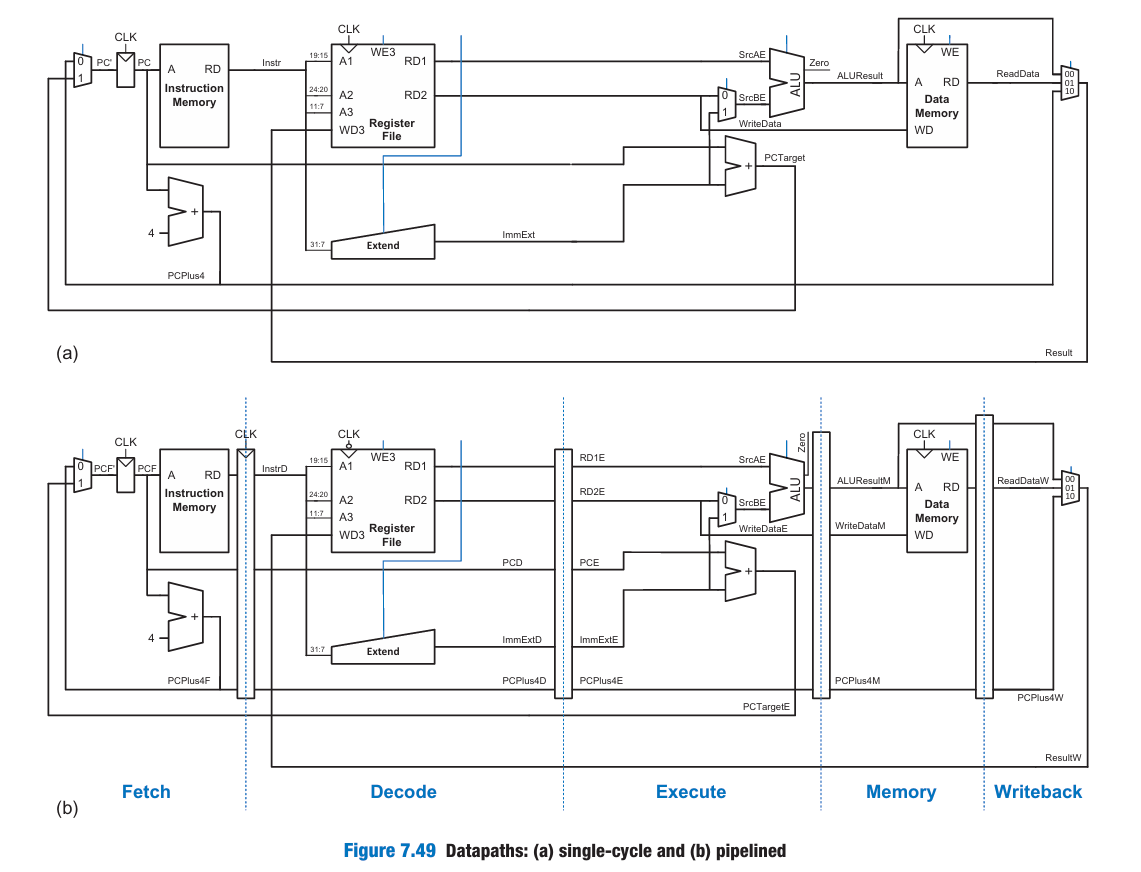
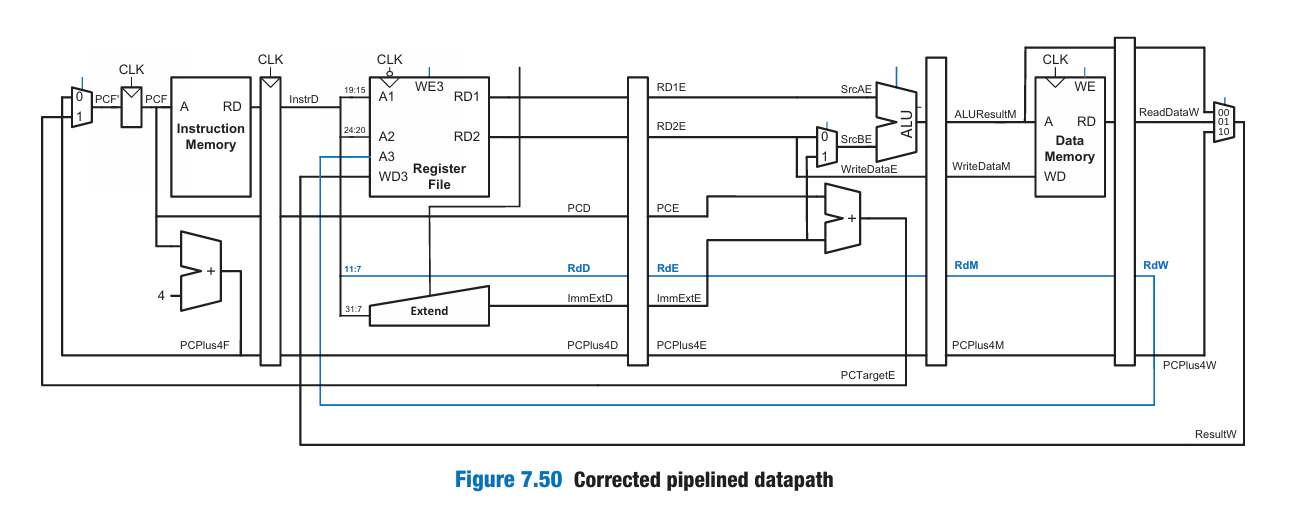
We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Ideally, the latency of each instruction is unchanged, but the throughput is five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. In the Fetch stage, the processor reads the instruction from instruction memory. In the Decode stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the Execute stage, the processor performs a computation with the ALU. In the Memory stage, the processor reads or writes data memory, if applicable. Finally, in the Writeback stage, the processor writes the result to the register file, if applicable. A central challenge in pipelined systems is handling hazards that occur when one instruction’s result is needed by a subsequent instruction before the former instruction has completed.

PIPELINED DATAPATH:



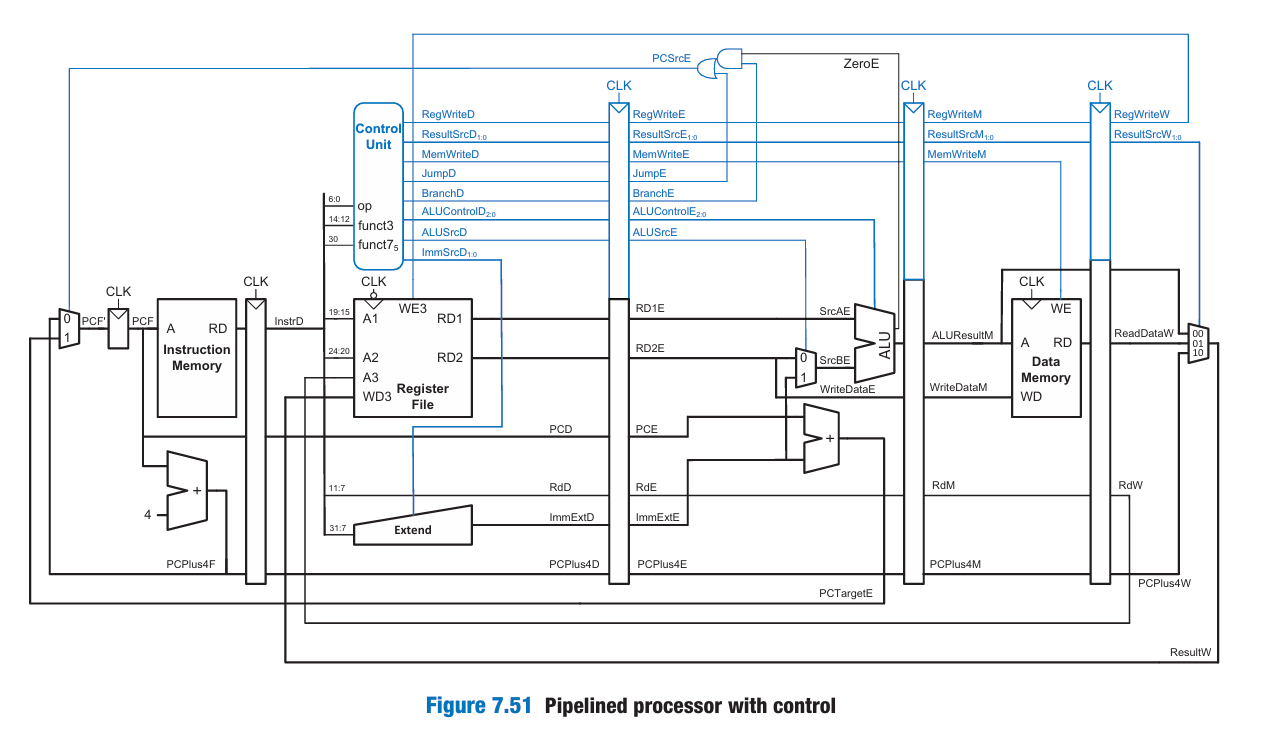
1. A3 comes from Instr[11:7] in decode stage but ResultW goes to WD3 in writeback stage. So when writeback happens for 1 instr it will take the A3 of some other instr causing a hazard.
2. The logic to produce PCF’ (the next PC) is also problematic because it could be updated with either a Fetch or an Execute stage signal (PCPlus4F or PCTargetE). This control hazard will be fixed later.



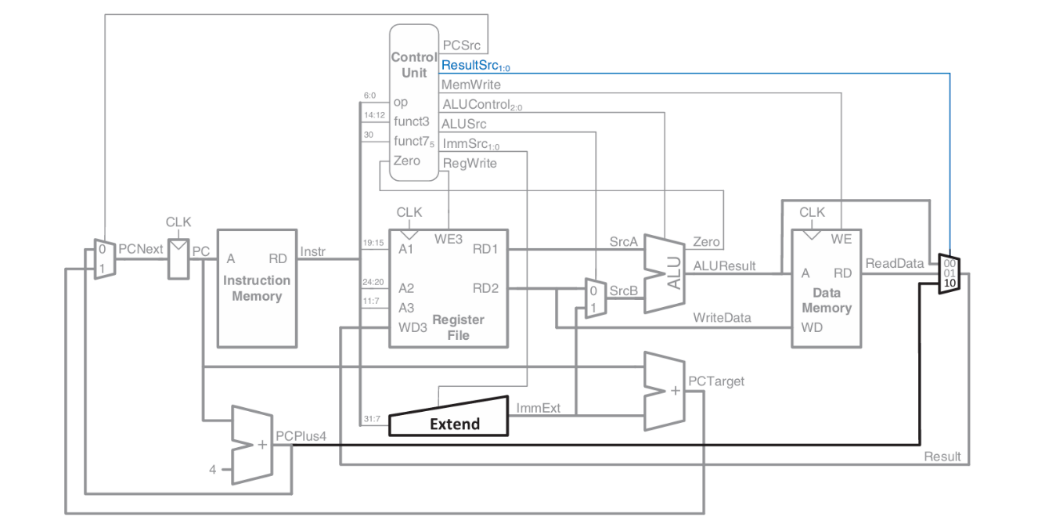
The Rd signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction. RdW and ResultW are fed back together to the register file in the Writeback stage.

The pipelined processor uses the same control signals as the single-cycle processor and, therefore, has the same control unit. But the control signals are also pipelined.

Pipelined Processor with Control



Single Cycle with Control



HAZARDS

The pipelined processor uses the same control signals as the single-cycle processor and, therefore, has the same control unit. not yet completed, a hazard occurs. The register file is written during the first half of the cycle and read during the second half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

A software solution would be to require the programmer or compiler to insert nop instructions between such instructions so that the dependent instruction does not read the result until it is available in the register file. Such a software interlock complicates programming and degrades performance, so it is not ideal.

We should be able to forward the result from one instruction to the next to resolve the RAW(read after write) hazard without waiting for the result to appear in the register file and without slowing down the pipeline. In other situations explored later in this section, we may have to stall the pipeline to give time for a result to be produced before the subsequent instruction uses the result.

A data hazard occurs when an instruction tries to read a register that has not yet been written back by a previous instruction.

A control hazard (also called a *branch hazard*) occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. Eg. 1000: beq x1, x2, +8

This means:

If x1 == x2, jump to instruction at 1000 + 8 = 1008  
Else, go to 1004 (normal next PC)

But when you start fetching instruction at 1004, you haven’t yet compared x1 and x2 — that happens later. So maybe you fetch the wrong instruction, and will have to flush it once you know the correct path.

FORWARDING

add s8, s1, s2 // Instruction 1

sub s3, s8, s4 // Instruction 2

or s5, s6, s8 // Instruction 3

and s7, s8, s9 // Instruction 4

* s8 is calculated in Instruction 1.
* The next 3 instructions immediately use s8.
* Without forwarding, you'd have to stall the pipeline and wait for s8 to reach the Writeback stage before using it. Instead of stalling, we "forward" the result of the previous instruction early from a later pipeline stage.

When to Forward?

If the current instruction (in the EX stage) needs a register value that:

* Is not yet in the register file (because it will be written later),
* But the value is already available in Memory stage or Writeback stage of a previous instruction, then we forward it using multiplexers.

Forwarding Multiplexers

You place MUXes in front of the ALU inputs (SrcA and SrcB), and the MUX chooses:

* From the Register File (normal case)
* Or from ALUResult in Memory stage (ALUResultM)
* Or from result in Writeback stage (ResultW)

if ((Rs1E == RdM) & RegWriteM) & (Rs1E != 0)

ForwardAE = 10; // Forward from Memory stage

else if ((Rs1E == RdW) & RegWriteW) & (Rs1E != 0)

ForwardAE = 01; // Forward from Writeback stage

else

ForwardAE = 00; // No forwarding, use RF

Let’s explain it line by line:

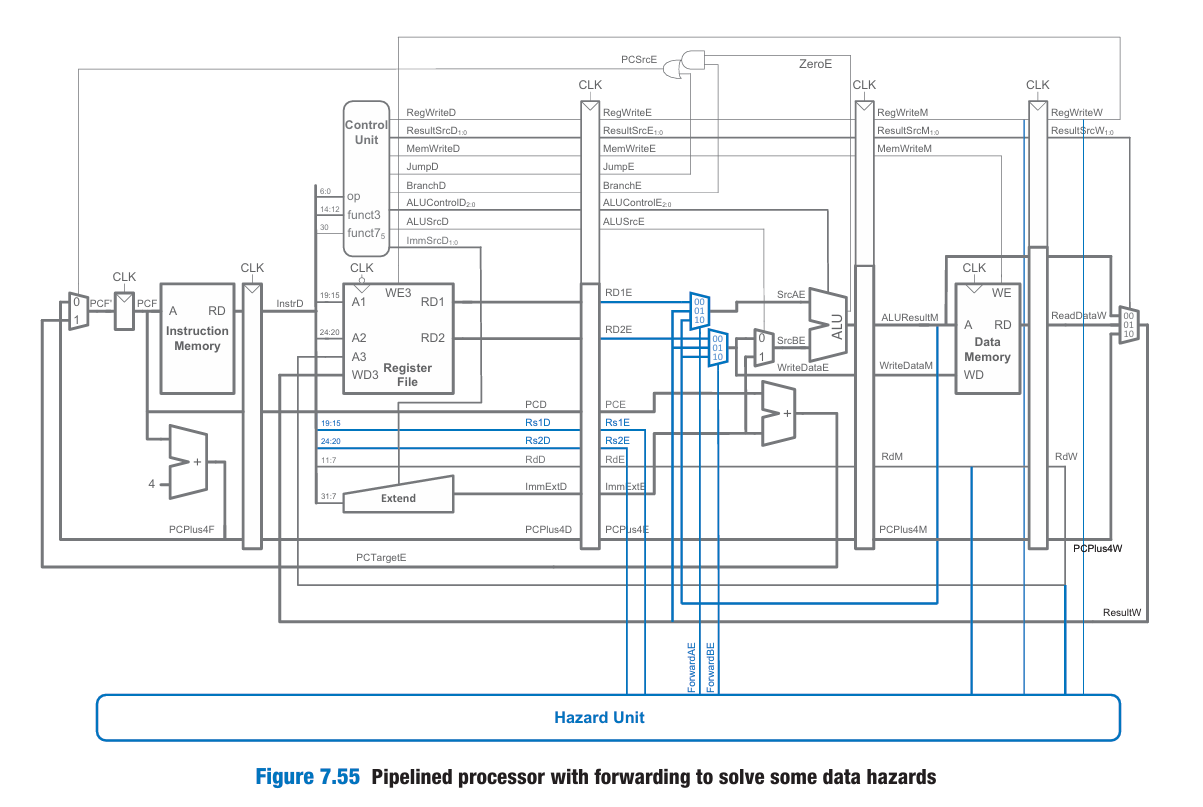
* Rs1E = Source register 1 from instruction in EX stage
* RdM = Destination register from instruction in MEM stage
* RdW = Destination register from instruction in WB stage
* RegWriteM / RegWriteW = Whether those instructions actually write to the register file
* Rs1E != 0 = Never forward if register is x0, because it's always 0

Interpretation:

| Condition | Action | Reason |
| --- | --- | --- |
| Rs1E == RdM and RegWriteM and not x0 | Forward from MEM stage | The most recent valid result is in MEM |
| Else if Rs1E == RdW and RegWriteW and not x0 | Forward from WB stage | If MEM stage doesn't match, but WB does |
| Else | Use Register File | No hazard, read value from register file |

Summary

* Data hazards occur when an instruction uses a register result before it is written back.
* Forwarding avoids stalls by bypassing the needed value from a later pipeline stage.
* Forwarding logic is implemented using:
  + Comparisons between source and destination registers
  + Control signals (ForwardAE, ForwardBE) to drive the MUX select lines
  + If both the Memory and Writeback stages contain matching destination reg isters, then the Memory stage should have priority because it contains the more recently executed instruction.



STALLING:

1. When forwarding works (Execute-to-Execute RAW):

add s7, s1, s2 // Instruction 1

sub s3, s7, s4 // Instruction 2 (uses s7)

* add writes result at end of EX stage (cycle 3).
* sub reads operand in its EX stage (cycle 4).
* Since both happen in EX stage, the data is ready in time.
* So the ALU result from add can be forwarded to the ALU of sub.

2. When forwarding fails (Load-to-Execute RAW):

lw s7, 0(x5) // Instruction 1 (load from memory)

and s3, s7, s4 // Instruction 2 (uses s7)

* lw fetches memory data in MEM stage (cycle 4), late.
* and needs s7 value in EX stage (also cycle 4), early.

This causes a conflict:

* lw hasn’t gotten the data yet (gets it *at the end* of cycle 4),
* But and wants to use it now (needs it *at the start* of cycle 4).

This situation is why we say:

“The lw instruction has a two-cycle latency.”

* It completes the memory access at the end of MEM stage.
* The next instruction can't use the loaded value until two cycles later — so we must insert a stall.

A solution is to stall the pipeline, holding up operation until the data is available. Note that the Execute stage is unused in cycle 4. Likewise, Memory is unused in cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a bubble, which behaves like a nop instruction. When we stall Decode (ID), the pipeline register between ID → EX (called ID/EX) is flushed. Flushing = setting control signals (like RegWrite, MemWrite, ALUSrc, etc.) to 0. So the EX stage receives a blank instruction: it computes nothing and writes nothing. This effectively makes it a nop. Stalling a stage is performed by disabling its pipeline register (i.e., the register to the left of a stage) so that the stage’s inputs do not change. When a stage is stalled, all previous stages must also be stalled so that no subsequent instructions are lost. So we don’t fetch or decode new instructions while the hazard is unresolved. The pipeline register directly after the stalled stage must be cleared (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary. Each stall wastes a cycle. If we stall often, throughput drops, and performance suffers.

Conditions to stall:

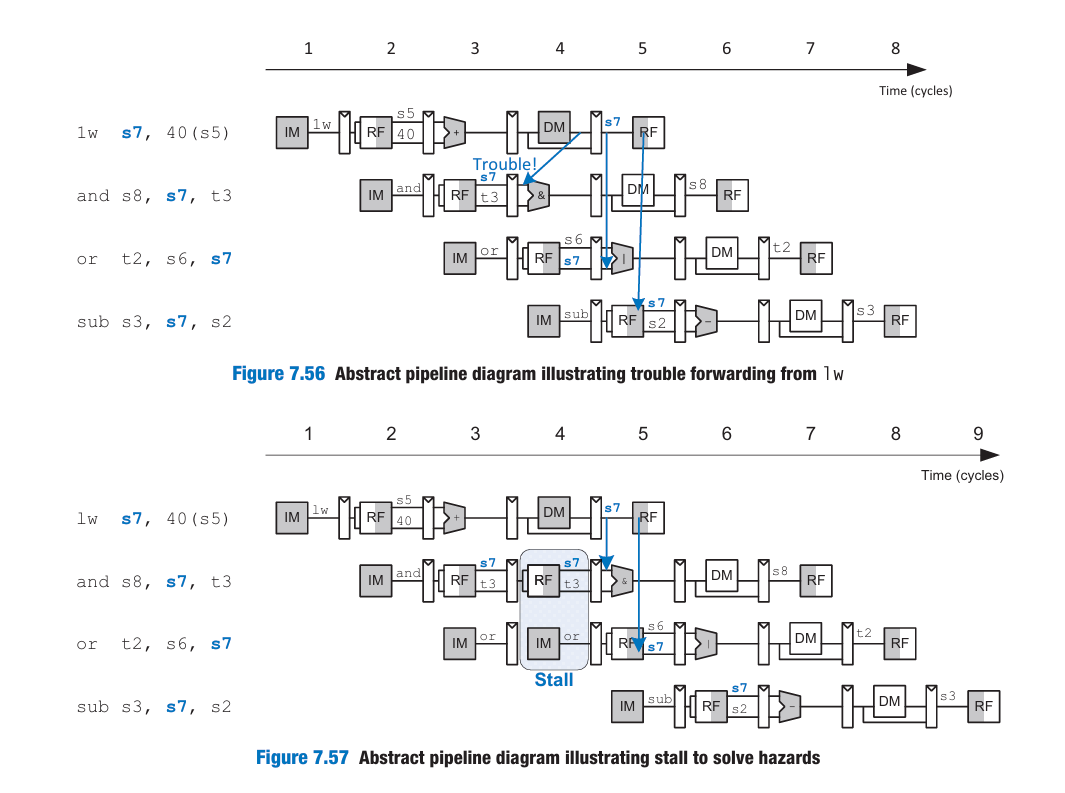
1. There is a lw instruction in the Execute stage  
   → Indicated by ResultSrcE0 == 1  
   (means this instruction needs to get its result from memory)
2. The next instruction (currently in the Decode stage) uses the destination register of that lw  
   → That means Rs1D == RdE or Rs2D == RdE  
   (i.e., the Decode instruction reads from the register that lw will write)

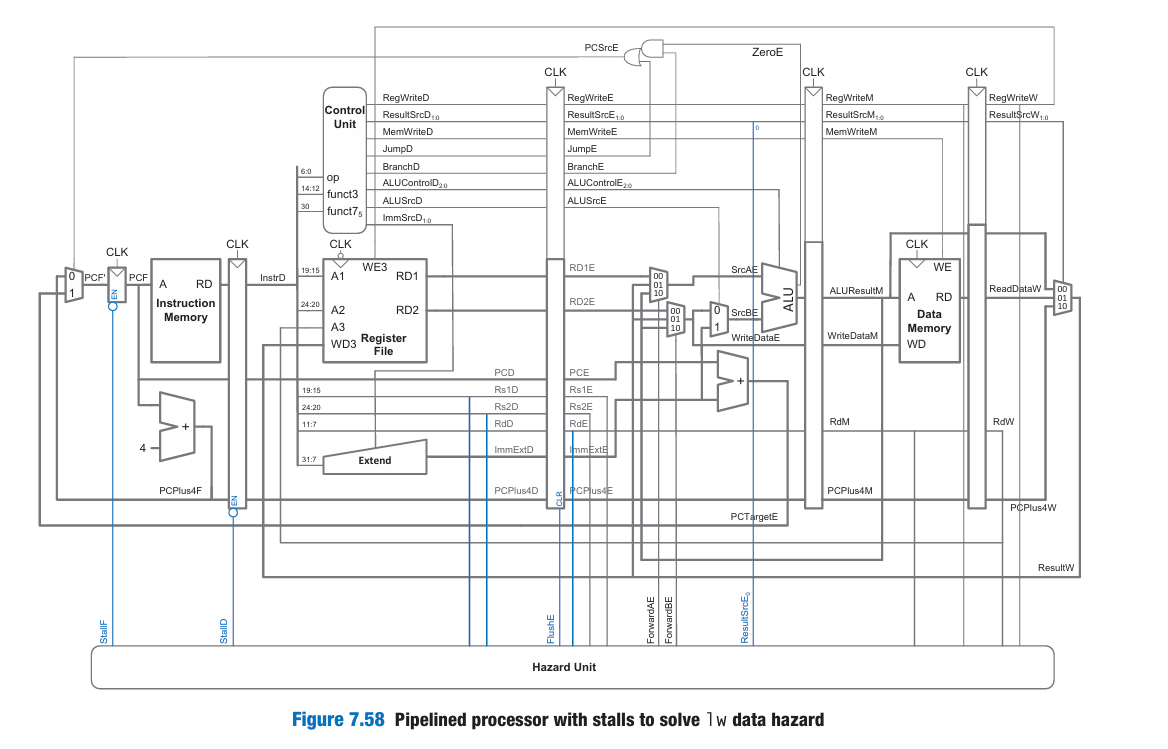
lwStall = ResultSrcE0 & ((Rs1D == RdE) | (Rs2D == RdE));

* ResultSrcE0 = 1 → means current instruction in EX is lw
* Rs1D / Rs2D = source registers of instruction in Decode stage
* RdE = destination register of instruction in EX stage

"If a load in EX stage is about to write to a register that the Decode instruction wants to use, then stall!"

|  |  |
| --- | --- |
| StallF | Freeze IF/ID → keep fetching same instruction (don’t update PC) |
| StallD | Freeze Decode stage → keep decoding same instruction |
| FlushE | Clear the Execute stage → insert a bubble (nop) |





CONTROL HAZARDS

| Cycle | IF | ID | EX | MEM | WB |
| --- | --- | --- | --- | --- | --- |
| 1 | beq |  |  |  |  |
| 2 | next | beq |  |  |  |
| 3 | ??? | next | beq |  |  |

* In Cycle 3, we have already fetched the next instruction, but:
* We don’t yet know if the branch will be taken or not.
* That decision (PCSrcE) only happens in the EX stage of beq in Cycle 3.
* So the instruction fetched in Cycle 3 may be wrong.

Naive Solution: Stalling

* Don’t fetch any new instruction until beq reaches the EX stage. That takes 2 cycles (IF → ID → EX).

Impact:

* Every branch stalls the pipeline by 2 cycles.
* Branches happen often (e.g., in loops and conditionals). So performance drops significantly.

Smarter Solution: Branch Prediction

Instead of stalling, we guess: Is the branch going to be taken or not? Based on that guess, we start executing the corresponding path.

Static Prediction

“Always predict that the branch is not taken.”

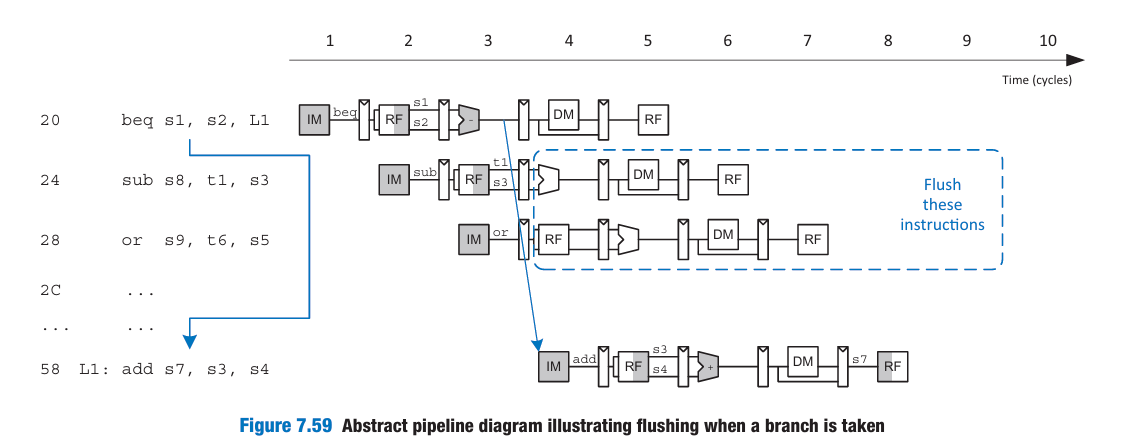
So: Keep incrementing the PC as usual (PC + 4). Fetch the next instruction just like any other (i.e., don’t jump)

What If the Prediction Is Wrong?

* The real branch decision is made in the Execute stage.
* If PCSrcE is asserted (i.e., branch was actually taken), then: The pipeline realizes the prediction was wrong. So the processor flushes the 2 instructions fetched after the beq

Flushing: What Does It Mean?

Flush the Decode and Execute pipeline registers. Replace them with nop (no operation) instructions.



Misprediction Penalty

Two wasted cycles (because 2 instructions were fetched and need to be thrown away). This is called the branch misprediction penalty. So even though we don't stall in advance: If we predicted wrong we pay a penalty after

Pipeline Timeline with Misprediction

| Cycle | IF | ID | EX | MEM | WB |
| --- | --- | --- | --- | --- | --- |
| 1 | beq |  |  |  |  |
| 2 | instr1 | beq |  |  |  |
| 3 | instr2 | instr1 | beq |  |  |
| 4 | target | bubble | bubble |  |  |
| 5 | ... | target | ... |  |  |

In cycle 4, once we realize beq was taken: We discard instr1 and instr2 (flush them). Then we start fetching from the branch target.

FlushD = PCSrcE // Flush Decode stage when branch taken

FlushE = lwStall | PCSrcE // Flush Execute stage on lwStall or branch taken

| Condition | Signal | Effect |
| --- | --- | --- |
| Branch taken | PCSrcE=1 | Flush ID + EX stages |
| Load-use hazard | lwStall | Stall IF + ID, Flush EX |
| Both conditions | Combined | `FlushE = lwStall |

When PCSrcE = 1:

* The processor *was wrong* about the branch.
  + FlushD = 1: Clears instruction in Decode stage
  + FlushE = 1: Clears instruction in Execute stage

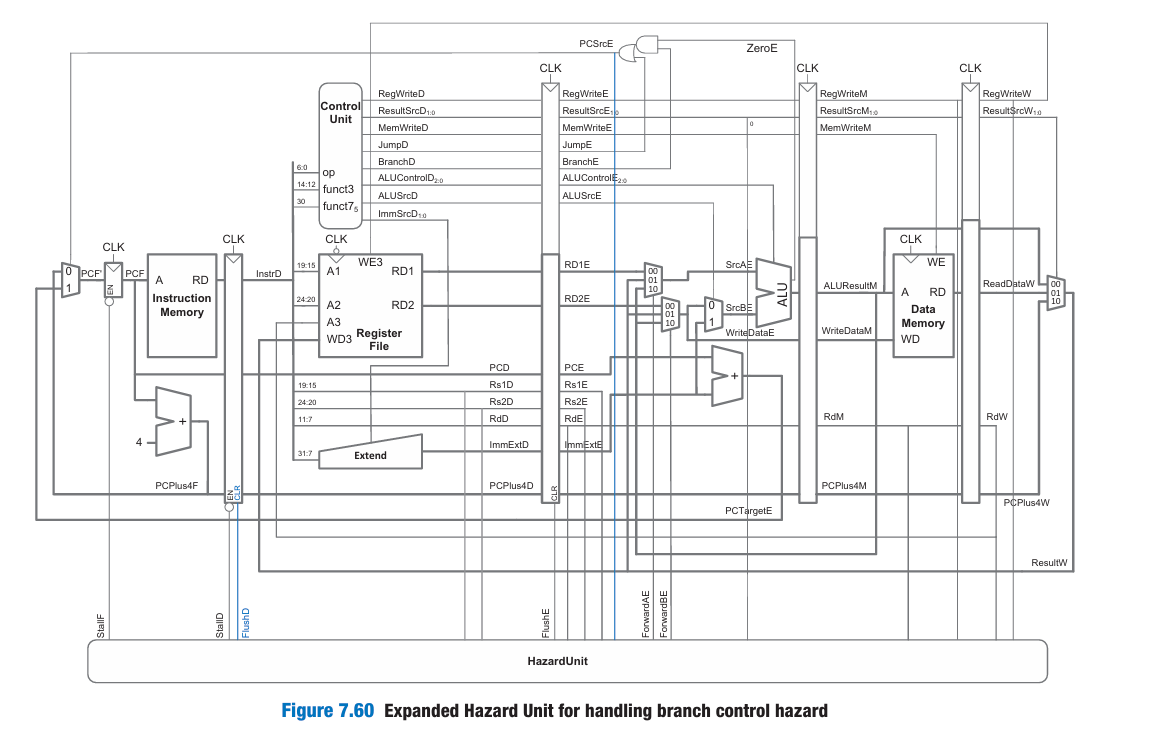
When lwStall = 1:

* Data not yet ready from a lw instruction
  + StallF = StallD = 1: Freeze Fetch and Decode stages
  + FlushE = 1: Insert bubble into Execute stage

The pipeline registers have:

* EN (enable) input → to implement stall (freeze)
* CLR (clear) input → to implement flush (insert bubble / nop)

| Term | Signal | Purpose |
| --- | --- | --- |
| FlushD | PCSrcE | Clear Decode stage when branch taken |
| FlushE | lwStall or PCSrcE | Clear Execute stage if branch taken or lw stall |
| StallD/F | lwStall | Freeze Decode & Fetch if lw hazard exists |
| Pipeline Safety | CLR to flush, EN to stall | Prevent wrong data from propagating. |



SUMMARY:

1. Data Hazards (RAW – Read After Write)

Occurs when: An instruction tries to read a register before a previous instruction has written the correct value to it.

Solution 1: Forwarding (Bypassing)

➤ Used when:

* Result is already computed (in EX or MEM stage) and
* Can be forwarded to the EX stage of the current instruction

🔧 Forwarding Logic (for source A):

if ((Rs1E == RdM) & RegWriteM & (Rs1E != 0))

ForwardAE = 2'b10; // Forward from Memory stage

else if ((Rs1E == RdW) & RegWriteW & (Rs1E != 0))

ForwardAE = 2'b01; // Forward from Writeback stage

else

ForwardAE = 2'b00; // No forwarding

Similarly, ForwardBE checks Rs2E.

Solution 2: Stalling for Load-Use Hazard

lw loads data from memory, which isn't ready until end of MEM stage, so dependent instructions can't proceed in EX.

🔧 Stall Logic:

lwStall = ResultSrcE0 & ((Rs1D == RdE) | (Rs2D == RdE)); // hazard detected

StallF = lwStall;

StallD = lwStall;

FlushE = lwStall;

* StallF, StallD: Freeze IF and ID
* FlushE: Clear EX stage → insert bubble (acts like NOP)

2. Control Hazards (Branch Instructions)

Occurs when: We don’t know whether a branch will be taken or not when we’re fetching the next instruction.

Solution 1: Stall Until Branch Decision

* Branch resolved in EX stage
* So you'd have to stall 2 cycles → inefficient

Solution 2: Branch Prediction

We predict that the branch is not taken, and fetch instructions in sequence.

* If branch actually taken (PCSrcE = 1):
  + The two wrong instructions (in ID and EX) are flushed.

🔧 Flush Logic:

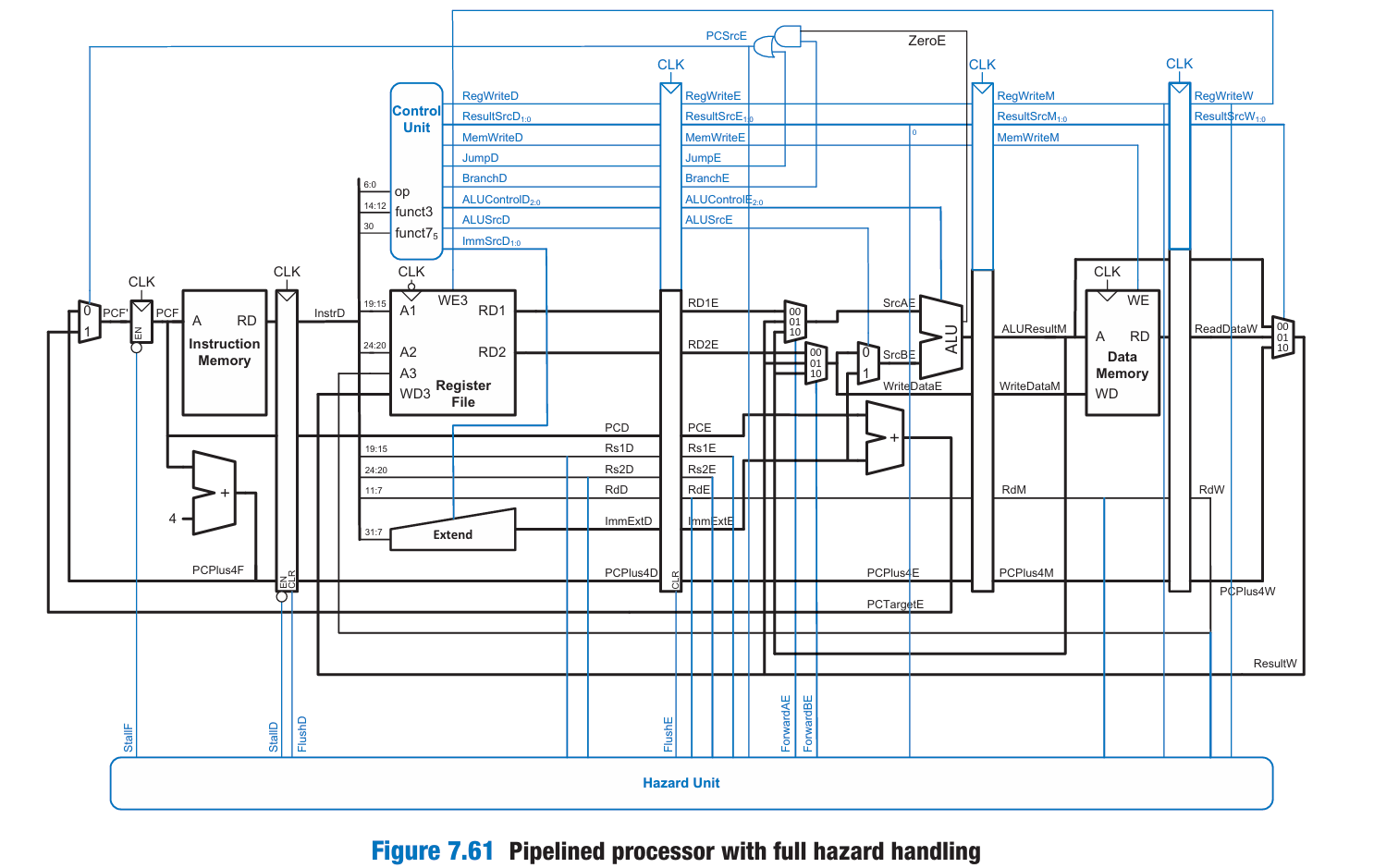
FlushD = PCSrcE; // Clear Decode stage (ID)

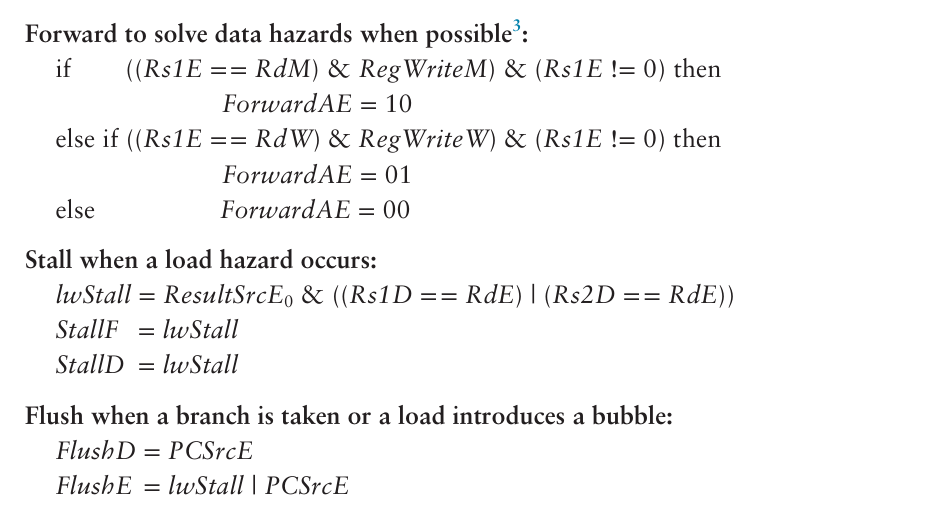
FlushE = lwStall | PCSrcE; // Clear Execute stage (EX) on branch or load hazard

| Hazard Type | Condition | Action Signals |
| --- | --- | --- |
| Data Hazard | Rs1E == RdM or RdW, with RegWrite | ForwardAE = 10 or 01 |
| Load Hazard | lwStall = ResultSrcE0 & (Rs1D == RdE or Rs2D == RdE) | StallF = StallD = FlushE = lwStall |
| Branch Taken | PCSrcE == 1 | FlushD = 1, FlushE = 1 |

🔩 Control Unit Outputs You Need

* Forwarding: ForwardAE, ForwardBE
* Stalling: StallF, StallD
* Flushing: FlushD, FlushE





Register File issue:

Why Asynchronous Read?

* The Decode (ID) stage reads register operands (rs1, rs2) using the values already present in the register file.
* These operands must be immediately available to the Execute (EX) stage in the same clock cycle.
* If you make read synchronous (posedge or negedge clk), you'll add a 1-cycle delay, which breaks the pipeline timing.

Write happens first, then read — within the same clock cycle — thanks to:

* Write on negedge clk
* Asynchronous read

Here's how it plays out in one clock cycle:

1. On the falling edge (negedge clk):
   * The Writeback stage writes the result of a previous instruction into the register file (via RegWrite, A3, and WD3).
2. During the rest of the cycle (asynchronously):
   * The Decode stage of a new instruction reads operands (A1, A2) from the register file immediately.
   * If this instruction needs the just-written value, it sees the updated data.

Why this matters:

If you write at posedge clk, and also read synchronously, the instruction in ID won't see the updated data until the next cycle → hazard!

With write on negedge + async read, you ensure that:

* The new value is available *immediately* after it's written.

1. posedge clk write + asynchronous read

Still works  
But: Register is written at the end of the clock cycle, which can create timing pressure

* The new value is not available *early* in the cycle for the next instruction
* In a pipelined design, the decode stage (which reads) happens right after the writeback stage — in the same cycle
* If you write late (on posedge), the Decode stage might not yet see the new value

2. negedge clk write + asynchronous read

Preferred in pipelined designs  
Ensures write completes before decode (read) happens  
The register is updated early in the cycle → decode stage sees the correct value

Summary of ImmSrc Mapping:

| ImmSrc | Type | Example Instruction |
| --- | --- | --- |
| 000 | I | addi, lw |
| 001 | S | sw |
| 010 | B | beq, bne |
| 011 | J | jal |
| 100 | U | lui |

Why don't we need to sign-extend for R-type instructions?

🔹 1. R-type instructions don’t have immediate fields

R-type instructions (like add, sub, and, or, sll, etc.) operate purely on register values.

R-type format:

| 31–25 | 24–20 | 19–15 | 14–12 | 11–7 | 6–0 |
| --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |

There is no immediate field to extract or extend.

| PC | Instruction | Mnemonic | Meaning |
| --- | --- | --- | --- |
| 0x00 | 32'h00500093 | addi x1, x0, 5 | x1 = 5 |
| 0x04 | 32'h00a00113 | addi x2, x0, 10 | x2 = 10 |
| 0x08 | 32'h002081b3 | add x3, x1, x2 | x3 = x1 + x2 |
| 0x0C | 32'h4011a233 | sub x4, x3, x1 | x4 = x3 - x1 |
| 0x10 | 32'h00412023 | sw x4, 0(x2) | MEM[x2] = x4 |
| 0x14 | 32'h00012283 | lw x5, 0(x2) | x5 = MEM[x2] |
| 0x18 | 32'h0042b2b3 | or x6, x5, x4 | x6 = x5 |
| 0x1C | 32'h0062c263 | beq x6, x6, +8 | if (x6==x6) PC += 8 |
| 0x20 | 32'h06300393 | addi x7, x0, 99 | x7 = 99 (skipped by branch) |
| 0x24 | 32'h02a00393 | addi x7, x0, 42 | x7 = 42 |
| 0x28 | 32'h008004ef | jal x8, +8 | x8 = PC+4, jump to PC+8 |
| 0x2C | 32'h07b00493 | addi x9, x0, 123 | x9 = 123 (skipped by jal) |
| 0x30 | 32'h00100493 | addi x9, x0, 1 | x9 = 1 |

Instruction 1: addi x1, x0, 5

* Enters IF in cycle 2, WB in cycle 6.
* Writes 5 into x1.
* No hazards.

Instruction 2: addi x2, x0, 10

* Starts IF in cycle 3, WB in cycle 7.
* Writes 10 into x2.
* No hazards.

Instruction 3: add x3, x1, x2

* Starts IF in cycle 4, needs x1, x2 from previous two instructions.
* RAW hazard on x1 and x2.
* Your design successfully forwards or stalls, and x3 gets 5 + 10 = 15 in EX.

Instruction 4: sub x4, x3, x1

* Starts IF in cycle 5.
* Needs x3 (produced in previous instruction).
* RAW hazard on x3 — handled via forwarding.
* x4 = 15 - 5 = 10.

Instruction 5: sw x4, 0(x2)

* Starts IF in cycle 6.
* Stores value of x4 at address in x2.
* x2 = 10, x4 = 10 → MEM[10] = 10.
* No writeback.

Instruction 6: lw x5, 0(x2)

* Starts IF in cycle 7.
* Loads MEM[10] = 10 → x5 = 10.
* Value was just written by sw, no memory hazard.

Instruction 7: or x6, x5, x4

* Starts IF in cycle 8.
* Load-use hazard: needs x5 immediately after it’s loaded.
* Your pipeline stalls F and D, flushes E.
* x6 = 10 | 10 = 10.

Instruction 8: beq x6, x6, +8

* Always true → Branch taken to PC + 8 = 0x24.
* Starts IF in cycle 9, branch resolved in EX.
* Instructions at 0x20 (next one) will be fetched, but flushed in cycle 12.

Instruction 9: addi x7, x0, 99

* Starts IF in cycle 10.
* Flushed due to taken branch.

Instruction 10: addi x7, x0, 42

* At 0x24, target of branch.
* x7 = 42.
* Valid instruction, executes fine.

Instruction 11: jal x8, +8

* PC jumps to 0x30 (0x28 + 8), x8 = 0x2C (PC + 4).
* Instruction at 0x2C fetched but flushed.

Instruction 12: addi x9, x0, 123

* Skipped due to jal (flushed).

Instruction 13: addi x9, x0, 1

* Final instruction at 0x30.
* x9 = 1.

Final Register Values

| Register | Value | Set By |
| --- | --- | --- |
| x1 | 5 | addi x1, x0, 5 |
| x2 | 10 | addi x2, x0, 10 |
| x3 | 15 | add x3, x1, x2 |
| x4 | 10 | sub x4, x3, x1 |
| x5 | 10 | lw x5, 0(x2) |
| x6 | 10 | or x6, x5, x4 |
| x7 | 42 | addi x7, x0, 42 (not 99 because that was skipped) |
| x8 | 0x2C | jal x8, +8 |
| x9 | 1 | addi x9, x0, 1 |

Memory:

* MEM[10] = 10 (from sw x4, 0(x2))

| Hazard Type | Instructions | Status |
| --- | --- | --- |
| RAW (x1, x2 → x3) | add x3, x1, x2 | Handled |
| RAW (x3 → x4) | sub x4, x3, x1 | Handled |
| Load-use (x5 → x6) | or x6, x5, x4 | Stall |
| Control hazard | beq x6, x6, +8 | Flushed |
| Control hazard | jal x8, +8 | Flushed |

Detailed explanation:

Instruction 1: addi x1, x0, 5

PC = 0x00, Binary: 0x00500093

-Add 5 to x0 (which is always 0), and write the result to x1  
-x1 = 5

| Cycle | Stage | Details |
| --- | --- | --- |
| 1 | IF | Fetches addi x1, x0, 5 into InstrF |
| 2 | ID | Decode: rs1 = x0 = 0, rd = x1, imm = 5 |
| 3 | EX | ALU computes: 0 + 5 = 5 |
| 4 | MEM | No memory op |
| 5 | WB | Writes 5 into x1 |

Instruction 2: addi x2, x0, 10

PC = 0x04, Binary: 0x00a00113

x2 = 10

| Cycle | Stage | Details |
| --- | --- | --- |
| 2 | IF | Fetch addi x2, x0, 10 |
| 3 | ID | Decode: rs1 = x0 = 0, rd = x2 |
| 4 | EX | ALU computes: 0 + 10 = 10 |
| 5 | MEM | No memory access |
| 6 | WB | Writes 10 into x2 |

Instruction 3: add x3, x1, x2

PC = 0x08, Binary: 0x002081b3  
x3 = x1 + x2 = 5 + 10 = 15

| Cycle | Stage | Details |
| --- | --- | --- |
| 3 | IF | Fetch add x3, x1, x2 |
| 4 | ID | Decode: rs1 = x1, rs2 = x2, rd = x3 (data hazard!) |
| 5 | EX | ALU uses forwarded values 5 and 10, computes 5+10=15 |
| 6 | MEM | No memory |
| 7 | WB | Writes 15 to x3 |

Instruction 4: sub x4, x3, x1

PC = 0x0C, Binary: 0x4011a233  
x4 = x3 - x1 = 15 - 5 = 10

| Cycle | Stage | Details |
| --- | --- | --- |
| 4 | IF | Fetch sub x4, x3, x1 |
| 5 | ID | rs1 = x3 = 15, rs2 = x1 = 5 |
| 6 | EX | ALU computes 15 - 5 = 10 |
| 7 | MEM | No memory |
| 8 | WB | Writes 10 to x4 |

Instruction 5: sw x4, 0(x2)

PC = 0x10, Binary: 0x00412023  
Store x4 (10) into memory at address x2 + 0 → MEM[10] = 10

| Cycle | Stage | Details |
| --- | --- | --- |
| 5 | IF | Fetch sw x4, 0(x2) |
| 6 | ID | rs1 = x2 = 10 (base), rs2 = x4 = 10 (data) |
| 7 | EX | ALU calculates address 10 + 0 = 10 |
| 8 | MEM | Store 10 into MEM[10] |
| 9 | WB | None (sw doesn’t write to register) |

Instruction 6: lw x5, 0(x2)

PC = 0x14, Binary: 0x00012283  
Load word from MEM[x2+0] → x5 = 10

| Cycle | Stage | Details |
| --- | --- | --- |
| 6 | IF | Fetch lw x5, 0(x2) |
| 7 | ID | rs1 = x2 = 10 |
| 8 | EX | ALU calculates address 10 + 0 = 10 |
| 9 | MEM | Load from MEM[10] → gets 10 |
| 10 | WB | Writes 10 to x5 |

Instruction 7: or x6, x5, x4

PC = 0x18, Binary: 0x0042b2b3  
x6 = x5 | x4 = 10 | 10 = 10

This is a load-use hazard — since x5 is written in cycle 10, but or needs it in EX at cycle 9

So pipeline inserts a stall, flushes EX once

| Cycle | Stage | Details |
| --- | --- | --- |
| 8 | IF | Fetch or x6, x5, x4 |
| 9 | ID | Hazard: x5 isn’t ready → stall one cycle |
| 10 | (stall) | Bubble inserted |
| 11 | EX | ALU computes `10 |
| 12 | MEM | None |
| 13 | WB | Write 10 to x6 |

Instruction 8: beq x6, x6, +8

PC = 0x1C, Binary: 0x0062c263  
Compare x6 and x6 → always equal → branch taken → PC = 0x24

| Cycle | Stage | Details |
| --- | --- | --- |
| 9 | IF | Fetch beq |
| 10 | ID | x6 == x6 = true |
| 11 | EX | Branch decision: taken → PC updated to 0x24 |
| 12 | MEM | None |
| 13 | WB | None |

-Instruction at 0x20 (addi x7, 99) is flushed  
-Next instruction fetched is at 0x24 (addi x7, 42)

Instruction 9: addi x7, x0, 42

PC = 0x24, Binary: 0x02a00393  
x7 = 42

| Cycle | Stage | Details |
| --- | --- | --- |
| 11 | IF | Fetch addi x7, x0, 42 |
| 12 | ID | rs1 = x0 = 0 |
| 13 | EX | ALU computes 0 + 42 = 42 |
| 14 | MEM | None |
| 15 | WB | Writes 42 to x7 |

Instruction 10: jal x8, +8

PC = 0x28, Binary: 0x008004ef  
Jump to PC+8 = 0x30, set x8 = PC+4 = 0x2C

| Cycle | Stage | Details |
| --- | --- | --- |
| 12 | IF | Fetch jal |
| 13 | ID | Calculate return addr = 0x2C |
| 14 | EX | ALU calculates target PC = 0x30 |
| 15 | MEM | None |
| 16 | WB | Writes 0x2C to x8 |

-Instruction at 0x2C (addi x9, 123) is flushed  
-Next fetched: 0x30: addi x9, 1

Instruction 11: addi x9, x0, 1

PC = 0x30, Binary: 0x00100493  
x9 = 1

| Cycle | Stage | Details |
| --- | --- | --- |
| 15 | IF | Fetch addi x9, 1 |
| 16 | ID | rs1 = x0 = 0 |
| 17 | EX | ALU computes 0 + 1 = 1 |
| 18 | MEM | None |
| 19 | WB | Writes 1 to x9 |

Instruction 12: 0x34 → 0x00000000 (NOP)

This is your HALT — when a NOP is fetched, the simulation ends.

Final Register Values (as expected):

| Register | Value | Why |
| --- | --- | --- |
| x1 | 5 | addi x1, x0, 5 |
| x2 | 10 | addi x2, x0, 10 |
| x3 | 15 | x1 + x2 = 5 + 10 |
| x4 | 10 | x3 - x1 = 15 - 5 |
| x5 | 10 | loaded from MEM[10] |
| x6 | 10 | `x5 |
| x7 | 42 | addi x7, 42 (after branch) |
| x8 | 44 | (0x2C = return address from jal) |
| x9 | 1 | addi x9, 1 |