CS152 – Computer Architecture and Engineering

Fall 2004

Lecture 11: Pipeline Control and Hazards

John Lazzaro
(www.cs.berkeley.edu/~lazzaro)

Dave Patterson
(www.cs.berkeley.edu/~patterson)

[Adapted from Mary Jane Irwin's slides www.cse.psu.edu/~cg431]

Fall 2004 © UC Regents

Recap last lecture

- All recent processors use pipelining
- Pipelining doesn't help latency of single task, it helps throughput or bandwidth of entire workload
  - Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced lengths of pipe stages reduces speedup
  - Time to "fill" pipeline and time to "drain" it reduces speedup
- Must detect and resolve hazards
  - Stalling negatively affects throughput
- Today: pipeline control, including hazards

Fall 2004 © UC Regents

Control Settings

	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

Pipeline registers written every clock cycle (like PC)

Control same for all instructions in IF and ID stages: fetch instruction, increment PC

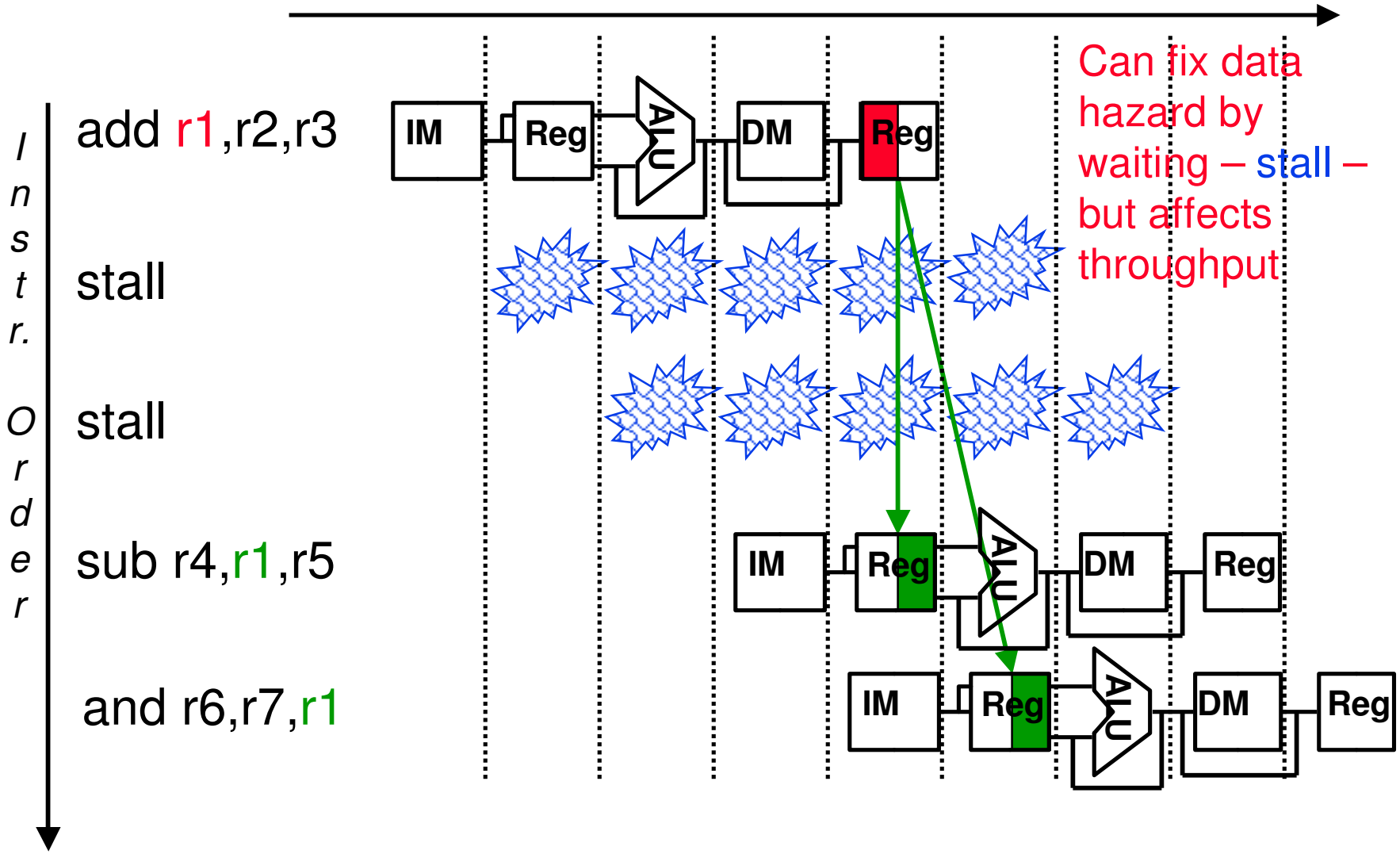
Fall 2004 © UC Regents

MIPS Pipeline Data and Control Paths

PCSrc
1
0
ID/EX
EX/MEM
Control
IF/ID
Add
4
Add
Branch
MEM/WB
RegWrite
Shift left 2
Instruction Memory
Read Address
PC
Read Addr 1
Register File
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
ALUSrc
ALU
Data Memory
Address
Read Data
Write Data
MemtoReg
ALU cntrl
MemWrite MemRead
Sign Extend
16
32
ALUOp
RegDst

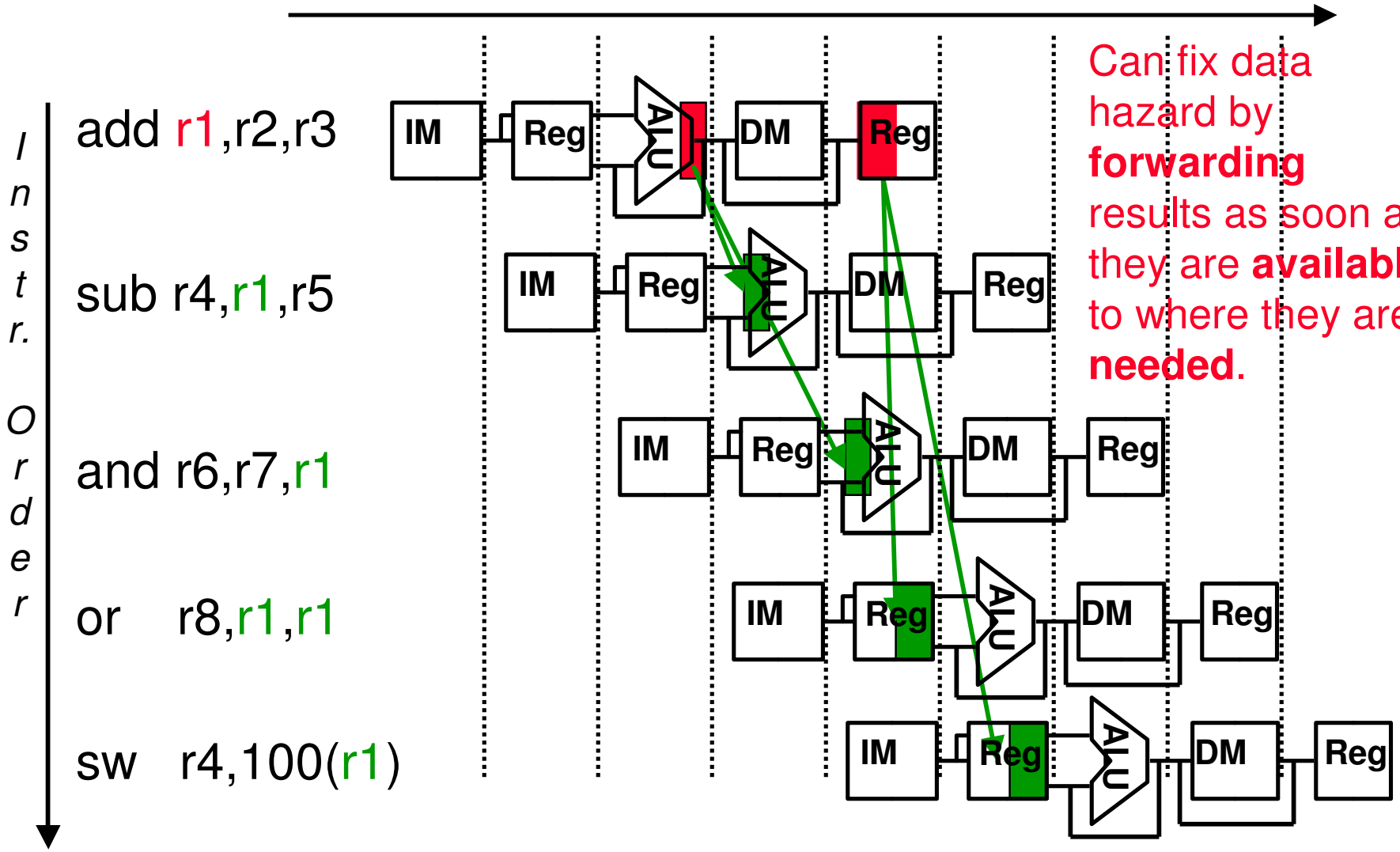
Fall 2004 © UC Regents

Review: One Way to "Fix" a Data Hazard



Fall 2004 © UC Regents

Review: Another Way to "Fix" a Data Hazard



Can fix data hazard by **forwarding** results as soon as they are **available** to where they are **needed**.

Fall 2004 © UC Regents

Data Forwarding (aka Bypassing)

- Any data dependence line that goes backwards in time
  - EX stage generating R-type ALU results or effective address calculation
  - MEM stage generating lw results
- Forward by taking the inputs to the ALU from any pipeline register rather than just ID/EX by
  - adding multiplexors to the inputs of the ALU so can pass Rd back to either (or both) of the EX's stage Rs and Rt ALU inputs
    00: normal input (ID/EX pipeline registers)
    10: forward from previous instr (EX/MEM pipeline registers)
    01: forward from instr 2 back (MEM/WB pipeline registers)
  - adding the proper control hardware
- With forwarding can run at full speed even in the presence of data dependencies

Fall 2004 © UC Regents

Data Forwarding Control Conditions (1/4)

1. EX/MEM hazard:

```
if (EX/MEM.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 10
```

- "RegisterRd" is number of register to be written (RD or RT)
- "RegisterRs" is number of RS register
- "RegisterRt" is number of RT register
- "ForwardA, ForwardB" controls forwarding muxes

```
if (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU.

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

2. MEM/WB hazard:

```
if (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

What's wrong with this hazard control?
(When might it forward when it shouldn't?)
(Which sequences would reveal this bug?)

Fall 2004 © UC Regents

Data Forwarding Control Conditions (2/4)

1. EX/MEM hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU **provided it writes**.

2. MEM/WB hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU **provided it writes**.

What's wrong with this hazard control?
(When might it forward when it shouldn't?)
(Which sequences would reveal this bug?)

Fall 2004 © UC Regents

Data Forwarding Control Conditions (3/4)

1. EX/MEM 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
```

Forwards the result from the previous instr. to either input of the ALU provided it writes **and != R0**.

2. MEM/WB 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
```

Forwards the result from the second previous instr. to either input of the ALU provided it writes **and != R0**.

What's wrong with this hazard control?

Fall 2004 © UC Regents

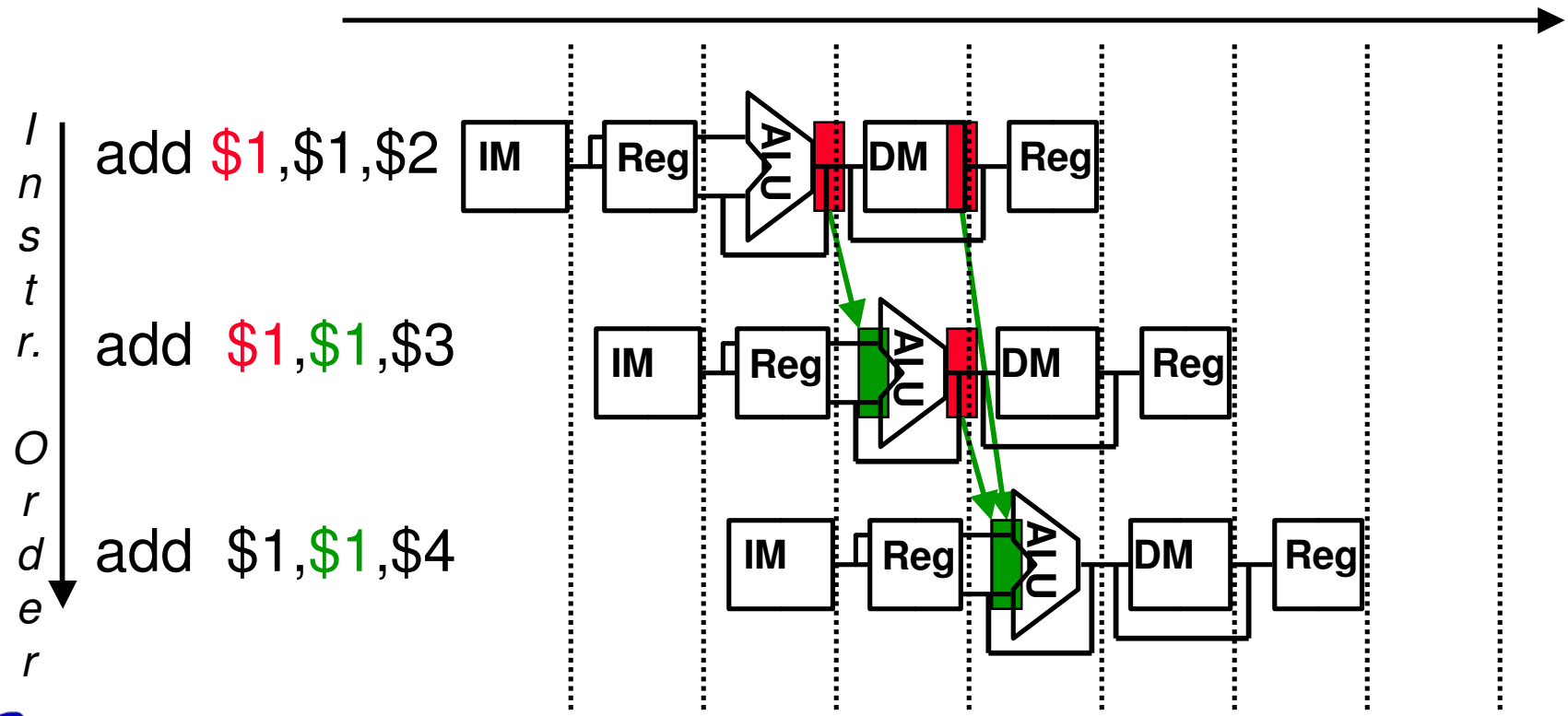
Administrivia

- Start Lab 3; Plan due Thursday evening; meet with TA Friday
- Reading Chapter 6, sections 6.5 to 6.9 for this week
- Midterm Tue Oct 12 5:30 - 8:30 in **101 Morgan** (you asked for it)
  - Northwest corner of campus, near Arch and Hearst
  - Midterm review Sunday Oct 10, 7 PM, 306 Soda
  - Bring 1 page, handwritten notes, both sides
  - Nothing electronic: no calculators, cell phones, pagers, …
  - Meet at LaVal's Northside afterwards for Pizza

Fall 2004 © UC Regents

Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded? More recent result!



Fall 2004 © UC Regents

Corrected Data Forwarding Control Conditions

2. MEM/WB hazard:

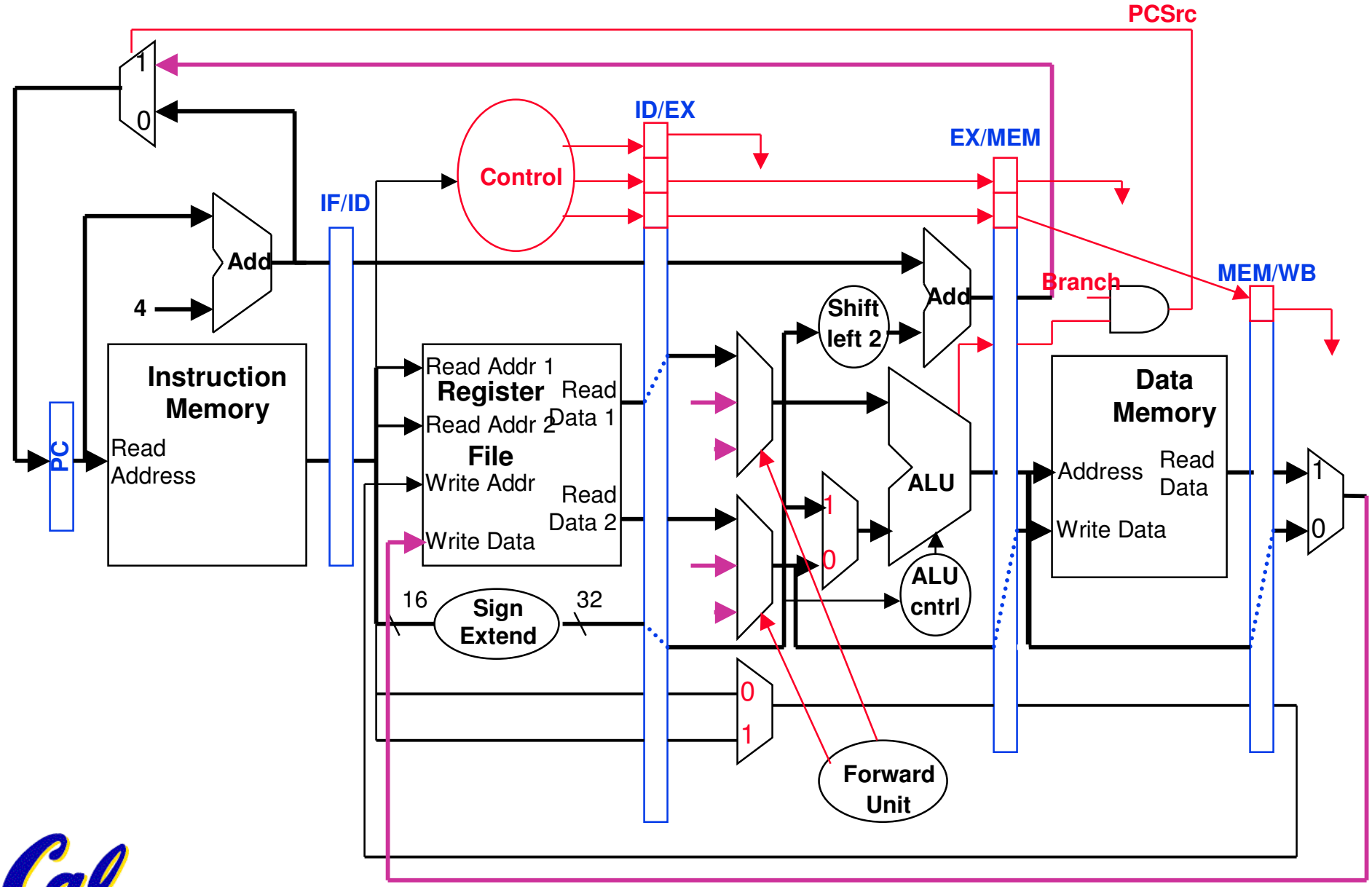
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs
   || ~ EX/MEM.RegWrite))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRt)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt
   || ~ EX/MEM.RegWrite)))
      ForwardB = 01
```

Forward if this instruction writes AND its not writing R0 AND this dest reg == source AND in between instr either dest. reg. doesn't match OR it doesn't write reg.

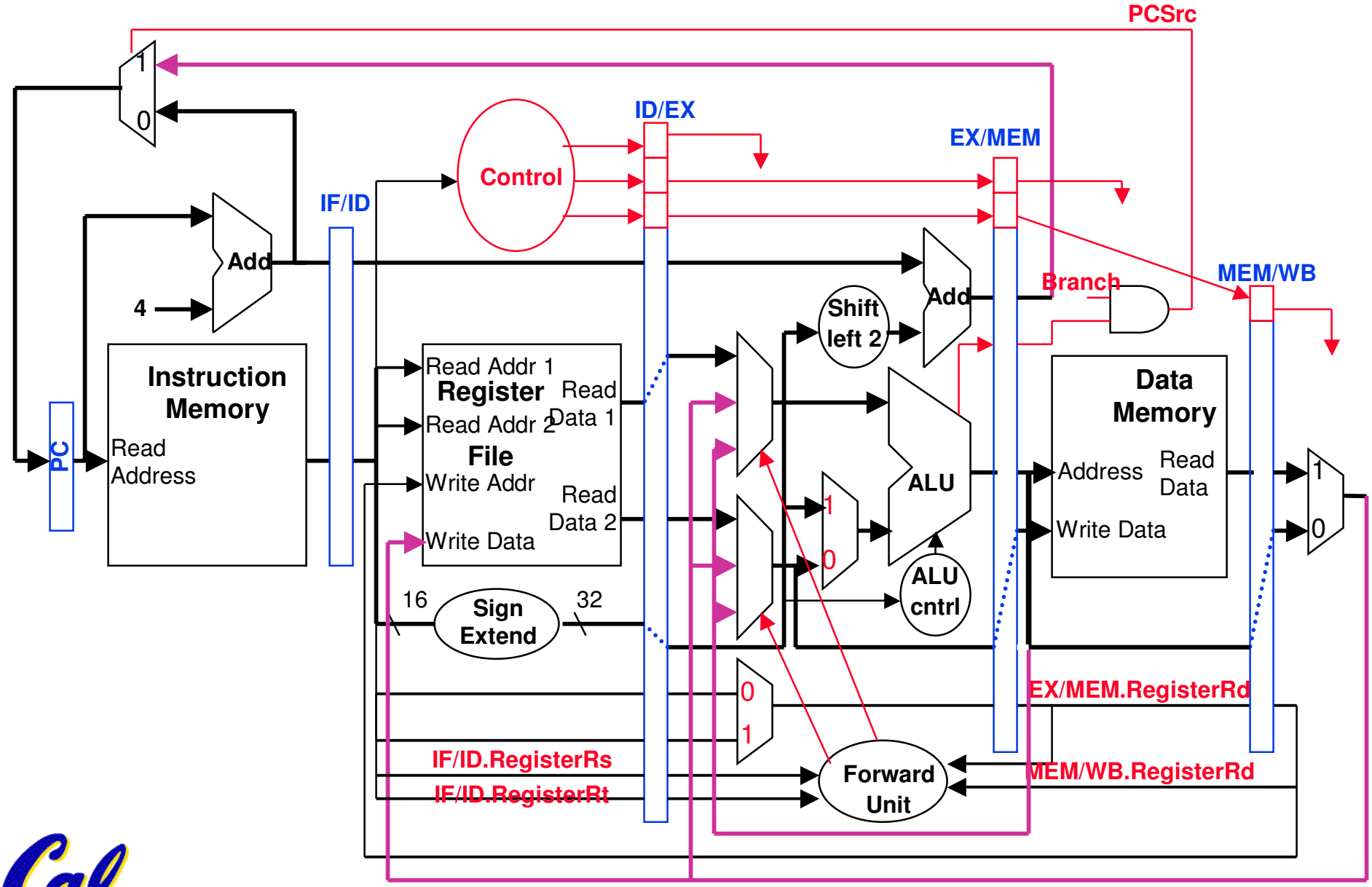
Fall 2004 © UC Regents

Datapath with Forwarding Hardware



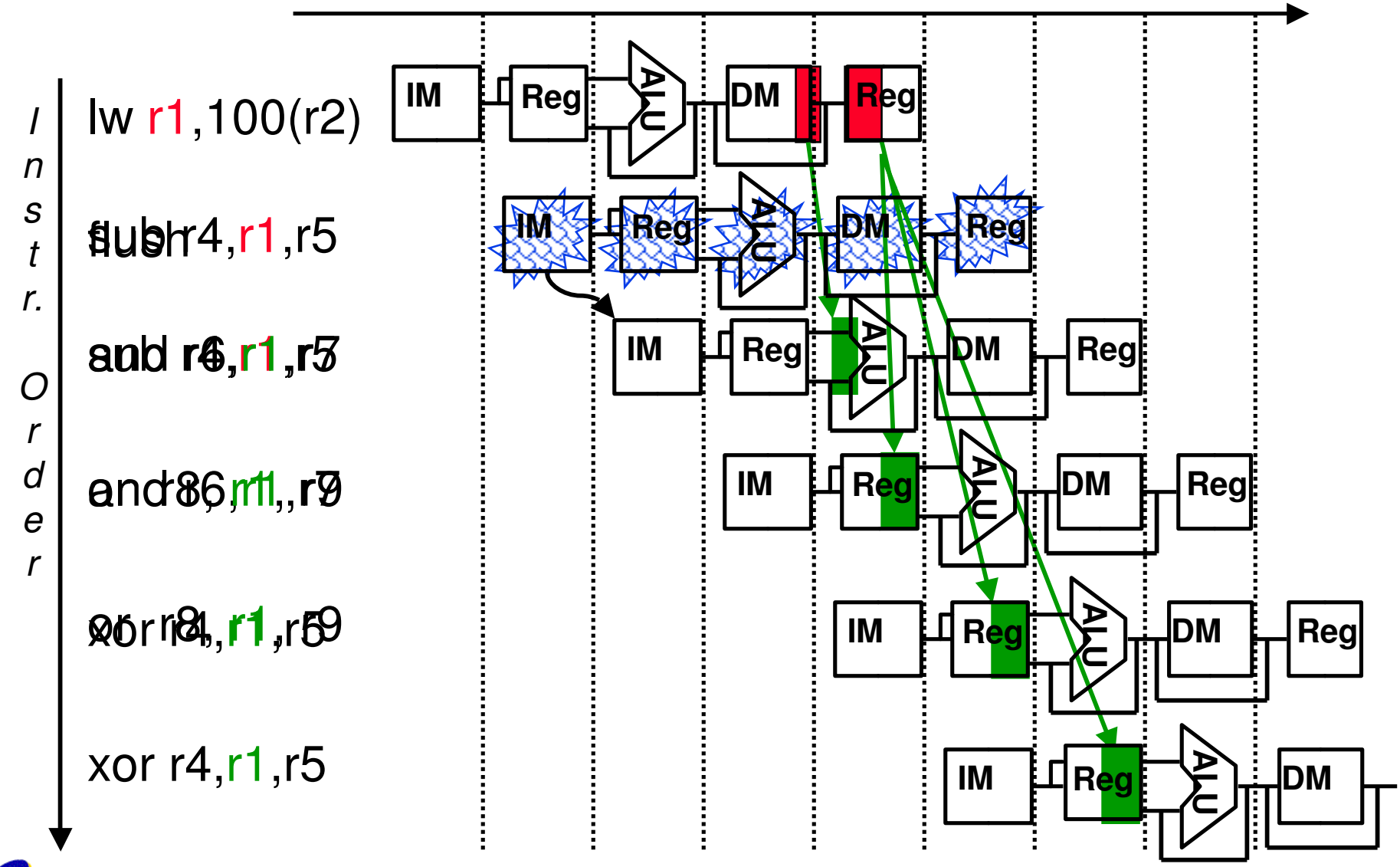
Fall 2004 © UC Regents

Datapath with Forwarding Hardware



Fall 2004 © UC Regents

Forwarding with Load-use Data Hazards



Fall 2004 © UC Regents

Load-use Hazard Detection Unit

- Need a hazard detection unit in the ID stage that inserts a stall between the load and its use

2. ID Hazard Detection

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

- The first line tests to see if the instruction is a load; the next two lines check to see if the destination register of the load in the EX stage matches either source registers of the instruction in the ID stage
- After this 1-cycle stall, the forwarding logic can handle the remaining data hazards

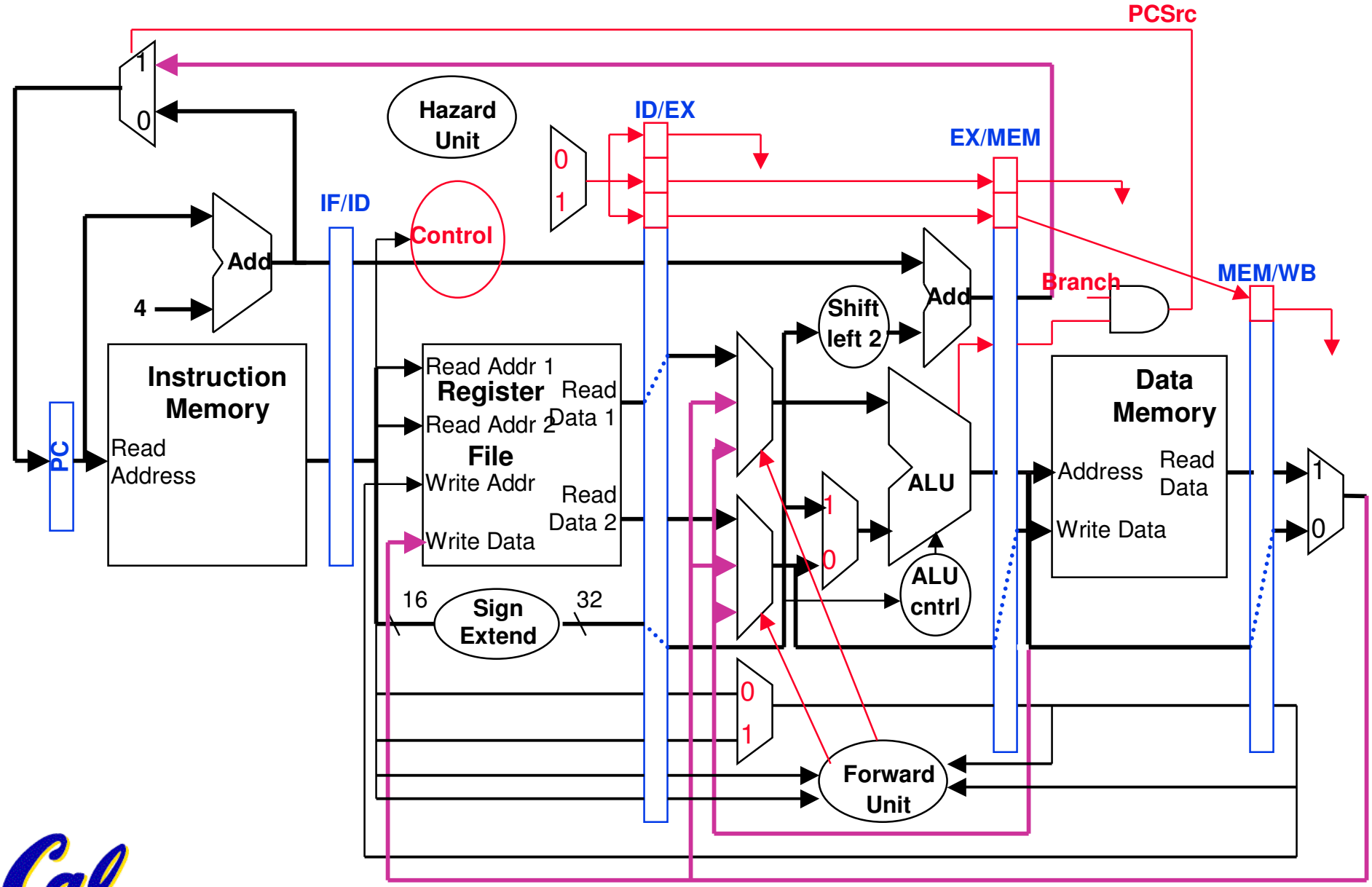
Fall 2004 © UC Regents

Stall Hardware

- In addition to the hazard detection unit, we have to implement the stall
- Prevent the IF and ID stage instructions from making progress down the pipeline, done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection unit controls the writing of the PC and IF/ID registers
- The instructions in the back half of the pipeline starting with the EX stage must be flushed (execute `noop`)
  - Must deassert the control signals (setting them to 0) in the EX, MEM, and WB control fields of the ID/EX pipeline register.
  - Hazard detection unit controls the multiplexor that chooses between the real control values and 0's.
  - Assume that 0's are benign values in datapath: nothing changes

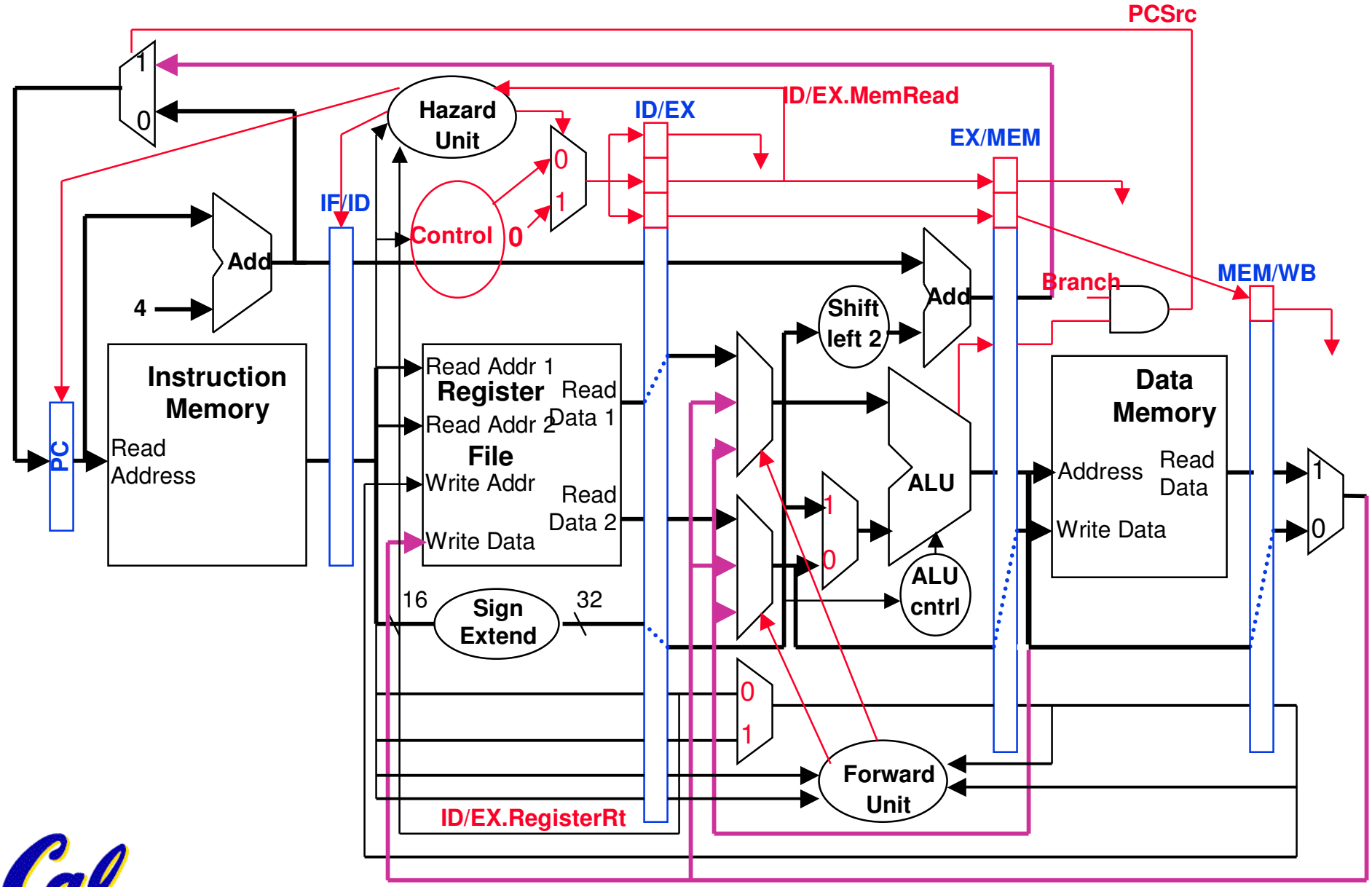
Fall 2004 © UC Regents

Adding the Hazard Hardware



Fall 2004 © UC Regents

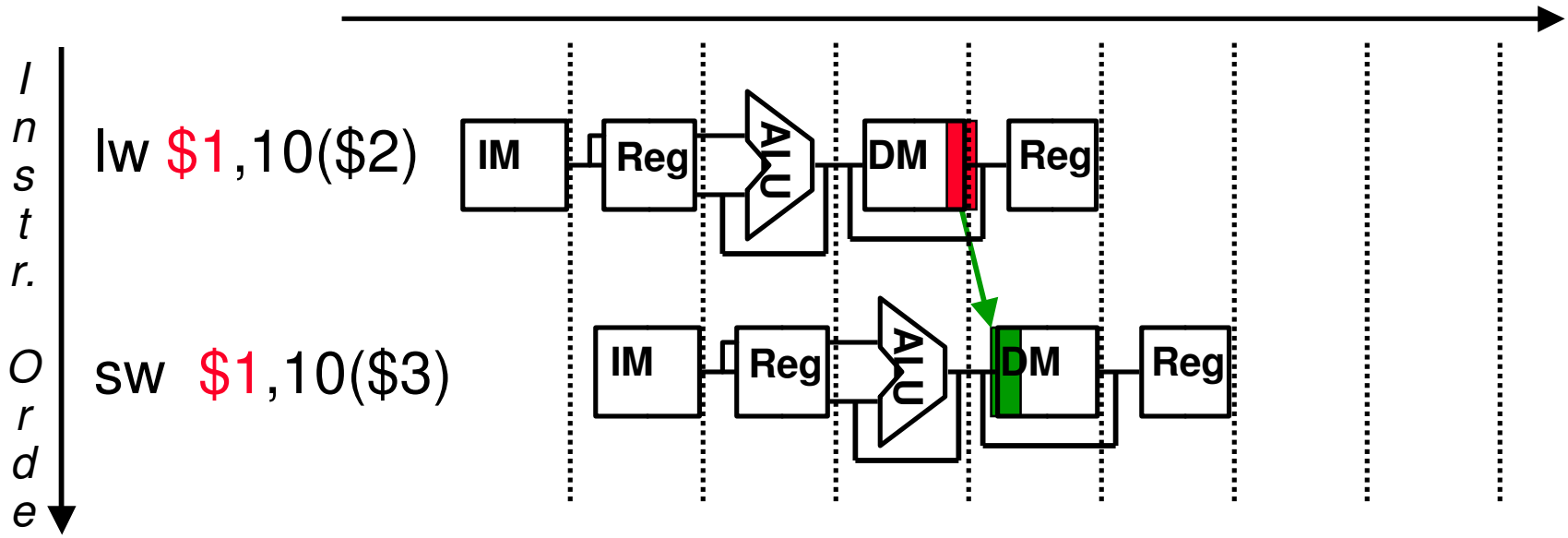
Adding the Hazard Hardware



Fall 2004 © UC Regents

Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit to the memory access stage
  - Should avoid stalling on such a load



Fall 2004 © UC Regents

Where else need to forward?

- What about this code?

```
addu $r5, …
sw $r5, …
```

Or

```
addu $r5, …
subu $r6, …
sw $r5, …
```

- Need forwarding path
- Elaboration on page 412 shows how if have separate mux for immediate vs. forwarded value, can use EX stage forwarding control to pass proper result to EX/MEM register for sw to store

Fall 2004 © UC Regents

Control Hazards

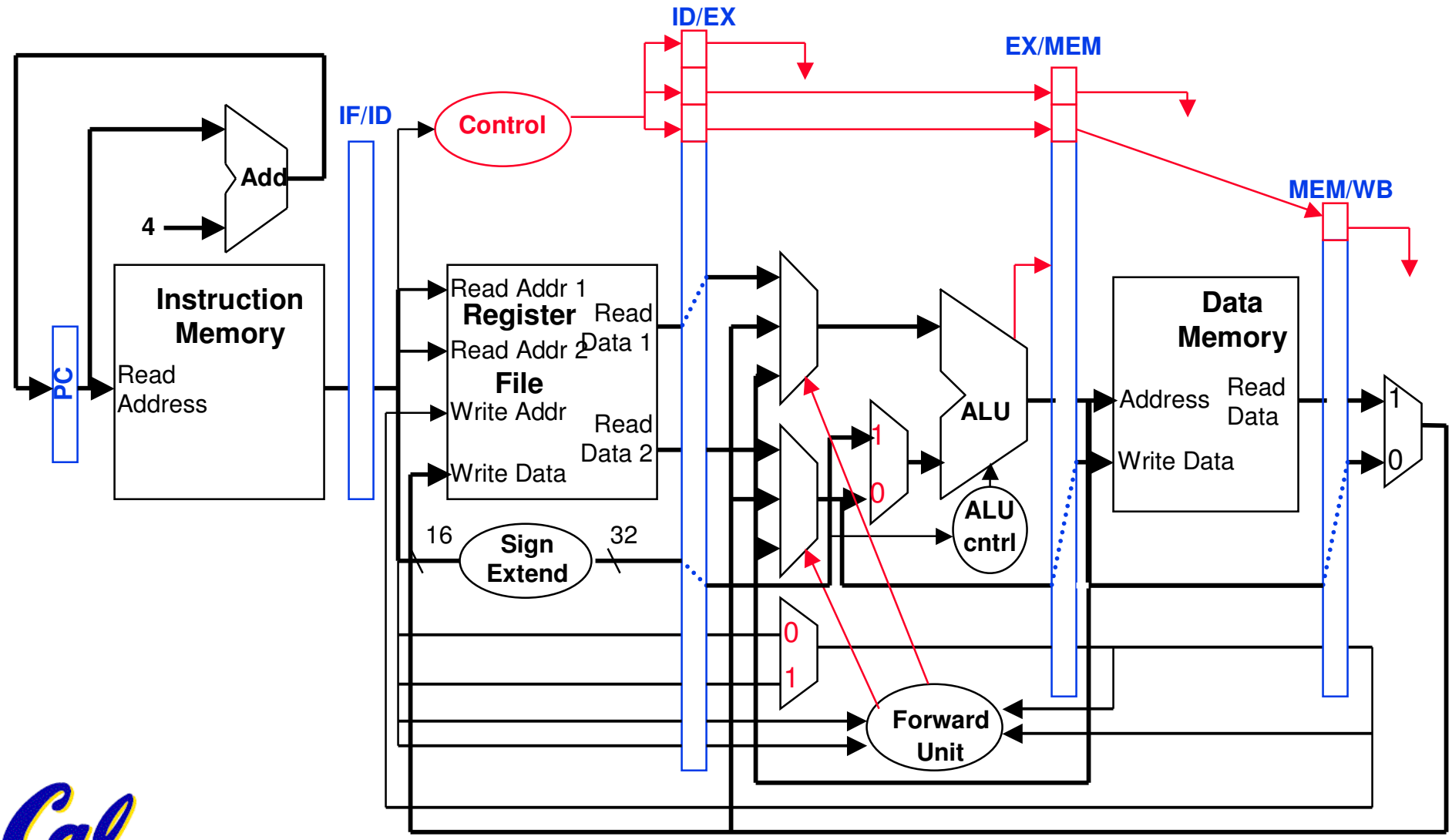
- When the flow of instruction addresses is not what the pipeline expects; incurred by change of flow instructions
  - Conditional branches (`beq`, `bne`)
  - Unconditional branches (`j`)

- Possible solutions
  - Stall
  - Move decision point earlier in the pipeline
  - Predict
  - Delay decision (requires compiler support)

- Control hazards occur less frequently than data hazards; there is nothing as effective against control hazards as forwarding is for data hazards

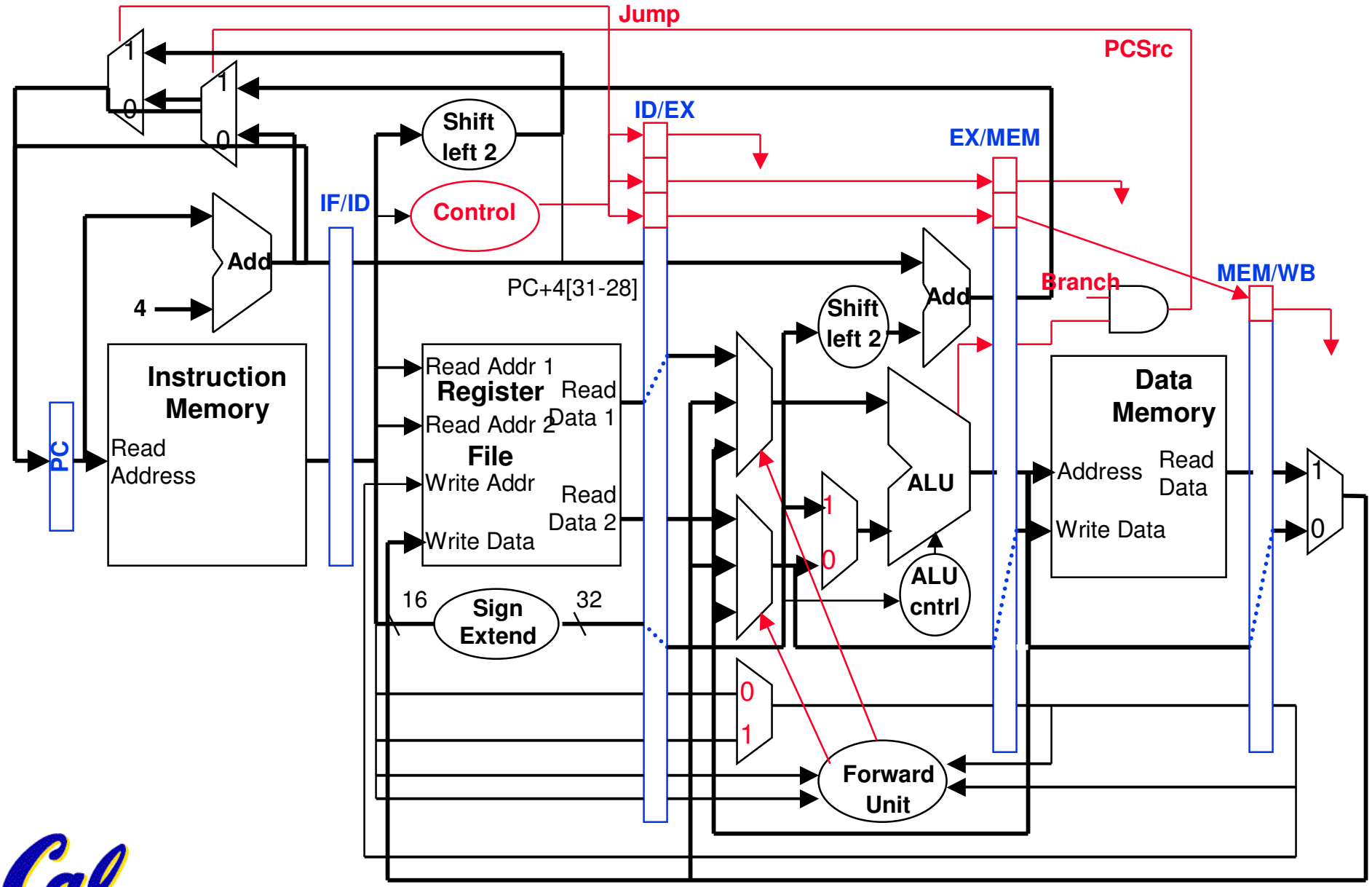
Fall 2004 © UC Regents

Datapath Branch and Jump Hardware



Fall 2004 © UC Regents

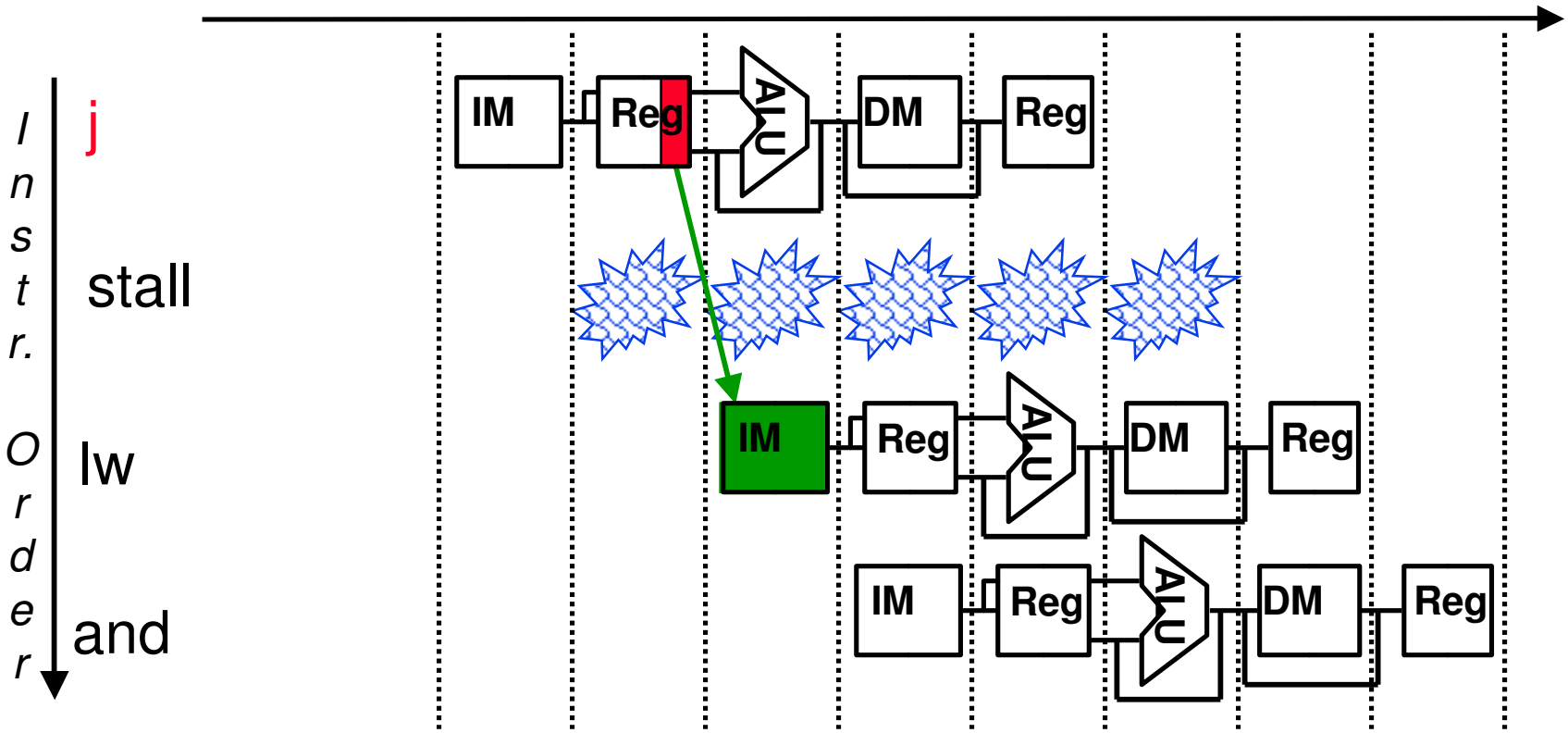
Datapath Branch and Jump Hardware



Fall 2004 © UC Regents

Jumps Incur One Stall

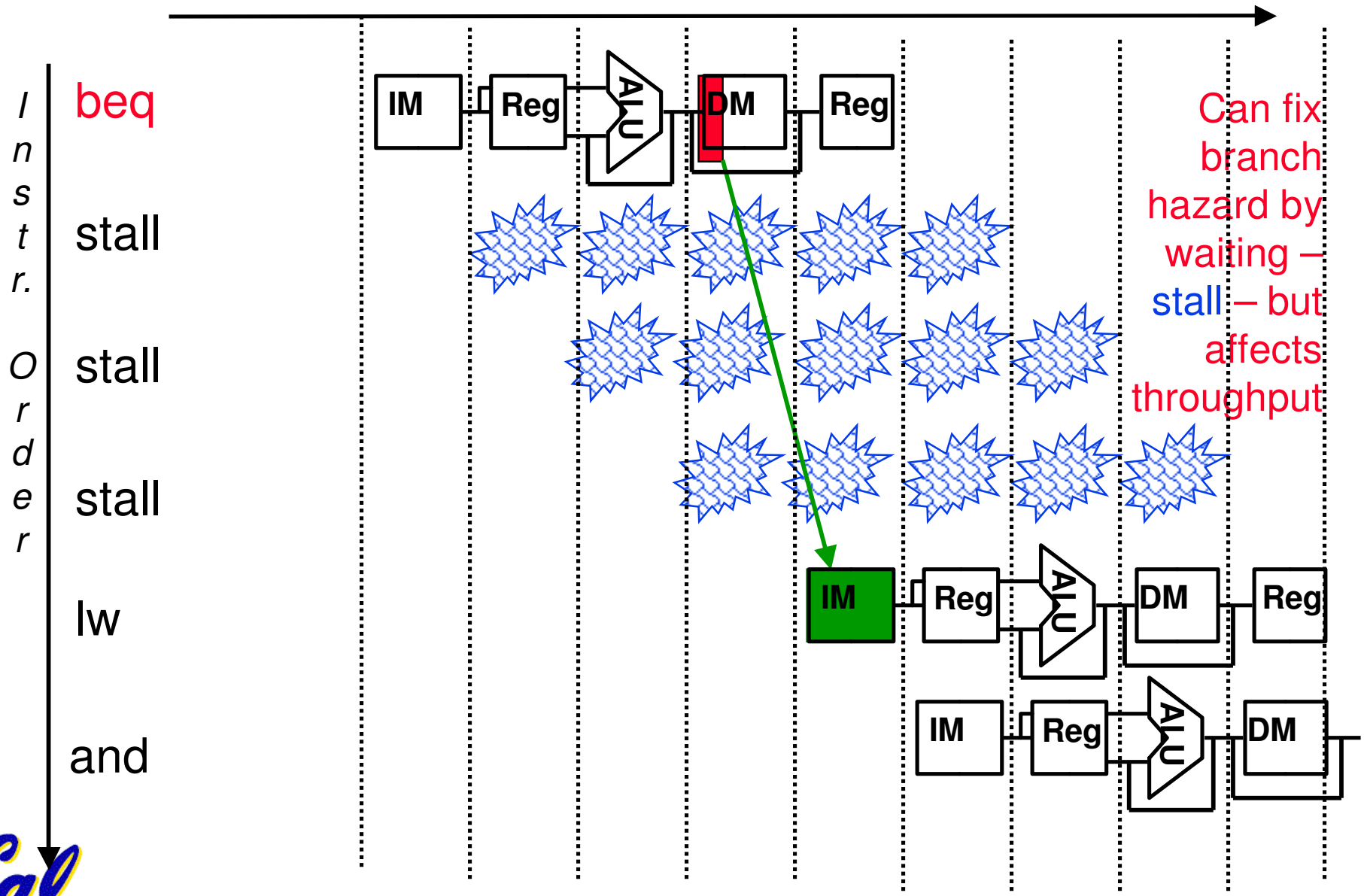
- Jumps not decoded until ID, so one stall is needed



- Fortunately, jumps are very infrequent – only 2% of the SPECint instruction mix

Fall 2004 © UC Regents

Review: Branches Incur Three Stalls



Fall 2004 © UC Regents

Moving Branch Decisions Earlier in Pipe

- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path
- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall cycles to one (like with jumps)
  - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  - Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a comparator, an and gate, and a 3x1 mux to the ID timing path
  - Need forwarding hardware in ID stage
- For longer pipelines, decision points are later in the pipeline, incurring more stalls, so we need a better solution

Fall 2004 © UC Regents

Early Branch Forwarding Issues

- Bypass of source operands from the EX/MEM

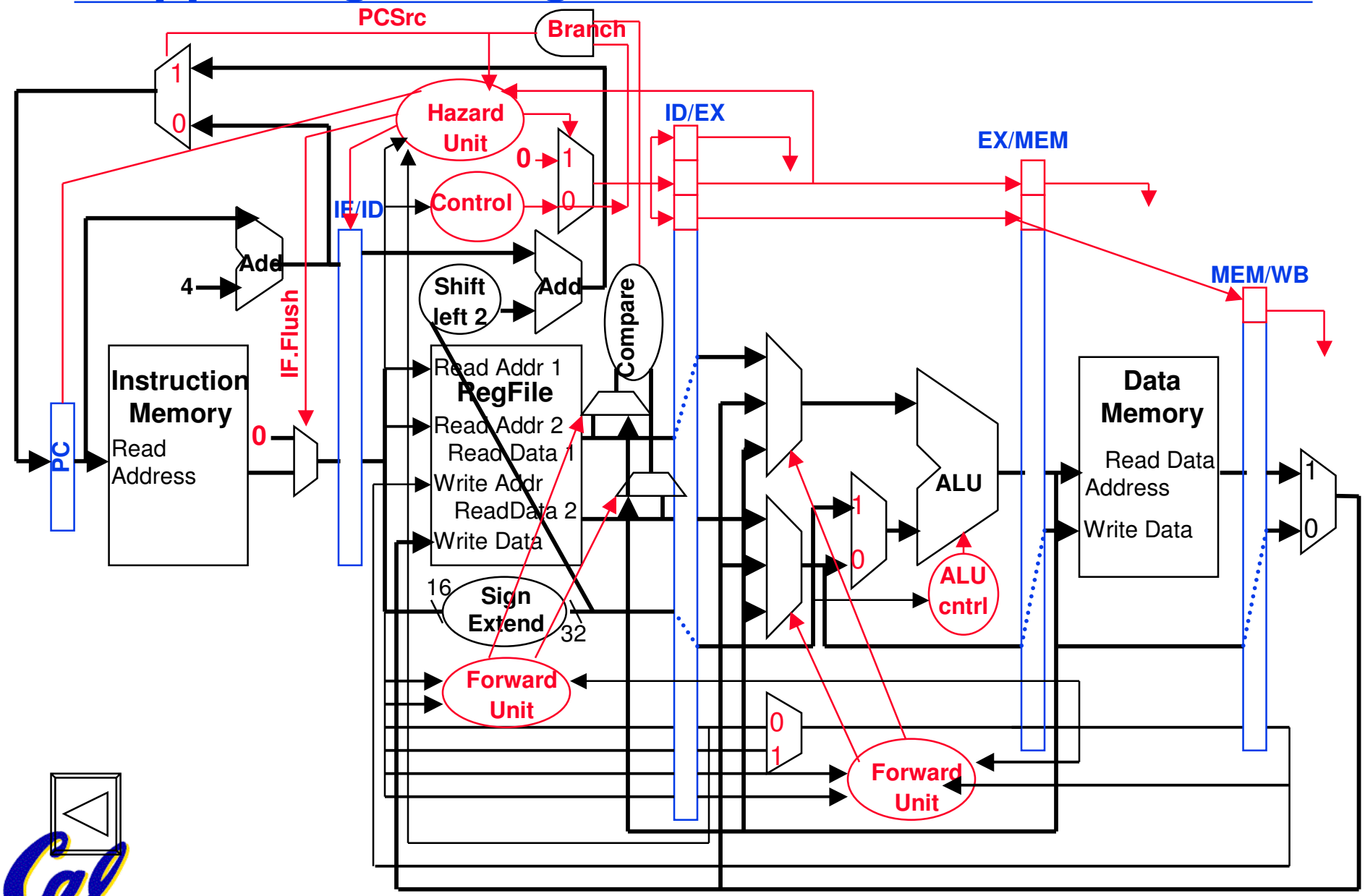
```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
      ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
      ForwardD = 1
```

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

- MEM/WB "forwarding" is taken care of by the normal RegFile write before read operation
- If the instruction immediately before the branch produces one of the branch compare source operands, then a stall will be required since the EX stage ALU operation is occurring at the same time as the ID stage branch compare operation

Fall 2004 © UC Regents

Supporting ID Stage Branches



Fall 2004 © UC Regents

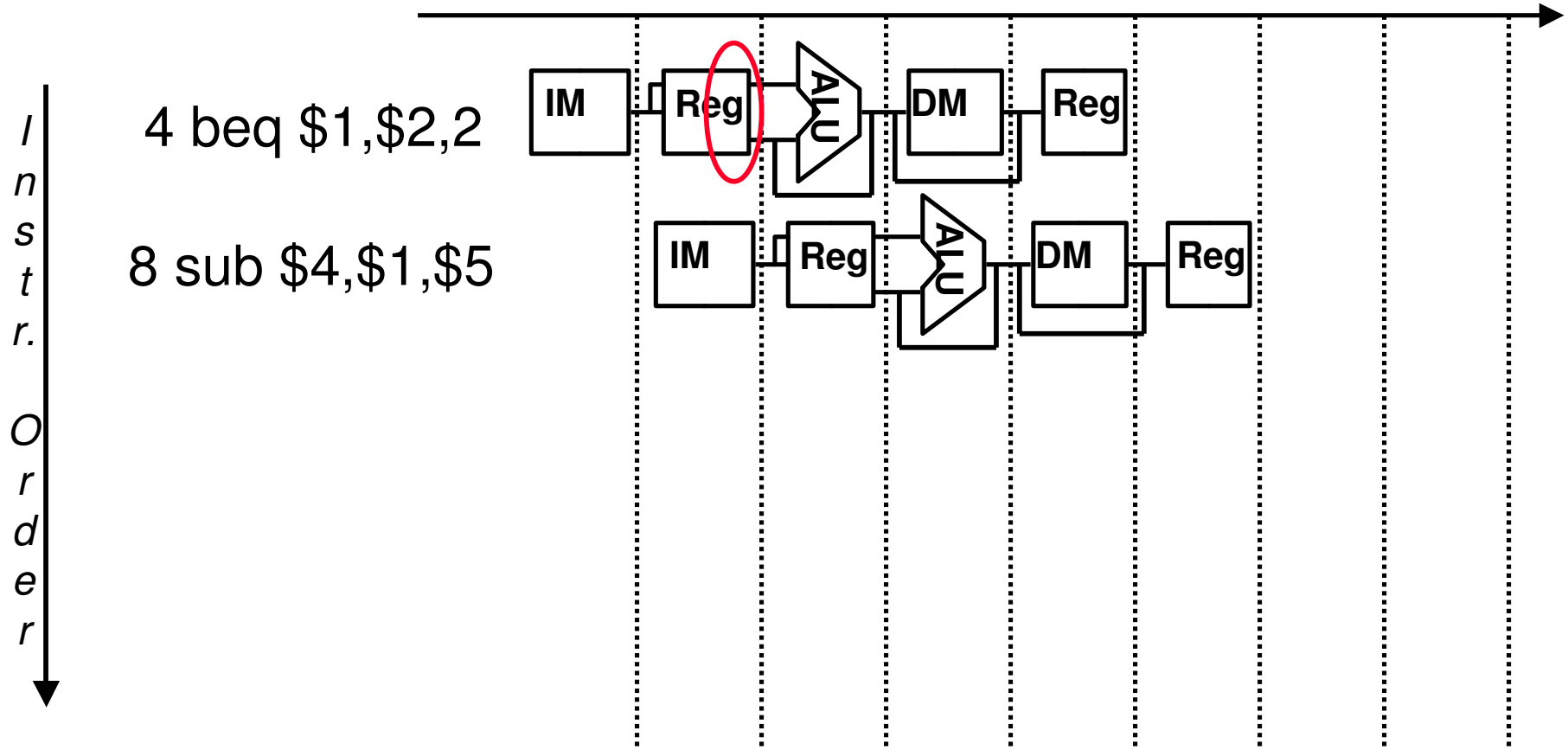
Branch Prediction

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

1. 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 in the pipeline after the branch
     - in IF, ID, and EX if branch logic in MEM – three stalls
     - in IF if branch logic in ID – one stall
   - ensure that those flushed instructions haven't changed machine state– automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite or RegWrite)
   - restart the pipeline at the branch destination

Fall 2004 © UC Regents

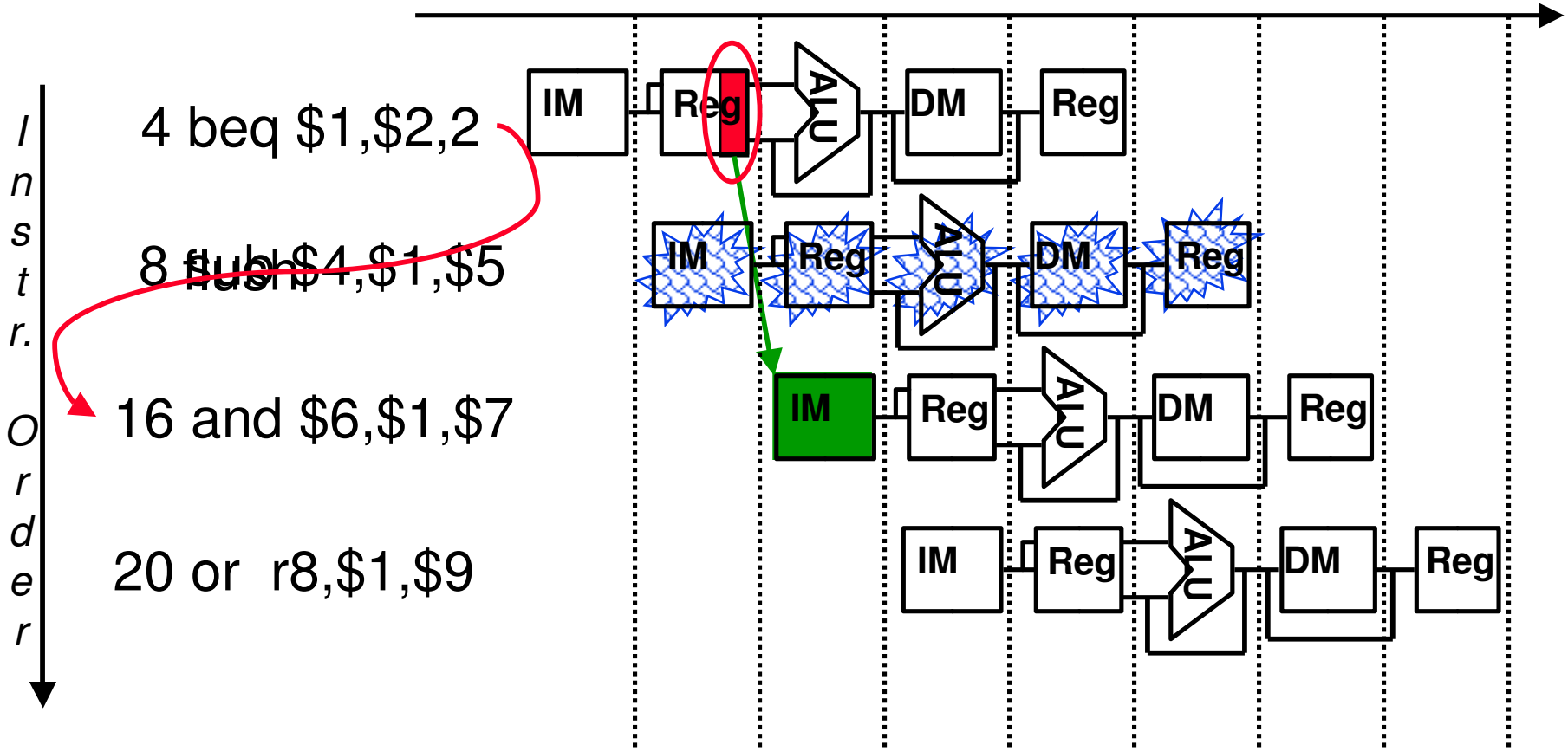
Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a `noop`)

Fall 2004 © UC Regents

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a `noop`)

Fall 2004 © UC Regents

Branch Prediction, con't

- Resolve branch hazards by statically assuming a given outcome and proceeding

2. Predict taken – always predict branches will be taken
   - Predict taken always incurs a stall (if branch destination hardware has been moved to the ID stage)

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance

- With more hardware, possible to try to predict branch behavior dynamically during program execution

3. Dynamic branch prediction – predict branches at run-time using run-time information

Fall 2004 © UC Regents

Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage, addressed by the lower bits of the PC, contains a bit that tells whether the branch was taken the last time it was execute
  - Bit may predict incorrectly (may be from a different branch with the same low order PC bits, or may be a wrong prediction for this branch) but the doesn't affect correctness, just performance
  - If the prediction is wrong, flush the incorrect instructions in pipeline, restart the pipeline with the right instructions, and invert the prediction bit
- The BHT predicts when a branch is taken, but does not tell where its taken to!
  - A branch target buffer (BTB) in the IF stage can cache the branch target address (or !even! the branch target instruction) so that a stall can be avoided

Fall 2004 © UC Regents

1-bit Prediction Accuracy

- 1-bit predictor in loop is incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code

1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
2. As long as branch is taken (looping), prediction is correct
3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

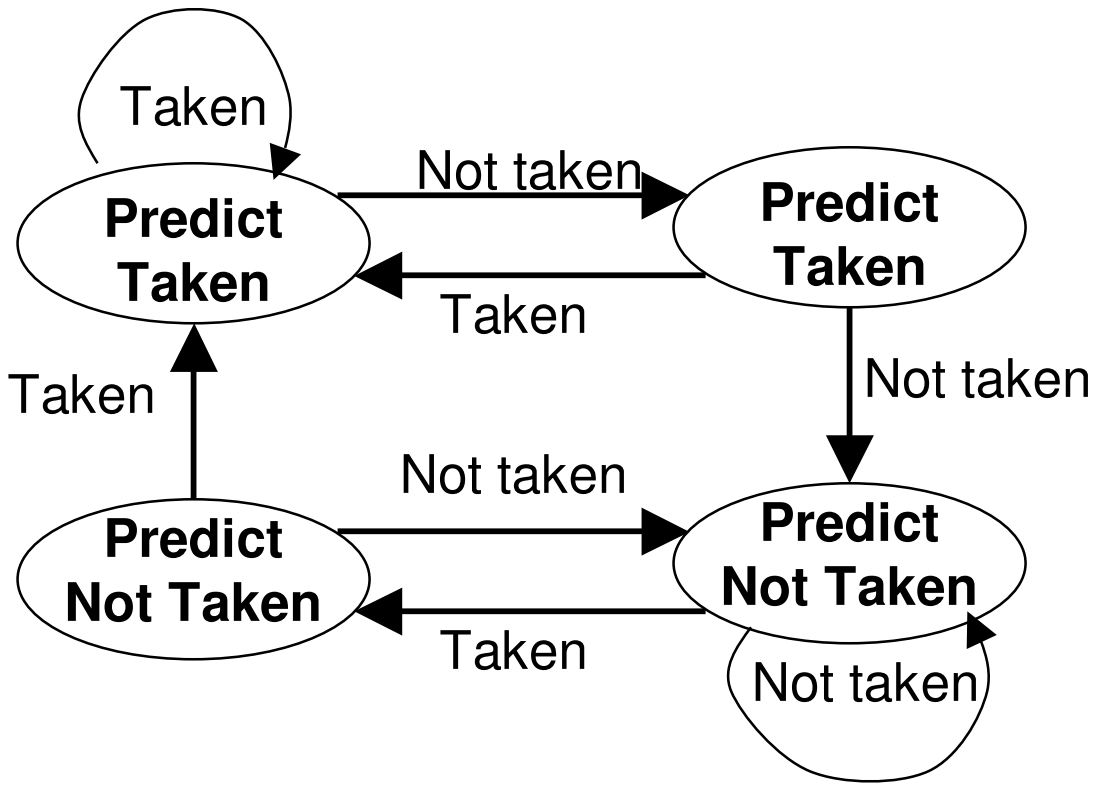
```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

Fall 2004 © UC Regents

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

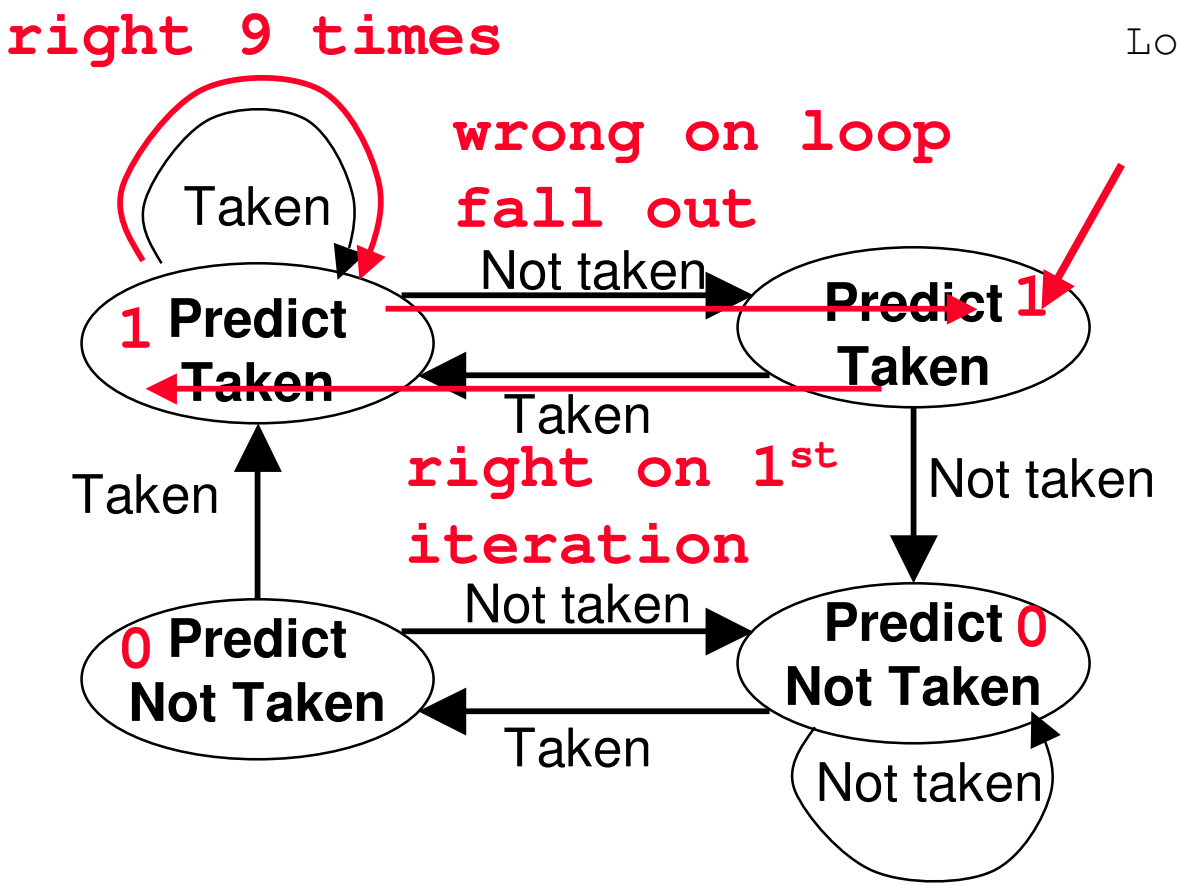



```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```


Fall 2004 © UC Regents

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




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


Fall 2004 © UC Regents

Delayed Decision

- First, move the branch decision hardware and target address calculation to the ID pipeline stage
- A delayed branch always executes the next sequential instruction – the branch takes effect after that next instruction
  - MIPS software moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay
- As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot.
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Fall 2004 © UC Regents

Scheduling Branch Delay Slots

A. From before branch

```
add  $1,$2,$3
if $2=0 then
delay slot
```

becomes

```
if $2=0 then
add  $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
delay slot
```

becomes

```
add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

C. From fall through

```
add  $1,$2,$3
if $1=0 then
delay slot

sub $4,$5,$6
```

becomes

```
add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Fall 2004 © UC Regents

Brain storm on bugs (if time permits)

- Depending on branch solution (move to ID, delayed, static prediction, dynamic prediction), where are bugs likely to hide?

1.

2.

…

- How can you write tests to uncover these likely bugs?

1.

2.

…

- Once it passes a test, don't need to run it again?

Fall 2004 © UC Regents

In Conclusion

- Data dependencies in pipelines often solved by forwarding
- Need to be sure prior instructions will write, destination matches source, and no earlier instruction has priority
- Need forwarding hardware every place where can forward, stall if stage needs to wait for result
  - EX stage, MEM stage for store, ID stage for early branch
- Loads require stall since overlap EX and MEM stages
  - Branches may require stall too
- Control hazards improved via delayed branch/jump in ISA, static prediction for branches, dynamic prediction for branches
  - If predict, hard part of design is recovering from misprediction

Fall 2004 © UC Regents