ECE-4300: Chapter-4: Processor (Cont.)

Prof: Mohamed El-Hadedy

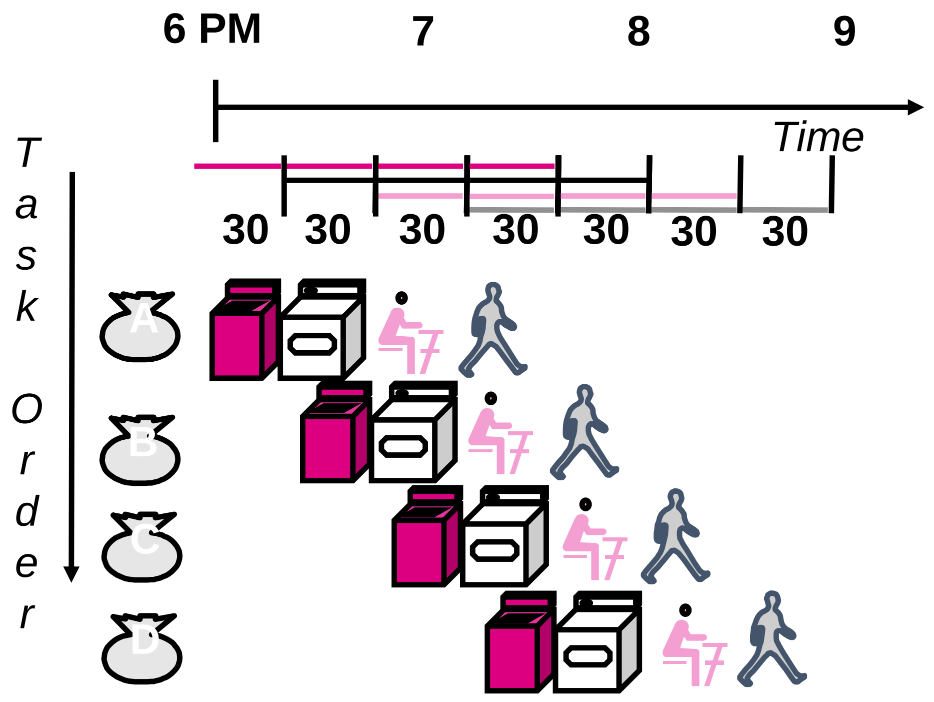
Email: mealy@cpp.edu

Office: 909-869-2594

Introduction

- Pipelining is an implementation technique:
  - Instructions are overlapped in time
  - similar to a manufacturing assembly line
  - employed to gain performance
  - Speed-up ~ = (non-pipelined MIPS) * (# of pipeline stages)

Pipelining Lessons

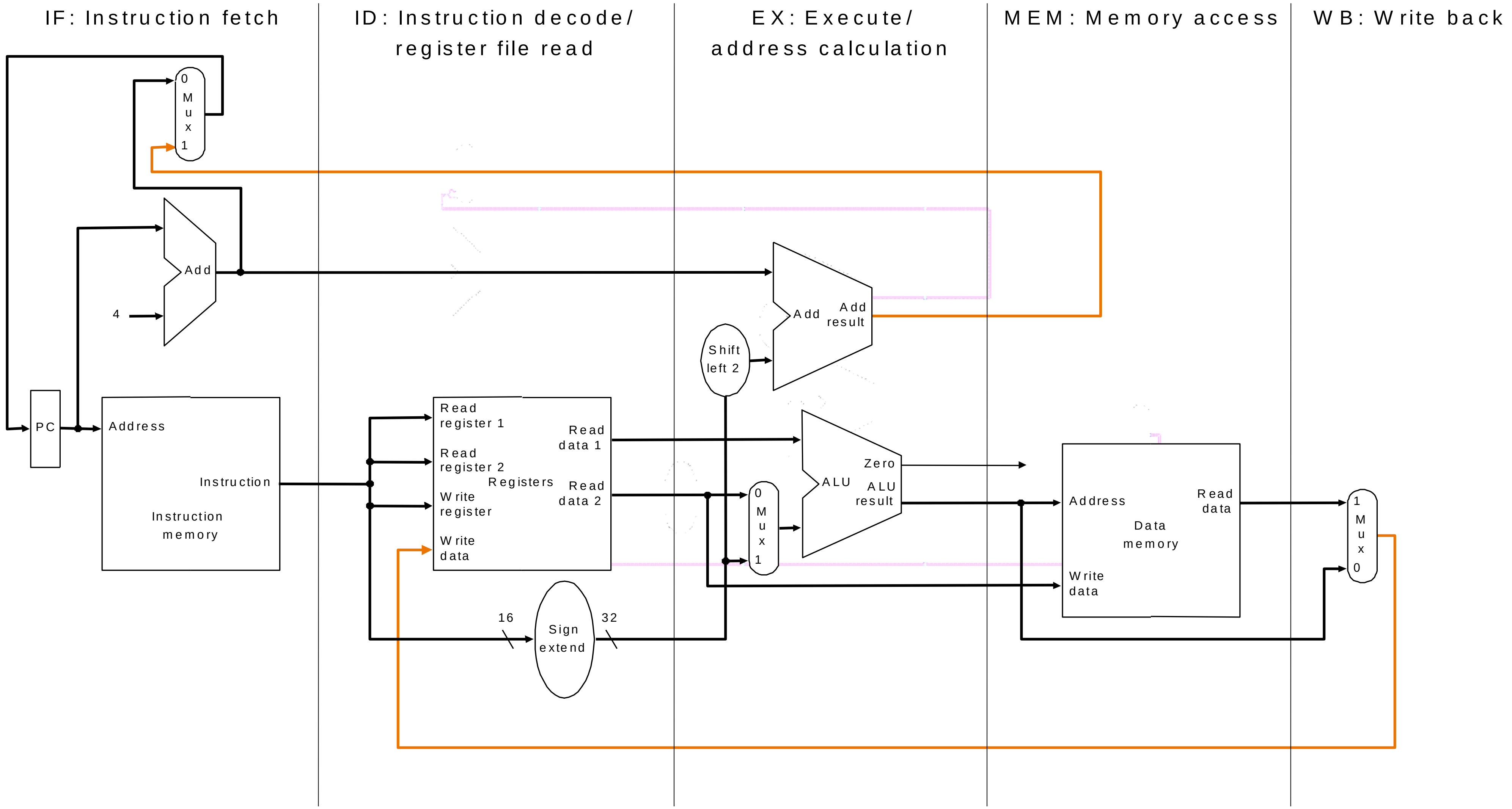


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

MIPS Pipeline

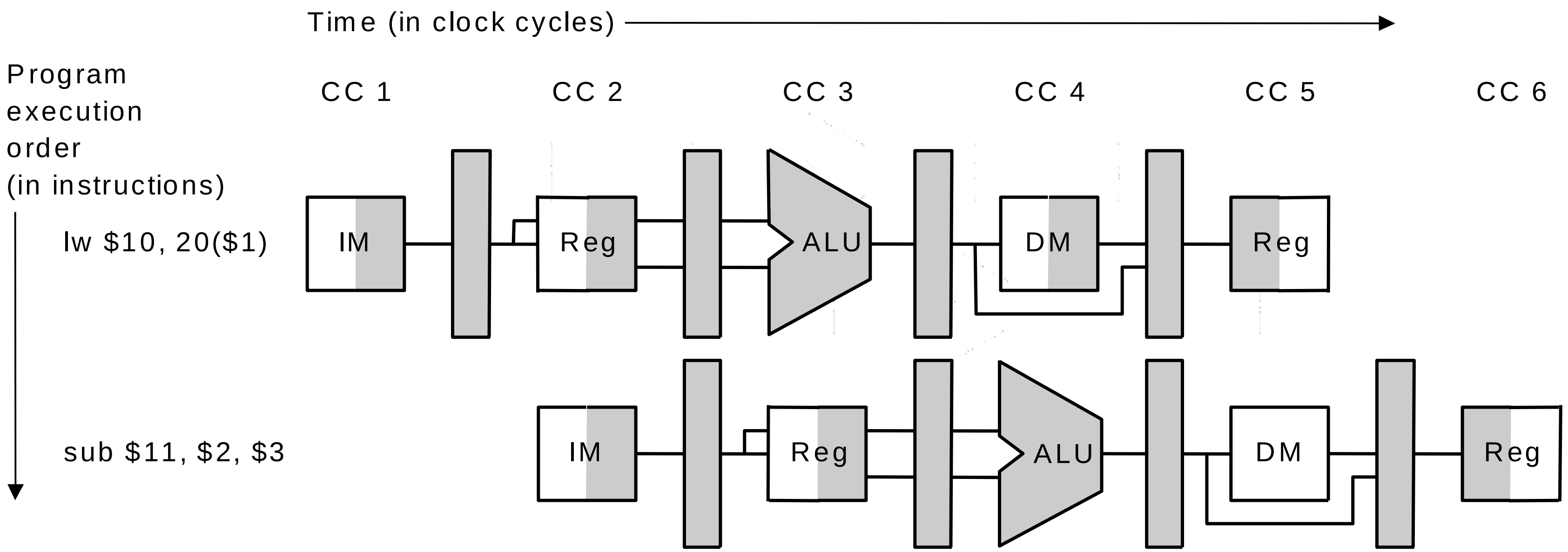
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Basic Idea



What do we need to add to actually split the datapath into stages?

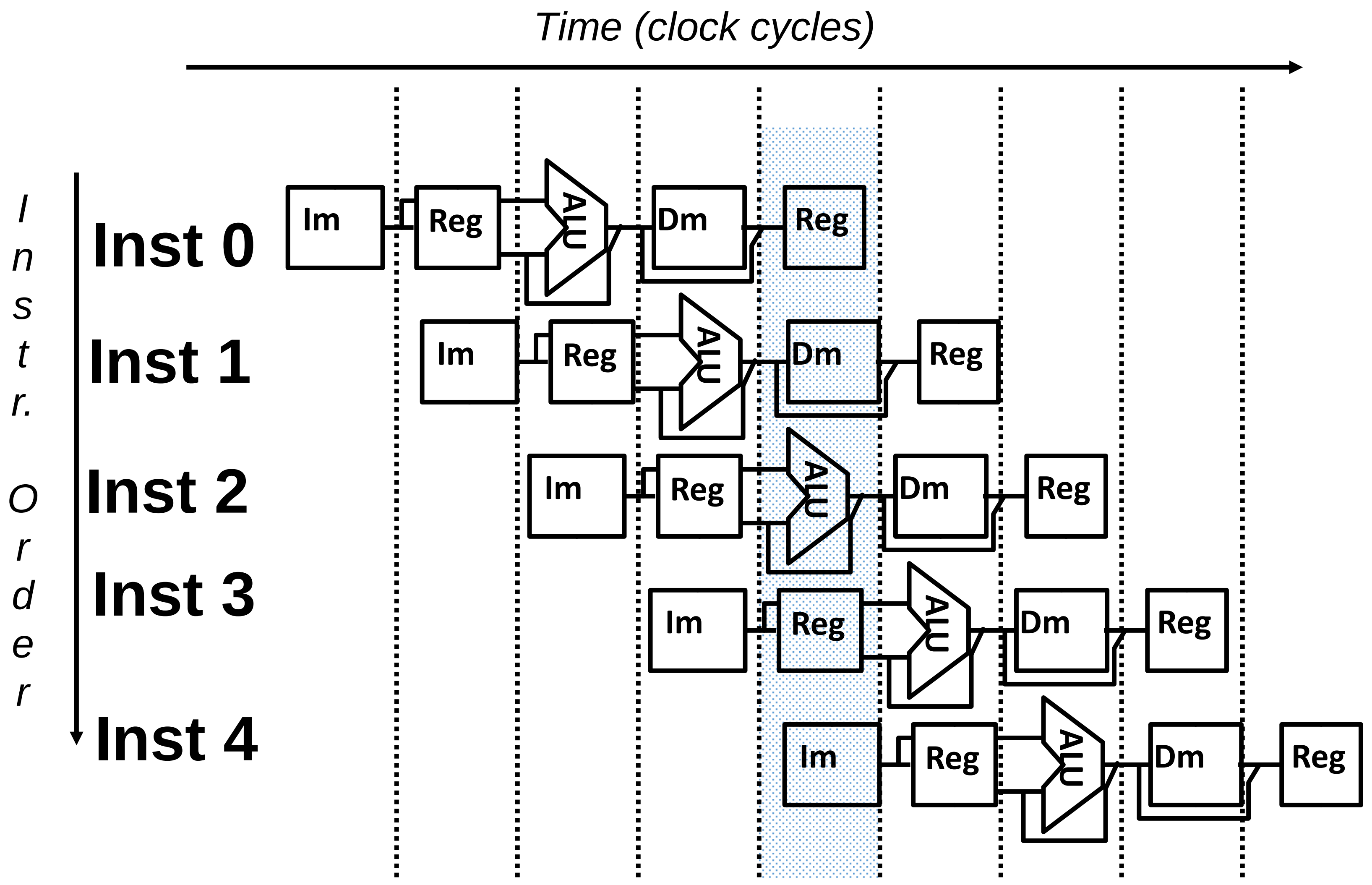
Pipelines



- How many cycles does it take to execute this code?
- Writes to RF occurs in the first half and reads from RF occurs in the second half of the cycle.

Why Pipeline?

- **Because the resources are there**



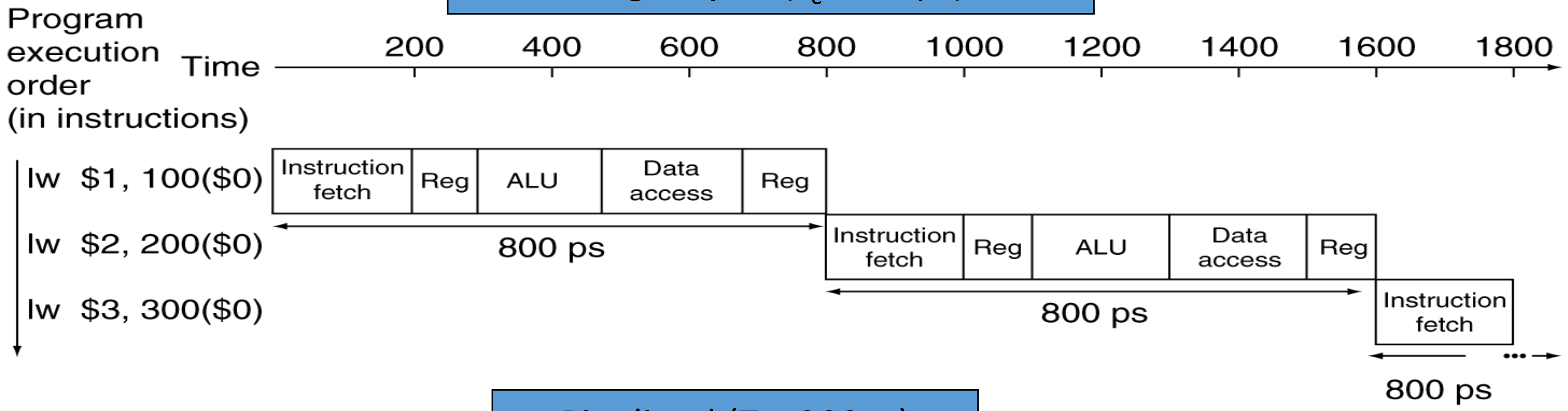
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

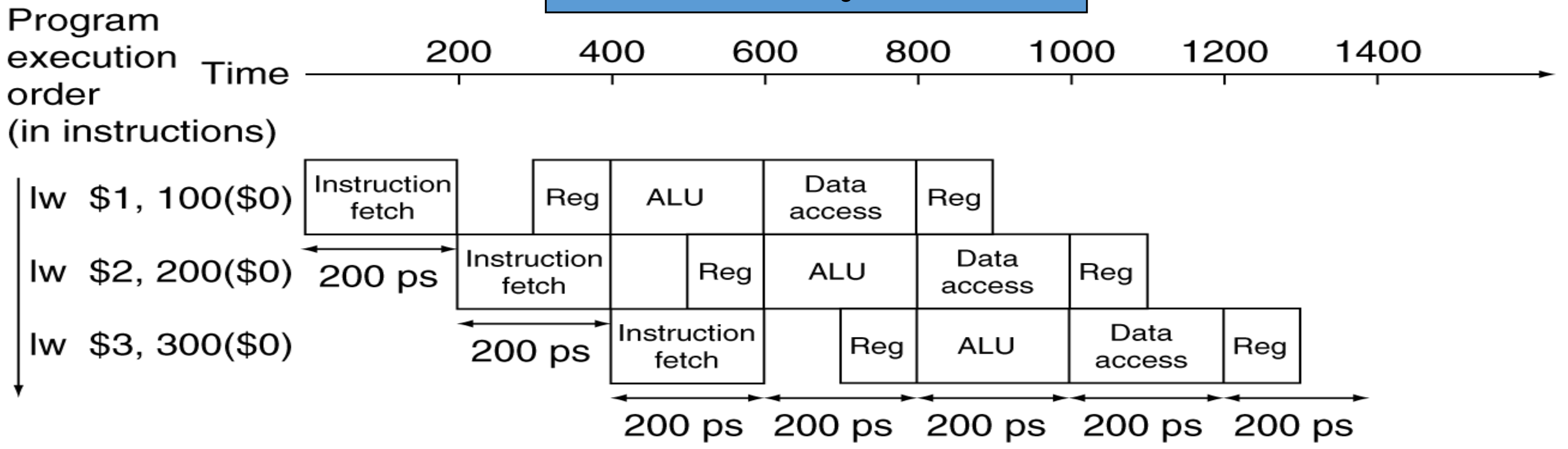
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

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

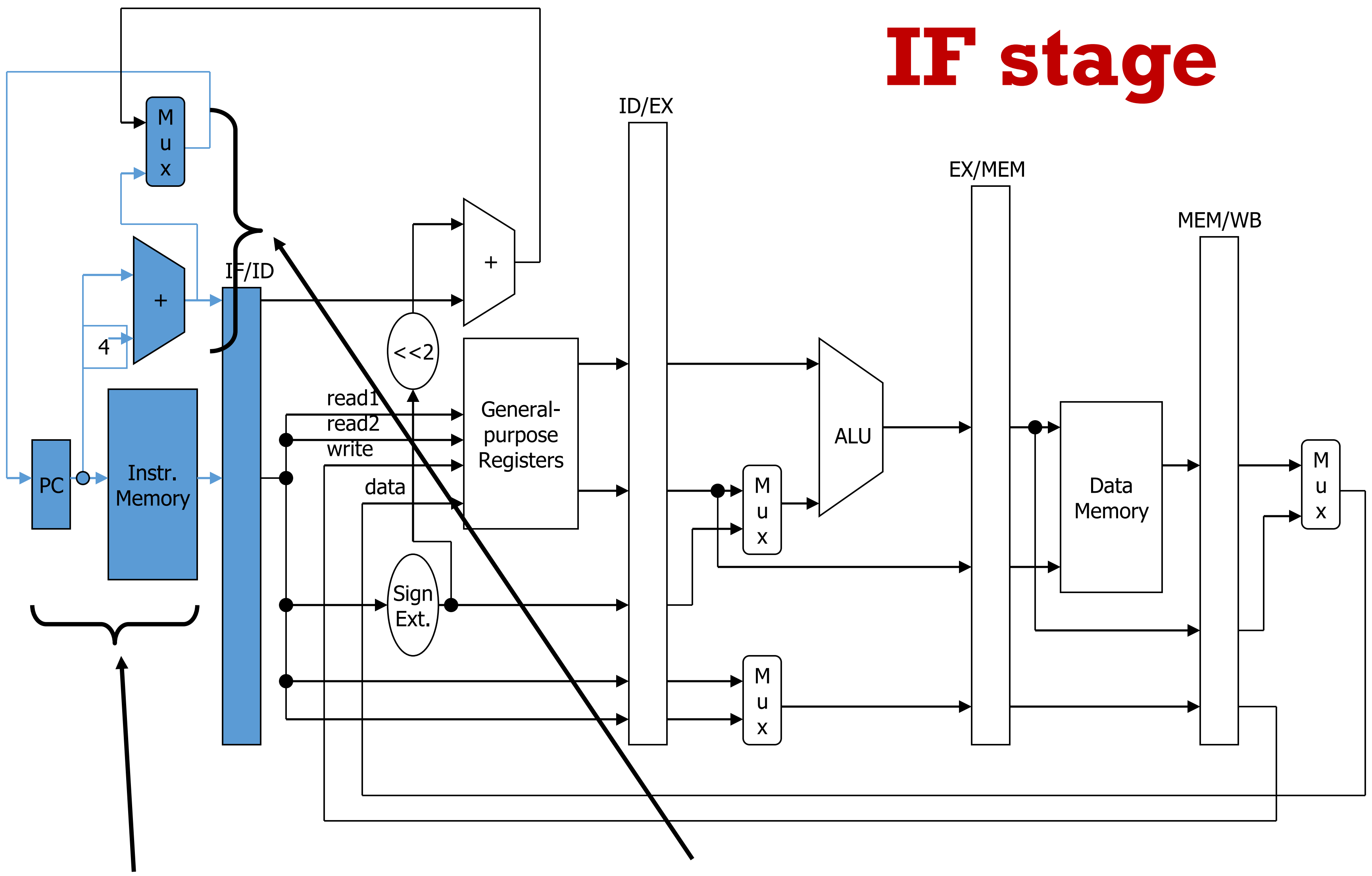
$$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

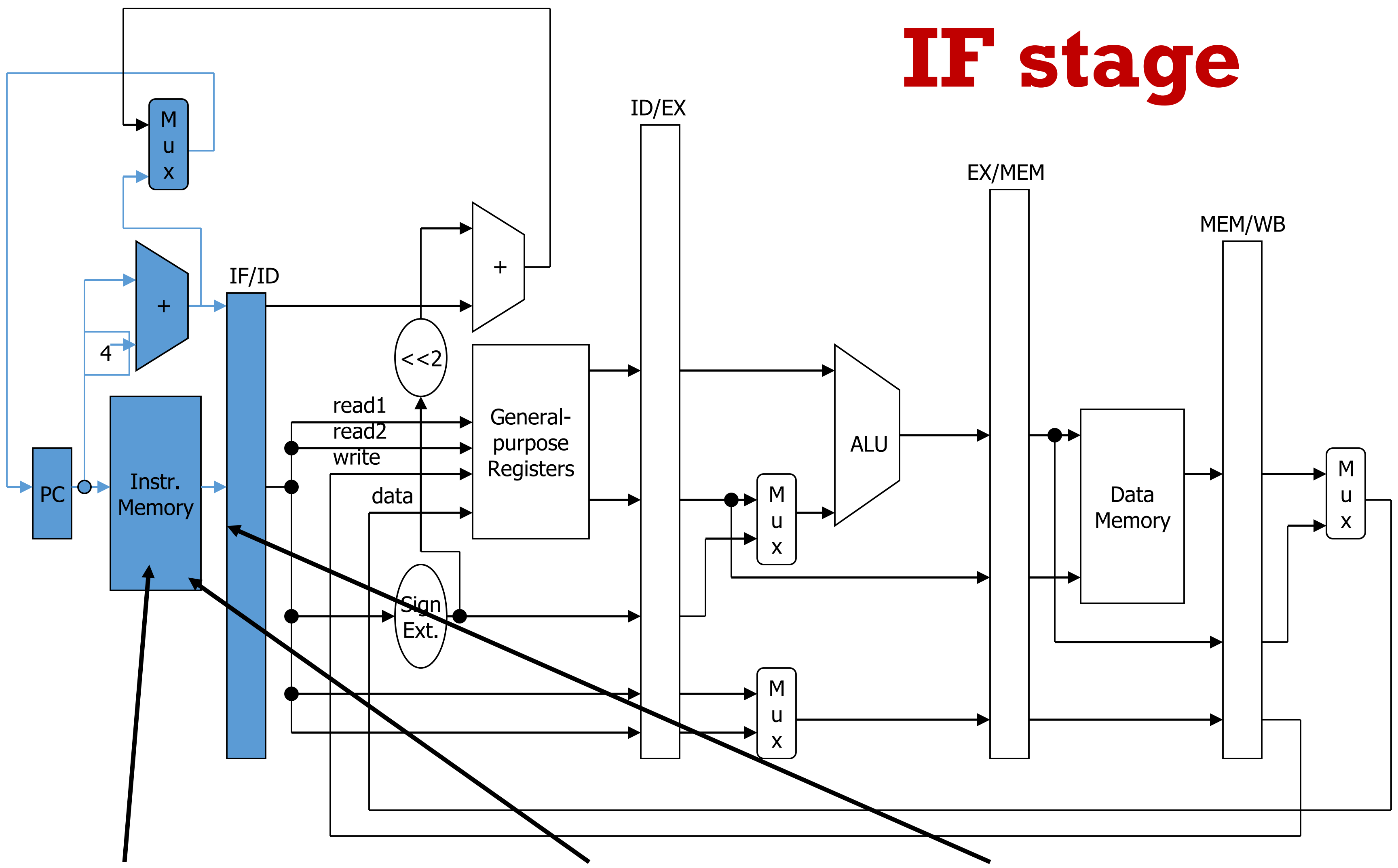
IF stage



Fetch instruction, Mem[PC].

New PC = either PC+4 or branch target of previous instruction.

IF stage

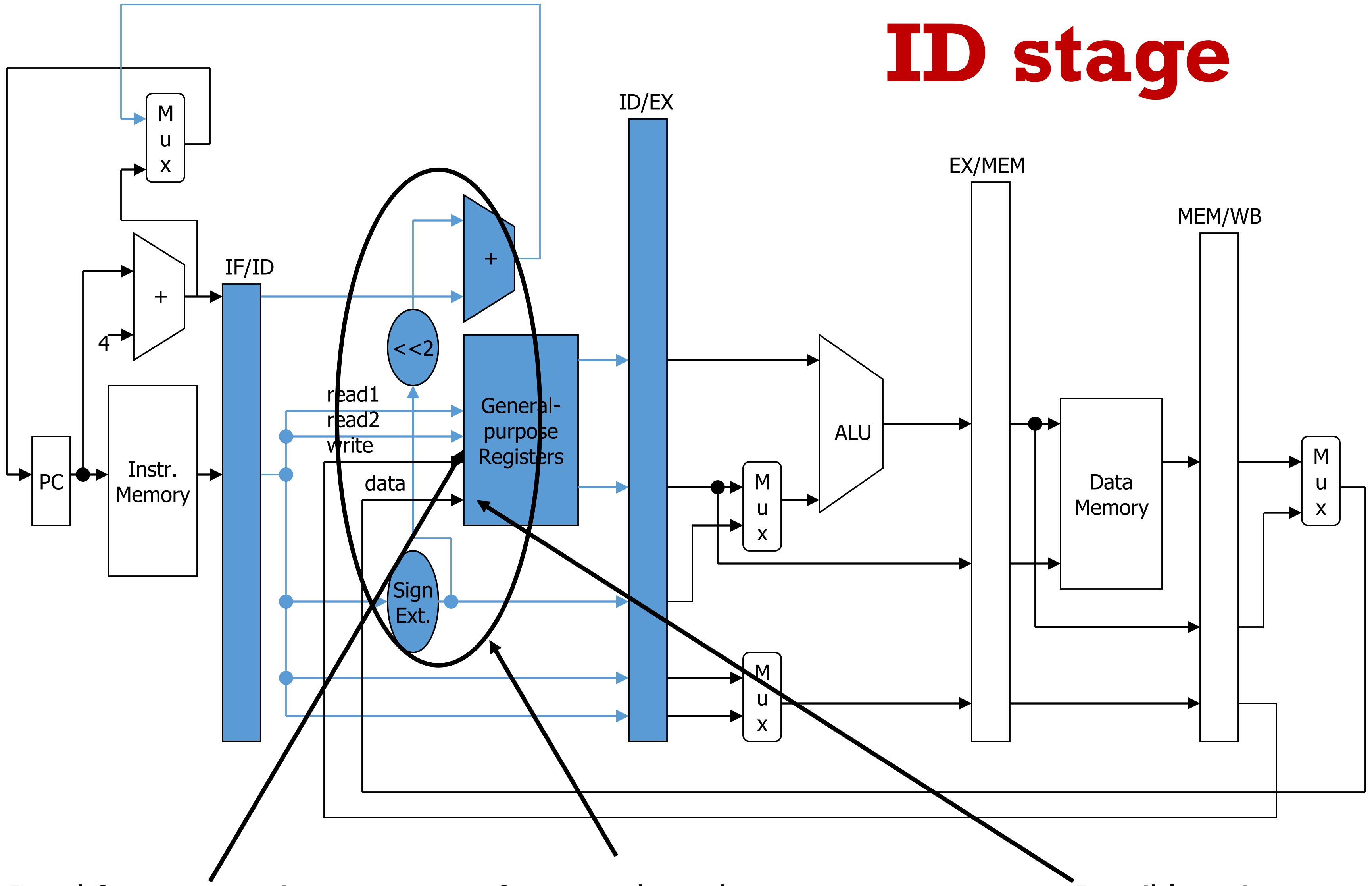


Separate instruction & data memories is a convenient. **Why?**

Accessing memory

Results of each stage must be stored in registers for next stage. **Why?**

ID stage

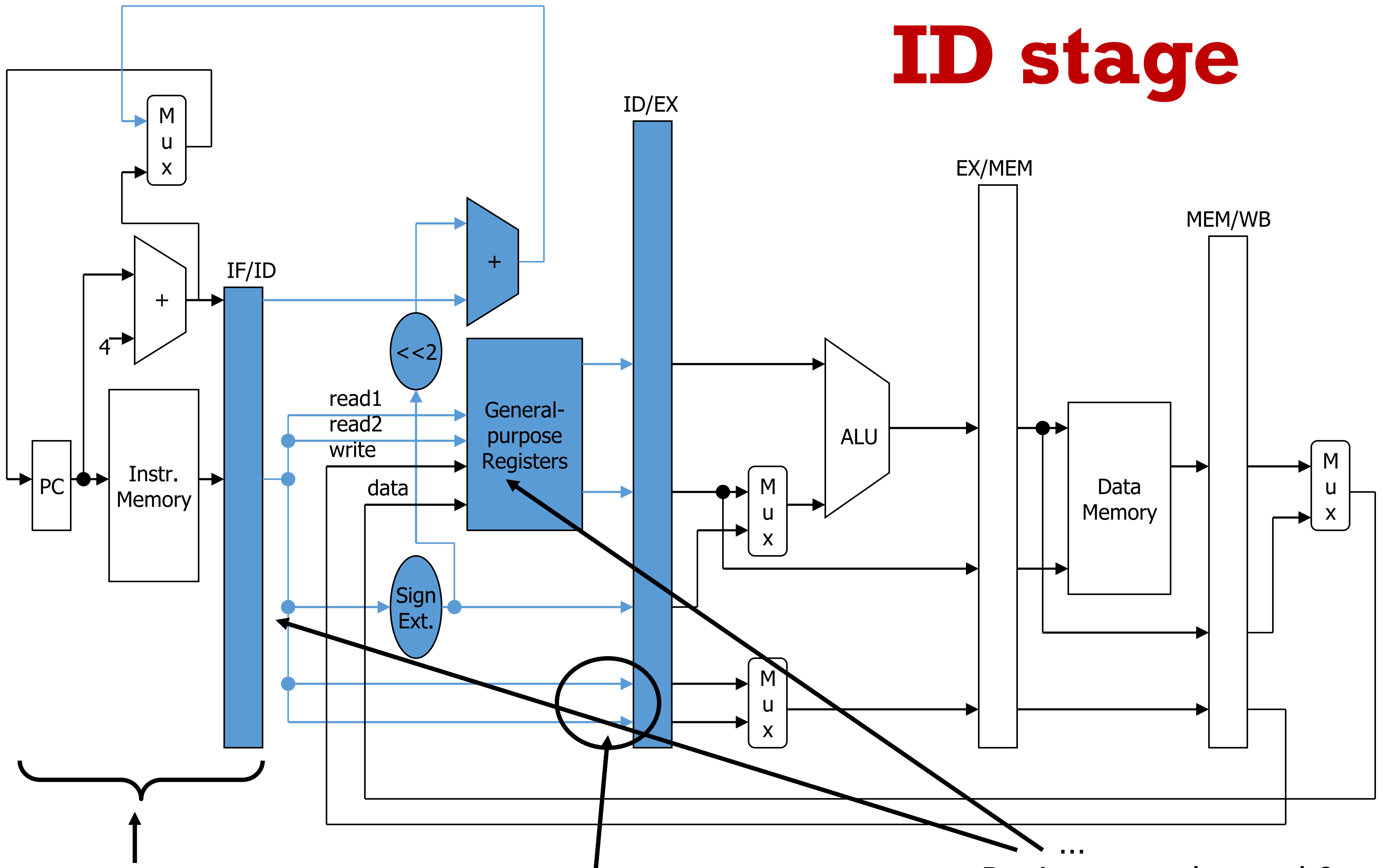


Read 2 source registers.
(Even if instr. only uses 1.)

Compute branch target =
PC + 4 + Sign-ext(address)<<2.

Possibly write
destination register
of earlier instruction.

ID stage

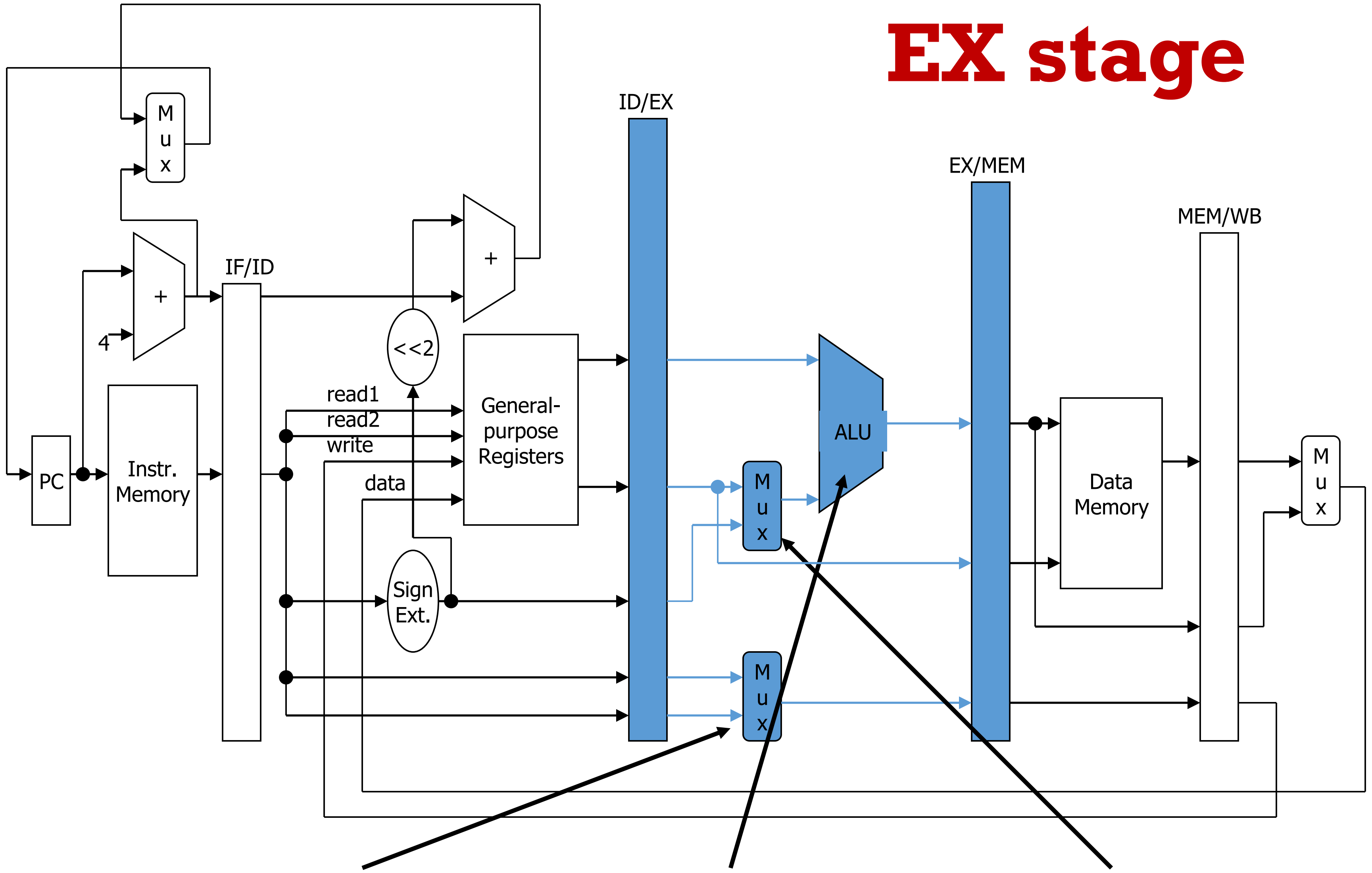


Remember, next instruction executing IF stage.

Store destination register for possible later use.

Registers can be read & written in same cycle.

EX stage

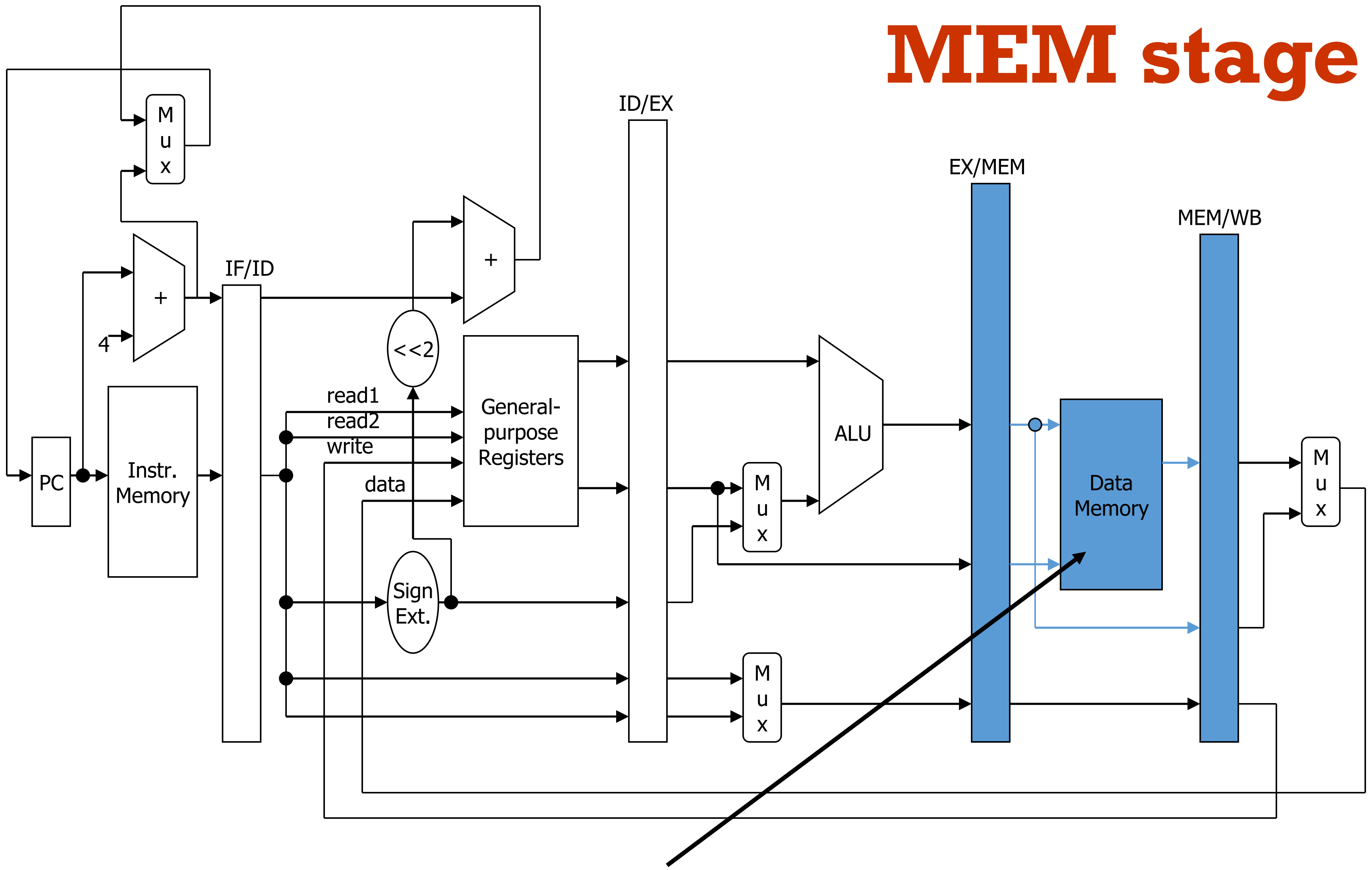


Choose which bits specify destination register.

Compute arithmetic/logic, address calculation, or condition testing…

…where 2nd operand is from register or immediate.

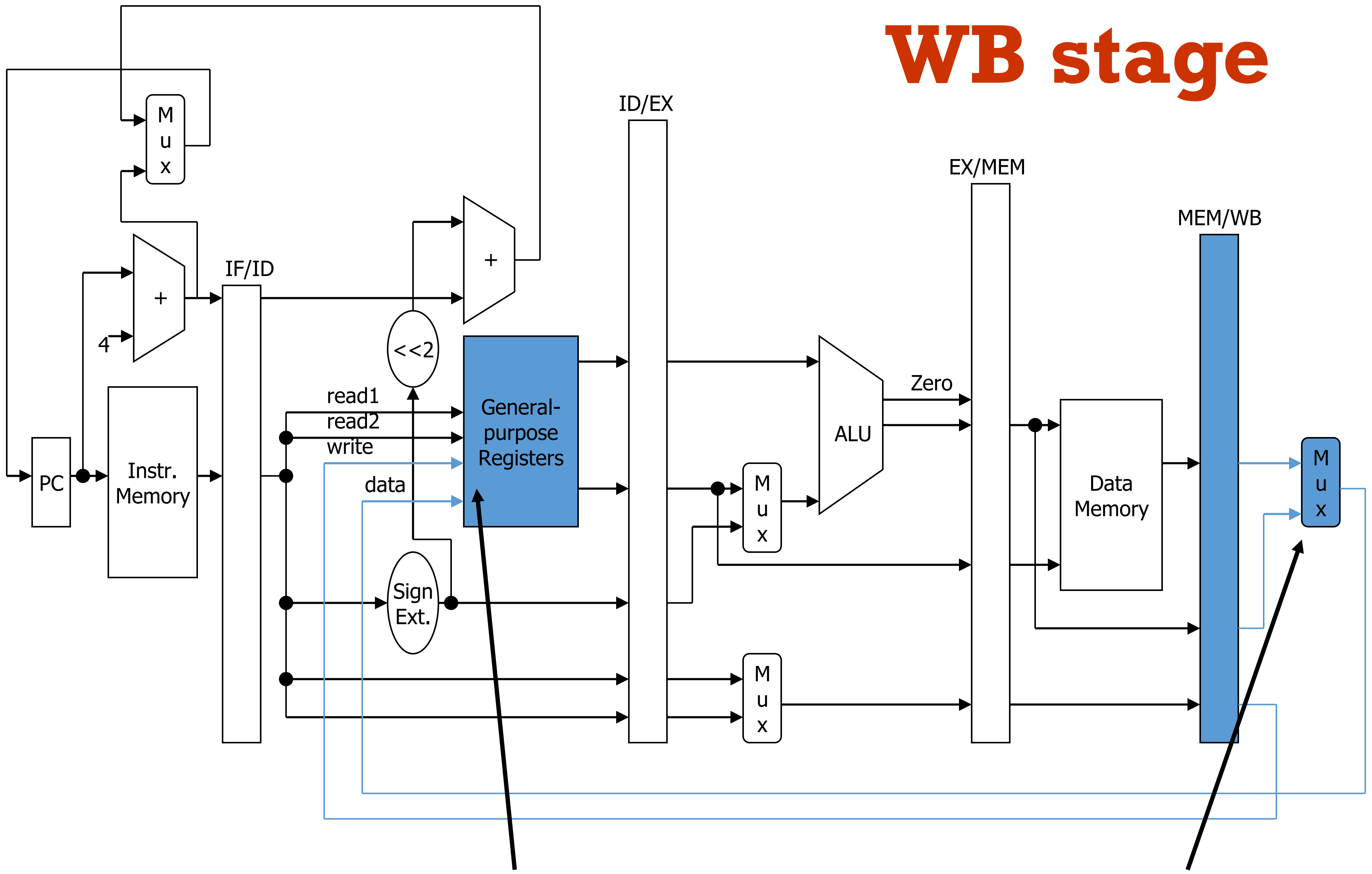
MEM stage



Possibly read or write memory.

Accessing memory

WB stage



Possibly write result into register.

Choose result from either memory or ALU.

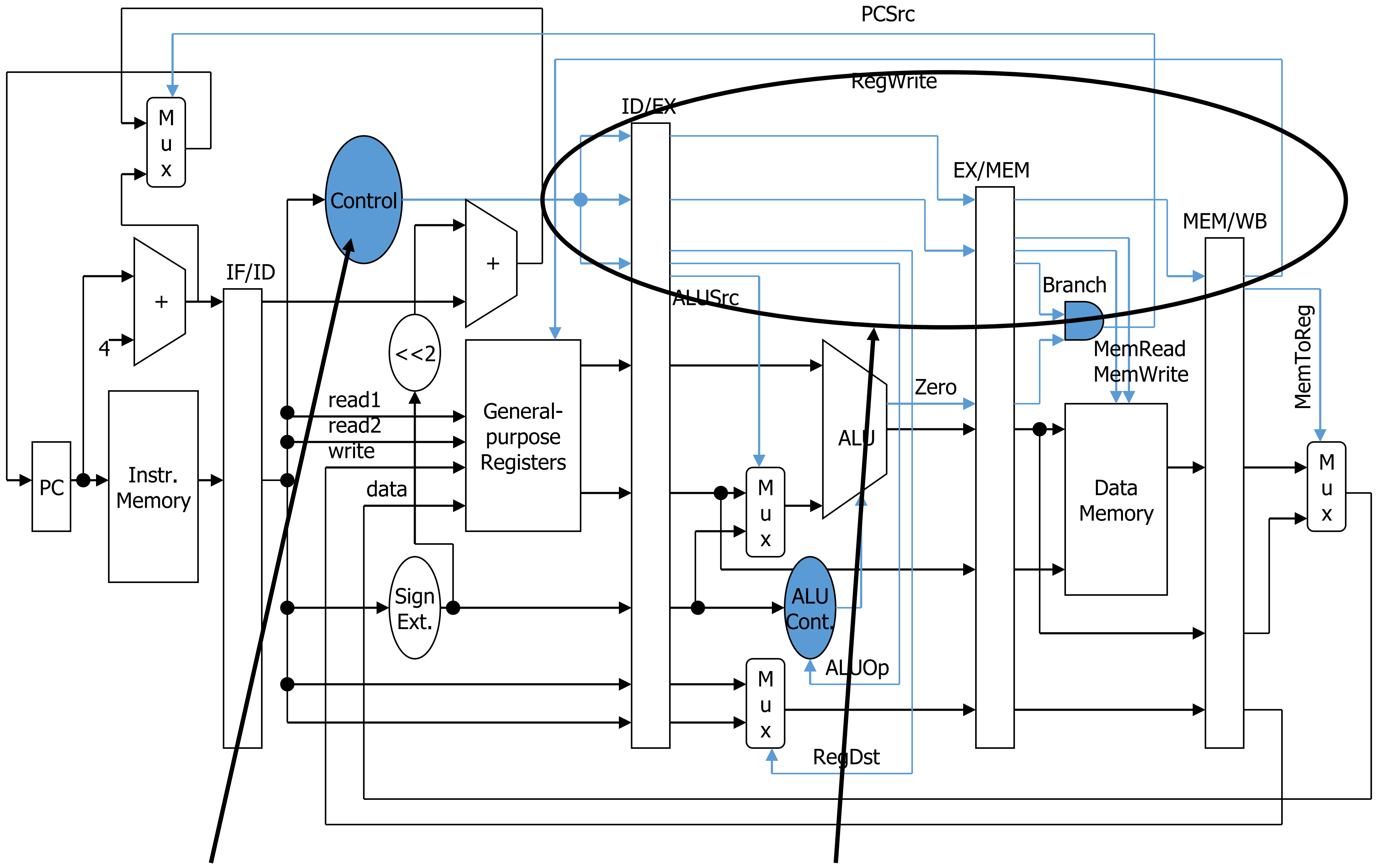
Pipeline Datapath

How would we add unconditional branches?

ID: Add adder to compute address.

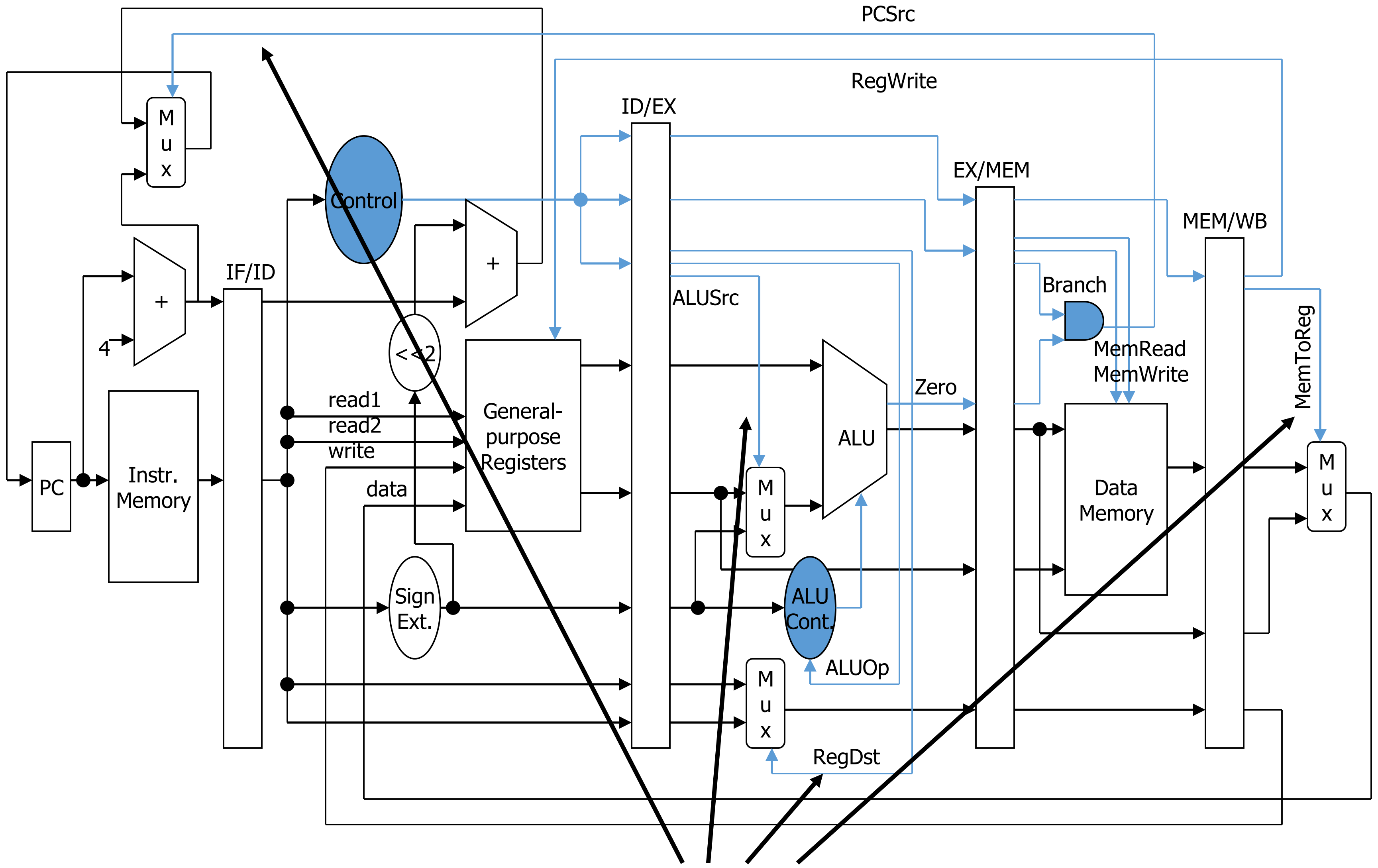
IF: Expand MUX that chooses PC's new value to have this as a third option.

Control logic activates appropriate functional units in each stage

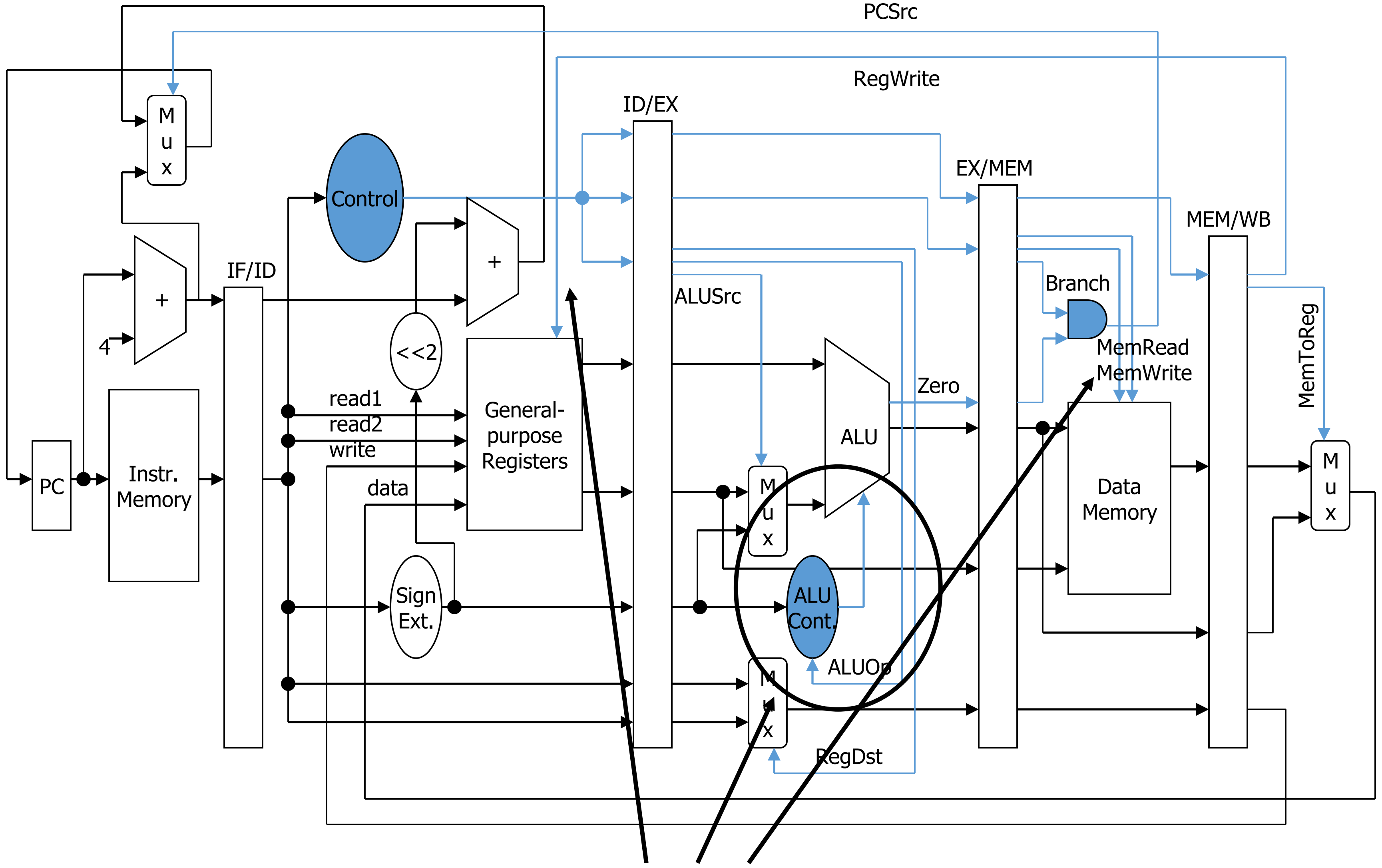


Control unit's simple combinational logic determines control bits from instruction…

…which are saved until appropriate stage.



Control bits are sent to all multiplexors to make choices in datapath.



Control bits are sent to some functional units to specify operations.

Pipeline Hazards

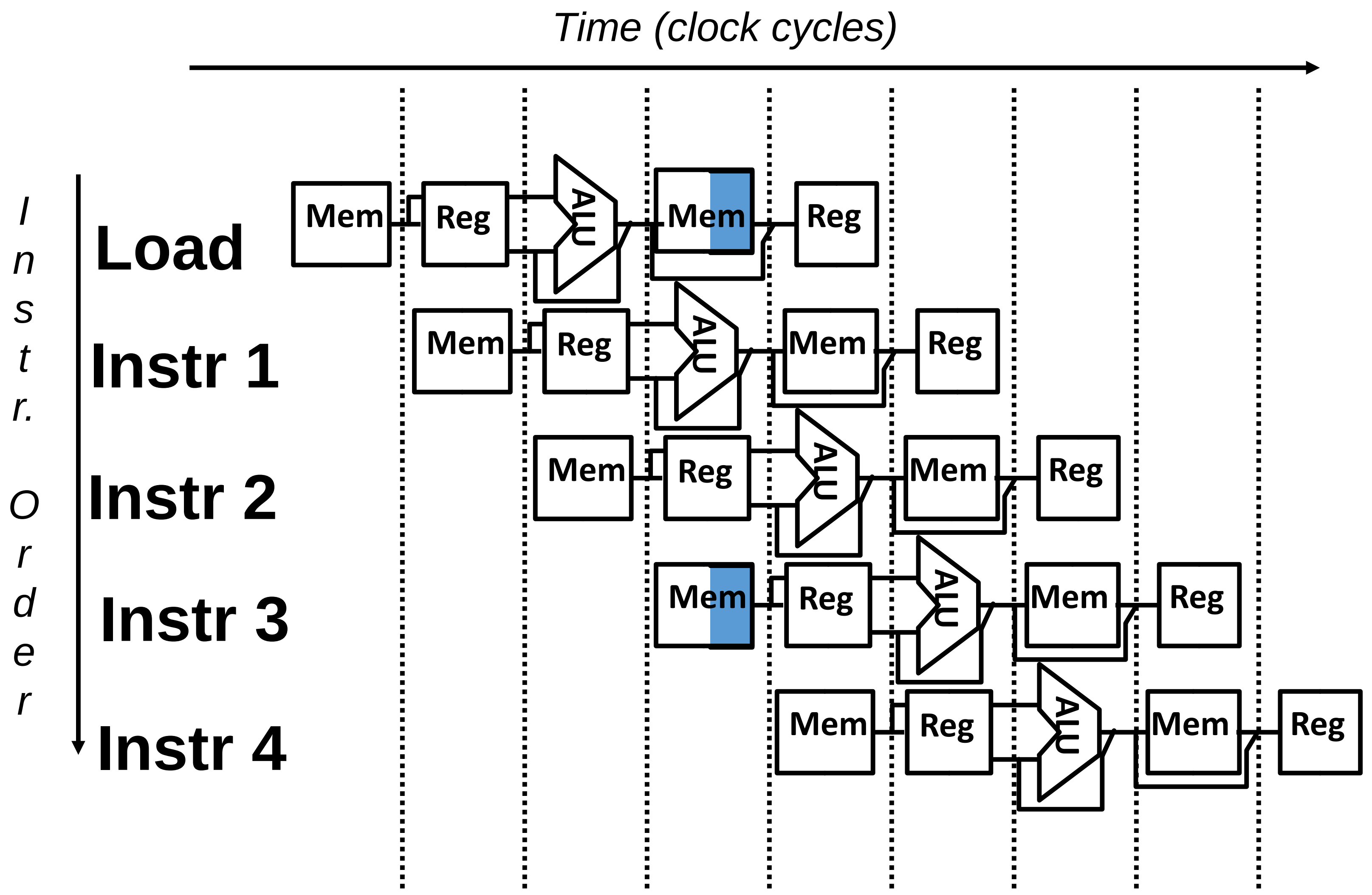
- Some instructions can't be executed in two consecutive cycles -- HAZARDS.
- Hazard types:
  - **Structural**: attempt to use the same resource two different ways at the same time
  - **Data**: attempt to use an item before it is ready

  (Results of one op not yet available to next)
  - **Control**: attempt to make a decision before condition is evaluated (Branch condition not yet available)
- Hazards generally require that the pipeline be stalled for one or more clock cycles

Structural Hazards

- If any combination of instructions in the pipeline cause a resource conflict, then we have a structural hazard
- Most common cause is memory or I/O
- Different examples: A one-port register file, common data and instruction memory, …
- The result is a stall (to wait for the resource to become free)
- Result is a 'bubble'

Single Memory is a Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)

Data Hazards

- **Consider:**

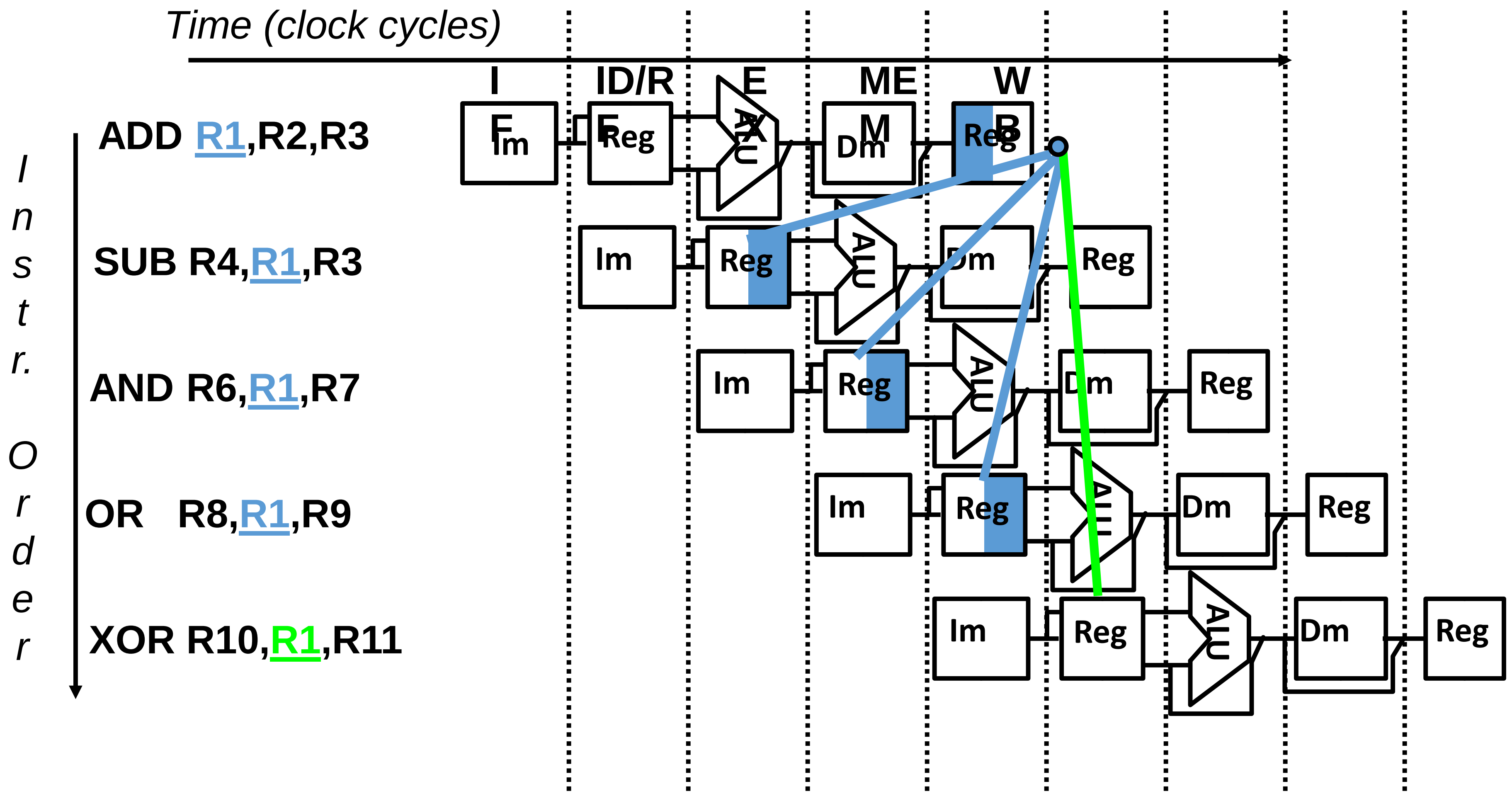
```
ADD   R1, R2, R3
SUB   R4, R1, R5
AND   R6, R1, R7
OR    R8, R1, R9
XOR   R10,R1,R11
```

- **Note the incorrect behavior of SUB, AND & OR**

- **Possible fixes: forwarding, re-ordering instructions or stalling**

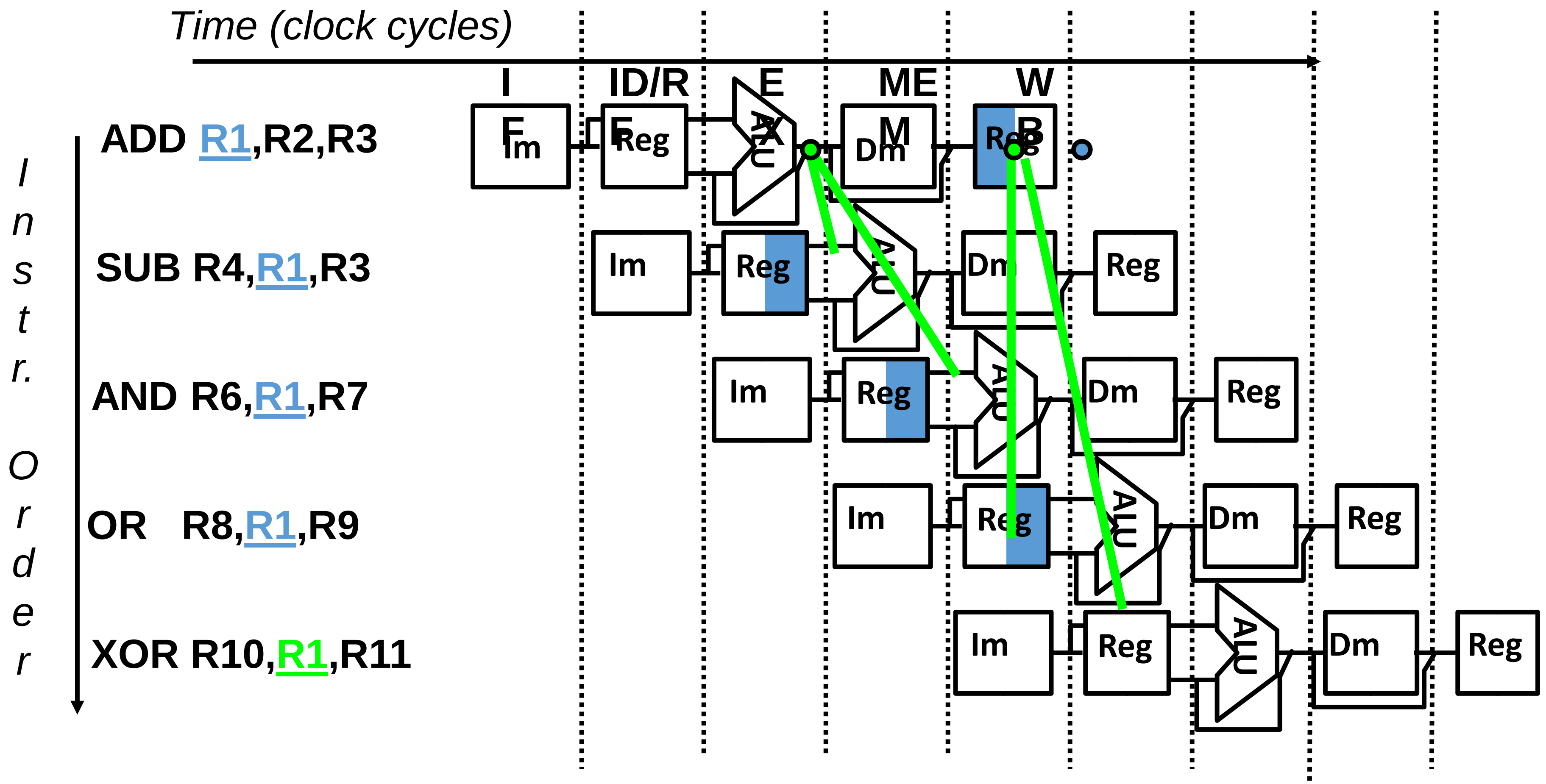
Data Hazard on R1

- **Dependencies backwards in time are hazards**



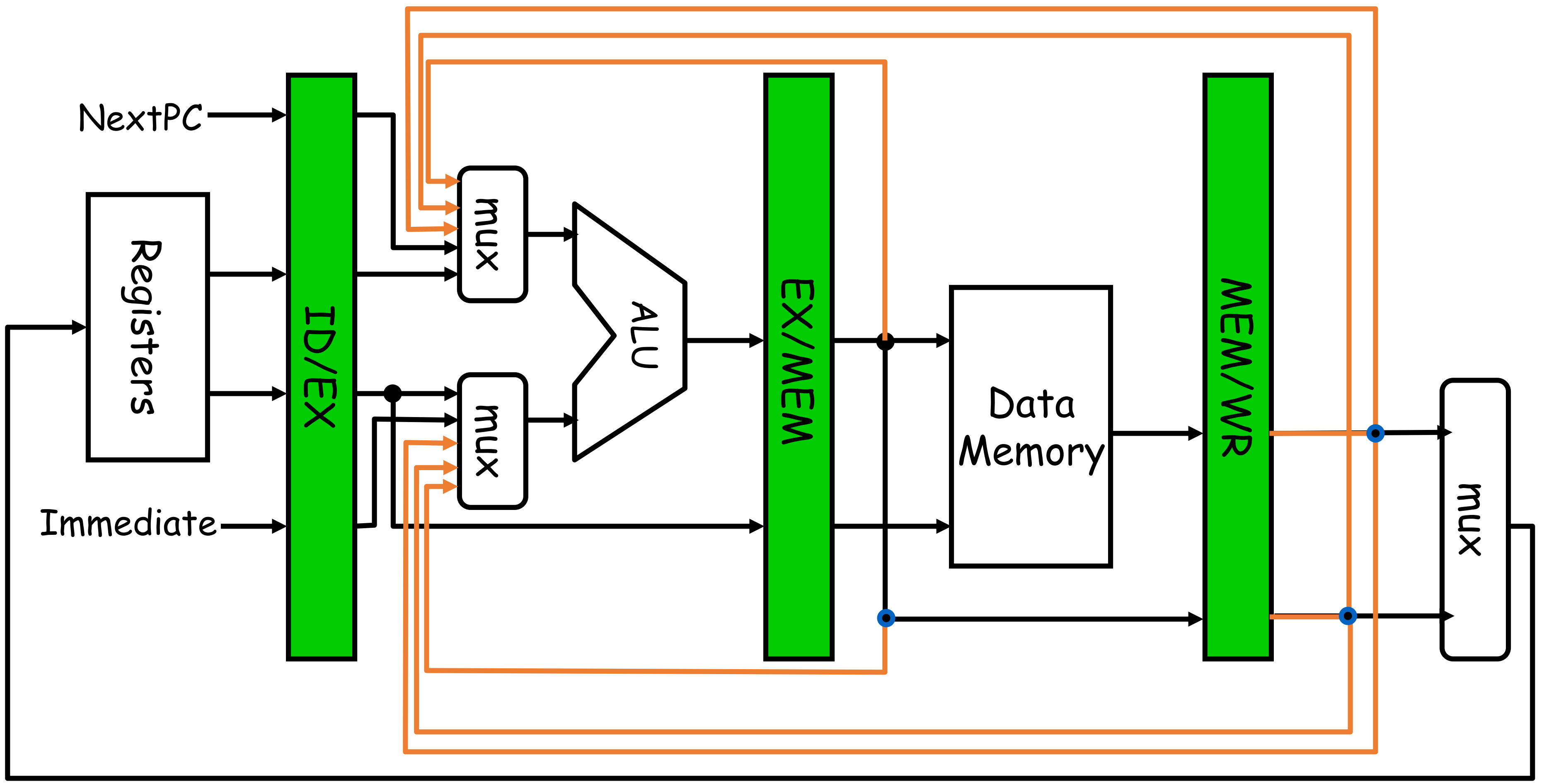
Data Hazard Solution

- "Forward" result from one stage to another



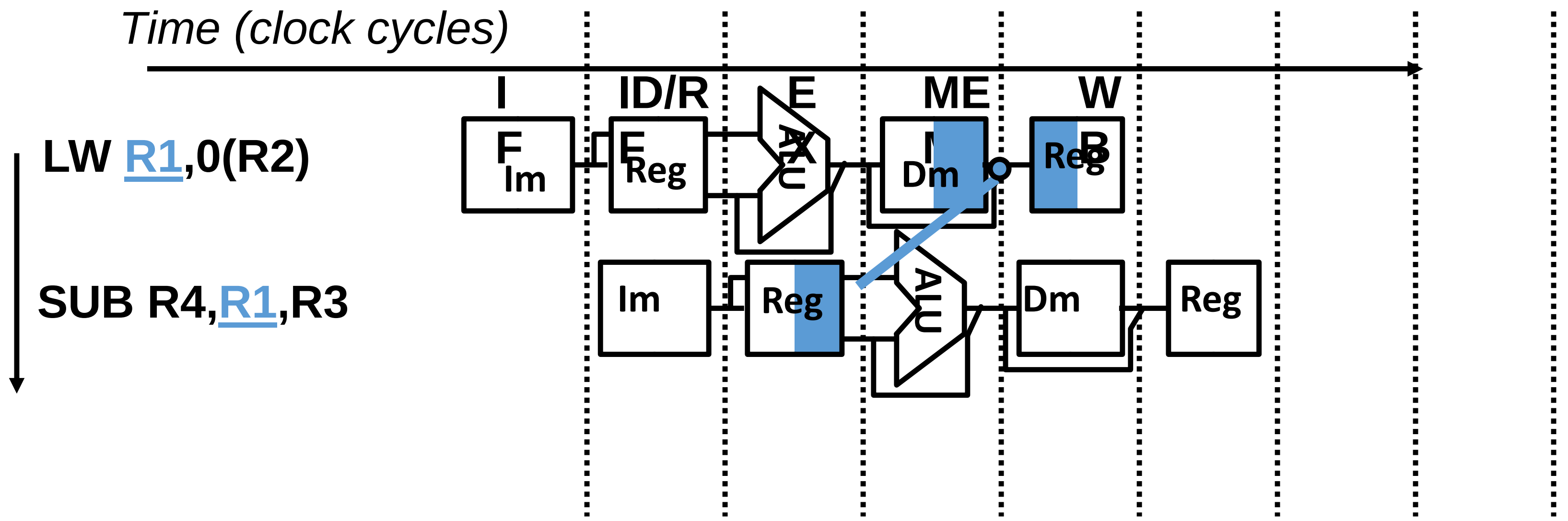
- "or" OK if define read/write properly

HW Change for Forwarding



Forwarding Loads

- **Dependencies backwards in time are hazards**



- **Can’t solve with forwarding:**
- **Must delay/stall instruction dependent on loads**

Forwarding

One way to fix a data hazard

- Basic idea is to bypass the register file, etc
- Detect the hazard, then
- Get ALU result and send it directly to the next input of the ALU (for SUB)

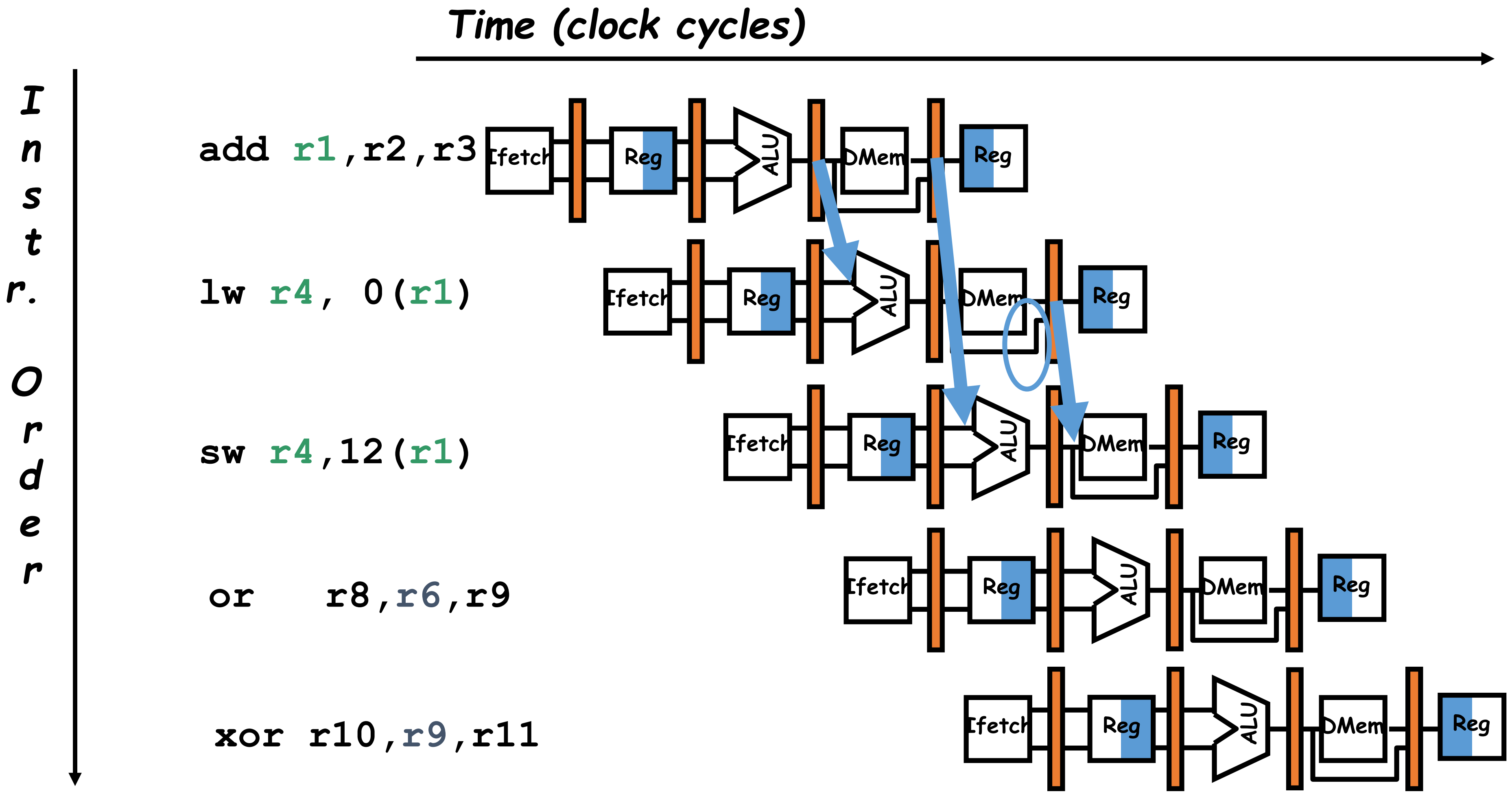
Data Hazards Requiring a Stall

- Consider:

```
LW      R1, 0(R2)
SUB     R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

- Forwarding can’t help the LW/SUB hazard
- The pipe must be stalled to wait for LW data

Forwarding to Avoid LW-SW Data Hazard

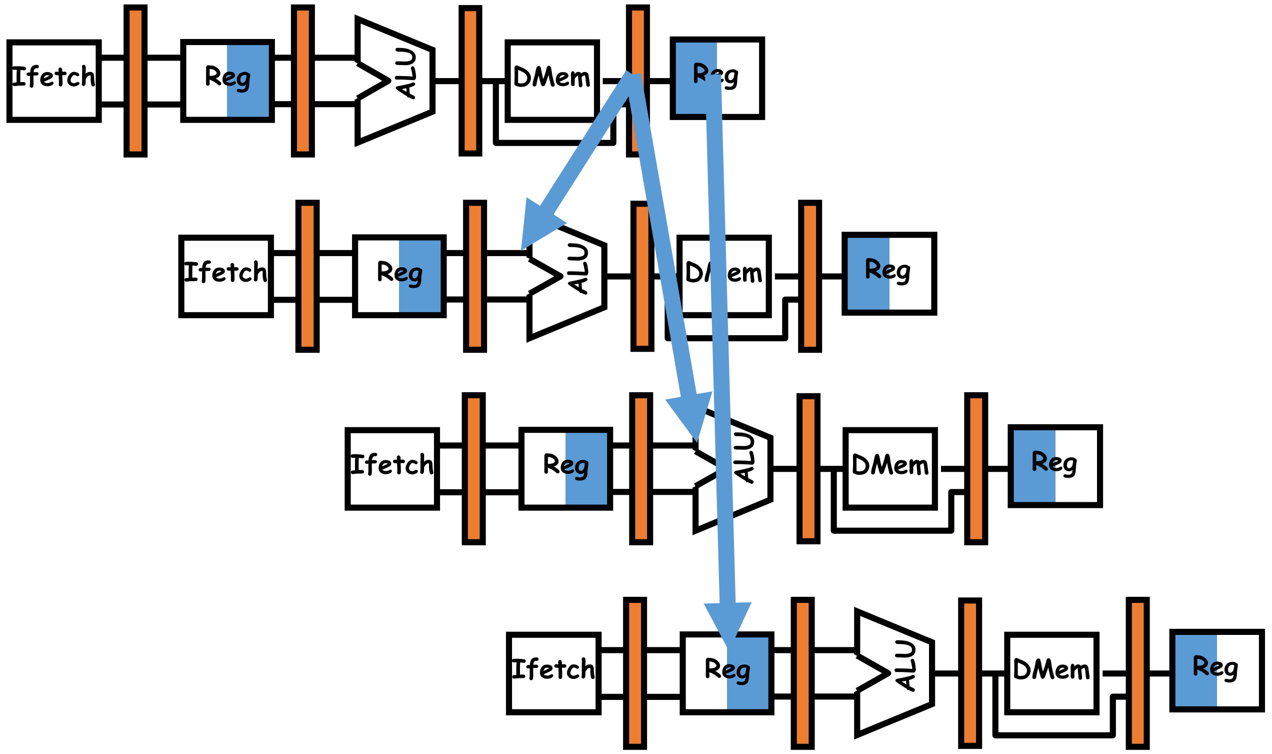


Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

```
lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
```

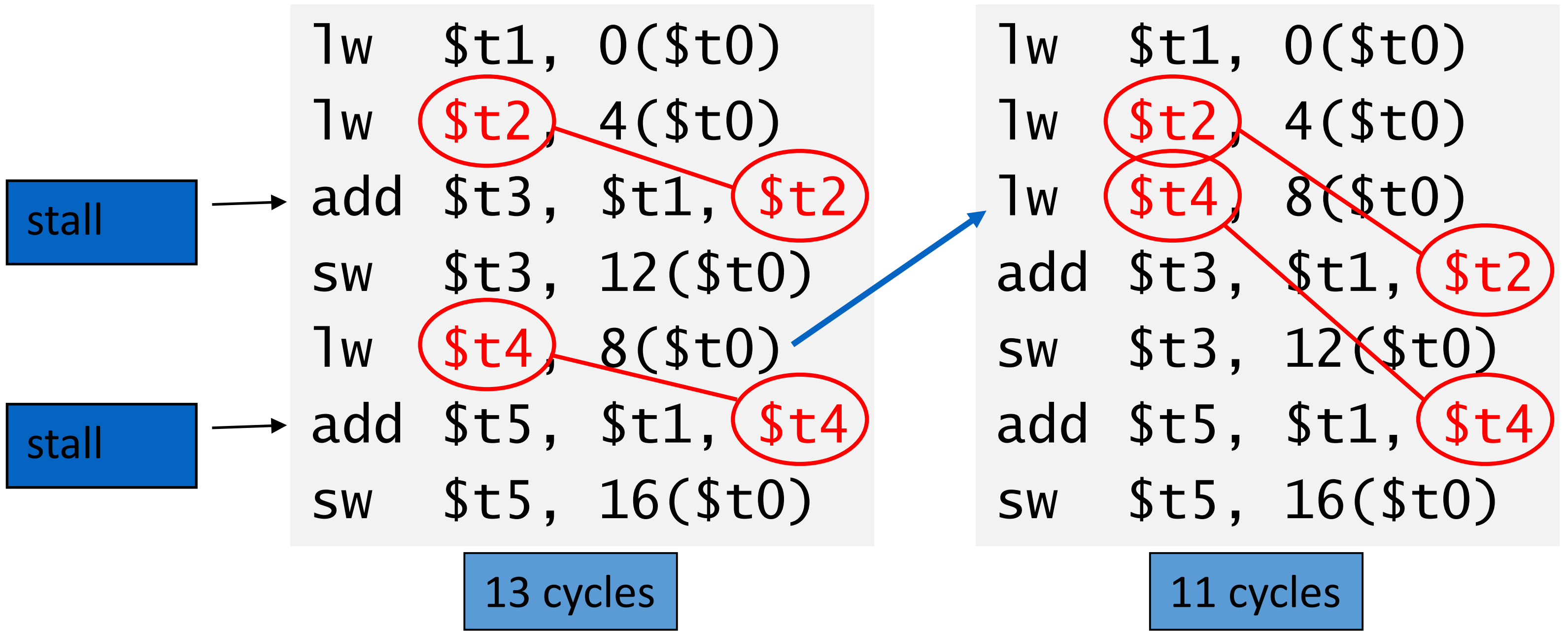


Scheduling for Data Hazards

- Stalls can sometimes be avoided by rescheduling the execution order of instructions
- Note that additional registers may be needed to reschedule the code

Code Scheduling to Avoid Stalls

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



```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

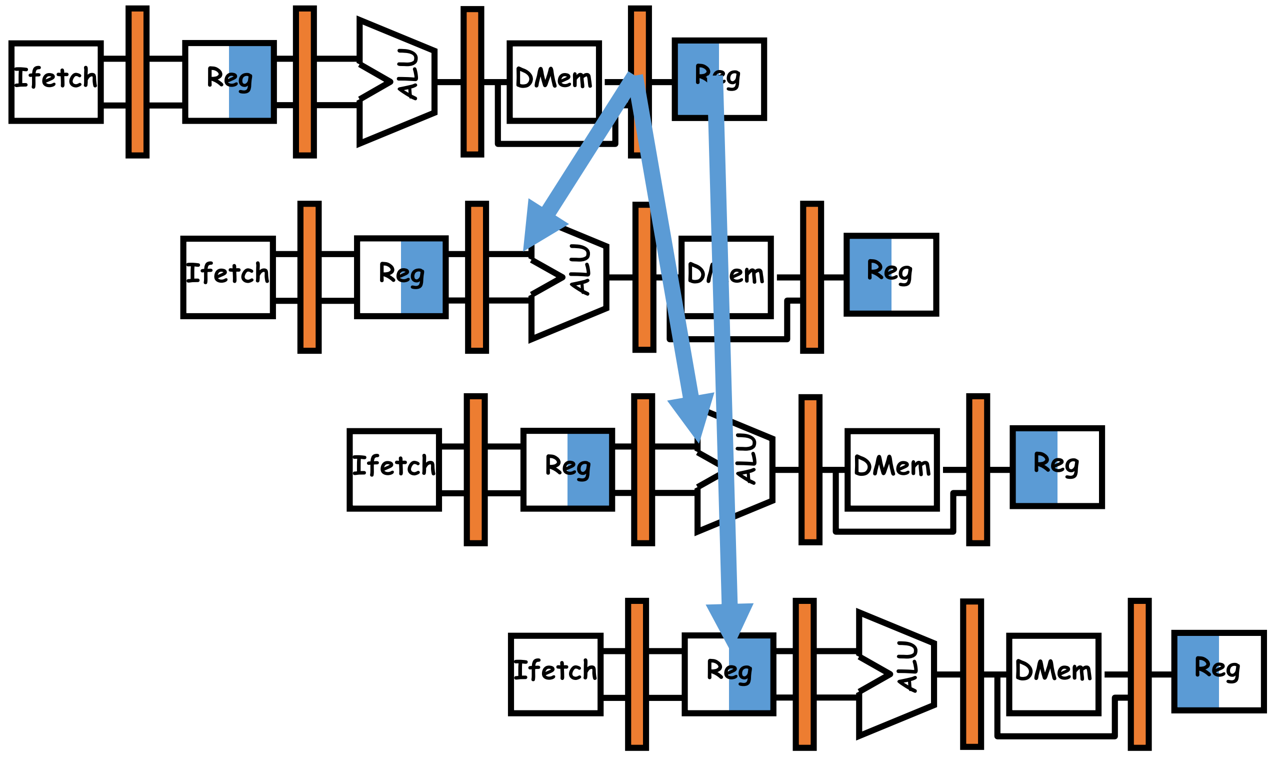
```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

```
lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
```

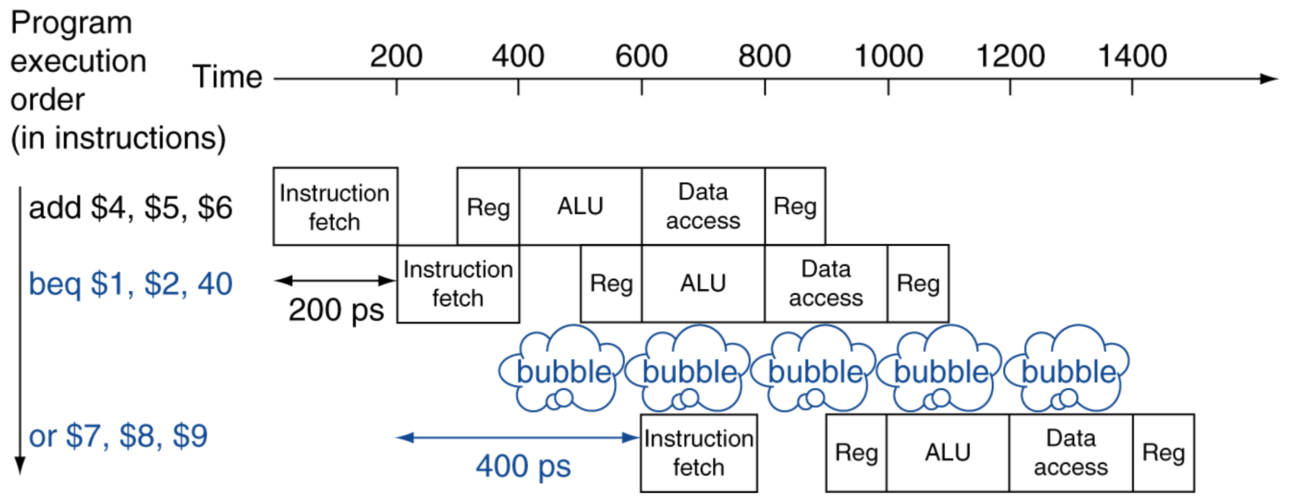


Control Hazards

- 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
- In MIPS pipeline
    - Need to compare registers and compute target early in the pipeline
    - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Control Hazards

- Control hazards cost more than data hazards
- If branch taken, PC normally changed at least after the second stage --- worse for deep pipe
- Simple solution is to stall on branches --- performance
- Possible fixes:
  - Determine branch condition earlier
  - Compute taken PC earlier
  - Both!

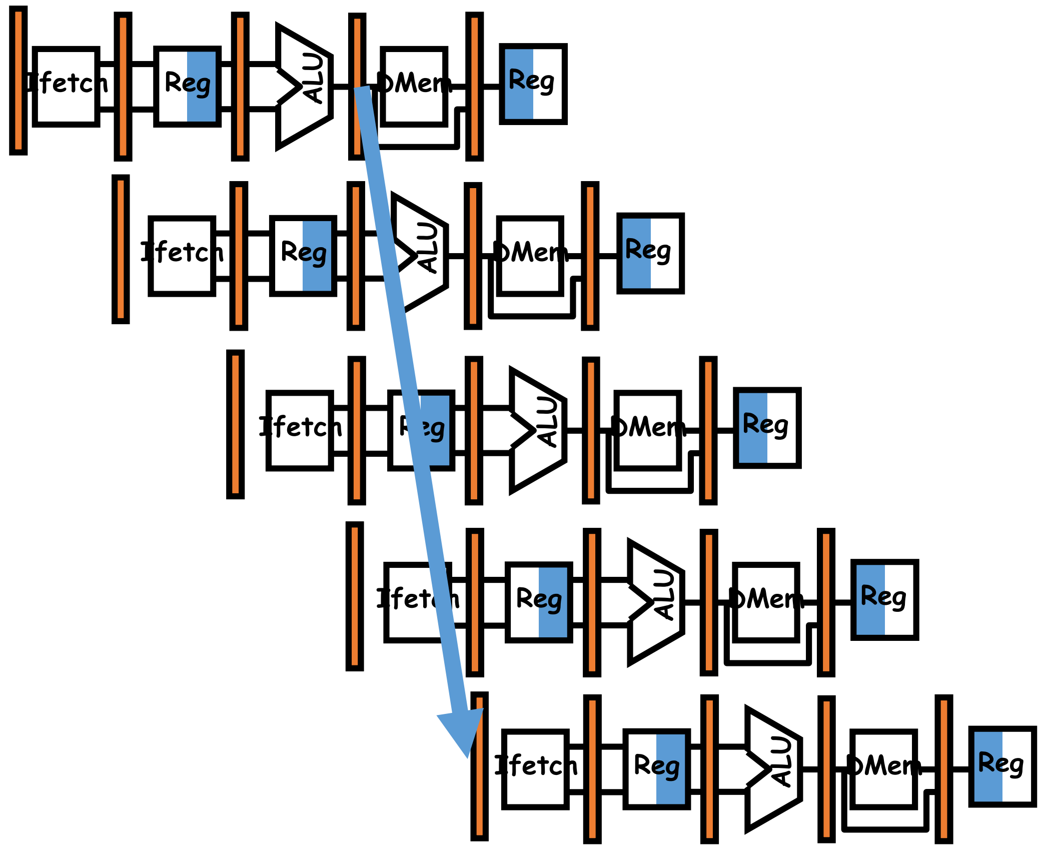
Control Hazards: Avoiding

Main techniques for avoiding stalls:

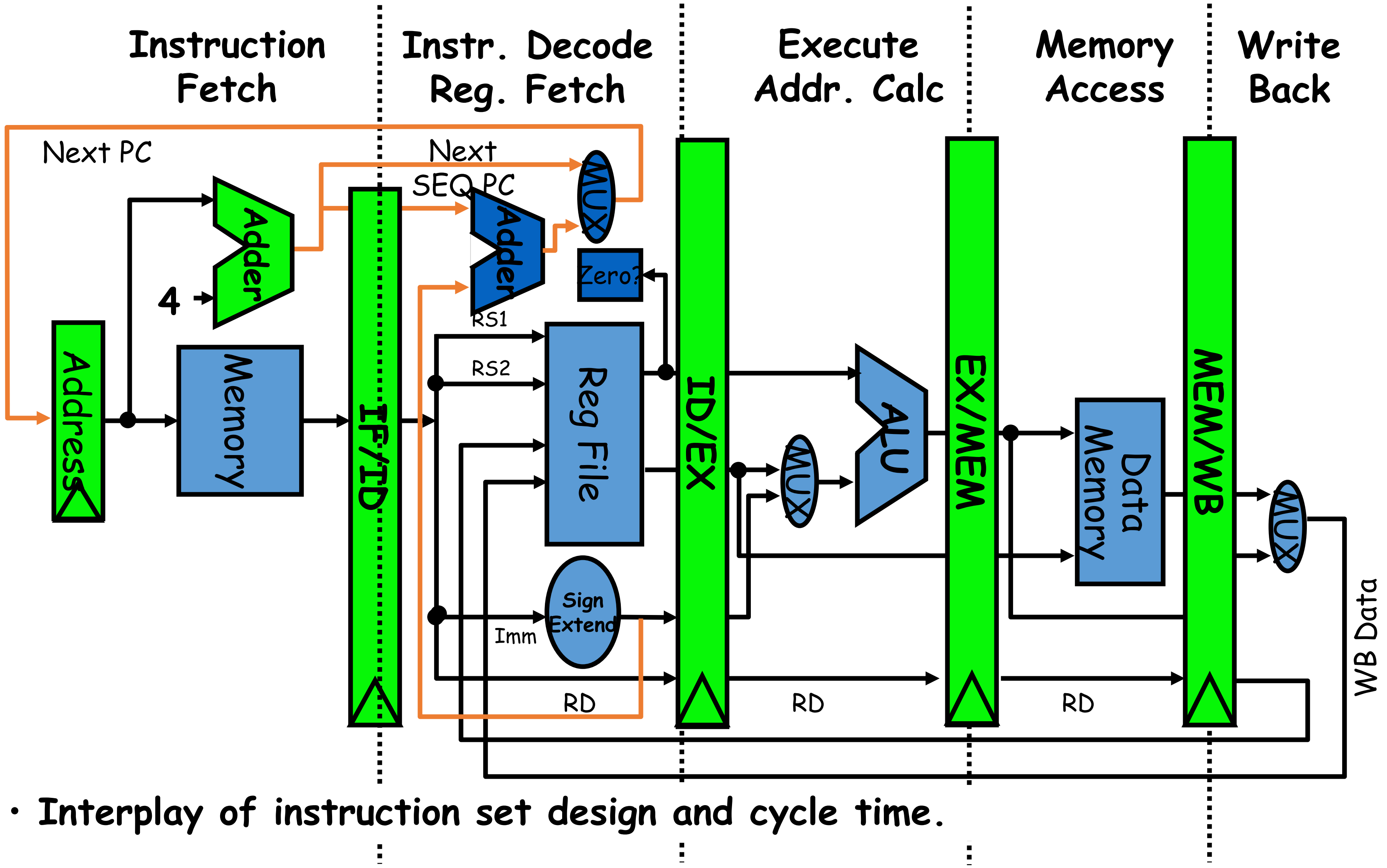
- Eliminating branches
- Branch prediction
- Move comparison testing to earlier stage
- Branch delay slots

Control Hazard on Branches Three Stage Stall

```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



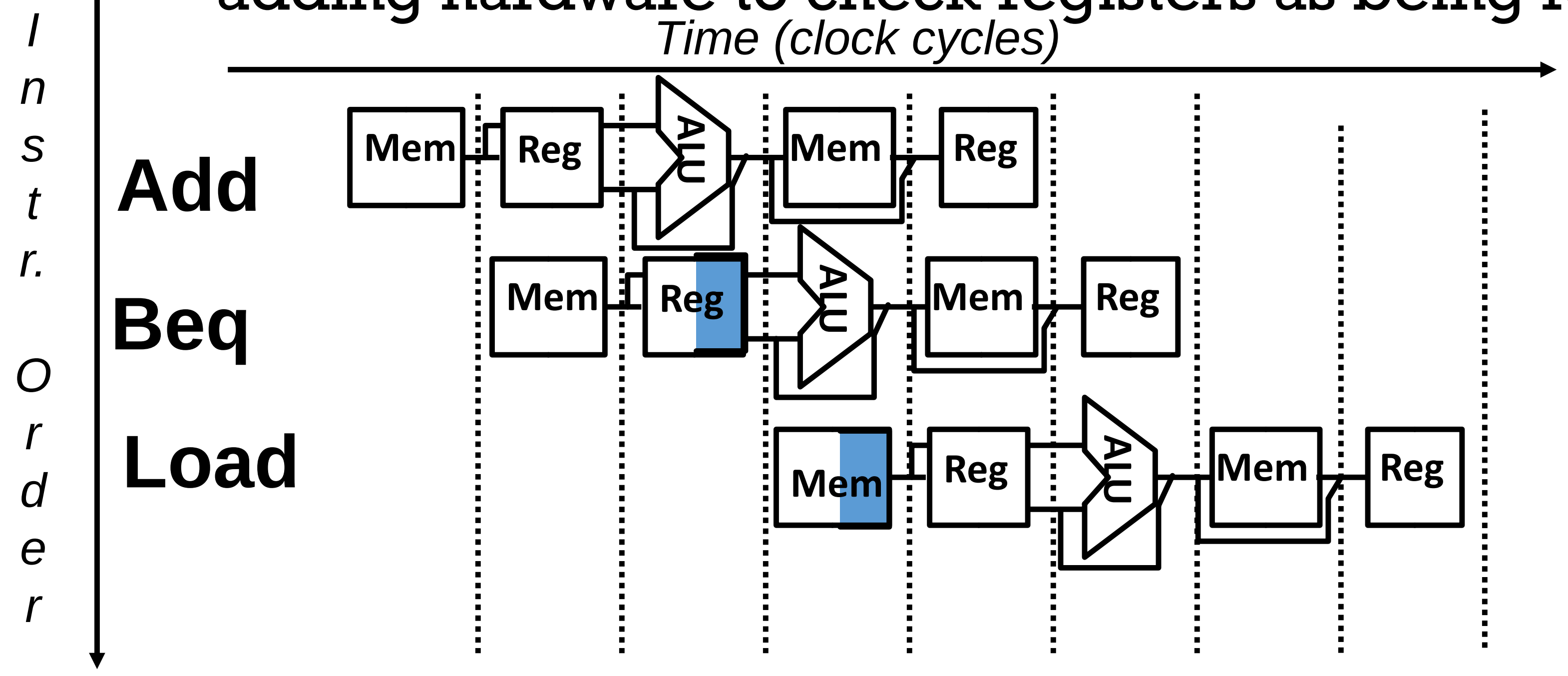
Pipelined MIPS Datapath



- Interplay of instruction set design and cycle time.

Control Hazard Solutions

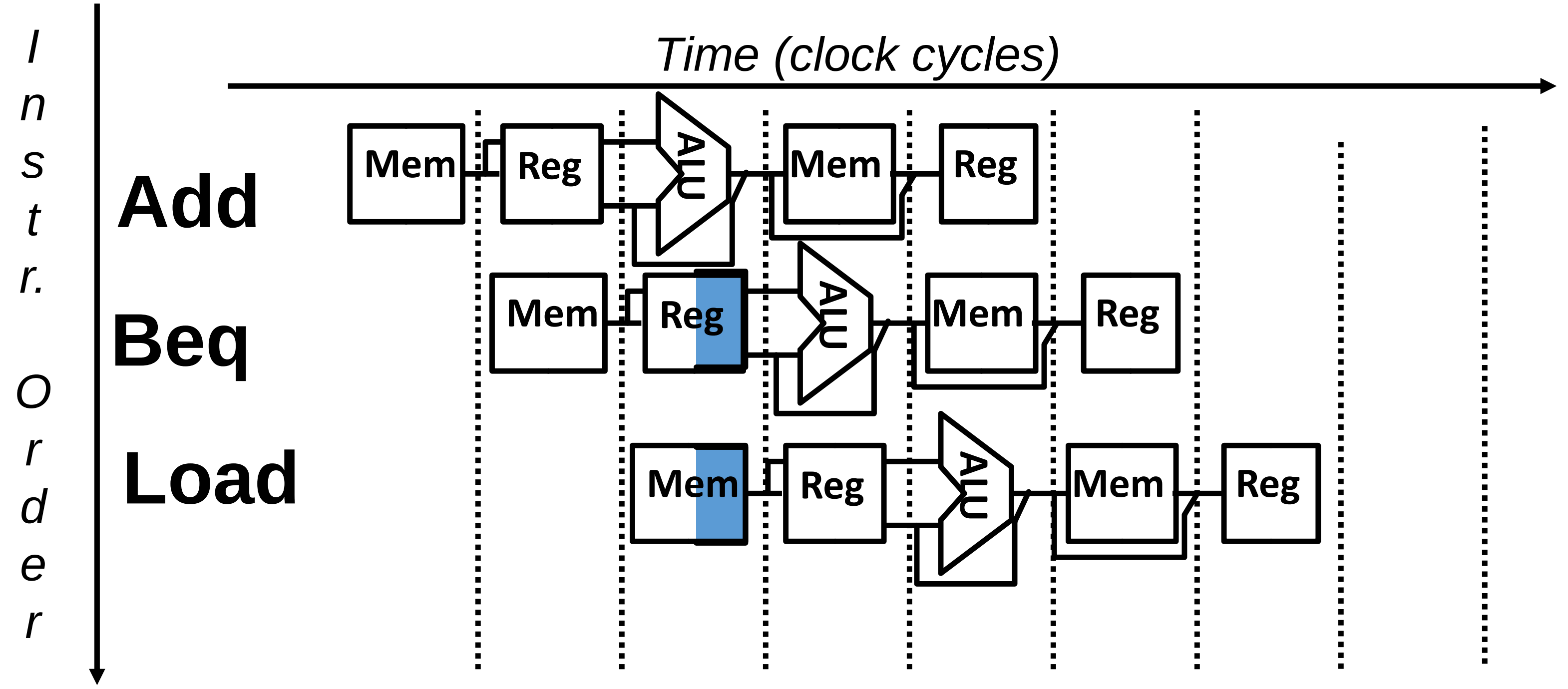
- Stall: wait until decision is clear
  - It is possible to move up decision to 2nd stage by adding hardware to check registers as being read



- Impact: 2 clock cycles per branch instruction => slow

Control Hazard Solutions

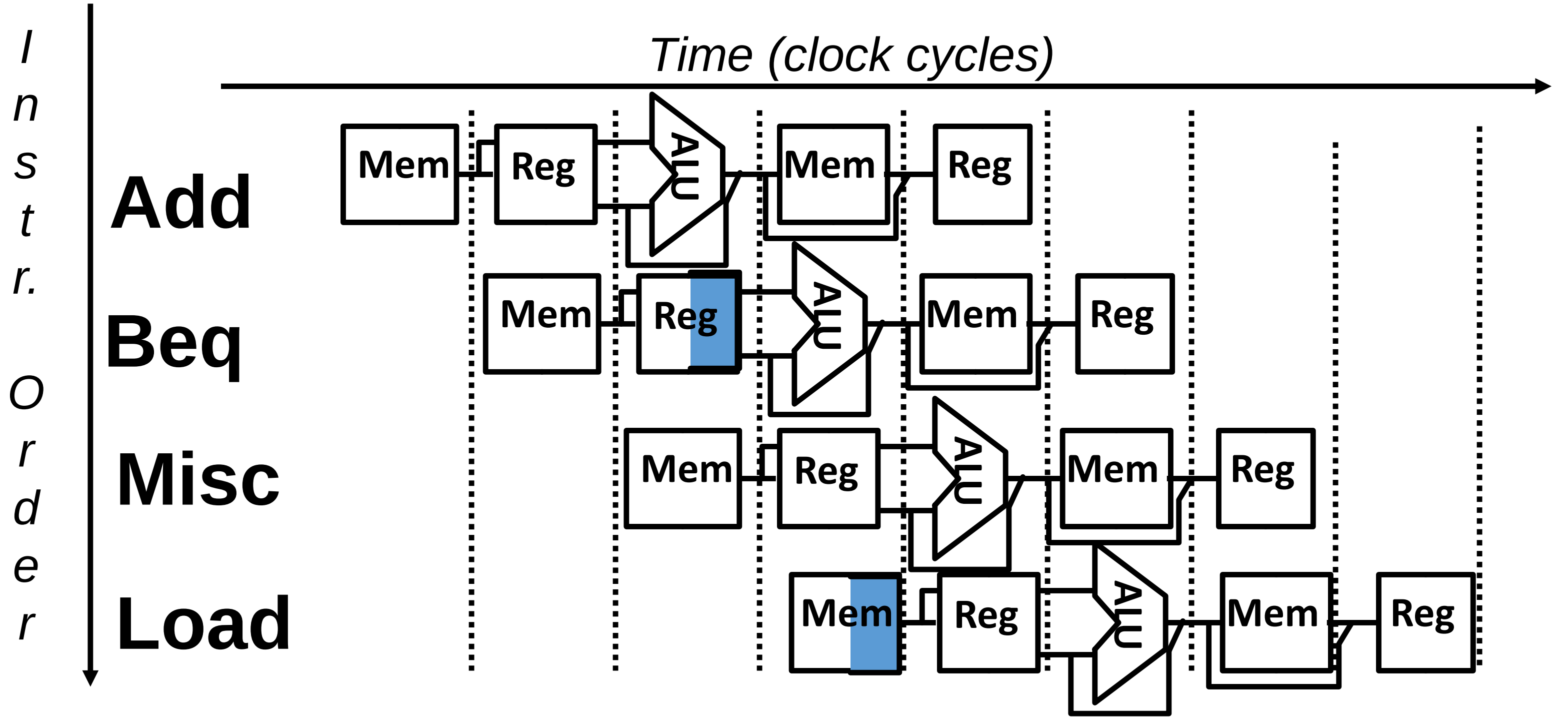
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time)
- More dynamic scheme: history of 1 branch (90%)

Control Hazard Solutions

- Redefine branch behavior (takes place after next instruction) "delayed branch"

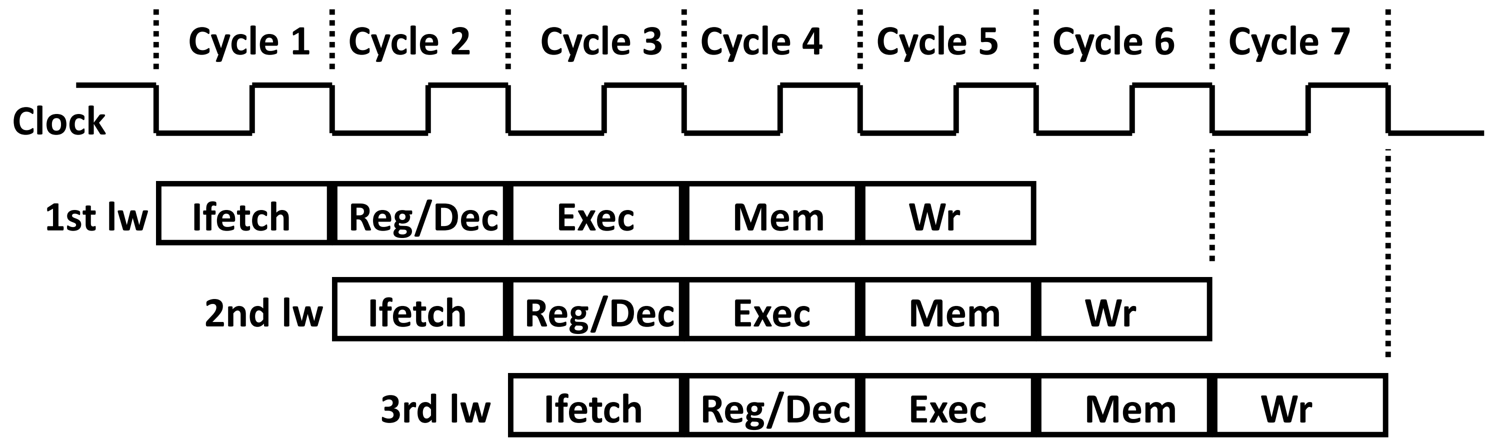


- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (50% of time- compiler)
- The longer the pipeline, less useful (need more inst. to fill slots

Designing a Pipelined Processor

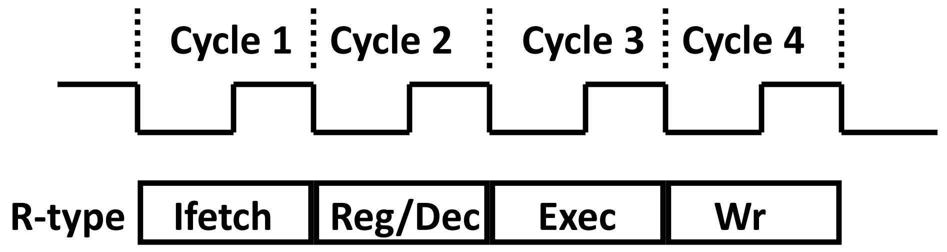
- Go back and examine your datapath and control diagram
- Associate resources with states
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

Pipelining the Load Instruction



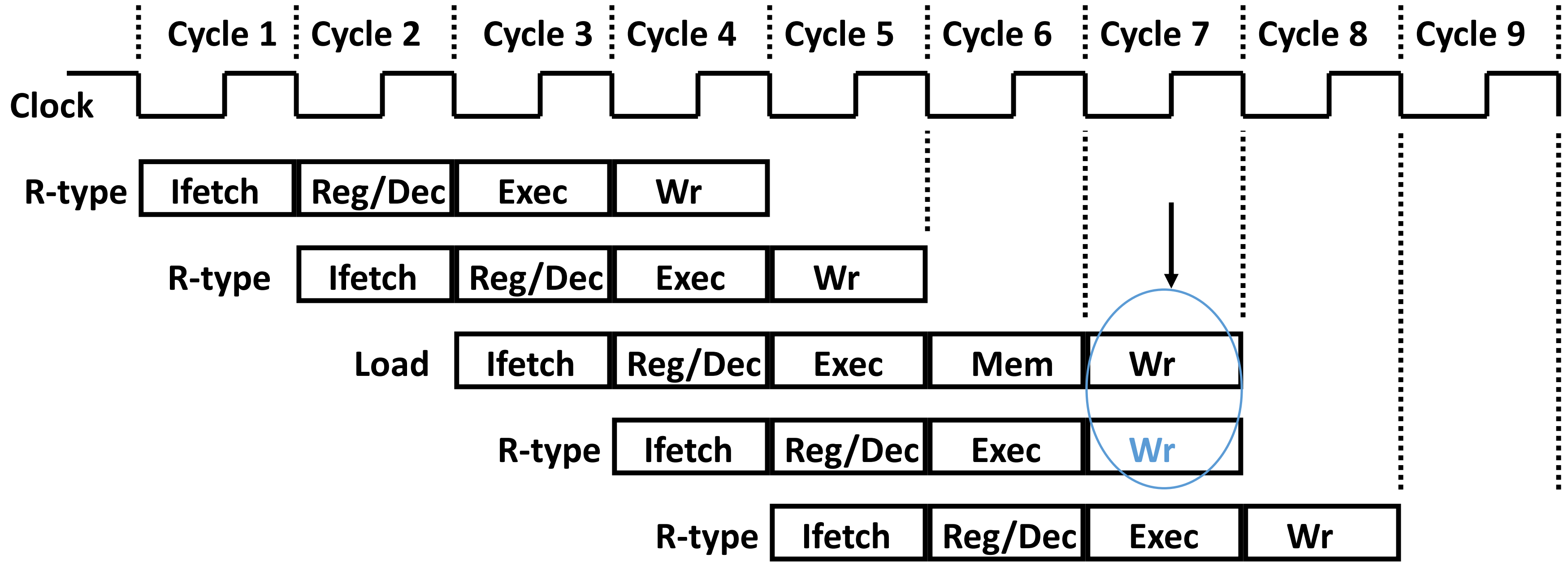
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

The Four Stages of R-type



- **Ifetch: Instruction Fetch**
  - **Fetch the instruction from the Instruction Memory**
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
  - **ALU operates on the two register operands**
  - **Update PC**
- **Wr: Write the ALU output back to the register file**

Pipelining the R-type and Load Instruction



- **We have pipeline conflict or structural hazard:**
  - **Two instructions try to write to the register file at the same time!**
  - **Only one write port**

Important Observation

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage

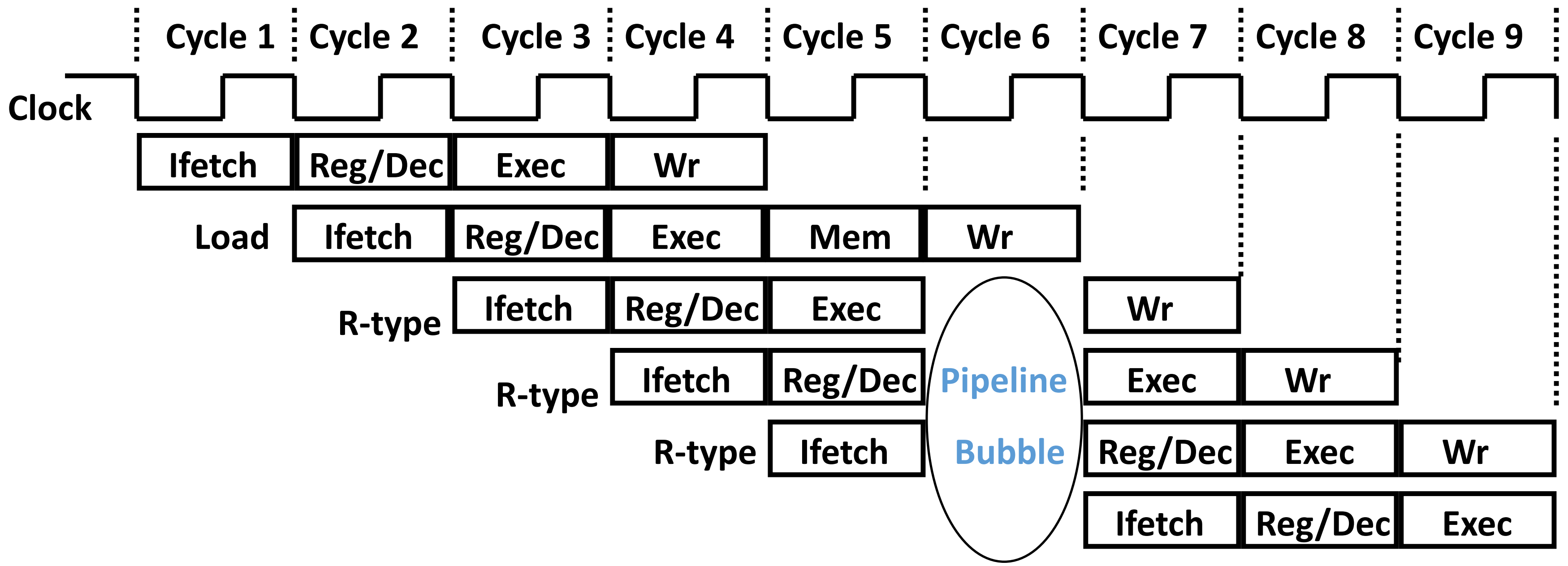
	1	2	3	4	5
Load	Ifetch	Reg/Dec	Exec	Mem	Wr

  - R-type uses Register File's Write Port during its 4th stage

	1	2	3	4
R-type	Ifetch	Reg/Dec	Exec	Wr

° 2 ways to solve this pipeline hazard.

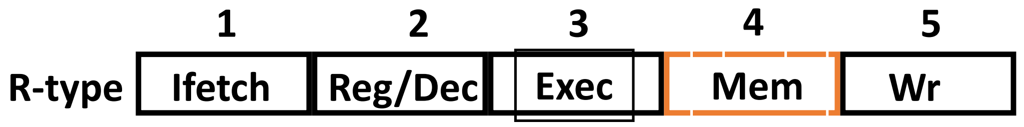
Solution 1: Insert "Bubble" into the Pipeline

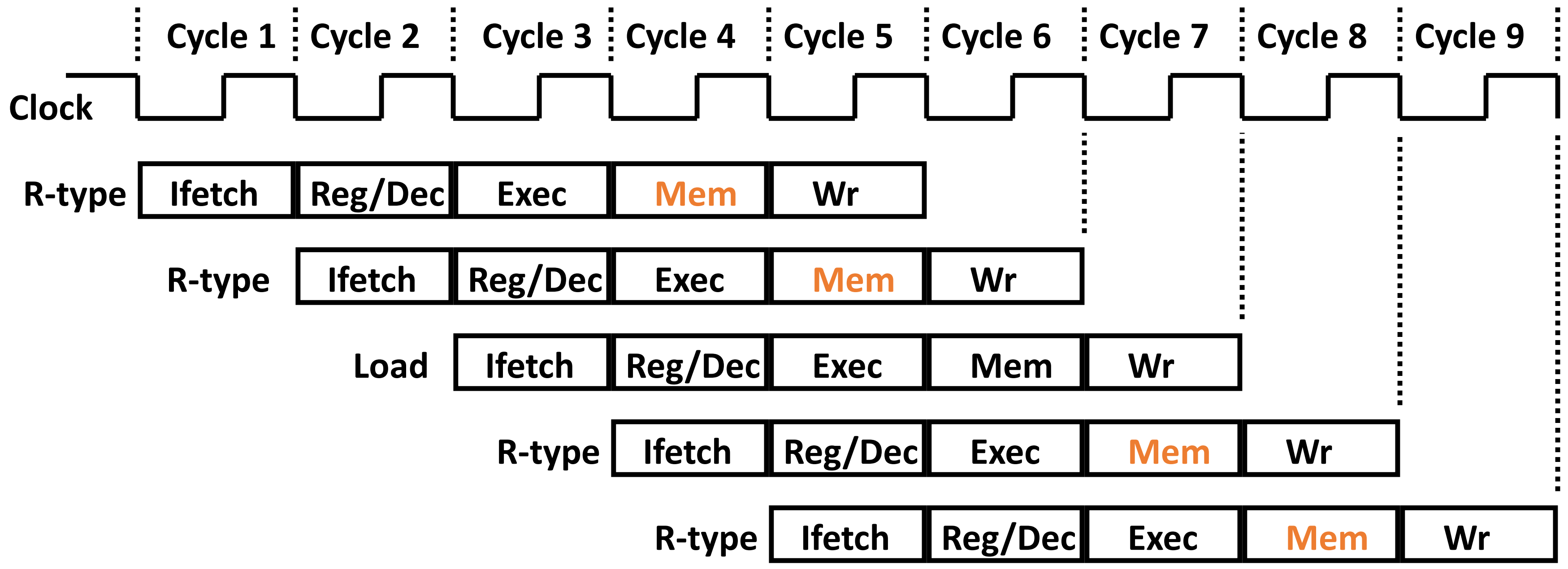


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

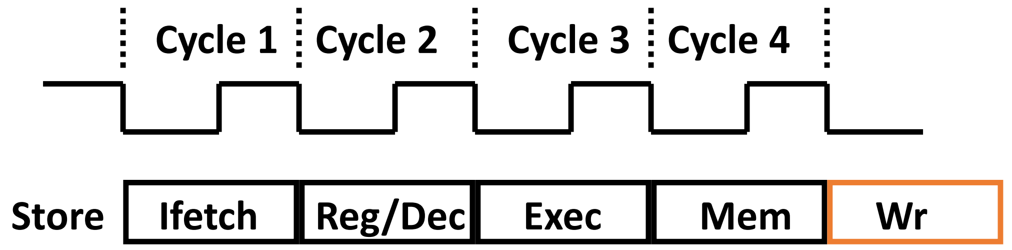
Solution 2: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.





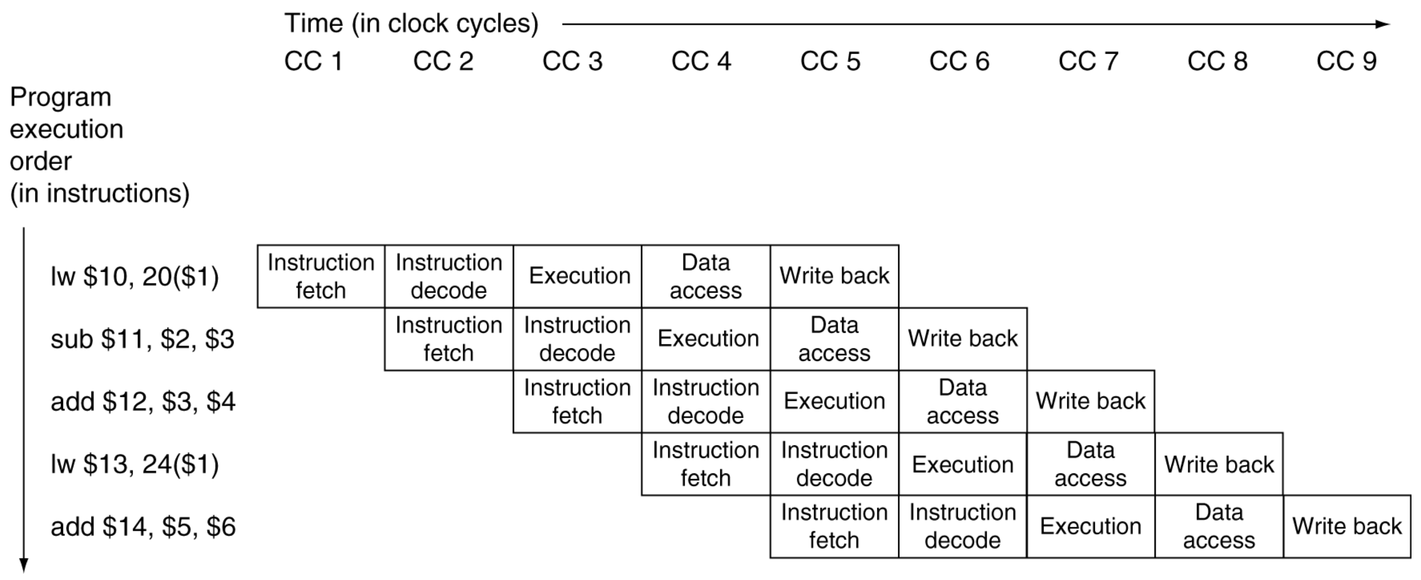
The Four Stages of Store



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

Multi-Cycle Pipeline Diagram

- Traditional form



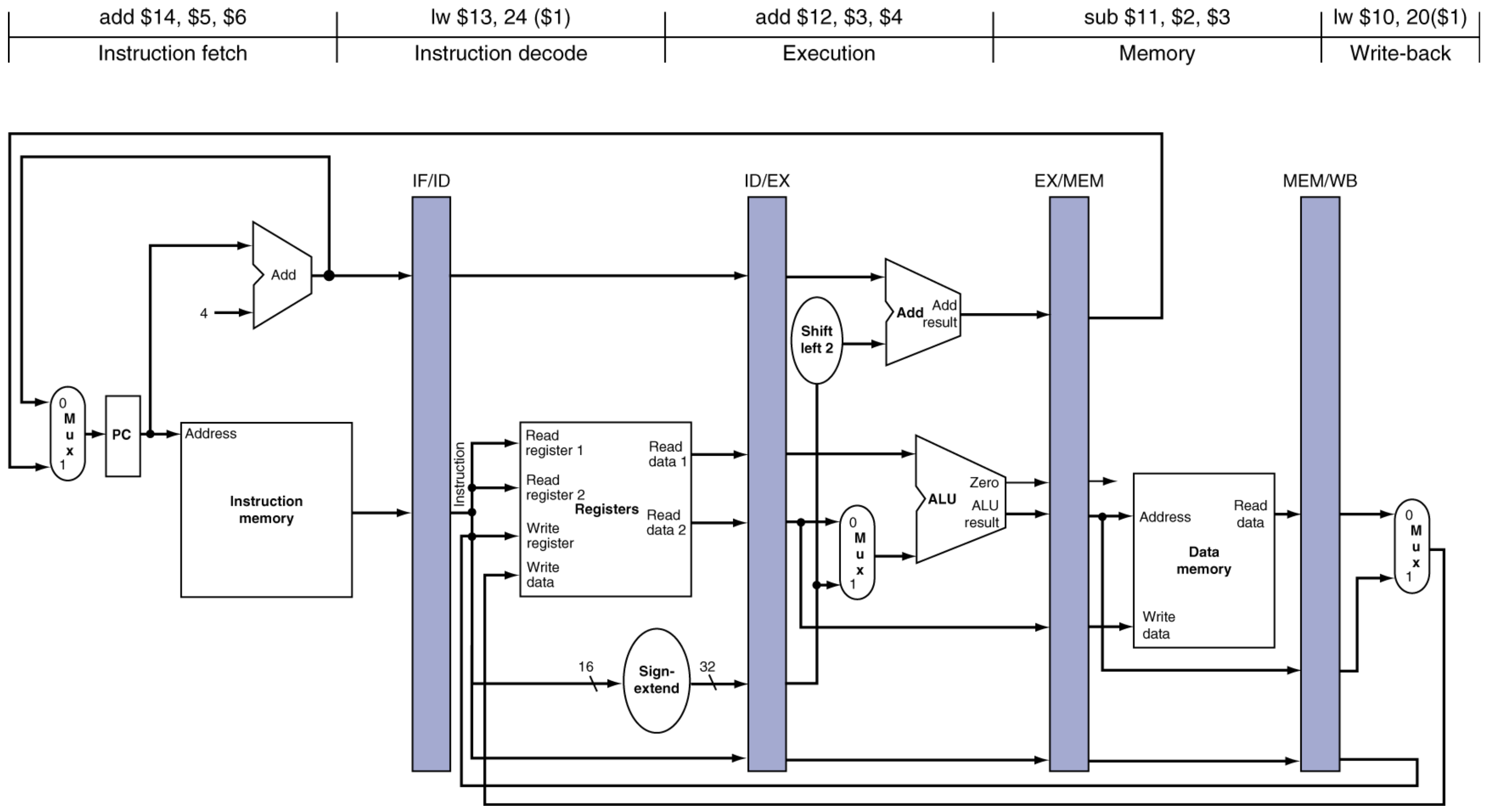
Time (in clock cycles)

Program execution order (in instructions)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle



Summary: Pipelining

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

Summary

- Pipelining is a fundamental concept
  - multiple steps using distinct resources
- Utilize capabilities of the Datapath by pipelined instruction processing
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- https://www.mips.com/
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html
- Professor El-Naga ECE-425 notes