Pipelining: Basic and Intermediate Concepts

Professors Hugh C. Lauer and Thérèse Smith
CS-4515, System Programming Concepts

Pedagogical Dilemma

- **Pipelining first, followed by Memory Hierarchy and Caching**
    - *or*
- **Memory Hierarchy and Caching first, followed by Pipelining**

From OS Course

- **Three models of parallelism in computing…**
  - Data parallelism
  - Task parallelism
  - Pipelining

- **… plus one new one**
  - Google – massive parallelism (warehouse scale)

From OS Course

- **Three models of parallelism in computing…**
  - Data parallelism
  - Task parallelism
  - Pipelining

- **… plus one new one**
  - Google – massive parallelism (warehouse scale)

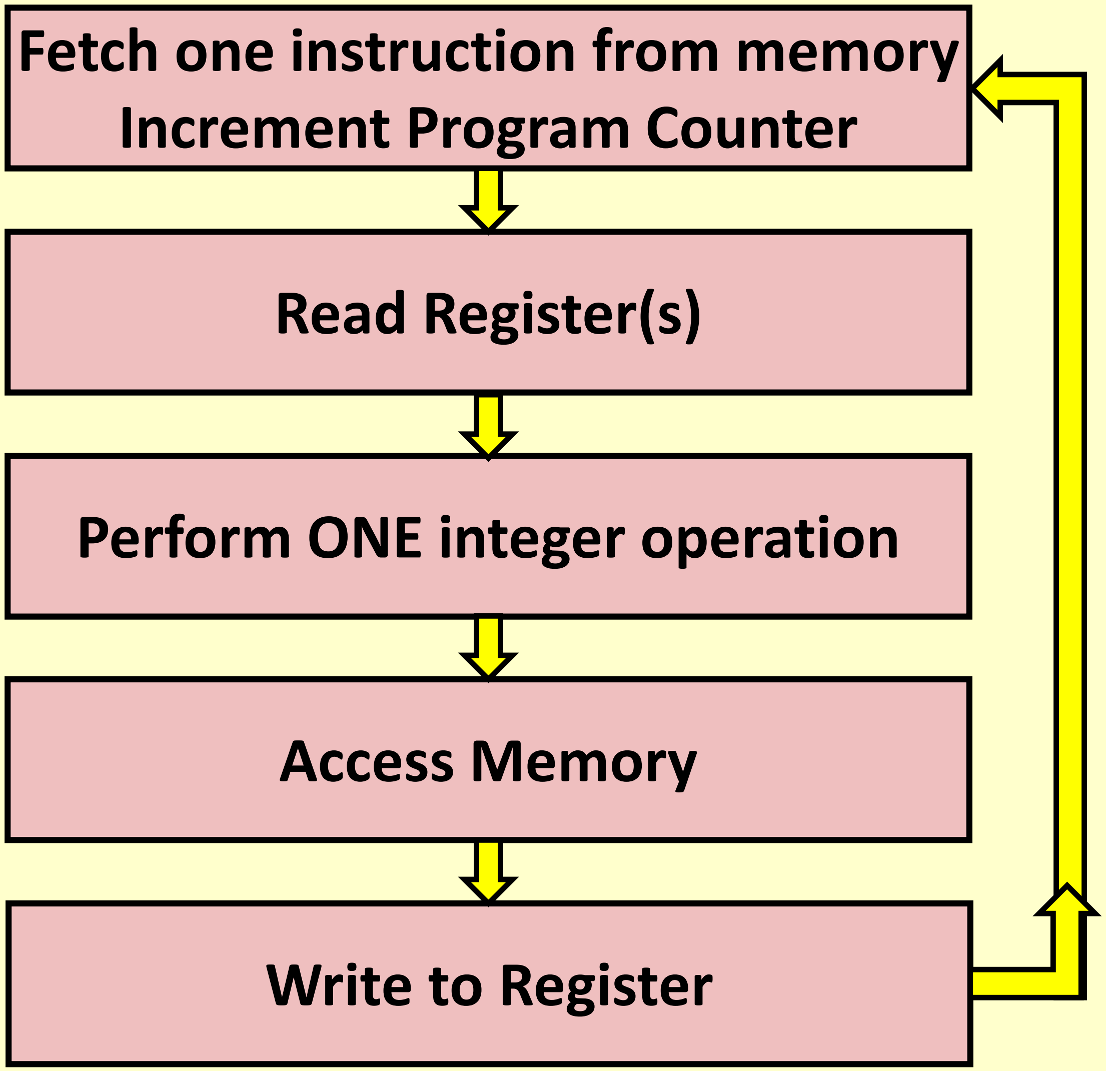
Recall von Neumann model

Five separate steps (phases)

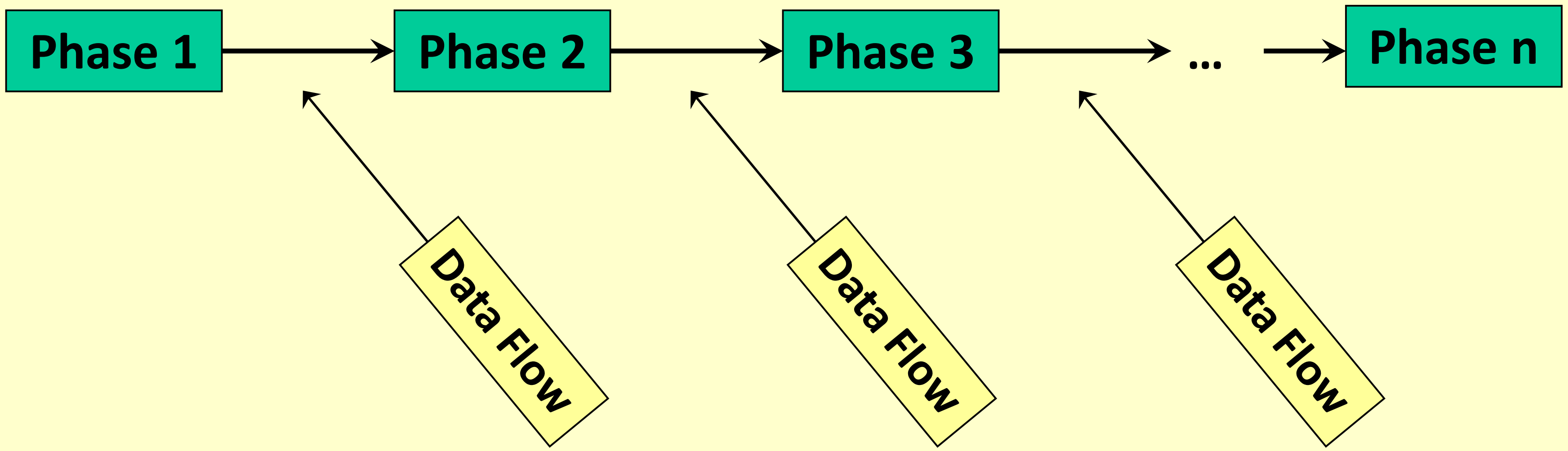
Each phase independent of others …

… requiring different circuitry

Data flows btw phases

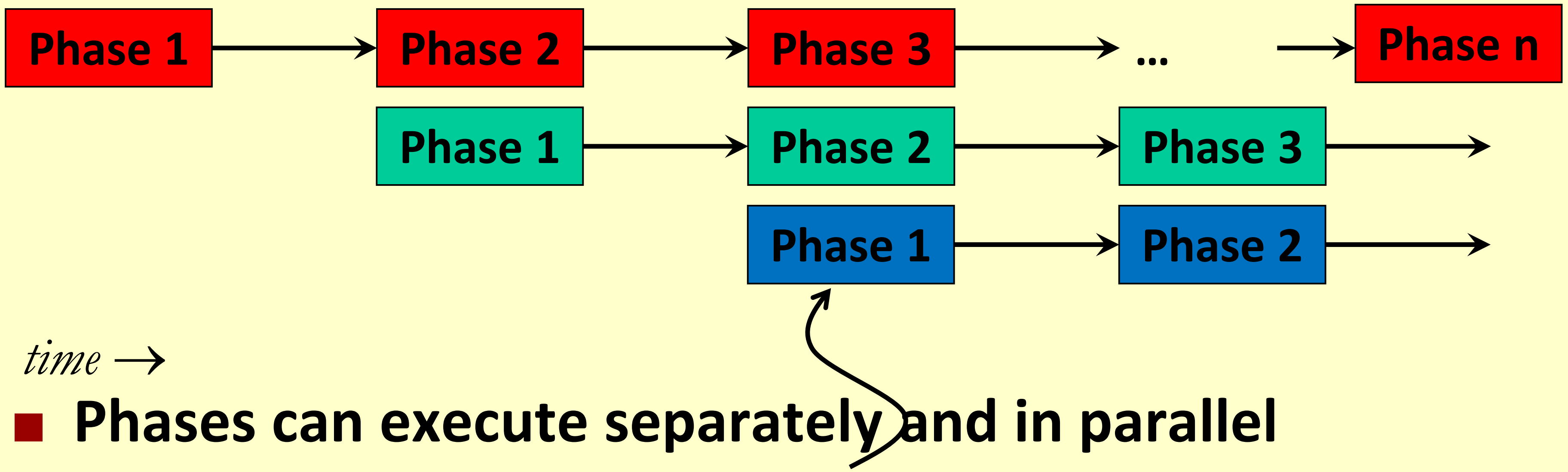

Fetch one instruction from memory
Increment Program Counter
Read Register(s)
Perform ONE integer operation
Access Memory
Write to Register

Lay out horizontally on time line



- **Assume phases do not share resources**
  - Except data flow between them
- **Observation:–**
  - With careful planning, these phases can be pipelined!

To execute an instruction



- **Phases can execute separately and in parallel**
  - I.e., Blue instruction is in Phase/Step 1 …
  - … while Green instruction is in Phase/Step 2 …
  - … … while Red instruction is in Phase/Step 3
  - … … … *etc.*
  - *All at the same time!*

Computer Architecture

- **This model applies to executing instructions within a processor**
  - CPUs
  - Graphics
- **(Nearly) all modern processors *pipelined* at instruction level since mid-1980's**
  - Some specialized computers were pipelined long before
- **RISC instruction sets designed specifically to accommodate pipelining**
  - And to work hand-in-hand with modern compilers

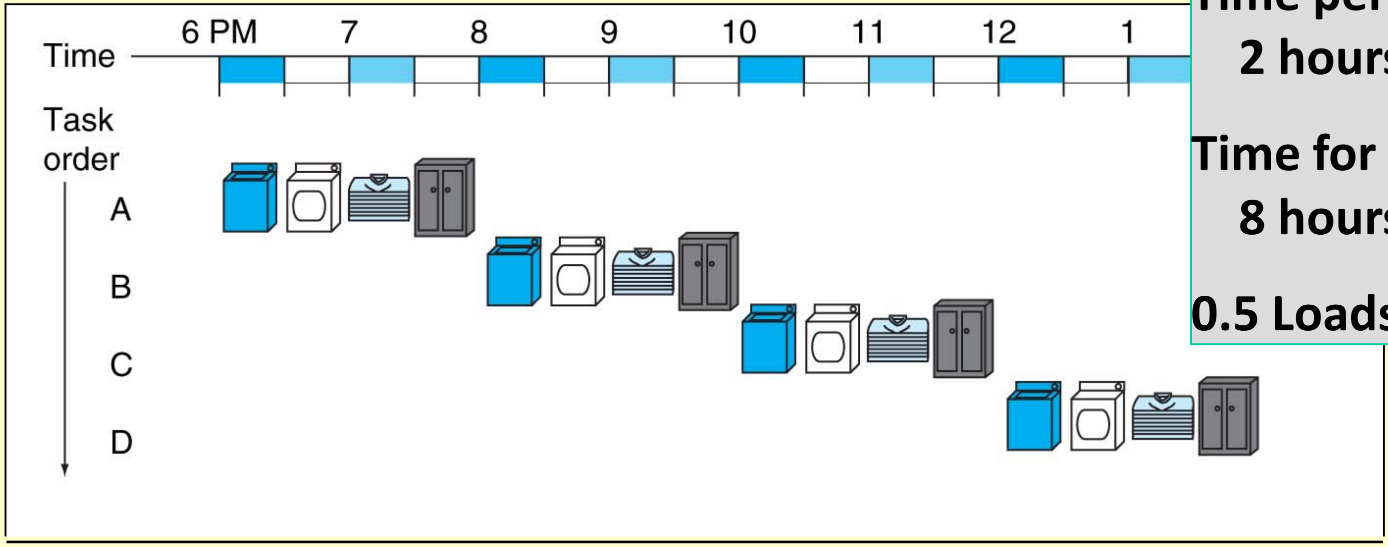
Reading Assignment

- **Appendix C – *Pipelining: Basic and Intermediate Concepts***
  - Especially §C.4-C.6
- **Also re-read/review §1.8 of Chapter 1, especially *Processor Performance Equation*, pp. 49–51**
  - Understand what is meant by *CPI*
  - I.e., Cycles per Instruction

Pipelining is also discussed in Byrant & O'Hallaron, §§4.6–4.5

Not included in custom version of textbook for WPI

Clothes-washing Analogy

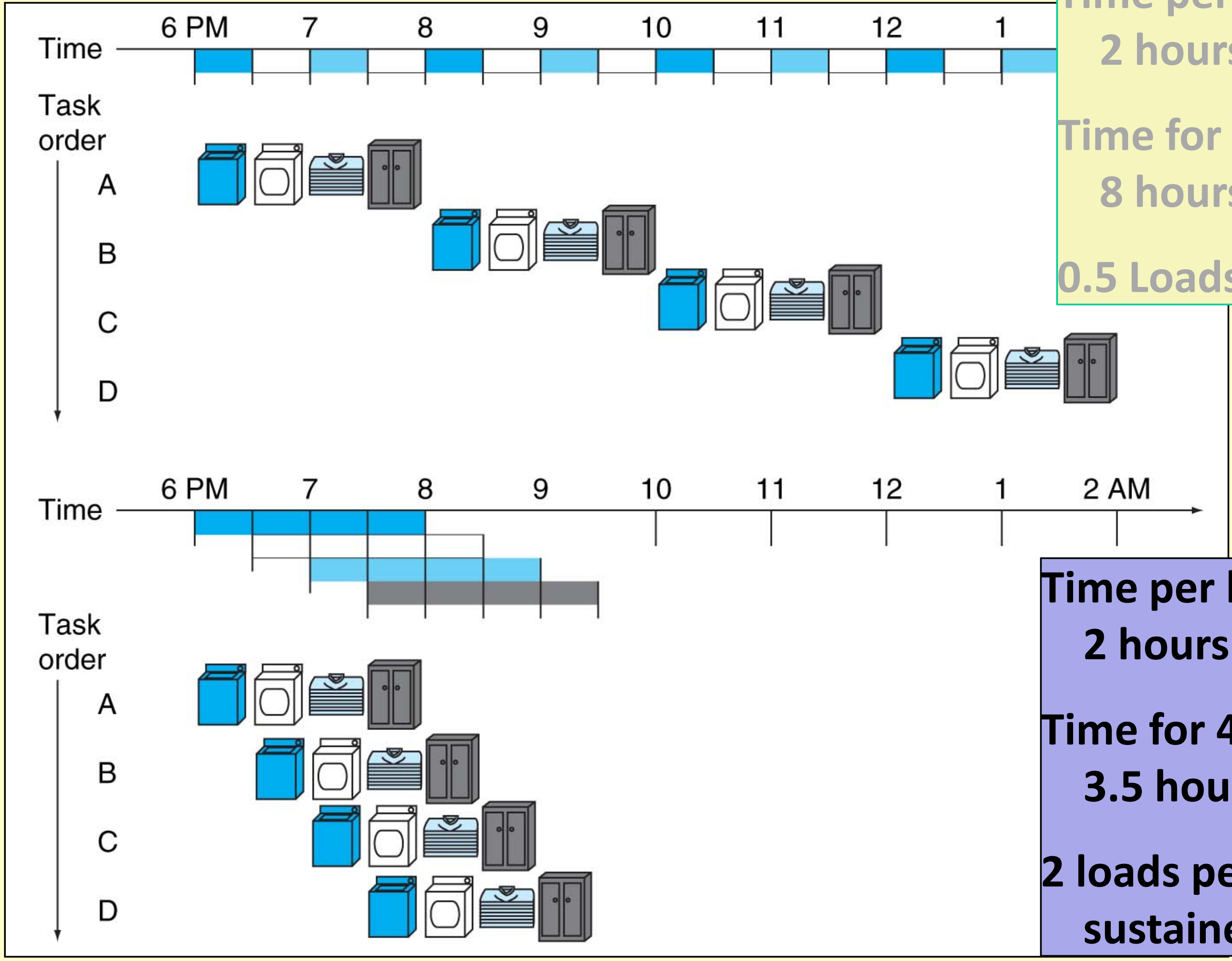


Time per load:–
  2 hours

Time for 4 loads:–
  8 hours!

0.5 Loads per hour

Clothes-washing Analogy (continued)



Time per load:–
  2 hours

Time for 4 loads:–
  8 hours!

0.5 Loads per hour

Time per load:–
  2 hours (still)

Time for 4 loads:–
  3.5 hours!

2 loads per hour
  sustained rate!

Better Analogy

- Production line for automobiles or other manufactured products

- Figure of merit:–
  - Number of cars per day off the line

Pipeline in Computer Processor

- **Partition instruction execution into *stages***
  - I.e., *phases* on the OS slide

- **Each *stage* uses a different set of hardware resources**
  - Avoid conflicts

- ***Stages* can execute concurrently, pipeline style**

Example — Typical RISC Pipeline

- **Instruction fetch**
  - Increment instruction counter as part of this stage
- **Instruction decode and Register fetch**
  - I.e., read operand(s) from register(s)
- **Execution *or* Address Calculation**
  - Arithmetic operation using fetched registers or immediate operand
- **Memory access**
  - Read from or write to (main) memory
- **Write-back to Register**
  - Store result into register again

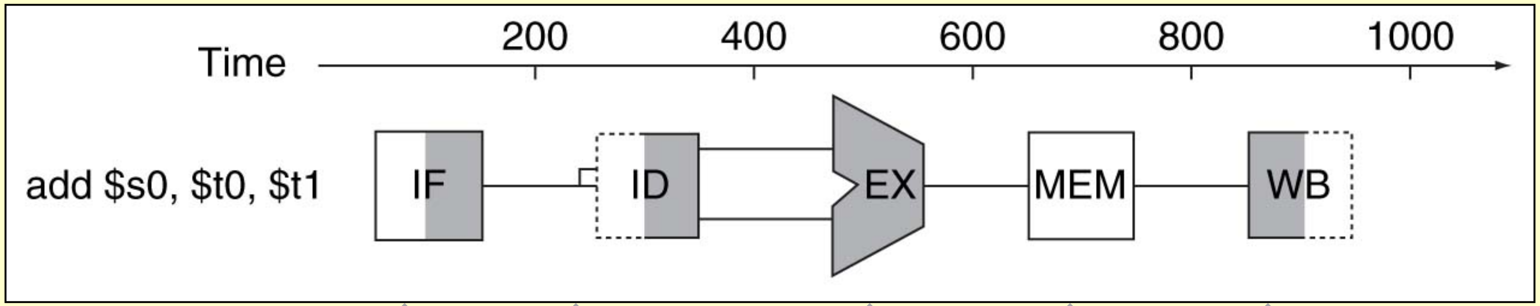
Example — Typical RISC Pipeline

- **Instruction fetch**
  - Increment instruction counter as part of this stage
- **Instruction decode and Register fetch**
  - I.e., read operand(s) from register(s)
- **Execution *or* Address Calculation**
  - Arithmetic operation using fetched registers or immediate operand
- **Memory access**
  - Read from or write to (main) memory
- **Write-back to Register**
  - Store result into register again

Every architecture partitions its instruction set differently

These are the five cycles of a many “textbook” pipelines!

Example — Typical RISC Pipeline (continued)



Instruction fetch

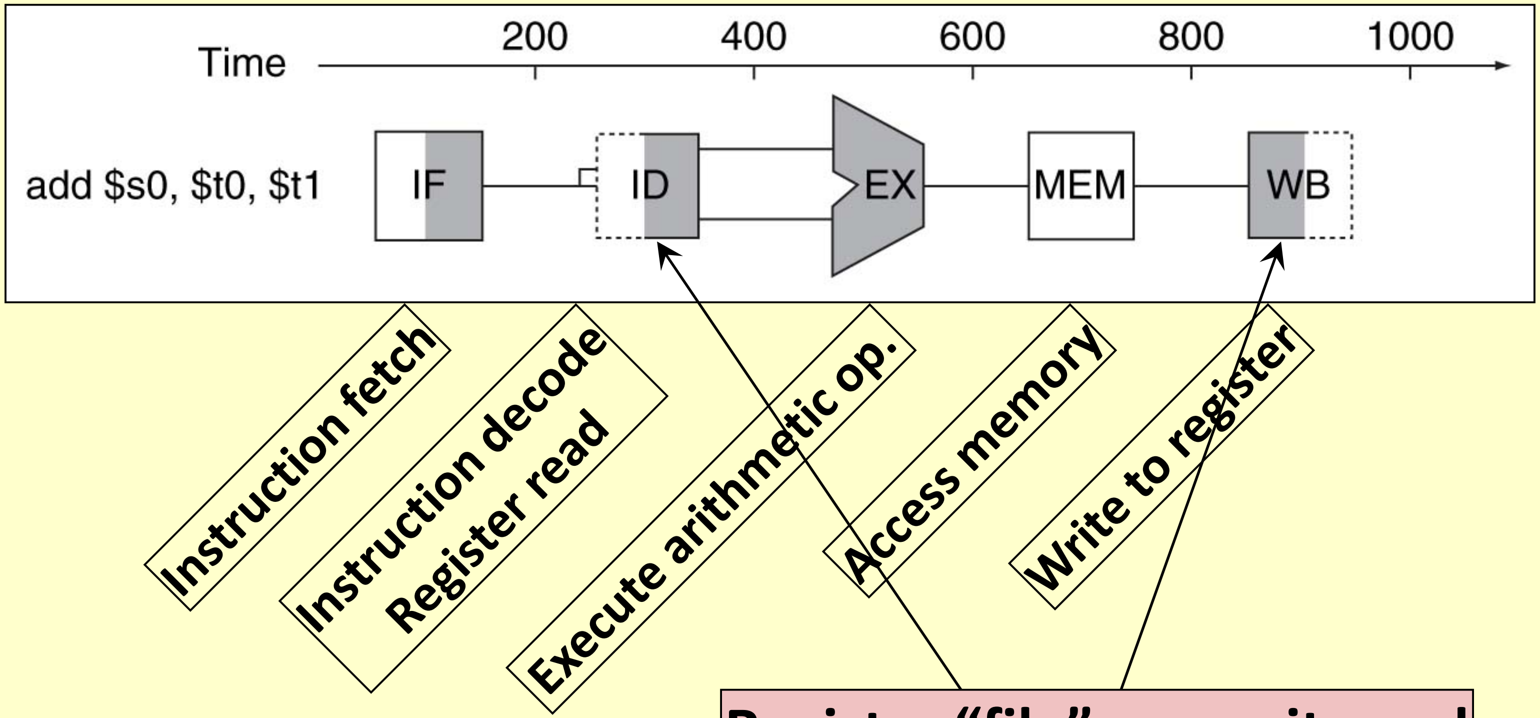
Instruction decode
Register read

Execute arithmetic op.

Access memory

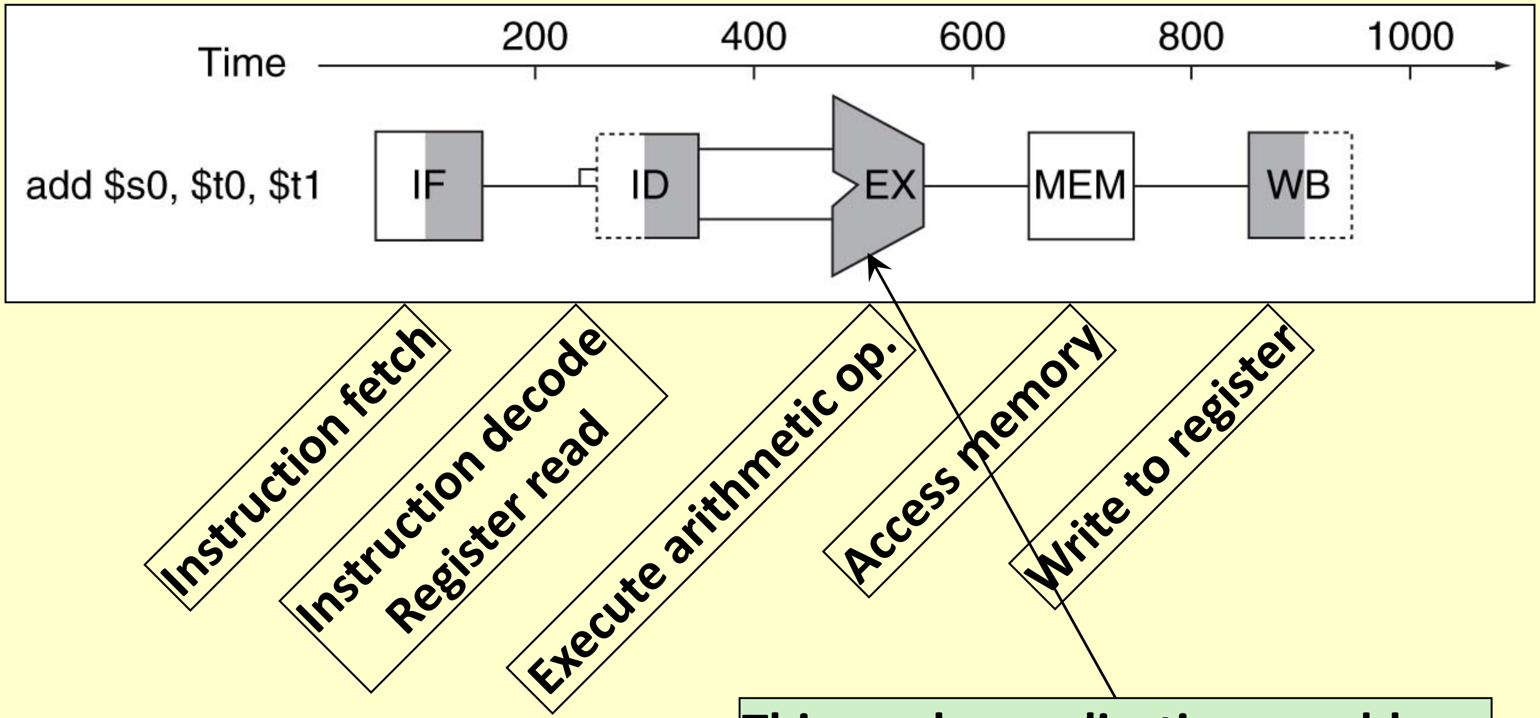
Write to register

Example — Typical RISC Pipeline (continued)



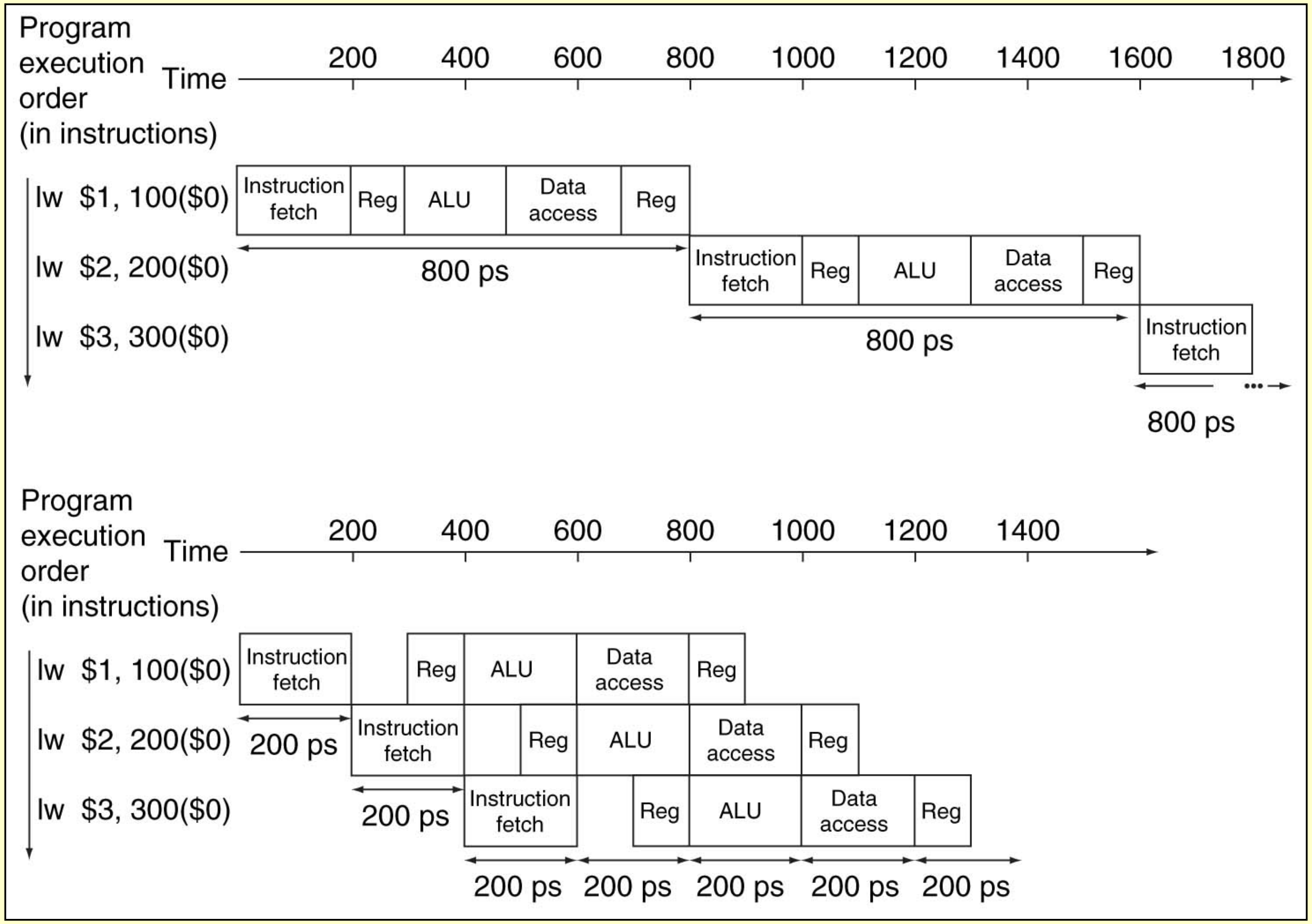
Register "file" can write and read in two half-cycles

Example — Typical RISC Pipeline (continued)

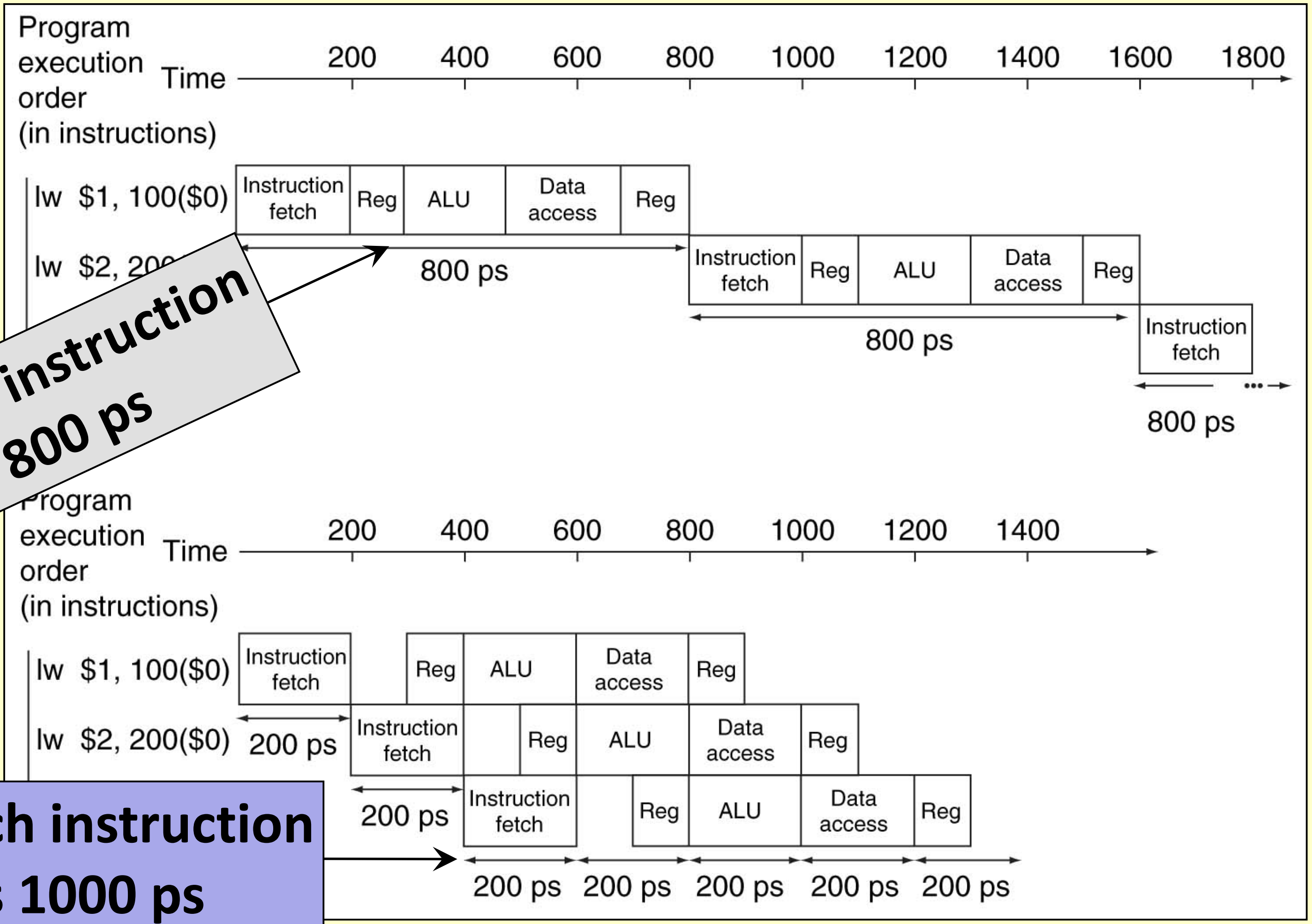


This can be application or address arithmetic (*not both*)

Example — Typical RISC Pipeline (continued)



Example — Typical RISC Pipeline (continued)

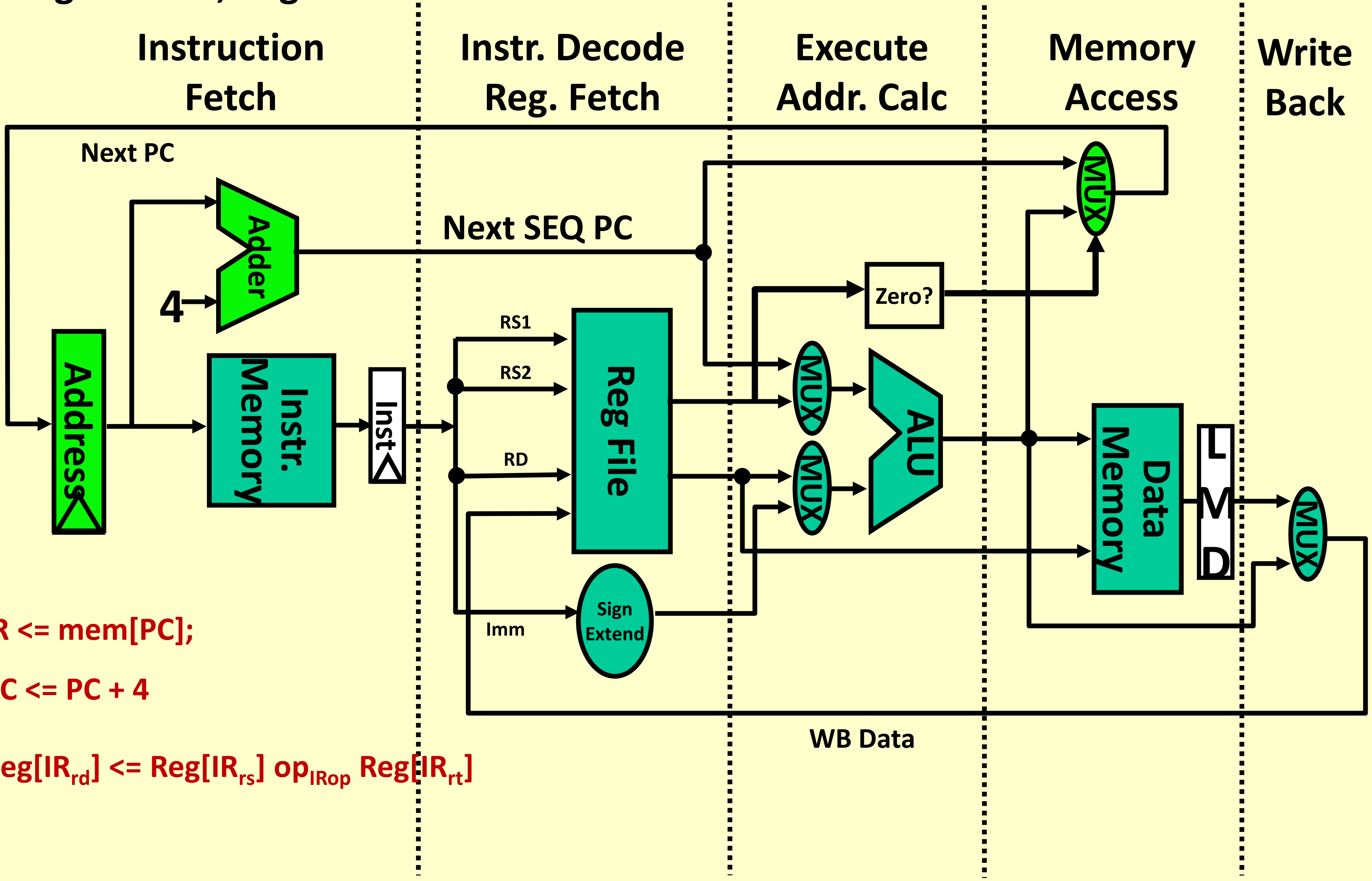


Each instruction is 800 ps

Each instruction is 1000 ps

Typical RISC Datapath (in five steps)

Figure C.18, Page C-30



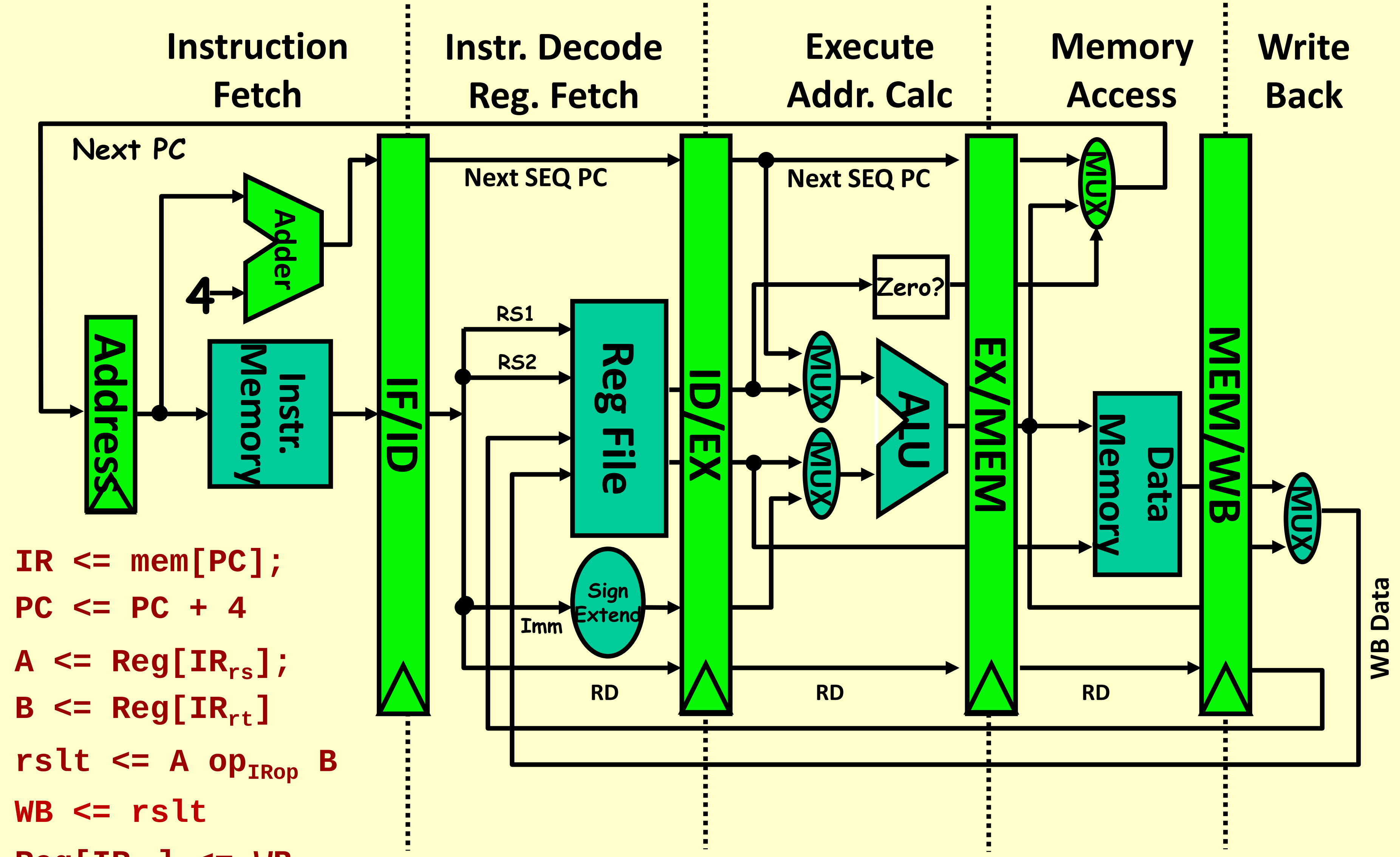
IR <= mem[PC];

PC <= PC + 4

Reg[IR$_{rd}$] <= Reg[IR$_{rs}$] op$_{IRop}$ Reg[IR$_{rt}$]

Typical RISC Datapath (showing pipeline registers)

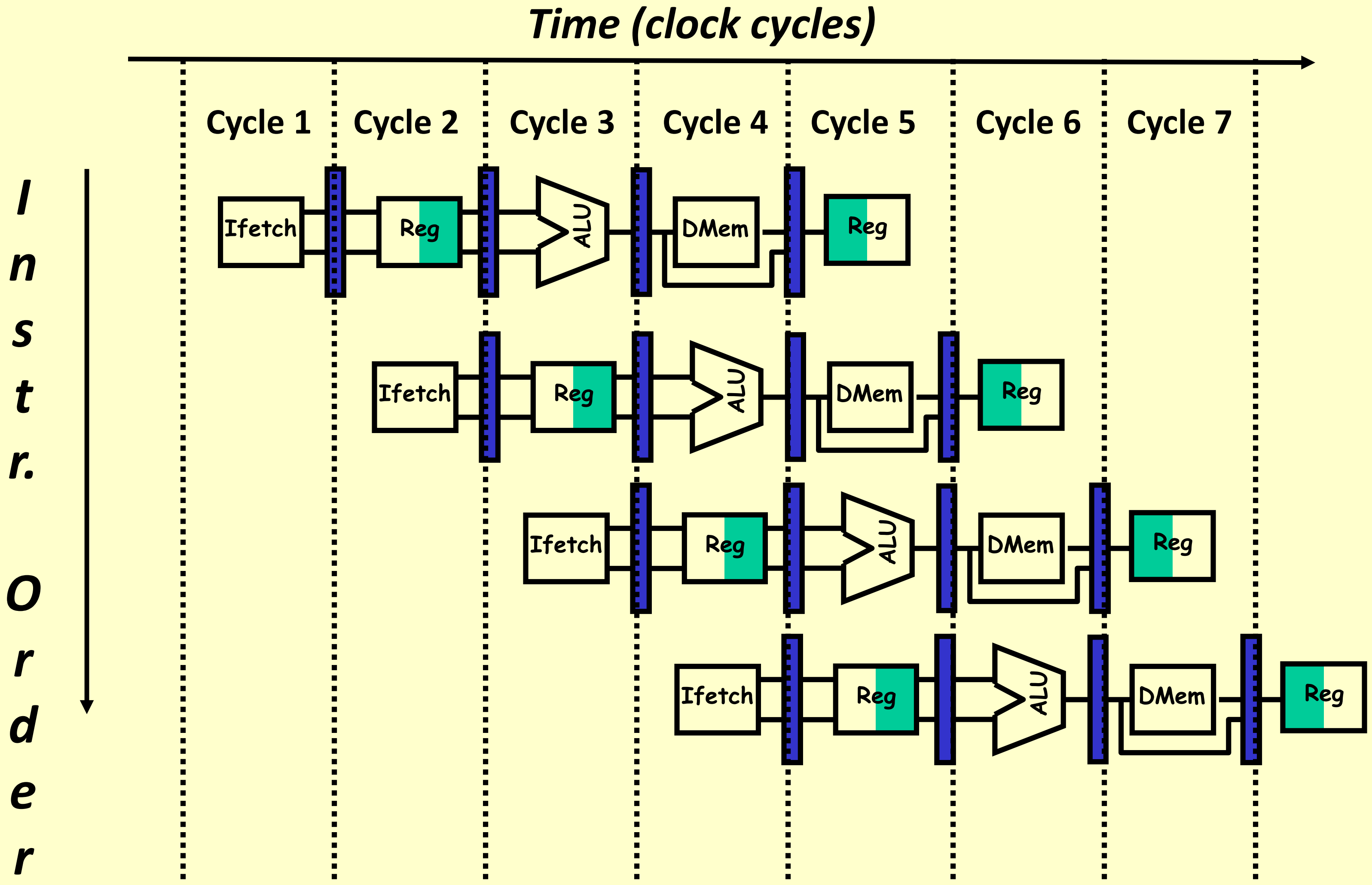
Figure C.19, Page C-31



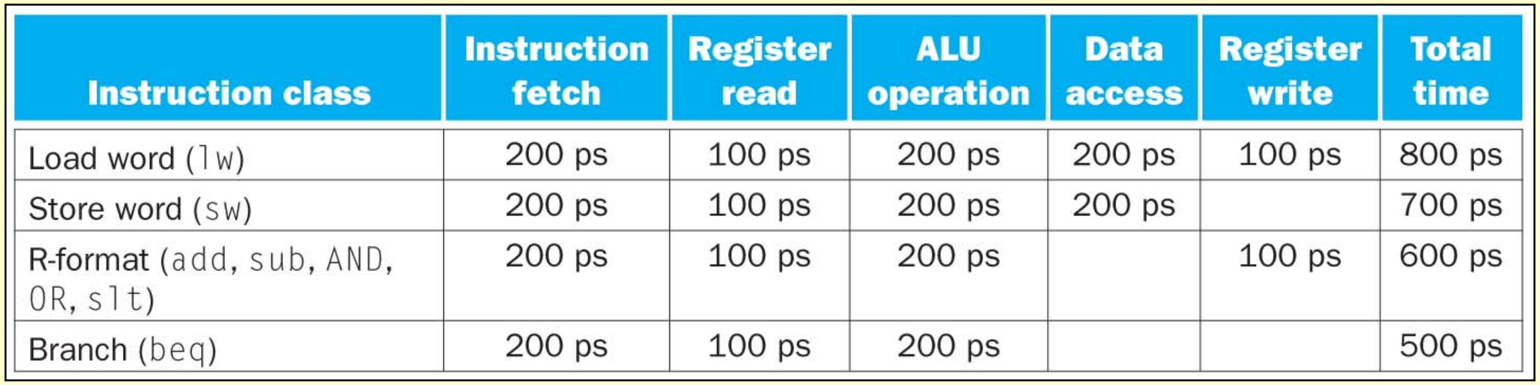
```
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_rs];
B <= Reg[IR_rt]
rslt <= A op_IRop B
WB <= rslt
Reg[IR_rd] <= WB
```

Visualizing Pipelining

Figure C-3, Page C-9



More on Typical RISC Pipeline



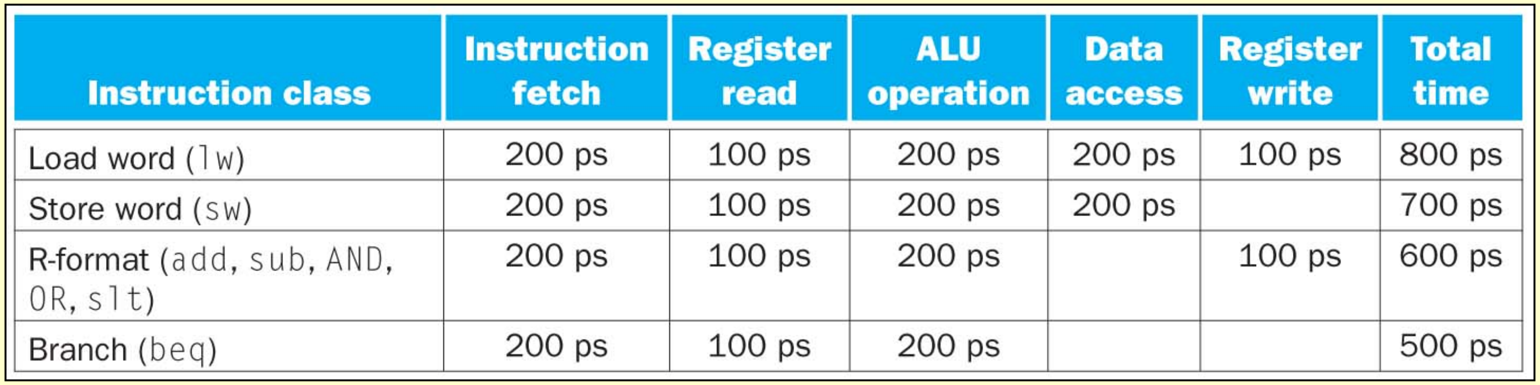
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Digression – RISC-V Instruction Formats

Fig A.23

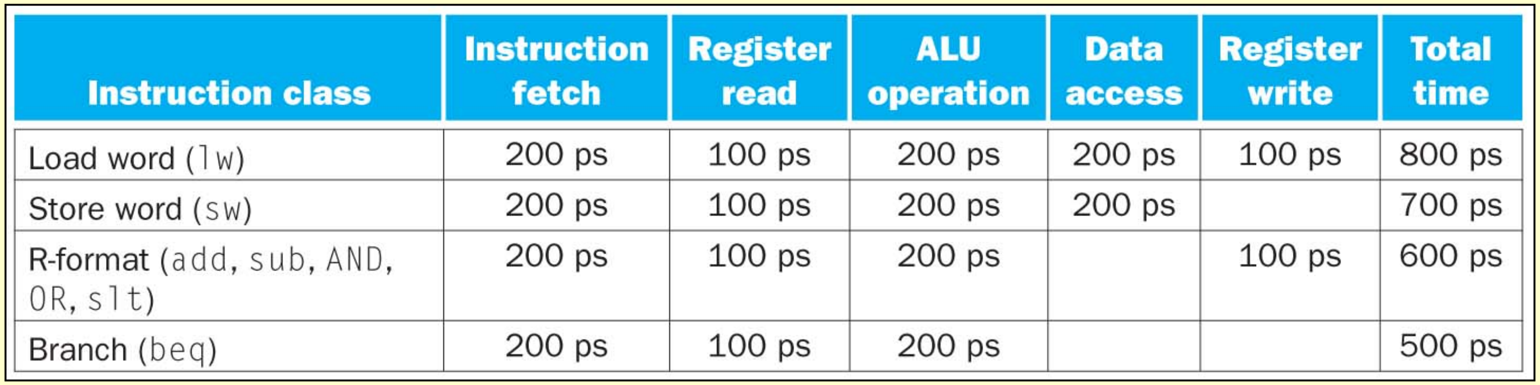
31 -- 25	24 -- 20	19 -- 15	14 -- 12	11 -- 7	6 -- 0		
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm [11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm		[31-12]		rd	opcode		U-type

More on Typical RISC Pipeline



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Quantifying the Speedup



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

- **Assume**
  - 40% ALU operations
  - 20% branches
  - 30% Load operations
  - 10% Store operations
  - No pipeline penalty

- **Average (non-pipelined) instruction duration = 650 ps**

$$.4 \times 600 + .2 \times 500 + .3 \times 800 + .1 \times 700 = 650$$

Quantifying the Speedup (continued)

$$Speedup = \frac{AveUnPipelinedTime}{AvePipelinedTime} = \frac{650}{200} = 3.25$$

- **Unpipelined architecture would allow variable duration instructions**
  - Branches much faster than Loads
- **Pipelined architecture requires every cycle to take exactly the same time**
  - Some wasted time in Branch, ALU, and Store operations
- **Speedup is less if there is a penalty for pipelining**

More on Speedup

- Suppose that there is a penalty for including pipelining (*vs.* non-pipelined design)
- The clock cycle of each stage in pipelined architecture would be slightly longer
- Speedup would be less
- Subject to Amdahl's Law

- See pp. C-8 - C.10

Questions?

Life is Never that Simple!

Definition!

- Introducing *hazards*
- i.e., factors that interfere with full and efficient pipelining
- Prevent instruction from starting or continuing on next cycle

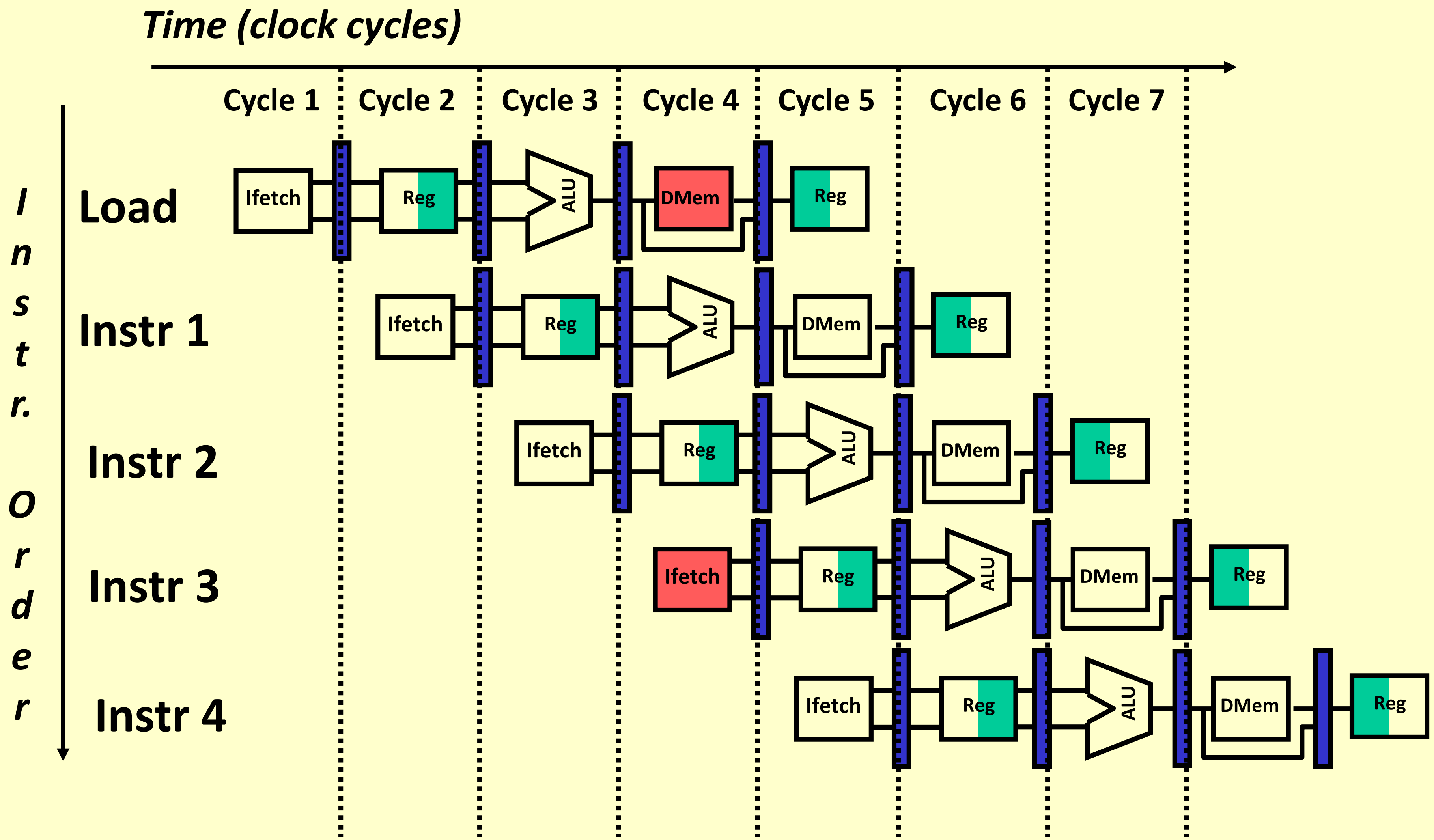
Hazards

- ***Structural* – resource conflicts among successive instructions**
  - Hardware cannot support all possible combinations of instructions in rapid succession
- ***Data* – dependencies of instructions on results of other instructions**
  - *x = a * b + c*   Cannot add until multiply is done
- ***Control* – branches that change instruction fetch order**
  - Will affect instructions that have already been fetched!

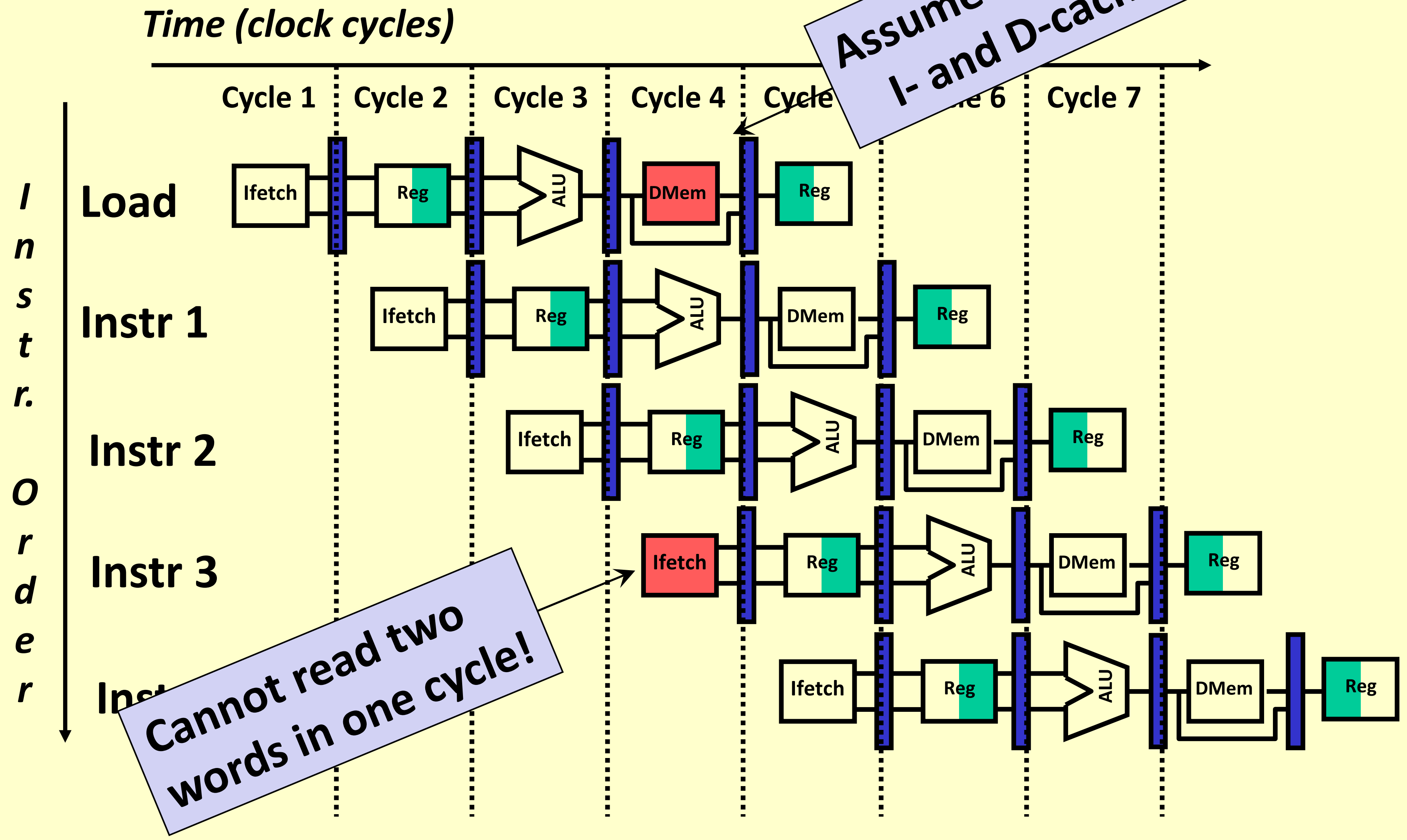
Structural Hazard Example:–

(what if instructions and data shared) **One Memory Port**

Page C-11

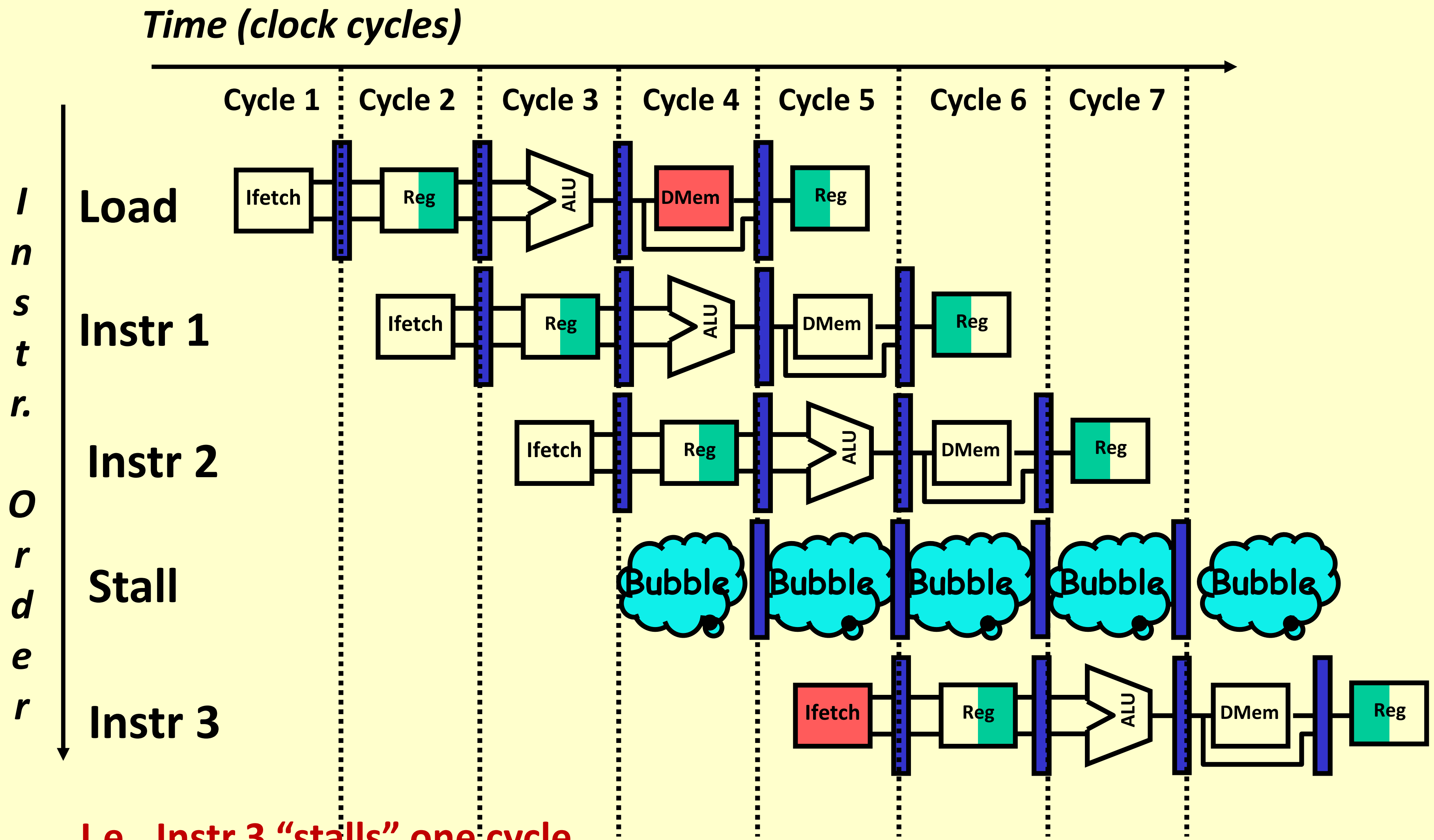


Structural Hazard Example:– (What if) One Memory Port Page C-11



Penalty – a Bubble in Pipeline

(Derived from Figure C.10, Page C-19,Idle -> Bubble)



I.e., Instr 3 “stalls” one cycle.

Speedup Equation for Pipelining

See pp. C-11 & C-12

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

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

For simple RISC pipeline, CPI = 1

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

Quantifying Dual-port vs. Single-port

- **Machine A: Dual ported memory**[1]
  - or “Harvard Architecture” [2]
- **Machine B: Single ported memory**
  - but pipelined implementation has 1.05 × faster clock rate
- **Ideal CPI = 1 for both**
- **Loads are 40% of instructions executed**

$Speedup_A = Pipeline\ Depth/(1 + 0) \times (clock_{unpipe}/clock_{pipe})$

$= Pipeline\ Depth$

$Speedup_B = Pipeline\ Depth/(1 + 0.4 \times 1) \times (clock_{unpipe}/(clock_{unpipe} / 1.05)$
(<-every load implies a stall)

$= (Pipeline\ Depth/1.4) \times 1.05$

$= 0.75 \times Pipeline\ Depth$

$Speedup_A / Speedup_B = Pipeline\ Depth/(0.75\ x\ Pipeline\ Depth) = 1.33$

- **Machine A is 1.33 times faster**

1. Dual-ported RAM (DPRAM) is a type of random-access memory that allows multiple reads or writes to occur at the same time, or early the same time, unlike single-ported RMA which allows only one access at a time.
2. physically separate storage and signal pathways for instructions and data

What to do about Memory Structural Hazard

- **Typical processor has two memory ports …**
  - I-port
  - D-port
- **… or two separate L1 cache memories**
  - I-cache
  - D-cache

Other Structural Hazards (examples)

- **Multiply**
  - Expensive in gates to make fully pipelined
- **Floating-point divide**
  - *Very* expensive to make fully pipelined
- **Floating-point square-root**
  - *Prohibitively* expensive to pipeline at all

Structural Hazards (conclusion)

- **Processor hardware *must* check**
  - Introduce “stalls”
- **Compiler *should be* aware**
  - Re-arrange compiled code

Questions?

Data Hazard on R1

Figure C.4, Page C-13

Time (clock cycles)

I n s t r. O r d e r

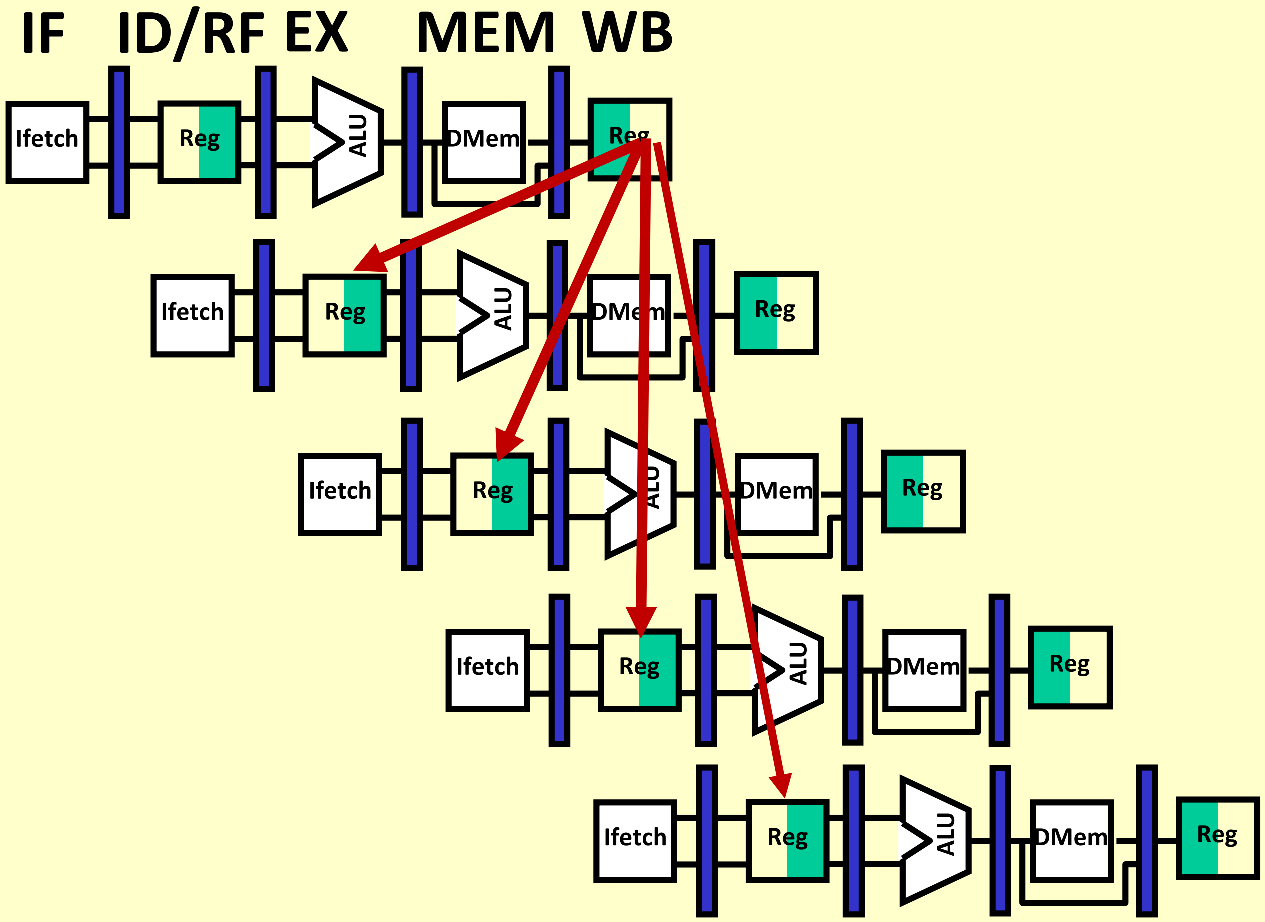
add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11



Three Generic Data Hazards (1)

- **Read After Write (RAW)**
  Instr$_J$ **tries to read operand before *Instr***$_I$ **writes it**

```
I: add r1,r2,r3    Write specified here
J: sub r4,r1,r3    Read specified here
                        but instruction I isn't finished
```

- **Caused by a "Dependence" (in compiler nomenclature)**
  - This hazard results from an actual need for data communication among instructions

Three Generic Data Hazards (2)

- **Write After Read (WAR)**
  Instr_J writes new value before Instr_I reads old value

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

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

- **Can't happen in MIPS 5 stage pipeline because:–**
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5
- **Many other processors susceptible to this hazard**

Three Generic Data Hazards (3)

- **Write After Write (WAW)**
  Instr$_J$ **writes a value before *Instr***$_I$ **writes a different value**

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

$$\begin{tikzcd} I \arrow[d, bend right] \\ J \arrow[u, bend left] \end{tikzcd}$$

He who laughs last…
(should be J)

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

Data Hazard on R1

Figure C.4, Page C-13

Time (clock cycles)

Instr. Order

add r1,r2,r3

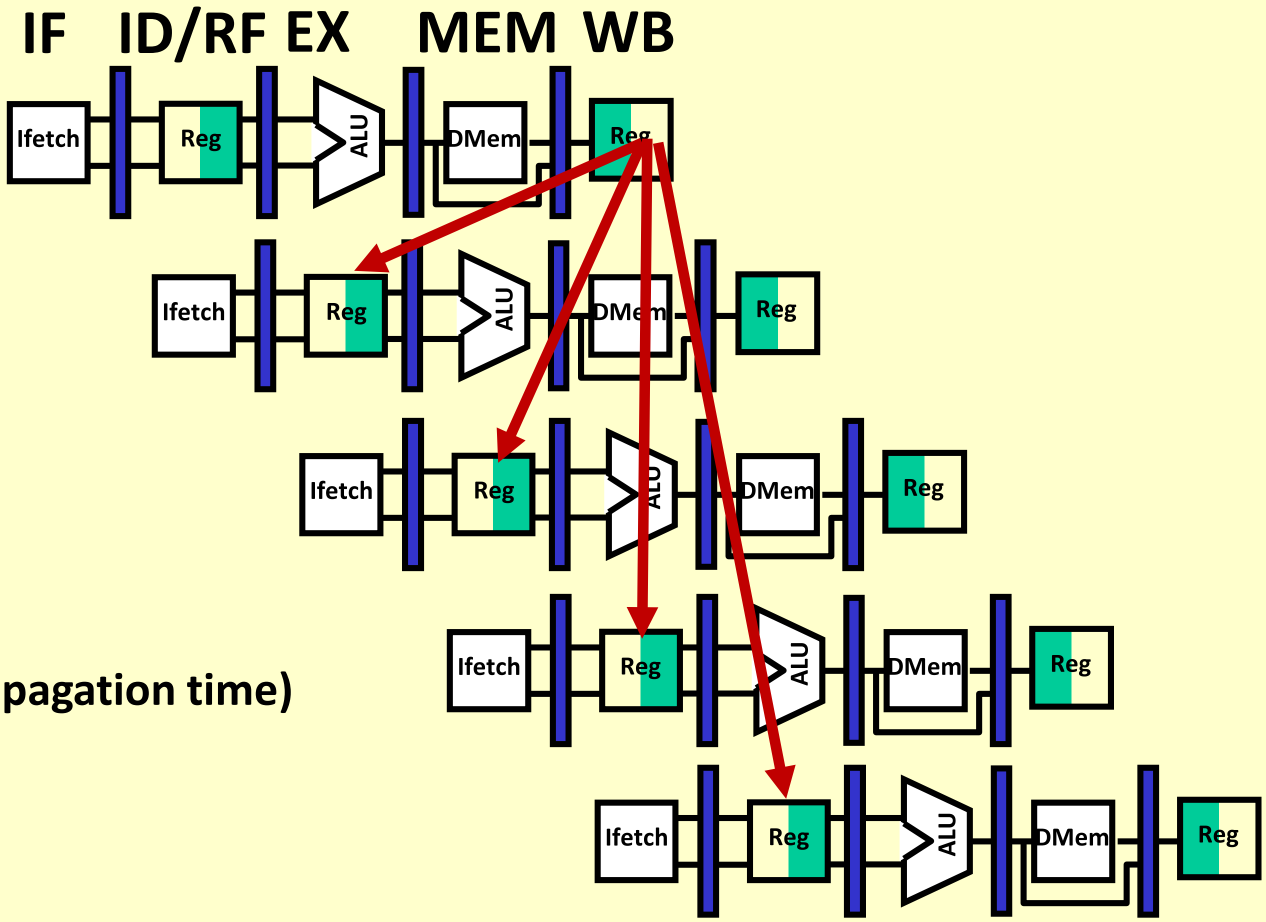
sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

(Propagation time)



Forwarding to Avoid Data Hazard

(Figure C.5-7, Page C-15, 16, 17)

Note in Fig. C.5 the diagram is inconsistent with its caption. Only ALU input 1 is used.

Time (clock cycles)

Instr. Order

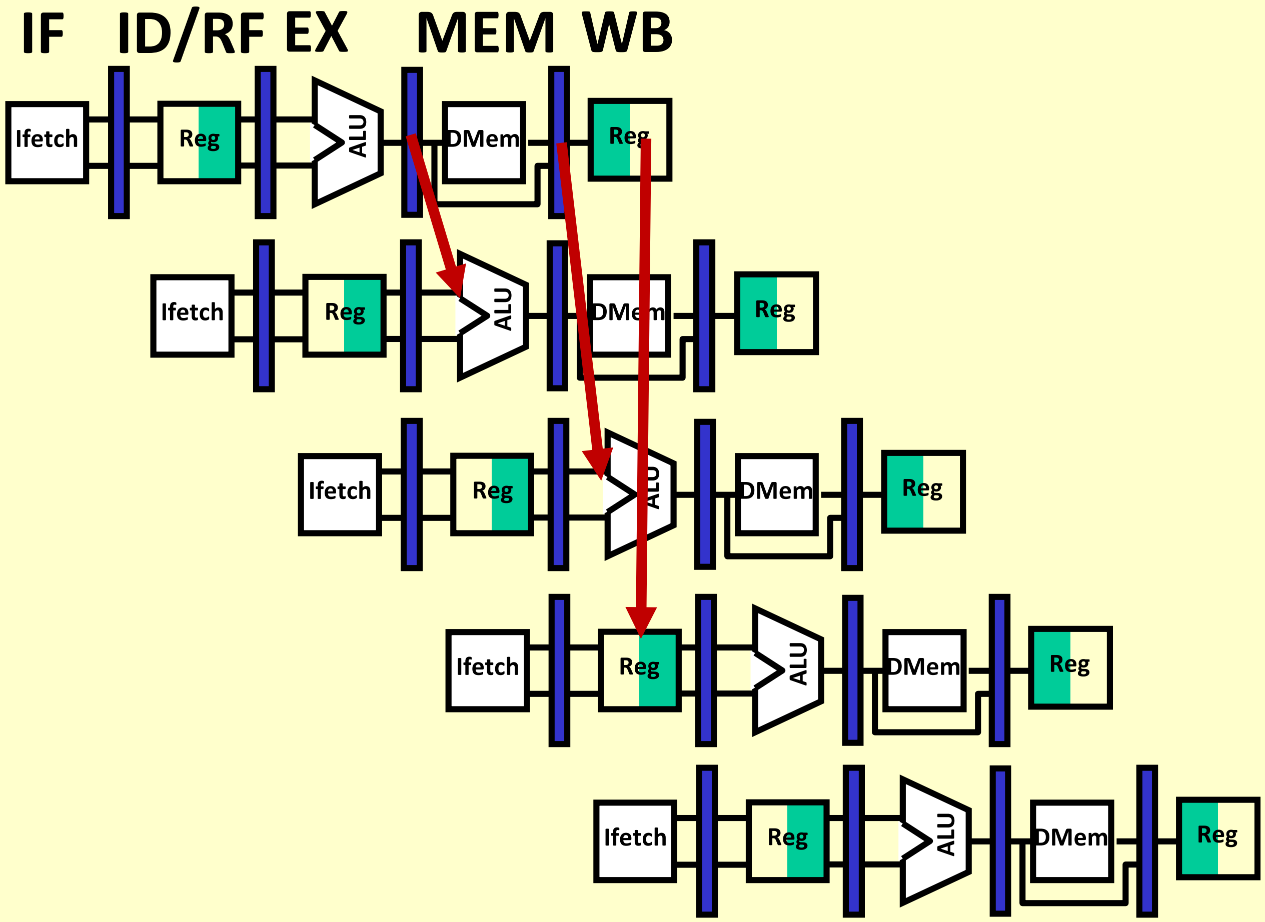
add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

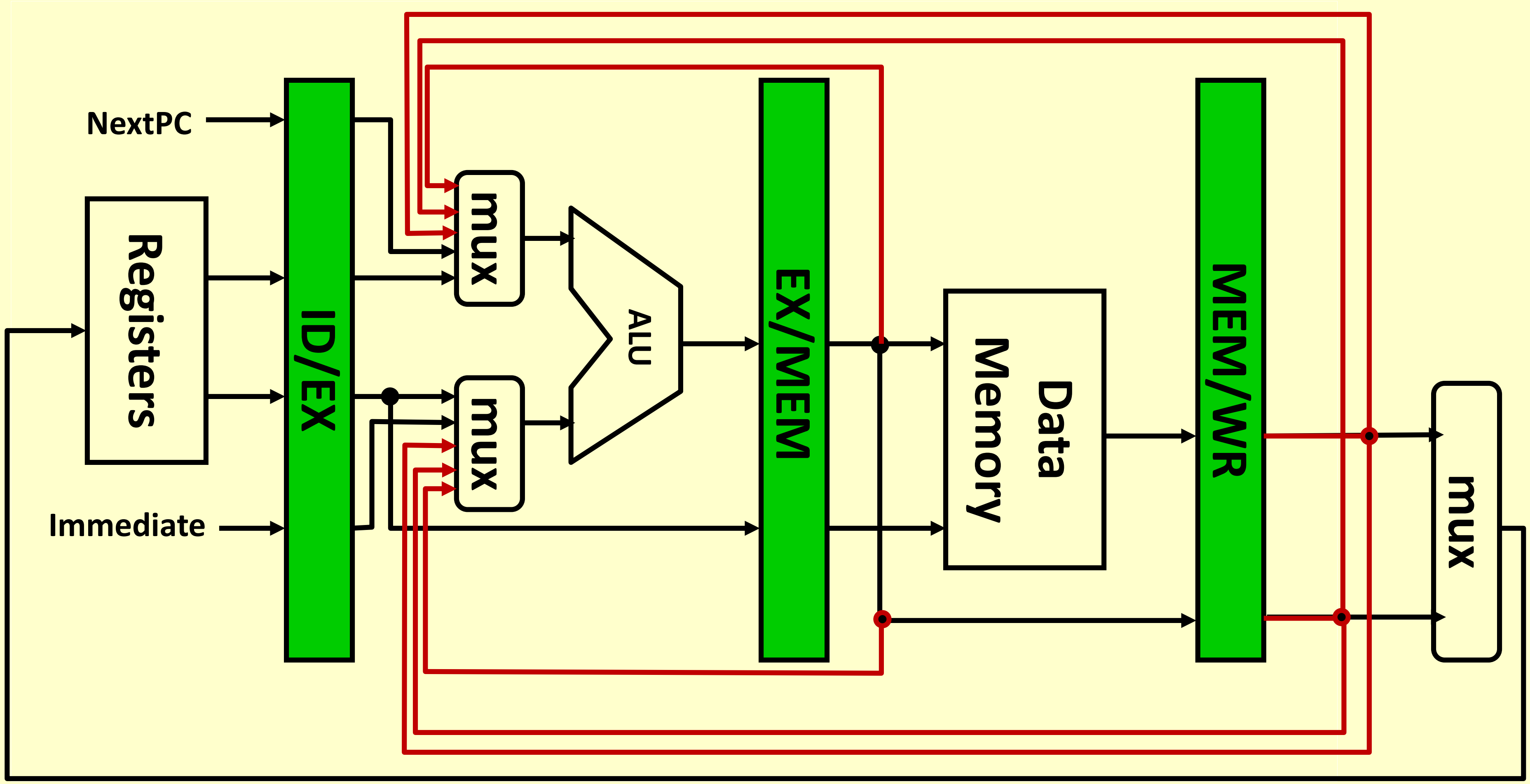
or r8,r1,r9

xor r10,r1,r11



HW Change for Forwarding

Figure C.24, Page C-37



Several parts bring data to ALU mux: Ex/Mem, also Mem/Wr, including DataMem at Mem/Wr

Forwarding to Avoid LoadW-StoreW Data Hazard Figure C.6, Page C-16

Time (clock cycles)

Instr. Order

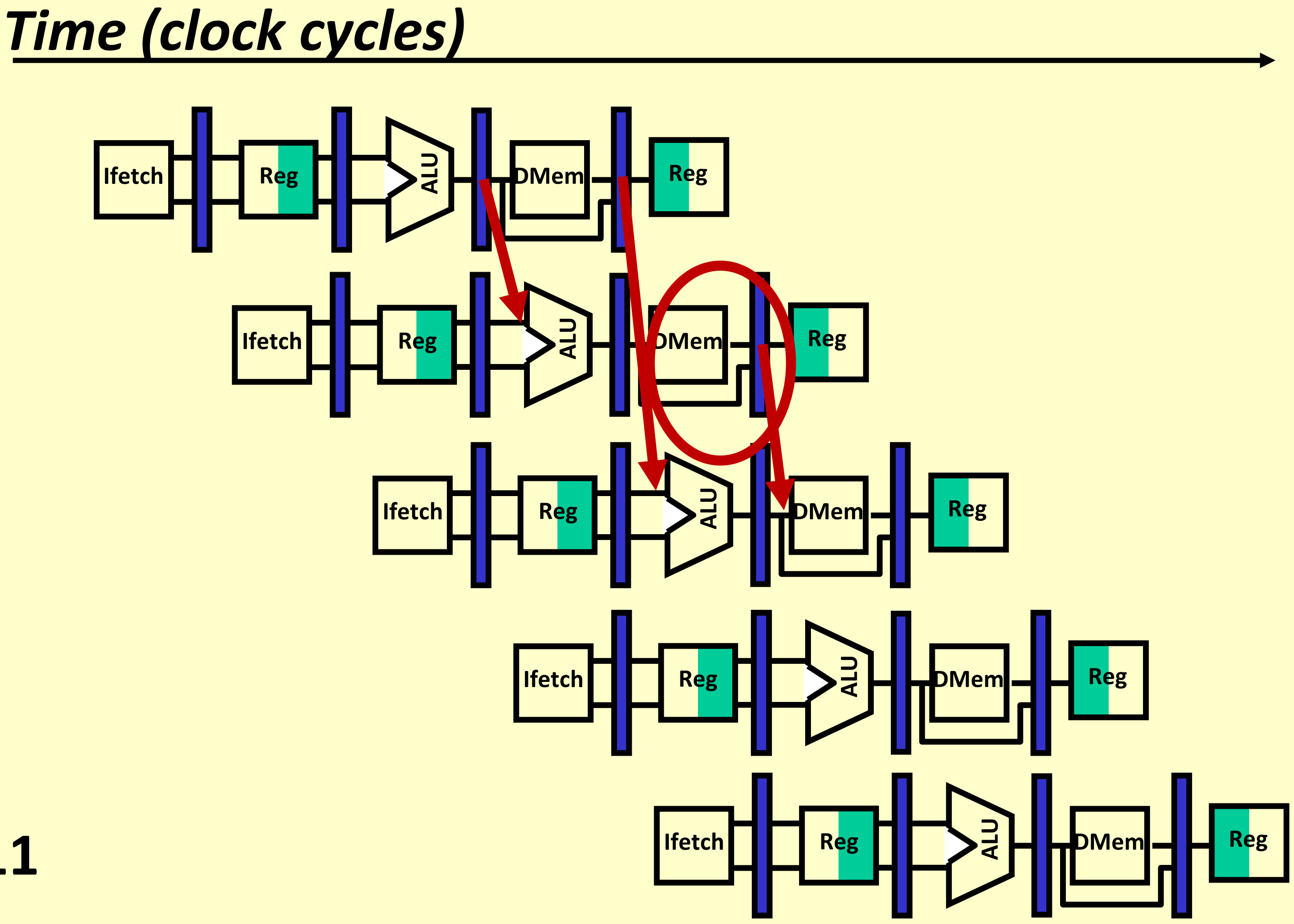
add r1,r2,r3

lw r4, 0(r1)

sw r4,12(r1)

or r8,r6,r9

xor r10,r9,r11



Data Hazard Even with Forwarding

(Figure C.7, Page C-17)

Time (clock cycles)

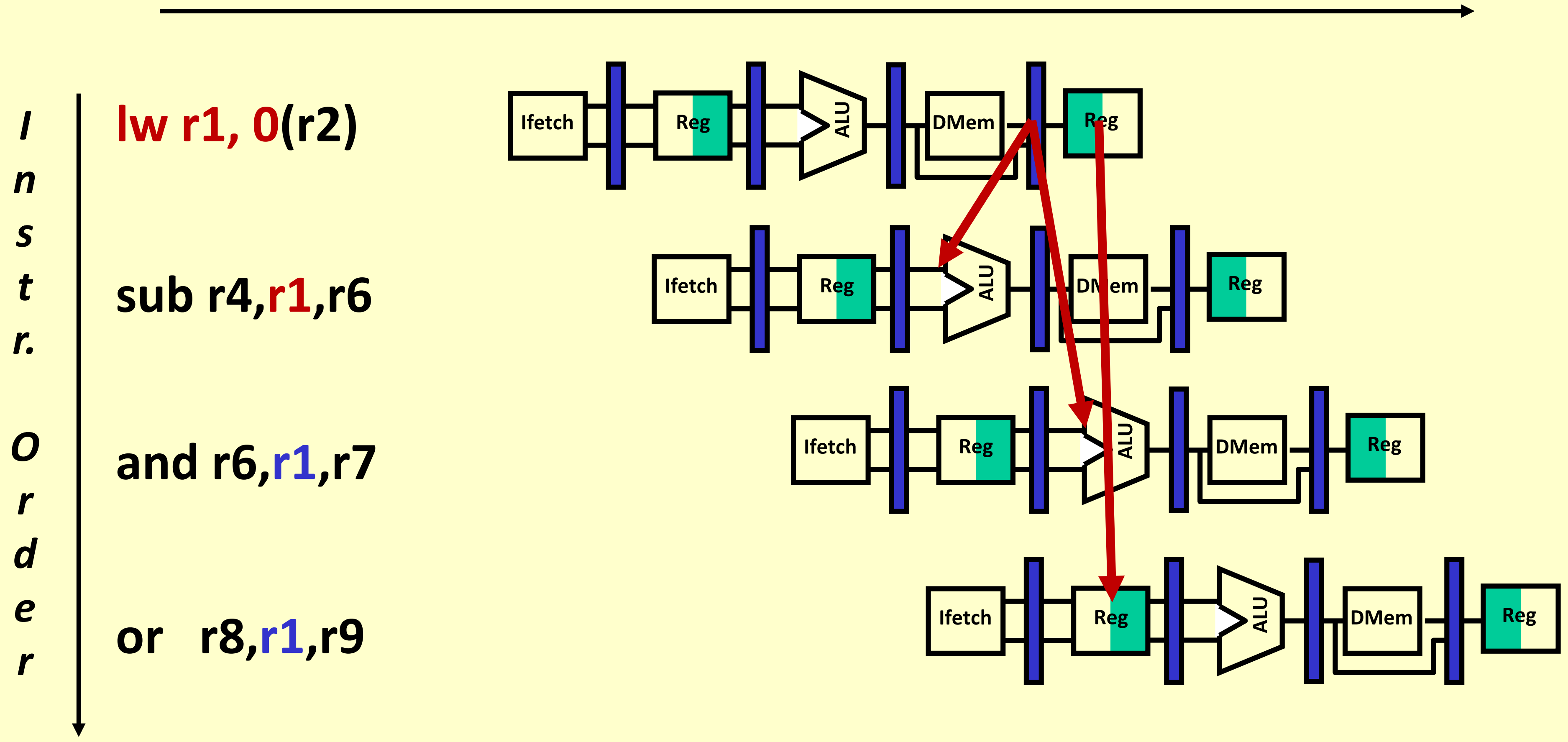
I n s t r. O r d e r

lw r1, 0(r2)

sub r4,r1,r6

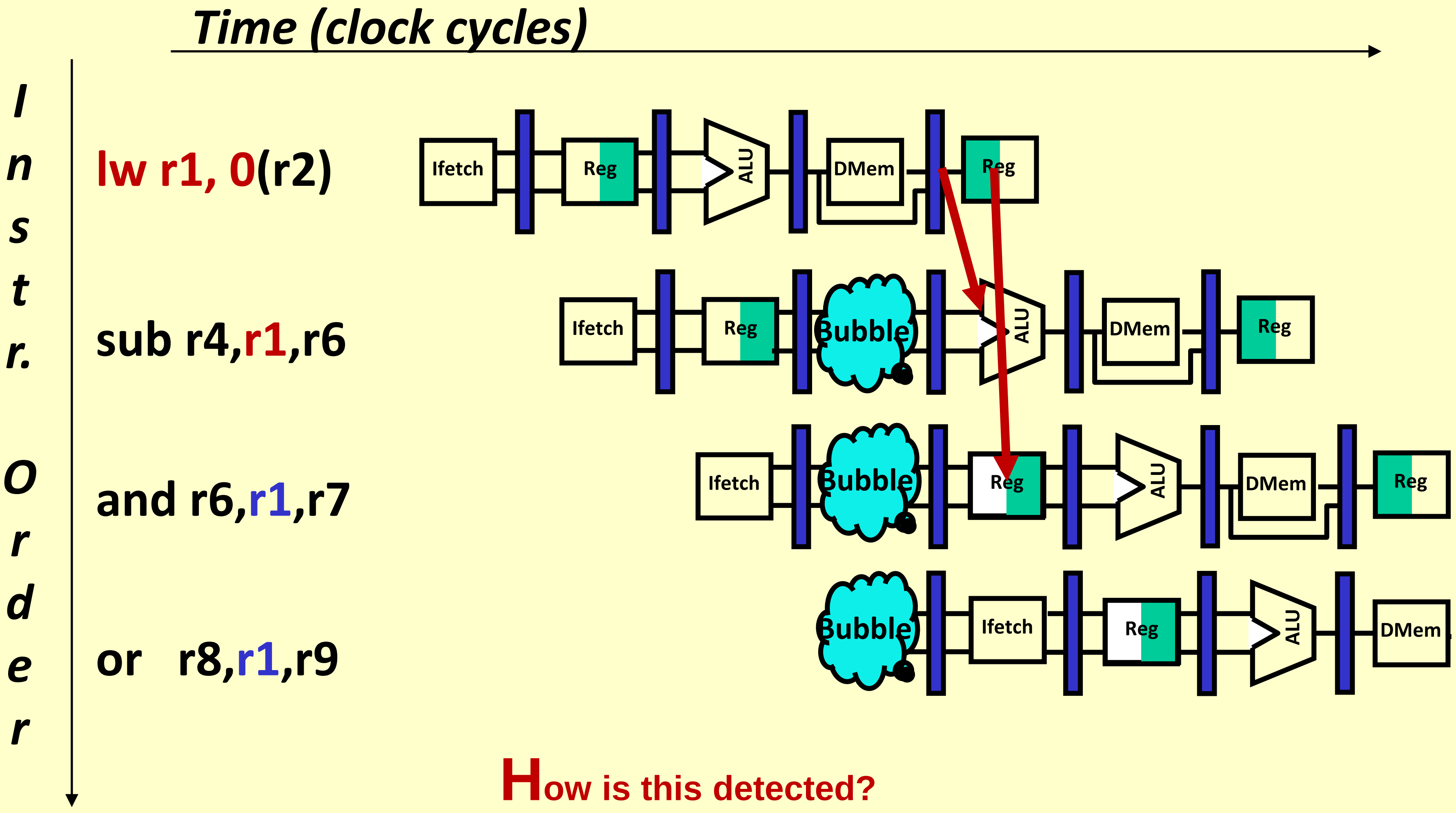
and r6,r1,r7

or r8,r1,r9



Data Hazard Causing Pipeline Stall

(See Figure C.7 and C.8, Pages C-17 and C-18) (Bubble ≡ Stall)



How is this detected?

See below

Software Scheduling to Avoid Data Hazards

Try producing fast code for

a = b + c;

d = e – f;

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

Slow code:

```
LW   Rb,b
LW   Rc,c
ADD  Ra,Rb,Rc    <--  Cannot ADD until Rc is loaded
SW   a,Ra
LW   Re,e
LW   Rf,f
SUB  Rd,Re,Rf    <--  Cannot SUB until Rf is loaded
SW   d,Rd
```

Data Hazard (continued)

Time (clock cycles)

Instr. Order

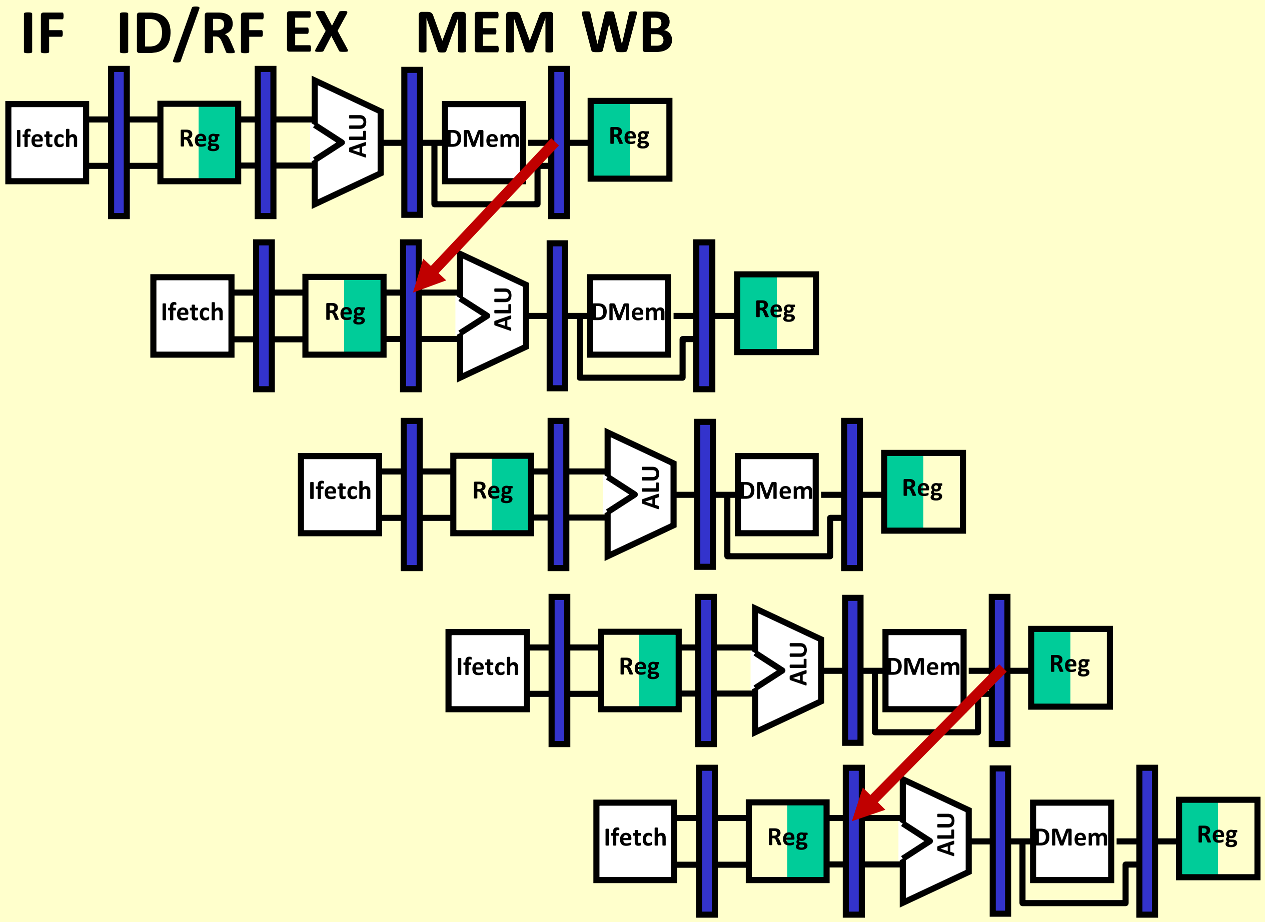
lw Rc, c

add Ra, Rb, Rc

…

lw Rf, f

sub Rd, Re, Rf



Software Scheduling to Avoid Data Hazards

Try producing fast code for

a = b + c;

d = e – f;

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

Slow code:

```
LW   Rb,b
LW   Rc,c
ADD  Ra,Rb,Rc
SW   a,Ra
LW   Re,e
LW   Rf,f
SUB  Rd,Re,Rf
SW   d,Rd
```

Fast code:

```
LW     Rb,b
LW     Rc,c
LW     Re,e
ADD    Ra,Rb,Rc
LW     Rf,f
SW     a,Ra
SUB    Rd,Re,Rf
SW     d,Rd
```

A useful instruction

A useful instruction

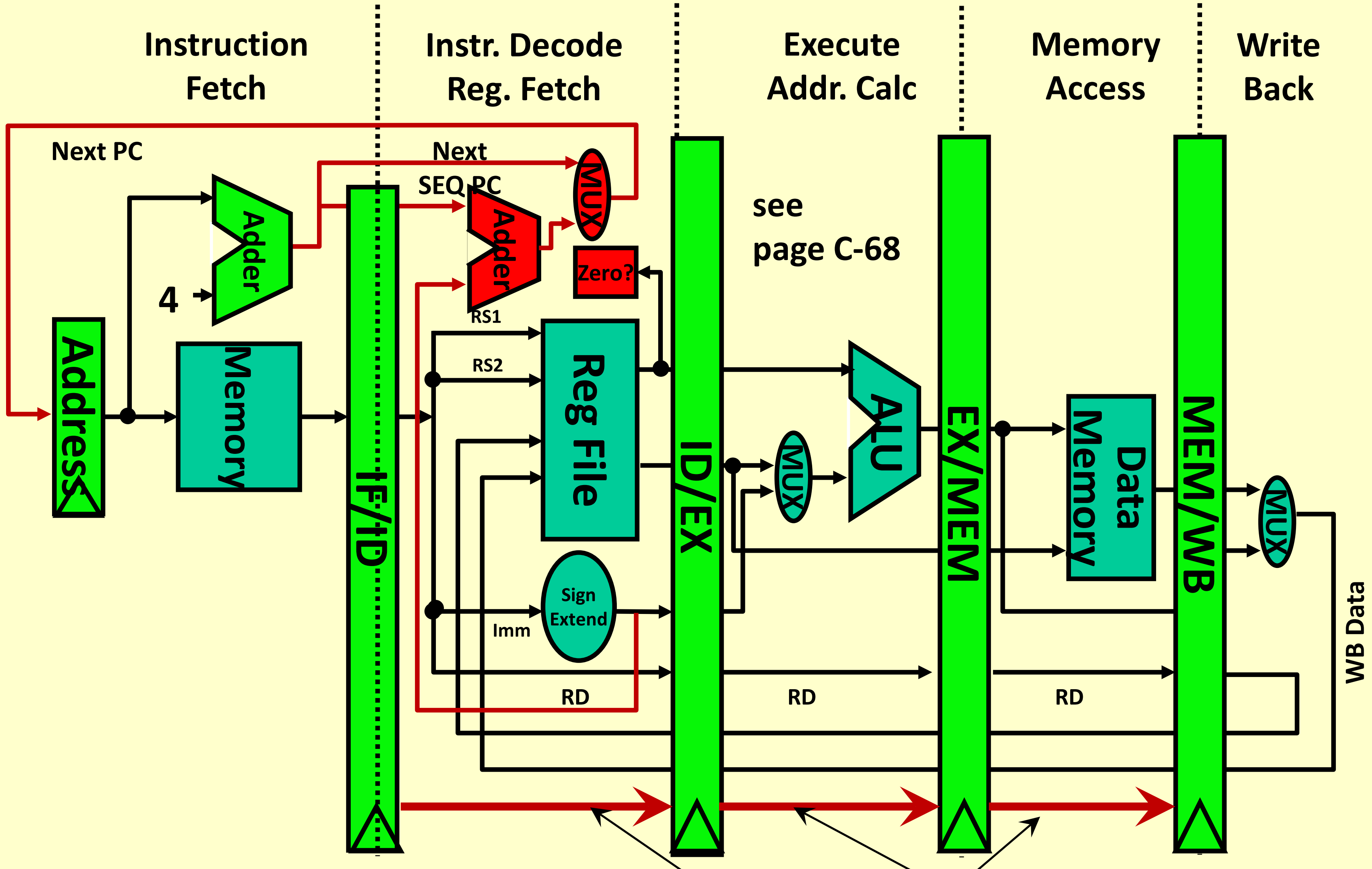
Compiler optimizes for performance.
Hardware checks and interlocks for safety.

Questions?

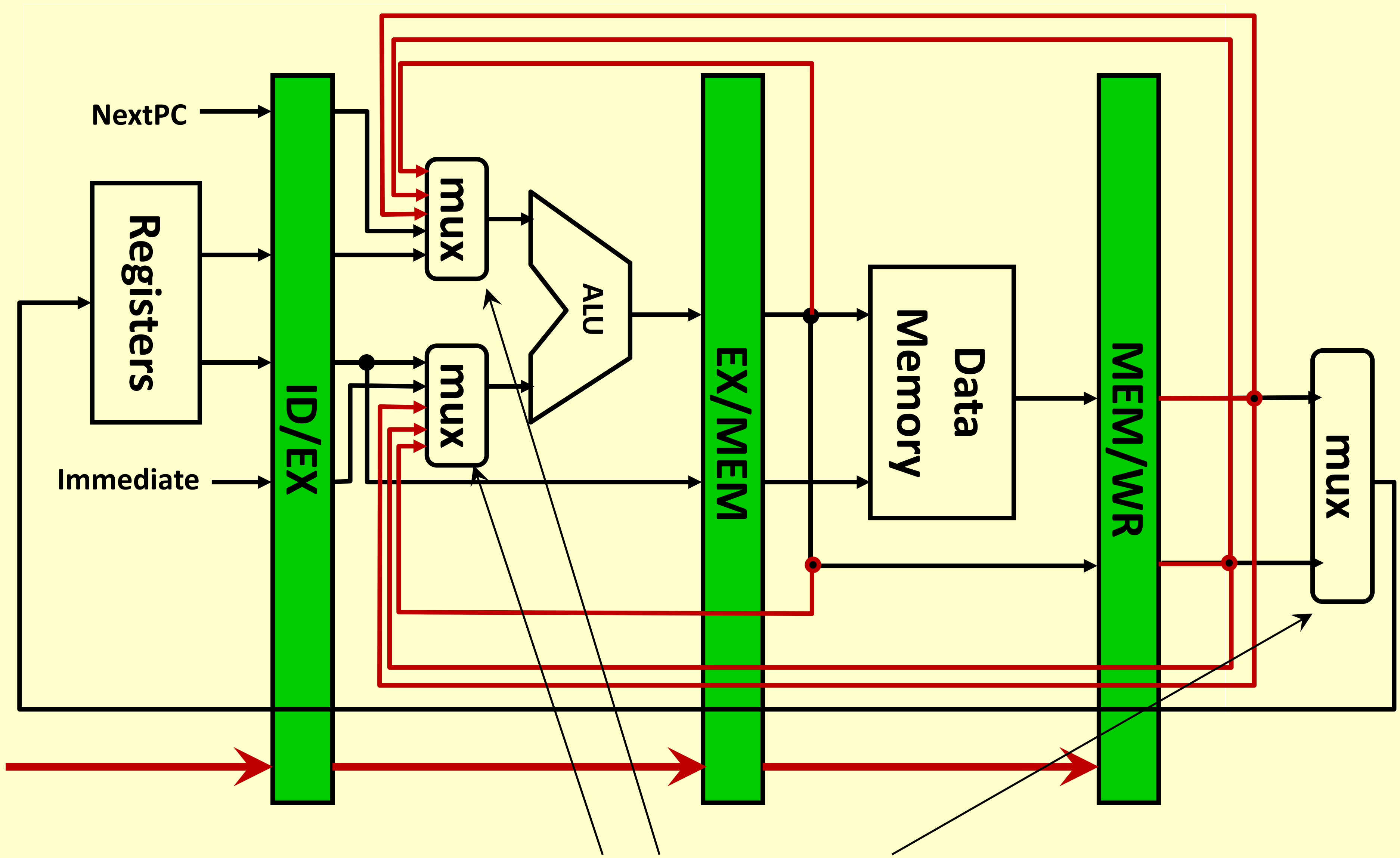
Detecting and Interlocking for Stalls

- Instruction Decode
- For each operand, determine where it comes from
- Send control codes through pipeline

1. Not used
2. Immediate
3. Read from register
4. Forward from ALU stage
5. Forward from Mem stage



Control codes (e.g., for multiplexers)

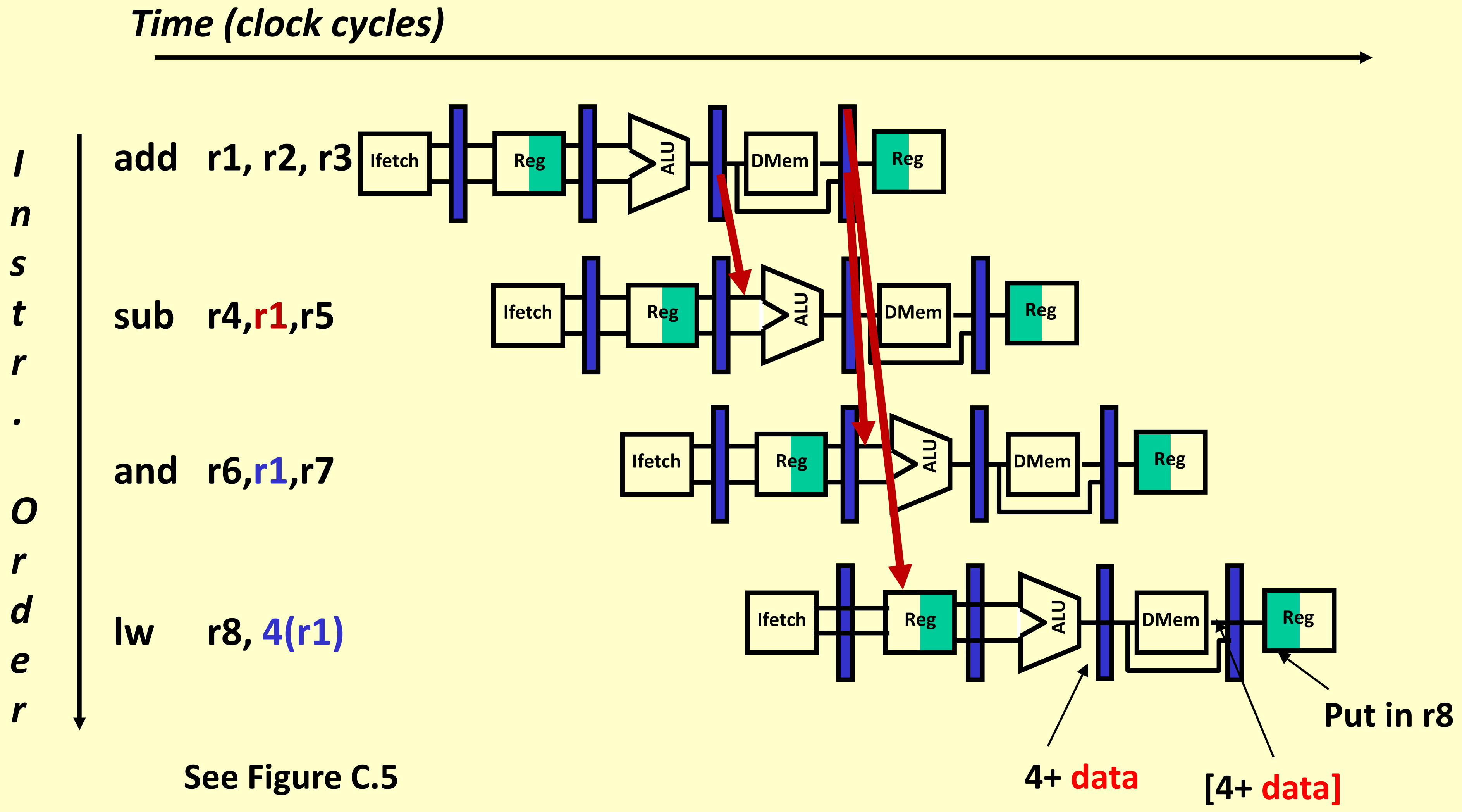


Codes control these MUXes
(and others)

How to decide

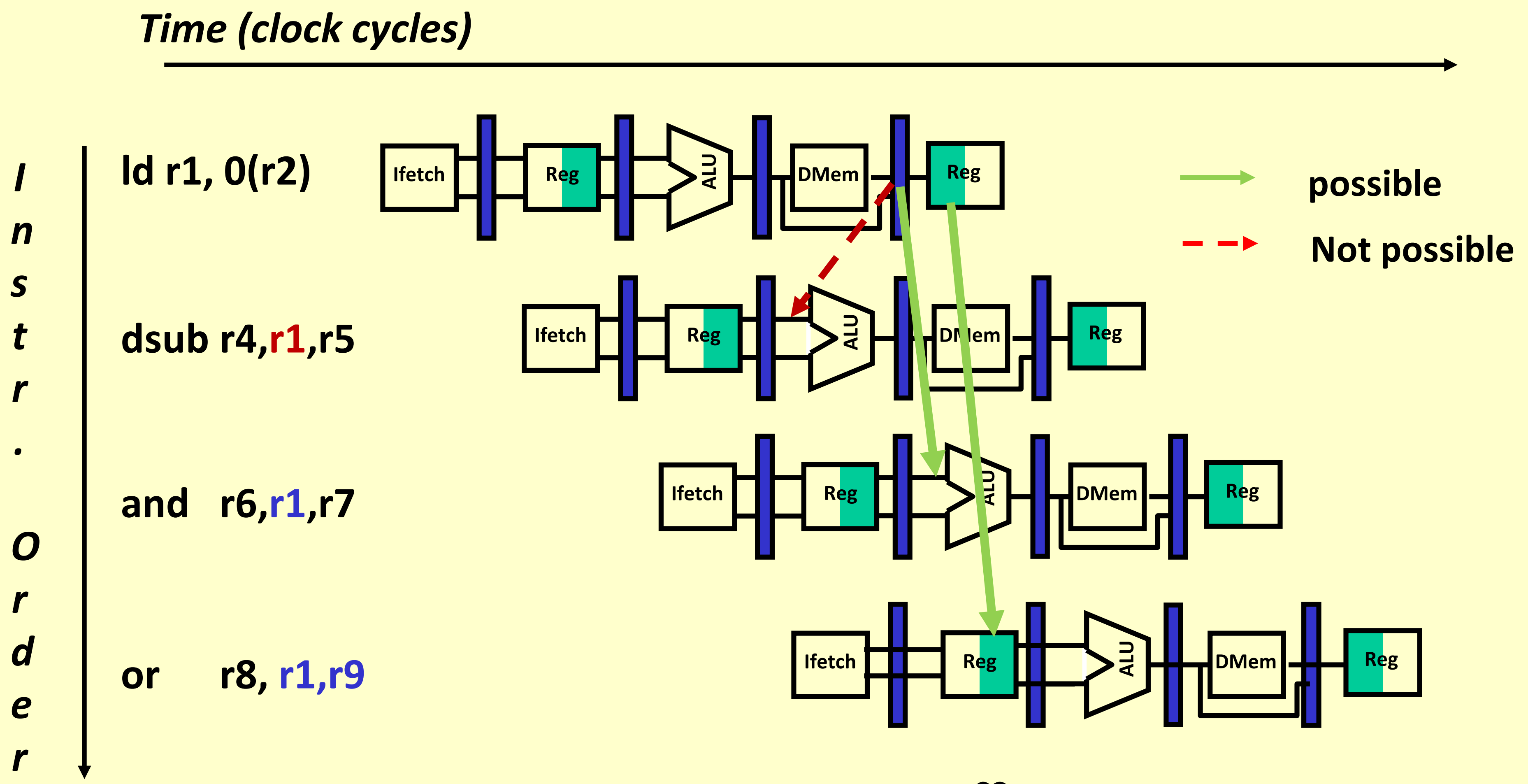
- **Keep track of three previous instructions**
- **If operand is *not* written by any of three previous instructions …**
  - … then read from register
- **If operand is output of ALU operation of *previous* instruction …**
  - … then forward from ALU stage
- **If operand is output from ALU or MEM of 2nd previous instruction …**
  - … then forward from MEM stage
- **Else stall one cycle**

Interlocking for Data Hazards (continued)



See Figure C.5

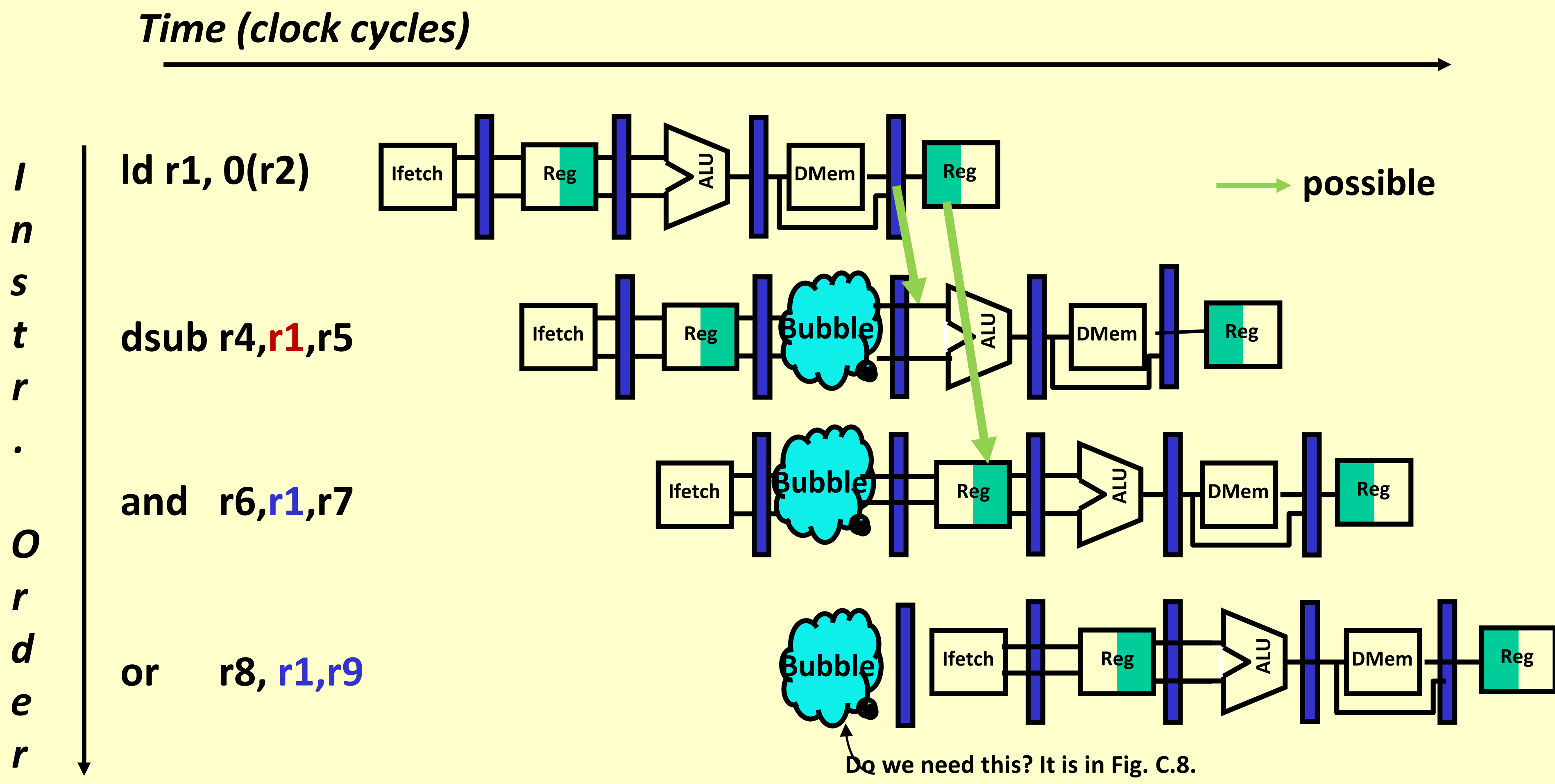
Data Hazards Requiring Stalls



See Figure C.7

Part of the solution

Data Hazards Requiring Stalls



Do we need this? It is in Fig. C.8.
Caption C.21 says "The only comparison needed is between the destination and the sources on the two instructions following the instruction that wrote the destination."

See Figure C.8

How to decide

- **Keep track of three previous instructions**
- **If operand is *not* written by any of three previous instructions …**
  - … then read from register
- **If operand is output of ALU operation of *previous* instruction …**
  - … then forward from ALU stage
- **If operand is output from ALU or MEM of 2nd previous instruction …**
  - … then forward from MEM stage
- **Else stall one cycle**

This entire decision tree can be programmed into gates in hardware

Questions?

Control Hazards

- Decision to take a *branch* or not may *not be known* until after the next instructions are fetched!

Control Hazard on Branches

(Three Stage Stall)

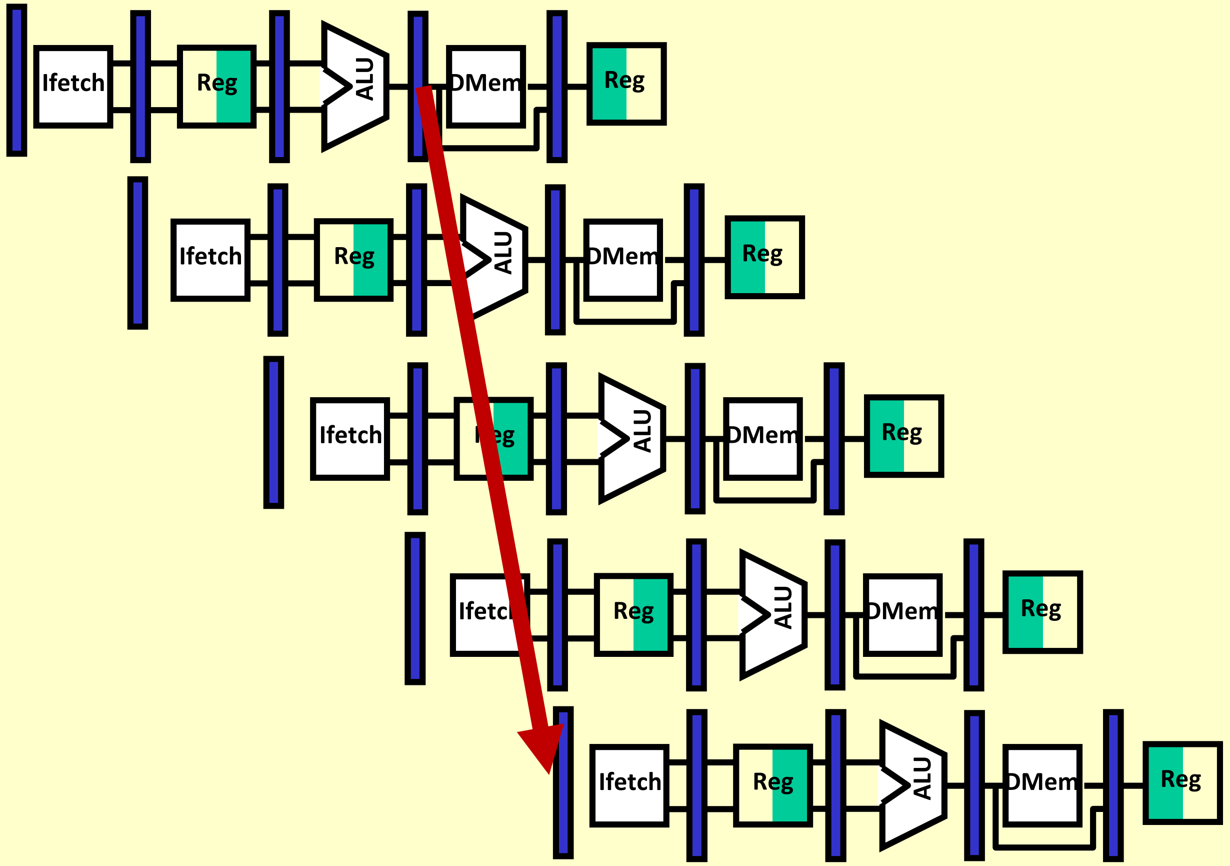
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



What do you do with the 3 instructions in between?

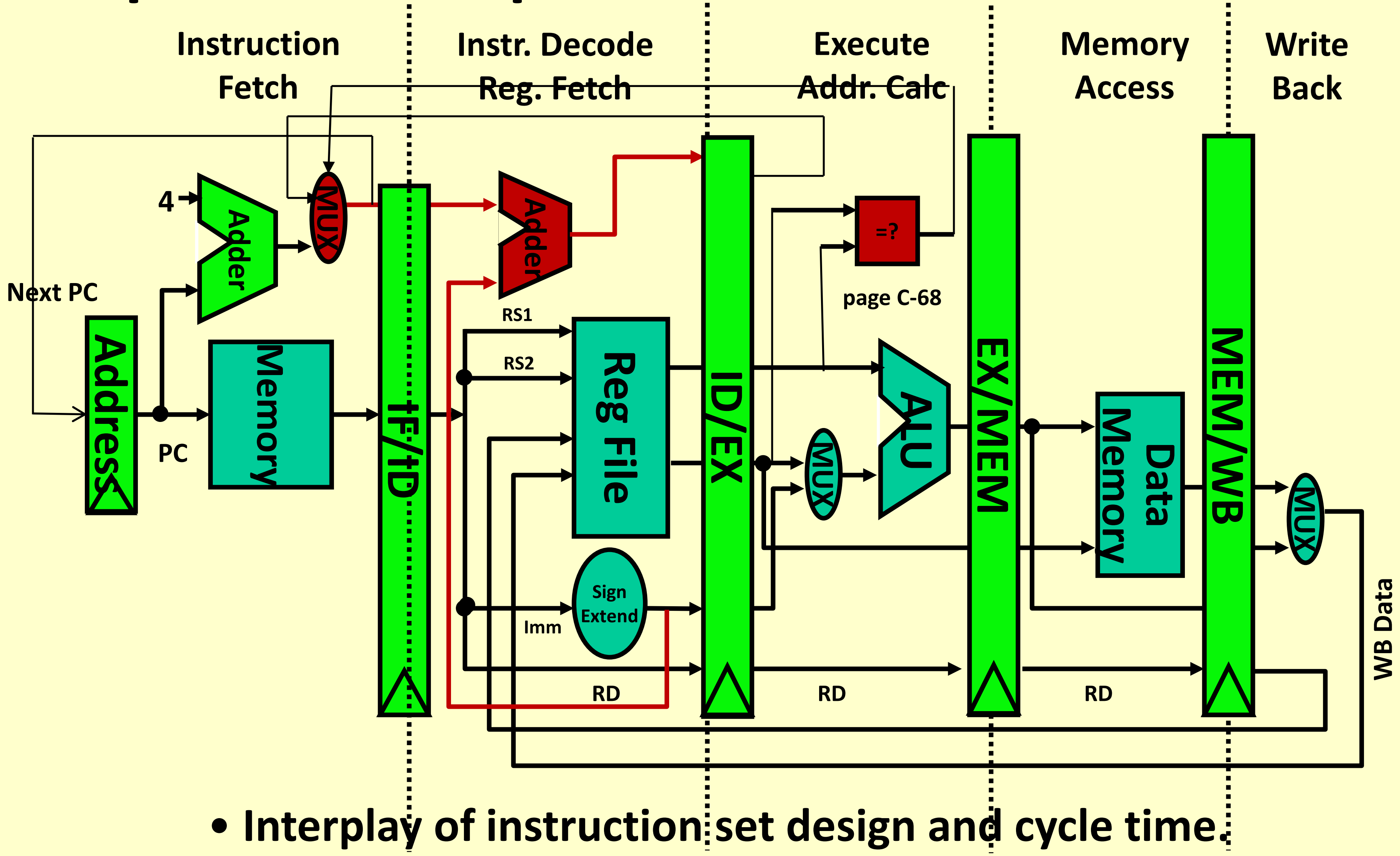
How do you do it?

Where is the “commit”? (we’ll discuss about 30 slides in the future)

Branch Stall Impact

- **If CPI = 1, 30% branch,**
  Stall 3 cycles => new CPI = 1.6!
- **Two part solution:–**
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- **MIPS branch tests if register = 0 or ≠ 0**
- **MIPS Solution:–**
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined PC Update (Figure C.25, page C-38)



- Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

- **#1: Stall until branch direction is clear**
- **#2: Predict Branch Not Taken**
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
- **#3: Predict Branch Taken**
  - 53% MIPS branches taken on average
  - **But haven't calculated branch target address in MIPS**
    - MIPS still incurs 1 cycle branch penalty
    - Other machines: branch target known before outcome
- **#4: (next slide)**

Speculative Execution

Four Branch Hazard Alternatives

#4: Delayed Branch

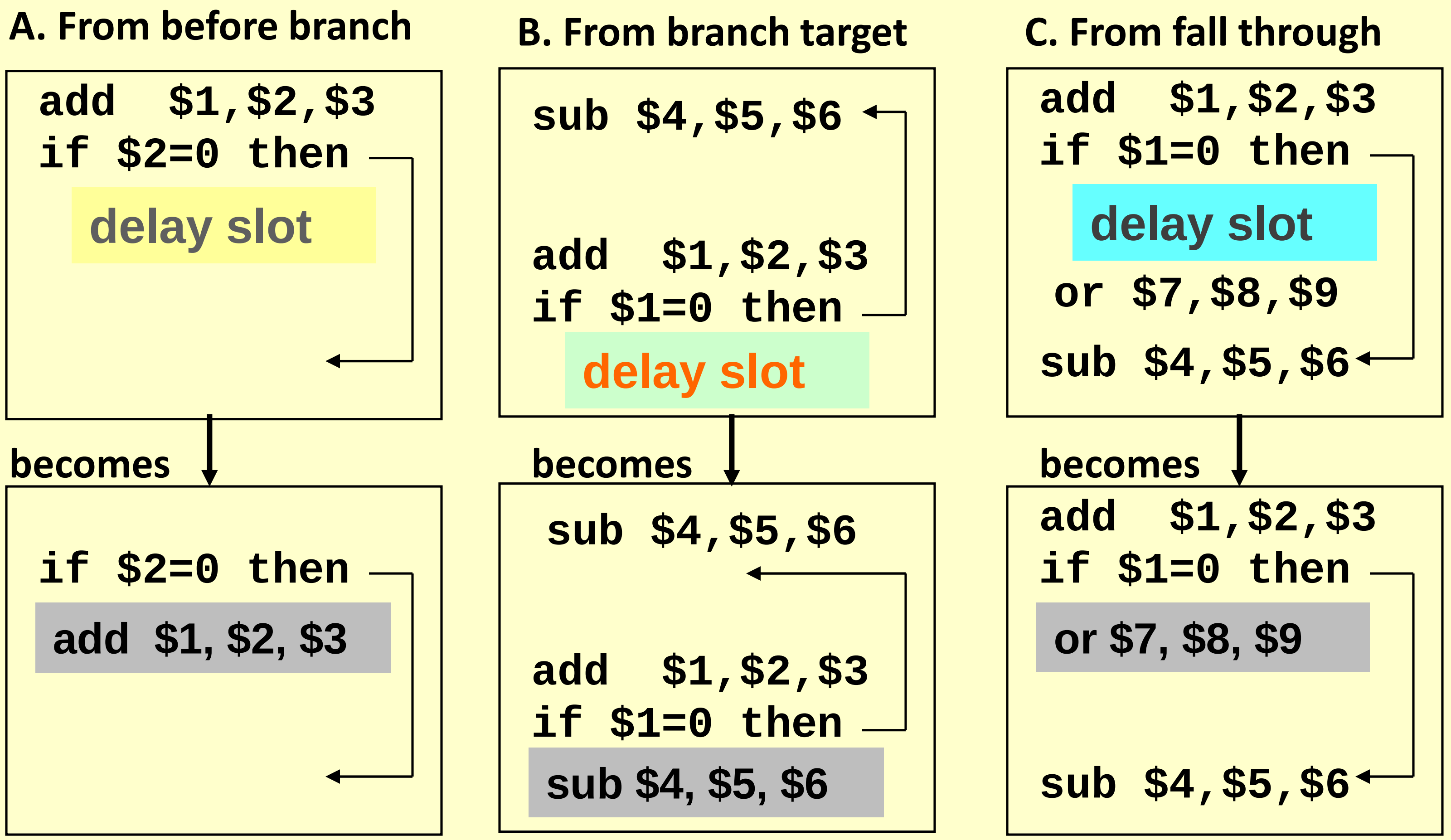
- Define branch to take place AFTER a following instruction

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

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots (Fig C.14)



- 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 instruction in delay slot whether or not the branch succeeds

Delayed Branch

- **Compiler effectiveness for single branch delay slot:**
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% × 80%) of slots usefully filled
- **…**

Delayed Branch — Downside

- **As processors go to deeper pipelines and multiple issue, 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

Evaluating Branch Alternatives

$$SpeedUp = \frac{PipelineDepth}{1 + BranchFreq \times BranchPenalty}$$

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken. Total 20%

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

See also Figure C.13

Questions?

Branch Prediction

- Necessary to improve performance in a pipeline.
- Can be done by the compiler or the hardware.
- *Static* prediction can be used to assist dynamic predictions.

Static Branch Prediction

- Simplest *static* prediction method is to assume a branch is taken
  - Helps to compile more efficient code
- Does not rely on information about executing code.
- Predicts always same direction for a branch during entire program execution
- High and variable misprediction rate

Static Branch Prediction (continued)

- **Alternatively, base on information from earlier runs**
  - Much more accurate
  - Reason: individual branches are highly biased one way or the other

- **Effectiveness depends on the accuracy and frequency of conditional branches.**

Mis-prediction rate (profile-based predictor)

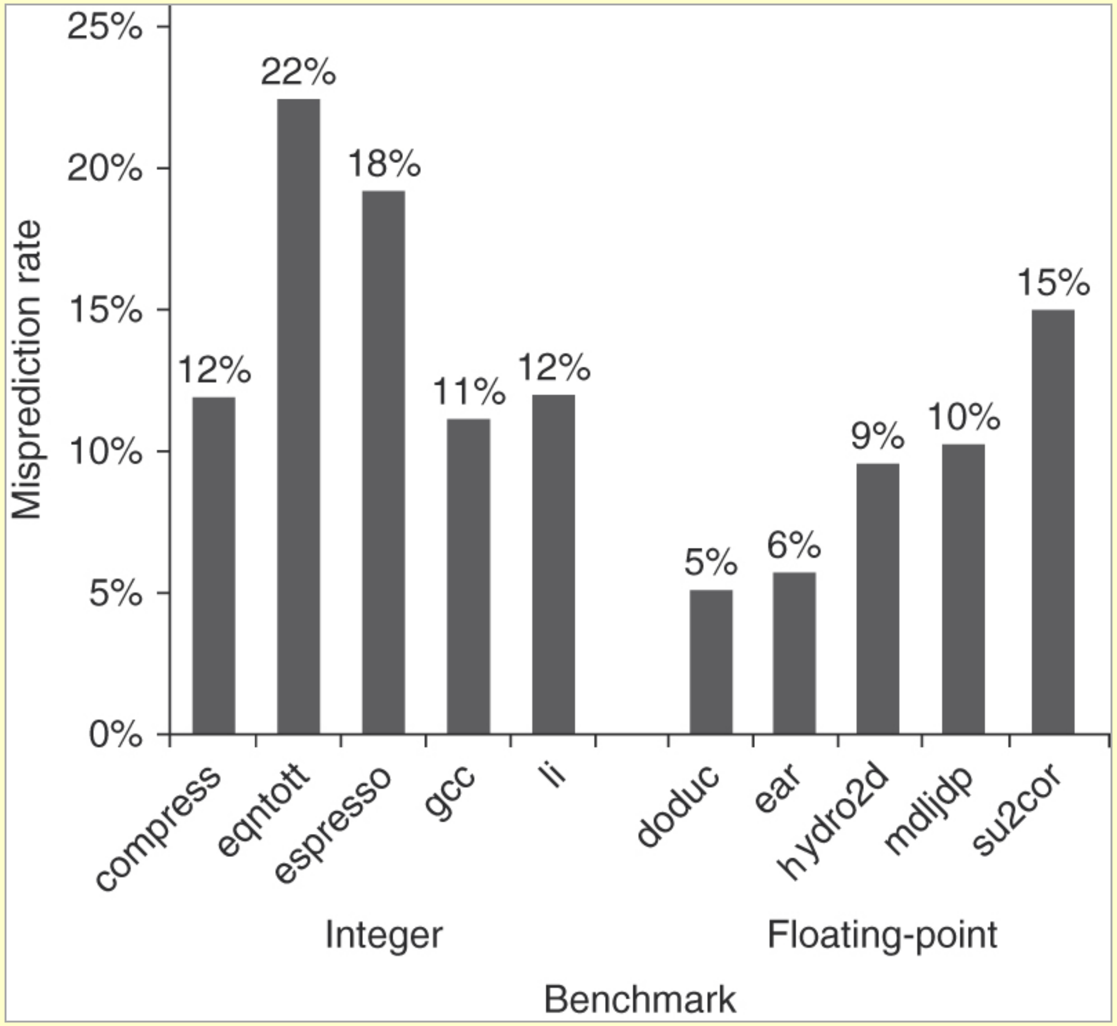


Figure C.14

Static Branch Prediction (continued)

- Alternatively, base on information from earlier runs
  - Much more accurate
  - Reason: individual branches are highly biased one way or the other

- Effectiveness depends on the accuracy and frequency of conditional branches.
- **Rarely used with modern processors.**

Why not?

Branch-prediction buffer

- **Simplest dynamic scheme ‘branch history table’**
  - 1-bit cache of recent branch instructions
- **Instructions in that direction are fetched**
- **If not correct**
  - Invert the bit and store it back into cache
- **Typically mispredicts twice**
  - Not just once!

Branch-Prediction Buffer (continued)

- **Improvement:–**
  - 2-bit cache entry,
  - Prediction must miss twice before change
  - See Fig C.15

- **Buffer implemented as a small, special "cache" accessed during the IF pipe stage**
  - Tag = address of branch instruction

2-bit Branch Prediction

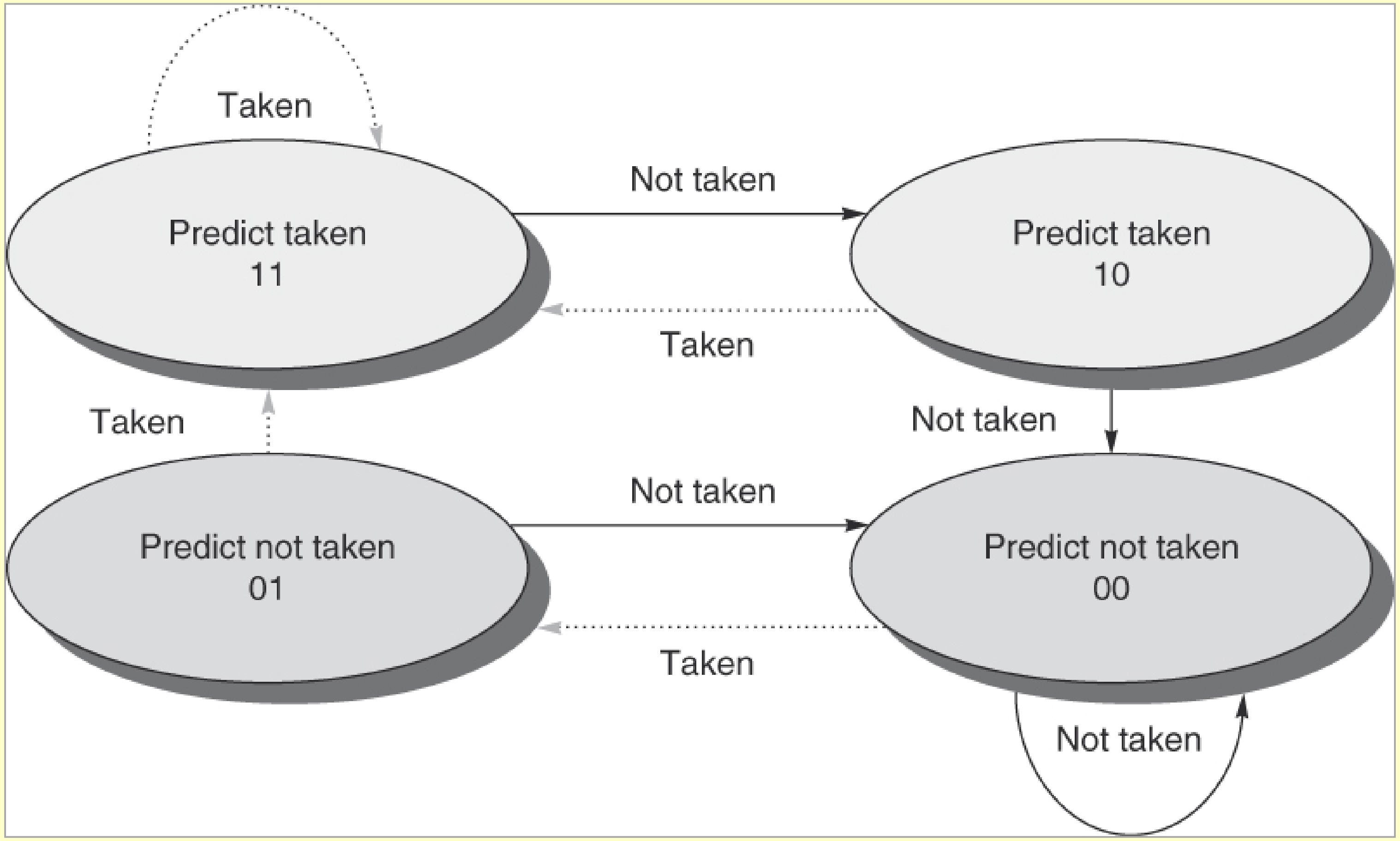


Figure C.15

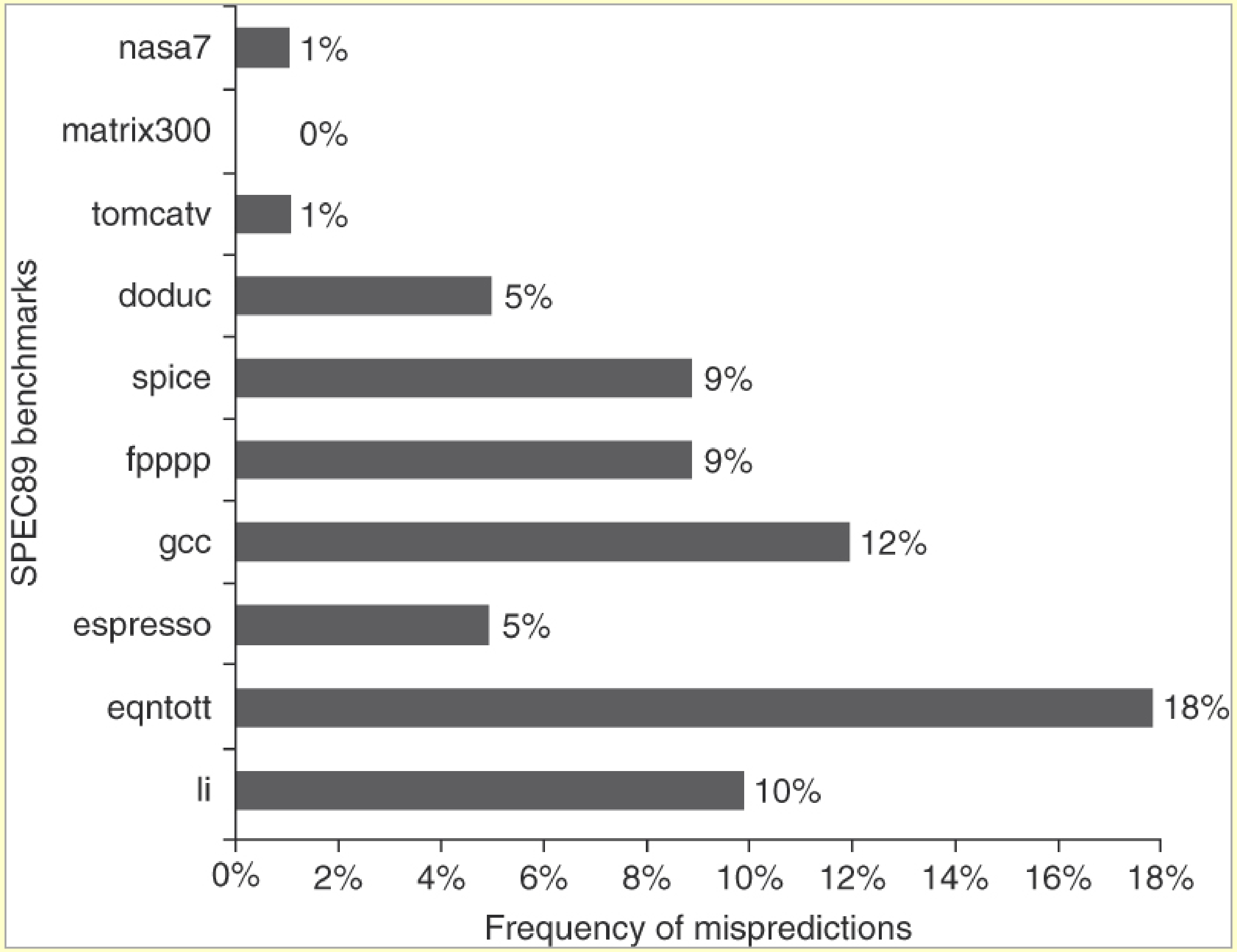
Example

```
…
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
…
```

Branch-prediction buffer (continued)

- **If predicted as ‘taken’, fetching of instructions begins as soon as PC is known**
  - Otherwise, sequential fetching continues.
  - When wrong prediction, bits are changed
- **With 4096 entries, 82%-99% accuracy rate in SPEC89 benchmarks**
    - See Fig C.17

Mis-prediction rates (2-bit prediction buffer)



Correlating predictors

- Increasing number of bits in the predictor => very little improvement
- 2-bit predictor only uses recent information for a single branch.
- Looking at behaviors from other branches can help improve prediction accuracy.

Correlating predictors (continued)

```
if(a==2) {
    a=0
}
if(b==2) {
    b=0
}
if(a!=b) {
    …
}
```

The last branch will only be taken if the first two are not taken as well, otherwise it should always execute.

Using the behavior of a single branch could never capture this behavior

Questions?

More Problems with Pipelining

- **Exception:– An unusual event happens to an instruction during its execution**
  - Examples: divide by zero, undefined opcode
- **Interrupt:– Hardware signal to switch the processor to a new instruction stream**
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **…**

More Problems with Pipelining (continued)

- **Requirement:– It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})**
  - The effect of all instructions up to and including I_i is totally complete
  - No effect of any instruction after I_i can take place
- **Interrupt or exception handler either aborts program or restarts at instruction I_{i+1}**

Questions?

CS-4515
Computer Architecture

Professor Hugh C. Lauer
CS-4515, System Programming Concepts

About the New Slides for Computer Architecture

CS-4515: System Programming Concepts

0th Lecture, March 17, 2017

Hugh C. Lauer

Department of Computer Science

On the Design

- **All slides are in Powerpoint 2007 (mix of PC and Mac versions)**
- **Probably could be edited using Powerpoint 2003 plus**
  - File format plugin
  - Calibri font
  - I would still recommend to use 2007 for editing
- **Design is suitable for printing out slides**
  - Only light colors, in particular for boxes
- **Some slides have covered areas (that disappear later) suitable for quizzing in class**
- **The design follows the Small Guide to Giving Presentations**

- **Next slides: Color/format conventions**

Style for title slides

System-Level I/O

CS-2011: Introduction to Machine Organization and Assembly Languag
14th Lecture, Oct. 12, 2010

Instructor:

Hugh C. Lauer

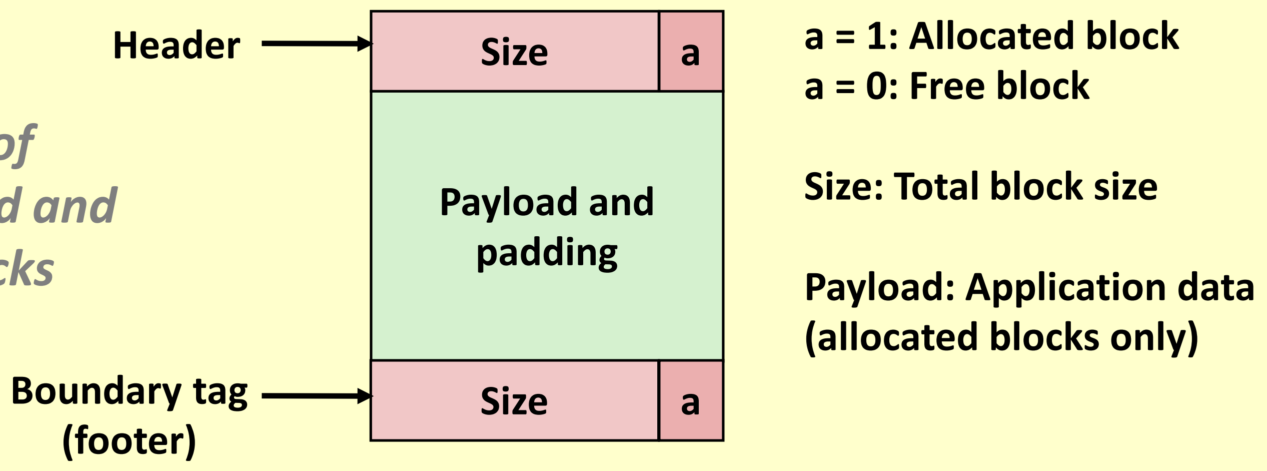
Style for outlining

Today

- Unix I/O
- RIO (robust I/O) package
- **Metadata, sharing, and redirection**
- Standard I/O
- Conclusions and examples

Style for Figure Labels

- **Capitalize only the first word in each figure label**
  - E.g., “Payload and padding”, not “Payload and Padding”, or “payload and padding”
  - This is the same style convention that we used in CS:APP2e.



Format of allocated and free blocks

Style for Code

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Style for Code and Alternative Code

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Style for Assembly Code: Version I

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp                  } Setup
    movl    %esp, %ebp            }
    movl    8(%ebp), %edx         }
    movl    12(%ebp), %eax        }
    cmpl    %eax, %edx            }
    jle     .L7                   } Body1
    subl    %eax, %edx            }
    movl    %edx, %eax            }
.L8:
    leave                         } Finish
    ret                           }
.L7:
    subl    %edx, %eax            } Body2
    jmp     .L8                   }
```

Style for Assembly Code: Version II

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

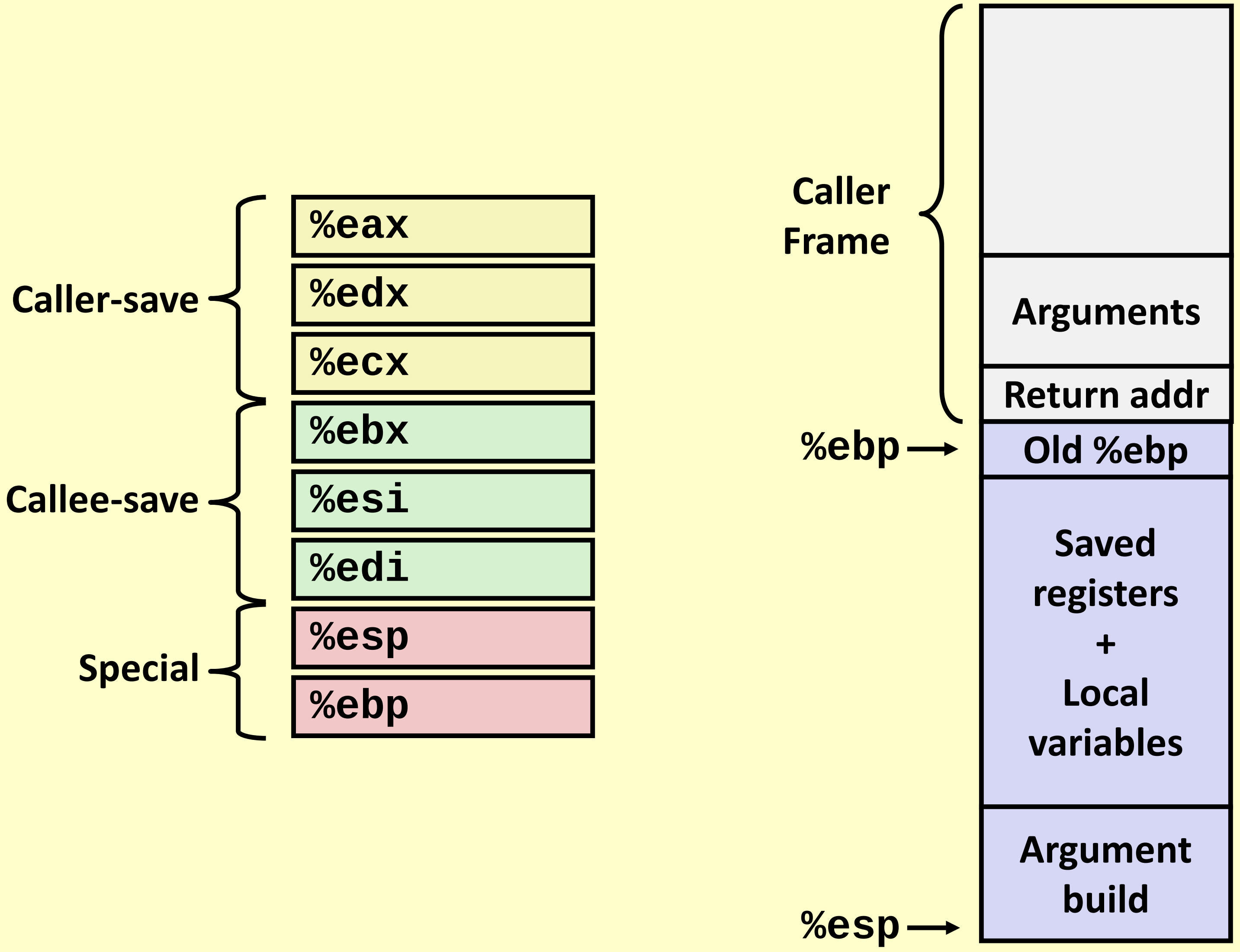
```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```

```
 # %edx = r
movl (%edx),%ecx        # r->i
leal 0(,%ecx,4),%eax    # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx)      # Update r->p
```

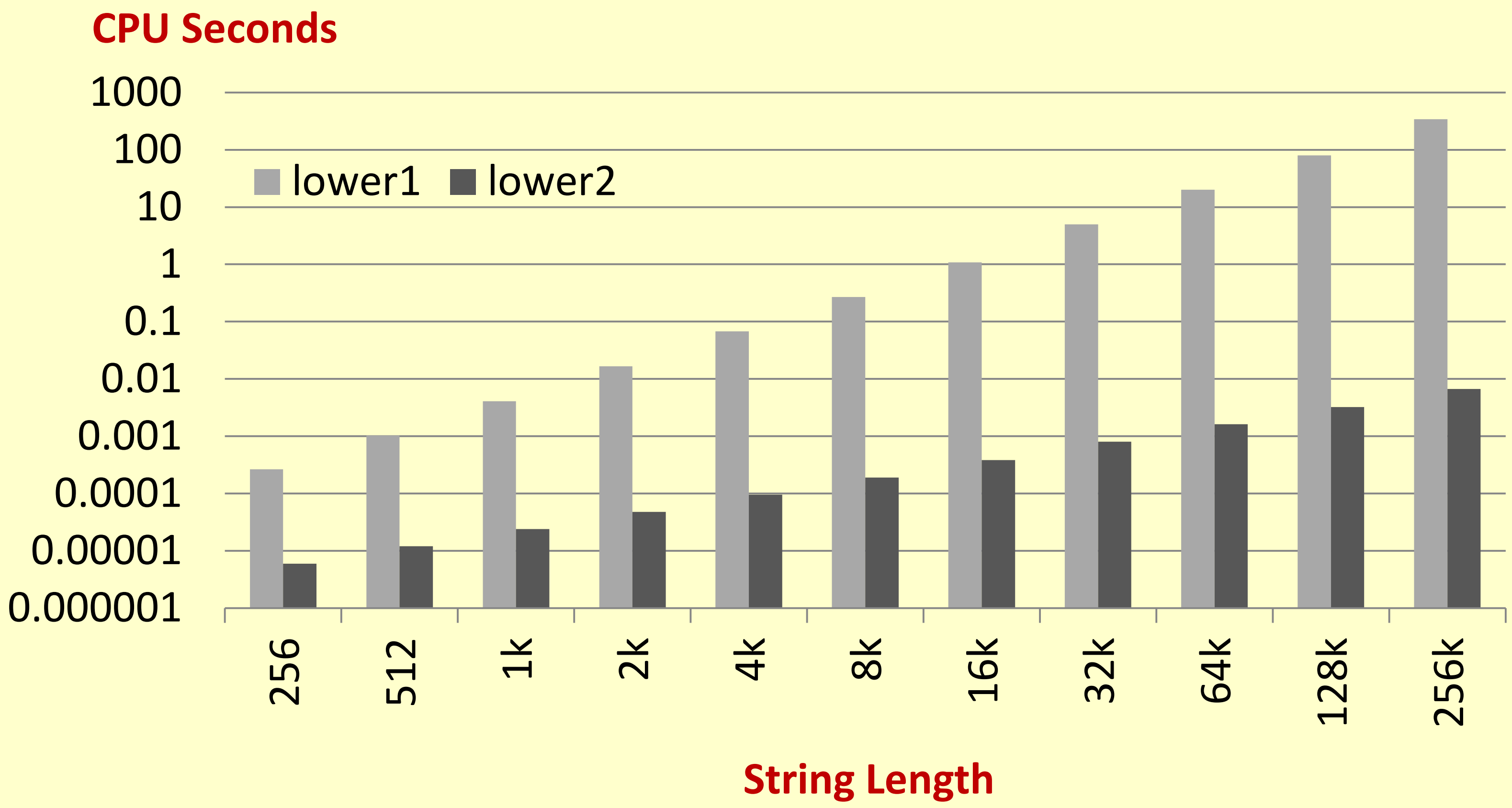
Linux Command Prompt

```
linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672
```

Stack and Registers



Bar Plot


CPU Seconds
1000
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
lower1
lower2
256
512
1k
2k
4k
8k
16k
32k
64k
128k
256k
String Length

Tables

Cycles per element (or per mult)

Machine	Nocona	Core 2
rfact	15.5	6.0
fact	10.0	3.0

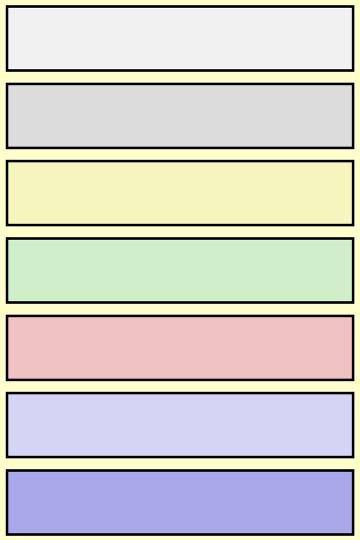
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
bound	1.0	1.0	2.0	2.0

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	**36/106**	**36/106**
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	**32/46**	**32/46**

Color Palette

- **Boxes/areas:**
  - Assembly, memory, …
  - Linux, memory, …
  - Code, …
  - Code, registers, …
  - Registers, …
  - Memory, …
  - Memory, …



- **Occasionally, use darker versions of above colors**
- **Text:**
  - Emphasizing something in the text
  - Comments inside yellow code boxes