# CSE 502 Graduate Computer Architecture

# Lec 4-6 – Performance + Instruction Pipelining Review

**Larry Wittie**

**Computer Science, StonyBrook University**

**http://www.cs.sunysb.edu/~cse502 and ~lw**

**Slides adapted from David Patterson, UC-Berkeley cs252-s06**

# Review from last lecture

- Tracking and extrapolating technology part of architect's responsibility
- Expect Bandwidth in disks, DRAM, network, and processors to improve by at least as much as the square of the improvement in Latency
- Quantify Cost (vs. Price)
  – IC $\approx$ f($Area^2$) + Learning curve, volume, commodity, price margins
- Quantify dynamic and static power
  – Capacitance x $Voltage^2$ x frequency, Energy vs. power
- Quantify dependability
  – Reliability (MTTF vs. FIT), Availability (MTTF/(MTTF+MTTR)

# Outline

- Review
- F&P: Benchmarks age, disks fail, singlepoint fail danger
- 502 Administrivia
- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion

# Fallacies and Pitfalls (1/2)

- **Fallacies** - commonly held misconceptions
  - When discussing a fallacy, we try to give a counterexample.
- **Pitfalls** - easily made mistakes.
  - Often generalizations of principles true in limited context
  - Text shows Fallacies and Pitfalls to help you avoid these errors
- **Fallacy: Benchmarks remain valid indefinitely**
  - Once a benchmark becomes popular, there is tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark: "benchmarksmanship."
  - 70 benchmarks in the 5 SPEC releases to 2000. 70% dropped from the next release because no longer useful
- **Pitfall: A single point of failure**
  - Rule of thumb for fault tolerant systems: make sure that every component is redundant so that no single component failure can bring down the whole system (e.g, power supply)

**Lab rule of thumb: "Don't buy one of anything."**

# Fallacies and Pitfalls (2/2)

- Fallacy - Rated MTTF of disks is 1,200,000 hours or $\approx$ 140 years, so disks practically never fail
- But disk lifetime is 5 years $\Rightarrow$ replace a disk every 5 years; average, 28 replacements (in 140 yrs) so not fail
- A better unit: % that fail (1.2M MTTF = 833 FIT)
- Fail over lifetime: if had 1000 disks for 5 years $= 1000*(5*365*24)*833 / 10^9 = 36{,}485{,}000 / 10^6 = 37$ = 3.7% (37/1000) fail over 5 yr lifetime (1.2M hr MTTF)
- But this is under pristine conditions
  - little vibration, narrow temperature range $\Rightarrow$ no power failures
- Real world: 3% to 6% of SCSI drives fail per year
  - 3400 - 6800 FIT or 150,000 to 300,000 hour MTTF [Gray & van Ingen 05]
- 3% to 7% of ATA drives fail per year (Advanced Tech Attachment)
  - 3400 - 8000 FIT or 125,000 to 300,000 hour MTTF [Gray & van Ingen 05]

# CSE502: Administrivia

- **Location:** Light Engineering, Room 152
- **Time:** 2:20-3:40PM Tuesday/Thursday
- **Textbook:** Computer Architecture: A Quantitative Approach, Hennessy and Patterson, **5th** Edition (CAQA5, H+P); Elsevier/Morgan-Kaufmann (Sept 2011, "2012"), paperback, ISBN 978-0123838728
- **Instructor:** Professor Larry Wittie
- **Office/Lab:** CompSci Building, Room 1308
- **Office Hrs:** 4-5+7-7:30pm Tu/Th or when 1308 door open
- **Phone:** 632-8750
- **Email:** lw AT icDOTsunysbDOTedu
- **Homepage: http://www.cs.sunysb.edu/~cse502**
- **Current reading: Appendix C and parts of App. A near end of CAQA5 text. (Chap 1 was last week.)**

# Outline

- Review
- F&P: Benchmarks age, disks fail, single-points fail
- 502 Administrivia
- **MIPS – An ISA for Pipelining**
- **5 stage pipelining**
- **Structural and Data Hazards**
- **Forwarding**
- **Branch Schemes**
- **Exceptions and Interrupts**
- **Conclusion**

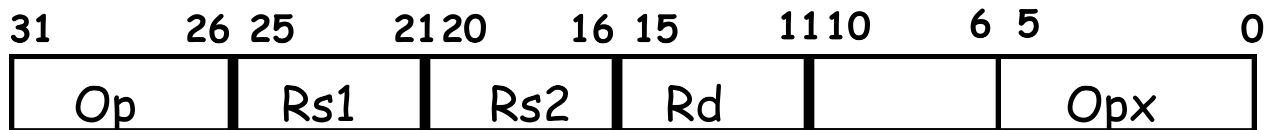
# A "Typical" RISC ISA – MIPS64

- 32-bit fixed format instruction (only 3 formats: RIJ)
  - Instruction 32-bit formats: Register, Immediate, Jump
- 32 64-bit General Purpose Reg. (GPR) R0-R31: integers
  - (R0 always = zero) (R31 <= PC+8 return address, Jump And Link calls)
- 32 64-bit Floating Point Register set (F0-F31)
- 3-address, reg-reg arithmetic instructions
- Single address mode for load/store (values: mem⇔ reg): base (in reg) + displacement (immediate, in instruction)
  - No indirection (thru ptr in memory, needing 2nd slow memory access)
- Simple branch conditions (e.g., single-bit: 0 or not?)
- (Delayed branch - ineffective in deep pipelines, so no longer used)

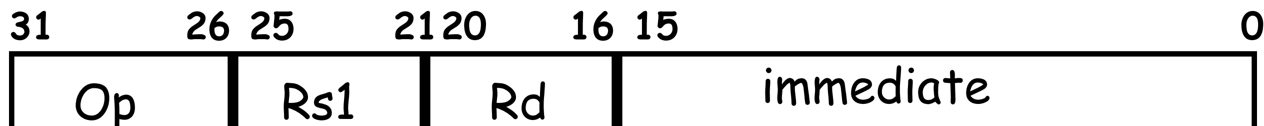
*see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3*

# Example of Fixed Format Instructions: MIPS

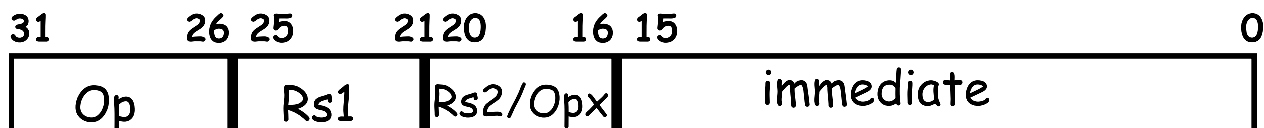
Register-Register – R Format – Arithmetic operations: Rd ← Rs1 Op Rs2




Register-Immediate – I Format – All immediate arithmetic ops

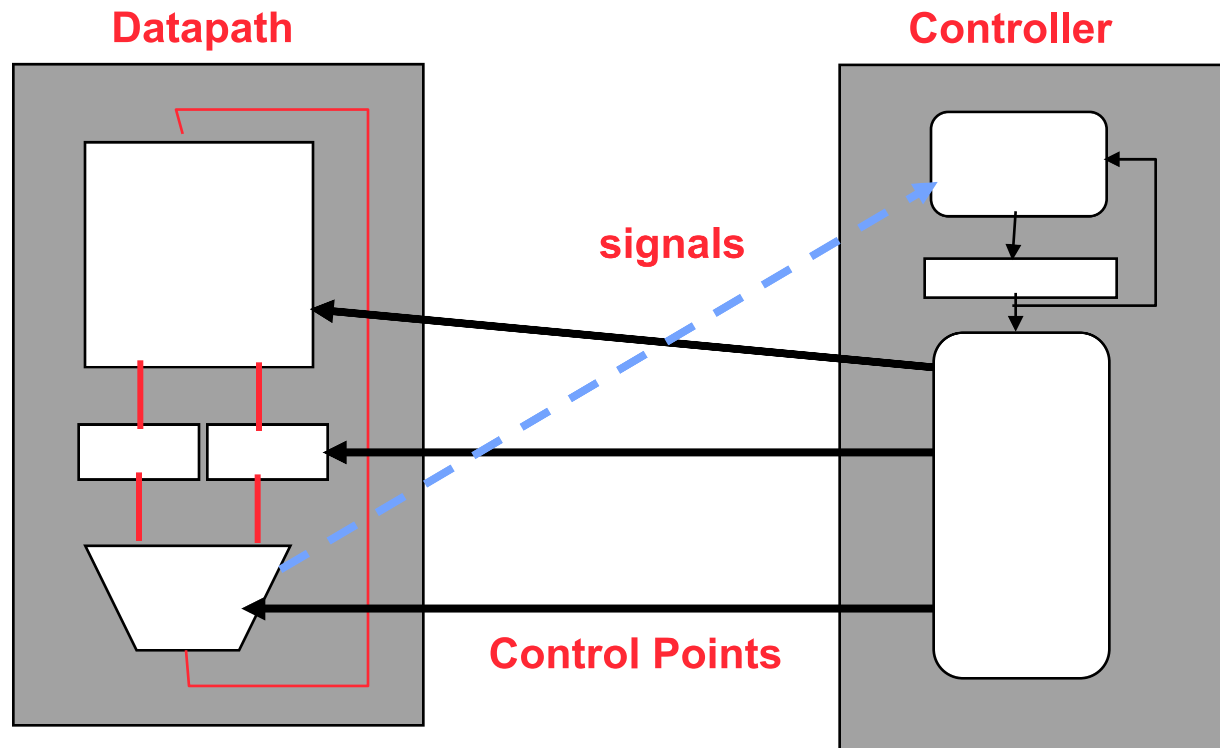



Branch – I Format – Moderate relative distance conditional branches
If OpCondition(Rs1,0 or Rs2) PC ← PC+4+Rs1*4; else PC+4




Jump/Call – J Format – Long distance jumps: $(PC+4)_{27..0}$ ← target*4

# Datapath vs Control



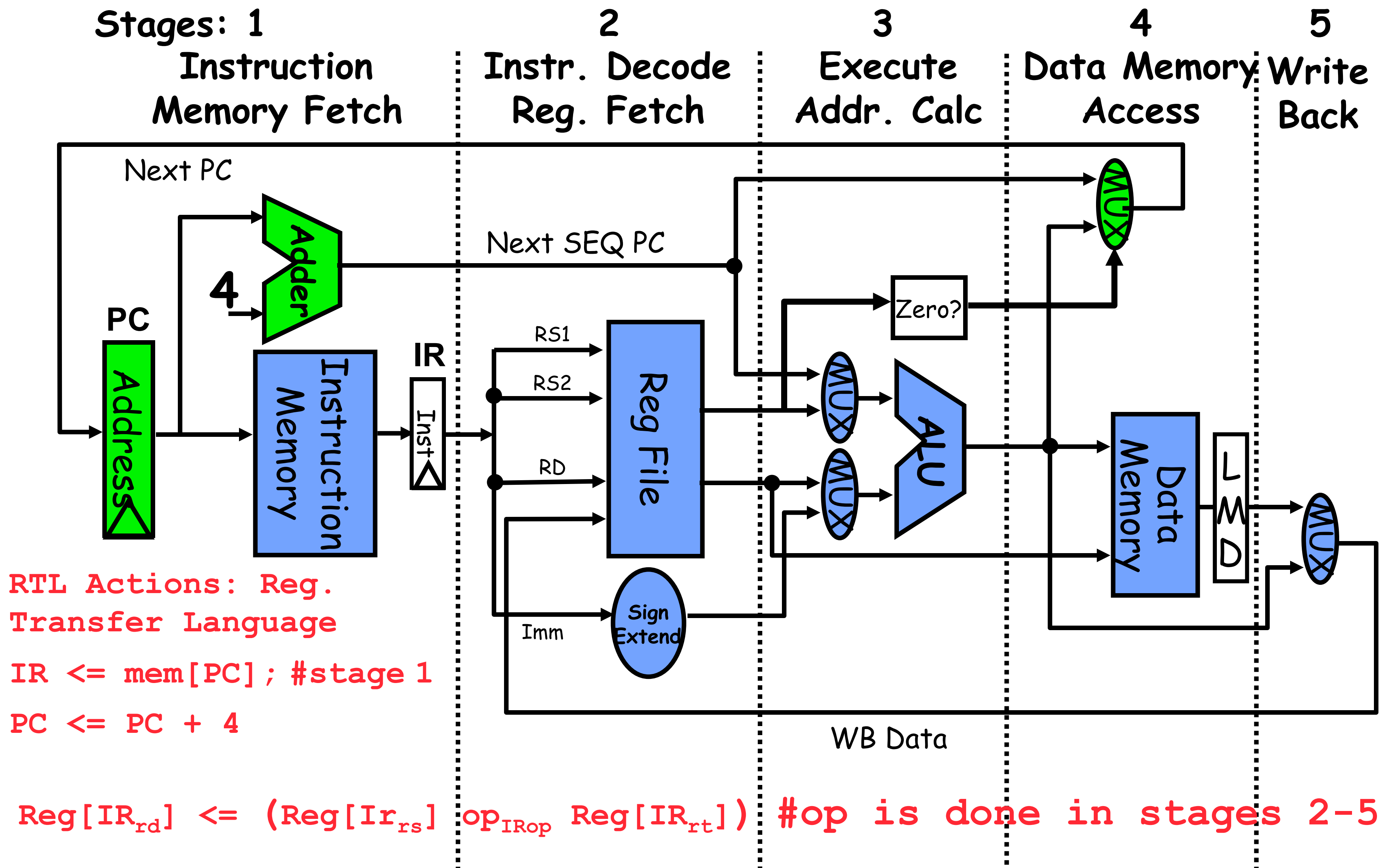

- **Datapath: Storage, Functional Units, Interconnections sufficient to perform the desired functions**
  - Inputs are Control Points
  - Outputs are signals
- **Controller: State machine to orchestrate operation on the data path**
  - Based on desired function and signals

# Approaching an ISA

- **Instruction Set Architecture**
  - **Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing**
- **Meaning of each instruction is described by RTL (register transfer language) on *architected registers* and memory**
- **Given technology constraints, assemble adequate datapath**
  - **Architected storage mapped to actual storage**
  - **Function Units (FUs) to do all the required operations**
  - **Possible additional storage (eg. Internal registers: MAR, MDR, IR, … {Memory Address Register, Memory Data Register, Instruction Register}**
  - **Interconnect to move information among registers and function units**
- **Map each instruction to a sequence of RTL operations**
- **Collate sequences into symbolic controller state transition diagram (STD)**
- **Lower symbolic STD to control points**
- **Implement controller**

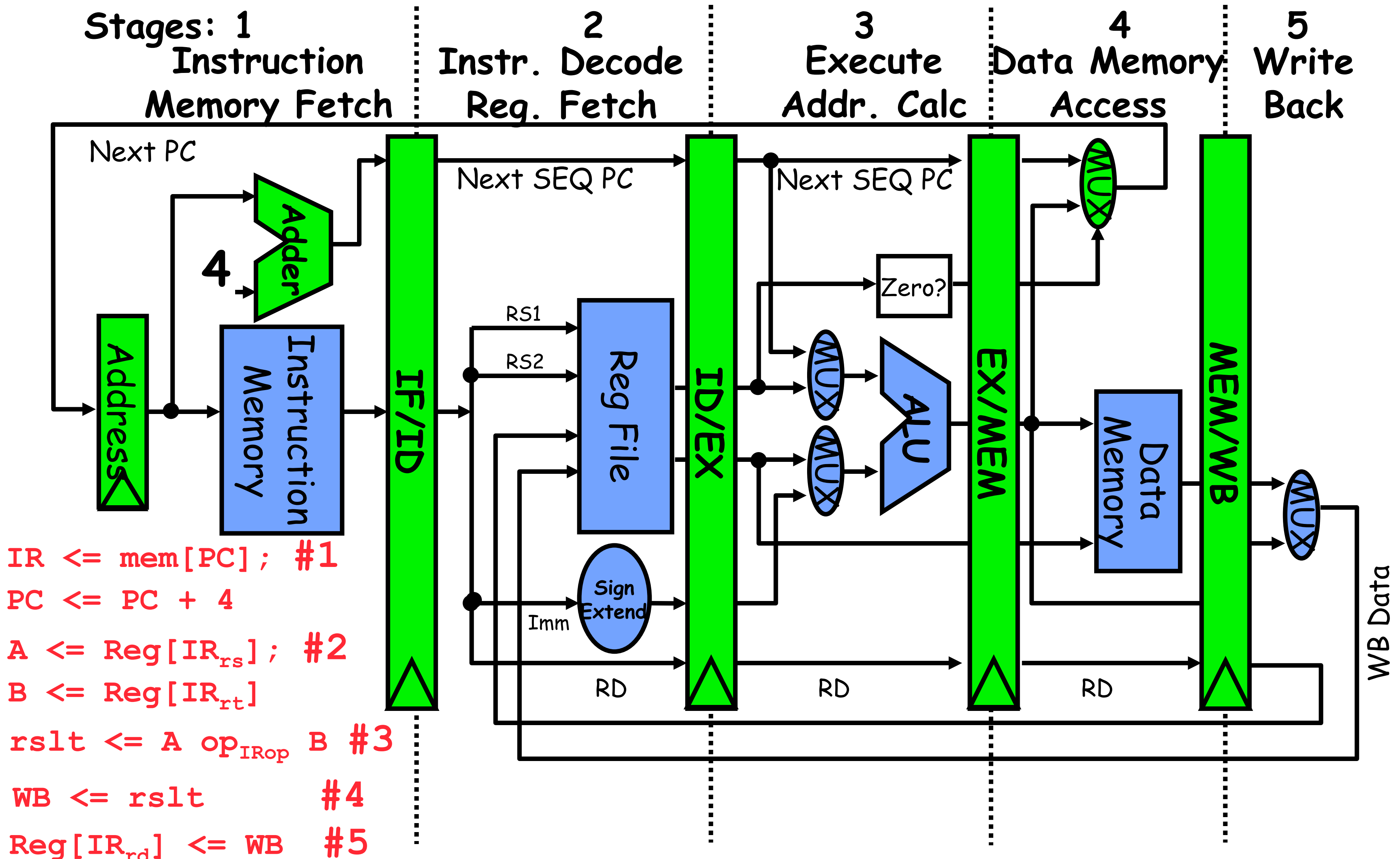
# 5 Steps of a (pre-pipelined) MIPS Datapath

**Figure C.21, Page C-34**



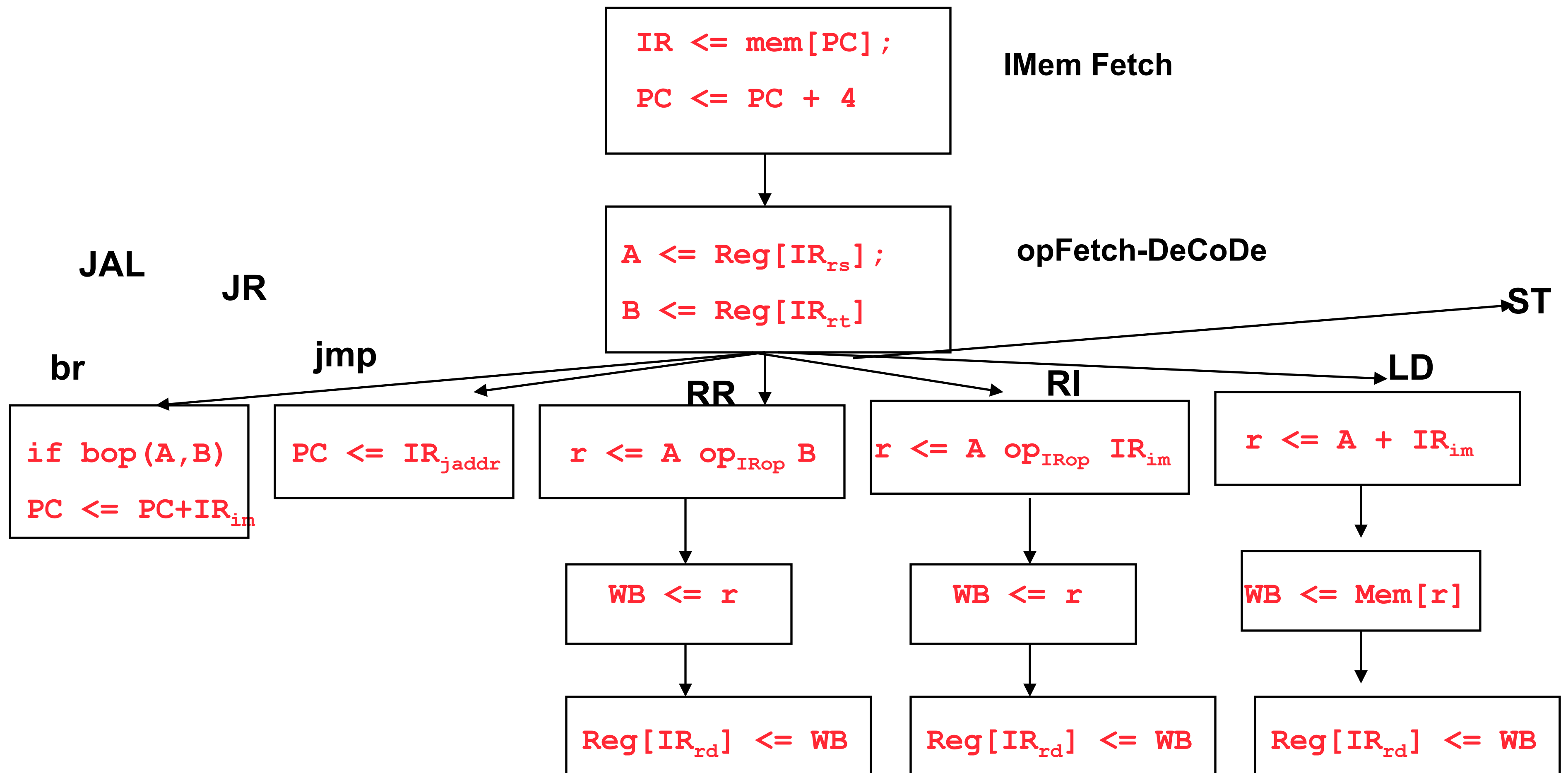

$Reg[IR_{rd}] <= (Reg[Ir_{rs}]\ op_{IRop}\ Reg[IR_{rt}])$ #op is done in stages 2-5

# 5-Stage MIPS Datapath(has pipeline latches)



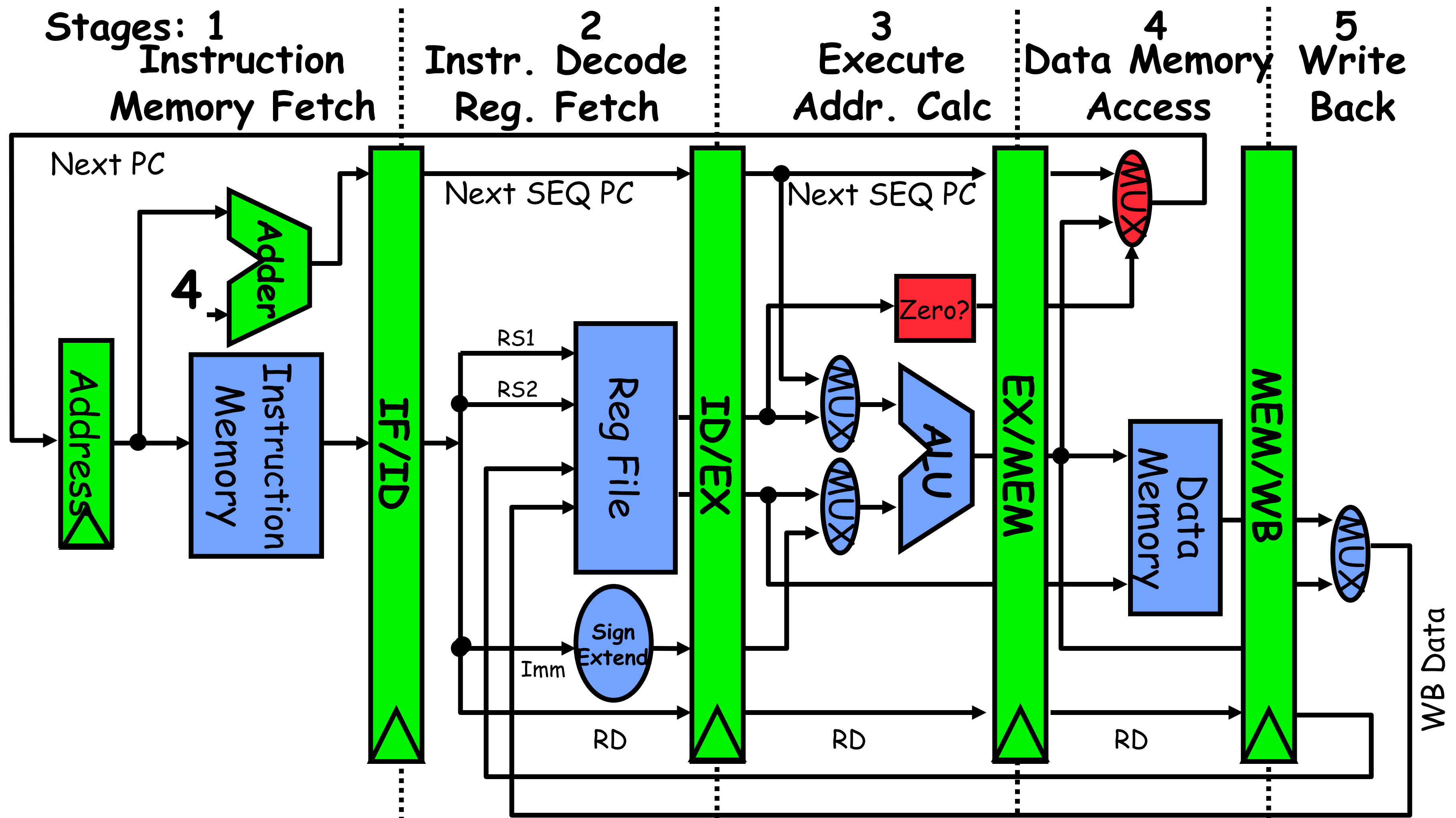
# Instruction Set Processor Controller

**Similar to Figure C.23, Page C-37**



# 5-Stage MIPS Datapath(has pipeline latches)

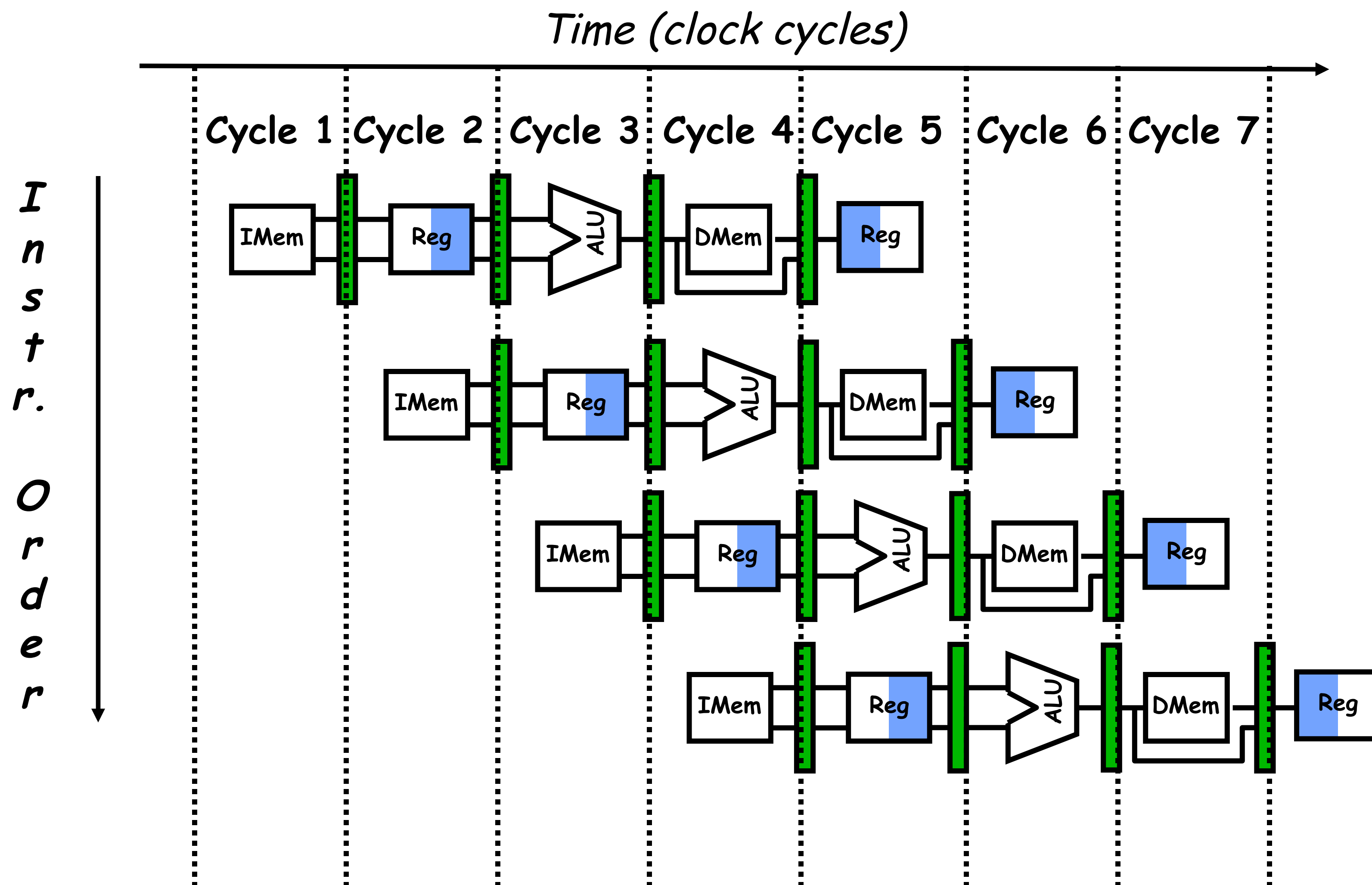
Figure C.22, Page C-35




- Data stationary control
  - local decode for each instruction phase / pipeline stage

# Visualizing Pipelining

**Figure C.3, Page C-9**

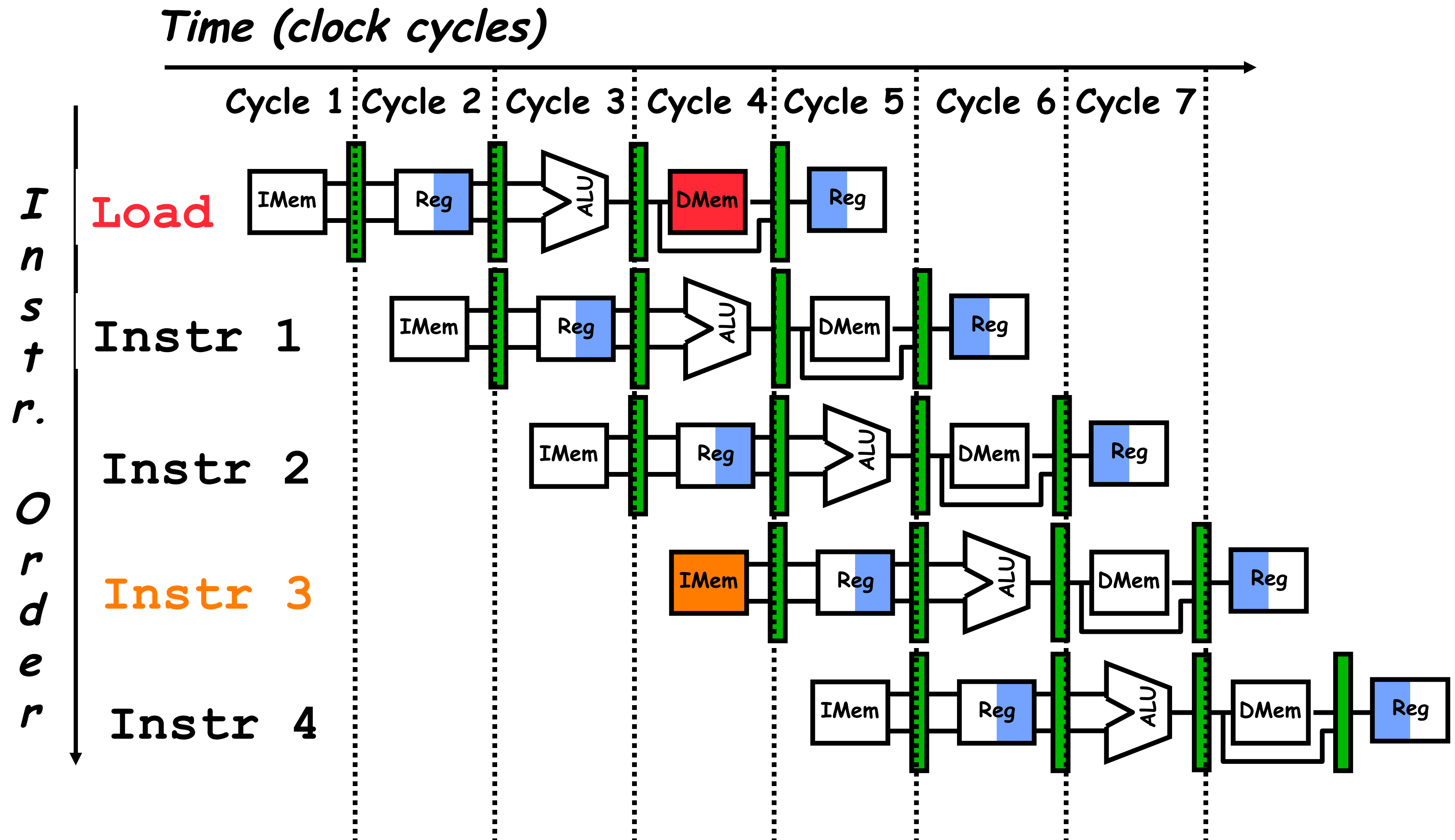


# Pipelining is not quite that easy!

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
  - **Structural hazards: HW cannot support this combination of instructions (having a single person to fold and put clothes away at same time)**
  - **Data hazards: Instruction depends on result of prior instruction still in the pipeline (having a missing sock in a later wash; cannot put away)**
  - **Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).**

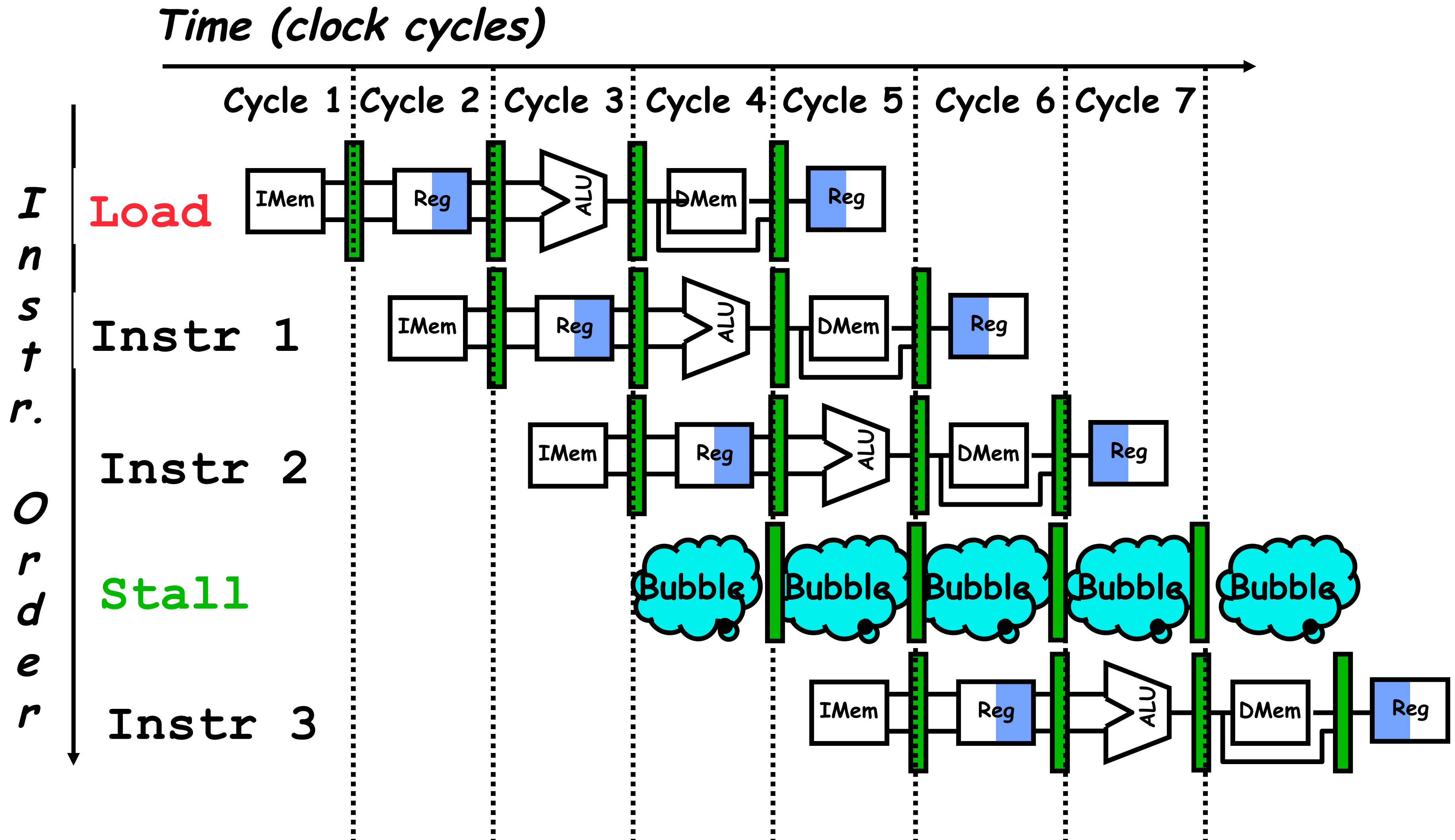
# One Memory_Port / Structural_Hazards

**Figure C.4, Page C-14**



# One Memory Port/Structural Hazards

**(Similar to Figure C.5, Page C-15)**




**How do you "bubble" the pipe?**

# Code SpeedUp Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

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

For simple RISC pipeline, Ideal CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

# Example: Dual-port vs. Single-port

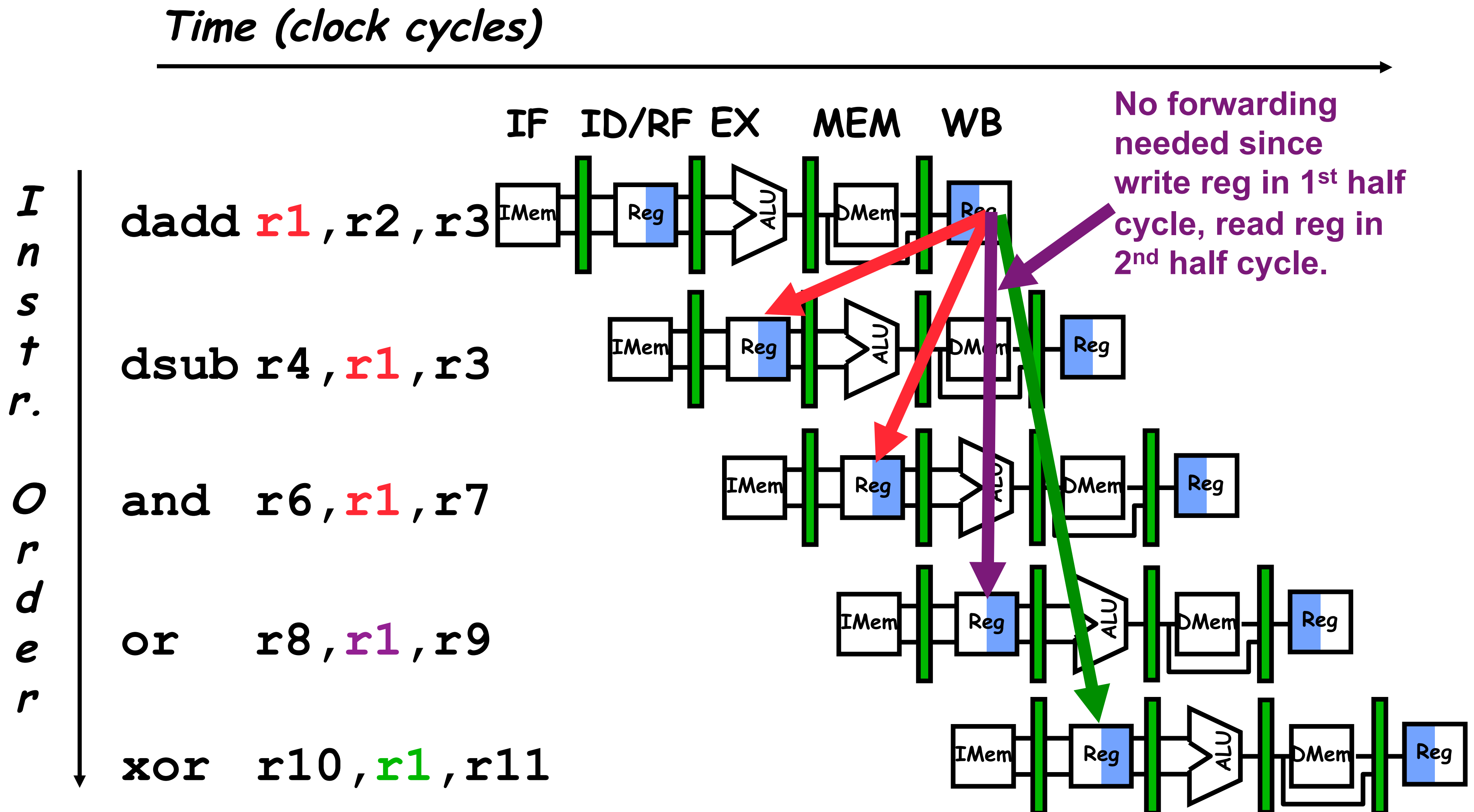
- **Machine A: Dual ported memory ("Harvard Architecture")**
- **Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate**
- **Ideal CPI = 1 for both**
- **Assume loads are 20% of instructions executed**

$$SpeedUp_A = \text{Pipeline Depth}/(1 + 0) \times (clock_{unpipe}/clock_{pipe})$$
$$= \text{Pipeline Depth}$$
$$SpeedUp_B = \text{Pipeline Depth}/(1 + 0.2 \times 1) \times (clock_{unpipe}/(clock_{unpipe} / 1.05)$$
$$= (\text{Pipeline Depth}/1.20) \times 1.05 \qquad \{105/120 = 7/8\}$$
$$= 0.875 \times \text{Pipeline Depth}$$
$$SpeedUp_A / SpeedUp_B = \text{Pipeline Depth}/(0.875 \times \text{Pipeline Depth}) = 1.14$$

- **Machine A is 1.14 times faster**

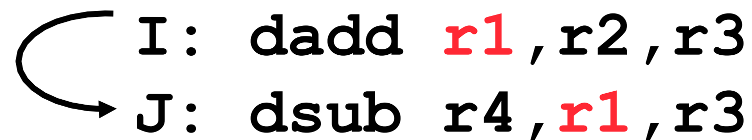
# Data Hazard on Register R1 (If No Forwarding)

Figure C.6, Page C-17



# Three Generic Data Hazards

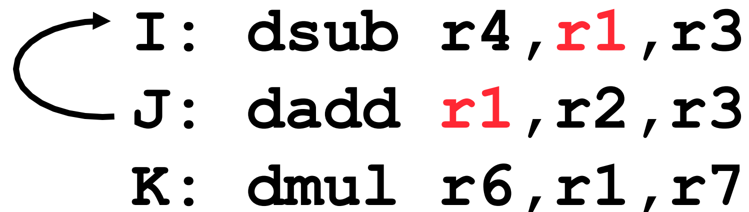
- Read After Write (RAW)
  $Instr_J$ tries to read operand before $Instr_I$ writes it




- Caused by a "(True) Dependence" (in compiler nomenclature). This hazard results from an actual need for communicating a new data value.

# Three Generic Data Hazards

- **Write After Read (WAR)**
  **$Instr_J$ writes operand *before* $Instr_I$ reads it**

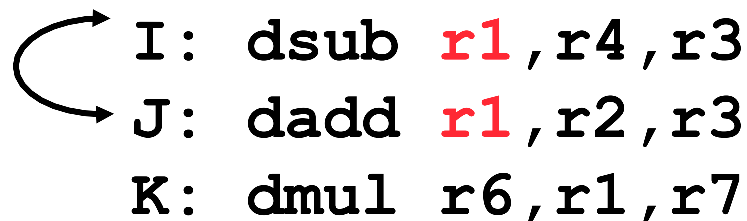



- **Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".**

- **Cannot happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Register reads are always in stage 2, and**
  - **Register writes are always in stage 5**

# Three Generic Data Hazards

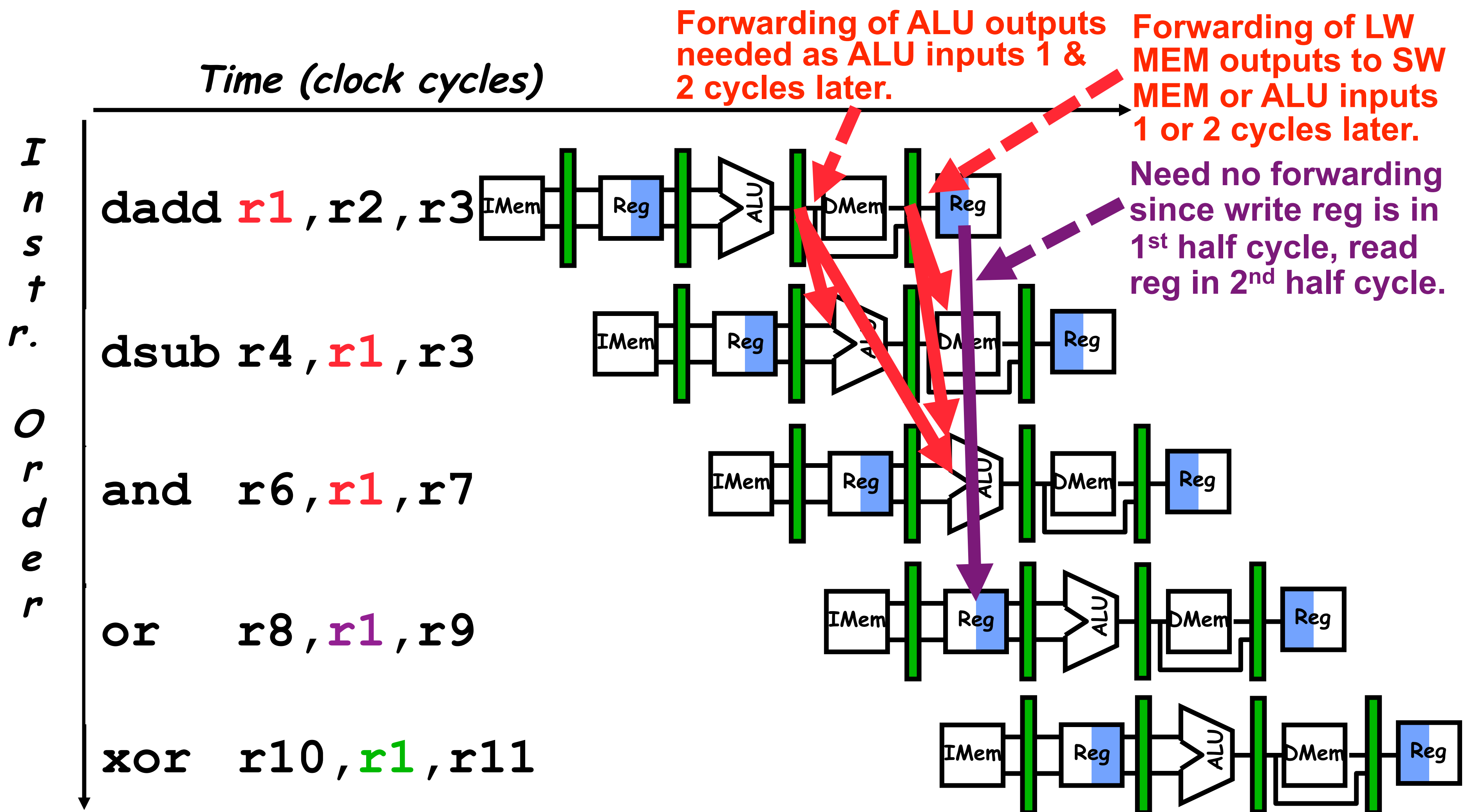
- **Write After Write (WAW)**
  **Instr$_J$ writes operand *before* Instr$_I$ writes it.**




- **Called an "output dependence" by compiler writers**
  **This also results from the reuse of name "r1".**
- **Cannot happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Register writes are always in stage 5**
- **Will see WAR and WAW in more complicated pipes**

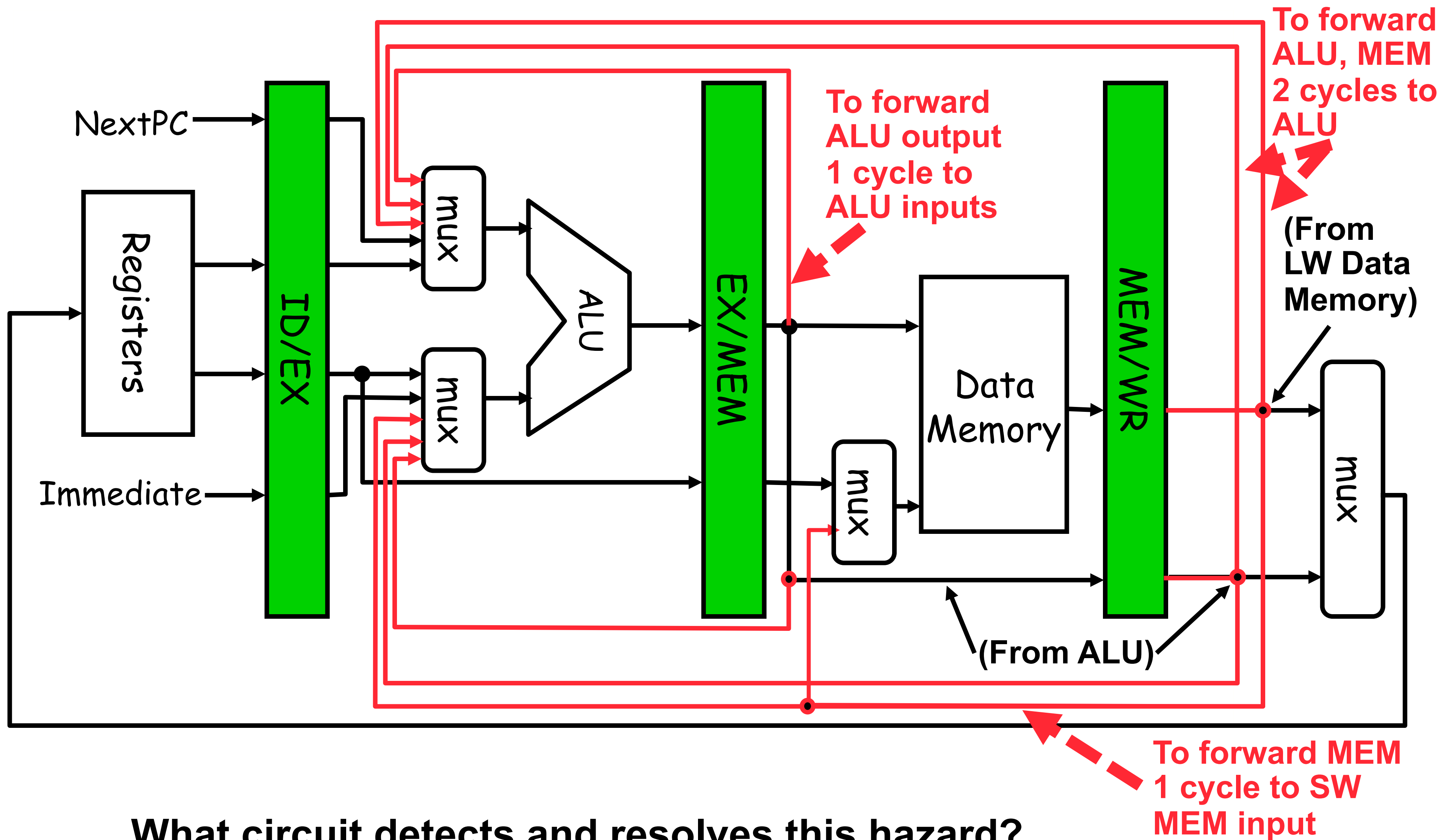
# Forwarding to Avoid Data Hazard

**Figure C.7, Page C-18 and Figure C.8, Page C-19**



# HW Datapath Changes (in red) for Forwarding

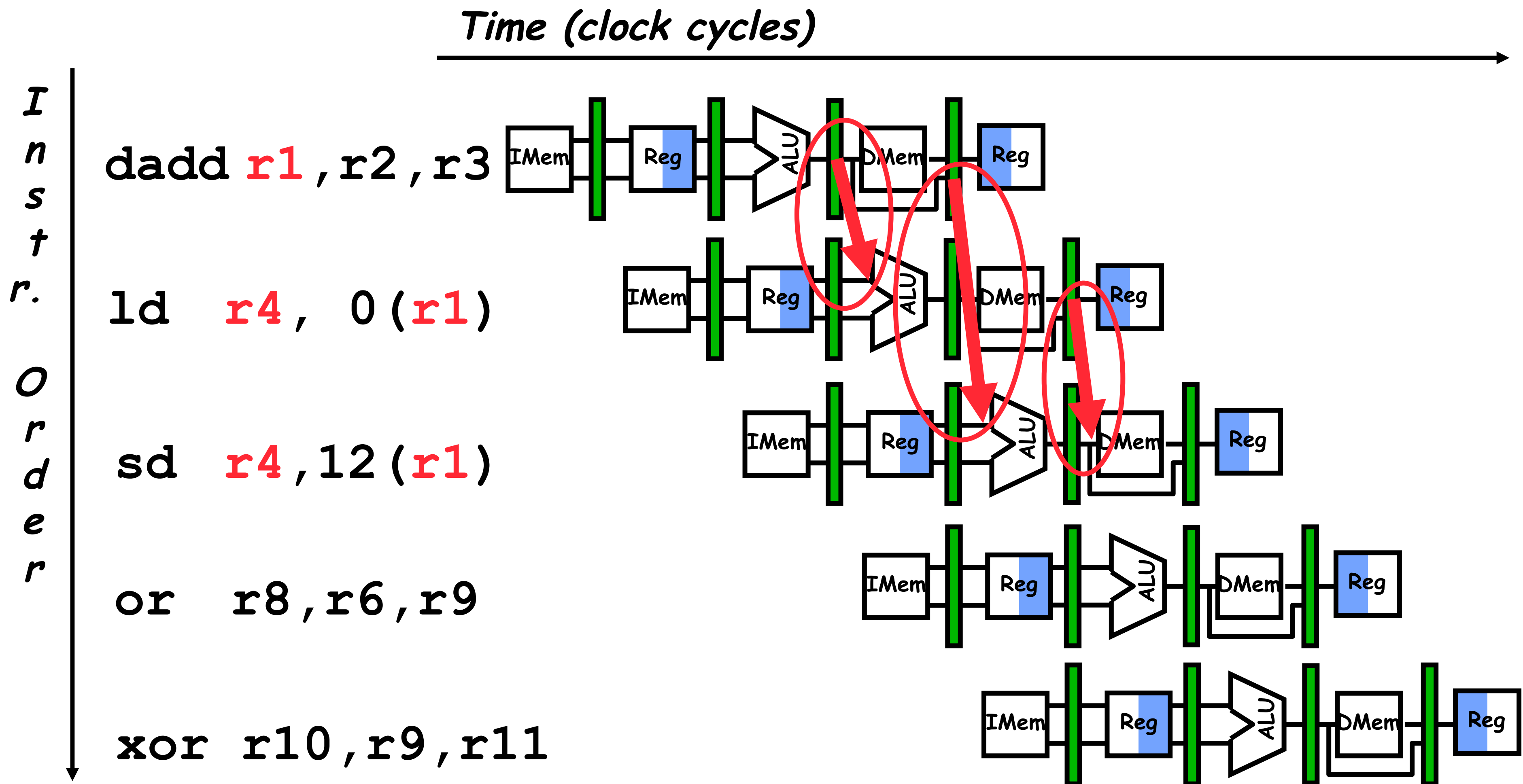
**Figure C.27, Page C-41**




**What circuit detects and resolves this hazard?**

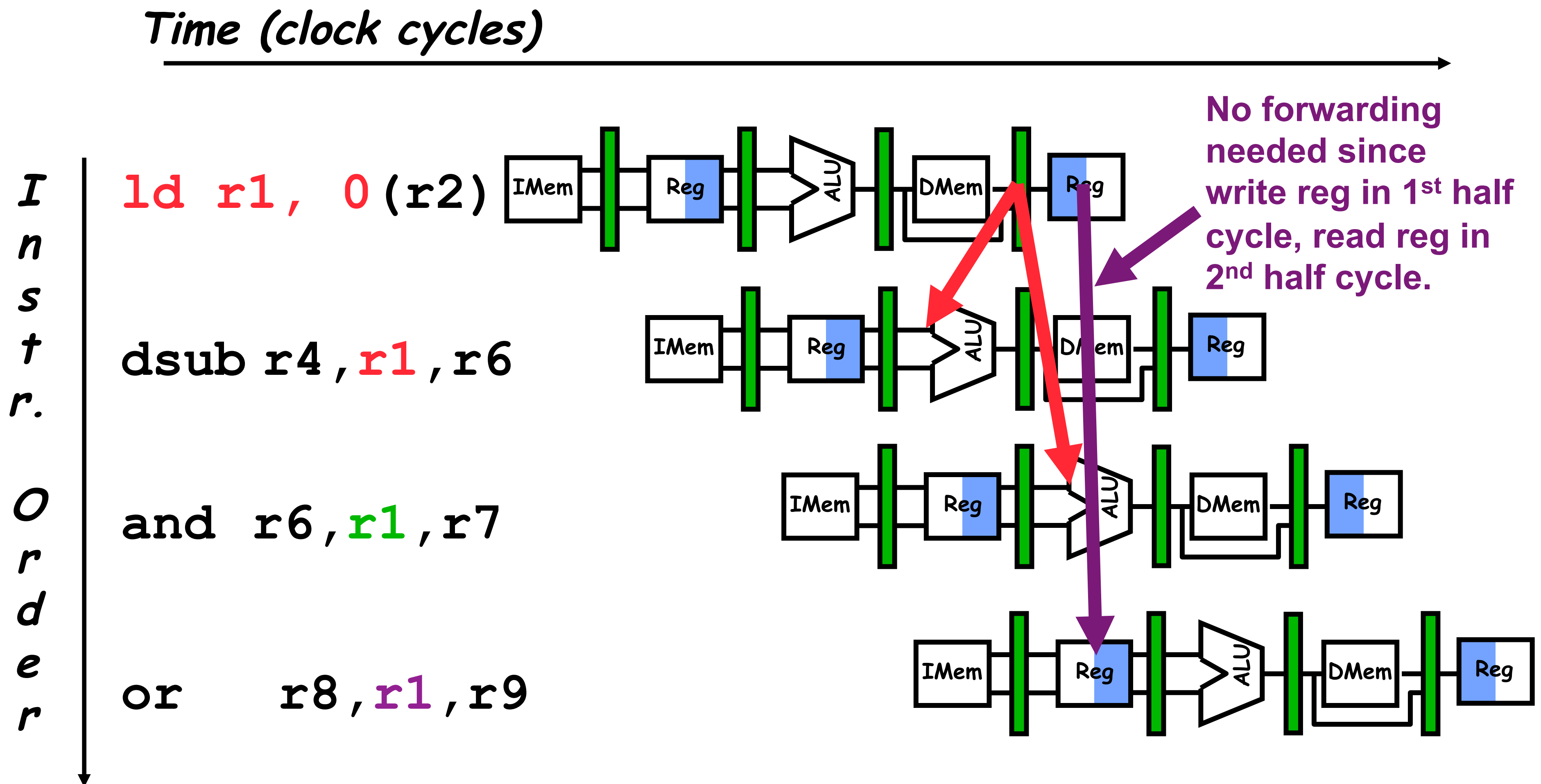
# Forwarding Avoids ALU-ALU & LD-SD Data Hazards

Similar to Figure C.8, Page C-19



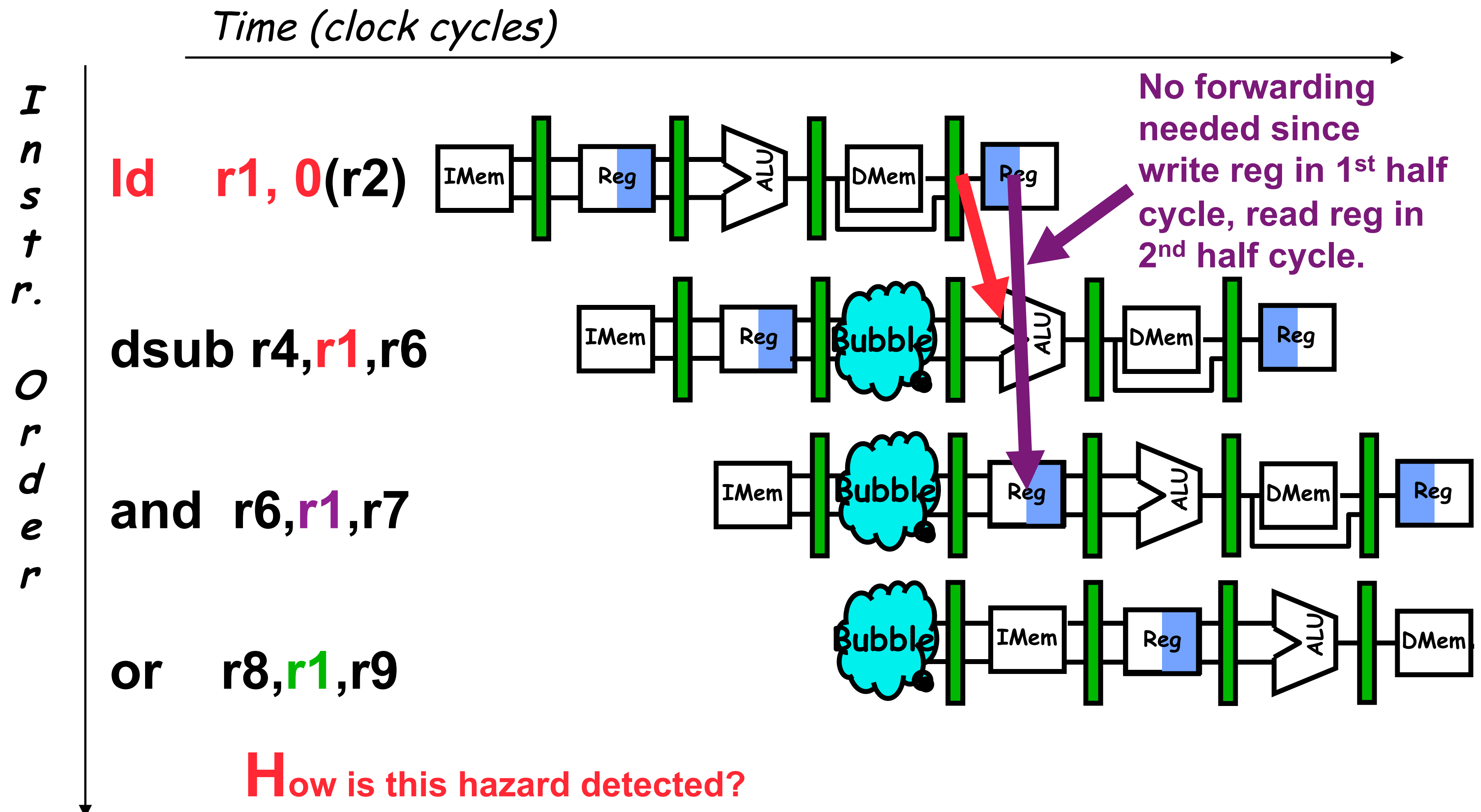
# LD-ALU Data Hazard Even with Forwarding

**Figure C.9, Page C-20**



# Data Hazard Even with Forwarding

- (Similar to Figure C.10, Page C-21)



# Software Scheduling to Avoid Load Hazards

Try producing fast code with no stalls for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f are in memory.

Slow code:

```
              LD      Rb,b
              LD      Rc,c
Stall ===>    DADD    Ra,Rb,Rc
              SD      a,Ra
              LD      Re,e
              LD      Rf,f
Stall ===>    DSUB    Rd,Re,Rf
              SD      d,Rd
```

Fast code (no stalls):

```
LD      Rb,b
LD      Rc,c
LD      Re,e
DADD    Ra,Rb,Rc
LD      Rf,f
SD      a,Ra
DSUB    Rd,Re,Rf
SD      d,Rd
```

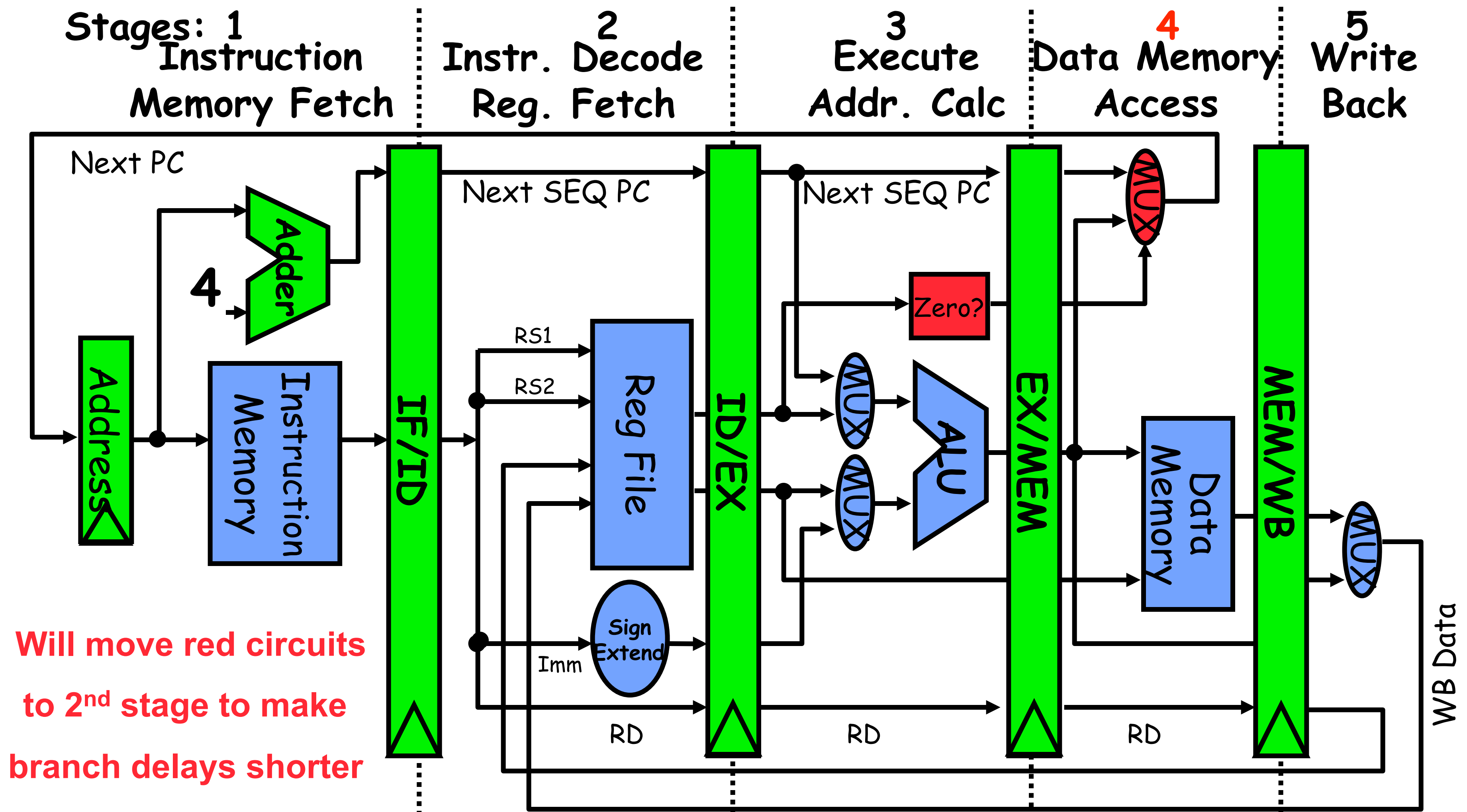
Compiler optimizes for performance. Hardware checks for safety.

Important technique ! Works since many general purpose registers.

# Outline

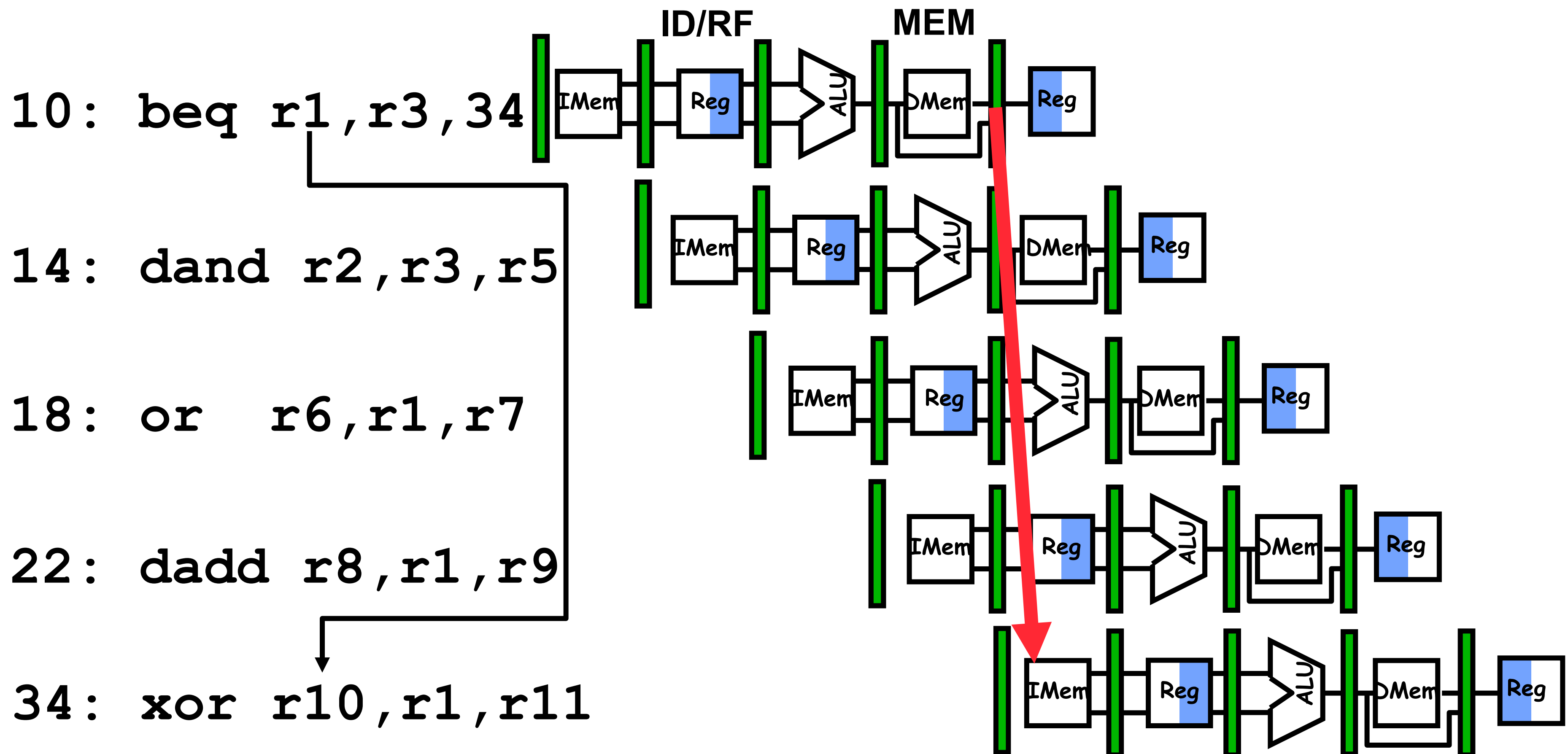
- Review
- F&P: Benchmarks age, disks fail, single-points fail
- 502 Administrivia
- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- **Branch Schemes**
- **Exceptions and Interrupts**
- **Conclusion**

# 5-Stage MIPS Datapath(has pipeline latches)




- Old simple design put branch completion in stage 4 (Mem)

# Control Hazard on Branch - Three Cycle Stall
## (Caused if Decide Branches in 4th Stage)




What can be done with the 3 instructions between beq & xor?

Code between beq&xor must not start until know beq not branch => 3 stalls

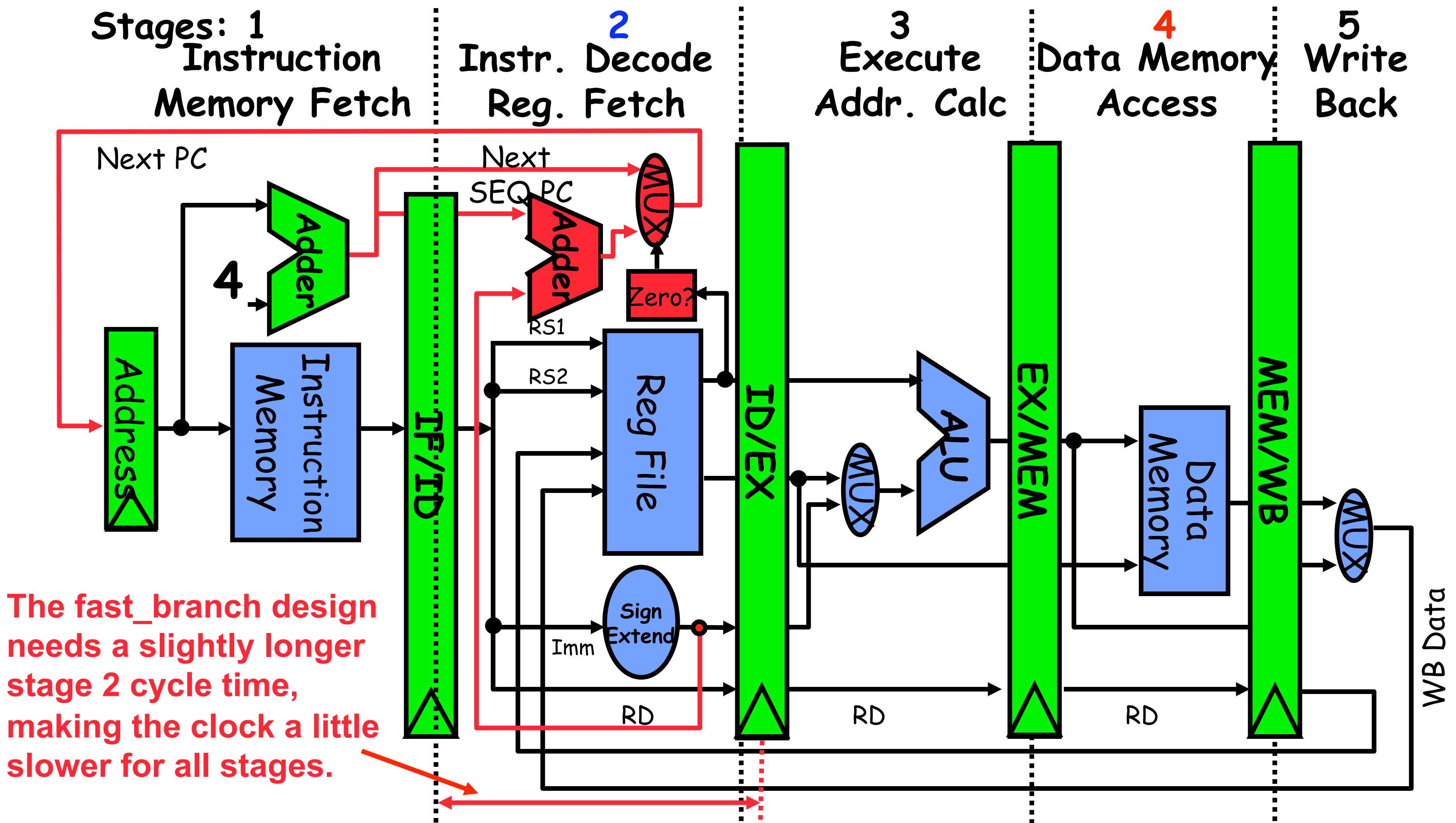
Adding 3 cycle stall after every branch (1/7 of instructions) => **CPI += 3/7. BAD!**

# Branch Stall Impact if Commit in Stage 4

- If CPI = 1 and 15% of instructions are branches,
  Stall 3 cycles => new CPI = 1.45 (1+3*.15) Too much!
- Two-part solution:
  - Determine sooner whether branch taken or not, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- Original 1986 MIPS R1000 Solution:
  - Move zero_test to ID/RF (Inst Decode & Register Fetch) stage(2) (4=MEM)
  - Add extra adder to calculate new PC (Program Counter) in ID/RF stage
  - Result is 1 clock cycle penalty for branch versus 3 when decided in MEM

# New Pipelined MIPS Datapath: Faster Branch

**Figure C.28, page C-42**




- Example of interplay of instruction set design and cycle time.

# Four Branch Hazard Alternatives – MIPS R1000

## #1: Stall until branch direction is clearly known

## #2: Predict Branch Not Taken – Easy Solution

- Execute the next instructions in sequence
- PC+4 already calculated, so use it to get next instruction
- Nullify bad instructions in pipeline if branch is actually taken
- Nullify easier since pipeline state updates are late (MEM, WB)
- 47% MIPS branches not taken on average

## #3: Predict Branch Taken

- 53% MIPS branches taken on average
- But have not calculated branch target address in MIPS
  - » MIPS still incurs 1 cycle branch penalty
  - » Some other CPUs: branch target known before outcome

## #4: Delayed Branch (Used Only in 1st MIPS "Killer Micro")

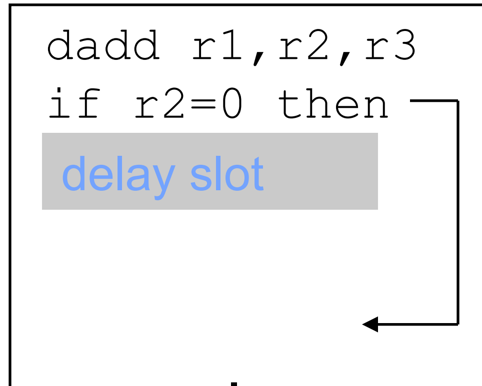
- Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  ........
  sequential successorn
branch target if taken
```

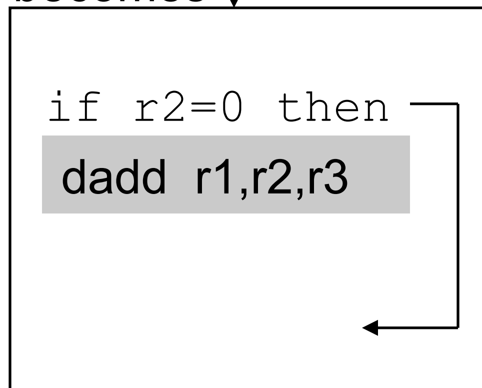
- 1 slot delay allows proper decision on branch target address in 5 stages
- MIPS R1000 used #4 (Later versions of MIPS did not; pipelines deeper)

# Scheduling Branch Delay Slots (Fig. C.14 Pg. C-24)

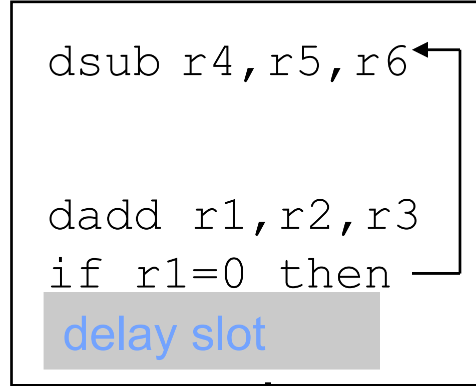
A. From before branch



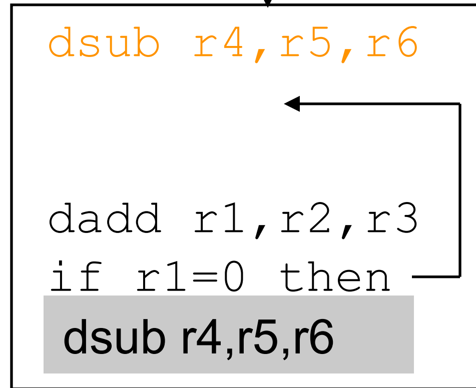

becomes



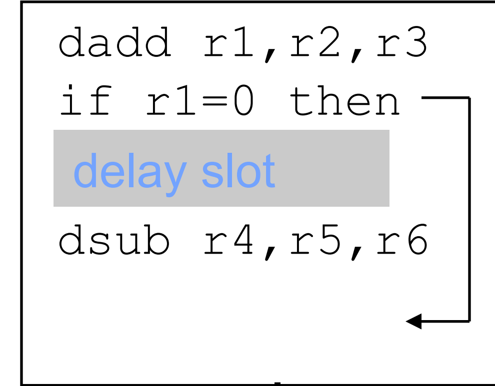

B. From branch target



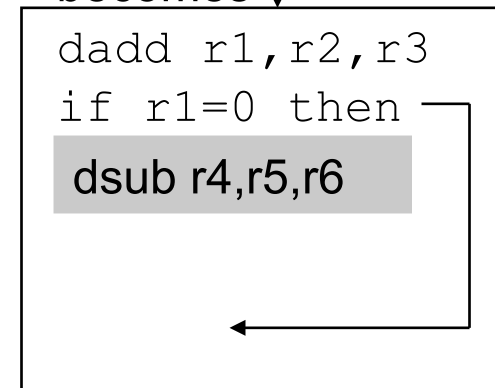

becomes




C. From fall through




becomes




The best choice, **A**, fills the delay slot & reduces instruction count (IC). In **B**, the *dsub* instruction may need to be copied, increasing IC. In **B** and **C**, an extra *dsub* must be okay when the branch fails.To help compilers fill branch delay slots, most processors with delay slots have two *canceling* branches = one for each prediction (taken, not taken). If predicted wrong, the instruction in the delay slot is treated as a *no-op*.

# Delayed Branch Not Used in Modern CPUs

Compiler effectiveness is 1/2 for a single branch delay slot:

- Fills about 60% of branch delay slots
- About 80% of instructions executed in branch delay slots are useful in computations
- Only half of (60% x 80%) slots usefully filled; cannot fill two or more

- Delayed Branch downside: As processor designs use deeper pipelines and multiple issue, the branch delay grows and needs many more delay slots
  - Delayed branching soon lost effectiveness and popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors soon permitted dynamic approaches that keep records of all branch locations, all taken/not-taken decisions, and target addresses predict branch targets with 95% or greater accuracy
  - Multi-issue 2 => 3 delay slots needed, 4 => 7 slots, 8 => 15 slots

# Eight-Stage Super-Pipeline of MIPS R4000

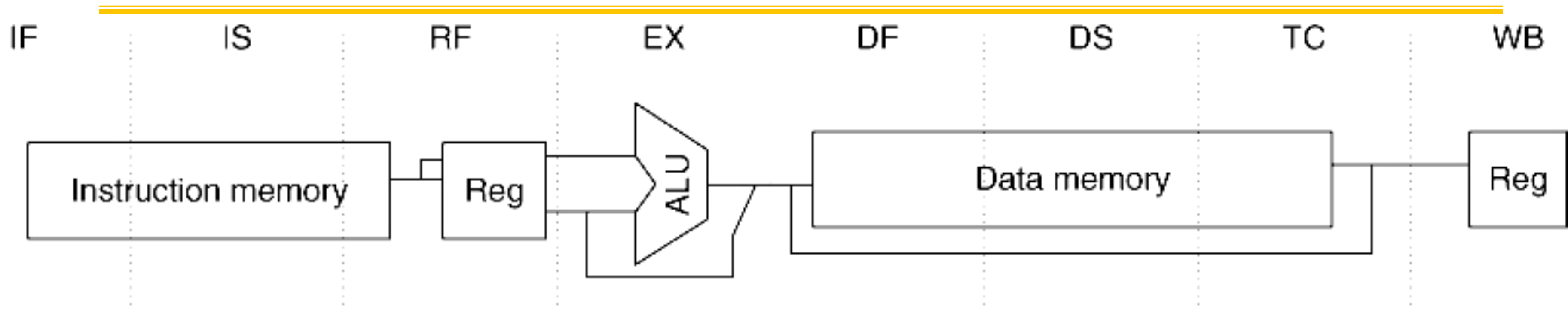



**Figure C.41** For faster clocks, the 8-stage R4000 uses pipelined instruction- and data-caches. Vertical dashed lines mark stage boundaries and pipeline latches. Each instruction is available at the end of IS, but the cache tag check is done in RF, while registers are fetched, so instruction memory is shown extending into RF. Stage TC is needed for data memory access, since MIPS cannot write data into memory or a register until it knows if the cache access was a hit (or not). The 8 R4000 stages are:

1. **IF** - First half of instruction fetch; PC selection & start of instruction cache access.
2. **IS** - Second half of instruction fetch, complete instruction cache access.
3. **RF** - Instruction decode & register fetch, plus hazard and instruction cache hit checks.
4. **EX** - Execution, which includes effective address calculation, ALU operation, or branch-target computation and condition evaluation.
5. **DF** - Data fetch, first half of data cache access.
6. **DS** - Second half of data fetch, completion of data cache access.
7. **TC** - Tag check, to determine whether the data cache access hit.
8. **WB** - Write-back value to register for loads and register-register operations.


© 2012, Elsevier Inc.

# Three-Cycle Branch Delay for MIPS R4000

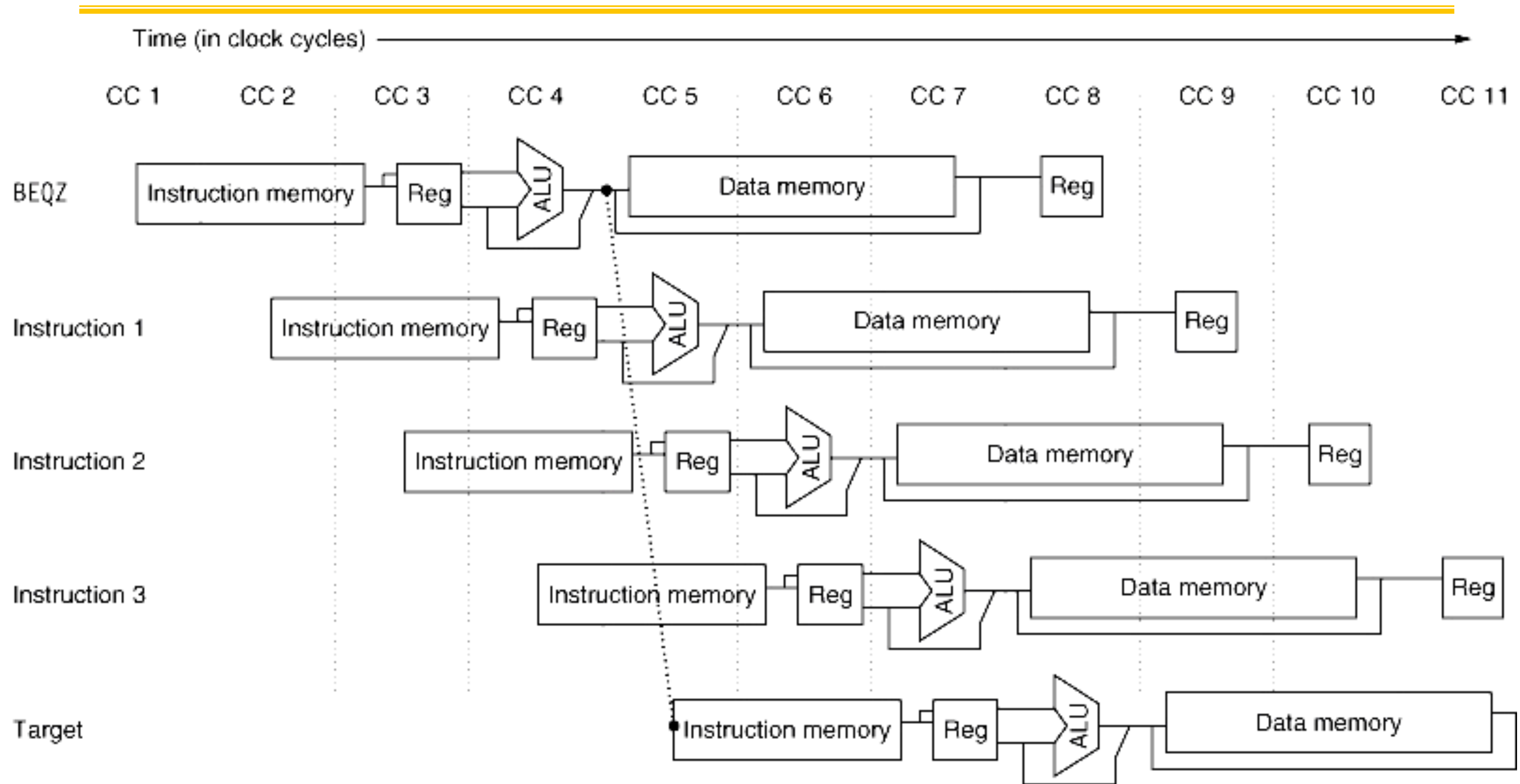



**Figure C.44** The basic branch delay is 3 cycles, since the condition evaluation is performed at the end of the EX stage.


© 2012, Elsevier Inc.

# Evaluating Branch Alternatives for 1st MIPS

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

**Assume 4% unconditional jump, 10% conditional branch-taken, 6% conditional branch-not-taken, base CPI = 1.**

| *Scheduling Scheme* | *Branch penalty* | *CPI* | *speedup vs. no-pipe 5 cycles* | *speedup vs. flush_pipeline* |
|---|---|---|---|---|
| **Flush pipeline** (Stage 4) | 3 | 1.60 | 3.1 | 1.00 |
| **Predict taken** (Stage 2) | 1 | 1.20 | 4.2 | 1.33 |
| **Predict not taken** (St.2) | 1 | 1.14 | 4.4 | 1.40 |
| **Delayed branch** (Stg 2) | 0.5 | 1.10 | 4.5 | 1.45 |

**(Sample calcu- lations)**

- **1.60=1+3(4+10+6)%** **(4.5=5/1.10)** **(1.45=1.6/1.1)**
- **1.20=1+1(4+10+6)%** **(to calculate taken target)**
- **1.14=1+1(4+10)%** **(refetch for jump, taken-branch)**

# Evaluating Branch Alternatives – MIPS R4000

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

The deeper MIPS R4000 pipeline takes at least three pipeline stages before the branch-target address is known. A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in Figure C.I5.

| Branch scheme | Penalty unconditional | Penalty untaken | Penalty taken |
|---|---|---|---|
| Flush pipeline | 2 | 3 | 3 |
| Predicted taken | 2 | 3 | 2 |
| Predicted untaken | 2 | 0 | 3 |

**Figure C.15** Branch penalties for three simple prediction schemes for a deeper pipeline. Unconditional branch targets are easily known by end of decode, CC 3.

| | Additions to the CPI from branch costs | | | |
|---|---|---|---|---|
| Static Branch scheme | Unconditional branches | Untaken conditional branches | Taken conditional branches | All branches |
| Frequency of event | 4% | 6% | 10% | 20% |
| Stall pipeline | 0.08 | 0.18 | 0.30 | 0.56 |
| Predicted taken | 0.08 | 0.18 | 0.20 | 0.46 |
| Predicted untaken | 0.08 | 0.00 | 0.30 | 0.38 |

**Figure C.16** CPI penalties for three branch-prediction schemes and deeper R4000 pipeline. Last entries are row sums. All other entries are Frequency_of_event **x** penalty from Figure C.15. CPI = 1 is 1.56 times faster than CPI=(1+0.56). CPI = 1.38 is 1.13 **X** faster than CPI = 1.56 .


© 2012, Elsevier Inc.

# MIPS R4000 CPI for 10 SPEC92 Codes

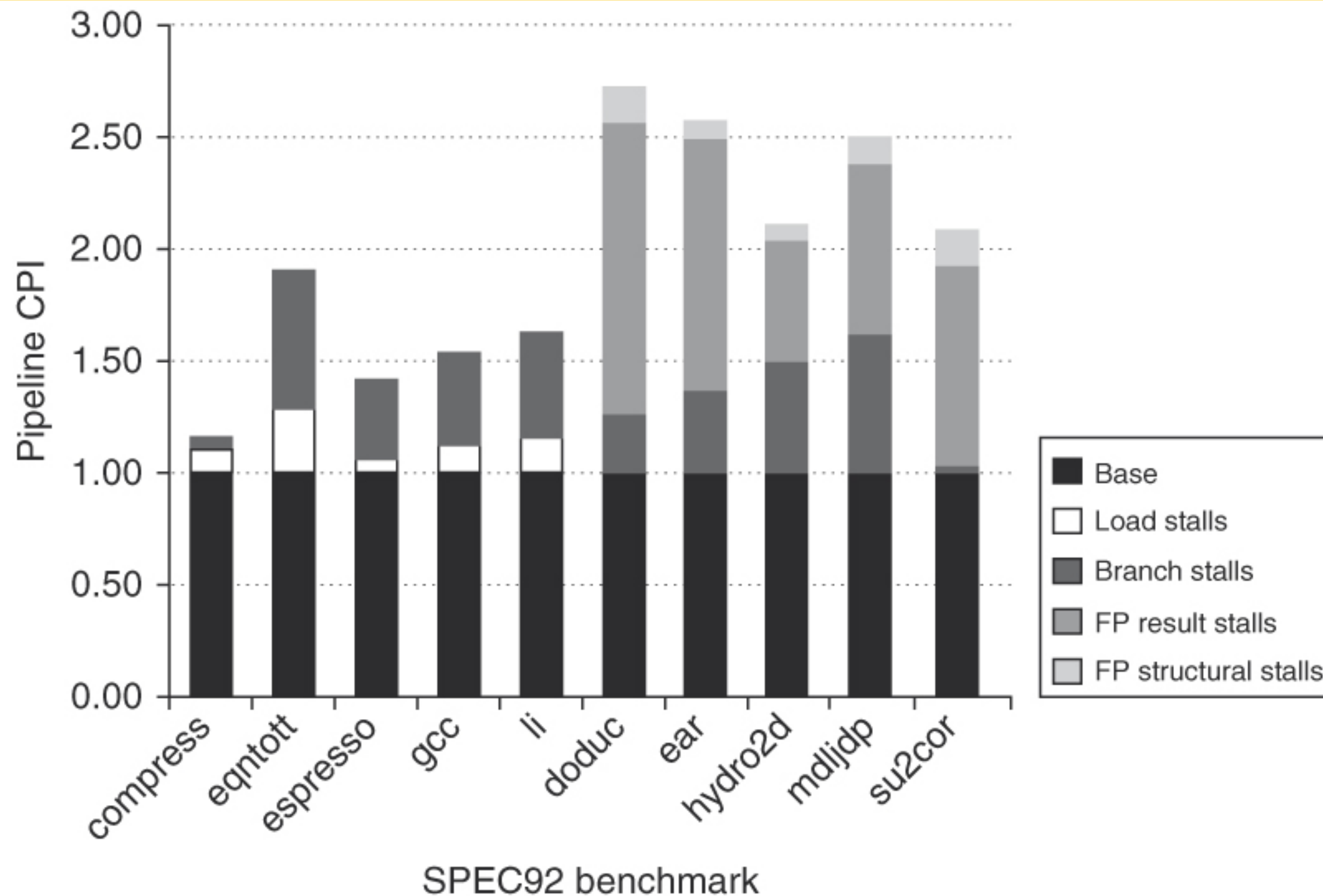



**Figure C.52** The MIPS R4000 pipeline CPI for 10 SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The leftmost five programs are integer programs, and branch delays are the major CPI contributor for these. The rightmost five programs are FP, and FP result stalls are their major contributor.

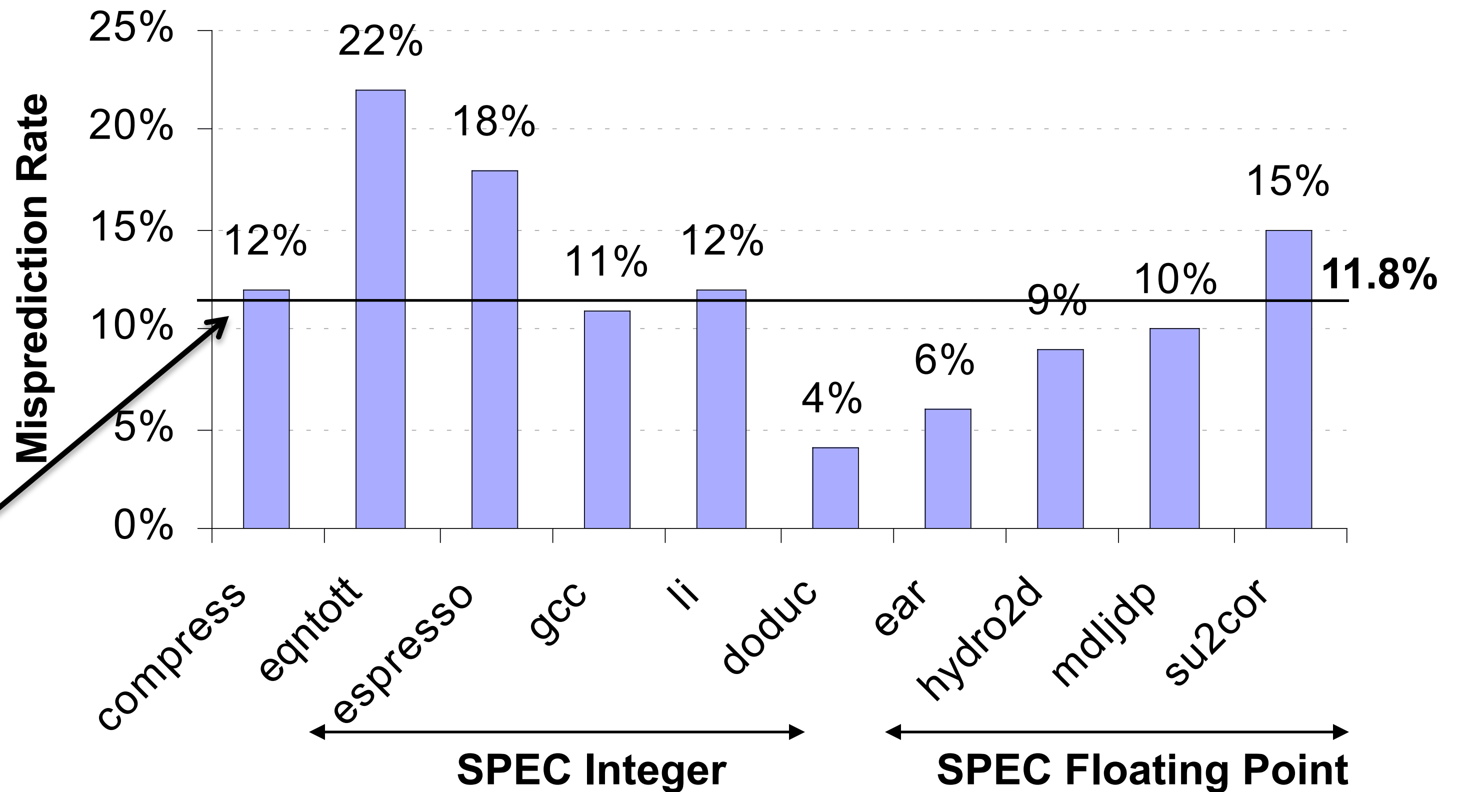


© 2012, Elsevier Inc.

# Static (Compile-Time) Branch Prediction

- An earlier slide showed scheduling code into a branch delay slot
- To reorder ("move") code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% in SPEC benchmarks

- A more accurate scheme predicts branches using profile information collected from earlier runs and modifies predictions based on last run:

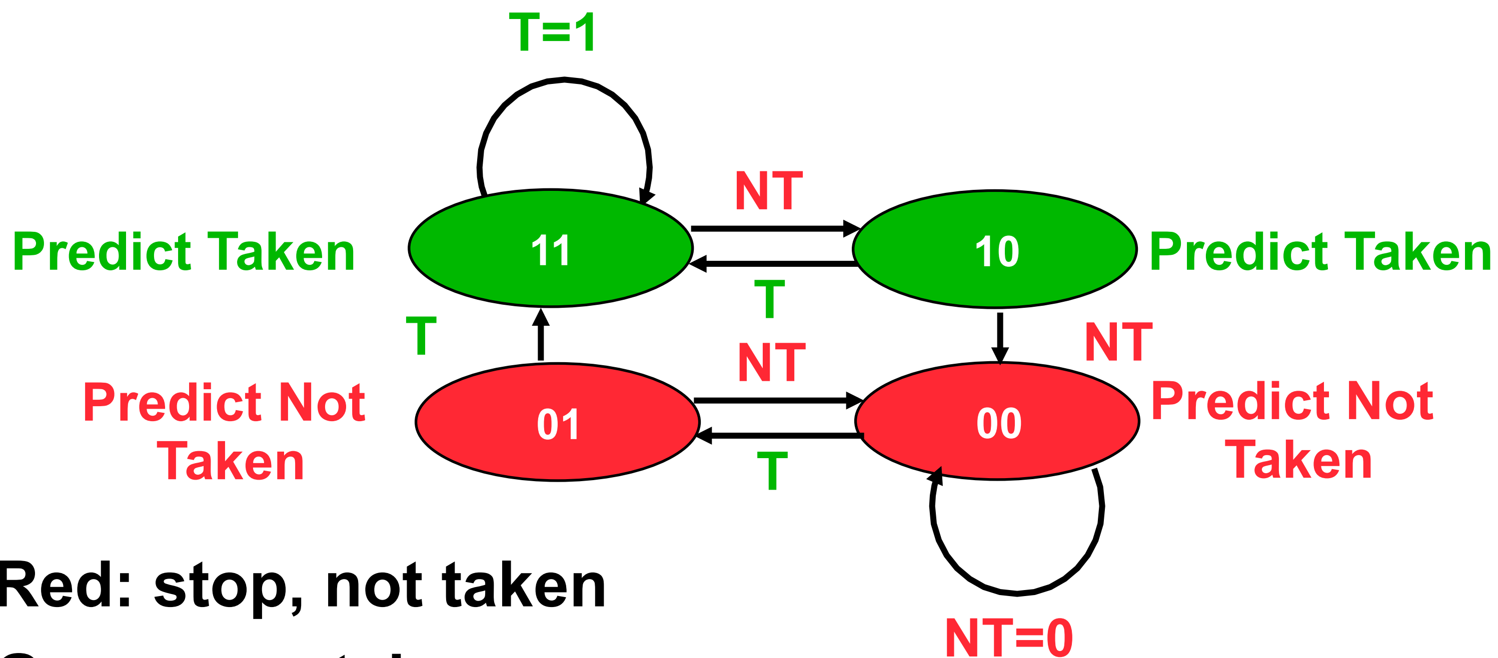


# Dynamic (Run-Time) Branch Prediction

- **Why does prediction work?**
  - **Underlying algorithm has regularities**
  - **Data that is being operated on has regularities**
  - **Instruction sequences have redundancies that are artifacts of way that humans and compilers solve problems**
- **Is dynamic branch prediction better than static prediction?**
  - **Seems to be**
  - **There are a small number of important branches in programs which have dynamic behavior**
- **Performance = *ƒ*(accuracy, cost of misprediction)**
- **Branch History Table:** Lower bits of PC address index a table of 1-bit values
  - **Says whether or not branch taken last time**
  - **No address check**
- **Problem: 1-bit BHT will cause two mispredictions per loop, (Average for loops is 9 iterations before exit):**
  - **End of loop case, when it exits instead of looping as before**
  - **First time through loop on *next* time through code, when it predicts exit instead of looping**

# Dynamic Branch Prediction With 2 Bits

- **Solution: 2-bit scheme where change prediction only if get misprediction *twice* in a row:**




- **Red: stop, not taken**
- **Green: go, taken**
- **Adds *hysteresis* to decision making process**

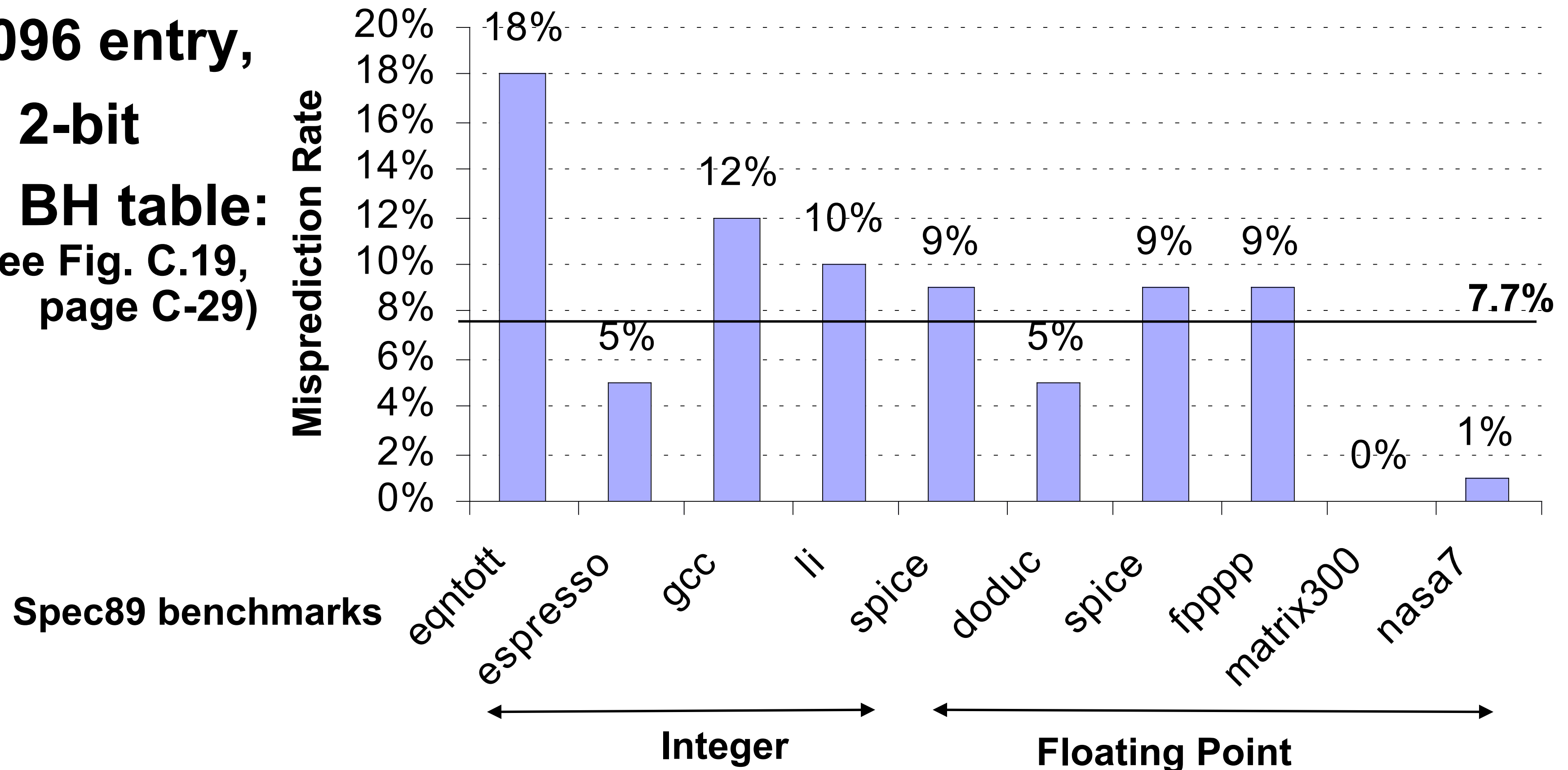
# Branch History Table (BHT) Accuracy

**Mispredict because either:**

- **Make wrong guess for that branch**
- **Got branch history of wrong branch when indexed the table (same low address bits used for index).**

**4096 entry, 2-bit BH table:**

**(See Fig. C.19, page C-29)**



# Very Accurate 4K-Entry Branch History Table

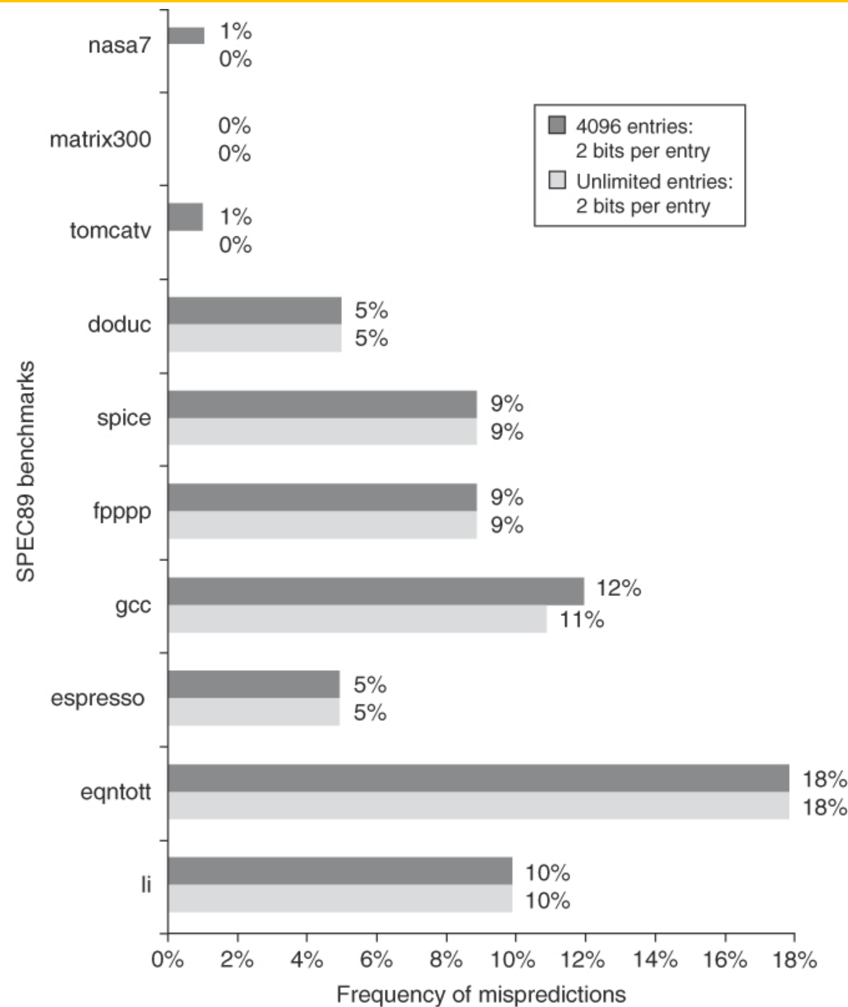



**Figure C.20 Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although these data are for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with at most 8K entries needed to match an infinite 2-bit predictor.**

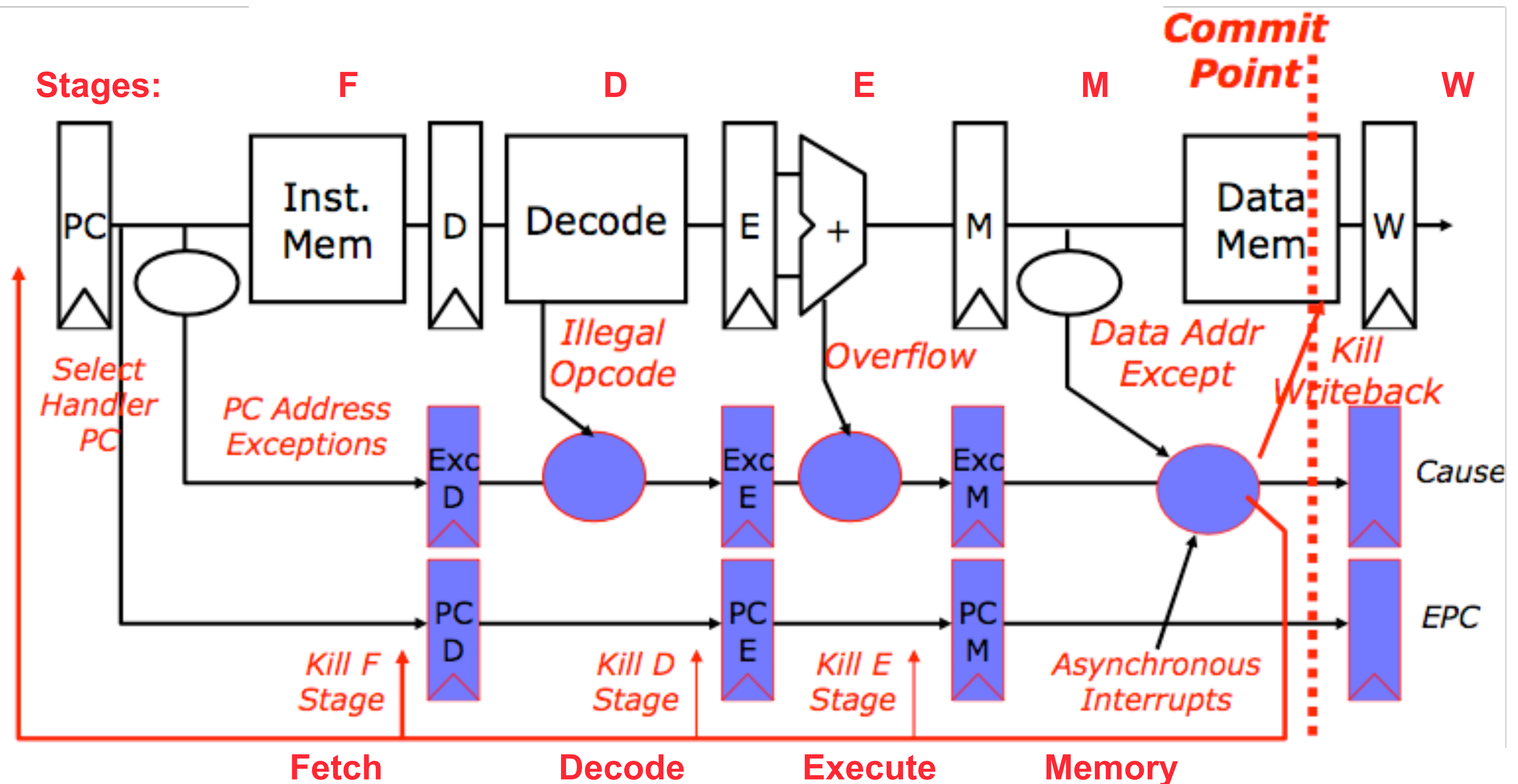
 © 2012, Elsevier Inc. 

# Another Problem for Pipelining - Interrupts

- **Exception: An unusual event happens to an instruction during its execution {caused by instructions executing}**
  - **Examples: divide by zero, undefined opcode**
- **Interrupt: Hardware signal to switch the processor to a new instruction stream {not directly caused by code}**
  - **Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)**

- **Precise Interrupt Problem: Must seem as if the exception or interrupt appeared between 2 instructions ($I_i$ and $I_{i+1}$) although several instructions were executing at the time**
  - **All instructions up to and including $I_i$ are totally completed**
  - **No effect of any instruction after $I_i$ is allowed to be saved**
- **After a precise interrupt, the interrupt (exception) handler either aborts the program or restarts at instruction $I_{i+1}$**

# Precise Exceptions in Static Pipelines




**Key observation: "Architected" states change only in memory (M) and register write (W) stages.**

# And In Conclusion: Control and Pipelining

- Quantify and summarize performance
  - Ratios to "VAX", Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, single-point failure
- Control via State Machines and Microprogramming
- Pipelines just overlap tasks - easy for independent tasks
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers by stalling:
  - Structural: need more HW resources
  - Data (RAR, RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch or branch (taken/not-taken) prediction
- Exceptions and interrupts add complexity
- For next time: Read Appendix B (Memory Caches).

# Unused Slides Spr'12

# Two-Cycle LD-ALU Delay for MIPS R4000

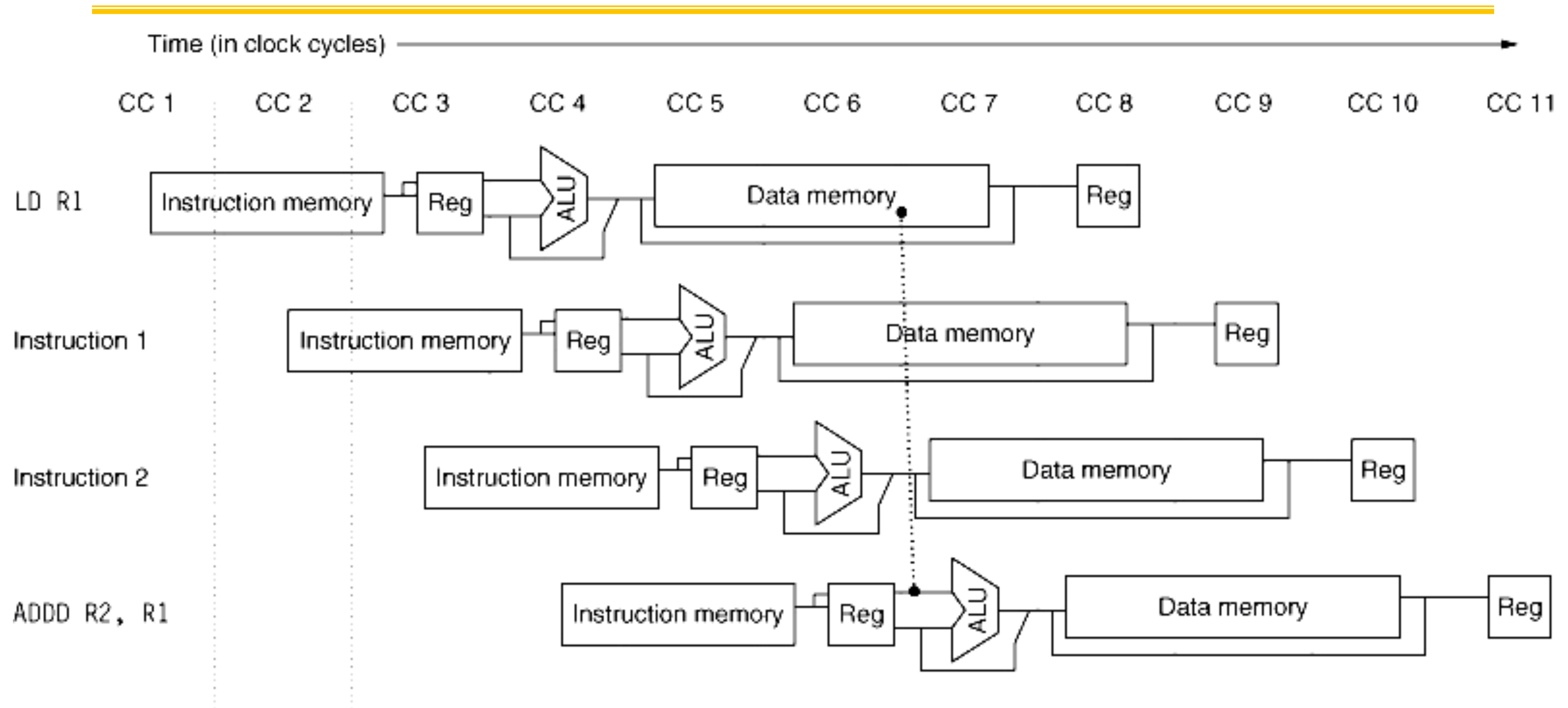



**Figure C.42** The structure of the R4000 integer pipeline permits a 2-cycle load-use delay. A delay of 2 cycles, not 3 cycles, is possible because the data value from memory is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up one cycle, when the correct data will become available.

 © 2012, Elsevier Inc. 