
The Laundry Analogy

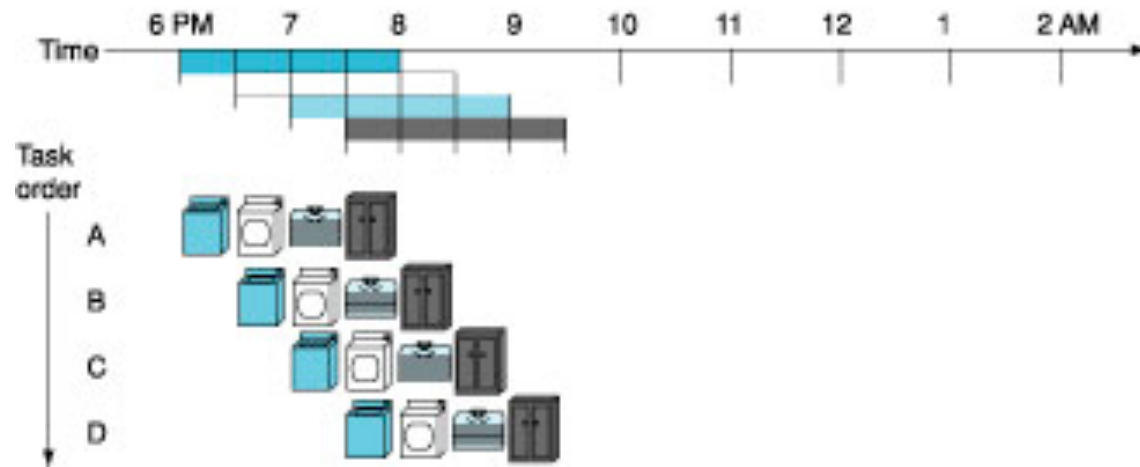
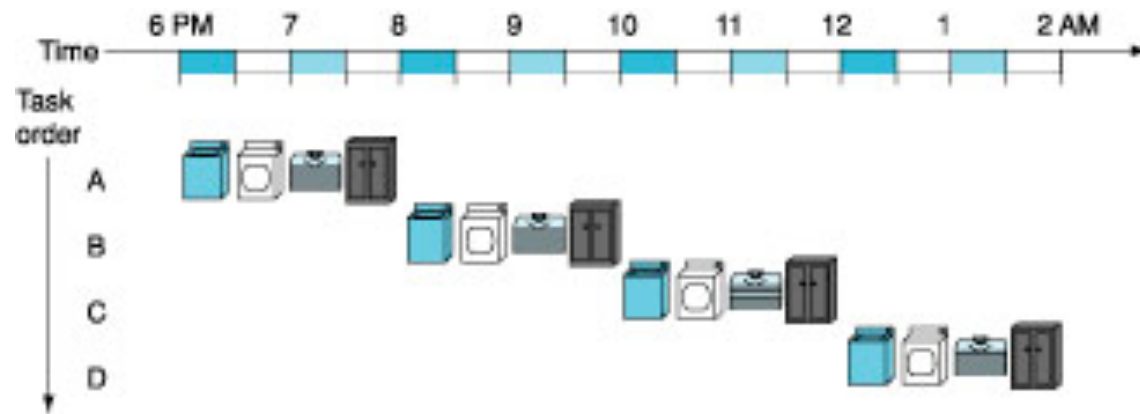
Non-pipelined approach:

1. run 1 load of clothes through washer
2. run load through dryer
3. fold the clothes
4. put the clothes away

Two loads? Start all over.

Pipelined Laundry

- While the first load is drying, put the second load in the washing machine.
- When the first load is being folded and the second load is in the dryer, put the third load in the washing machine.



Laundry Performance

- For 4 loads:
 - non-pipelined approach takes 16 units of time.
 - pipelined approach takes 7 units of time.
- For 816 loads:
 - non-pipelined approach takes 3264 units of time.
 - pipelined approach takes 819 units of time.

Execution Time vs. Throughput

- It still takes the same amount of time to get your favorite pair of socks clean, pipelining won't help.
- However, the total time spent away from studying for COE3DR4 is reduced 😊

Pipelining

- We can overlap the execution of multiple instructions.
- At any time, there are multiple instructions being executed – each in a different stage.

Instruction Pipelining

First we need to break instruction execution into stages:

1. Instruction Fetch
2. Instruction Decode/ Register Fetch
3. ALU Operation
4. Data Memory access
5. Write result into register

Operation Timings

- Some estimated timings for each of the stages:

Register Write 100ps

Register Read 100ps

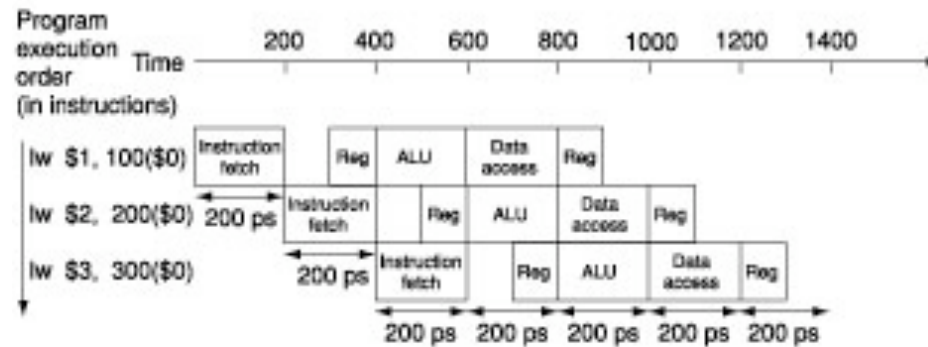
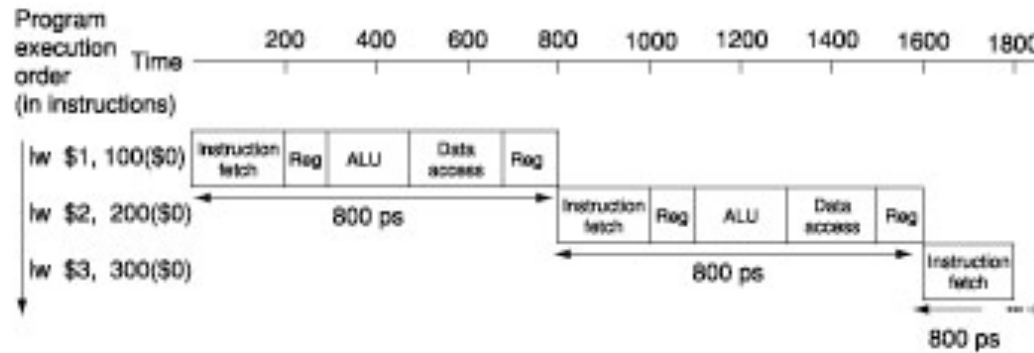
ALU Operation 200ps

Memory access 200ps

Operation Timings

•	Instruction class	Instruction fetch	Register read	ALU operation	Data	Register memory	Total	write
•	Load word	200	100	200	200	100	800ps	
•	Store word	200	100	200	200		700ps	
•	R type	200	100	200		100	600ps	
•	Branch	200	100	200			500ps	

Comparison

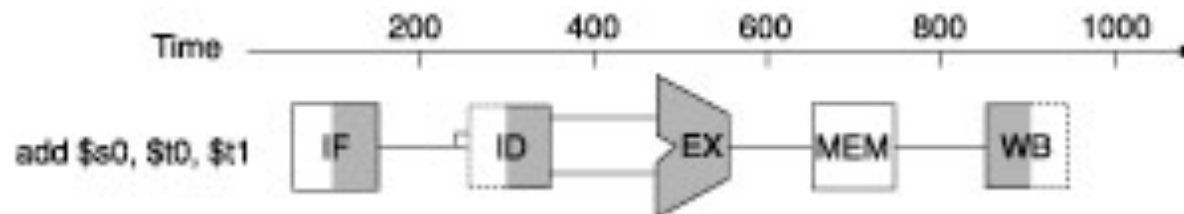


MIPS and Pipelining

- MIPS design features that make pipelining easy include:
 - single length instruction (always 1 word)
 - Makes it easier to fetch instructions in the first pipeline stage and decode them in the second one
 - relatively few instruction formats
 - Second stage can begin reading register file at the same time as the hardware is determining type of the instruction
 - Memory operands only appear in load/store instruction set
 - We can calculate memory address and access memory in the following stage
 - operands must be aligned in memory (a single data transfer instruction requires a single memory operation).
 - We need not worry about a single data transfer instruction requiring two data memory access

Task-time diagram

- IF: instruction fetch (box representing instruction memory)
- ID: instruction decode/register file read stage (box representing register file)
- EX: execution stage (drawing represents the ALU)
- MEM: memory access stage (box representing data memory)
- WB: write back stage (drawing represents the register file)
- Shading means the element is used
- Shading of left half means writing and shading of right half means reading



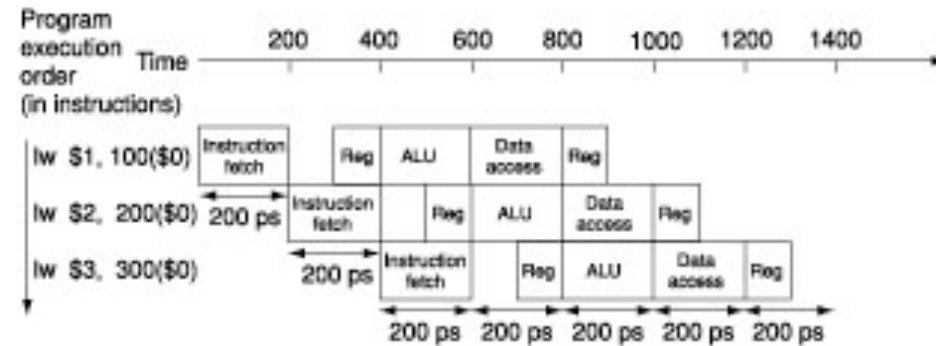
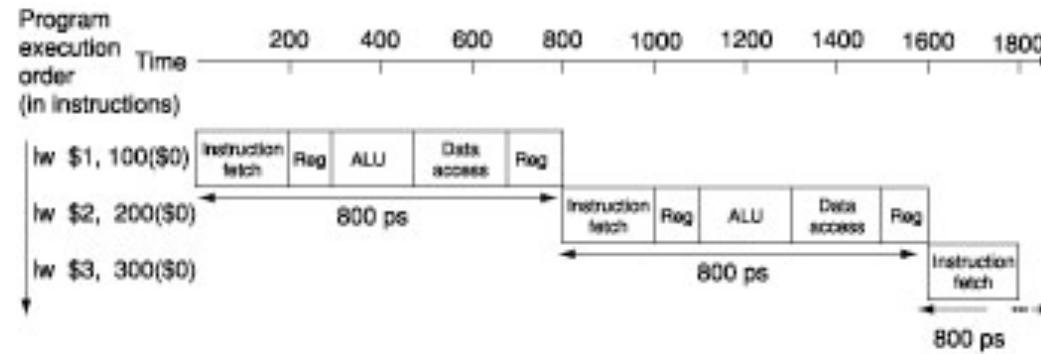
Pipeline Hazard

- Something happens that means the next instruction cannot execute in the following clock cycle.
- Three kinds of hazards:
 - structural hazard
 - control hazard
 - data hazard

Structural Hazards

- Two stages require the same resource.
 - What if we only had enough electricity to run either the washer or the dryer at any given time?
 - What if MIPS datapath had only one memory unit instead of separate instruction and data memory?

Structural Hazards



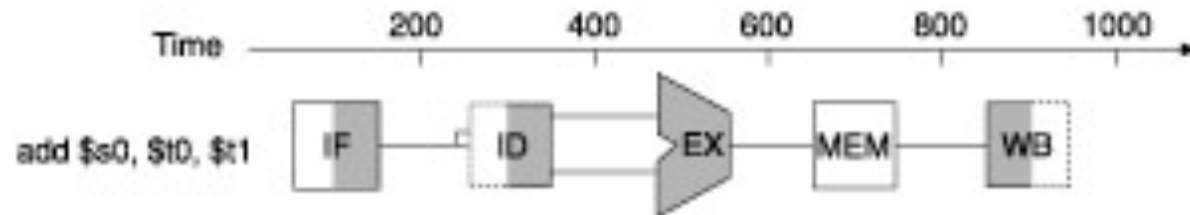
Avoiding Structural Hazards

- Design the pipeline carefully.
- Might need to duplicate resources
- Detecting structural hazards at execution time (and delaying execution) is not something we want to do (structural hazards are minimized in the design phase).

Data Hazard

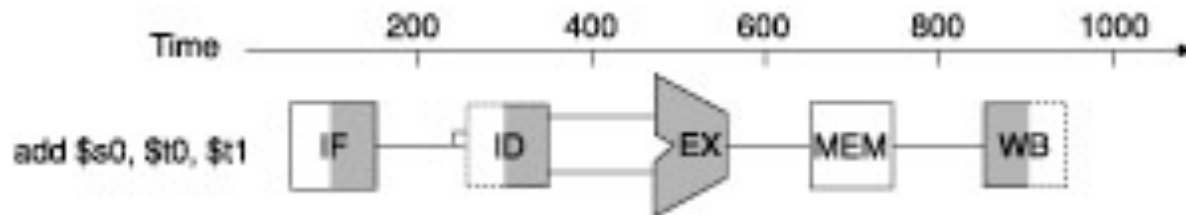
- One of the values needed by an instruction is not yet available (the instruction that computes it isn't done yet).
- This will cause a data hazard:

```
add $s0, $t0, $t1  
sub $t2, $s0, $t3
```



Data Hazard

- The add instruction doesn't write its result until the fifth stage.
- We would have to stall (add three bubbles to) the pipeline
- **Pipeline stall** (also called bubble): A stall initiated in order to resolve a hazard
- Bubbles (harmless instructions doing nothing) or no-op codes are added by the compiler



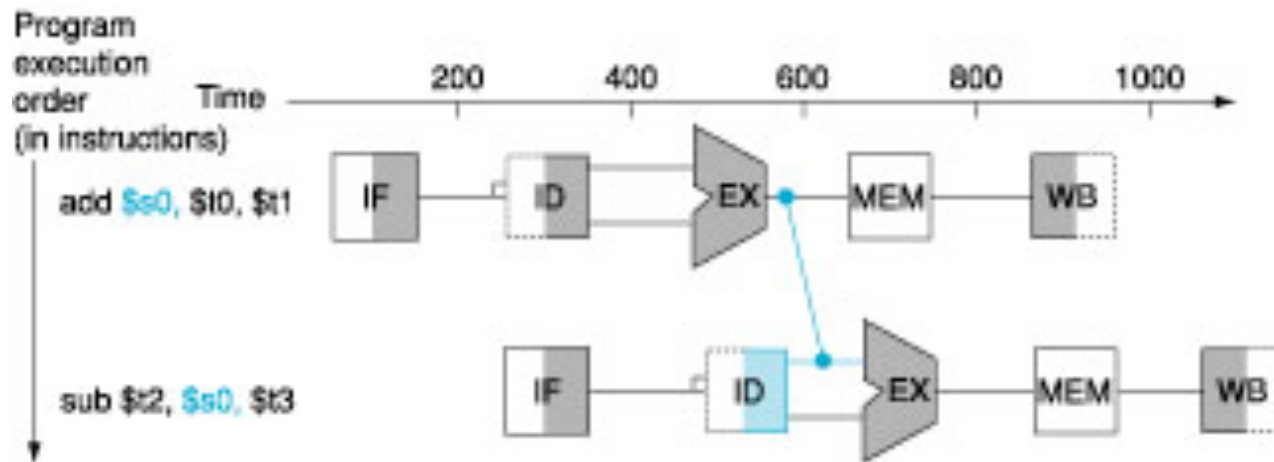
Handling Data Hazards

- We can hope that the compiler can arrange instructions so that data hazards never appear.
- Some data hazards aren't real - the value needed is available, just not in the right place.

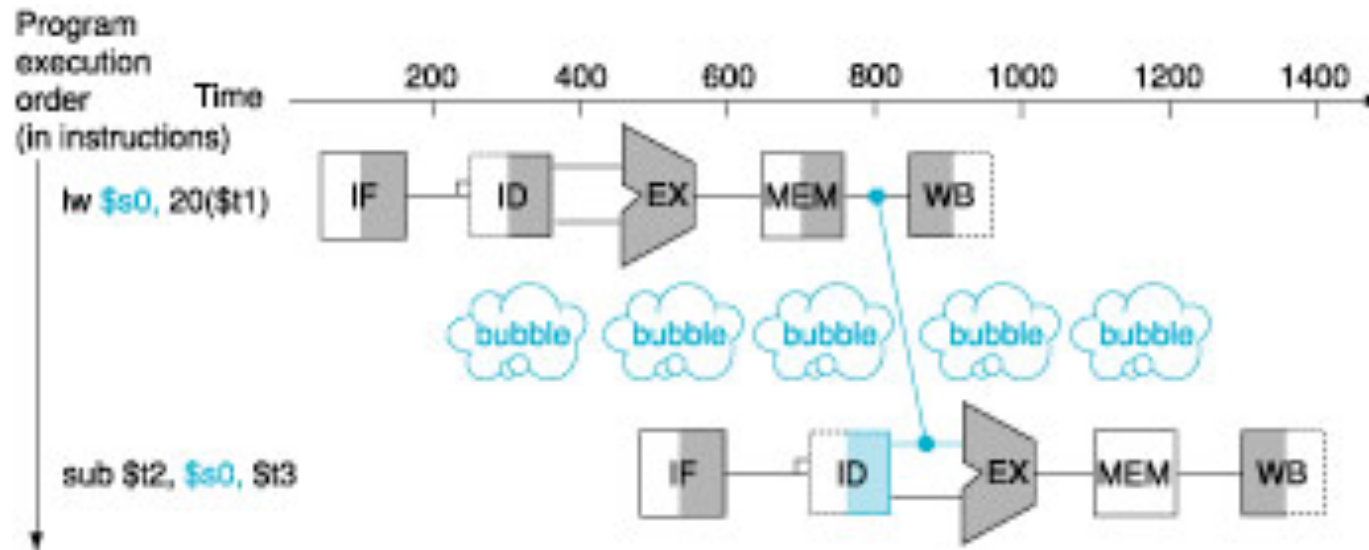
Forwarding

- It's possible to *forward* the value directly from one resource to another (in time).
- Forwarding or bypassing: a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible register or memory
- Hardware needs to detect (and handle) these situations automatically!
 - This is difficult, but necessary.

Picture of Forwarding



Another Example

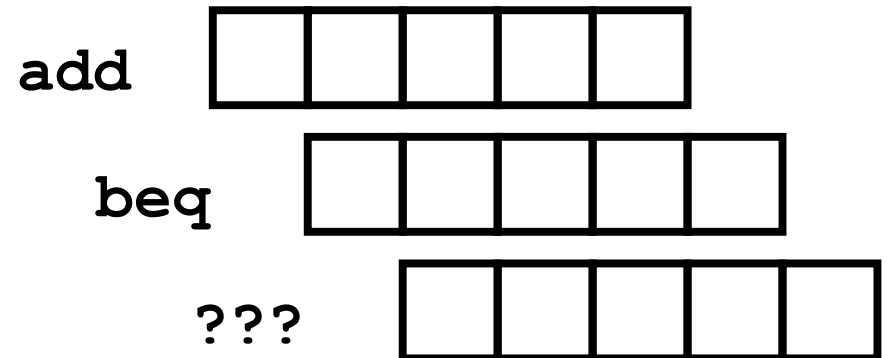


Control Hazards

- When a decision has to be made based on the results of an instruction that has not yet finished.
- Example: conditional branch
 - The instruction that is fed to the pipeline right after a **beq** depends on whether or not the branch is taken.

beq Control Hazard

```
    add $4, $5, $6  
    beq $1, $2, 40  
    lw  $3, 300($0)  
40:  or  $7, $8, $9
```



The instruction to follow the **beq** could be either the **lw** or the **or**, it depends on the result of the **beq** instruction.

One possible solution - stall

- We can include in the control unit the ability to *stall* (to keep new instructions from entering the pipeline) until we know which one.
- Unfortunately conditional branches are very common operations, and this would slow things down considerably.

Another strategy

- Predict whether or not the branch will be taken.
- Go ahead with the *predicted* instruction (feed it into the pipeline next).
- If your prediction is right, you don't lose any time.
- If your prediction is wrong, you need to undo some things and start the correct instruction

Dynamic Branch Prediction

- The idea is to build hardware that will come up with a prediction based on the past history of the specific branch instruction.
- Predict the branch will be taken if it has been taken more often than not in the recent past.
 - This works great for loops! (90% + correct).

Pipeline Datapath

- Now we'll see a basic implementation of a pipelined processor.
- The datapath and control unit share similarities with both the single cycle and multicycle implementations that we already saw.
- An example execution highlights important pipelining concepts.
- In future lectures, we'll discuss several complications of pipelining that we're hiding from you for now.

Pipeline Datapath

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath.
- This increases throughput, so programs can run faster.
- One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.

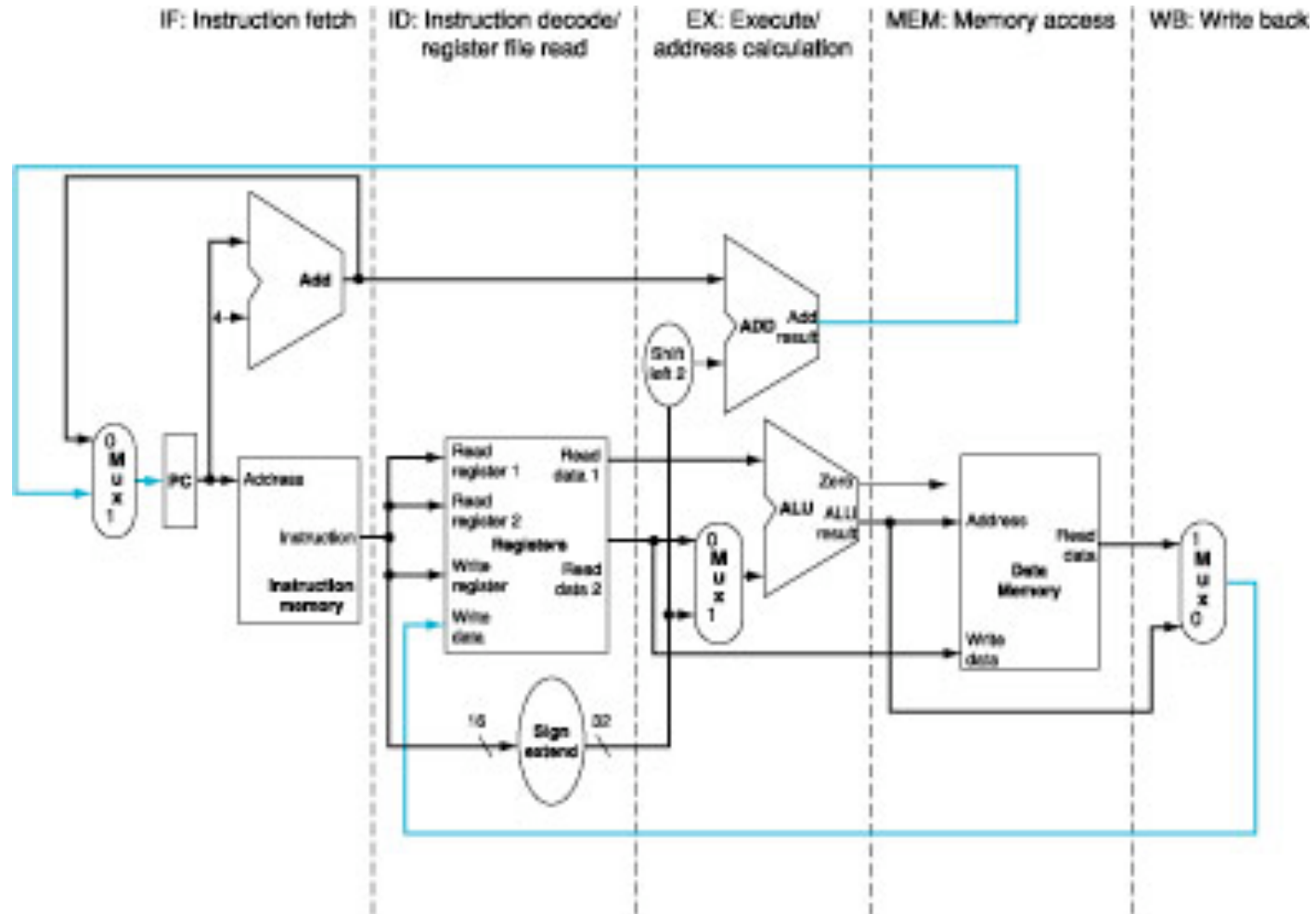
Pipeline Datapath

- The whole point of pipelining is to allow multiple instructions to execute at the same time.
- We may need to perform several operations in the same cycle.
 - Increment the PC and add registers at the same time.
 - Fetch one instruction while another one reads or writes data.
- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.

Pipeline Datapath

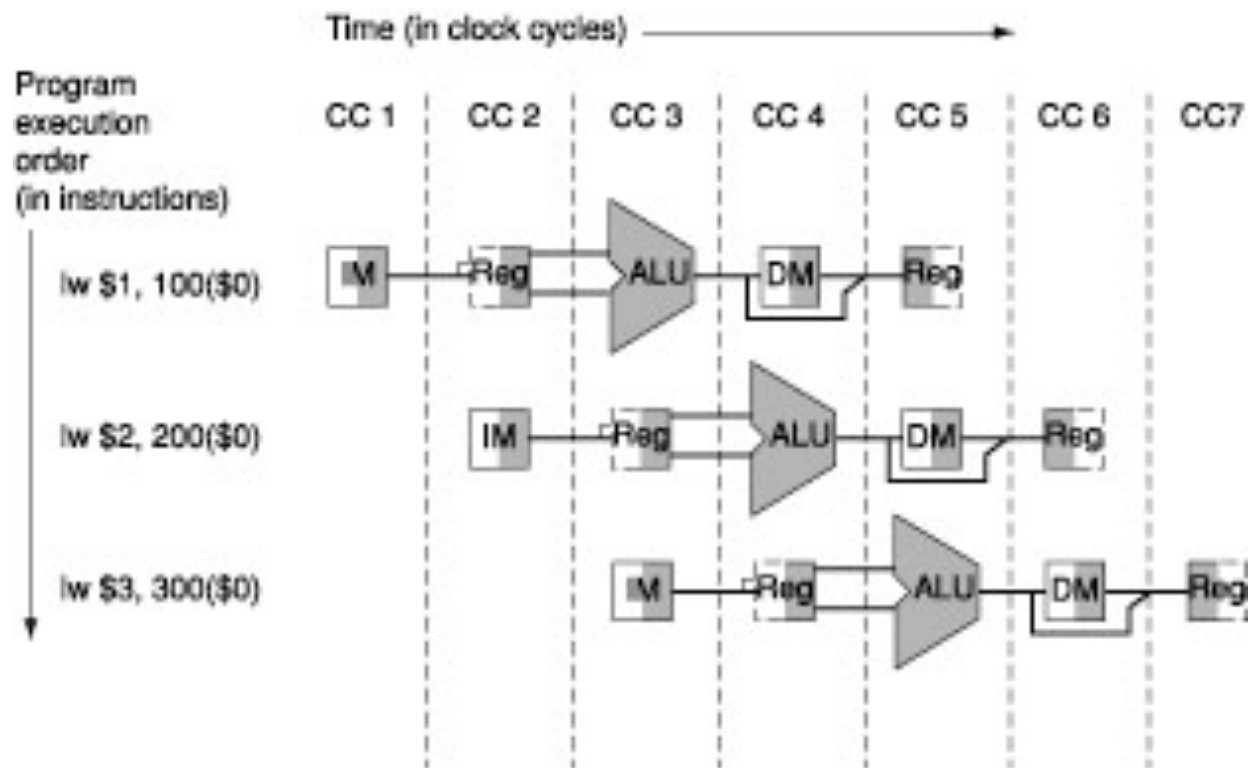
- Division of an instruction into five stages means a five-stage pipeline
- Up to five instructions will be in execution during any single cycle
- We must separate the datapath to five pieces:
 1. IF: instruction fetch
 2. ID: instruction decode and register file read
 3. EX: execution or address calculation
 4. MEM: data memory access
 5. WB: Write back

Pipeline Datapath



Pipeline Datapath

- Instructions and data generally move from left to right except for the write back stage, which places the result back into the register file, and selection of next PC value.
- These exceptions relate to potential data hazards (write back) and control hazards (PC selection).
- One way to understand what happens in a pipelined system is to place the instructions on a timeline.



Pipeline Datapath

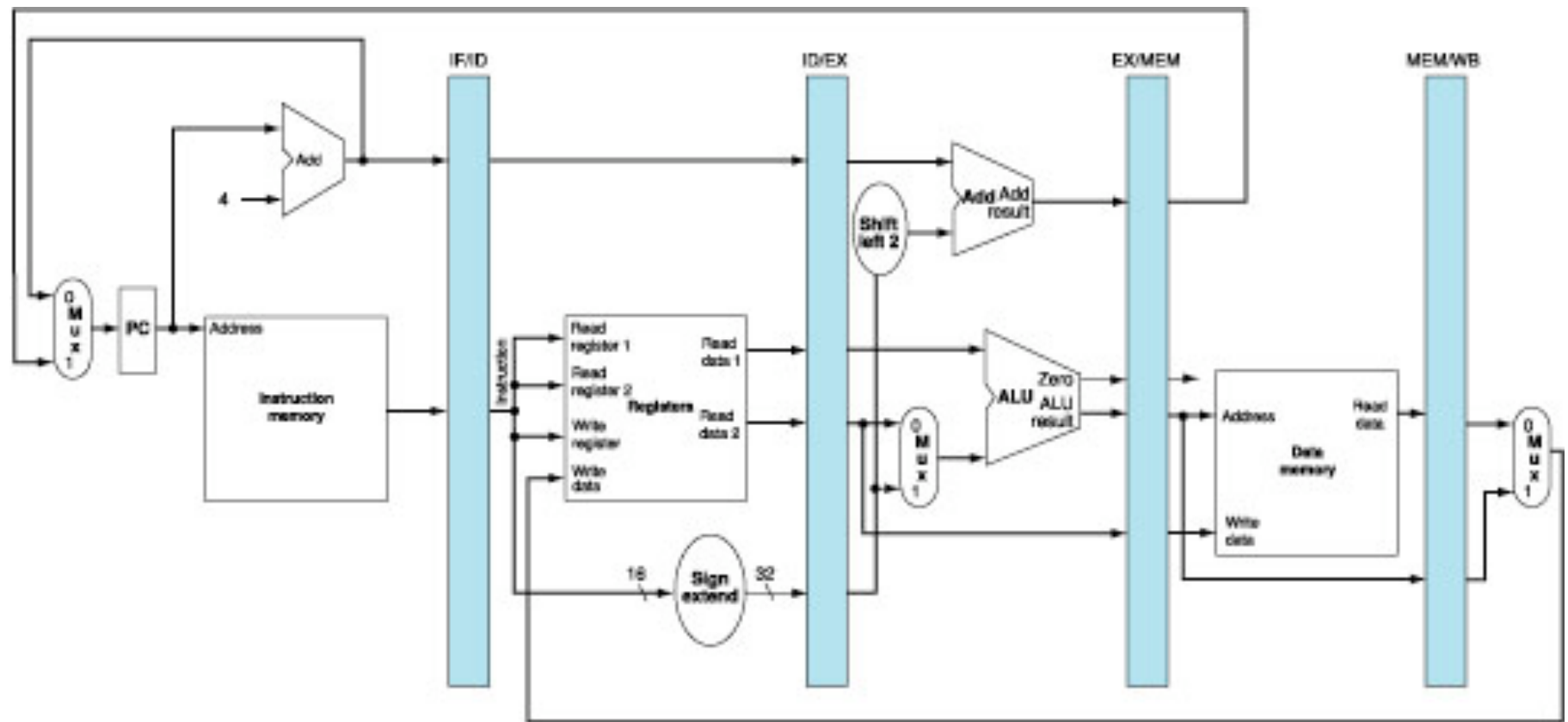
- Since each instruction doesn't really have a separate datapath, registers are added in between the pipeline stages.
- Any information needed in a later pipe stage **MUST** be passed to that stage via a pipeline register.
- All instructions must ultimately update some state in the machine (register file, memory, PC), so there is no pipeline register at the end of the sequence.
- Note that each logical component can only be used once within a single pipeline stage.

Pipeline Datapath

- In pipelining, we divide instruction execution into multiple cycles.
- Information computed during one cycle may be needed in a later cycle.
- The instruction read in the IF stage determines which registers are fetched in the ID stage, what constant is used for the EX stage, and what the destination register is for WB.
- The registers read in ID are used in the EX and/or MEM stages.
- The ALU output produced in the EX stage is an effective address for the MEM stage or a result for the WB stage.

Pipeline Datapath

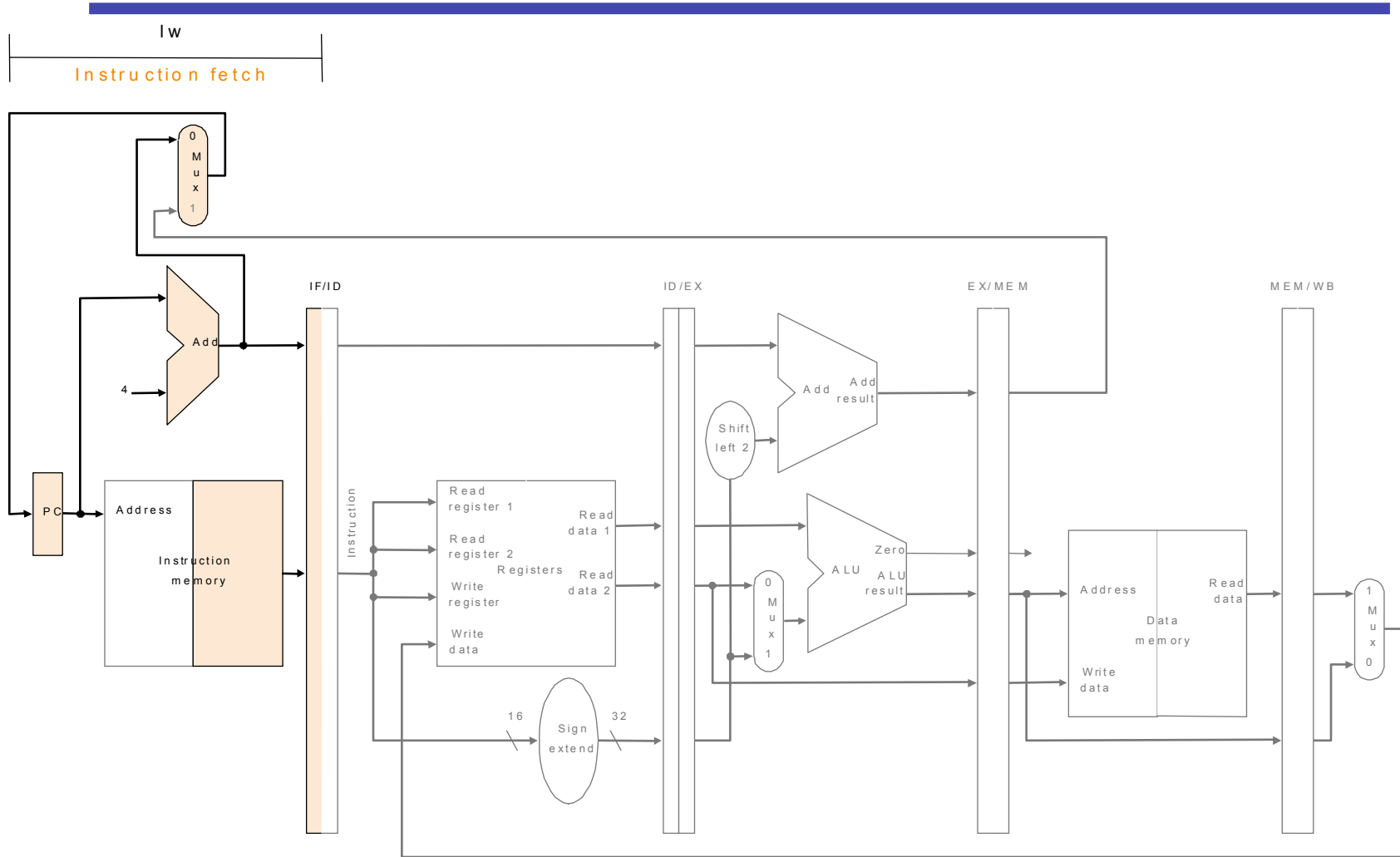
- We'll add intermediate registers to our pipelined datapath.
- The registers are named for the stages they connect: IF/ID ID/EX EX/MEM MEM/WB
- No register is needed after the WB stage, because after WB the instruction is done.



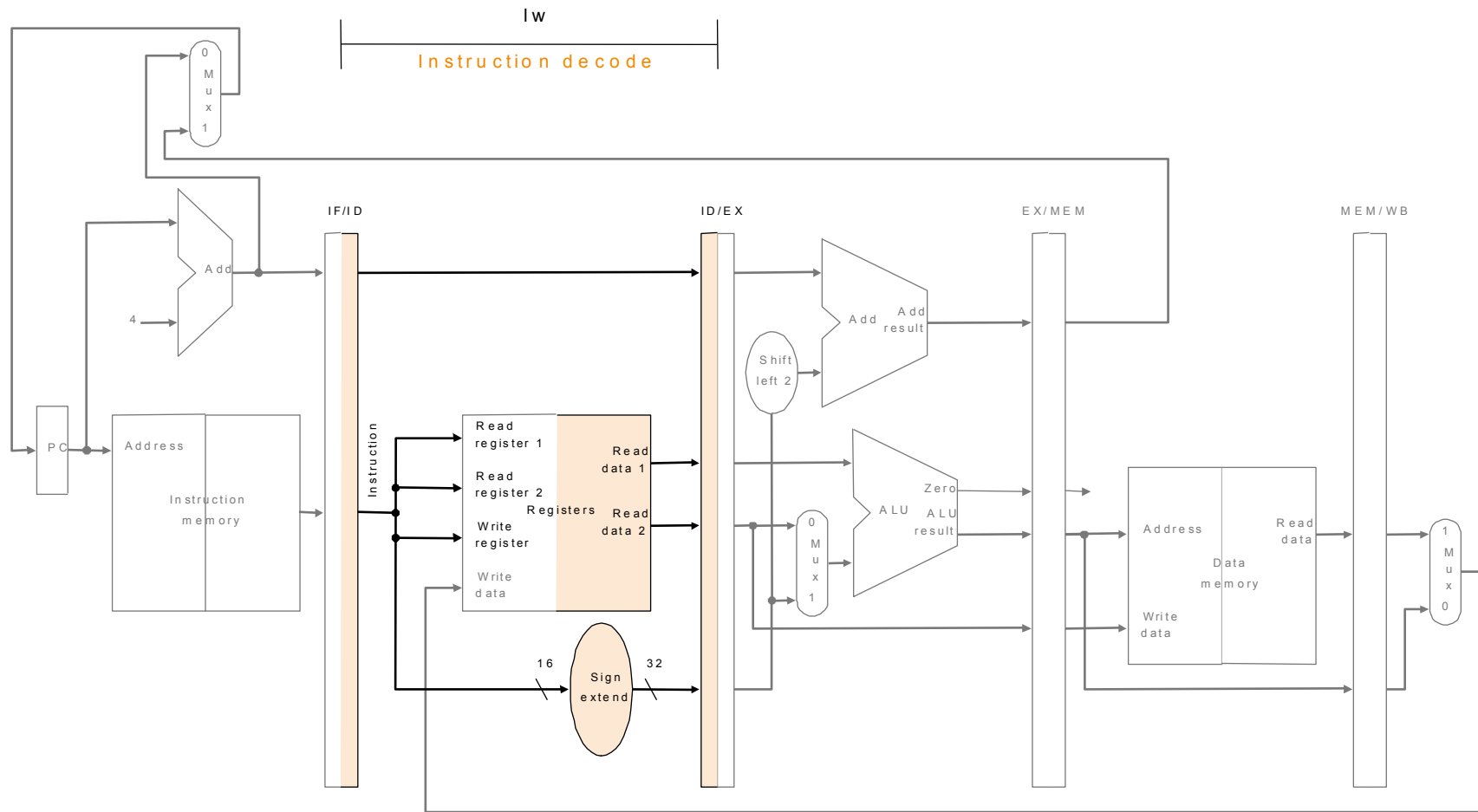
-
- Any data values required in later stages must be propagated through the pipeline registers.
 - The most extreme example is the destination register.
 - The rd field of the instruction word, retrieved in the first stage (IF), determines the destination register. But that register isn't updated until the *fifth* stage (WB).
 - Thus, the rd field must be passed through all of the pipeline stages, as shown in red on the next slide.
 - Notice that we can't keep a single "instruction register" like we did before in the multicycle datapath, because the pipelined machine needs to fetch a new instruction every clock cycle.

lw and pipelined datapath

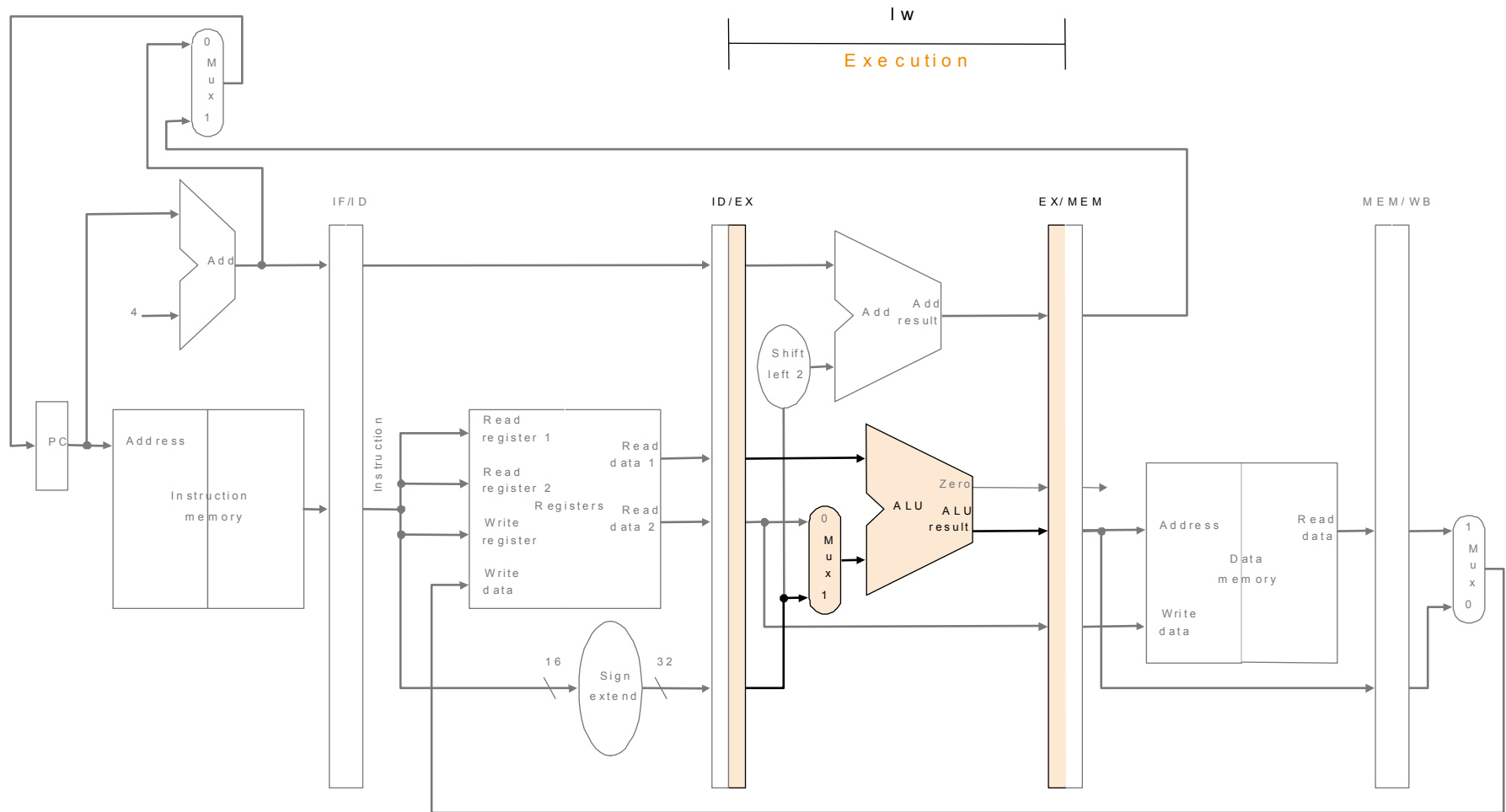
- We can trace the execution of a load word instruction through the datapath.
- We need to keep in mind that other instructions are using the stages not in use by our **lw** instruction!



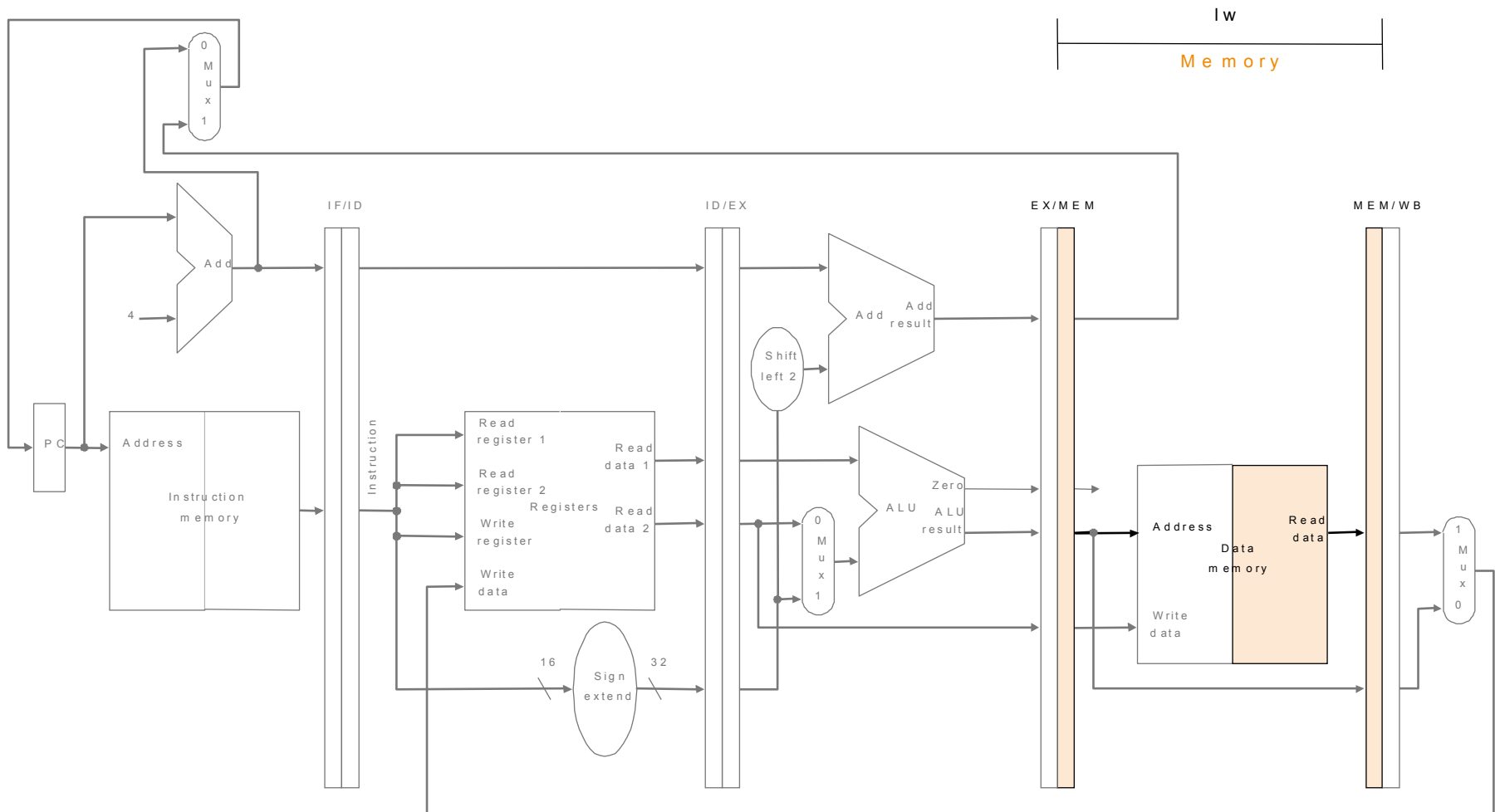
-
- Instruction is read from memory using PC and place in the IF/ID register
 - PC is incremented by 4 and written back to PC and to IF/ID (it might be needed later for an instruction such as beq)



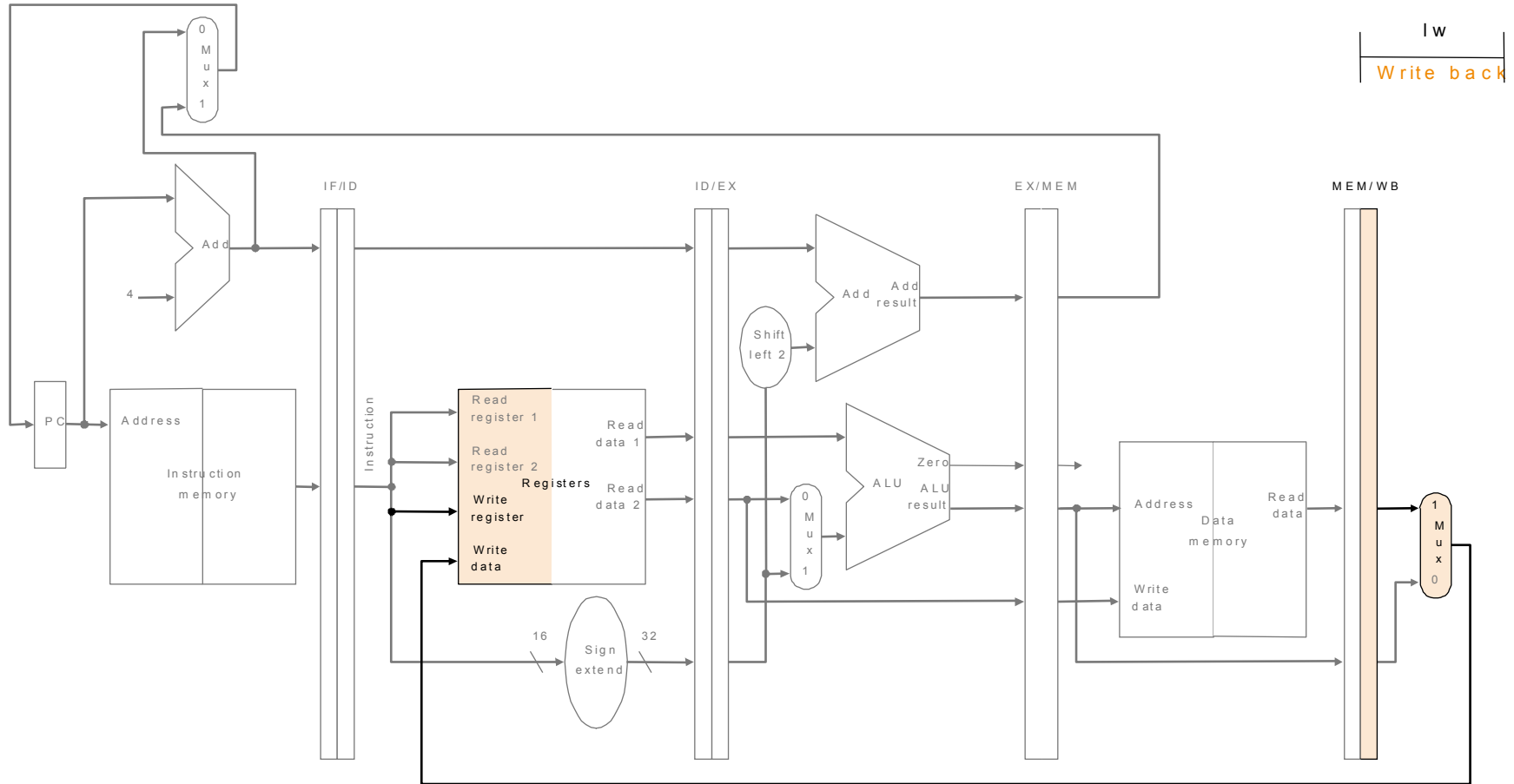
-
- Instruction portion of IF/ID supplies the 16-bit immediate which is sign extended to 32 bits and the register numbers to read the two registers
 - All three values and incremented PC are stored in ID/EX



-
- Load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU
 - The sum is placed in EX/MEM



-
- The load instruction reads the data memory using the address from the EX/MEM register and loads the data into the MEM/WB pipeline register

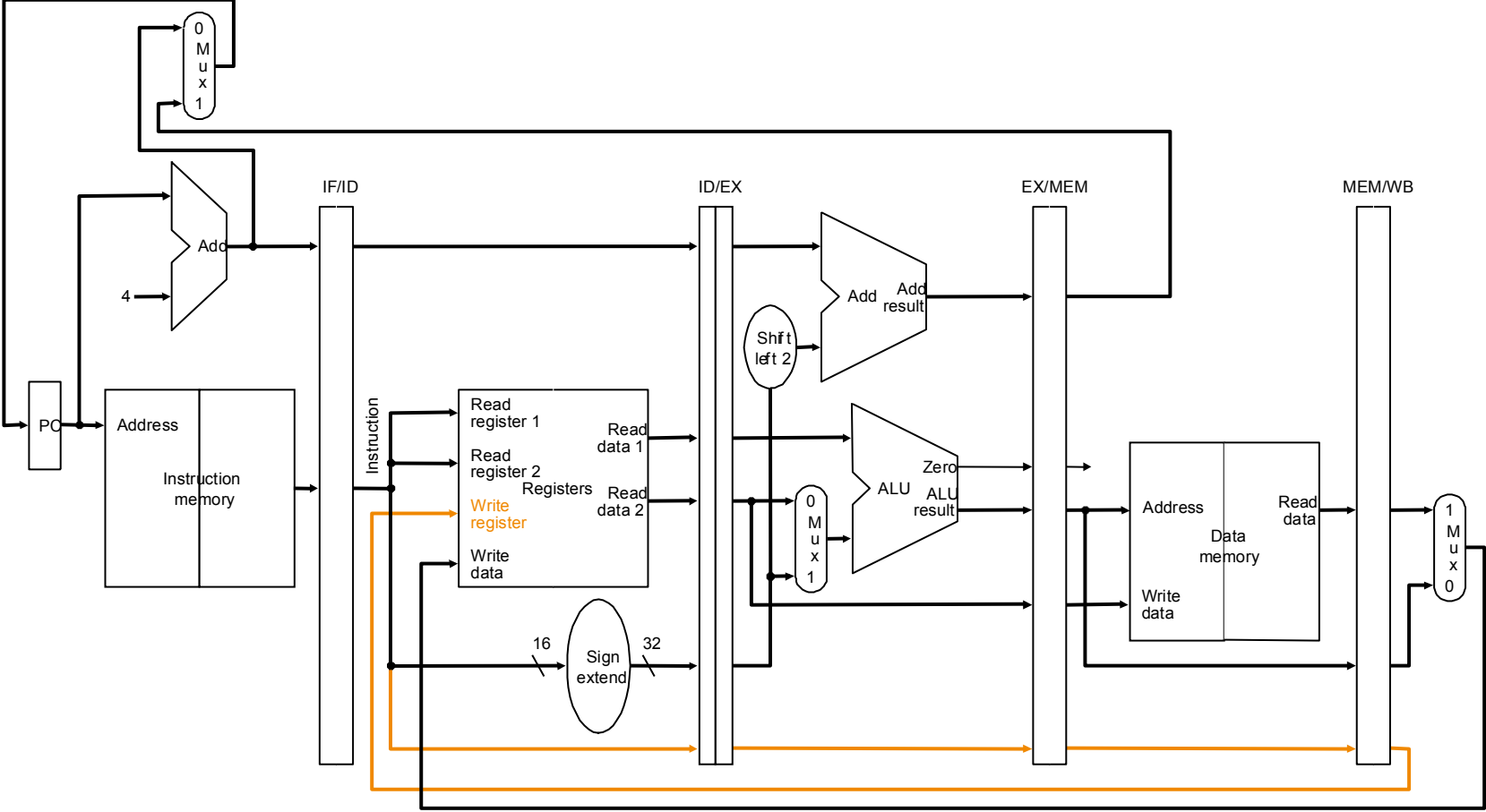


Iw
Write back

A Bug!

- When the value read from memory is written back to the register file, the inputs to the register file (write register #) are from a different instruction!
- To fix the bug we need to save the part of the **lw** instruction (5 bits of it specify which register should get the value from memory).

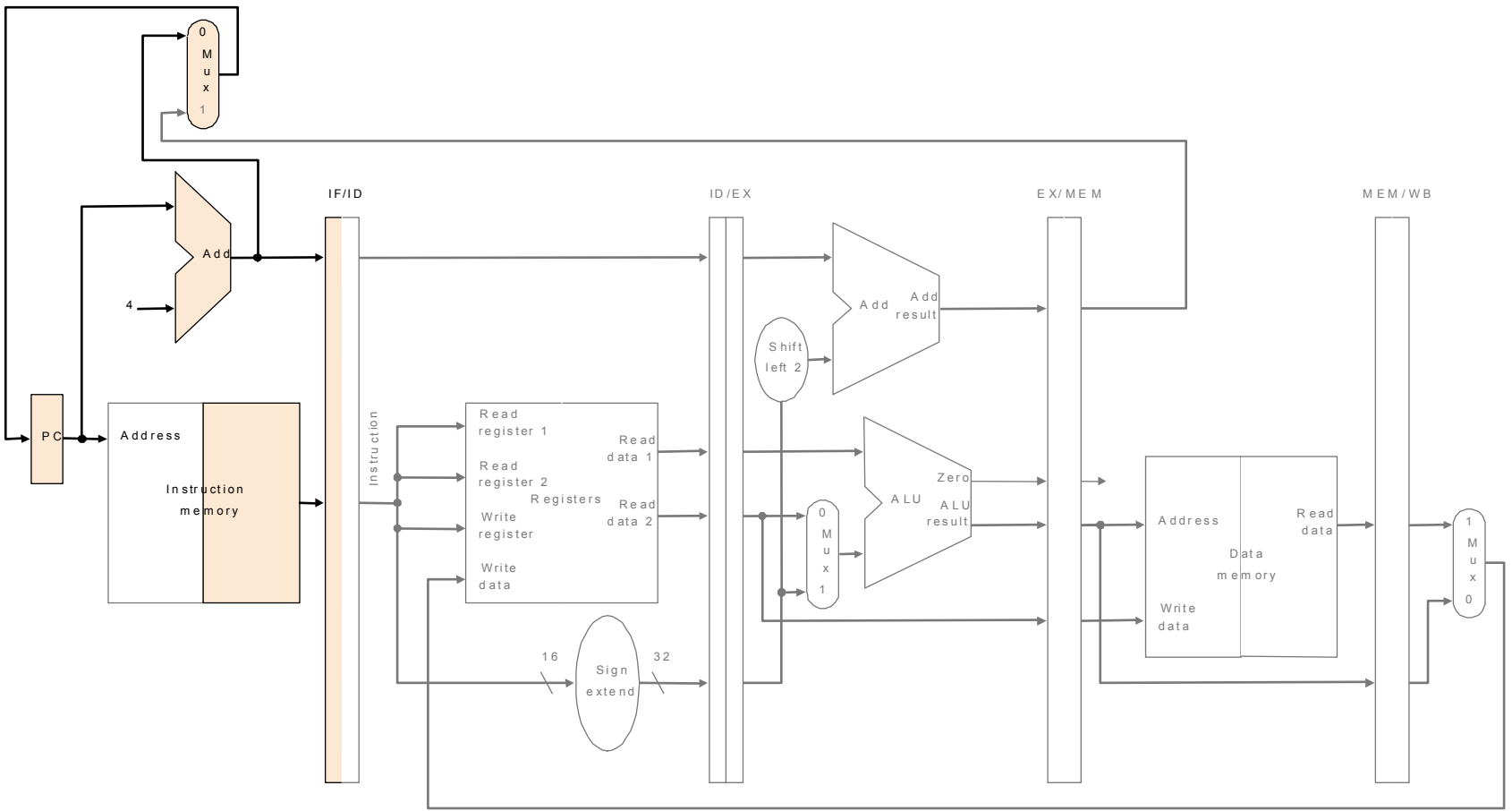
New Datapath

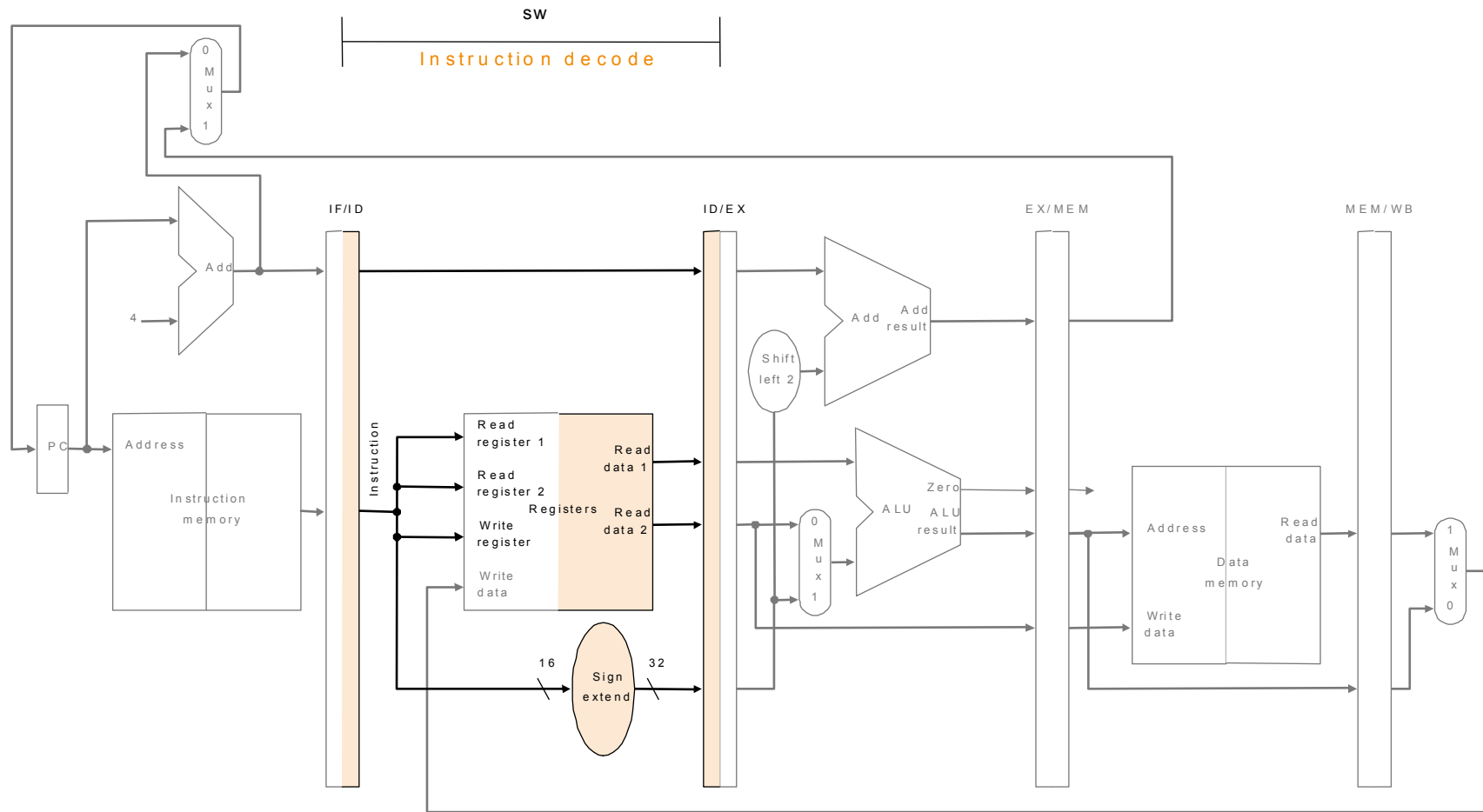


sw and pipelined datapath

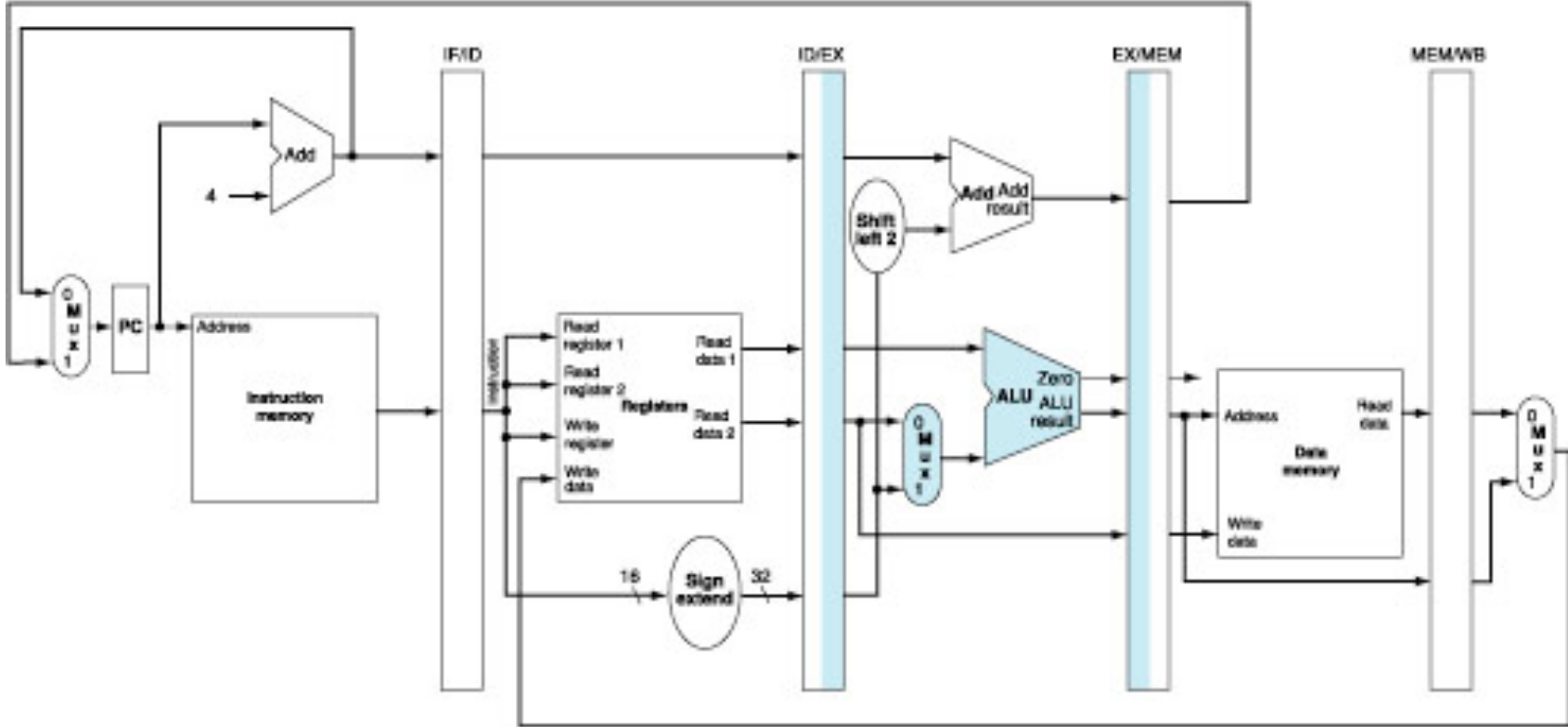
- The first two stages are the same.
- In third stage the effective address is place in EX/MEM
- In the forth stage data is written into memory. Register containing the data was read in an earlier stage and stored in ID/EX
 - To make this data available during MEM stage it is placed in EX/MEM register
- Nothing happens in this stage
- In a pipeline an instruction passes through a stage even if there is nothing to do

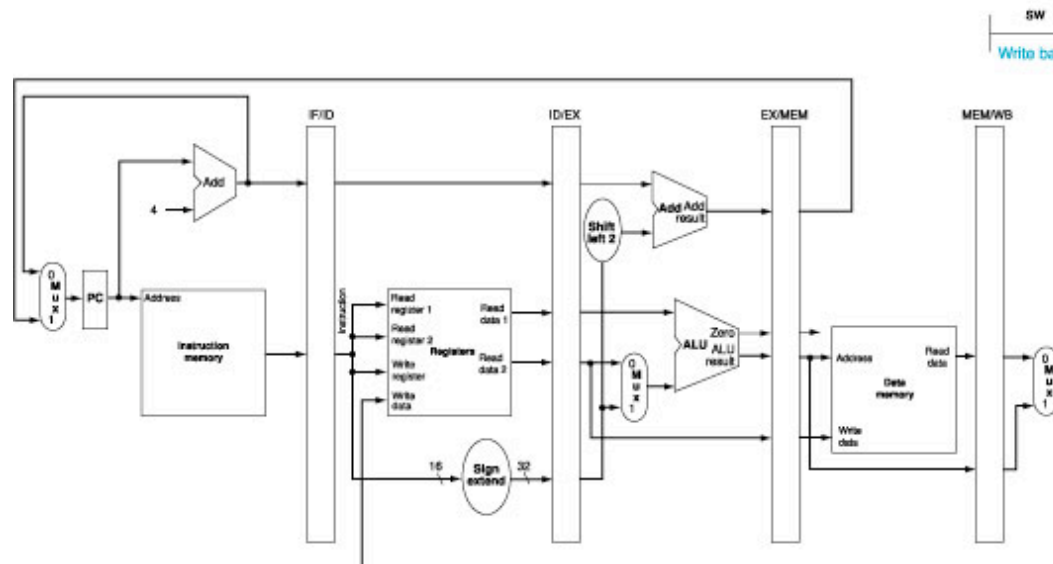
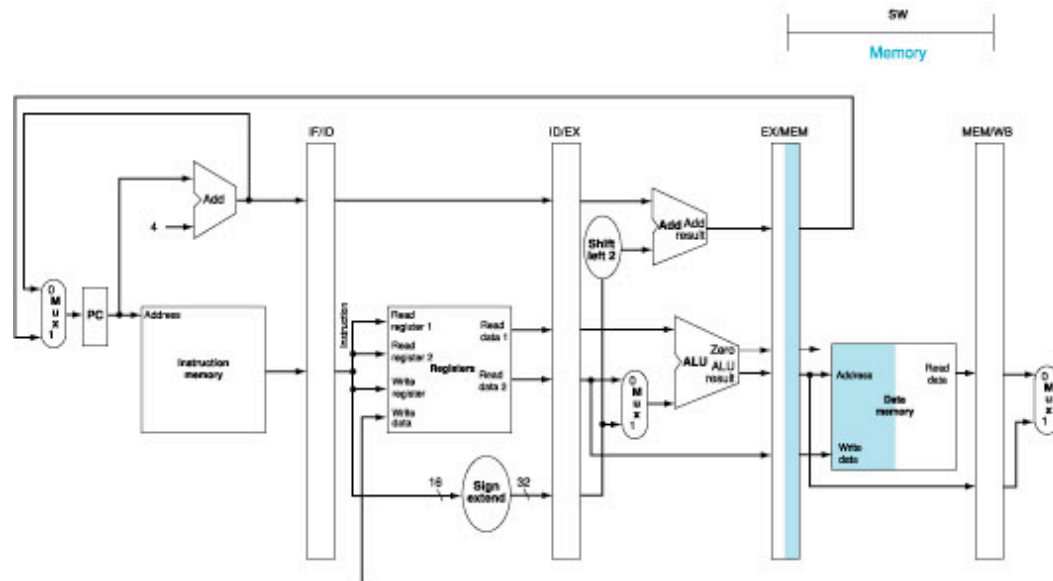
SW
Instruction fetch

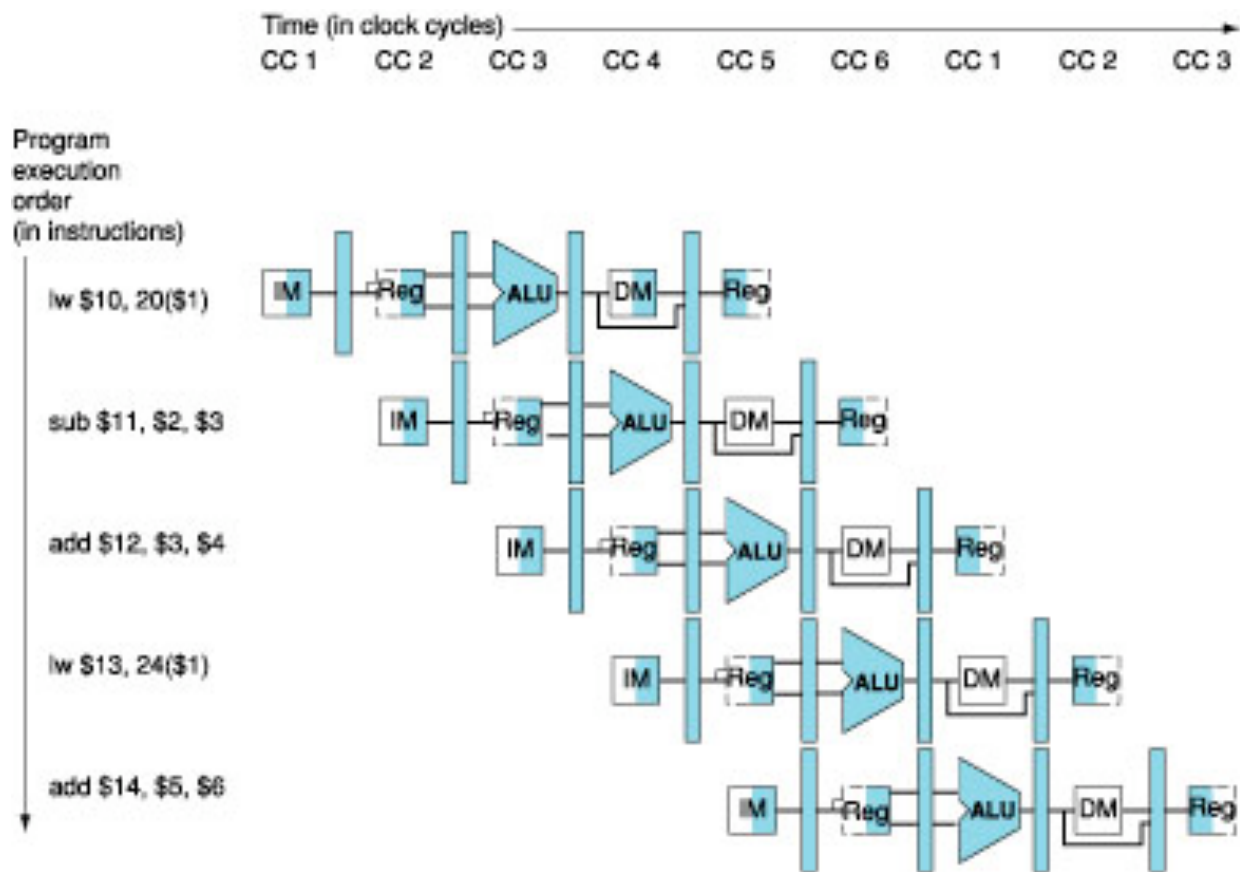


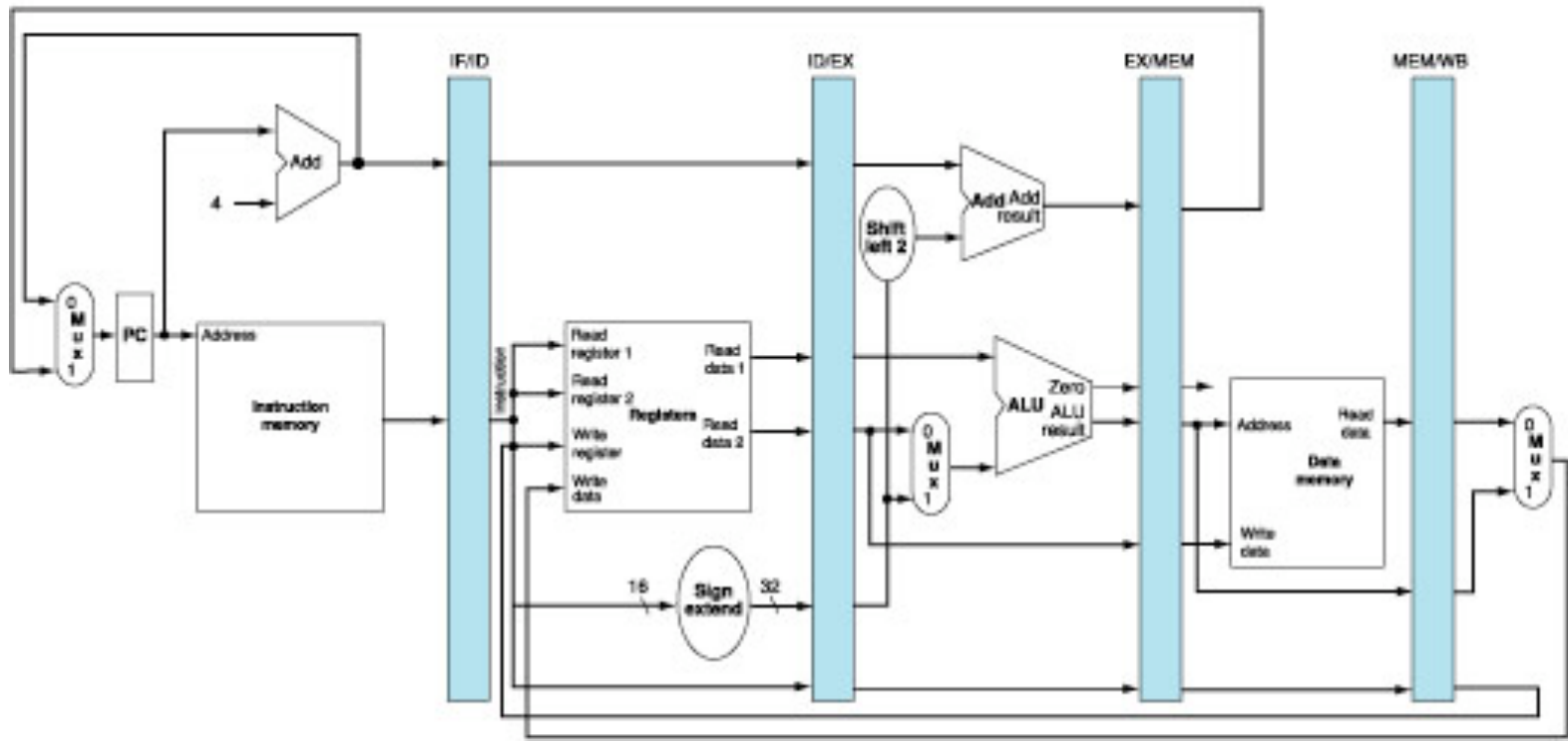


SW Execution







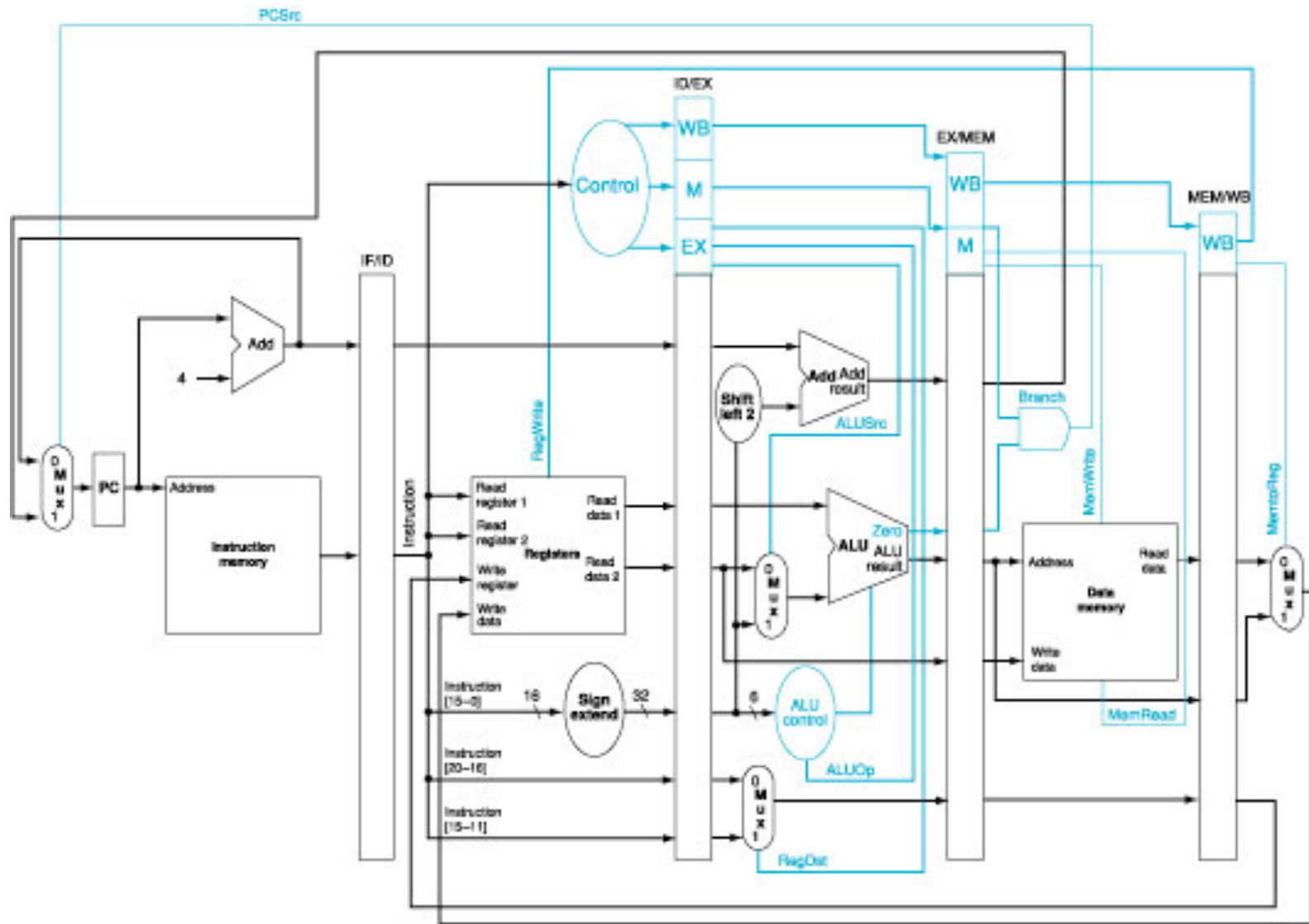


Pipelined Control

- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- Some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can pass them in the pipeline registers, along with the other data.
- Control signals can be categorized by the pipeline stage that uses them.

Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

Pipelined Control



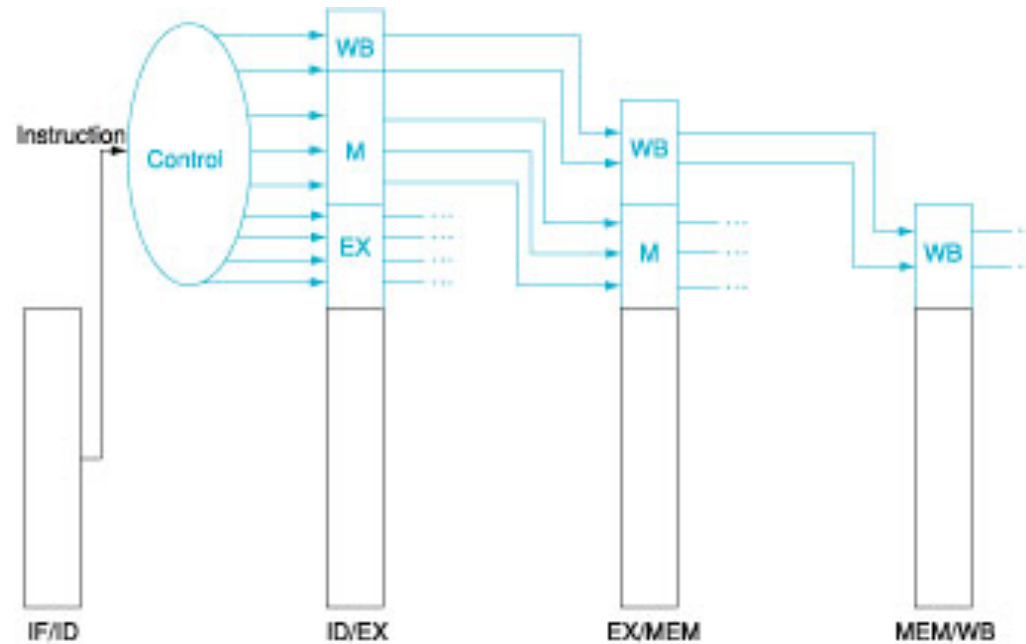
Pipelined Control

- Because the datapath fetches one instruction per cycle, the PC must also be updated on each clock cycle. Including a write enable for the PC would be redundant.
- Similarly, the pipeline registers are also written on every cycle, so no explicit write signals are needed.

Pipelined Control

- The control signals are grouped together in the pipeline registers, just to make the diagram a little clearer.
- Divide the control lines into five groups according to pipeline stage:
 1. IF: no control needed
 2. ID: no control needed
 3. EX: RegDst (result register), ALUOp (operation), ALUSrc (register/immediate)
 4. MEM: Branch (beq), MemRead (load), MemWrite (store)
 5. WB: MemtoReg (ALU/memory result), RegWrite
- How to get the right control information to right stage at the right time?
 - Create during instruction decode, pass along via pipeline registers.

Pipelined Control



Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back 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

Data hazard and forwarding

- Example with dependences

sub \$2, \$1, \$3

and \$12, \$2, \$5

or \$13, \$6, \$2

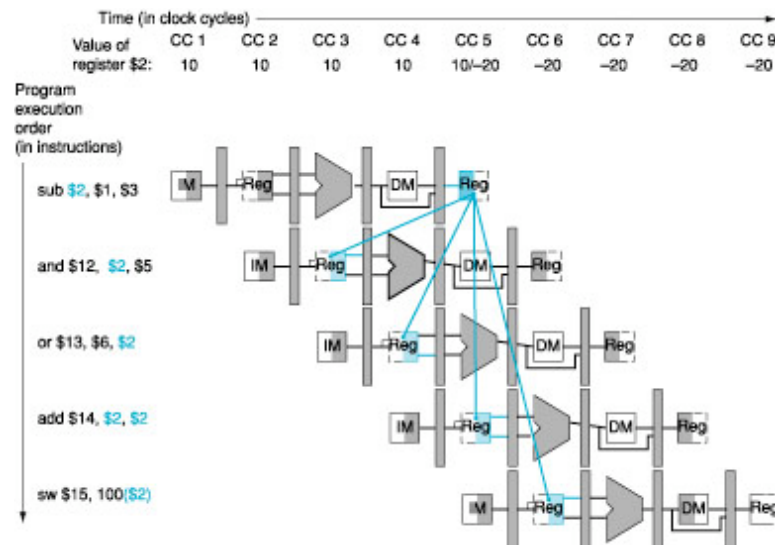
add \$14, \$2, \$2

sw \$15, 100(\$2)

- There are several dependencies in this code fragment.
- The first instruction, SUB, stores a value into \$2.
- That register is used as a source in the rest of the instructions.
- This is not a problem for the single-cycle and multicycle datapaths.
- Each instruction is executed completely before the next one begins.
- This ensures that instructions 2 through 5 above use the new value of \$2 (the sub result), just as we expect.
- How would this code sequence fare in our pipelined datapath?

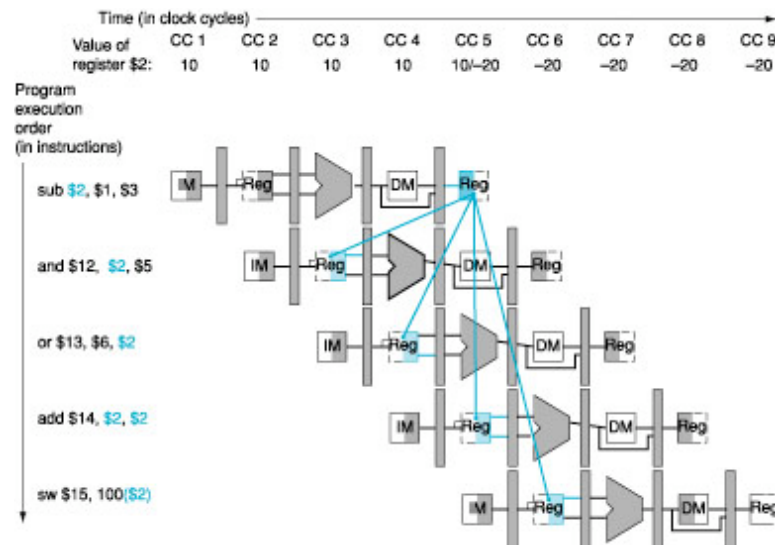
Data hazard and forwarding

- The SUB instruction does not write to register \$2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
- The AND reads register \$2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of \$2, not the new one.



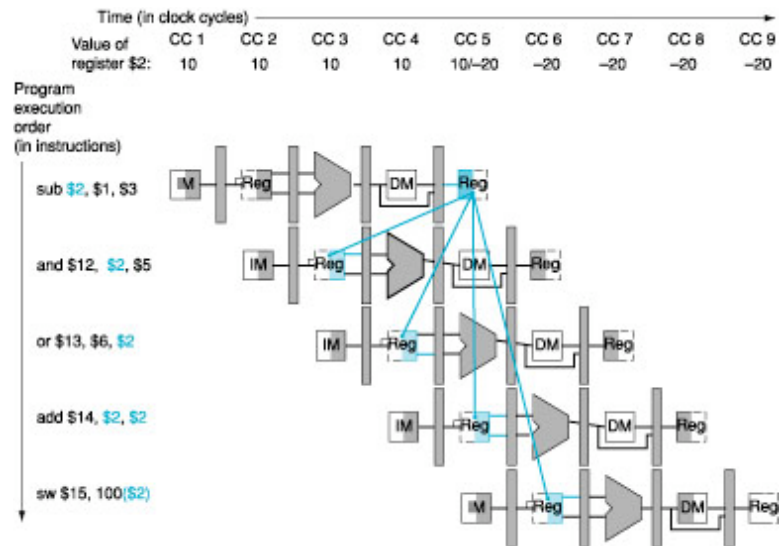
Data hazard and forwarding

- Similarly, the OR instruction uses register \$2 in cycle 4, again before it's actually updated by SUB.
- The ADD instruction is okay, because of the register file design.
 - Registers are written at the beginning of a clock cycle.
 - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads \$2 after the SUB finishes.



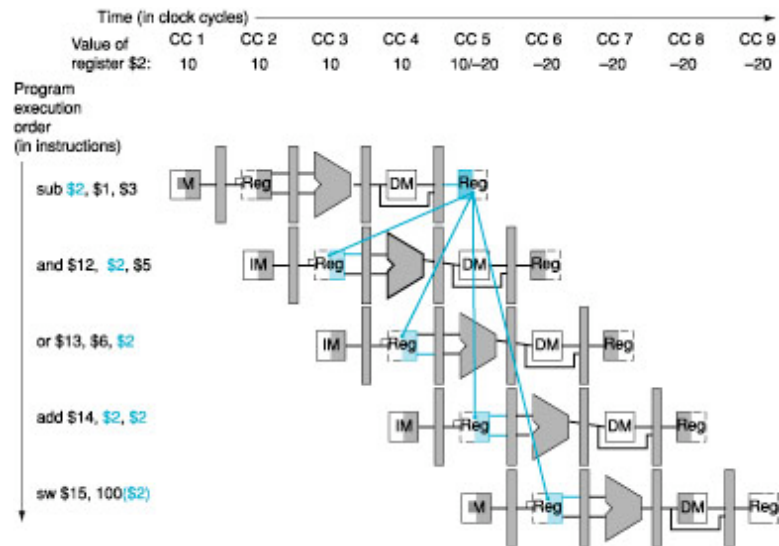
Data hazard and forwarding

- Arrows indicate the flow of data between instructions.
- The tails of the arrows show when register \$2 is written.
- The heads of the arrows show when \$2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions 1 & 2 and 1 & 3.



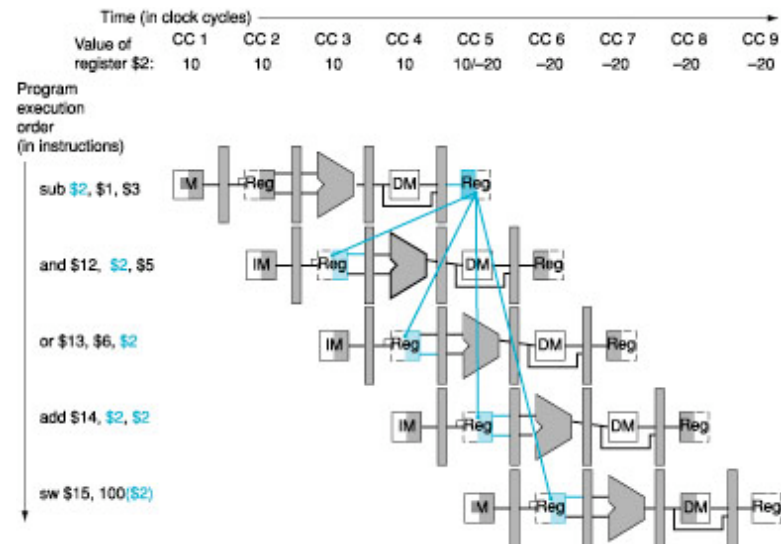
Data hazard and forwarding

- Let's look at when the data is actually produced and consumed.
- The SUB instruction produces its result in its EX stage, during cycle 3 in the diagram below.
- The AND and OR need the new value of \$2 in their EX stages, during clock cycles 4 and 5.



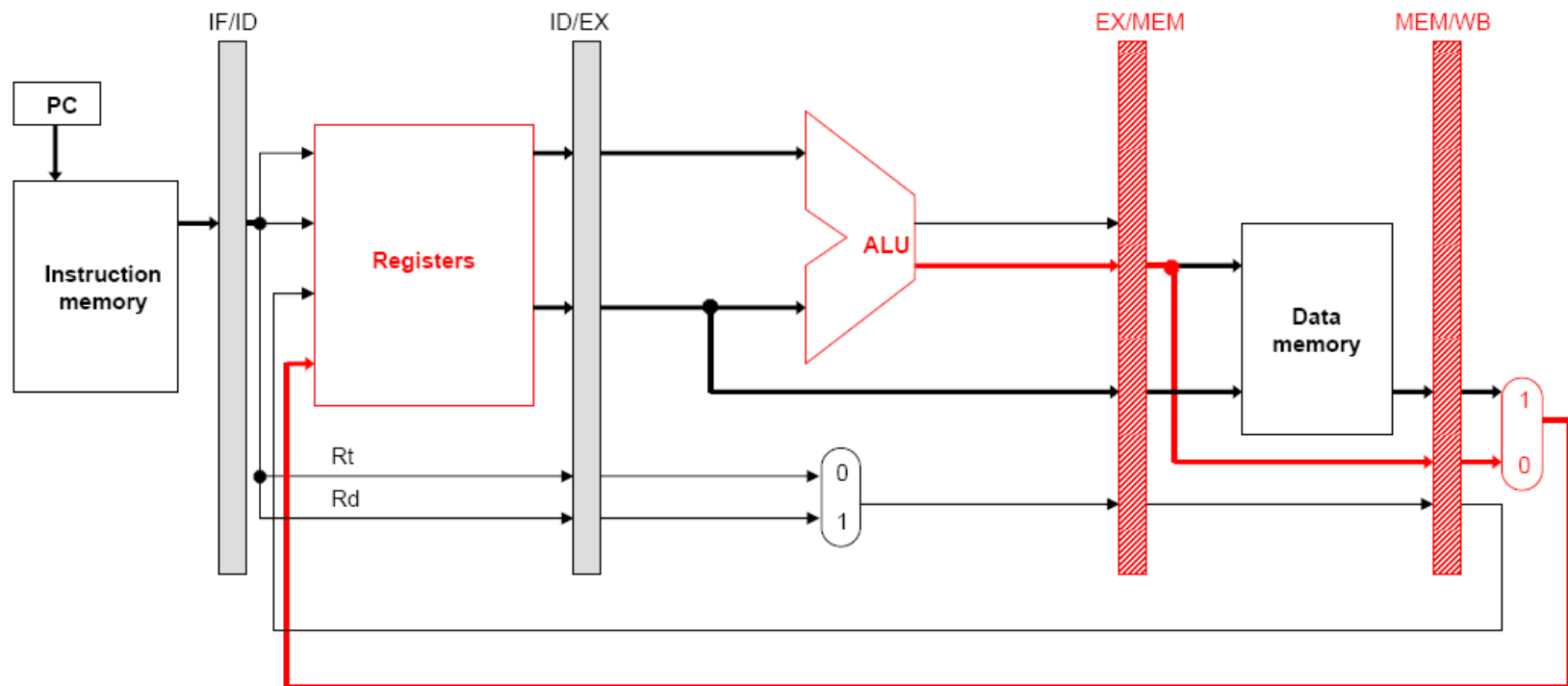
Data hazard and forwarding

- The actual result \$1 - \$3 is computed in clock cycle 3, *before* it's needed in cycles 4 and 5.
- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
- Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.



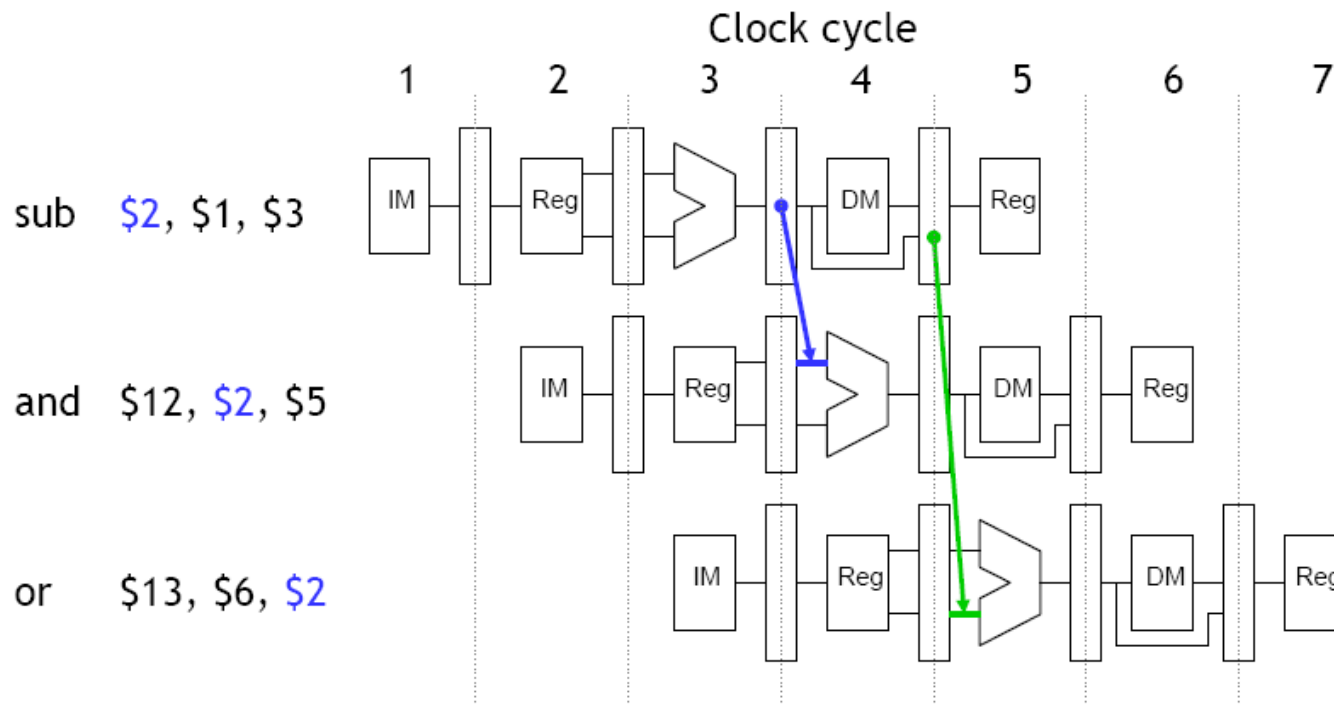
Data hazard and forwarding

- The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.



Data hazard and forwarding

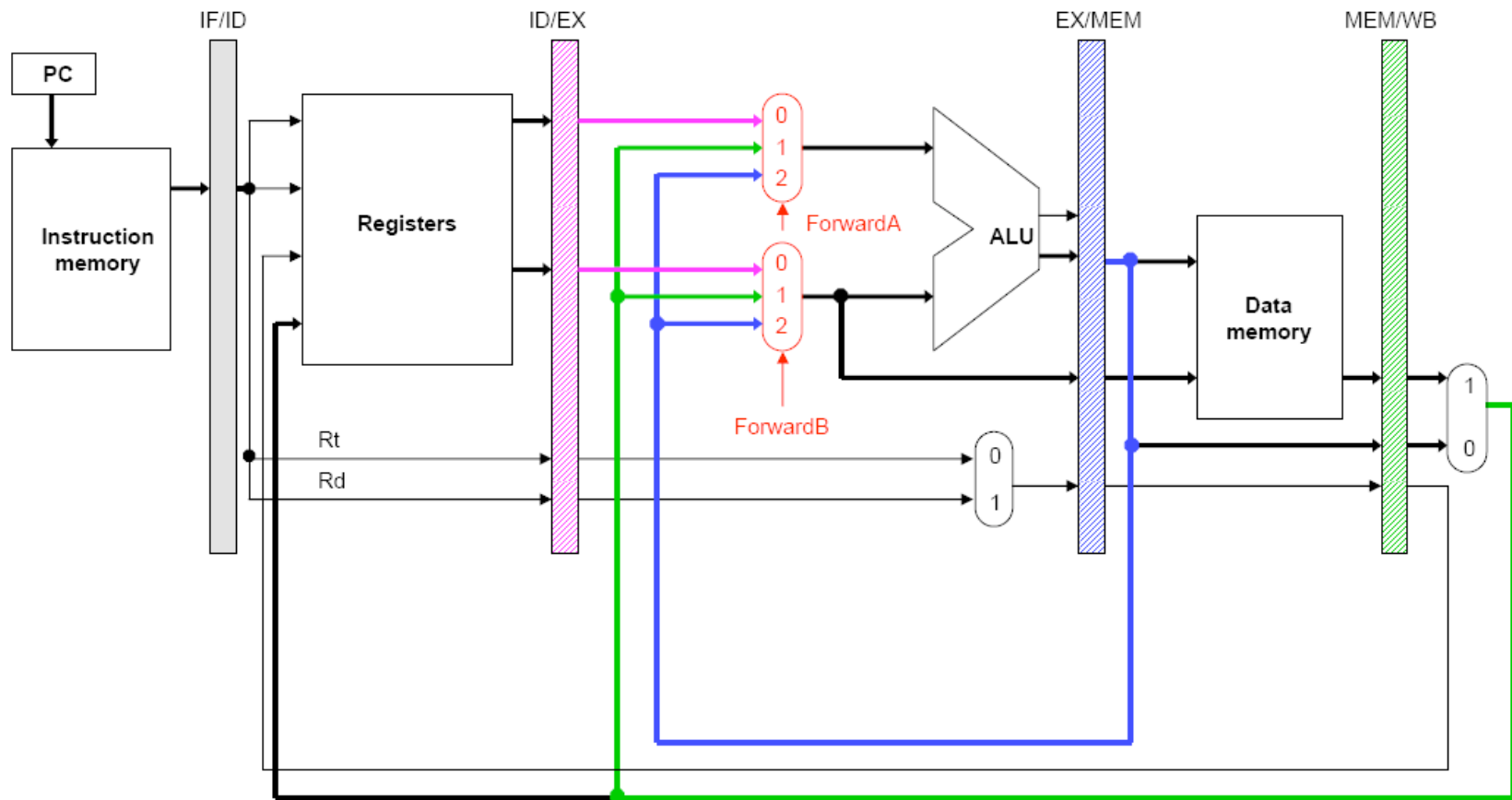
- Since the pipeline registers already contain the ALU result, we could just forward that value to subsequent instructions, to prevent data hazards.
- In clock cycle 4, the AND instruction can get the value \$1 - \$3 from the EX/MEM pipeline register used by sub.
- Then in cycle 5, the OR can get that same result from the MEM/WB pipeline register being used by SUB.



Data hazard and forwarding

- A forwarding unit selects the correct ALU inputs for the EX stage.
- If there is no hazard, the ALU's operands will come from the register file, just like before.
- If there is a hazard, the operands will come from either the EX/MEM or MEM/WB pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB.

Data hazard and forwarding



Data hazard and forwarding

- So how can the hardware determine if a hazard exists?
- An EX hazard occurs between the instruction currently in its EX stage and the previous instruction if:
 1. The previous instruction will write to the register file, *and*
 2. The destination is one of the ALU source registers in the EX stage.
- There is an EX hazard between the two instructions below:
sub \$2, \$1, \$3
and \$12, \$2, \$5

Data hazard and forwarding

- New notation for different fields of data in pipeline registers: data in a pipeline register can be referenced using a class-like syntax.
- For example, ID/EX.RegisterRt refers to the rt field stored in the ID/EX pipeline.

Data hazard and forwarding

- The first ALU source comes from the pipeline register when necessary.
 - if ($EX/MEM.RegWrite = 1$
and ($EX/MEM.RegisterRd \neq 0$)
and $EX/MEM.RegisterRd = ID/EX.RegisterRs$)
then $ForwardA = 2$
- The second ALU source is similar.
 - if ($EX/MEM.RegWrite = 1$
and ($EX/MEM.RegisterRd \neq 0$)
and $EX/MEM.RegisterRd = ID/EX.RegisterRt$)
then $ForwardB = 2$

Data hazard and forwarding

- Here is an equation for detecting and handling MEM hazards for the first ALU source.
if ($MEM/WB.RegWrite = 1$
and ($MEM/WB.RegisterRd \neq 0$)
and $MEM/WB.RegisterRd = ID/EX.RegisterRs$
then $ForwardA = 1$
- The second ALU operand is handled similarly.
if ($MEM/WB.RegWrite = 1$
and $MEM/WB.RegisterRd = ID/EX.RegisterRt$
and ($MEM/WB.RegisterRd \neq 0$)
then $ForwardB = 1$

Data hazard and forwarding

- One new problem is if a register is updated twice in a row.
 - add \$1, \$2, \$3
 - add \$1, \$1, \$4
 - sub \$5, \$5, \$1
- Register \$1 is written by *both* of the previous instructions, but only the most recent result (from the second ADD) should be forwarded.

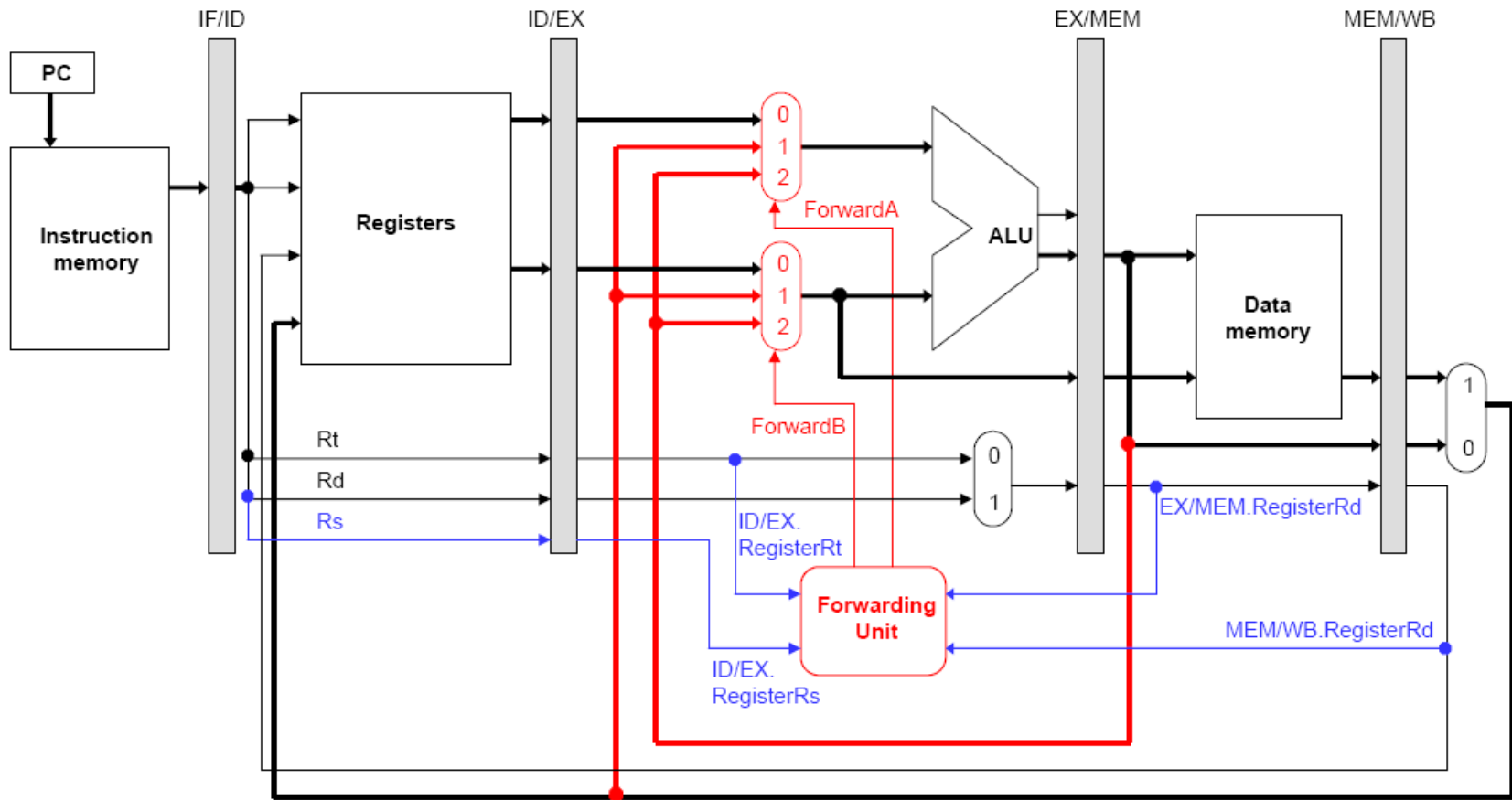
Data hazard and forwarding

- Here is an equation for detecting and handling MEM hazards for the first ALU source.

if ($MEM/WB.RegWrite = 1$
and ($MEM/WB.RegisterRd \neq 0$)
and $MEM/WB.RegisterRd = ID/EX.RegisterRs$
and ($EX/MEM.RegisterRd \neq ID/EX.RegisterRs$)
then $ForwardA = 1$

- The second ALU operand is handled similarly.

if ($MEM/WB.RegWrite = 1$
and $MEM/WB.RegisterRd = ID/EX.RegisterRt$
and ($MEM/WB.RegisterRd \neq 0$)
and ($EX/MEM.RegisterRd \neq ID/EX.RegisterRs$)
then $ForwardB = 1$

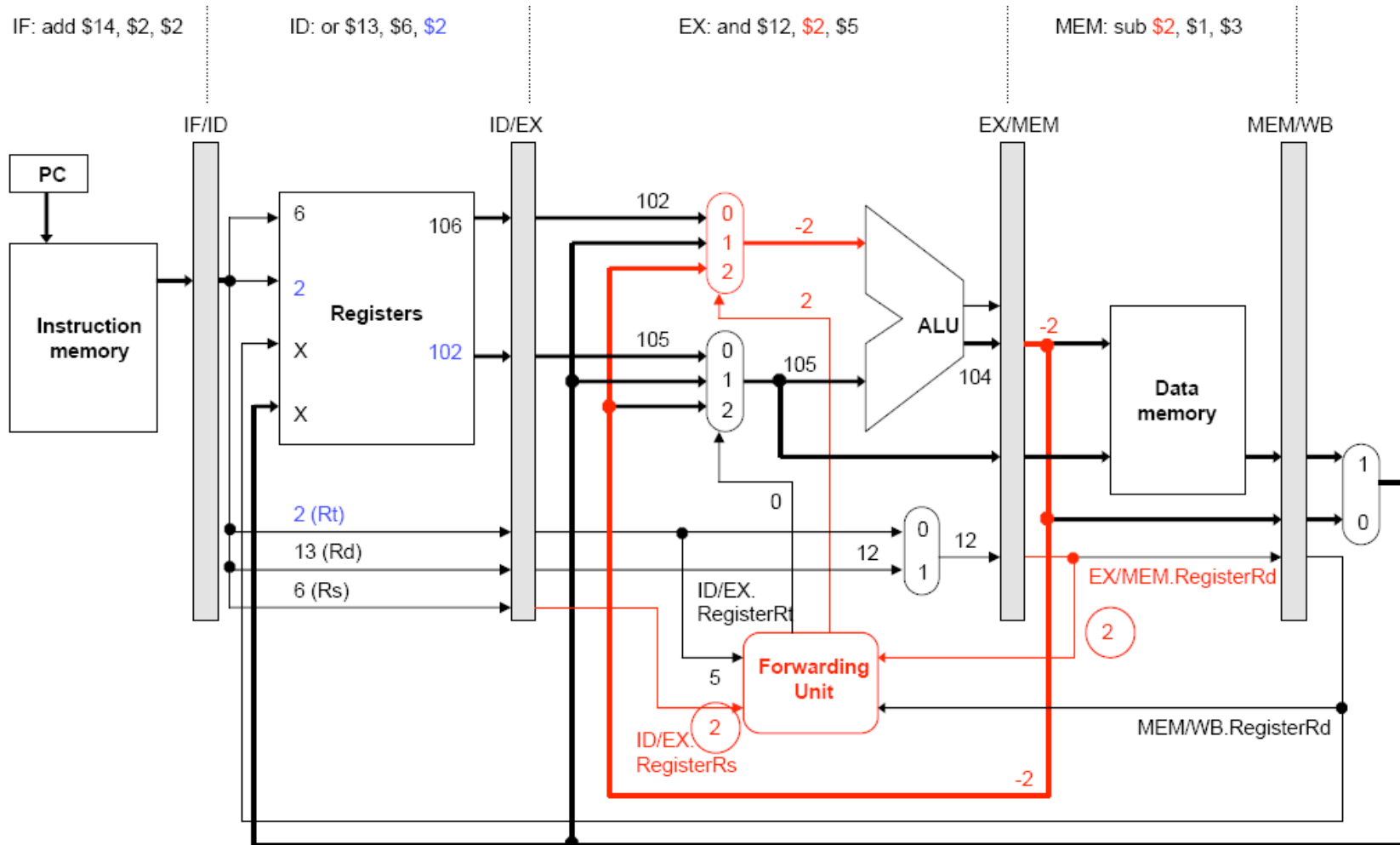


Example

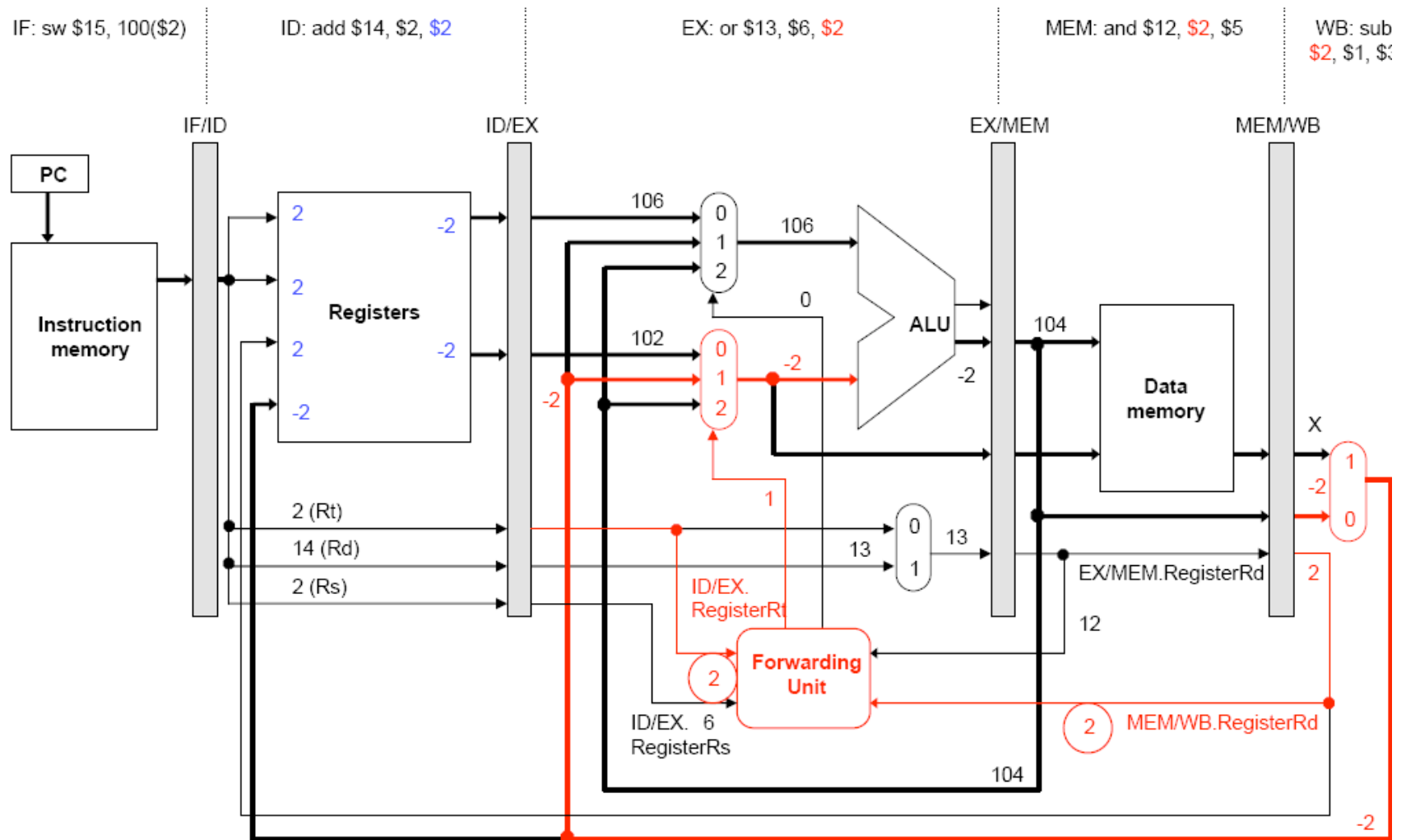
sub \$2, \$1, \$3
and \$12, \$2, \$5
or \$13, \$6, \$2
add \$14, \$2, \$2
sw \$15, 100(\$2)

- Assume each register initially contains its number plus 100.
- After the first instruction, \$2 should contain $101 - 103 = -2$.
- The other instructions should all use -2 as one of their operands.
- We'll try to keep the example short.
 - Assume no forwarding is needed except for register \$2.
 - We'll skip the first two cycles, since they're the same as before.

Clock cycle 4: forwarding \$2 from EX/MEM



Clock cycle 5: forwarding \$2 from MEM/WB



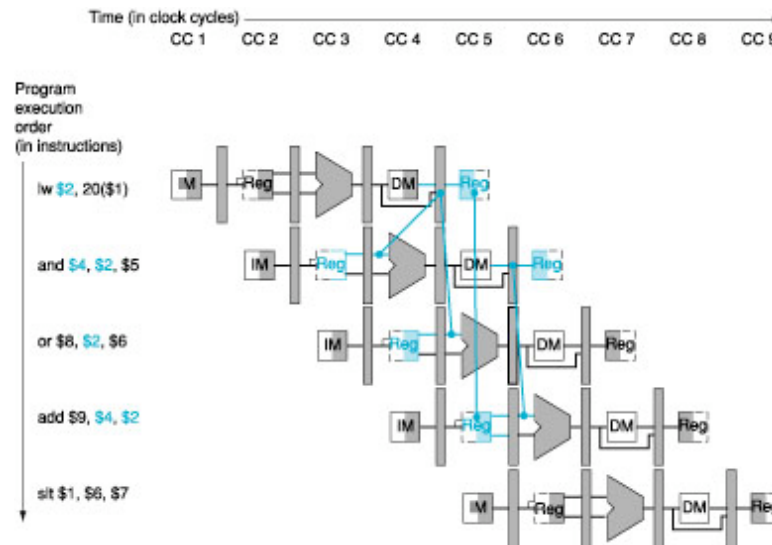
-
- The first data hazard occurs during cycle 4.
 - The forwarding unit notices that the ALU's first source register for the AND is also the destination of the SUB instruction.
 - The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.
 - A second hazard occurs during clock cycle 5.
 - The ALU's second source (for OR) is the SUB destination again.
 - This time, the value has to be forwarded from the MEM/WB pipeline register instead.
 - There are no other hazards involving the SUB instruction.
 - During cycle 5, SUB writes its result back into register \$2.
 - The ADD instruction can read this new value from the register file in the same cycle.

Data Hazard and Stall

- So far, we have discussed data hazards that can occur in pipelined CPUs if some instructions depend upon others that are still executing.
- Many hazards can be resolved by forwarding data from the pipeline registers, instead of waiting for the writeback stage.
- The pipeline continues running at full speed, with one instruction beginning on every clock cycle.
- Now, we'll see some real limitations of pipelining.
 - Forwarding may not work for data hazards from load instructions.
- In these cases we may need to slow down, or stall, the pipeline.

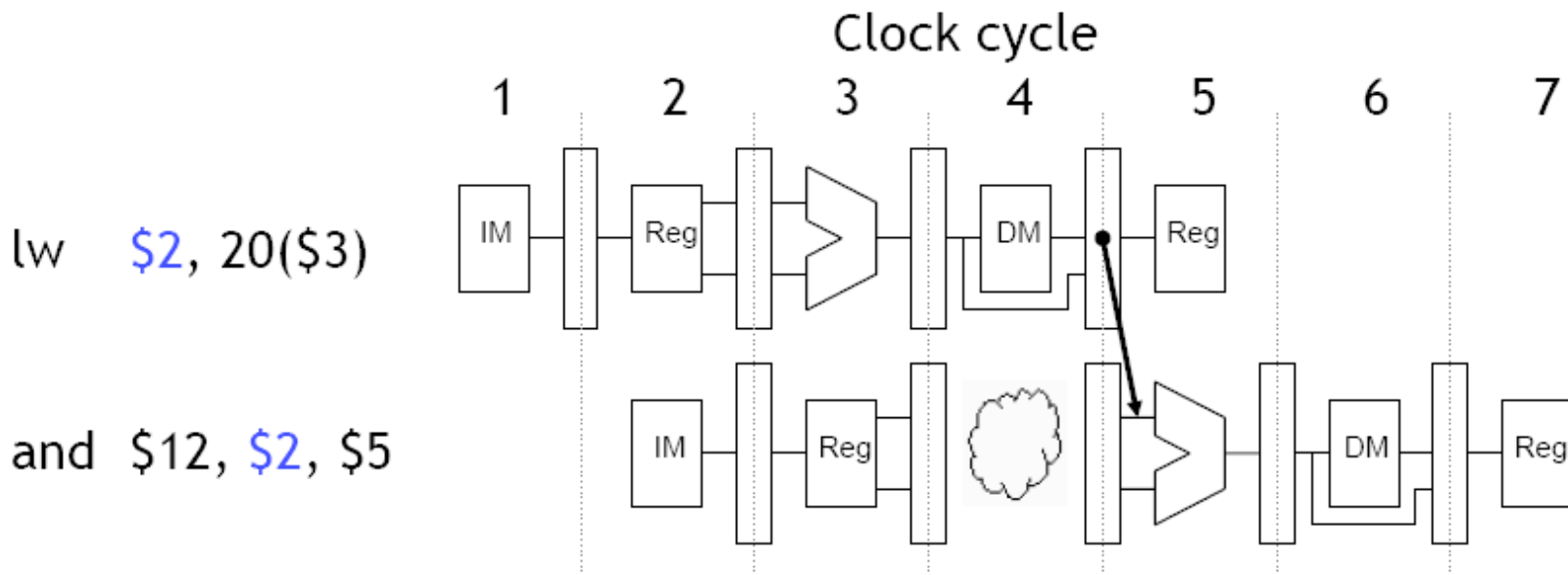
Data Hazard and Stall

- Consider the following example.
- The load data doesn't come from memory until the *end* of cycle 4.
- But the AND needs that value at the *beginning* of the same cycle!
- This is a “true” data hazard—the data is not available when we need it.



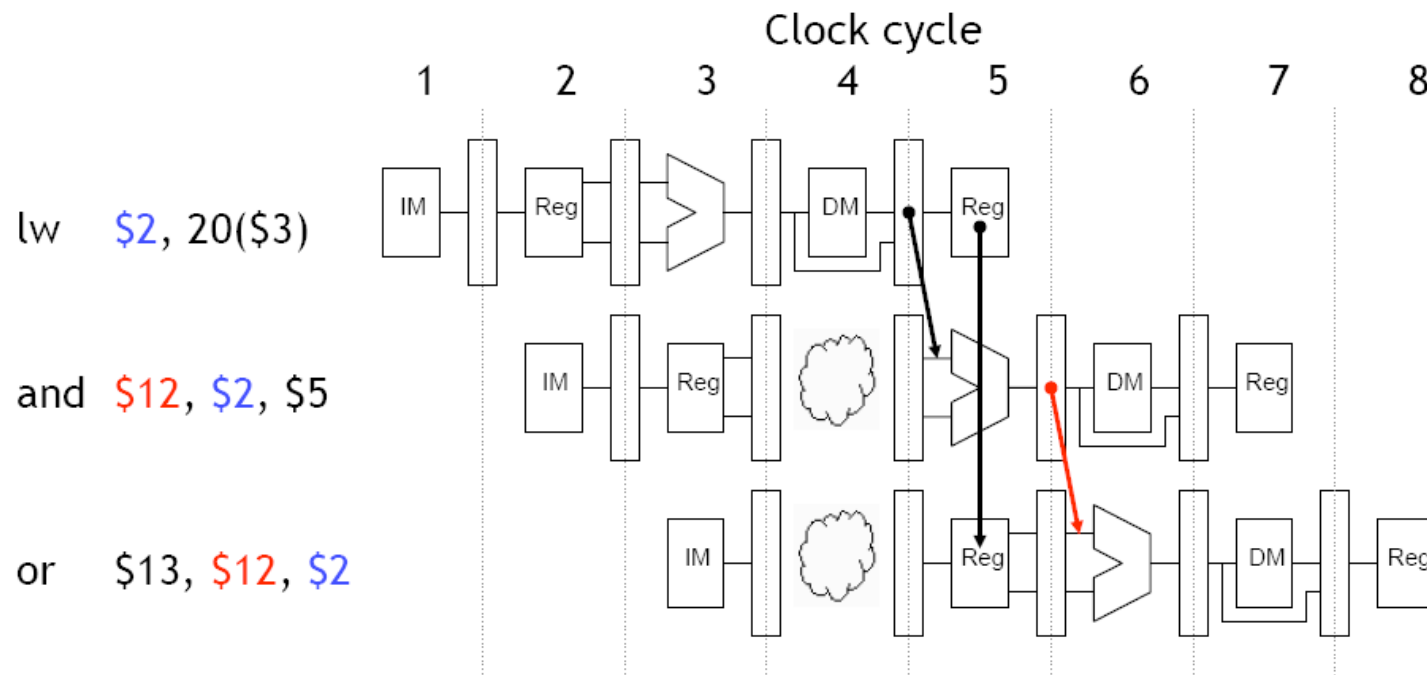
Data Hazard and Stall

- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.
- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.



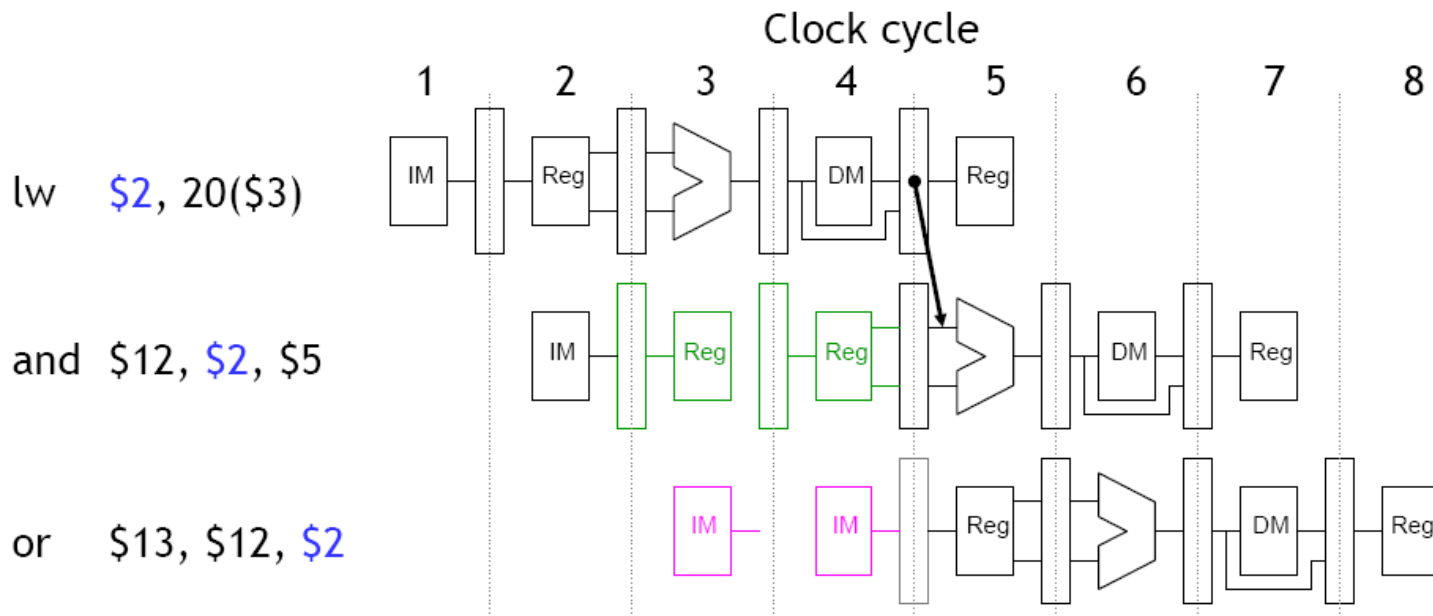
Data Hazard and Stall

- If we delay the second instruction, we'll have to delay the third one too.
- This is necessary to make forwarding work between AND and OR.
- It also prevents problems such as two instructions trying to write to the same register in the same cycle.



Data Hazard and Stall

- One way to implement a stall is to force the two instructions after LW to pause and remain in their ID and IF stages for one extra cycle.
- This is easily accomplished.
 - Don't update the PC, so the current IF stage is repeated.
 - Don't update the IF/ID register, so the ID stage is also repeated.



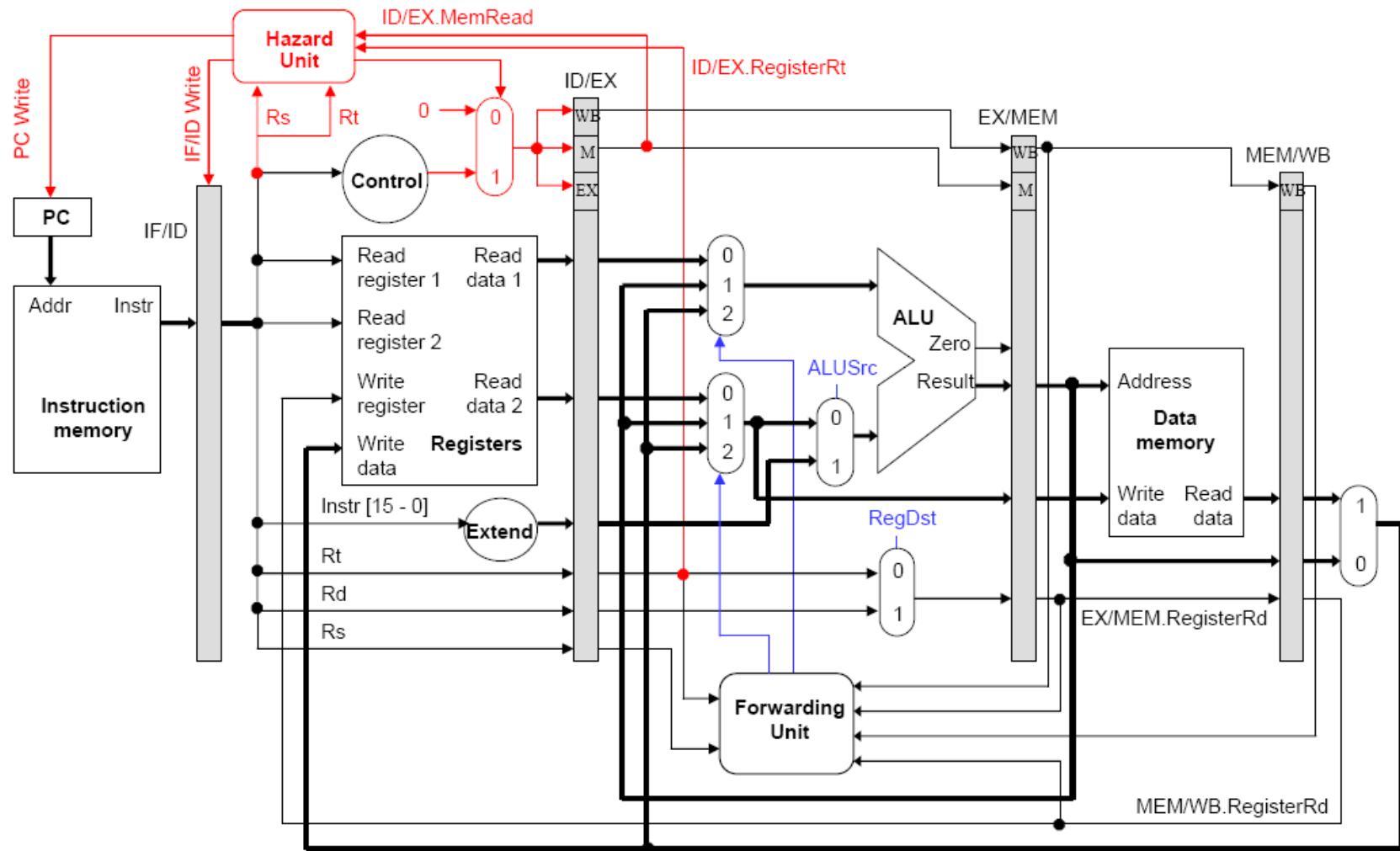
Data Hazard and Stall

- But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?
- Those units aren't used in those cycles because of the stall, so we can set the EX, MEM and WB control signals to all 0s.
- Detecting stall is much like detecting data hazards.

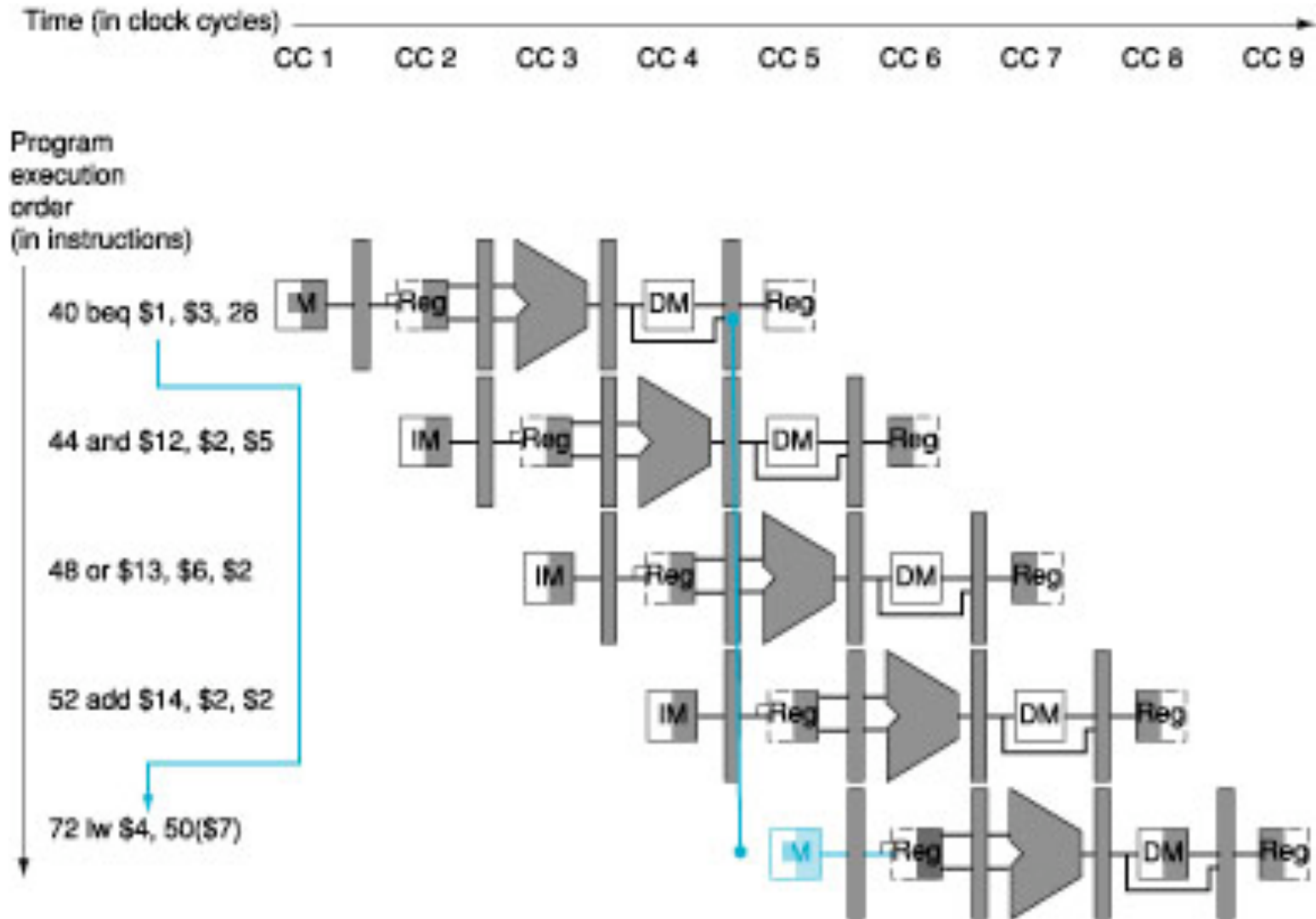
Data Hazard and Stall

- We can detect a load hazard between the current instruction in its ID stage and the previous instruction in the EX stage just like we detected data hazards.
- A hazard occurs if the previous instruction was LW...
 $ID/EX.MemRead = 1$
...and the LW destination is one of the current source registers.
 $ID/EX.RegisterRt = IF/ID.RegisterRs$
or
 $ID/EX.RegisterRt = IF/ID.RegisterRt$
- The complete test for stalling is the conjunction of these two conditions.
if ($ID/EX.MemRead = 1$ and
($ID/EX.RegisterRt = IF/ID.RegisterRs$ or
 $ID/EX.RegisterRt = IF/ID.RegisterRt$))
then stall

Data Hazard and Stall



Branch Hazard

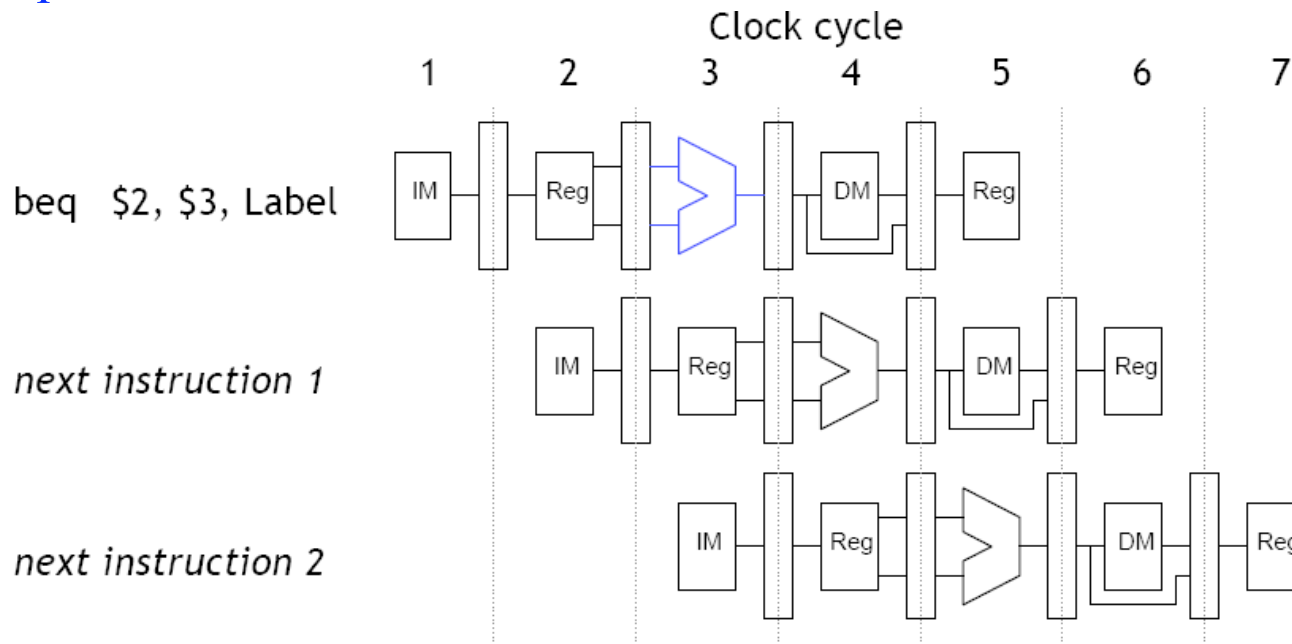


Branch Hazard

- Most of the work for a branch computation is done in the EX stage.
 - The branch target address is computed.
 - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
- But we need to know which instruction to fetch next, in order to keep the pipeline running!
- This leads to what's called a control hazard.

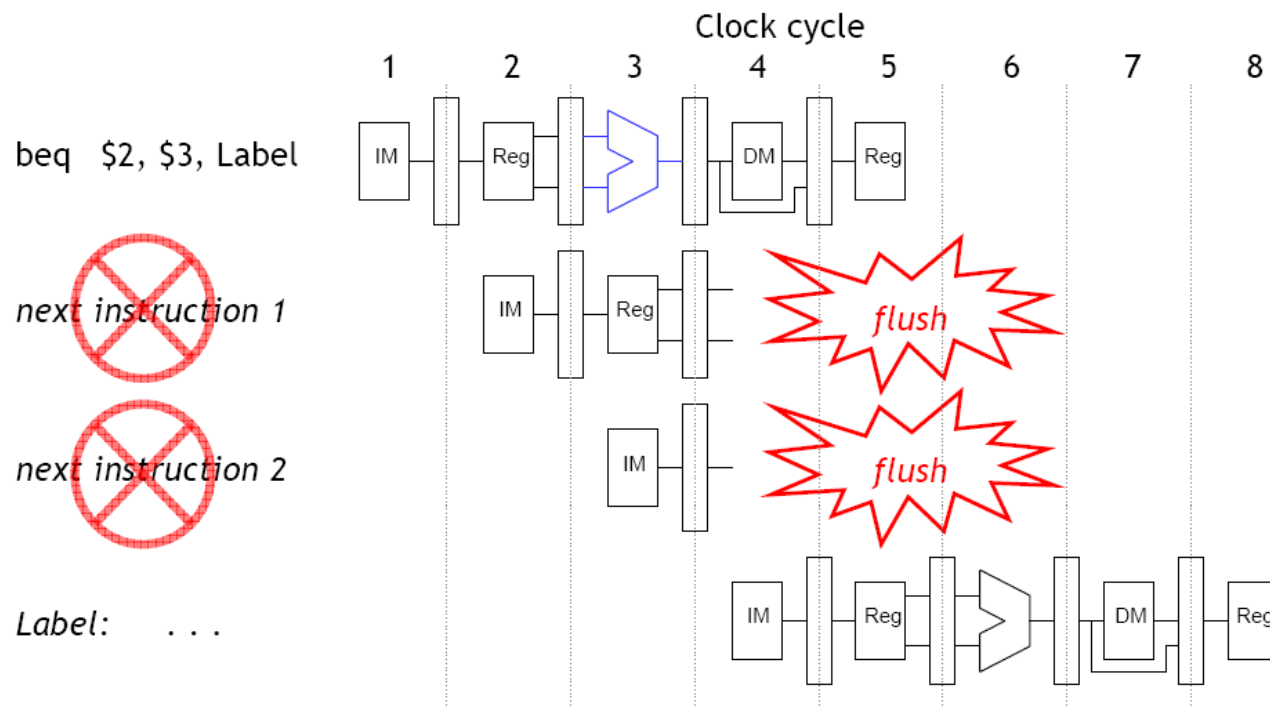
Branch Hazard

- Again, stalling is always one possible solution.
- Another approach is to *guess* whether or not the branch is taken.
 - In terms of hardware, it's easier to assume the branch is *not* taken.
 - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch Hazard

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.

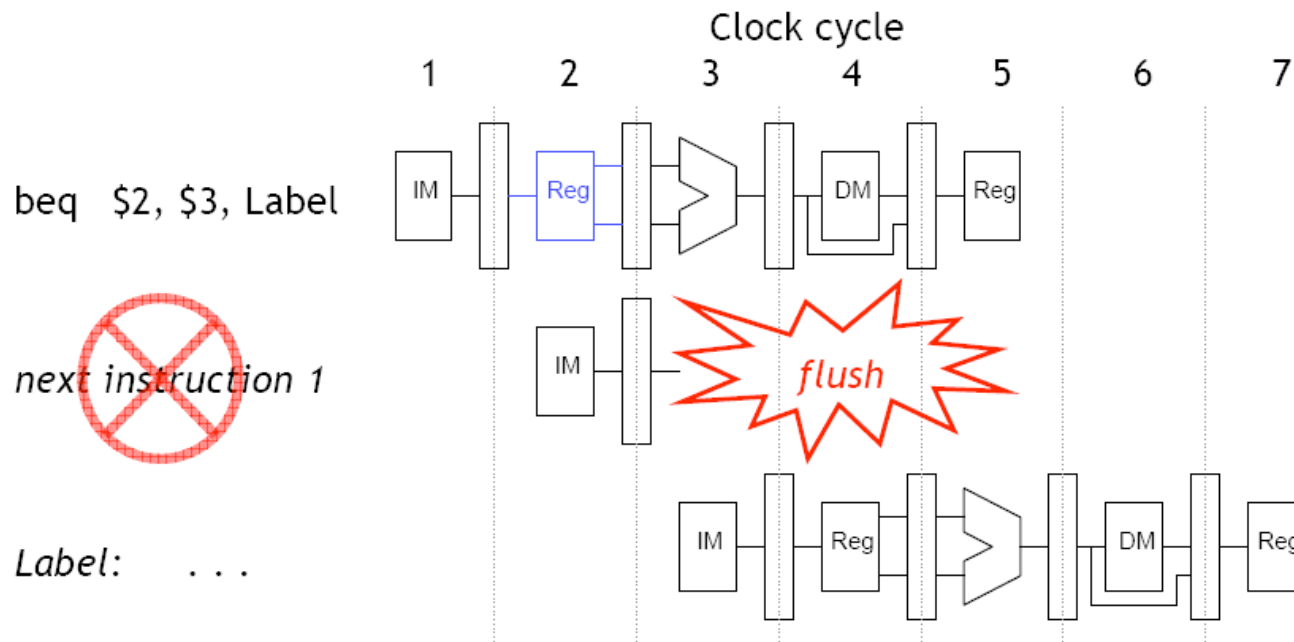


Branch Hazard

- Overall, branch prediction is worth it.
 - Mispredicting a branch means that two clock cycles are wasted.
 - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for *every* branch.
- All modern CPUs use branch prediction.
 - Accurate predictions are important for optimal performance.
 - Most CPUs predict branches dynamically—statistics are kept at runtime to determine the likelihood of a branch being taken.
- The pipeline structure also has a big impact on branch prediction.
 - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
 - We must also be careful that instructions do not modify registers or memory before they get flushed.

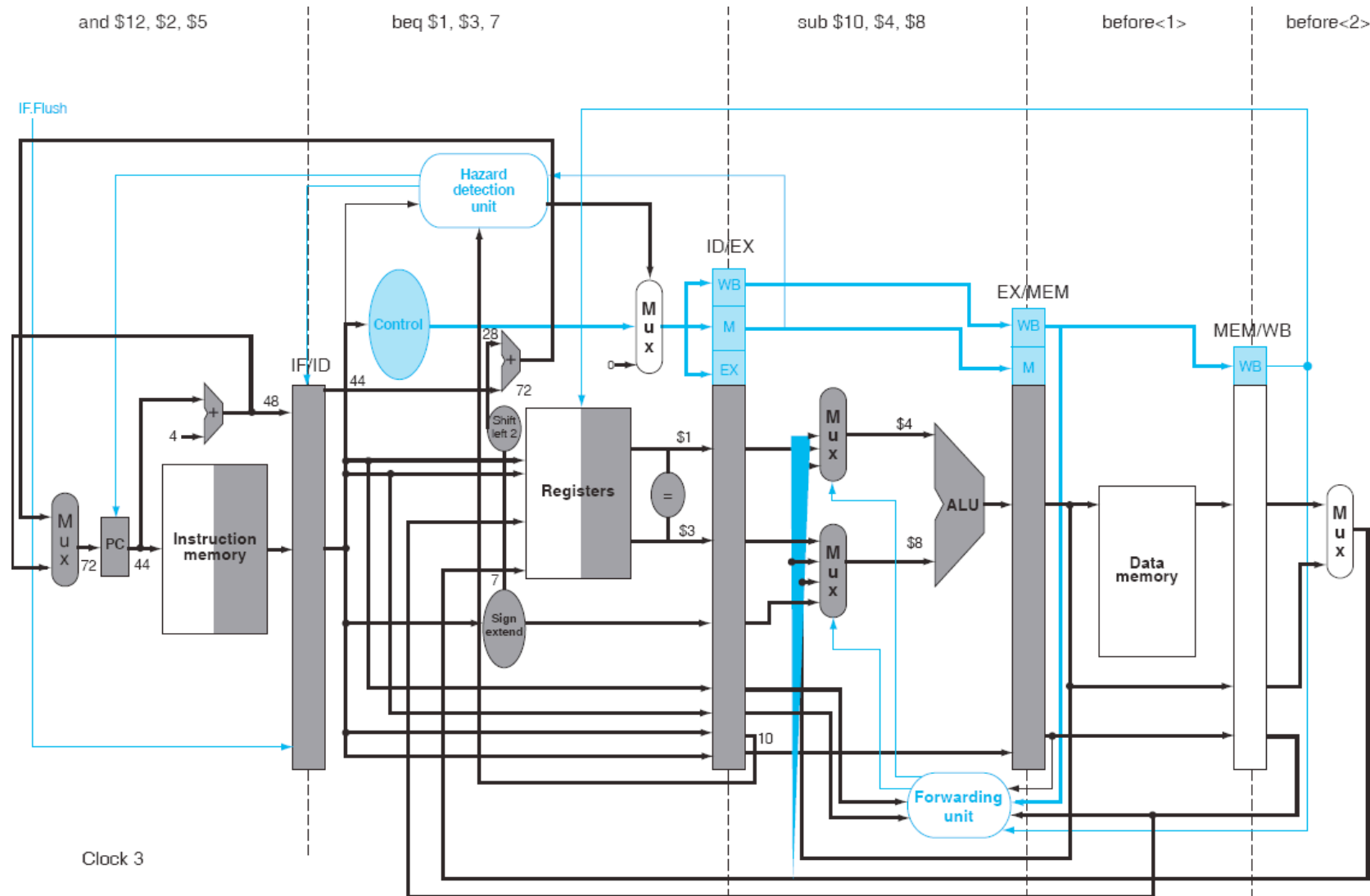
Branch Hazard

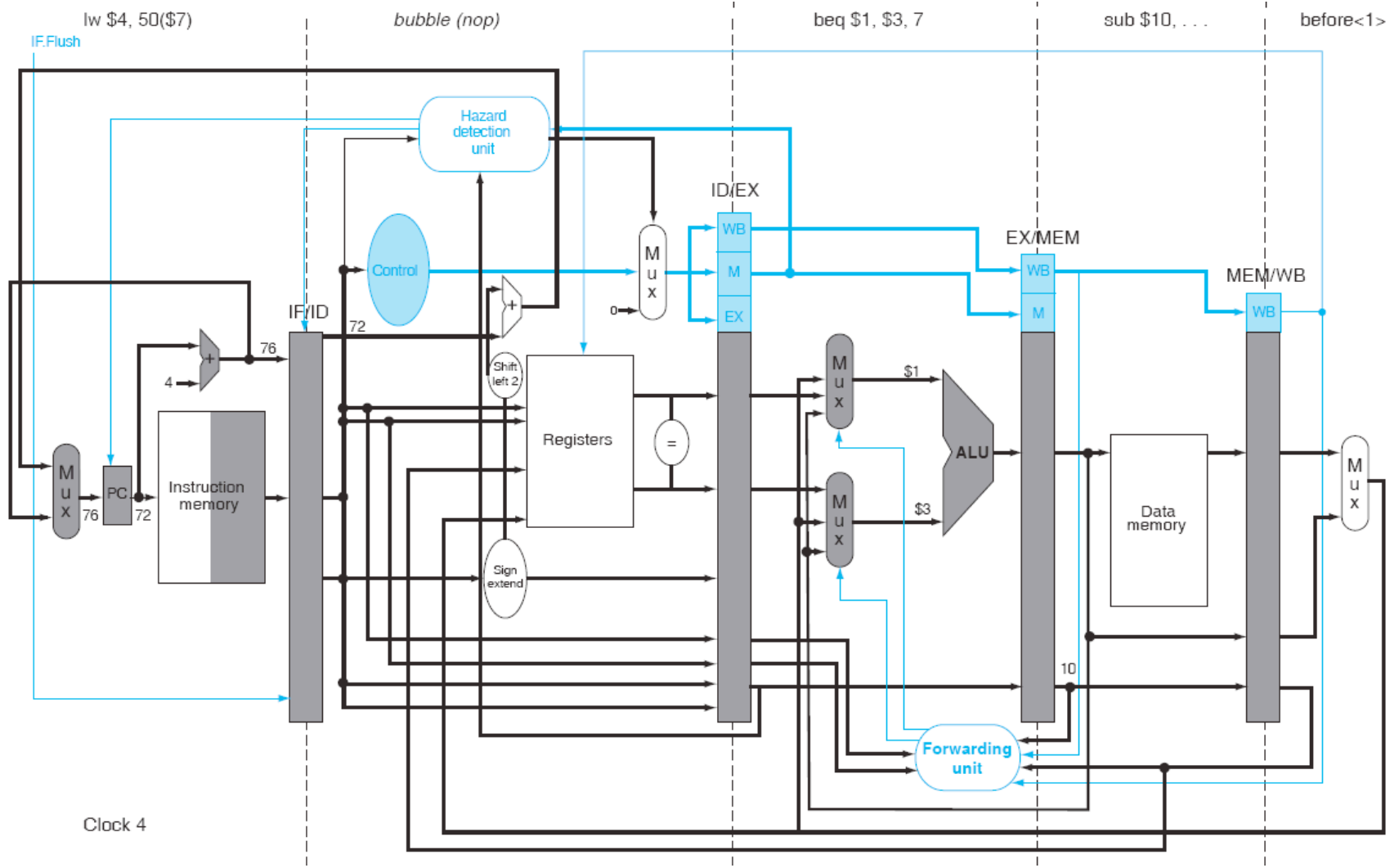
- We can actually decide the branch a little earlier, in ID instead of EX.
 - Our sample instruction set has only a BEQ.
 - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.



Branch Hazard

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
 - MIPS uses `sll $0, $0, 0` as the nop instruction.
 - This happens to have a binary encoding of all 0s: `0000 0000`.
- Flushing introduces a bubble into the pipeline, which represents the onecycle delay in taking the branch.
- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.





Clock 4

Dynamic Branch Prediction

- ❑ Dynamic branch prediction: prediction of branches at runtime using runtime information
- ❑ 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 execute
 - 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 the 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

1-bit Prediction Accuracy

- ❑ A 1-bit predictor will be incorrect twice when not taken

- Assume `predict_bit = 0` to start (indicating branch not taken) and loop control is at the bottom of the loop code

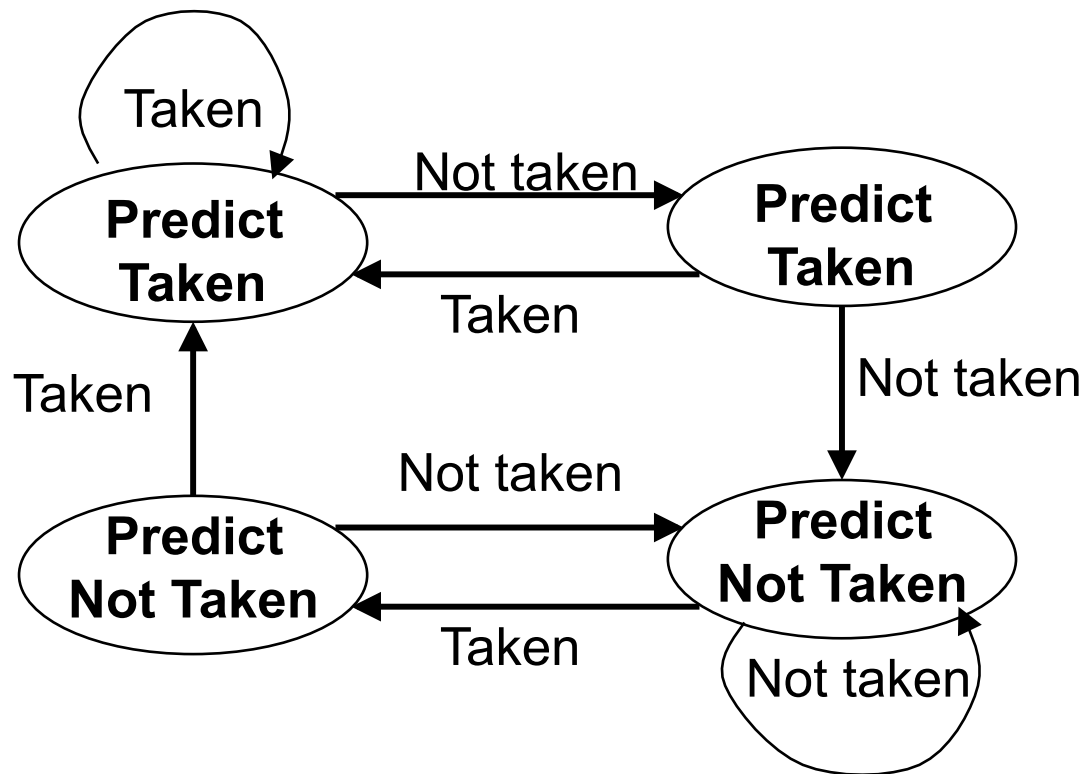
1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (`predict_bit = 1`)
2. As long as branch is taken (looping), prediction is correct
3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (`predict_bit = 0`)

```
Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- ❑ For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed

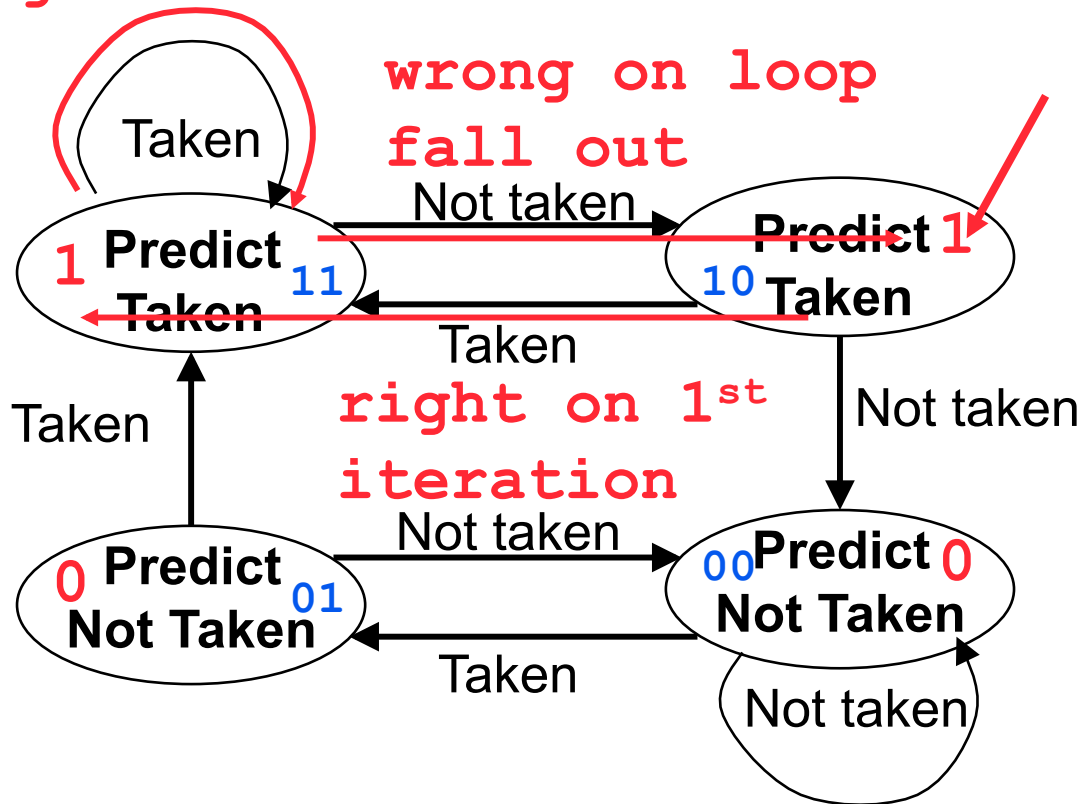


```
Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed

right 9 times



```

Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
last loop instr
bne $1,$2,Loop
fall out instr
    
```

- BHT also stores the initial FSM state