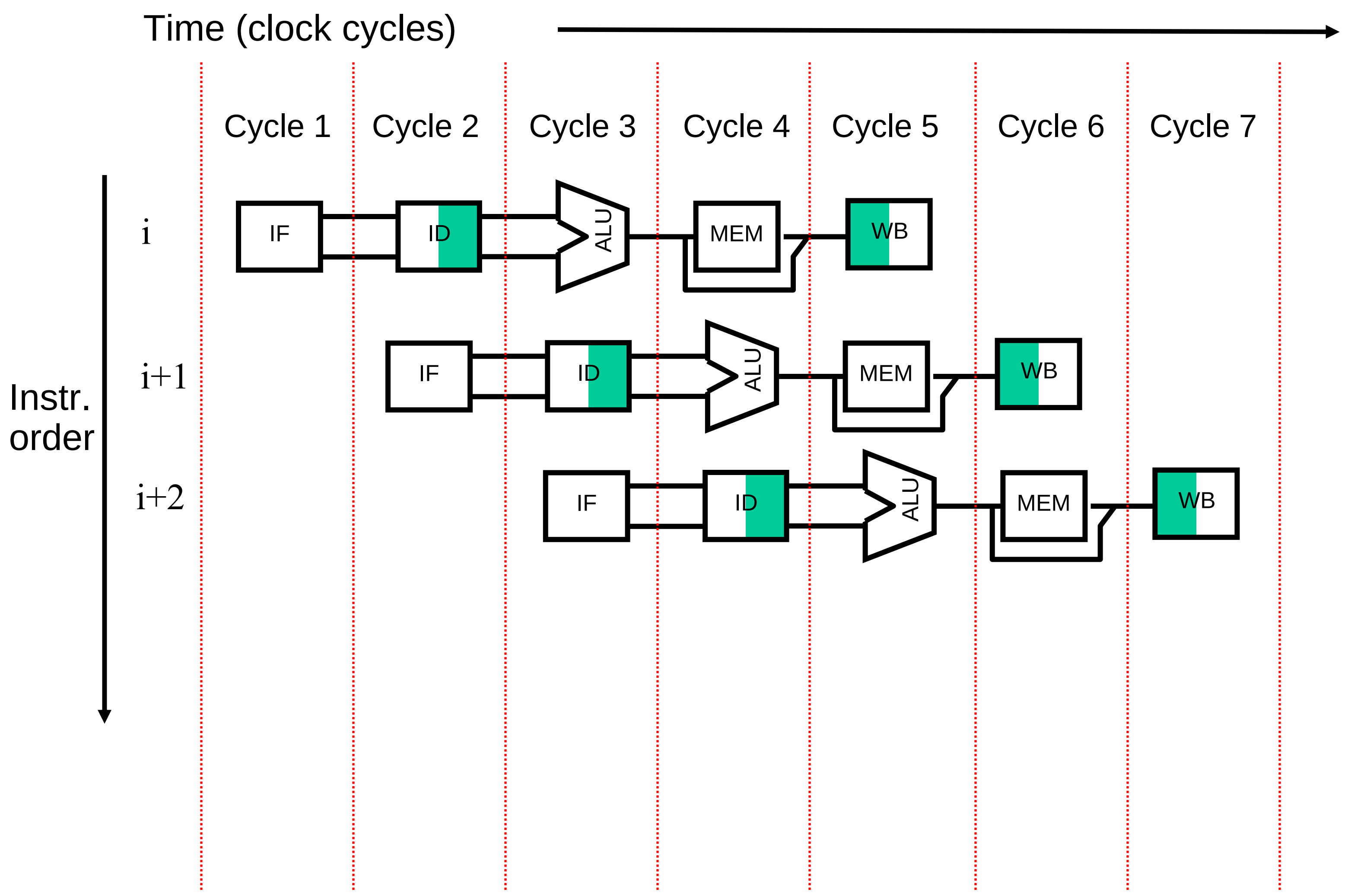
## Recap

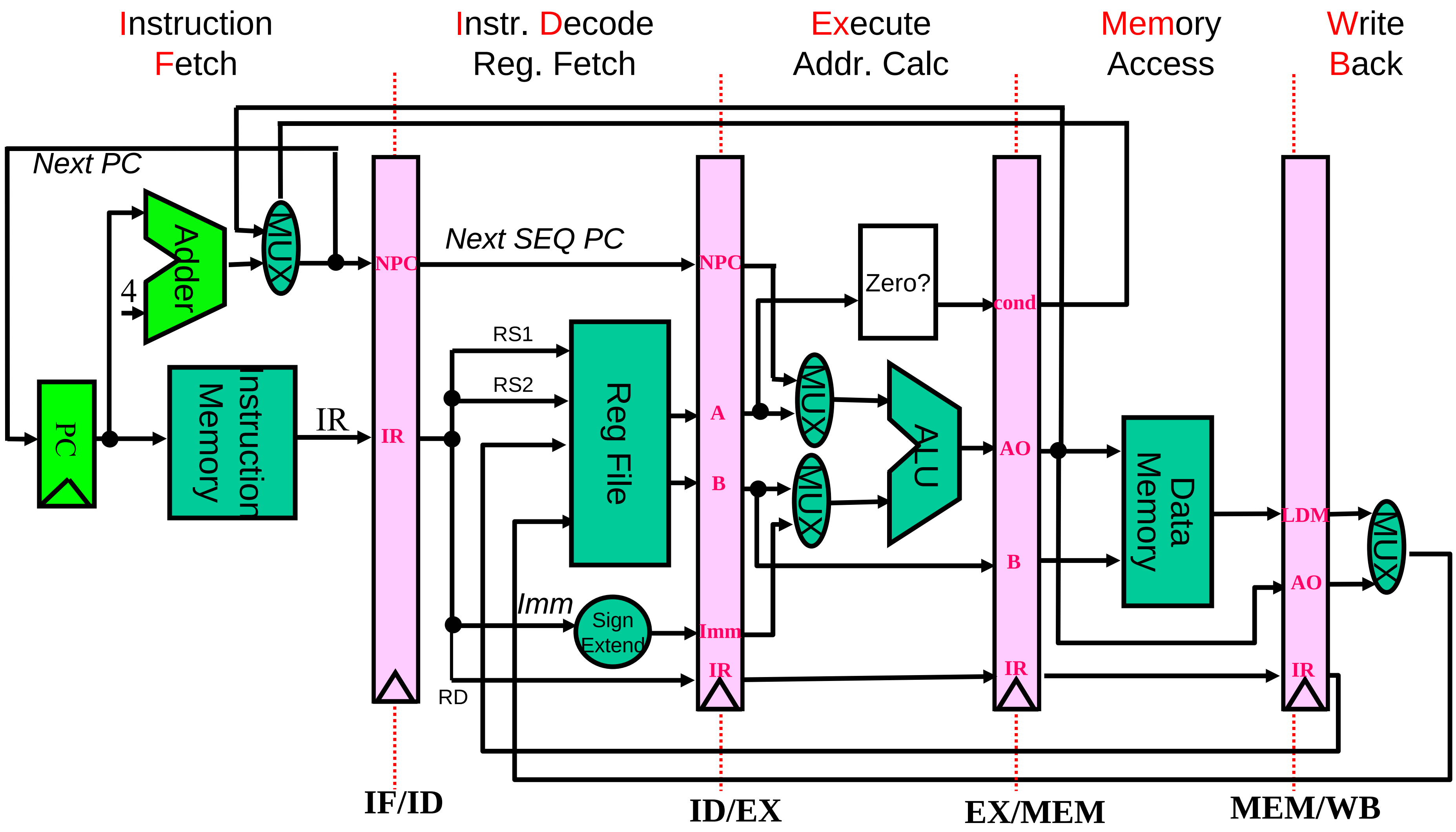
- Computers execute billions of instructions, so instruction throughput is what matters
- Main idea behind pipelining: Divide instruction execution across several stages
  - each stage accesses only a subset of the CPU's resources
- Example: Classic 5-stage RISC pipeline

  IF ID EX MEM WB
- Simultaneously have different instructions in different stages
  - Ideally, can issue a new instruction every cycle

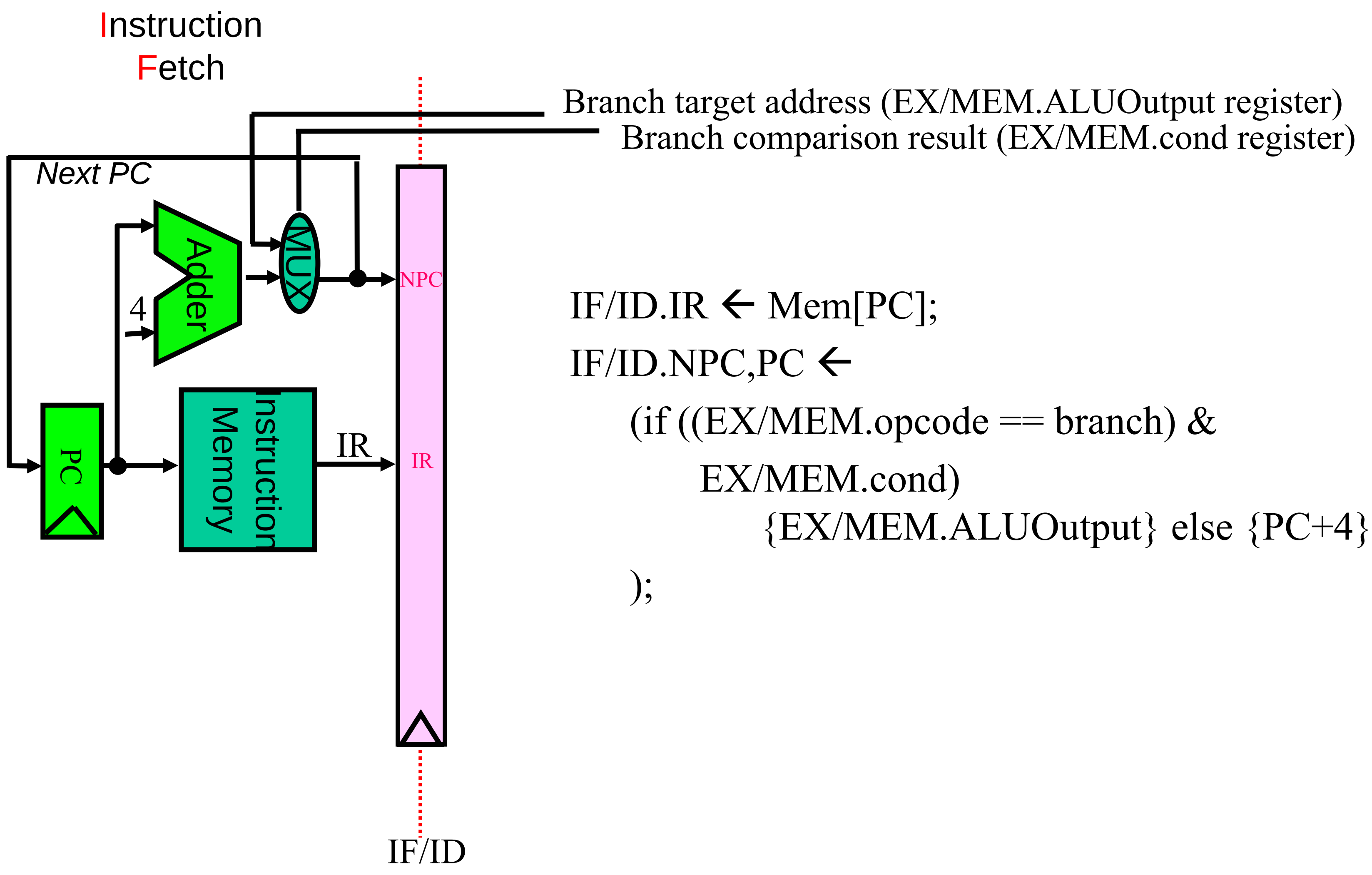
## Recap (Cont'd)


Time (clock cycles)
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Instr. order
i
i+1
i+2
IF
ID
ALU
MEM
WB

# Pipelined Implementation of a RISC ISA

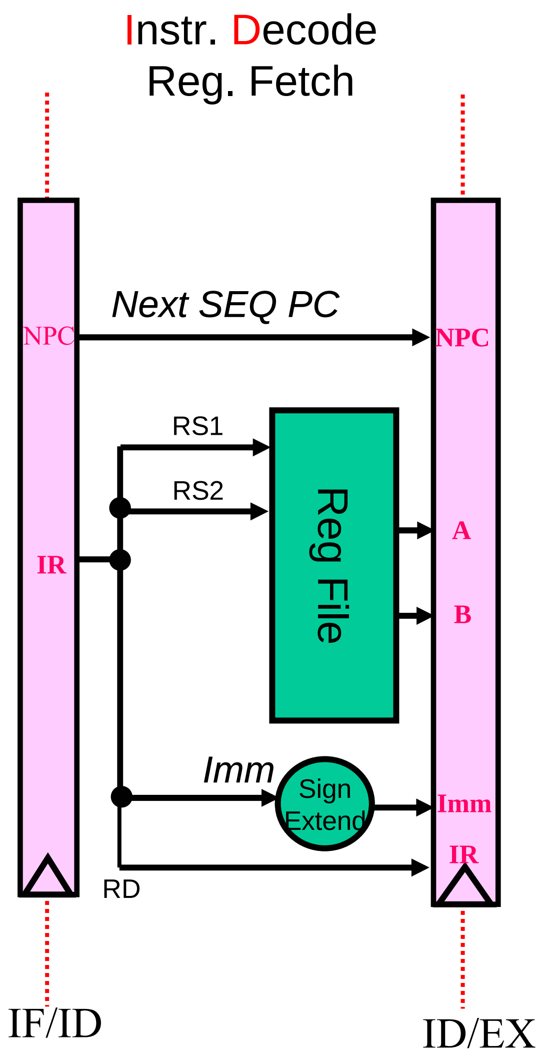


# Pipeline stage: Instruction Fetch (IF)


Instruction
Fetch
Branch target address (EX/MEM.ALUOutput register)
Branch comparison result (EX/MEM.cond register)
Next PC
Adder
4
MUX
NPC
PC
Instruction Memory
IR
IR
IF/ID

```
IF/ID.IR ← Mem[PC];
IF/ID.NPC,PC ←
    (if ((EX/MEM.opcode == branch) &
        EX/MEM.cond)
            {EX/MEM.ALUOutput} else {PC+4}
    );
```

## Pipeline stage: Instruction Decode (ID)



ID/EX.A← Regs[IF/ID.IR[rs]];

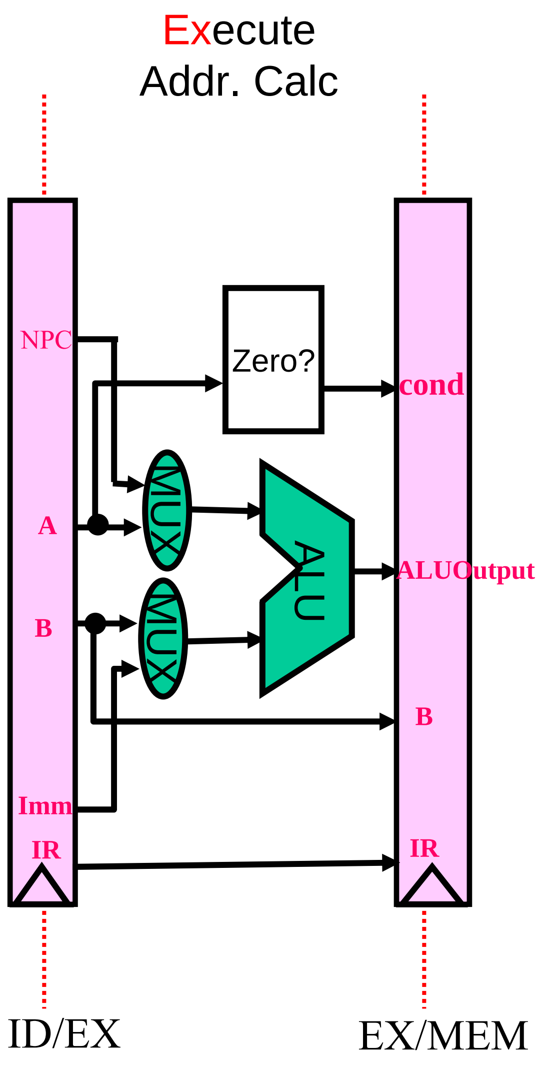
ID/EX.B ←Regs[IF/ID.IR[rt]];

ID/EX.NPC←IF/ID.NPC;

ID/EX.IR←IF/ID.IR;

ID/EX.Imm← sign-extend (IF/ID.IR[immediate field])

## Pipeline stage: Execute (EX)



**ALU instruction**

EX/MEM.IR ← ID/EX.IR;

EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B;

or

EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;

**Branch instruction**

EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm <<2);

EX/MEM.cond ← (ID/EX.A ==0);

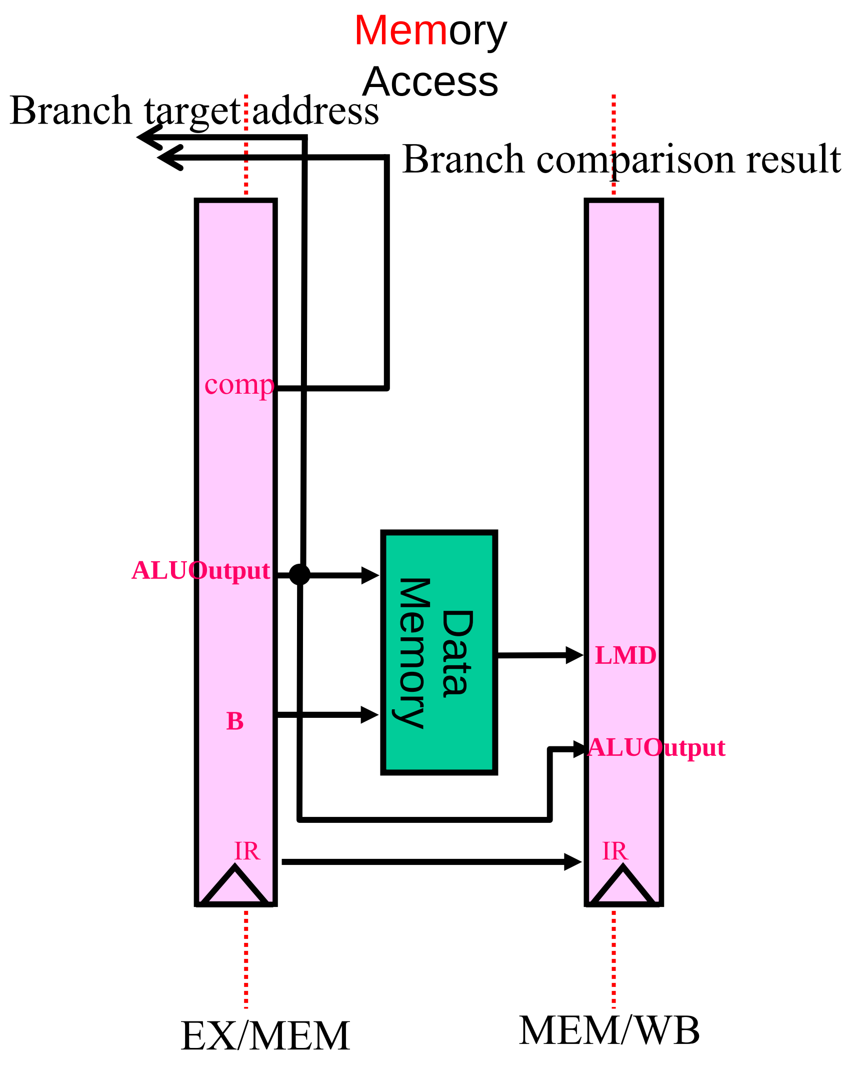
**Load/store instruction**

EX/MEM.IR←ID/EX.IR;

EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;

EX/MEM.B ← ID/EX.B;

## Pipeline stage: Memory access (MEM)



<u>ALU instruction</u>

MEM/WB.IR←EX/MEM.IR;

MEM.WB.ALUOutput← EX/MEM.ALUOutput;

<u>Load/store instruction</u>

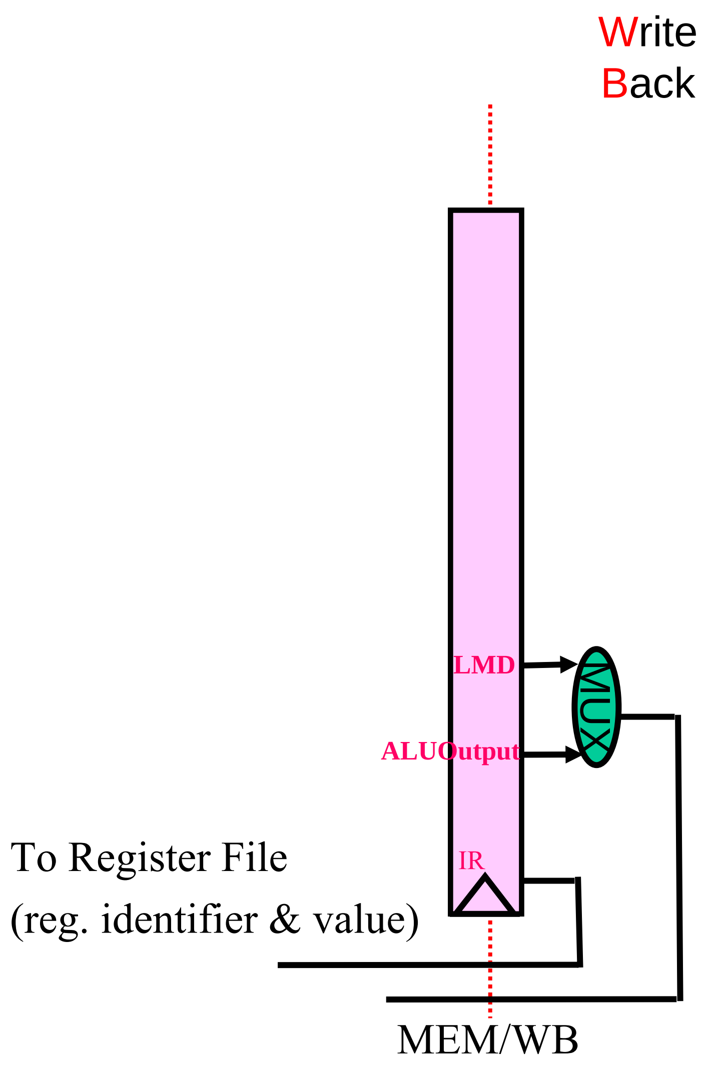
MEM/WB.IR←EX/MEM.IR;

MEM/WB.LMD ← Mem[EX/MEM.ALUOutput];

or

Mem[EX/MEM.ALUOutput]← EX/MEM.B;

## Pipeline stage: Write Back (WB)



ALU instruction

Regs[MEM/WB.IR[rd]] ← MEM.WB.ALUOutput;

or

Regs[MEM/WB.IR[rt]] ← MEM.WB.ALUOutput;

Load instruction only

Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD

## Pipeline Hazards

- Should we expect a CPI of 1 in practice?
- Unfortunately, the answer to the question is NO.
- Limit to pipelining: Hazards
  - Prevent next instruction from executing during its designated clock cycle
- Three classes of hazards

  Structural: Hardware cannot support this combination of instructions - two instructions need the same resource.

  Data: Instruction depends on result of prior instruction still in the pipeline

  Control: Pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline
  - To do this, hardware or software must detect that a hazard has occurred

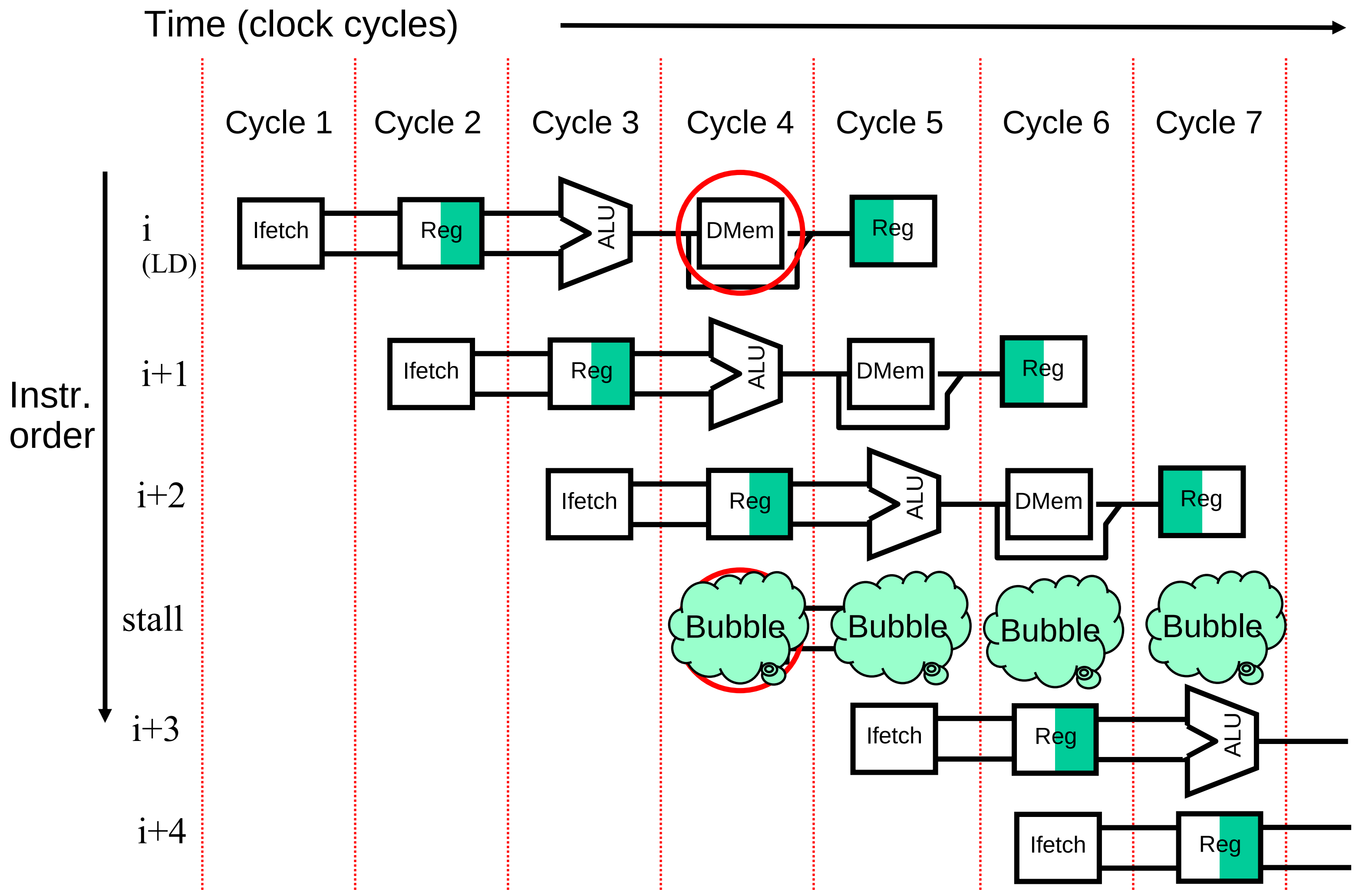
## Pipeline Hazards (A): Structural Hazards

- Occur when two or more instructions need the same resource
- Common methods for eliminating structural hazards are:
  - Duplicate resources
  - Pipeline the resource
  - Reorder the instructions

- It may be too expensive to eliminate a structural hazard, in which case the pipeline should stall
  - no new instructions are issued until the hazard has been resolved

- What are some examples of structural hazards?

# One Memory Port Structural Hazard



# Pipeline Speedup Example: One or Two Memory Ports

- Two machines
  - Machine A: Dual ported memory
  - Machine B: Single ported memory, but its pipelined implementation has a clock rate that is 1.05 times faster
  - Ideal CPI = 1 for both
  - Loads are 40% of instructions executed (cause stalls in machine B)
- Which is faster?

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

$\text{Speedup}_A$ = (Pipeline Depth/(1 + 0)) x 1

= Pipeline Depth

$\text{Speedup}_B$ = (Pipeline Depth/(1 + 0.4 x 1)) x 1.05

= 0.75 x Pipeline Depth

Machine A is 1.33 times faster

# Pipeline Hazards (B): Data Hazards

Three generic types of data hazards

- Read After Write (RAW)
  - $\text{Instr}_J$ tries to read operand before $\text{Instr}_I$ (I < J) writes it
  - Called a dependence

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Write After Read (WAR)
  - $\text{Instr}_J$ writes operand before $\text{Instr}_I$ reads it
  - Called an anti-dependence
    - Name dependence (renaming)
    - No value being transmitted

```
I: sub r4,r1,r3
J: add r1,r2,r3
```

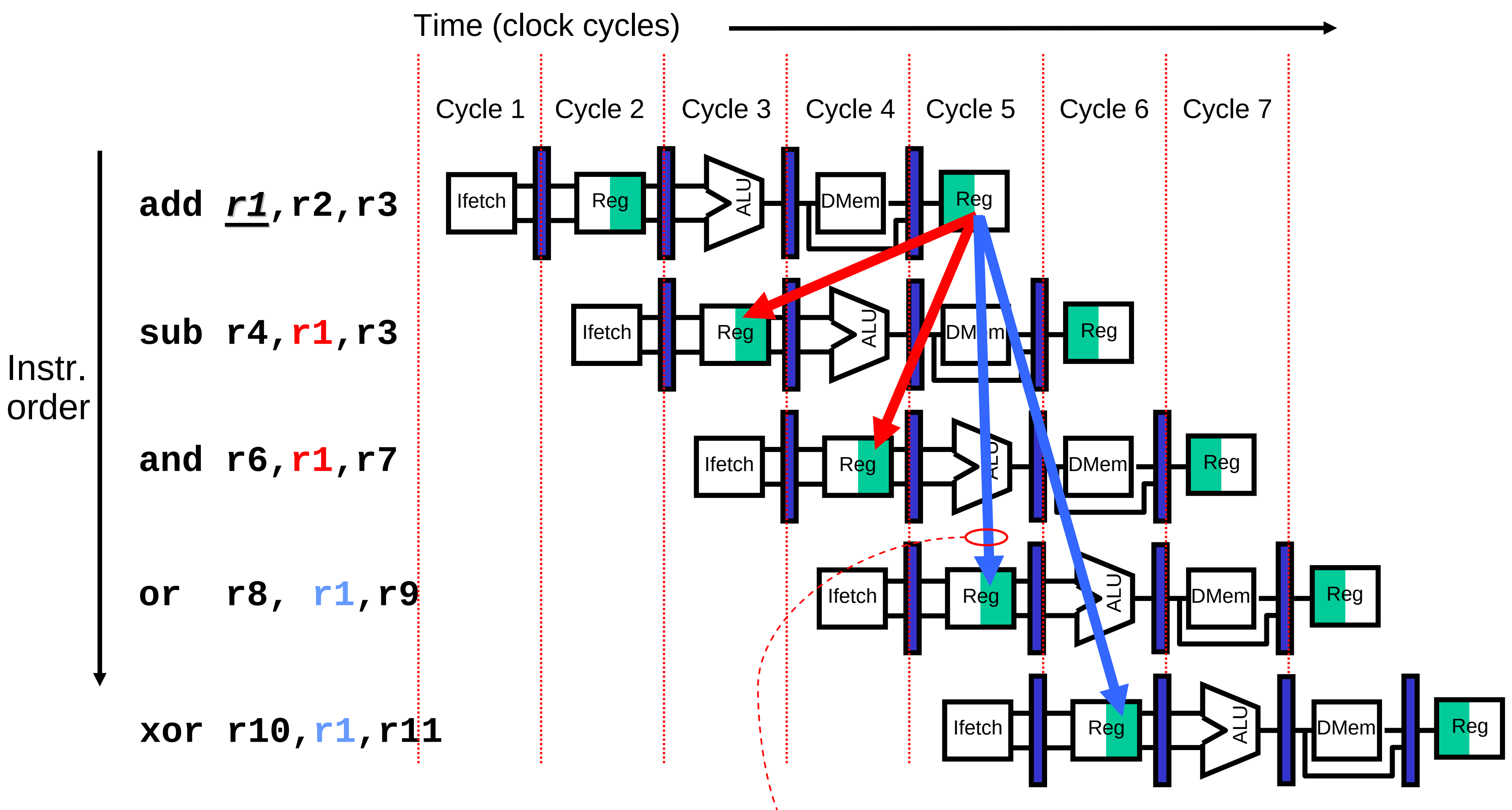
- Write After Write (WAW)
  - $\text{Instr}_J$ writes operand before $\text{Instr}_I$ writes it
  - Called an output dependence
    - Name dependence (renaming)
    - No value being transmitted

```
I: sub r1,r4,r3
J: add r1,r2,r3
```

## Data Hazards and Pipeline Stalls

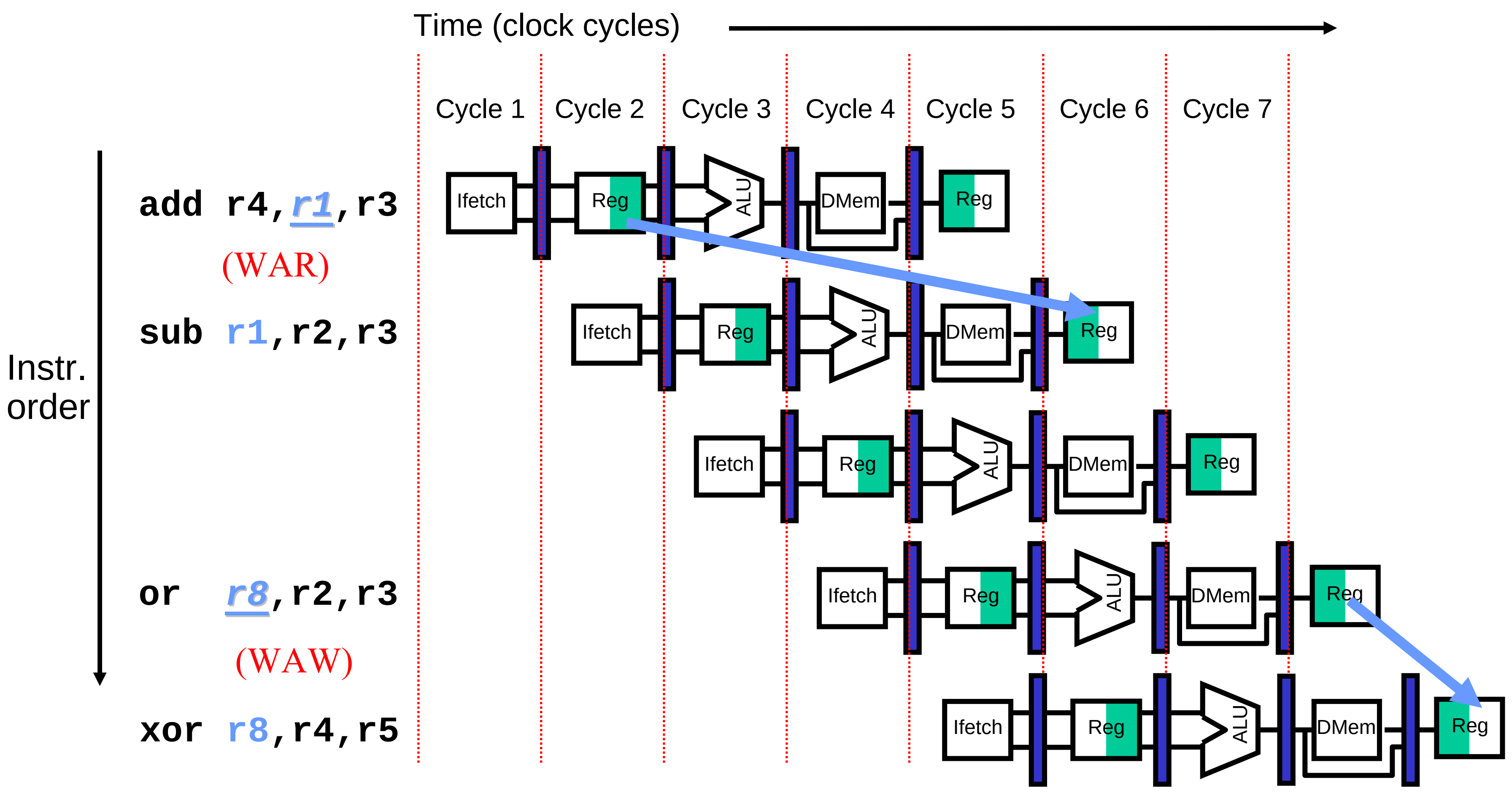
- Do all kinds of data hazards translate into pipeline stalls?
- NO, whether or not a data hazard results in a stall depends on the pipeline structure
- For the simple five-stage RISC pipeline
  - Only RAW hazards result in a pipeline stall
    - Instruction reading a register needs to wait until it is written
  - WAR and WAW hazards cannot occur because
    - All instructions take 5 stages
    - Reads happen in the 2nd stage (ID)
    - Writes happen in the 5th stage (WB)
    - No way for a write from a subsequent instruction to interfere with the read (or write) of a prior instruction
- For more complicated pipelines (later in the course)
  - Both WAR and WAW hazards are possible if instructions execute out of order or access (read) data later in the pipeline

# RAW Hazards in the 5-stage Pipeline



Forwarding through the register file

# Absence of WAR and WAW Hazards



## Reducing Impact of RAW Hazards: Data Forwarding

- Data forwarding (also called bypassing or short-circuiting)
  - Directly transfers data from each stage to earlier pipeline stages
    - Result is accessible before it gets written into the register file.

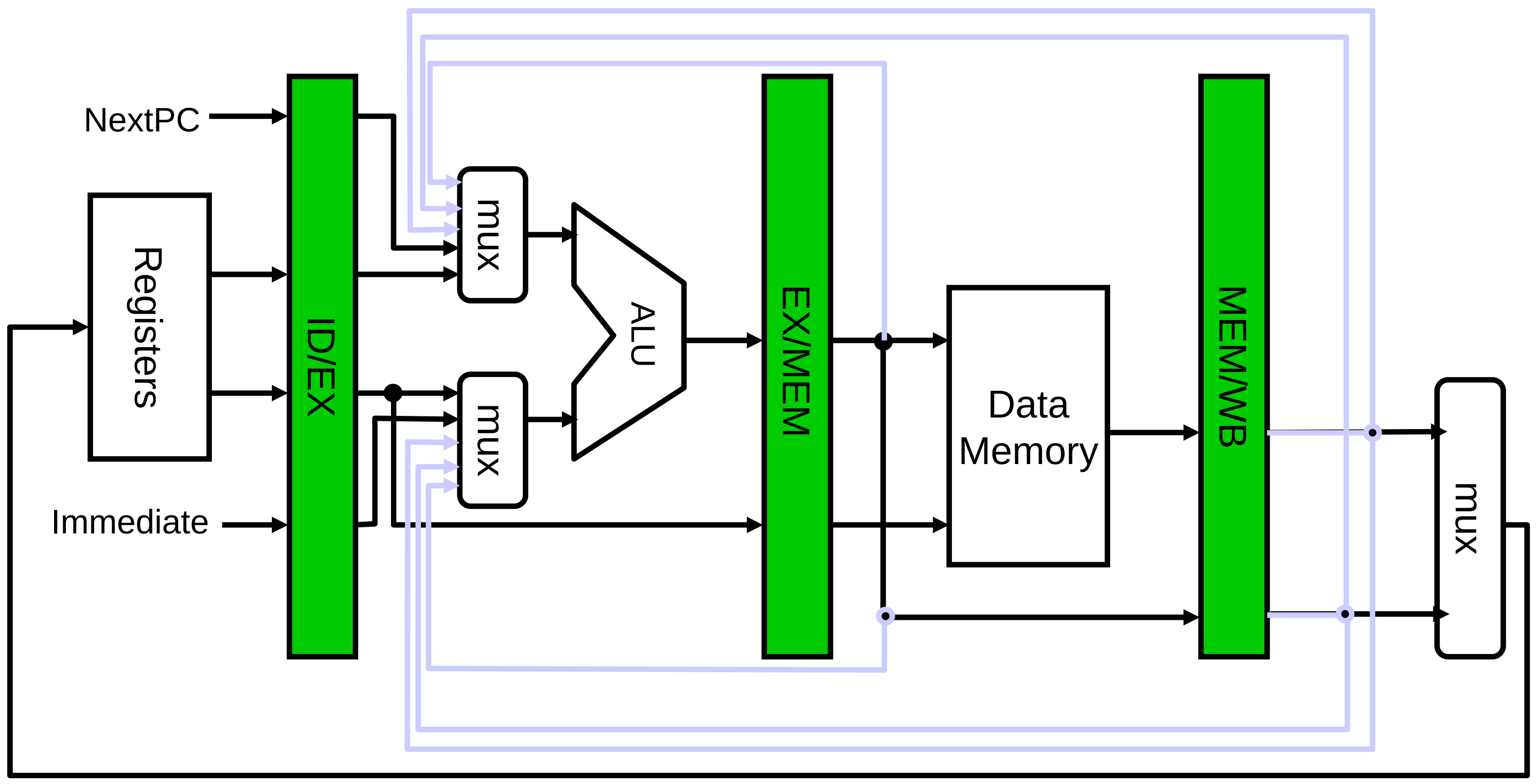
Instr i: **`add r1,r2,r3`** (result ready after EX stage)

----------------------

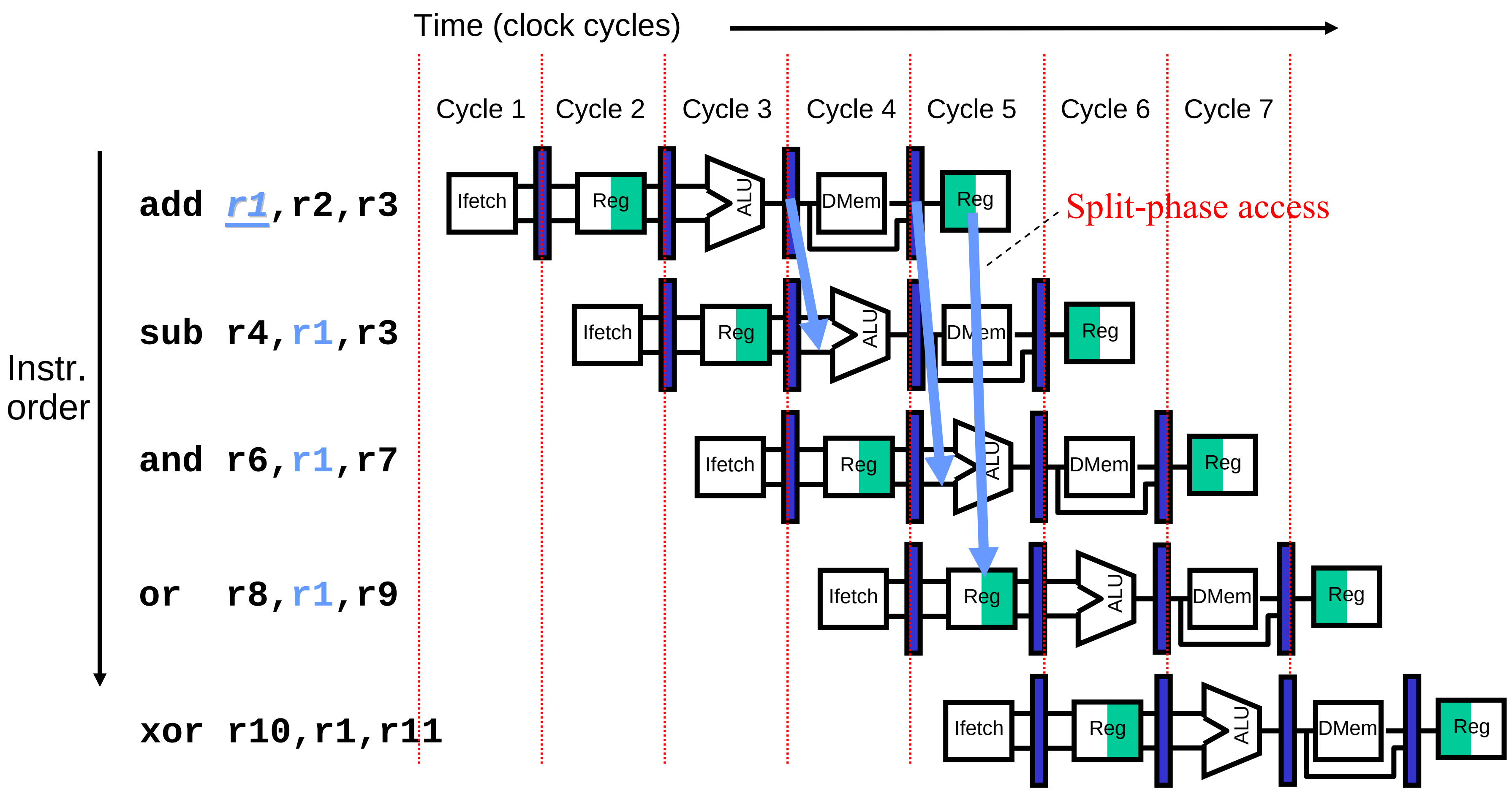
Instr j: **`sub r4,r1,r5`** (result needed in EX stage)

- To support data forwarding, additional hardware is required.
  - Multiplexers to allow data to be transferred back
  - Control logic for the multiplexers

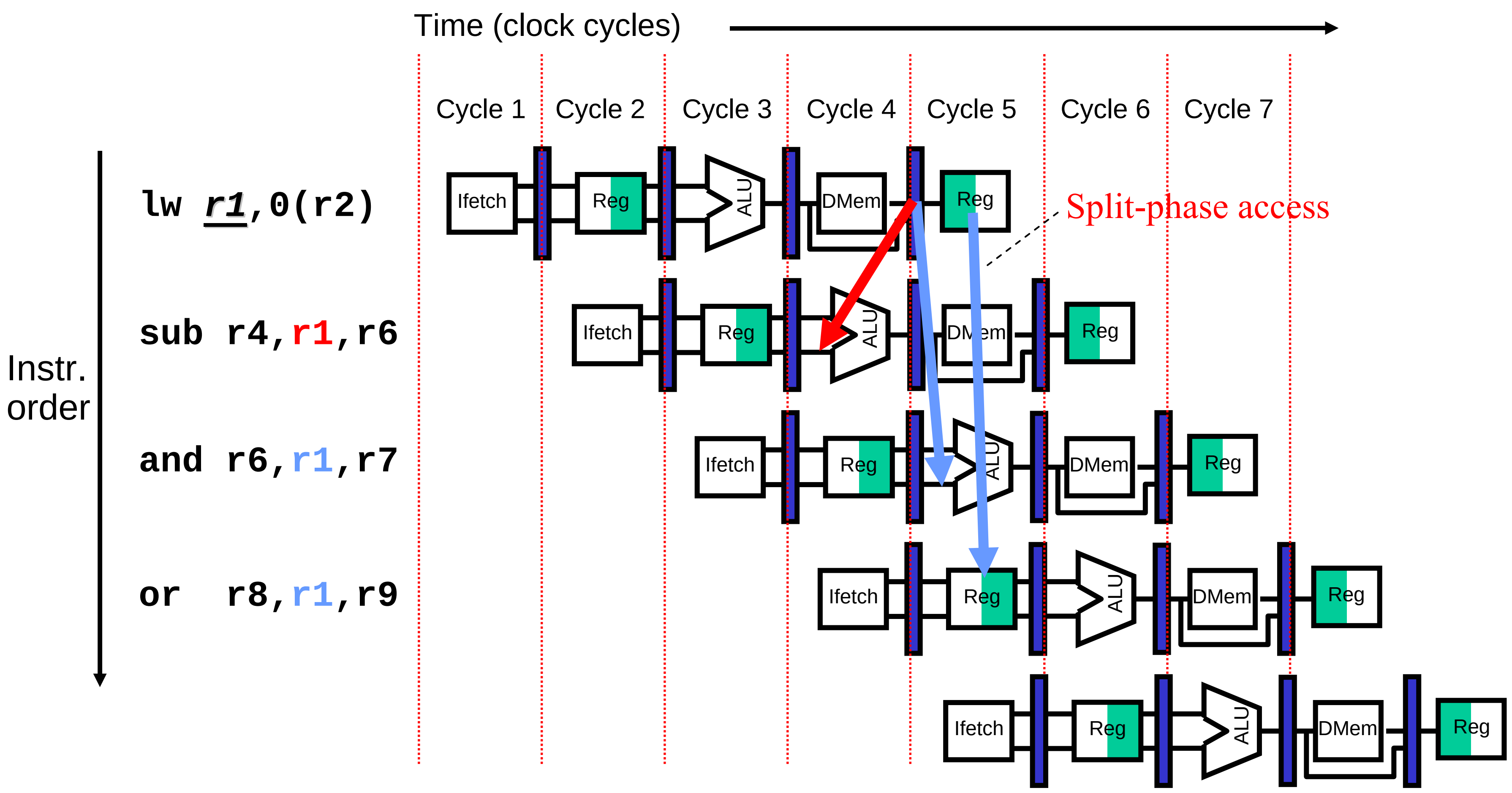
# Hardware Changes for Forwarding



## Avoidance of RAW Hazards Using Forwarding



## Forwarding Does Not Eliminate All Hazards

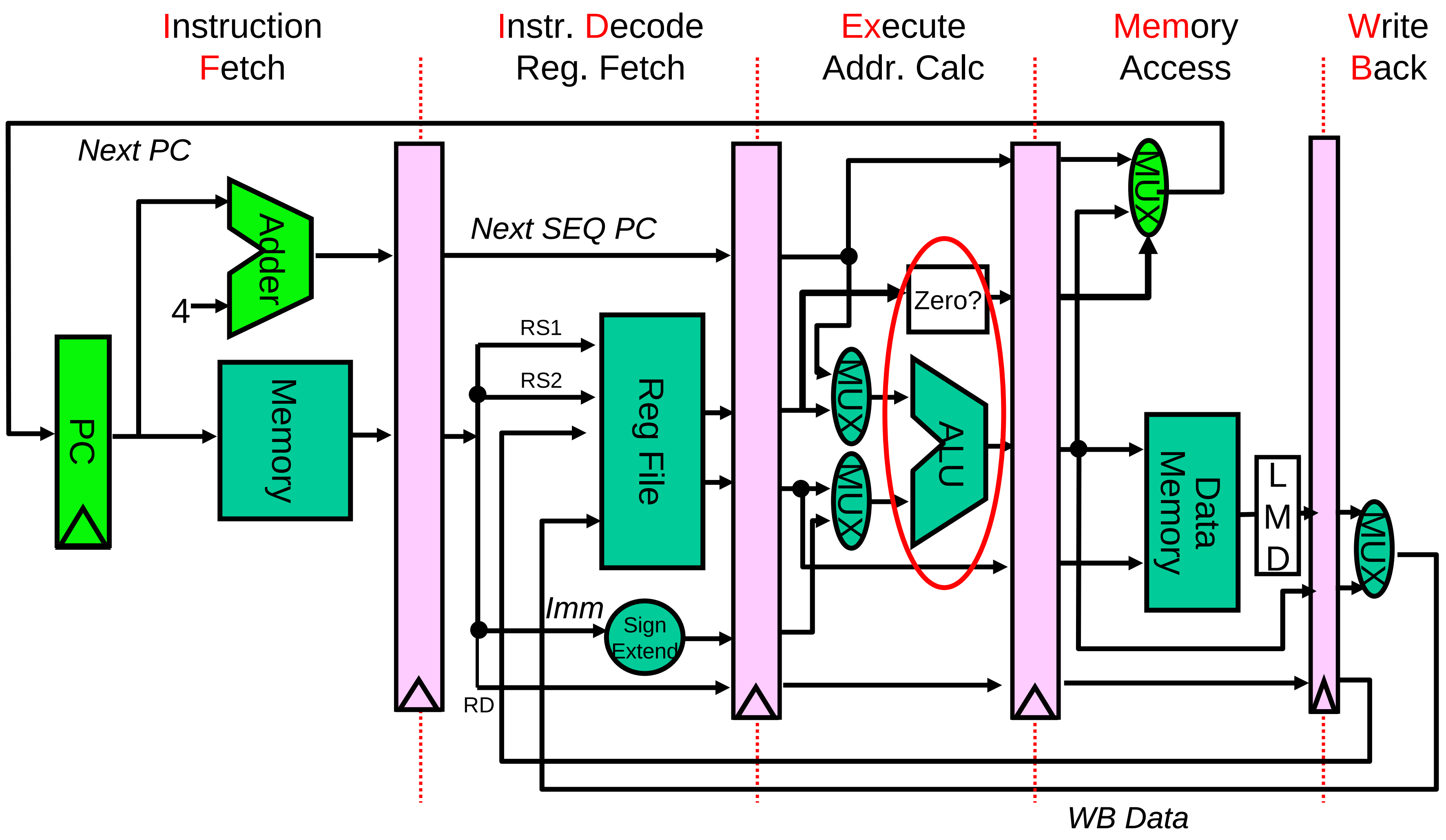


Cope with this by stalling the EXE stage till results are available

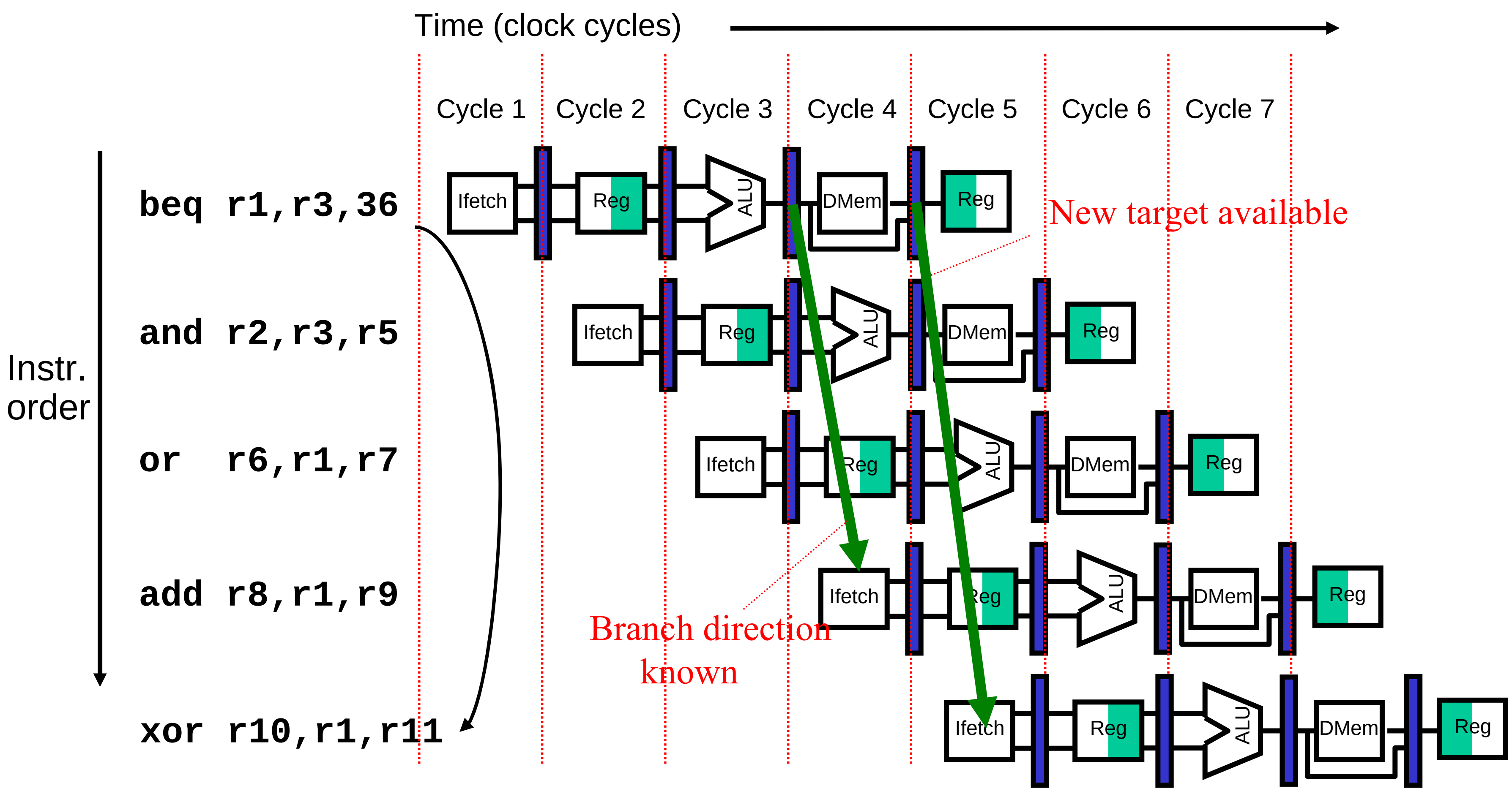
## Pipeline Hazards (C): Control Hazards

- Control hazards occur due to instructions changing the PC
  - can result in a large performance loss
- A branch is either
  - Taken: PC $\leftarrow$PC + Imm
  - Not Taken: PC $\leftarrow$ PC + 4
- Cannot fetch the next instruction till value of PC is known
- Simplest solution is to stall the pipeline upon detecting a branch
  - ID stage detects the branch
  - Don't know if the branch is taken until the EX stage
  - New PC is not changed until the end of the MEM stage, after determining if the branch is taken and the new PC value
  - If the branch is taken, we need to repeat some stages and fetch new instructions

## (Review) Pipelined Implementation of a RISC ISA



# 3 Cycle Stall on Branch-Induced Control Hazards



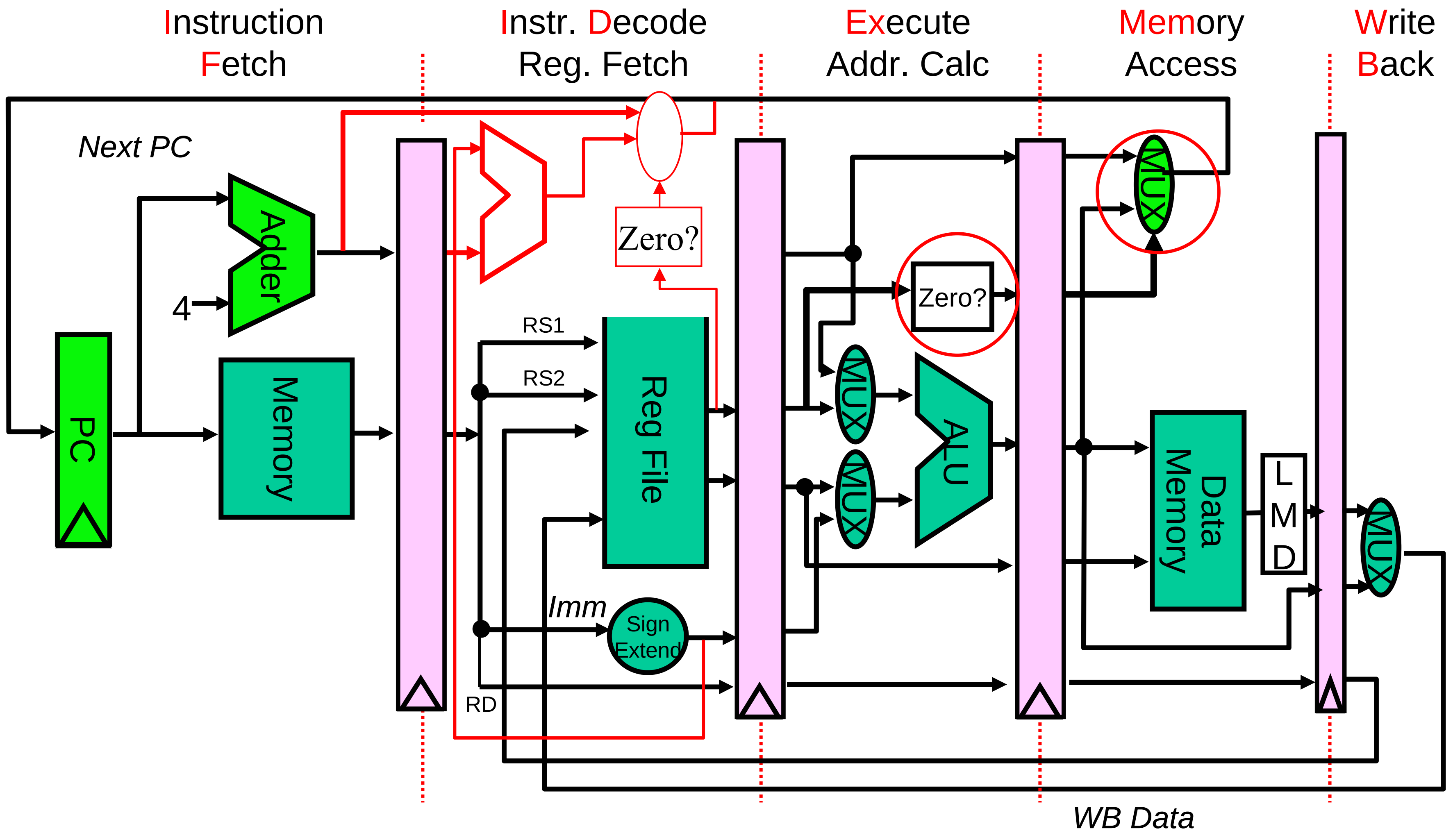
## Impact of Branch Stalls

- If CPI = 1, 30% branches
  - Stall 3 cycles => new CPI = (1 + 0.3*3) = 1.9!
  - 50% of these branches taken => new CPI = 1 + 0.15*3 + 0.15*2 = 1.7

- Penalty would be worse for current-day (longer) pipelines
  - IF and ID-like stages are each multiple-cycle

- How do we reduce impact of branch stalls?
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier

# Pipelined Implementation of a RISC ISA: Reducing Branch Penalty to 1 cycle



## Branch Behavior in Programs

- Based on SPEC benchmarks on DLX (CA-AQA, 2nd Edition)
  - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
  - About 75% of the branches are forward branches
  - 60% of forward branches are taken
  - 80% of backward branches are taken

- Why are branches (especially backward branches) more likely to be taken than not taken?

## Dealing with Branch Stalls

- Approach 1: Stall until branch direction is clear
- Approach 2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - PC+4 already calculated, so use it to get next instruction; chances are the branch is not taken
  - “Squash” instructions in pipeline if branch actually taken
    - Can do this because CPU state not updated till late in the pipeline

| *Instr.* | *Clock Number* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i (T) | IF | ID | EX | MEM | WB | | | | |
| i+1 | | IF | idle | idle | idle | idle | | | |
| T | | | IF | ID | EX | MEM | WB | | |
| T+1 | | | | IF | ID | EX | MEM | WB | |
| T+2 | | | | | IF | ID | EX | MEM | WB |

## Dealing with Branch Stalls (cont'd)

- Approach 3: Predict Branch Taken
  - Most branches are taken
  - But haven't yet calculated target address in a 5-stage RISC pipeline
    - So, will still incur a 1-cycle latency
    - Makes sense on machines where branch target is known before outcome
      - (later: Branch Target Buffers)

- Approach 4: Delayed Branch
  - Define branch to take place AFTER n following instructions

`branch instruction`
  `sequential successor`$_1$
  `sequential successor`$_2$
  `........`
  `sequential successor`$_n$
`branch target if taken`

$\left.\begin{matrix} \\ \\ \\ \end{matrix}\right\}$ n branch delay slots

## Branch Delay Slots

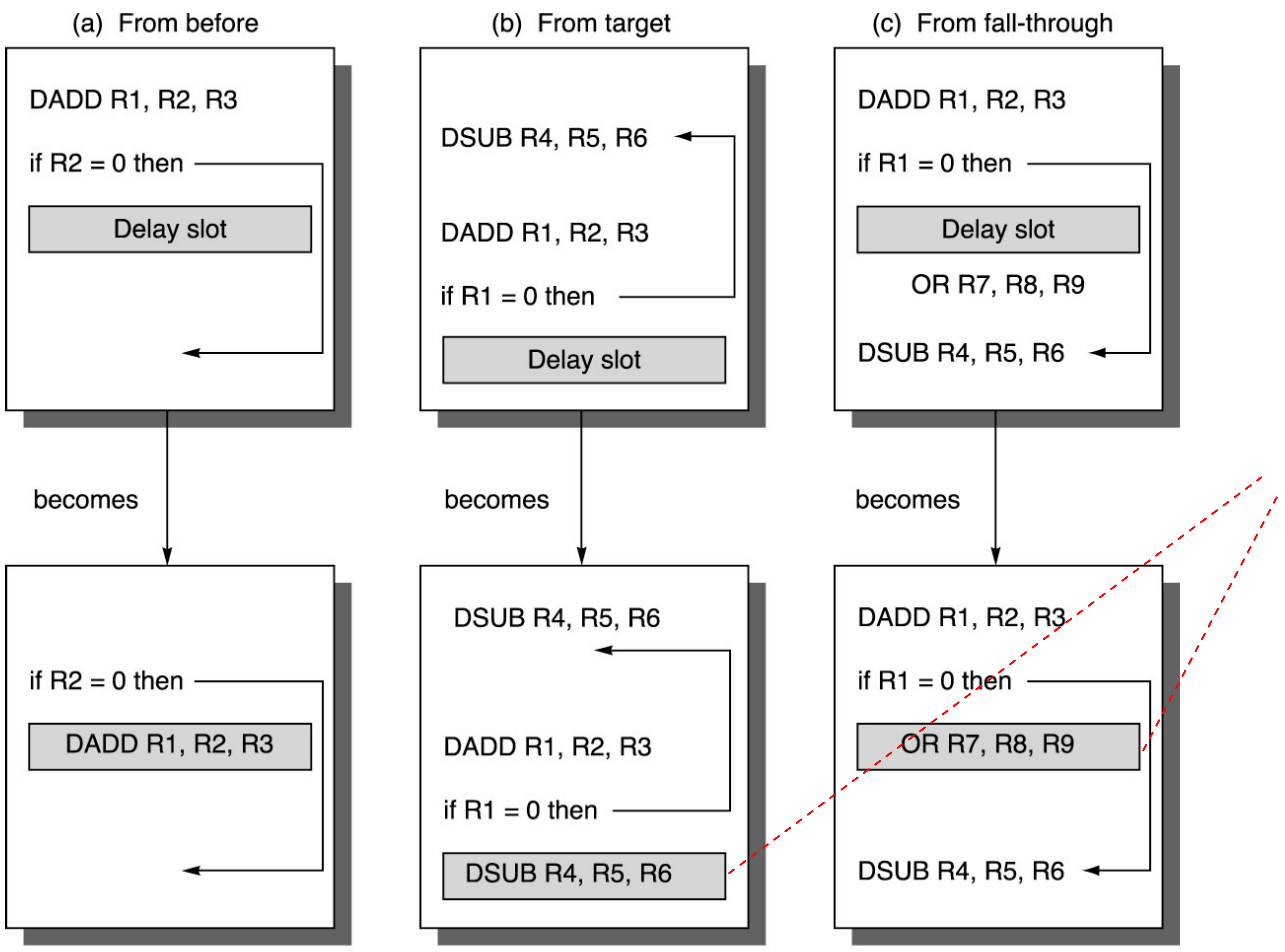
- Instructions in the branch delay slot(s) get executed whether or not branch is taken

| *Instr.* | *Clock Number* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i (T) | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | |
| D(i+1) | | **IF** | **ID** | **EX** | **MEM** | **WB** | | | |
| T | | | **IF** | **ID** | **EX** | **MEM** | **WB** | | |
| T+1 | | | | **IF** | **ID** | **EX** | **MEM** | **WB** | |
| T+2 | | | | | **IF** | **ID** | **EX** | **MEM** | **WB** |

- Heavily used in early RISC machines
  - 1 delay-slot suffices for a 5-stage pipeline (target available at end of ID)
  - Machines with deep pipelines require additional delay slots to avoid branch penalties
    - Benefits are unclear

# Scheduling the Branch Delay Slot

Where does the instruction for the delay slot come from?



Nullifying or cancelling branches

– Converts delay slot instruction into a nop

## Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

- Assumptions
  - 14% of instructions are branches
  - 30% of branches are not taken
  - 50% of delay slots can be filled with useful instructions

| *Scheduling scheme* | *Branch penalty* | *CPI* | *speedup v. unpipelined* | *speedup v. stall* |
|---|---|---|---|---|
| Slow stall pipeline | 3 | 1.42 | 3.5 | 1.0 |
| Fast stall pipeline | 1 | 1.14 | 4.4 | 1.26 |
| Predict taken | 1 | 1.14 | 4.4 | 1.26 |
| Predict not taken | 0.7 | 1.10 | 4.5 | 1.29 |
| Delayed branch | 0.5 | 1.07 | 4.7 | 1.34 |

- A compiler can reorder instructions to further improve speedup

## Importance of Avoiding Branch Stalls

- Crucial in modern microprocessors, which issue/execute multiple instructions every cycle
  - Need to have a steady stream of instructions to keep the hardware busy
  - Stalls due to control hazards dominate
- So far, we have looked at static schemes for reducing branch penalties
  - Same scheme applies to every branch instruction
- Potential for increased benefits from dynamic schemes
  - Can choose most appropriate scheme separately for each instruction
    - Branches to top of loop have different behavior (Taken) than
      "if (x == 0) return;" (Not Taken)
  - Can "learn" appropriate scheme based on observed behavior
  - Dynamic (hardware) branch prediction schemes
    - For both direction (T or NT) and target prediction
    - Key element of all modern microprocessors