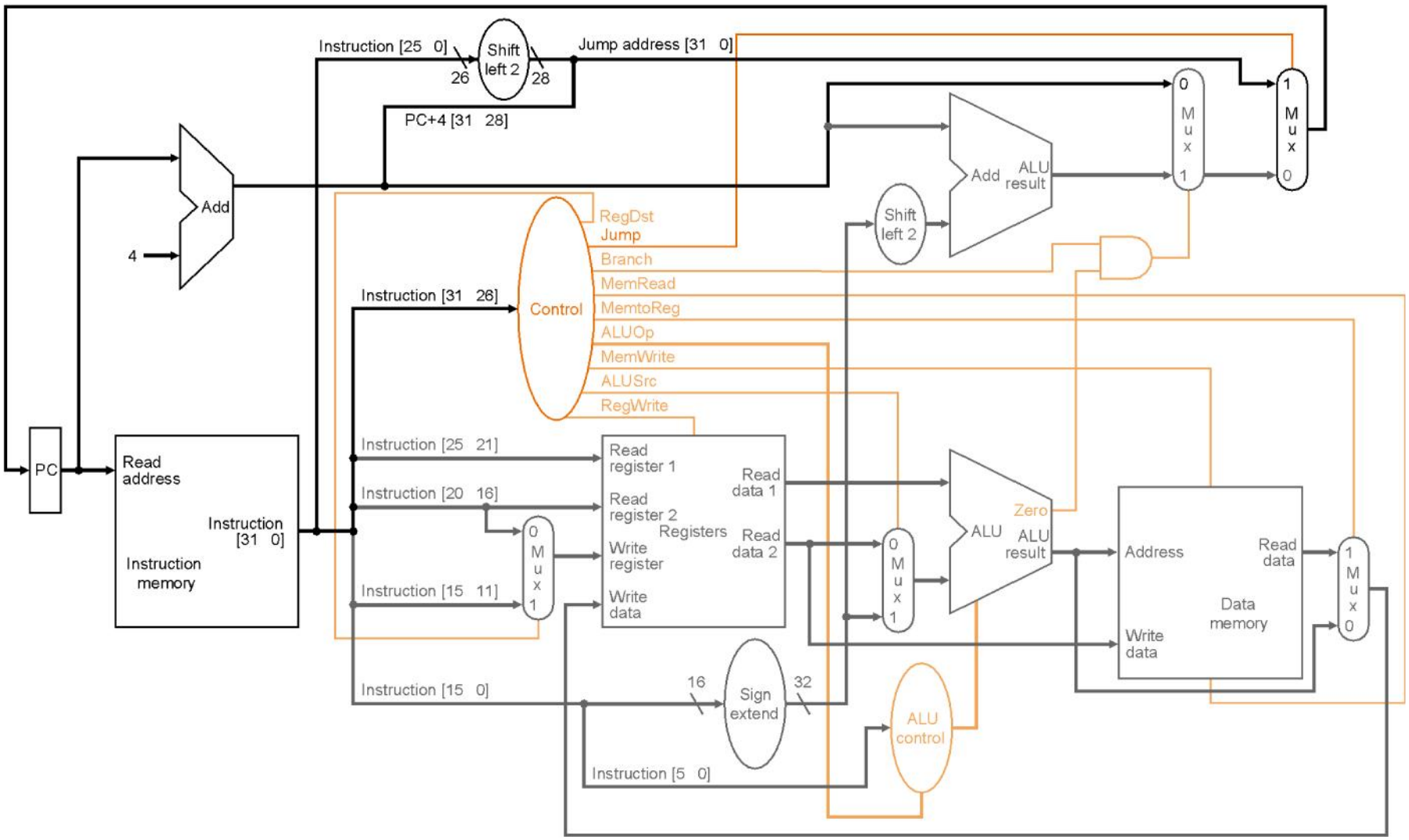
Pipelining -- Outline

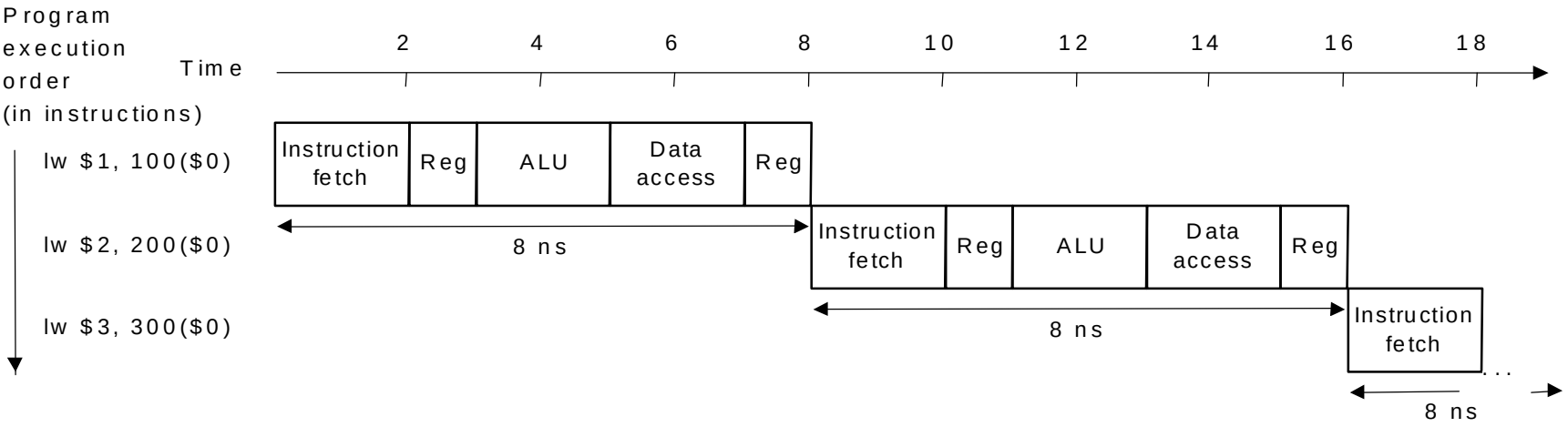
- Basic idea
- Pipelined datapath
- Pipelined control
- Data hazards
- Three methods to handle data hazards

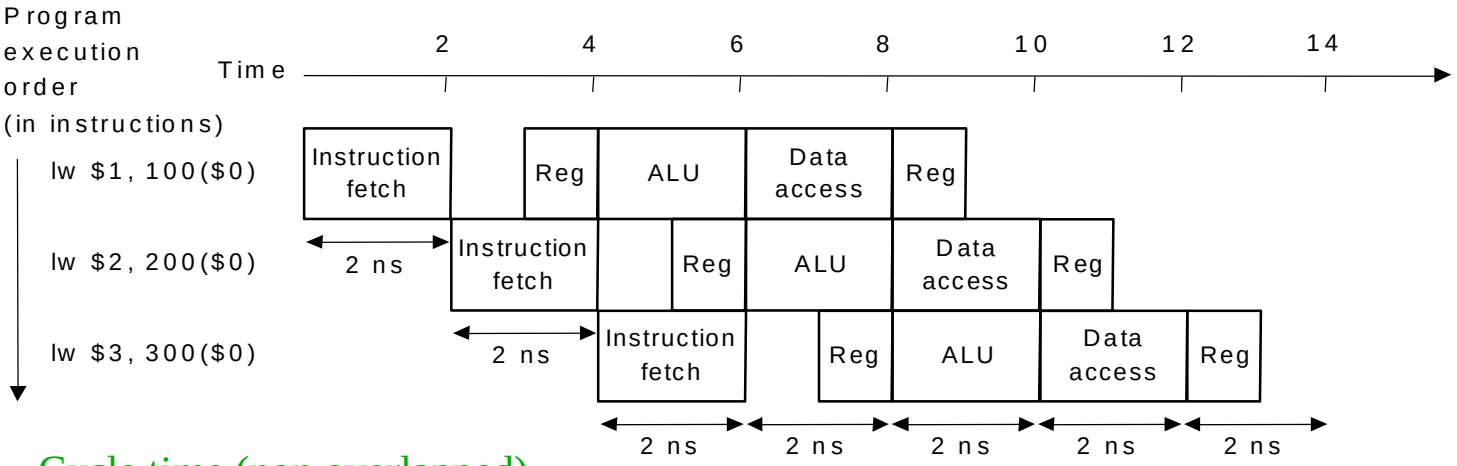

Instruction [25 0]
Shift left 2
26
28
Jump address [31 0]
PC+4 [31 28]
Add
4
0
M
u
x
1
1
M
u
x
0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31 26]
Control
PC
Read address
Instruction [31 0]
Instruction memory
Instruction [25 21]
Instruction [20 16]
Instruction [15 11]
Instruction [15 0]
Instruction [5 0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data

Basic idea of pipeline

- For single cycle computer, roughly, 5 steps
  - IF, ID, EX, MEM, WB
- For the current step, hardware for other steps are idle.
- Use the idle hardware to work on other instructions
  - Overlap the instruction executions.
  - Add buffers to hold partial results of instruction execution

Pipeline Basic idea: Non overlapped vs. Overlapped





Cycle time (non overlapped)

Cycle time (overlapped)

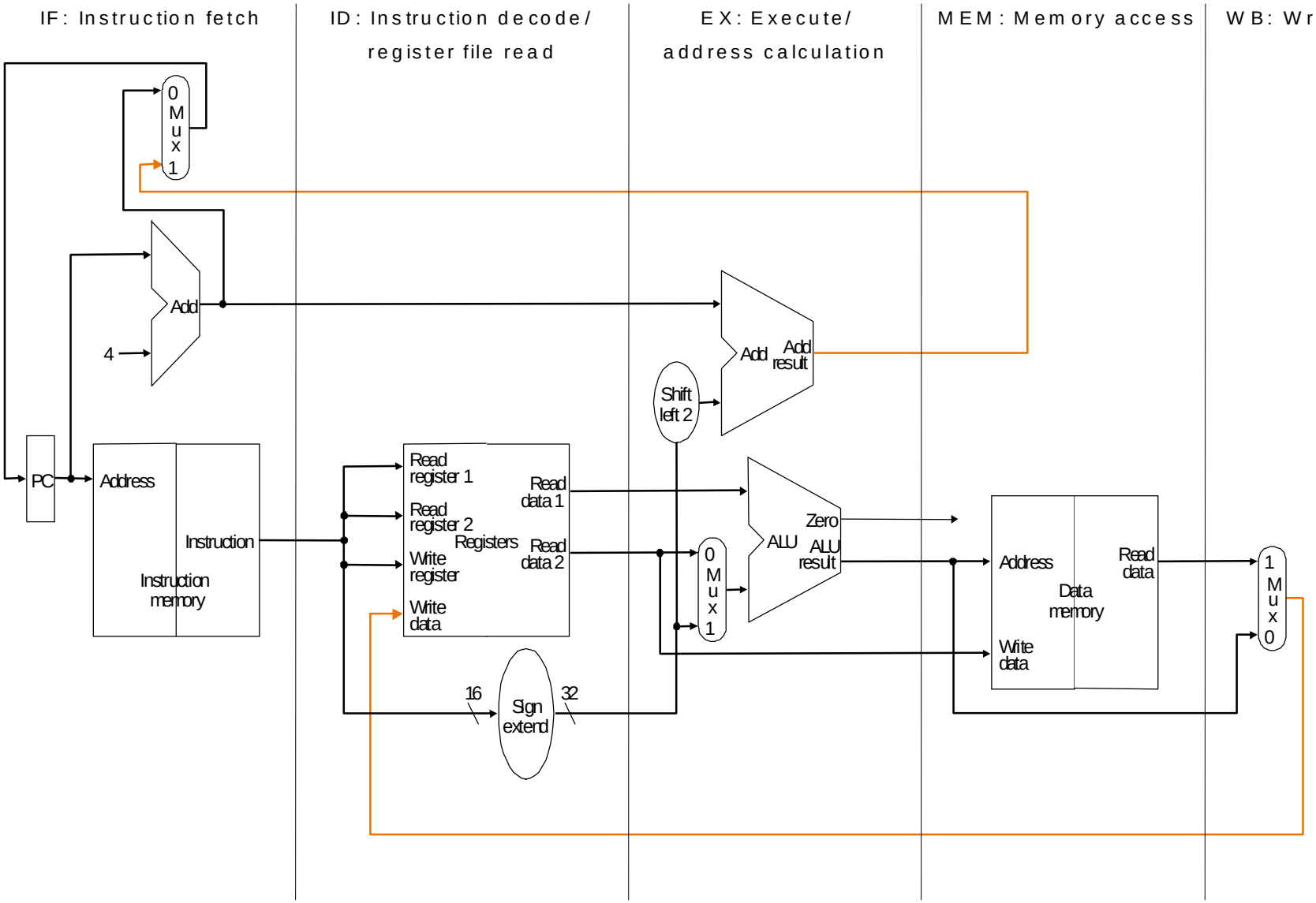
#cycles (overlapped with n `lw`s) =

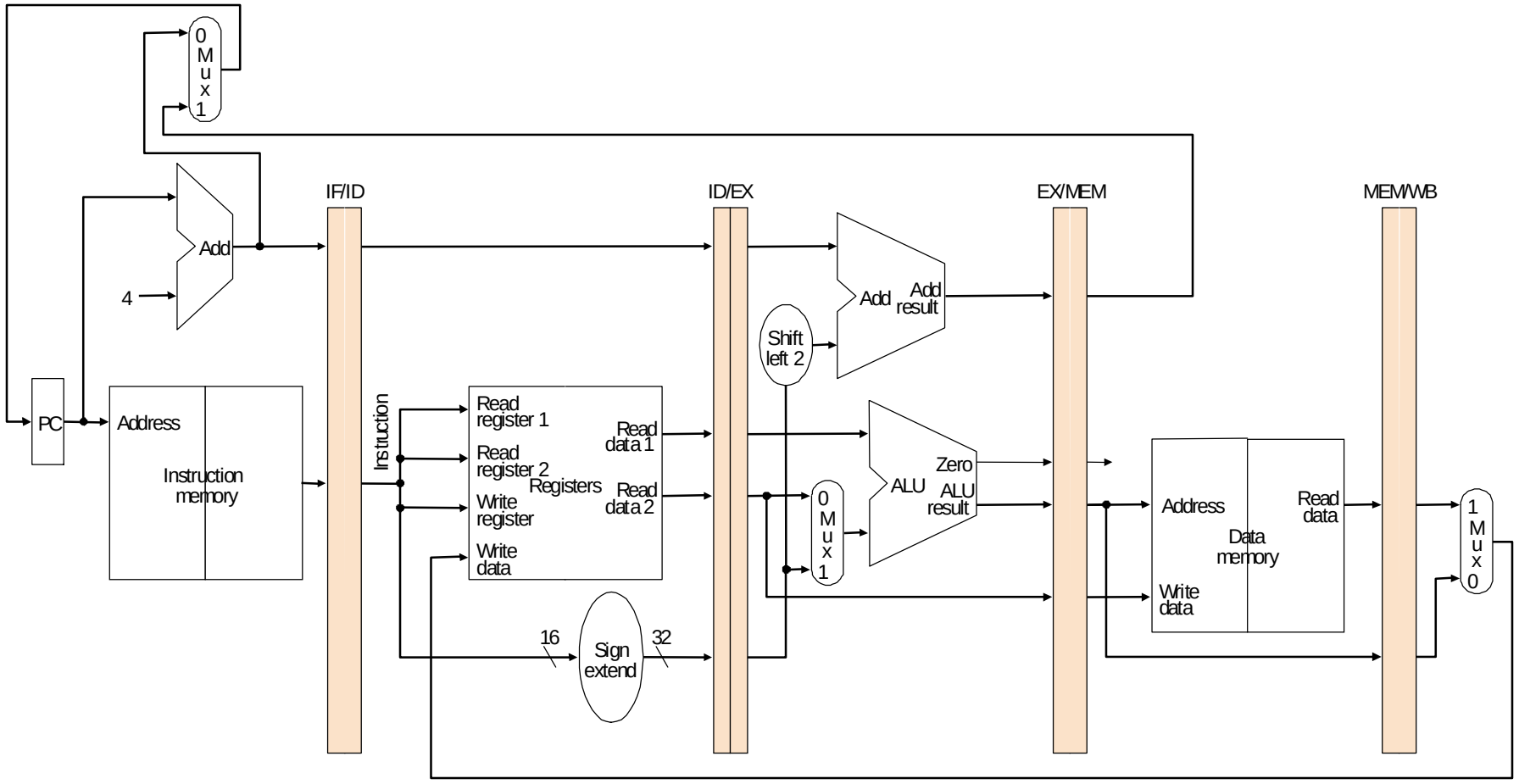
Pipelining: Basic Idea (con't)

- Pipeline: implementation technique in which multiple inst. are overlapped in execution
  - Improve inst. throughput rather than inst. execution time
  - speedup = $\frac{Execution\ tim\ (nonoverlpped\ )}{Execution\ time\ (overlapped\ )} \approx \#\ stages$
  - pipe stage: balancing length of each stage with equal length, limited # pipe stages

MIPS Pipe: 5 stages

- IF: instruction fetch
- ID: inst. decode and register fetch
- EX: execution and effective address calculation
- MEM: memory access
- WB: write back to register


IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
0
M
u
x
1
Add
4
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
0
M
u
x
1
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign extend
32
1
M
u
x
0

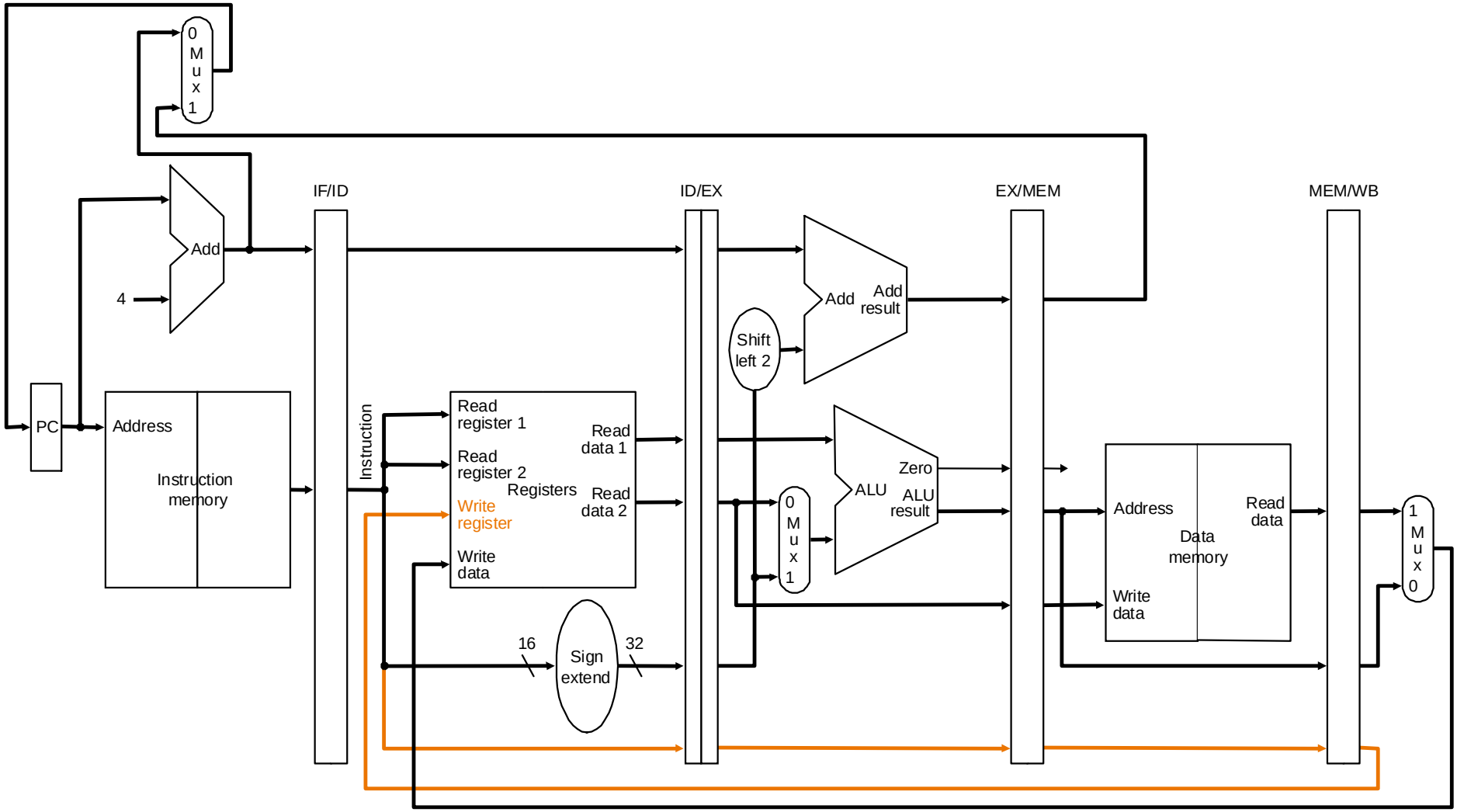


Add buffers (**pipeline registers**) between pipeline stages

Any information needed in a later stage must be passed to that stage via a pipeline register

Any problem with register write back?

Pipelining -- Control?



Pipeline control

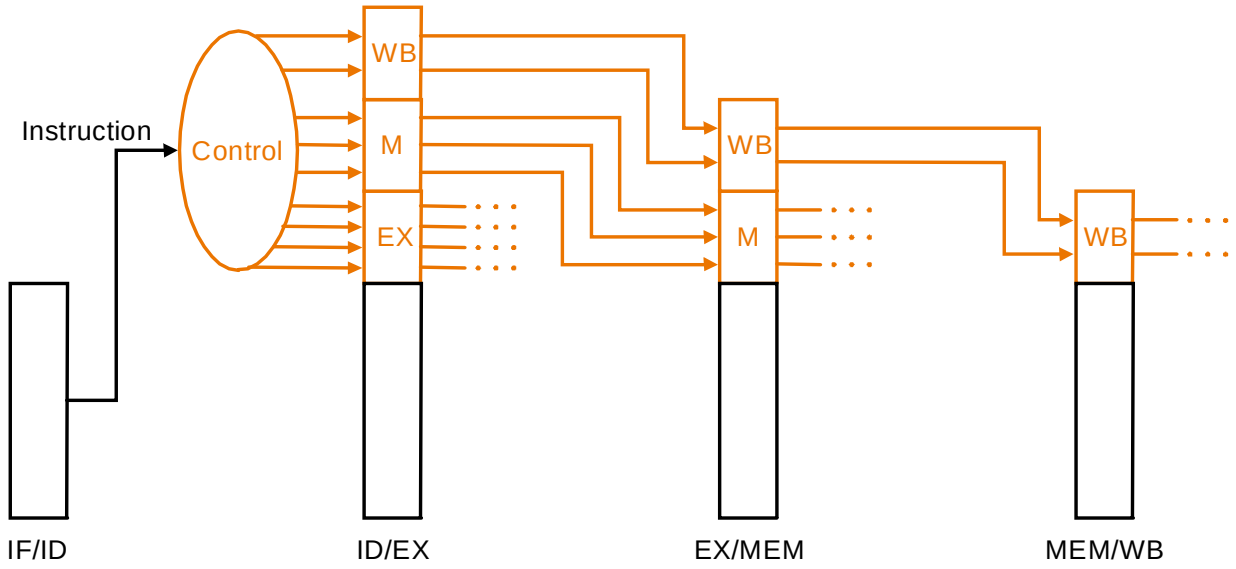
- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

- Generate all control signals in the ID stage and pipeline them

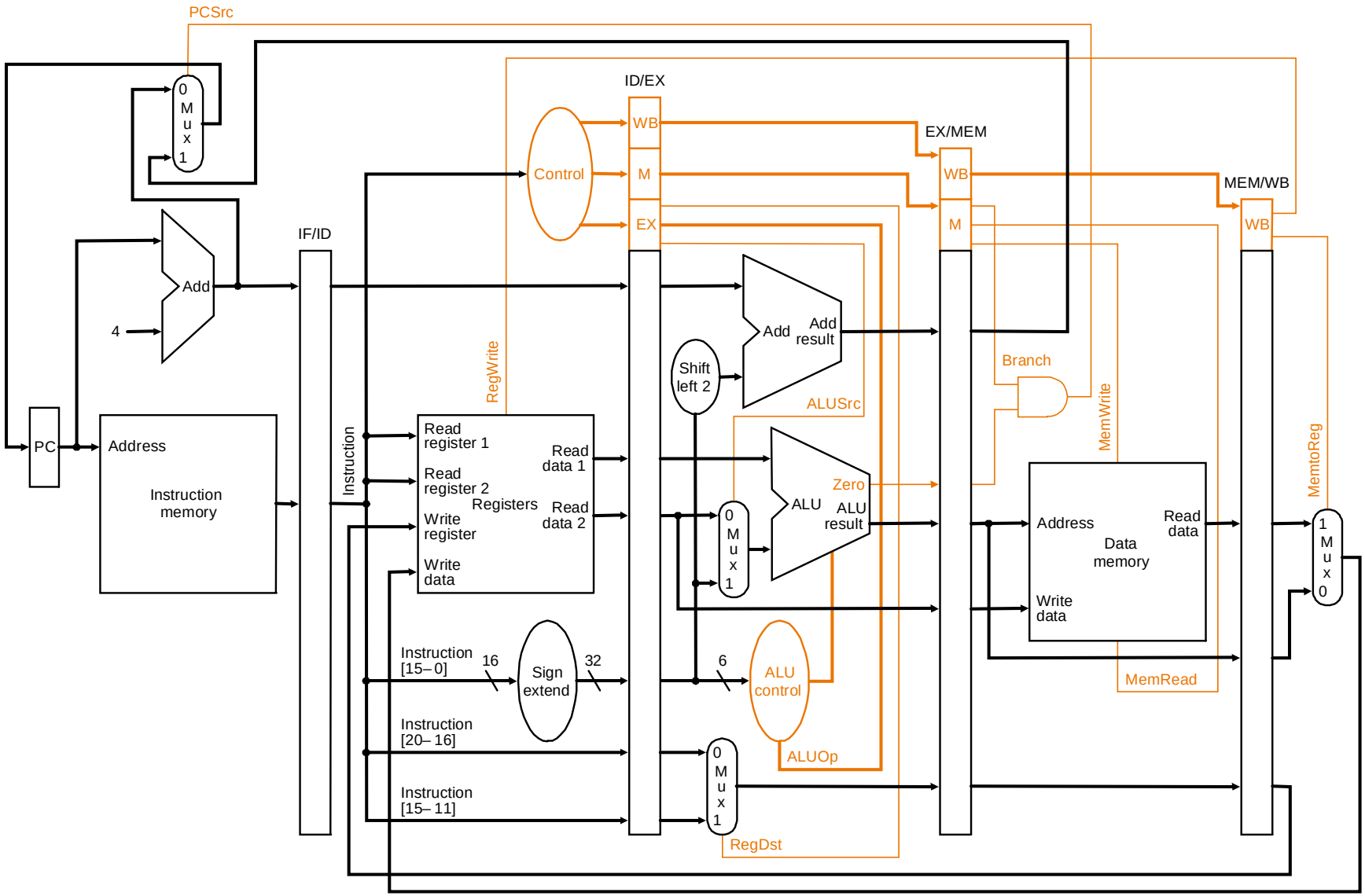
Pipeline Control

- Pass control signals along just like the data

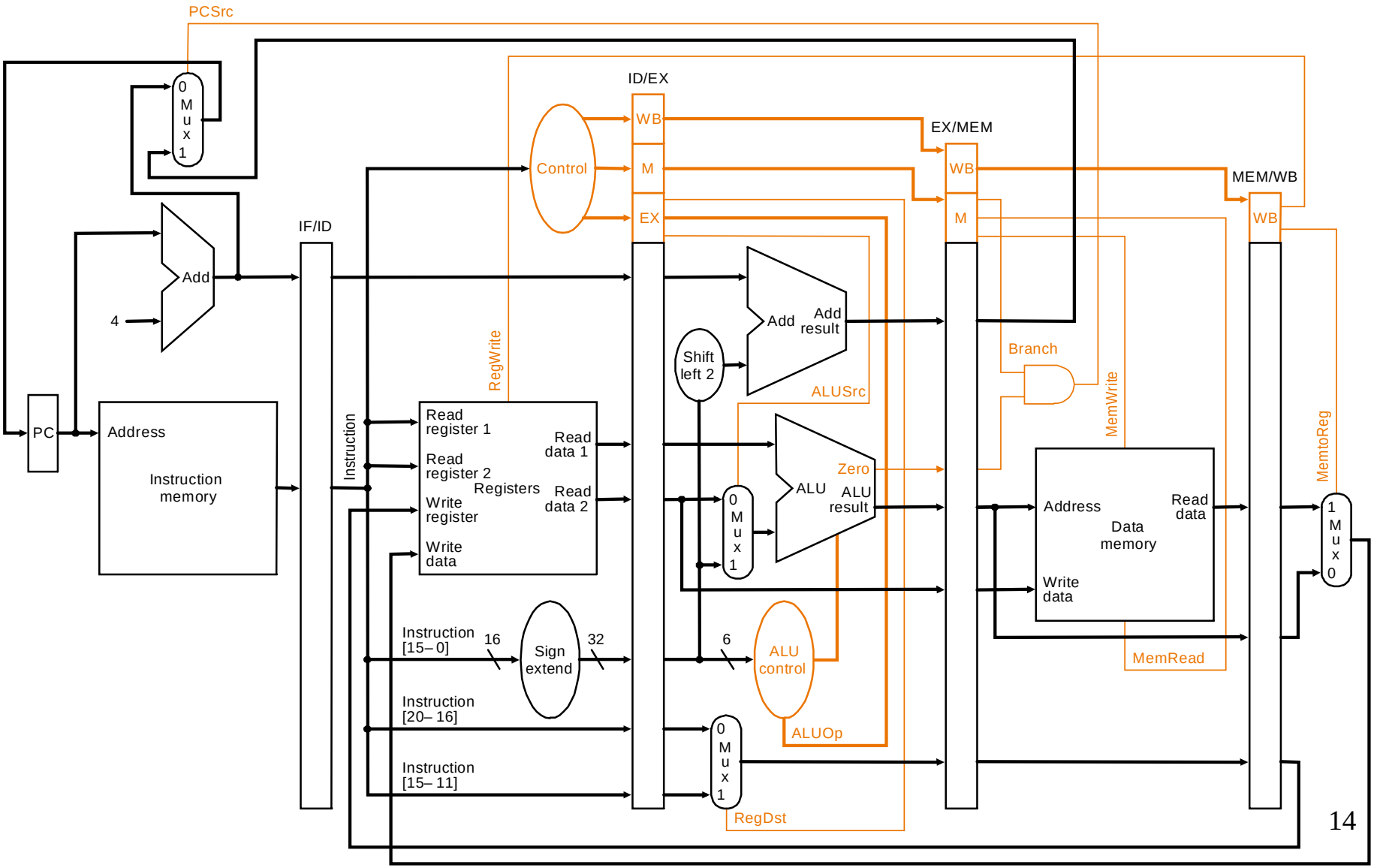
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



Pipeline: Datapath + Control



Example: How *add $1, $2, $3* Through the Pipe


PCSrc
0
M
u
x
1
Add
4
PC
Address
Instruction
memory
IF/ID
Instruction
RegWrite
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Control
ID/EX
WB
M
EX
Add
Add
result
Shift
left 2
ALUSrc
ALU
Zero
ALU
result
Instruction
[15– 0]
16
Sign
extend
32
6
ALU
control
ALUOp
Instruction
[20– 16]
Instruction
[15– 11]
RegDst
EX/MEM
WB
M
Branch
MemWrite
Address
Data
memory
Read
data
Write
data
MemRead
MEM/WB
WB
MemtoReg
1
M
u
x
0

Performance

- Execution time of execution *n* load word instructions
- Speedup
- CPI
- Size of each pipeline buffer:
  - IF/ID
  - ID/EX
  - EX/MEM
  - MEM/WB

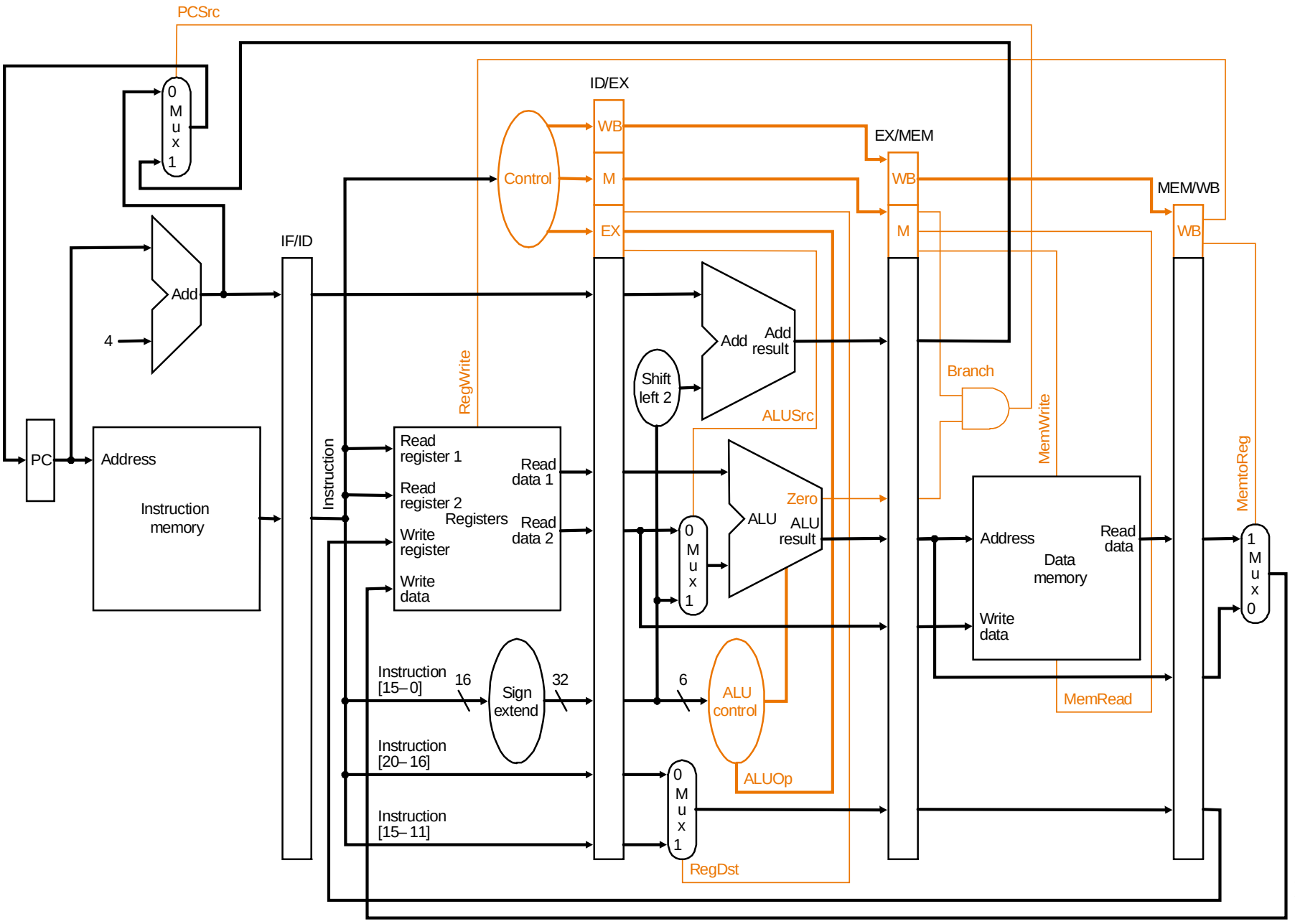
Example: How the five instructions go through the pipe, and what are the register values

- Determine the values of every field in 4 pipeline registers in cycle 5 for instructions

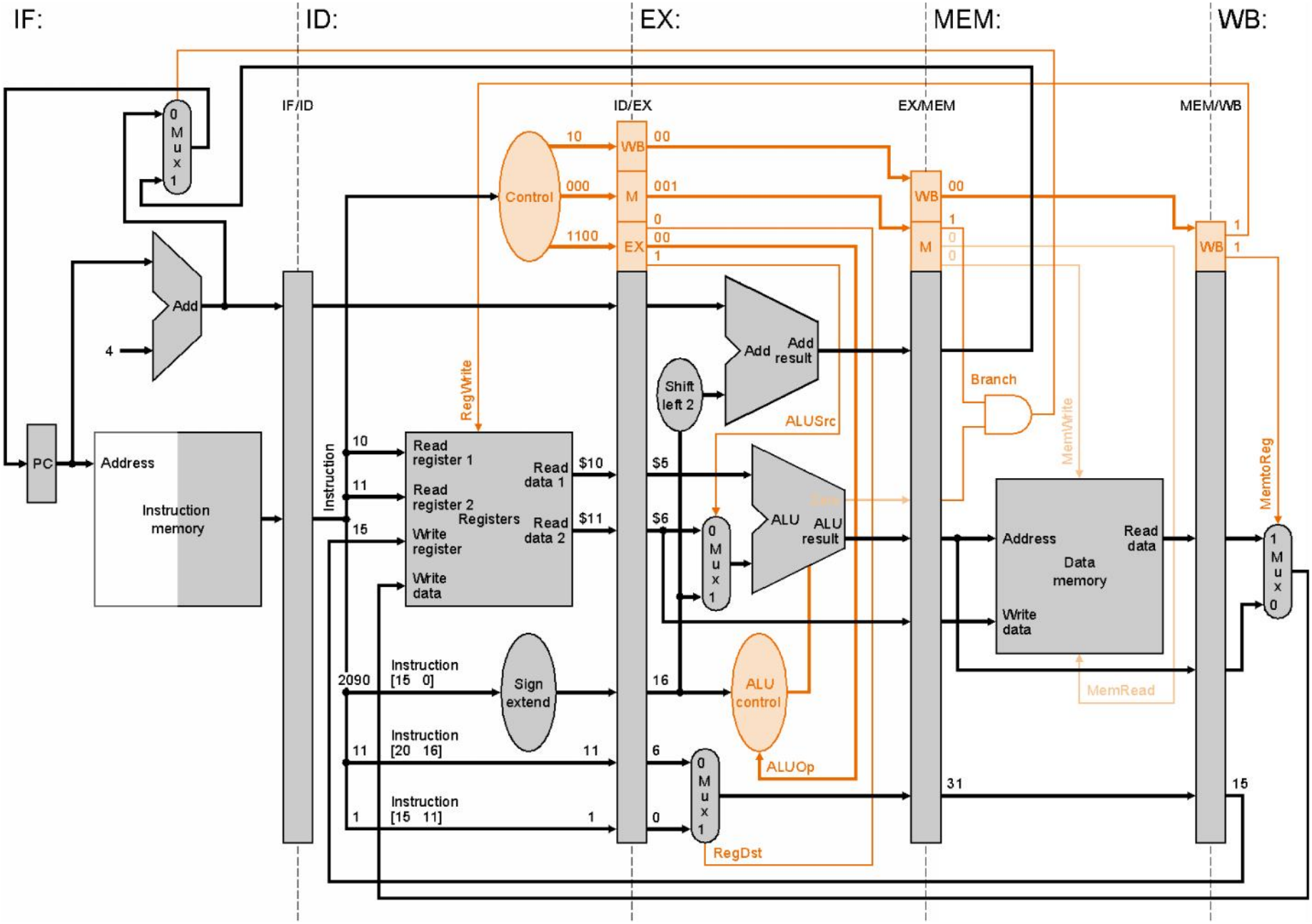
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

- Initial value of PC: 500
- Initial values of registers
  10+ register number
- Initial values of memory cells
  1000+memory address

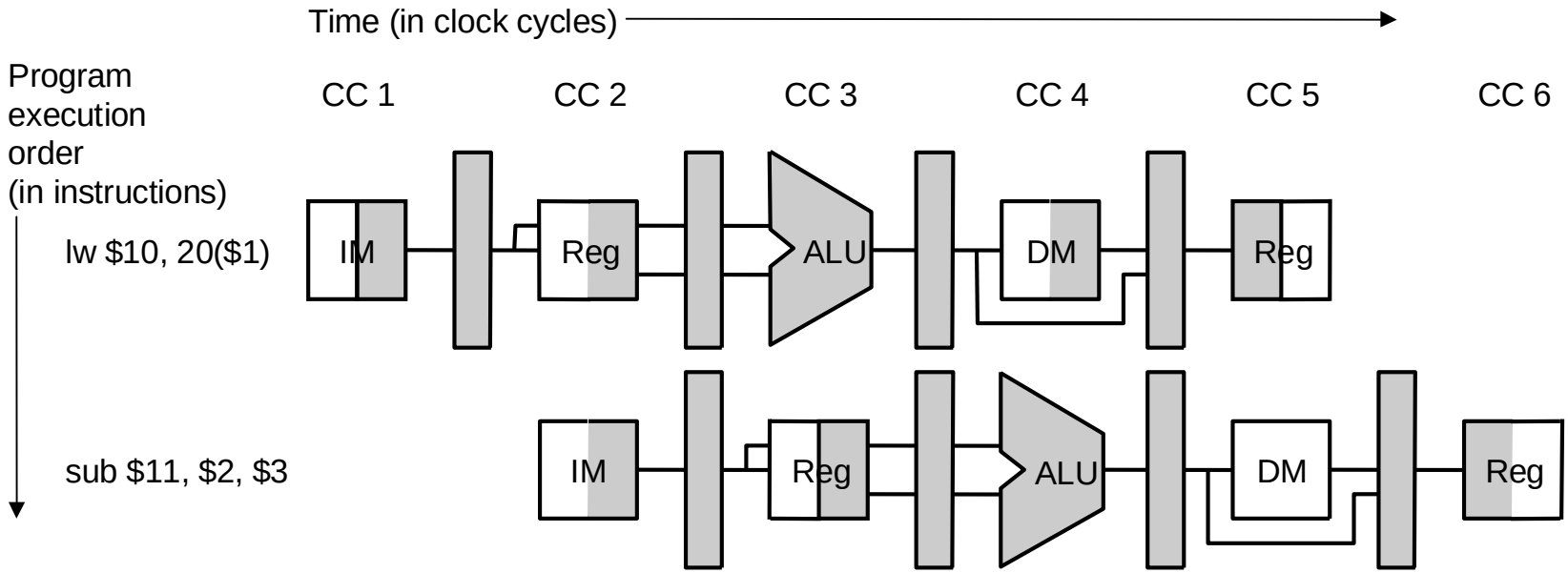
- Assume register
  - MEM/WB constrains info of "`lw $10, 20($1)`"
  - EX/MEM: `sub $11, $2, $3`
  - ID/EX: `and $12, $4, $5`
  - IF/ID: `or $13, $6, $7`


PCSrc
0
M
u
x
1
ID/EX
WB
M
EX
Control
EX/MEM
WB
M
MEM/WB
WB
IF/ID
Add
4
PC
Address
Instruction
memory
Instruction
RegWrite
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Shift
left 2
Add
Add
result
ALUSrc
Branch
MemWrite
MemtoReg
Zero
ALU
ALU
result
Address
Data
memory
Read
data
Write
data
MemRead
Instruction
[15–0]
16
Sign
extend
32
6
ALU
control
ALUOp
Instruction
[20–16]
Instruction
[15–11]
RegDst

Example: given register values, find the instructions



Graphically Representing Pipelines



- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Pipeline Hazards

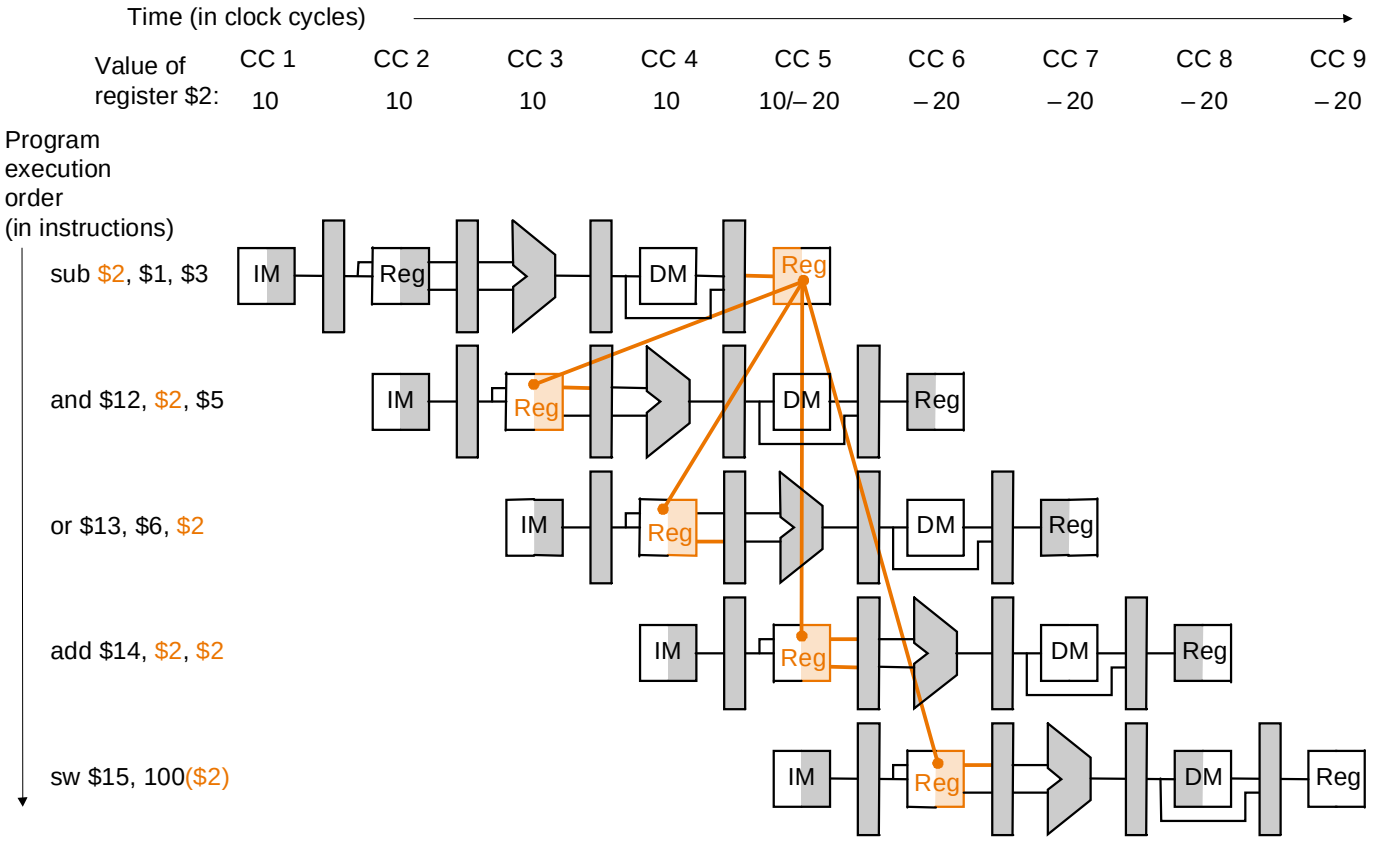
- Structural hazards -- caused by hardware resource conflicts
- Data hazards -- caused by data dependence
- Control hazards -- caused by branch / jump instructions

Data Dependence

- *Sub $2, $1, $3* *I1*
  and $12, $2, $5 *I2*
  or $13, $6, $2 *I3*
  add $14, $2, $2 *I4*
  sw $15, 100($2) *I5*
- Read after write dependence (RAW)
- Write after read dependence (WAR)
- Write after write dependence (WAW)
- Data dependence does not cause any problem in non overlapped execution

Data Dependence --Con't

- Problem: starting next instruction before the current is finished



I4 is ok when register write happens when clock is high & register read happens when clock is low

How to Handle Data Dependence?

- Compiler approach
  - insert dummy operations or “nops”
  - reschedule instructions
- Hardware approach
  - forwarding

Compiler Approach 1 for Data Hazards

- Compiler generates code without data hazards by inserting dummy operations
  sub $2, $1, $3
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100($2)

  This code has data hazards, not allowed
- Insert dummy operations “nop”
- Problem: this really slows us down!

Compiler Approach 2-- Scheduling

- Move independent inst. around to eliminate data hazards and fill in bubbles

  I1 *and $18, $9, $10*
  I2 *sub $2, $1, $3*
  I3 *and $12, $2, $5*
  I4 *or $13, $6,$2*
  I5 *add $14, $2, $2*
  I6 *sw $15, 100($2)*
  I7 *sub $16, $7, $8*
  I8 *add $17, $8, $9*

- RAW dependence b/w I2 and I3, I4, I5, I6

How to identify all dependence

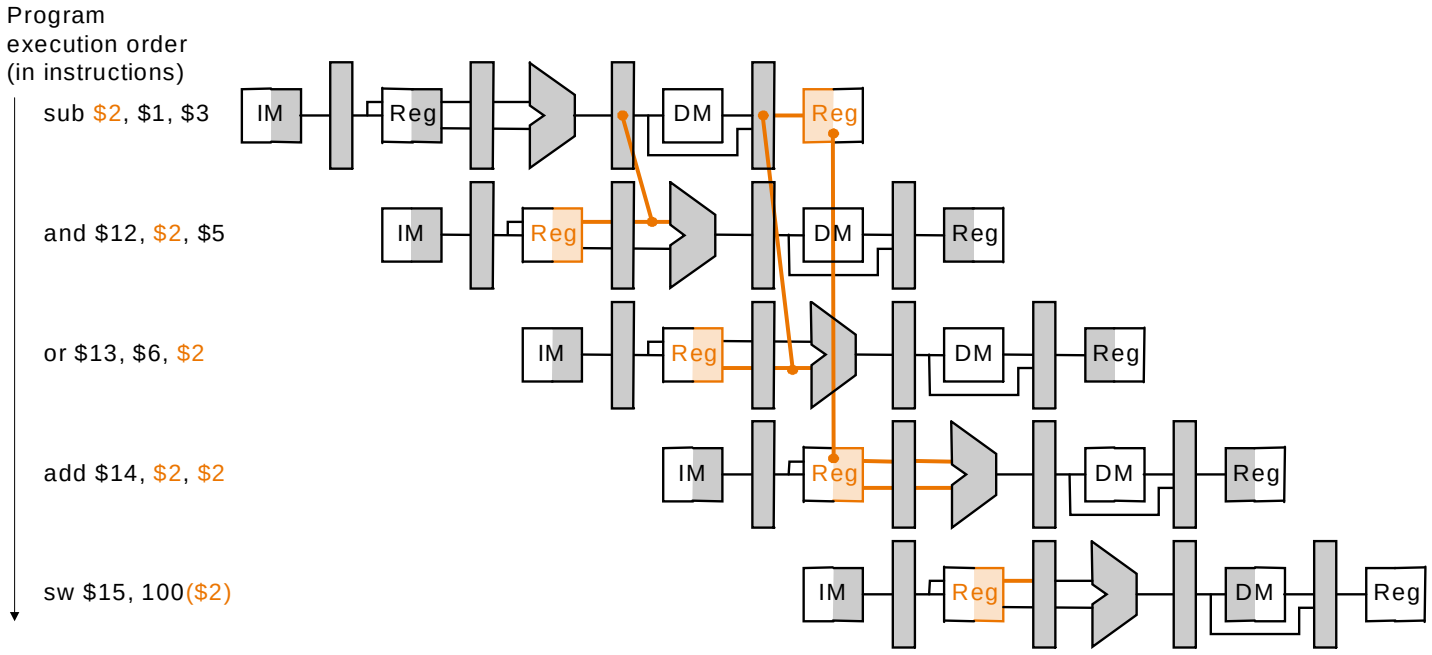
- Find all instructions that write registers
- For each such instruction writing register $j, identify all other instructions that either read or write register $j

Hardware Solution: Forwarding

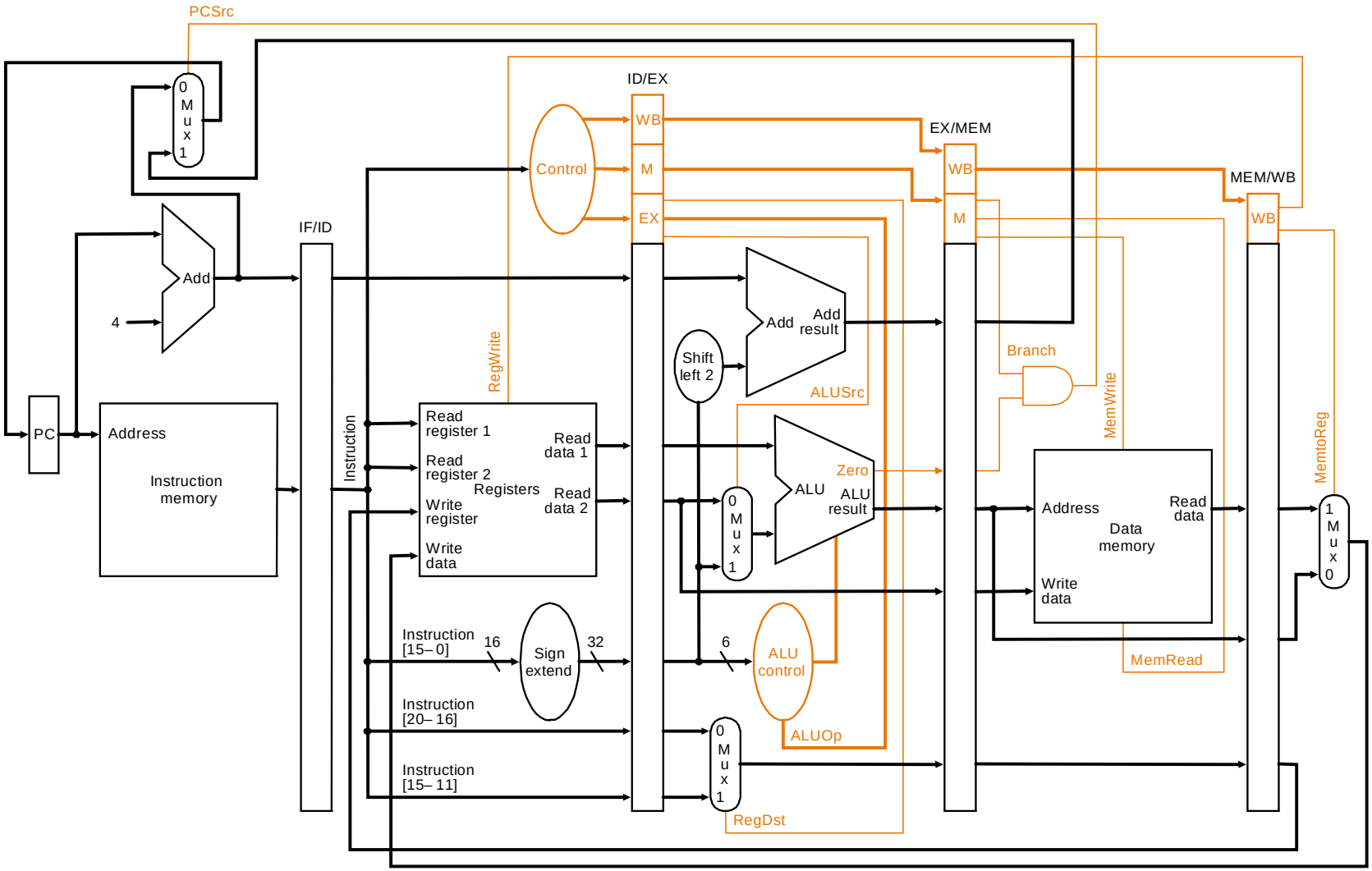
- Use temporary results, don't wait for them to be written
  - ALU forwarding: ALU output flows in pipe registers and forwarded to ALU input
  - #stalls to resolve the hazard: 2

Time (in clock cycles) →

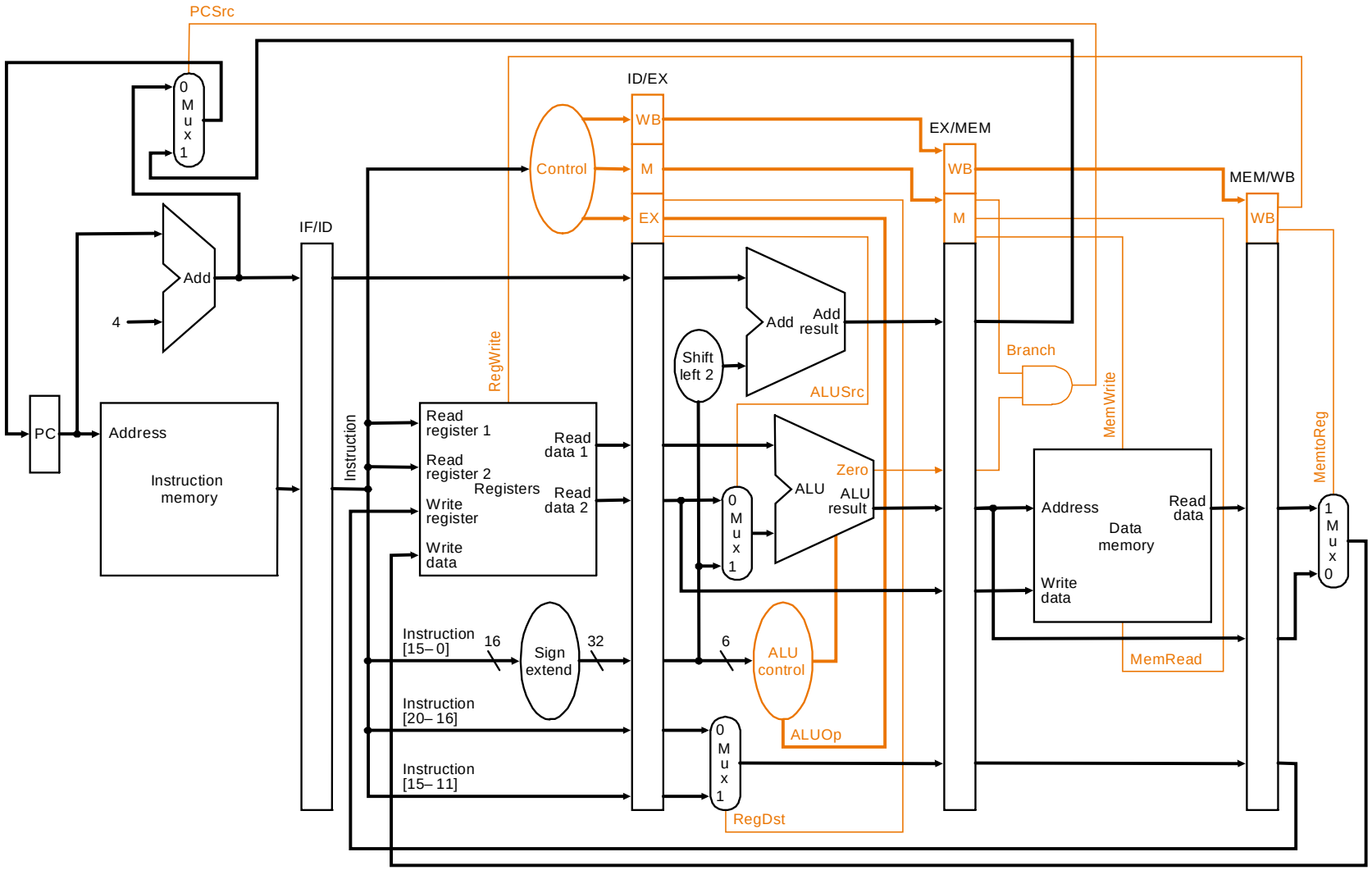
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



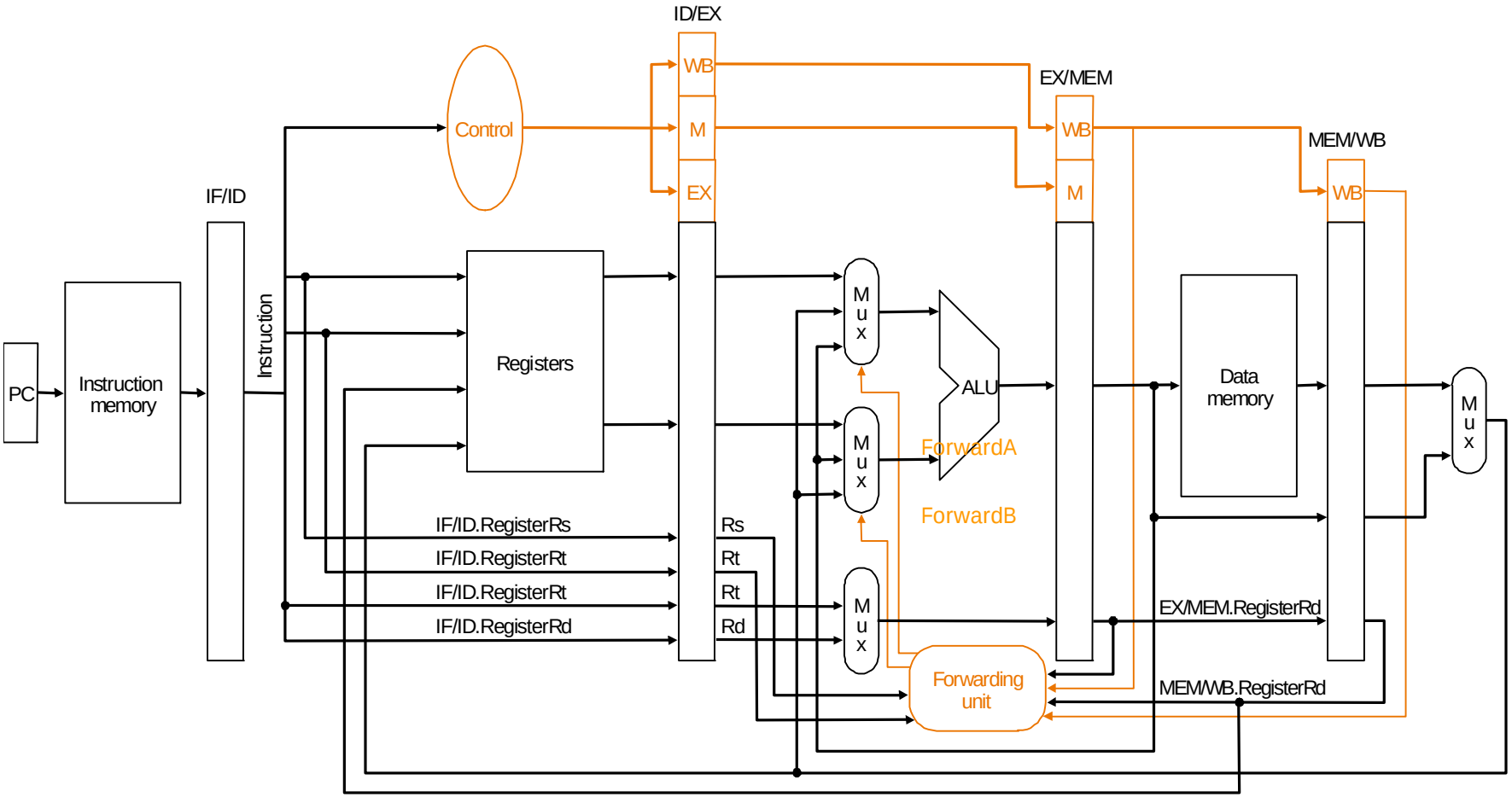
What is going on in cycle 4



What is going on in cycle 5



ALU Forwarding -- Datapath & Control



Keep rs in ID/EX

Forwarding Unit

- Compares the 2 read addresses of the instruction in EX with write addresses of instructions in MEM and WB
- Input
  - rs and rt in ID/EX
  - write address (rd or rt) and RegWrt in EX/MEM
  - write address (rd or rt) and RegWrt in MEM/WB
- Output
  - ForwardA to control the top mux before ALU
  - ForwardB to control the bottom mux before ALU

Forwarding--con't

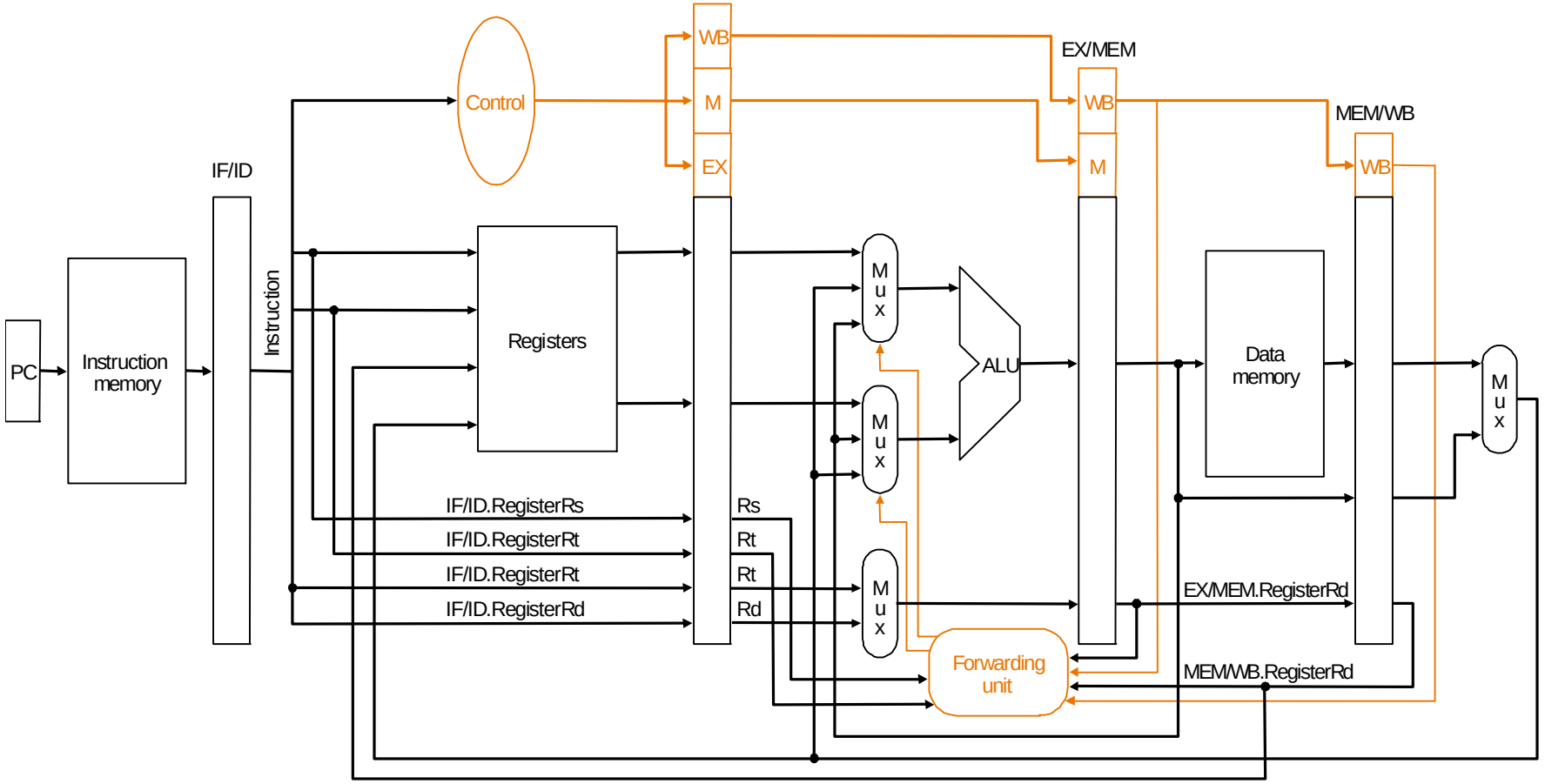
- Suppose instruction I1 in stage MEM, I2 in EX
- Forwarding unit tries to see if there are dependence b/w
  - I1 & I2: Rd addresses of I2, wrt address of I1, Regwrt of I1
- Wrtdata of I1 might be fed back to both ALU inputs

```
I1 sub $2, $1, $3
I2 and $12, $2, $5
I3 or $13, $6, $2
…
```

Or …

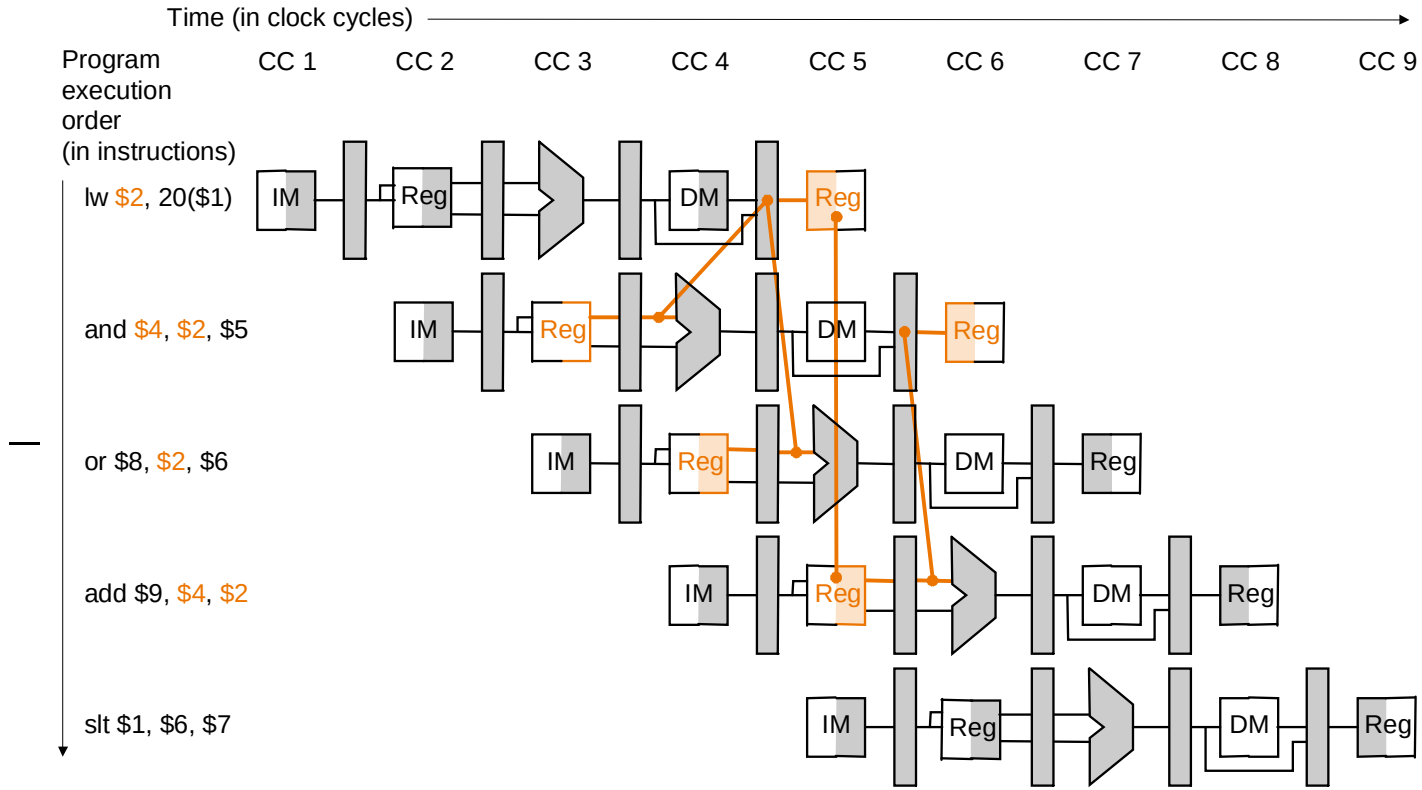
And $12, $2, $5

```
Sub $2, $1, $3
```



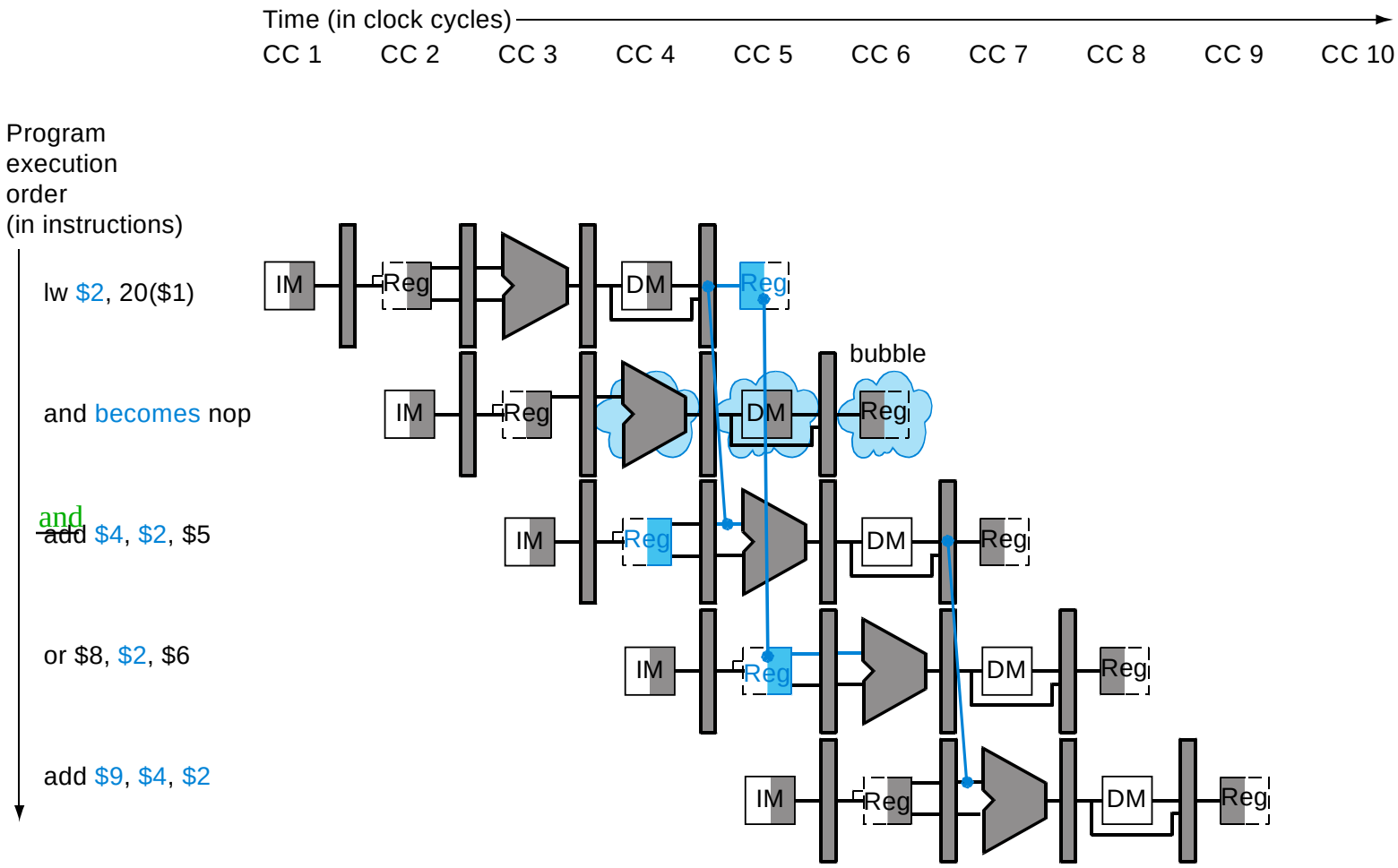
Can't always forward

- Load word can still cause a hazard:
  - a read after a *lw* write to the same register.



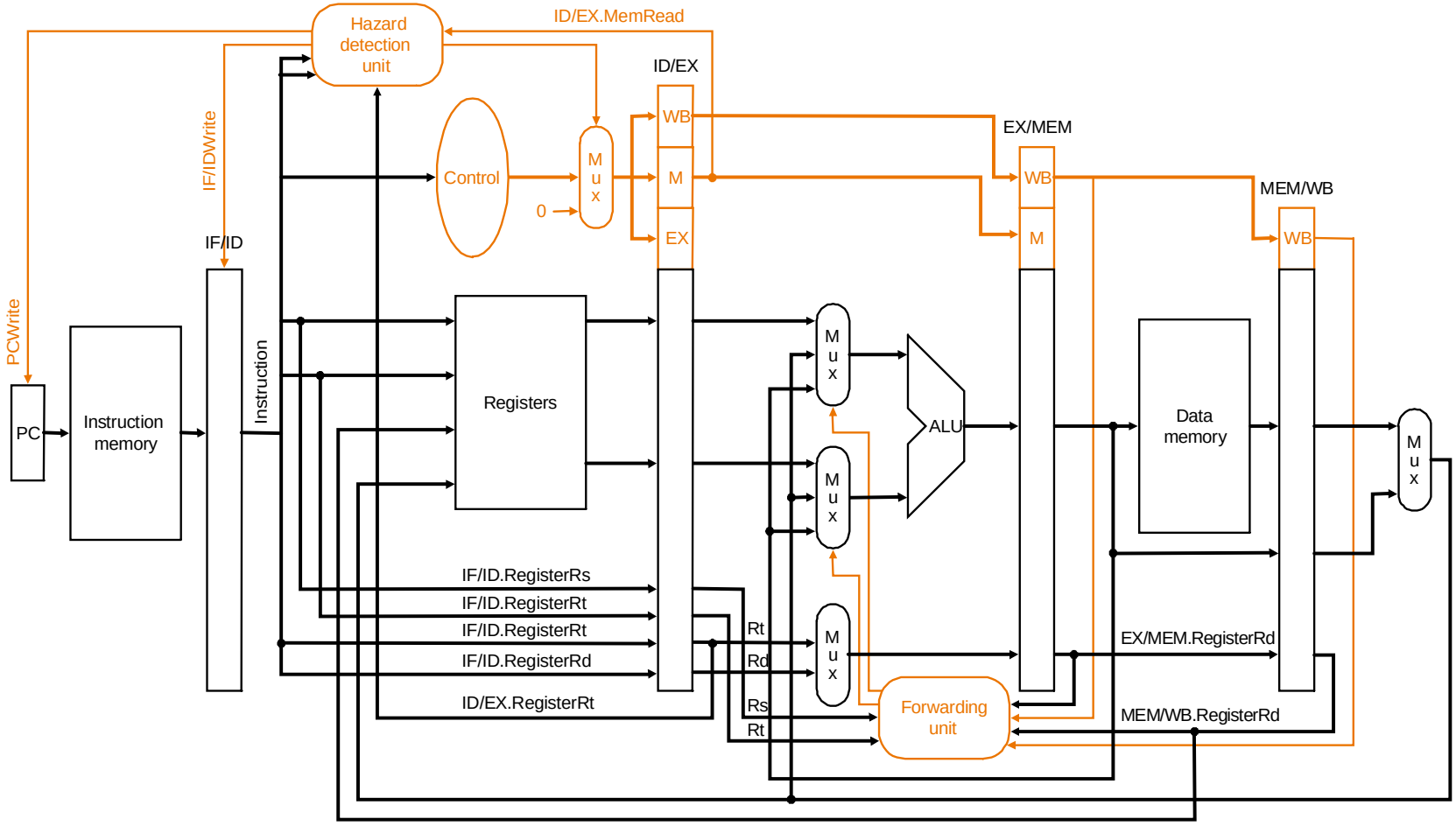
Thus, we need a hazard detection unit to “stall” the load instruction

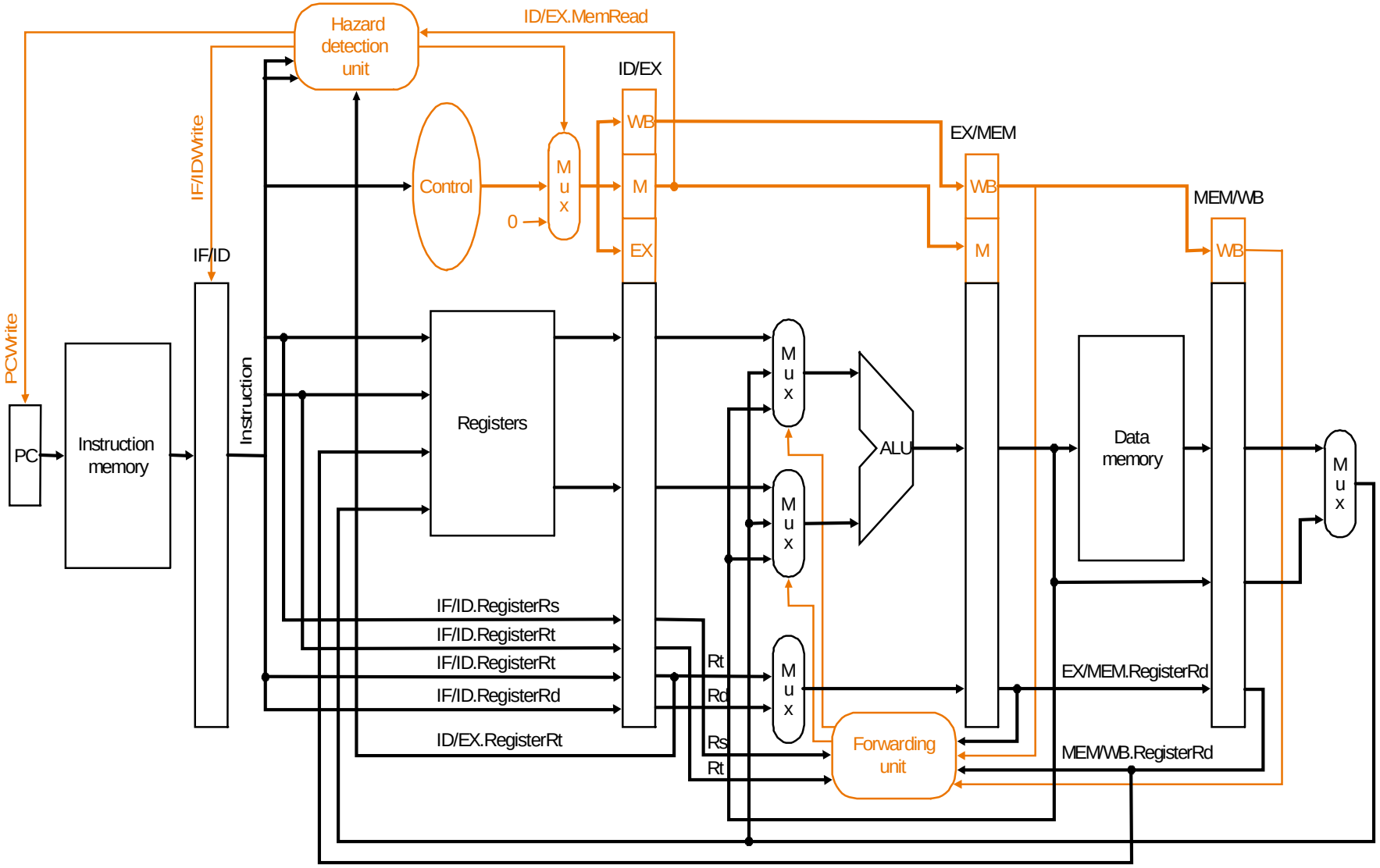
We can stall the pipeline by keeping an instruction in the same stage



Hazard Detection Unit

- Stall by letting an instruction that won't write anything go forward




Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
M u x
0
WB
M
EX
EX/MEM
MEM/WB
IF/ID
PCWrite
PC
Instruction memory
Instruction
Registers
ALU
Data memory
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
ID/EX.RegisterRt
Rt
Rd
Rs
Rt
Forwarding unit
EX/MEM.RegisterRd
MEM/WB.RegisterRd

Hazard Detection--con’t

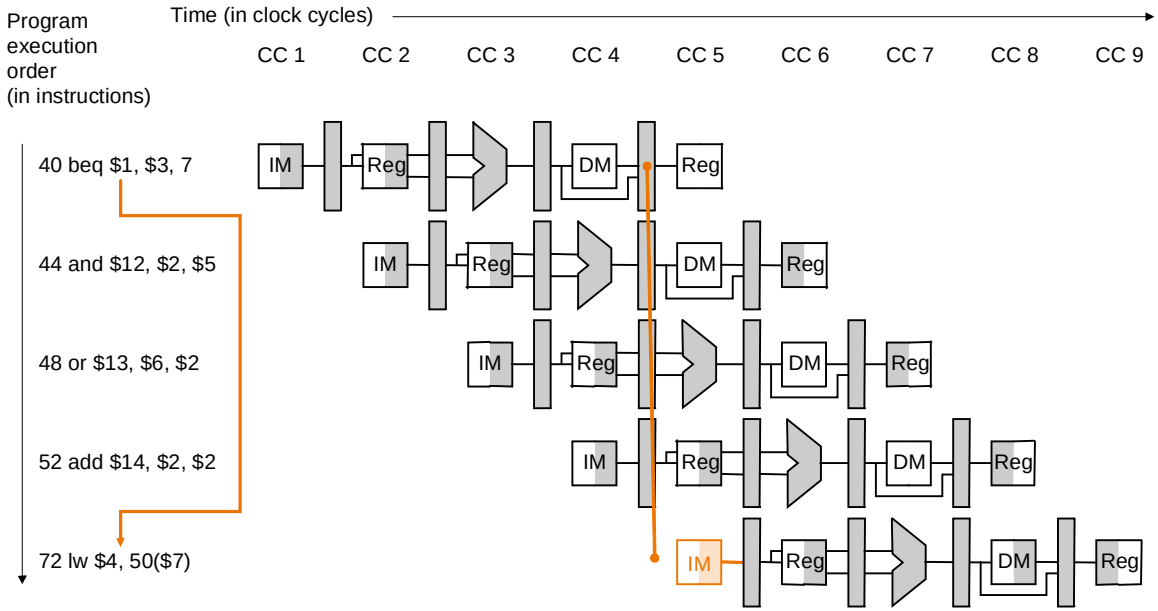
- Detects if I3 is a lw instruction and if I4 depends on I3(I3 in EX and I4 in ID)
  - checking memRd of I3
  - compare wrt address of I3 with Rd addresses of I4
- If both yes, insert a bubble by
  - do not write PC (I5 is fetched twice)
  - do not write IF/ID (I4 stays in ID for 1 more cycle)
  - inster a nop (let 00 000 0000 as the control values passing the mux)

Data Hazard Summary

- Data dependence
  - RAW, WAR, WAW
- Data hazard caused by RAW
- Methods to resolve hazards
  - Compiler approach
  - Hardware forwarding
    - ALU forwarding unit
      - Inputs: read and write addresses, Regwrt, …
      - Outputs: control the selection of inputs to ALU
    - Special case: dependence caused by `lw`
      - Hazard detection unit: its inputs and outputs.

Branch Hazards

- When we decide to branch, other instructions are in the pipeline!



If branch is not taken: ok

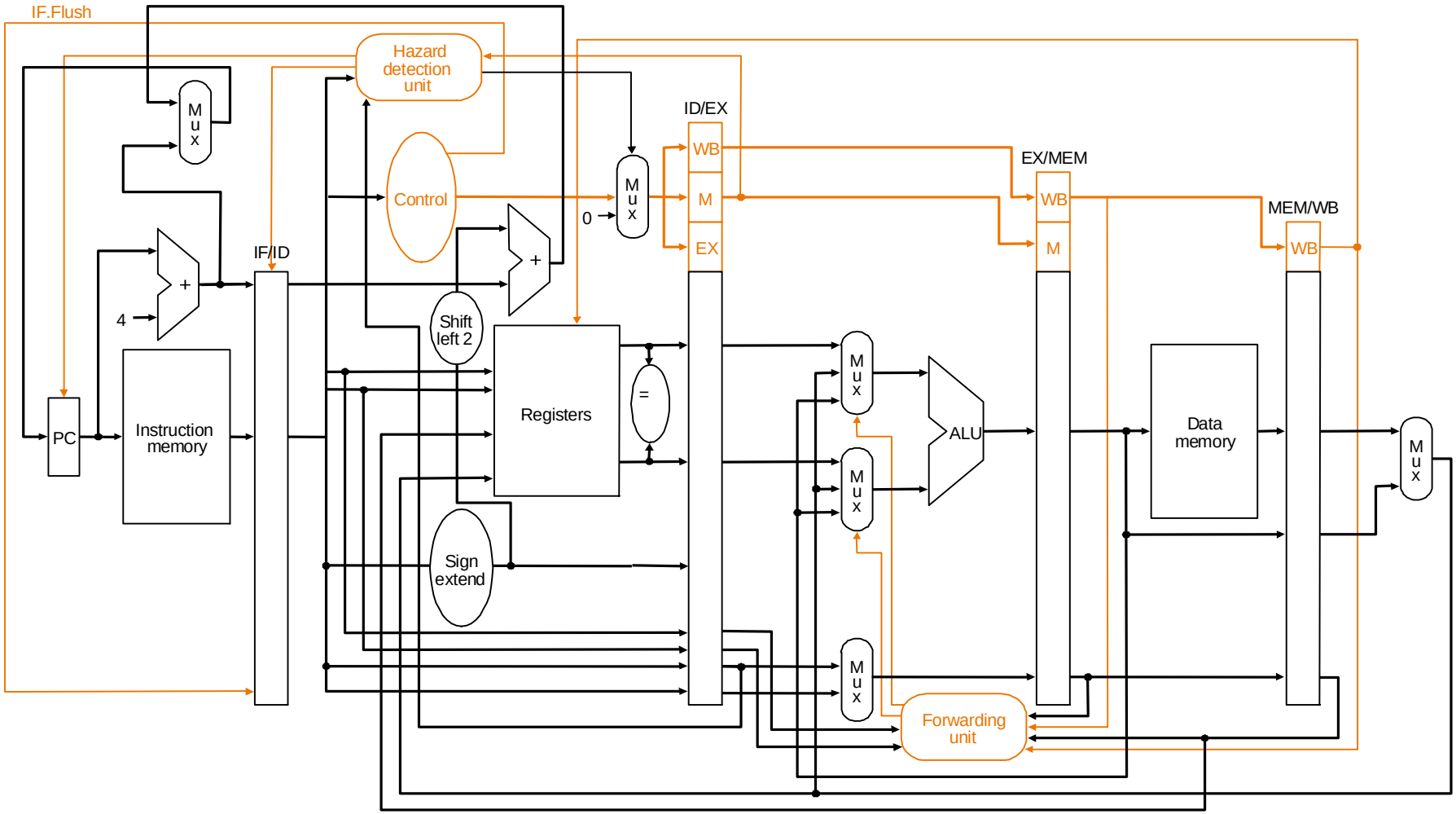
If branch is taken, 3 wrong instructions in pipe already

Solution: flushing the 3 instructions--lost 3 cycles

Control Hazards--How to handle?

- 2 hardware schemes
  - Static: always assume the branch is taken
  - Dynamic: branch prediction
- One optimization
  - move branch addr calculation and condition test to ID stage
    - The lost 3 cycles is cut to 1.
- One software scheme: branch delay slot
  - Compiler inserts an instruction directly after a branch instruction that is independent of the branch inst.
    - No mater the branch is taken or not, that instruction should always be executed.

Flushing instructions



Note: we've also moved branch decision to ID stage

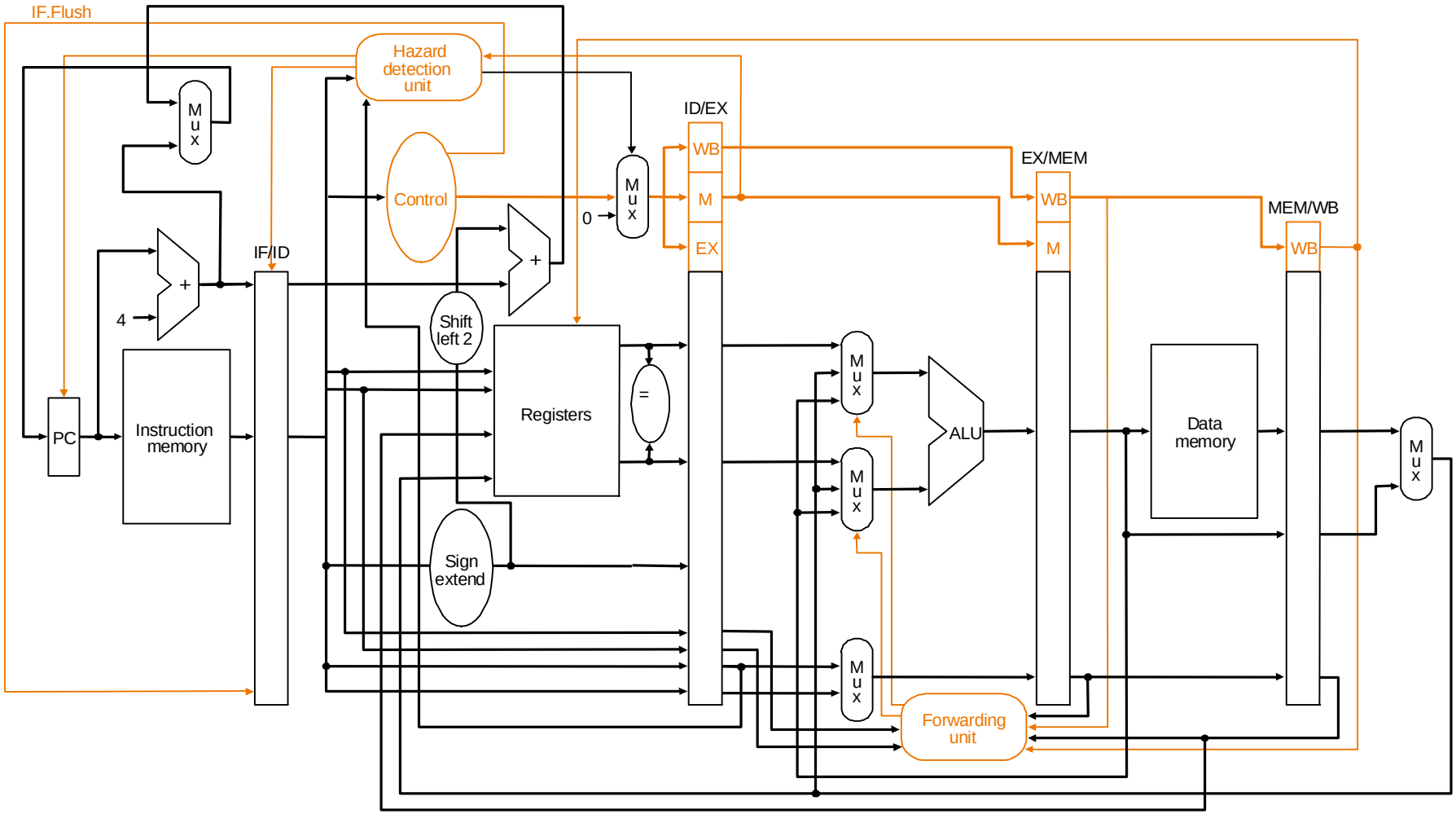
Flushing instructions

- Move branch addr. calculation/condition testing to stage 2 (ID)
- Condition testing: use 32 XOR gates to compare bitwisely
  - delay: negligible
- IF.Flush==branch control line in single cycle datapath
- $F = Zero \bullet IF.Flush$ is used to control write of PC & IF.ID buffer
- If F=1, take branch by doing two operations
  - zero IF.ID buffer (flushing)
  - write PC with branch address
- If branch taken, lose one cycle
- CPI of `beq`

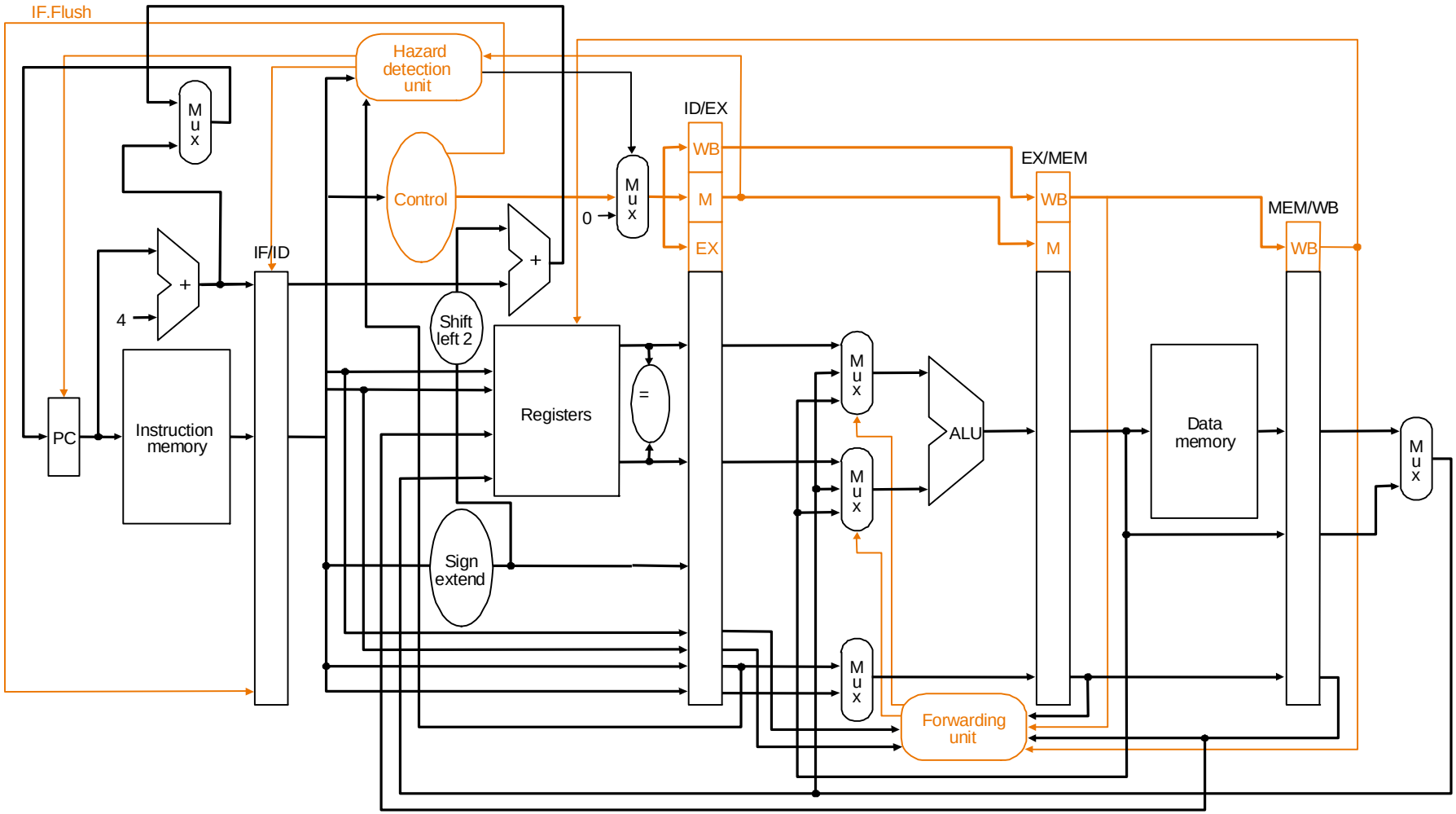
Example

- 32 lw $1, 200($2)
  36 add $3, $1, $4
  40 beq $5, $6, 7
  44 And $12, $2, $5
  48 or $13, $6, $2
  …
  72 lw $4, 50($7)
  assume $5 =$6

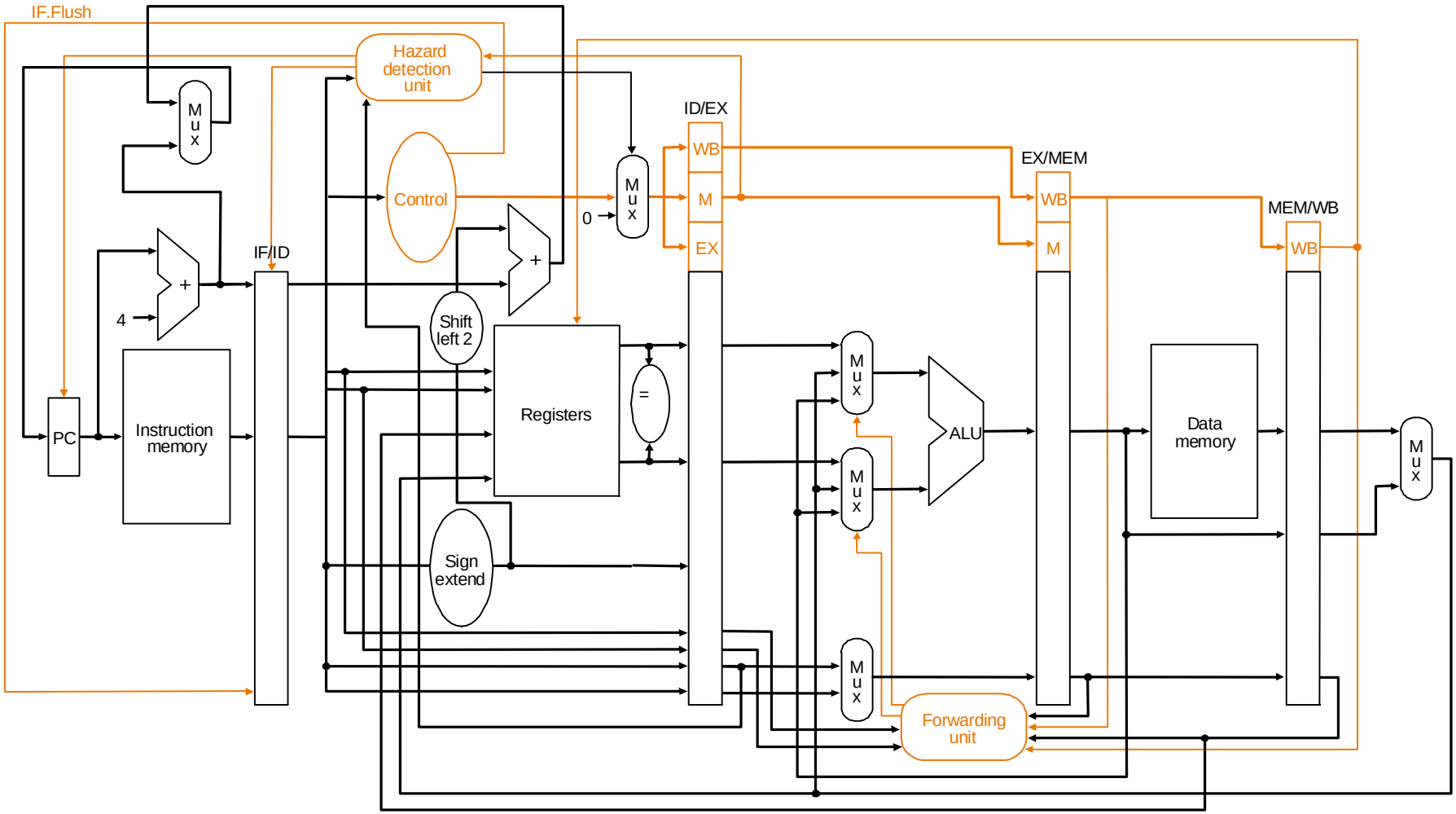
Cycle 3


IF.Flush
Hazard detection unit
M u x
Control
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
+
4
Shift left 2
Registers
=
PC
Instruction memory
Sign extend
ALU
Data memory
Forwarding unit

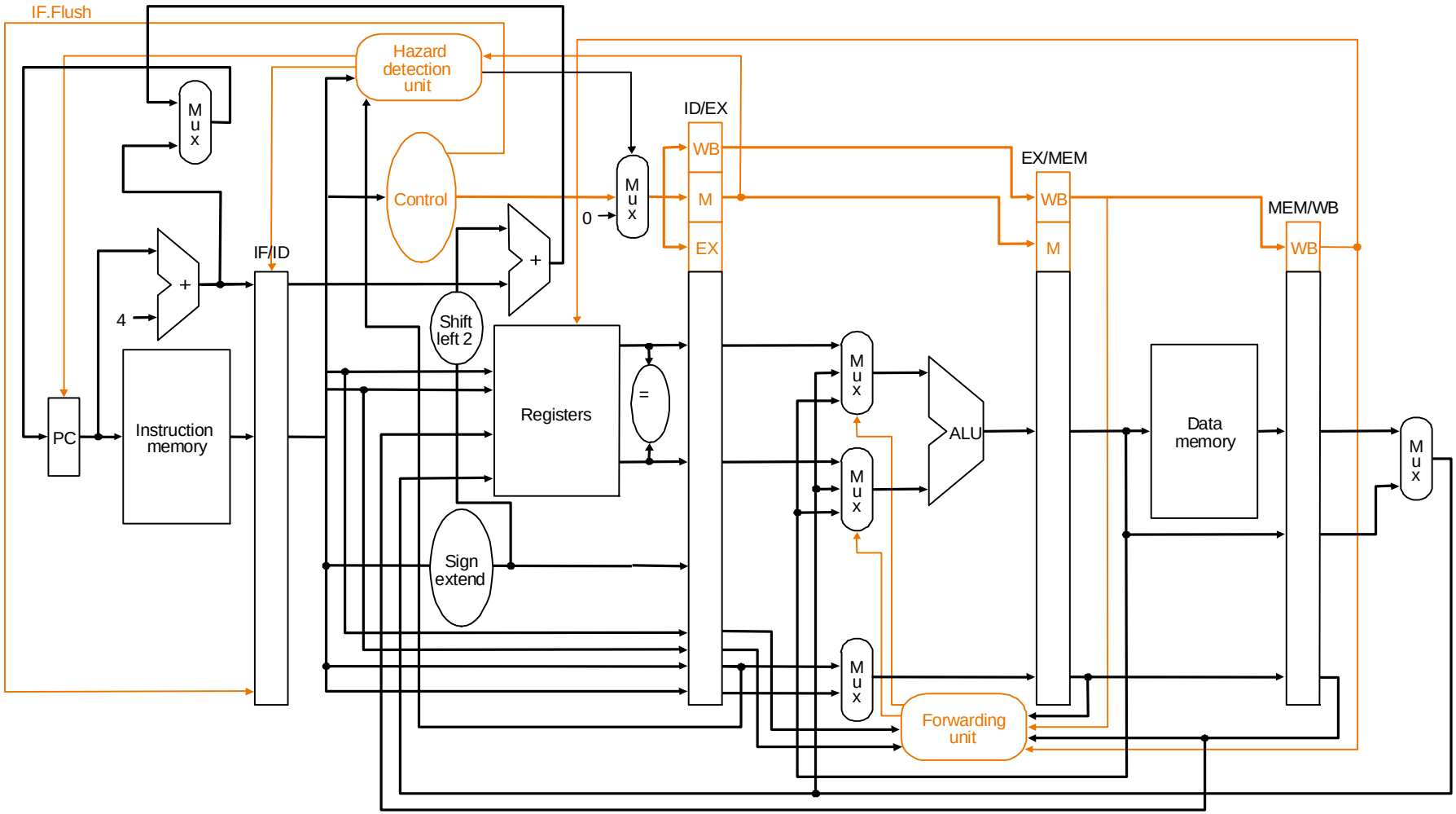
Cycle 4


IF.Flush
Hazard detection unit
M u x
Control
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
+
4
Shift left 2
Registers
=
PC
Instruction memory
Sign extend
ALU
Data memory
Forwarding unit

Cycle 5


IF.Flush
Hazard detection unit
M u x
Control
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
+
4
Shift left 2
Registers
=
PC
Instruction memory
Sign extend
ALU
Data memory
Forwarding unit

Cycle 6


IF.Flush
Hazard detection unit
M u x
Control
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
+
4
Shift left 2
Registers
=
PC
Instruction memory
Sign extend
ALU
Data memory
Forwarding unit

Performance Analysis-- Individual CPI
Basic Pipeline

Instruction	R	lw	sw
CPI/no dependence			
CPI/with dependence*			

Instruction	beq	jump
CPI (branch is not taken)		
CPI (branch is taken)		

Performance Analysis-- Individual CPI
Advanced Pipeline

Instruction	R	lw	sw
CPI/no dependence			
CPI/with dependence*			

Instruction	beq	jump
CPI (branch is not taken)		
CPI (branch is taken)		

Calculating CPI of our Pipeline

	load	store	R-type	branch	jumps
gcc instruction mix	22%	11%	49%	16%	2%

- CPI when there is no any hazard in the pipeline
- Assume: half of `lw` are immediately followed by a dependent instruction; 75% of branches are taken; jump is treated as a taken branch inst, 30% R type instructions have dependence with their following instructions.

- CPI of the basic pipeline

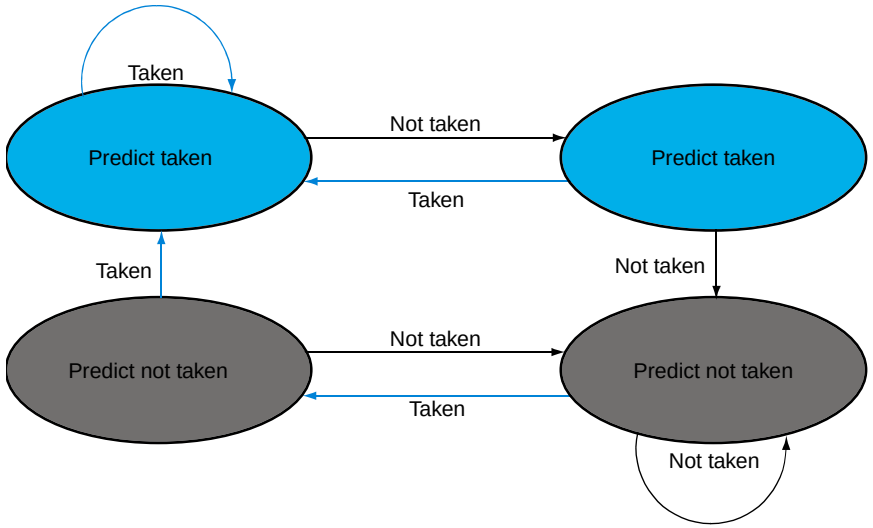
- CPI of the advanced pipeline

Dynamic Branch Prediction

- Basic idea: look at if the branch instruction takes the branch last time it was executed. If so, fetching instructions from same place as last time.
- 1-bit prediction: a 1-bit prediction buffer+ branch target buffer
  - if the prediction=1, take branch; other wise do not taken branch; if made mistake, reverse the prediction bit.
  - Shortcoming: predict incorrectly twice for a loop.
- 2 bit prediction

Branches

- If the branch is taken, we have a penalty of one cycle
- For our simple design, this is reasonable
- With deeper pipelines, penalty increases and static branch prediction drastically hurts performance
- Solution: dynamic branch prediction



A 2-bit prediction scheme

Branch Prediction

- Sophisticated Techniques:
  - A “branch target buffer” to help us look up the destination
  - Correlating predictors that base prediction on global behavior and recently executed branches (e.g., prediction for a specific branch instruction based on what happened in previous branches)
  - Tournament predictors that use different types of prediction strategies and keep track of which one is performing best.
  - A “branch delay slot” which the compiler tries to fill with a useful instruction (make the one cycle delay part of the ISA)
- Branch prediction is especially important because it enables other more advanced pipelining techniques to be effective!
- Modern processors predict correctly 95% of the time!

Improving Performance

- Dynamic Pipeline Scheduling
  - Hardware chooses which instructions to execute next
  - Will execute instructions out of order (e.g., doesn't wait for a dependency to be resolved, but rather keeps going!)
  - Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)
- Trying to exploit instruction-level parallelism

Advanced Pipelining

- Increase the depth of the pipeline
- Start more than one instruction each cycle (multiple issue)
- Loop unrolling to expose more ILP (better scheduling)
- “Superscalar” processors
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
- All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., different “pipes”)
- VLIW: very long instruction word, static multiple issue (relies more on compiler technology)

Example

- Considering the following loop

```
                mov $5, $0
SUM:            lw $10, 1000($20)
                add $5, $5, $10
                addi $20, $20, -4
                bne $20, $0, SUM
```

- Assume a pipeline without forwarding and hazard detection
- Question (1) how to reorder inst and insert as few as needed *nop* to eliminate hazards?

  (2) find #cycles = f (N: #words added)

Concluding Remarks

- Big picture
  - Pipelines improve inst. throughput but not the inst. latency
  - Benefit of pipeline is limited by
    - data and control dependencies & inst. latencies
  - The upper limit can be raised (not eliminated) by reducing data and control hazards
- Compiler writers must understand pipeline
  - compiler techniques to reduce data and control hazards