Computer Architecture & Assembly Language 14:332:331

Lecture 7
Pipelining

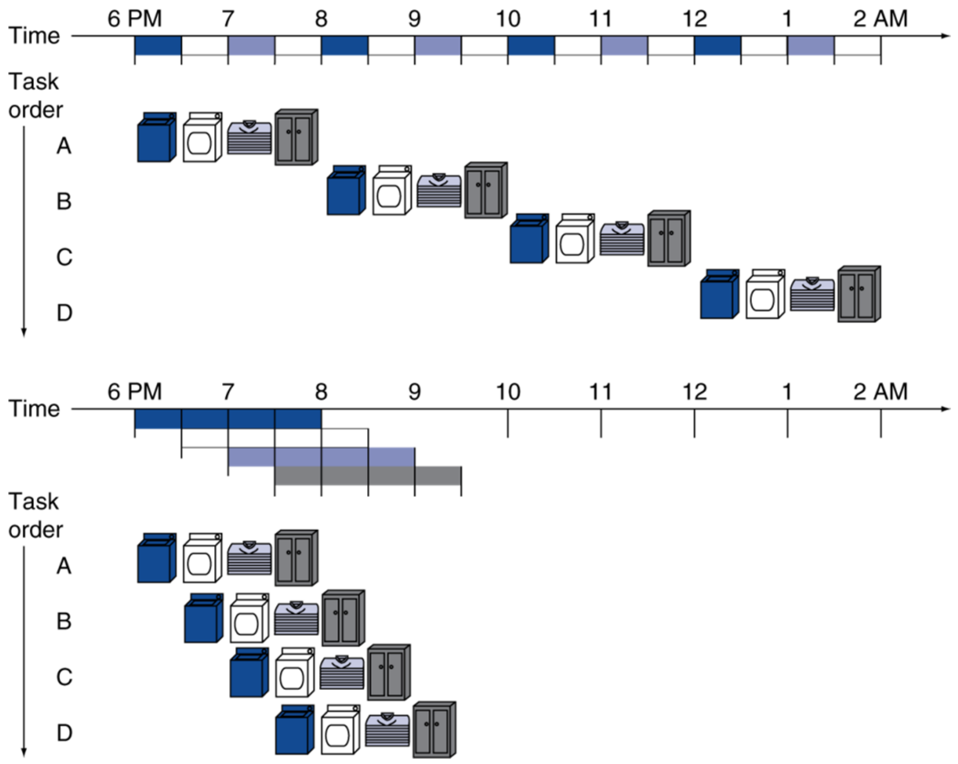
Naghmeh Karimi

Fall 16

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier, *and Computer Organization and Design*, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier and Mary Jane Irwin's slides from Penn State University.

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3

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

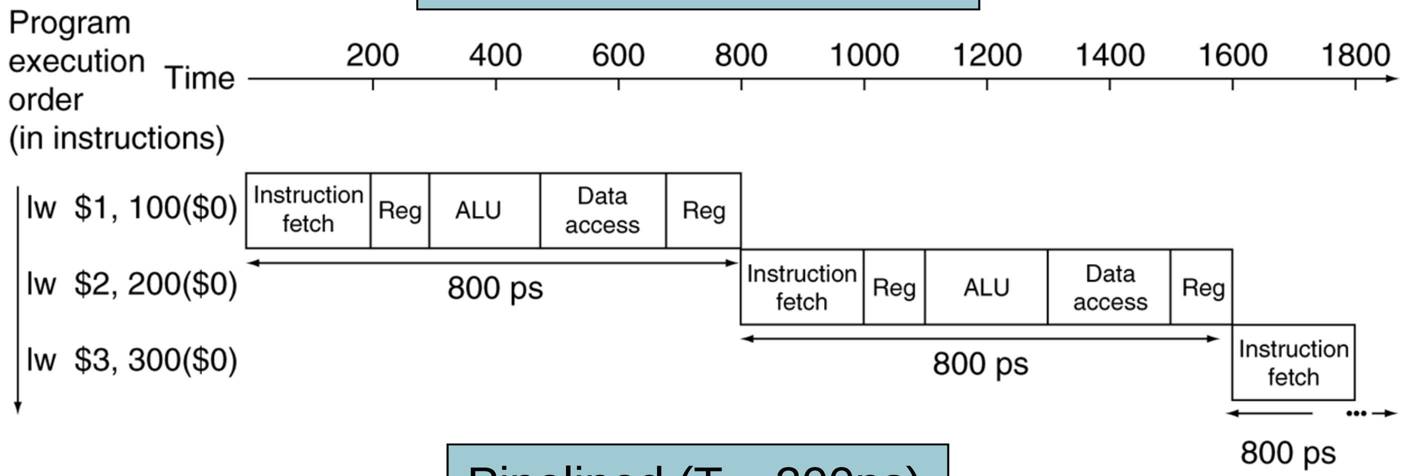
Pipeline Performance

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

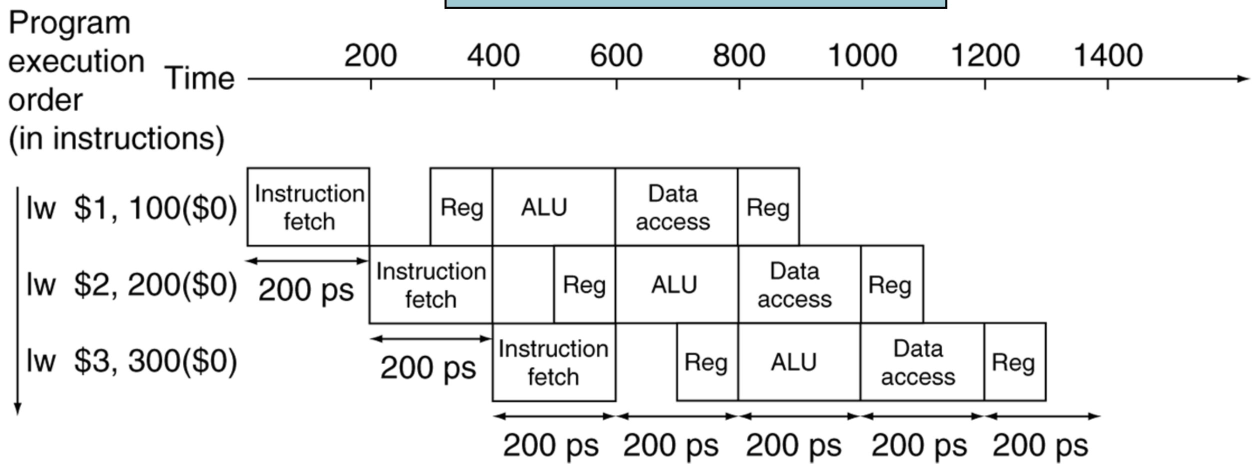
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



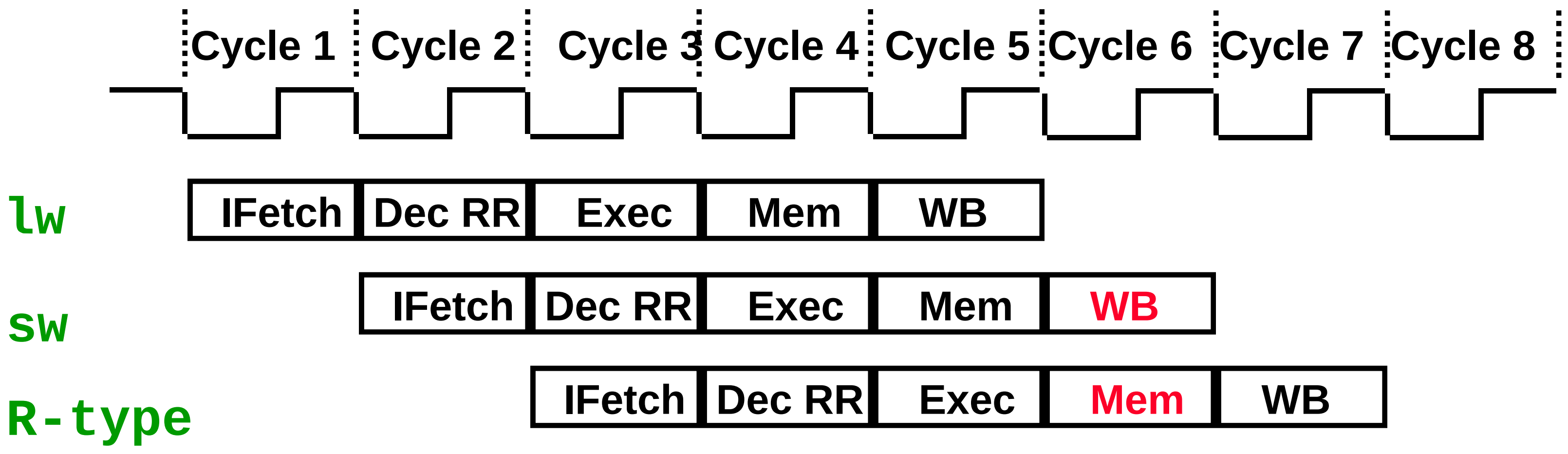
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $\text{Time between instructions}_{\text{pipelined}}$

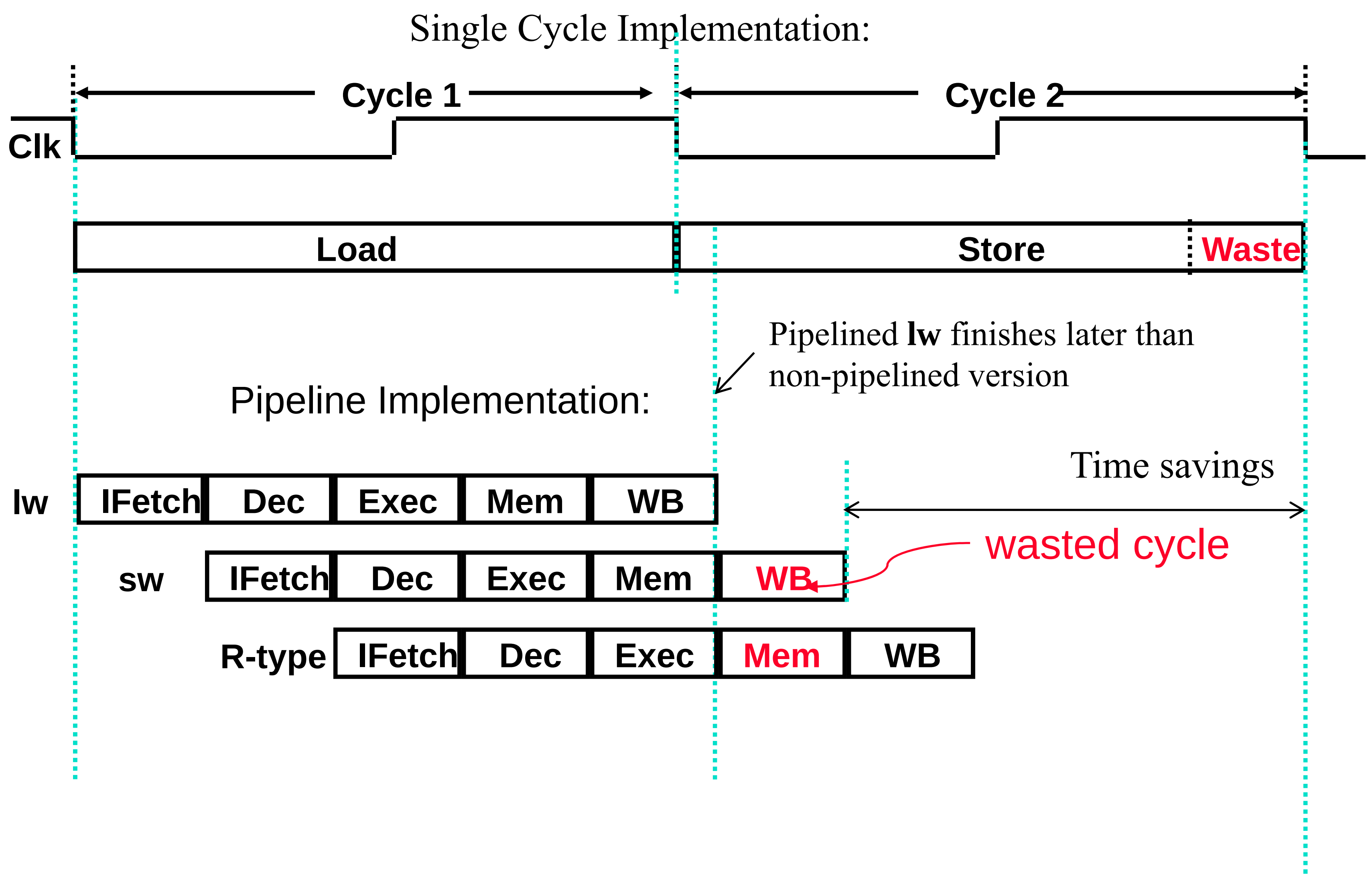
    $$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelined MIPS Processor

- Start the next instruction while still working on the current one



Single Cycle vs. Pipelined



MIPS Pipeline Datapath Modifications

- What do we need to add/modify in our single-cycle per instruction datapath to make it pipelined?

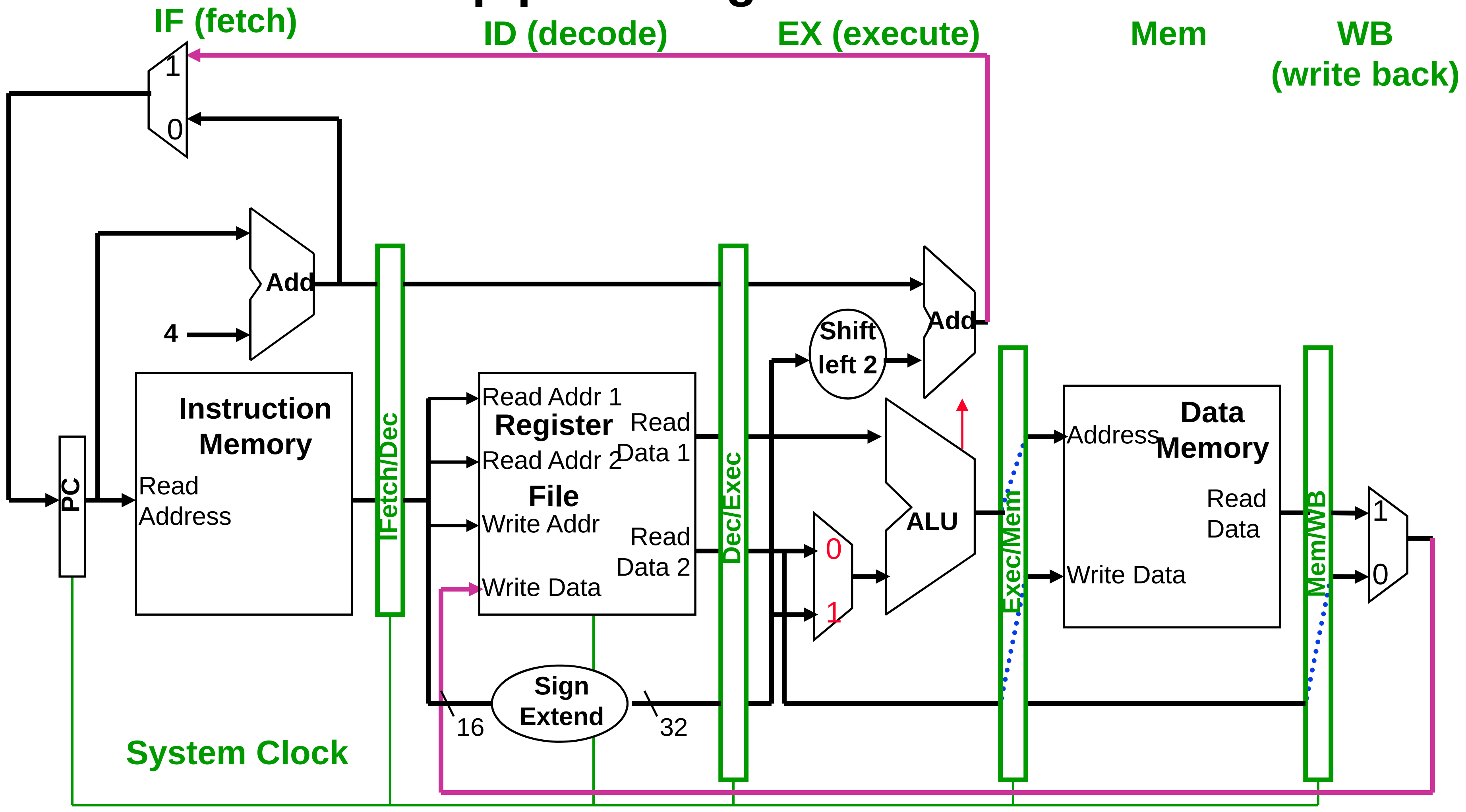
- The MIPS instruction has (up to) five stages, thus pipeliene has 5 stages:
  - **Ifetch** to fetch the instruction from Instruction memory
  - **Dec** to decode the instruction and read Register File registers
  - **Exec** to do the ALU operations
  - **Mem** to read from/write into Data Memory
  - **WB** to write back into the register file.

→ So we need a way to separate the data path into five pieces, without losing intermediate results.

- We will introduce Pipeline registers between stages to isolate them and store intermediate results

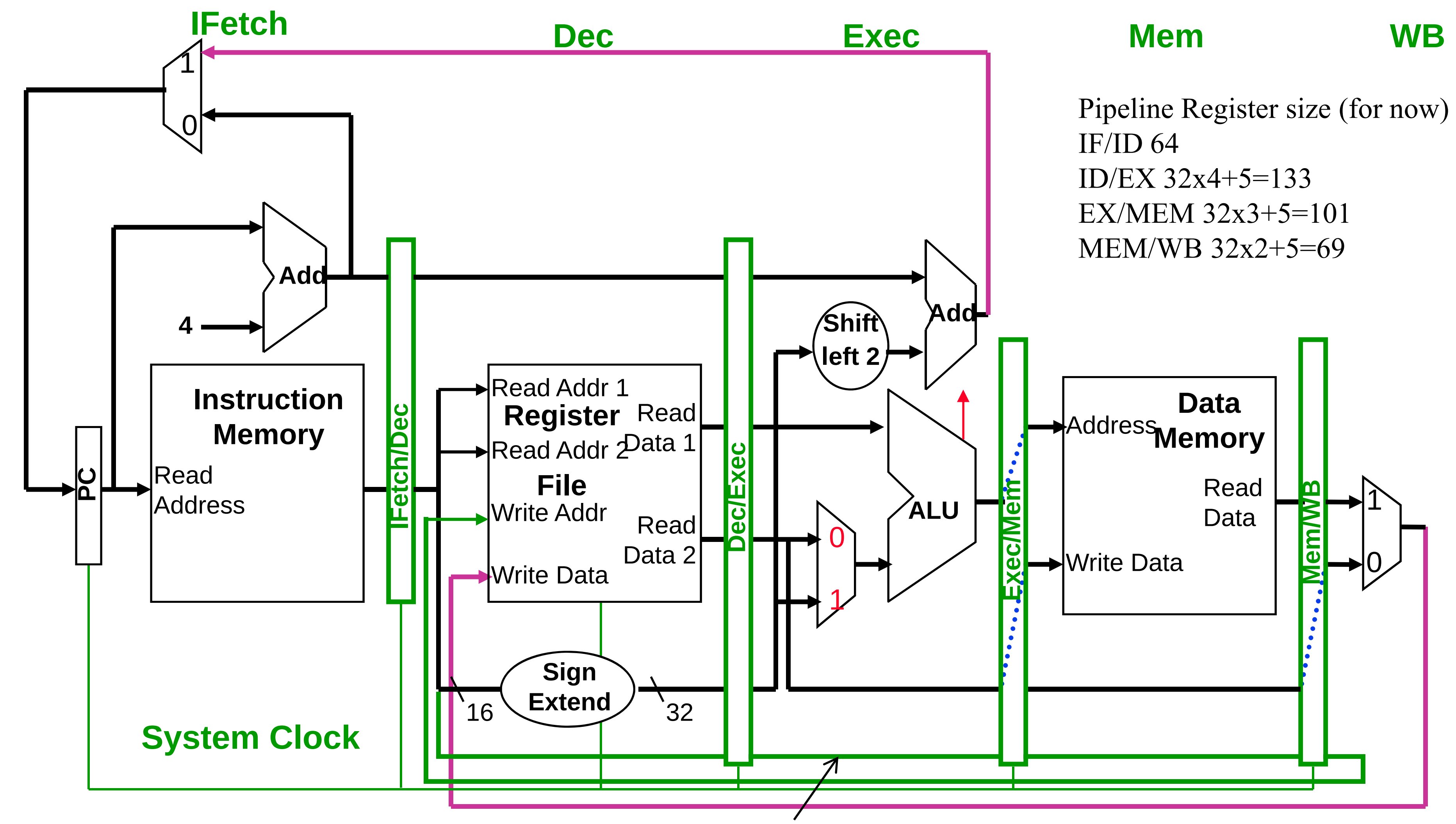
MIPS Pipeline Datapath Modifications

- All instructions *advance* during one clock cycle between one pipeline register and the next



MIPS Pipeline Datapath Modifications

- Because all data is passed through the pipeline, the address of the register where data needs to be loaded (lw) also needs to be passed

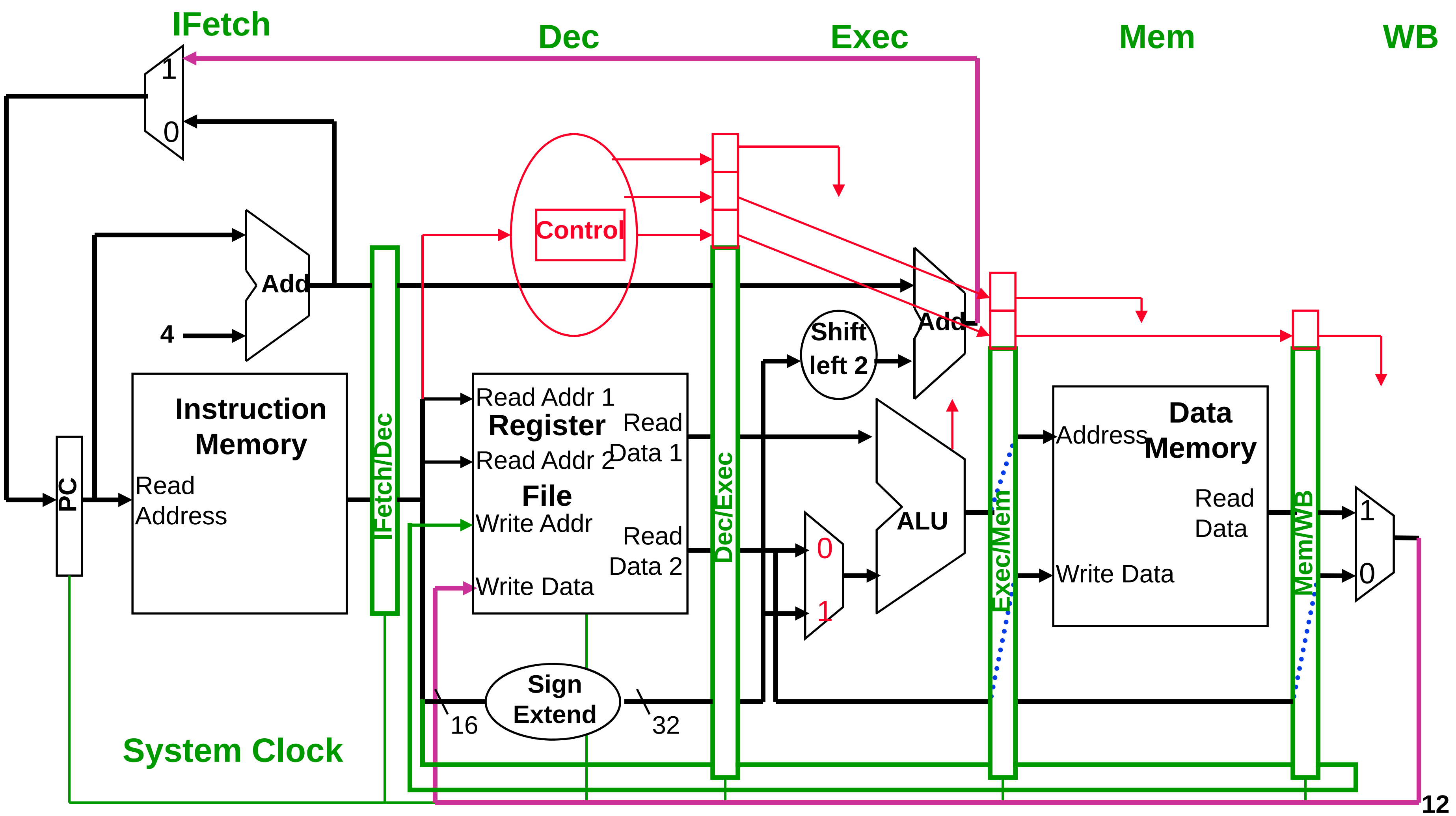


Pipeline Register size (for now)
IF/ID 64
ID/EX 32x4+5=133
EX/MEM 32x3+5=101
MEM/WB 32x2+5=69

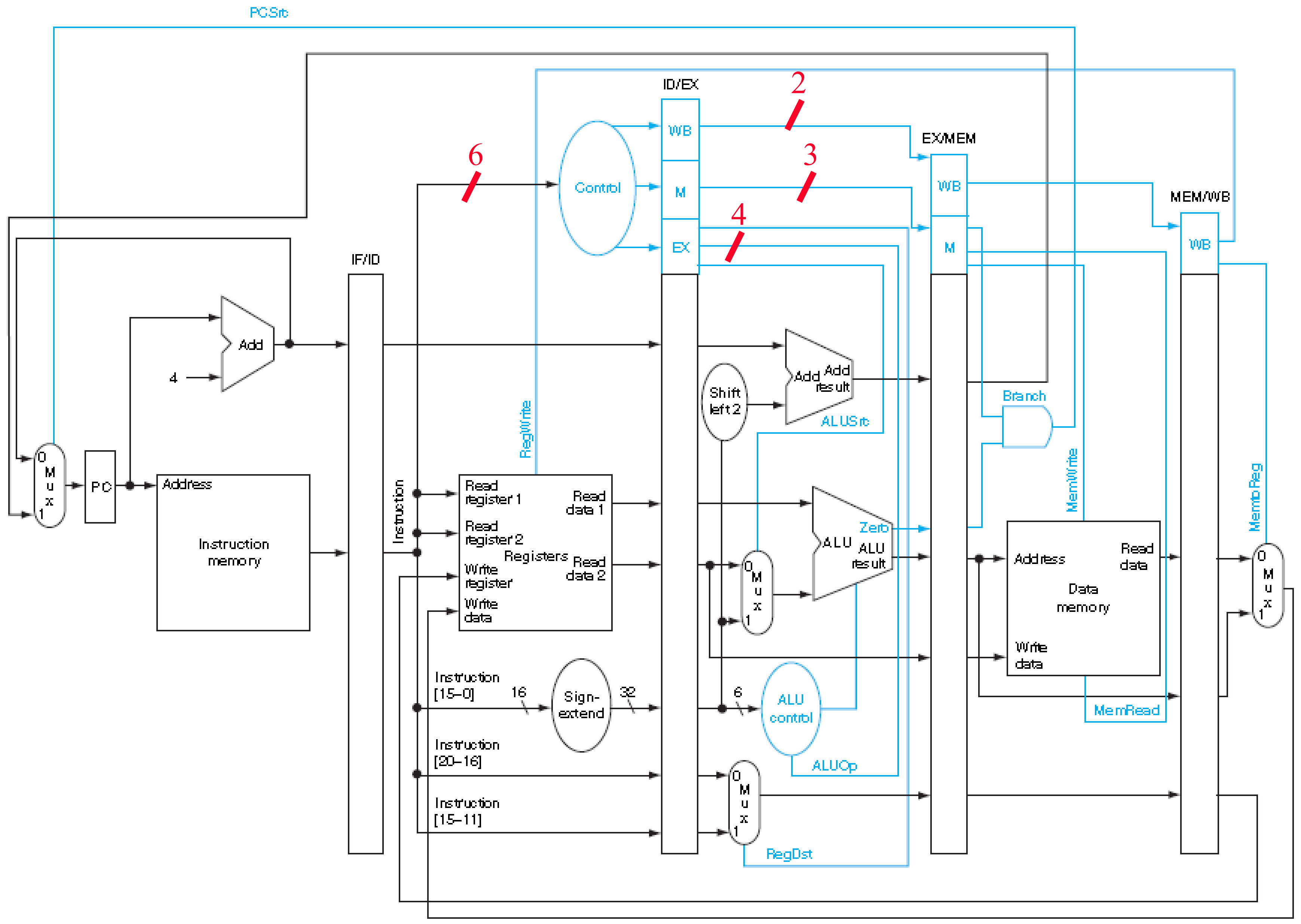
Extends pipeline reg. to hold address of destination reg.

MIPS Pipeline Control Path Modifications

- All control signals are determined during Decode and held in the pipeline registers between pipeline stages



MIPS Pipeline Control Path Modifications

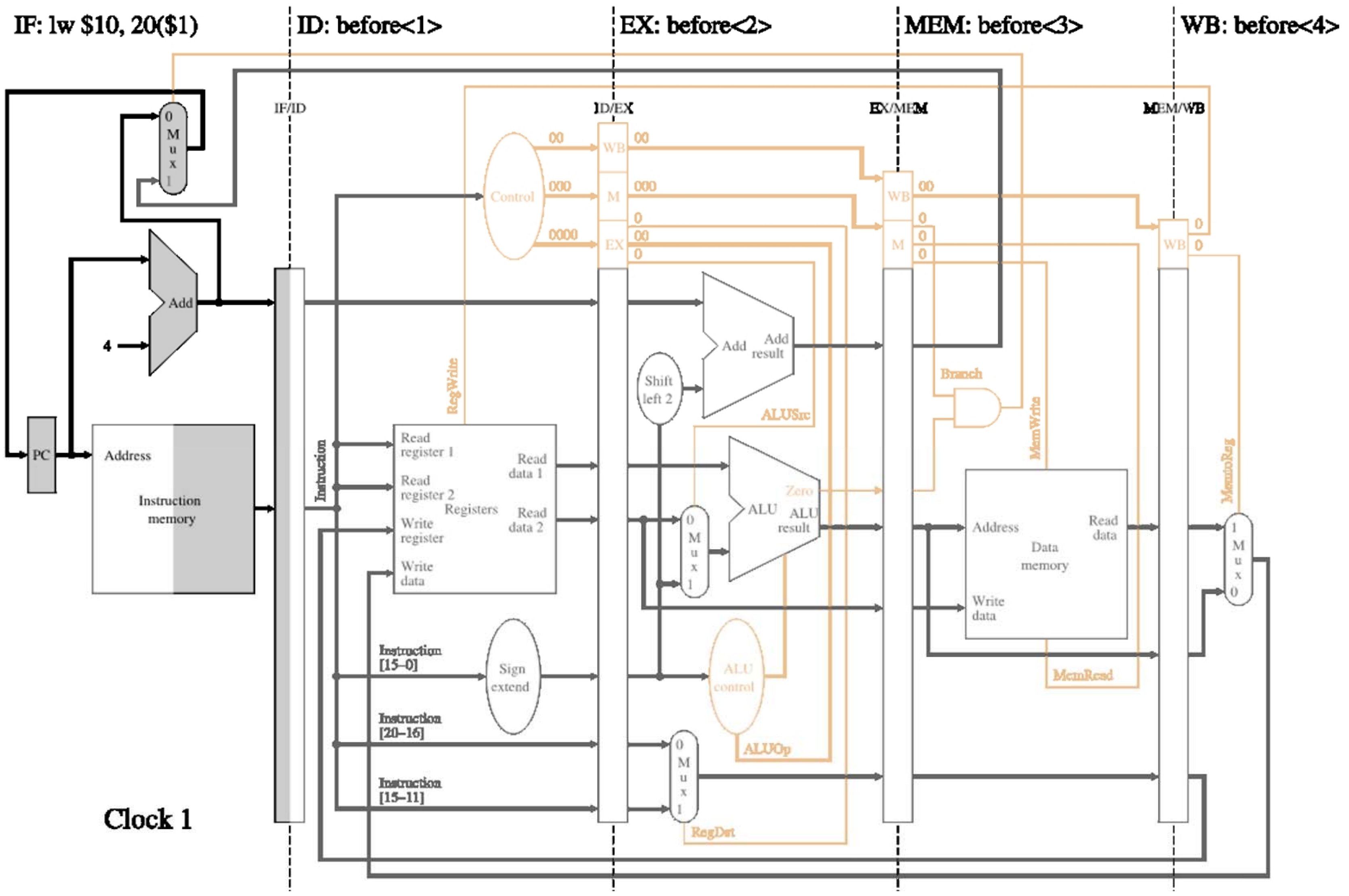


Pipeline Example

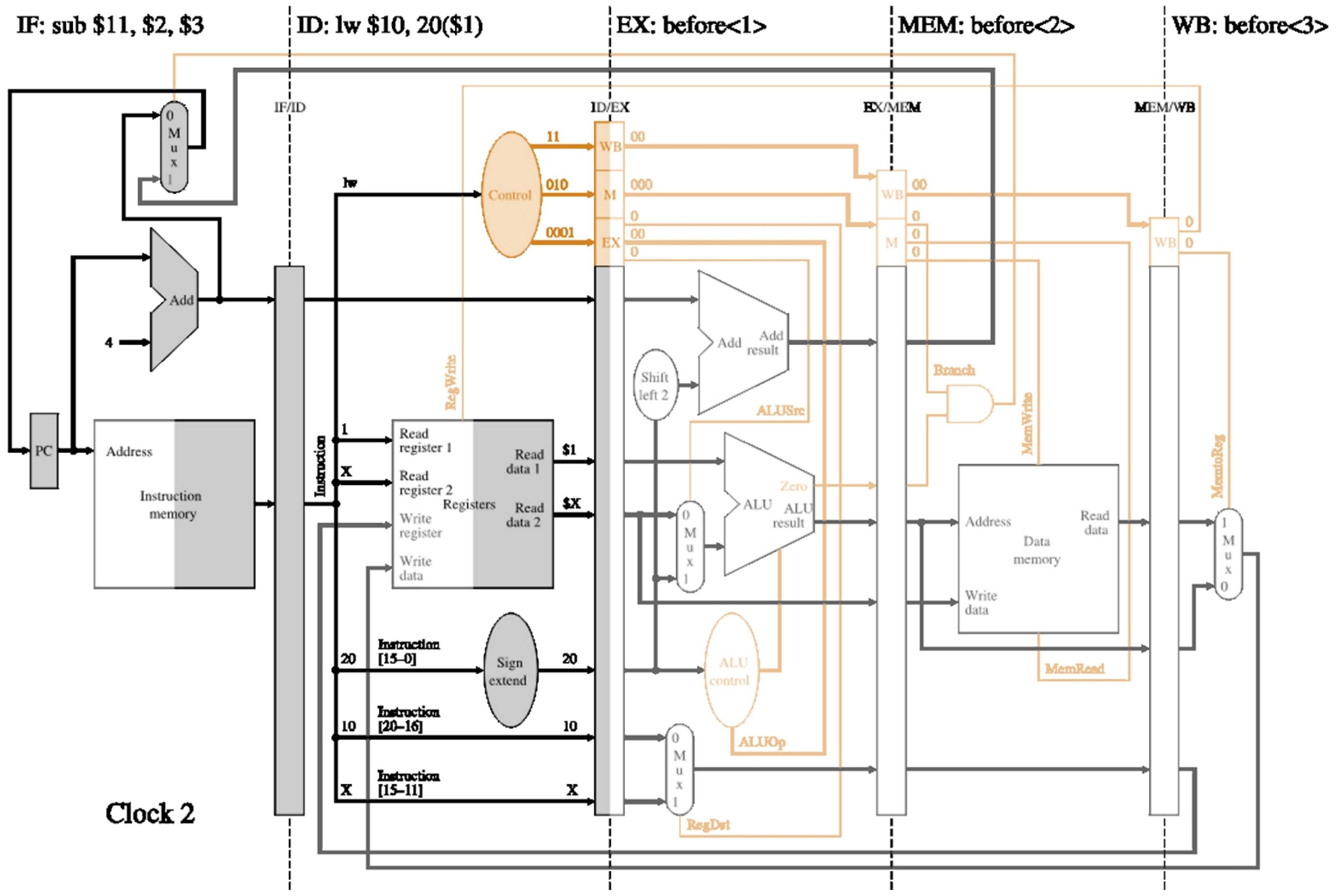
- How does the non-dependent instruction sequence execute in a pipeline ? (no support for forwarding)

```
before <4>
before <3>
before <2>
before <1>
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
after <1>
after <2>
```

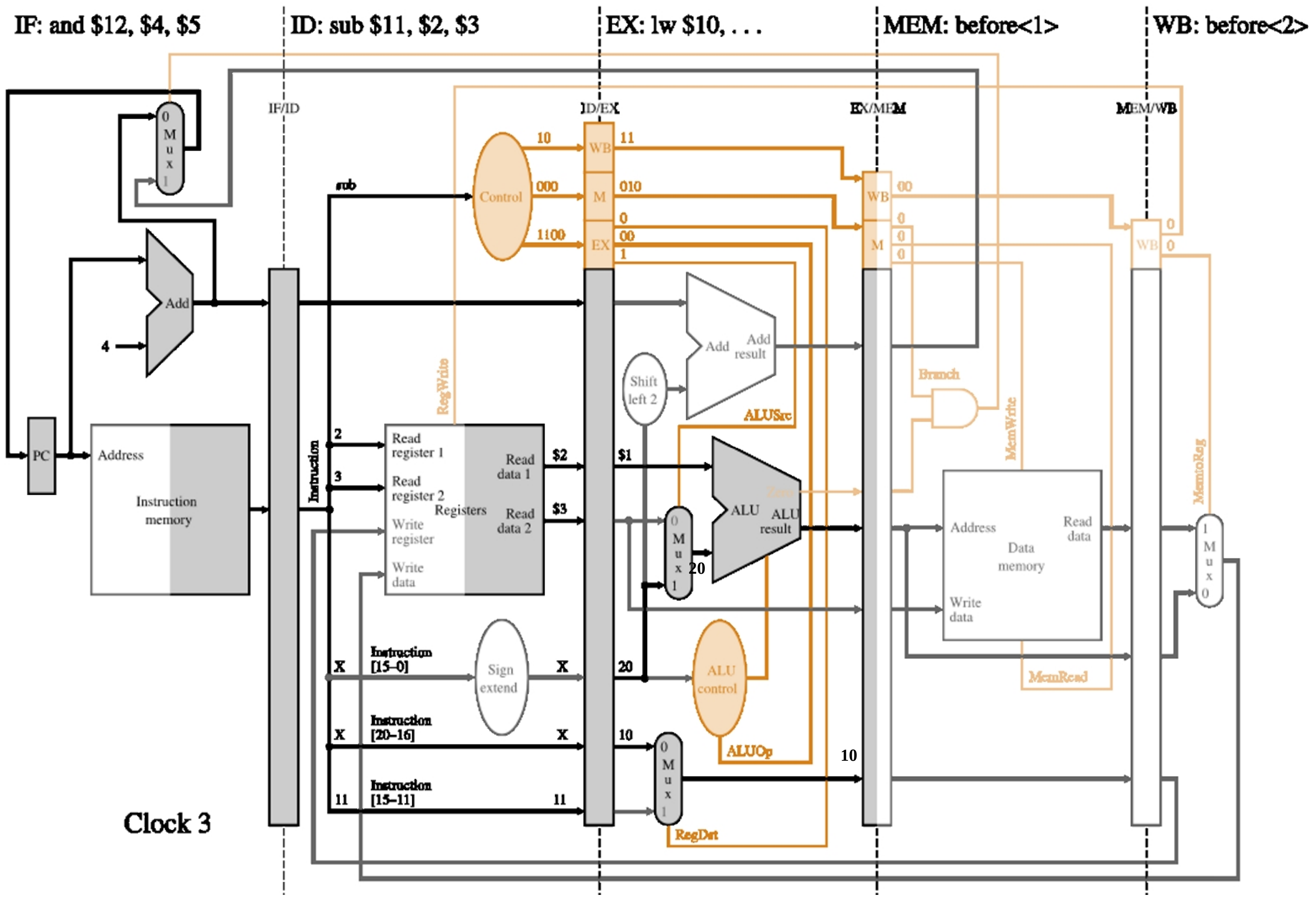
Pipeline Example - before <4> completes



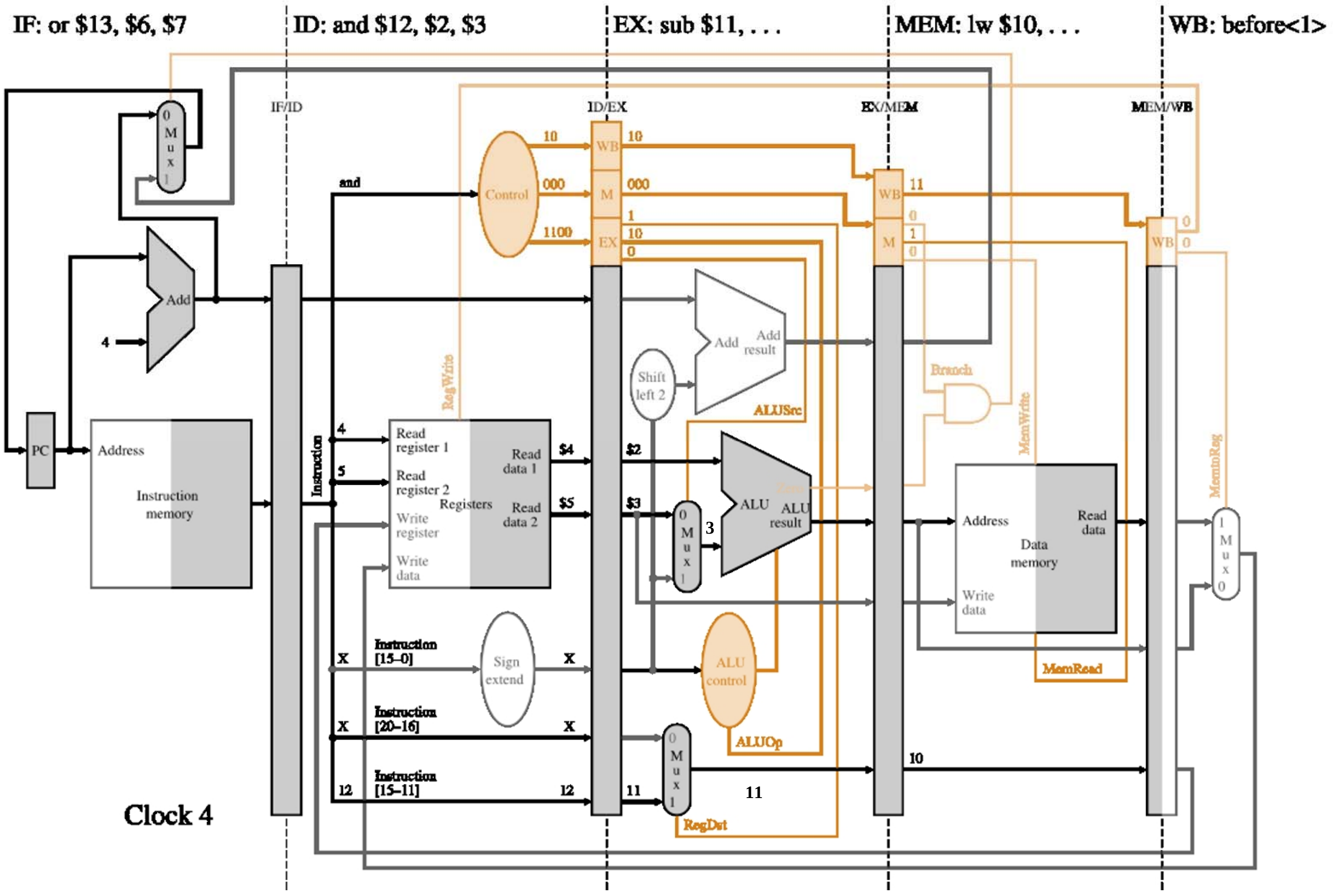
Pipeline Example - before <3> completes



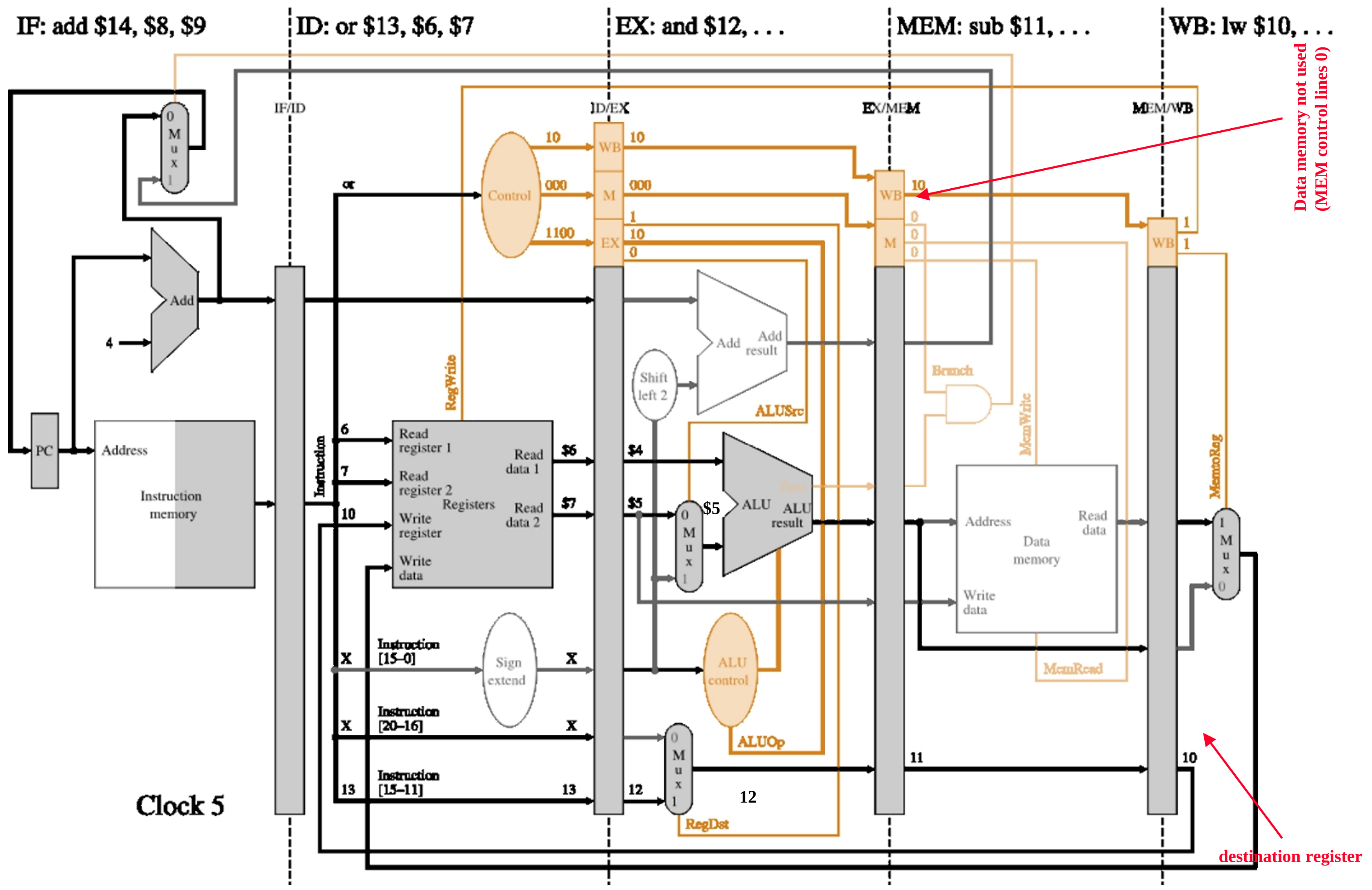
Pipeline Example - before <2> completes



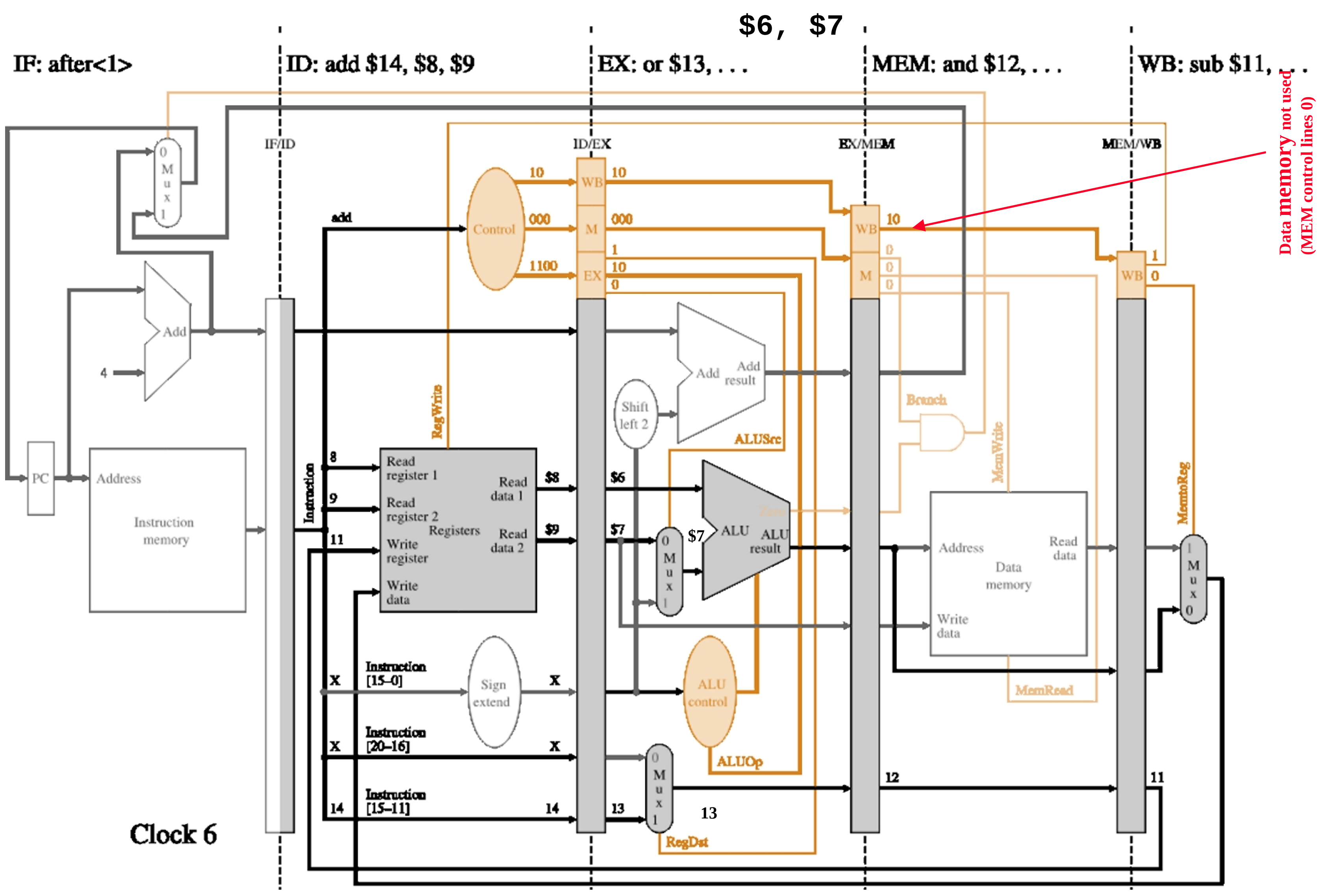
Pipeline Example - before <1> completes



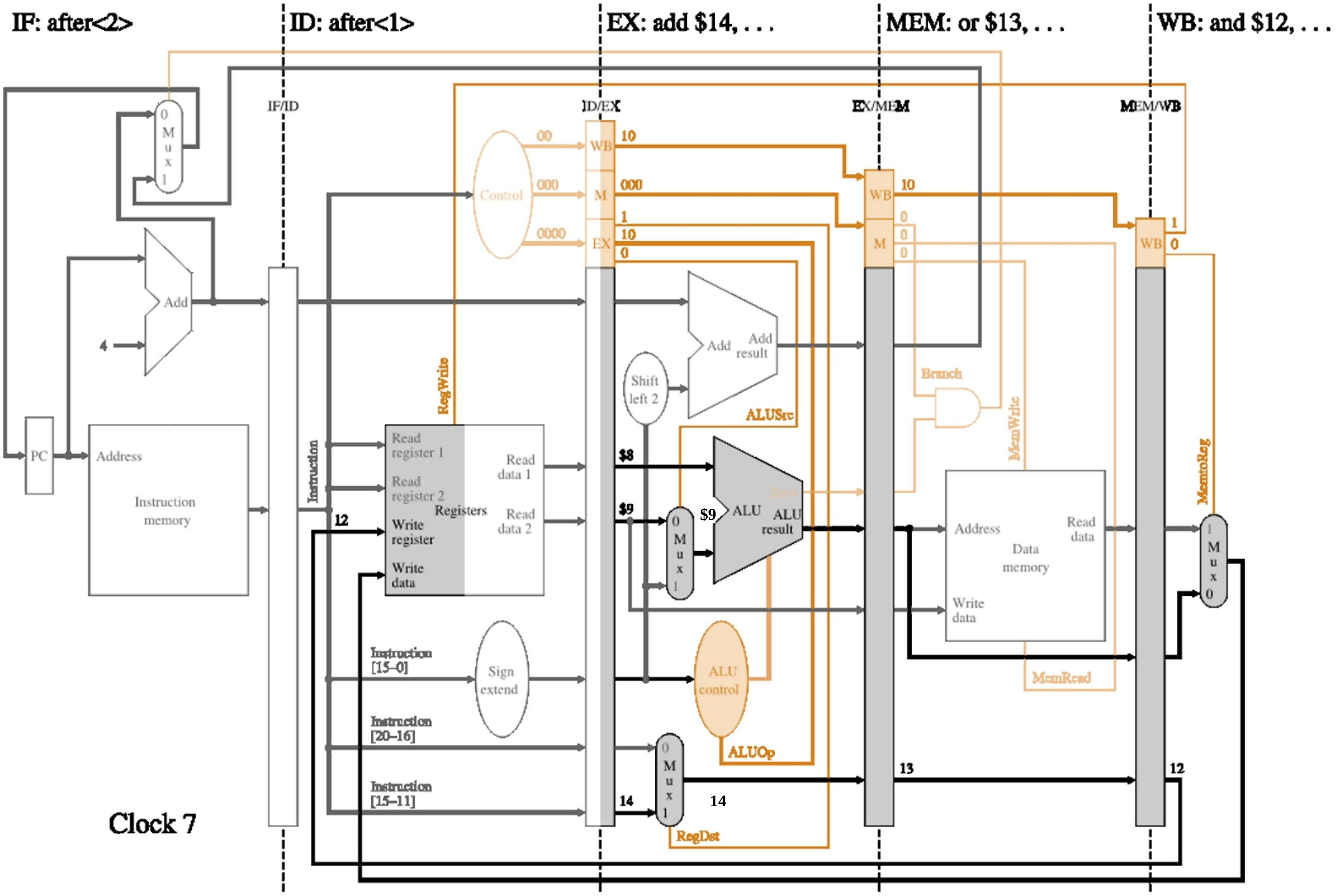
Pipeline Example - lw completes



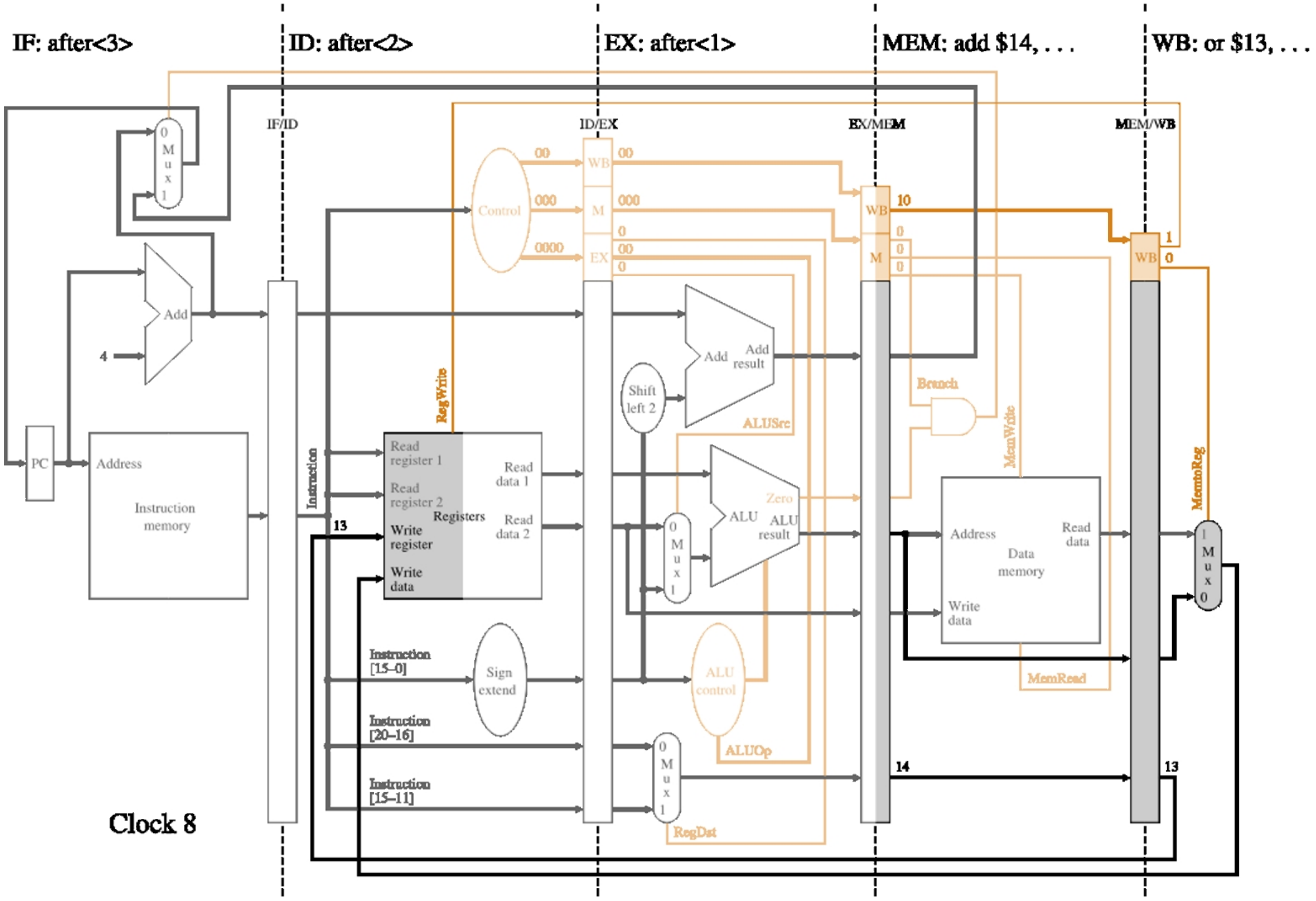
Pipeline Example - sub completes



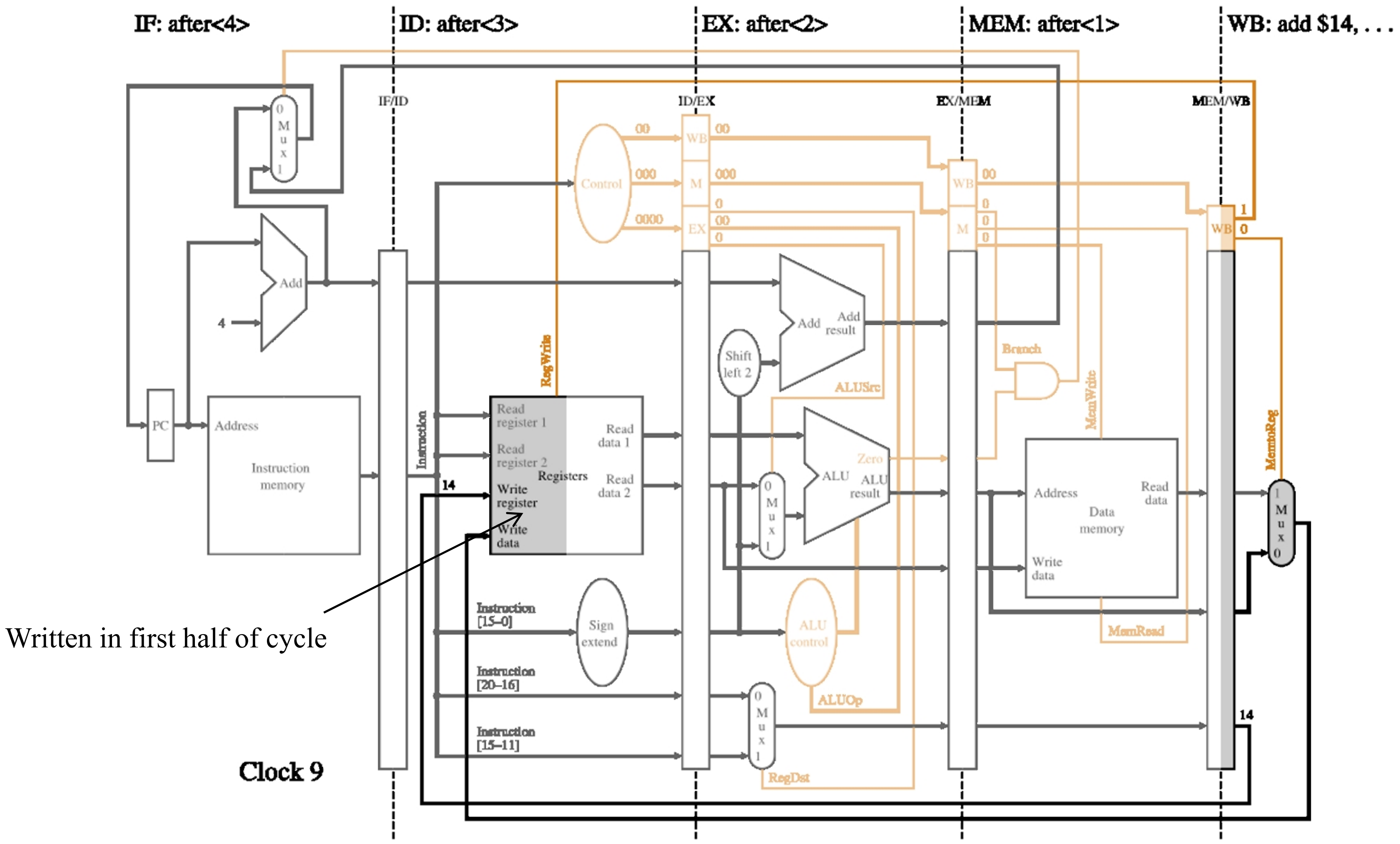
Pipeline Example - and completes



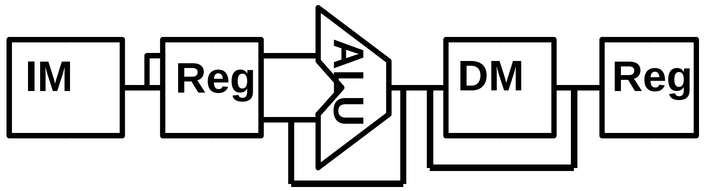
Pipeline Example - or completes



Pipeline Example - add completes



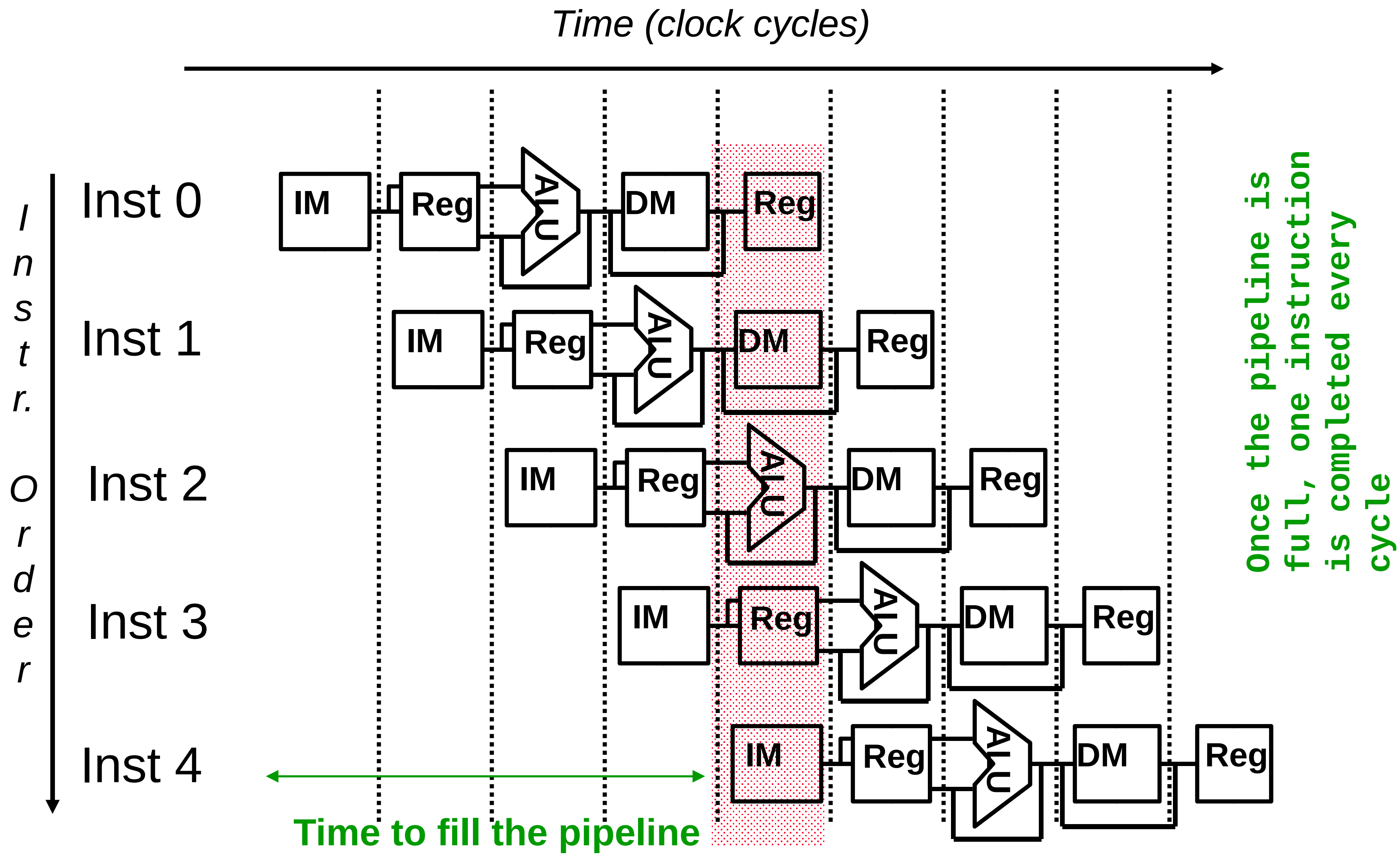
Graphically Representing MIPS Pipeline



- So-far we saw the *single-clock-cycle pipeline diagrams* – show the state of the entire datapath during a clock cycle (instructions are identified above the pipeline stages).

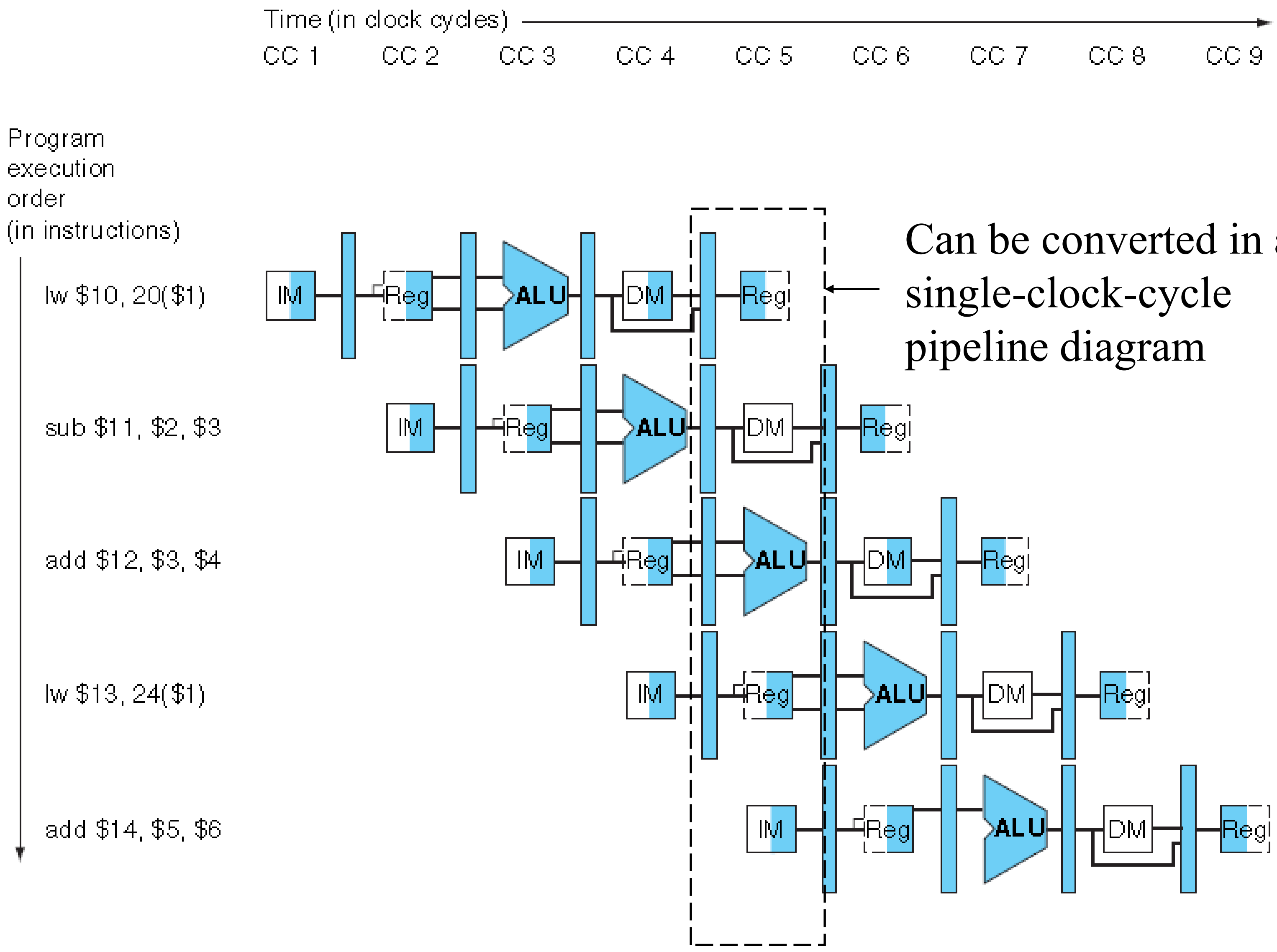
- Can represent multiple instructions in a single figure

Why Pipeline? For Throughput



Time (clock cycles)
Instr. Order
Inst 0
Inst 1
Inst 2
Inst 3
Inst 4
IM
Reg
ALU
DM
Reg
Time to fill the pipeline
Once the pipeline is full, one instruction is completed every cycle

Example of graphical representation

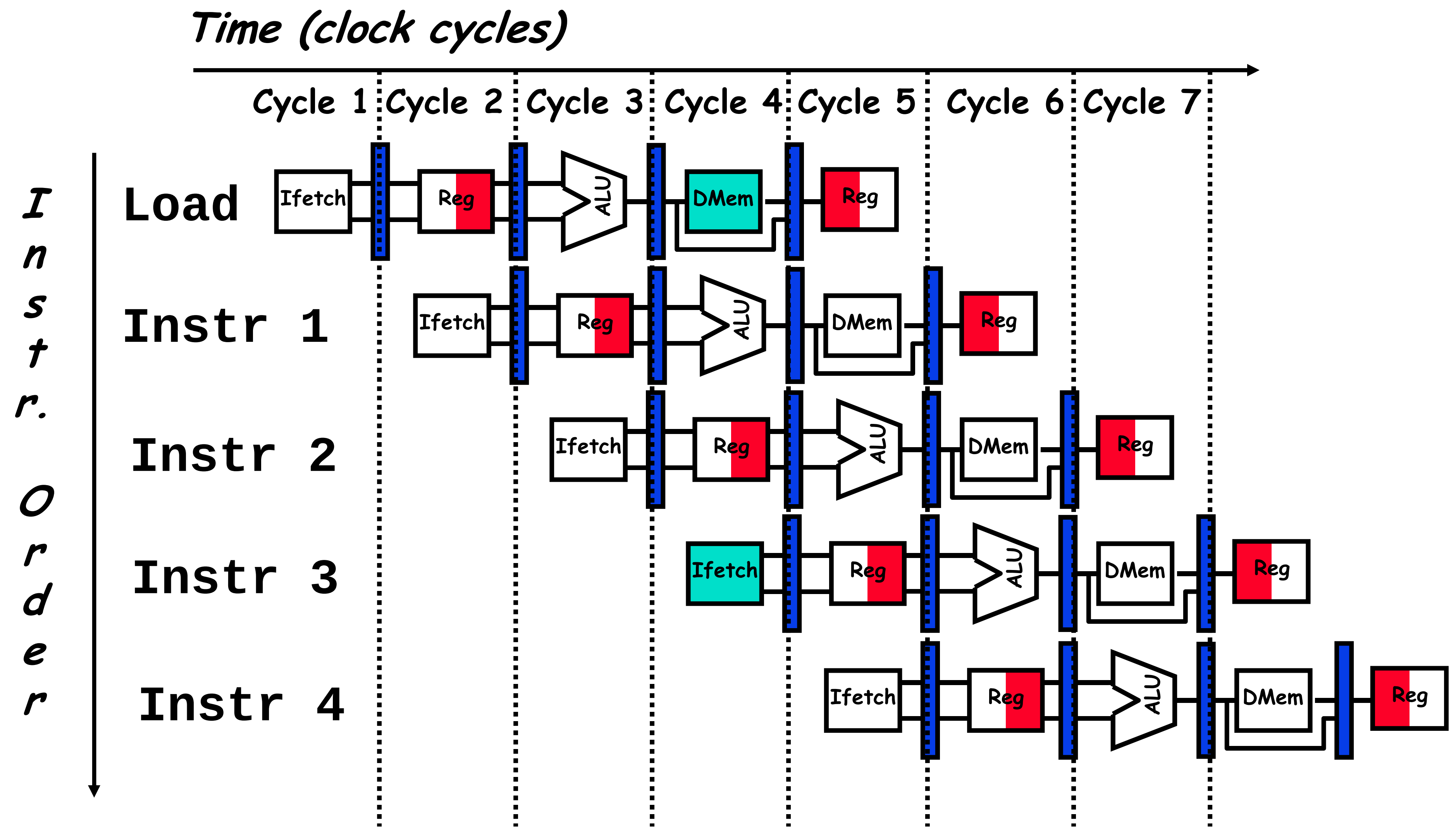


Can be converted in a single-clock-cycle pipeline diagram

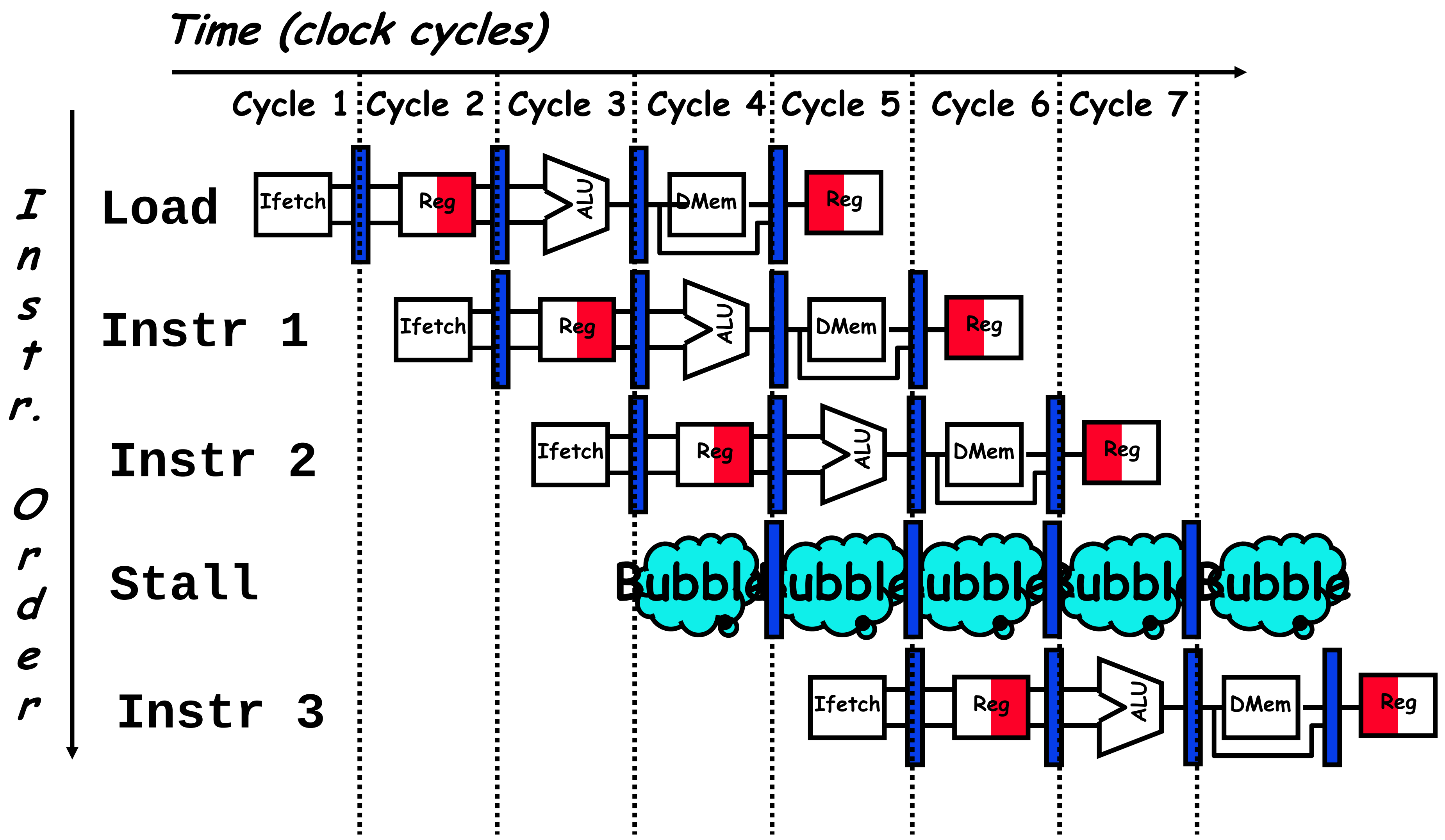
Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

One Memory Port/Structural Hazards

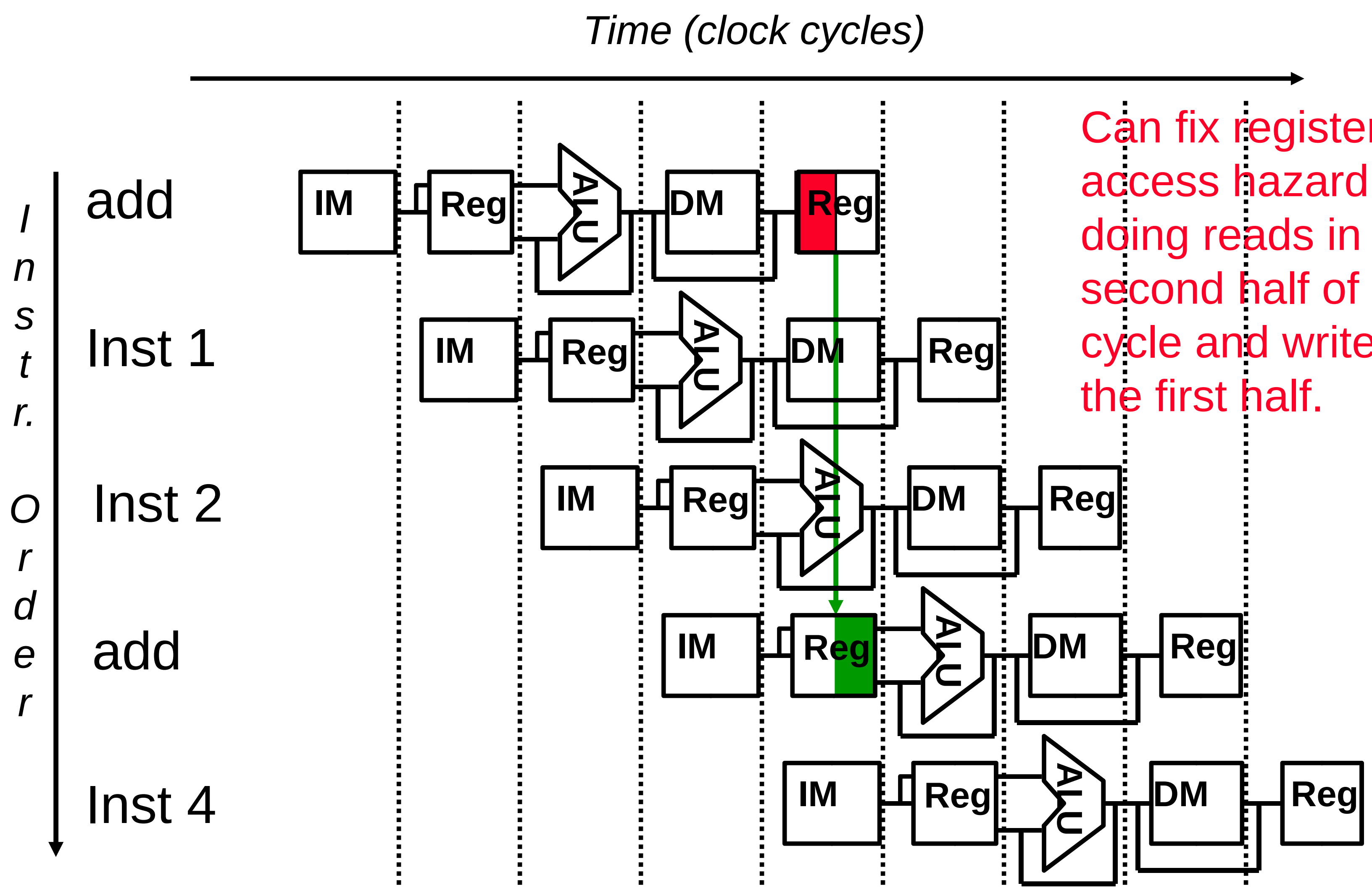


One Memory Port/Structural Hazards



How do you “bubble” the pipe?

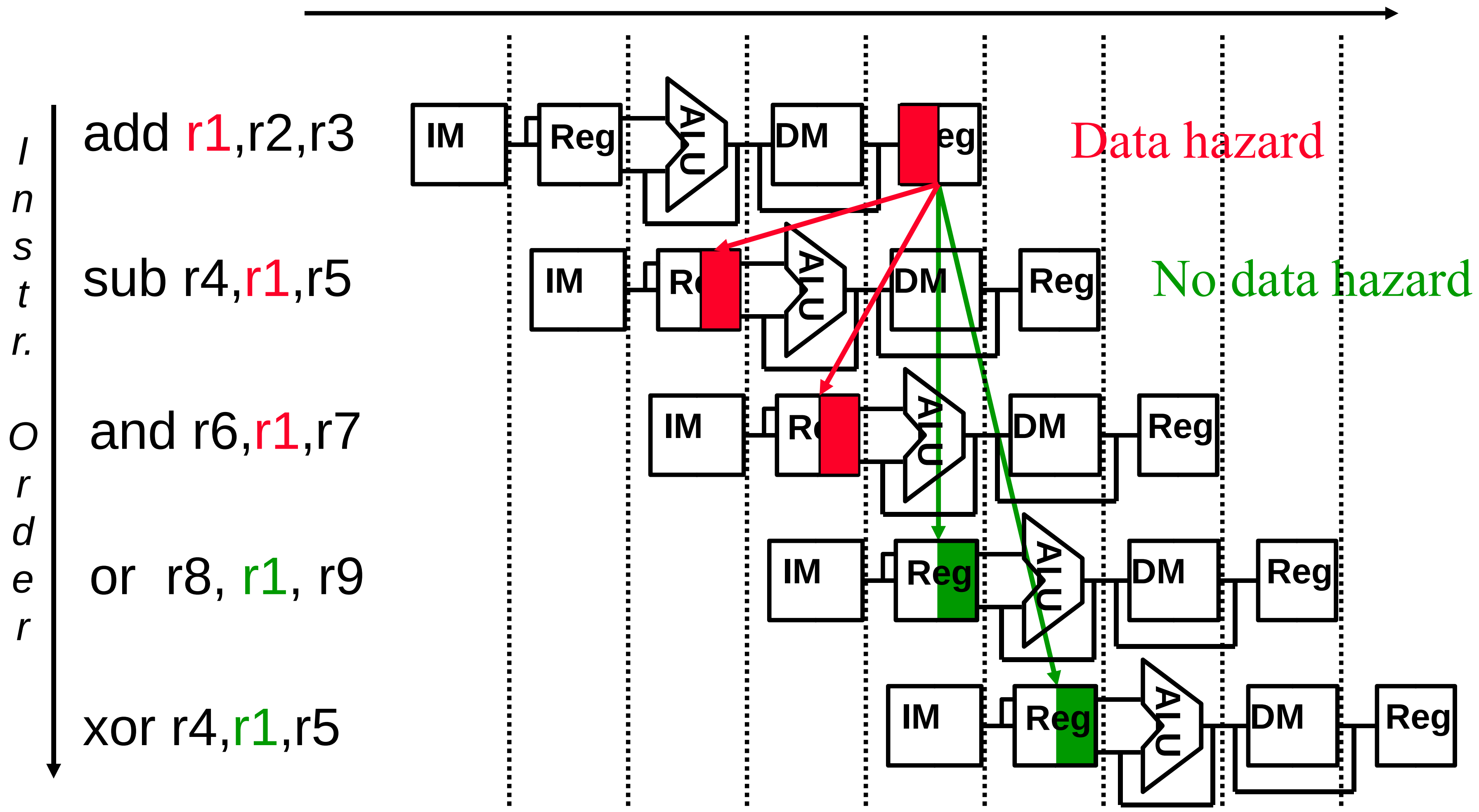
How About Register File Access?



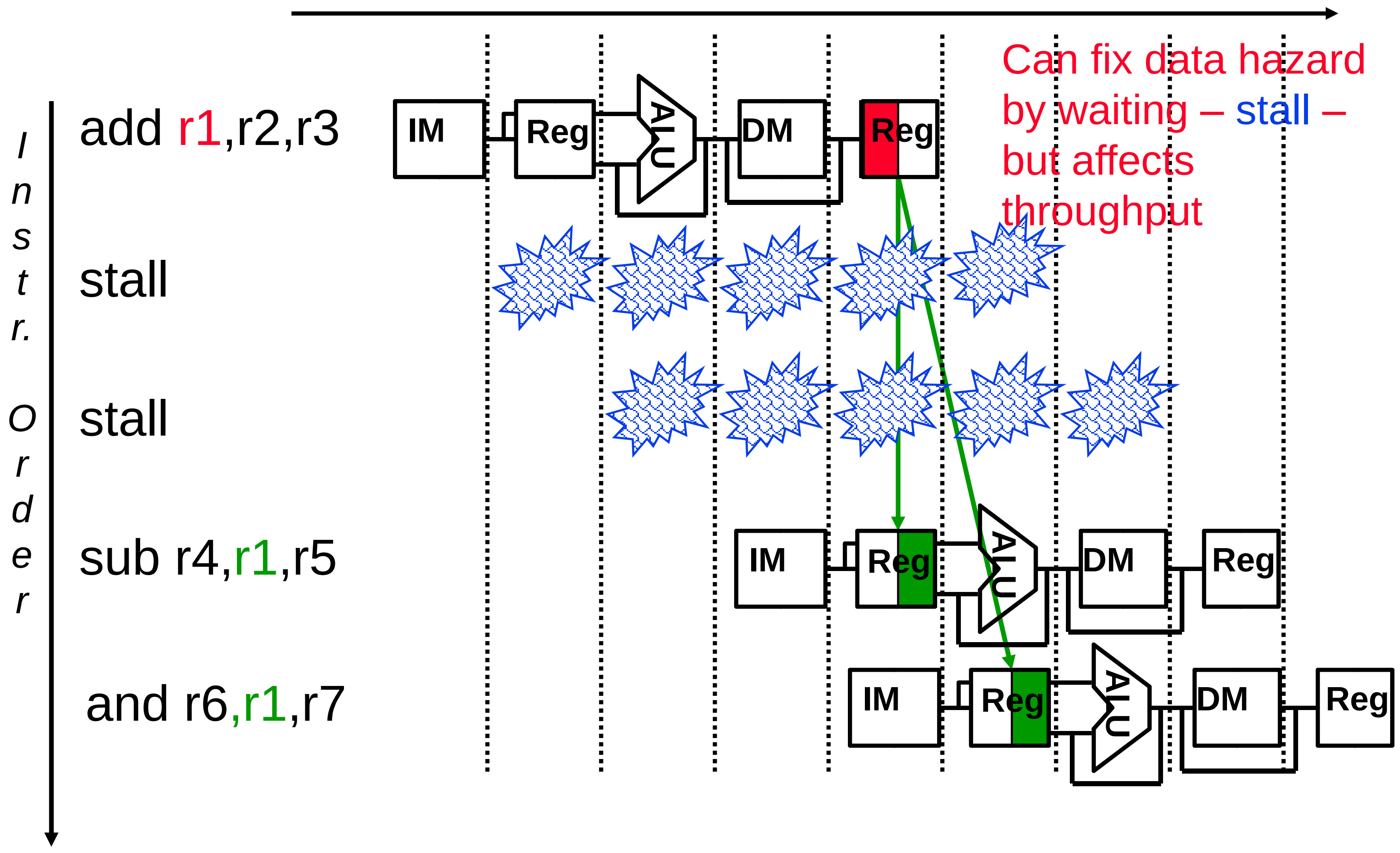
Can fix register file access hazard by doing reads in the second half of the cycle and writes in the first half.

Register Usage Can Cause Data Hazards

- **Dependencies backward in time cause hazards**

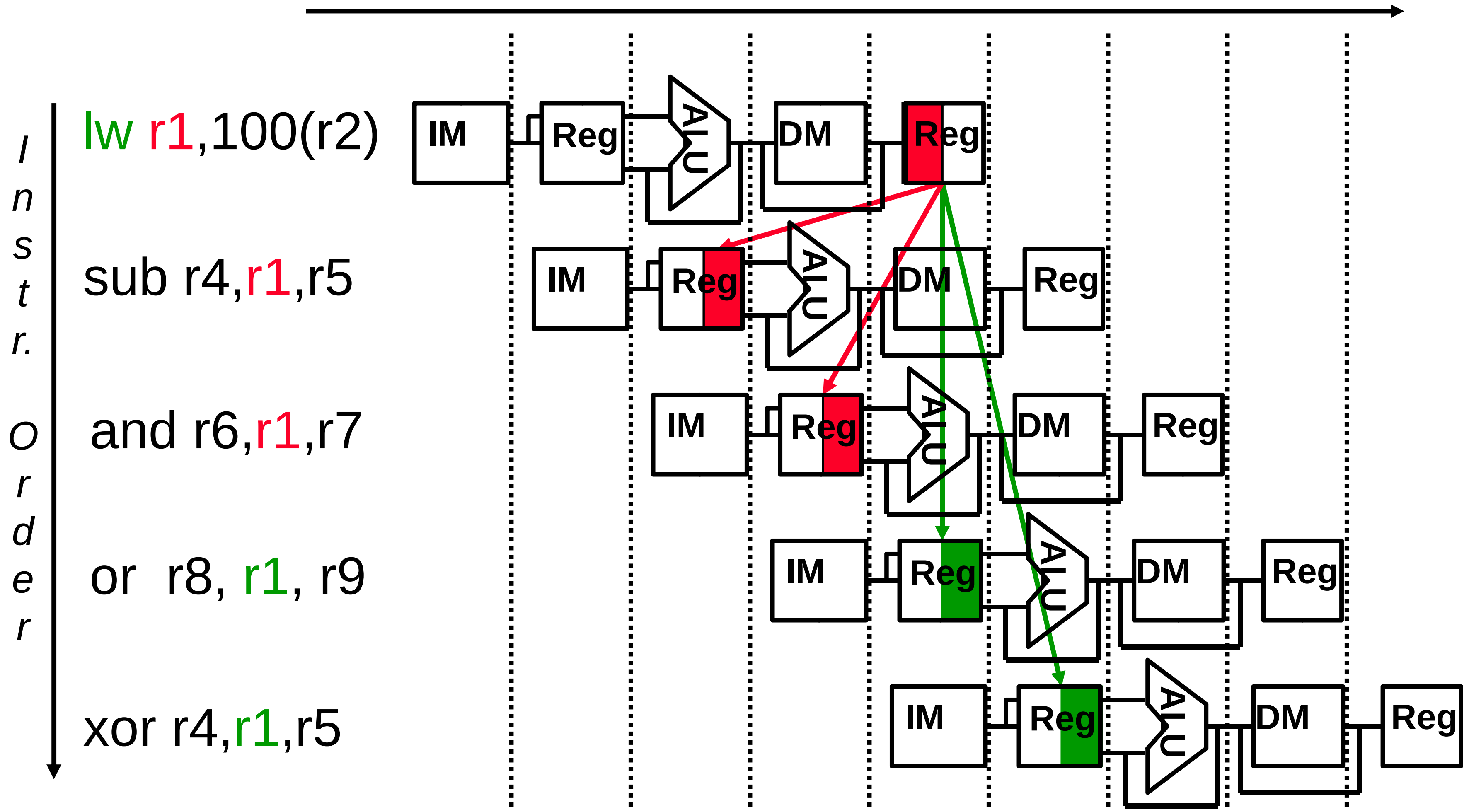


One Way to "Fix" a Data Hazard



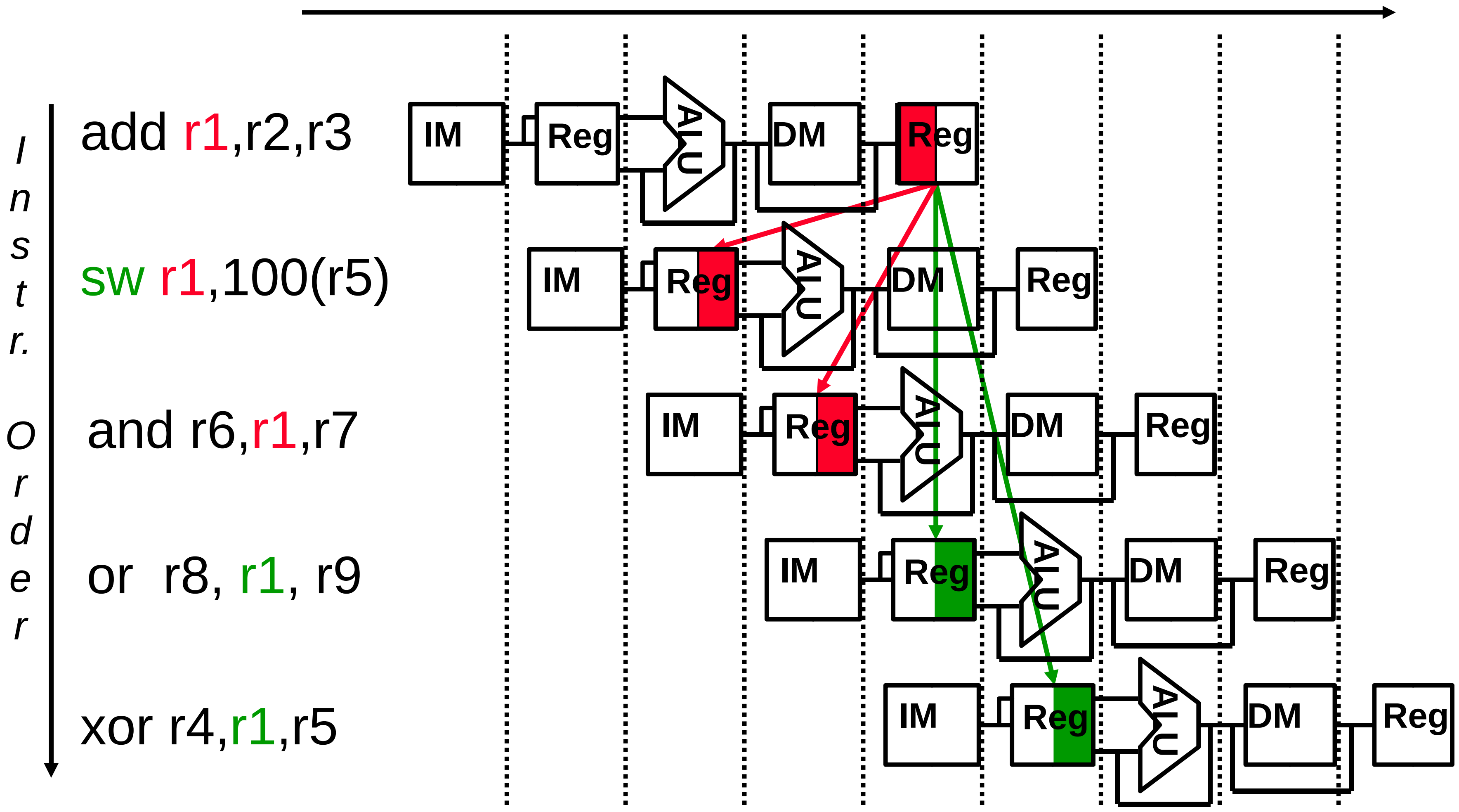
Loads Can Cause Data Hazards

- **Dependencies backward in time cause hazards**



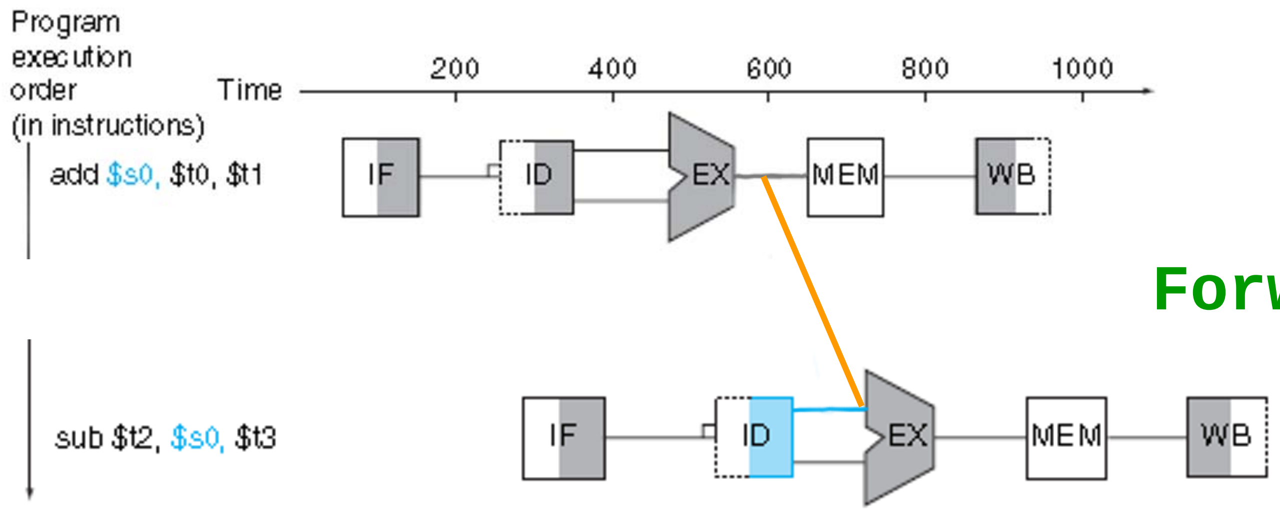
Stores Can Cause Data Hazards

- Dependencies backward in time cause hazards



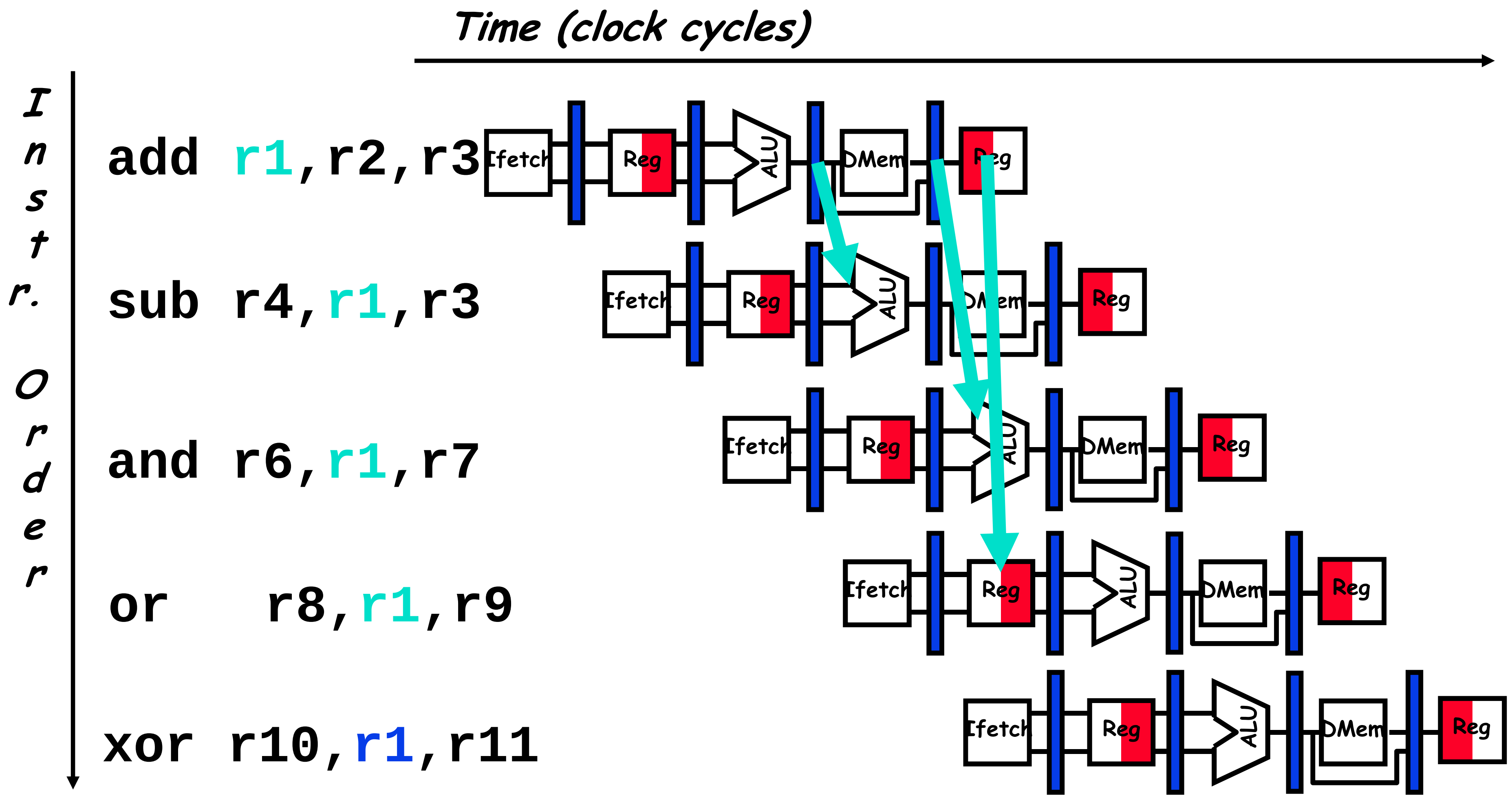
Pipelining the MIPS ISA

- data hazards: what if an instruction's input operands depend on the output of a previous instruction that did not finish? Example an add followed by a sub.



Forwarding

Forwarding to Avoid Data Hazard

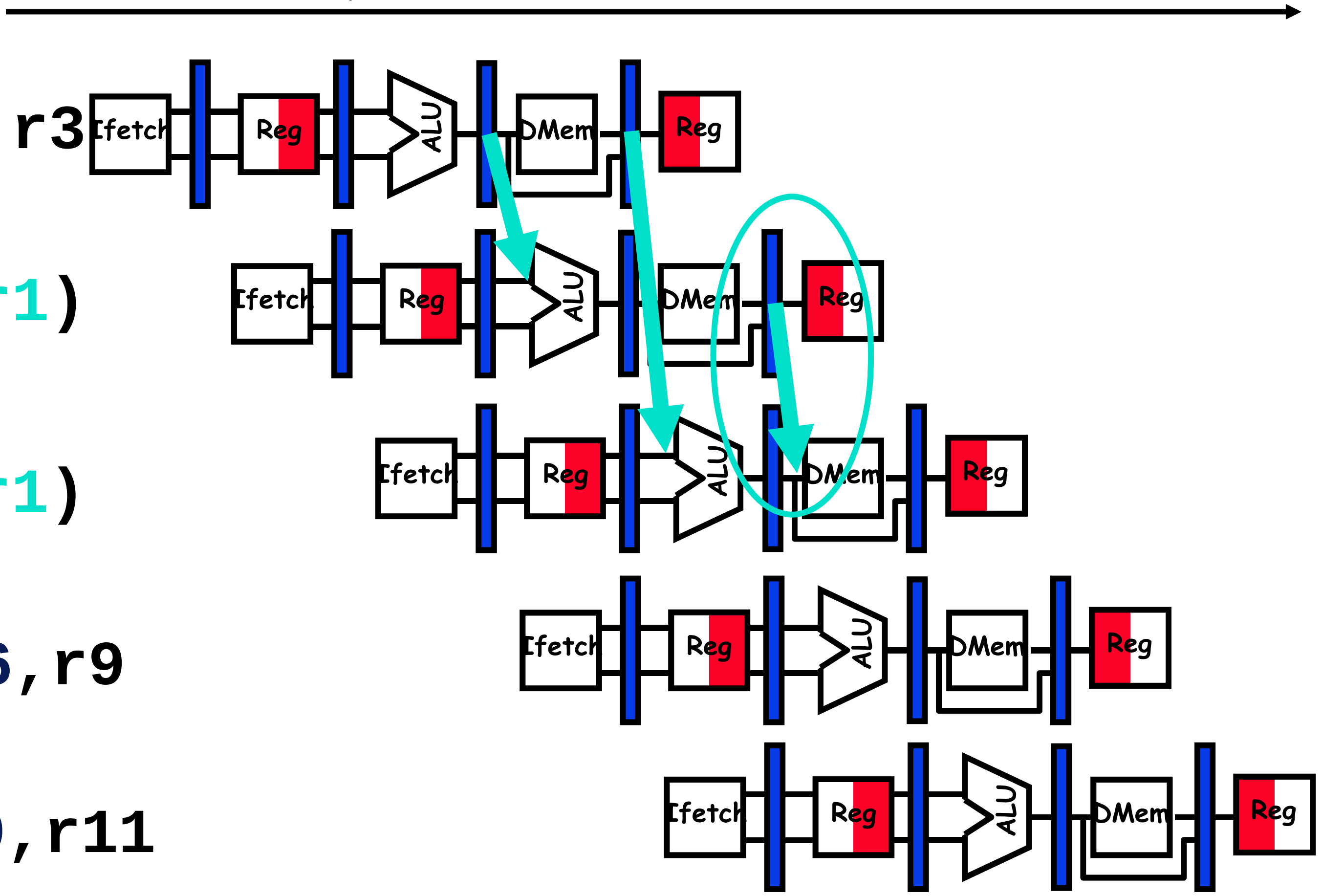


Forwarding to Avoid LW-SW Data Hazard

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
lw  r4, 0(r1)
sw  r4,12(r1)
or    r8,r6,r9
xor r10,r9,r11
```



Data Hazard Even with Forwarding

Time (clock cycles)

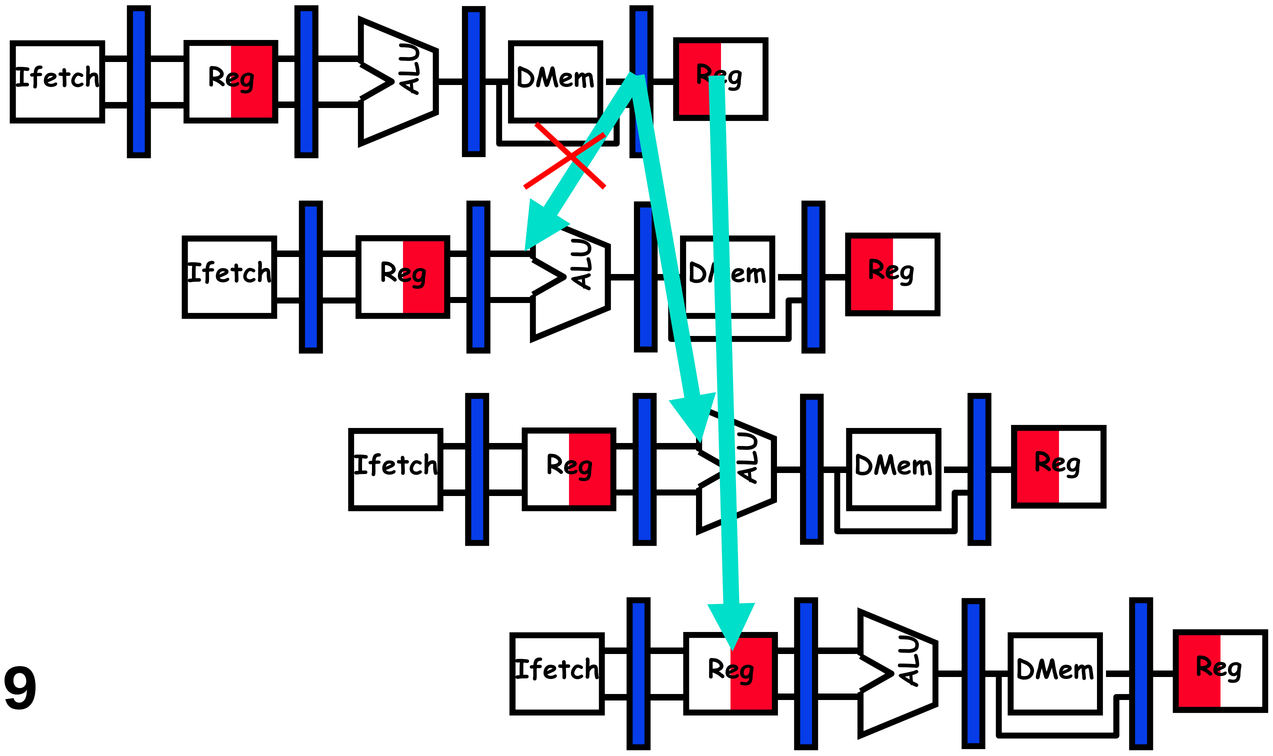
Instr. Order

lw r1, 0(r2)

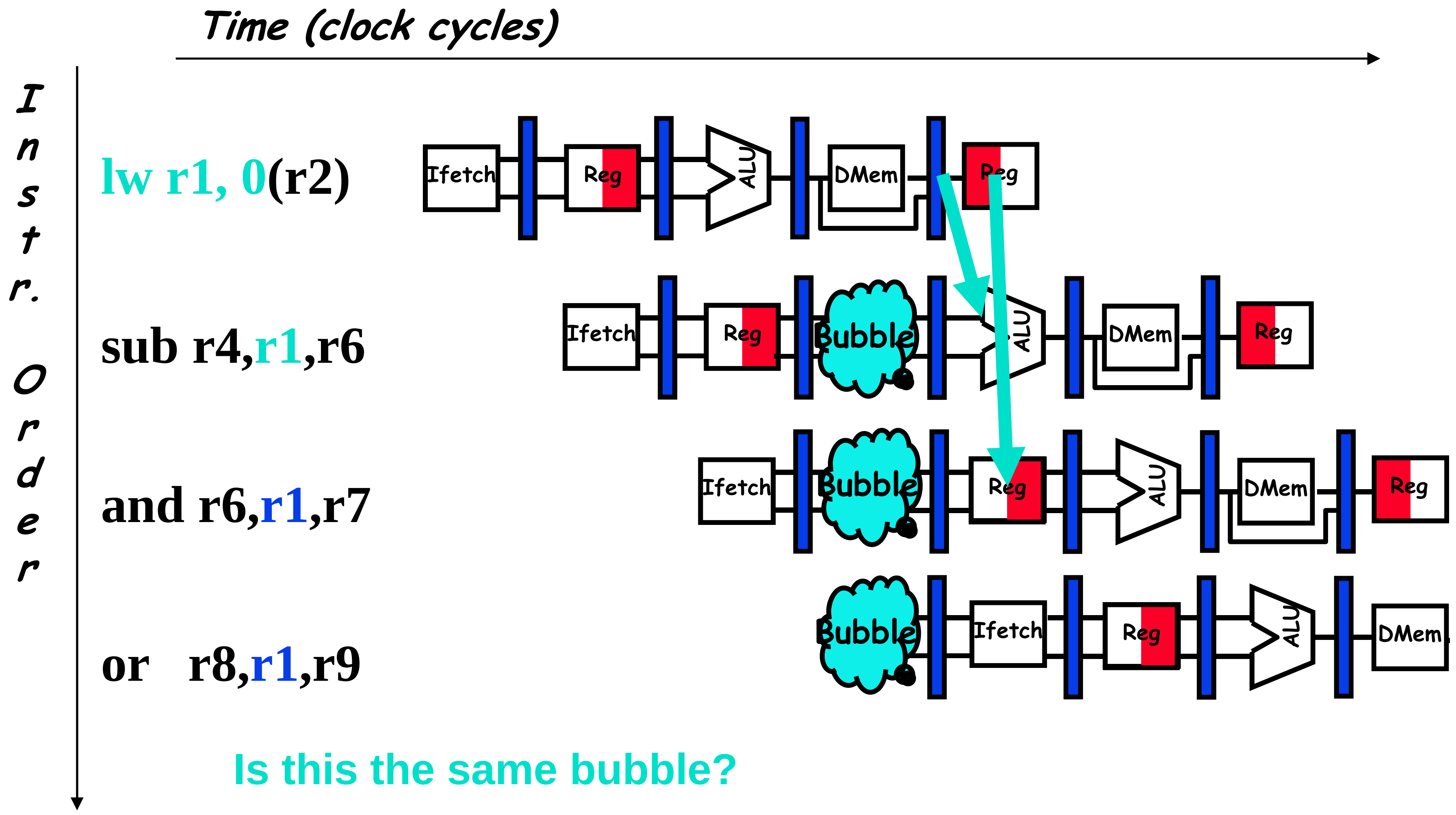
sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9



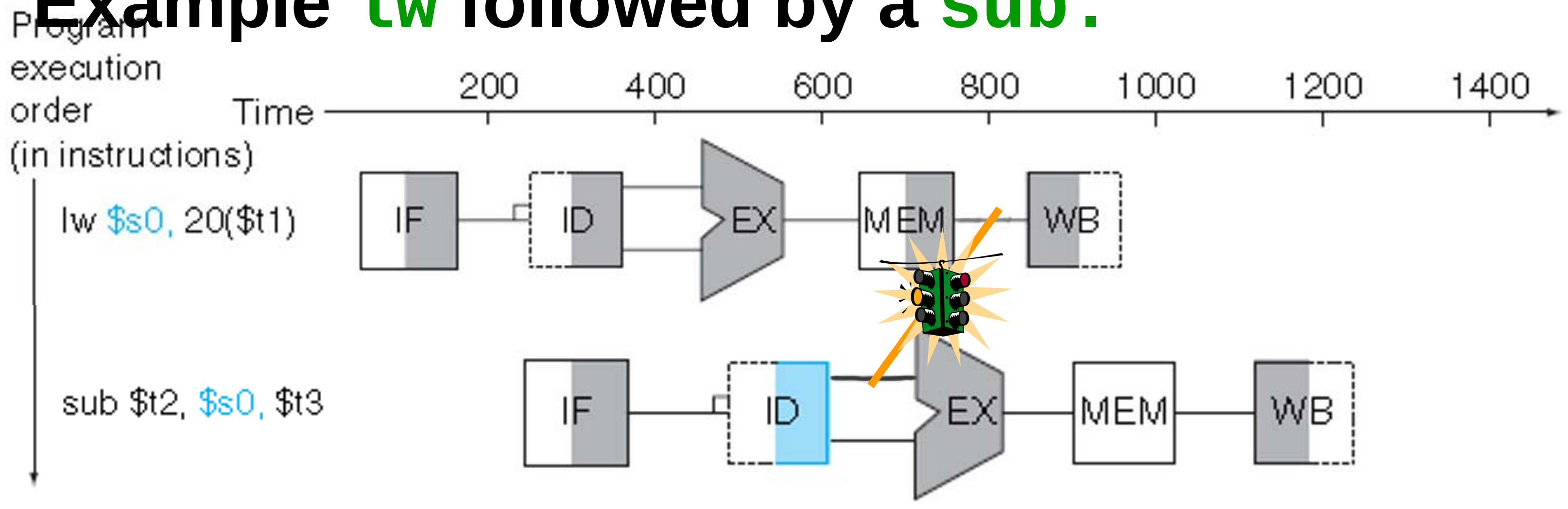
Data Hazard Even with Forwarding



Is this the same bubble?

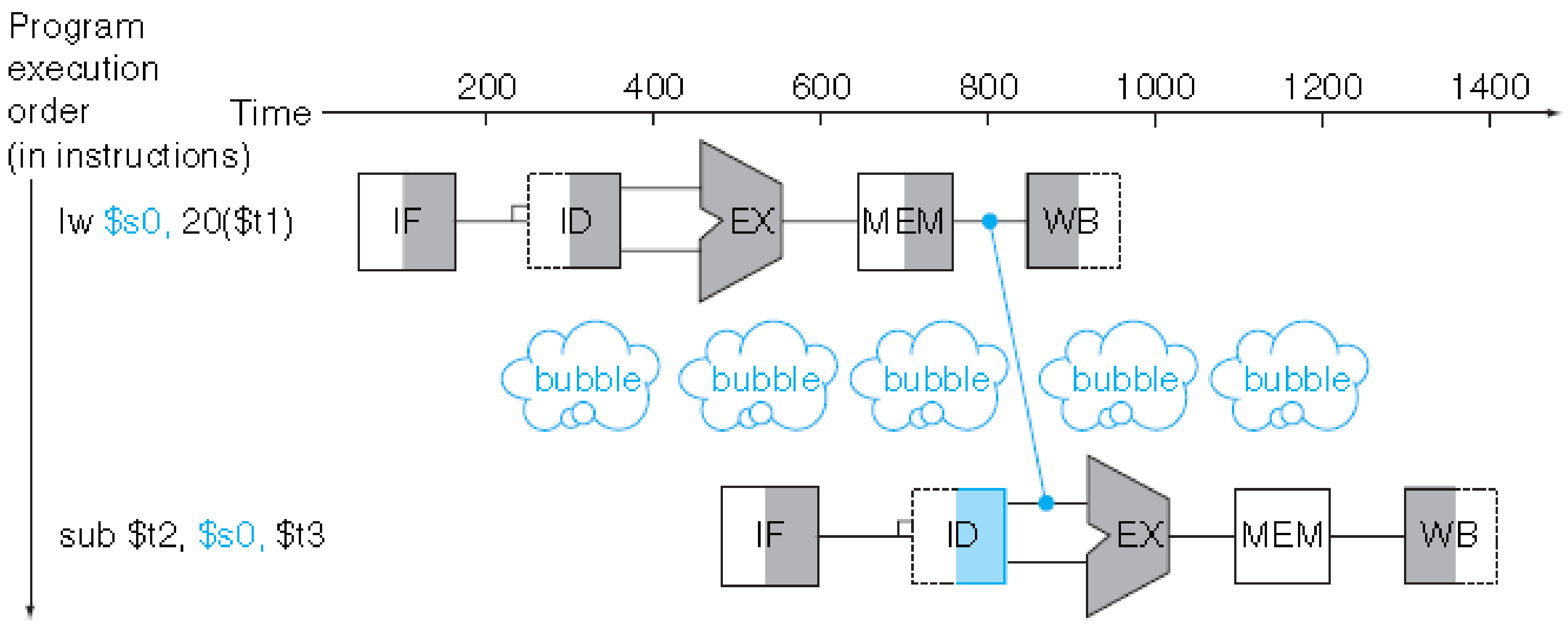
Pipelining the MIPS ISA

- `Forwarding` **will *fail* for a `lw` followed immediately by an instruction that uses the results of the `lw` operation.**
- **Example `lw` followed by a `sub.`**



Pipelining the MIPS ISA

- Solution - stall pipeline one clock cycle, *then* forward
    - **Another solution - optimize compiler, such that `lw` is followed by an instruction which does not depend on the loaded word.**



Pipeline Changes to accommodate Forwarding

- To avoid slowing down throughput, we need to add a hardware that *detects* data hazards.
- We call this the forwarding unit.
- Data needs to be forwarded to the ALU when a data hazard is detected. Thus the forwarding unit controls forwarding data through additional multiplexing at the ALU input.
- This logic unit needs input from the three pipeline registers.
- It also needs to detect if the RegWrite control signal is asserted – so it needs input from the control lines also.
- No forwarding if EX/MEM.RegisterRd=$0 and MEM/WB.RegisterRd=$0

Pipeline Changes to accommodate Forwarding

- It needs to detect one of *four cases* of data hazards:

  if (EX/MEM.RegWrite

  and (EX/MEM.RegisterRd $\neq$ 0

  and (EX/MEM.RegisterRd=ID/EX.RegisterRs) Forward

- similarly

  if (EX/MEM.RegWrite

  and (EX/MEM.RegisterRd $\neq$ 0

  and (EX/MEM.RegisterRd=ID/EX.RegisterRt) Forward

Pipeline Changes to accommodate Forwarding

- similarly

  if (MEM/WB.RegWrite

  and (MEM/WB.RegisterRd $\neq$ 0

  and (MEM/WB.RegisterRd=ID/EX.RegisterRs)
  Forward

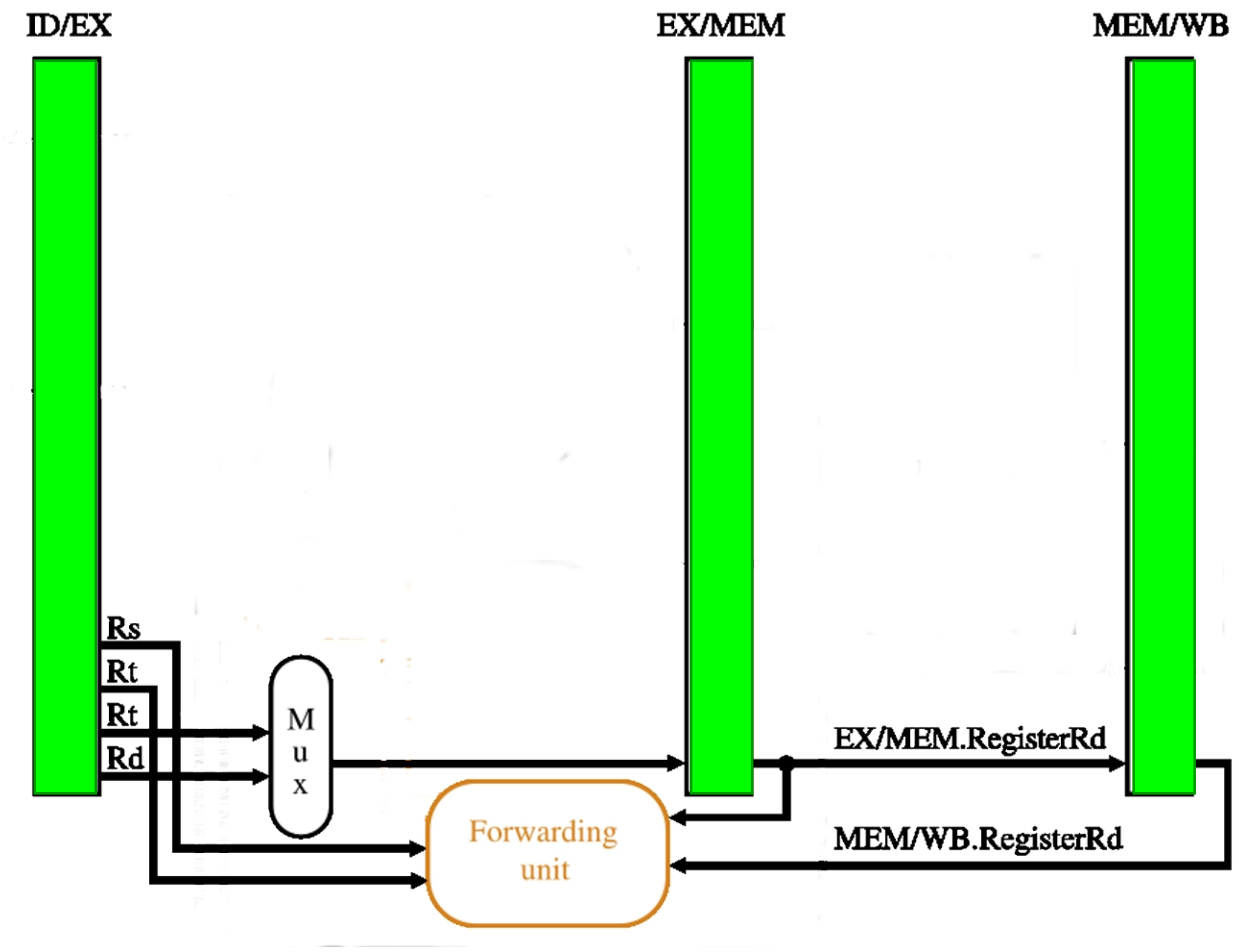
- similarly

  if (MEM/WB.RegWrite

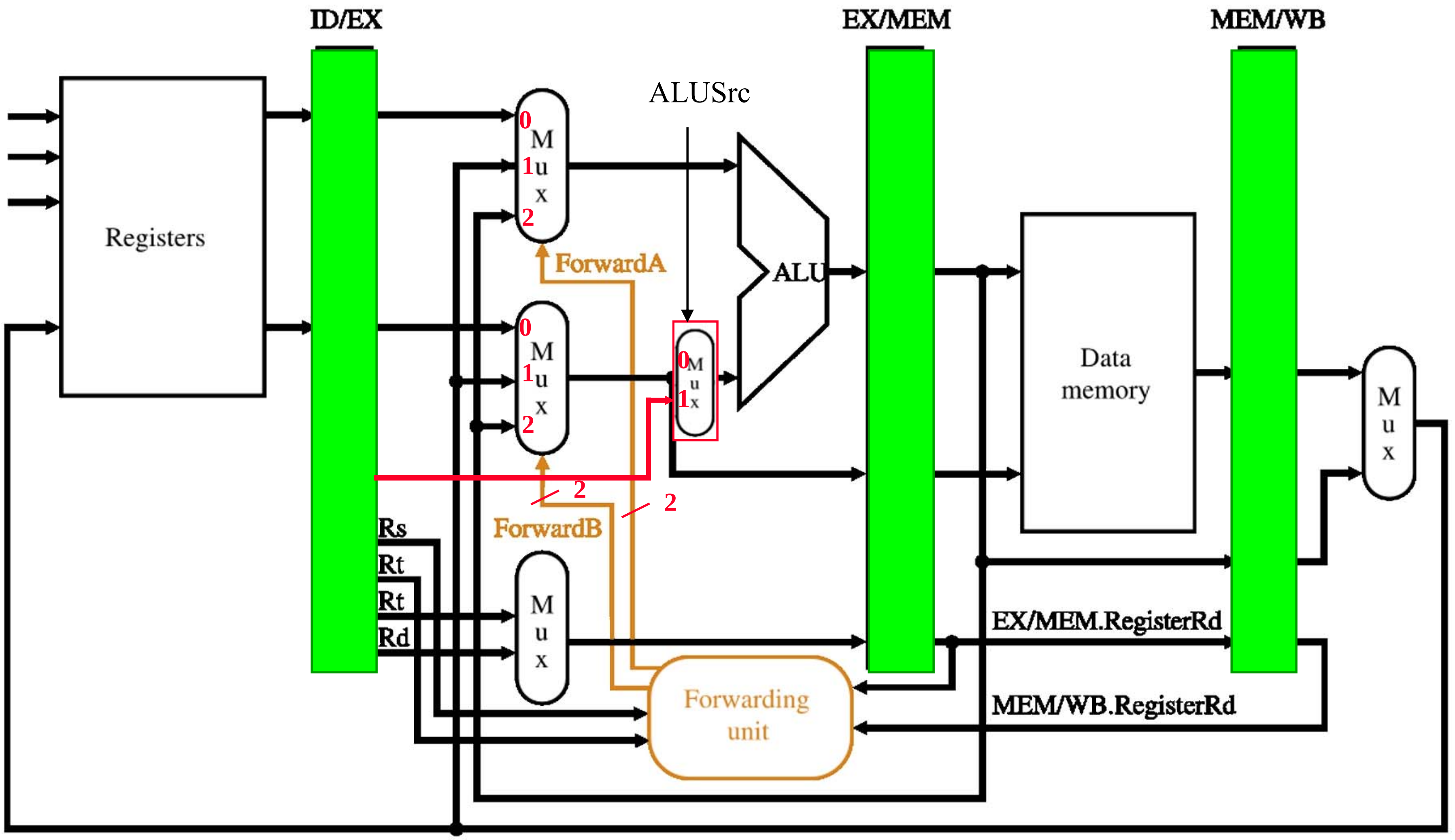
  and (MEM/WB.RegisterRd $\neq$ 0

  and (MEM/WB.RegisterRd=ID/EX.RegisterRt)
  Forward

Pipeline Changes to accommodate Forwarding



Pipeline Changes to accommodate Forwarding



ForwardA 00 ID/EX input to ALU1 - no fwd
01 MEM/WB input to ALU1
10 EX/MEM input to ALU1

ForwardB 00 ID/EX input to ALU2
01 MEM/WB input to ALU2
10 EX/MEM input to ALU2

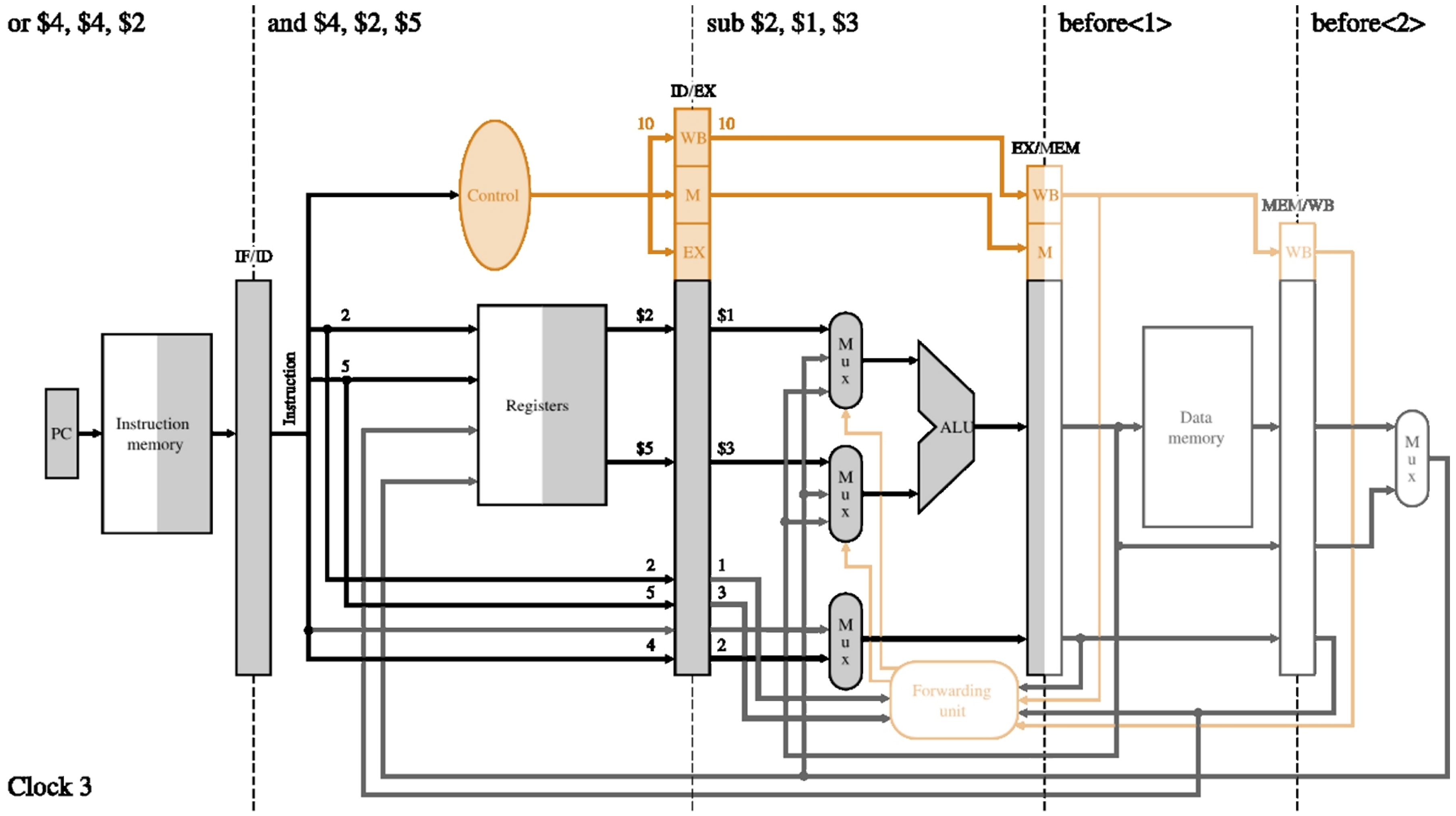
ForwardB 11 sign extension input to ALU2
OR add another multiplexer

Forwarding Pipeline Example

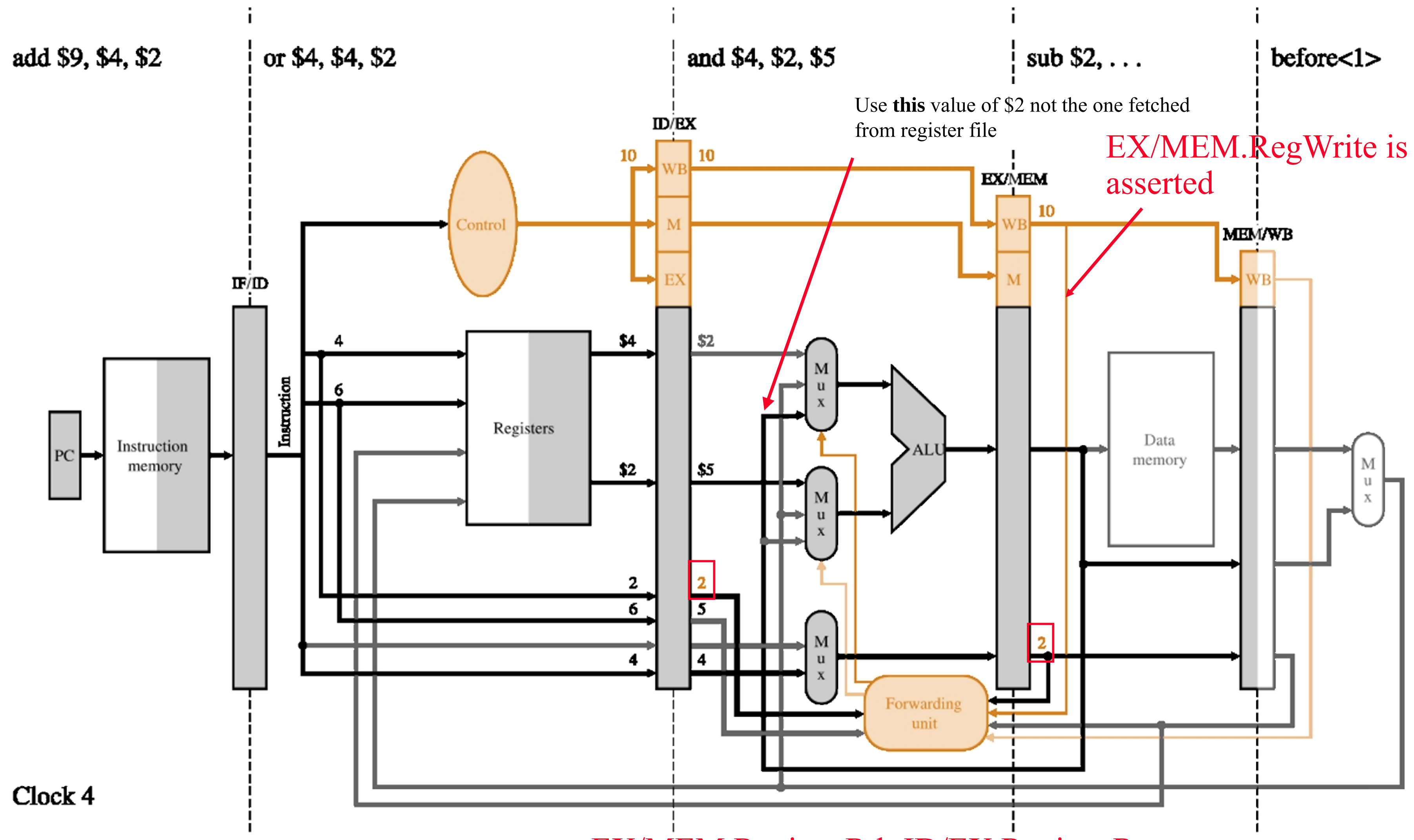
- How does the dependent instruction sequence execute in a pipeline with support for forwarding?

```
before <4>
before <3>
before <2>
before <1>
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
after <1>
after <2>
…
```

Forw. Pipeline Example - before <2> completes

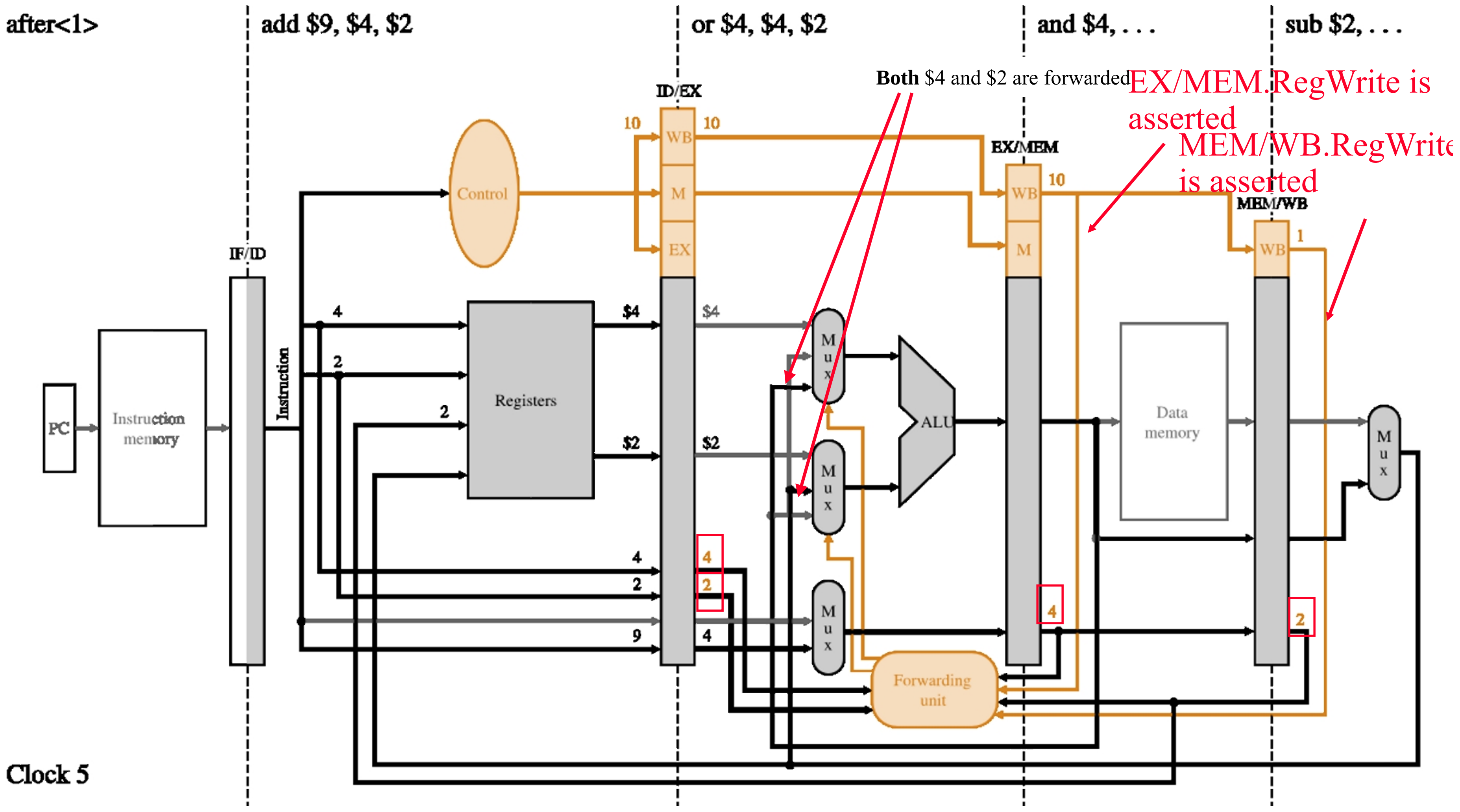


Forw. Pipeline Example - before <1> completes



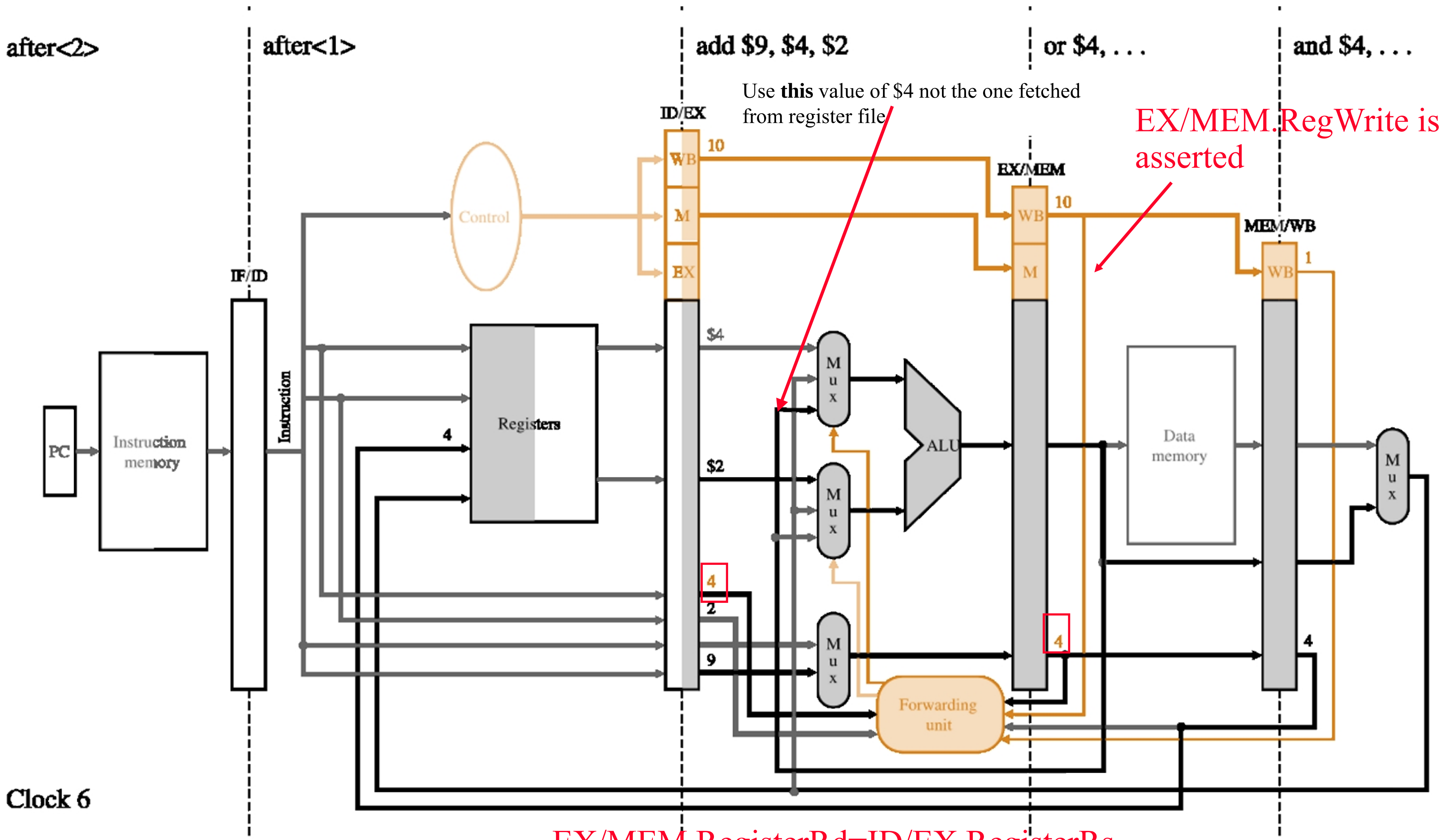
EX/MEM.RegisterRd=ID/EX.RegisterRs

Forw. Pipeline Example - sub completes



EX/MEM.RegisterRd=ID/EX.RegisterRs MEM/WB.RegisterRd=ID/EX.RegisterRt

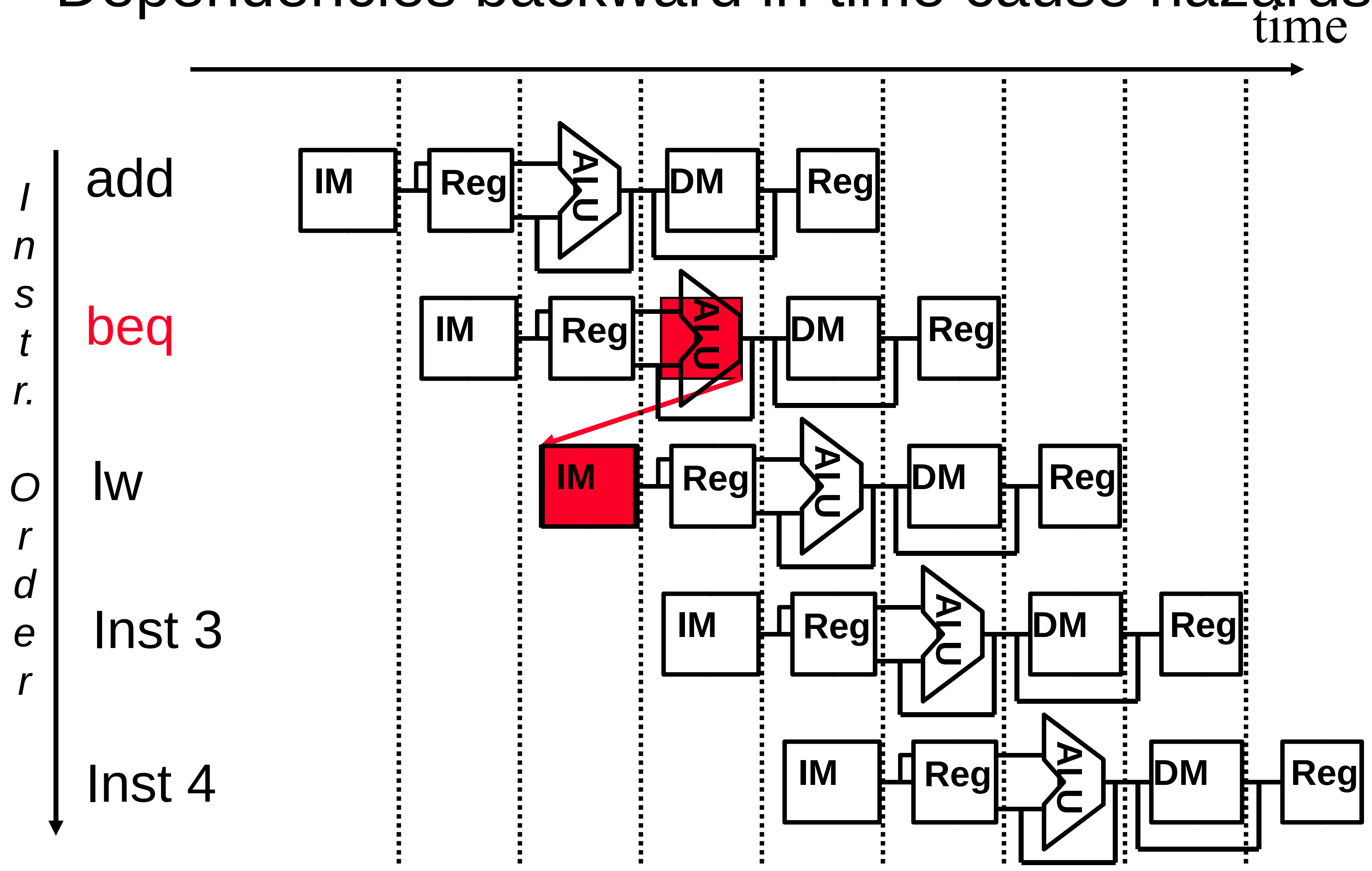
Forwarding Pipeline Example - and completes



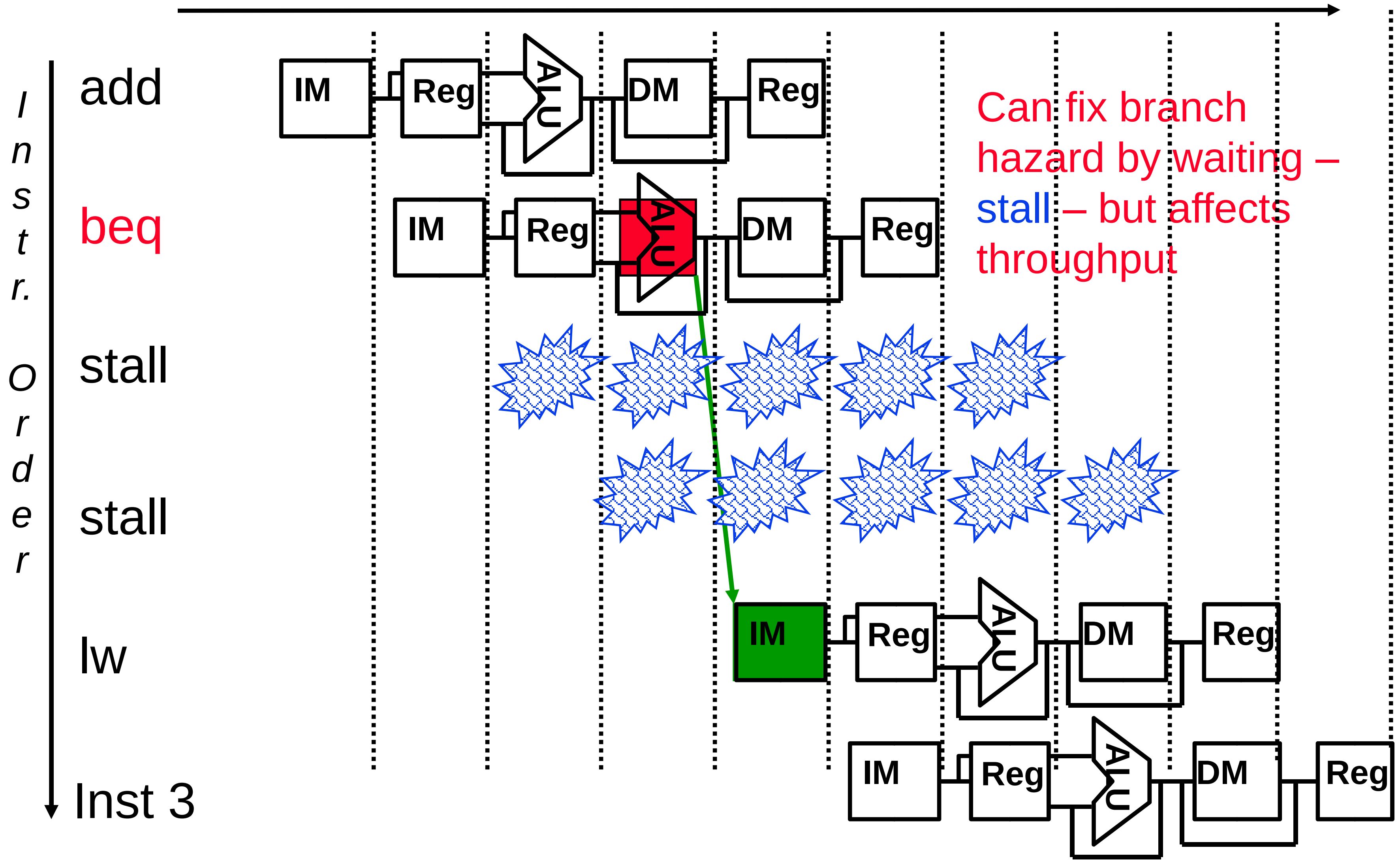
EX/MEM.RegisterRd=ID/EX.RegisterRs

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



One Way to “Fix” a Control Hazard



Can fix branch hazard by waiting – stall – but affects throughput

Impact of branch stalling

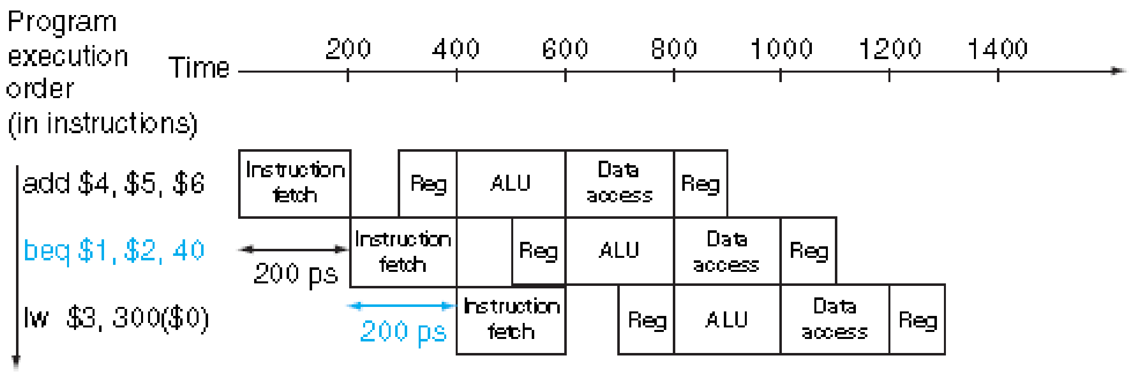
- We assume that all instructions in the pipeline have a CPI of 1. Branches which are followed by a stall have a CPI of 3.
- In a typical program branches occur 13% of the time. Thus we can compute the aggregate CPI of the always-stall for branch architecture as:

$$\text{Then CPI} = \sum_{i=1}^{n} \text{CPI}_{i} \times \text{F}_{i}$$

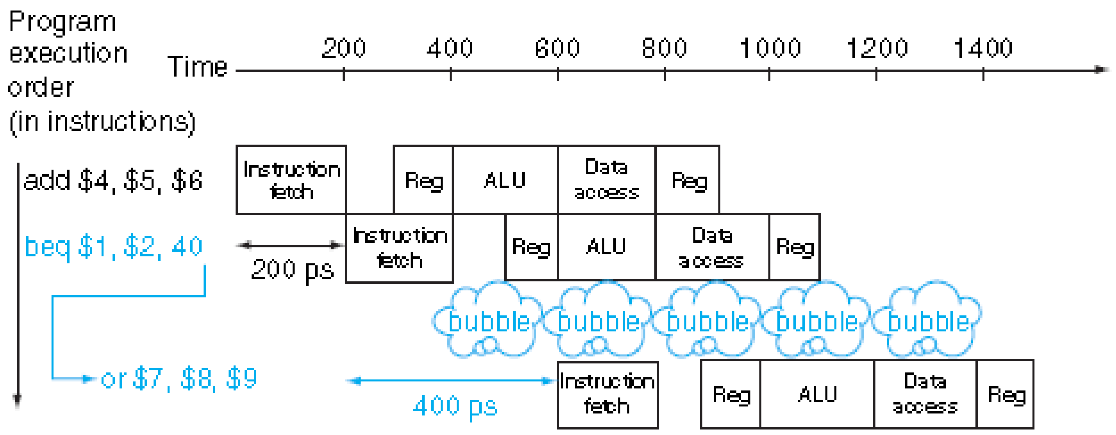
- $\text{CPI}_{\textbf{always stall}} = 1 \times 87\% + 3 \times 13\% = 1.26$ cycles/instruction
-

Pipelining the MIPS ISA

- control hazards: Another approach is "prediction" - either *static* - always execute the instruction following a branch (assume always that the branch is not taken), or predict *dynamically* (keep a history of each branch as taken or not taken - accurate 90% of time).



Branch not taken



Branch taken

Dynamic branch prediction

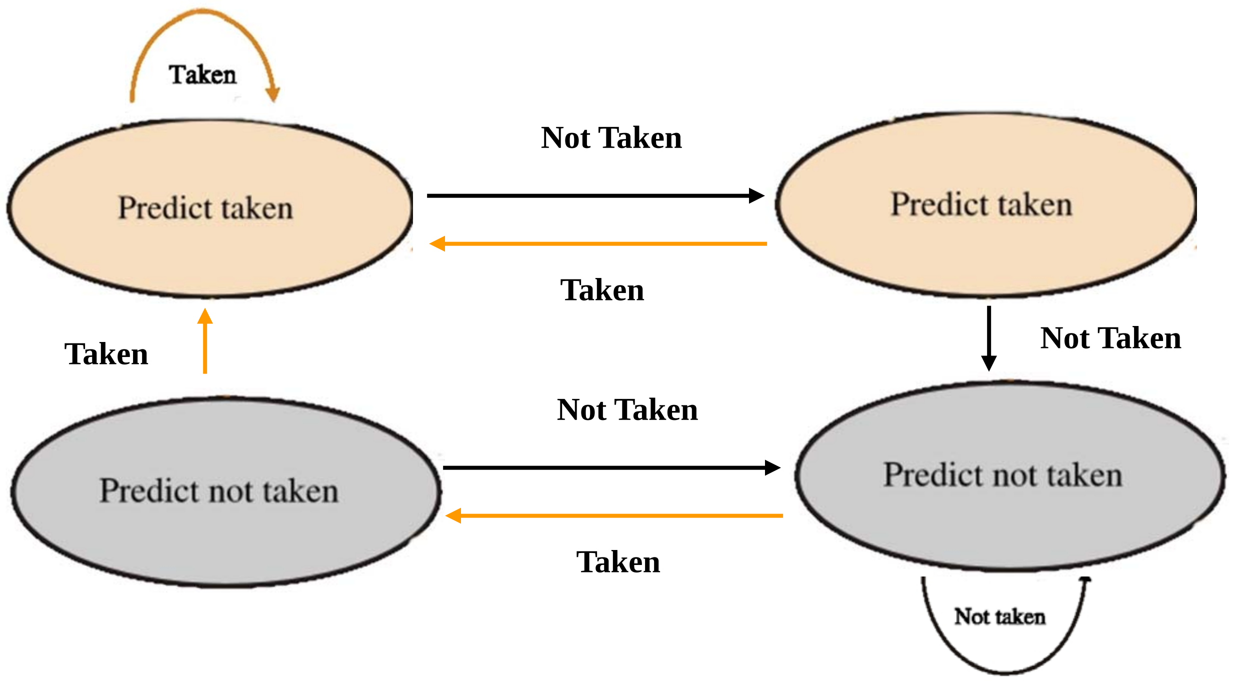
- The static branch “predicts” that it will not be taken and then flush if it was taken works for simple pipelines, but is wasteful for performance for aggressive pipelining architecture (such as the multiple issue of Pentium IV).

- One approach is to have a *branch prediction buffer* (a small memory unit indexed by the lower portion of the address in the branch instruction). It contains a bit that says if the branch was recently taken or not.

- The value of the prediction bit is inverted if the prediction turned out to be wrong. When the branch is almost always taken, this 1-bit predictor will predict wrong twice (at the start and end of the run of branches).

Dynamic branch prediction

- A better approach is to use a two-bit scheme, which must be wrong *twice* to change the direction of prediction.
- The branch prediction is stored in a special buffer which is accessed with the beq instruction in the IF stage. If the beq is predicted as taken, then fetching begins from the target once beq is in ID.



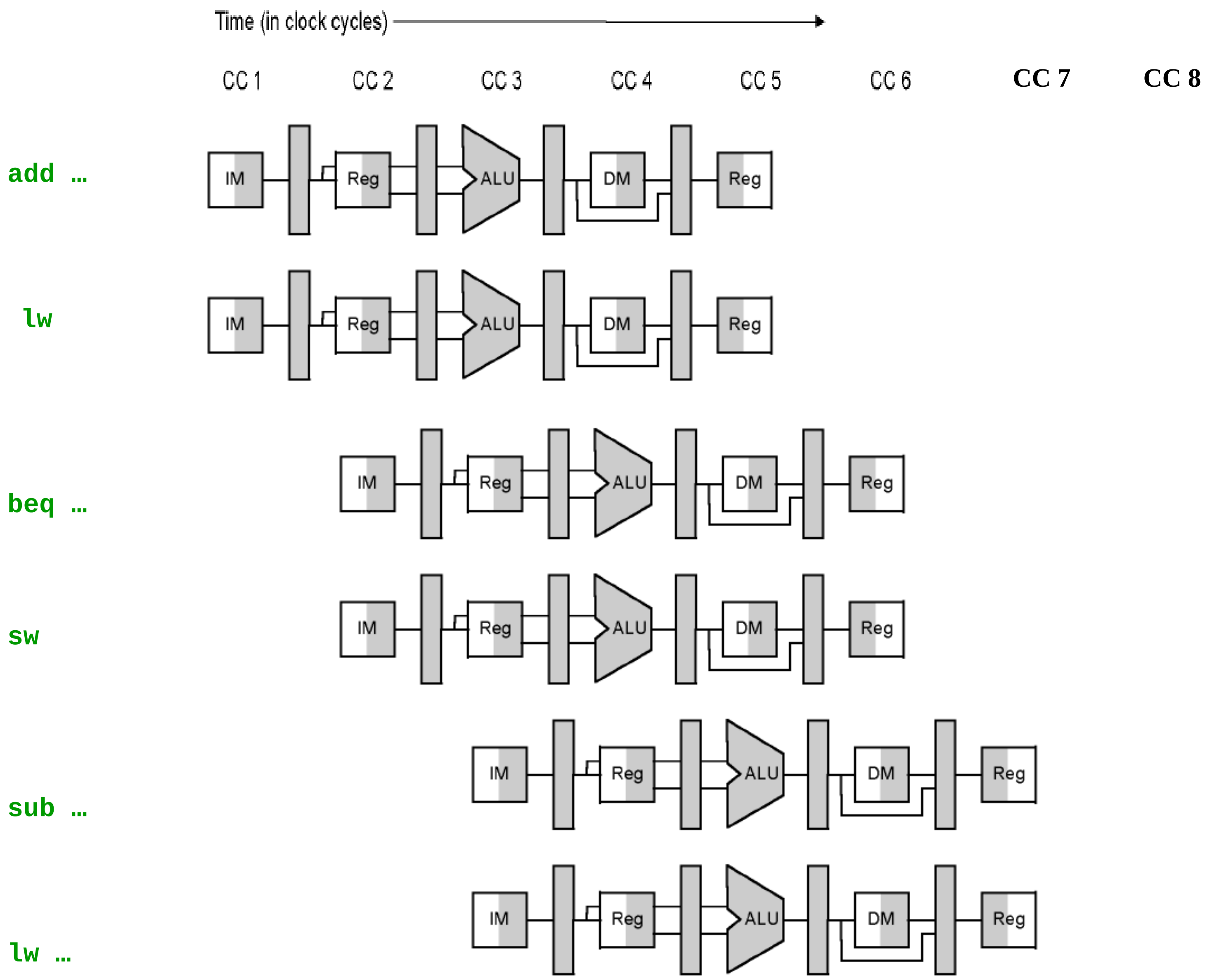
Pipelining Speed-ups

- One way to speed up pipelines is to have more *stages* (up to eight) – results in shorter clock cycles.
- Another way is *superscalar architectures* which have CPI less than 1.
- *Multiple* instructions can be launched at the *same time (multiple issue)* - Instruction execution rate exceeds the clock rate! We're talking of number of Instructions per Clock Cycle (IPC instead of CPI)
- Architectures try to issue 3 to 8 instructions at every clock cycle.
- A third way is to balance load through *dynamic pipeline scheduling*, to avoid hazards (stalls).
- The price for these speed-ups is more hardware, more complicated control and a more complicated instruction execution model.
- If instructions are launched in pairs, only the first instruction is launched if dynamic conditions are not met.

Static Multiple Issue

- Used in embedded processors and VLIW processors
- Can improve performance by up to 200%
- Layout is restricted to simplify the decoding and instruction issue
- Instructions are issued in pairs, aligned on a 64-bit boundary with the ALU and branch portion operating first;
- If one of the instruction of the pair cannot be used, it is replaced by a no-op.
- The hardware detects data hazards and generates stalls between two issue packets, but the compiler is required to avoid all dependencies *within* the instruction pair.
- A load will cause the next *two* instructions to stall if they were to use the loaded word**.**

Example



Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
add …
lw
beq …
sw
sub …
lw …
IM
Reg
ALU
DM
Reg