Computer Architecture
Fall 2022

Lecture 03: RISC-V Basic Architecture

Jie Zhang

jiez@pku.edu.cn

CHASELab

Computer Hardware And System Evolution Laboratory

PEKING UNIVERSITY

Question review

- Edge case study of instruction *LR.d, SC.d*
- Case 1: LR/SC addresses don't match – can this succeed?

```
lr.w t0,(a0)
sc.w t1,a1,(a3)
```

- Case 2: unbalanced *LR.d, SC.d*

```
lr.w t0,(a0)
sc.w t1,a1,(a0)
addi a1,a1,1
sc.w t2,a1,(a0)
```

- Case 3: multiple LRs, SCs from one core

```
lr.w t0,(a0)
lr.w t1,(a2)
sc.w t2,a1,(a0)
sc.w t3,a1,(a2)
```

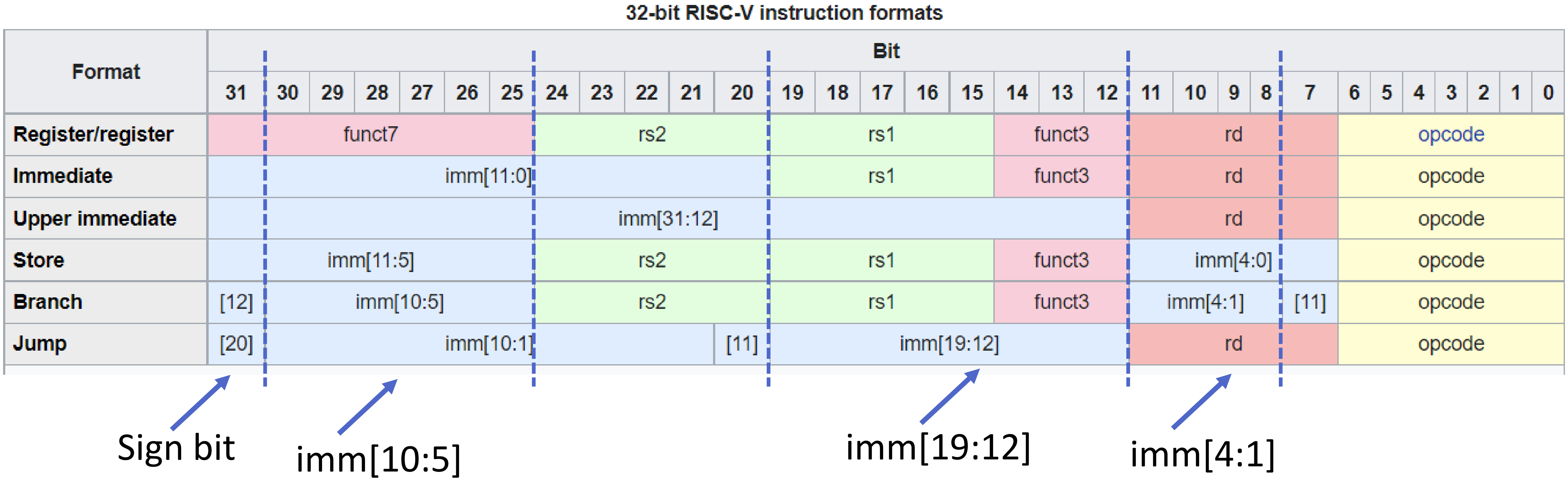
Note that:

- *the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written.*
- *Regardless of success or failure, executing an SC.W instruction invalidates any reservation held by this hart.*

https://five-embeddev.com/riscv-isa-manual/latest/a.html#

Question review

- “Wired” orders of immediate



32-bit RISC-V instruction formats

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

Sign bit

imm[10:5]

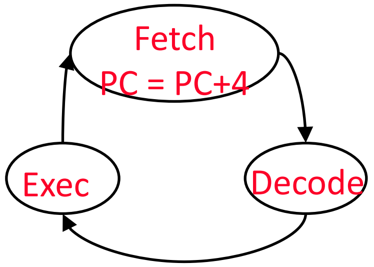
imm[19:12]

imm[4:1]

https://five-embeddev.com/riscv-isa-manual/latest/a.html#

The Processor: Datapath & Control

- Our implementation is simplified
  - memory-reference instructions: **ld, sd**
  - arithmetic-logical instructions: **add, sub, and, or**
  - control flow instructions: **beq**
- 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 use the ALU after reading the registers



How? memory-reference? arithmetic? control flow?

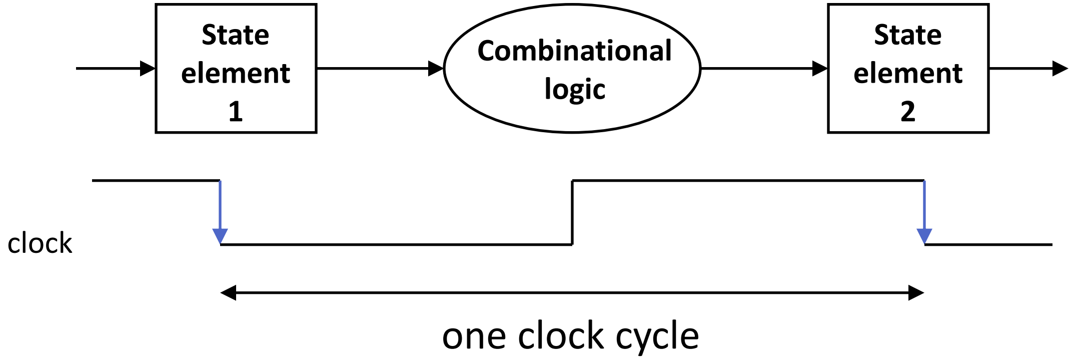
Clocking Methodologies

The clocking methodology defines when signals can be read and when they are written

- An edge-triggered methodology

Typical execution

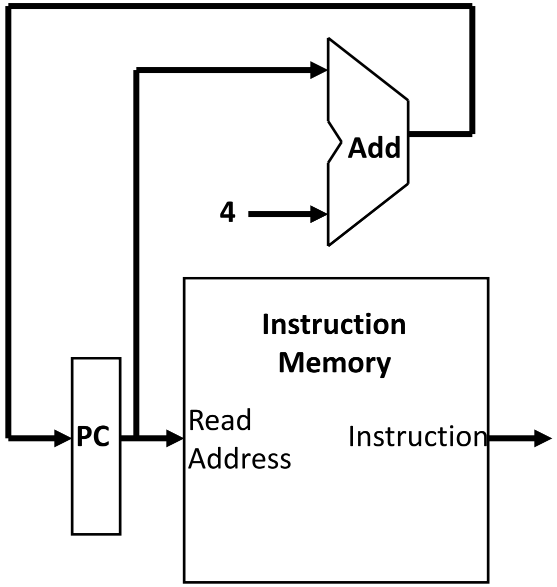
- read contents of state elements
- send values through combinational logic
- write results to one or more state elements



- Assume 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

Fetching Instructions

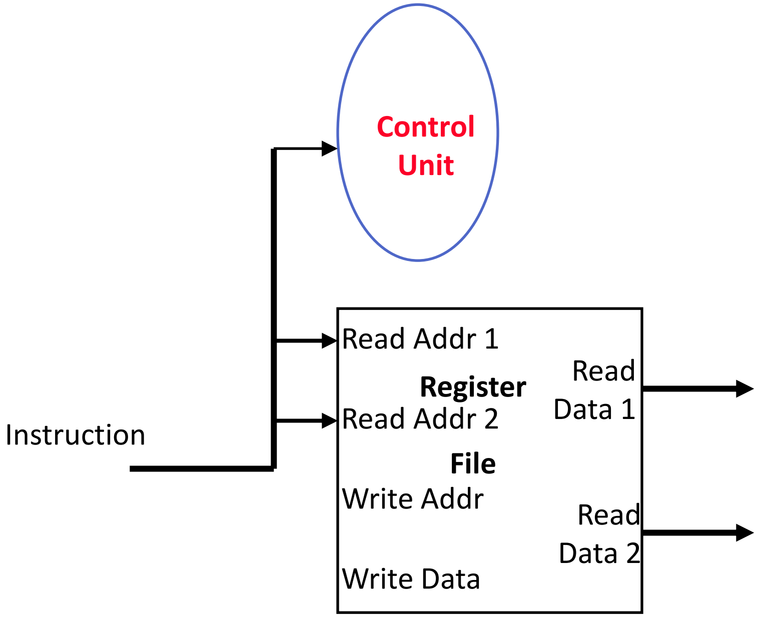
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC to hold the address of the next instruction



- PC is updated every cycle, so it does NOT need an explicit write control signal
- Instruction Memory is read every cycle, so it does NOT need an explicit read control signal

Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



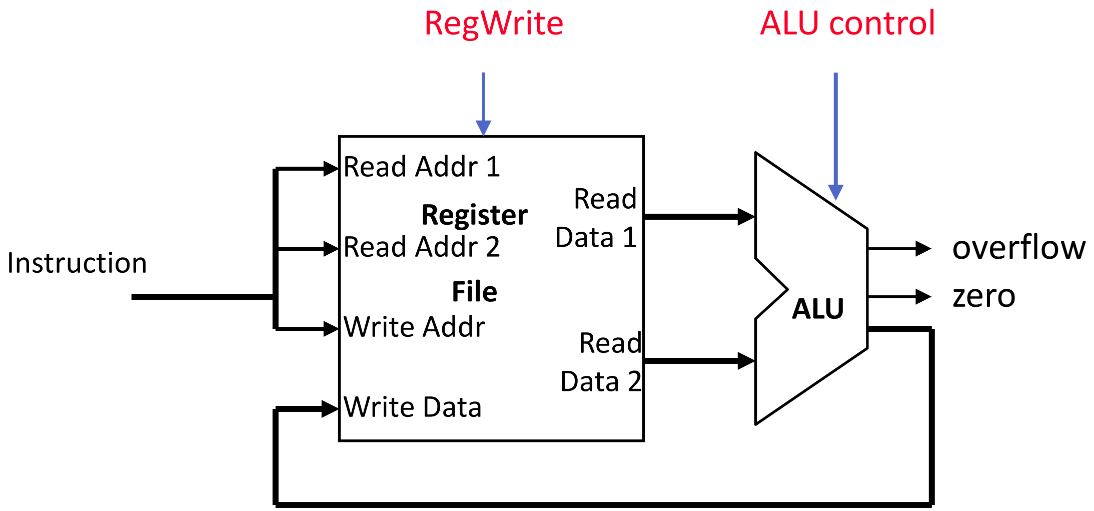
- reading two values from the Register File
  - Register File addresses are contained in the instruction

Executing R Format Operations

- R format operations (**add, sub, and, or**)

31	24	19	14	11	6	0
funct7	rs2	rs1	funct3	rd	opcode	R format

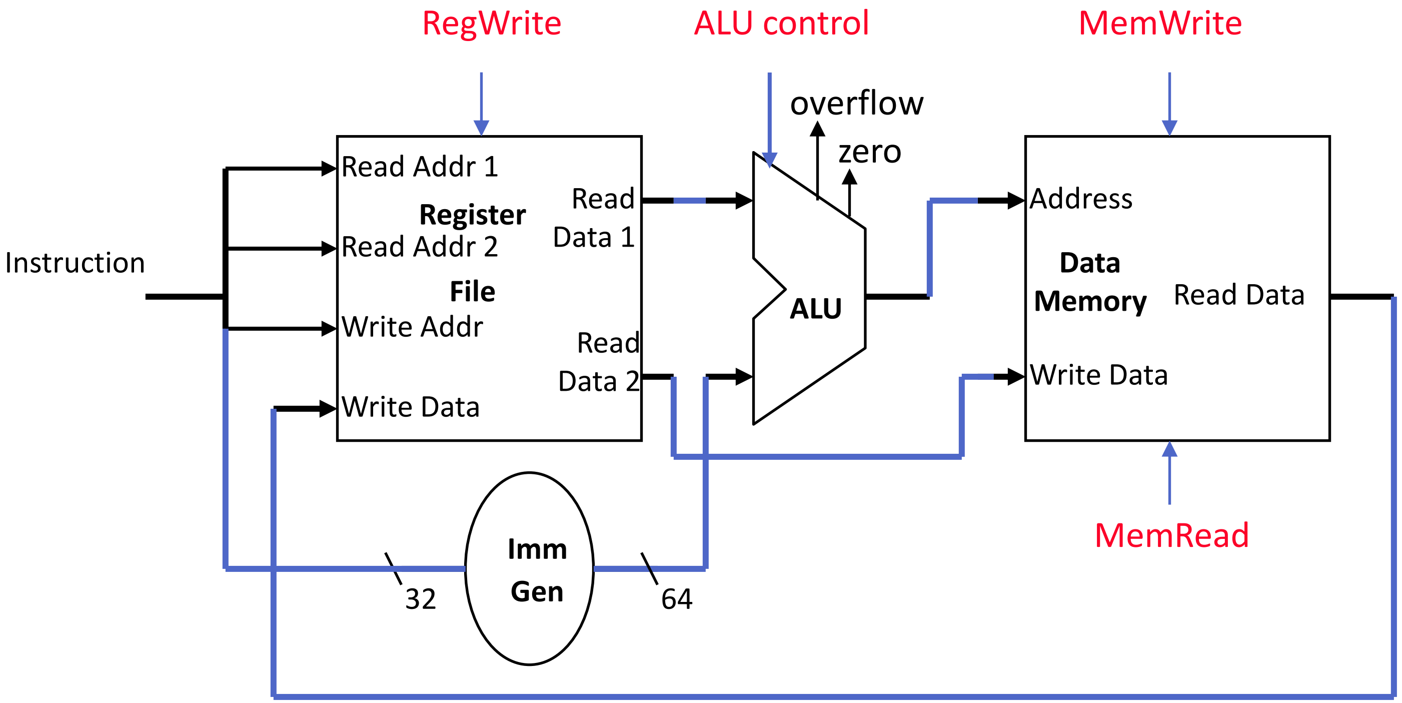
  - perform the (op and funct) operation on values in rs1 and rs2
  - store the result back into the Register File (into location rd)



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

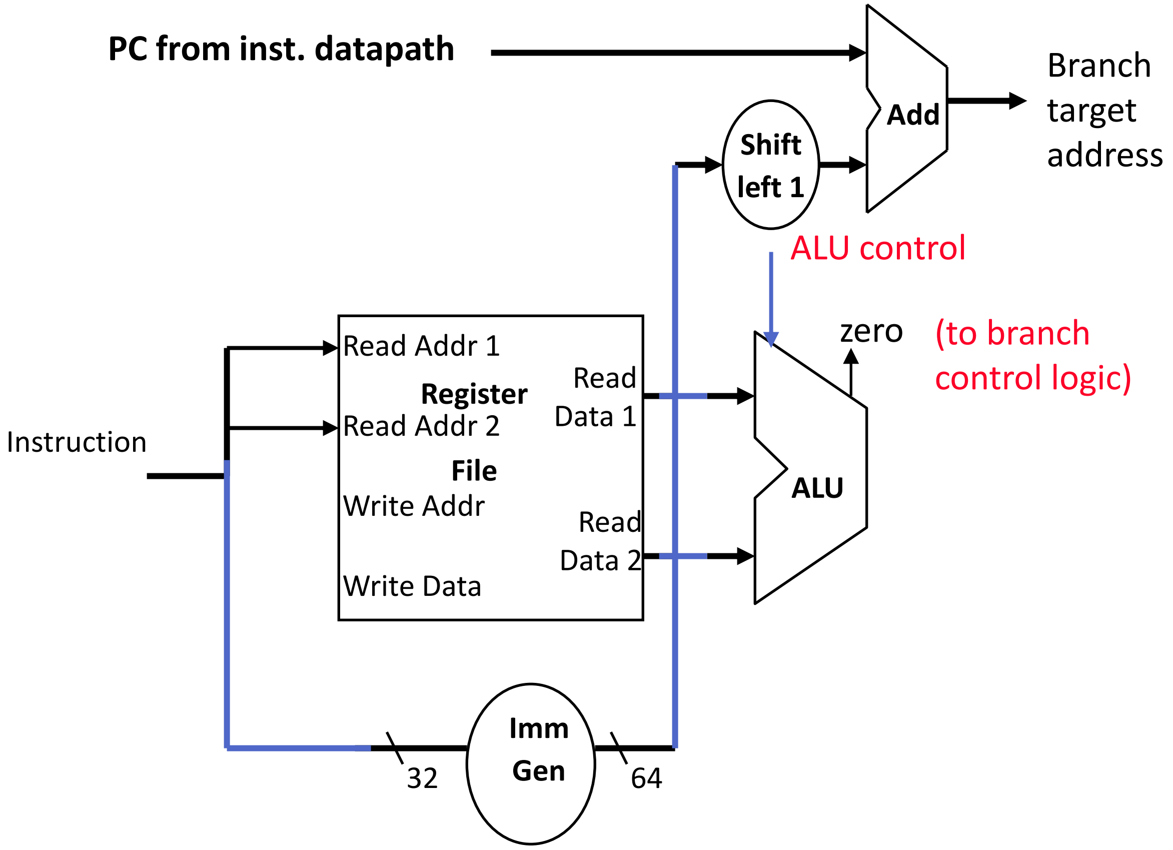
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 12-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



Executing Branch Operations

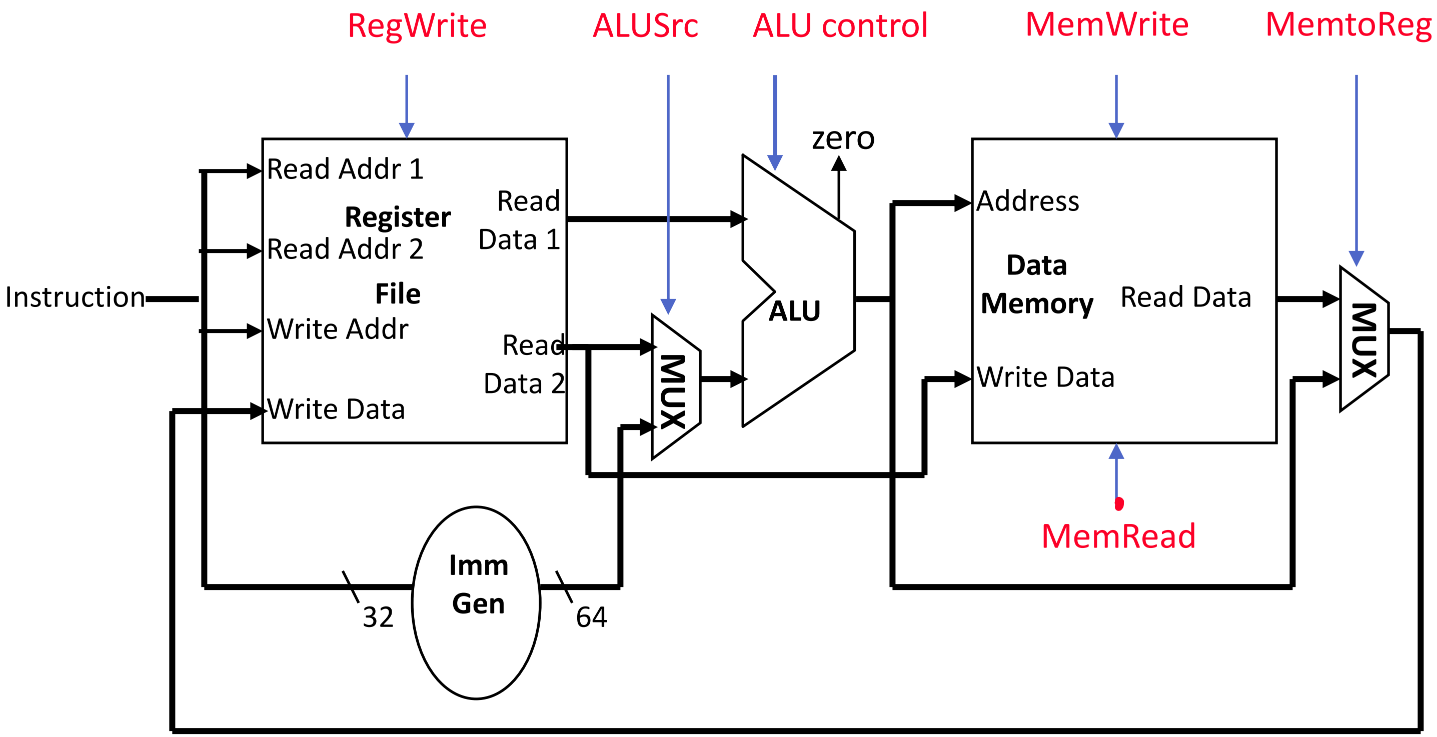
- Branch operations involves
  - compare the operands read from the Register File during decode for equality (**zero** ALU output)
  - compute the branch target address by adding the updated PC to the 12-bit signed-extended offset field in the instruction



Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions 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)
  - multiplexors 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

R-type and Memory Access



Adding the Control

Selecting the operations to perform (ALU, Register File and Memory read/write)

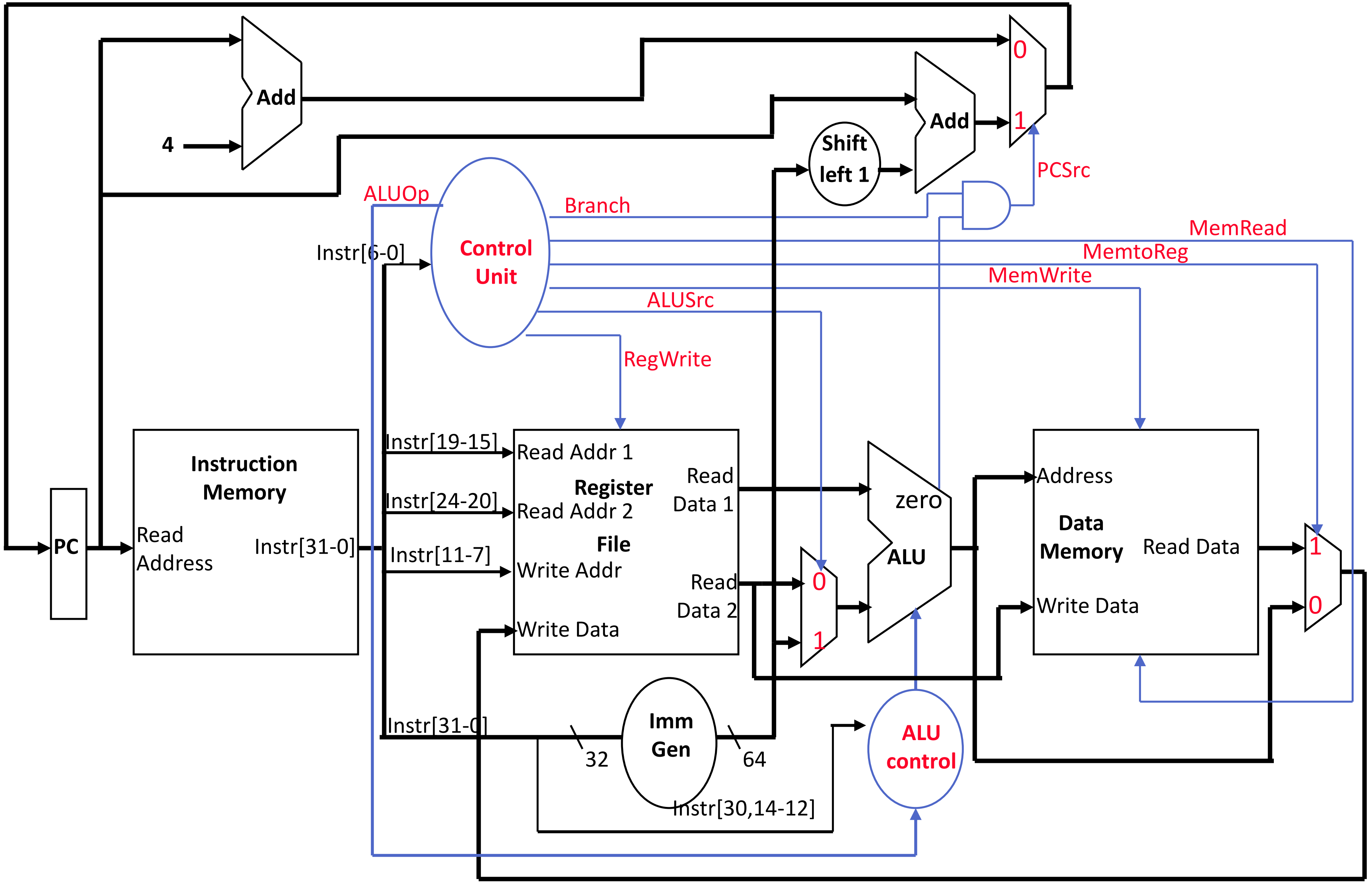
Controlling the flow of data (multiplexor inputs)

31 24 19 14 11 6 0

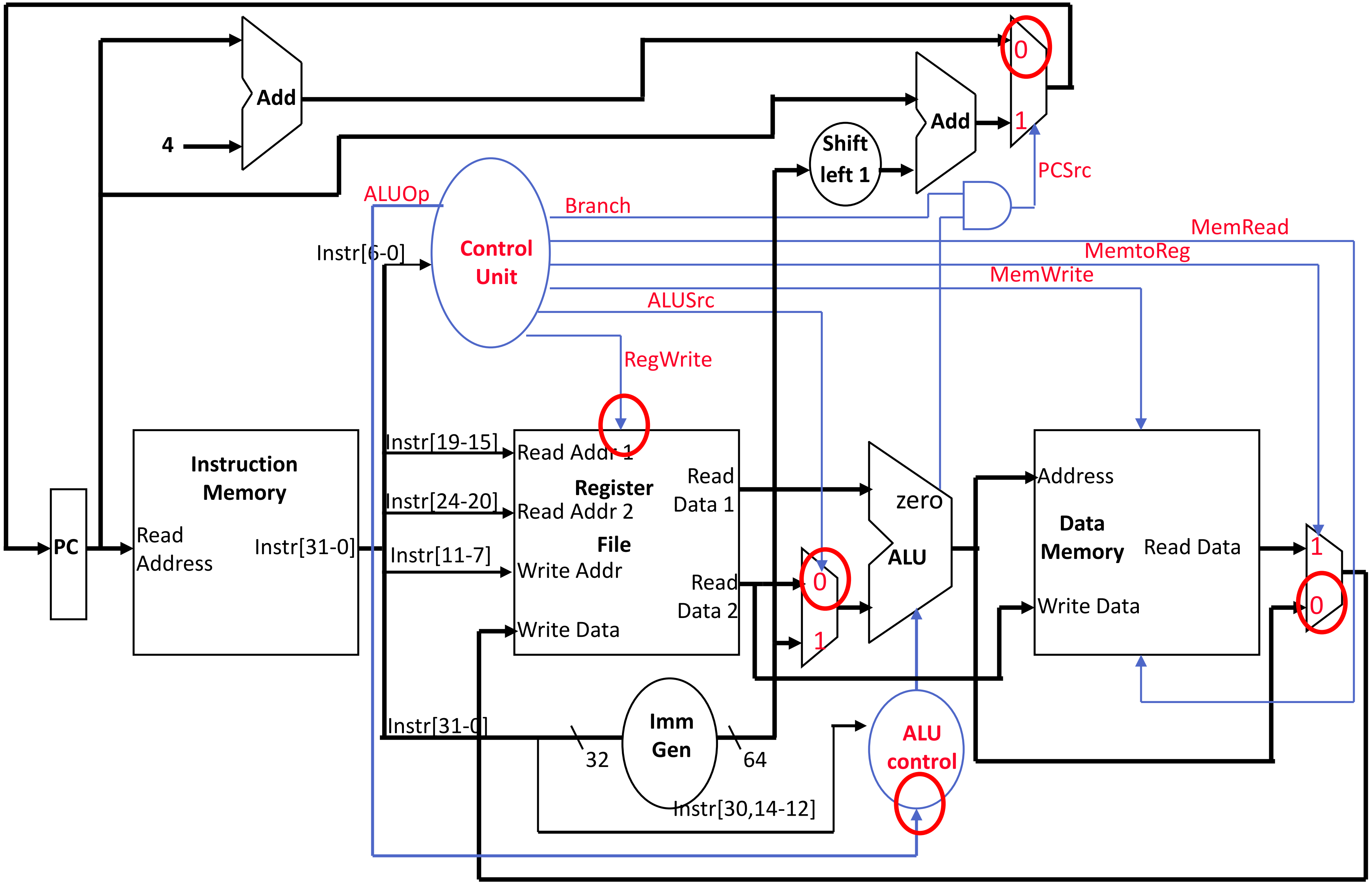
funct7	rs2	rs1	funct3	rd	opcode	R format
immediate		rs1	funct3	rd	opcode	I format
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S format
imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	SB format

- Observations
  - op field always in bits 6-0
  - addr of registers to be read are always specified by the rs1 field (bits 19-15) and rs2 field (bits 24-20);
  - addr. of register to be written is in rd (bits 11-7)
  - Another operand can also be a 12-bit offset for branch or load-store instructions.

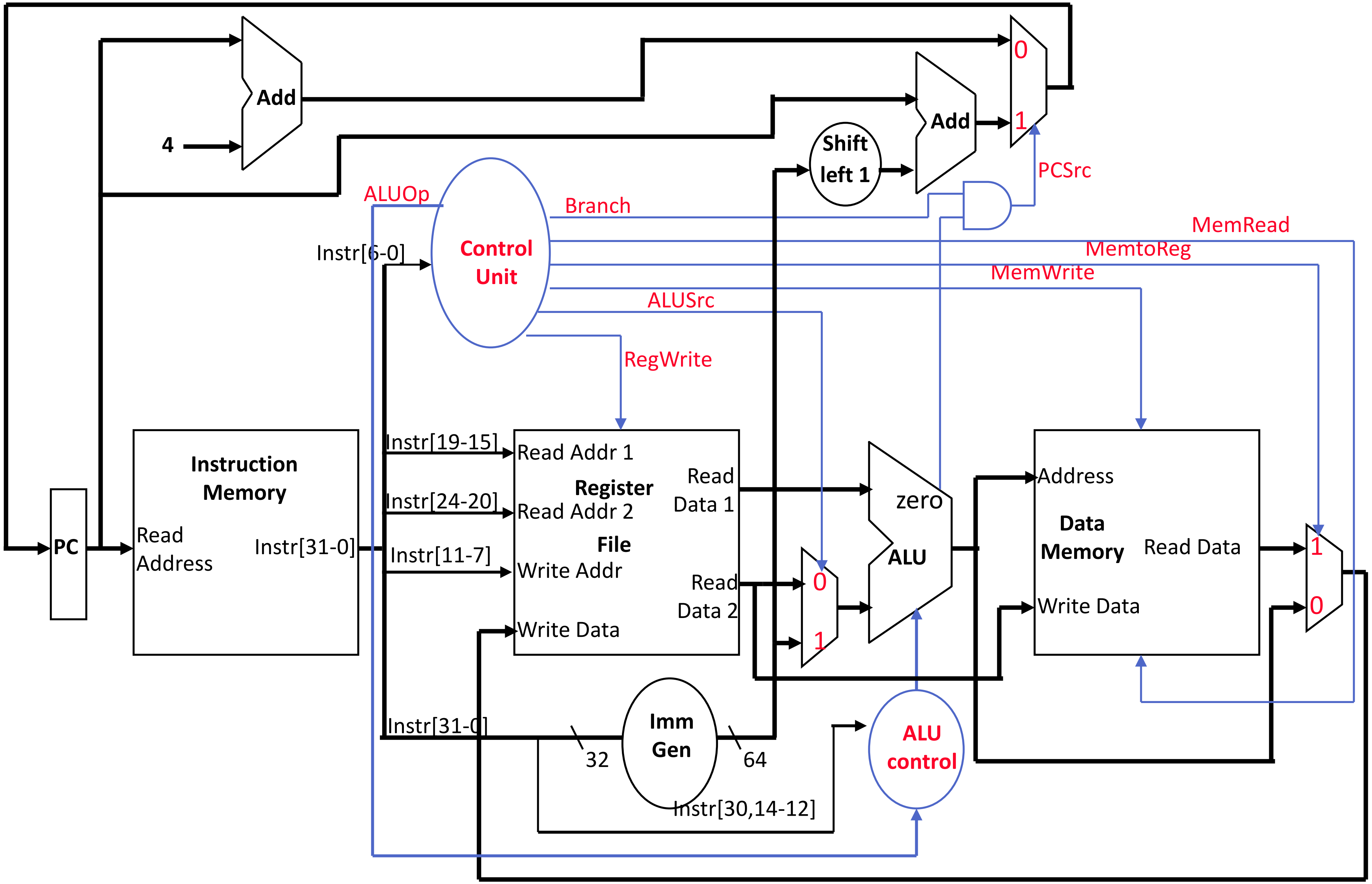
Single Cycle Datapath with Control Unit



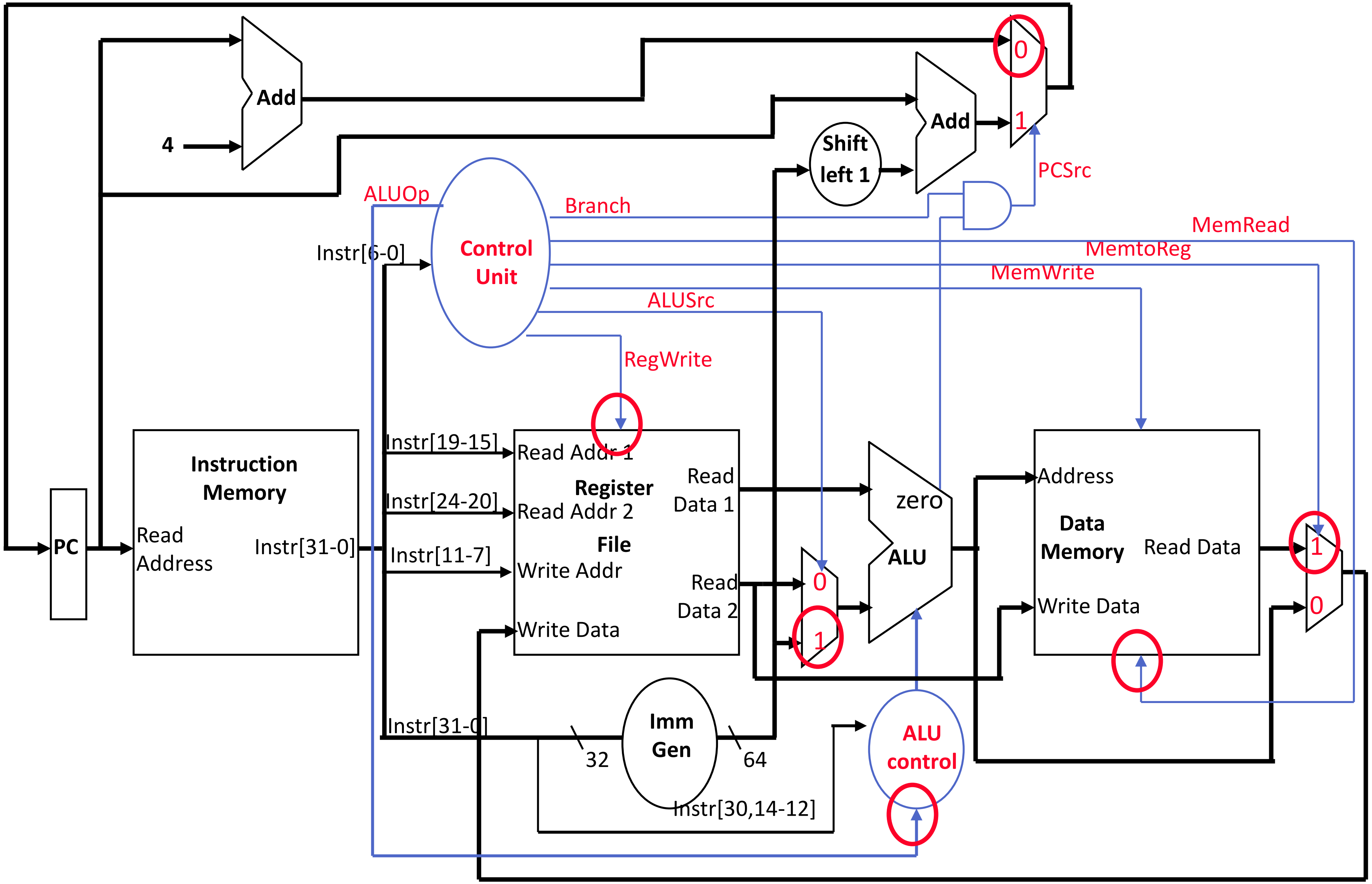
R-type Instruction Data/Control Flow



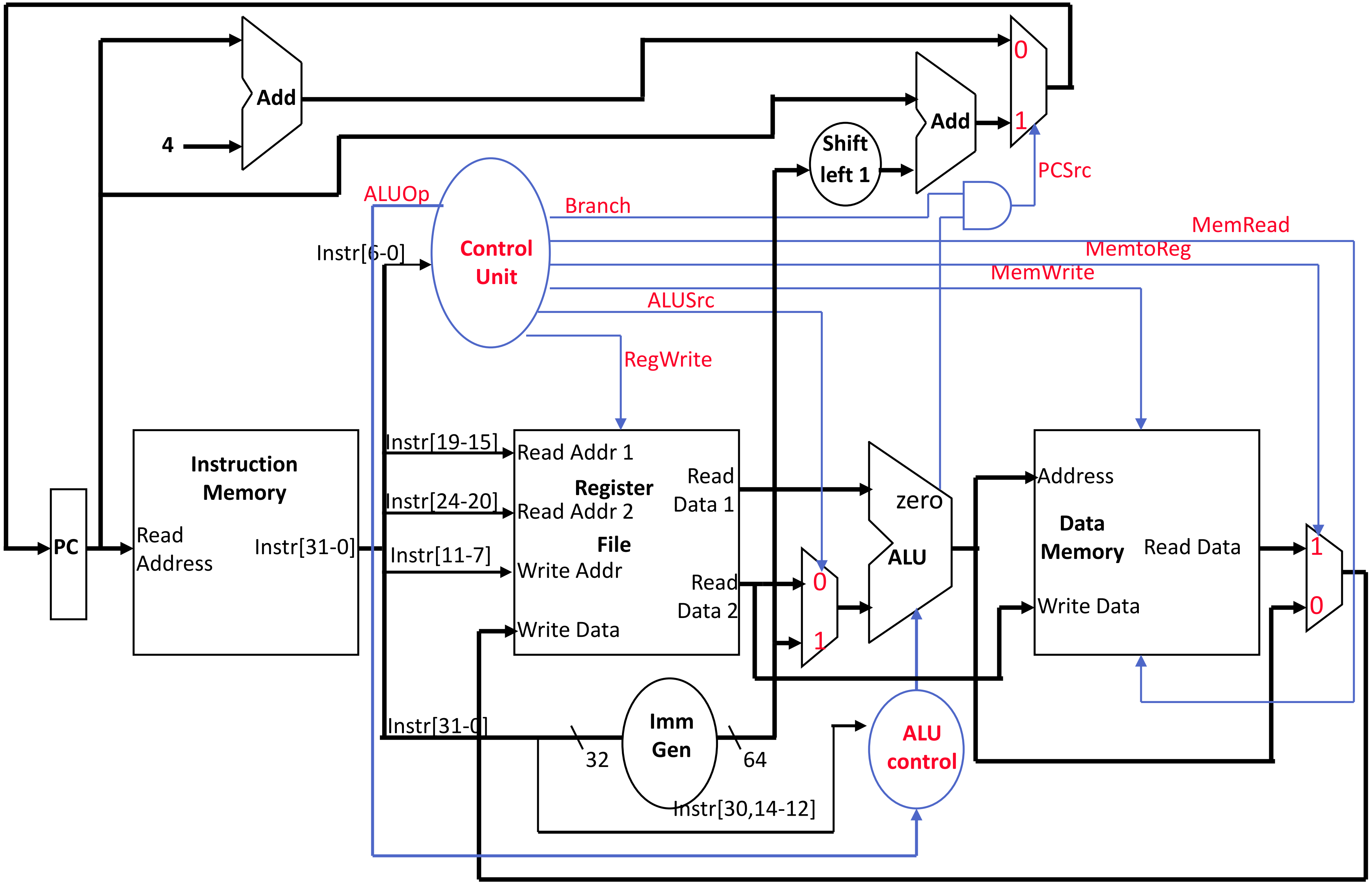
Load Word Instruction Data/Control Flow



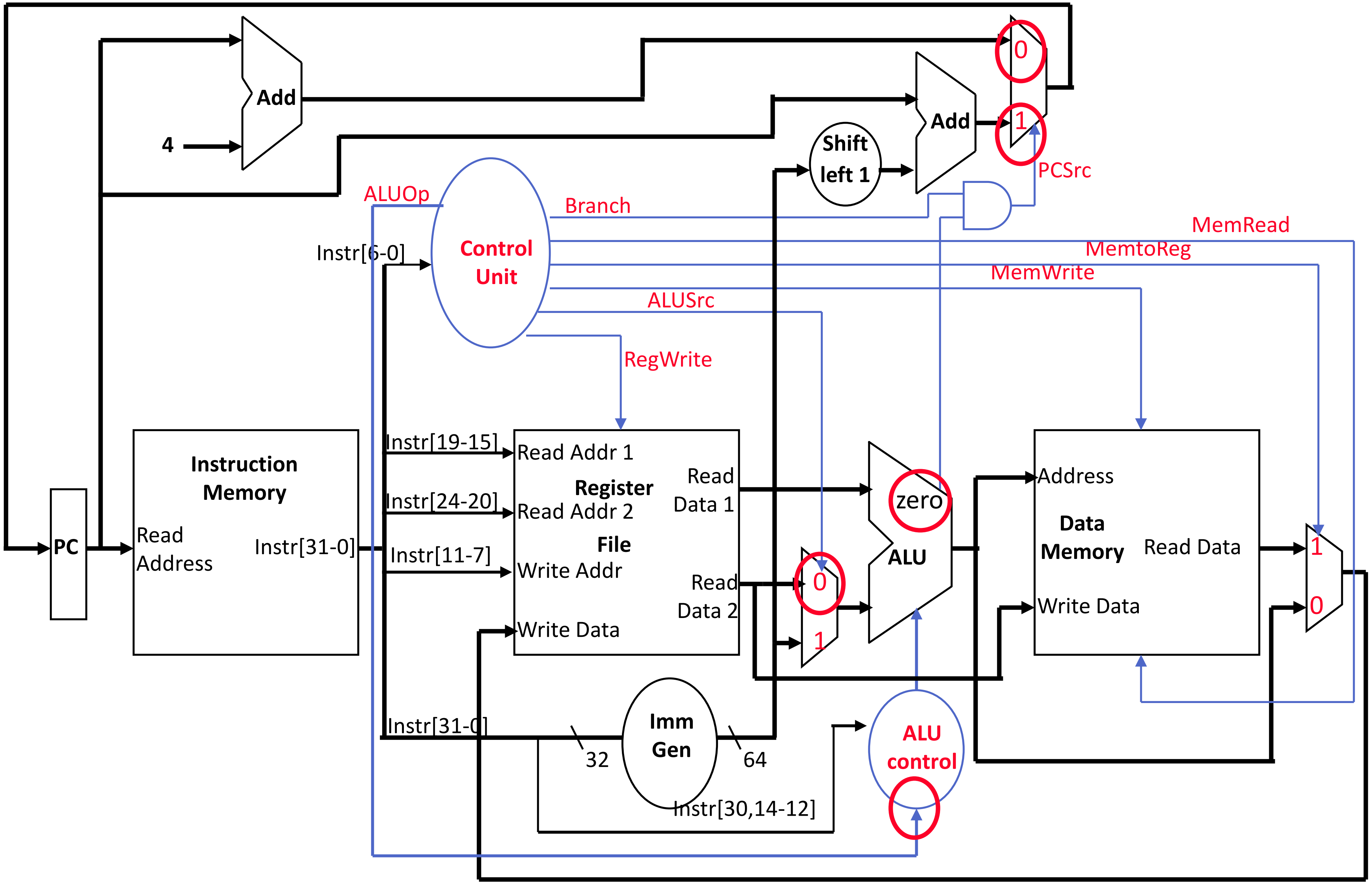
Load Word Instruction Data/Control Flow



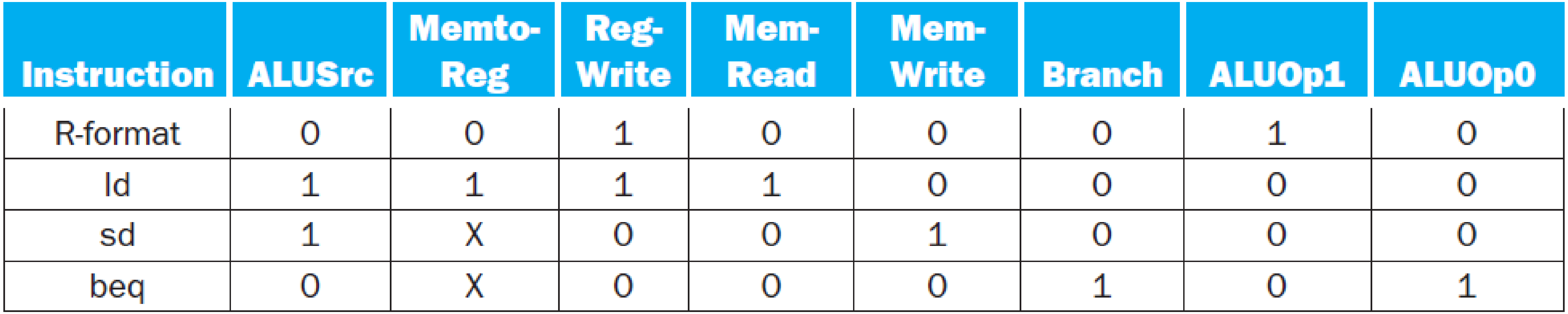
Branch Instruction Data/Control Flow



Branch Instruction Data/Control Flow



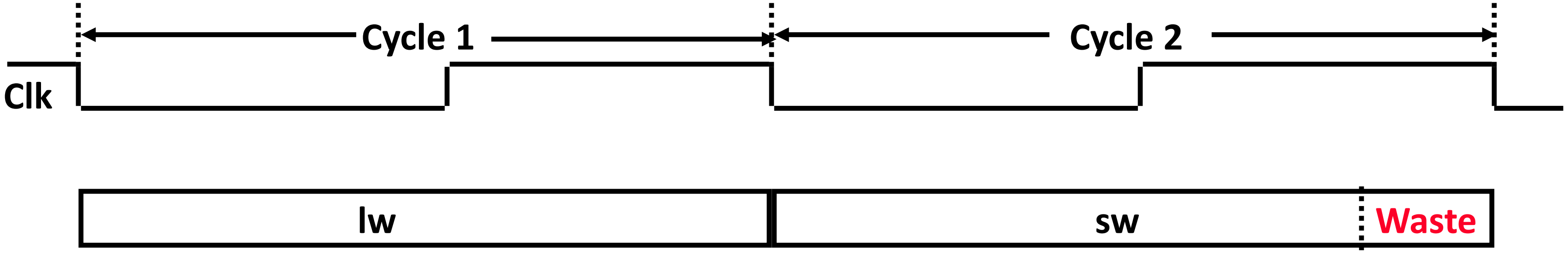
Control Signal



Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Single Cycle Disadvantages & Advantages

- **Disadvantages:** 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



- **Disadvantages:** 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

- **Advantages:** Is simple and easy to understand

Multicycle Datapath Approach

Let an instruction take more than 1 clock cycle to complete

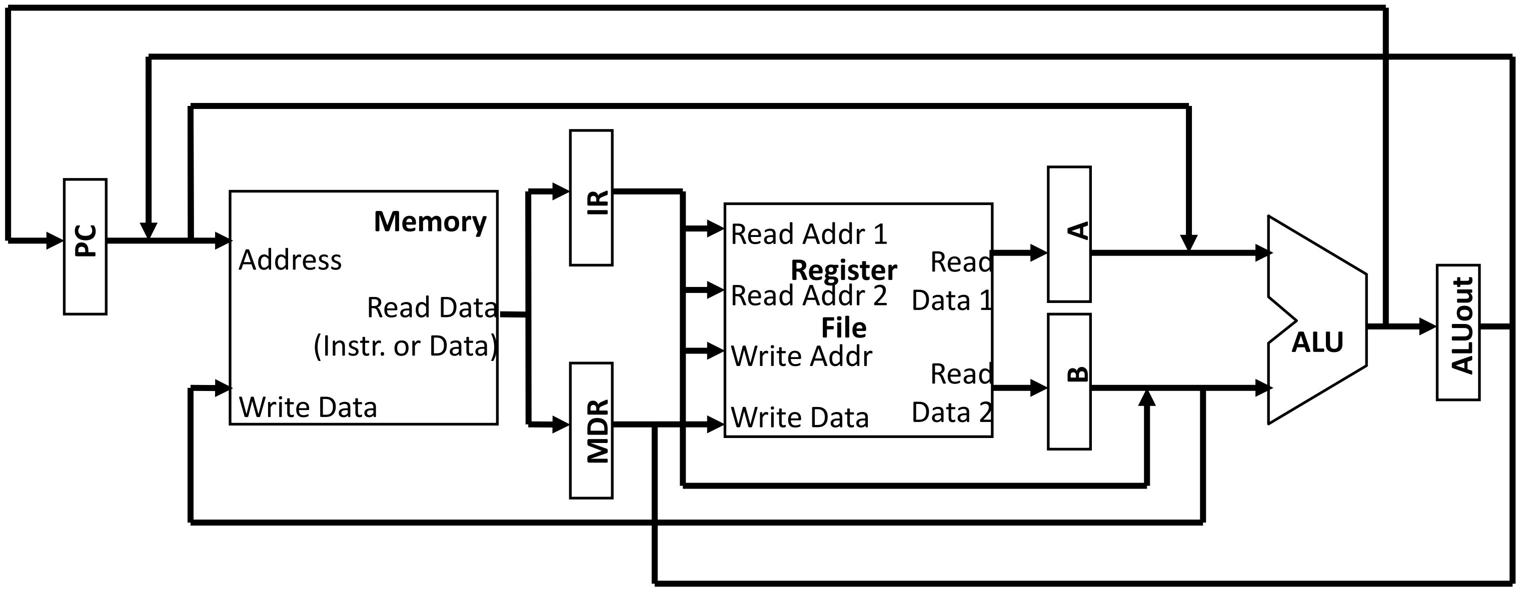
- Break up instructions into steps where each *step* takes a cycle while trying to
  - balance the amount of work to be done in each step
  - restrict each cycle to use only one major functional unit
- Not every instruction takes the *same* number of clock cycles

In addition to faster clock rates, multicycle allows functional units that can be used more than once per instruction as long as they are used on *different* clock cycles, as a result

- only need one memory – but only one memory access per cycle
- need only one ALU/adder – but only one ALU operation per cycle

Multicycle Datapath Approach, con’t

- At the end of a cycle
  - Store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer). All (except IR) hold data only between a pair of adjacent clock cycles (no write control signal needed)



IR – Instruction Register

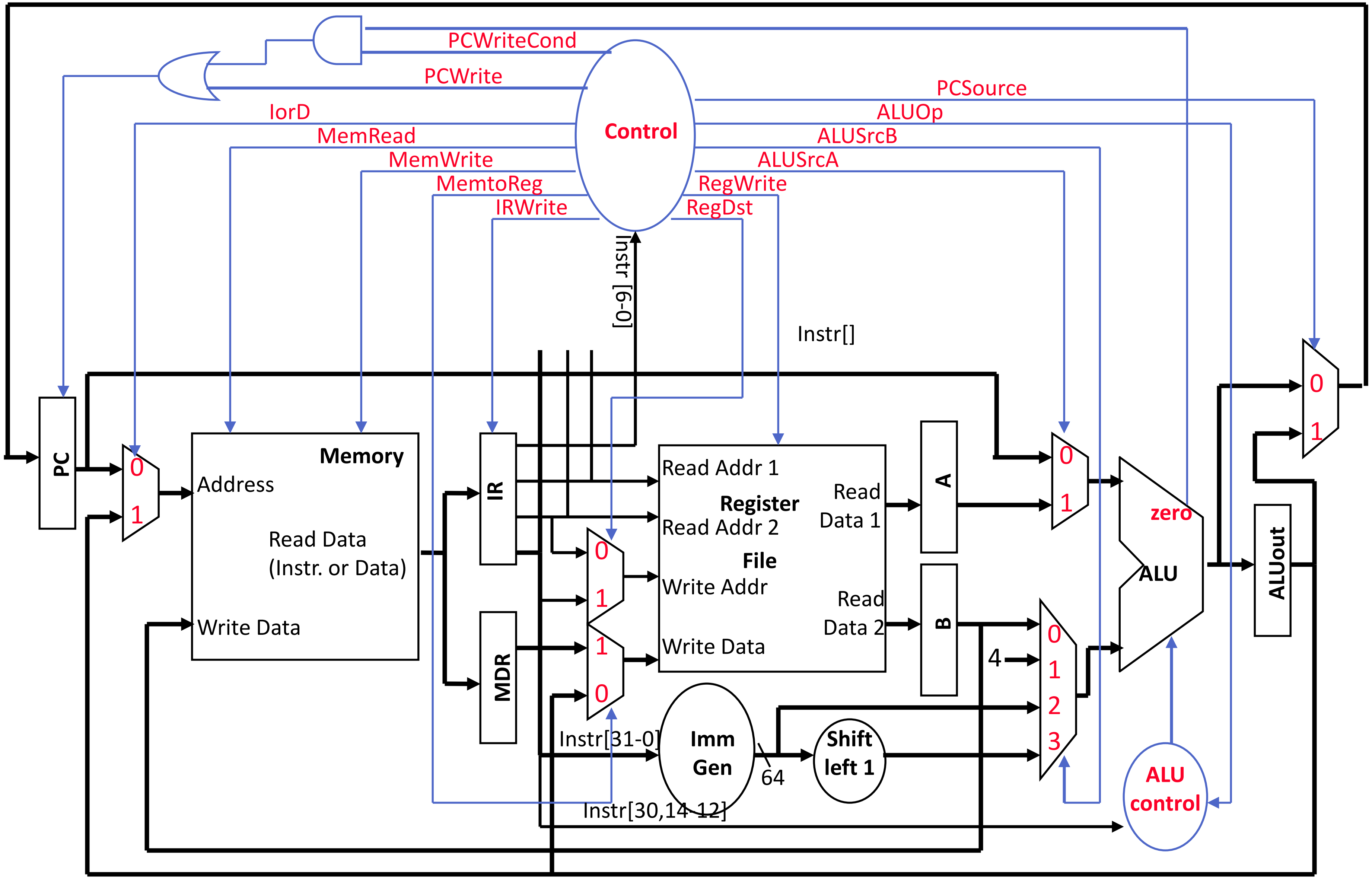
MDR – Memory Data Register

A, B – regfile read data registers

ALUout – ALU output register

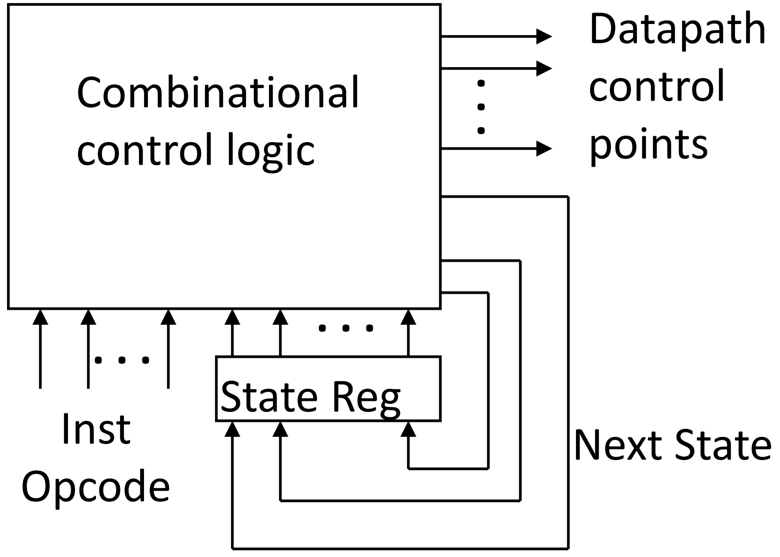
- Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

The Multicycle Datapath with Control Signals

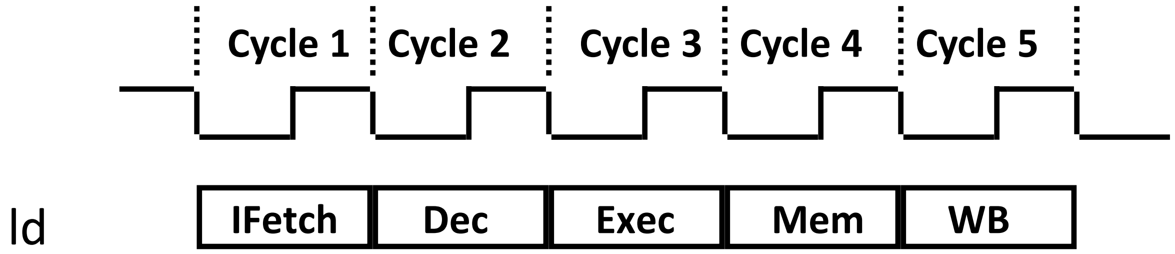


Multicycle Control Unit

- Multicycle datapath control signals are not determined solely by the bits in the instruction
  - e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next
- Must use a finite state machine (FSM) for control
  - a set of states (current state stored in State Register)
  - next state function (determined by current state and the input)
  - output function (determined by current state and the input)



The Five Steps of the Load Instruction

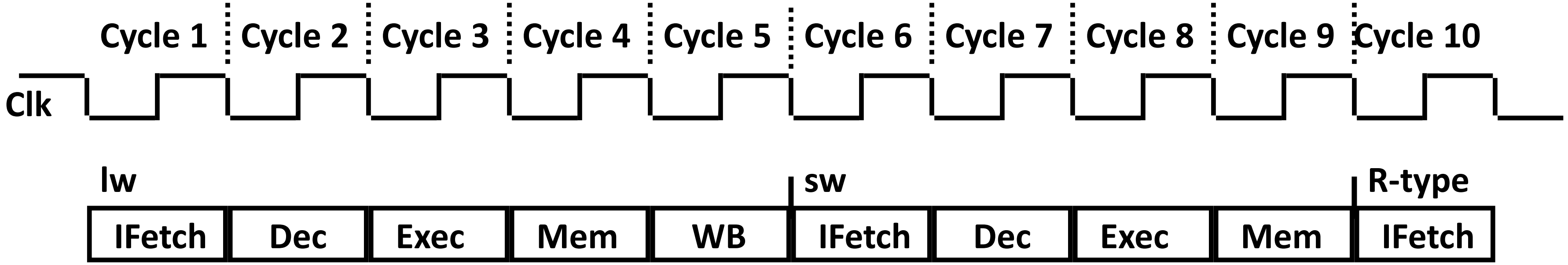


- IFetch: Instruction Fetch and Update PC
- Dec: Instruction Decode, Register Read, Sign Extend Offset
- Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch Completion
- Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- WB: Memory Read Completion (RegFile write)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Multicycle Advantages & Disadvantages

- **Advantages:** uses the clock cycle efficiently – the clock cycle is timed to accom modate the slowest instruction step

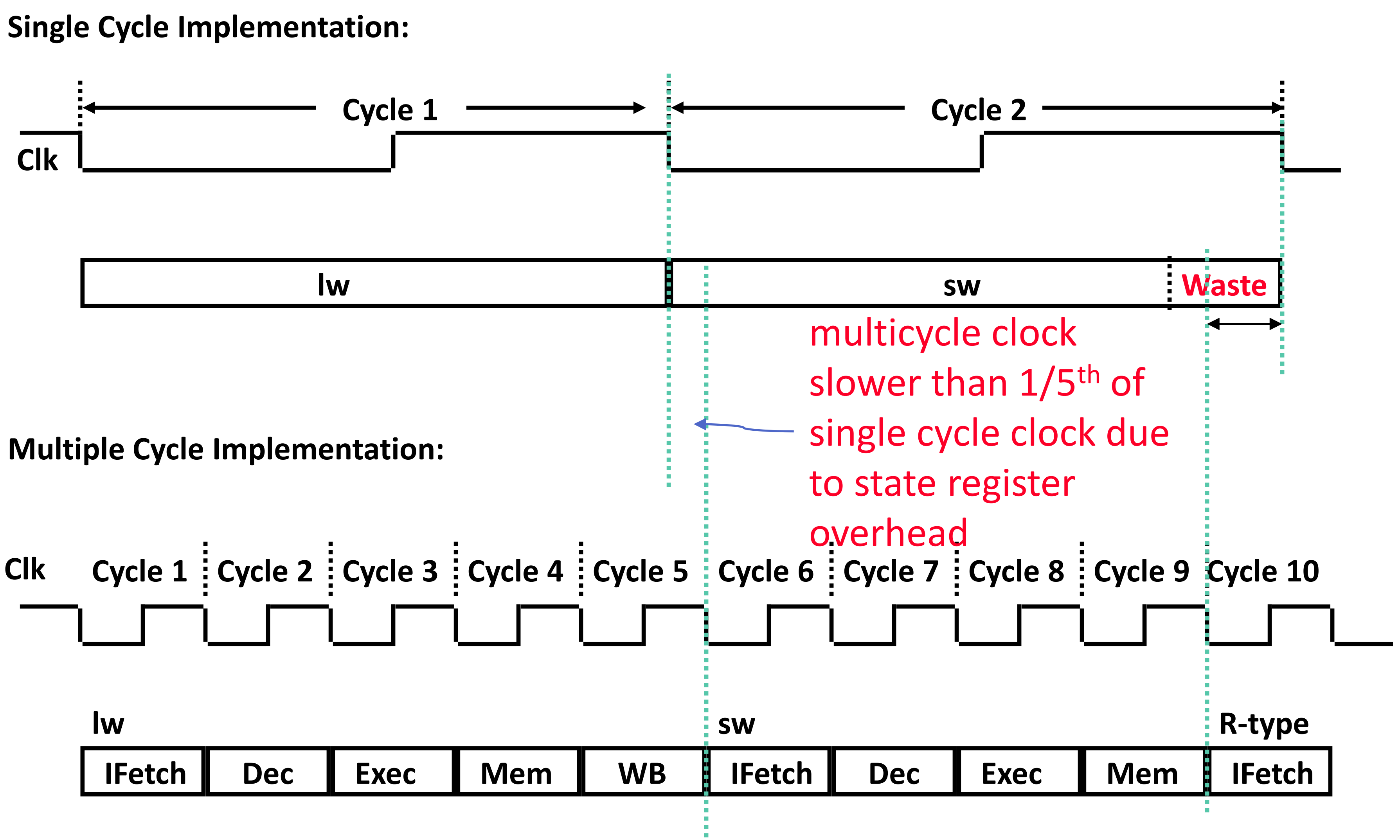


- **Advantages:** multicycle implementations allow functional units to be used mor e than once per instruction as long as they are used on different clock cycles

but

- **Disadvantages:** requires additional internal state registers, more muxes, and m ore complicated (FSM) control

Single Cycle vs. Multiple Cycle Timing



How Can We Make It Even Faster?

- **Option 1:** split the multiple instruction cycle into smaller and smaller steps
  - There is a point of diminishing returns where as much time is spent loading the state registers as doing the work

- **Option 2:** 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:

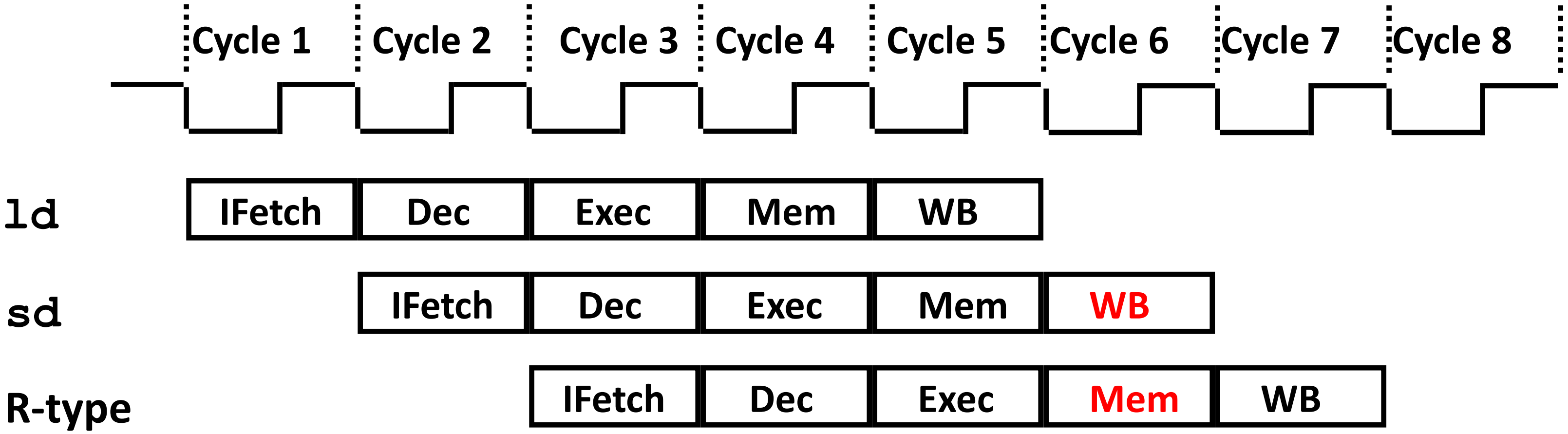
  CPU time = CPI * CC * IC

- **Option 3:** fetch (and execute) more than one instruction at a time

  Superscalar processing – stay tuned

A Pipelined Processor

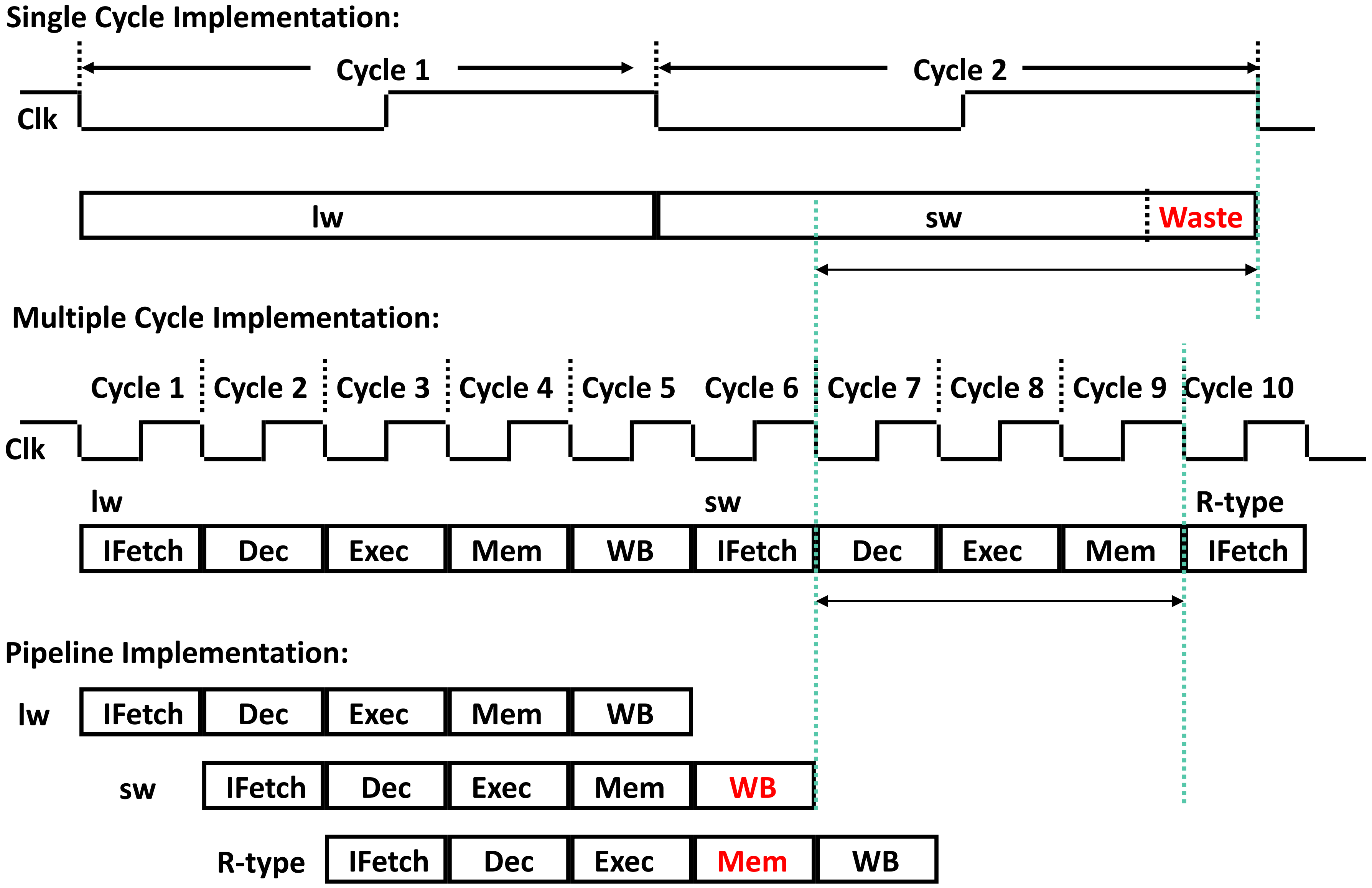
- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
`ld`	IFetch	Dec	Exec	Mem	WB			
`sd`		IFetch	Dec	Exec	Mem	WB		
R-type			IFetch	Dec	Exec	Mem	WB	

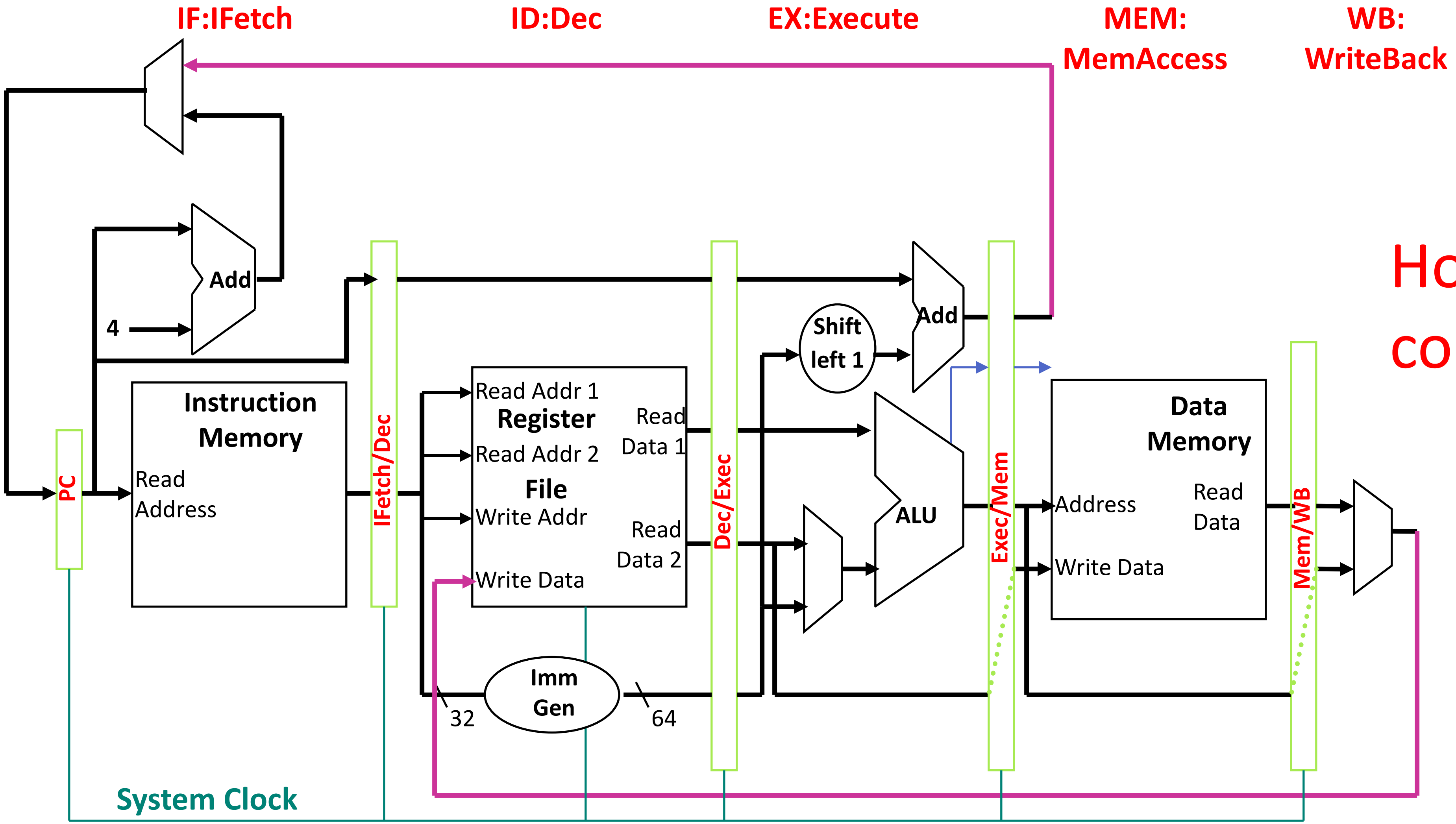
- clock cycle (pipeline stage time) is limited by the slowest stage
- for some instructions, some stages are wasted cycles

Single Cycle, Multiple Cycle, vs. Pipeline



Pipeline Datapath Modifications

- What do we need to add/modify in our datapath?
  - State registers between each pipeline stage to isolate them



How about control signals

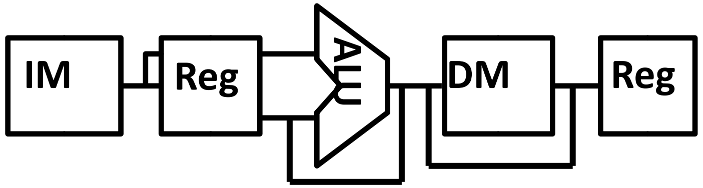
Control Signals

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Pipelining the RISC-V ISA

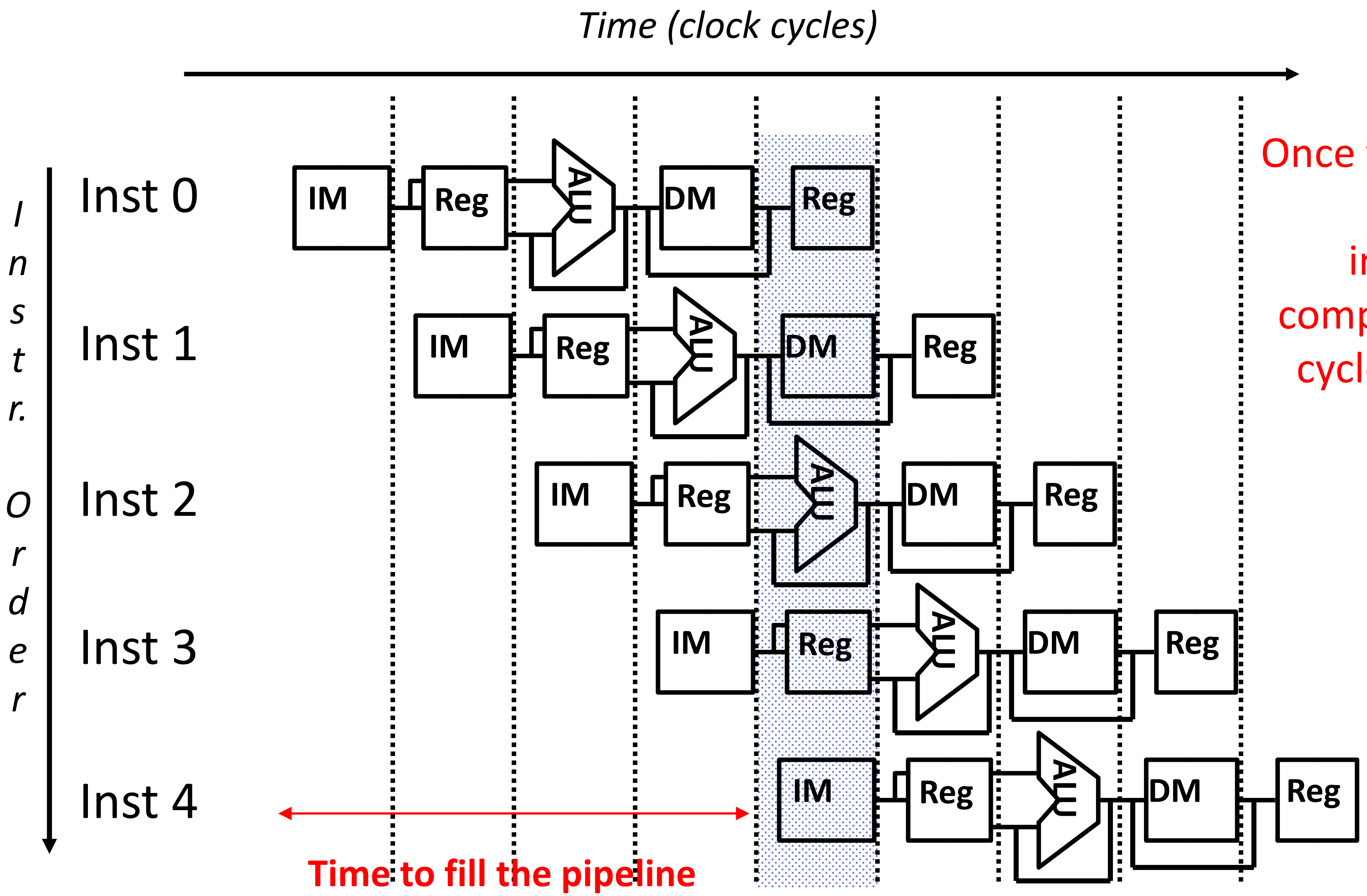
- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats with carefully designed fields
    - can begin reading register file in 2nd stage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)
- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if an instruction's input operands depend on the output of a previous instruction?

Graphically Representing a 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?

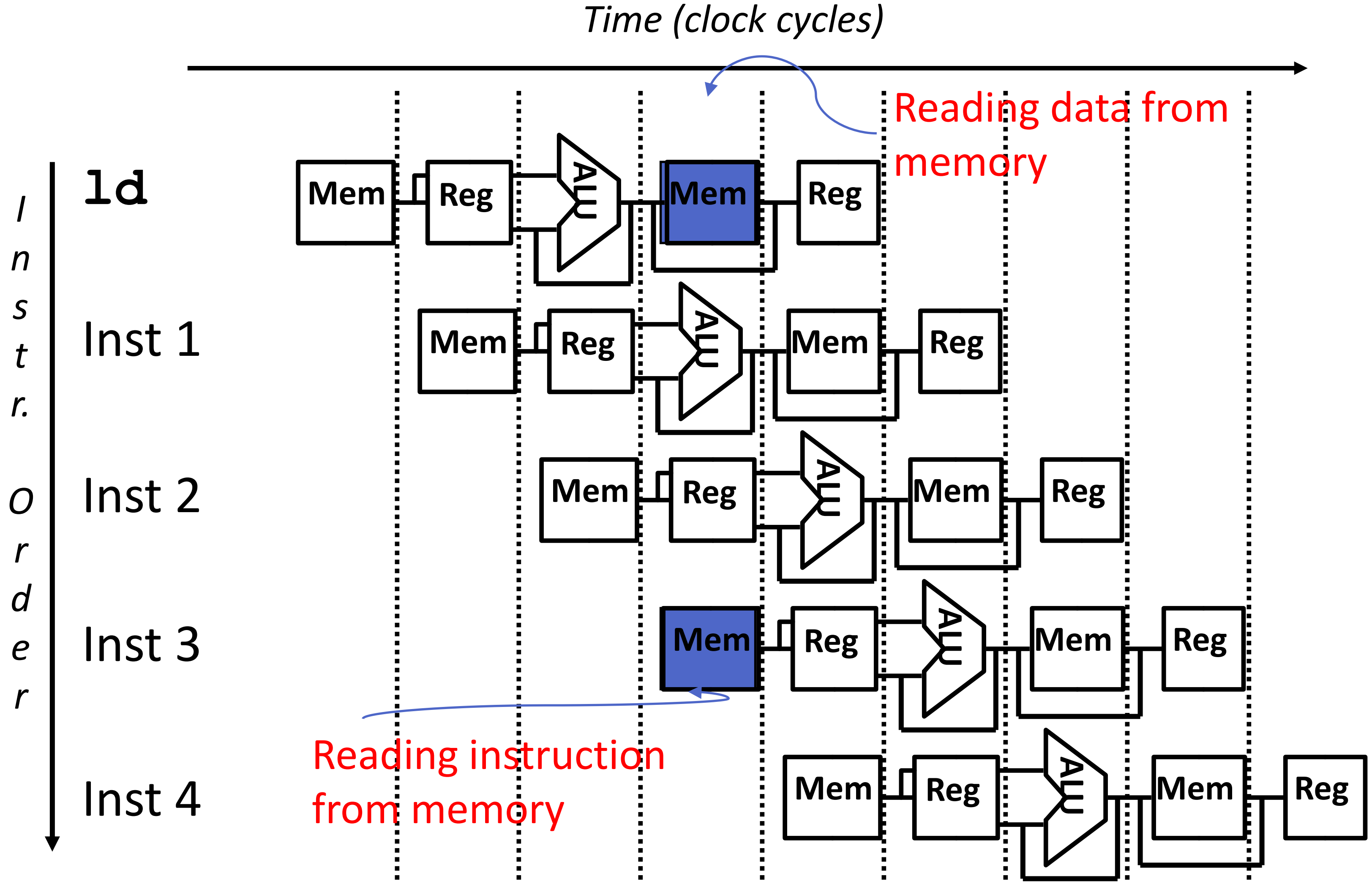
- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch instructions

- Can always resolve hazards by waiting

  pipeline control must detect the hazard

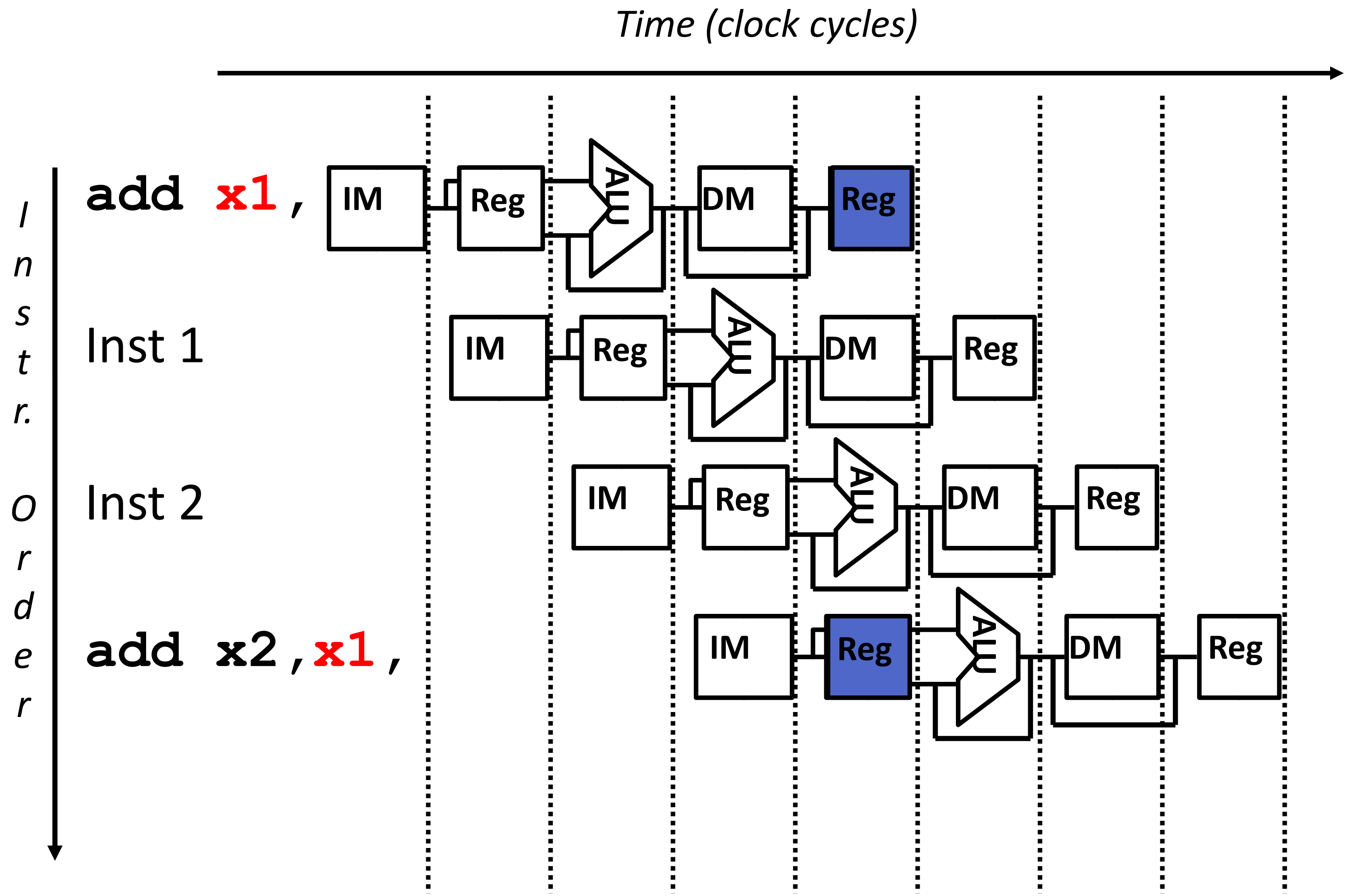
  and take action to resolve hazards

A Single Memory Would Be a Structural Hazard

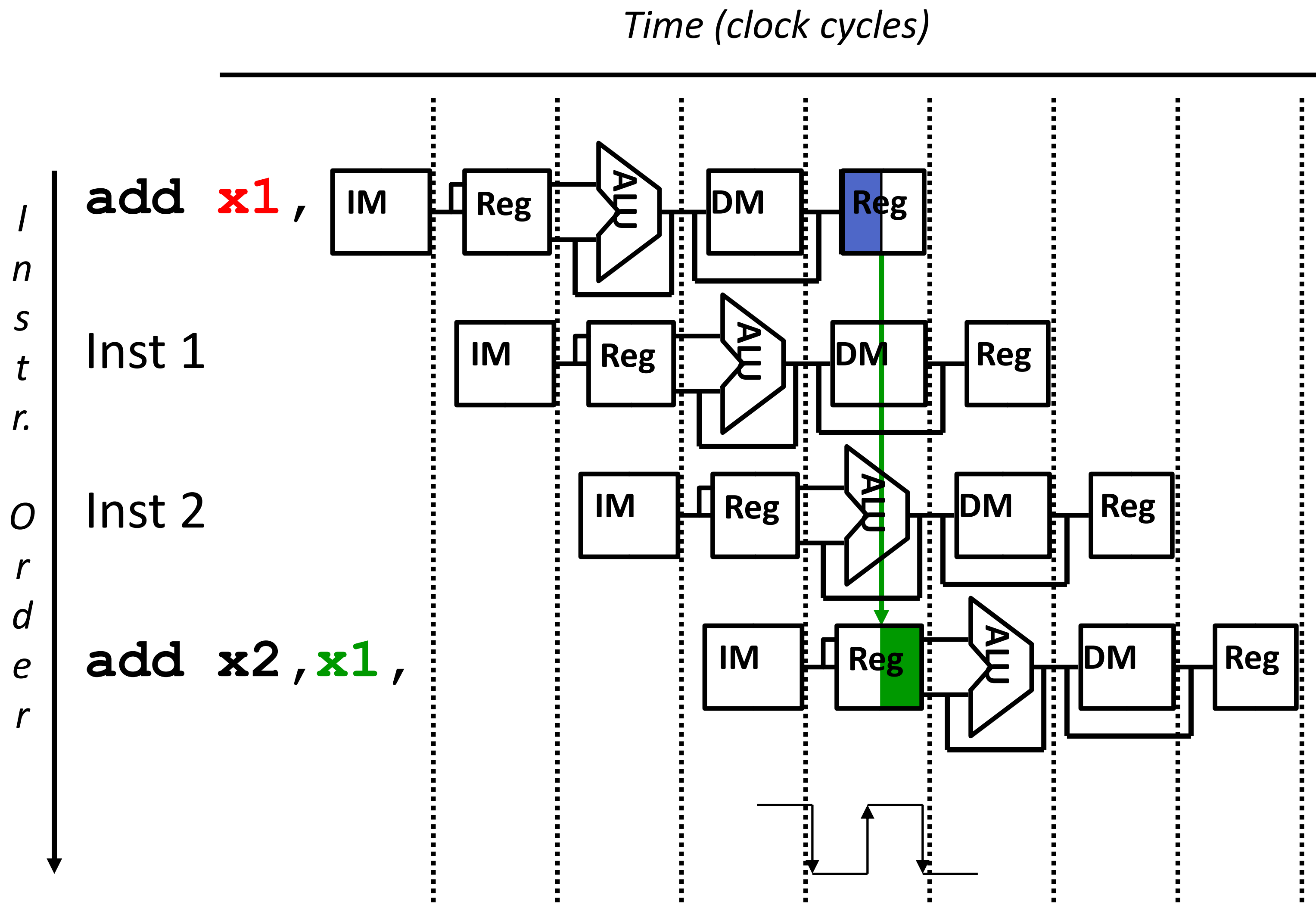


- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



How About Register File Access?



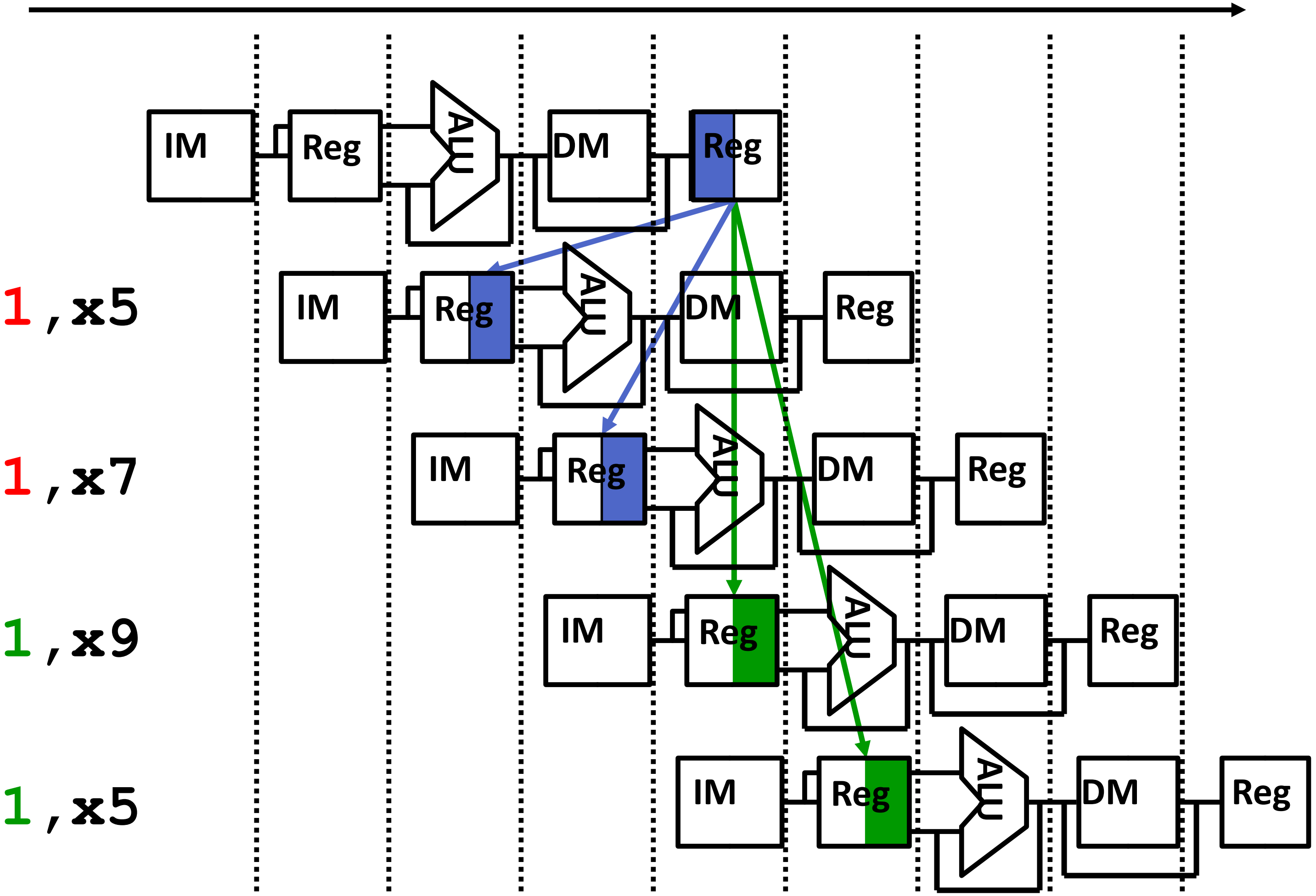
Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

Instr. Order

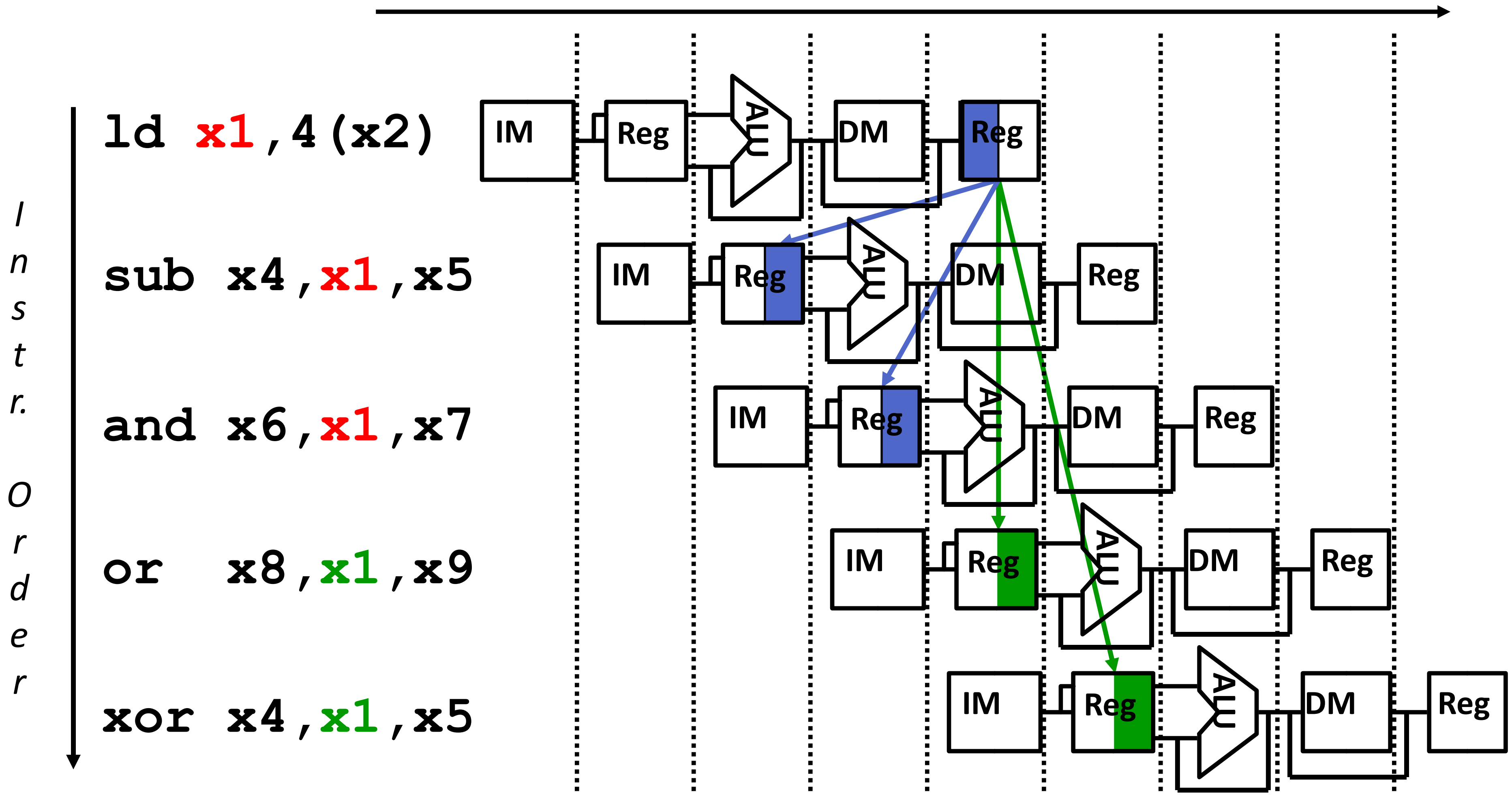
```
add x1,
sub x4,x1,x5
and x6,x1,x7
or  x8,x1,x9
xor x4,$1,x5
```



➢ Read after write data hazard

Loads Can Cause Data Hazards

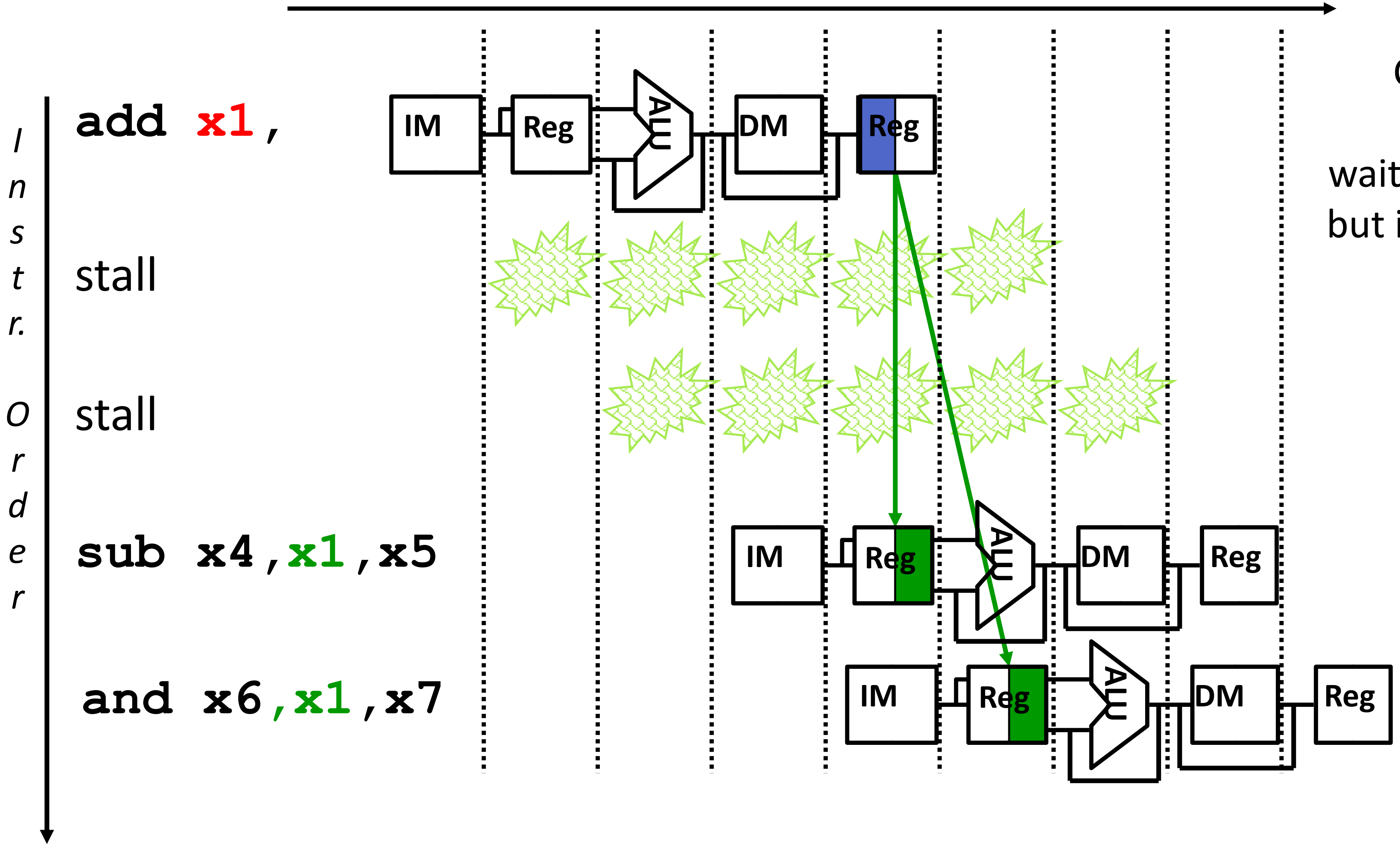
- Dependencies backward in time cause hazards



```
ld  x1,4(x2)
sub x4,x1,x5
and x6,x1,x7
or  x8,x1,x9
xor x4,x1,x5
```

> Read after write data hazard

One Way to “Fix” a Data Hazard



`add x1,`

stall

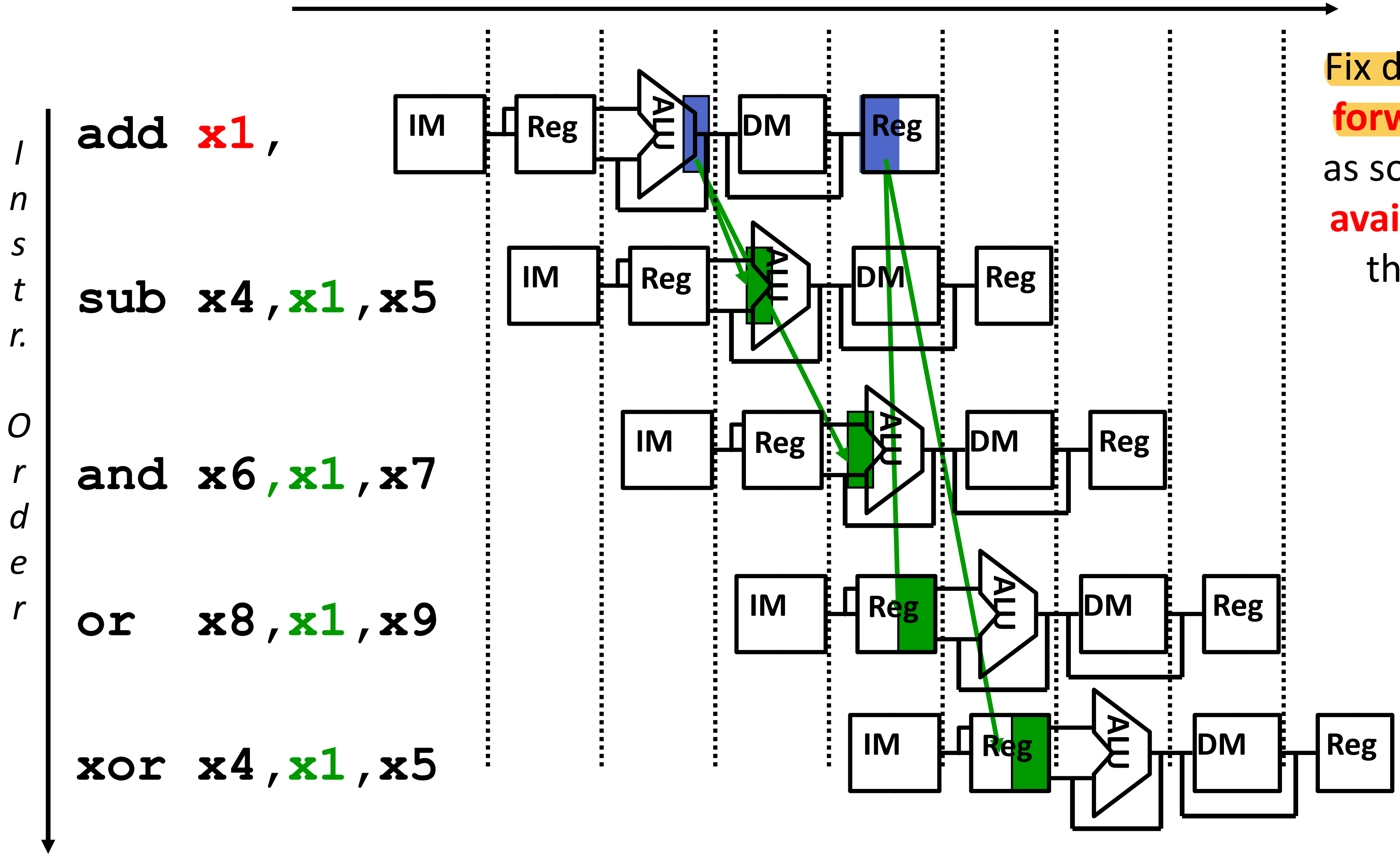
stall

`sub x4,x1,x5`

`and x6,x1,x7`

Can fix data hazard by waiting – stall – but impacts CPI

Another Way to “Fix” a Data Hazard



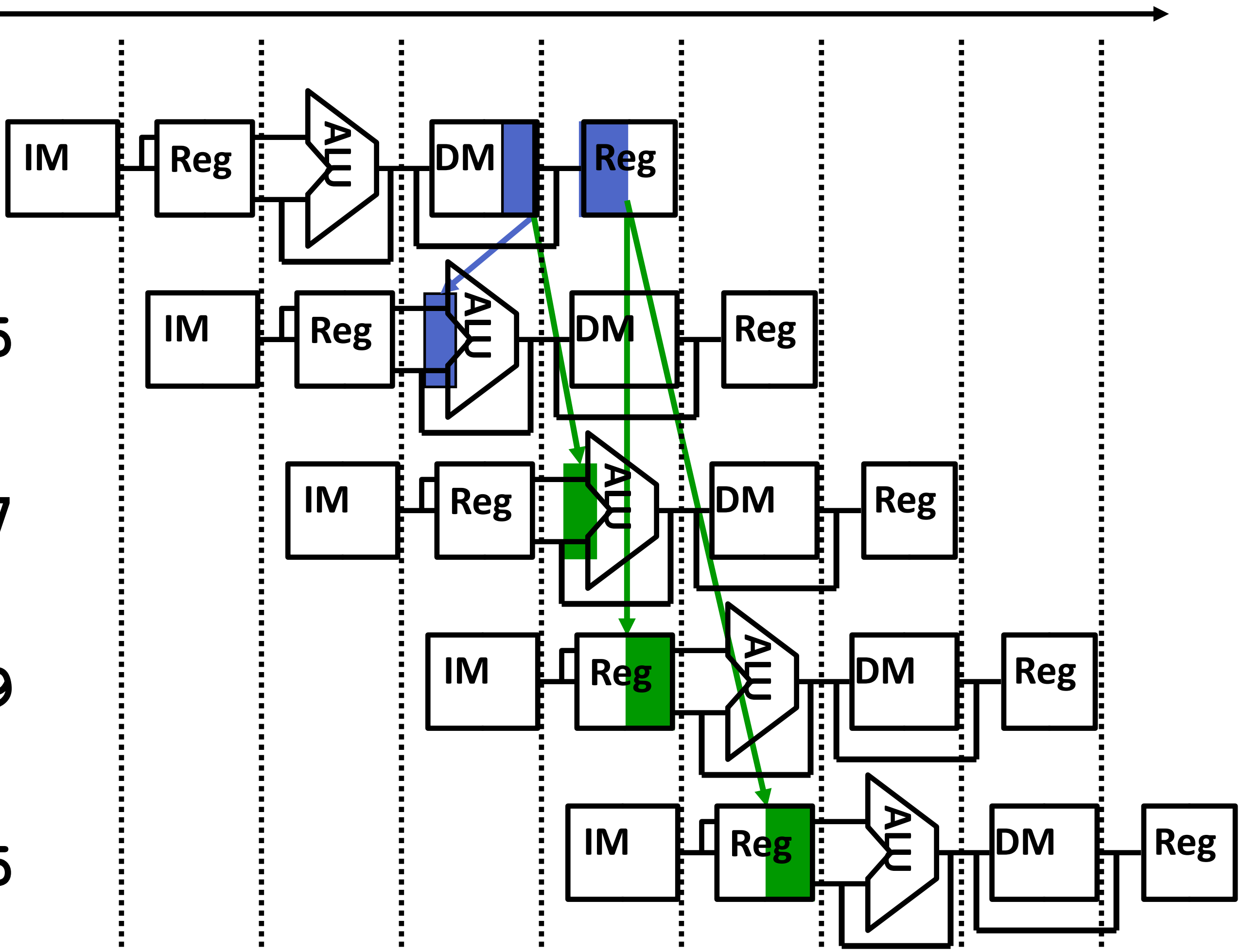
```
add x1,
sub x4,x1,x5
and x6,x1,x7
or  x8,x1,x9
xor x4,x1,x5
```

Fix data hazards by **forwarding** results as soon as they are **available** to where they are **needed**

Forwarding with Load-use Data Hazards

Instr. Order

```
ld   x1,4(x2)
sub  x4,x1,x5
and  x6,x1,x7
or   x8,x1,x9
xor  x4,x1,x5
```



- Will still need one stall cycle even with forwarding

Data Forwarding (aka Bypassing)

- 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 multiplexors 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 can achieve a CPI of 1 even in the presence of data dependencies

Data Forwarding Control Conditions

1. EX/MEM hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

2. MEM/WB hazard:

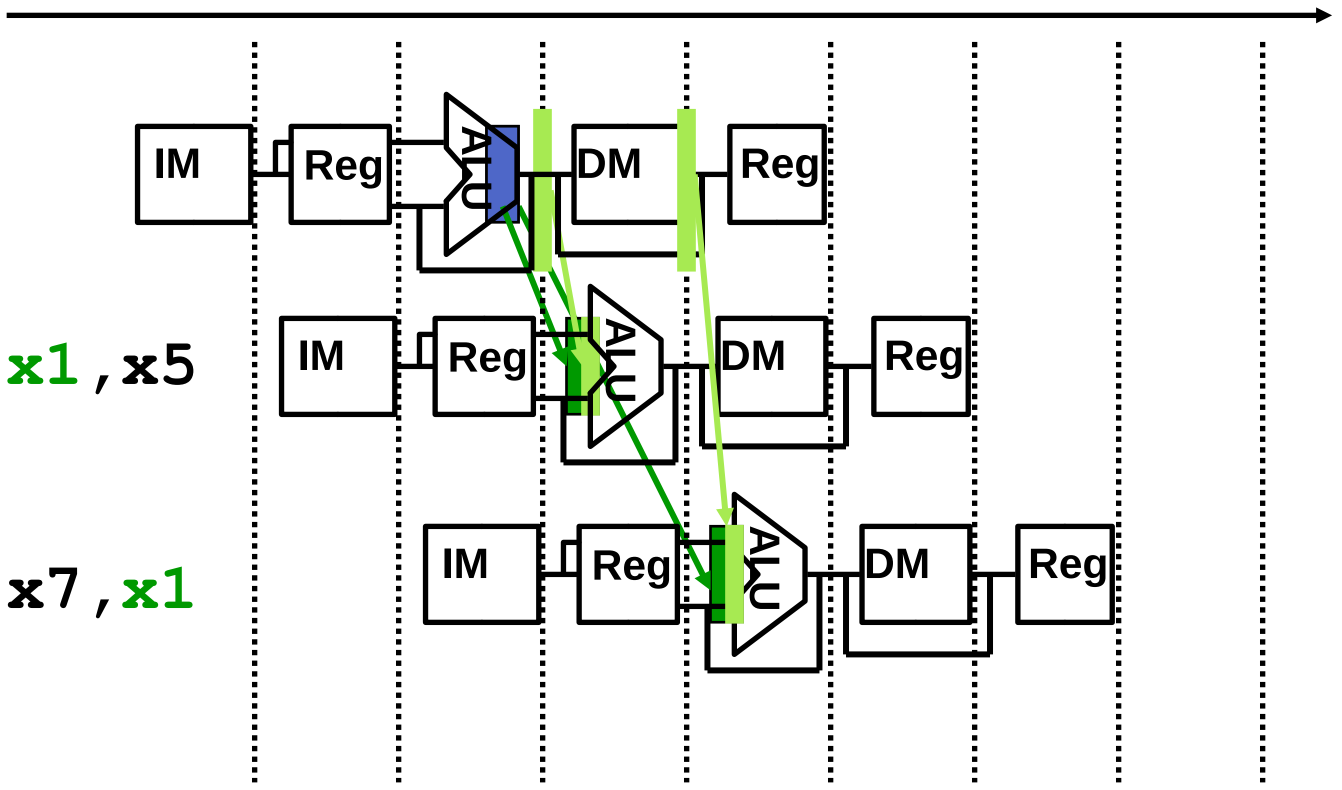
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU

Forwarding Illustration

Instr. Order

```
add x1,
sub x4,x1,x5
and x6,x7,x1
```



EX/MEM hazard forwarding

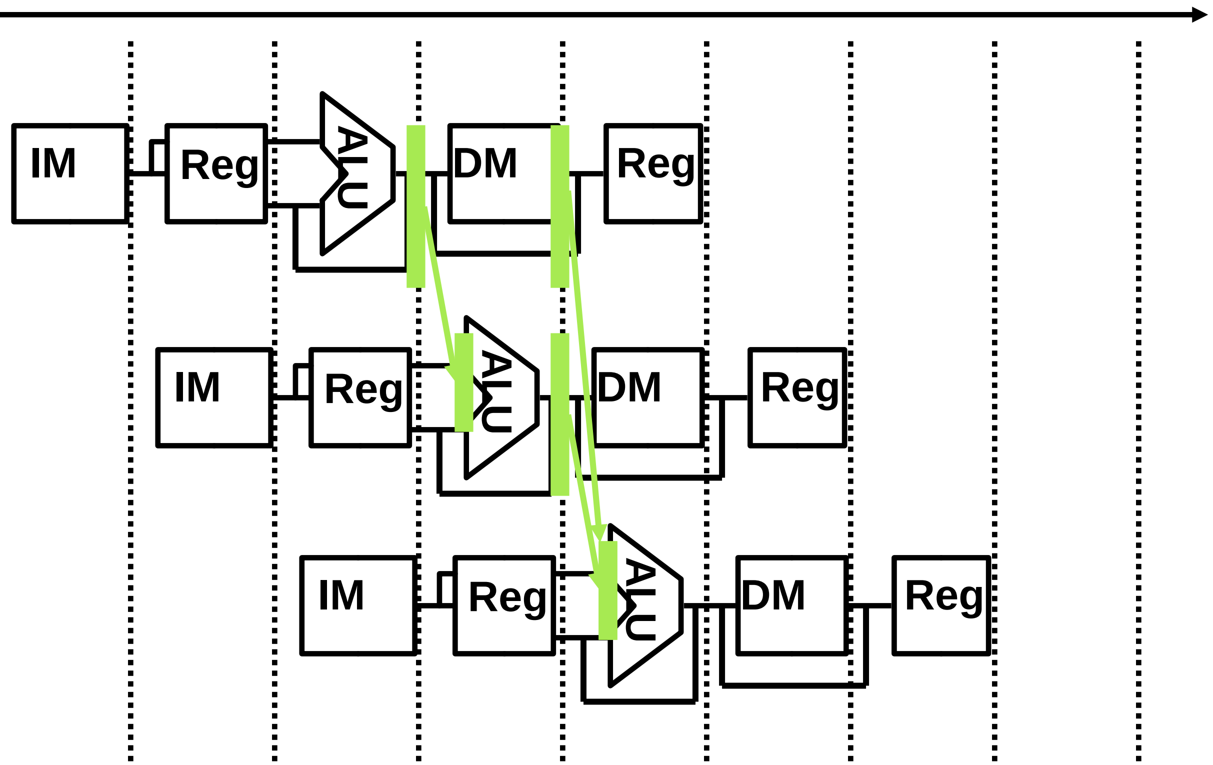
MEM/WB hazard forwarding

Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

Instr. Order

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```



Corrected Data Forwarding Control Conditions

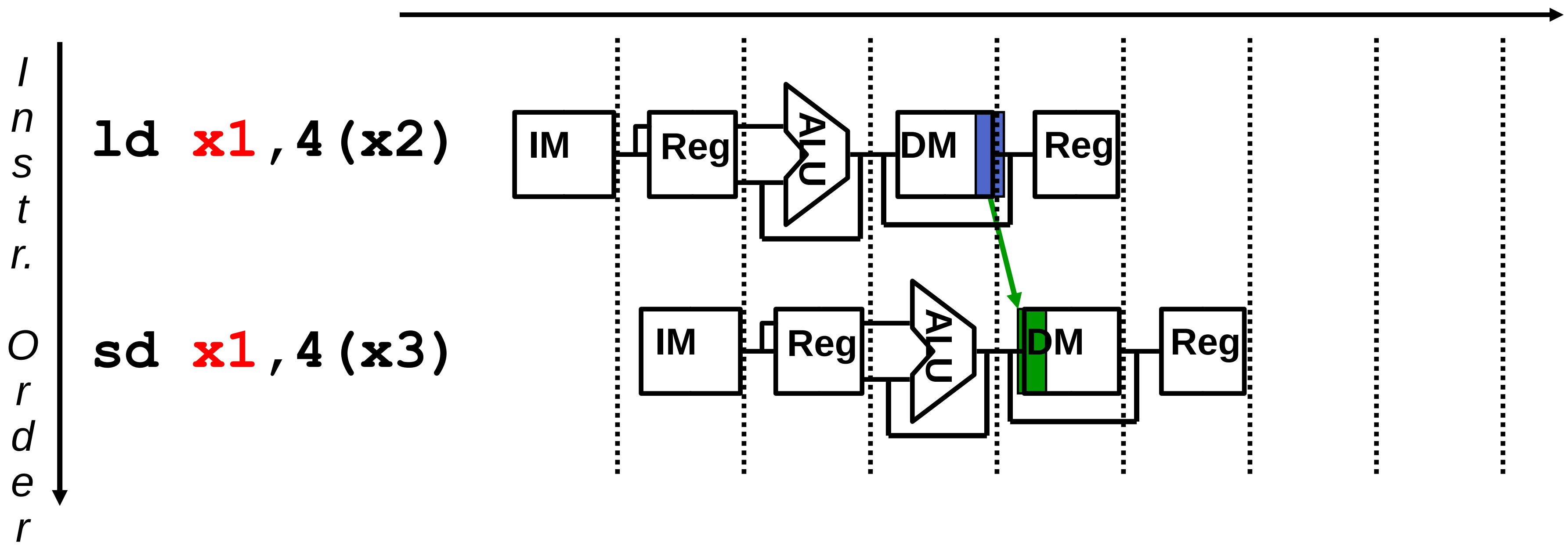
2. MEM/WB hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
     ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs1)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
     ForwardB = 01
```

Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit and a mux to the memory access stage



Instr. Order

```
ld x1,4(x2)
sd x1,4(x3)
```

Forwarding with Load-use Data Hazards

Instr. Order

ld x1,4(x2)

stall

sub x4,x1,x5

sub x4,x1,x5

and x6,x1,x7

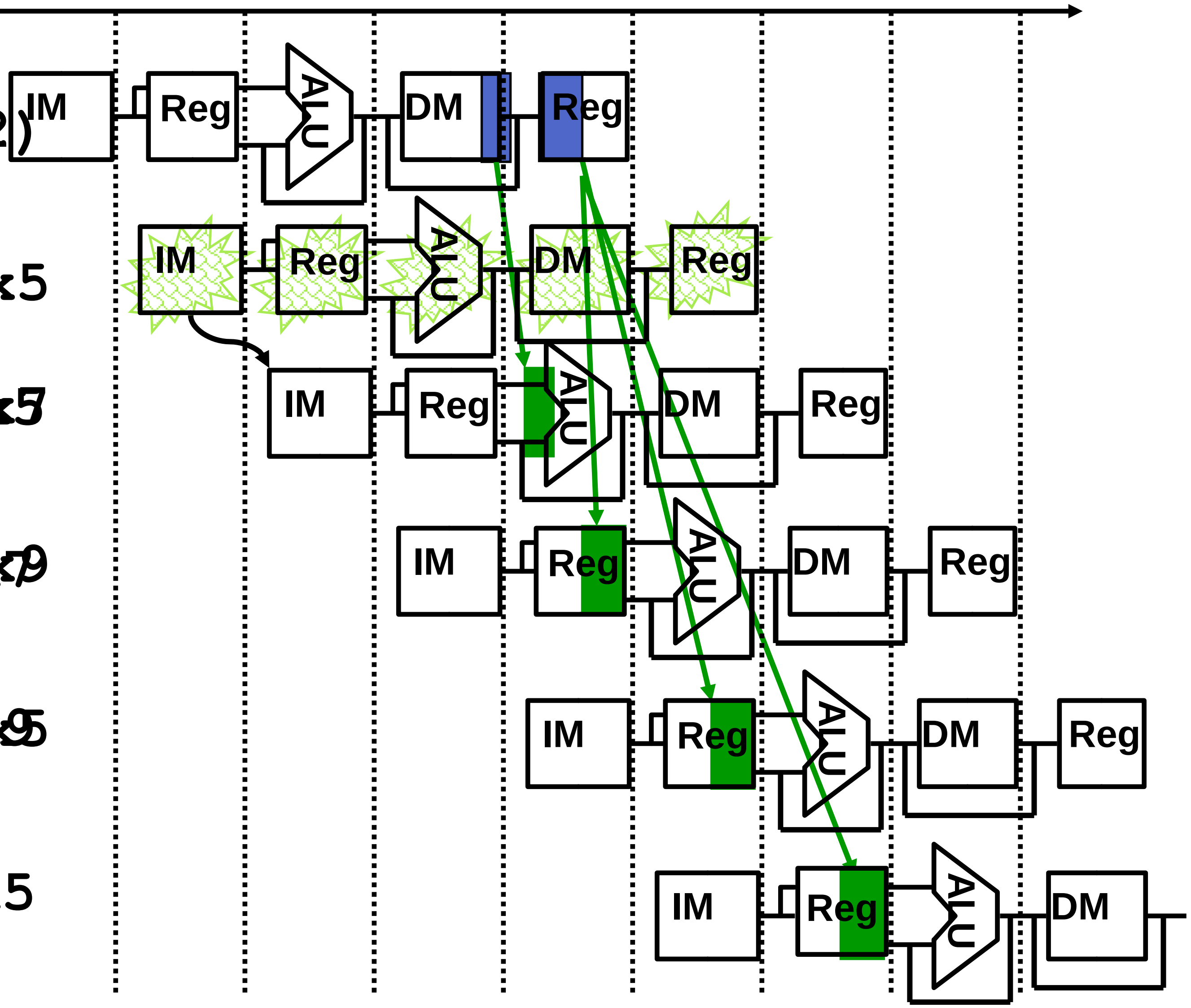
and x6,x1,x7

or x8,x1,x9

or x8,x1,x9

xor x4,x1,x5

xor x4,x1,x5



Load-use Hazard Detection Unit

- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

2. ID Hazard Detection

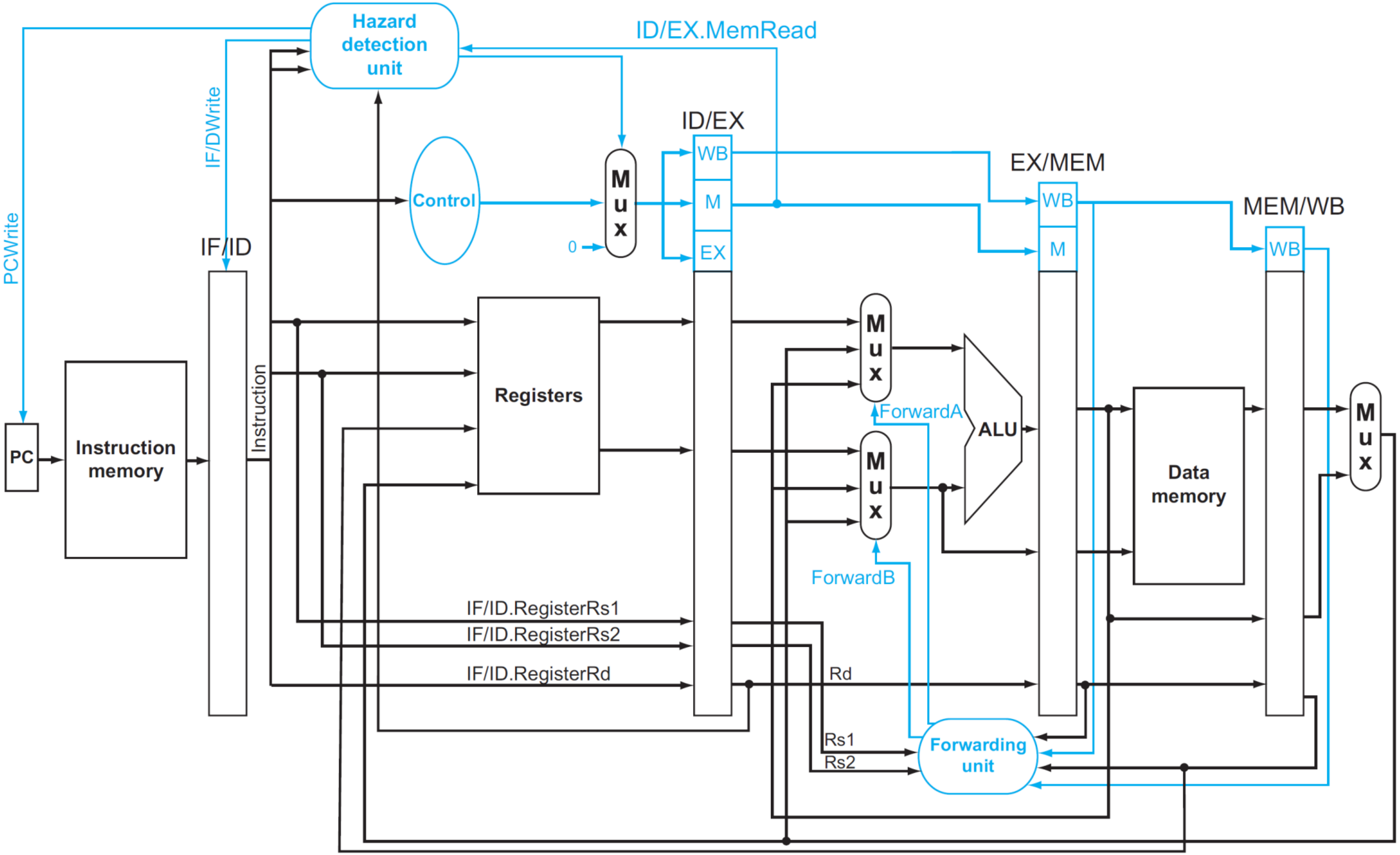
```
if (ID/EX.MemRead
and ((ID/EX.RegisterRd = IF/ID.RegisterRs1)
or   (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
stall the pipeline
```

- The first line tests to see if the instruction now in the EX stage is a `ld`; the next two lines check to see if the destination register of the `ld` matches either source register of the instruction in the ID stage (the load-use instruction)
- After this one cycle stall, the forwarding logic can handle the remaining data hazards

Stall Hardware

- Along with the Hazard Unit, we have to implement the stall
- Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection Unit controls the writing of the PC (`PC.write`) and IF/ID (`IF/ID.write`) registers
- Insert a "bubble" between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). The Hazard Unit controls the mux that chooses between the real control values and the 0's.
- Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

Datapath Modifications with Forwarding and Hazard Detection

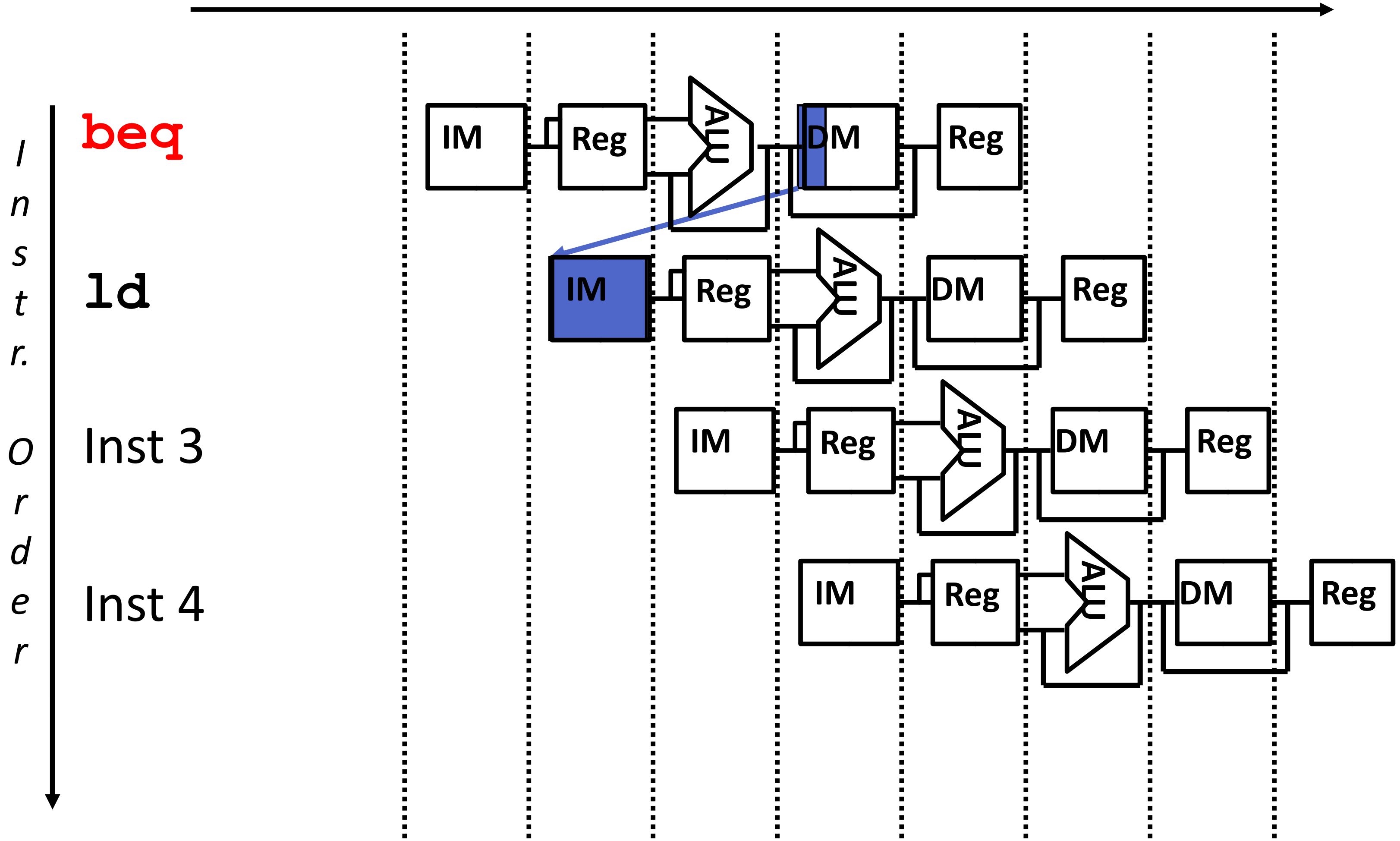


Control Hazards

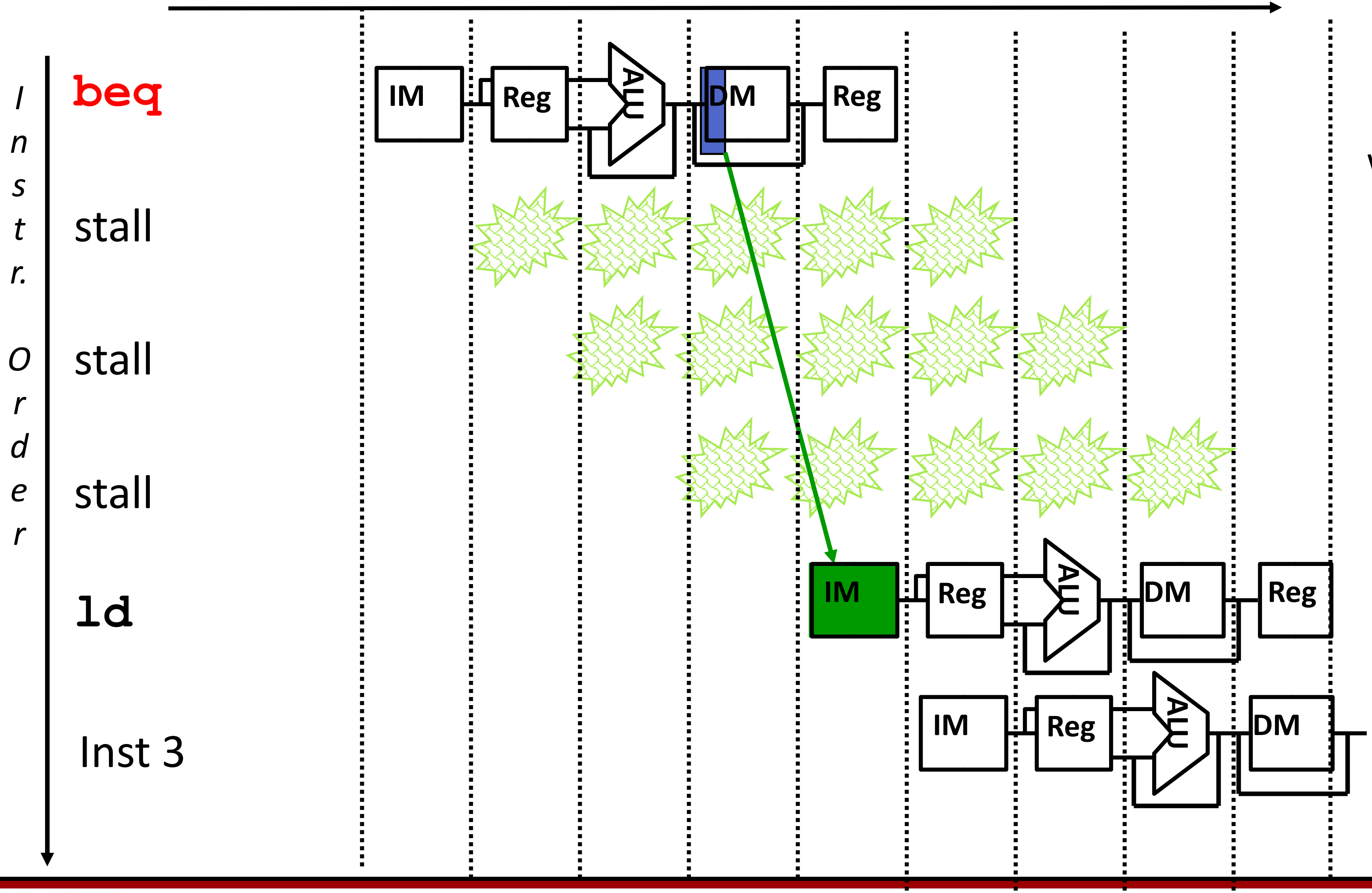
- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Conditional branches (`beq`)
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !
- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



One Way to “Fix” a Control Hazard



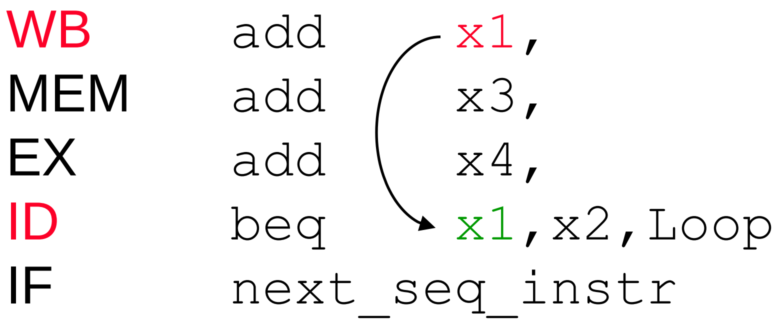
Fix branch hazard by waiting – stall – but affects CPI

Moving Branch Decisions Earlier in Pipe

- Design: Add hardware to compute the branch target address and evaluate the branch decision to the ____ stage
  - Reduces the number of stall (flush) cycles to ____ ?
    - But now need to add forwarding hardware in ID stage
  - Computing branch target address can be done in parallel with RegFile read
  - Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an `and` gate to the ID timing path
- For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

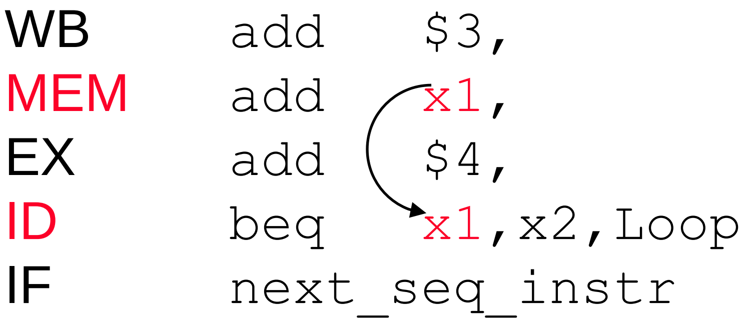
ID Branch Forwarding Issues

- MEM/WB "forwarding" is taken care of by the normal RegFile write before read operation



WB add x1,
MEM add x3,
EX add x4,
ID beq x1,x2,Loop
IF next_seq_instr

- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like



WB add $3,
MEM add x1,
EX add $4,
ID beq x1,x2,Loop
IF next_seq_instr

```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs1))
    ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs2))
    ForwardD = 1
```

Forwards the result from the second previous instr. to either input of the compare

ID Branch Forwarding Issues, con't

- If the instruction immediately before the branch produces one of the branch source operands, then a stall needs to be inserted (between the beq and add) since the EX stage ALU operation is occurring at the *same time* as the ID stage branch compare operation

```
WB     add     x3,
MEM    add     x4,
EX     add     x1,
ID     beq     x1,$2,Loop
IF     next_seq_instr
```

  - "Bounce" the beq (in ID) and next_seq_instr (in IF) in place (ID Hazard Unit deasserts PC.Write and IF/ID.Write)
  - Insert a stall between the add in the EX stage and the beq in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)

Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
    - If taken, flush instructions after the branch (earlier in the pipeline)
        - in IF, ID, and EX stages if branch logic in MEM – three stalls
        - in IF and ID stages if branch logic in EX – two stalls
        - in IF stage if branch logic in ID – one stall
    - Ensure that those flushed instructions haven’t changed the machine state – automatic in the 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
    - 30-40% accuracy…not so good

ASIDE: Two “Types” of Stalls

- Noop instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert noop by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline

- Flushes (or instruction squashing) were an instruction in the pipeline is replaced with a noop instruction

Static Branch Prediction, con't

- Resolve branch hazards by assuming a given outcome and proceeding

2. Predict taken – predict branches will always be taken

    60-70% accuracy

    Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)

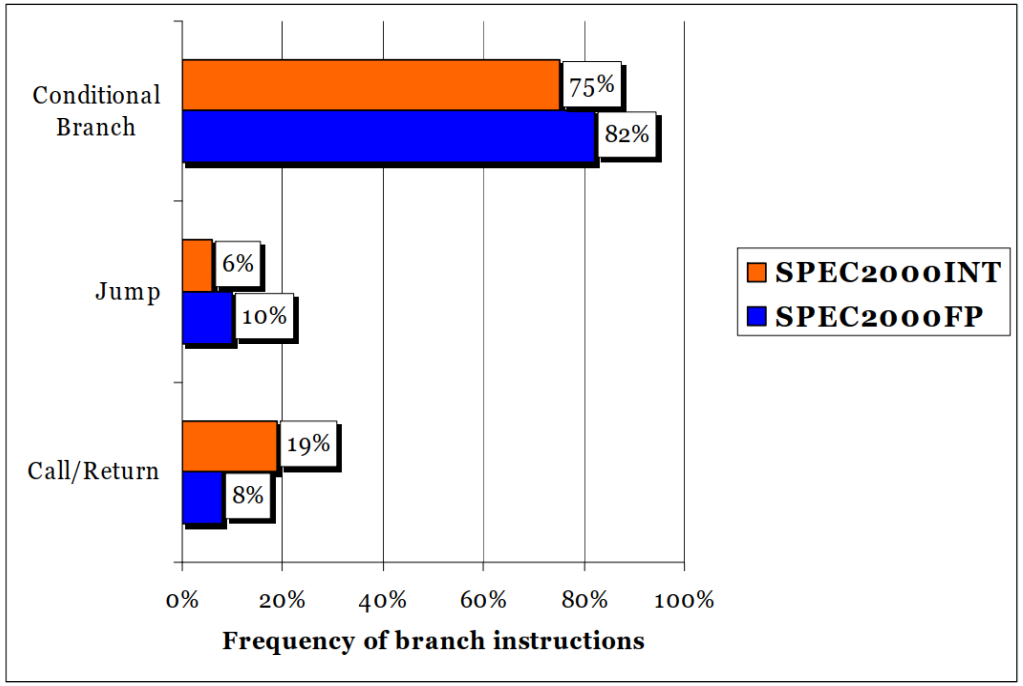
    Is there a way to "cache" the address of the branch target instruction ??

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution

3. Dynamic branch prediction – predict branches at run-time using *run-time* i nformation

Dynamic Branch Prediction

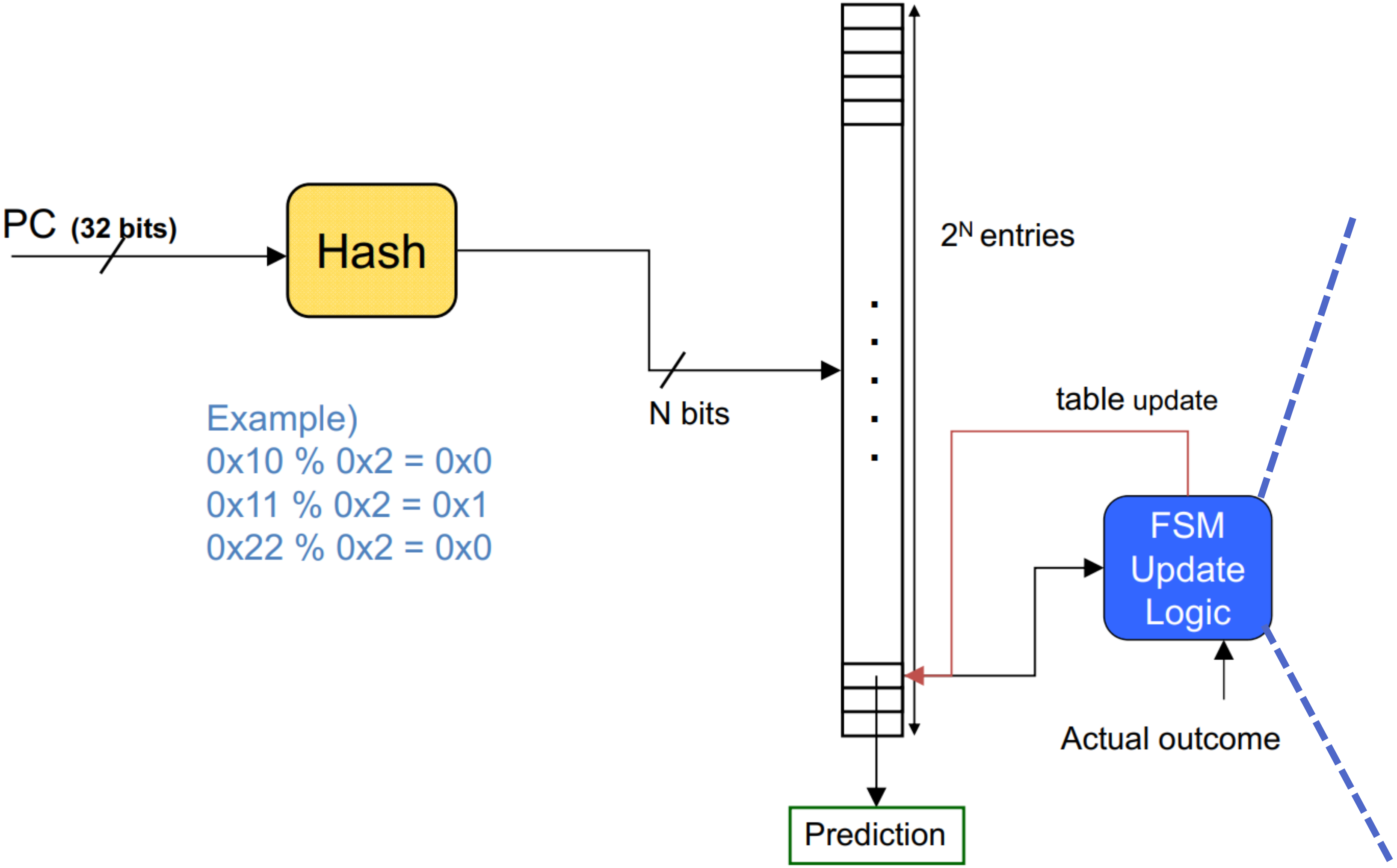
- Need to know two things
  - Whether the branch is taken or not (direction)
  - The target address if it is taken (target)
- Direct jumps, Function calls
  - Direction known (always taken), target easy to compute
- Conditional Branches (typically PC-relative)
  - Direction difficult to predict, target easy to compute
- Indirect jumps, function returns
  - Direction known (always taken), target difficult



Dynamic Branch Prediction

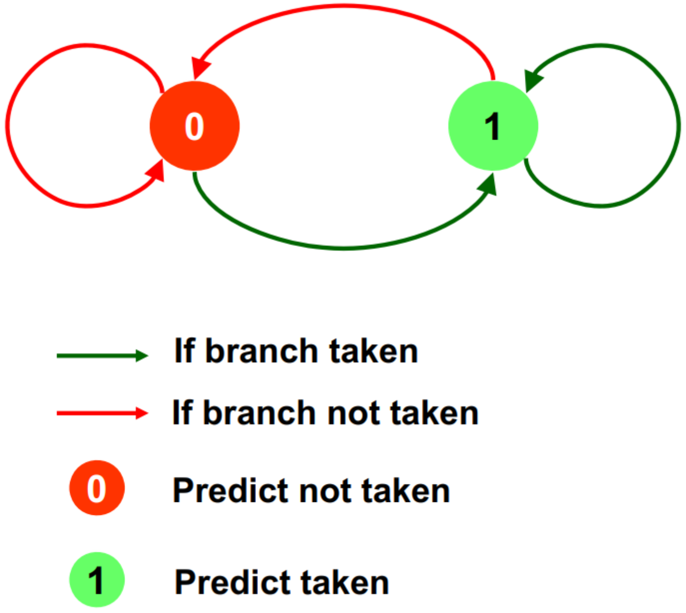
- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains a bit passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was executed
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but it doesn't affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit
    - A 4096-bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

1-bit Predictors



Example)
0x10 % 0x2 = 0x0
0x11 % 0x2 = 0x1
0x22 % 0x2 = 0x0

- A 2-state machine
- Change mind fast

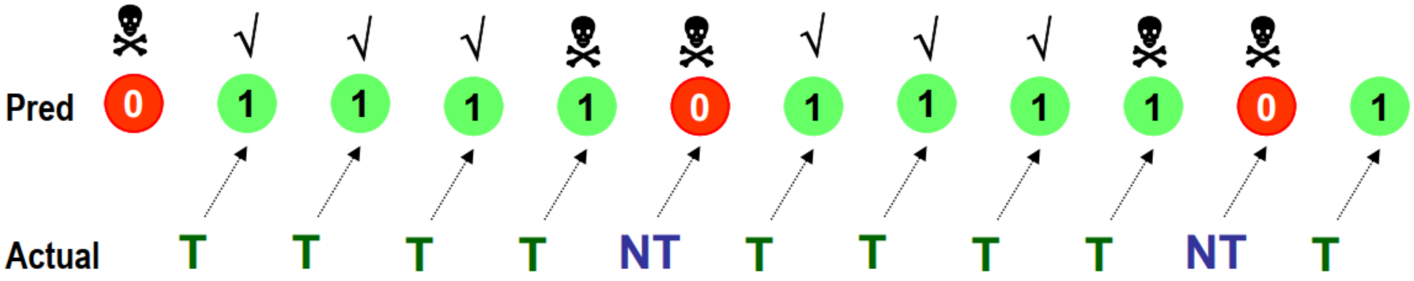


1-bit Predictors

A loop example using 1-bit branch history table

```
for (i=0; i<4; i++) {
        ....
}
```

```
 addi   r10, r0, 4
 addi   r1,   r1, r0
L1:
 ... ...
 addi   r1, r1,   1
 bne    r1, r10, L1
```

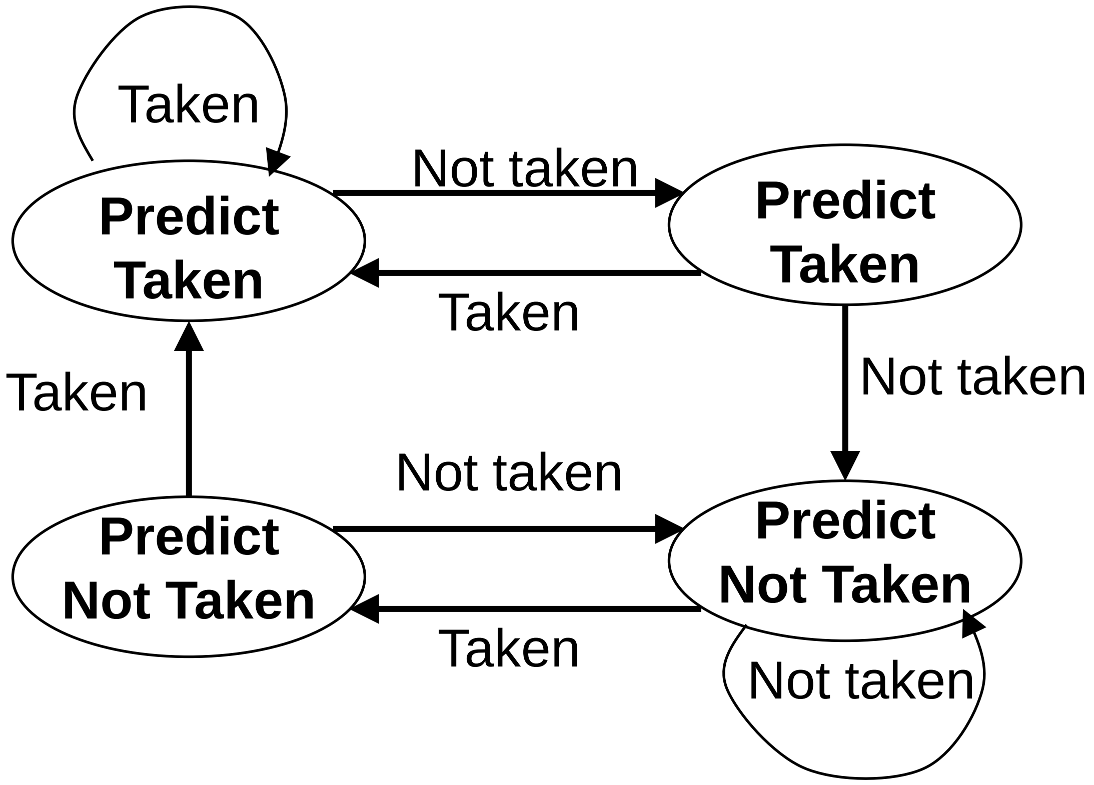


	☠	√	√	√	☠	☠	√	√	√	☠	☠	
Pred	0	1	1	1	1	0	1	1	1	1	0	1
Actual	T	T	T	T	NT	T	T	T	T	NT	T	

60% accuracy

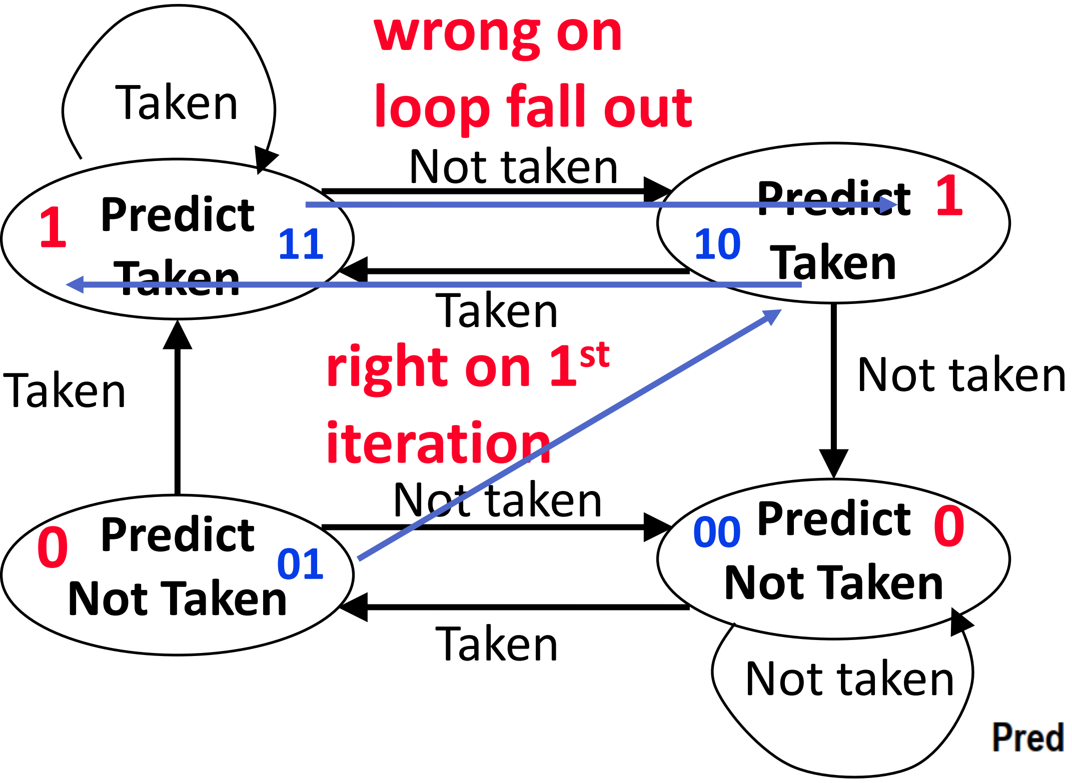
2-bit Predictors

- A 2-bit scheme can give about 80% accuracy
- A prediction must be wrong twice before the prediction bit is changed



2-bit Predictors

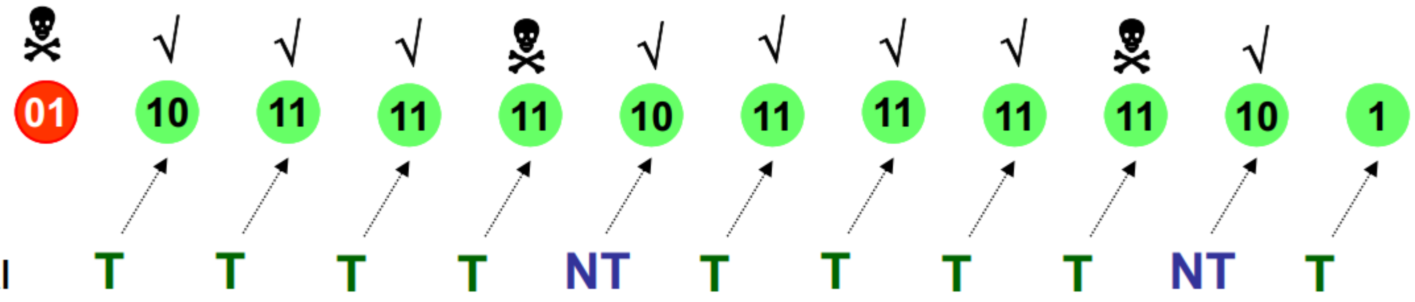
- A 2-bit scheme can give about 80% accuracy
- A prediction must be wrong twice before the prediction bit is changed



wrong on loop fall out

right on 1st iteration

- ➢ BHT also stores the initial FSM state

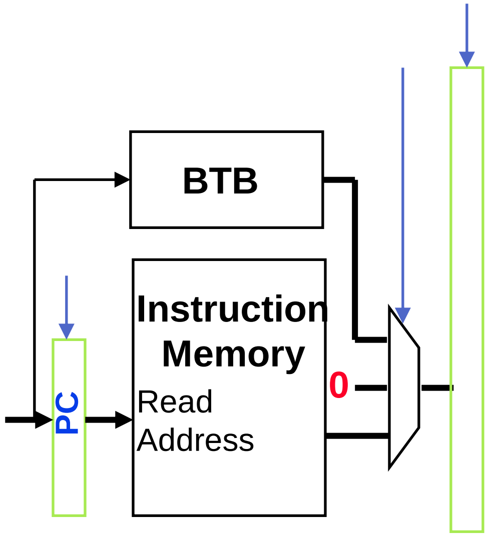


	☠	√	√	√	☠	√	√	√	√	☠	√	
Pred	01	10	11	11	11	10	11	11	11	11	10	1
Actual	T	T	T	T	NT	T	T	T	T	NT	T	

Branch Target Buffer

- 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 can cache 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

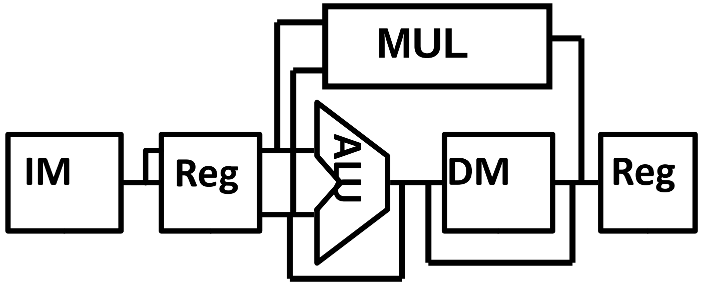
Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



- If the prediction is correct, stalls can be avoided no matter which direction they go

Other Pipeline Structures Are Possible

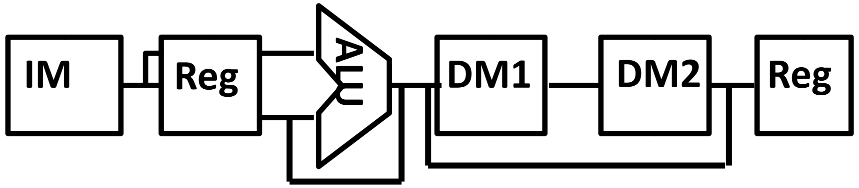
- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)



- 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)



Summary

- All modern-day processors use pipelining for performance (a CPI of 1 and fast a CC)
- Pipeline clock rate limited by slowest pipeline stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Delay decision (requires compiler support)
    - Static and dynamic prediction (requires hardware support)