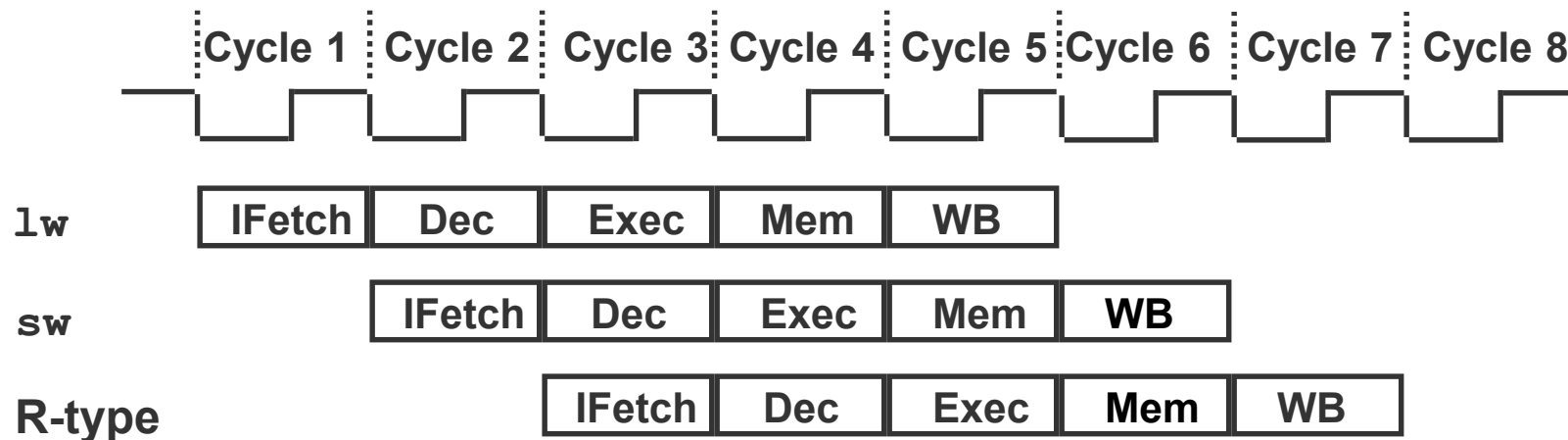


► Makes processors faster

- (All?) modern processors are pipelined for performance

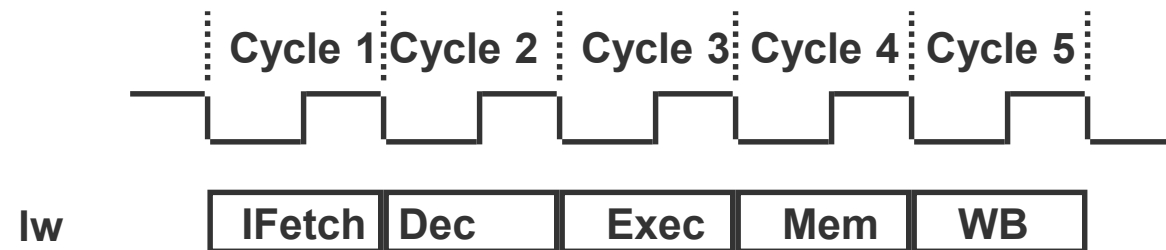
► Idea

- Subdivide single instruction into multiple stages
- Make clock cycle shorter and process one stage of an instruction per clock cycle
- Process stages of consecutive instructions in parallel



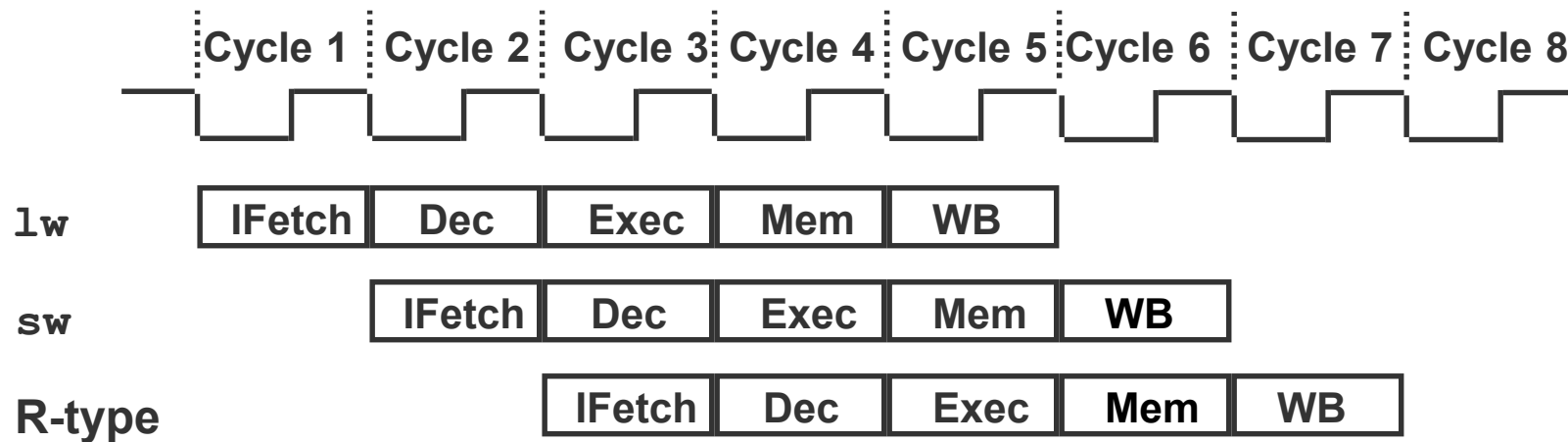


- ▶ IFetch: Instruction Fetch and Update PC
- ▶ Dec: Registers Fetch and Instruction Decode
- ▶ Exec: Execute R-type; calculate memory address
- ▶ Mem: Read/write the data from/to the Data Memory
- ▶ WB: Write the result data into the register file





- ▶ Start the **next** instruction before the current one has completed
 - Improves **throughput** – total amount of work done in a given time
 - Instruction **latency** (execution time, delay time, response time – time from the start of an instruction to its completion) is *not* reduced

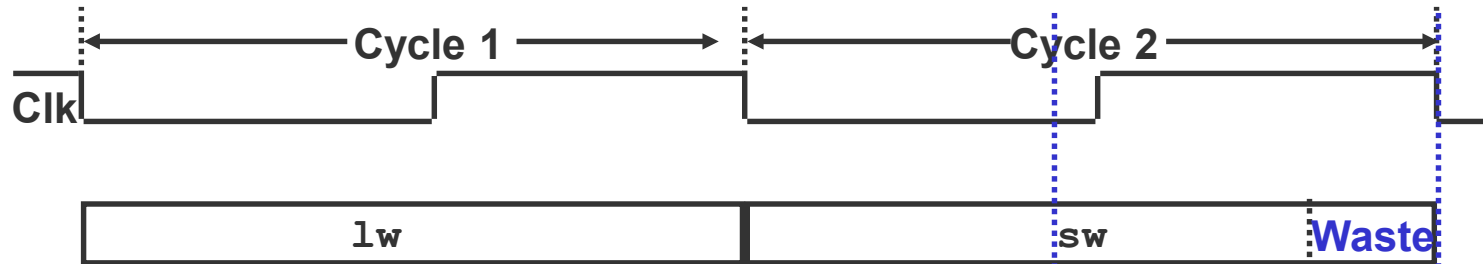


- ▶ Clock cycle (pipeline stage time) is limited by the **slowest** stage
 - For some stages don't need the whole clock cycle (e.g., WB)
 - For some instructions, some stages are **wasted** cycles (i.e., nothing is done during that cycle for that instruction)



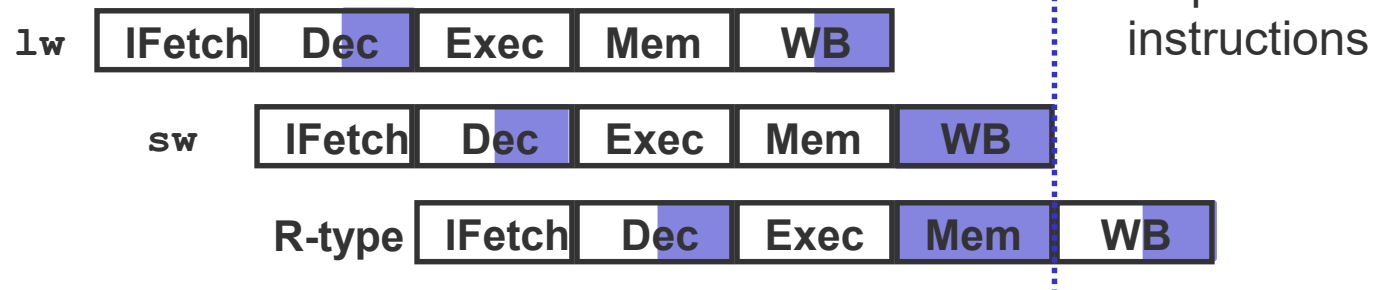
► Single Cycle Implementation (CC = 800 ps)

- Some cycle time is wasted as some instructions complete faster than others



► Pipeline Implementation (CC = 200 ps)

- Some cycle time is wasted as
 - Some instructions complete faster than others
 - Some stages use only half a clock cycle
- But instructions are processed in parallel





- ▶ Remember **the** performance equation
 - $\text{CPU time} = \text{CPI} * \text{CC} * \text{IC}$
- ▶ CPI in the pipelined case
 - is almost 1 if one instruction completes per CC, ...
 - but may increase as pipelining requires
 - Initial time to fill the pipeline
 - Some (stall) cycles without processed instructions due to hazards
- ▶ Example: five stage pipeline
 - Pipelining allows for shorter CCs: 200 ps instead of 800 ps
 - Single instruction takes 5 CCs à 200 ps = 1000 ps
instead of 1 CCs à 800 ps
 - But 1000 add operations take 1004 CCs à 200 ps = 200,8 ns
instead of 1000 CCs à 800 ps = 800 ns

⇒ 5 stage pipeline is 4 times faster



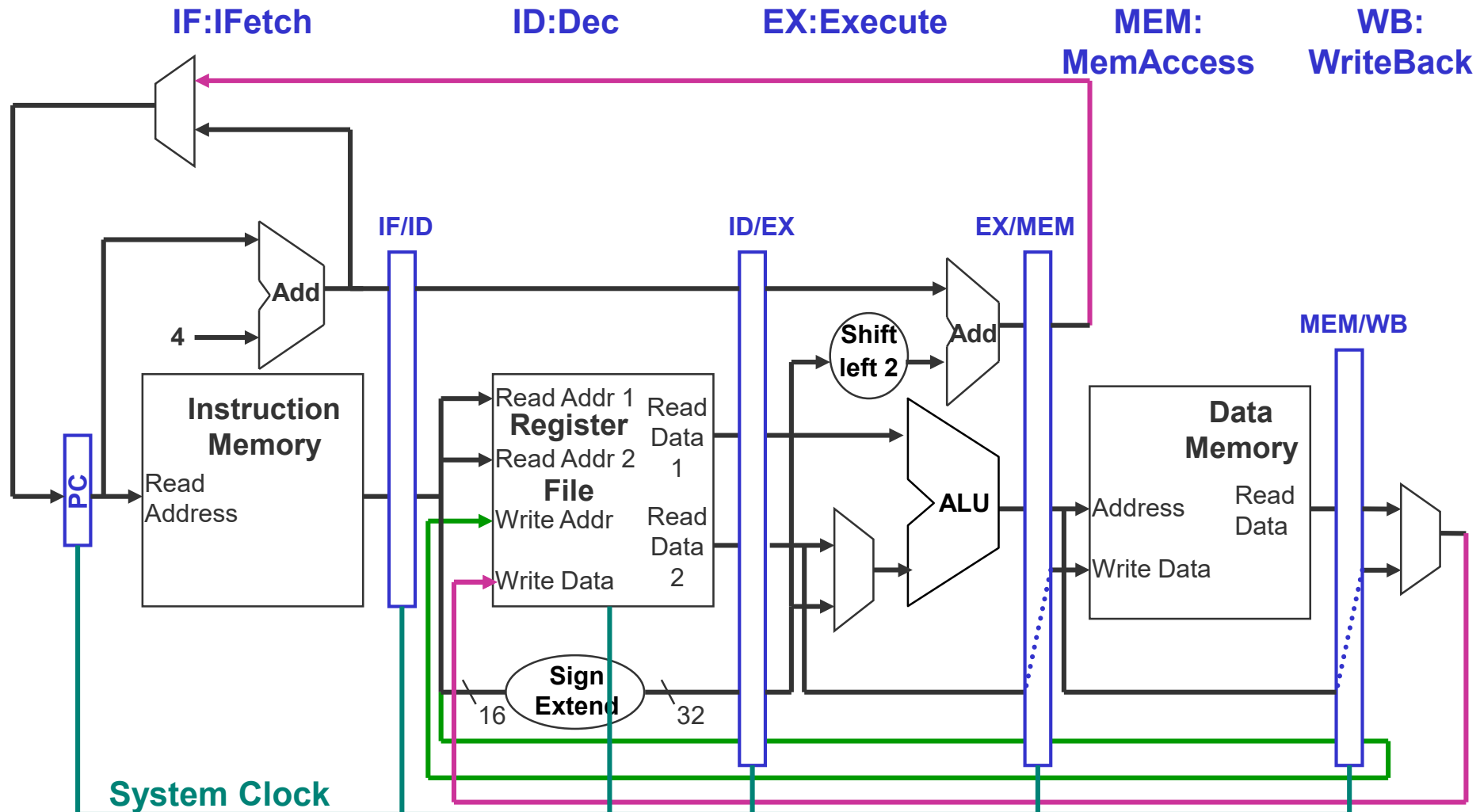
► What makes it easy

- All instructions are the same length (32 bits)
 - Can fetch in the 1st stage and decode in the 2nd stage
- Few instruction formats (three) with **similarities** across formats
 - Can begin reading register file in 2nd stage
- Memory operations occur only in loads and stores
 - Can use the execute stage to calculate memory addresses
- Each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
- Operands must be aligned in memory so that a single data transfer takes only one data memory access



MIPS Pipeline Datapath Additions/Mods

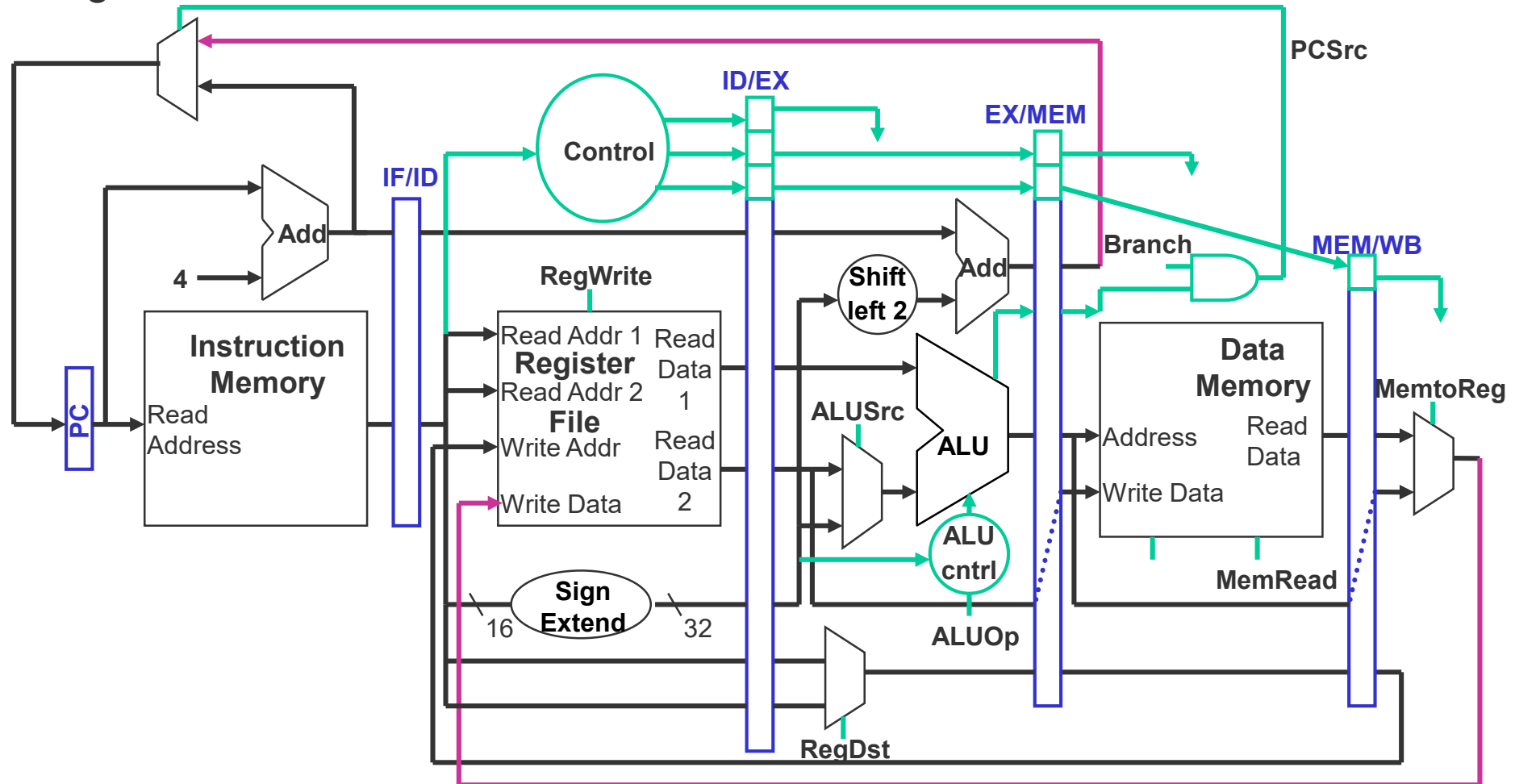
- State registers between each pipeline stage to **isolate** them





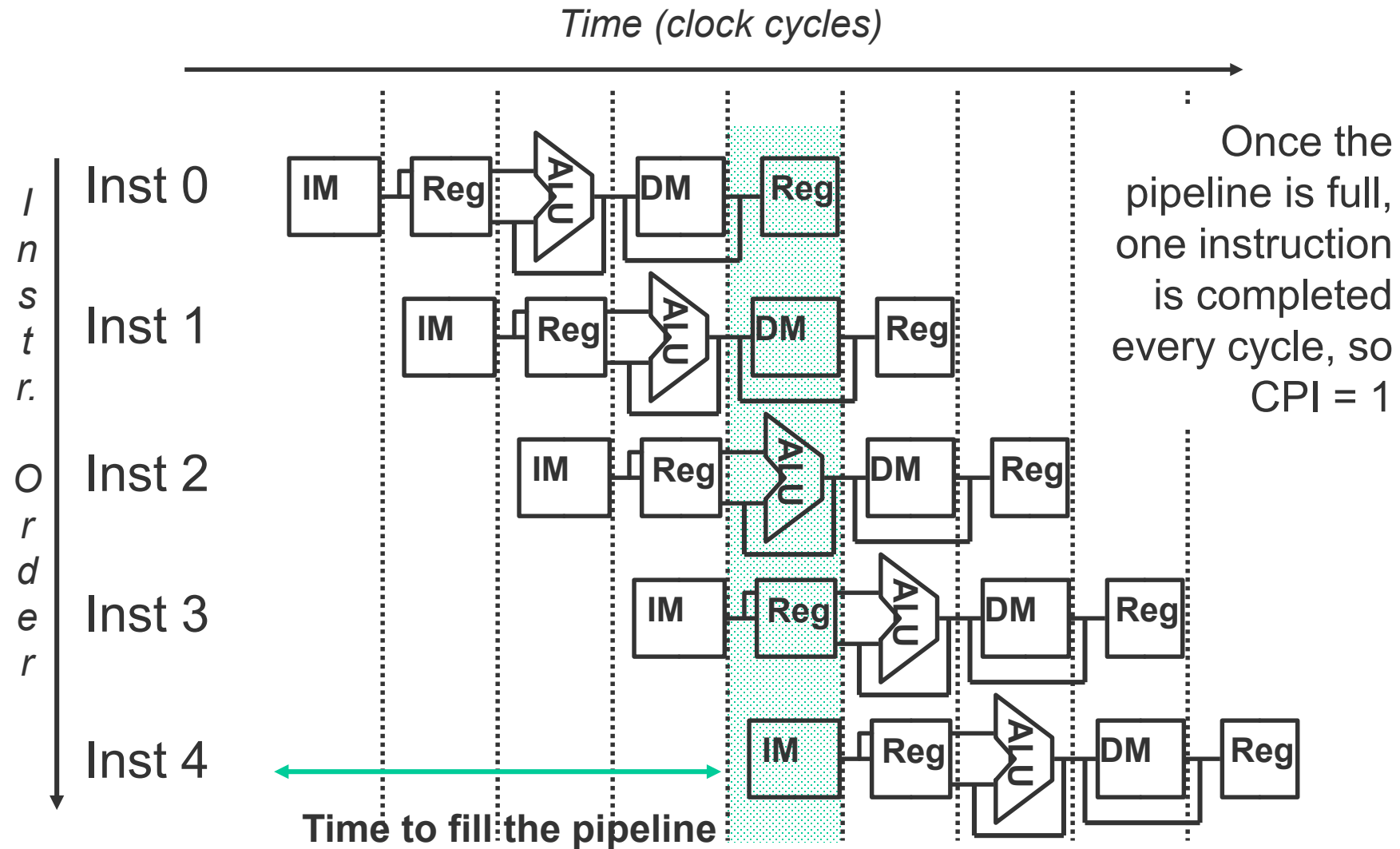
MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode and held in the **state registers** between pipeline stages



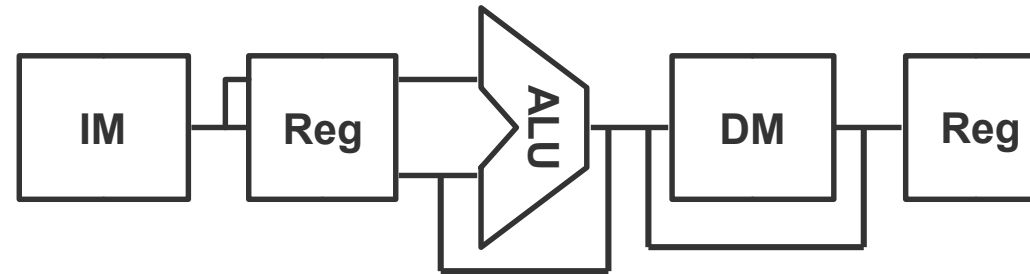


Why Pipeline? For Performance!





Graphically Representing MIPS Pipeline



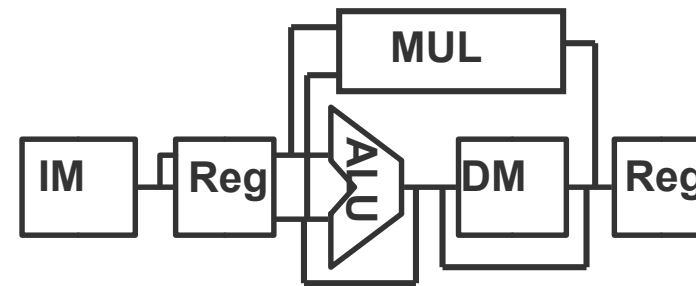
► Can help with answering questions like:

- How many cycles does it take to execute this code?
- What is the ALU doing during cycle 4?
- Is there a hazard, why does it occur, and how can it be fixed?



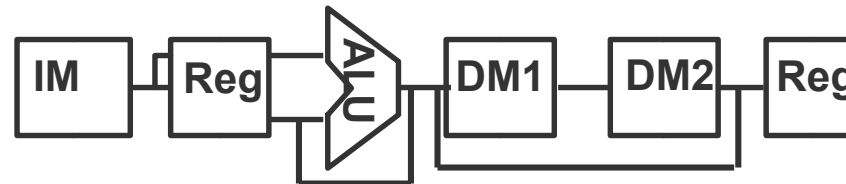
► What about the (slow) multiply operation?

- Make the clock twice as slow or ...
- Let it take two cycles (since it doesn't use the DM stage)



► What if the data memory access is twice as slow as the instruction memory?

- Make the clock twice as slow or ...
- Let data memory access take two cycles (and keep the same clock rate)





► Yes: pipeline hazards

- **Structural hazards**: attempt to use the same resource by two different instructions at the same time
- **Data hazards**: attempt to use data before it is ready
 - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- **Control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - Branch and jump instructions, exceptions

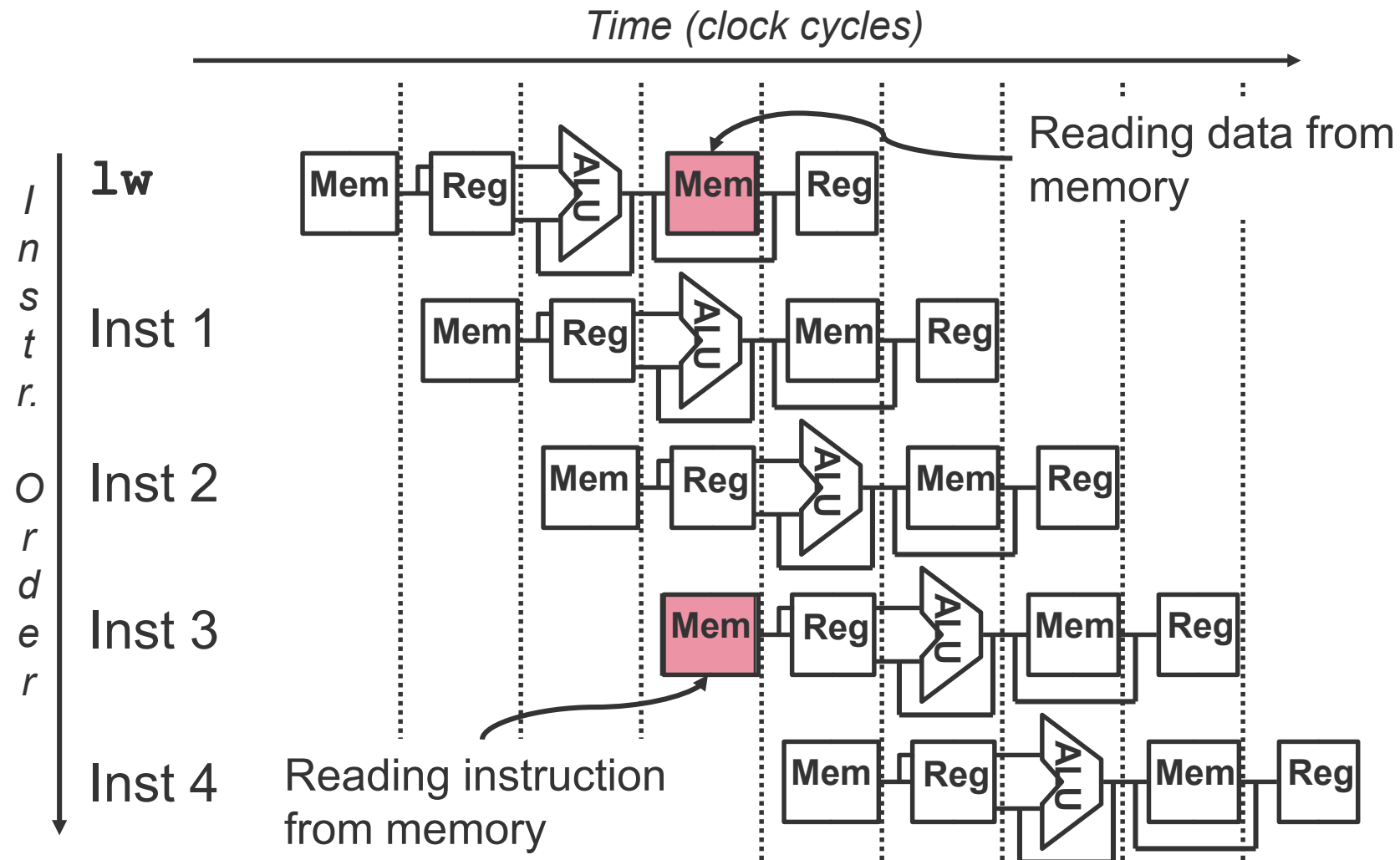
► Can usually resolve hazards by waiting

- Pipeline control must **detect** the hazard and
- Take action to **resolve** hazards



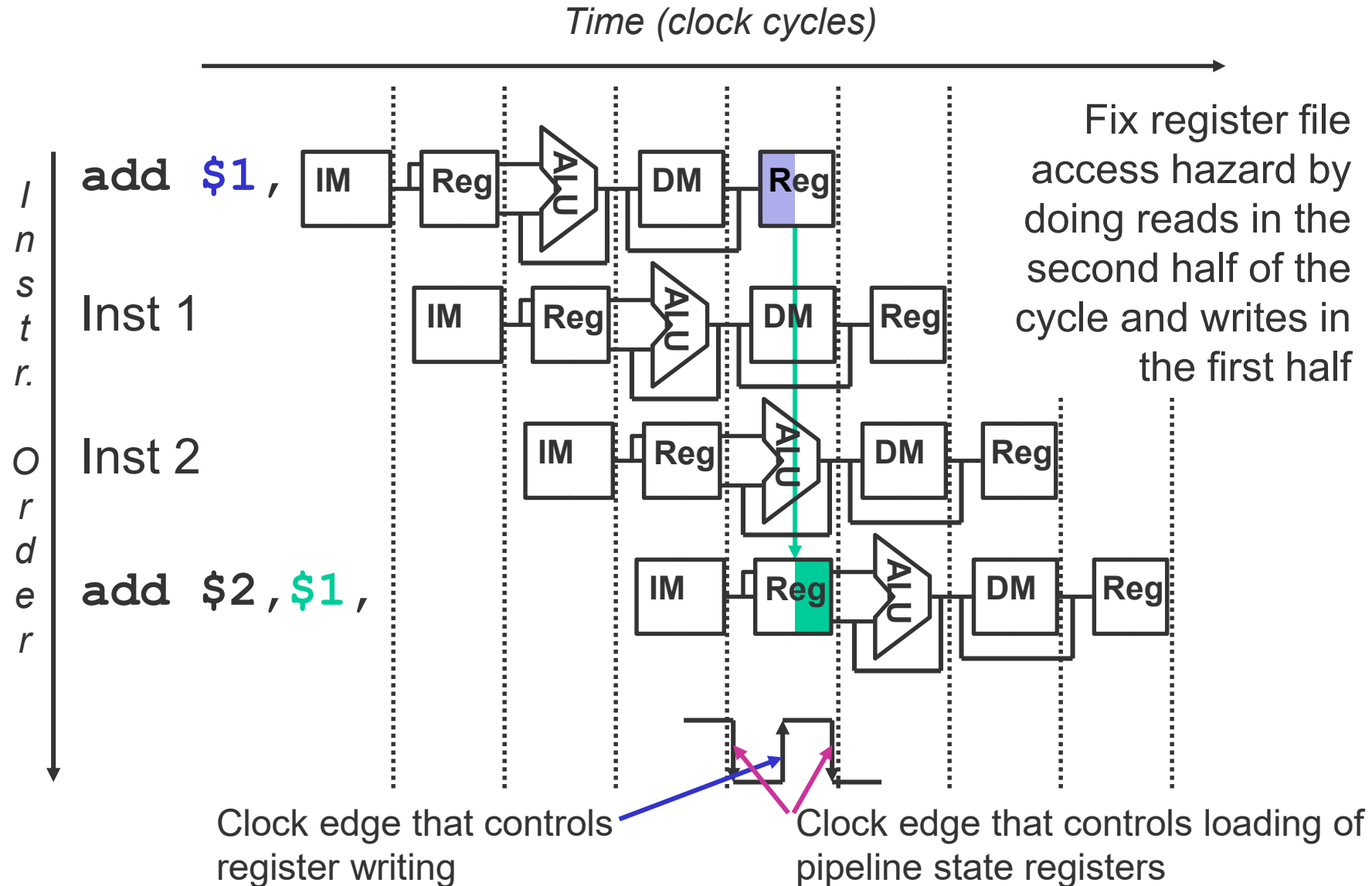
Structural Hazard: Single Memory

- Fix with separate instr and data memories (I\$/IM and D\$/DM)





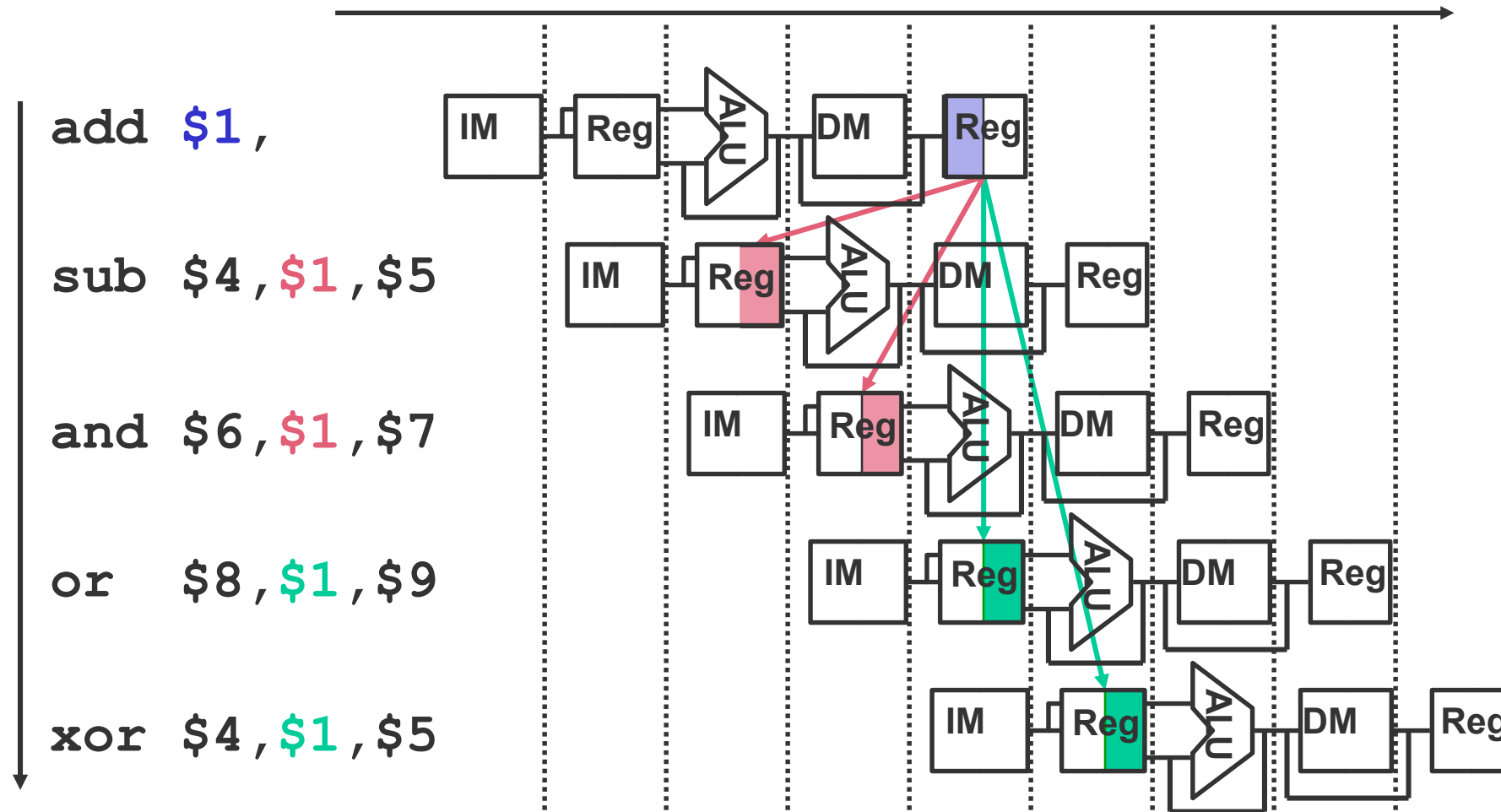
Structural Hazard: Register File Access?





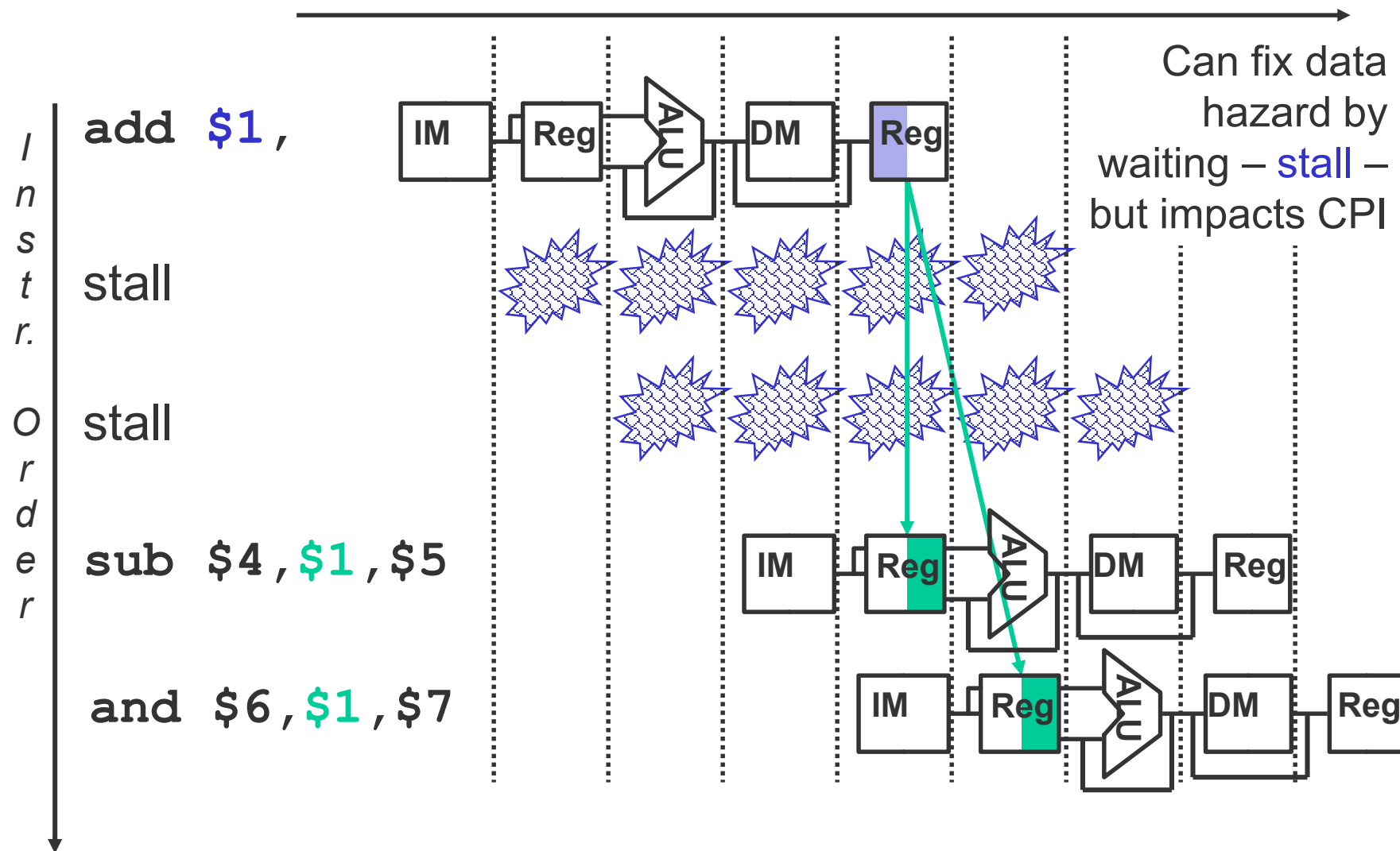
Data Hazards due to Register Usage

- Read before write data hazard
 - Caused by dependency backward in time



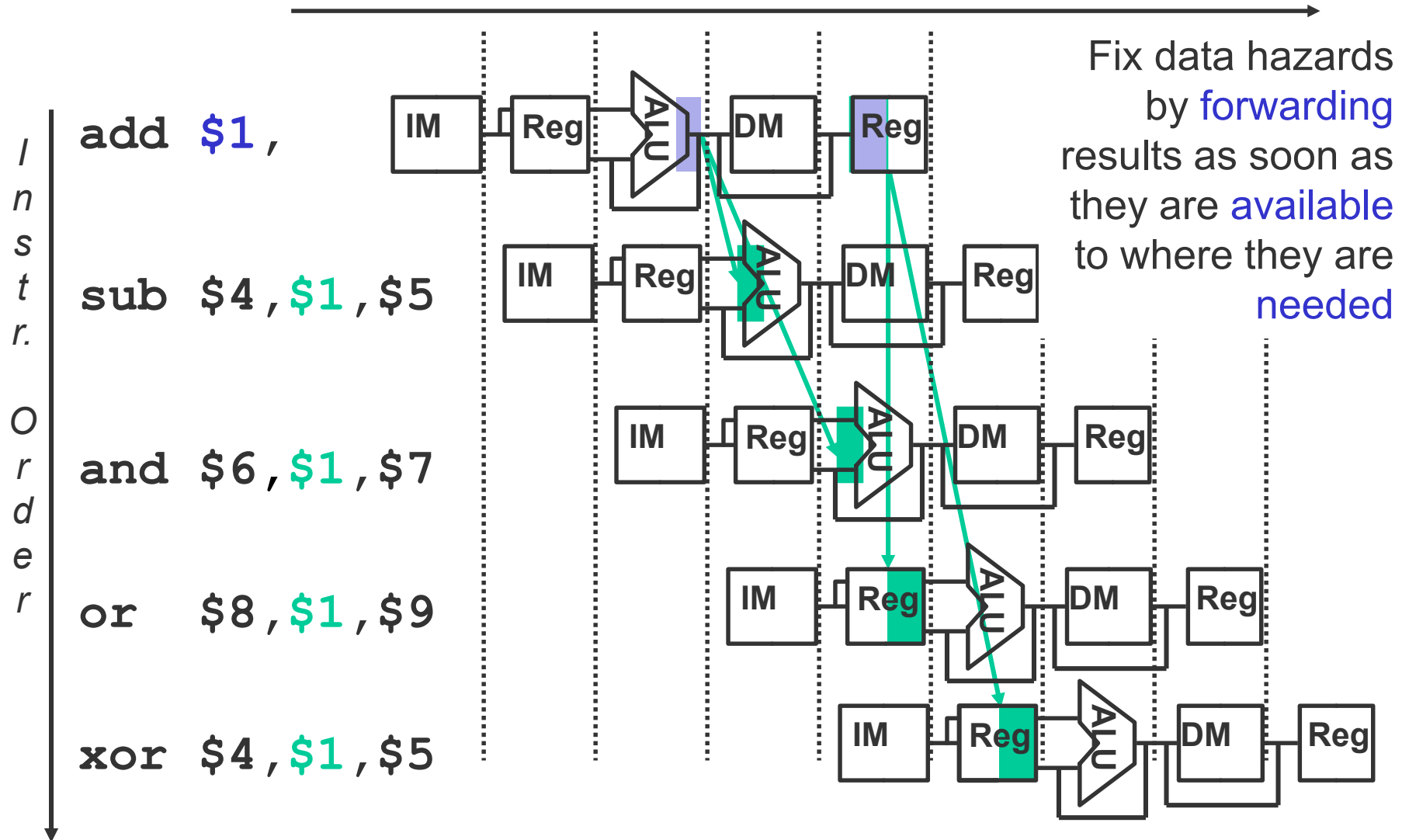


Pipeline Stalling to “Fix” a Data Hazard



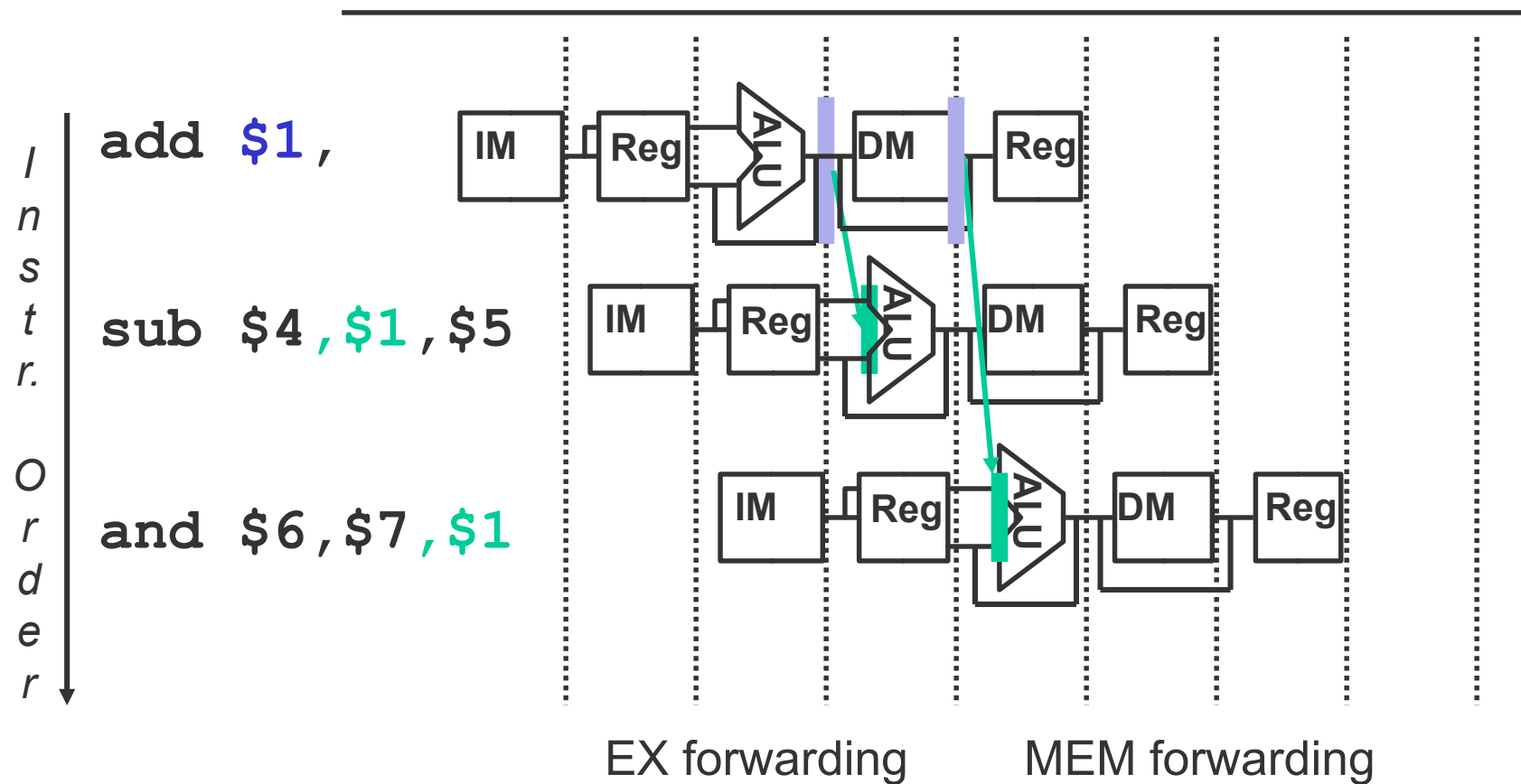


Data Forwarding to fix a Data Hazard





- ▶ Take the result from the earliest point that it exists in **any** of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- ▶ For ALU functional unit: the inputs can come from **any** pipeline register rather than just from ID/EX by
 - Adding multiplexers to the inputs of the ALU
 - Connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs
 - Adding the proper control hardware to control the new muxes
- ▶ Other functional units may need similar forwarding logic (e.g., the DM)
- ▶ With forwarding a CPI of 1 can be achieved even in the presence of data dependencies

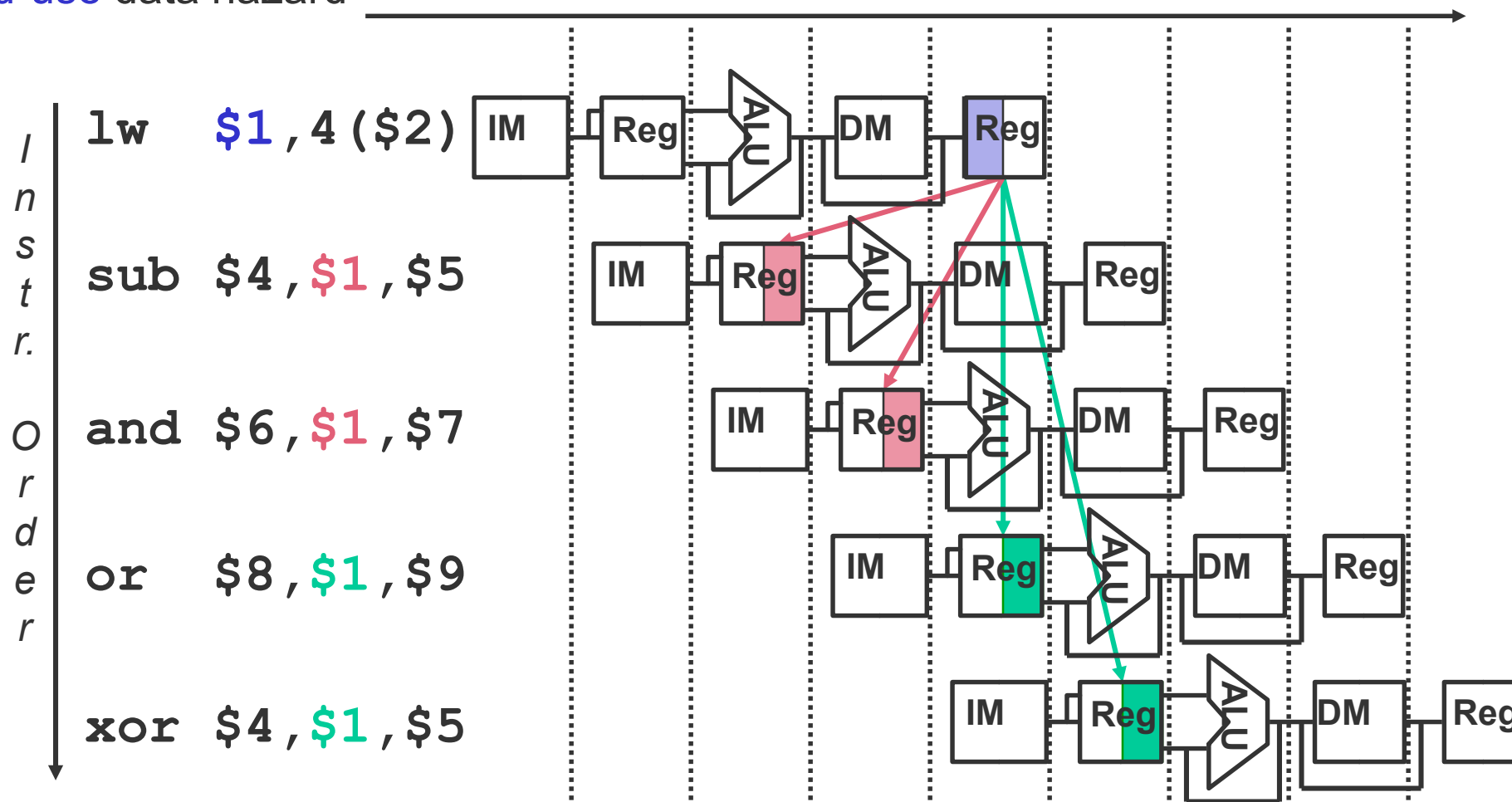


► Forwarded data from ALU found in the pipeline registers of different stages



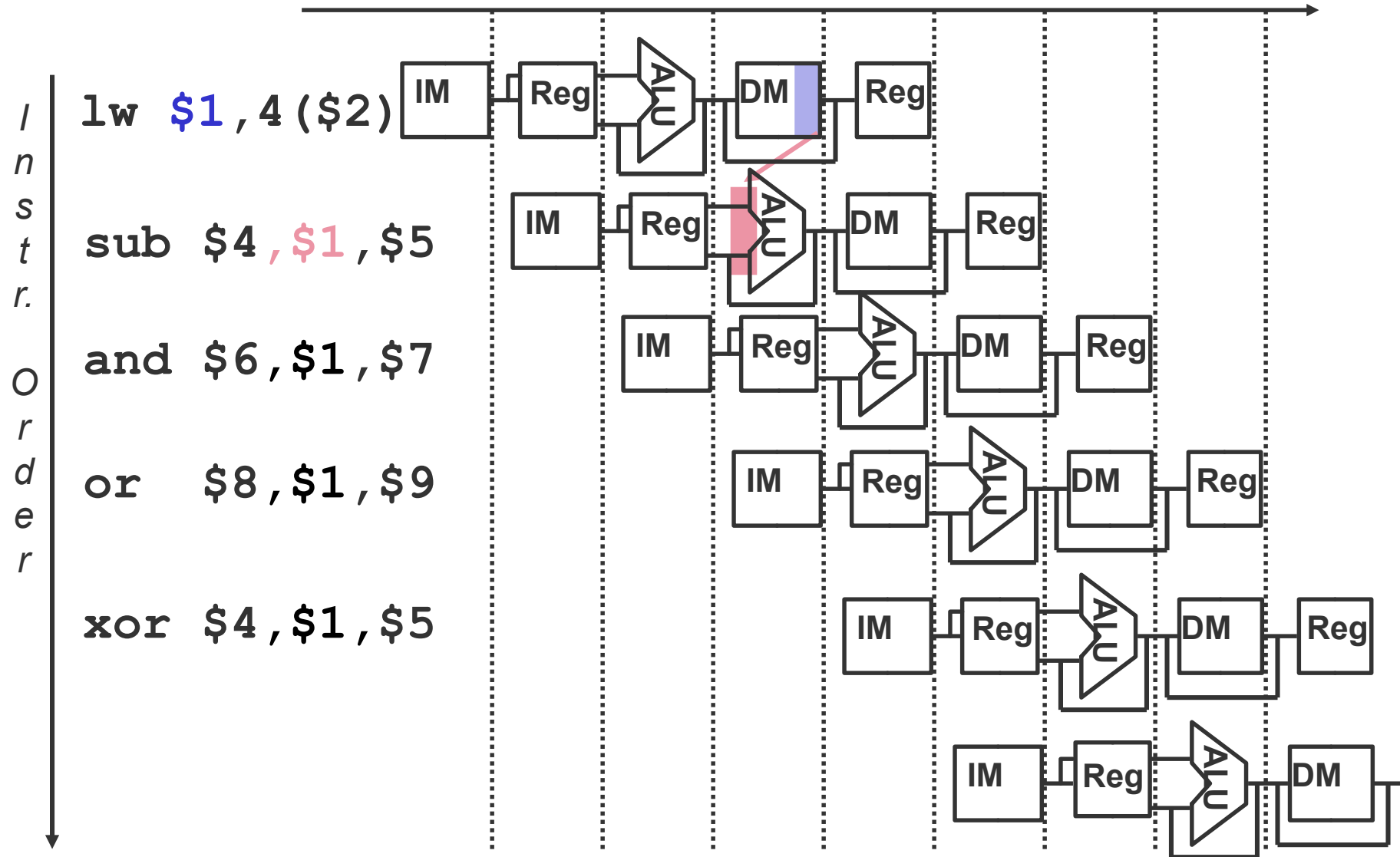
► Dependencies backward in time cause hazards

- Load-use data hazard



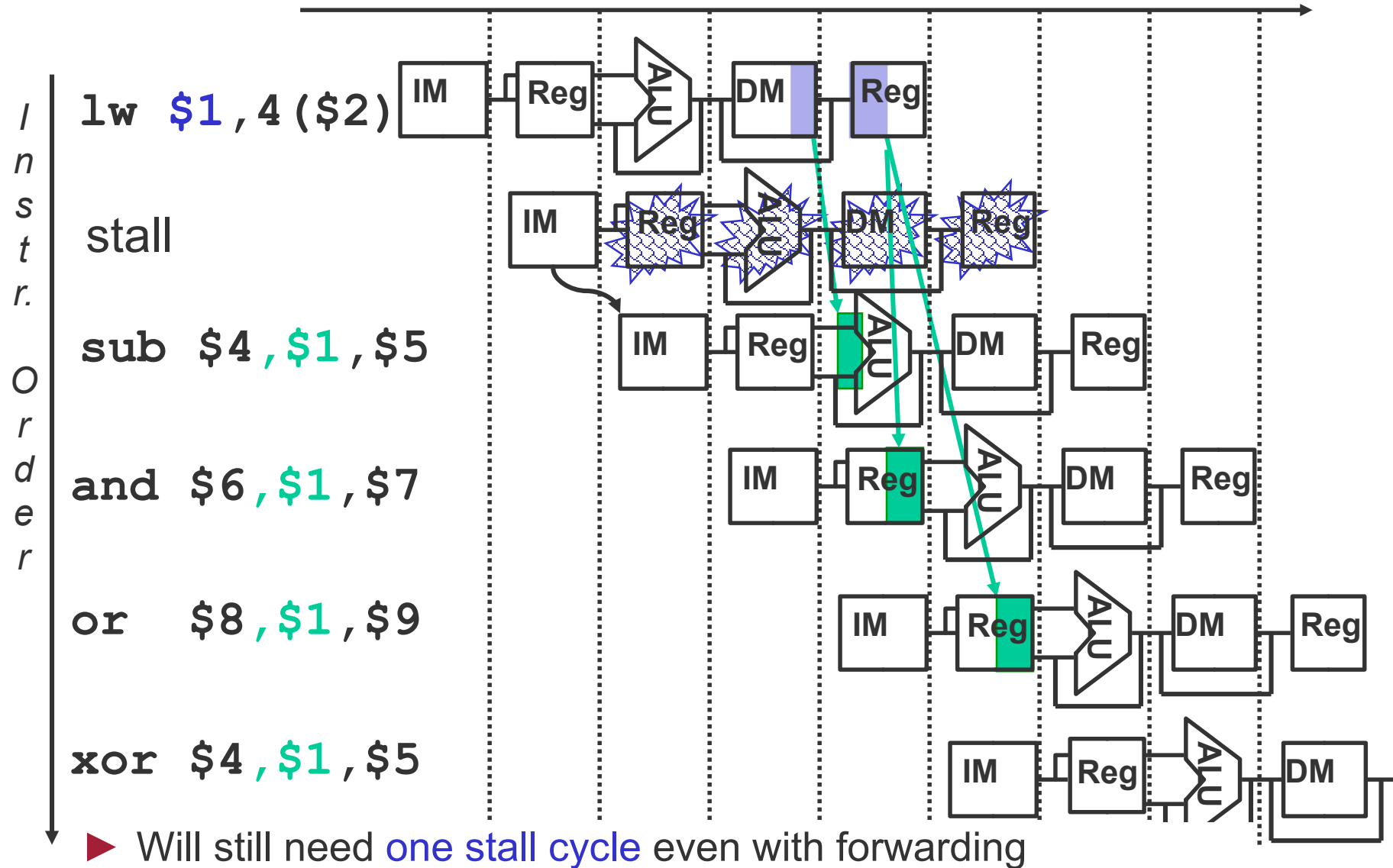


Forwarding with Load-Use Data Hazards (1)





Forwarding with Load-Use Data Hazards (2)

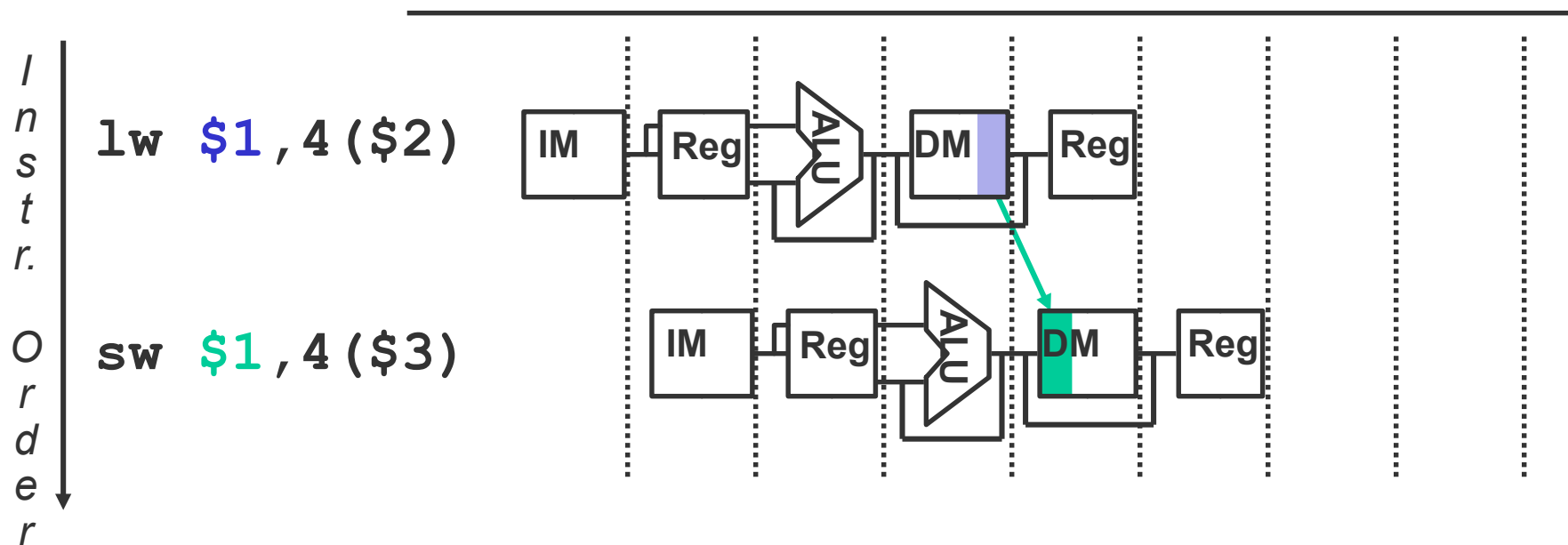




Forwarding: Memory-to-Memory Copies

► Memory-to-memory copies

- Loads immediately followed by stores
- Stalling can be avoided by forwarding data from the MEM/WB pipeline register to the data memory input
- Would need to add a Forward Unit and a mux to the MEM stage





► Control hazards

- When the flow of instruction addresses is not sequential (i.e., $PC = PC + 4$)
- Caused by change of flow instructions
 - Unconditional branches (`j`, `jal`, `jr`)
 - Conditional branches (`beq`, `bne`)
 - Exceptions
- Occur less frequently than data hazards,
- But there is nothing as effective against control hazards as forwarding is for data hazards

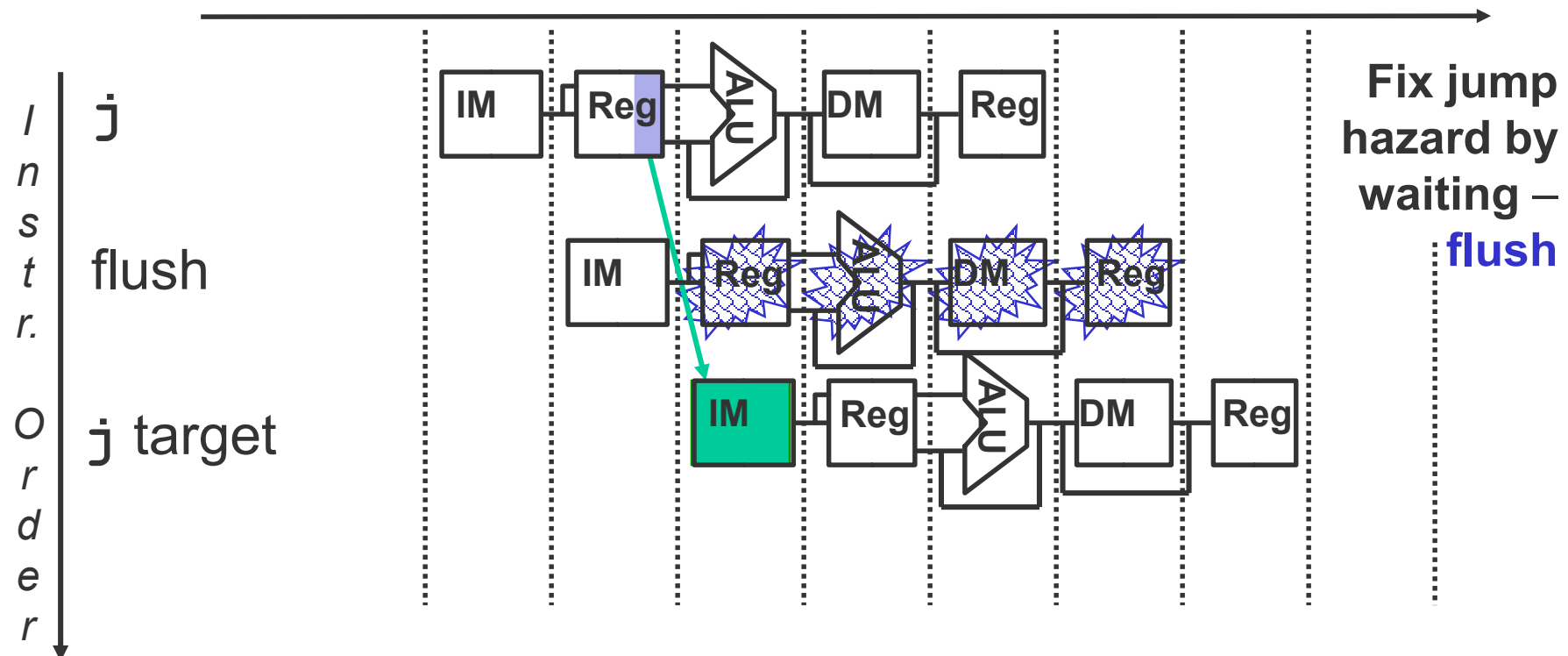
► How to deal with control hazards

- Stall (impacts CPI)
- Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
- Predict and hope for the best!



► Jumps not decoded until ID stage

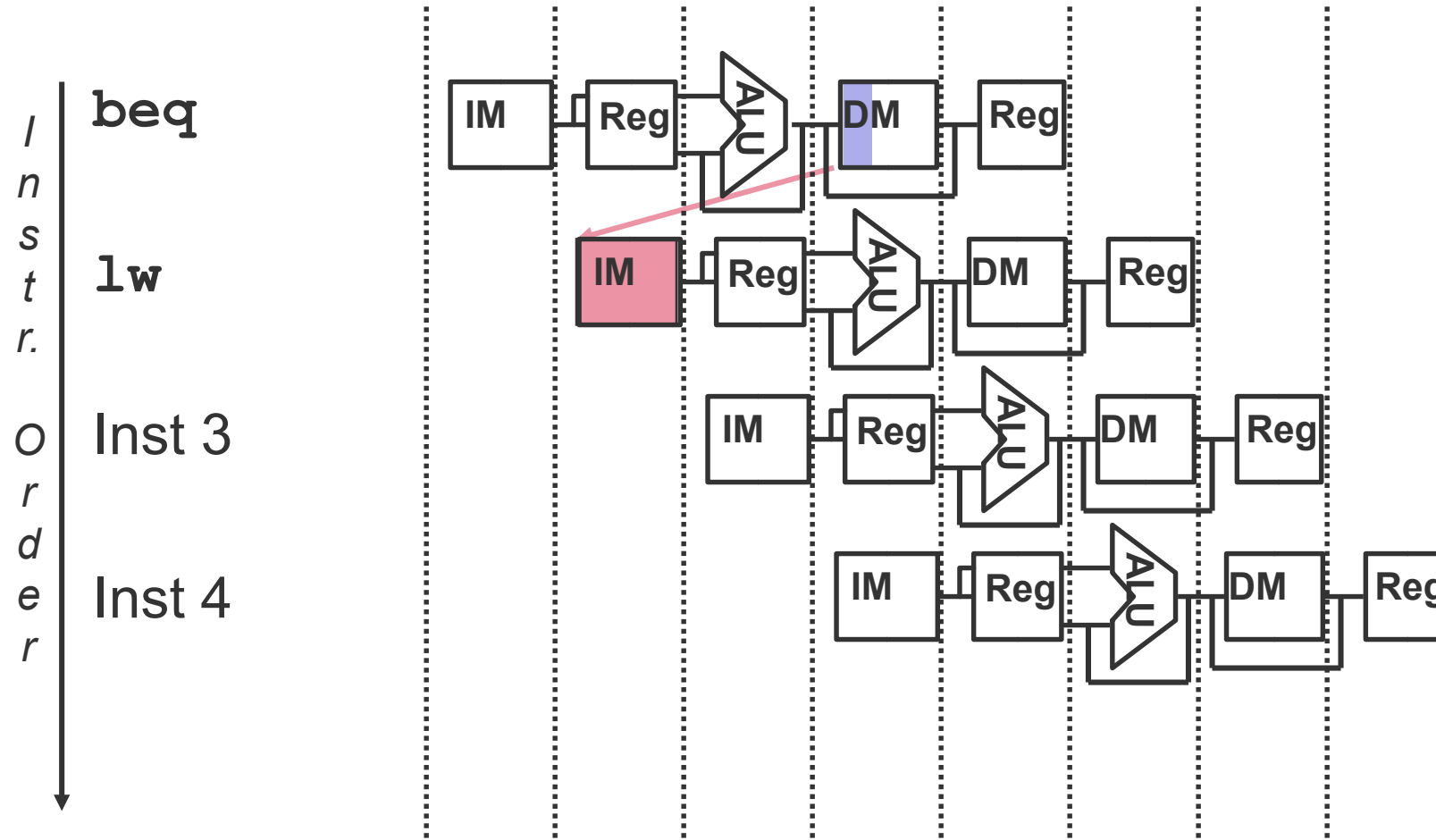
- Wrong successor instruction already fetched $\Rightarrow$ one flush is needed
 - Flush: erase wrong instruction from pipeline, turning it into a noop
 - Noop: no operation
- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix





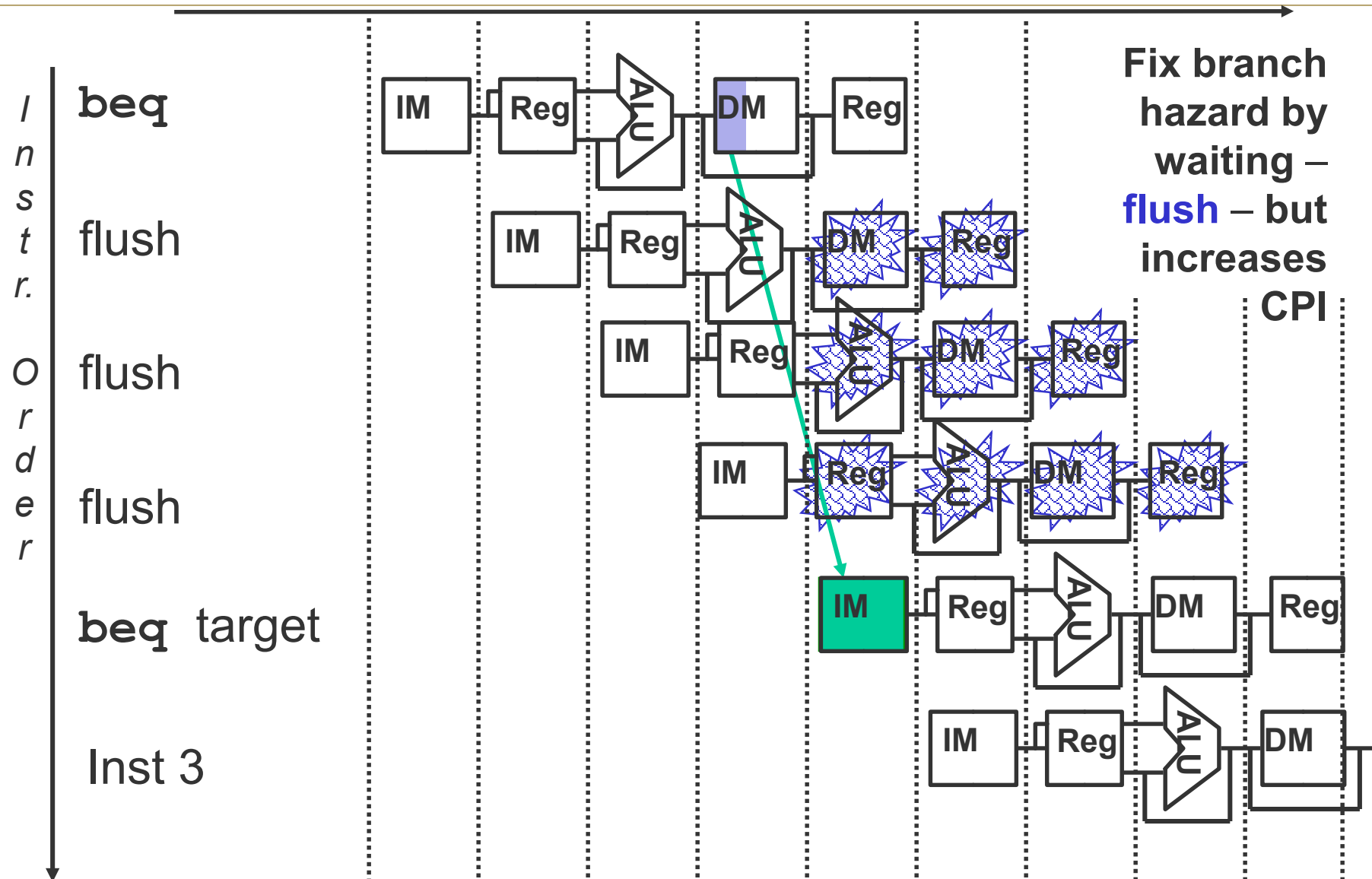
► Aka branch control hazards

- Caused by dependencies backward in time





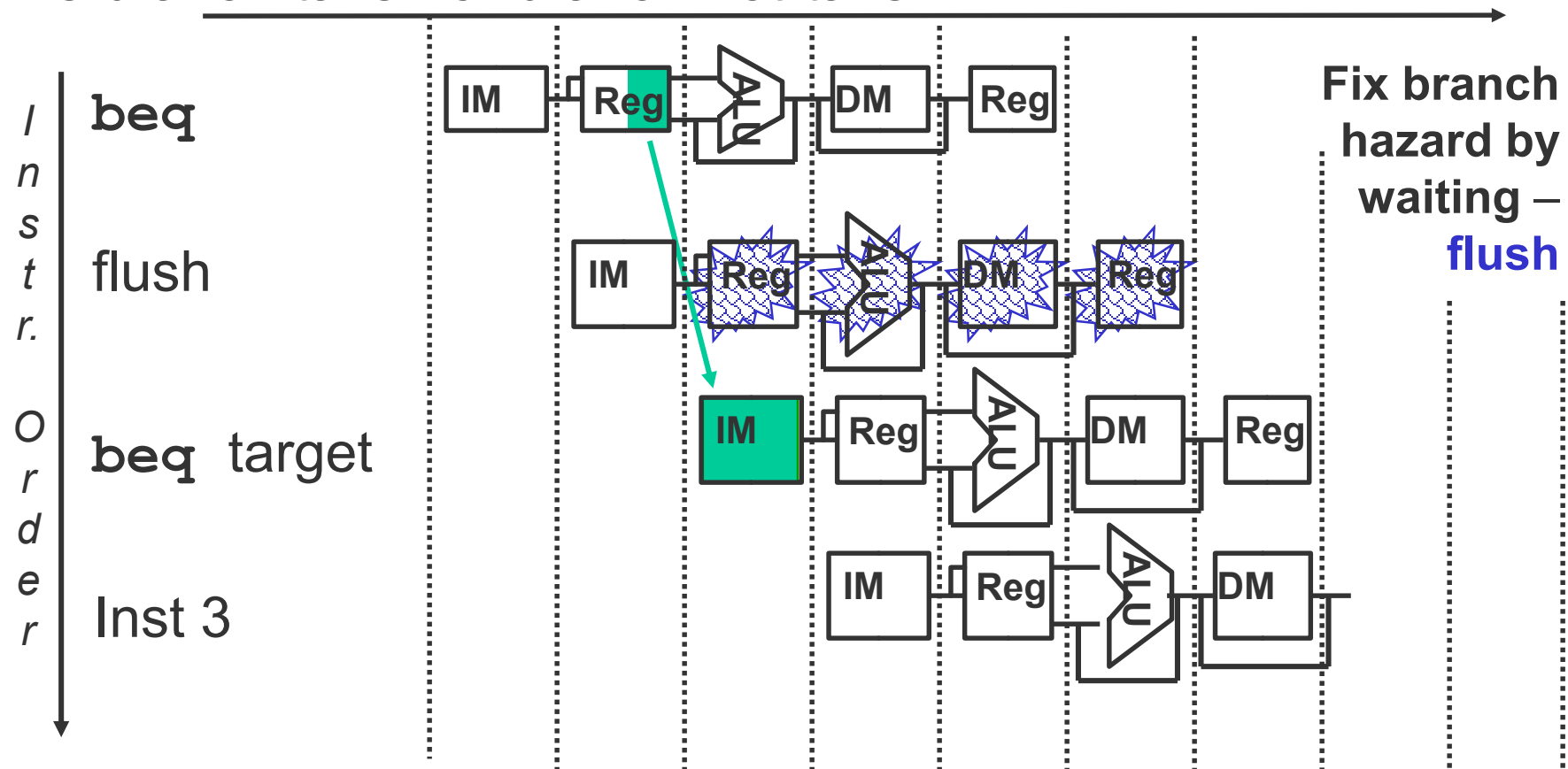
One Way to “Fix” a Branch Control Hazard





Another Way to “Fix” a Branch Control Hazard

- Move branch decision hardware back to as early in the pipeline as possible – i.e., during the decode cycle
 - Assume branch-taken or branch-not-taken





► Top-of-the-loop branching

- Branch taken at most once
- Works well with “predict branch not taken”

```
Loop: beq $1,$2,Out
      1st loop instr
      .
      .
      .
      last loop instr
      j Loop
Out:  fall out instr
```

► Bottom-of-the-loop branching

- Branch not taken at most once
- Works well with “predict branch taken”

```
Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

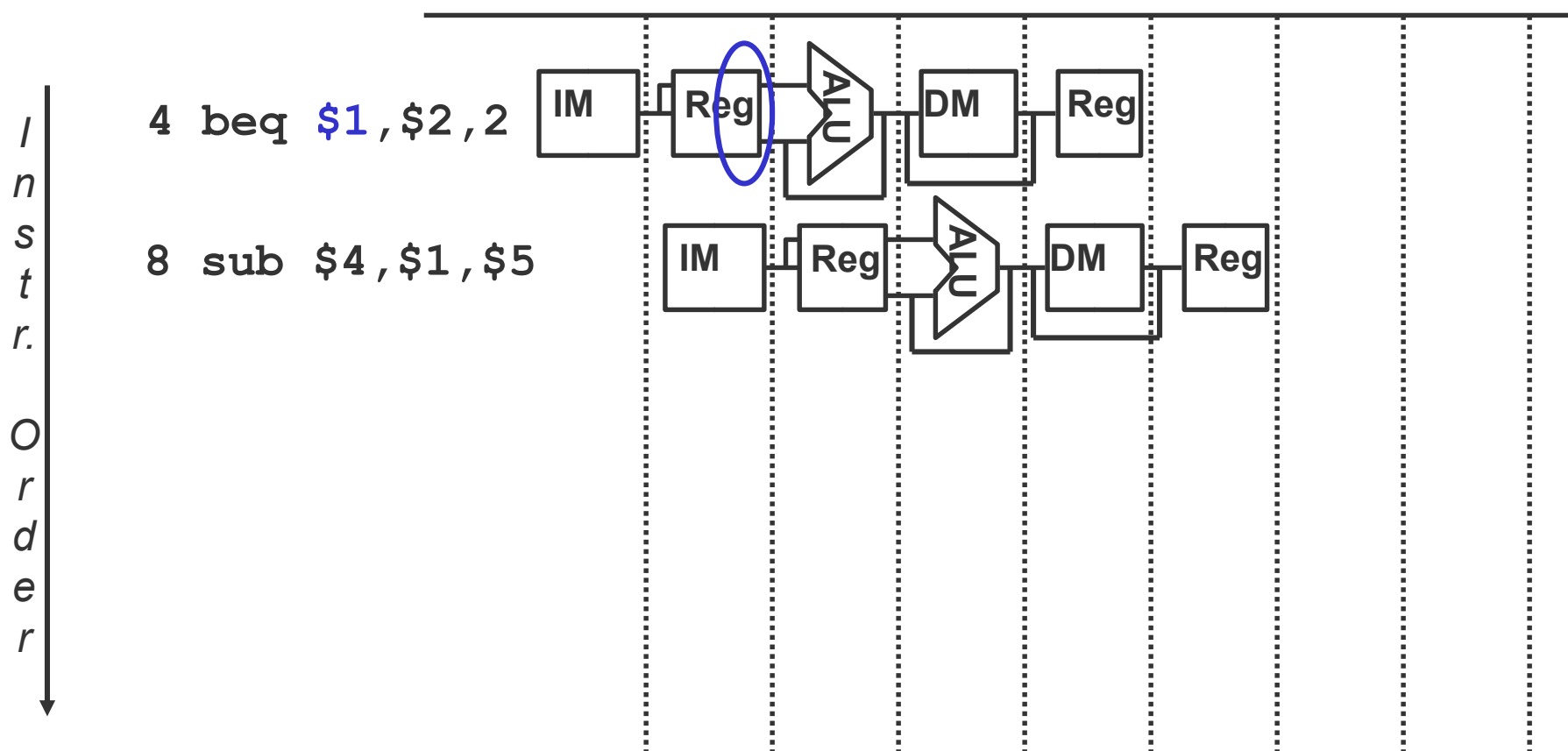


- ▶ Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome
- ▶ Predict “not taken”
 - Assumes branch will **not** be taken
 - Continues to fetch from the sequential instruction stream
 - Works well for “**top-of-the-loop**” branching
 - Pipeline stalls only when branch is taken
 - Ensure that flushed instructions haven’t changed the machine state
 - Automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
 - Restart the pipeline at the branch destination
- ▶ Predict “taken”
 - Assumes branch will be taken
 - **Requires one stall cycle** if branch decision hardware at ID stage
 - Possibly “cache” the address of the branch target instruction
 - Works well for “**bottom-of-the-loop**” branching



Flushing Needed with “Predict Taken”

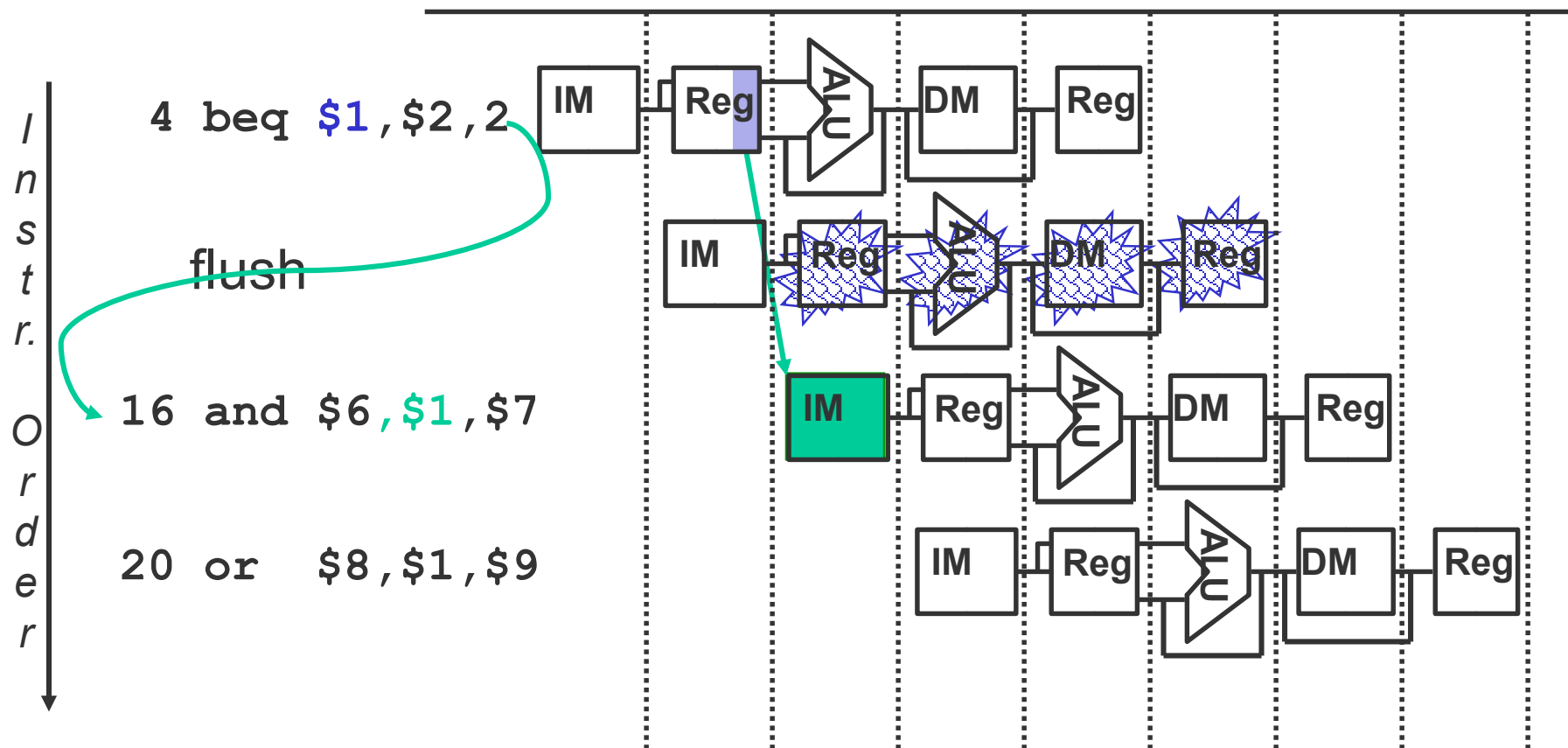
„Predict Taken“ requires a jump. A jump requires a flush cycle.
However, the guess „Predict Taken“ may still be wrong.





Flushing Needed with “Predict Taken”

„Predict Taken“ requires a jump. A jump requires a flush cycle.
However, the guess „Predict Taken“ may still be wrong.

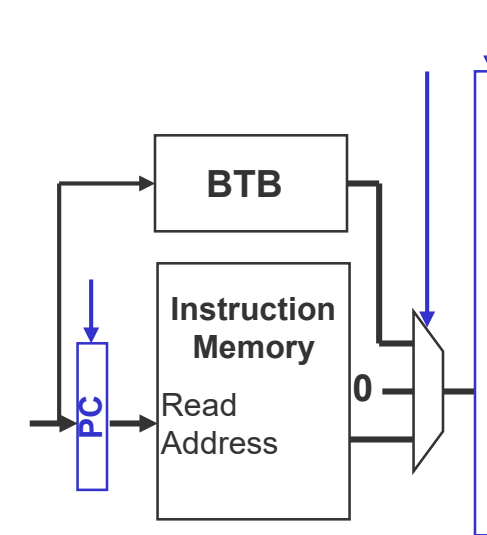




- ▶ Predicts branches dynamically using run-time information
 - Remembers last (one or two) outcomes of a particular inst addr
 - Assumes same outcome for next decision of that inst addr
- ▶ Branch prediction buffer (aka branch history table (BHT))
 - Remembers outcome of last one/two decisions (one/two-bit predictors)
 - In the IF stage addressed by the lower bits of the PC for prediction
 - Contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was executed
 - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
- ▶ When outcome of decision is available
 - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction
 - Modify the prediction bit(s) in the BHT accordingly
- ▶ Wrong predictions don't affect correctness, just performance
- ▶ A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

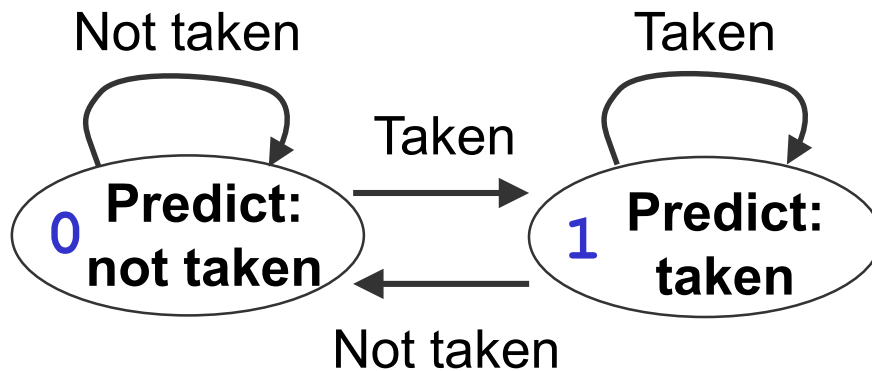


- ▶ The BHT predicts **when** a branch is taken, but does not tell **where** it's taken to!
- ▶ A **branch target buffer (BTB)** in the IF stage caches the branch target address
- ▶ But we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which “next” instruction will be loaded into IF/ID at the next clock edge
 - Would need a two read port instruction memory
 - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction
 - If the prediction is correct, stalls can be avoided no matter which direction they go





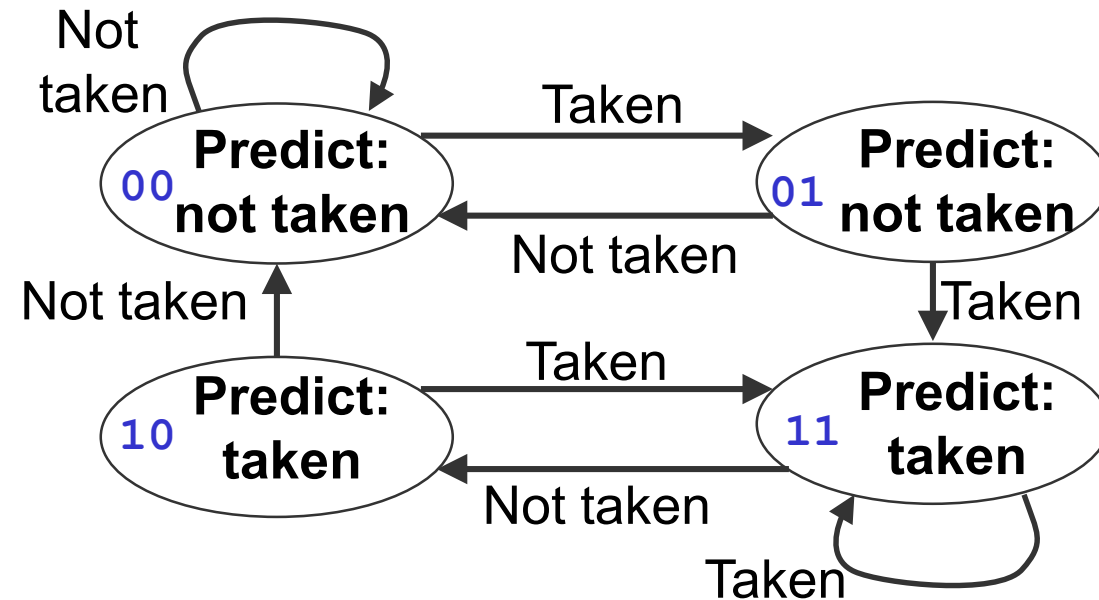
- ▶ Takes last decision outcome for prediction
- ▶ Predict_bit
 - Updated by each branch and used for prediction
 - 0: branch not taken; predict not taken
 - 1: branch taken; predict: taken



- ▶ Accuracy analysis
 - Assumption
 - Bottom-of-the-loop branching
 - Initial predict_bit = 0 (like after leaving the loop)
 - Branch taken at least once
 - Result
 - Misprediction at first and last time through the loop
 - Correct prediction otherwise
 - Example
 - 10 times through the loop
 - Branch taken first 9 times
 - Branch not taken last time
 - 20% wrong predictions



- ▶ Requires two other decisions to change prediction
- ▶ State encoding (x,y)
 - First bit x: prediction
 - Second bit y: last decision



- ▶ Accuracy analysis
 - Assumption
 - Bottom-of-the-loop branching
 - Initial predict_bits = 10 (like after leaving the loop)
 - Branch taken at least once
 - Result
 - Misprediction only last time through the loop
 - Correct prediction otherwise
 - Example
 - 10 times through the loop
 - 10% wrong predictions



► Exceptions

- Events requiring immediate attention by the processor
- Control passed from the user program to an exception handler of the OS

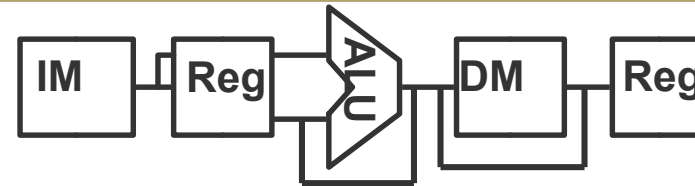
► Interrupts (exception type)

- Caused by **external events** – **asynchronous** to program execution
- **Instructions** in pipeline **can complete** before control passed to the OS interrupt handler
- Simply suspend and resume user program

► Traps (exception type)

- Caused by **internal events** – **synchronous** to program execution
- Offending instruction must be stopped midstream and control immediately passed to the OS trap handler to remedy the cause for the trap
- The offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

► Nomenclature for exceptions, interrupts, and traps not consistently used



Occurrence in stage(s)
of pipeline?

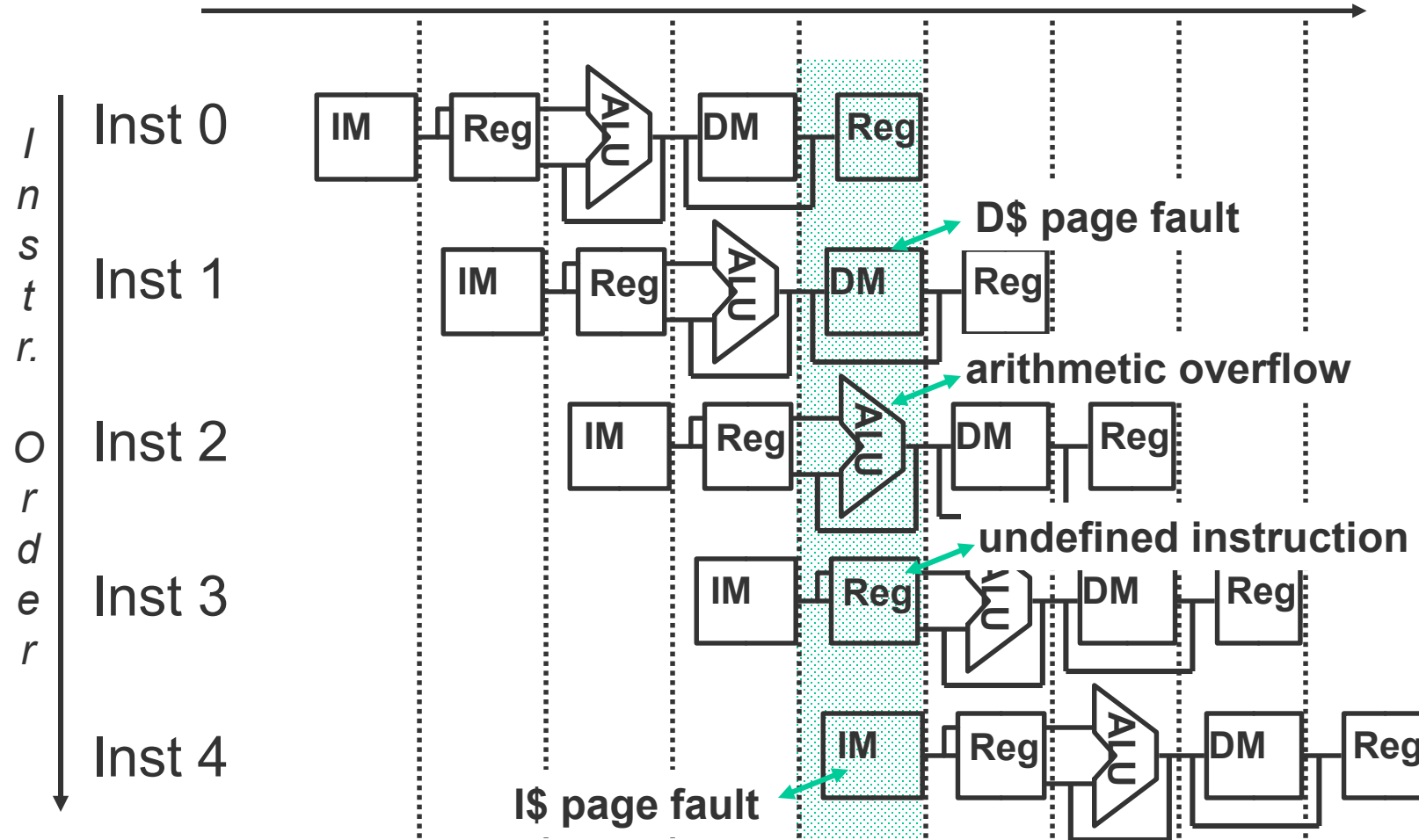
Synchronous?

- | | | |
|---|---------|----------|
| ▶ Arithmetic overflow | EX | yes |
| ▶ Undefined instruction | ID | yes |
| ▶ OS service request (e.g. TLB or page fault) | IF, MEM | yes |
| ▶ I/O device request | any | no |
| ▶ Hardware malfunction | any | yes / no |
| ▶ Beware that multiple exceptions can occur simultaneously in a <i>single</i> clock cycle | | |



Multiple Simultaneous Exceptions

- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first





▶ Cause register

- Records the exception type

▶ EPC register (Exception PC)

- Records the address of the offending instruction

▶ A way to flush the offending instruction and the ones that follow it

▶ A way to load the PC with the address of the exception handler

- Expand the PC input mux where the new input is hardwired to the exception handler address
- e.g., 8000 0180hex for arithmetic overflow

▶ The software (OS) looks at the cause of the exception and “deals” with it



► Pipelining

- Used by all modern processors for performance
- Doesn't reduce latency of single instruction
- Increases throughput in terms of instruction per time
- May lead to hazards
- Complicates processor structure & exception handling

► Clock rate

- Limited by slowest pipeline stage
- Pipeline allows for faster clock rate
- Speedup by faster clock rate if pipeline always filled

► Resolution of hazards

- Structural hazards
 - Design pipeline correctly
- Data hazards
 - Stalling
 - Forwarding (hardware support)
- Control hazards
 - Put branch decision hardware in an early stage of the pipeline (hardware support)
 - Pipeline stalling & flushing
 - Static and dynamic branch prediction (hardware support)

► Pipelining efficiency reduced by

- Time to “fill” and “drain” the pipeline (esp. for short code)
- Stalling & pipeline flushing for hazards