RISC-V Pipelined Processor

CMPU 224 -- Computer Organization

Jason Waterman

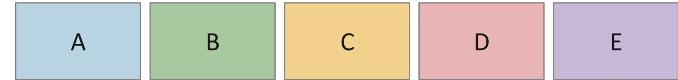
Real-World Pipelines

Idea
- Divide process into independent stages
- Move objects through stages in sequence
- Multiple objects processed simultaneously

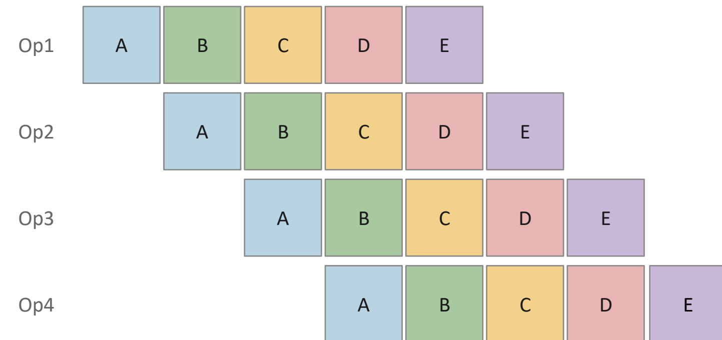
Throughput
- Number of results produced per unit time

Latency
- Time required to process a single item

Sequential

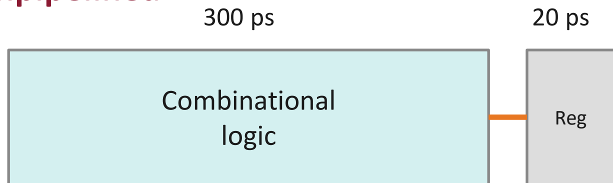


Pipelined



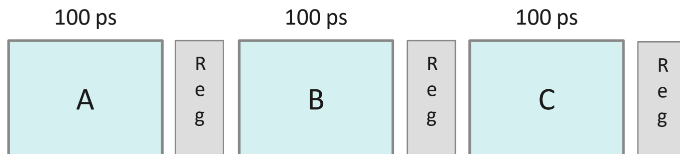
Computational Example

Unpipelined



Delay = 320 ps | Throughput = 3.12 GIPS

3-Way Pipelined

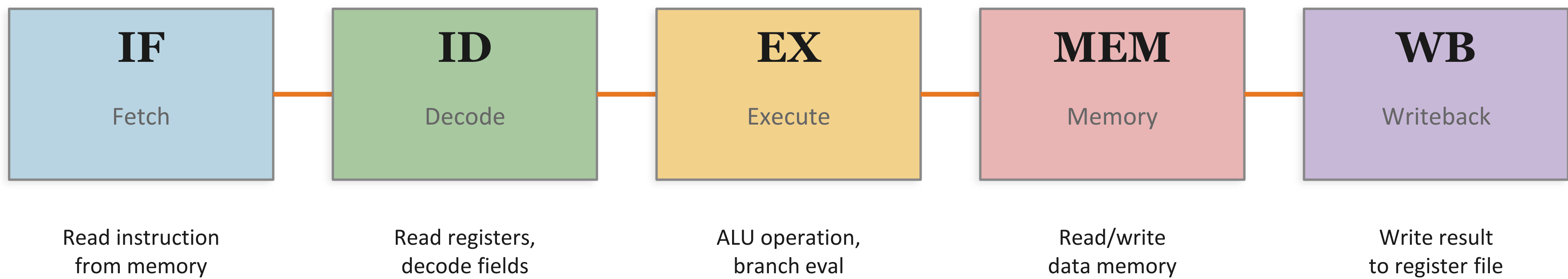


Delay = 360 ps | Throughput = 8.33 GIPS

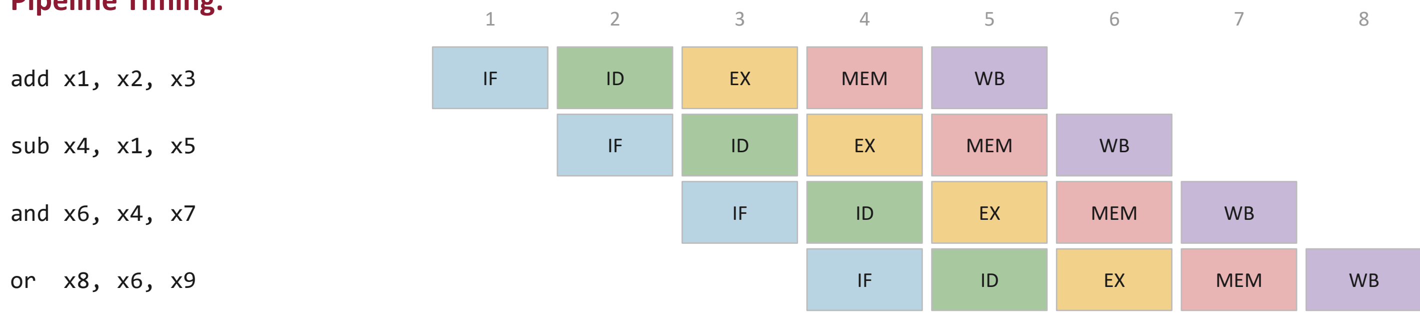
Key Observations

- Throughput improved 2.67x with 3 stages
- Latency slightly increases (320 to 360 ps) due to register overhead
- Throughput limited by slowest stage
- Register overhead becomes significant with deeper pipelines
- Modern CPUs: 15+ pipeline stages

RISC-V 5-Stage Pipeline



Pipeline Timing:



	1	2	3	4	5	6	7	8
`add x1, x2, x3`	IF	ID	EX	MEM	WB			
`sub x4, x1, x5`		IF	ID	EX	MEM	WB		
`and x6, x4, x7`			IF	ID	EX	MEM	WB	
`or  x8, x6, x9`				IF	ID	EX	MEM	WB

RISC-V 5-Stage Pipeline Datapath

4
2
+
PC
Instr.
memory
addr
instr
Compressed
decoder
IFID
enable
clear
Decode
opcode
Registers
Wr data
Wr idx
R1 idx
R2 idx
Wr
En
Reg 1
Reg 2
Imm.
IDEX
enable
clear
Op 1
ALU
Res
Op 2
Branch
Branch
taken
EX/
MEM
Addr.
Data
in
Wr
en
Data
memory
Read
out
MEM/
WB

RISC-V 5-Stage Pipeline Control and Datapath

PC
Instr. memory
addr
instr
Compressed decoder
IFID
enable
clear
Decode
opcode
Control
Registers
Wr data
Wr idx
R1 idx
R2 idx
Wr En
Reg 1
Reg 2
Imm.
IDEX
clear
enable
ALU
ctrl
Op 1
Op 2
Res
Branch
Branch taken
EX/ MEM
clear
Data memory
Wr en
Addr.
Data in
Read out
MEM/ WB

Fetch Stage (IF)

Input

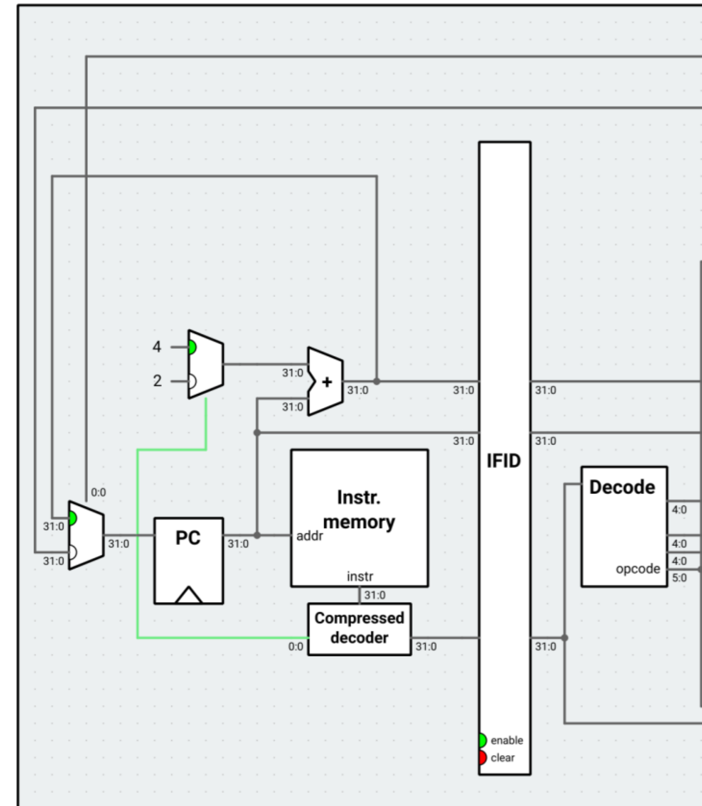
- PC: Program Counter (address of current instruction)

Computed Values

- pc4_in: PC + 4 (next sequential instruction)
- pc_in: Current PC (used later for branches, jal, jalr, auipc)
- instr_in: Fetched instruction word (sent to Decode)

Key Points

- Instruction memory is read using PC as the address
- PC+4 is computed for use as the return address in jal/jalr (written in Writeback stage)
- The current PC is forwarded for use in branch/jump target calculation in Execute



Decode Stage (ID)

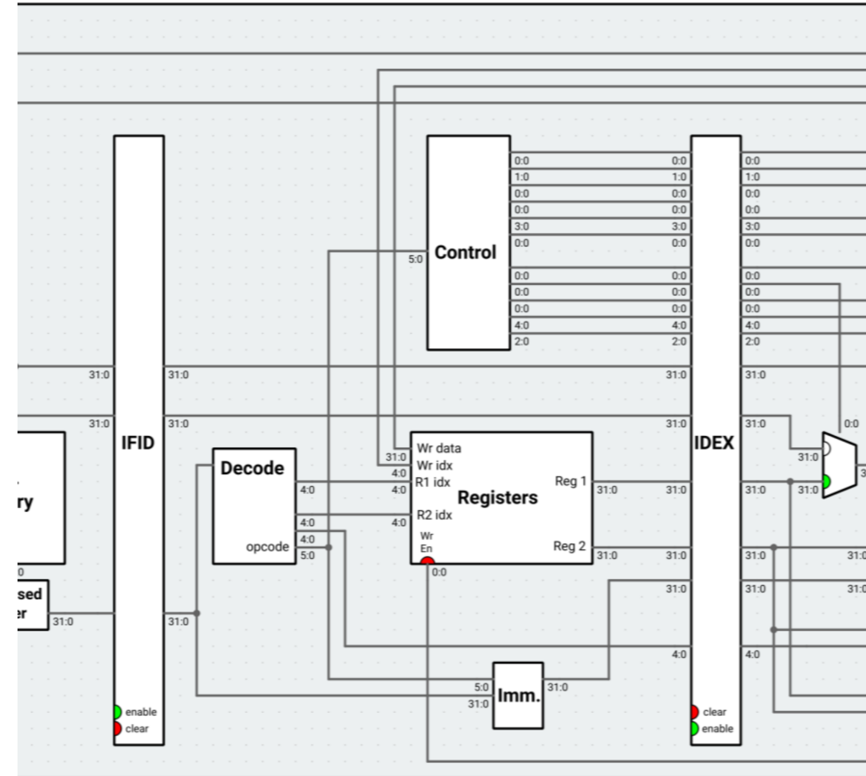
Input

- instr_out: The instruction word from Fetch

Computed Values

- r1_in, r2_in: Register values (rs1, rs2)
- imm_in: Sign-extended 32-bit immediate
- br_op_in: Branch type (BEQ, BNE, BLT, BGE...)

- wr_reg_idx_in: Destination register index (rd)
- mem_op_in: Memory operation (LW, SW, NOP)
- mem_do_write_in: memory write enable (used in MEM)
- do_jmp_in: Is this a jump instruction?
- do_br_in: Is this a branch instruction?



Execute Stage (EX)

Inputs

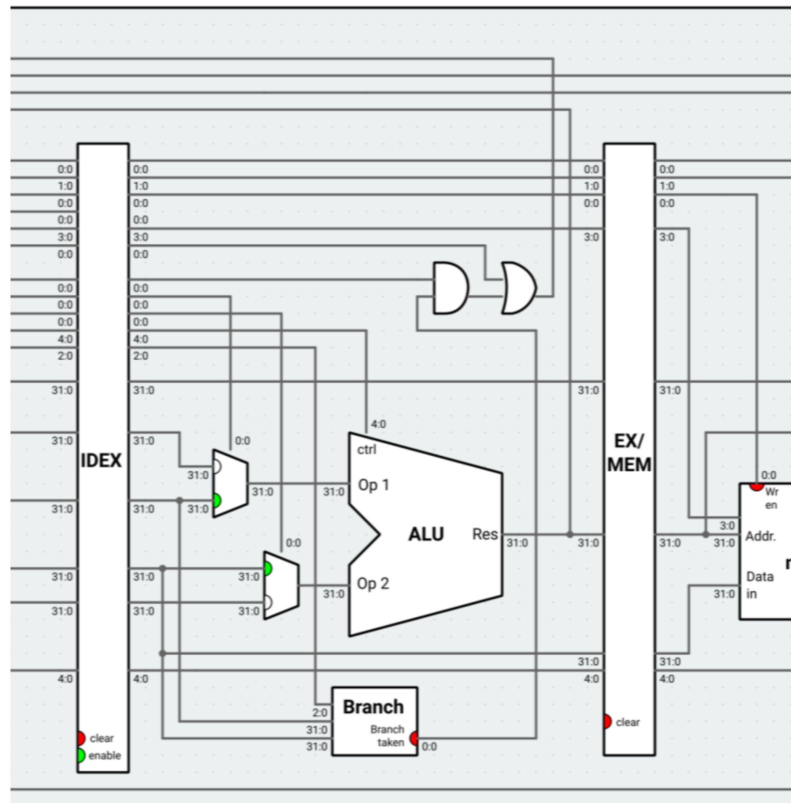
- r1_out, r2_out: Register values for ALU / branch
- alu_op1_ctl_out: Selects PC or rs1
- alu_op2_ctl_out: Selects rs2 or immediate
- alu_ctl_out: ALU operation to perform
- do_br_out: is the instruction a branch?
- do_jump_out: is the instruction a jal/jalr?

Computed Values

- alures_in: ALU result
- Branch taken

Note

- Output of OR gate goes to the PC MUX to decide if next instruction is PC+4 or Branch/Jump target from ALU



Memory Stage

Inputs:

- alures_out: Address for load/store
- mem_do_write: Write enable
- mem_op_out: LW, SW, or NOP
- r2_out: data for store

Computed Values:

- mem_read_in: Data read from memory

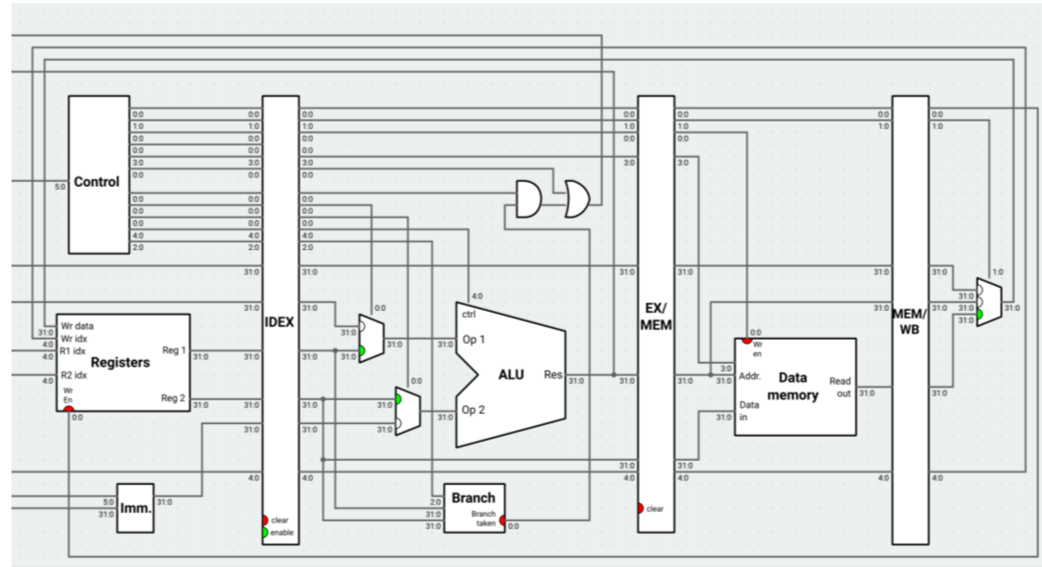
Writeback Stage

Inputs:

- reg_do_write_out: Writeback enable
- reg_wr_src_ctl_out: Data source mux
- wr_reg_idx_out: Destination register

Computed Output:

- ALU result, Memory read, PC+4



Data Dependencies

```
addi  x1, x0, 50
add   x2, x1, x3    # reads x1
lw    x4, 0(x2)     # reads x2
```

Read-After-Write (RAW) Dependency

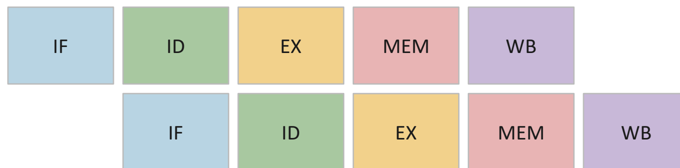
Result from one instruction used as operand for a following instruction

- Very common in real programs
- Must handle correctly for pipeline to produce right results
- Want to minimize performance impact

The Problem:

```
addi x1, x0, 50

add  x2, x1, x3
```

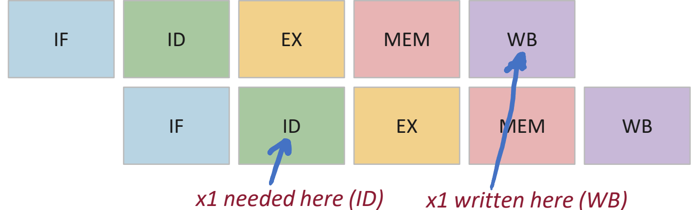


x1 needed here (ID) *x1 written here (WB)*

Stalling for Data Dependencies

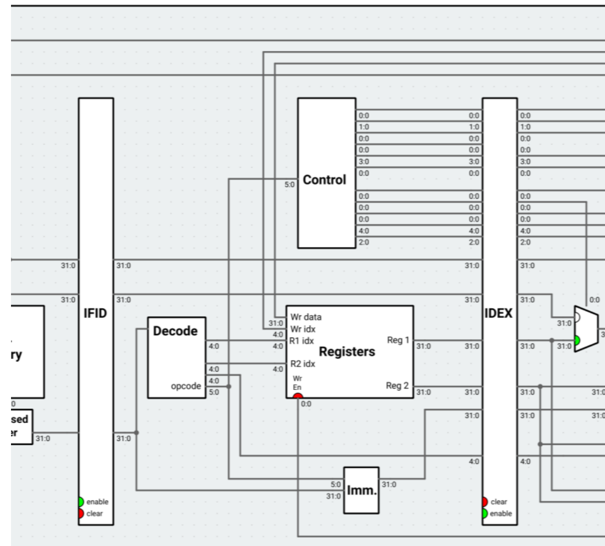
If a following instruction needs a register value that has not yet been written:

- Hold the dependent instruction in the Decode stage
- Inject a bubble (NOP) into the Execute stage
- Following instructions stall in earlier stages



Pipeline Register Modes

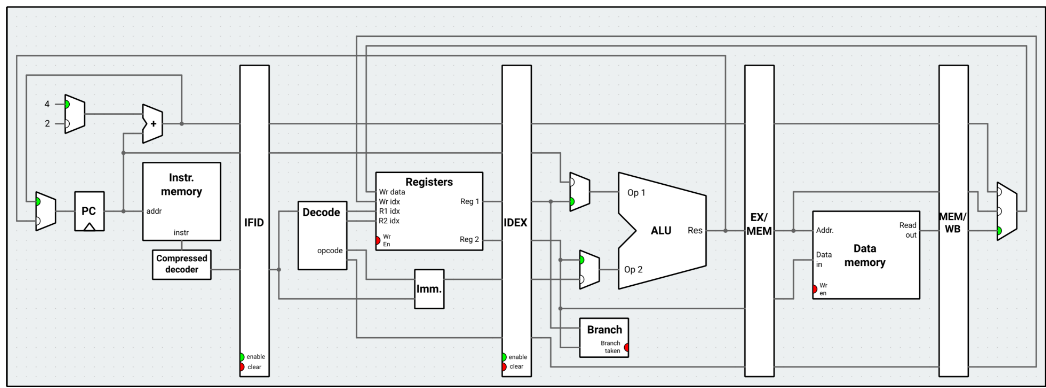
Normal	Loads new value on clock edge
Stall	Retains current value (holds instruction)
Bubble	Loads NOP value (cancels instruction)



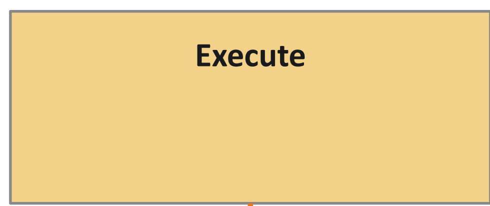
Data Forwarding

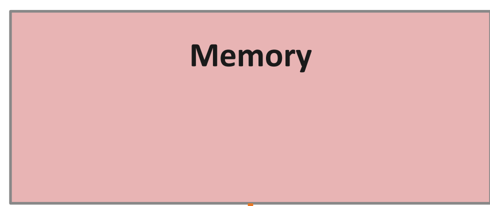
Observation:
The result value is computed in Execute or Memory stage, but not written to the register file until Writeback Stage

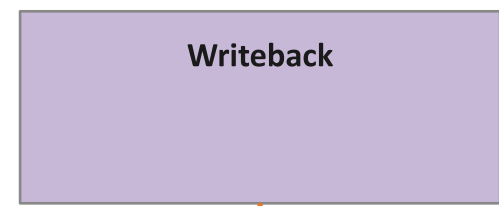
Solution:
Forward (bypass) the value directly from where it is produced to the Decode stage, where it is needed



Forwarding Sources





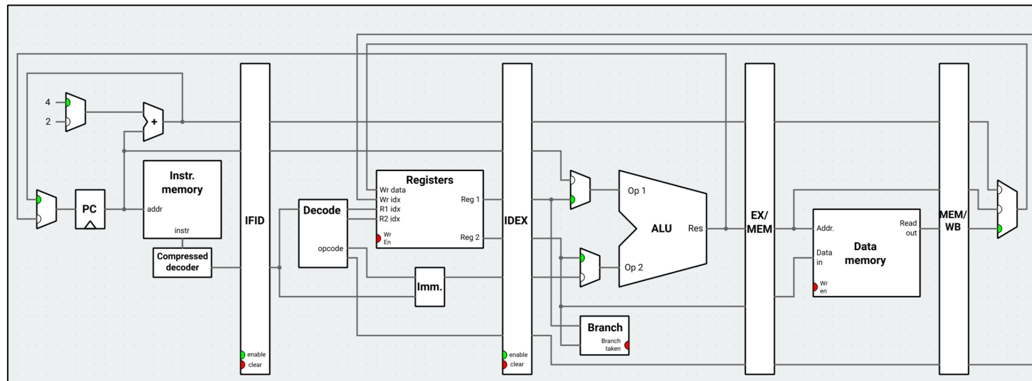


Load/Use Hazard

Forwarding cannot solve all hazards!

A load instruction (lw) reads data from memory in the MEM stage. If the next instruction needs that value in ID, the data is not available yet.

Solution: Stall for one cycle, then forward the loaded value from MEM stage.



Detection:

Condition	Trigger
Load/Use Hazard	EX stage has lw AND EX.rd matches ID.rs1 or ID.rs2

```
lw    x1, 0(x2)      # loads x1
add   x3, x1, x4     # uses x1 immediately!
```

Control Actions:

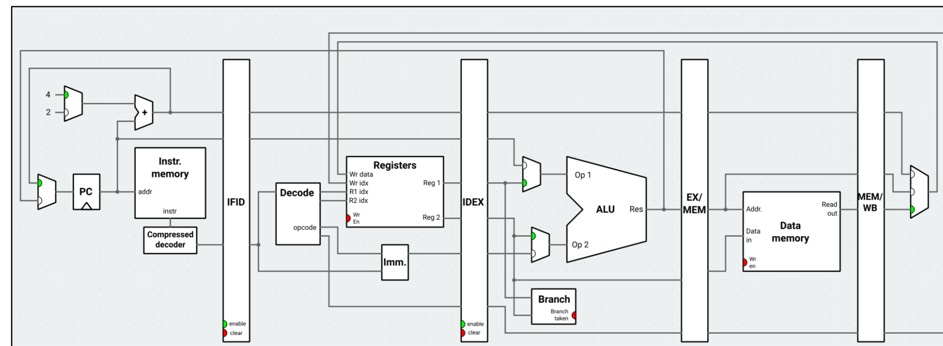
IF	ID	EX	MEM	WB
stall	stall	bubble	normal	normal

Control Hazards: Branch Misprediction

Problem:
Branch outcome not known until Execute stage. If we guess wrong, we have fetched incorrect instructions.

Our Strategy:

- Predict branch not taken (always fetch PC+4)
- If branch IS taken, cancel the two incorrectly fetched instructions
- Replace them with bubbles in Decode and Execute stages



Condition	IF	ID	EX	MEM	WB
Mispredicted Branch	normal	**bubble**	**bubble**	normal	normal

```
beq  x1, x0, target  # taken!
addi x2, x0, 1       # wrong! (cancel)
addi x3, x0, 2       # wrong! (cancel)
target: ...
```

Penalty: 2 wasted cycles per misprediction

Pipeline Hazard Control Summary

Detection Conditions

Condition	Trigger
Load/Use Hazard	EX stage has lw AND EX.rd in {ID.rs1, ID.rs2} (one cycle penalty)
Mispredicted Branch	EX stage has branch AND branch was taken (two cycle penalty)

Control Actions (next cycle)

Condition	IF	ID	EX	MEM	WB
Load/Use	**stall**	**stall**	**bubble**	normal	normal
Mispredict	normal	**bubble**	**bubble**	normal	normal

5-Stage Pipeline with Hazards and Forwarding

PC
enable
Instr. memory
addr
instr
Compressed decoder
IFID
enable
clear
Decode
opcode
Control
Registers
Wr data
Wr idx
R1 idx
R2 idx
Wr En
Reg 1
Reg 2
Imm.
IDEX
clear
enable
ALU
ctrl
Op 1
Op 2
Res
Branch
Branch taken
EX/MEM
clear
Data memory
Wr en
Addr.
Data in
Read out
MEM/WB
Forwarding unit
Hazard unit

Performance Analysis

CPI = 1.0 + B/I

C = I + B clock cycles, I = instructions completed, B = bubbles injected

Penalty Breakdown (typical rv32i workload):

Penalty	Fraction	Hazard Rate	Bubbles	Penalty
Load/Use (LP)	0.25	0.20	1	**0.05**
Branch Mispredict (MP)	0.20	0.40	2	**0.16**

Total penalty: 0.05 + 0.16 = 0.21

CPI = 1.21

Pipelined Processor Summary

Pipeline Design	5-stage in-order pipeline: IF, ID, EX, MEM, WB. Each instruction has a 5-cycle latency, but throughput approaches 1 instruction/cycle.
Data Hazards	Most RAW hazards resolved by forwarding (zero penalty). Load/use hazards require one cycle stall.
Control Hazards	Branch mispredictions waste 2 cycles. Handled by injecting bubbles into Decode and Execute stages.
Pipeline Control	Combinational logic detects hazard conditions and sets stall/bubble signals for each pipeline register.
Performance	CPI close to 1.0 with forwarding. Typical workloads achieve CPI around 1.2.