ECE-425: Chapter-4: Processor (Cont.)

Prof: Mohamed El-Hadedy

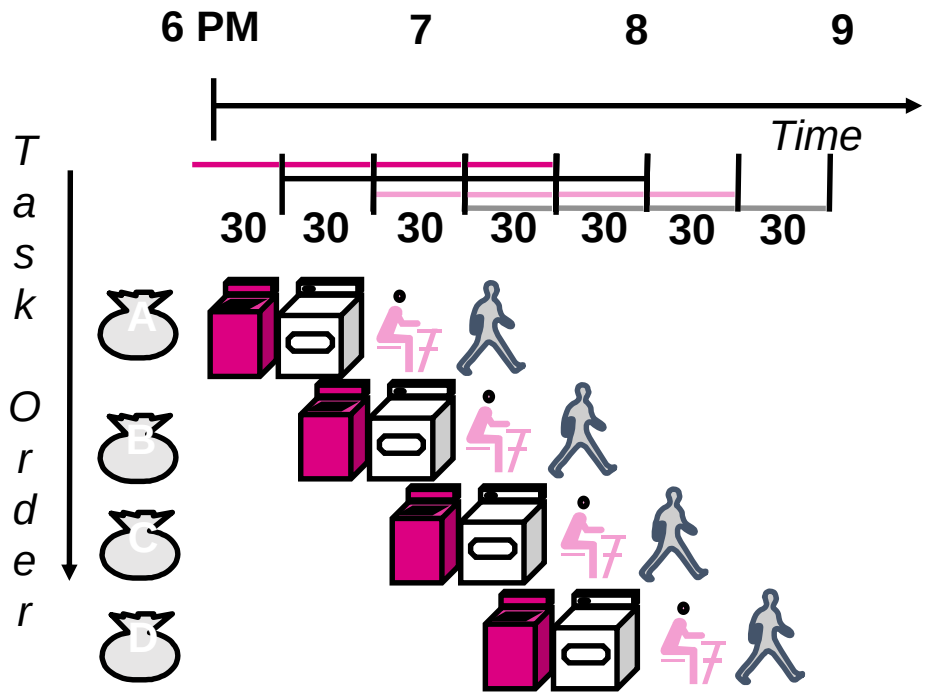
Email: mealy@cpp.edu

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



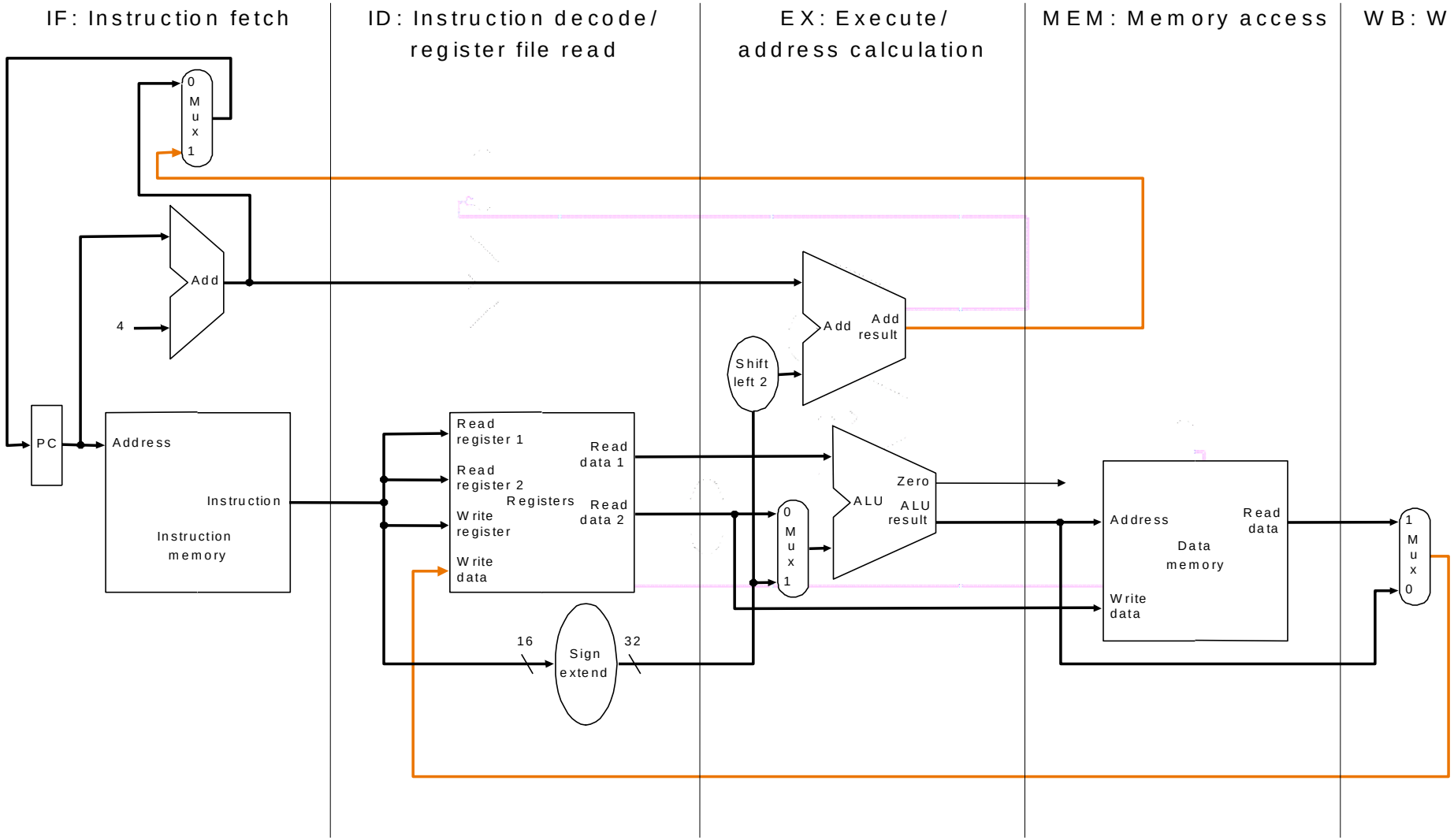
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
-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

MIPS Pipeline

- Five stages, one step per stage

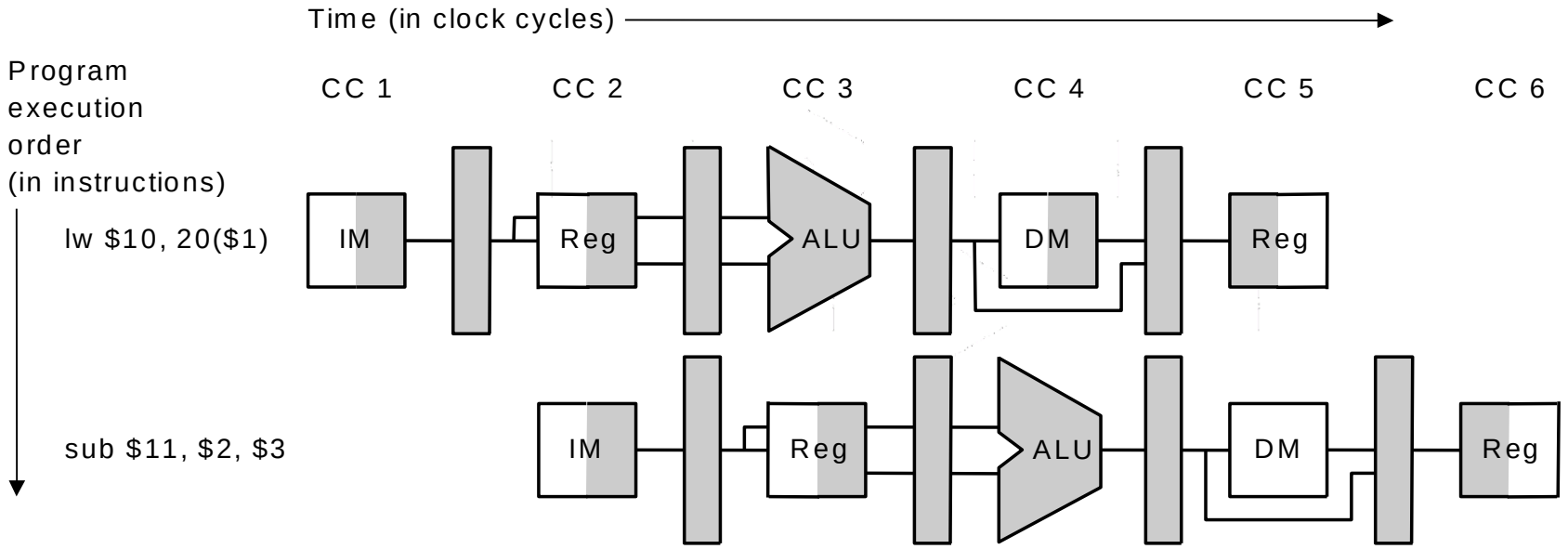
1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

Basic Idea



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

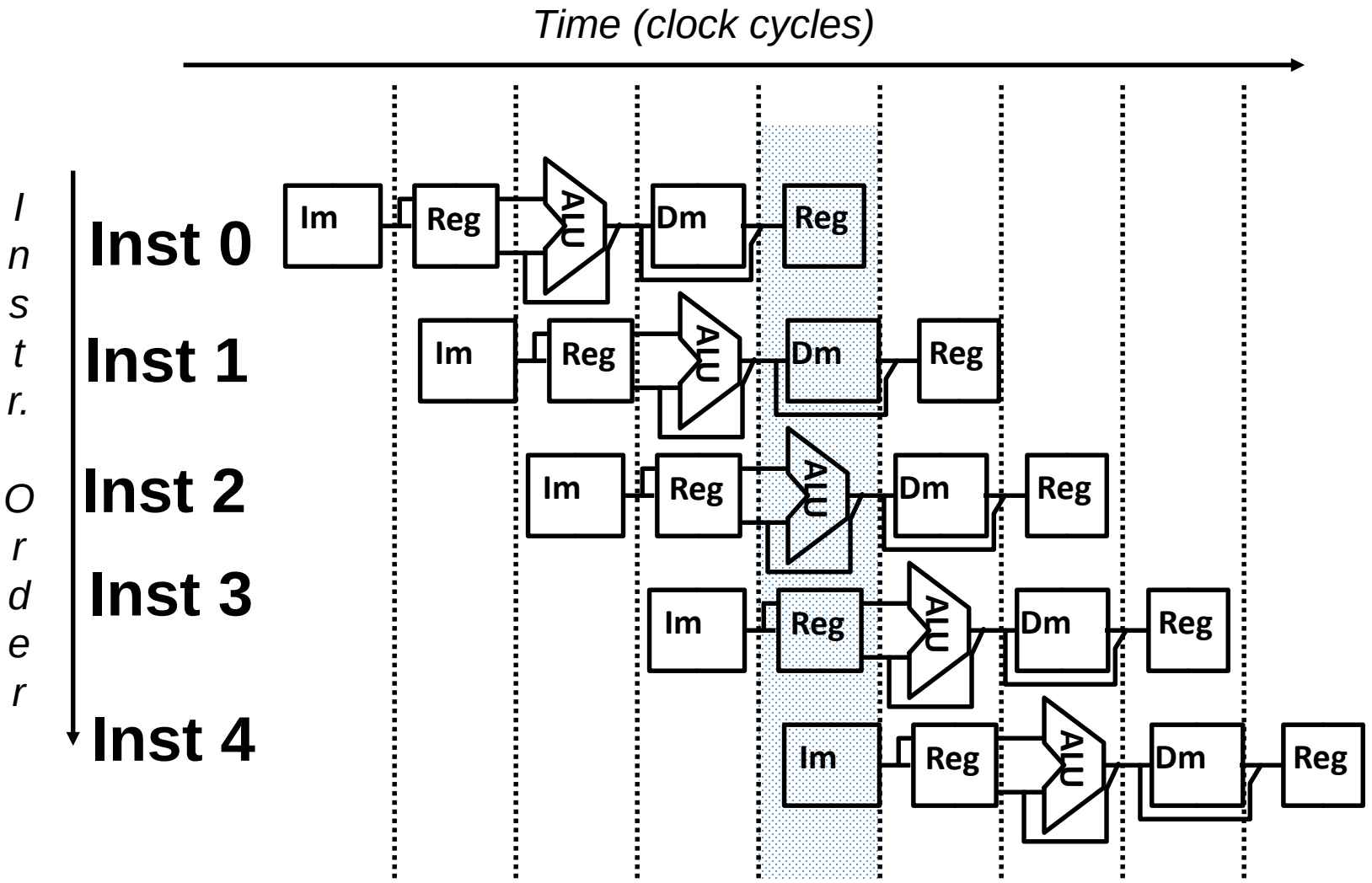
Pipelines



- 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**

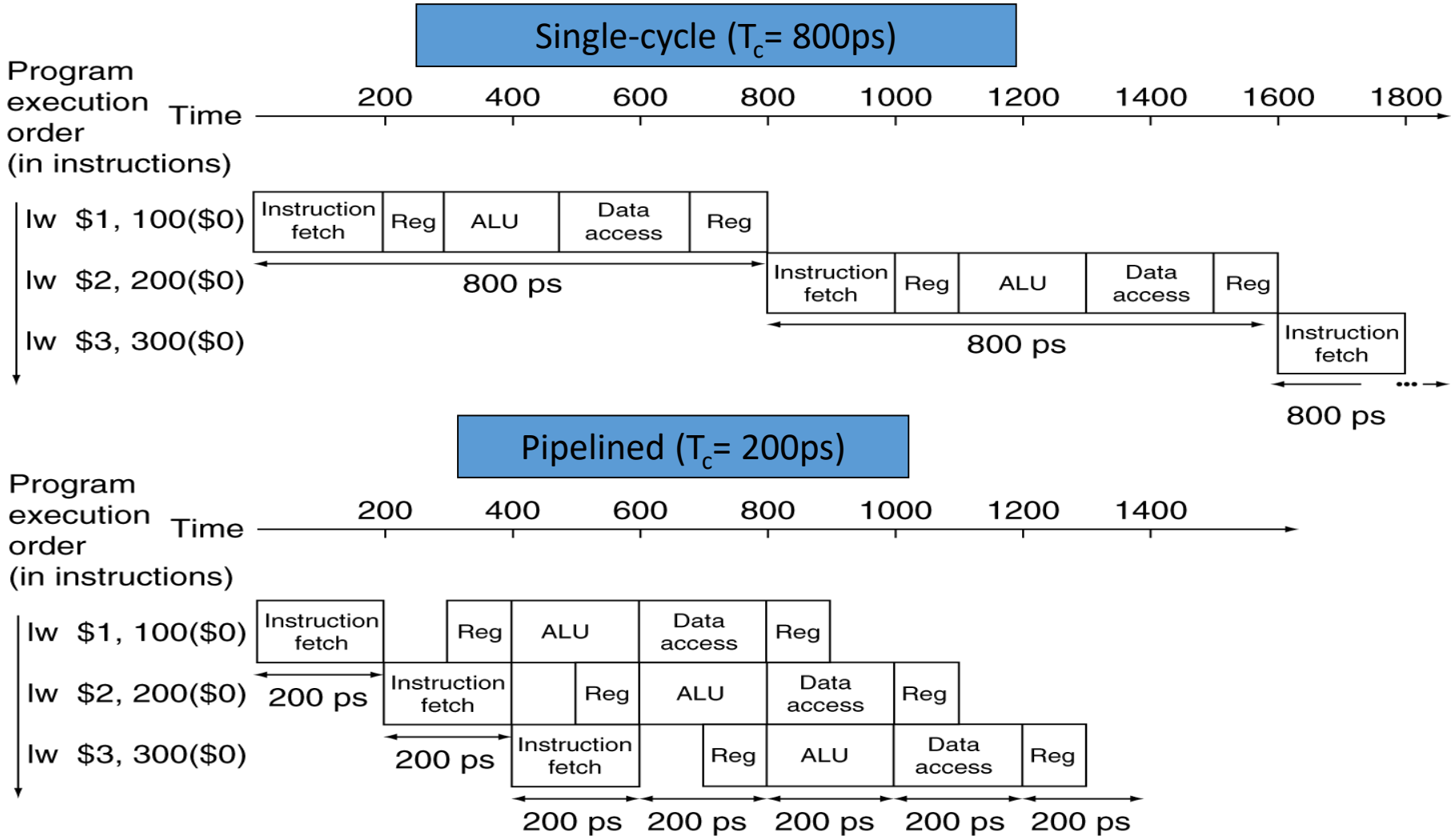


Pipeline Performance

- 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



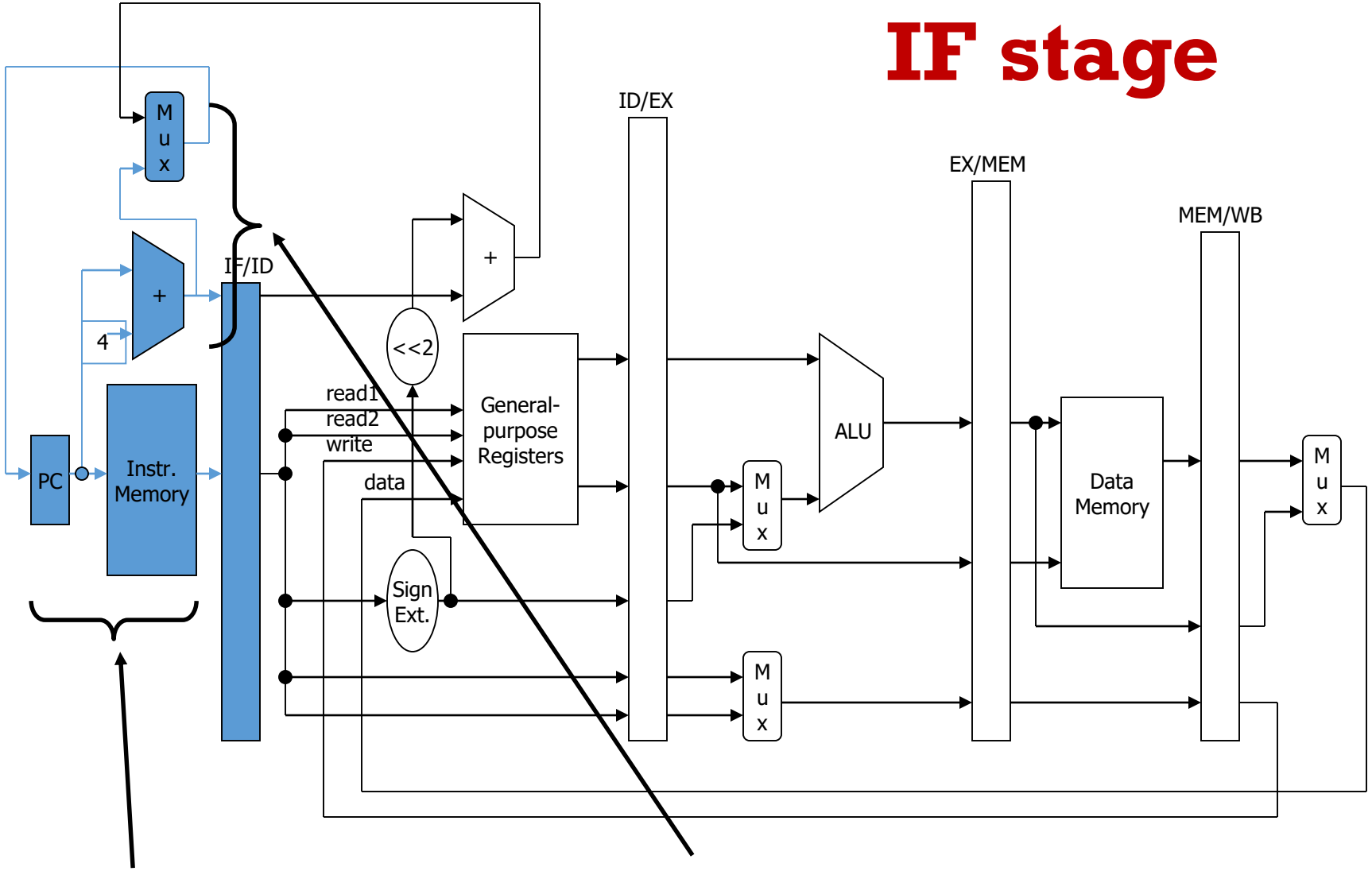
Pipeline Speedup

- 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

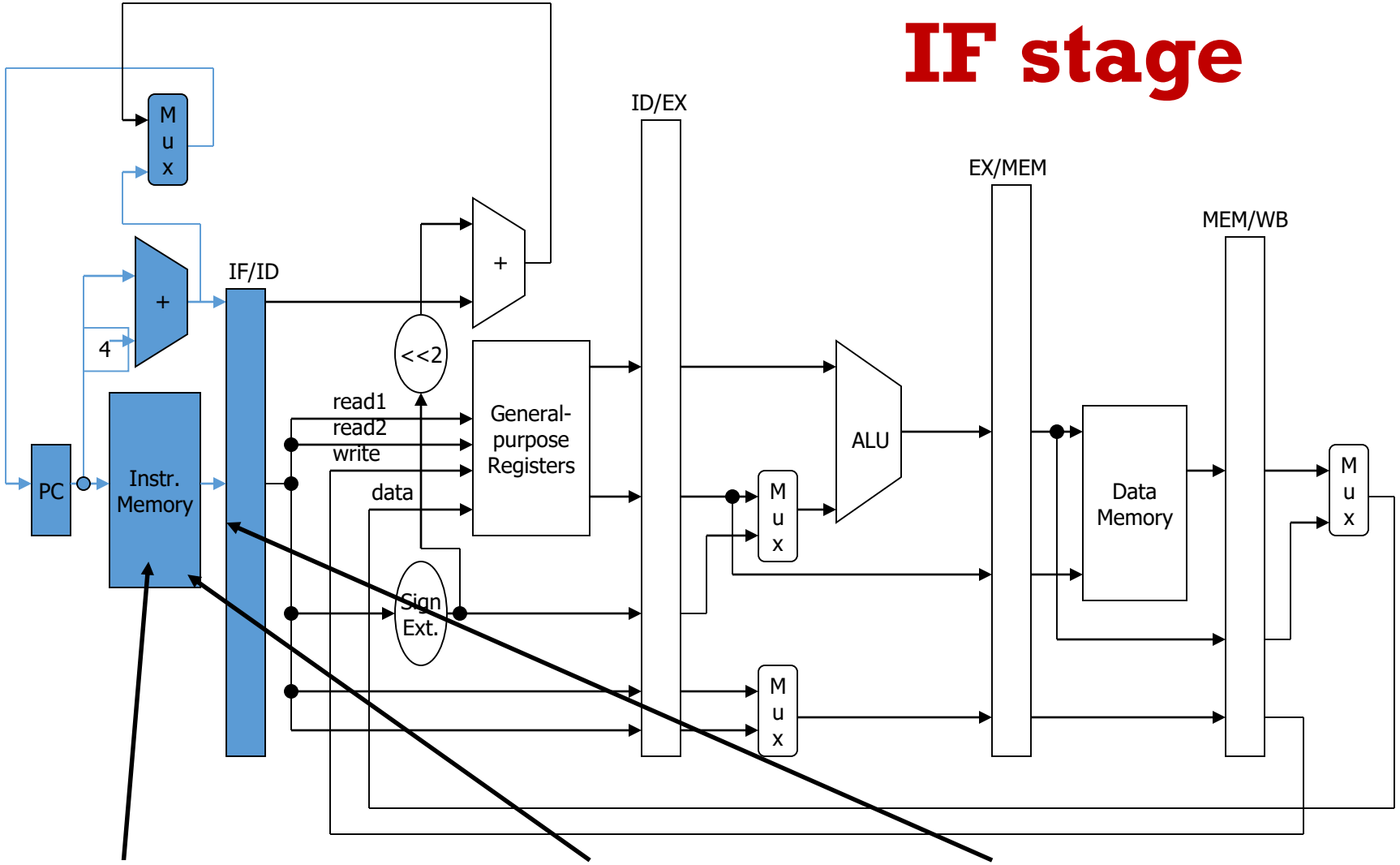
IF stage



Fetch instruction, Mem[PC].

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

IF stage

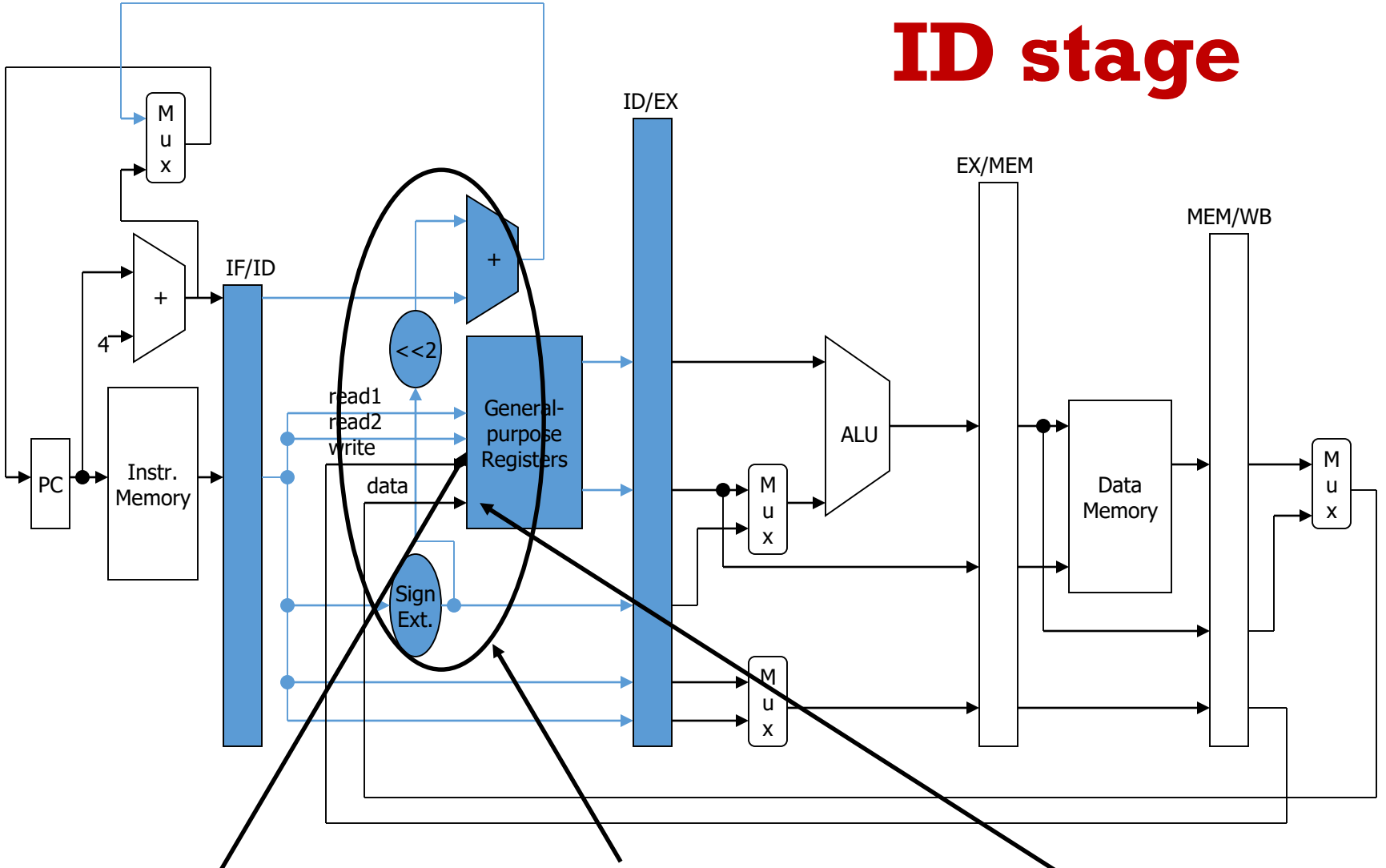


Separate instruction & data memories is a convenient. **Why?**

Accessing memory

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

ID stage

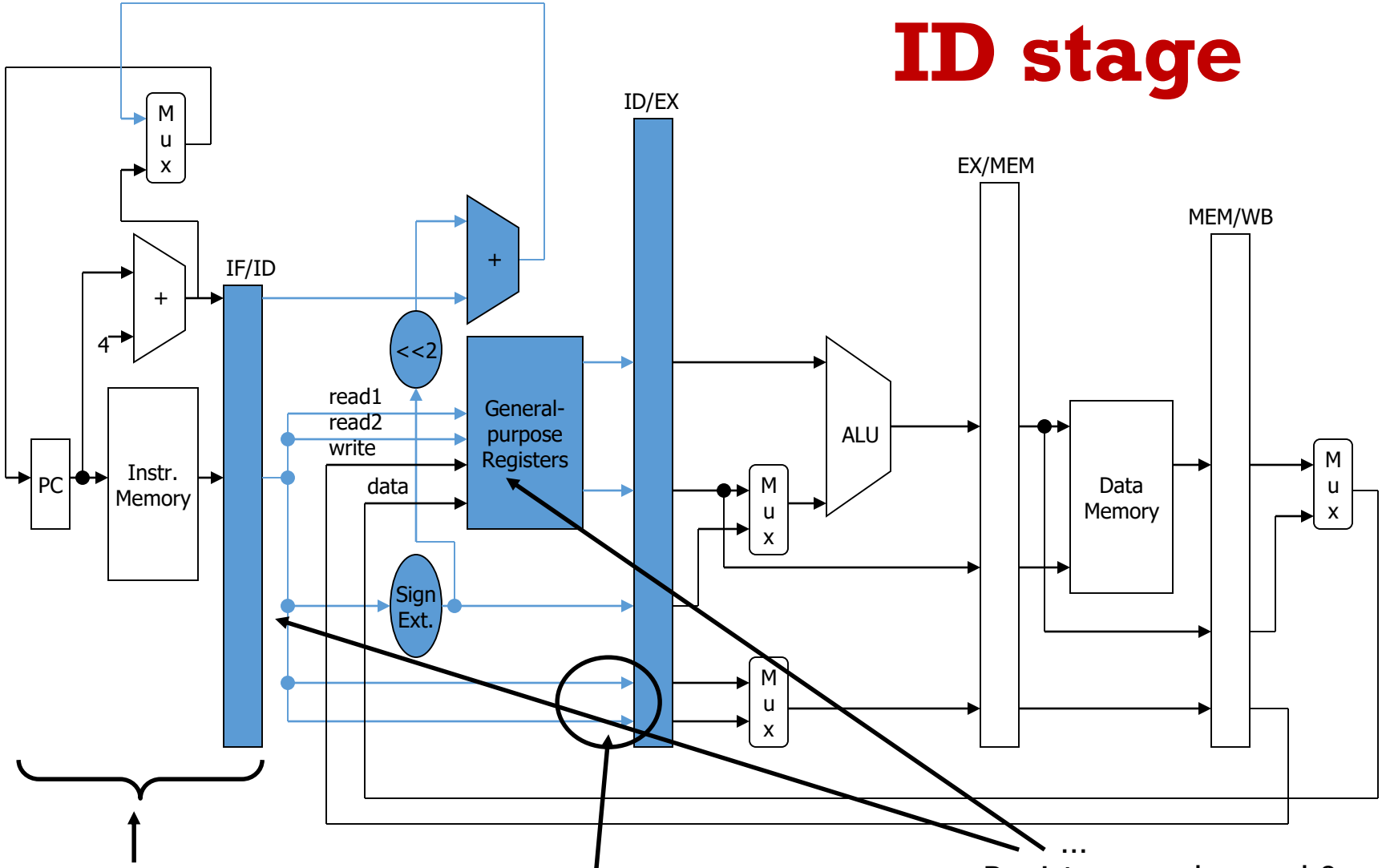


Read 2 source registers.
(Even if instr. only uses 1.)

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

Possibly write
destination register
of earlier instruction.

ID stage

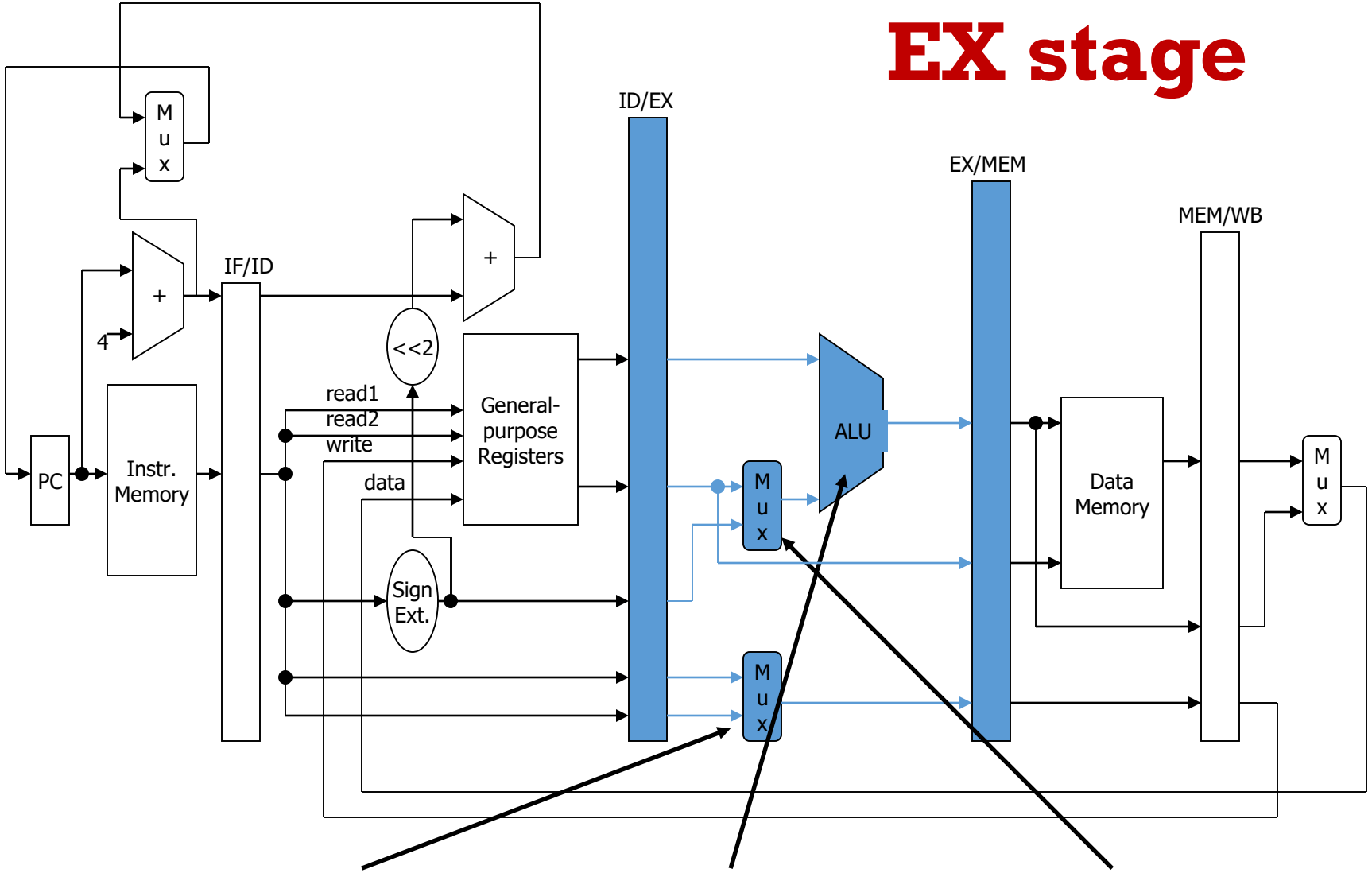


Remember, next instruction executing IF stage.

Store destination register for possible later use.

Registers can be read & written in same cycle.

EX stage

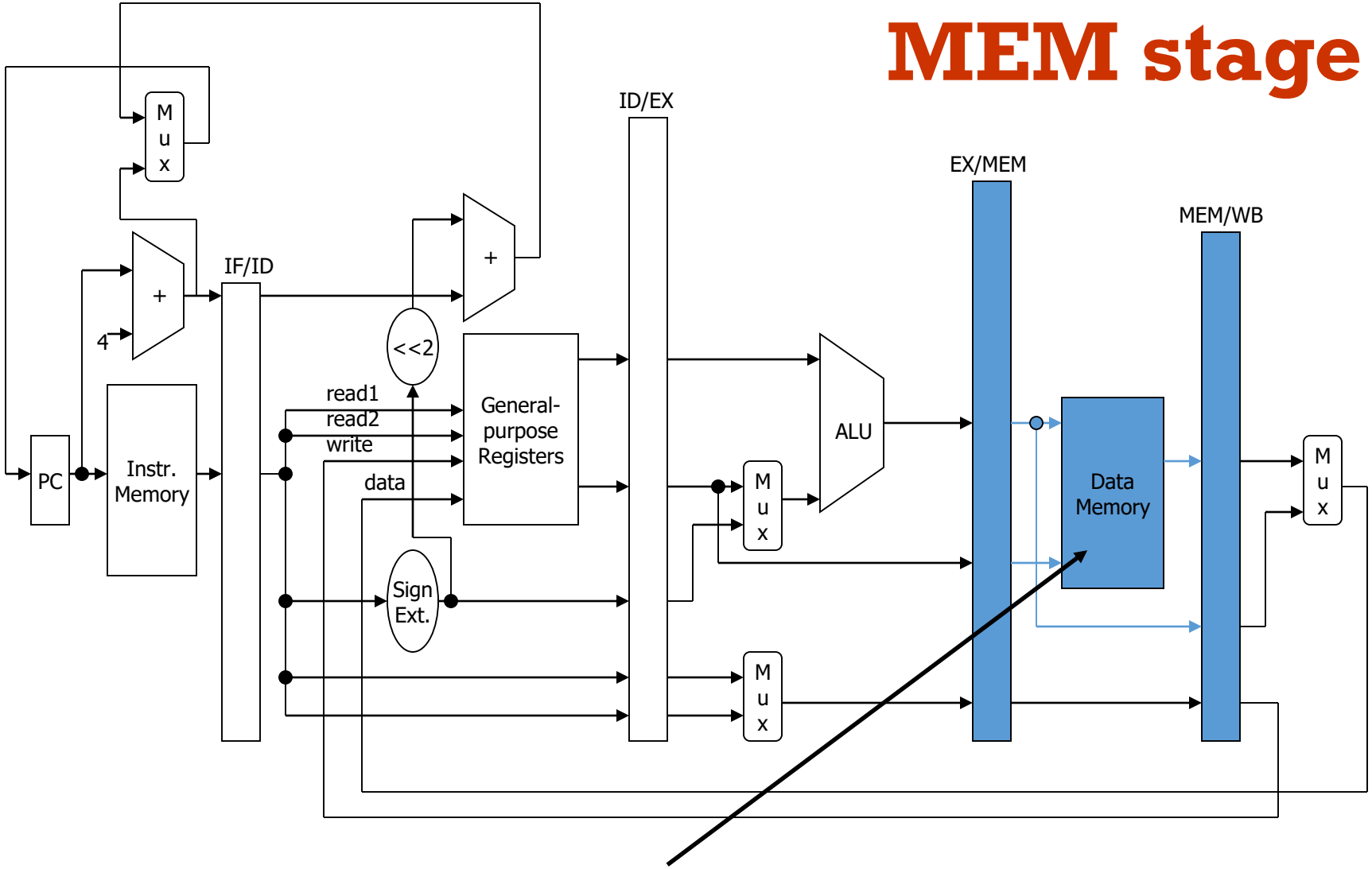


Choose which bits specify destination register.

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

…where 2nd operand is from register or immediate.

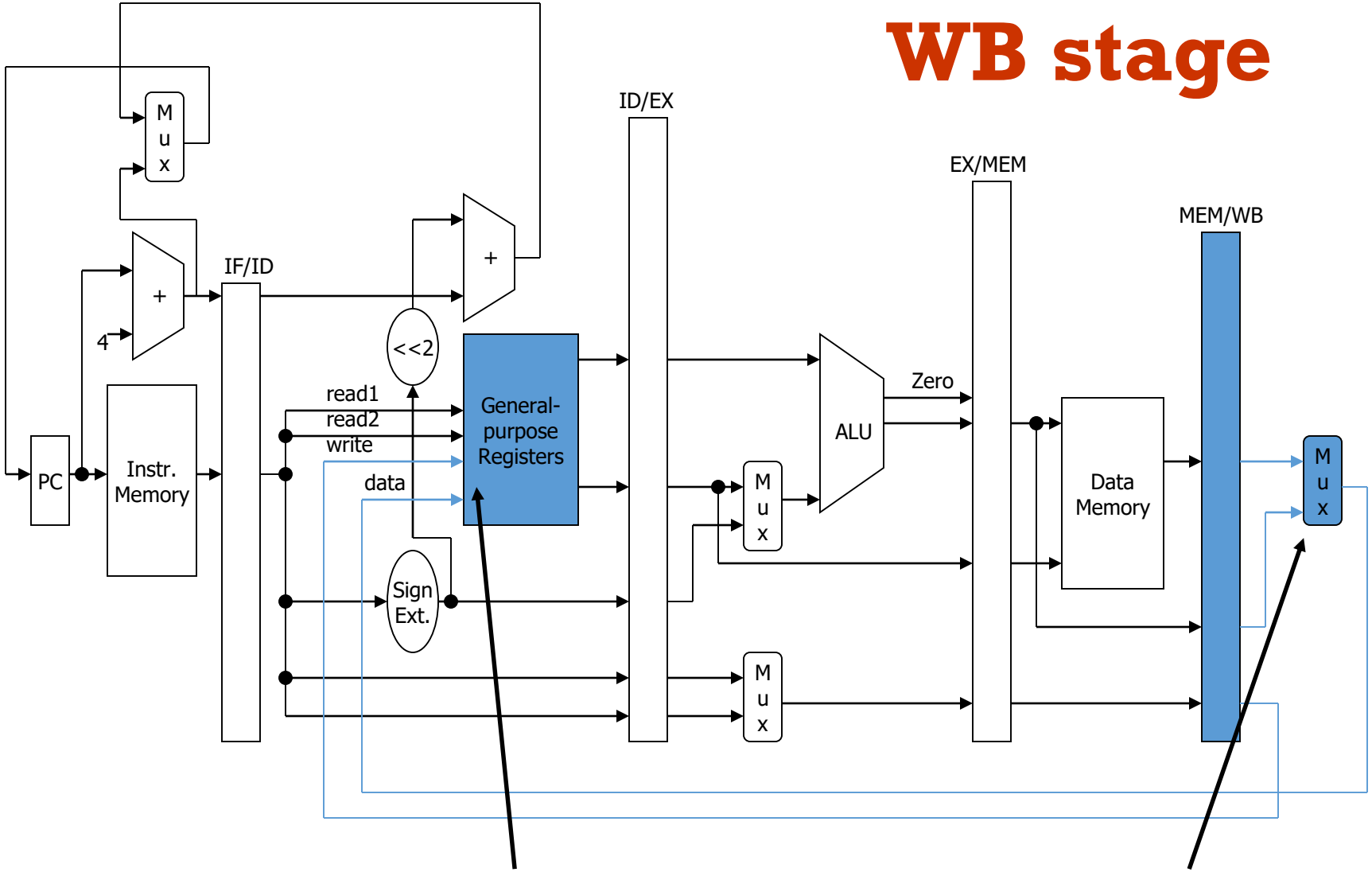
MEM stage



Possibly read or write memory.

Accessing memory

WB stage



Possibly write result into register.

Choose result from either memory or ALU.

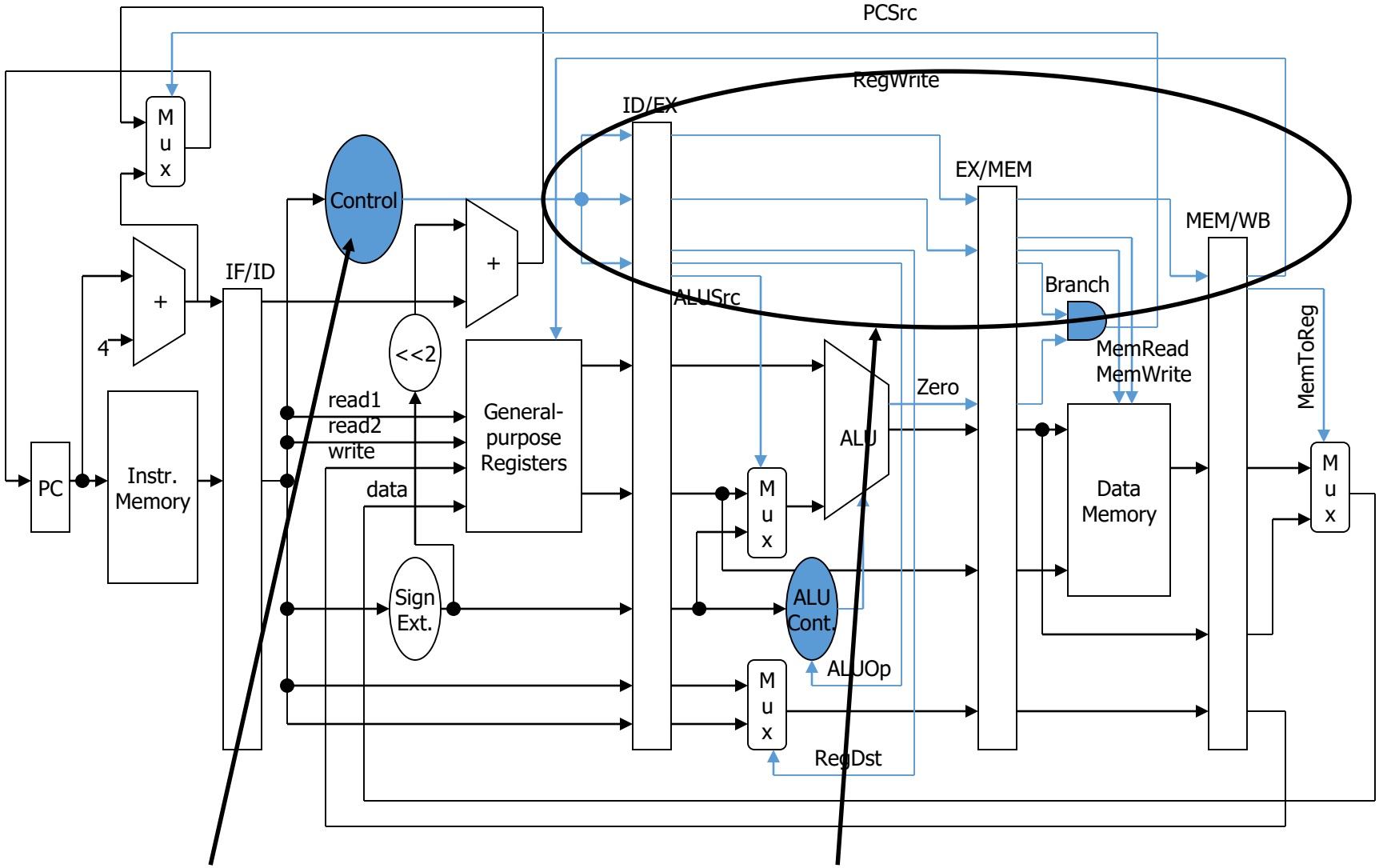
Pipeline Datapath

How would we add unconditional branches?

ID: Add adder to compute address.

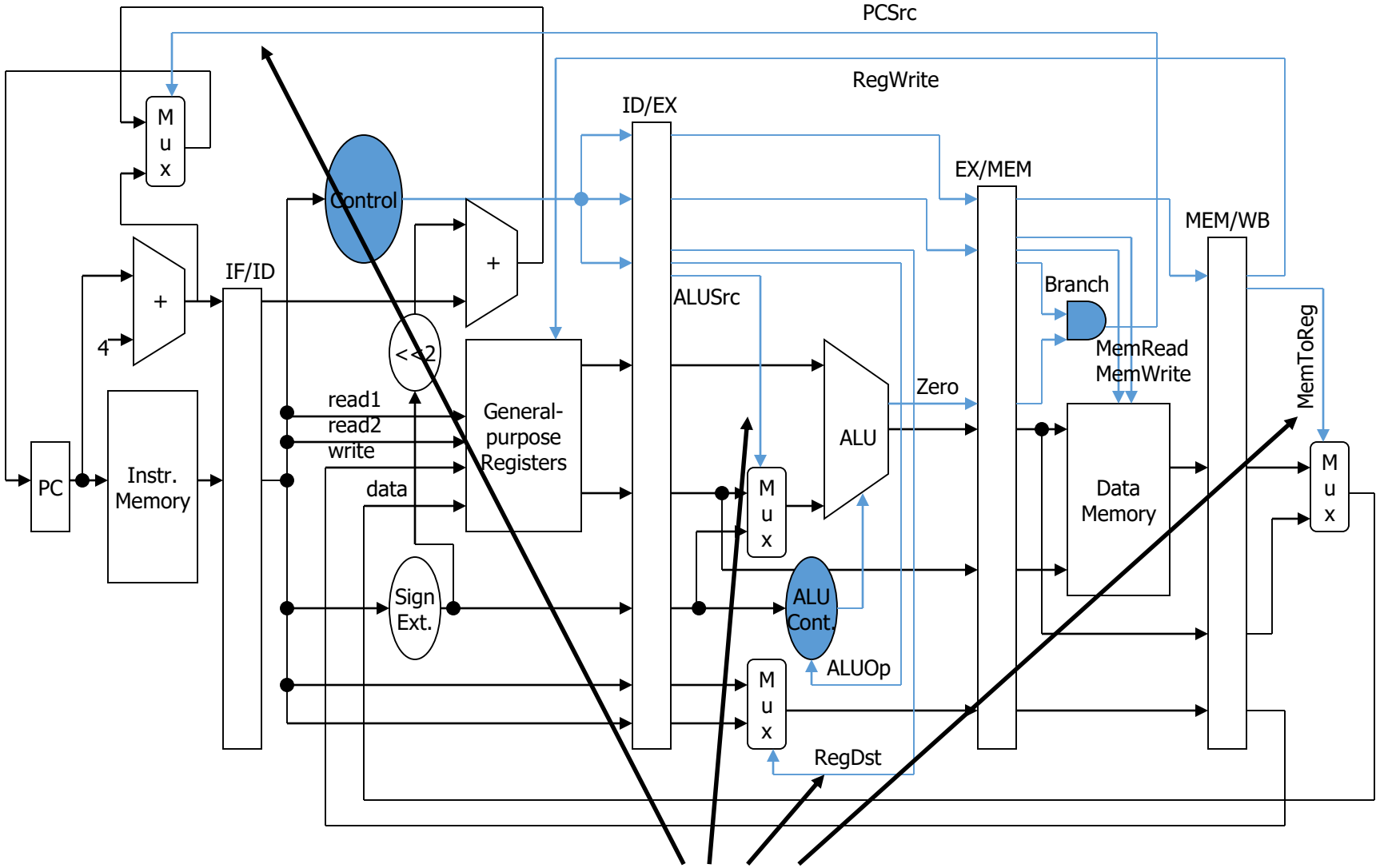
IF: Expand MUX that chooses PC's new value to have this as a third option.

Control logic activates appropriate functional units in each stage

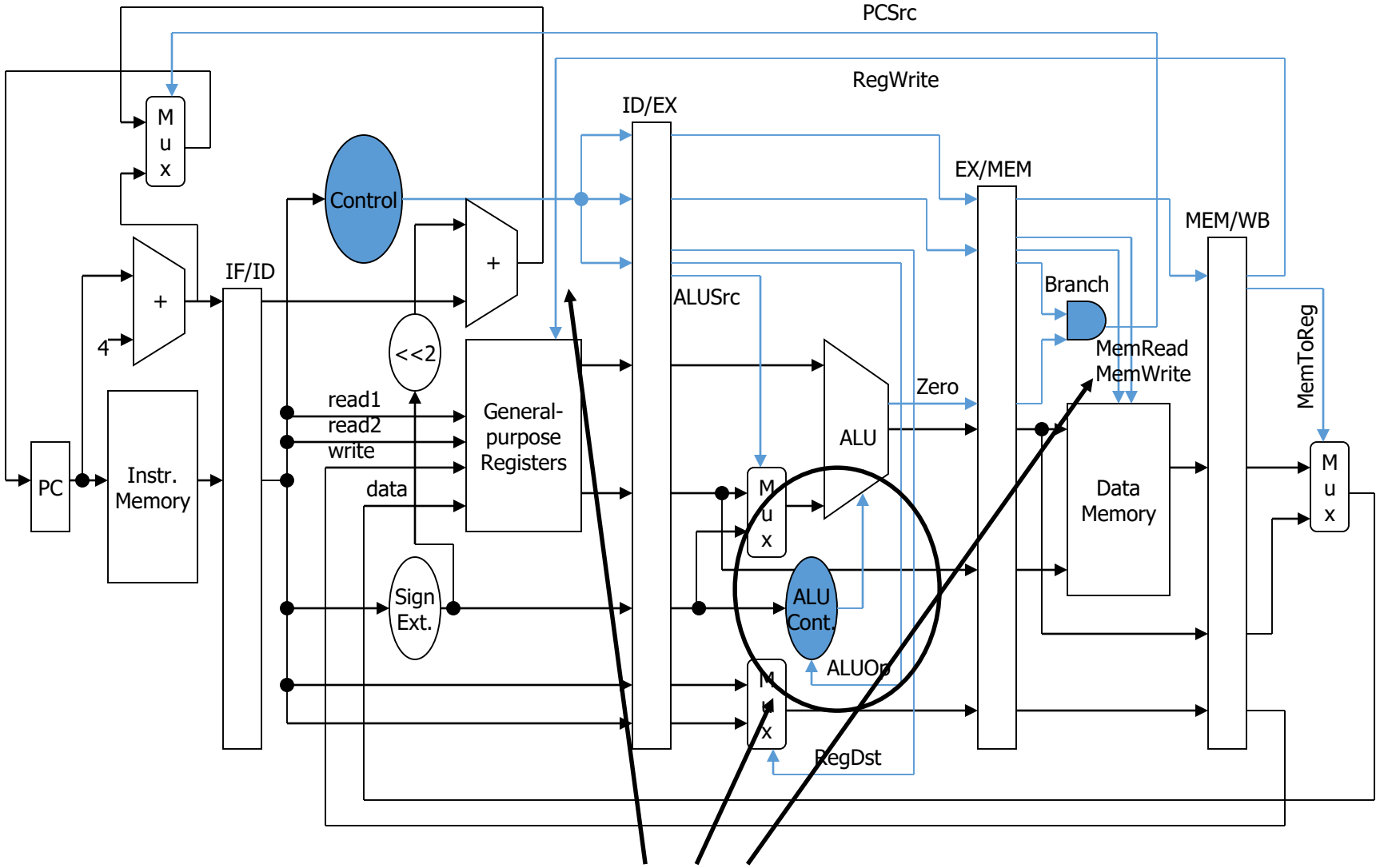


Control unit's simple combinational logic determines control bits from instruction…

…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

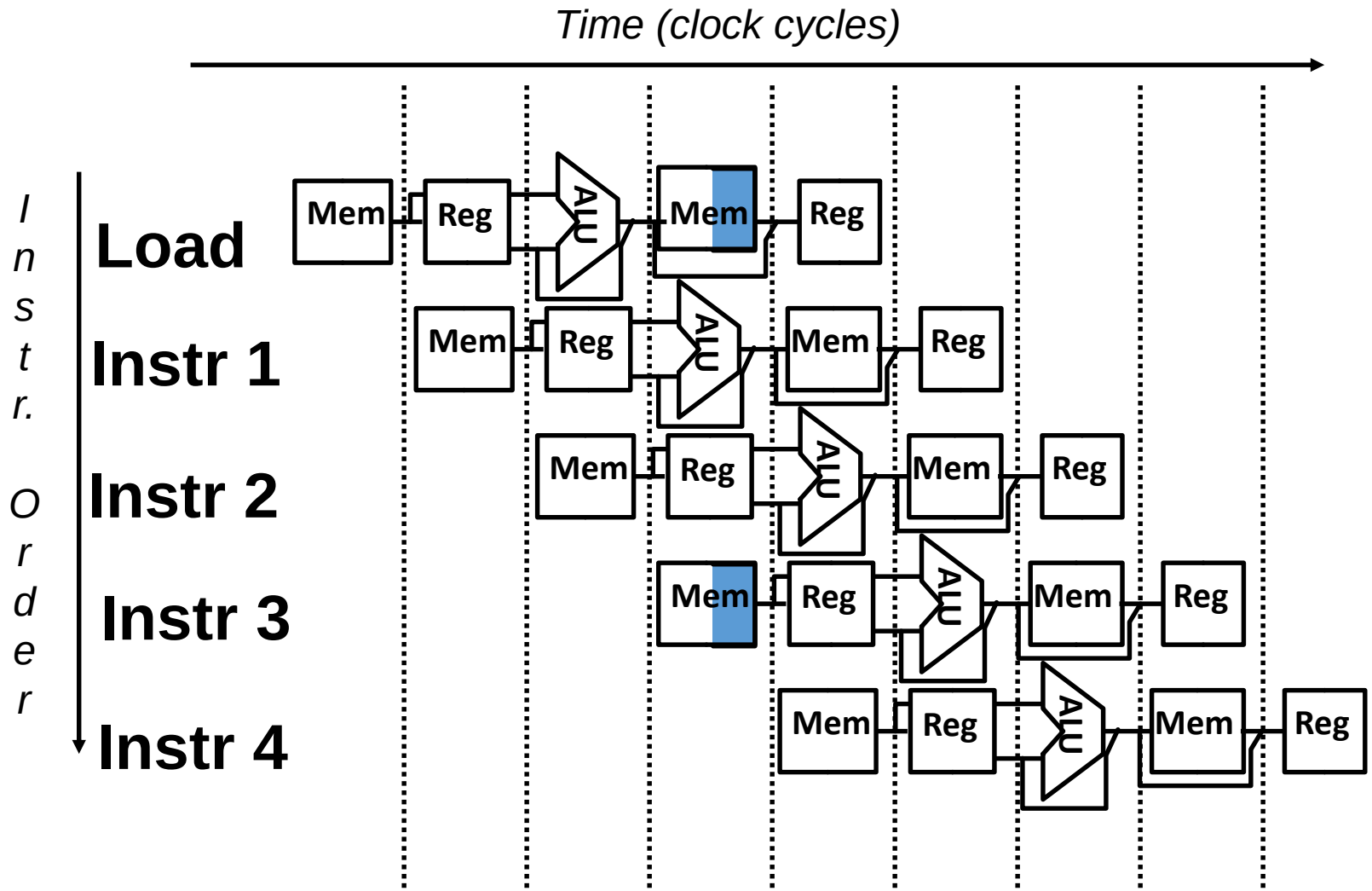
- Some instructions can't be executed in two consecutive cycles -- HAZARDS.
- Hazard types:
  - **Structural**: attempt to use the same resource two different ways at the same time
  - **Data**: attempt to use an item before it is ready

  (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:**

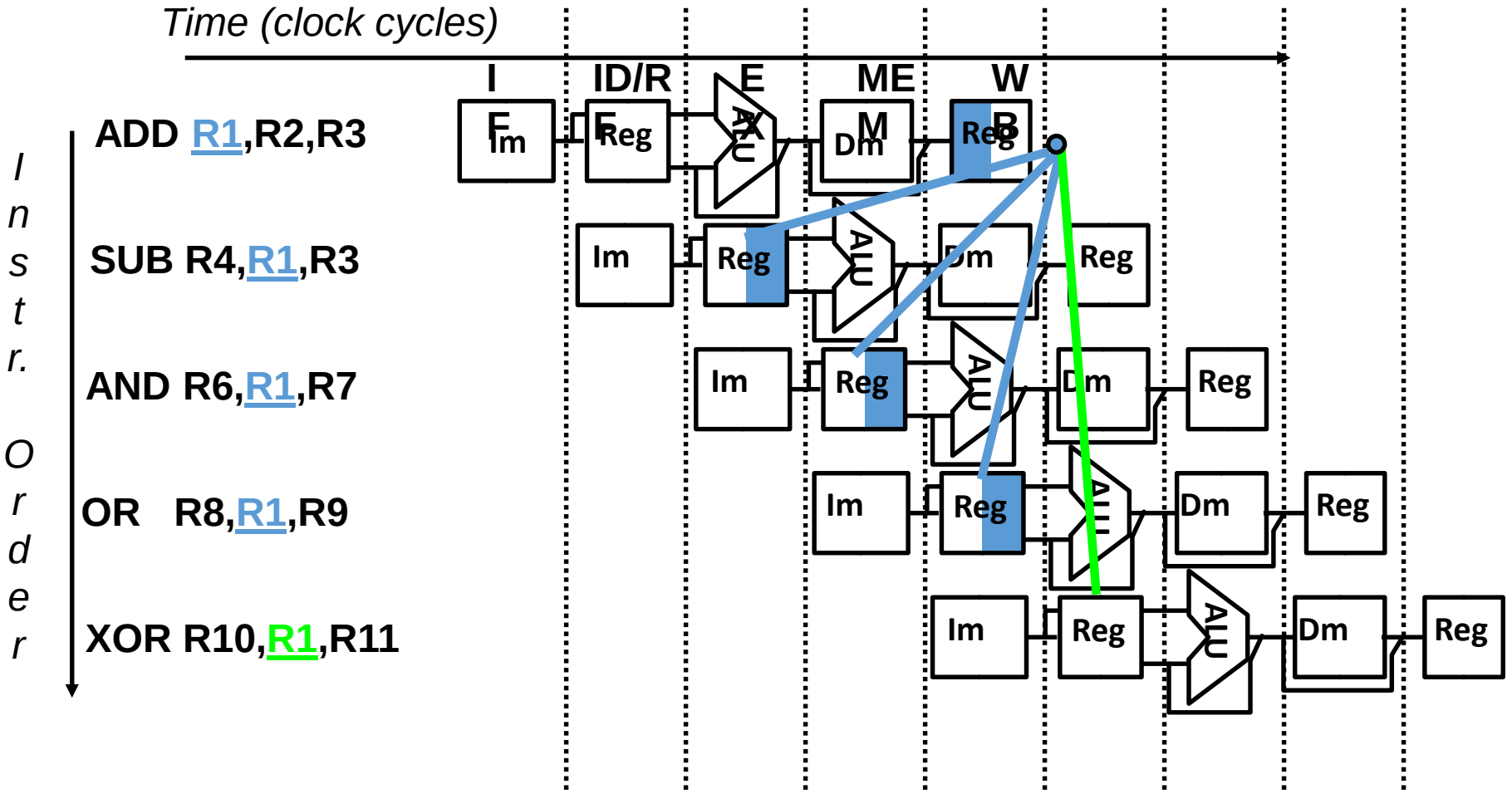
```
ADD     R1, R2, R3
SUB     R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10,R1,R11
```

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

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

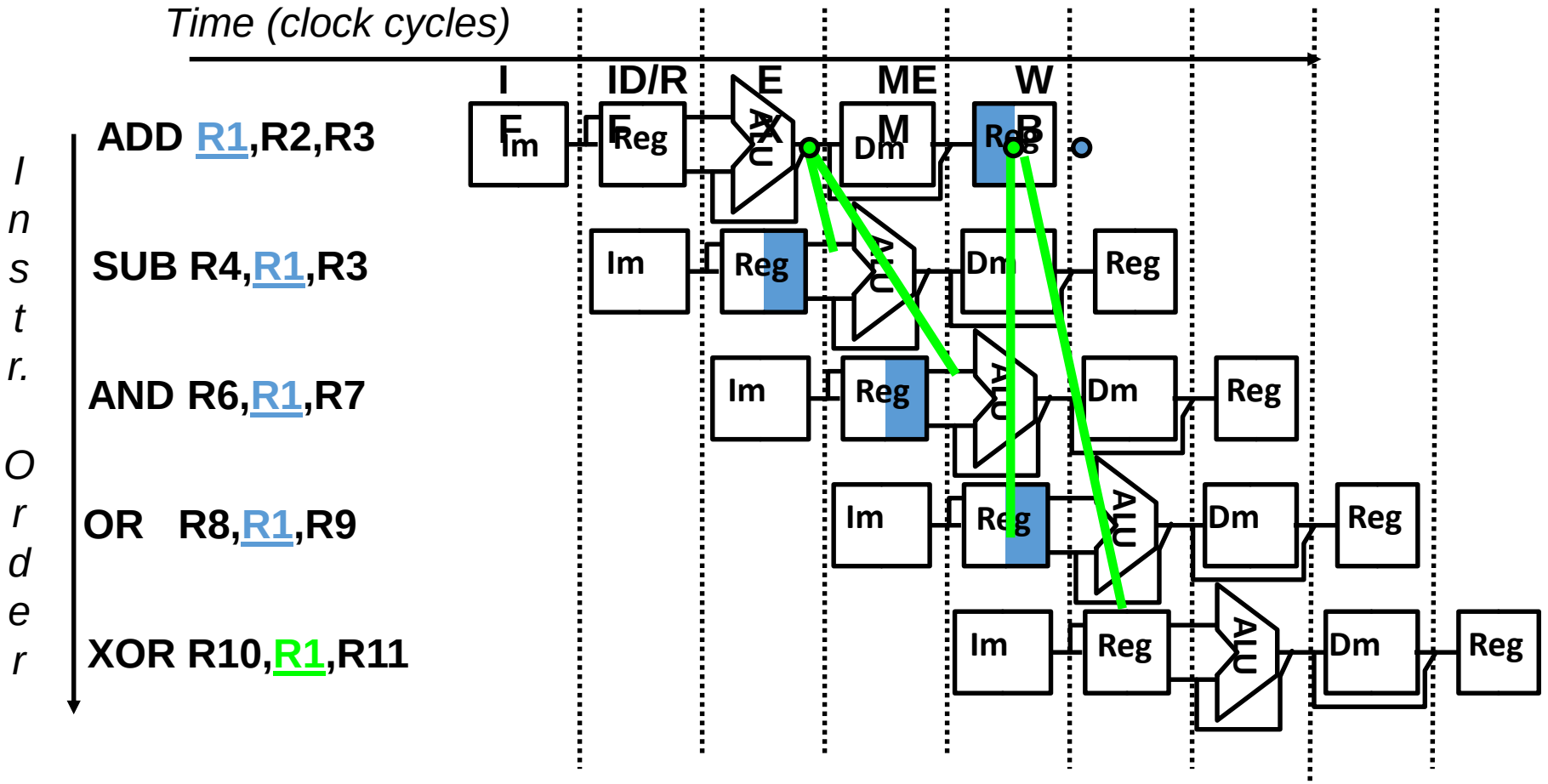
Data Hazard on R1

- **Dependencies backwards in time are hazards**



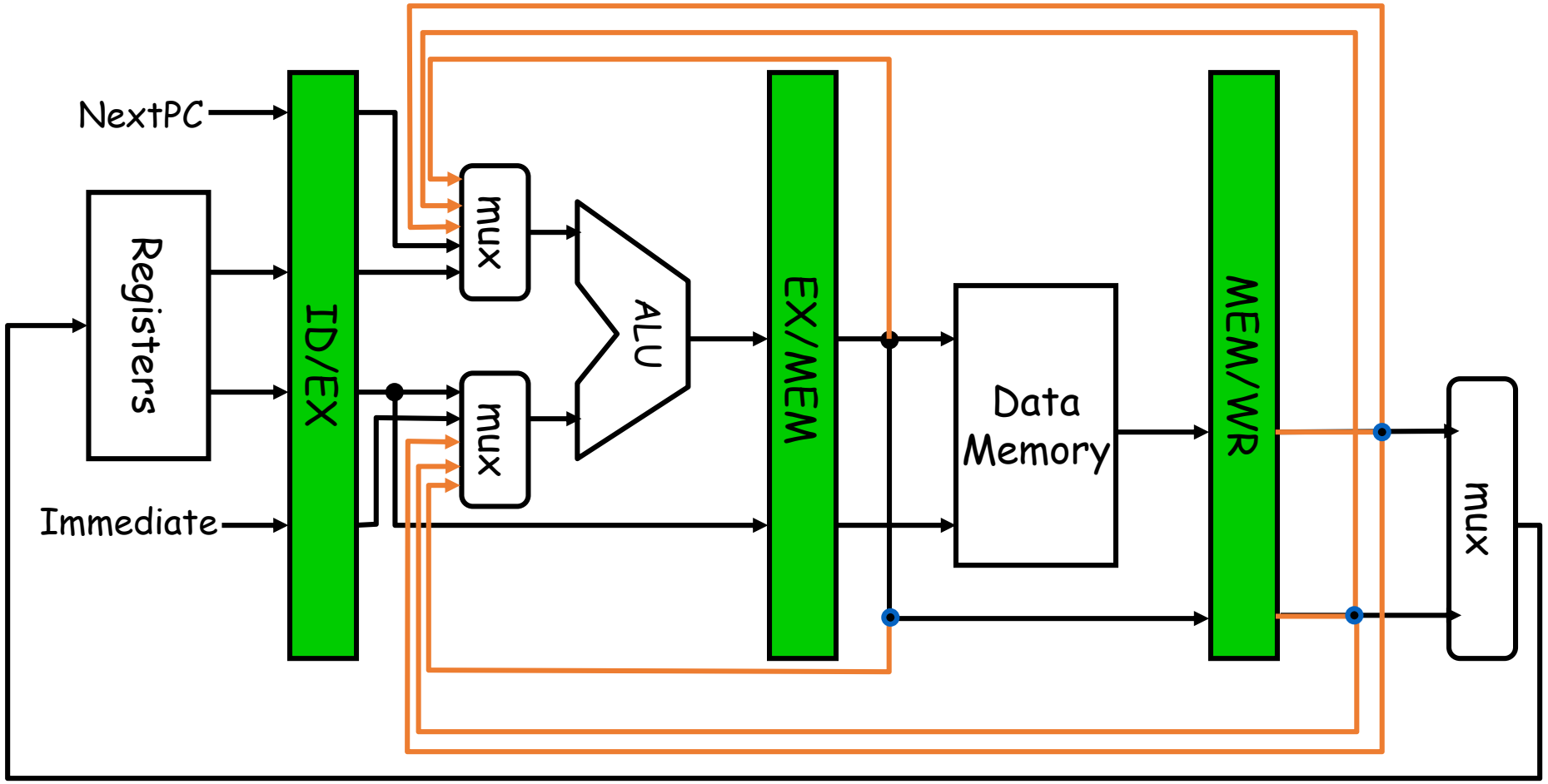
Data Hazard Solution

- **"Forward" result from one stage to another**



- **"or" OK if define read/write properly**

HW Change for Forwarding



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