Very Large Scale Integration II - VLSI II

Basic Pipelining (cont'd)

Y. Fırat Kula

ITU VLSI Laboratories

Istanbul Technical University

Basic Pipelining (cont'd)

- Data Hazard Handling
  - RAW Hazards
  - Load-Use Hazards
- Control Hazard Handling
  - Jumps
  - Branches

Hazards Review

- Structural
  - Use seperate memories
  - Half-cycle register operation
- Data
  - RAW Hazards (Bypassing)
  - Load-Use Hazards (Bypassing & Stalling)
- Control
  - Jumps
  - Branches

Stalling solves it all, but we want to avoid that…

Hazards Review

- Structural
  - Use seperate memories
  - Half-cycle register operation
- Data
  - RAW Hazards (Bypassing)
  - Load-Use Hazards (Bypassing & Stalling)
- Control
  - Jumps
  - Branches

Read-After-Write (RAW) Hazard

- Consider the following sequence…

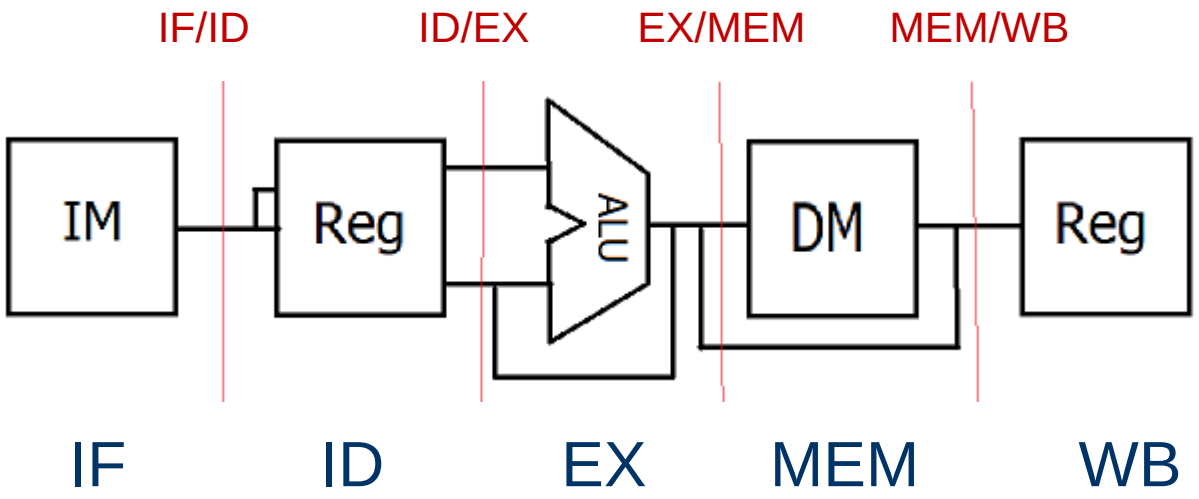
```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

- We can resolve this with forwarding

How do we decide when to forward?

Read-After-Write (RAW) Hazard

- Reminder…



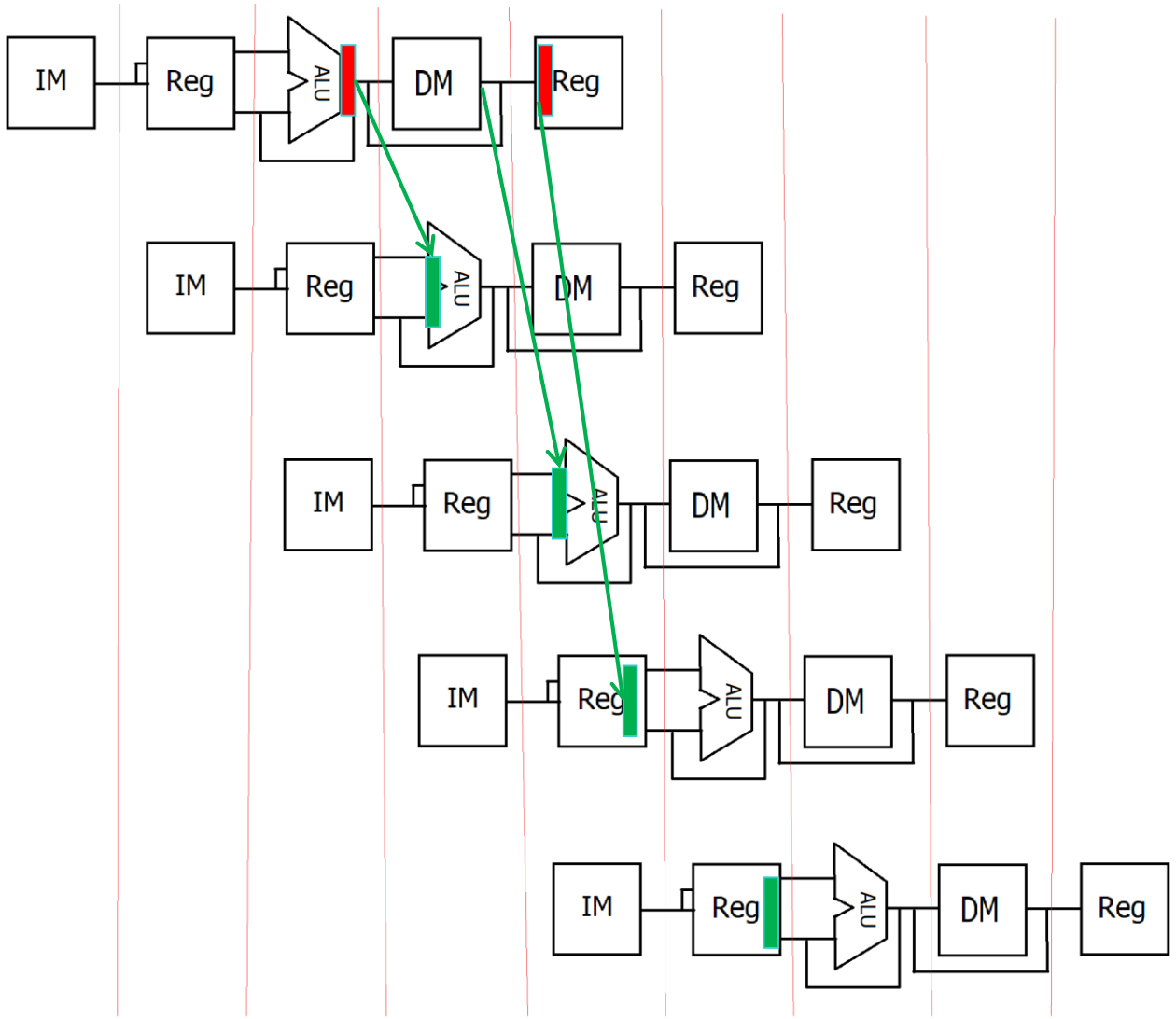
sub x2, x1,x3

and x12, x2,x5

or x13, x6,x2

add x14, x2,x2

sd x15, 100(x2)

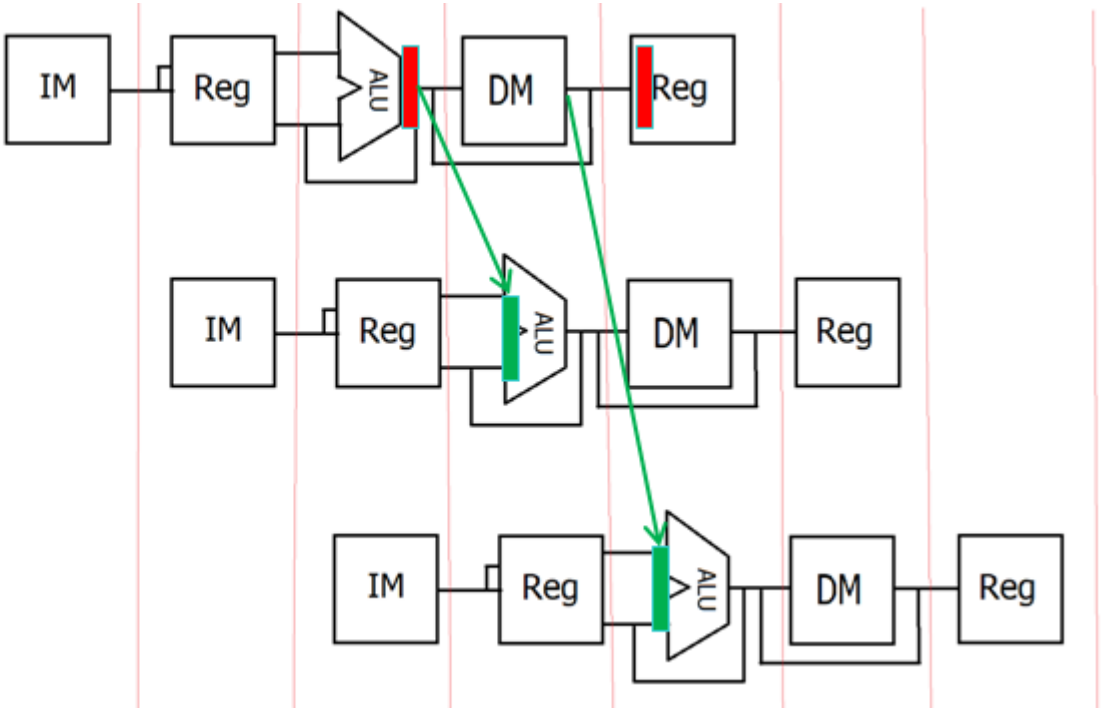


Read-After-Write (RAW) Hazard

sub x2, x1,x3

and x12, x2,x5

or x13, x6,x2



Forwarding: How to decide?

- Pass register numbers along pipeline
  - ID/EX.RegisterRs = register number for Rs in ID/EX
  - ID/EX.RegisterRt = register number for Rt in ID/EX
  - ID/EX.RegisterRd = register number for Rd in ID/EX

ALU inputs

Destination Register

- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- Second previous is in the MEM/WB register
- RAW Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Forwarding: How to decide?

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

Forwarding: How to decide?

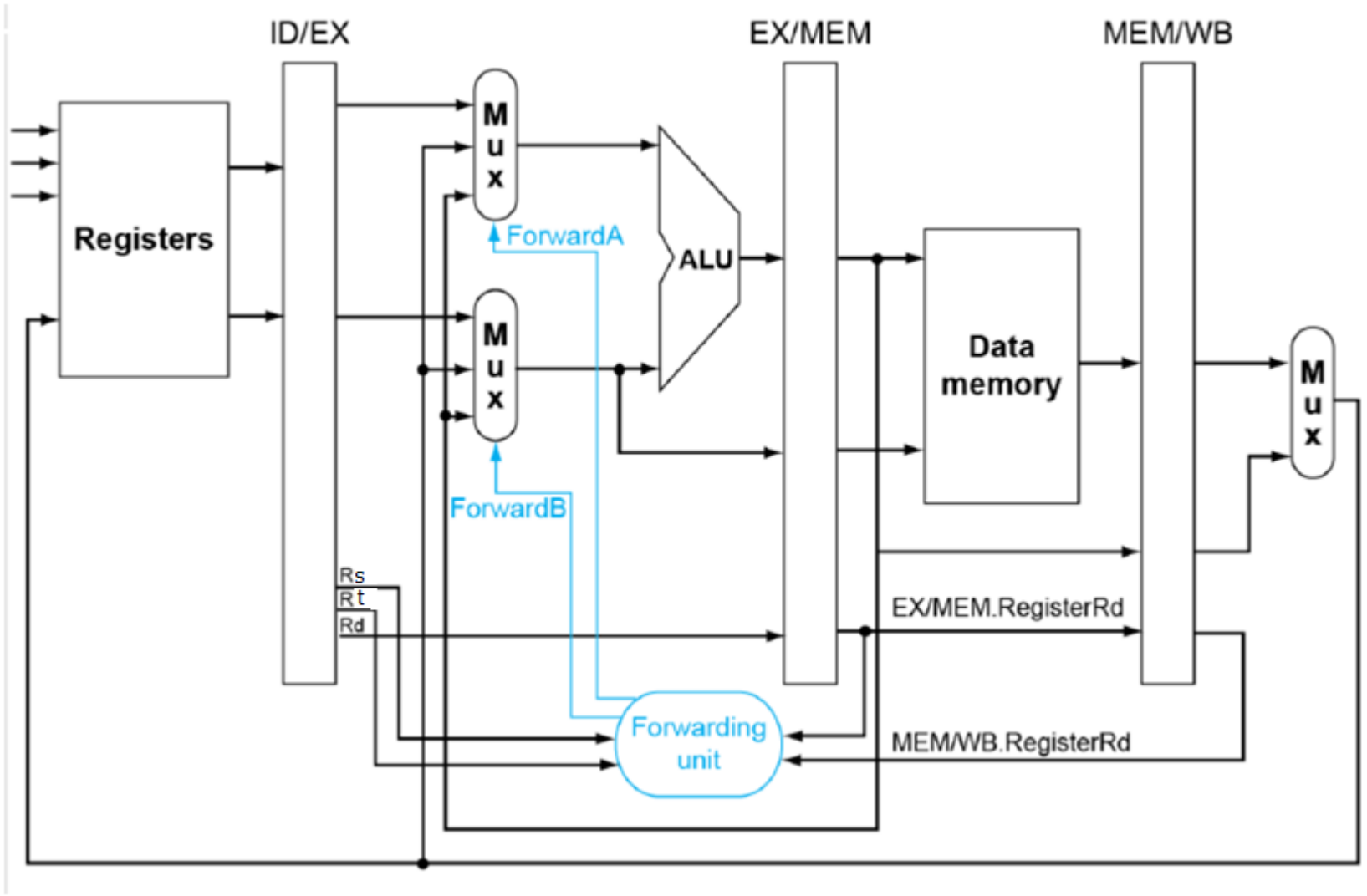
- Detecting RAW hazard with Previous Instruction
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01 (Forward from EX/MEM pipe stage)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01 (Forward from EX/MEM pipe stage)

Forward from EX logic (Forward from ALU output)

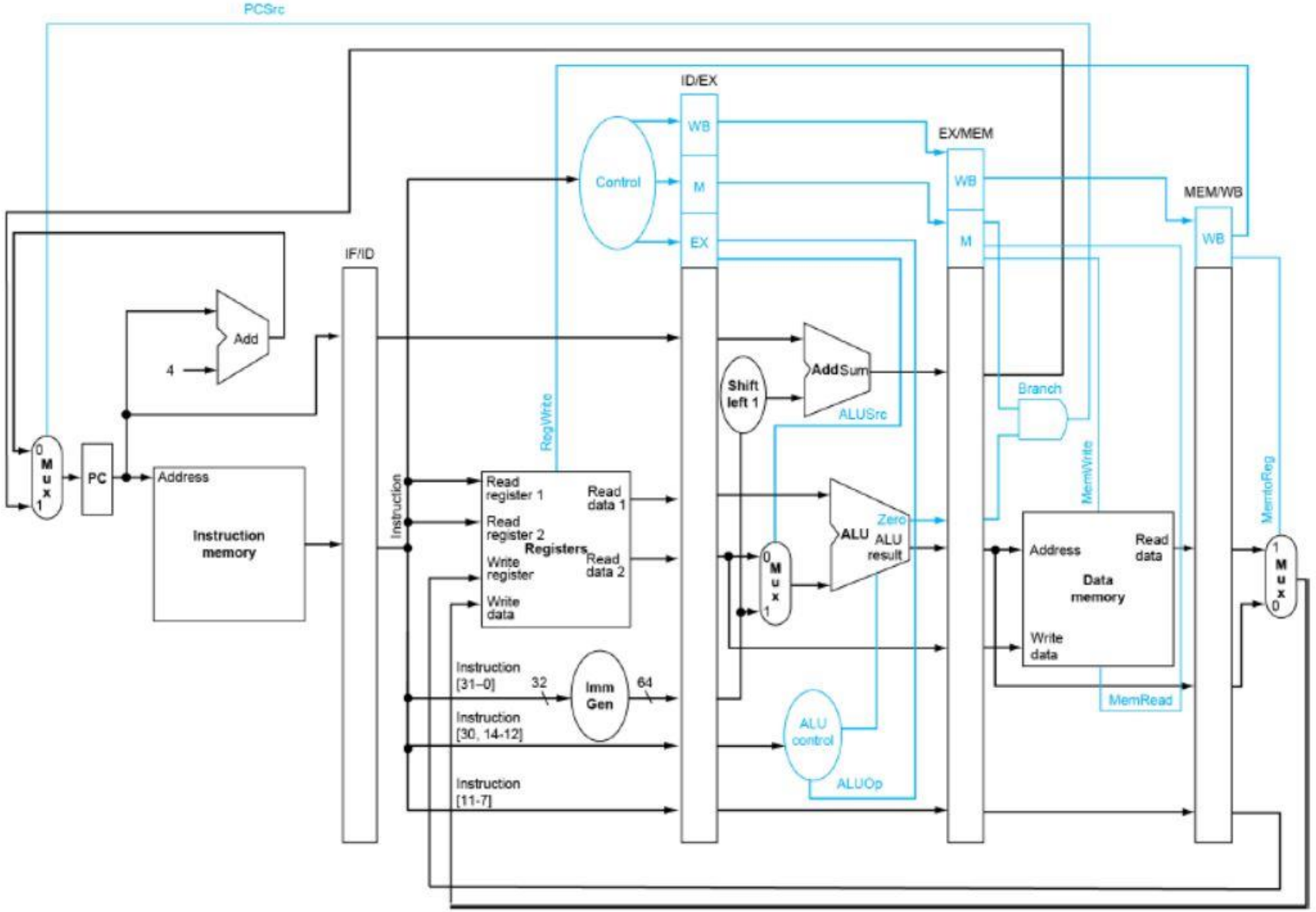
- Detecting RAW hazard with Second Previous
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10 (Forward from MEM/WB pipe stage)
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10 (Forward from MEM/WB pipe stage)

Forward from MEM logic (Forward from Data Memory Output)

Forwarding Hardware



Reminder: Pipelined Control Signals



Forwarding Control Signals

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Yet Another Complication…

- Consider the following sequence…

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

Which one to forward?

› Both hazards occur
  – Want to use the most recent

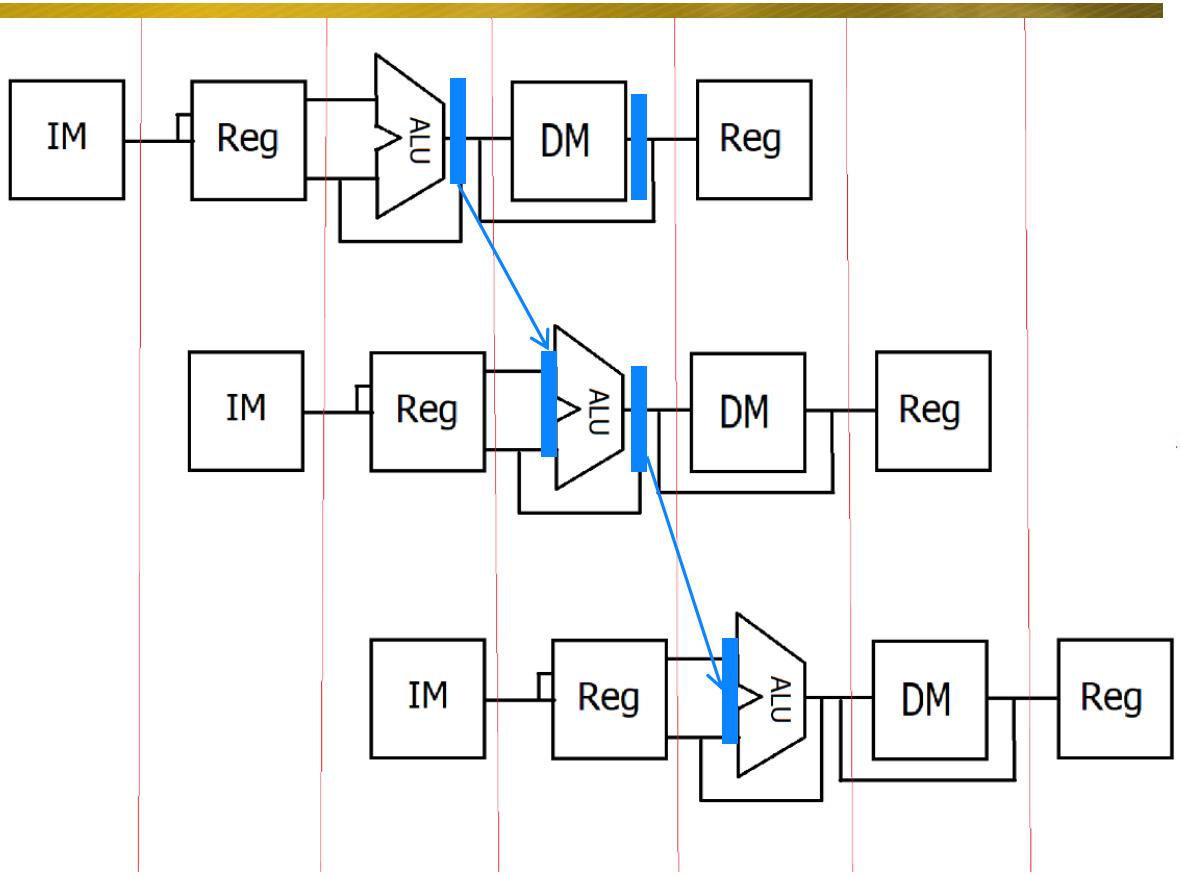
We must revise our MEM forward condition – Only forward if EX hazard condition is not true

Yet Another Complication…

add x1, x1,x2

add x1, x1,x3

add x1, x1,x4



Corrected Forwarding Logic

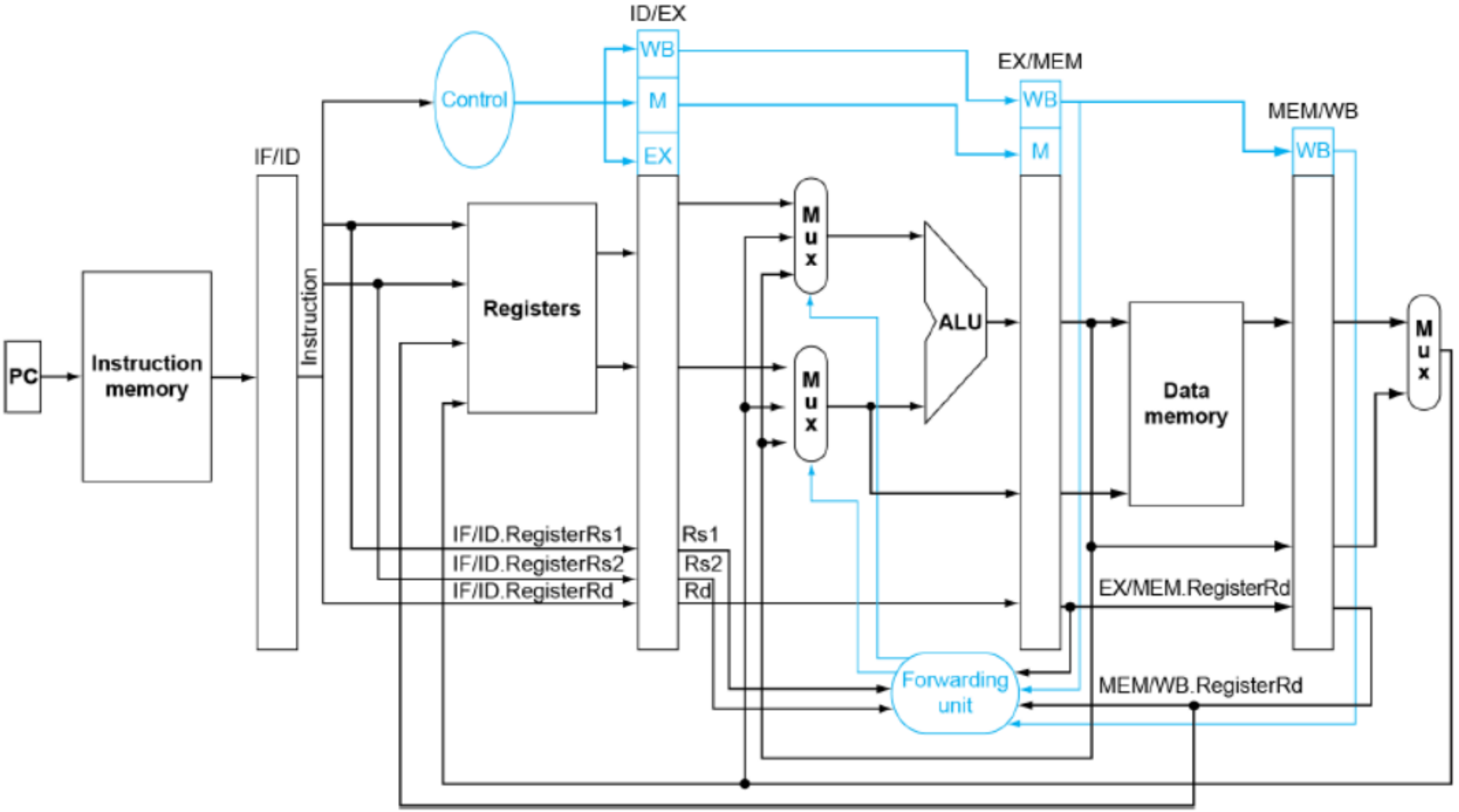
- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01 (Forward from EX/MEM pipe stage)
- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01 (Forward from EX/MEM pipe stage)

Forward from EX logic (Forward from ALU output)

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
  ForwardA = 10 (Forward from MEM/WB pipe stage)
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
  ForwardB = 10 (Forward from MEM/WB pipe stage)

Forward from MEM logic (Forward from Data Memory Output)

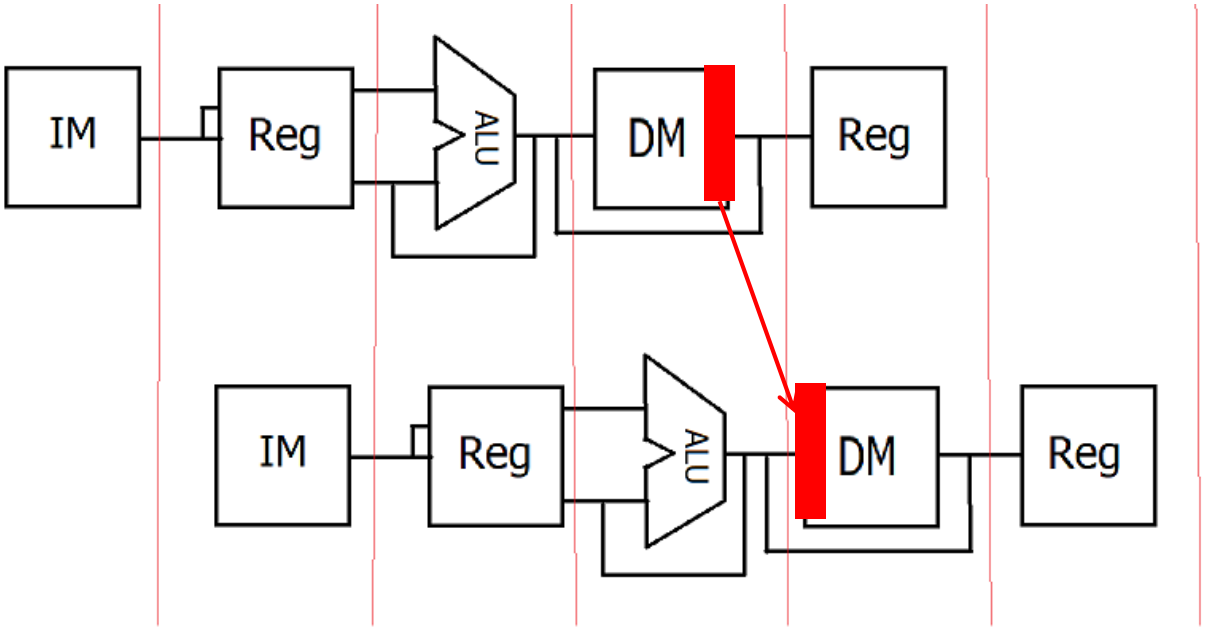
Datapath with Complete Forwarding Hardware



Memory-to-Memory Copies

lw x1, 4(x2)

sw x1, 4(x3)



- Can be resolved by adding a Forward Unit and a MUX to the MEM stage

Hazards Review

- Structural
  - Use seperate memories
  - Half-cycle register operation
- Data
  - RAW Hazards (Bypassing)
  - Load-Use Hazards (Bypassing & Stalling)
- Control
  - Jumps
  - Branches

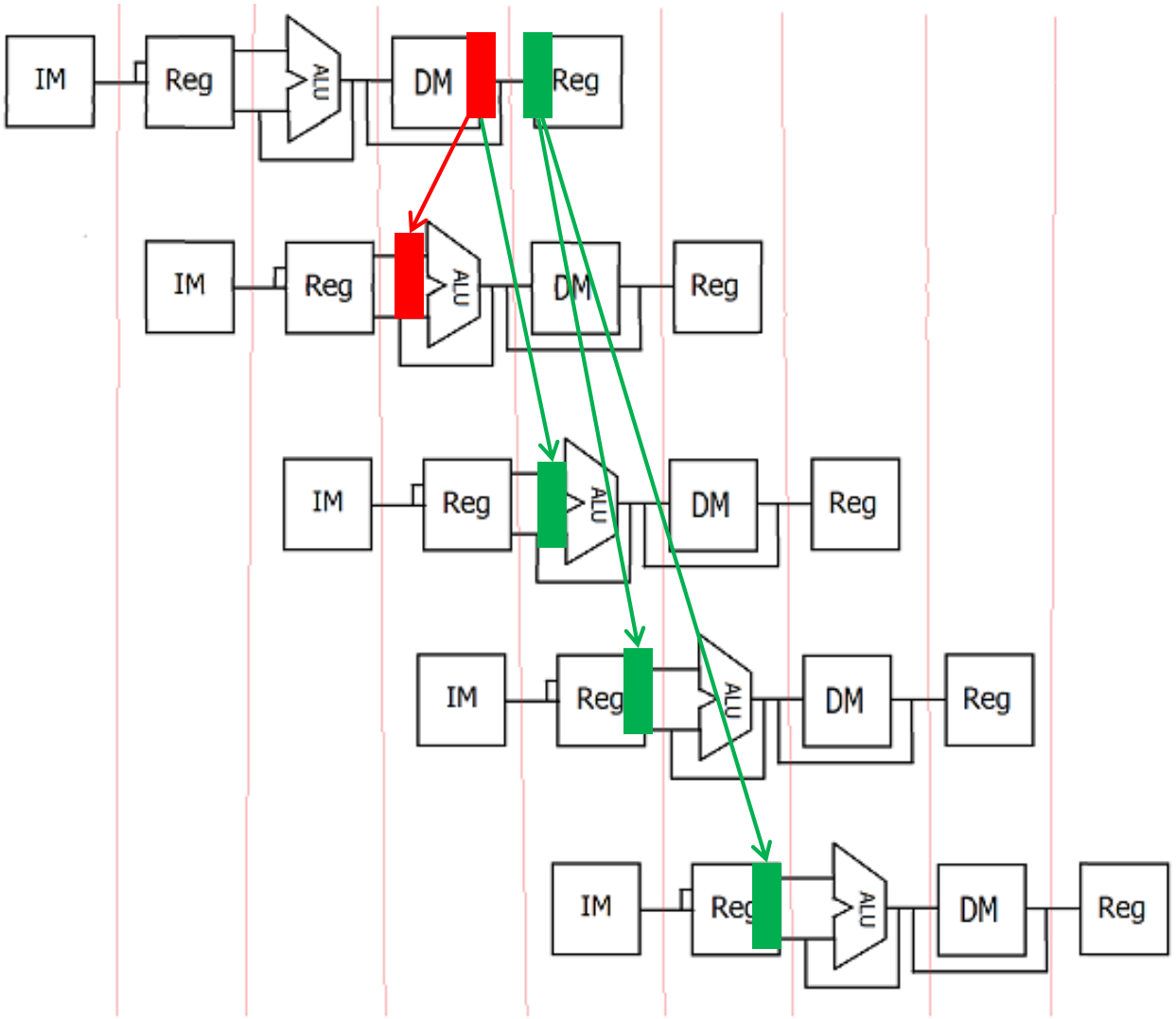
Load-Use Hazards

lw x2, 4(x1)

and x4, x2,x5

or x8, x2,x6

add x9, x4,x2



Load-Use Hazards

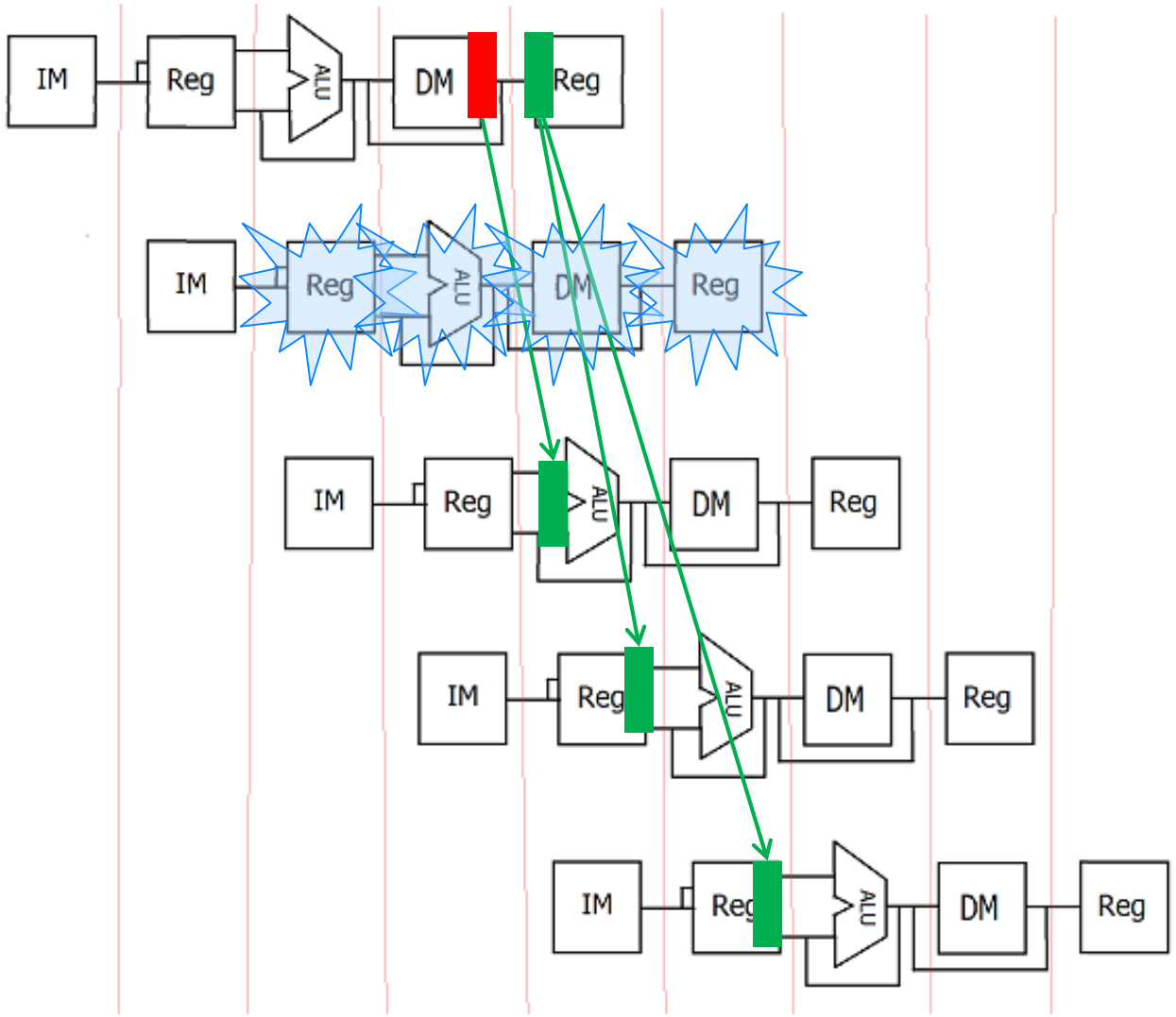
lw x2, 4(x1)

STALL

and x4, x2,x5

or x8, x2,x6

add x9, x4,x2



Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when

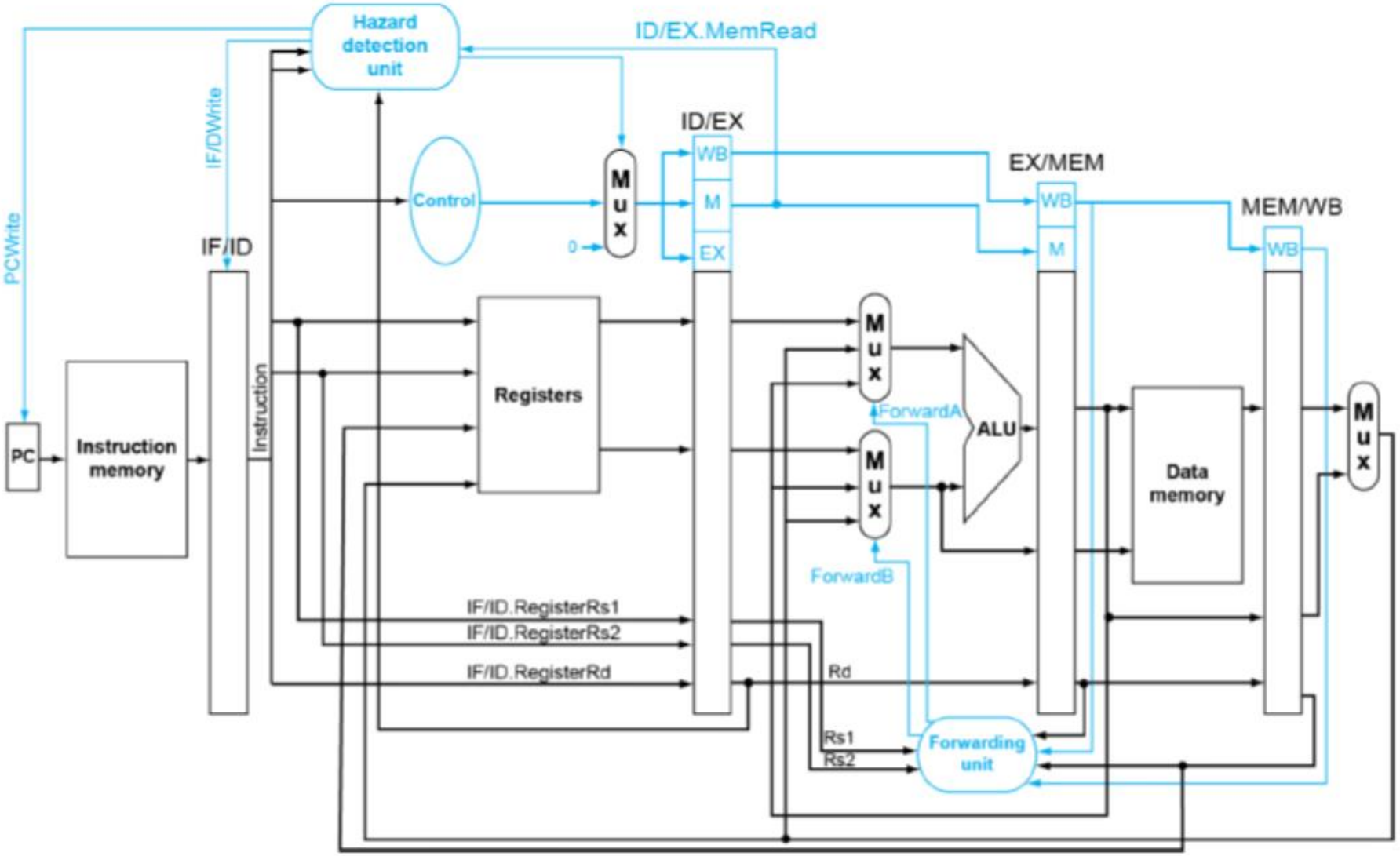
```
if(ID/EX.MemRead)
and (ID/EX.RegisterRd = IF/ID.RegisterRs)
or (ID/EX.RegisterRd = IF/ID.RegisterRt)
stall the pipeline (insert bubble)
```

How to Stall in Hardware?

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

A Hazard stall unit, and a MUX that selects between original control signals and zeros

Datapath with Hazard Stall Hardware



Load-Use Hazards

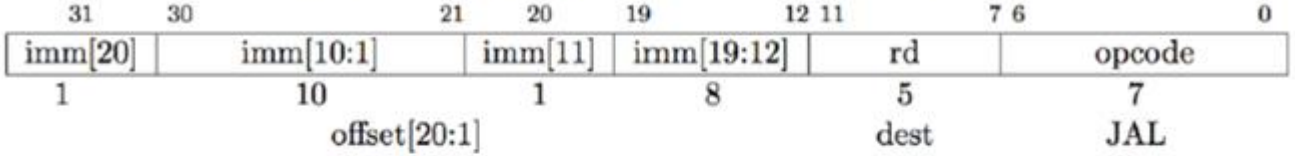
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards

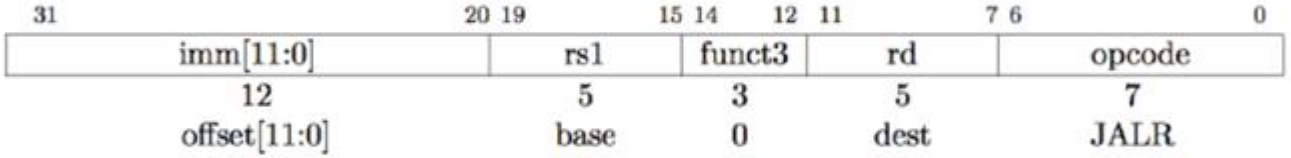
- When the flow of instruction adress is not sequential (PC = PC + 4)
  - Jumps (jal, jalr…)
  - Conditional Branches (beq, bge…)
  - Exceptions (unknown opcodes)
- Much less frequent than data hazards, but harder to deal with

RISC-V Jumps and Branches

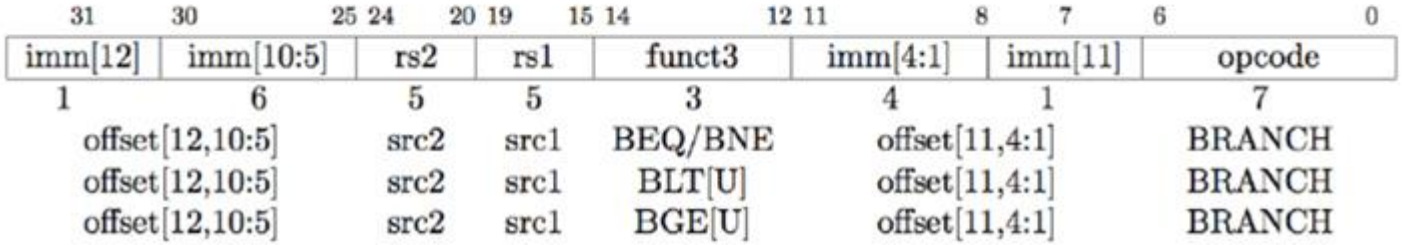
- **JAL: unconditional jump to PC+immediate**



- **JALR: indirect jump to rs1+immediate**



- **Branch: if (rs1 conds rs2), branch to PC+immediate**



RISC-V Jumps and Branches

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

- **JAL: unconditional jump to PC+immediate**
- **JALR: indirect jump to rs1+immediate**
- **Branch: if (rs1 conds rs2), branch to PC+immediate**

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Reg. Fetch
B<cond.>	After Execute	After Inst. Decode

Hazards Review

- Structural
  - Use seperate memories
  - Half-cycle register operation
- Data
  - RAW Hazards (Bypassing)
  - Load-Use Hazards (Bypassing & Stalling)
- Control
  - Jumps
  - Branches

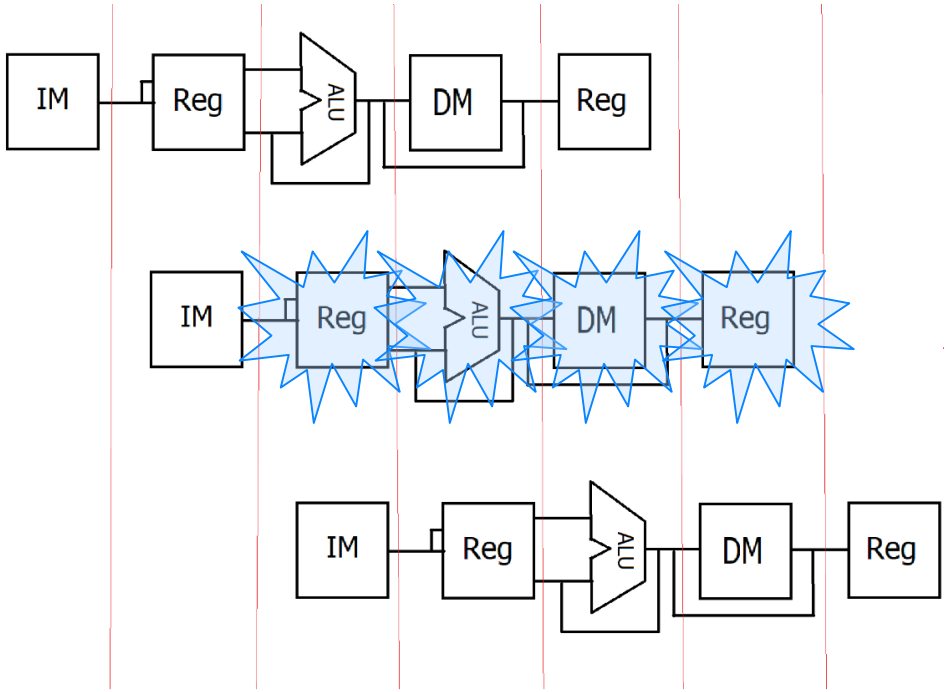
Jumps

- Jumps are decoded in ID, so wee need one flush…

(jump)

“flush” this insturction

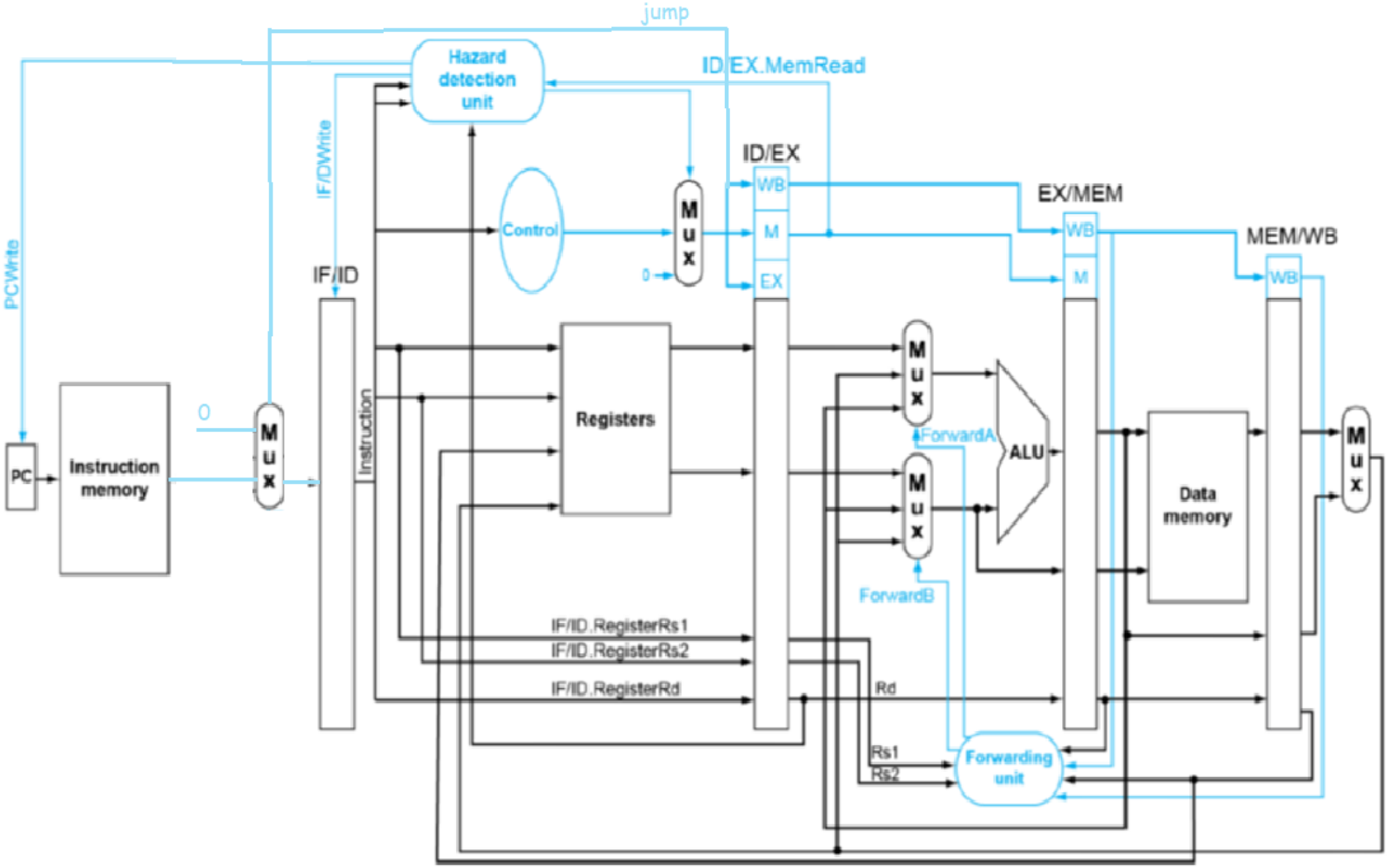
(jump target)



“Bubble” and “Flush”

- Bubble is when we insert a “nop” in the pipeline
  - We prevent the instructions earlier in the pipeline from progressing down the pipeline, for one cycle, by zeroing control signals
  - Later instructions progresses normally
- Flush is when we replace an ongoing instruction with a “nop”
  - Zero the control signals for the instruction to be flushed (we zeroed all bits in IF/ID register, before decoding the control signals)

Flushing for Jumps



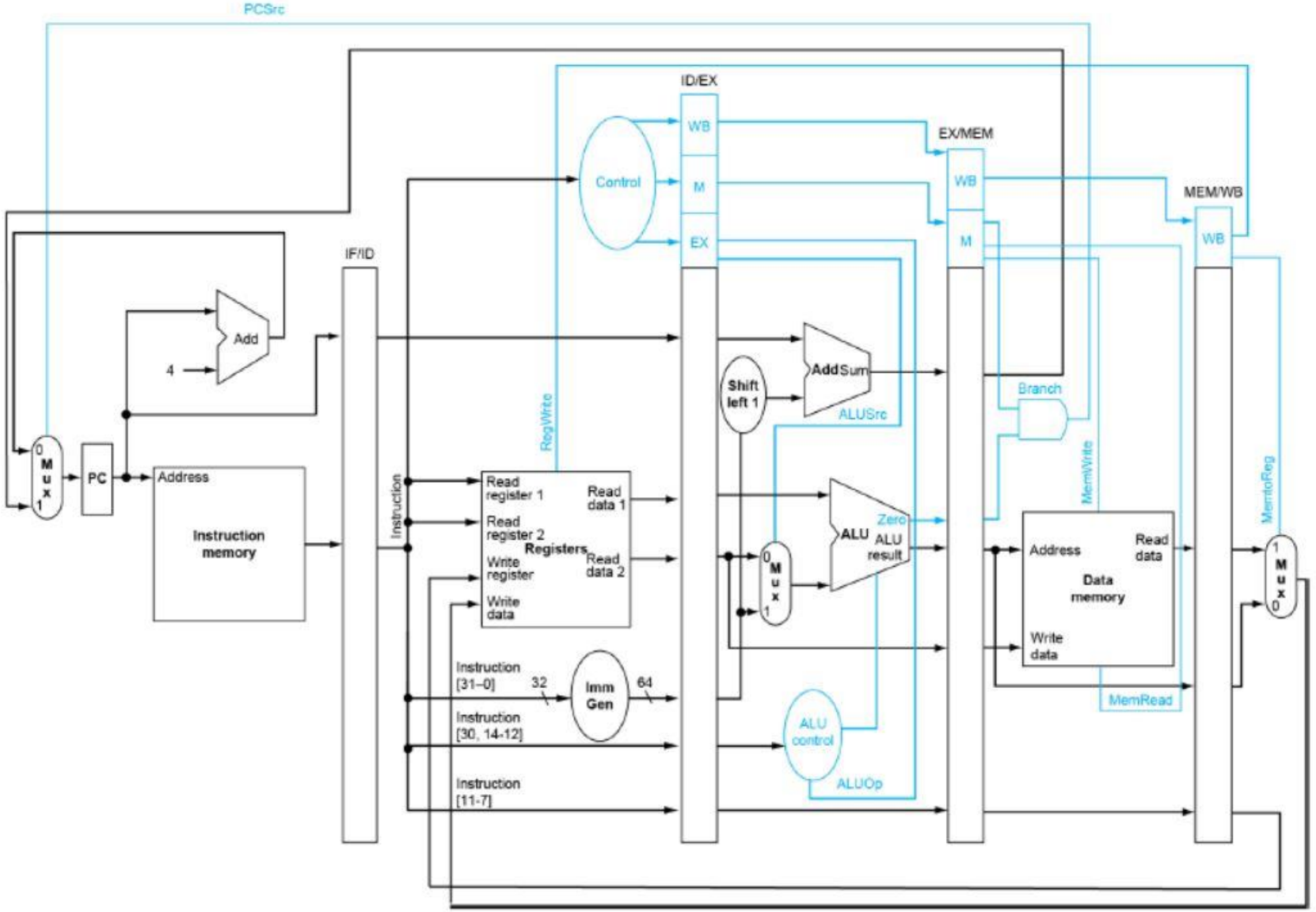
Jumps

- Can we resolve jumps in IF stage (CPI=1) ?
  - Pre-calculated Look-up Tables can be added in IF stage (compiler+hardware support)

Hazards Review

- Structural
  - Use seperate memories
  - Half-cycle register operation
- Data
  - RAW Hazards (Bypassing)
  - Load-Use Hazards (Bypassing & Stalling)
- Control
  - Jumps
  - Branches

Reminder: Pipelined Control Signals



Branches

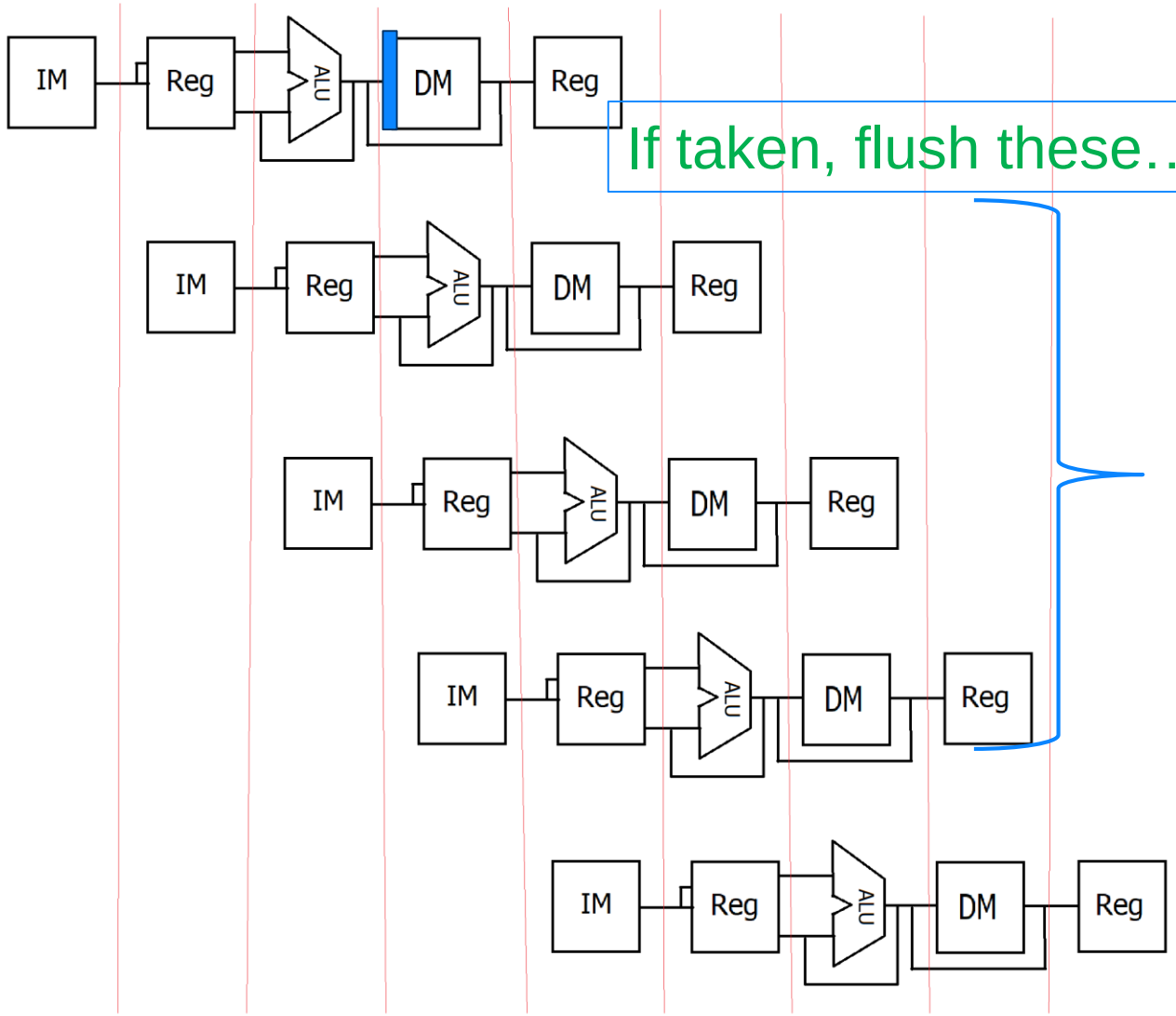
beq x1, x0,16

and x12, x2,x5

or x13, x6,x2

add x14, x2,x2

lw x4, 100(x7)



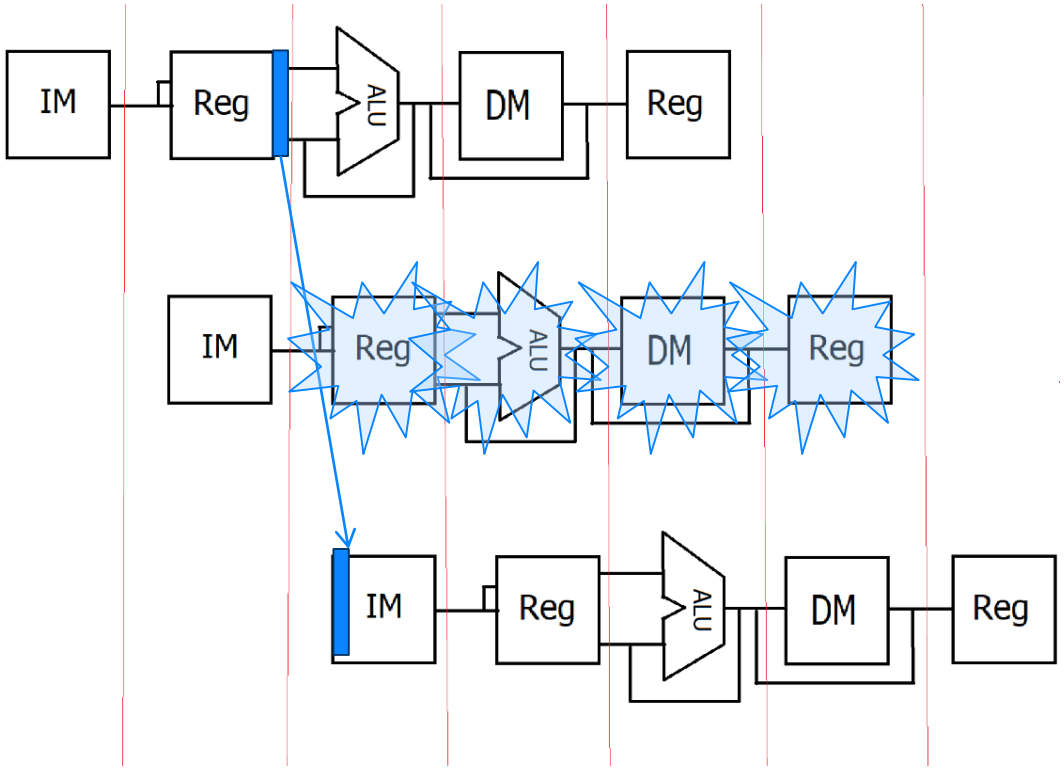
Branches: A Better Way

- Move branch decision hardware earlier in the pipeline

beq

flush

(beq target)

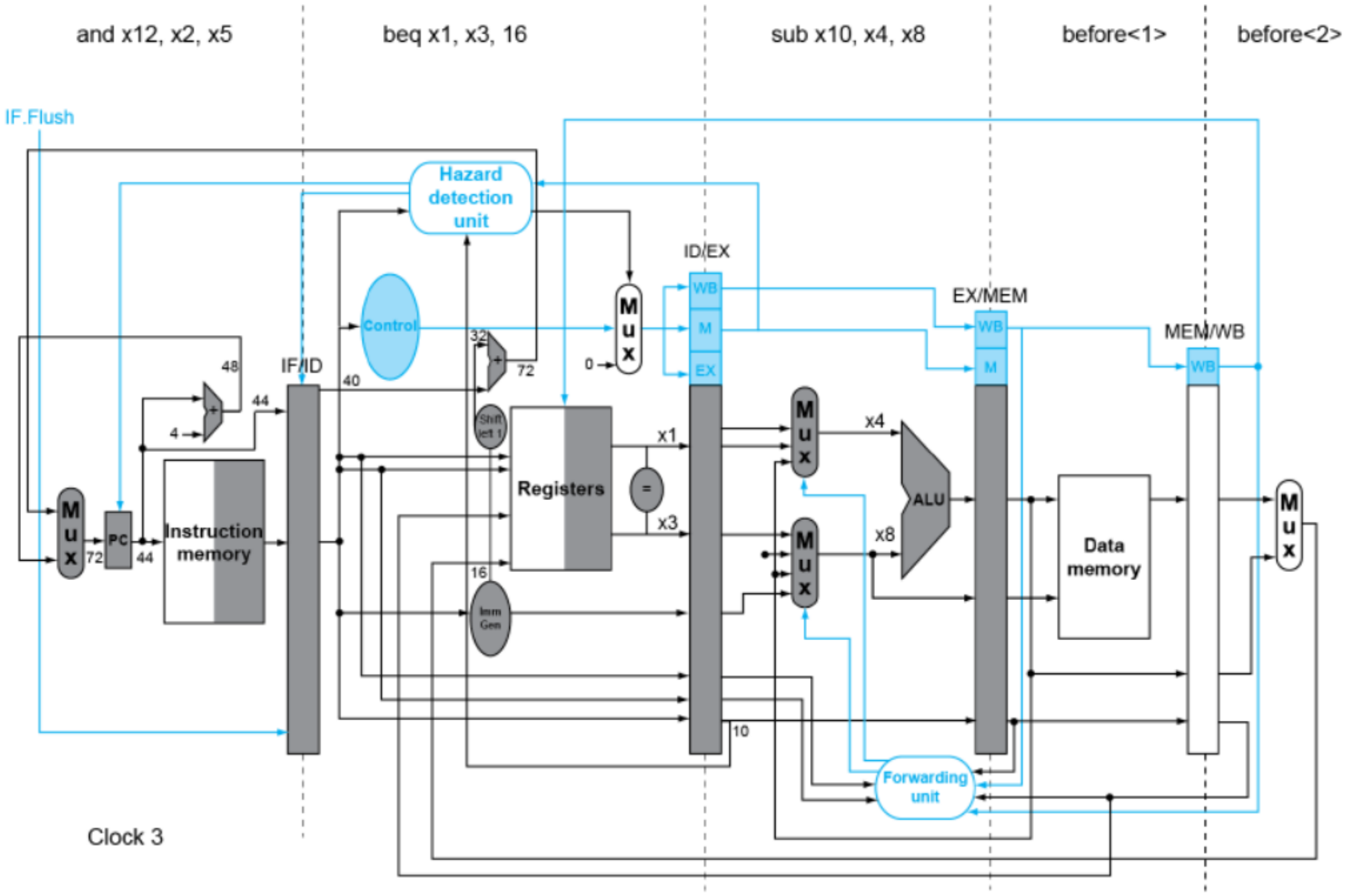


Revised Branch Decision

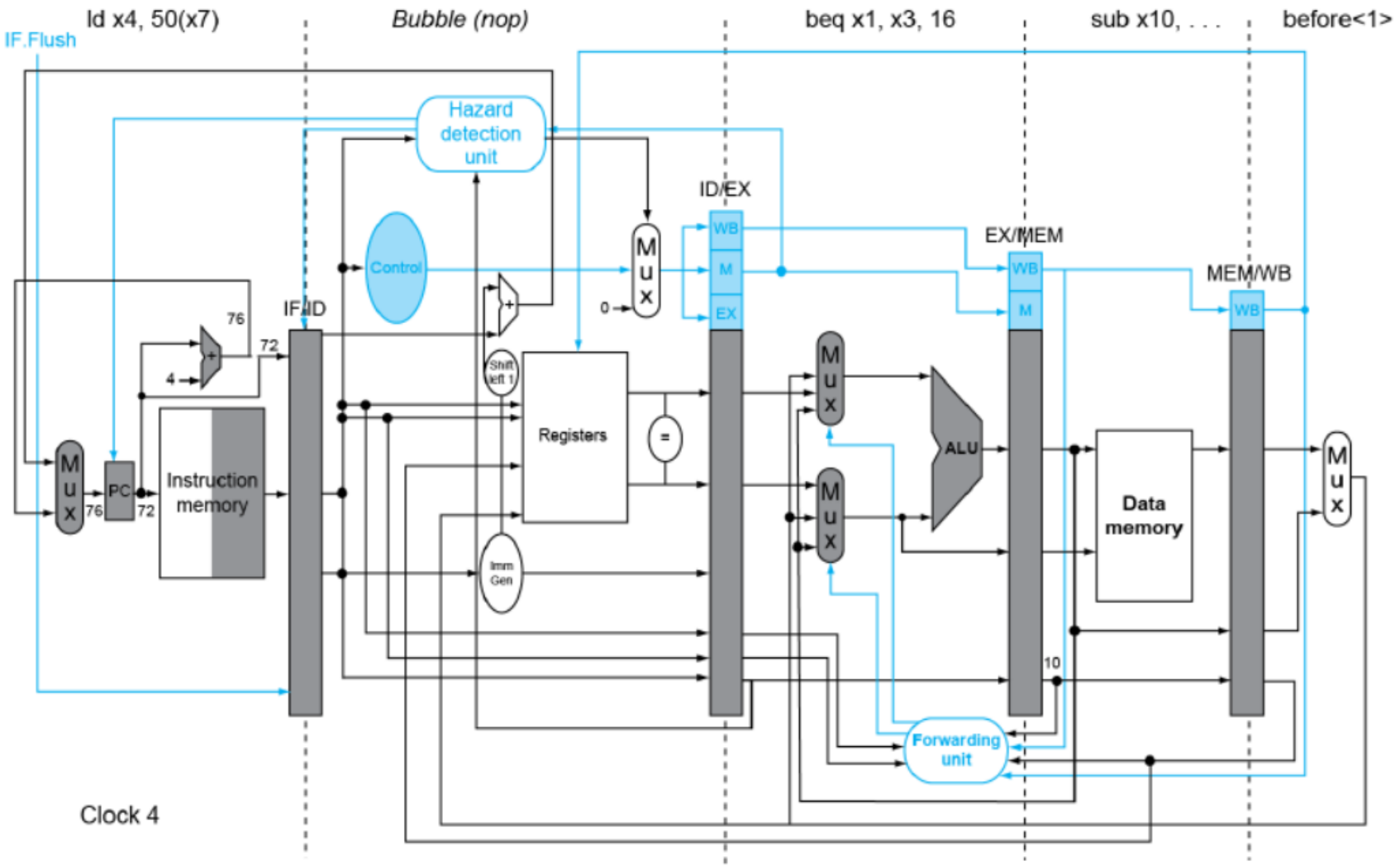
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16   // PC-relative
branch
                          // to 40+16*2=72
44:   and   x12, x2, x5
48:   orr   x13, x2, x6
52:   add   x14, x4, x2
56:   sub   x15, x6, x7
      ...
72:   ld    x4, 50(x7)
```

Revised Branch Decision Hardware + Example cont'd



Revised Branch Decision Hardware + Example cont'd



INNOVATION • QUALITY • RELIABILITY

Branch Hazards: More Details

```
WB      add3  x1, .....
MEM     add2  x3, ....
EX      add1  x4, ....
ID      beq   x1, x2, .....
IF      (next_instruction_in_sequence)
```

- No issue, Reg file write before read solves this.

```
WB      add3  x3, .....
MEM     add2  x1, ....
EX      add1  x4, ....
ID      beq   x1, x2, .....
IF      (next_instruction_in_sequence)
```

- EX/MB forwarding: For cases like this, we must forward from EX/MB to the comparator inputs

Branch Hazards: More Details

- Branch EX/MEM forwarding decision logic

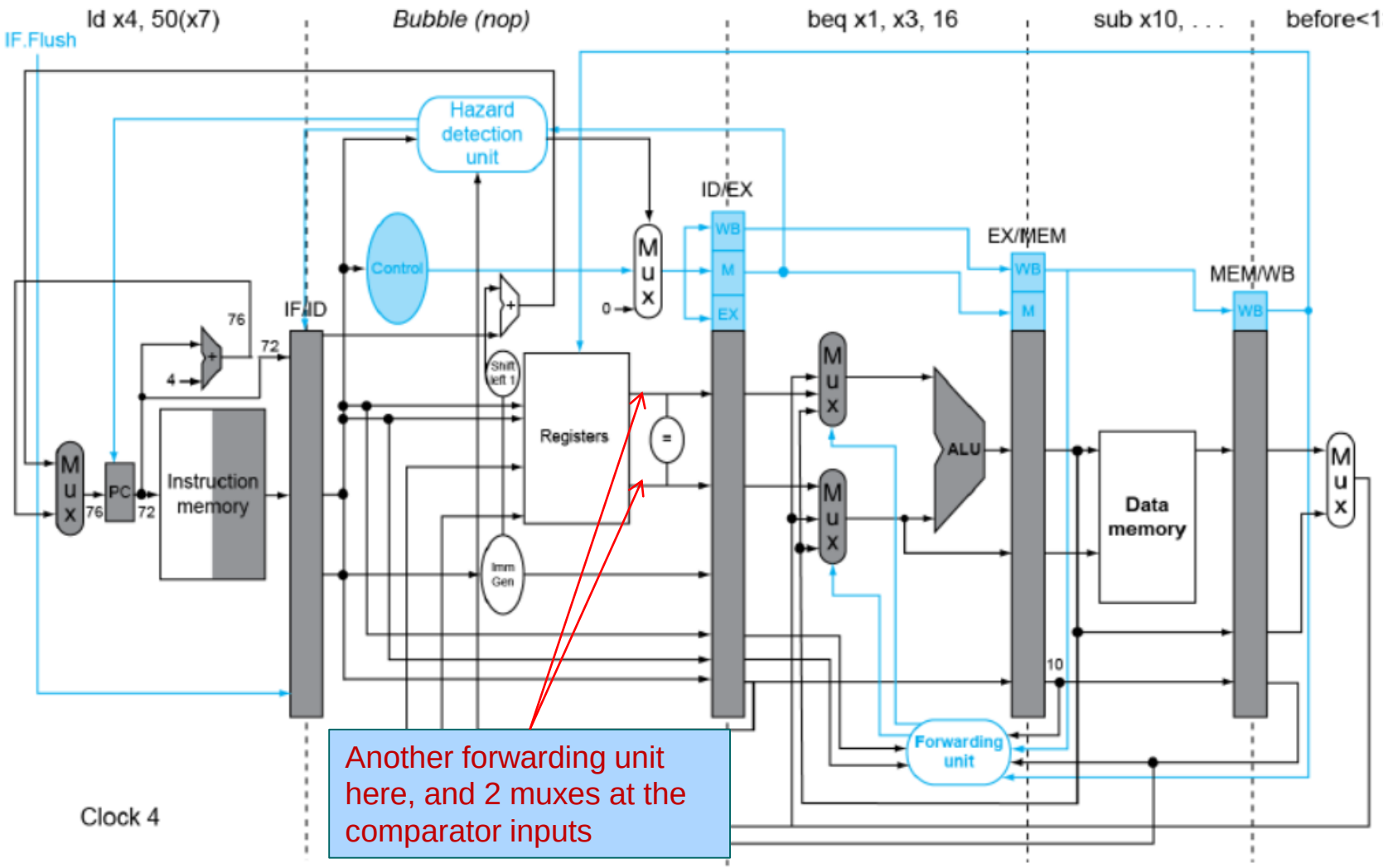
```
if (IDControl.Branch)
and (EX/MEM.RegisterRd ! = 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs)
        ForwardC = 1

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

Branch Hazards: More Details

WB	add3 x3, …
MEM	add3 x4, …
EX	add3 x1, …
ID	beq x1, x2, …
IF	…. …. ….

- Similar case with Load-Use hazards, stall is unavoidable

Revised Branch Decision Hardware



Branch Hazards

- Do nothing
  - Three cycle stall (flush)
- Resolve decision in EX stage
  - Stall reduced to two cycles
  - Increases critical time of EX stage, which is already high
- Resolve decision in ID stage
  - Stall reduced to one cycle
  - Extra comparator and selection logic
  - Needs another forwarding unit for ID stage
- Can we make branch decision in IF stage (No stall -> CPI=1)?

Branch Hazards

- **Compiler based solution:**

  Delay slots: The one stall cycle after a branch instruction is called a delay slot. Compiler can find some other instruction to put there, that will not have any effect on endangered registers.

  - But it may not be always feasible to find suitable instructions to place in delay slots, especially in longer pipelines...

Branch Prediction

- *Motivation:*
  - Branch penalties limit performance of deeply pipelined processors
  - Modern branch predictors have high accuracy
  - (>95%) and can reduce branch penalties significantly

- *Required hardware support:*
  - *Prediction structures:*
    - Branch history tables, branch target buffers, etc.
  - *Mispredict recovery mechanisms:*
    - *Keep result computation separate from commit*
    - Kill instructions following branch in pipeline
    - Restore state to that following branch

Static Branch Prediction

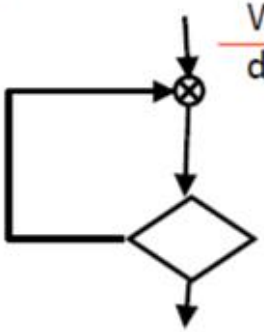
- Resolve branch hazards by assuming an outcome, then proceeding without waiting for the actual branch outcome

Overall probability a branch is taken is ~60-70% but:

What C++ statement does this look like

backward

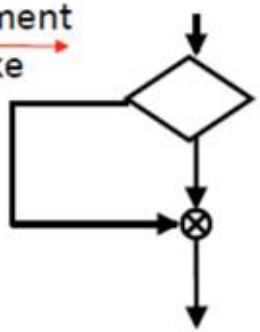
90%



What C++ statement does this look like

forward

50%



ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

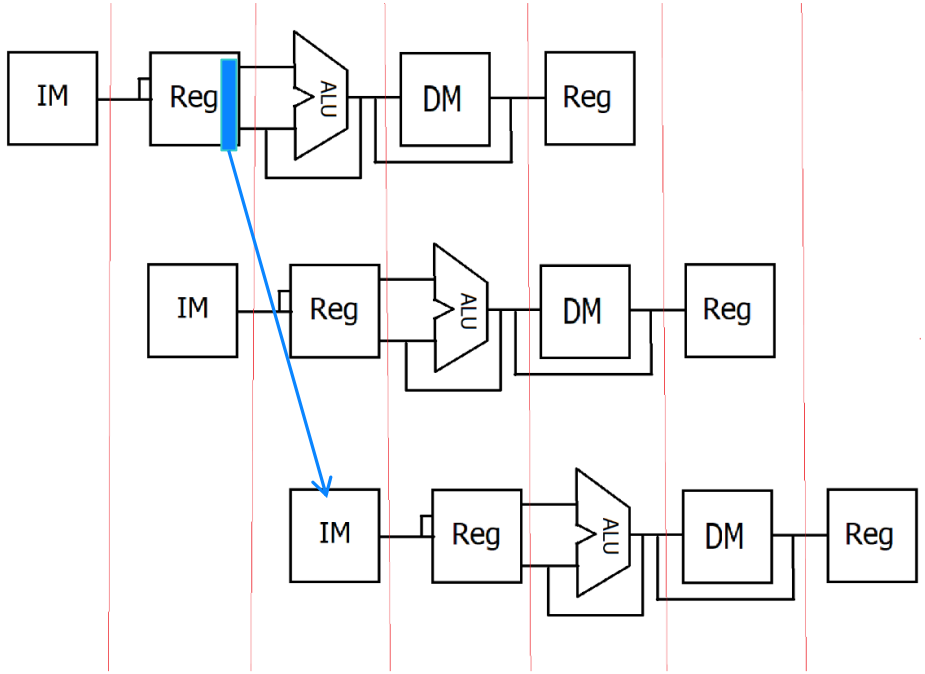
Static Branch Prediction

- Flushing with misprediction (branch not taken)

beq x1, x0,16

and x12, x2,x5 (flush)

or x13, x6,x2 (destination)



Flush by asserting IF.Flush signal, converting bits in IF/ID instruction field to zero and turning to “nop” instruction

Static Branch Prediction

- "Predict not taken" works well for top controlled loops
- "Branch taken" prediction
  - Always incurs one cycle stall (assuming branch decision is done in ID stage)
  - Is there a way to store the adress of the branch beforehand?
- For longer pipelines, branch penalty greatly increases, so a better way is required…

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

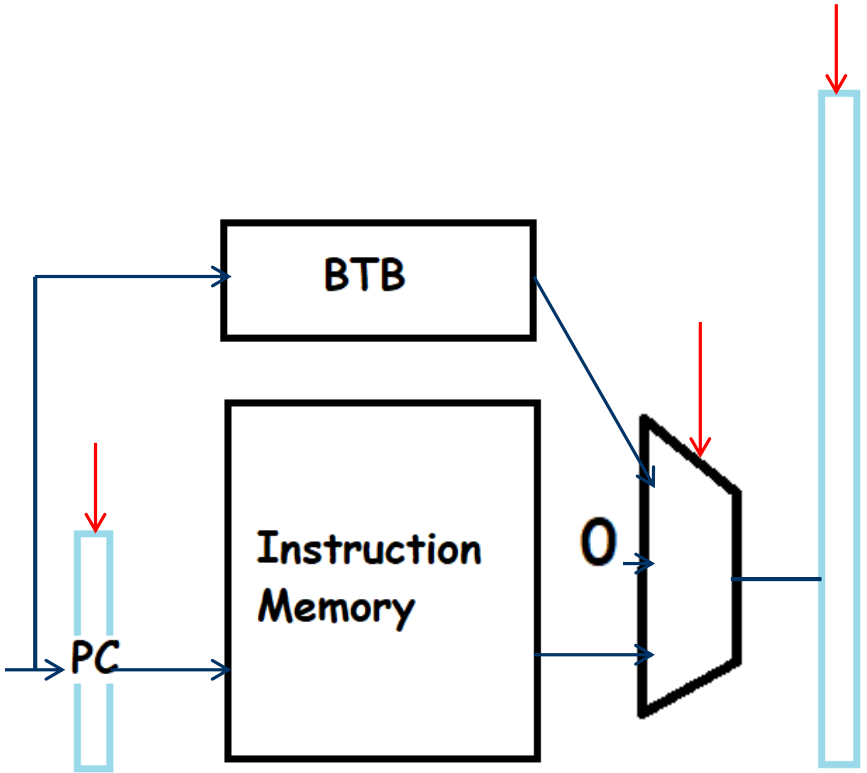
Branch History Table

- A branch prediction buffer (or branch history table - BHT) in the IF stage indexed by the PC, conatins bits passed to the ID stage through IF/ID pipeline register that tells whether the branch was taken last time it was executed.
- If the prediction is wrong, flush and return the old instruction (handling this did not change, but the prediction correctness rate greatly increases)
- A 4096 bit BHT varies from %1 to %18 percent mispredction, tried on various benchmark programs

Branch Target Buffer

BHT tells us if we take the branch or not, but it does not tell us where to go!

A branch target buffer (BHT) in IF stage catches the branch target addresses. Using the prediction bits from IF/ID stage, we choose whether or not take the jump destination or the sequential instruction.



Dynamic Branch Prediction

- 1-bit prediction scheme
  - Low-portion address as address for a one-bit flag for Taken or NotTaken historically
  - Simple
- 2-bit prediction
  - Miss twice to change

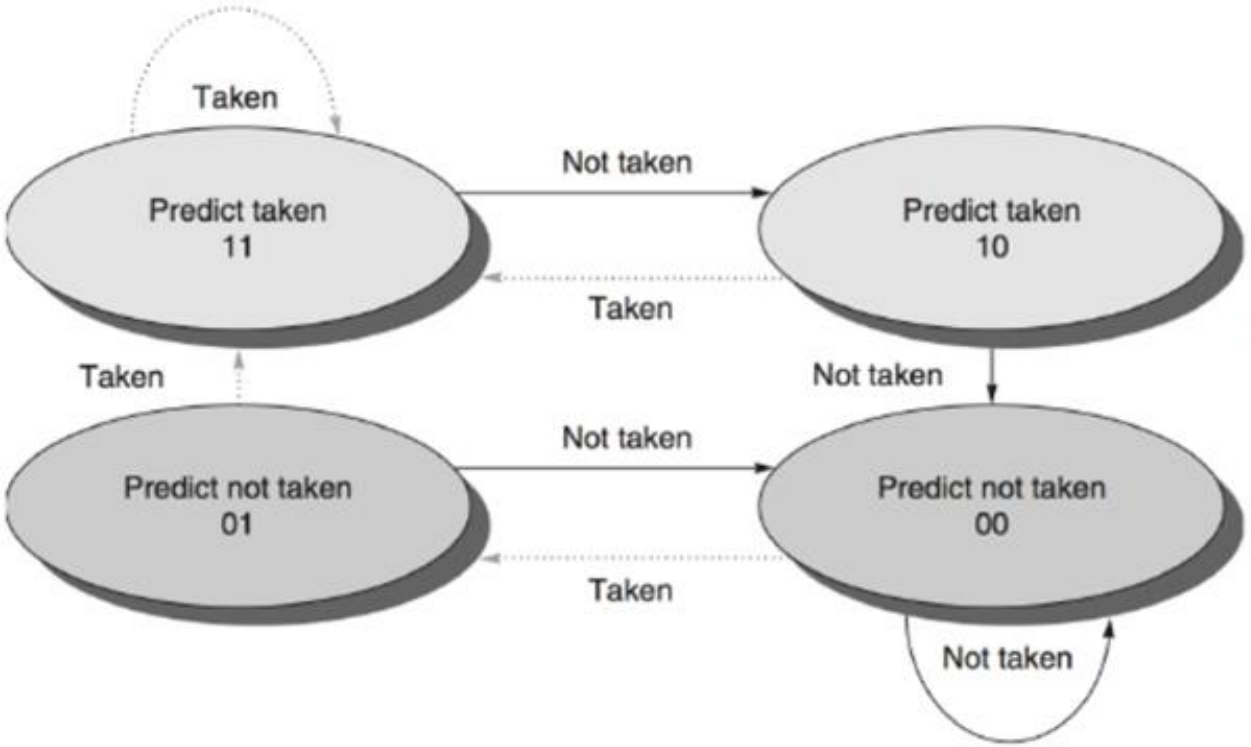
1-bit Predictor

› Inner loop branches mispredicted twice!

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-bit Predictor

- It takes two wrong predictions to change decision



References and Further Reading

- http://passlab.github.io/CSE564/notes/lecture09_RISCV_Impl_pipeline.pdf
- https://inst.eecs.berkeley.edu/~cs61c/fa17/lec/13/L13%20Pipelining%20(1up).pdf