Pipelining: Implementation

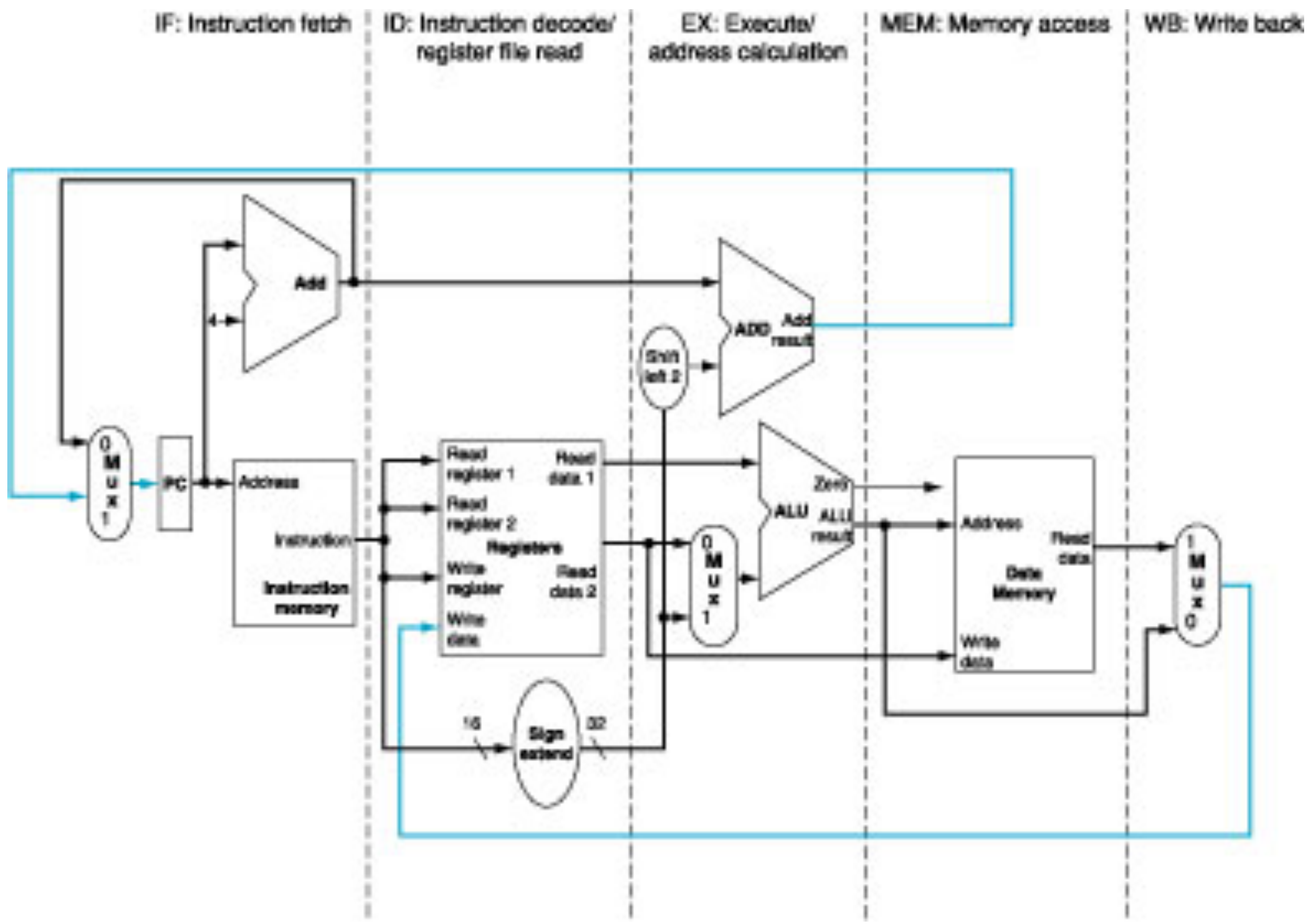
CPSC 252 Computer Organization

Ellen Walker, Hiram College

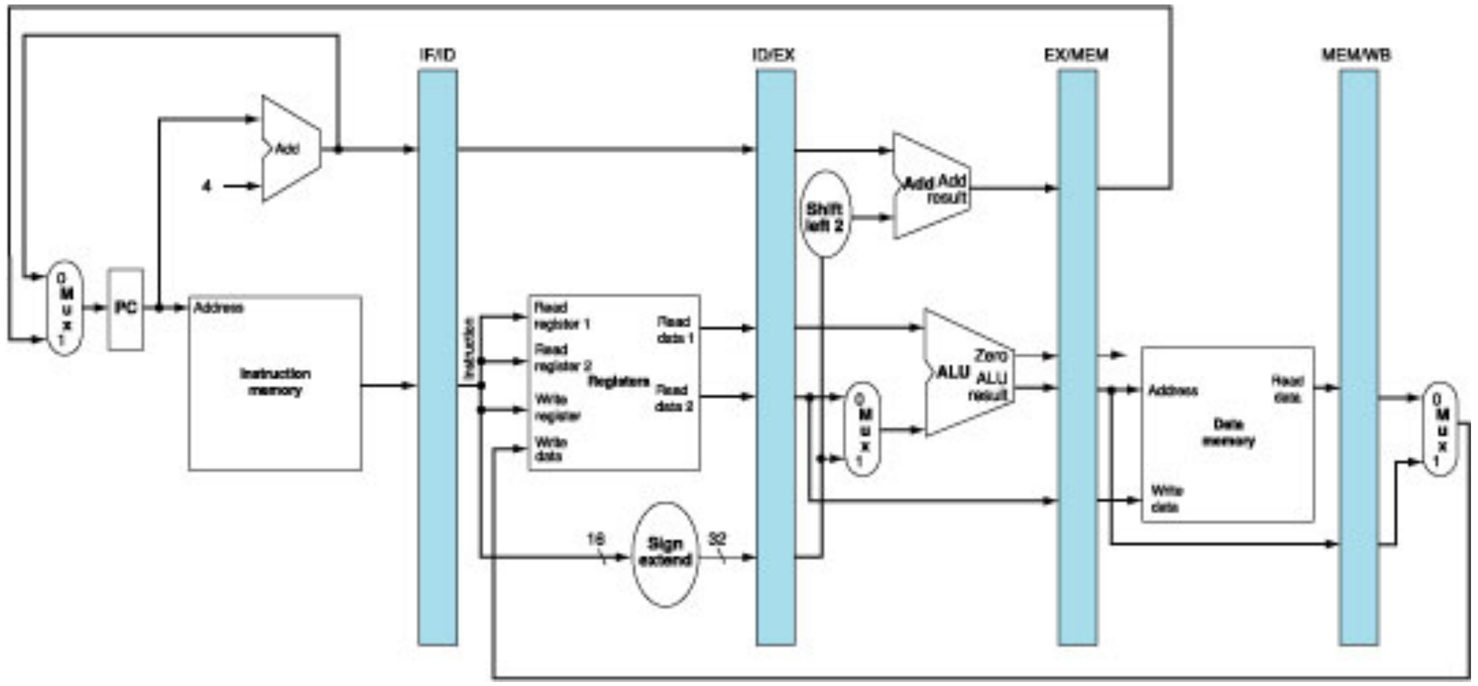
Outline

- Datapath (ignoring hazards)
- Control (ignoring hazards)
- Data hazards
  - Forwarding
  - Detecting when to stall
- Branch hazards

Single Cycle Datapath (stages)



Datapath with Pipeline Registers



Each register holds values of *all* lines that pass through it so that hardware can be reused for next stage

Datapath Concepts

- Each logical component of the data path can only be used in one stage (sub-datapaths)
- Register between sub-datapaths for each pair of stages; stores all information that passes between
- To pass information to a later stage, it must go through (all) intermediate registers

Instruction Fetch (IF)

- Get instruction from mem[PC], place into IF/ID register (32 bits)
- PC <- PC+4
- Save PC+4 in IF/ID register for possible use by branch instruction (32 bits)
- IF/ID register has 64 bits

Instruction Decode (ID)

- Provide register fields from instruction in IF/ID to register file, save contents of rs and rt in ID/EX (64 bits)
- Sign-extend the immediate field from instruction in IF/ID and save value in ID/EX (32 bits)
- Pass the PC field from IF/ID into ID/EX (32 bits)
- ID/EX has 128 bits

Execute or Address (EX)

- Perform an ALU operation controlled by the multiplexor on its inputs. Save result and "zero" bit in EX/MEM (33 bits)
  - Perform operation on contents of rs and rt
  - **OR** add rs to sign-extended immediate field (to compute memory address)
- Compute (PC+4)+ (sign-extended immediate field)*4, for branch address. Save to EX/MEM (32 bits)
- Pass value of rt to EX/MEM for store (32 bits)
- EX/MEM has 97 bits

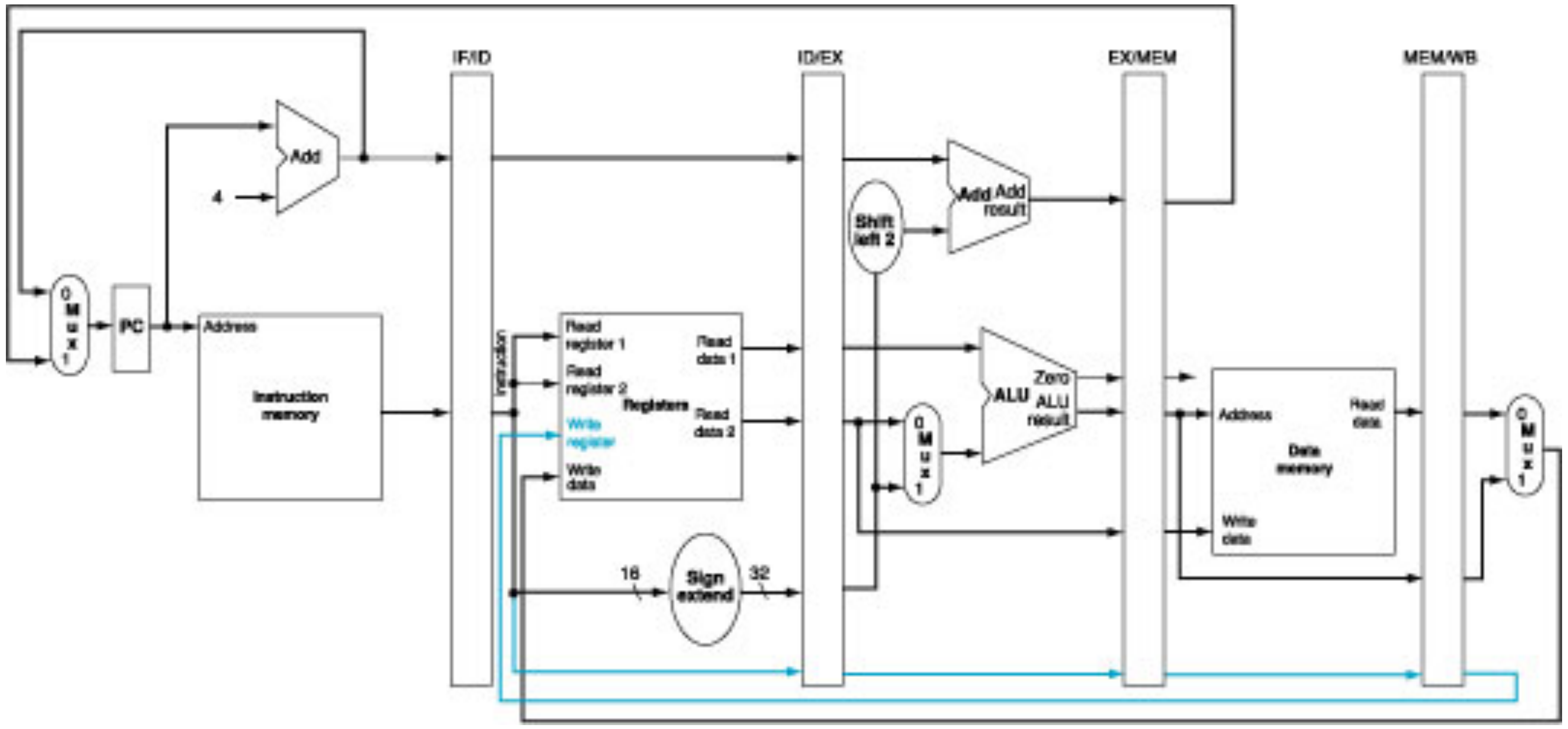
Memory Access (MEM)

- Pass address and register value (write data) from EX/MEM to memory; save memory read data in MEM/WB (32 bits)
- Pass ALU result from EX/MEM to MEM/WB (32 bits)
- Branch address passed back to PC (through branch control MUX)
- MEM/WB has 64 bits

Write Back (WB)

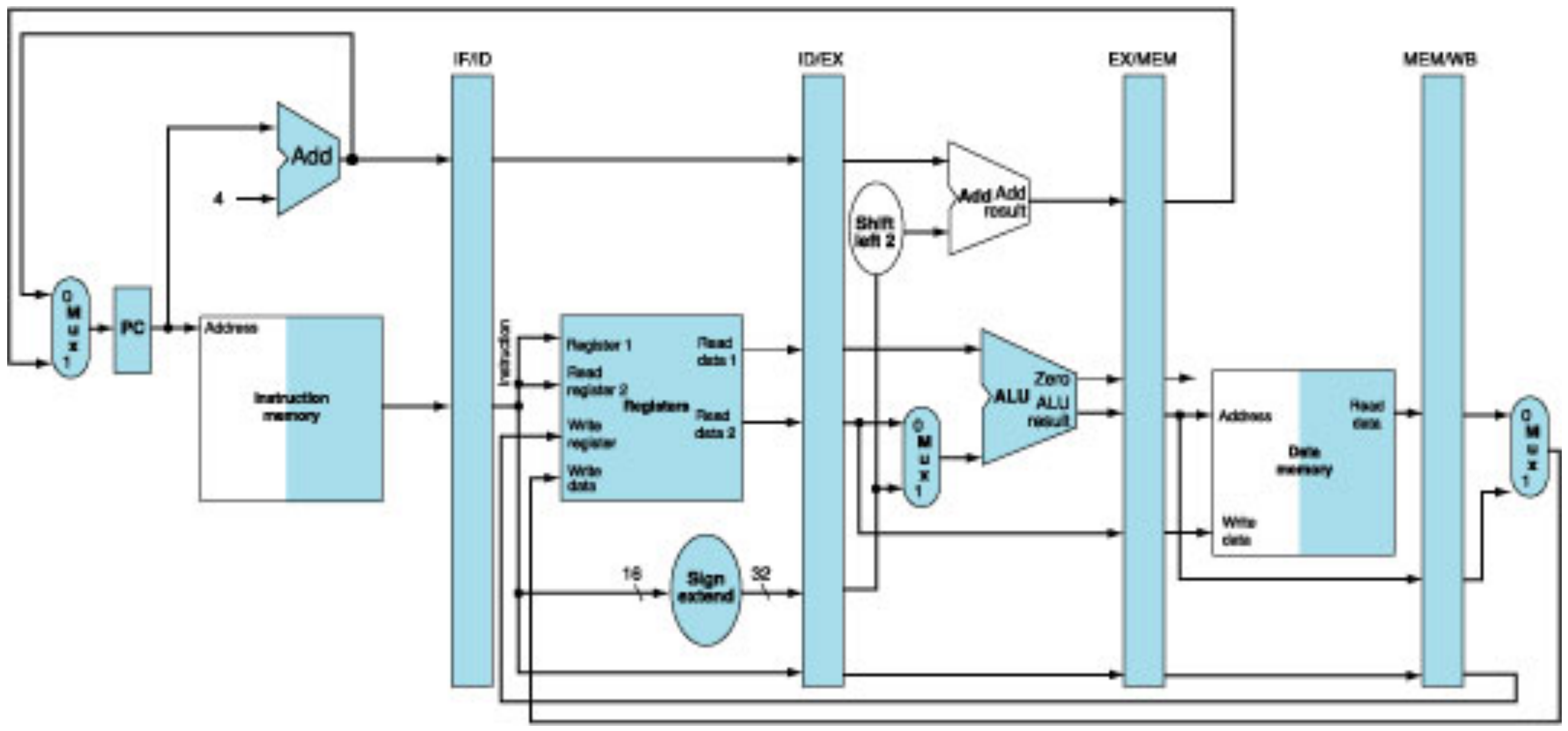
- Memory data **OR** ALU result is passed to write data on register file, rd is written [BUG HERE (for lw)]
- This stage does nothing for a sw or branch instruction (but time must be taken because other instructions are executing)

Corrected Pipeline

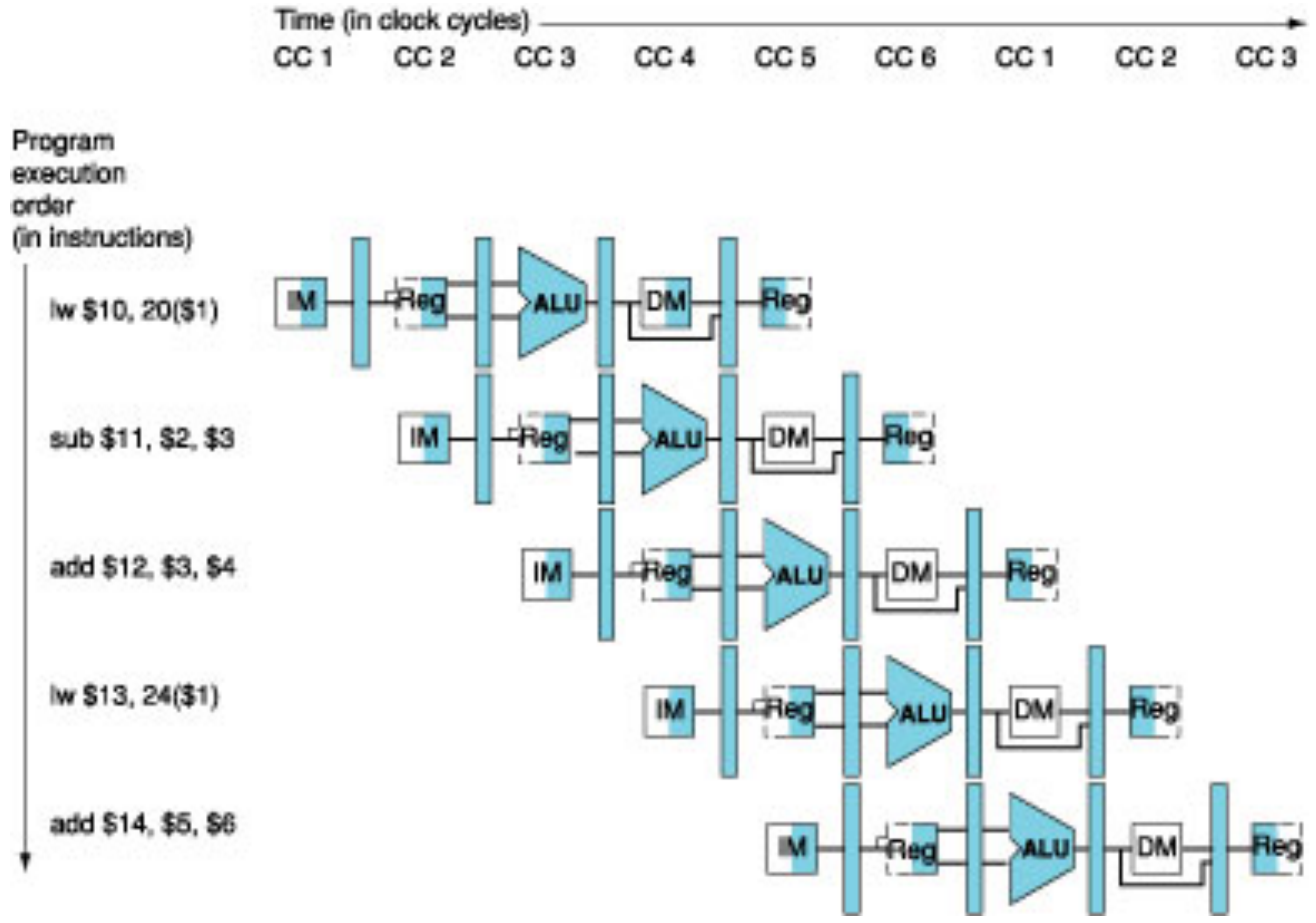


Write register number passed through to WB stage, then used. Increase all registers except IF/ID by 5 bits.

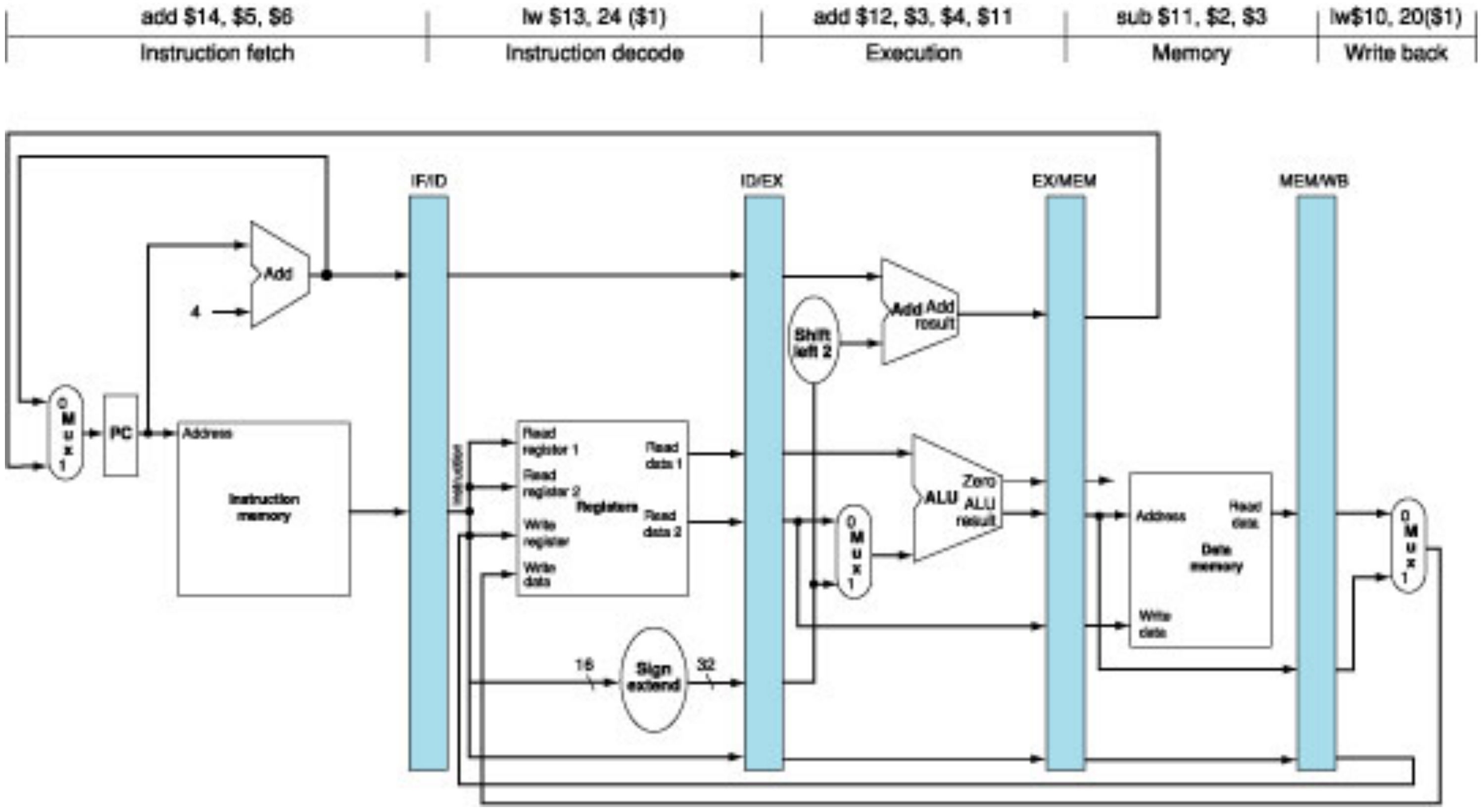
Pipelined lw Datapath



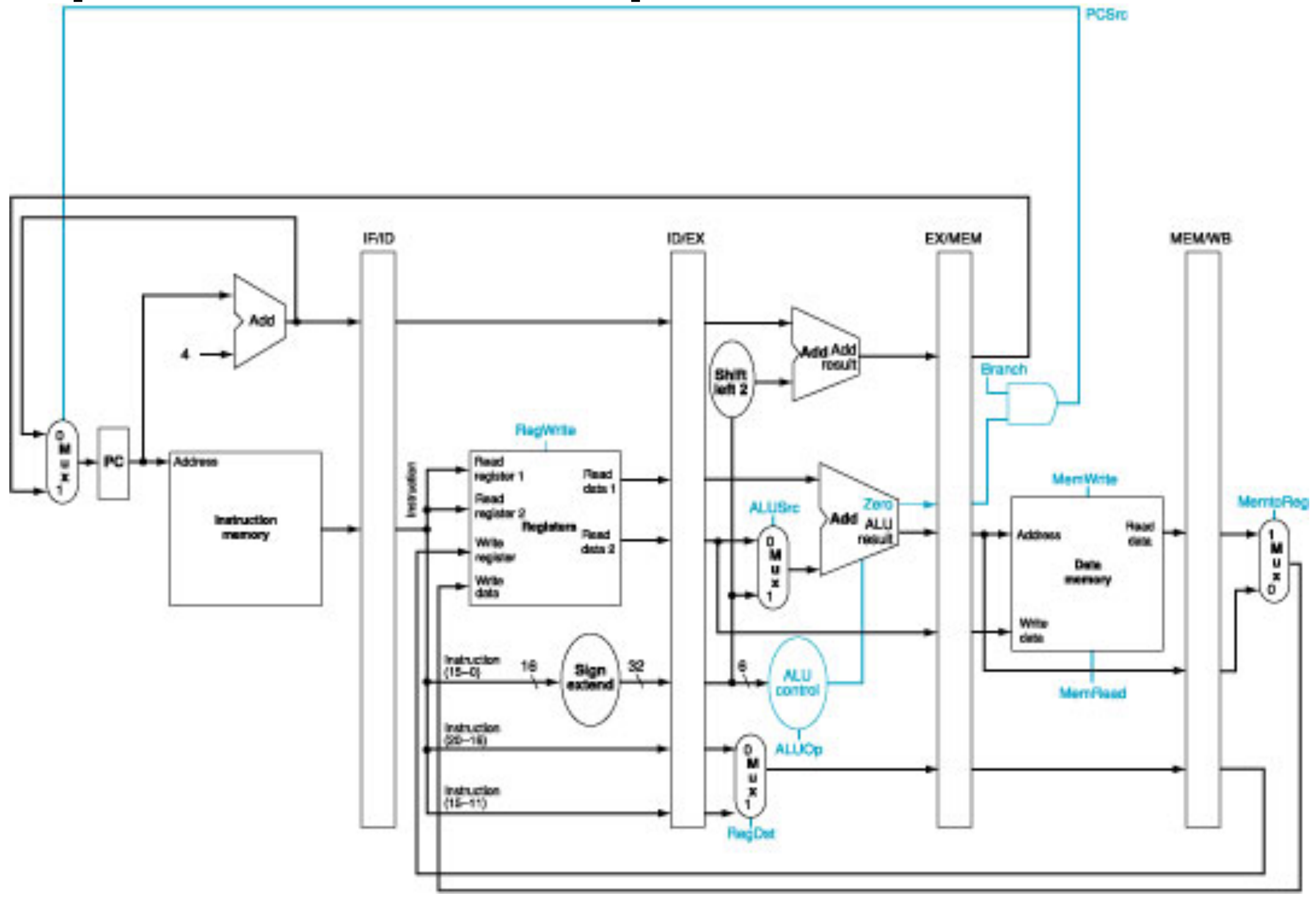
Multiple Clock Cycle Diagram



Single Clock Cycle Diagram



Pipelined Datapath & Control

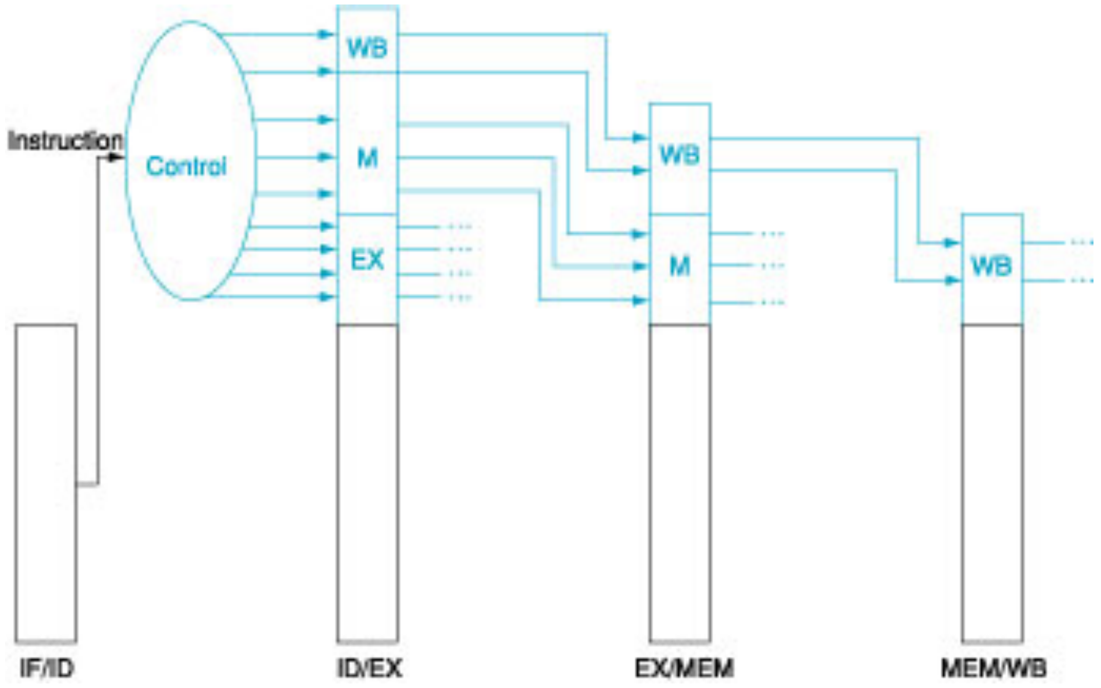


Control Signals per Stage

- IF: none (same actions always)
- ID: none (same actions always)
- EX: RegDst, ALUOp, ALUSrc
  - R-format vs. load/store
- MEM: MemWrite, MemRead, Branch
  - Branch vs. load vs. store
- WB: MemtoReg, RegWrite
  - Load vs. Store vs. R-format

Control Values in Pipeline Registers

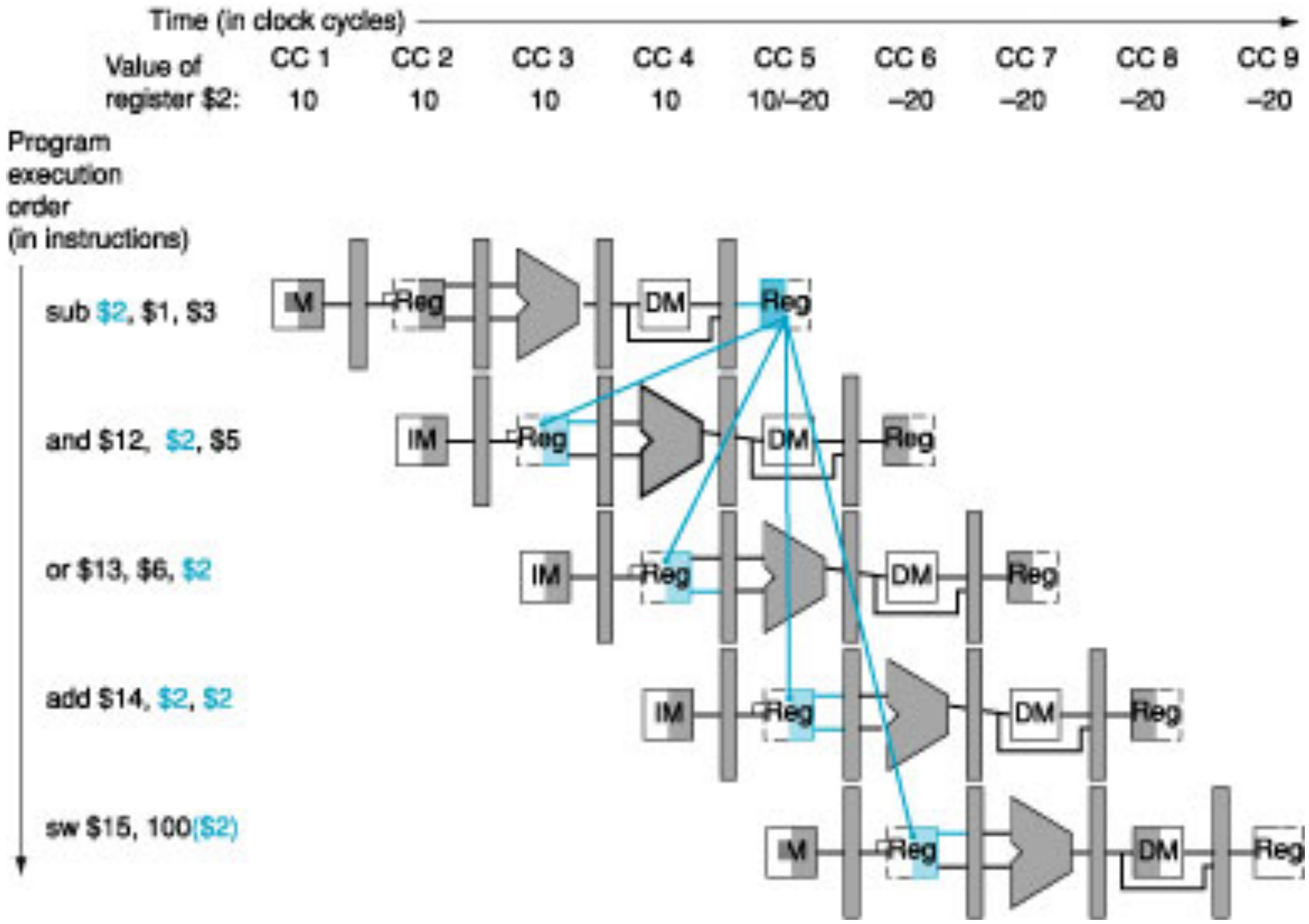
- Determine bits from instruction in ID stage
- Save bits in pipeline registers, passing through all necessary stages.



Considering Data Hazards

- Data hazards are caused by dependencies on earlier instructions
- Registers do not (yet) have the expected value when read
- Connect register-read to register-write; if the “arrow” goes back in time, there’s a data hazard (write-read dependency).

Dependency Example



Solution: Forwarding

- Make the value available (to beginning of EX) as soon as it is computed (end of EX)
- This solves all hazards in the previous example.
- There is also a potential forward from a computation to a store instruction
- More complex instruction sets have more forwarding possibilities

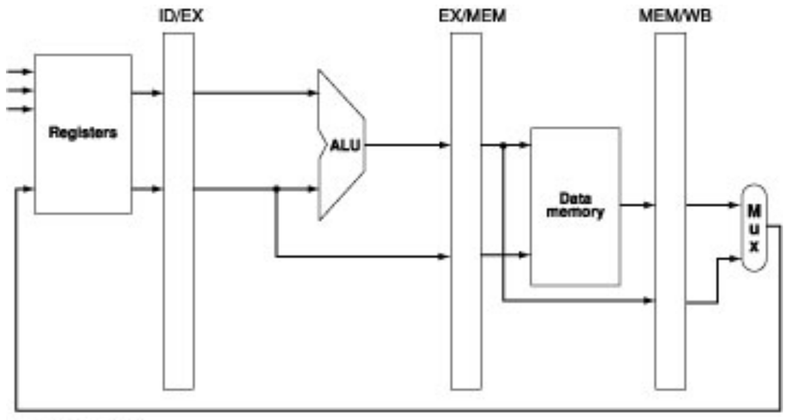
Recognizing Data Hazards

- Source register of current instruction = destination register of previous instruction
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
- Source register of current instruction = destination register of 2 instructions ago
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRd
- **AND** prior instruction will write the register
  - EX/MEM.RegWrite or MEM/WB.RegWrite is set

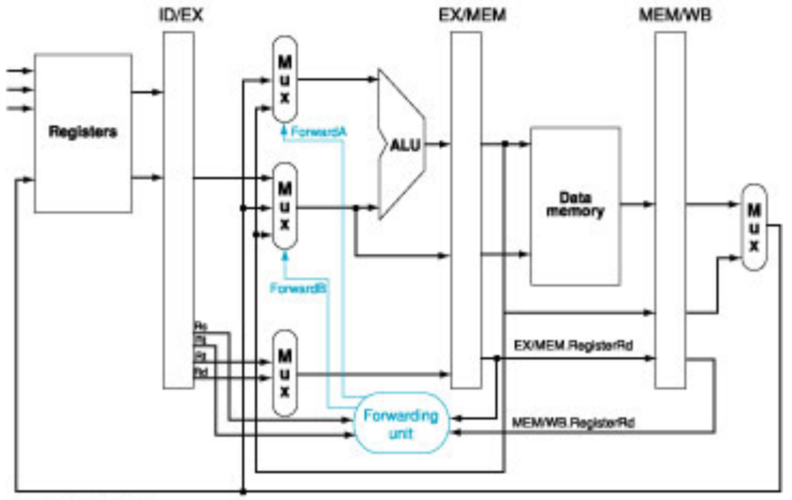
Implementing Forwarding

- Connect appropriate value from pipeline registers after EX (EX/Mem and Mem/WB) to ALU input
- Control ALU input multiplexor with logic that checks for hazards (previous slide)

Forwarding: Datapath & Control



a. No forwarding



b. With forwarding

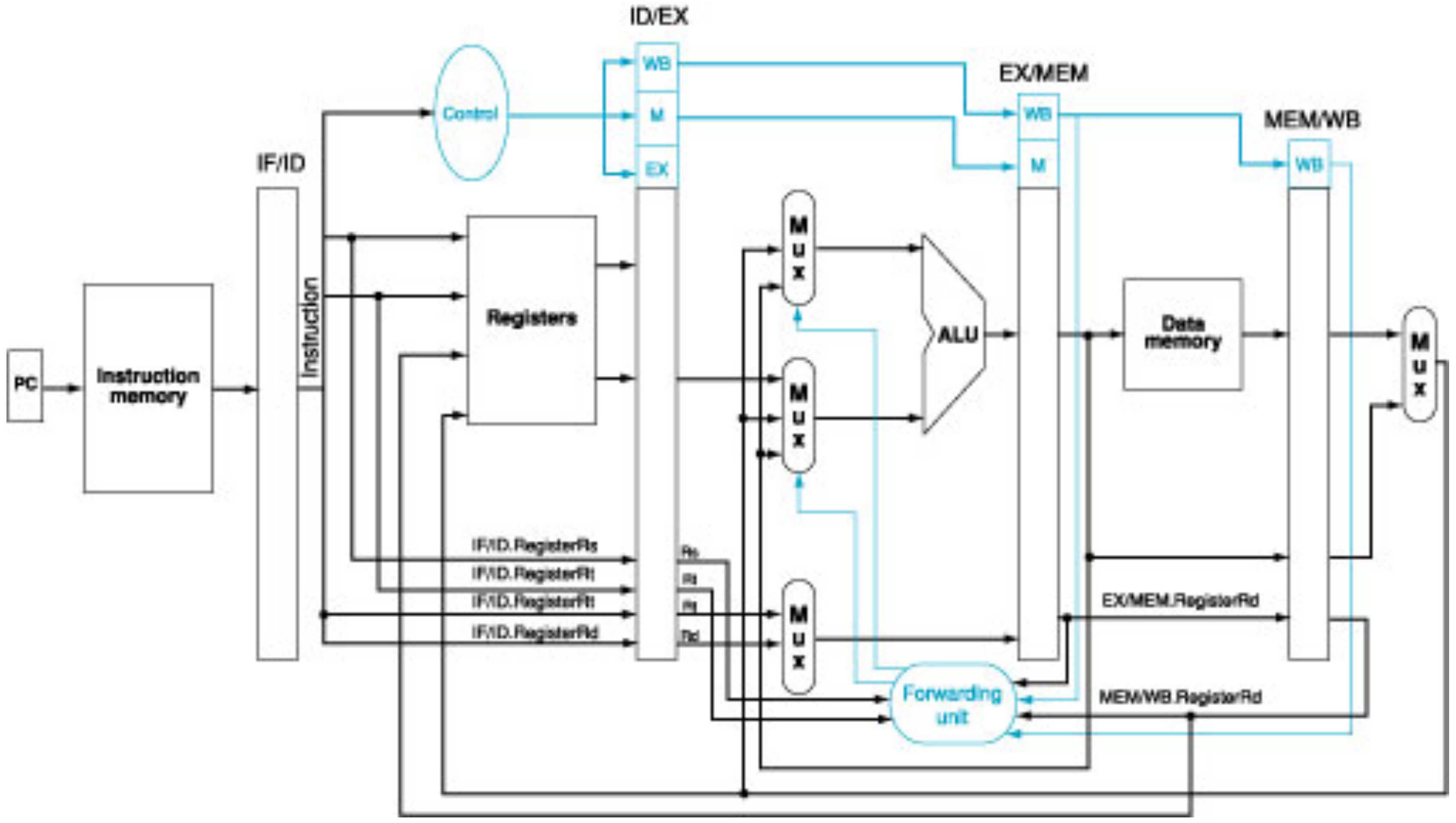
Forwarding: EX Hazard

- 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

Forwarding: MEM Hazard

- If (MEM/WB.RegWrite) and (MEM/WB.RegisterRd ≠ 0) and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs) and (MEM/WB.RegisterRd = ID/EX.RegisterRs) ForwardA = 01
- If (MEM/WB.RegWrite) and (MEM/WB.RegisterRd ≠ 0) and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs) and (MEM/WB.RegisterRd = ID/EX.RegisterRs) ForwardB = 01

Pipeline with Forwarding



Hazard Detection Unit

- Recognize situations where pipeline must be stalled
- Prevent execution of stalled instructions (already in the pipeline) for 1 cycle
- Insert bubbles (NOP instructions) in the back half of the pipeline

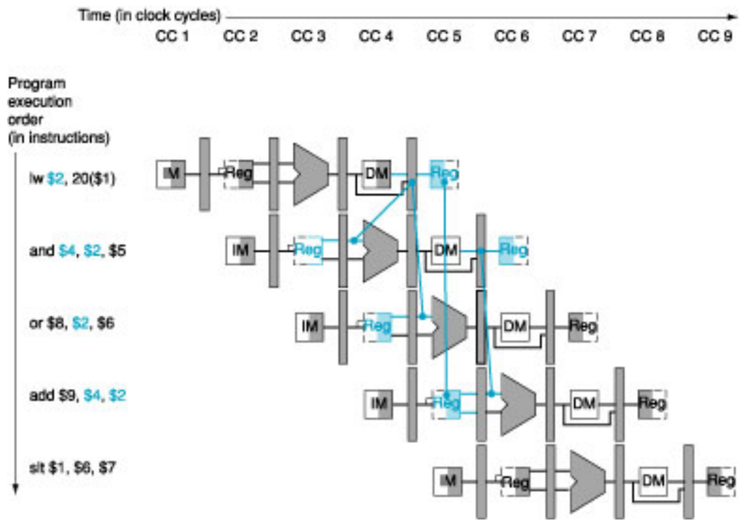
Detecting Load Hazard

- If the previous instruction was load, and one of the source registers for the current instruction is the value being loaded in the previous instruction, then stall
  - If (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))) then stall

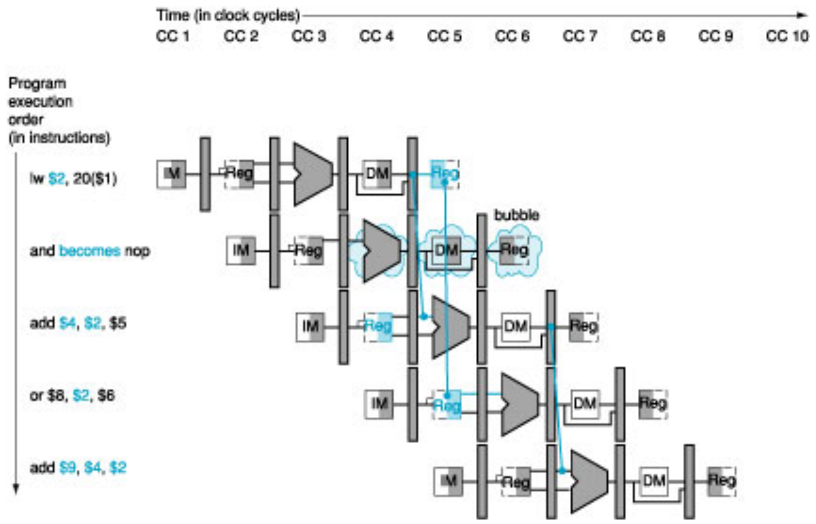
Stopping the Pipeline

- To avoid fetching or decoding new instructions (losing those in the pipeline), prevent PC and IF/ID registers from being written
- To prevent later stages from doing any work with possibly bad values, set all write control signals to 0 in the pipeline registers
  - No writes; therefore no changes!

Resolving the Hazard

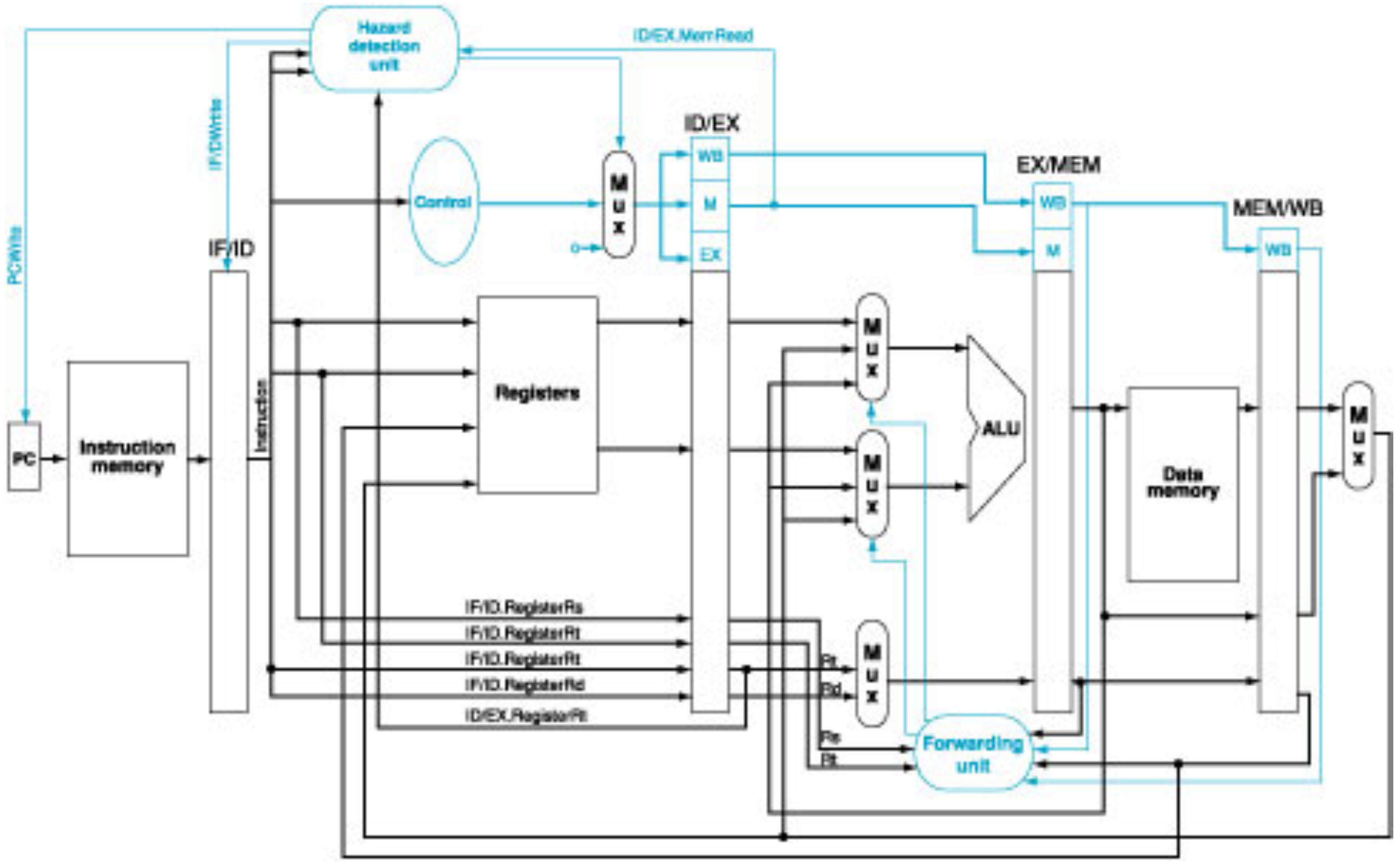


Original Hazard



Fixed with stall

Processor with Hazard Detection Unit



Control (Branch) Hazards

- Branch instruction
  - Stage 1: load the instruction
  - Stage 2: decode the instruction & get registers (being compared)
  - Stage 3: compute branch address, and determine equality (Zero)*******
  - Stage 4: feed back info to PC

Incorrectly Assume Branch Not Taken

IF (beq)	ID (beq)	EX (beq)	[Mem]	[WB]		
	IF (pc+1)	ID (pc+1)	EX (pc+1)	Mem (pc+1)		
		IF (pc +2)	ID (pc +2)	EX (pc +2)	Mem (pc+2)	
			IF (pc +3)	ID (pc +3)	EX (pc +3)	Mem (pc+3)
				IF (new)	ID (new)	EX (new)

Assume Branch Not Taken

- If the branch really is not taken, no change in pipeline
- If the branch is taken, insert bubbles for all 3 instructions in the pipeline
  - Set regWrite and memWrite to 0 in IF/ID, ID/EX, and EX/Mem
  - This will prevent any changes in state from taking place

Cost of Assumption

- Branch is not taken: 0 cost
- Branch is taken: 3 cycles cost
- Example: 50% of branches taken
  - Average penalty 1.5 cycles
- Example: 25% of branches taken
  - Average penalty 0.75 cycles (0.25*3)

Reducing the Delay

- Determine branch address in ID instead of EX
  - Move the extra adder back into ID instead of in parallel with CPU in EX
- Determine result of branch test earlier
  - Add specialized logic for detection in ID
  - Copy forwarding logic into ID in cases of register data hazard
  - Add logic regarding this test to hazard detection unit (until relevant register values are computed by EX)
- Transform the instruction currently in IF/ID to NOP

Reduced Delay Datapath

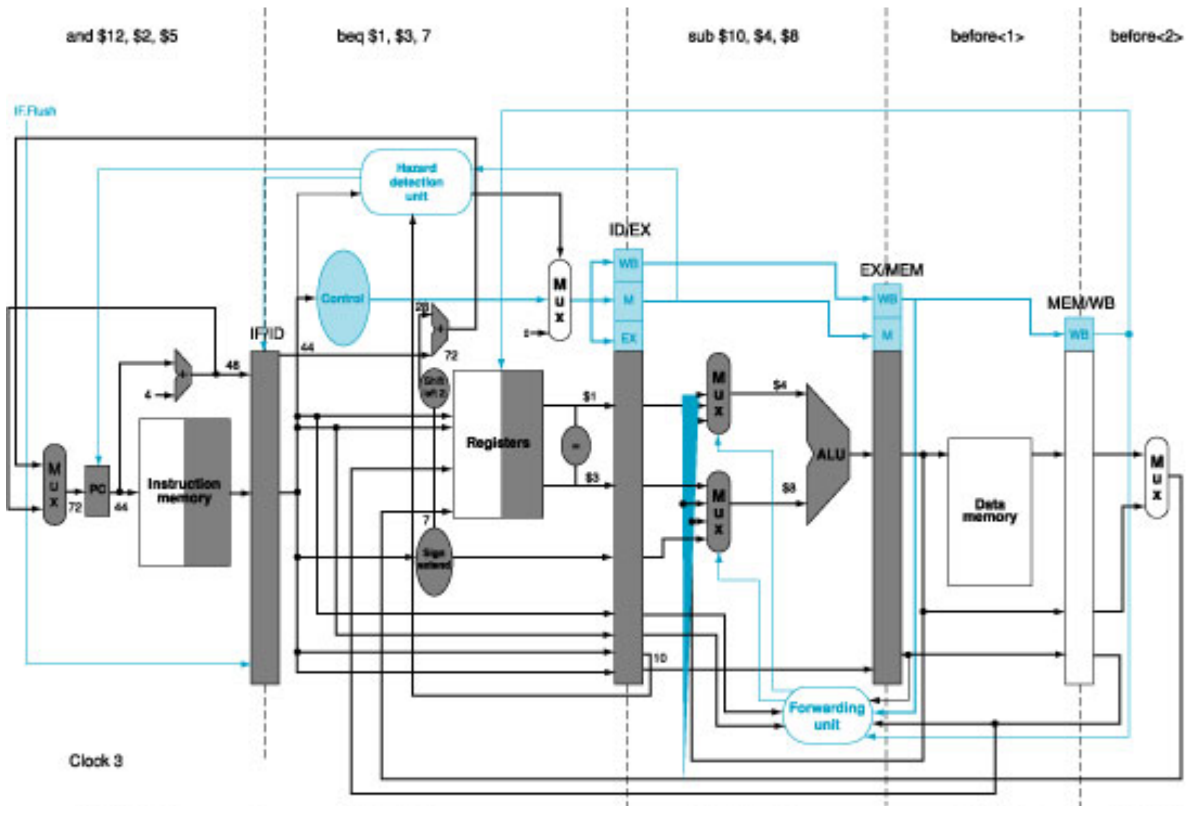


Figure 6.38 (top)

Dynamic Branch Prediction

- For each branch, keep track of how often the branch is taken, and make the prediction
- If prediction says “don’t take”, use the scheme from before
- If prediction says “take”, immediately reset PC (based on PC address, so it is done in IF step)

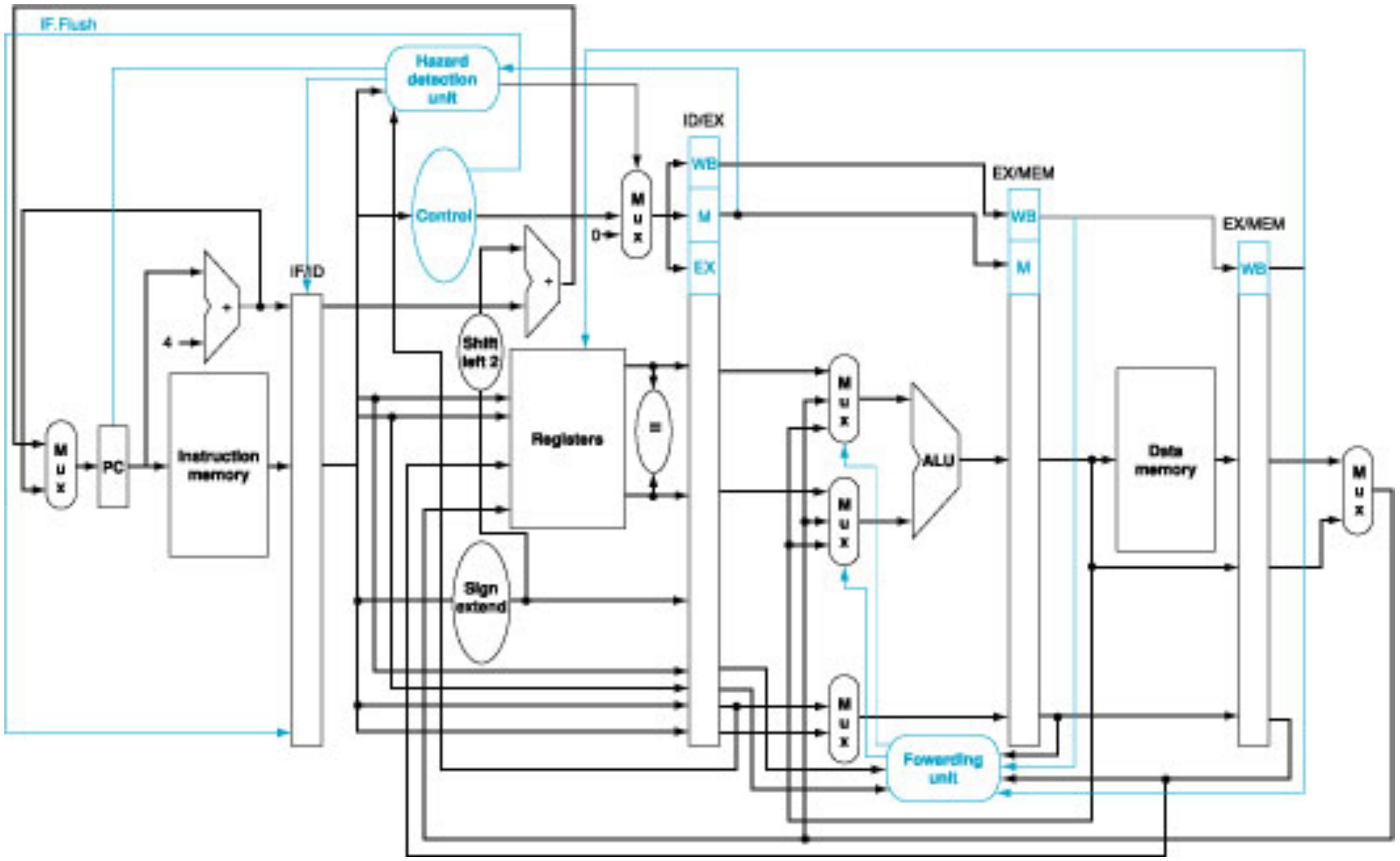
Branch Prediction Buffer

- Indexed by low-order bits of address
  - Simple scheme: 1 bit is set when taken, reset when not. For a loop, it’s wrong twice (beginning and end).
  - Two-bit buffer: bit is only flipped when the prediction is wrong twice in a row. (This only misses once in a loop).
  - Many more complex schemes…

Branch Delay Slot

- To avoid NOP’ing the instruction immediately after the branch…
- Put an instruction that should be always executed there! (compiler or assembler)
  - Instruction immediately before branch
  - If that’s not possible for some reason
    - Instruction at target (if branch-taken likely)
    - Instruction after branch (if branch-taken unlikely)

Final Data/Control



Pipelined Performance

- Determine clock cycle (e.g. 200ps)
- Determine latency for each instruction
  - ALU, store = 1 cycle
  - Jump = 2 cycles
  - Load = 1.5 cycles (assuming 50% load/read hazards)
  - Branch = 1.25 cycles (assuming 25% wrong predictions and 1 cycle penalty)

Pipelined Performance (cont)

- Compute weighted average, based on instruction mix
  - 25% loads, 10% stores, 11% branches, 2% jumps, 52% ALU
  - .25*1.5+.1*1+.11*1.25+.02*2+.52*1 = 1.17
  - 1.17 * 200ps = 234ps (average instruction time)