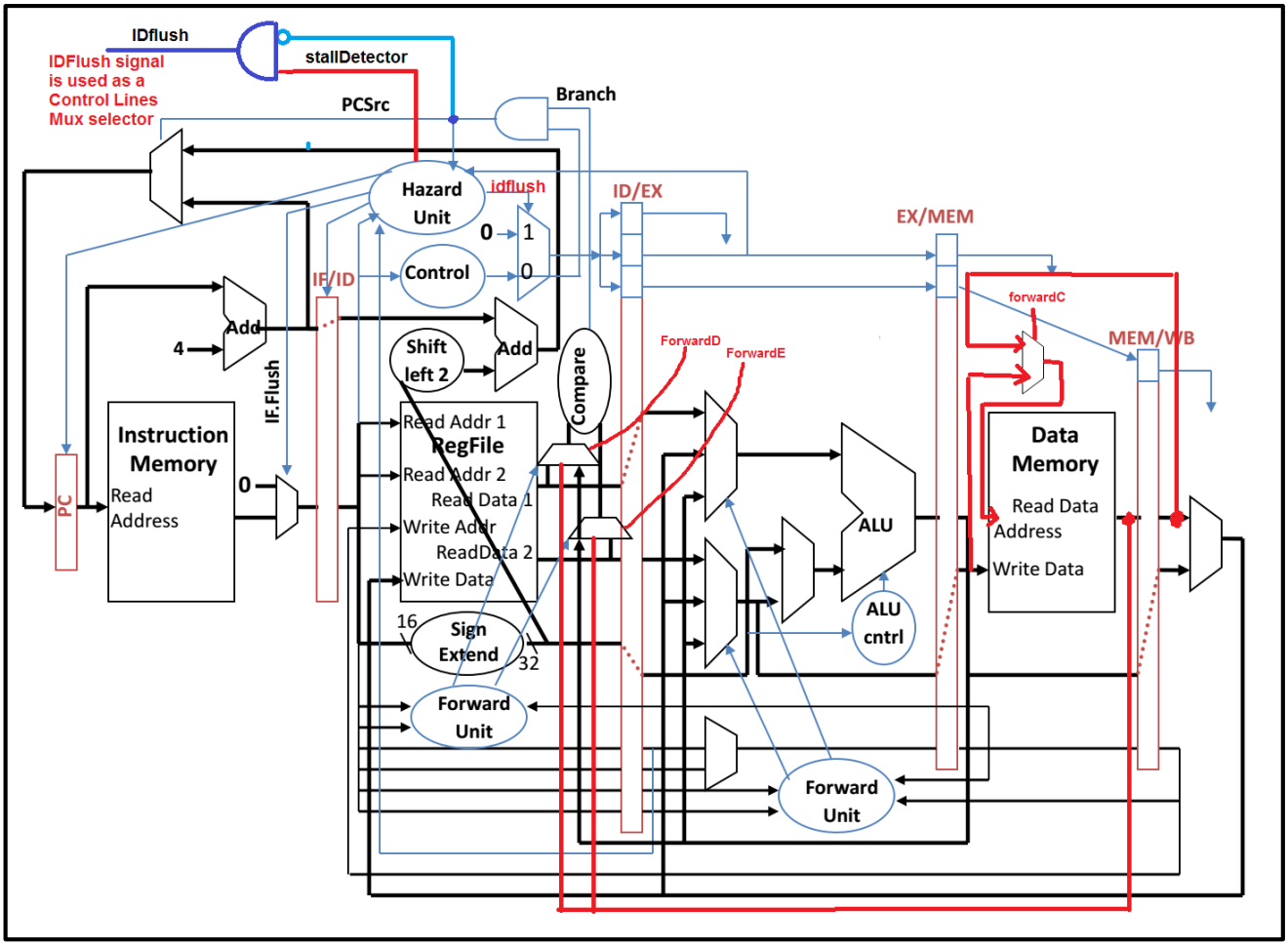


MIPS PIPLELINE **PROCESSOR**

Designing for Hazards

MIPS PIPELINE PROCESSOR *DESIGNING FOR DATA HAZARDS AND CONTROL HAZARDS AND STATIC BRANCH HAZARDS*

Group40

CONTENTS

Online Resources

Site	Location	Description
Washington Website	https://courses.cs.washington.edu/courses/cse378/07au/lectures/	*Contain very useful slides that handle most common concepts of the pipelining*
Github	Some Repositories on Github	*Get some ideas from available codes on github*
Ahmed Fathy Videos	https://www.youtube.com/user/noone988	*Best hardware qualified person in the universe.*
Xilinx	Xilinx website	Searching for synthesizing errors and try to understand them and handle them

PREFACE

We tried hardly to support all possible data hazards, control hazards and stalling conditions, and we hope that we succeed in that.

- **Data Hazards:** hazards occurs due to read after write dependencies.
- **Control Hazards:** Control pipelined registers in case if the processor needs Stalling or flushing the pipeline.
- **Branching:** Supporting static branch and assumed that the branch is not taken,also also supporting branch hazards
- **Supported R-Formats Instructions :** add,sub,and,or,xor,nor,sll,srl,sra,slt,jr
- **Supported I-Formats Instructions :** addi,andi,ori,xori,slti,lui,lw,sw,beq,bne
- **Supported J-Formats Instructions :** j,jal
- **Supporting signed and unsigned operands**
- **I/O standards:** The only input to the MIPS Top Module is **Clk** signal.
- **Test cases :** Support test cases with very hard situations to handle , you will Find it in Test cases Document.

ABOUT THE AUTHOR

This project is made by a group of students (Group40) at 3rd year Computer and systems department students at Faculty of Engineering Ain Shams University.

This is a complete rewrite in Verilog HDL. The aim of this project is to simulate a pipelined MIPS processor dealing with various MIPS instructions.

This Project supporting data hazards, control hazards for LW, BEQ and R-Format instructions.

The HDL code supporting filling the memories (Instruction Memory and Data Memory) and register files from outside text files or filling it manually by the user by typing them inside their modules .These text files must be found inside the Project files directory under constant names.

There is also a GUI program with simple interface that can be simply used by any user.

GUI support converting ASSEMBLY language into machine code language, filling text files contain the data that will be stored in the Register file and the memories (Instruction Memory and Data Memory),and viewing the monitor of the internal test bench.

Group Members

Abdelrahman Ibrahim Yassin

Abdelrahman Mahmoud

Abdallah Reda Abdallah

Diaa Ahmed Abdelzaher

Samuel Medhat Farid

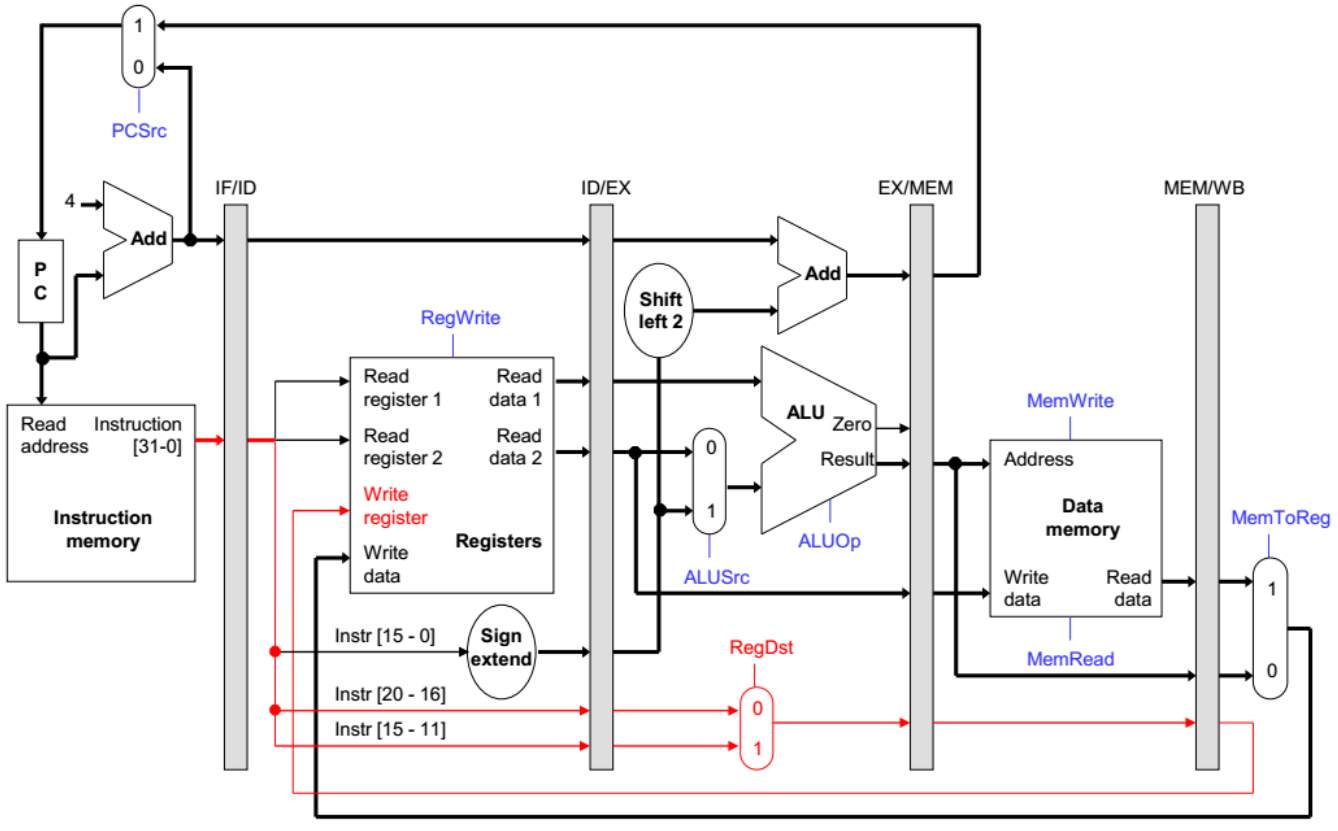
Chapter 2: Pipelining

- Pipeline Register

- We'll add intermediate registers to our pipelined data path too.
- There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big pipeline register between each stage.
- 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.



- Control Lines

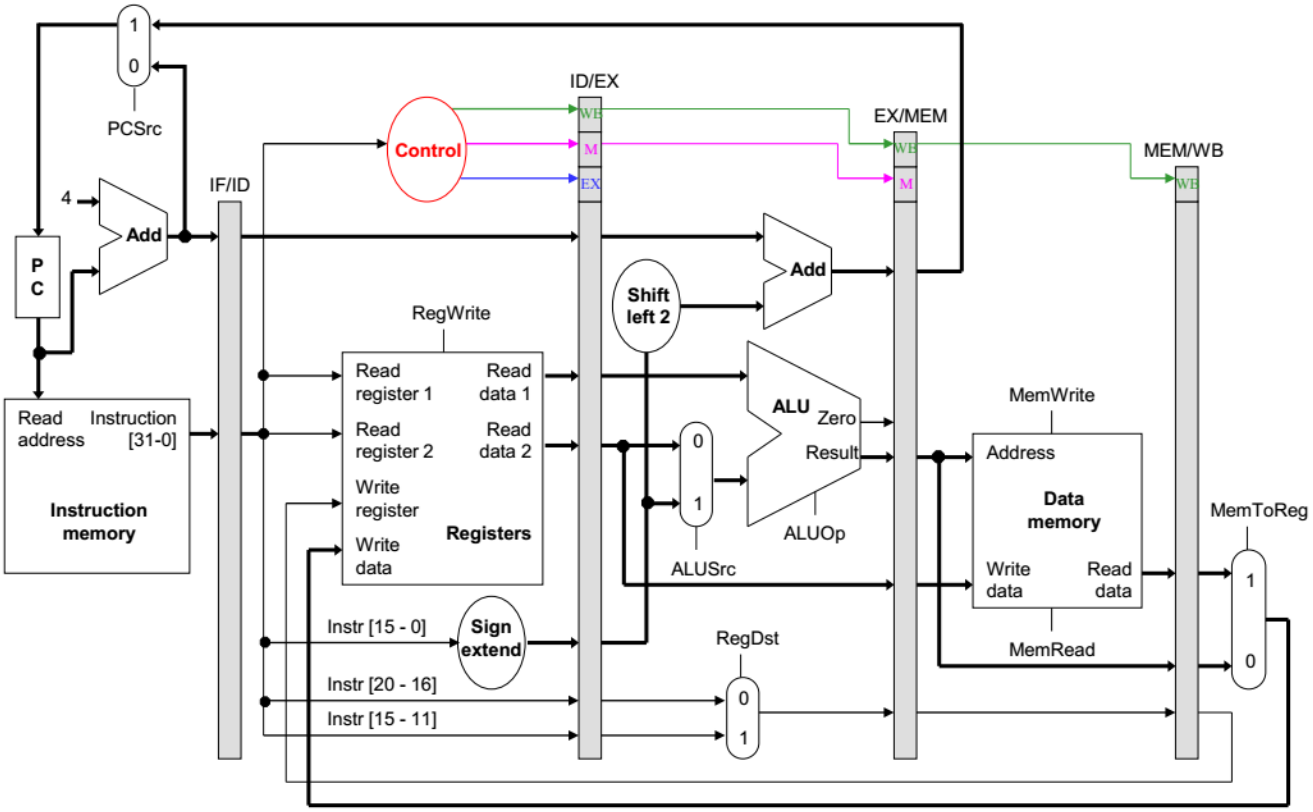
- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But just like before, some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just 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[3:0]	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

- **IN OUR** Verilog description ,Control lines are 12bits not as standard because we increase the ALUOp bit from 2bits to 4bits to cancel the ALUControlUnit in the EXE stage ,so we send the ALUOp to the ALU directly and it decides the operation should do
- Our ControlLines

  RegDst =ControlLines[0]; Jump =ControlLines[1];
  Branch =ControlLines[2]; MemRead =ControlLines[3];
  MemtoReg =ControlLines[4]; ALUOp[3:0]= ControlLines[8:5]
  MemWrite =ControlLines[9]; ALUSrc =ControlLines[10[
  RegWrite =ControlLines[11];

- Here, the Pipelined data path and control



- So, the following instructions will be executed in pipeline like illustrated in the diagram

Clock cycle

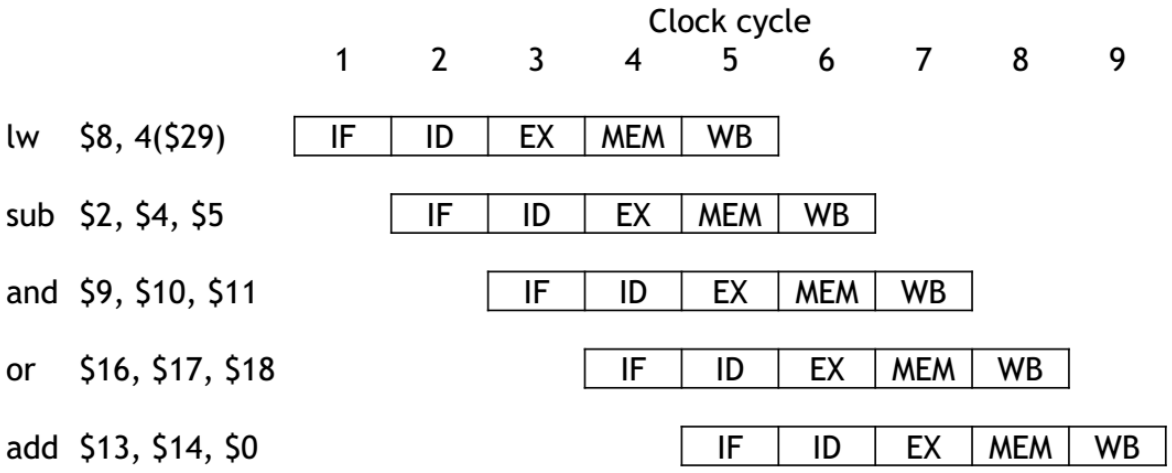
	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $t5, $t6, $0					IF	ID	EX	MEM	WB

- Summary

- The pipelined data path combines ideas from the single and multi-cycle processors that we saw earlier.
  - It uses multiple memories and ALUs.
  - Instruction execution is split into several stages.
- Pipeline registers propagate data and control values to later stages.
- The MIPS instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.
- Next chapter, we'll start talking about Hazards.

- Chapter 2: Forwarding and Data Hazards

- Now, we'll introduce some problems that data hazards can cause for our pipelined processor, and show how to handle them with forwarding.



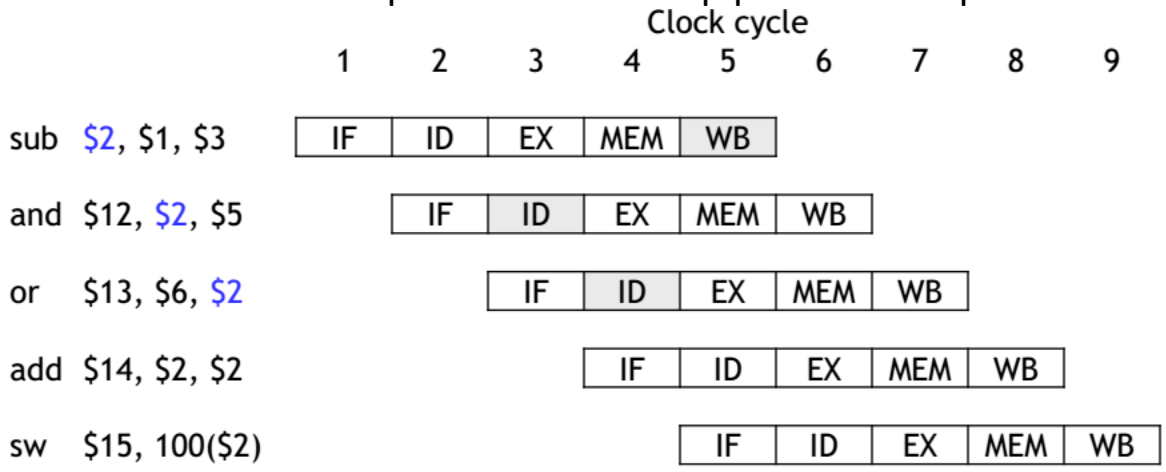
- This diagram shows the execution of an ideal code.
  - Each instruction needs a total of five cycles for execution.
  - One instruction begins on every clock cycle for the first five cycles.
  - One instruction completes on each cycle from that time on.

- The instructions in this example are independent.
  - Each instruction reads and writes completely different registers.
  - Our data path handles this sequence easily, as we saw last time.
- But most sequences of instructions are *not* independent!

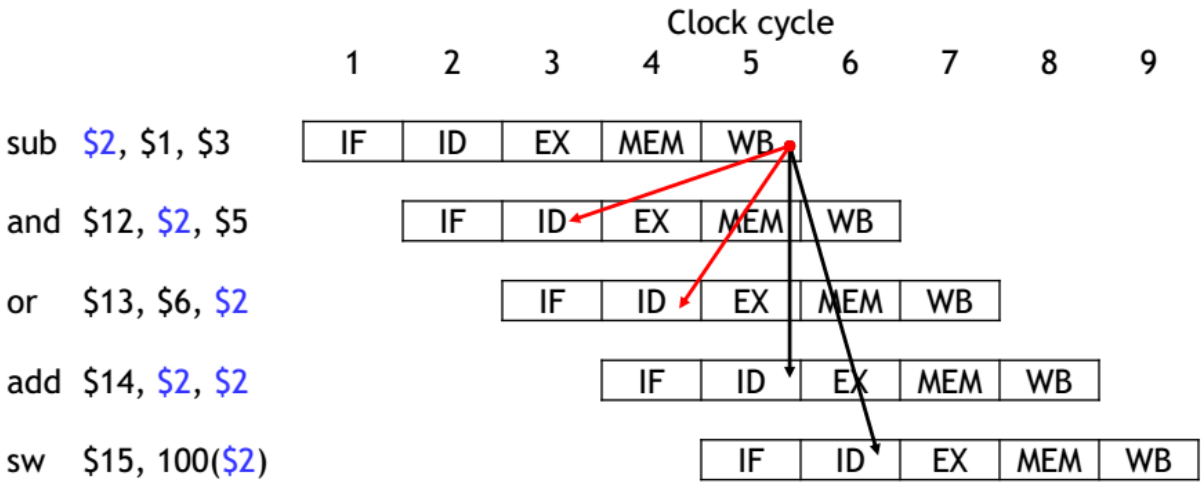
- An example with dependencies

```
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 new 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 multi cycle data paths.
  - 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 data path?

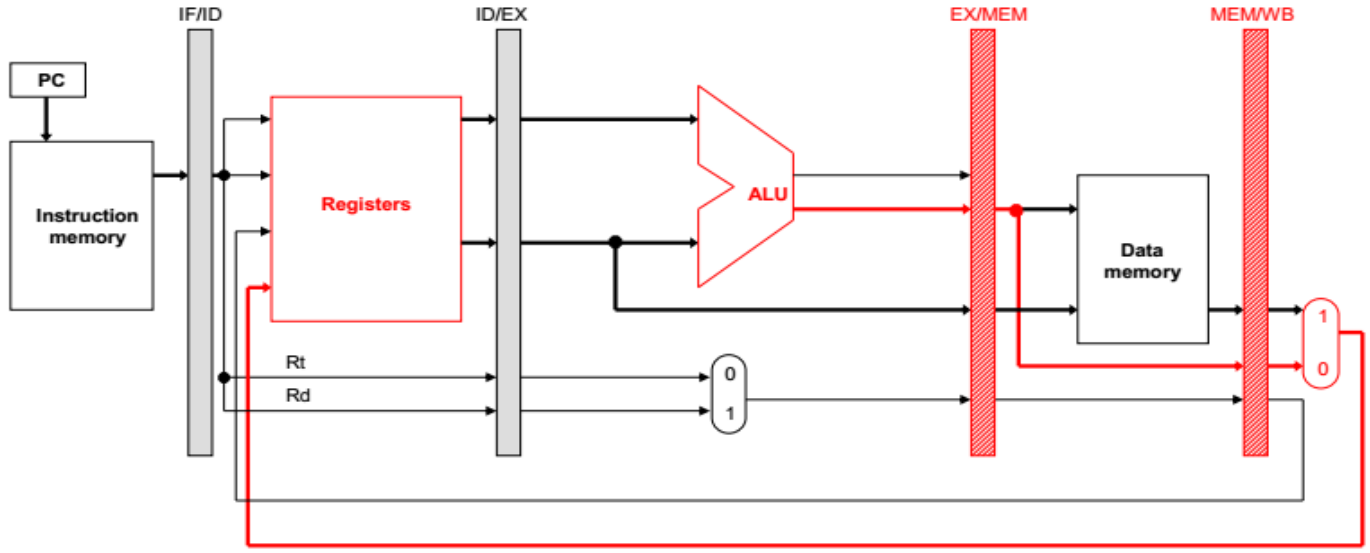


- The SUB instruction does not write to register $2 until clock cycle 5 .this causes two data hazards in our current pipelined data path.
  - The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction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



- 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 data path. Here, hazards exist between instructions 1 & 2 and 1 & 3.
- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $2.
- When is the data is actually produced and consumed?
- What can we do?

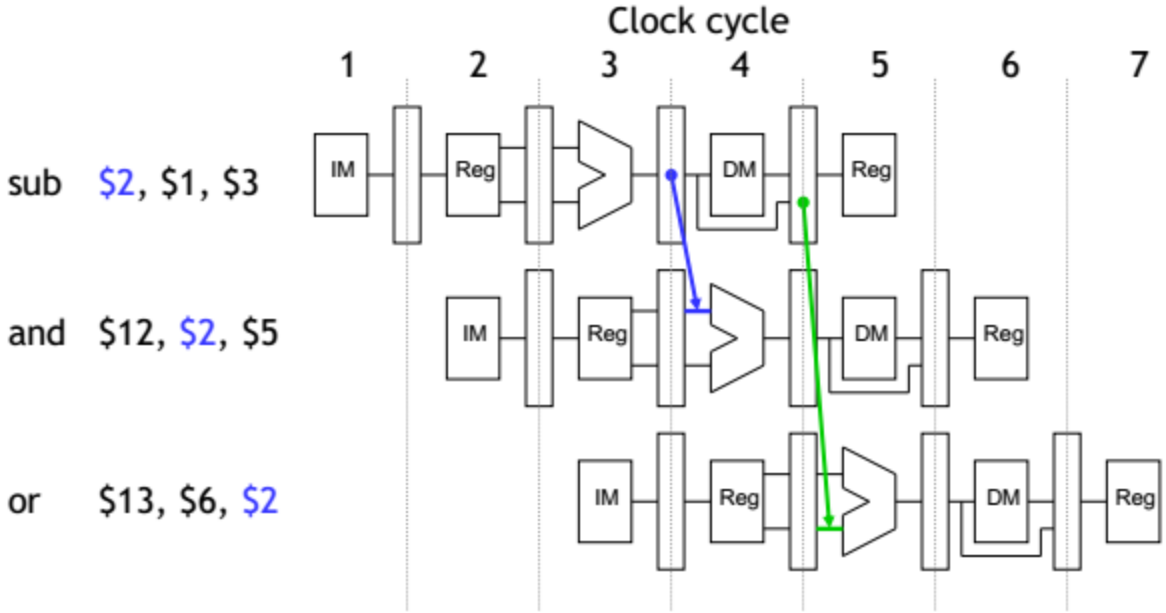
- The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.



- 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.



- A forwarding unit selects the correct ALU inputs for the EX stage.

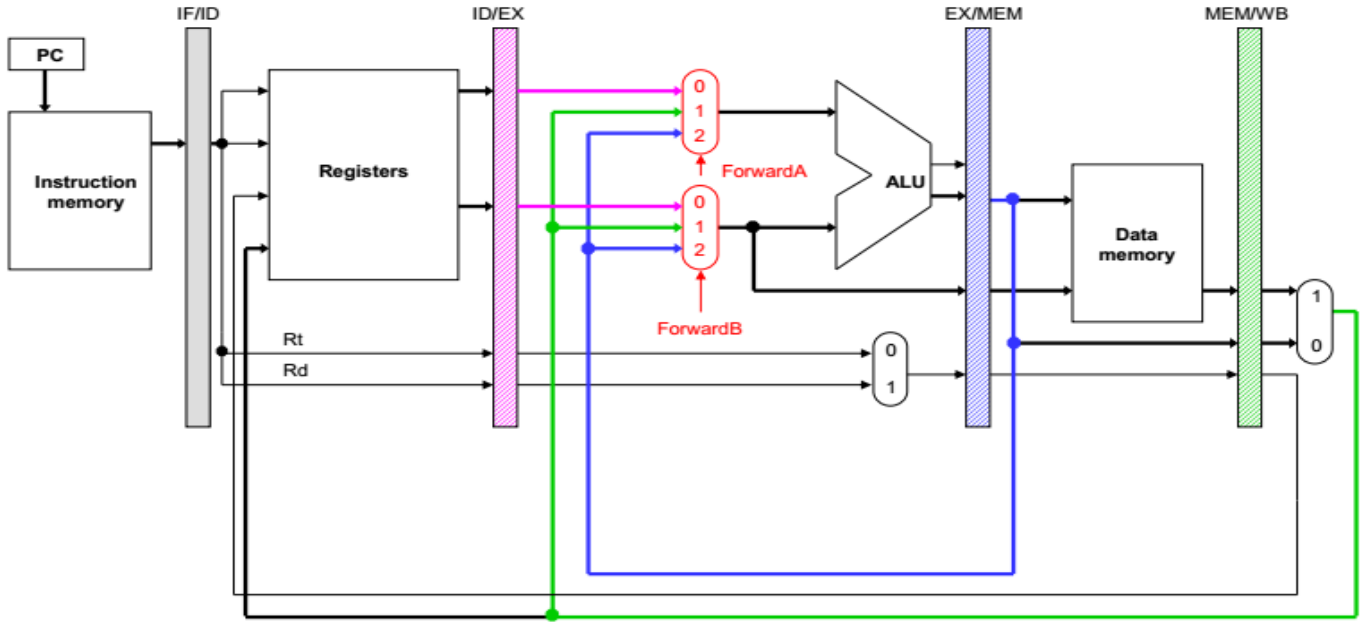
— If there is no hazard, the ALU's operands will come from the register file, just like before.

— 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.

- ForwardA signal will be one if the condition of forwarding from EXMEMALUresult to the operands of the ALU in case of read after write this happens in case of that an instruction uses a value is being modified in an instruction that before the preceding instruction it is set to 2 in case of that an instruction uses a value is being modified in an instruction that just the preceding instruction this happens in case of the first operand of the ALU it is set to be zero if there is no hazards so it will take the first operand as normal from the IDEX pipeline register.
- ForwardB signal is similar to ForwardA signal but it control the second Mux.

-The following figure shows the previous explanation



- So how can the hardware determine if a hazard exists?

- An EX/MEM 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/MEM hazard between the two instructions below.
- 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.

- The first ALU source comes from the pipeline register when necessary.
  if (EX/MEM.RegWrite = 1 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 = ID/EX.RegisterRt)
  then ForwardB = 2

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago.
- One new problem is if a register is updated twice in a row.

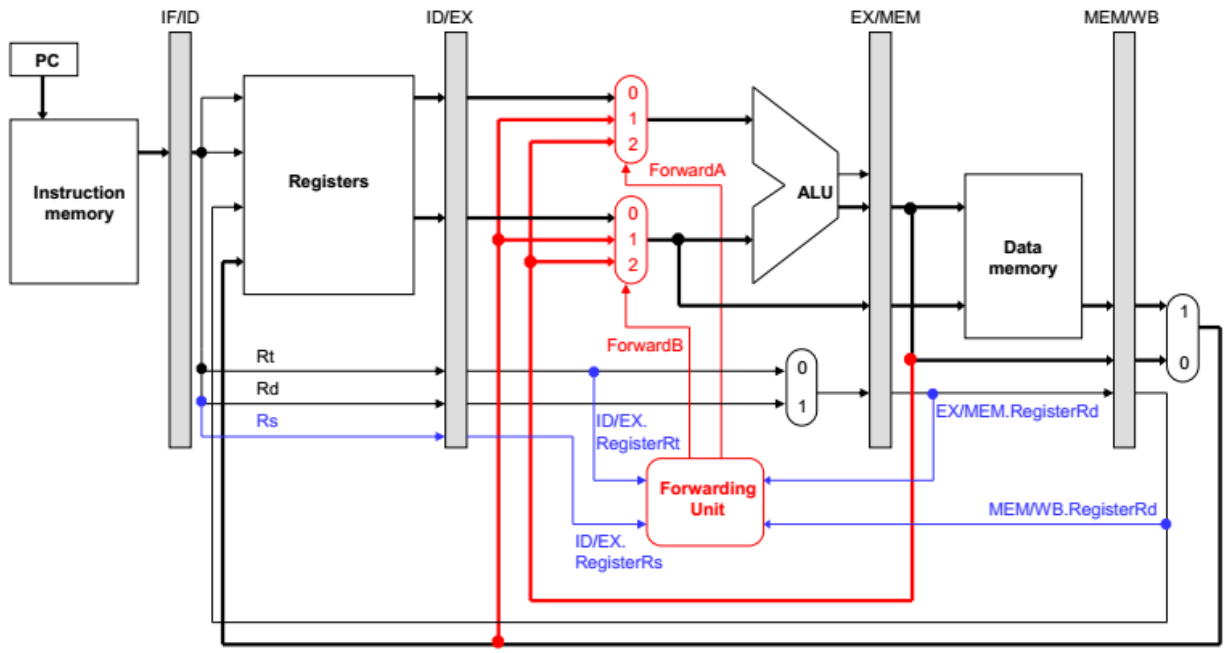
```
add $1, $2, $3
add $1, $1, $4
sub $5, $5, $1
```

- Register $1 is written by *both* of the previous instructions, but only the most recent result (from the second ADD) should be forwarded.
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

if (MEM/WB.RegWrite = 1
and MEM/WB.RegisterRd = ID/EX.RegisterRs
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegWrite = 0)
then ForwardA = 1

- The second ALU operand is handled similarly.

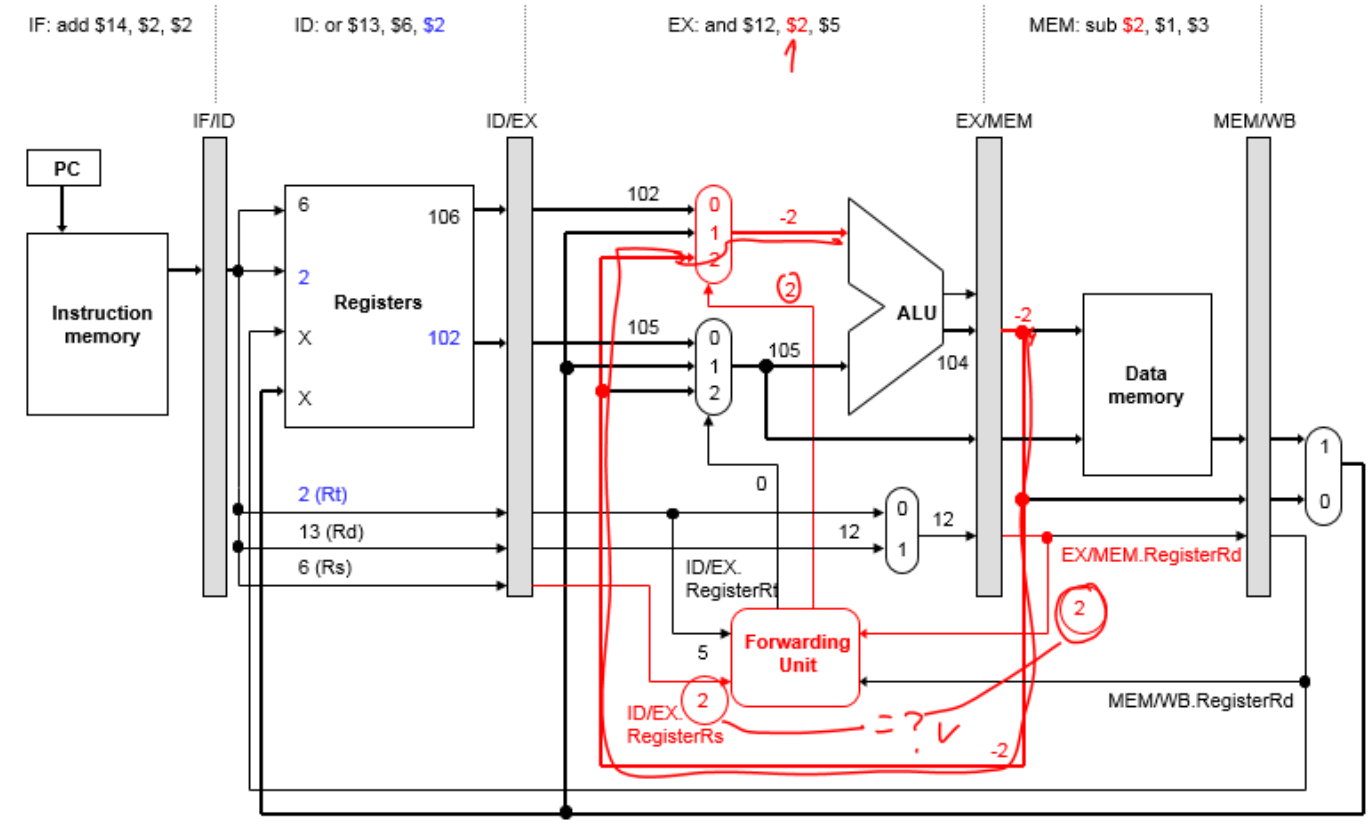
if (MEM/WB.RegWrite = 1
and MEM/WB.RegisterRd = ID/EX.RegisterRt
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt or EX/MEM.RegWrite = 0)
then ForwardB = 1



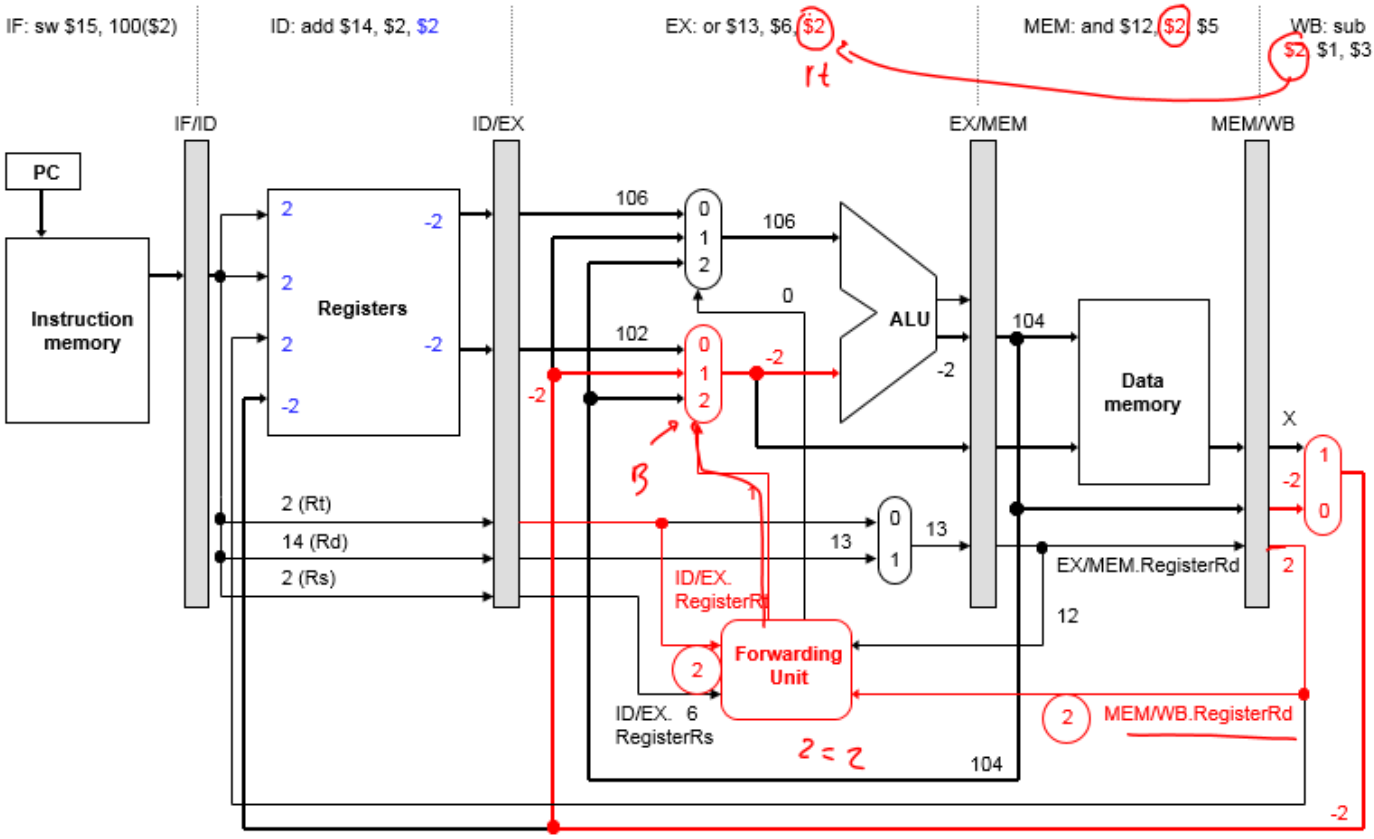
- The forwarding unit has several control signals as inputs.
  ID/EX.RegisterRs
  ID/EX.RegisterRt
  MEM/WB.RegWrite
  EX/MEM.RegisterRd
  EX/MEM.RegWrite
  MEM/WB.RegisterRd

  (The two RegWrite signals are not shown in the diagram, but they come from the control unit.)
- The fowarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.
- Some new buses route data from pipeline registers to the new muxes.

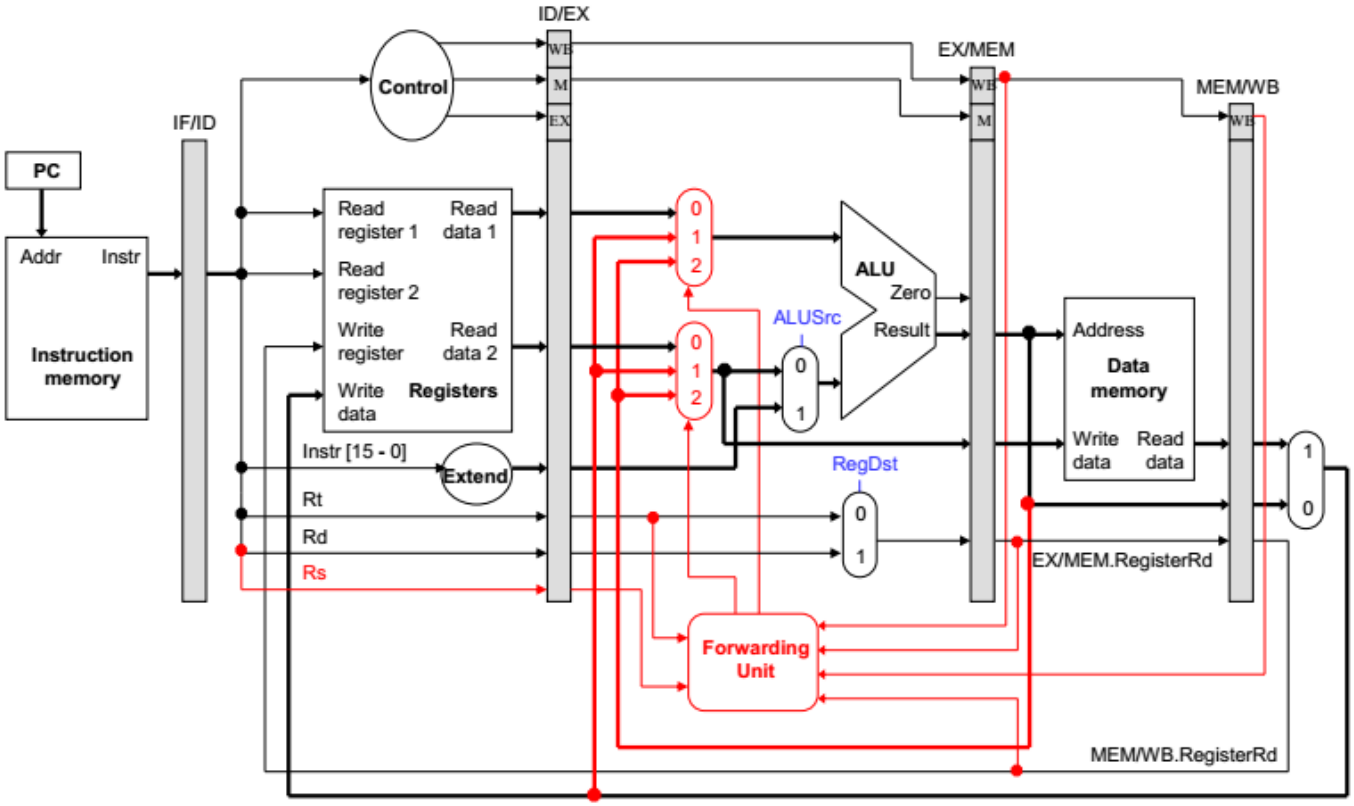
@Clk cycle 4:



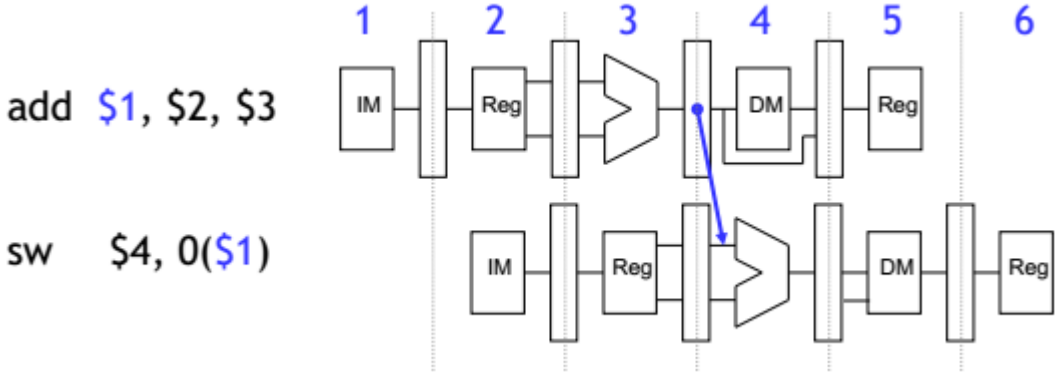
@Clk cycle 5:

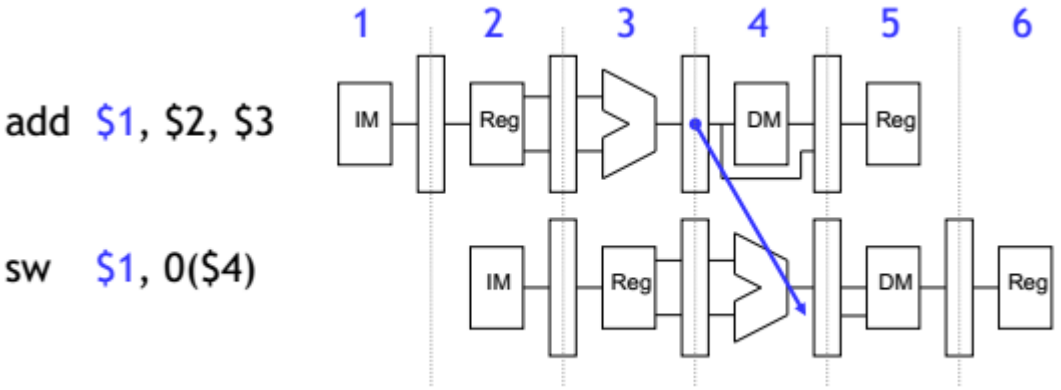


- The first data hazard occurs during cycle 4.
  - The forwarding unit notices that the ALU's first source register for the AND is also the destination of the SUB instruction.
  - The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.
- 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.

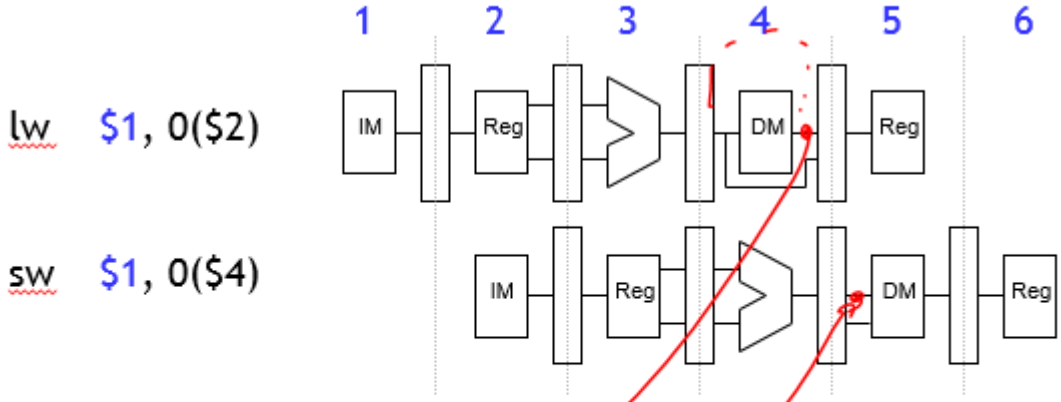


- **What about stores?**
  - Two "easy" cases: they are handled as previously illustrated



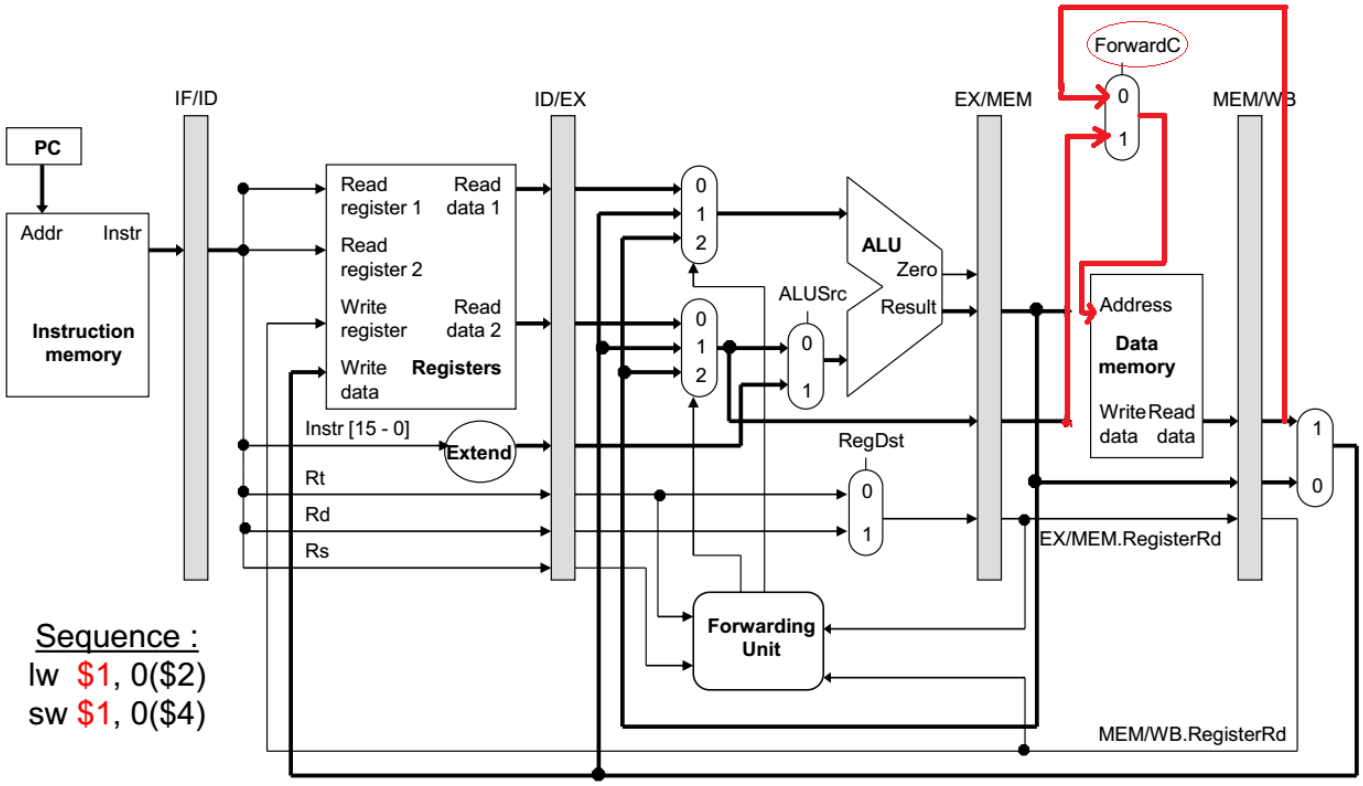


  - A harder case:



    - In what cycle is:
      - The load value available?
      - The store value needed?

- **We will Bypassing Load/Store by Extending the Data path**
  - ForwardC it used in case of a sw instruction wants to store a word in the memory and this address come from the value is being modified in another instruction just before the sw if there is hazards this signal will set to 1 to forward the value from the mem stage to exec stage to calculate the address properly.



- **Summary**
  - In real code, most instructions are dependent upon other ones.
    - This can lead to data hazards in our original pipelined data path.
    - Instructions can't write back to the register file soon enough for the next two instructions to read.
  - Forwarding eliminates data hazards involving arithmetic instructions.
    - The forwarding unit detects hazards by comparing the destination registers of previous instructions to the source registers of the current instruction.
    - Hazards are avoided by grabbing results from the pipeline registers *before* they are written back to the register file.
  - Next, we'll finish up pipelining.
    - Forwarding can't save us in some cases involving lw.
    - We still haven't talked about branches for the pipelined data path.

- Summarize the forwarding conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

- Chapter 3: Stalls and flushes

- 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 write back 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.
  - Branches affect the instruction fetch for the next clock cycle.
- In both of these cases we may need to slow down, or stall, the pipeline.

- Detecting stall is much like detecting data hazards.
- Recall the format of hazard detection equations:

  if (EX/MEM.RegWrite = 1 and EX/MEM.RegisterRd = ID/EX.RegisterRs) then Bypass Rs from EX/MEM stage latch

- 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.

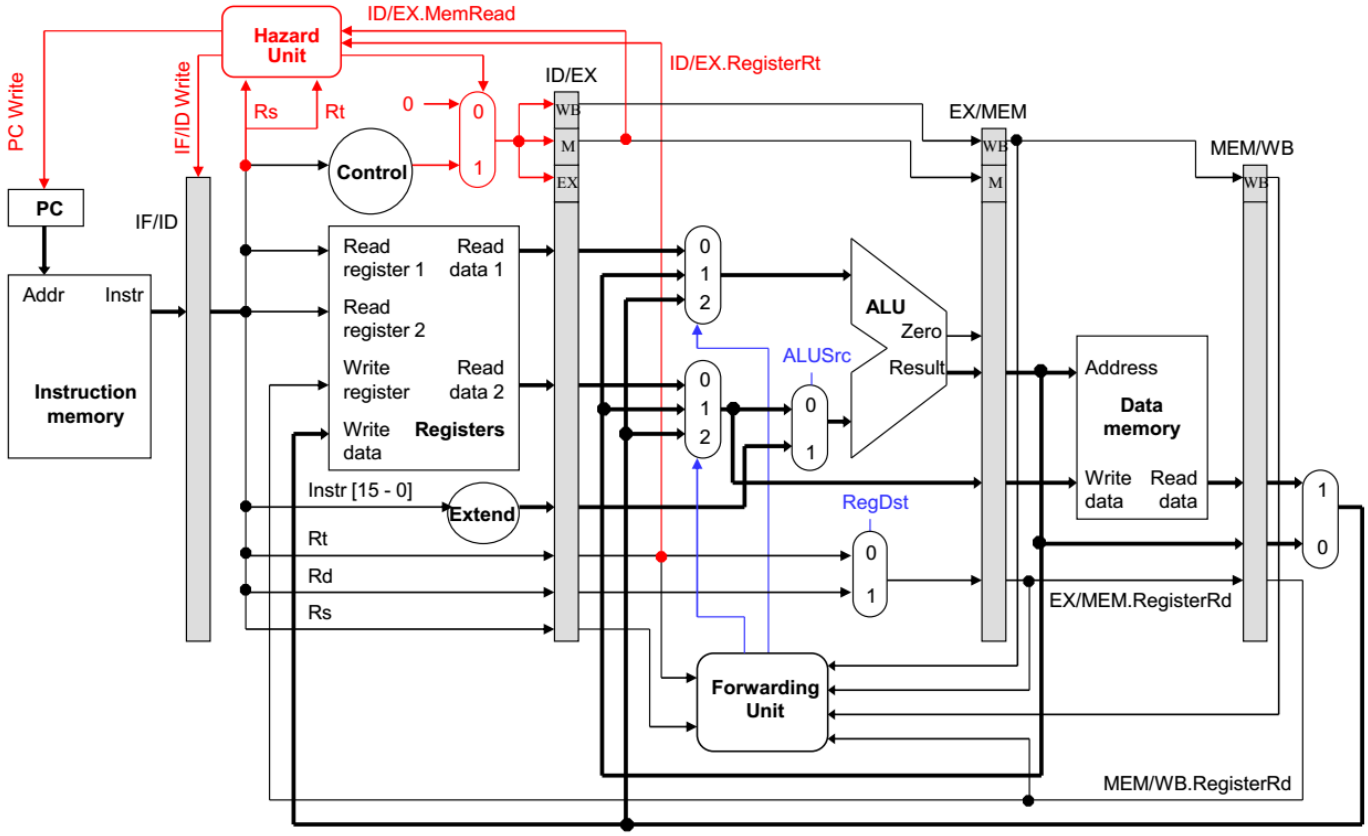
ID/EX.RegisterRt = IF/ID.RegisterRs or ID/EX.RegisterRt = IF/ID.RegisterRt

- The complete test for stalling is the conjunction of these two conditions.

  if (ID/EX.MemRead = 1 and (ID/EX.RegisterRt = IF/ID.RegisterRs orID/EX.RegisterRt = IF/ID.RegisterRt)) then stall

- How to stall the pipeline ?
  - Force control values in ID/EX registerto 0 so EX, MEM and WB do nop (no-operation)
  - Prevent update of PC and IF/ID register
  - Using instruction is decoded again so Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw Can subsequently forward to EX stage

- Adding hazard detection to the CPU



- The hazard detection unit's inputs are as follows.
  - IF/ID.RegisterRs and IF/ID.RegisterRt, the source registers for the current instruction.
  - ID/EX.MemRead and ID/EX.RegisterRt, to determine if the previous instruction is LW and, if so, which register it will write to.
- By inspecting these values, the detection unit generates three outputs.
  - Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls or continues.
  - A mux select for a new multiplexer, which forces control signals for the current EX and future MEM/WB stages to 0 in case of a stall.

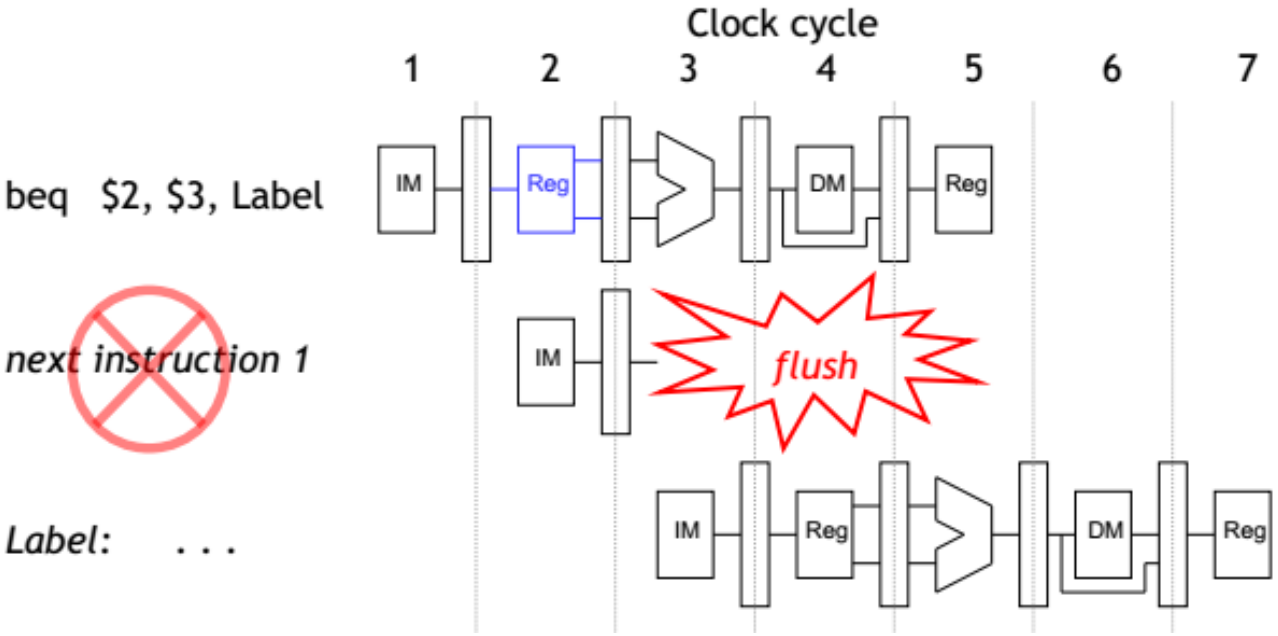
- Branch prediction

- We can 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 miss prediction.



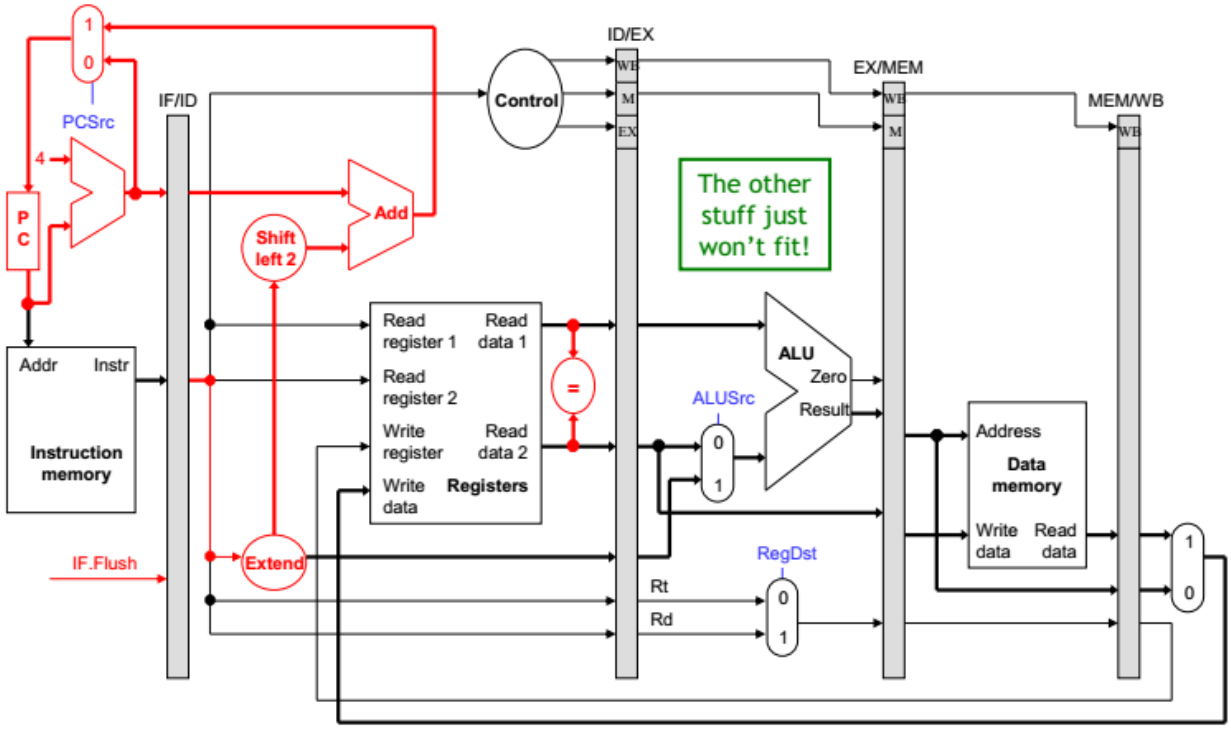
- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.

— MIPS uses sll $0, $0, 0 as the nop instruction.

— This happens to have a binary encoding of all 0s: 0000 .... 0000.

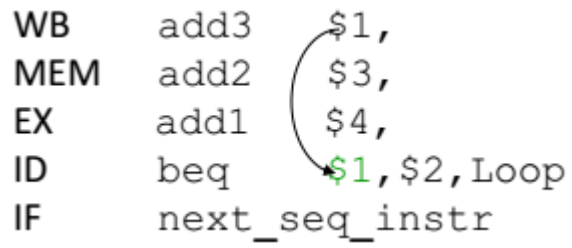
- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.
- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.

-We assume in our Verilog description that the Branch is not taken

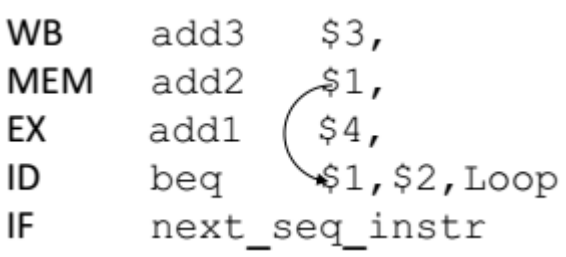


ID Branch Forwarding Issues

- MEM/WB "forwarding" is taken care of by the normal RegFile write before read operation.



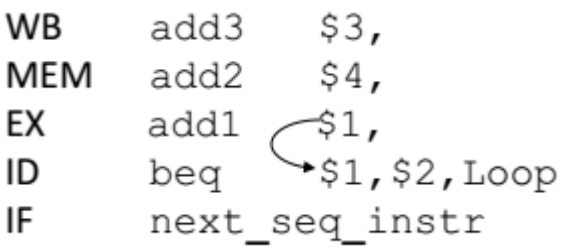
- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like



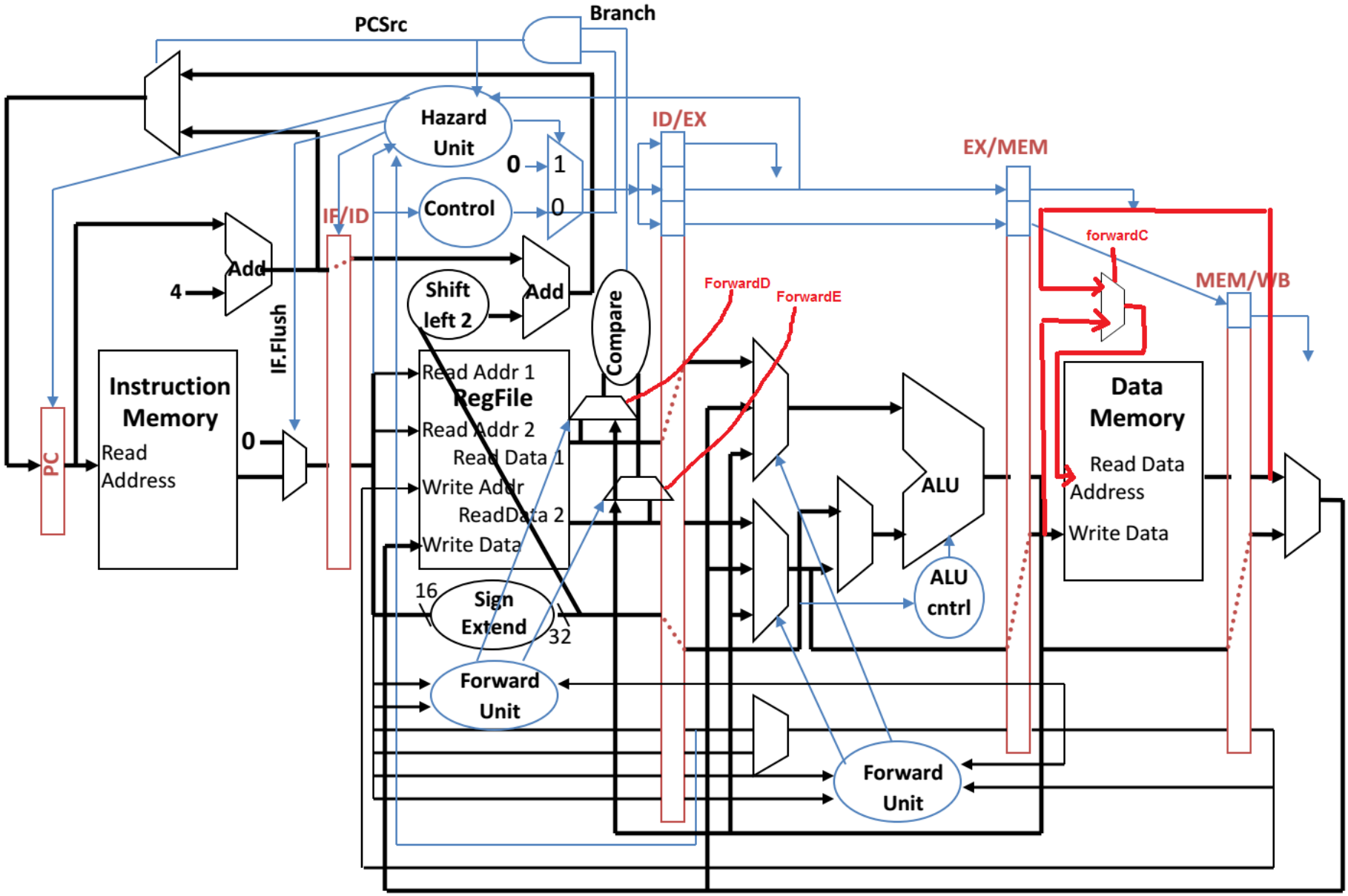

```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
ForwardD = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
ForwardE = 1
```

Forwards the result from the second previous instr. to either input of the compare

- If the instruction immediately before the branch produces one of the branch source operands, then a stall needs to be inserted (between the beq and add1) since the EX stage ALU operation is occurring at the *same time* as the ID stage branch compare operation



Supporting ID Stage Branches



- Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall.

- If taken, flush instructions after the branch (earlier in the pipeline)
  - In IF, ID, and EX stages if branch logic in MEM – three stalls
  - In IF and ID stages if branch logic in EX – two stalls
  - in IF stage if branch logic in ID – one stall

- Ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))

- Restart the pipeline at the branch destination

- Predict not taken works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
            .
            .
            .
      last loop instr
      j  Loop
Out:  fall out instr
```

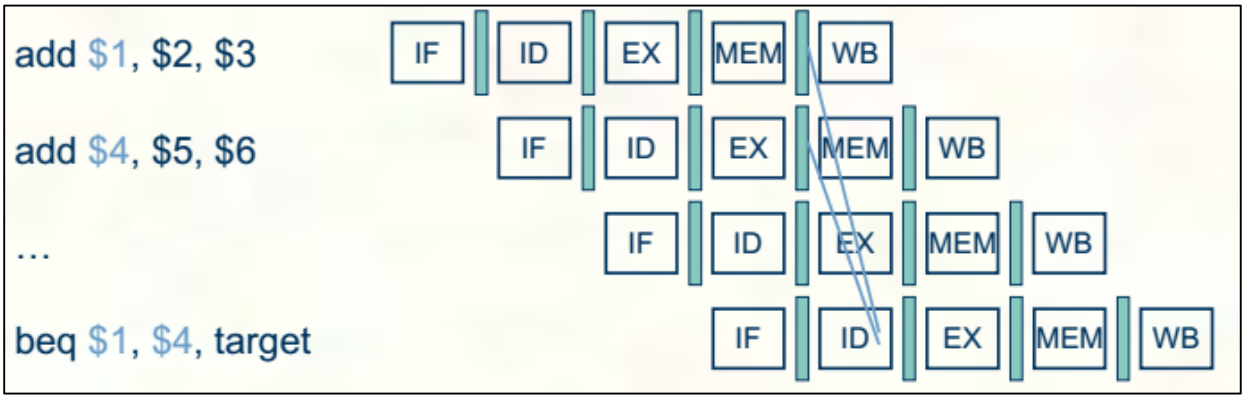
- Predict not taken doesn't work well for "bottom of the loop" branching structures

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

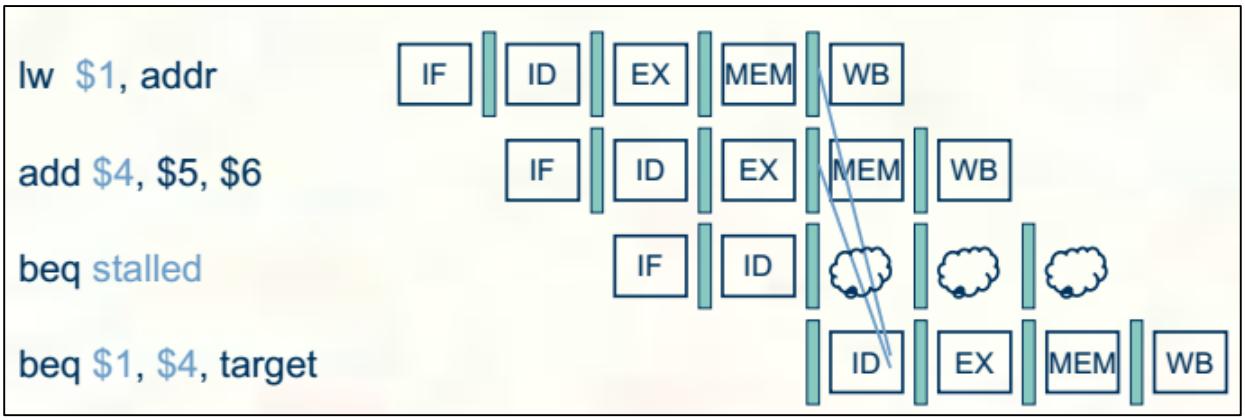
We could have implement the Dynamic branch prediction idea but our time lost very quickly (but who cares?) May be implementing it in other coming projects ISA.

Supported Data Hazards for Branches

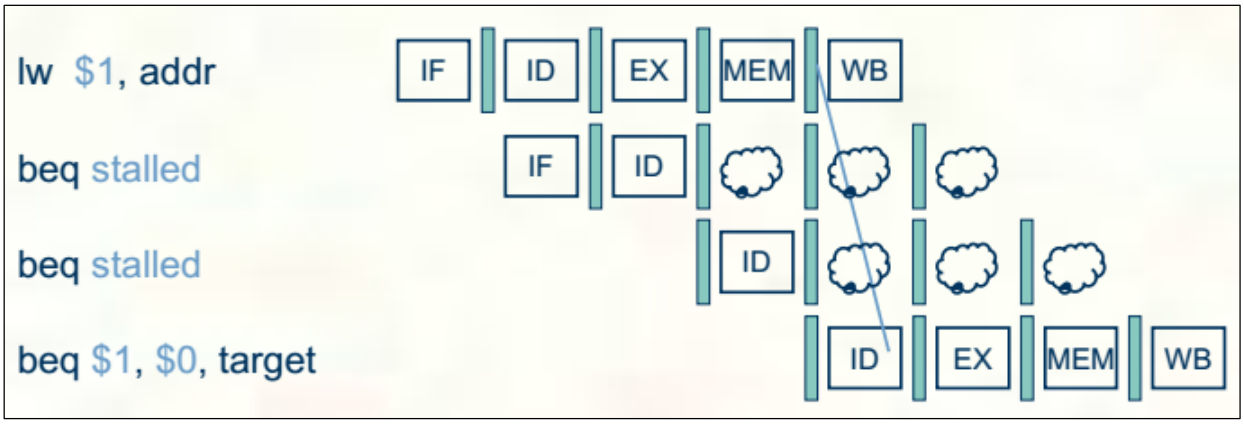
Case1: If a comparison register is a destination of 2nd or 3rd preceding ALU instruction can resolve using forwarding



Case2: If a comparison register is a destination of 2nd or 3rd preceding ALU instruction **needs one stall**

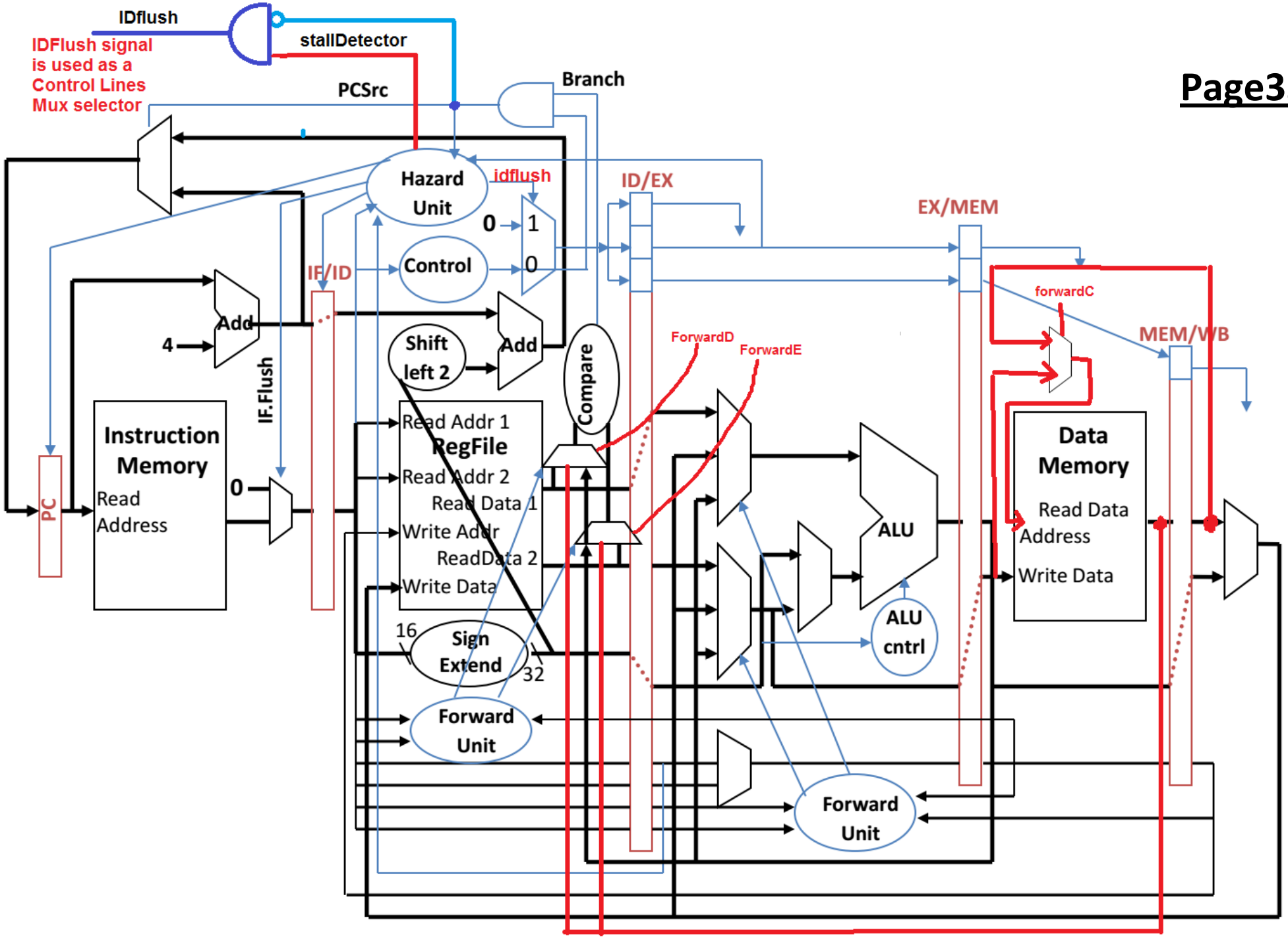


Case3: If a comparison register is a destination of immediately preceding load instruction **needs two stalls**



- Summary

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.


IDflush
IDFlush signal is used as a Control Lines Mux selector
stallDetector
PCSrc
Branch
Hazard Unit
idflush
ID/EX
EX/MEM
MEM/WB
Control
IF/ID
Add
4
Shift left 2
Add
Compare
ForwardD
ForwardE
forwardC
IF.Flush
PC
Instruction Memory
Read Address
Read Addr 1
RegFile
Read Addr 2
Read Data 1
Write Addr
ReadData 2
Write Data
16
Sign Extend
32
ALU
ALU cntrl
Data Memory
Read Data
Address
Write Data
Forward Unit
Forward Unit