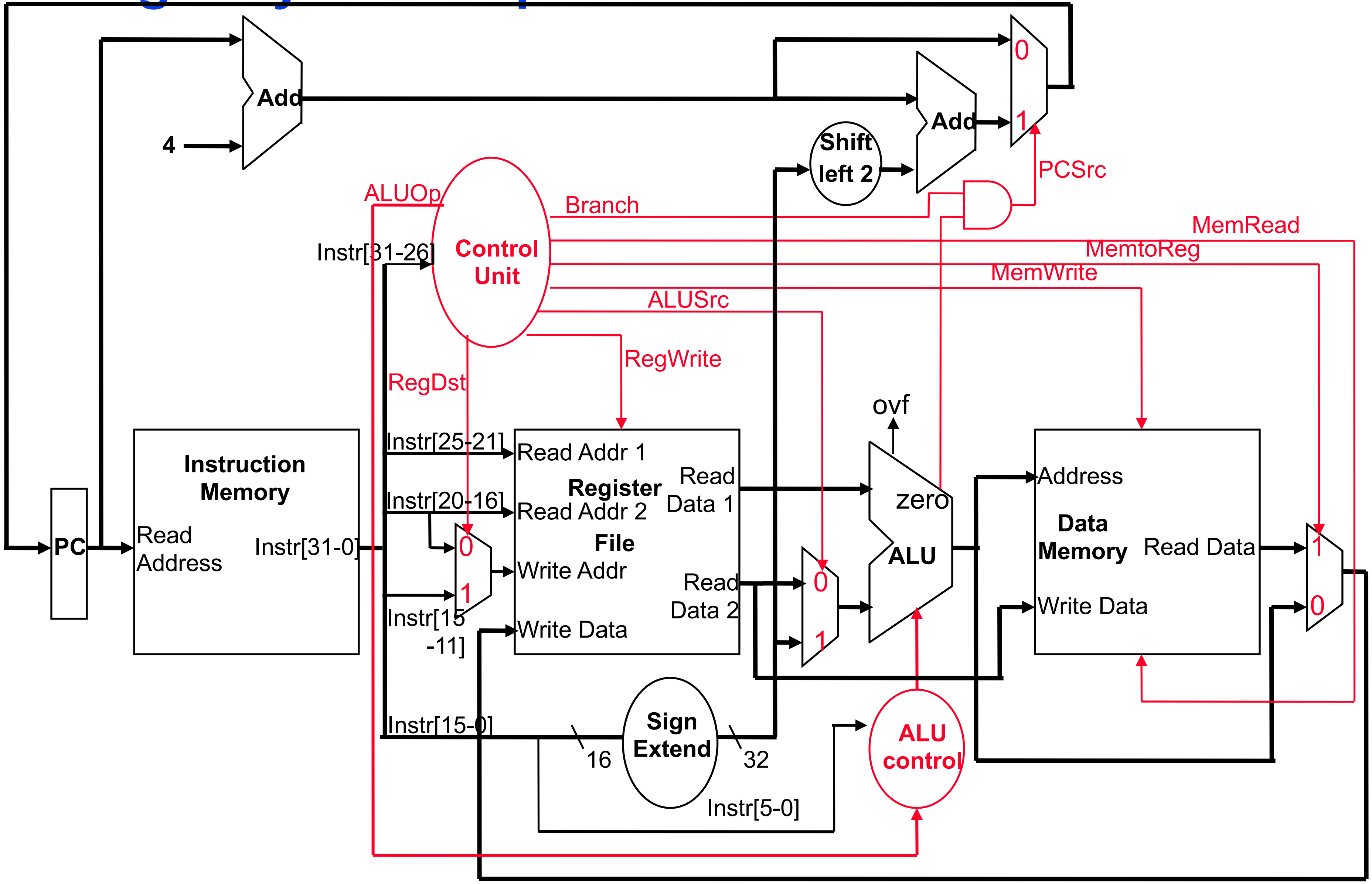
CSL7070: Computer Architecture
Lecture 9, 10th March 2023

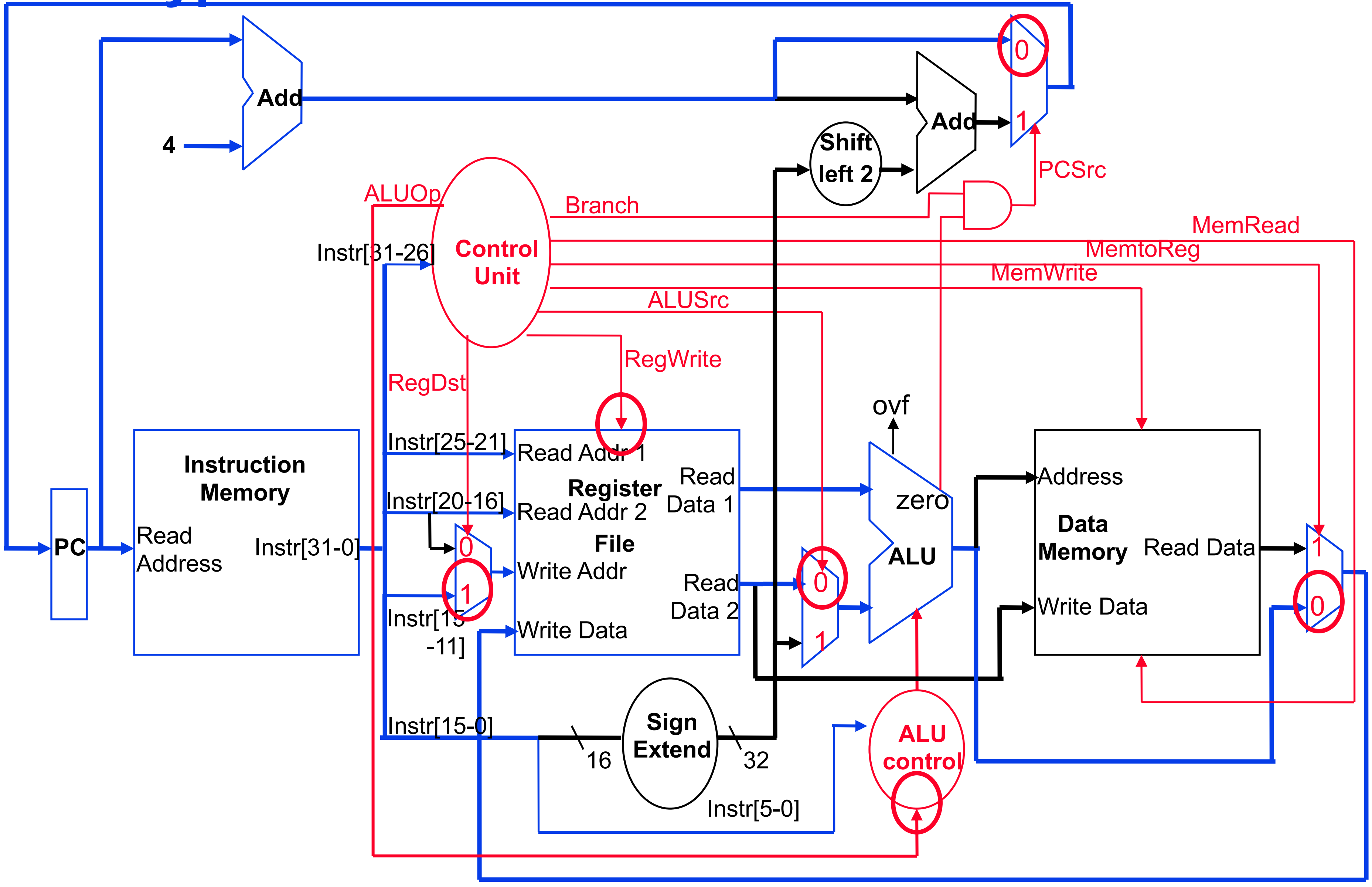
Dip Sankar Banerjee

Indian Institute of Technology, Jodhpur
January-April 2023

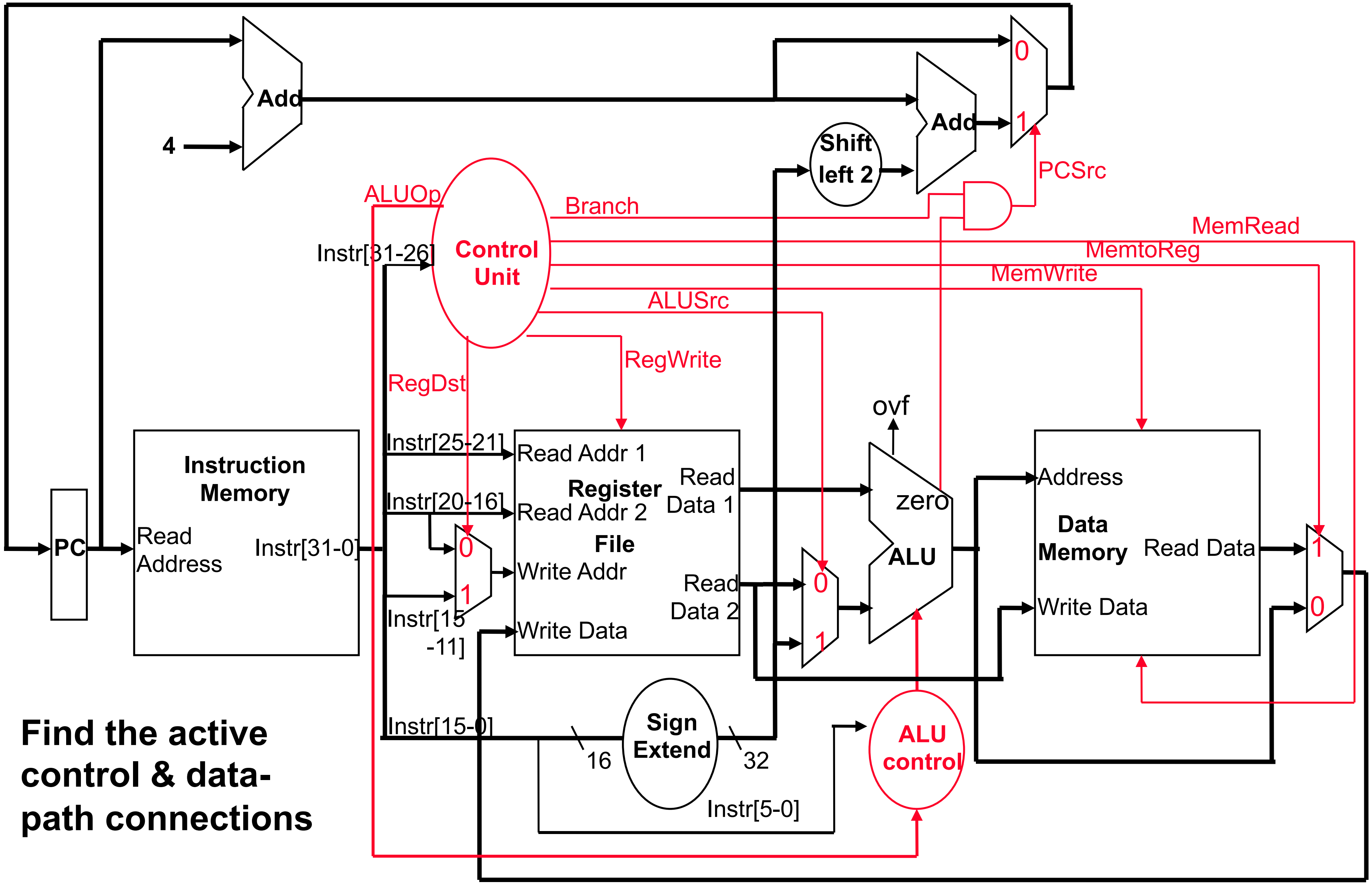
Single Cycle Datapath with Control Unit



R-type Instruction Data/Control Flow

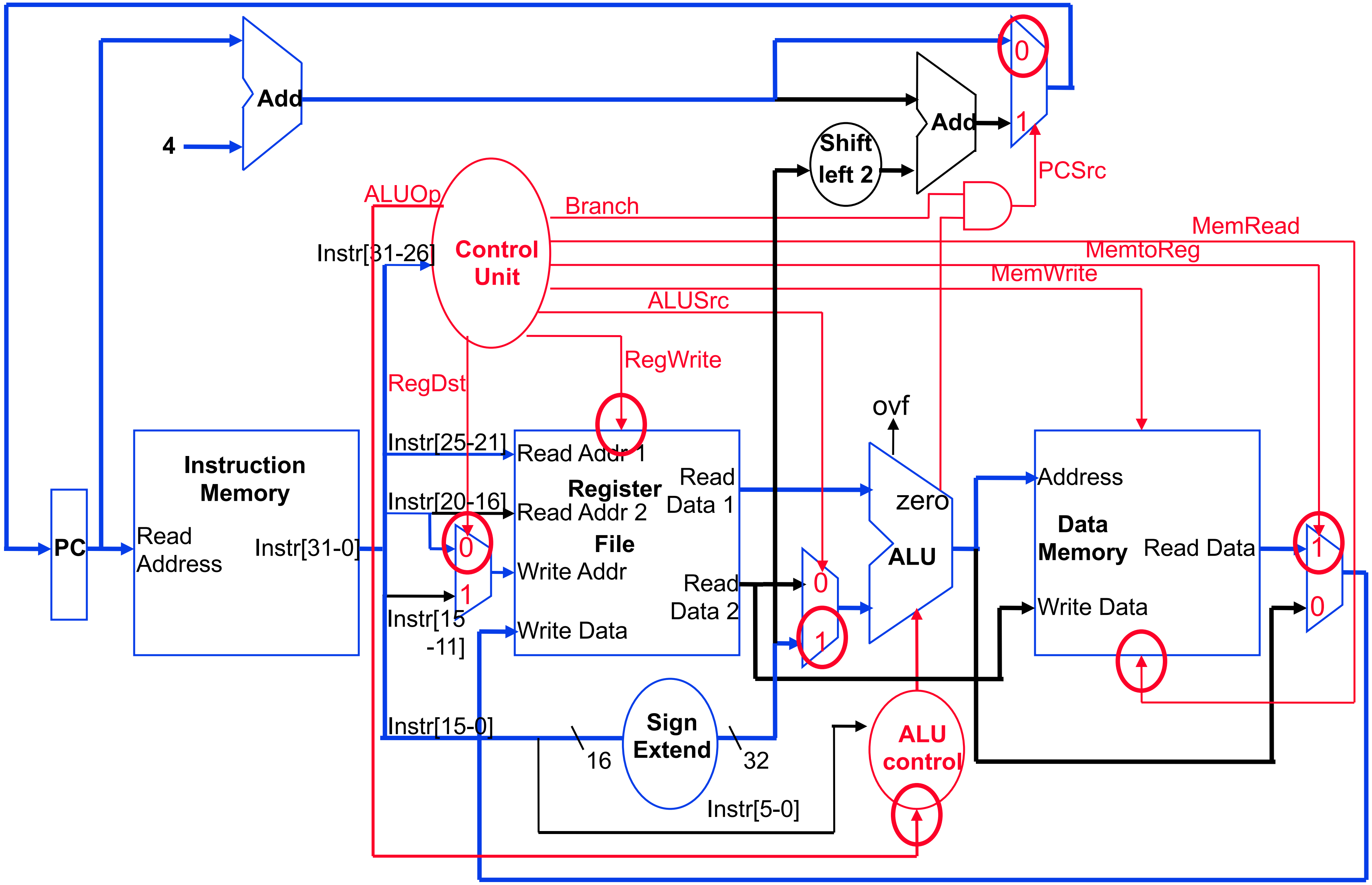


Load Word Instruction Data/Control Flow

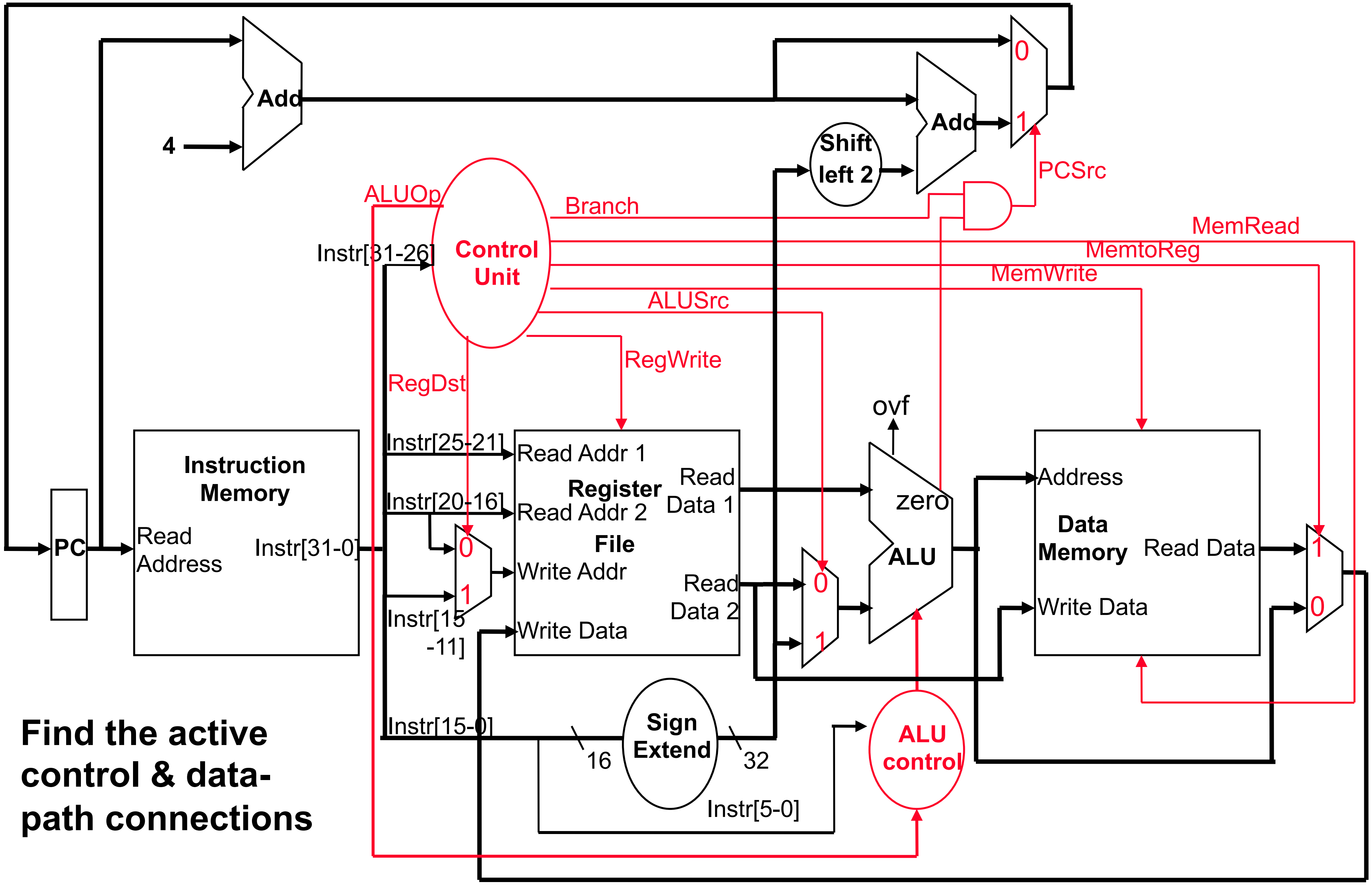


Find the active control & data-path connections

Load Word Instruction Data/Control Flow

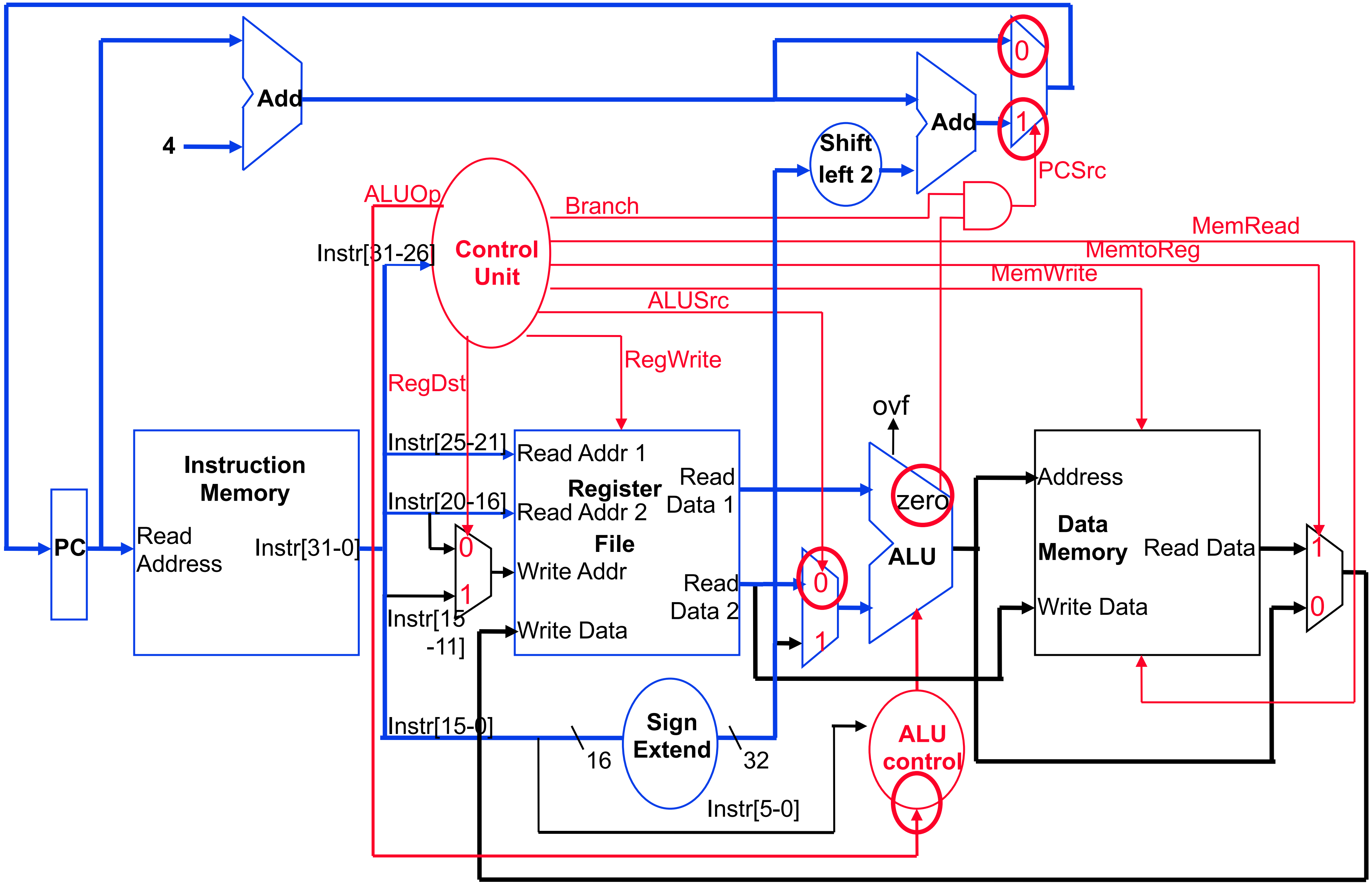


Branch Instruction Data/Control Flow

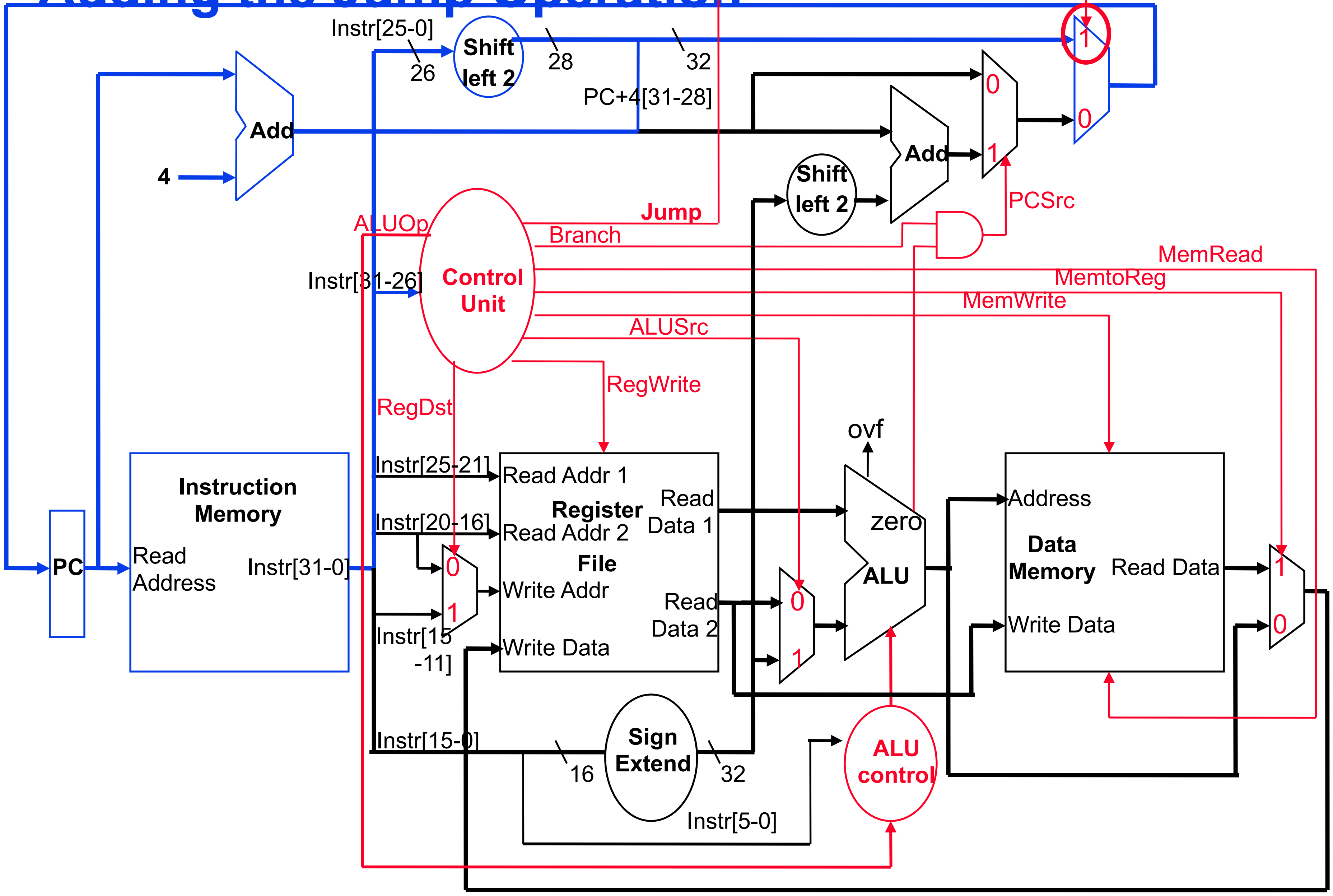


Find the active control & data-path connections

Branch Instruction Data/Control Flow



Adding the Jump Operation



Instruction Times (Critical Paths)

- 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) but with:
  - 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						
load						
store						
beq						
jump						

Instruction Critical Paths

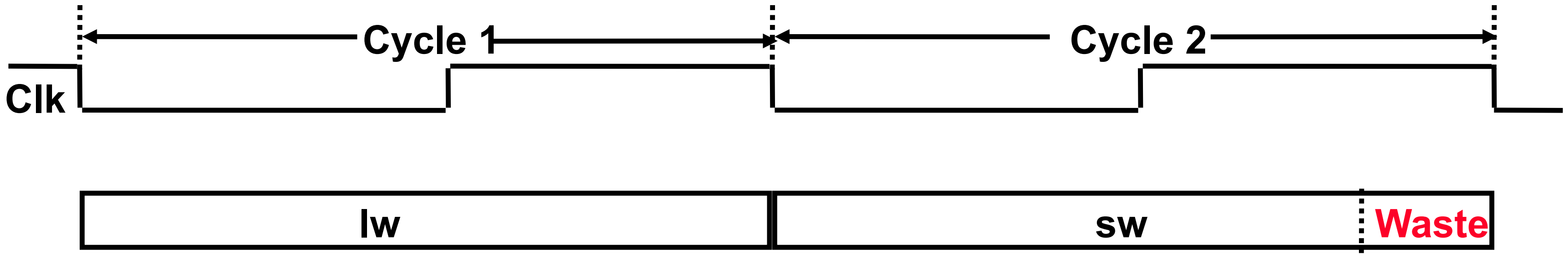
❑ What is the clock cycle time (assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times) but with:

- 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



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

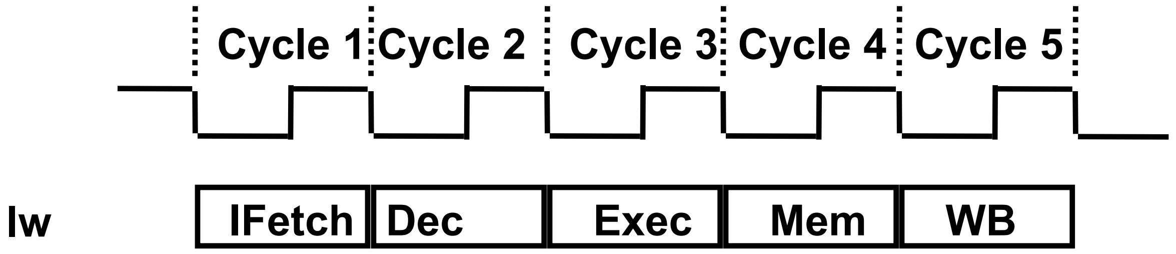
but

- Is simple and easy to understand

How Can We Make It Faster?

- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – all modern processors are pipelined for performance
  - Remember *the* performance equation:
    CPU time = CPI * CC * IC
- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CC is nearly five times faster
- Fetch (and execute) more than one instruction at a time
  - Superscalar processing – stay tuned

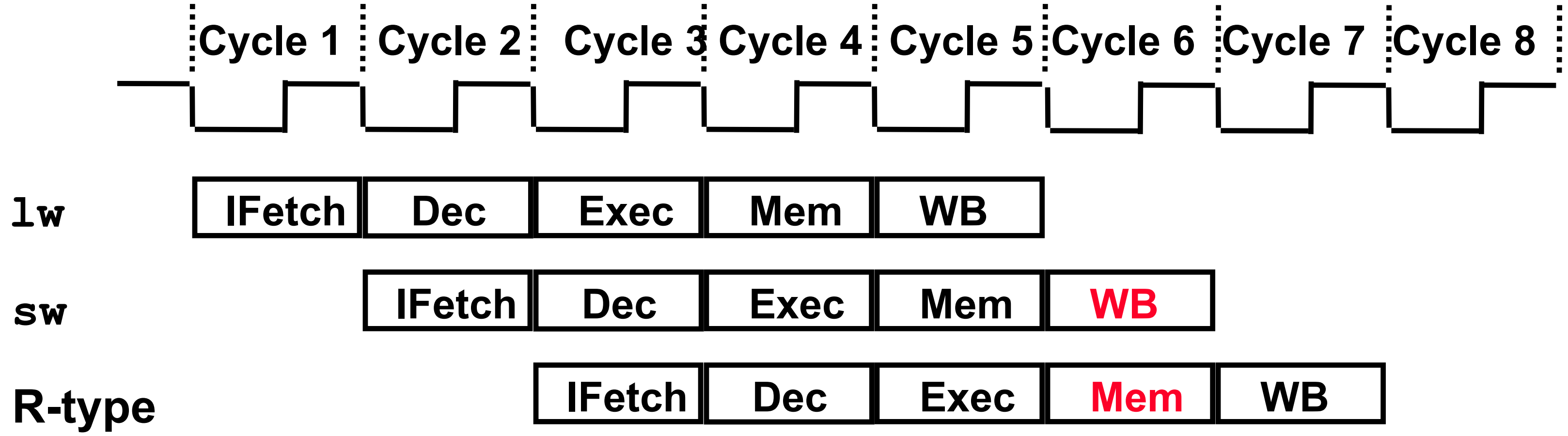
The Five Stages of Load Instruction



- 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

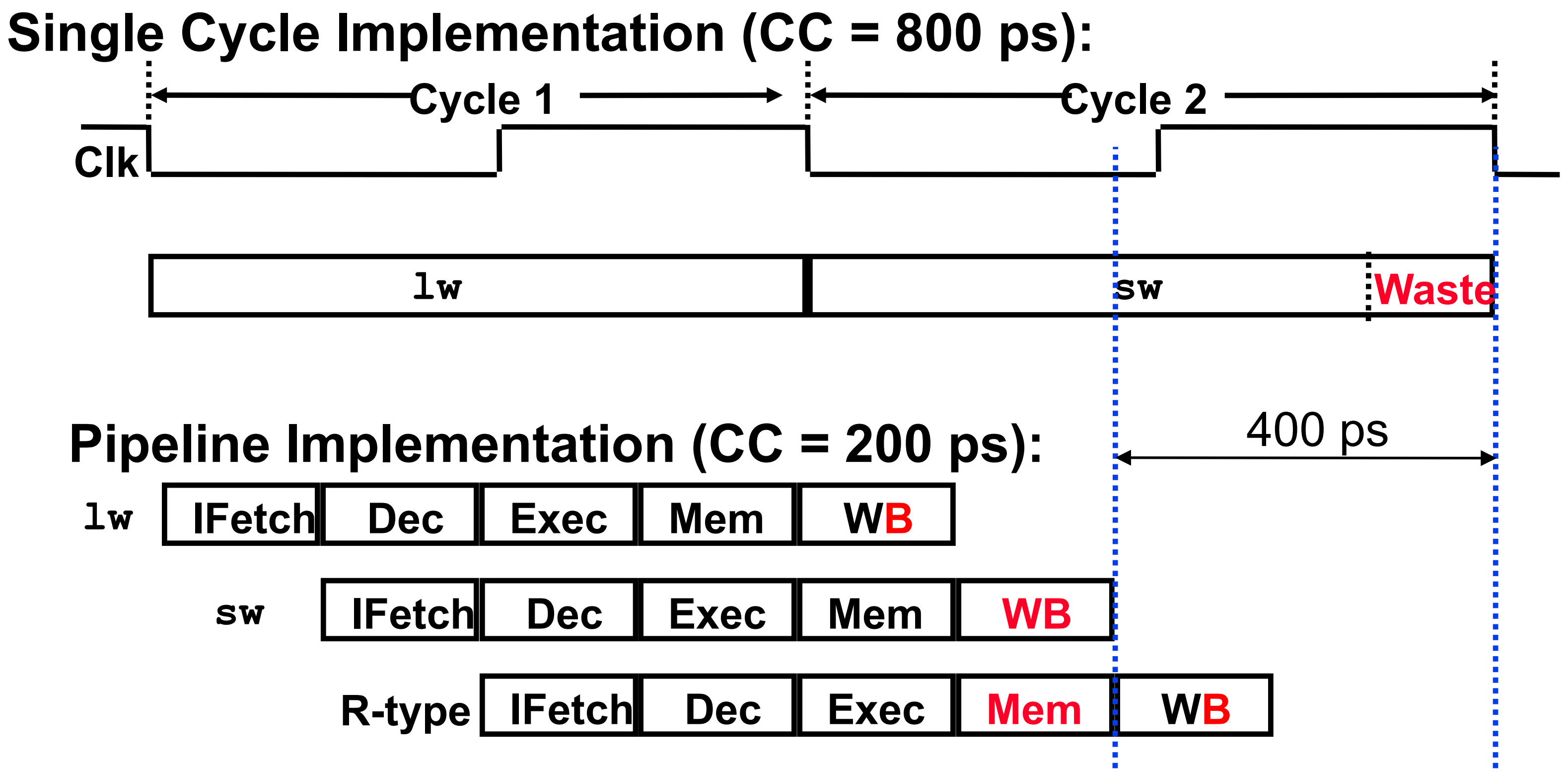
A Pipelined RISC Processor

- 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 versus Pipeline



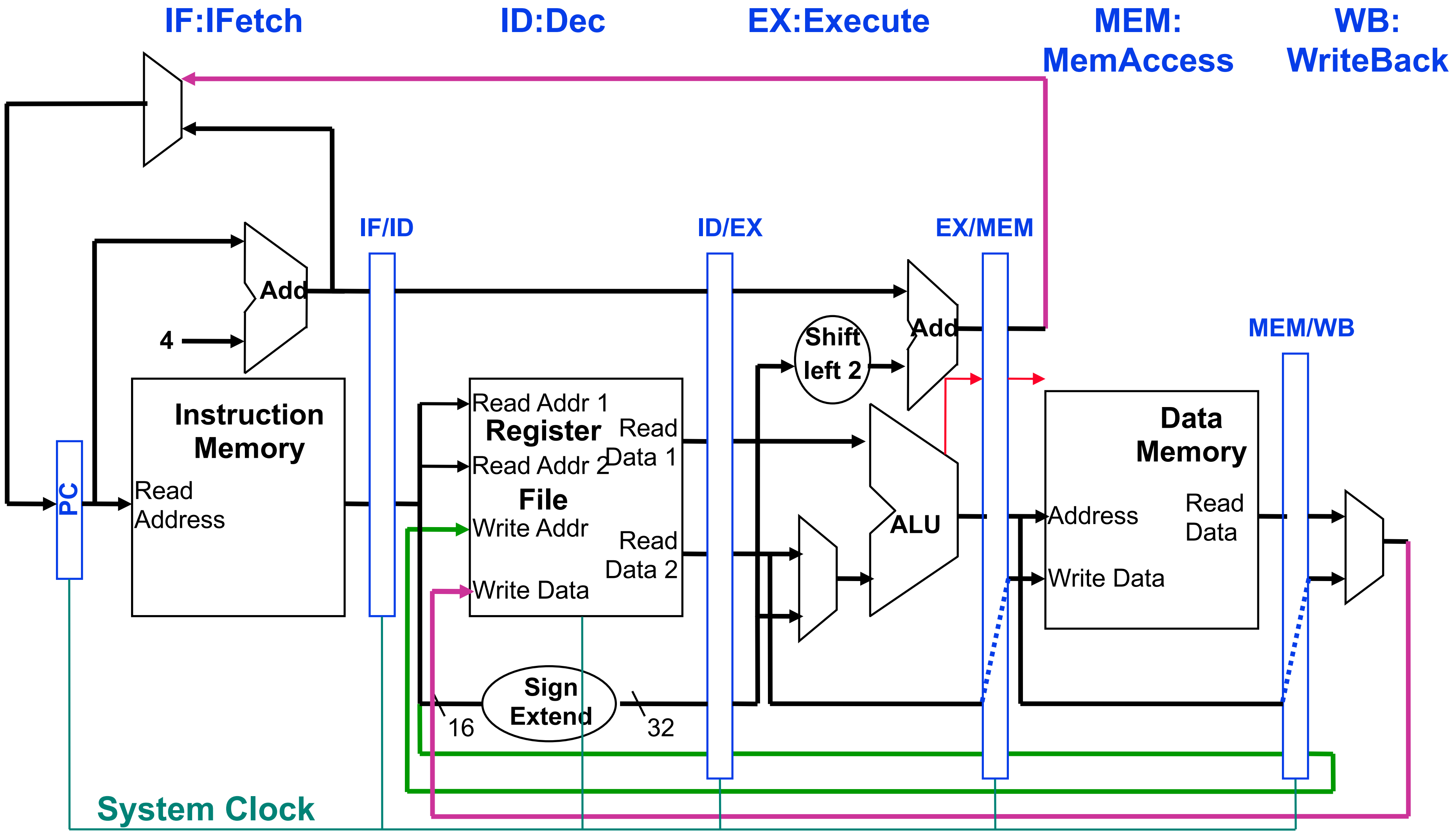
- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

Pipelining the RISC ISA

- 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 (only 3) with symmetry 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 a single data transfer takes only one data memory access

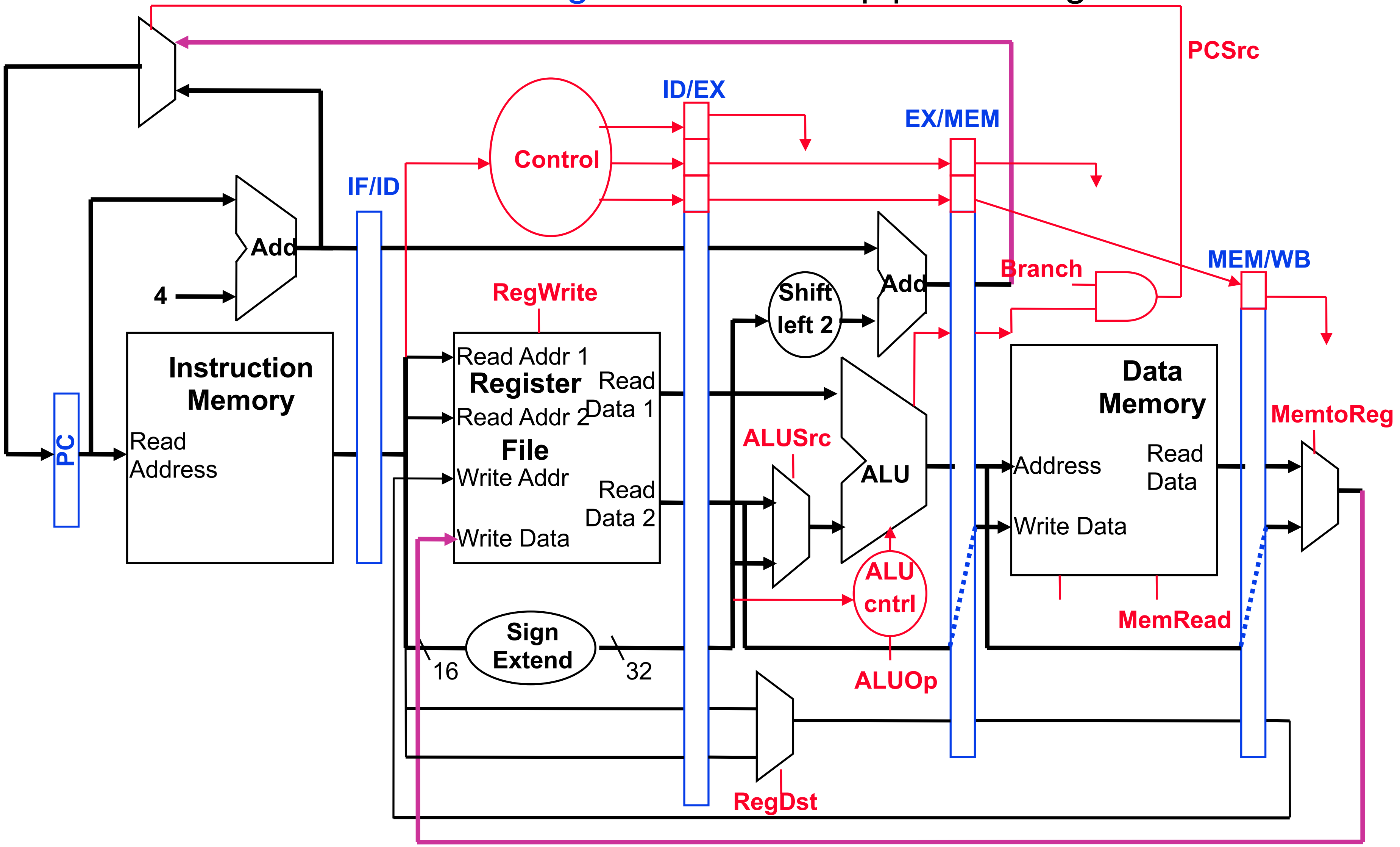
RISC Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them



RISC Pipeline Control Path Modifications

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



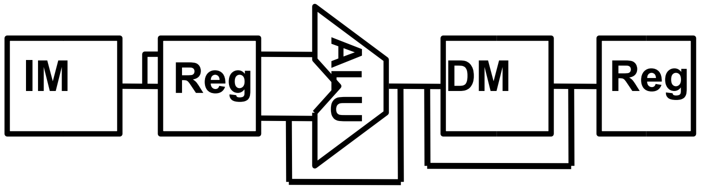
Pipeline Control

- IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)

- ID Stage: no optional control signals to set

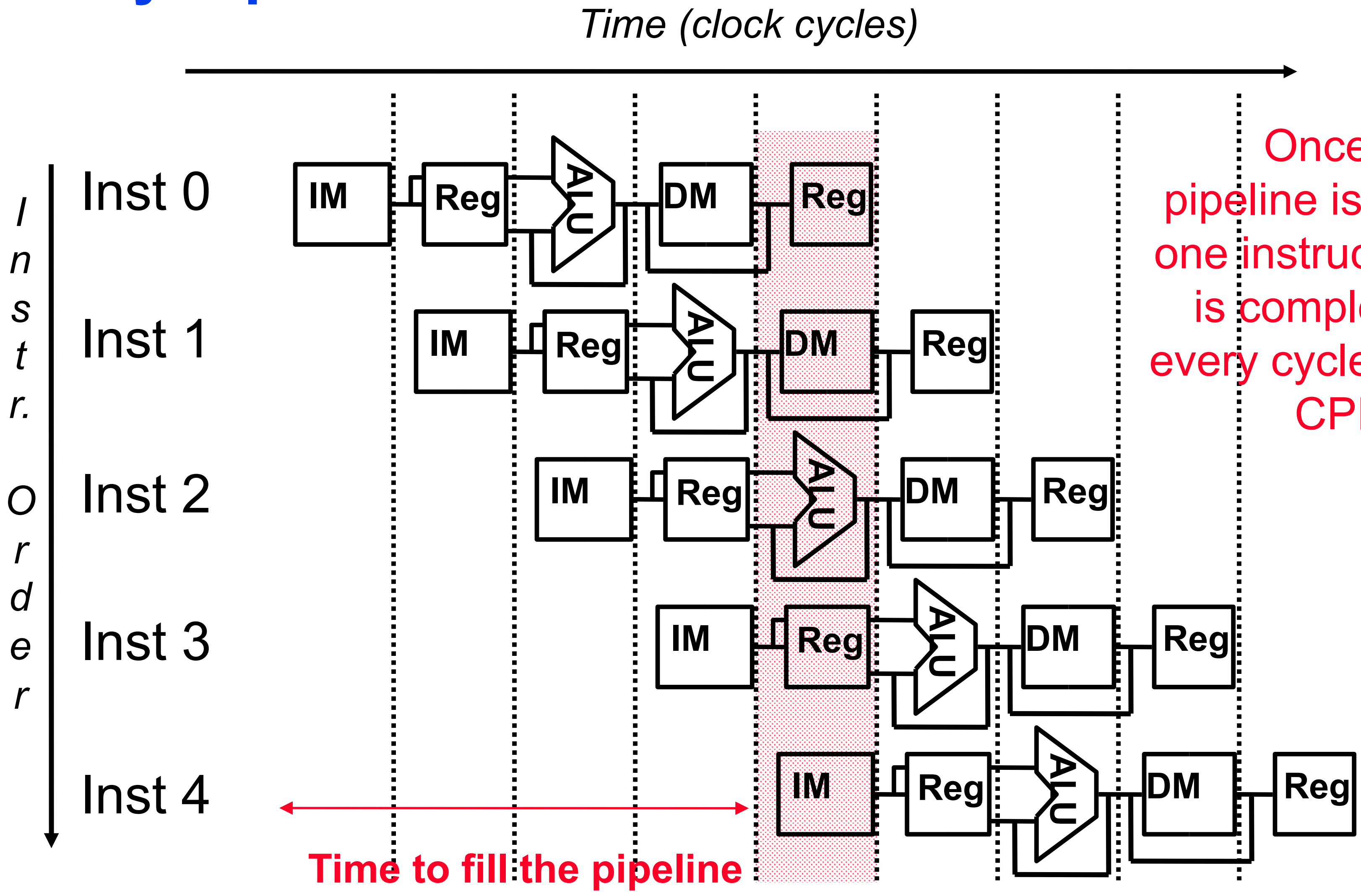
	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

Graphically Representing RISC 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?

Why Pipeline? For Performance!



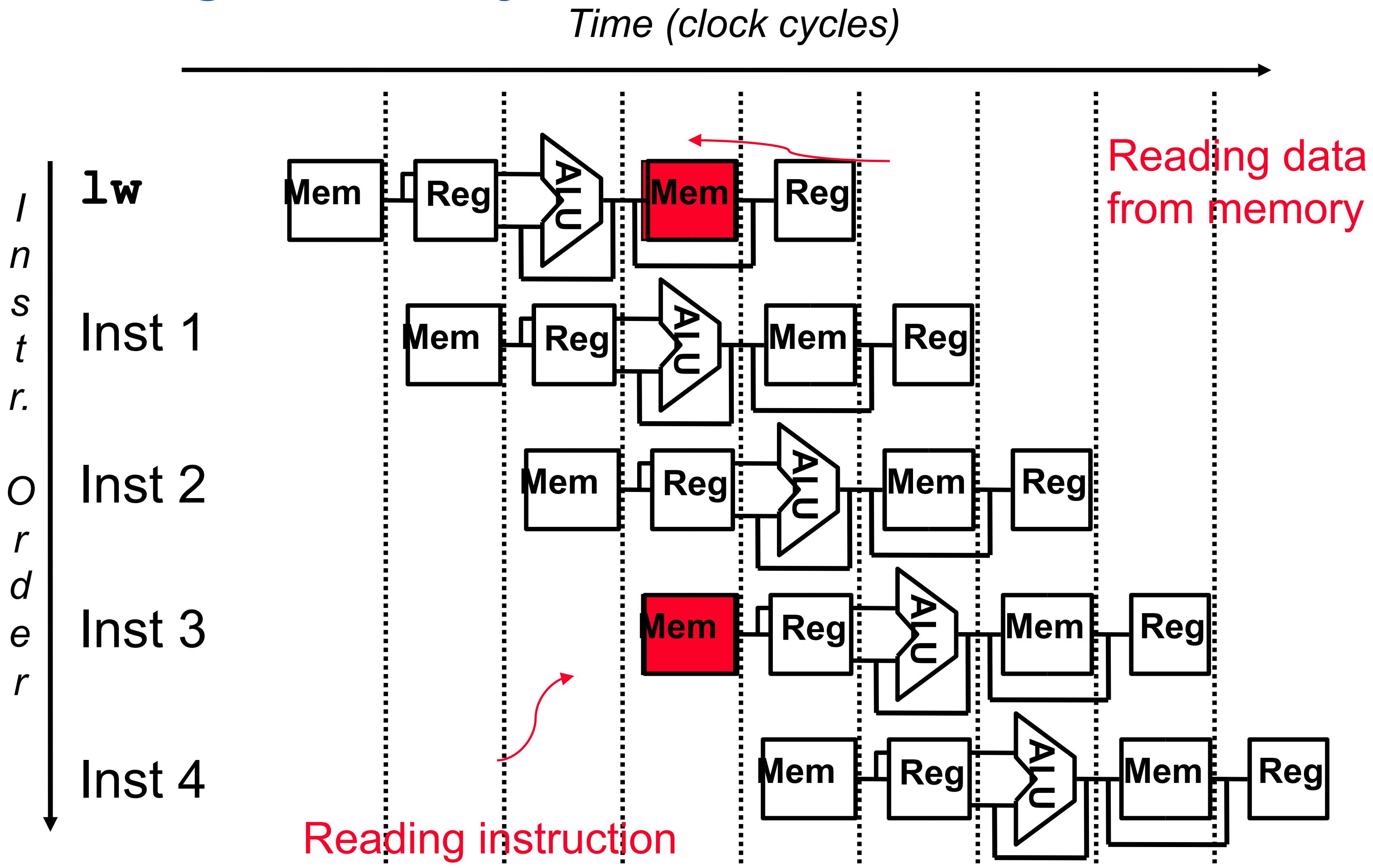
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

Can Pipelining Get Us Into Trouble?

- 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

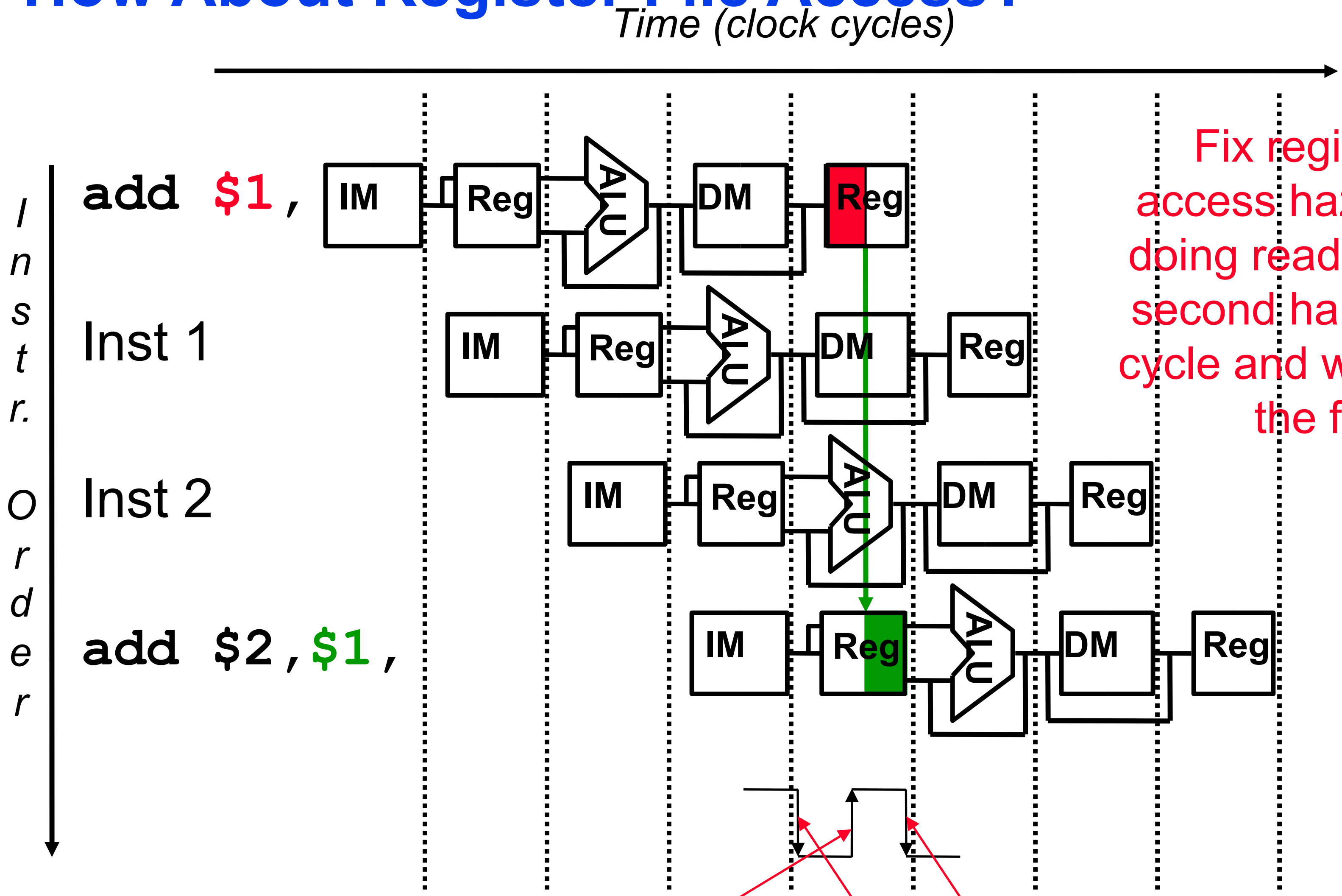
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

A Single Memory Would Be a Structural Hazard



- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

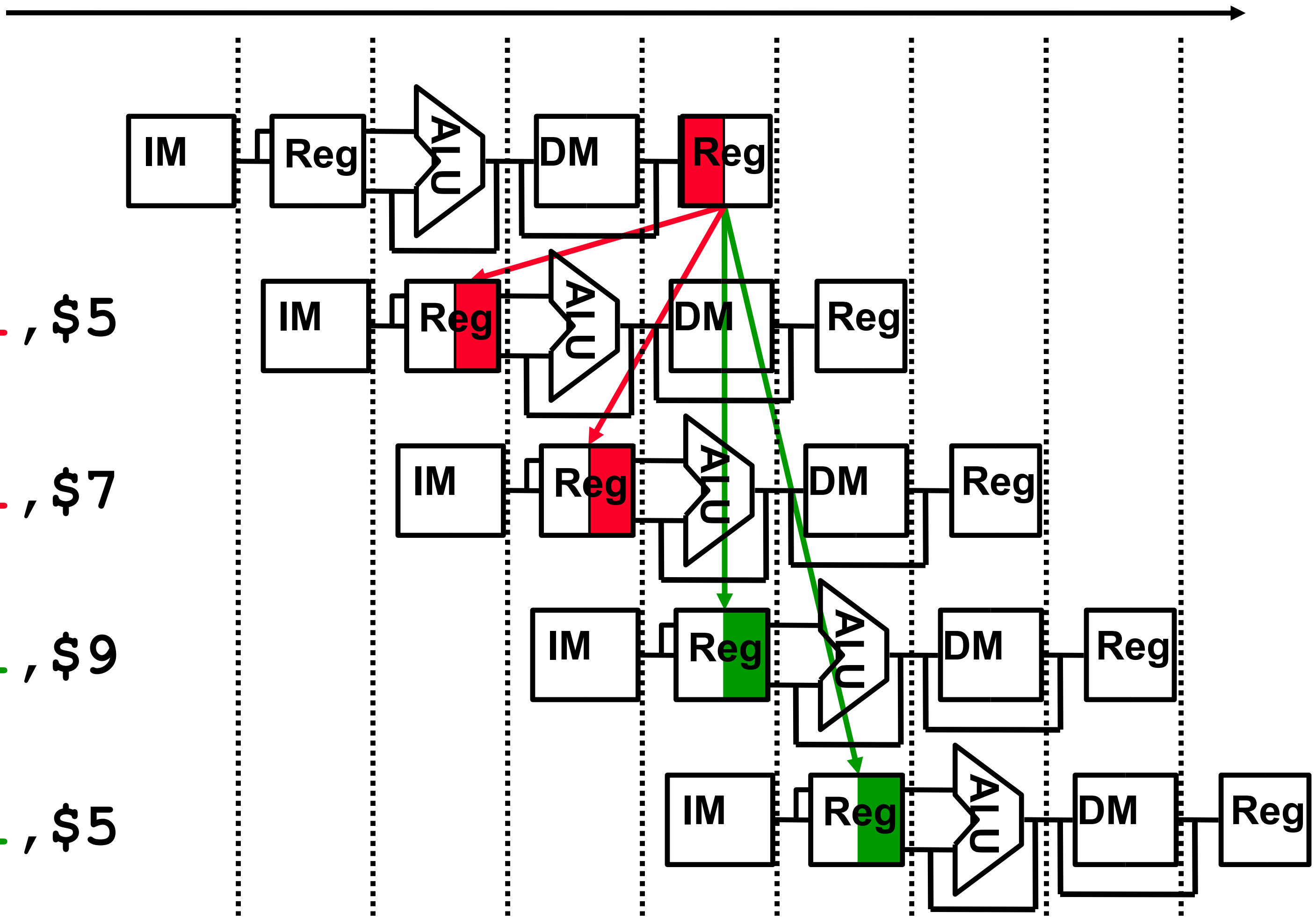
clock edge that controls register writing

clock edge that controls loading of pipeline state registers

Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

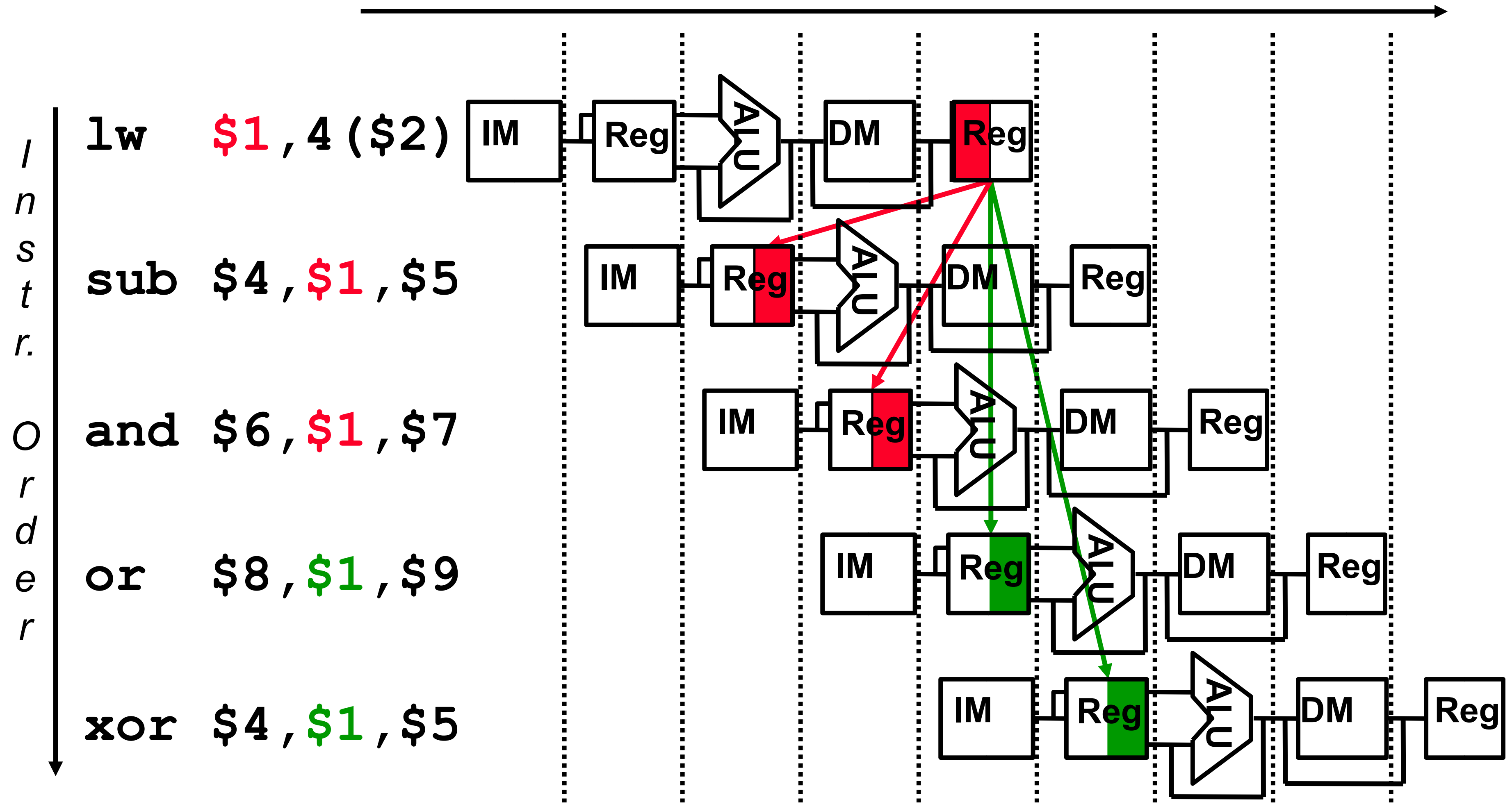
```
add $1,
sub $4,$1,$5
and $6,$1,$7
or  $8,$1,$9
xor $4,$1,$5
```



- Read after write data hazard (RAW in the code)

Loads Can Cause Data Hazards

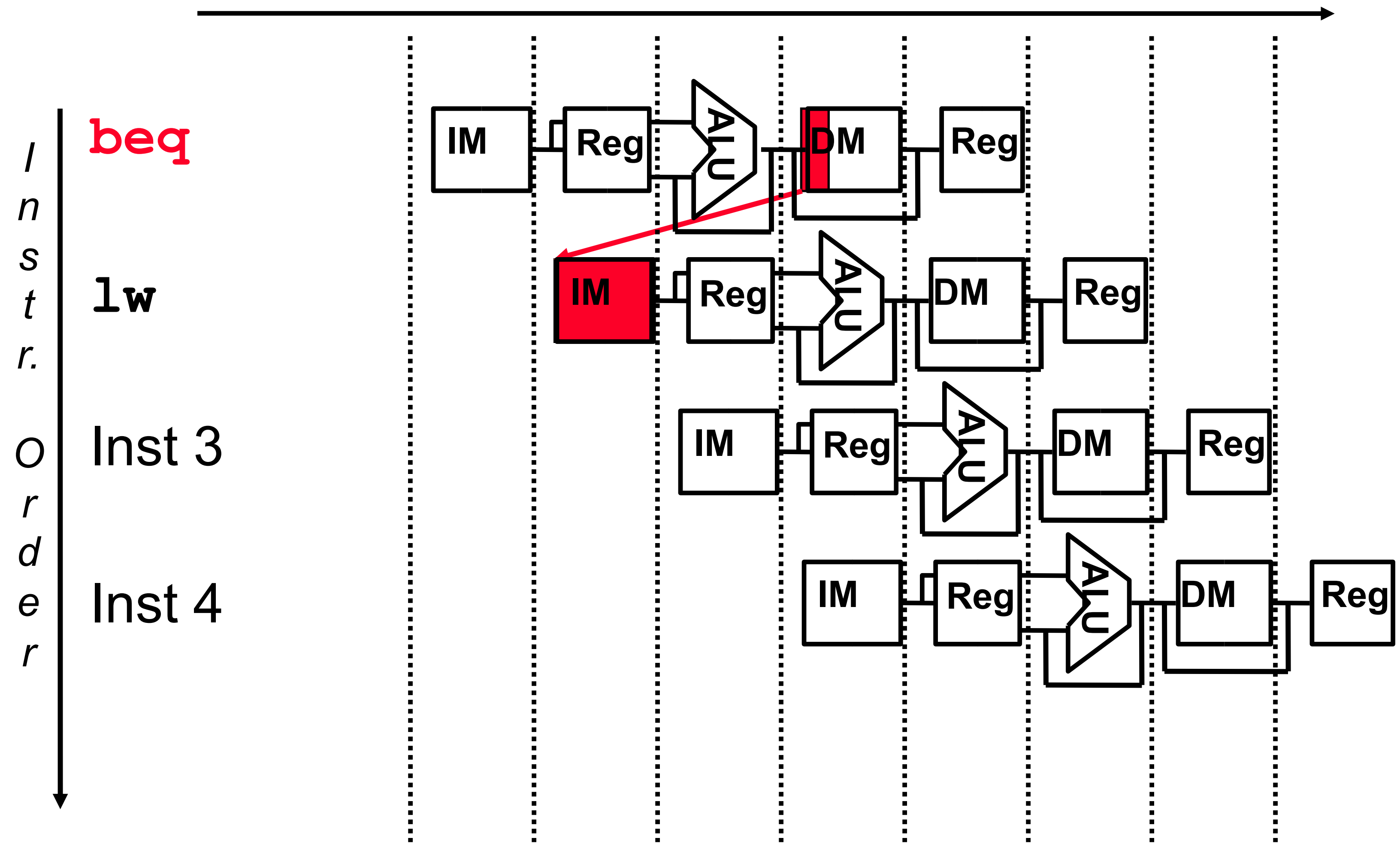
- ❑Dependencies backward in time cause hazards



- ❑Load-use data hazard

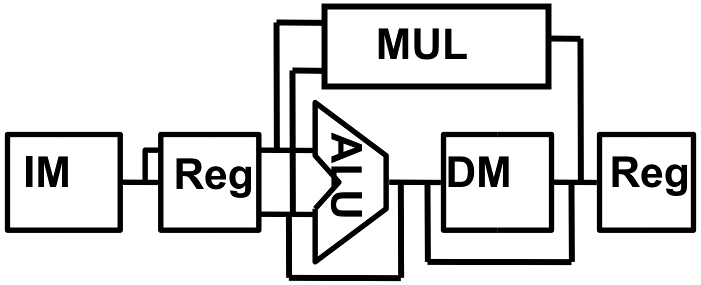
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards

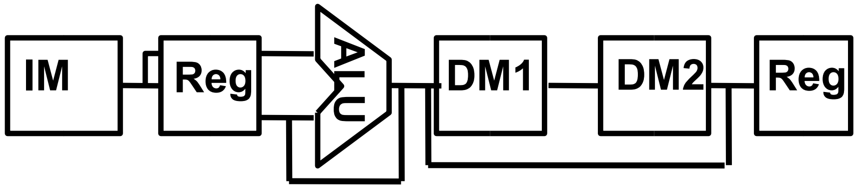


Other Pipeline Structures Are Possible

- 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)

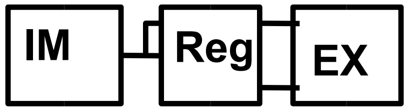


- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Other Sample Pipeline Alternatives

- ARM7

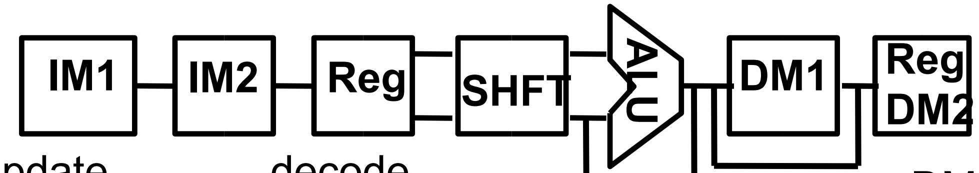


PC update
IM access

decode
reg
access

ALU op
DM access
shift/rotate
commit result
(write back)

- XScale



PC update
BTB access
start IM access

IM access

decode
reg 1 access

shift/rotate
reg 2 access

ALU op

start DM access
exception

DM write
reg write

Summary

- All modern day processors use pipelining
- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and fast a CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to “fill” pipeline and time to “drain” it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)