

COURSE OVERVIEW

**Classes on
Sunday,
Monday and
Wednesday**

3 Cr. Hr.



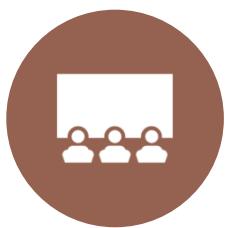
GRADING OVERVIEW



TOTAL MARKS
300



ATTENDANCE 30
(10%)



CLASS TEST 60
(20%)



TERM FINAL
EXAM 210 (70%)



INSTRUCTOR

Name: Saem Hasan

Designation: Lecturer, CSE, BUET

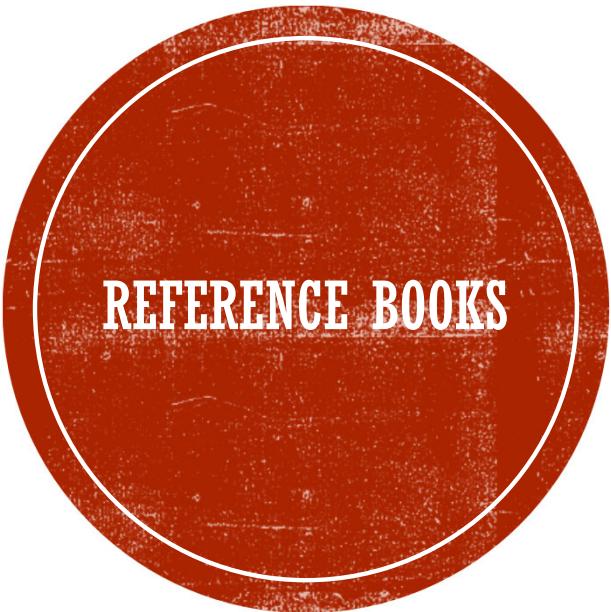
Office Room: 516 (CSE Wing, ECE Building)

Contact: +8801626187505

Email: saem@teacher.cse.buet.ac.bd

Email: saemhasan027@gmail.com





Computer Organization and Design (5th Edition) by David A Patterson and John L. Hennessy

Digital Logic and Computer Design by M. MORRIS MANO

Computer Systems A Programmer's Perspective (3rd Edition)

CSE 306

Computer Architecture

Design of an ALU

Prepared by
Madhusudan Basak
Assistant Professor
CSE, BUET

*Some modifications made by Saem Hasan



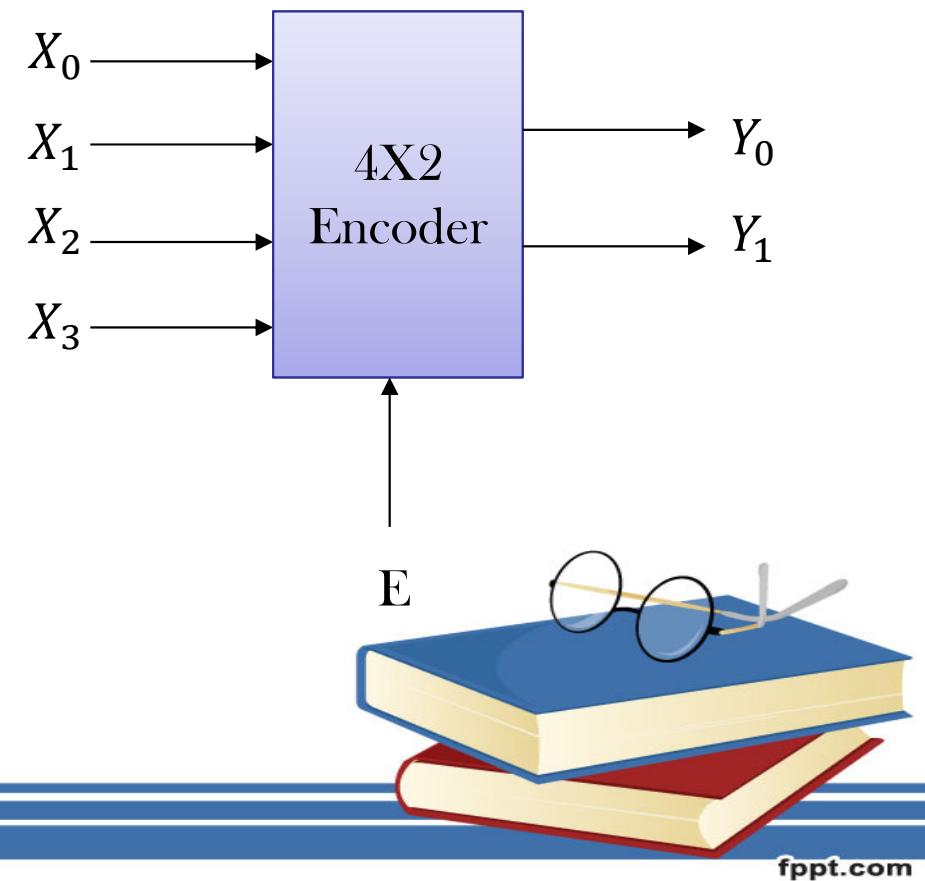
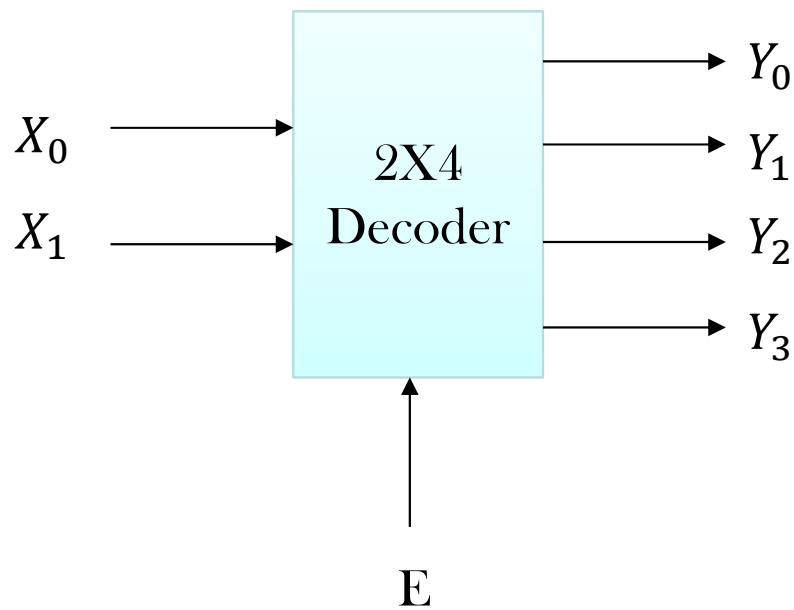
ALU and the Processor

- ALU stands for Arithmetic Logic Unit
 - A part of the processor or CPU



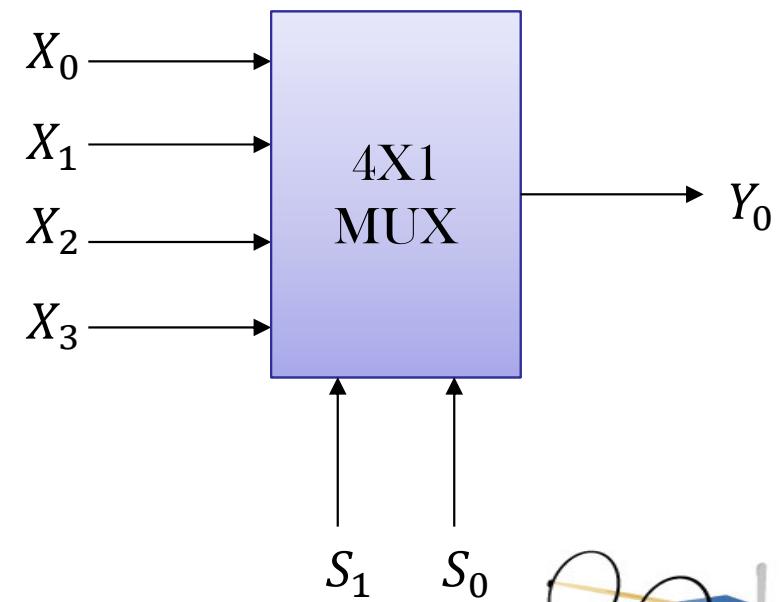
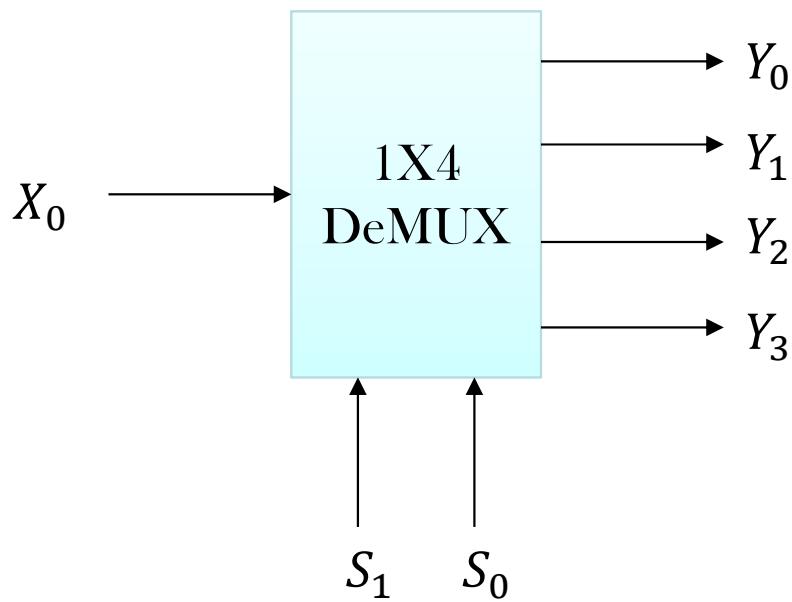
Some Terminologies Revisiting

- Decoder, Encoder



Some Terminologies Revisiting

□ DeMUX, MUX



A Simplified Processor in Operation

$$R_1 \leftarrow R_2 + R_3$$

0000 0001 0010 0011

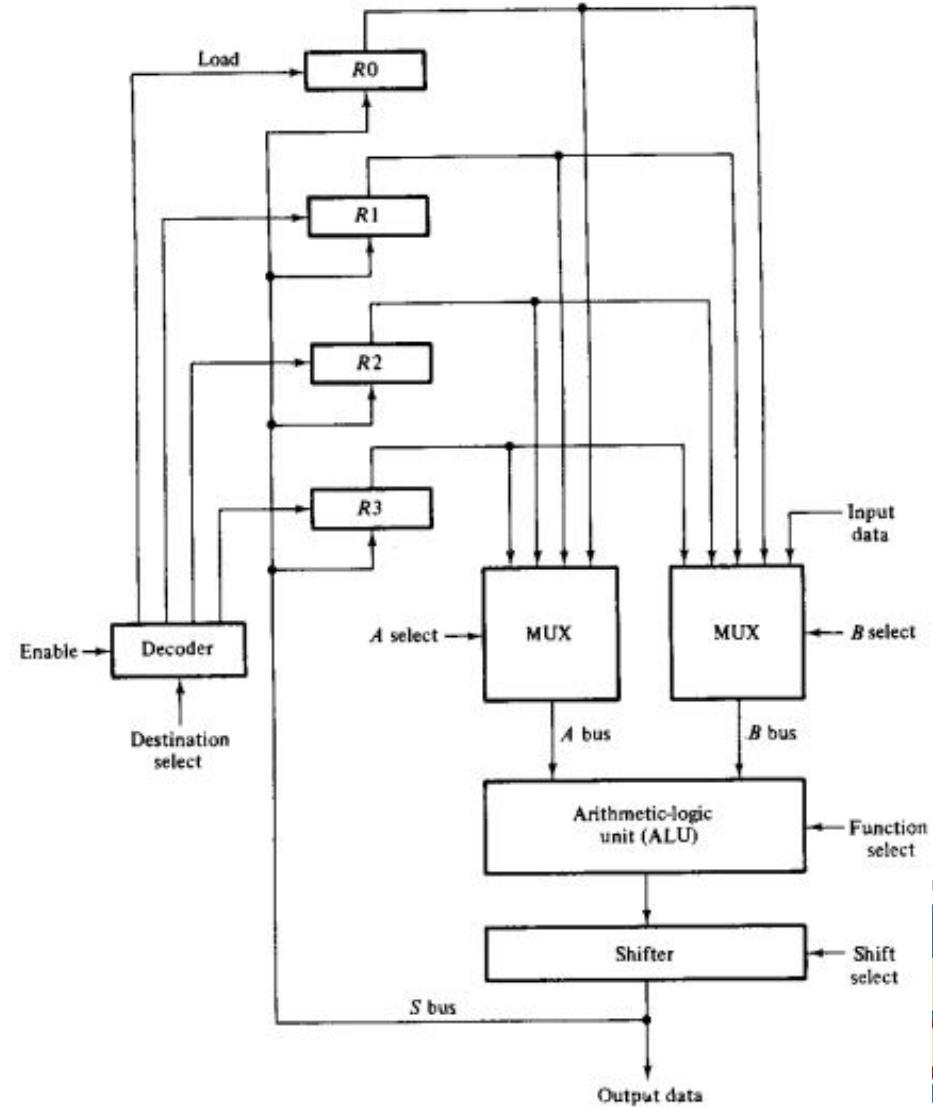


Figure 9-1 Processor registers and ALU connected through common buses

Simplified Processor with Scratchpad Memory

$$R_1 \leftarrow R_2 + R_3$$

- $T_1: A \leftarrow M[010]$ read R_2 into register A
 $T_2: B \leftarrow M[011]$ read R_3 into register B
 $T_3: M[001] \leftarrow A + B$ perform operation in ALU
and transfer result to R_1

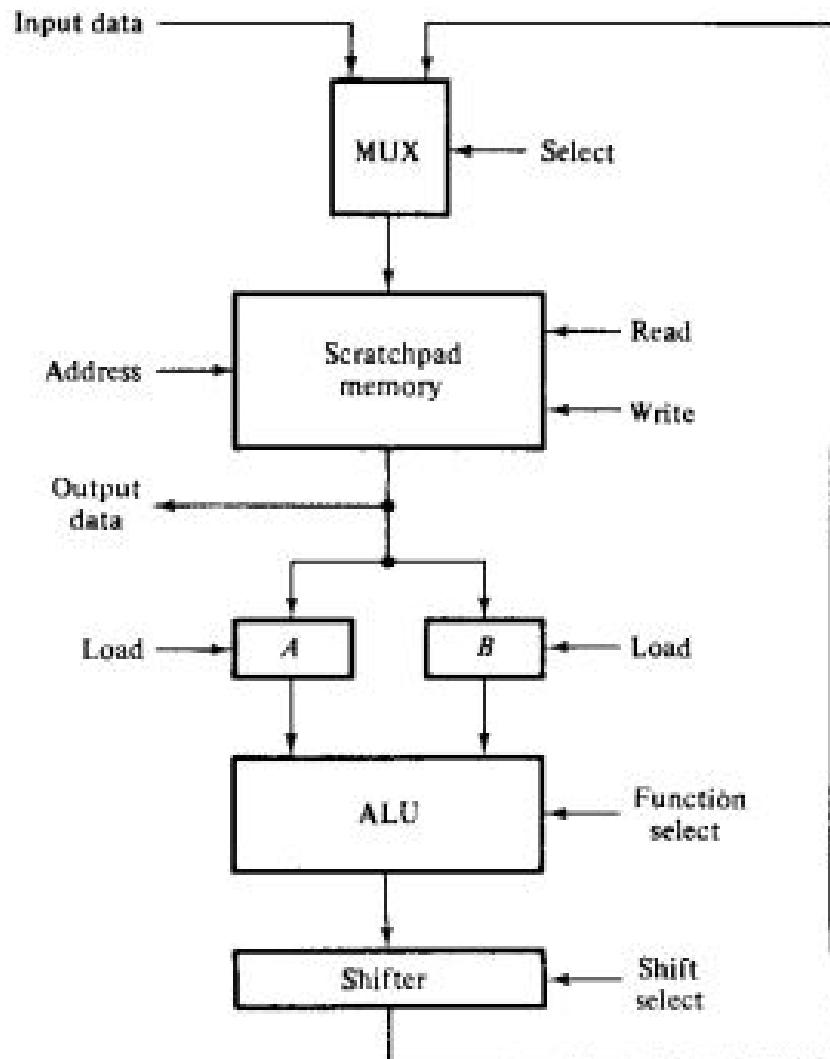


Figure 9-2 Processor unit employing a scratchpad memory

Simplified Processor with Scratchpad Memory

$$R_1 \leftarrow R_2 + R_3$$

- $T_1: A \leftarrow M[010]$ read R_2 into register A
 $T_2: B \leftarrow M[011]$ read R_3 into register B
 $T_3: M[001] \leftarrow A + B$ perform operation in ALU
and transfer result to R_1

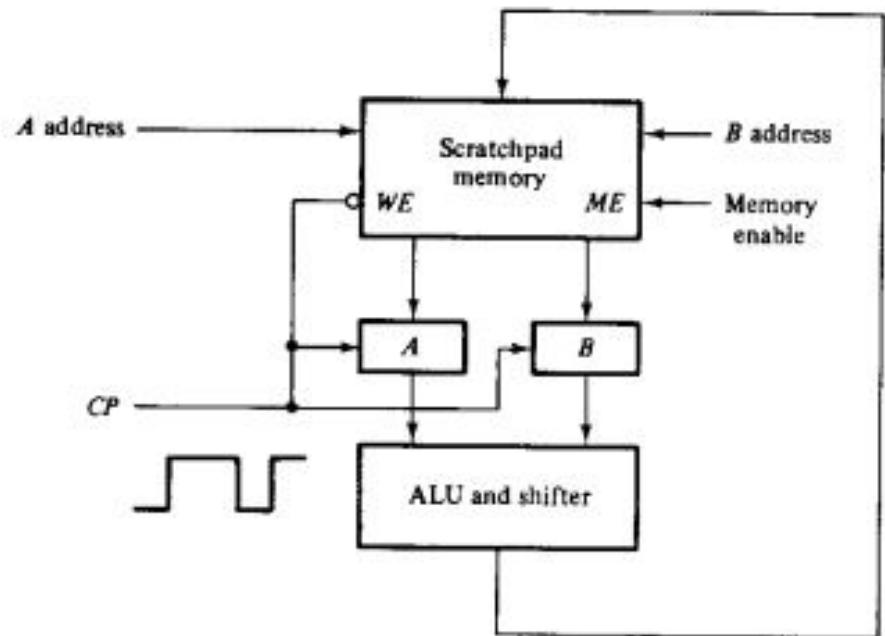


Figure 9-3 Processor unit with a 2-port memory



Simplified Processor with Accumulator

$$R_1 \leftarrow R_2 + R_3$$

$T_1: A \leftarrow 0$ clear A

$T_2: A \leftarrow A + R1$ transfer $R1$ to A

$T_3: A \leftarrow A + R2$ add $R2$ to A

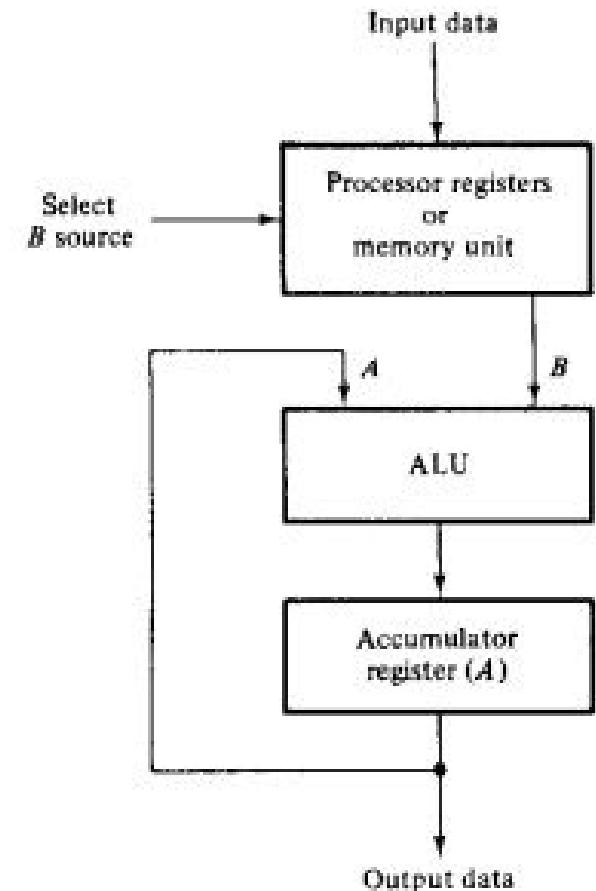
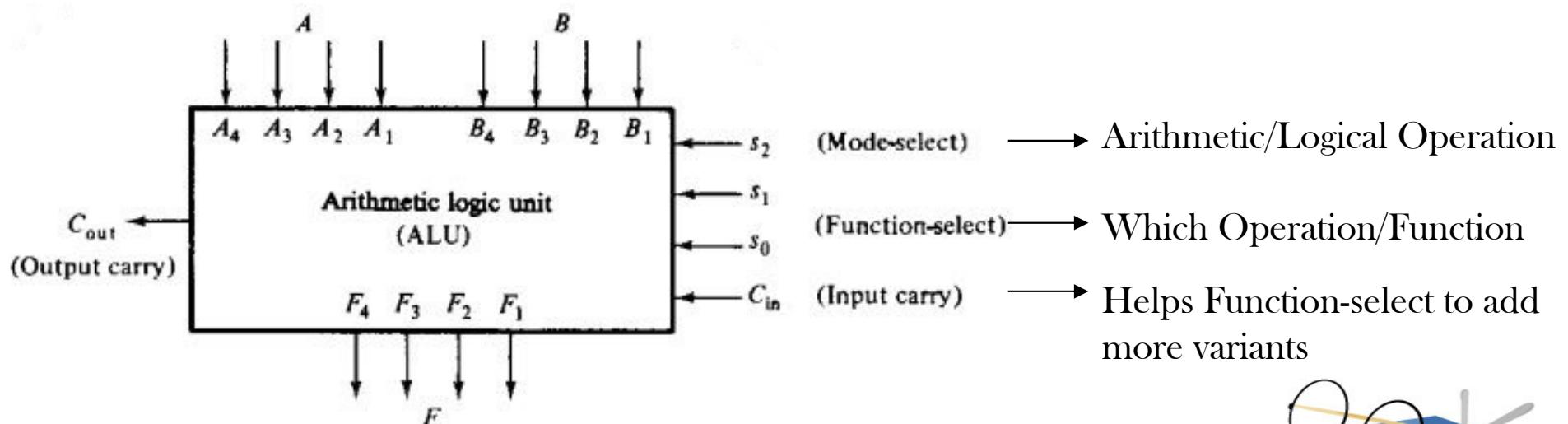


Figure 9-4 Processor with an accumulator register

An ALU Unit

- Can perform both Arithmetic and Logic Operations



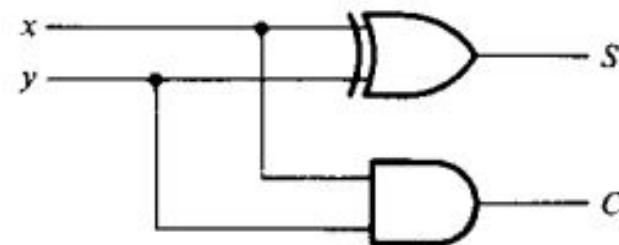
Parallel Adder

- Parallel adder
 - A number of full-adder circuits connected in cascade



1-bit Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

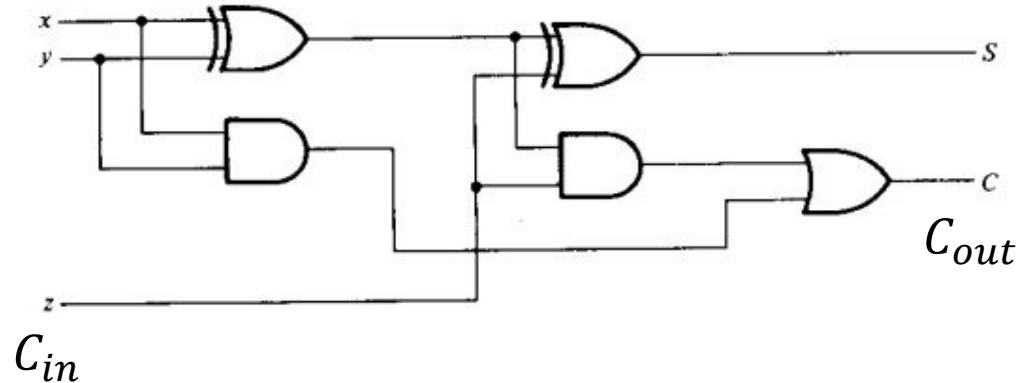


- $S = xy' + x'y = x \oplus y$
- $C = xy$

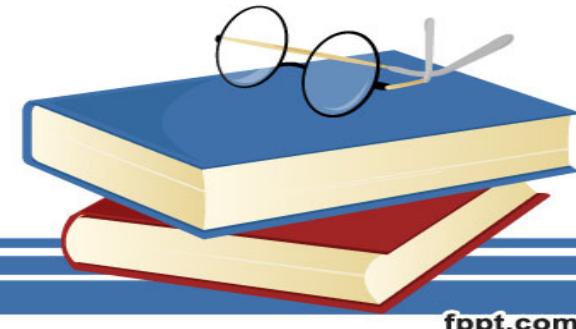


1-bit Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

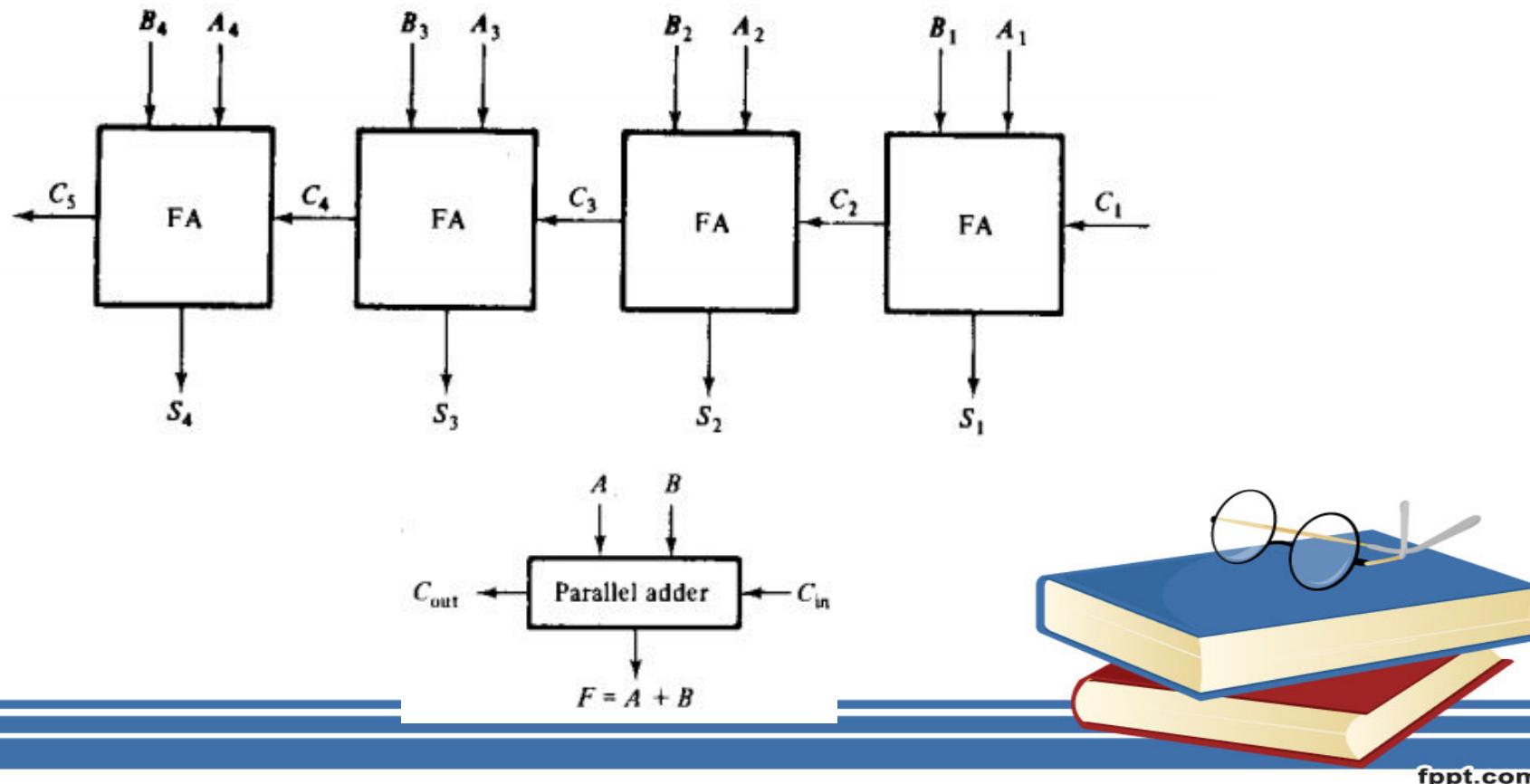


- $S = x'y'z + x'yz' + xy'z' + xyz$
- $C = xy + xz + yz$

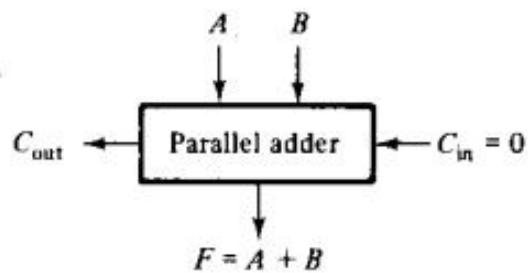


Parallel Adders

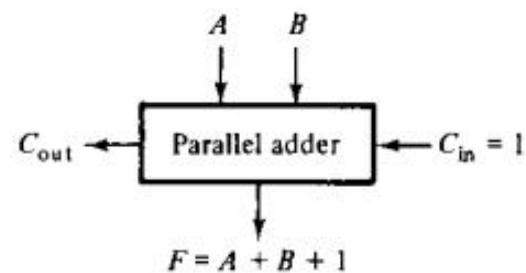
- n number of 1-bit full-adders are connected in cascade to form a n -bit parallel adder



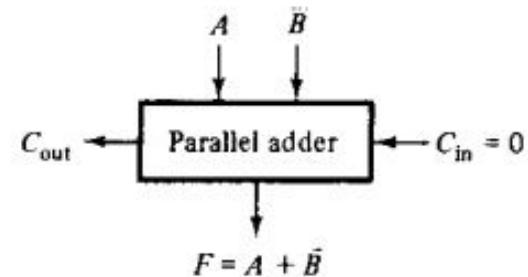
Arithmetic Operations by ALU



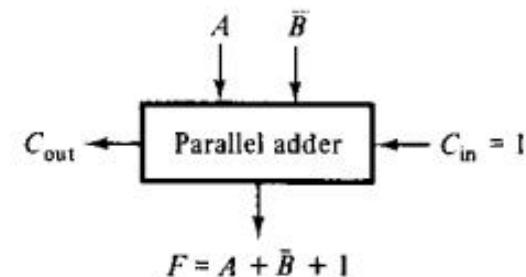
(a) Addition



(b) Addition with carry



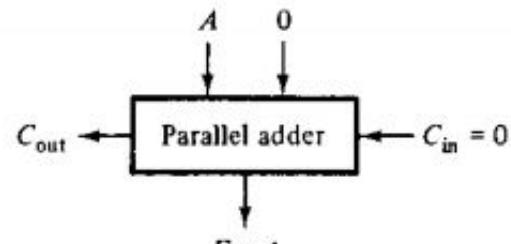
(c) A plus 1's complement of B



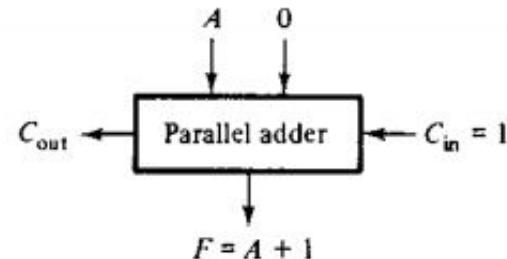
(d) Subtraction



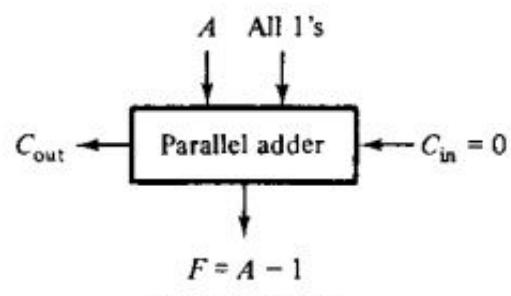
Arithmetic Operations by ALU



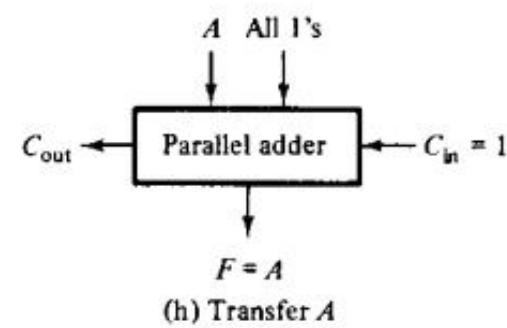
(e) Transfer A



(f) Increment A



(g) Decrement A



(h) Transfer A

Figure 9-6 Operations obtained by controlling one set of inputs to a parallel adder



What are we doing?

- Keeping A fixed and changing B to generate different operations
- Changes in B
 - Keeping B as it is
 - Inverting all bits of B
 - Changing each bit of B to 0
 - Changing each bit of B to 1
- Let's assume Y_i represents modified representation of B_i and thus Y represents B
- So, following 4 combinations can be obtained
 - Keeping B as it is ($Y_i = B_i$)
 - Inverting all bits of B ($Y_i = B_i'$)
 - Changing each bit of B to 0 ($Y_i = 0$)
 - Changing each bit of B to 1 ($Y_i = 1$)



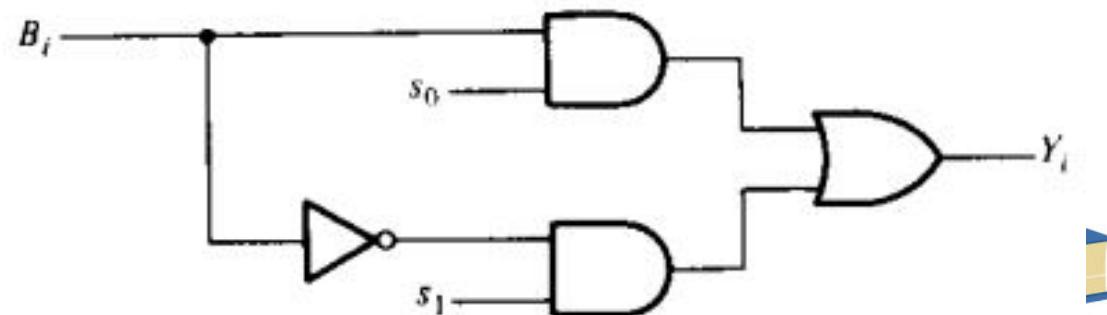
What are we doing?

- So Following 4 combinations can be obtained

- Keeping B as it is ($Y_i = B_i$)
- Inverting all bits of B ($Y_i = B_i'$)
- Changing each bit of B to 0 ($Y_i = 0$)
- Changing each bit of B to 1 ($Y_i = 1$)

s_1	s_0	Y_i
0	0	0
0	1	B_i
1	0	B_i'
1	1	1

- $Y_i = B_i s_0 + B_i' s_1$



Function Table

s_1	s_0	Y_i
0	0	0
0	1	B_i
1	0	B'_i
1	1	1

Modified B

Function select			Y equals	Output equals	Function
s_1	s_0	C_{in}			
0	0	0	0	$F = A$	Transfer A
0	0	1	0	$F = A + 1$	Increment A
0	1	0	B	$F = A + B$	Add B to A
0	1	1	B	$F = A + B + 1$	Add B to A plus 1
1	0	0	\bar{B}	$F = A + \bar{B}$	Add 1's complement of B to A
1	0	1	\bar{B}	$F = A + \bar{B} + 1$	Add 2's complement of B to A
1	1	0	All 1's	$F = A - 1$	Decrement A
1	1	1	All 1's	$F = A$	Transfer A



Function Table: Designer's perspective

s_1	s_0	Y_i
0	0	0
0	1	B_i
1	0	B'_i
1	1	1

Modified B

Function select			Y equals	Output equals	Function
s_1	s_0	C_{in}			
0000	0000	0000	0000	0000	Transfer A
0001	0010	0000	0000	0000	Increment A
0010	0100	0000	0000	0000	Add B to A
0011	0110	0000	0000	0000	Add B to A plus 1
0100	1000	0000	0000	0000	Add 1's complement of B to A
0101	1010	0000	0000	0000	Add 2's complement of B to A
0110	1100	0000	0000	0000	Decrement A
0111	1110	0000	0000	0000	Transfer A



Logic Diagram of an independent Arithmetic Circuit

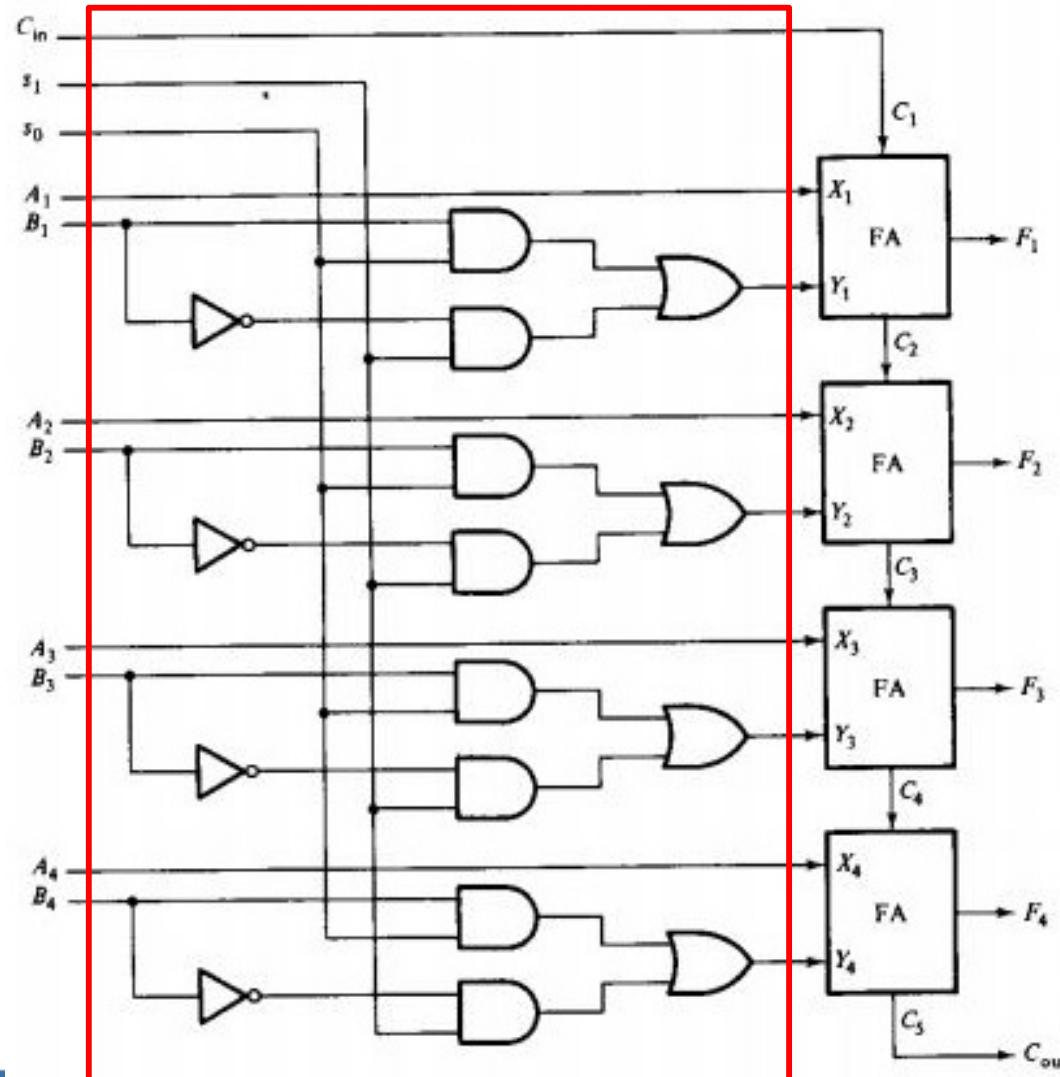
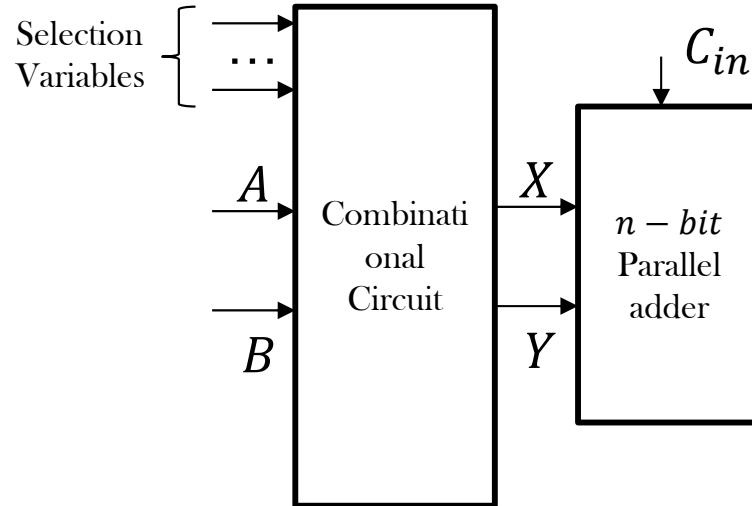


Figure 9-8 Logic diagram of arithmetic circuit

Design Example

- Design an adder/subtractor circuit with one selection variable s and two inputs A and B . When $s=0$, the circuit performs $A+B$. When $s=1$, the circuit performs $A-B$ by taking the 2's complement of B .
- Functions

$$F = A + B$$

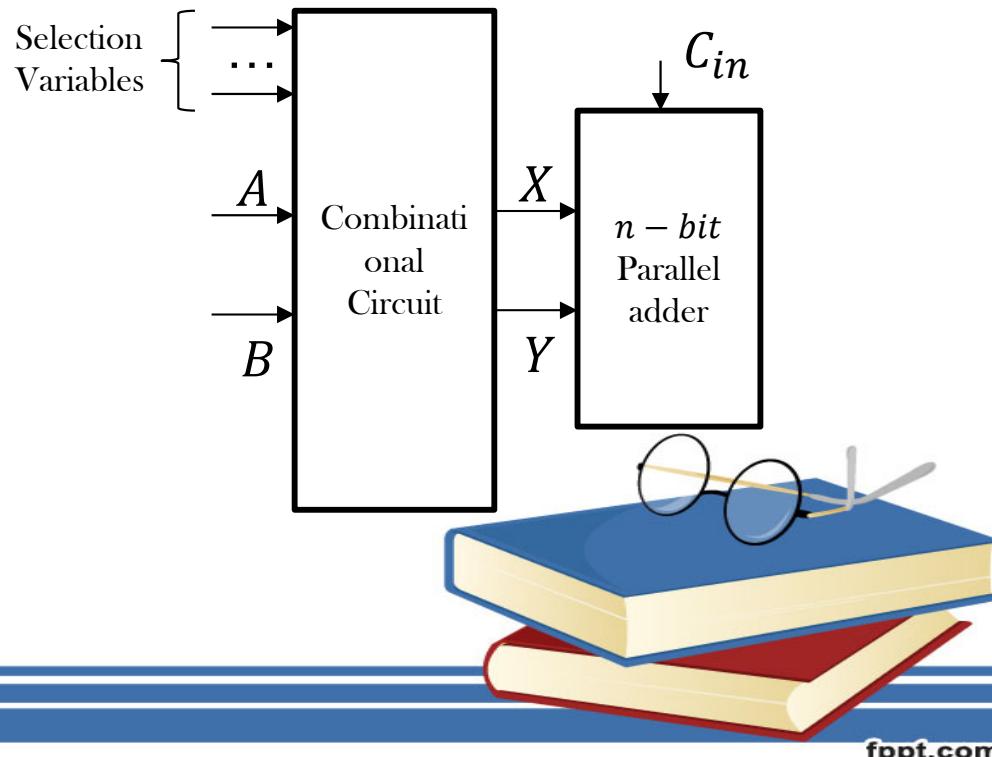
$$\begin{aligned} F &= A - B \\ &= A + \bar{B} + 1 \end{aligned}$$

s	Y_i
0	B_i
1	B'_i

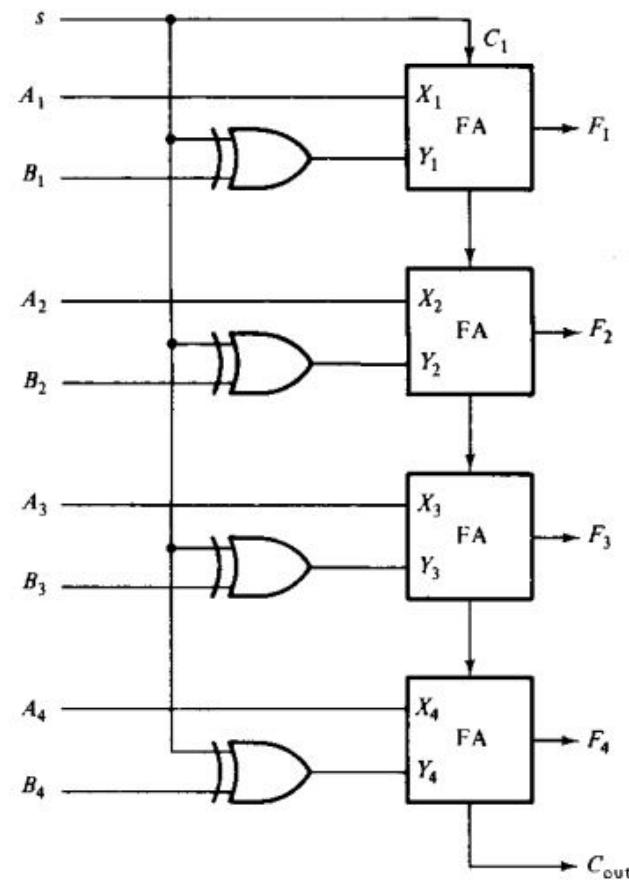
s	C_{in}
0	0
1	1

$$Y_i = s'B_i + sB'_i = s \oplus B_i$$

$$C_{in} = s$$

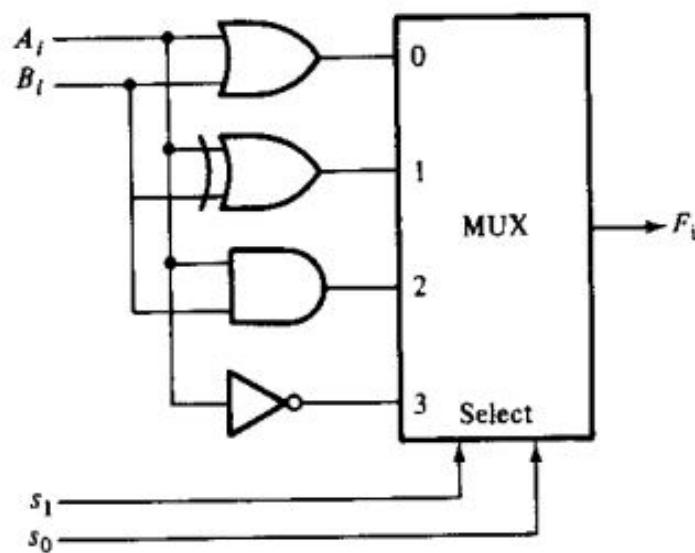


Design Example



Designing a Logic Circuit

- We shall implement three basic logical operations (AND, OR and NOT) and an XOR operation

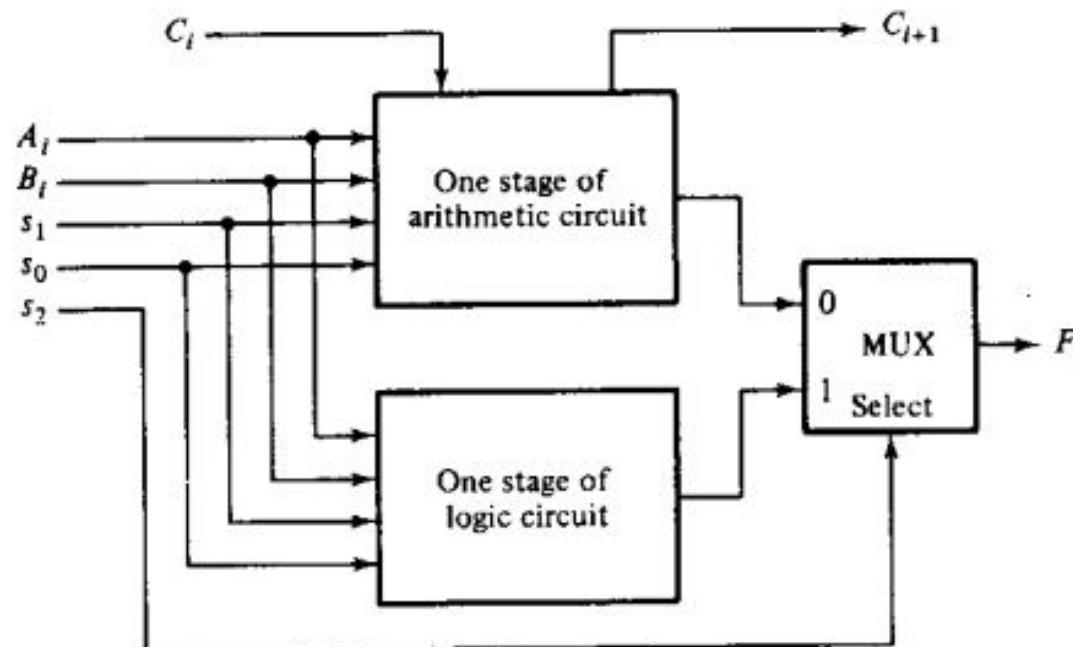


s_1	s_0	Output	Operation
0	0	$F_i = A_i + B_i$	OR
0	1	$F_i = A_i \oplus B_i$	XOR
1	0	$F_i = A_i B_i$	AND
1	1	$F_i = A_i'$	NOT



Designing a Logic Circuit

- Let's combine it with arithmetic operations



More Efficient Design

- ❑ Use already available arithmetic circuit and incorporate logical operations
- ❑ Procedure
 - Design the arithmetic section independently
 - Take the circuit, consider $C_{in} = 0$, and determine which logic operations are automatically generated from the arithmetic circuit
 - Modify the circuit to incorporate required but not automatically generated logic operations



More Efficient Design

- Use already available arithmetic circuit and incorporate logical operations

s_1	s_0	Y_i
0	0	0
0	1	B_i
1	0	B'_i
1	1	1

s_1	s_0	X_i	Y_i
0	0	A_i	0
0	1	A_i	B_i
1	0	A_i	B'_i
1	1	A_i	1

Required
operation

OR
XOR
AND
NOT



Incorporating remaining functions

□ Unresolved cases

s_2	s_1	s_0	X_i	Y_i	Automatically Obtained F_i	Required F_i
1	0	0	A_i	0	$F_i = A_i$	$F_i = A_i + B_i$
1	1	0	A_i	B'_i	$F_i = A_i \odot B_i$	$F_i = A_i B_i$



Incorporating remaining functions

From Table 9-3, we see that when $s_2 = 1$, the input carry C_i in each stage must be 0. With $s_1 s_0 = 00$, each stage as it stands generates the function $F_i = A_i$. To change the output to an OR operation, we must change the input to each full-adder circuit from A_i , to $A_i + B_i$. This can be accomplished by ORing B_i and A_i when $s_2 s_1 s_0 = 100$.

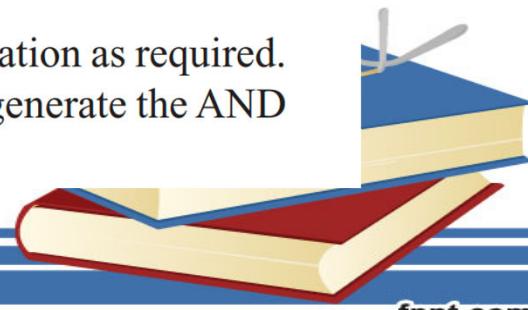
The other selection variables that give an undesirable output occur when $s_2 s_1 s_0 = 110$. The unit as it stands generates an output $F_i = A_i \odot B_i$ but we want to generate the AND operation $F_i = A_i B_i$. Let us investigate the possibility of ORing each input A_i with some Boolean function K_i . The function so obtained is then used for X_i when $s_2 s_1 s_0 = 110$:

$$F_i = X_i \oplus Y_i = (A_i \oplus K_i) \oplus B'_i = A_i B_i + K_i B_i + A'_i K'_i B'_i \\ (\text{A} + \text{K})$$

Careful inspection of the result reveals that if the variable $K_i = B'_i$, we obtain an output:

$$F_i = A_i B_i + B'_i B_i + A_i B_i B'_i = A_i B_i$$

Two terms are equal to 0 because $B_i B'_i = 0$. The result obtained is the AND operation as required. The conclusion is that, if A_i is ORed with B'_i when $s_2 s_1 s_0 = 110$, the output will generate the AND operation.



Final Boolean Functions

- Combining the arithmetic and logical cases, we get the final form of the Boolean function as:

$$X_i = A_i + s_2 s'_1 s'_0 B_i + s_2 s_1 s'_0 B'_i$$

$$Y_i = s_0 B_i + s_1 B'_i$$

$$Z_i = s'_2 C_i$$

Selection				Output	Function
s_2	s_1	s_0	C_{in}		
0	0	0	0	$F = A$	Transfer A
0	0	0	1	$F = A + 1$	Increment A
0	0	1	0	$F = A + B$	Addition
0	0	1	1	$F = A + B + 1$	Add with carry
0	1	0	0	$F = A - B - 1$	Subtract with borrow
0	1	0	1	$F = A - B$	Subtraction
0	1	1	0	$F = A - 1$	Decrement A
0	1	1	1	$F = A$	Transfer A
1	0	0	X	$F = A \vee B$	OR
1	0	1	X	$F = A \oplus B$	XOR
1	1	0	X	$F = A \wedge B$	AND
1	1	1	X	$F = \bar{A}$	Complement A

Let's See Another Example

- Derive the input equations (X_i , Y_i and Z_i) for the parallel adders to be used in the ALU which satisfies the following functional design specification.

s_2	s_1	c_{in}	Required Functions
0	0	0	$F = AB + C$
0	0	1	$F = AB + C + 1$
0	1	0	$F = AB$
0	1	1	$F = AB + 1$
1	0	x	$F = (AB)'$
1	1	x	$F = AB$



Solution

s₂	s₁	c_{in}	X_i	Y_i	Z_i	Required Functions
0	0	0	AB	C	0	F = AB + C
0	0	1	AB	C	1	F = AB + C + 1
0	1	0	AB	0	0	F = AB
0	1	1	AB	0	1	F = AB + 1
1	0	x	AB	1	x	F = (AB)'
1	1	x	AB	0	x	F = AB



Solution

- $X_i = AB$
- $Y_i = s_1' C$
- $Z_i = c_{i-1}$
- $c_0 = c_{in}$

- Then for logical operations,
 - $X_i = AB$
 - $Y_i = s_1' C + s_2 s_1'$
 - $Z_i = s_2' c_{i-1}$

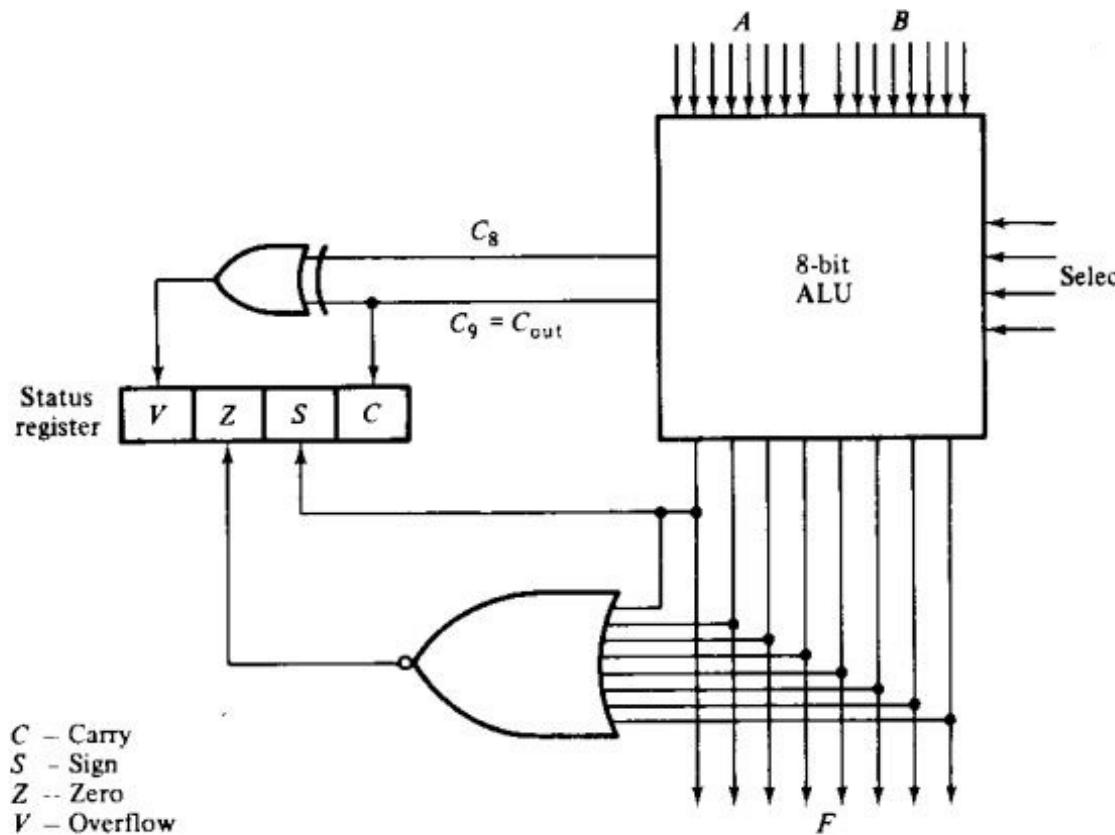


Status Register

- Four bits represents four status bits
 - C: Contains the output carry of the operation
 - S: Contains the sign of the result of the operation
 - Z: Indicates whether the n -bit of the result is 0 or not
 - V: Indicates any overflow has occurred due to the operation
- Status bits help to determine relationships among inputs
- Example:
 - Compare the value of A with the value of B
 - Determine the value of x^{th} bit of an input



Status Register



Comparing two unsigned numbers

- Compare the value of A with the value of B
- Check the status bits (mainly C and Z) after the performing the following operation

$$\begin{aligned} F &= A + B' + 1 = A - B \\ &= 2^n + (A - B) \end{aligned}$$

Relation	Condition of status bits	Boolean function



Comparing two signed numbers

- Compare the value of A with the value of B
- Check the status bits (mainly Z, V and S) after the performing the following operation
 $F = A - B$

Relation	Condition of status bits	Boolean function



Effect of Output Carry

Function select			Arithmetic function	$C_{out} = 1$ if	Comments
s_1	s_0	C_{in}			
0	0	0	$F = A$		C_{out} is always 0
0	0	1	$F = A + 1$	$A = 2^n - 1$	$C_{out} = 1$ and $F = 0$ if $A = 2^n - 1$
0	1	0	$F = A + B$	$(A + B) > 2^n$	Overflow occurs if $C_{out} = 1$
0	1	1	$F = A + B + 1$	$(A + B) > (2^n - 1)$	Overflow occurs if $C_{out} = 1$
1	0	0	$F = A - B - 1$	$A > B$	If $C_{out} = 0$, then $A < B$ and $F = 1$'s complement of $(B - A)$
1	0	1	$F = A - B$	$A > B$	If $C_{out} = 0$, then $A < B$ and $F = 2$'s complement of $(B - A)$
1	1	0	$F = A - 1$	$A \neq 0$	$C_{out} = 1$, except when $A = 0$
1	1	1	$F = A$		C_{out} is always 1



Overflow Flag

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0



References

- Digital Logic and Computer Design *by M. Morris Mano*
 - Chapter 9 (9.1-9.7)



Thank You ☺



CSE 305

Computer Architecture

Introduction

Prepared by
Madhusudan Basak
Assistant Professor
CSE, BUET

* Some modifications made by Saem Hasan



Why Computer Architecture?

- ❑ To apply the Architectural sense to a computer



- ❑ To apply computer for Architectural design



- ❑ To know about the basic Architecture of a computer



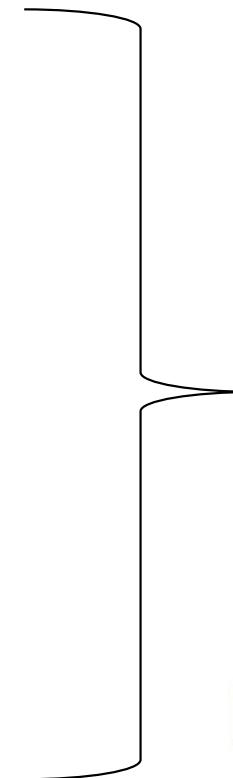
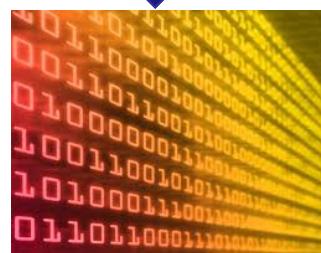
Why Computer Architecture?

❑ Purpose

- How **hardware** (processors, memories, disk drives, network infrastructure) plus **software** (operating systems, compilers, libraries, network protocols) **combine** to support the execution of application programs
- How you as a programmer can best use these resources



Why Computer Architecture?



What to know?



What the computer does

- Logical View
- Instruction Set Architecture (ISA)

How it does

- Physical View
- Computer Organization

Computer Architecture



Instruction Set Architecture



Computer Organization



Instruction Set Architecture

- ❑ Instruction set architecture is the **attributes of a computing system** as seen by the assembly language programmer or compiler.
 - Instruction Set (**what operations can be performed?**)
 - Instruction Format (**how are instructions specified?**)
 - Data storage (**where is data located?**)
 - Addressing Modes (**how is data accessed?**)
 - Exceptional Conditions (**what happens if something goes wrong?**)

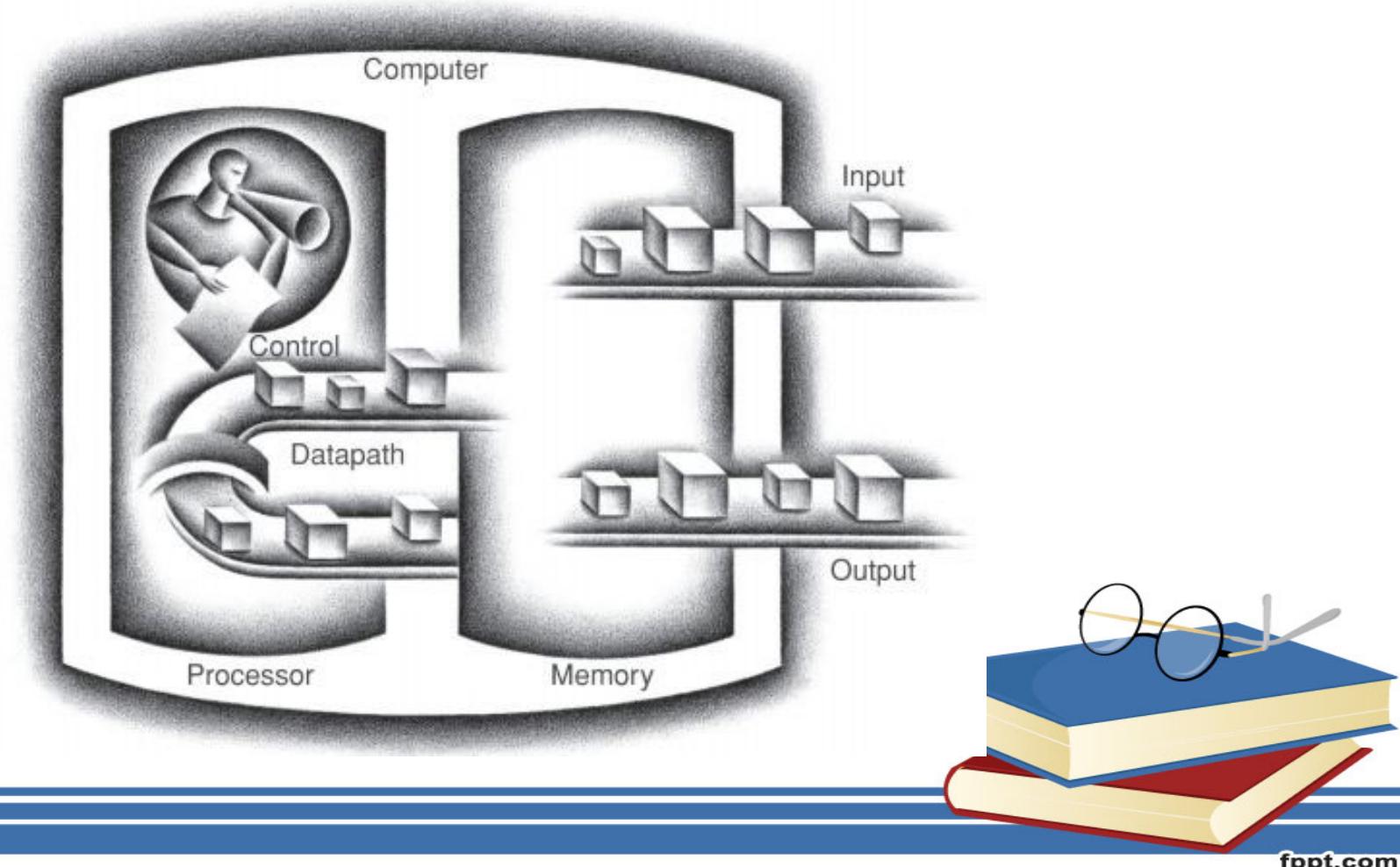


Machine Organization

- Machine organization is the view of the computer that is seen by the logic designer. This includes
 - Capabilities & performance characteristics of functional units (e.g., registers, ALU, shifters, etc.)
 - Ways in which these components are interconnected
 - How information flows between components
 - Logic and means by which such information flow is controlled
 - Coordination of functional units



Components of a Computer



Components of a Computer



- Gives directions to the other components
- e.g., bus controller, memory interface unit



- Performs arithmetic and logic operations
- e.g., adders, multipliers, shifters



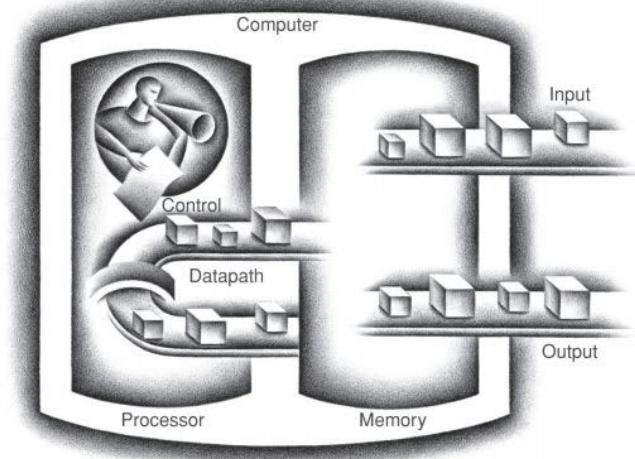
- Holds data and instructions
- e.g., cache, main memory, disk



- Sends data to the computer
- e.g., keyboard, mouse



- Gets data from the computer
- e.g., screen, sound card



Information in a computer -- Instructions

- Instructions specify commands to

- Transfer information **within a computer**
 - e.g., from memory to ALU
 - Transfer of information **between the computer and I/O devices**
 - e.g., from keyboard to computer, or computer to printer
 - Perform **arithmetic and logical operations**
 - e. g., add two numbers, perform a logical AND



Information in a computer -- Instructions

- ❑ A sequence of instructions to perform a task is called a **program**, which is stored in the memory.
- ❑ Processor fetches instructions from the memory and performs the operations stated in those instructions.
- ❑ What do the instructions operate upon?



Information in a computer -- Data

- Data are the “operands” upon which instructions operate.
- Data could be:
 - Numbers,
 - Encoded characters.
- Data, in a broad sense means any digital information.
- Computers use data that is encoded as a string of binary digits called bits.



Classes of Computers

□ Desktop / Notebook Computers

- Low-end systems, high performance workstations.
- Subject to cost/performance tradeoff

□ Server Computers

- Network based
- High capacity, performance, reliability
- Range from small servers to building sized

□ Embedded Computers

- Hidden as components of systems
- Minimize memory and power. Often not programmable



Eight Great Ideas in Computer Architecture

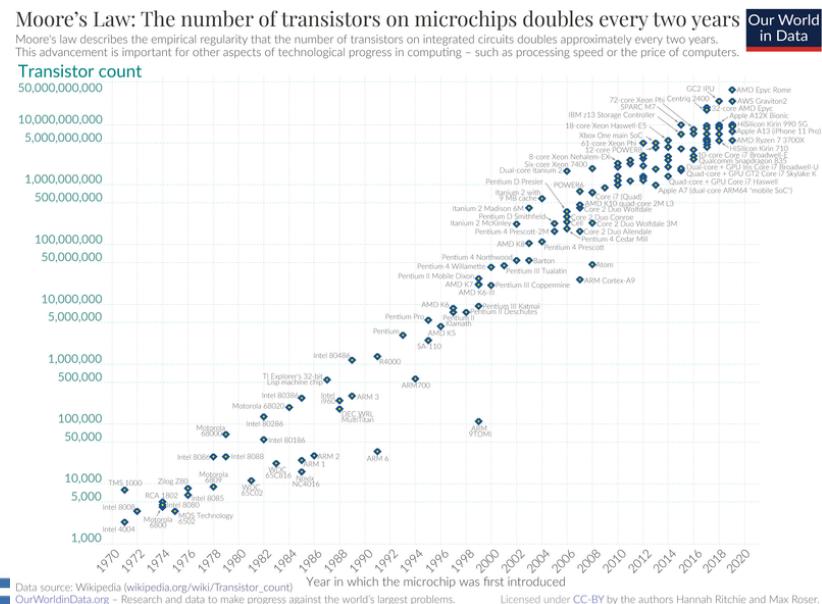
- Design for Moore's Law
- Use Abstraction to Simplify Design
- Make the Common Case Fast
- Performance via Parallelism
- Performance via Pipelining
- Performance via Prediction
- Hierarchy of Memories
- Dependability via Redundancy



Design for Moore's Law



- Provided by Gordon Moore (co-founder of Intel) in 1965
- Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years.



Use Abstraction to Simplify Design



- Computing system maintains a hierarchical structure
- Lower-level **details** are hidden to the higher levels
- Higher level only gets the abstract view
- Both Hardware and Software consist of hierarchical layers using abstraction



Make the Common Case Fast



- ❑ More efficiency in common case, more impact in overall design.



Performance via Parallelism



- ❑ Current multi processor system exploits parallelism.
- ❑ Often needs special care for coordination.

- ❑ $x = a + b$
- ❑ $y = c^*d$

- ❑ $x = a + b$
- ❑ $y = x^*d$

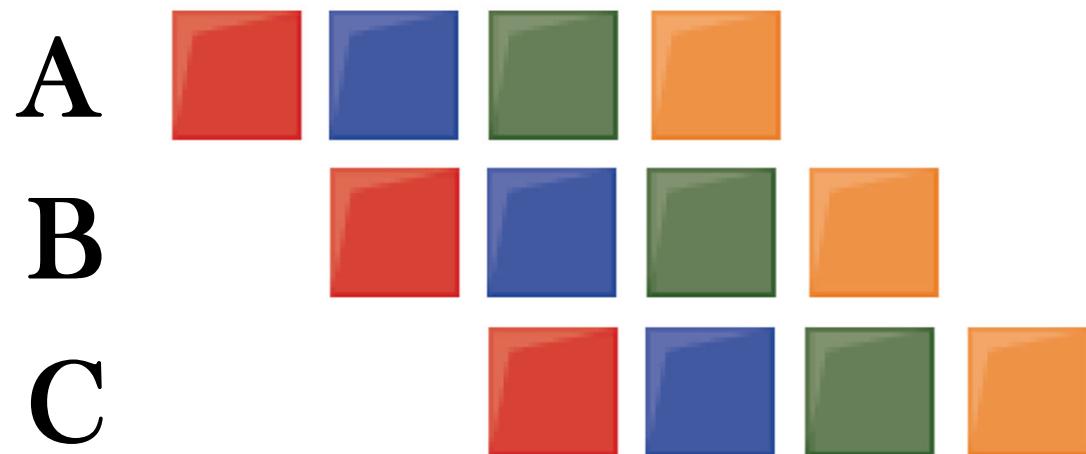


Performance via Pipelining



❑ Pipelining

- A special case of parallelism
- Performing multiple non-dependent operations at the same time



Performance via Prediction



- Perform operation just based on prediction/assumption
- Applicable when the impact is **not costly**



Hierarchy of Memories



- We want faster and cheaper memory
- Faster memory is costlier
- Cheaper memory is slower
- Trade-off is memory hierarchy



Dependability via Redundancy

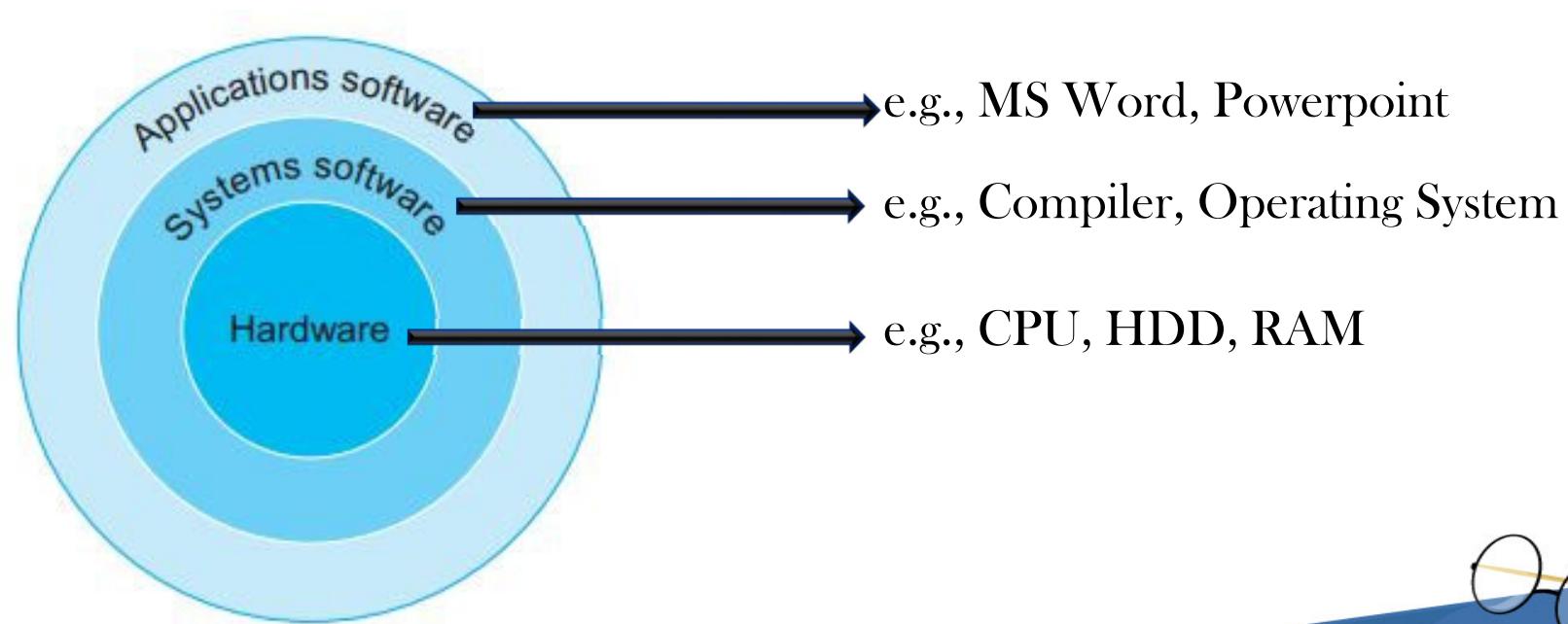


- Redundancy means keeping multiple copies
- One fails, another exists => Dependable



Hierarchical Structure of Program Execution

- ❑ Simplified view including Hardware



Software Abstraction

- Hierarchical structure for a program execution

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

Assembly language program (for MIPS)

```
swap:
    multi $2, $5,4
    add $2, $4,$2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

Binary machine language program (for MIPS)

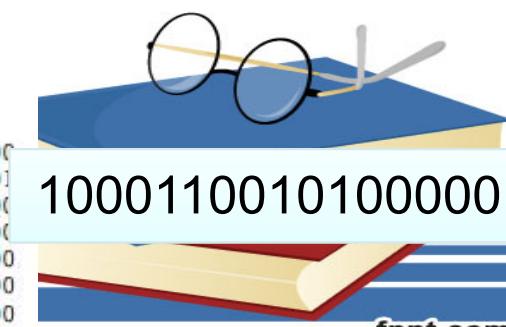
```
0000000001010001000000000100011000
00000000010000010000100000010000
1000110111100010000000000000000000
10001110000100100000000000000000000
10101110000100100000000000000000000
10101101111000100000000000000000000
000000111110000000000000000000000000
```

Compiler

Assembler

A + B

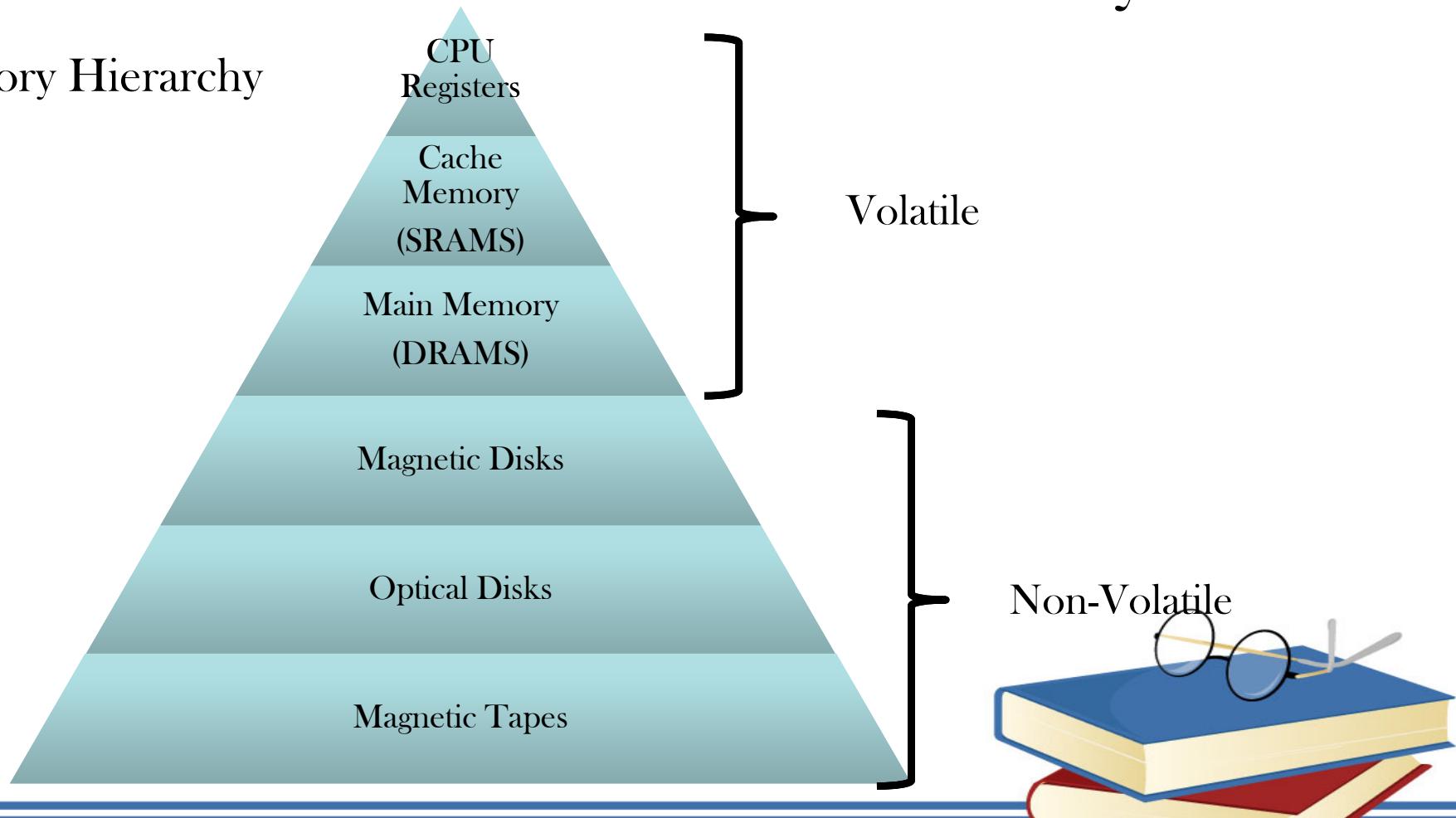
Add A, B



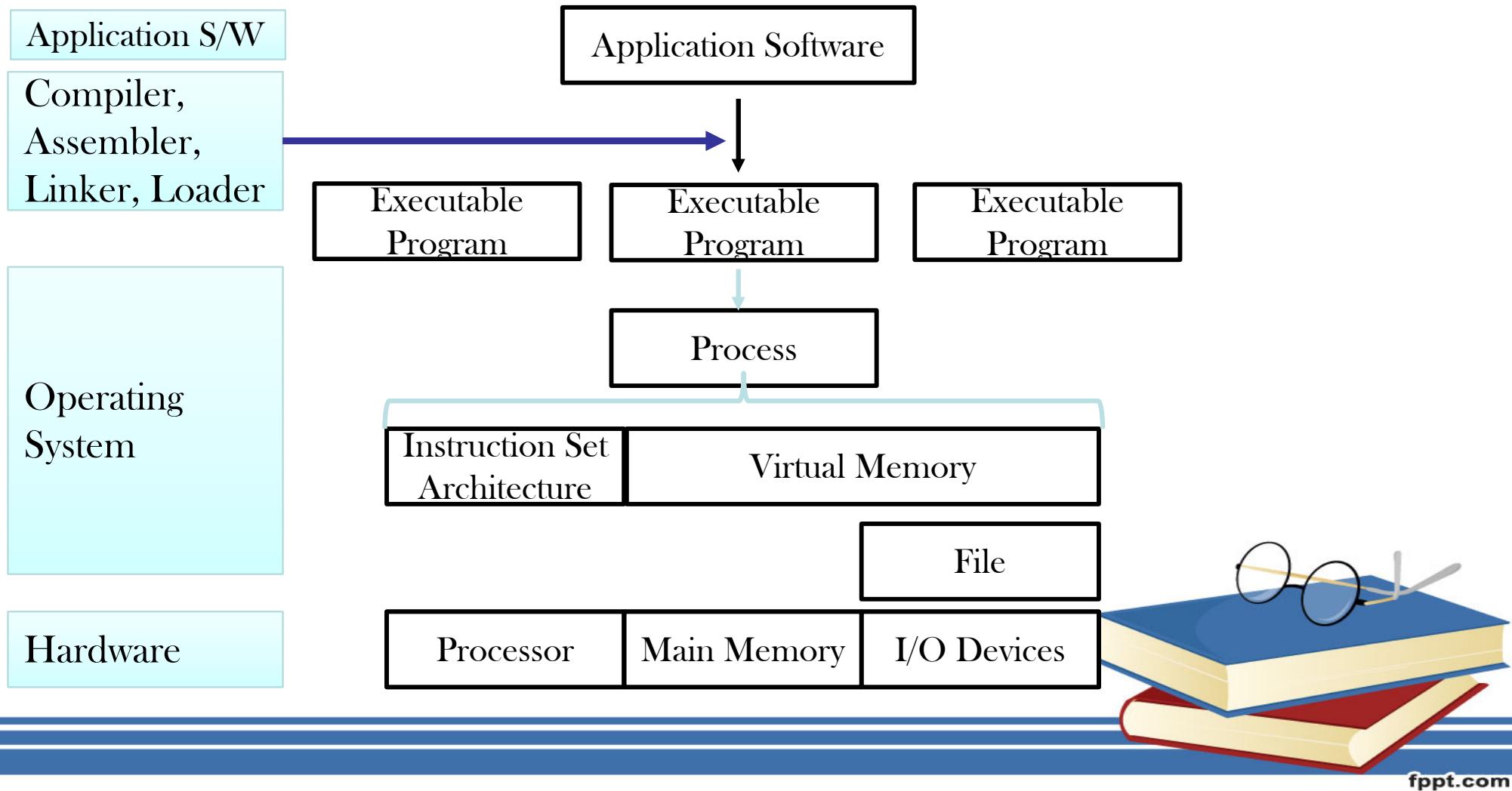
Hardware Abstraction: Memory

□ Memory Hierarchy

Increased speed and cost



Simplified overall abstraction



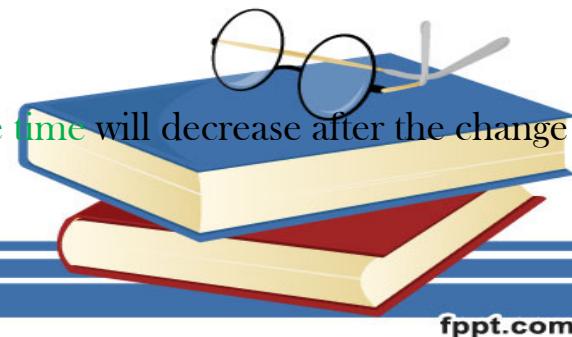
Performance

- ❑ What is the metric of the performance of a computing system?
- ❑ Depends on the purpose
- ❑ Two commonly used metrics are:
 - Execution or Response Time
 - How long it takes to do a task
 - Throughput
 - Total work done per unit time
 - e.g., tasks/transactions/... per hour



Example

- Do the following changes to a computer system increase **throughput**, decrease **response time**, or both?
 - Replacing the processor in a computer with a faster version
 - **Response time** decreases or improves
 - Decreasing **response time** generally increases **throughput**
 - Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the web
 - **Throughput** increases
 - **Response time** depends on scenario
 - Generally no impact on **response time**
 - But in case tasks were waiting in the queue previously, **response time** will decrease after the change



Relative Performance

- We shall focus on Response or Execution time

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

- “X is n times faster than Y” means

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = n$$

- Example: Time taken to run a program

- A: 10s, B=15s
- $\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A} = \frac{15s}{10s} = 1.5$
- So, A is 1.5 times faster than B



Measuring Execution Time

❑ Elapsed time

- Counts everything (disk and memory accesses, I/O, operating system overhead etc.)
- A useful number, but often not good for comparison purposes
 - Time sharing among multiple programs

❑ CPU time

- Doesn't count I/O or time spent in running other programs
- Can be broken into *system CPU time* and *user CPU time*

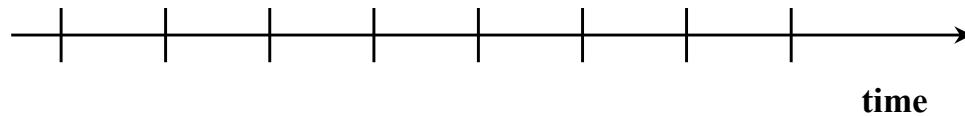
❑ Our focus: user CPU time

- CPU time spent in executing the lines of code that are “in” our program



Clock Cycles

- Time is not continuous, rather discrete to a Computer's perspective
- Activities are performed during the discrete clock ticks



- cycle time = time between ticks = seconds per cycle
- clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 2 Ghz. clock has a $1 / 2 \times 10^9 = 0.5$ nano-second (ns) cycle time

- So, for a program

$$CPU\ Time = CPU\ clock\ cycles * Clock\ cycle\ time$$

$$CPU\ Time = \frac{CPU\ clock\ cycles}{Clock\ rate}$$



Clock Cycles

- ❑ For a program

$$CPU\ Time = CPU\ clock\ cycles * Clock\ cycle\ time$$

$$CPU\ Time = \frac{CPU\ clock\ cycles}{Clock\ rate}$$

- ❑ Performance improvement means

- Decreasing number of clock cycles
- Increasing clock rate
- Hardware designer often trade off clock rate against cycle count



CPU Time Example

❑ Computer A: 2GHz clock, 10s CPU time

❑ Designing Computer B

➤ Aim for 6s CPU time

➤ Can do faster clock, but causes $1.2 \times$ clock cycles

❑ How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$



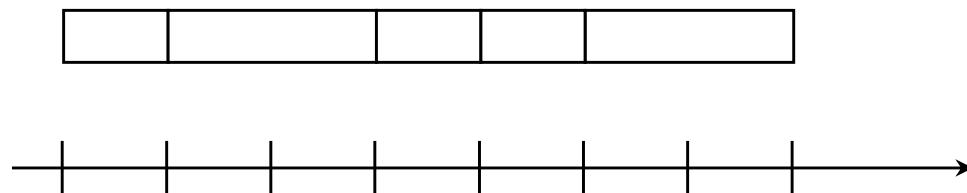
Instructions vs Cycles

- ❑ Is the number of cycles identical with the number of instructions?
- ❑ No!



Why?

- Operations take different time
 - Multiplication takes longer than addition
 - Floating point operations take longer than integer operations
 - The access time to a register is much shorter than to memory location



Instruction Count and CPI

$CPU\ Time = CPU\ clock\ cycles * Clock\ cycle\ time$

$CPU\ clock\ cycles = Instruction\ Count * CPI$

$CPU\ Time = Instruction\ Count * CPI * Clock\ cycle\ time$

$$CPU\ Time = \frac{Instruction\ Count * CPI}{Clock\ Rate}$$



Instruction Count and CPI

- Instruction Count for a program
 - Determined by program, ISA and compiler
- CPI is an average since the number of cycles per instruction varies from instruction to instruction
- CPI varies by application, as well as among implementation with the same instruction set
 - Number of cycles for each instruction
 - Frequency of instructions (instruction mix)
 - Memory access time



CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}\end{aligned}$$

A is faster...

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}\end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

...by this much



Program Execution time

*CPU clock cycles = Instruction Count * CPI*

$$CPU \text{ clock cycles} = \sum_{i=1}^n (CPI_i * C_i)$$

$$CPI_{avg} = \frac{Clock \text{ Cycles}}{instruction \text{ count}} = \frac{\sum_{i=1}^n (CPI_i * C_i)}{\sum_{i=1}^n C_i}$$



CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
 - Clock Cycles
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$
 $= 10$
 - Avg. CPI = $10/5 = 2.0$
- Sequence 2: IC = 6
 - Clock Cycles
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$
 $= 9$
 - Avg. CPI = $9/6 = 1.5$



Performance Summary

❑ The BIG Picture

$$CPU\ Time = Instruction\ Count * CPI * Clock\ cycle\ time$$

$$CPU\ Time = \frac{Instructions}{Program} * \frac{Clock\ cycles}{Instruction} * \frac{Seconds}{Clock\ cycle}$$

❑ Performance depends on

- Algorithm
- Programming language
- Compiler
- Instruction set architecture



Tradeoffs

□ Instruction count, CPI, and clock cycle present tradeoffs

➤ RISC - reduced instruction set computer (MIPS)

- Simple instructions
- Higher instruction counts for an application
- Lower CPI

➤ CISC - complex instruction set computer (IA-32)

- More complex instructions
- Lower instruction counts for an application
- Higher CPI



Comparing Computing Systems

- Comparing systems => comparing execution time of the **workload** is required
- **Benchmarks** can also help to evaluate measure the performance
- 12 benchmarks of SPECINTC2006 are given in the next slide
- **SPECratio** can be used to measure the performance

$$SPEC_{ratio} = \frac{\text{Execution time of a reference computing system}}{\text{Execution time of the measured computing system}}$$

- The geometric mean of the **SPECratios** (of the Benchmarks) can be calculated

$$\text{GeometricMean} = \sqrt[n]{\prod_{i=1}^n \text{Execution time ratio}_i}$$



Benchmarks

Description	Name	Instruction Count x 10 ⁹	CPI	Clock cycle time (seconds x 10 ⁻⁹)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	-	-	-	-	-	-	25.7

FIGURE 1.18 SPECINTC2006 benchmarks running on a 2.66 GHz Intel Core i7 920. As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios.



Fallacies and Pitfalls

- Fallacy

- Commonly held misconceptions

- Pitfall

- a hidden or unsuspected danger or difficulty
 - Easily made mistakes



Fallacies

❑ Fallacy 1

Computers at low utilization use little power

❑ Fallacy 2

Designing for performance and designing for energy efficiency are unrelated goals



Pitfalls

❑ Pitfall 1

Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of the improvement.

❑ Pitfall 2

Using a subset of the performance equation as a performance metric.



Amdahl's Law

*Execution time
after improvement*



*Execution time
affected by improvement*
 $\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}}$



*Execution time
unaffected*



Amdahl's Law

$$\begin{aligned}\text{Overall Speedup} &= \frac{\text{Old execution time}}{\text{New execution time}} \\ &= \frac{1}{\left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)}\end{aligned}$$



Example

- Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the speed of multiplication if I want my program to run five times faster?

$$20 = \frac{80}{n} + 20$$

$$n = \frac{80}{0} = \infty$$



Acknowledgements

- These slides contain material developed and copyright by:
 - Krste Asanovic (UCB), James Hoe (CMU), Li-Shiuan Peh (MIT), Sudhakar Yalamanchili (GATECH), and Amirali Baniasadi (UVIC) in part of their respective courses
 - Lecture slides by Dr. Tanzima Hashem, Professor, CSE, BUET
 - Lecture slides by Ms. Mehnaz Tabassum Mahin, Assistant Professor, CSE, BUET



Thank You ☺



CSE 305

Computer Architecture

Instructions

Prepared by
Madhusudan Basak
Assistant Professor
CSE, BUET

*Some modifications done by Saem Hasan



Instructions

- ❑ A computer is run by instructions
- ❑ Instruction Set
 - All possible instructions for a CPU



Instruction Set Architecture

- ❑ Instruction set architecture is the **attributes of a computing system** as seen by the assembly language programmer or compiler.
 - Instruction Set (**what operations can be performed?**)
 - Instruction Format (**how are instructions specified?**)
 - Data storage (**where is data located?**)
 - Addressing Modes (**how is data accessed?**)
 - Exceptional Conditions (**what happens if something goes wrong?**)



Design Philosophy

It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations. ... The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

Burks, Goldstine, and von Neumann, 1946 ▶



MIPS Architecture

- Acronym of “Microprocessor without Interlocked Pipelined Stages”
 - Is a **RISC** (Reduced Instruction Set Computer) **ISA** (Instruction Set Architecture)
 - Developed by MIPS Technologies (then MIPS Computer Systems) in 1985



Design Principles

- *Design Principle 1:* Simplicity favors regularity.
- *Design Principle 2:* Smaller is faster.
- *Design Principle 3:* Good design demands good compromises.



Simplicity favors regularity

- ❑ The instructions of MIPS are fixed and rigid
- ❑ Rigidity ensures Regularity
- ❑ Simplicity favors regularity



Example Instruction

- *Add* and *subtract* instructions always follow the following fixed format

add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands

$$a = b + c + d + e$$



```
add a, b, c      # The sum of b and c is placed in a  
add a, a, d      # The sum of b, c, and d is now in a  
add a, a, e      # The sum of b, c, d, and e is now in a
```



Example Instruction

- *Add* and *subtract* instructions always follow the following fixed format

add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands

a = b + c;
d = a - e;

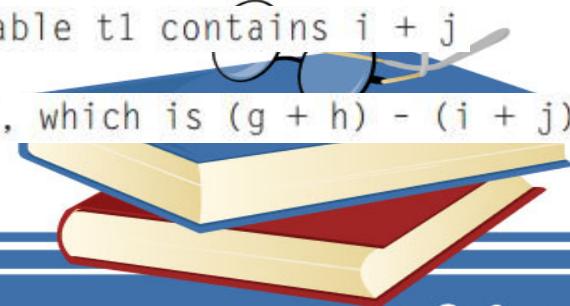


add a, b, c
sub d, a, e

f = (g + h) - (i + j);



add t0,g,h # temporary variable t0 contains g + h
add t1,i,j # temporary variable t1 contains i + j
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)



MIPS Instruction Properties

- ❑ Arithmetic operations only takes register values as operands
- ❑ Size of a register : 32 bit
- ❑ Number of registers: 32
 - Satisfies *Design Principle 2*: Smaller is faster.
 - Large number of registers -> longer time for electronic signal to travel -> Increased clock cycle
 - Large number of registers -> Increased number of control bits
- ❑ A number is dedicated to a particular register.
 - 5 bit are reserved to indicate each register when the count is 32
 - Compiler maps a program variable to a register
 - There is a number assigned against each register



MIPS Instruction Properties

□ Example

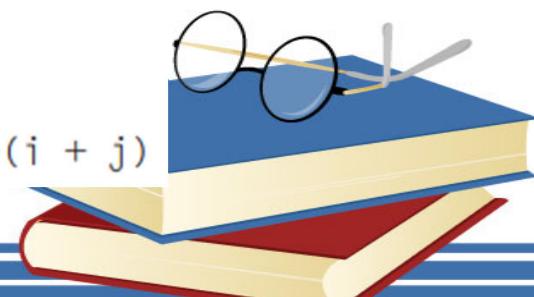
```
f = (g + h) - (i + j);
```



```
add t0,g,h # temporary variable t0 contains g + h  
add t1,i,j # temporary variable t1 contains i + j  
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```



```
add $t0,$s1,$s2 # register $t0 contains g + h  
add $t1,$s3,$s4 # register $t1 contains i + j  
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```



MIPS Instruction Properties

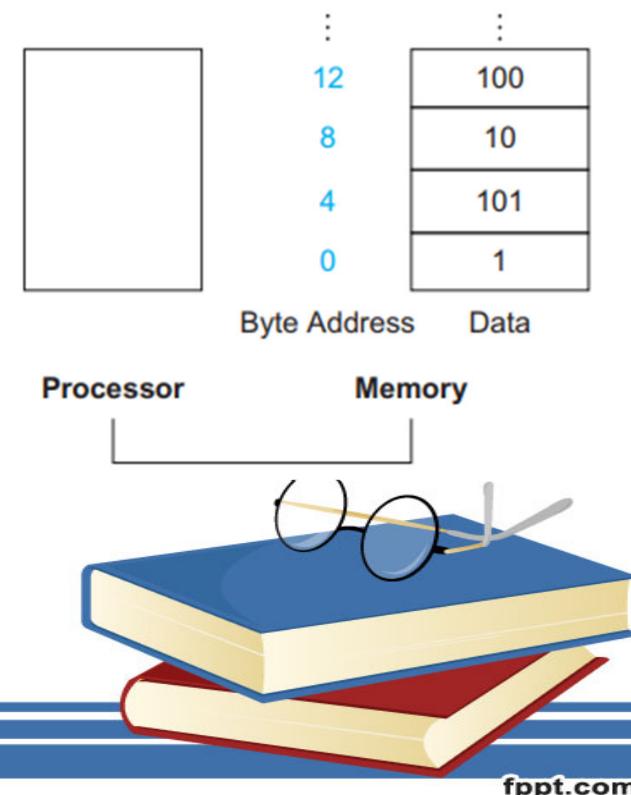
- What about data structures (Arrays and Structures)?
 - These are kept in memory and a particular element is transferred to registers whenever required

- Compiler

- places an array/structure into memory
- keeps the starting address of the memory in a register
- tracks which register is used for which array/structure

- Data Transfer Instructions

- load word: *lw*
- store word: *sw*



Data Transfer Instructions

- Example
- $A[12] = h + A[8]$

```
lw    $t0,32($s3)  # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
sw    $t0,48($s3)  # Stores h + A[8] back into A[12]
```



Spilling Registers

- ❑ What if the number of variables and data structures is more than 32?
 - The process of putting less commonly used variables into memory is called Spilling Registers.
 - Keeps the most frequently used variables in registers and less frequent ones in memory
 - Moves variables around registers and memory



Operation with Constants

- Add *immediate* or *addi* instruction
 - Takes one register and one constant as input

addi \$s3,\$s3,4 # \$s3 = \$s3 + 4

- Constant 0 is used very often (e.g., for transfer operations)

- A register *\$zero* is dedicated for this purpose

add \$s2,\$s3,\$zero #\$s2=\$s3

- Is an instruction represented using register names?



Representing Instructions in the Computer

- Key principles of Today's computer instructions
 - Instructions are represented as numbers
 - Programs are stored in memory to be read or written, just like data



Representing Instructions in the Computer

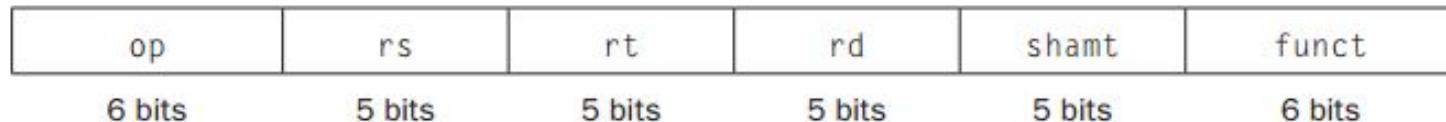
- An instruction is a sequence of bits
- Each register is mapped to a number
 - There is a convention to map register names into numbers (e.g., registers \$s0 to \$s7 map onto registers 16 to 23)
- Each instruction is 32 bit long (size of a word)
 - Satisfies *Design Principle 1*: Simplicity favors regularity.
- Instruction Format: format used by an instruction



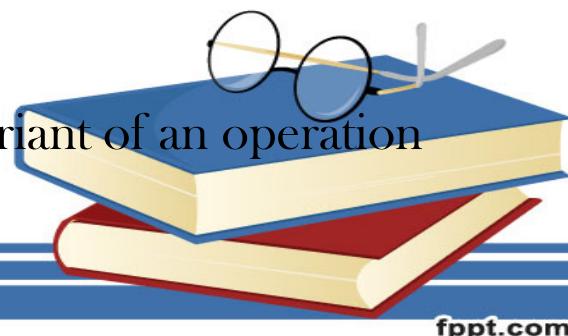
An MIPS Instruction Format

□ R-Format

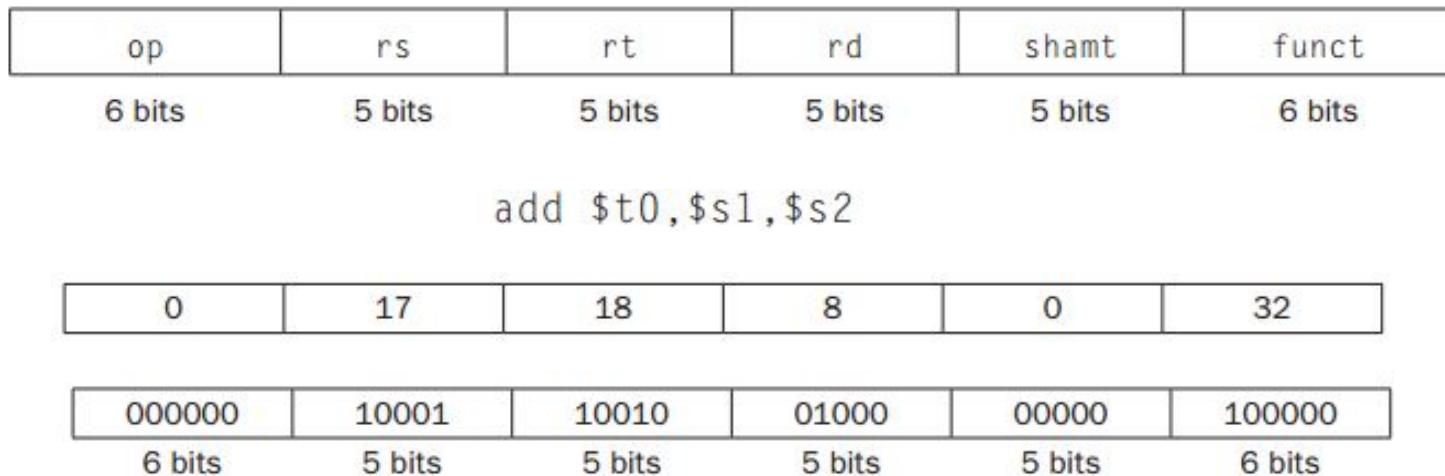
- Deals with arithmetic operations with registers



- *op*: shorthand of opcode, denotes operation type and format type
- *rs*: Source Register1
- *rt*: Source Register 2
- *rd*: Destination Register
- *shamt*: Shorthand of shift amount
- *funct*: shorthand of function code, denotes the specific variant of an operation



An MIPS Instruction Format



Would it be good to use just one format for all kind of operations?

No!

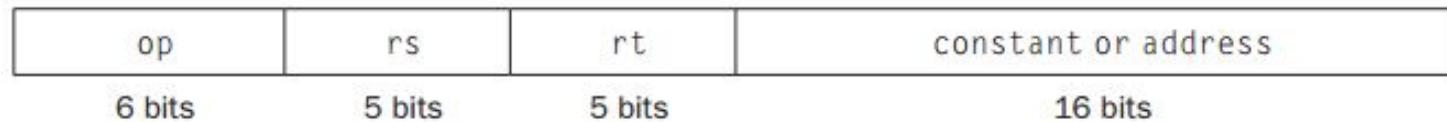
Design Principle 3: Good design demands good compromises.



Another MIPS Instruction Format

□ I-Format

- Deals with immediate and data transfer operations



- *rt*: Here *rt* represents the destination register



MIPS Instruction Encoding (Simplified)

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

$$A[300] = h + A[300];$$

```
lw $t0,1200($t1) # Temporary reg $t0 gets A[300]
add $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw $t0,1200($t1) # Stores h + A[300] back into A[300]
```

Op	rs	rt	rd	address/ shamt	funct	
35	9	8	1200			
0	18	8	8	0	32	
43	9	8	1200			
100011	01001	01000	0000 0100 1011 0000			
000000	10010	01000	01000	00000	100000	
101011	01001	01000	0000 0100 1011 0000			



Summary

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format



Logical Operations

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Shift right by constant

No NOT Operation?



Shift Left

- ❑ sll \$t2,\$s0,4 # reg \$t2 = reg \$s0 << 4 bits

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0



Branching Operations

beq register1, register2, L1

bne register1, register2, L1

if (i==j)
 f = g + h;
else
 f = g - h;

bne \$s3,\$s4,Else *#go to Else if i ≠ j*
add \$s0,\$s1,\$s2 *#f = g + h*
j Exit
Else: *sub \$s0,\$s1,\$s2* *#f = g - h*
Exit:

slt \$t0, \$s3, \$s4 # \$t0 = 1 if \$s3 < \$s4

slti \$t0,\$s2,10 # \$t0 = 1 if \$s2 < 10

sltu \$t1, \$s0, \$s1 # unsigned comparison

Why not *blt*?



Branching Operations

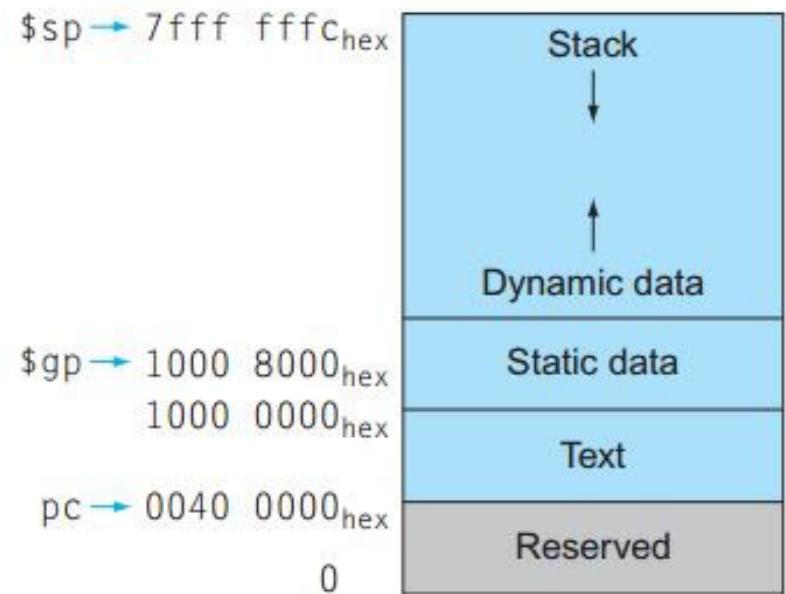
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call



Memory Allocation for Program and Data

- ❑ Reserved Memory
- ❑ Text Segment: Program Code
- ❑ Static Data Segment:
 - Contains static variables, constants, arrays
- ❑ Dynamic Data Segment (Heap):
 - Contains linked lists, variable length data
- ❑ Stack
 - Contains function local variables, parameters

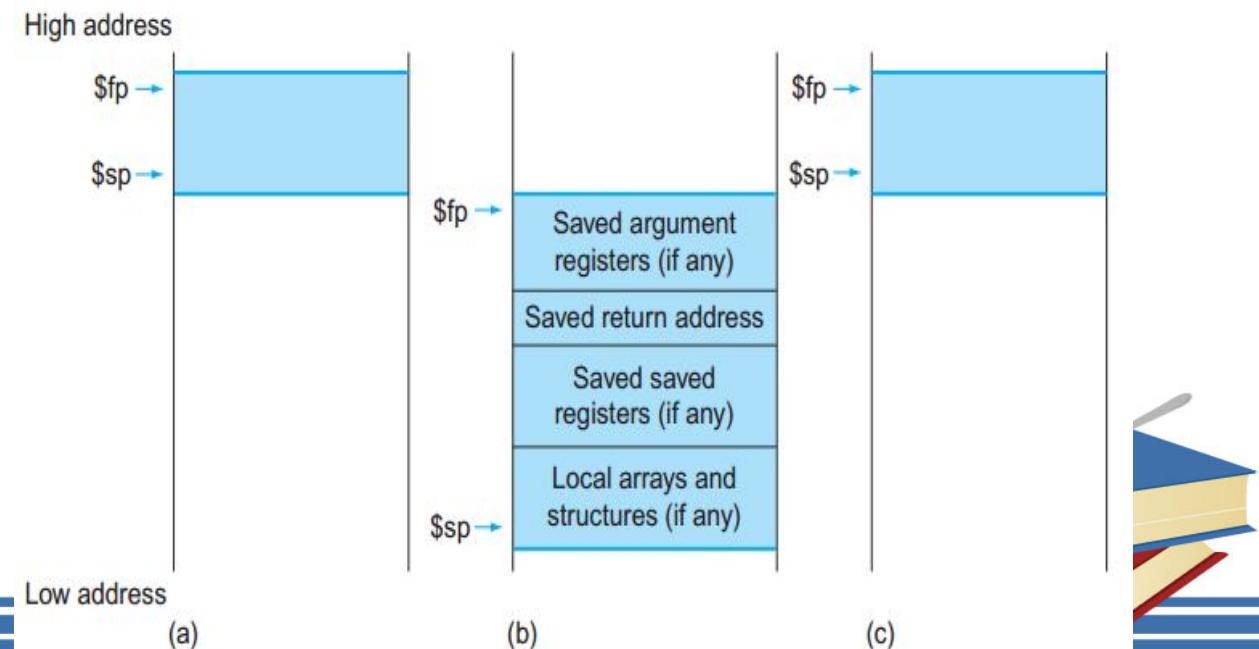


Stack During Procedure Call

- Stores/saves the values of registers and Restores those later
- Either \$sp and \$fp combination, or only \$sp is used
- The allocated stack area by a procedure is known as activation record or procedure frame

```
int non_leaf()
{
    int a=10, b=5, c=2, d=7, x;
    x= leaf(a,b,c,d);
    return x;
}

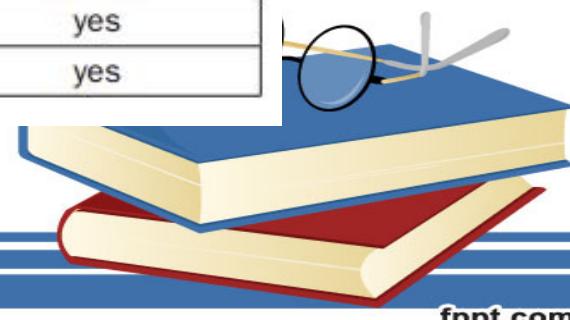
int leaf(int g, int h, int i, int j)
{
    int f;
    f=(g+h)-(i+j);
    return f;
}
```



Stack During Procedure Call

- Stores the values of registers

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

❑ Place for parameters:

- \$a0-\$a3: four argument registers in which to pass parameters
- Memory (e.g., stack) can be used if more values to be passed



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

❑ Transfer Control:

➤ Jump-and-link instruction (jal)

jal ProcedureAddress

- Keeps the return address for the procedure in \$ra and jumps to the *ProcedureAddress*



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

❑ Acquiring the storage resources

- Different register values (manipulated by caller) are stored in the stack
- Prepares the registers for operations
- Can use the memory for data storage



Procedures in MIPS

□ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
 2. Transfer control to the procedure.
- 3. Acquire the storage resources needed for the procedure.**
4. Perform the desired task.
 5. Put the result value in a place where the calling program can access it.
 6. Return control to the point of origin, since a procedure can be called from several points in a program.

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
 2. Transfer control to the procedure.
 3. Acquire the storage resources needed for the procedure.
- 4. Perform the desired task.**
5. Put the result value in a place where the calling program can access it.
 6. Return control to the point of origin, since a procedure can be called from several points in a program.

❑ Performing Tasks

- Can perform all the operations allowed by the MIPS instruction set



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

❑ Result storing by the callee

- \$v0-\$v1: two value registers in which to return values
- Memory (e.g., stack) can be used if more values to be returned



Procedures in MIPS

❑ Six steps during a procedure call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

❑ Procedure returned

- Stack is adjusted using \$sp and \$fp pointers
- An unconditional jump to the address from where the caller will resume execution

jr \$ra



An Example

non_leaf:

```
int non_leaf()
{
    int a, b, c, d, x;
    ...
    x=leaf(a,b,c,d);
    ...
}

int leaf(int g, int h, int i, int j)
{
    int f;
    f=(g+h)-(i+j);
    return f;
}
```

...
add \$a0, \$s1, \$zero
add \$a1, \$s2, \$zero
add \$a2, \$s3, \$zero
add \$a3, \$s4, \$zero
jal leaf
add \$s1, \$zero, \$v0
...

leaf:
addi \$sp, \$sp, -12
sw \$t1, 8(\$sp)
sw \$t0, 4(\$sp)
sw \$s0, 0(\$sp)

add \$t0,\$a0,\$a1
add \$t1,\$a2,\$a3
sub \$s0,\$t0,\$t1

add \$v0,\$s0,\$zero

lw \$s0, 0(\$sp)
lw \$t0, 4(\$sp)
lw \$t1, 8(\$sp)
addi \$sp,\$sp,12

jr \$ra

Put parameters in a place where the procedure can access them.

Transfer control to the procedure.

Acquire the storage resources needed for the procedure

Perform the desired task.

Put the result value in a place where the calling program can access it.

Return control to the point of origin, since a procedure can be called from several points in a program.

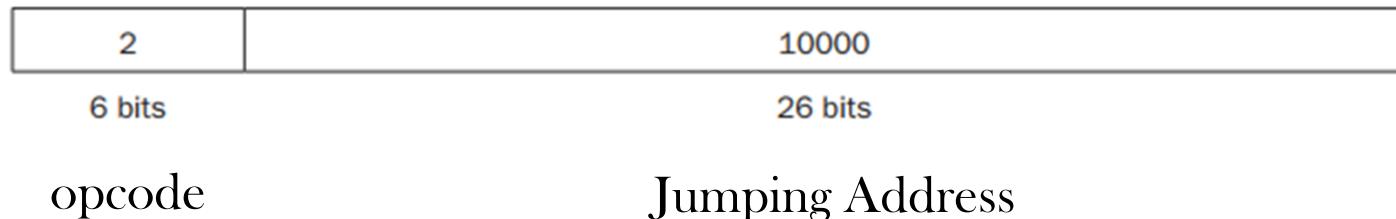
Nested Call

```
int fact(int n)
{
    if(n<1)
        return 1;
    else
        return n*fact(n-1);
}
```

fact:	addi \$sp, \$sp, -8 sw \$ra, 4(\$sp) sw \$a0, 0(\$sp) slti \$t0,\$a0,1 beq \$t0,\$zero,L1 addi \$v0,\$zero,1 addi \$sp,\$sp,8 jr \$ra	# adjust stack for 2 items # save the return address # save the argument n # test for n < 1 # if n >= 1, go to L1 # Set the return value # pop 2 items off stack # return to caller
L1:	addi \$a0,\$a0,-1 jal fact lw \$a0, 0(\$sp) lw \$ra, 4(\$sp) addi \$sp, \$sp, 8 mul \$v0,\$a0,\$v0 jr \$ra	# n >= 1: argument gets (n - 1) # call fact with (n - 1) # return from jal: restore argument n # restore the return address # adjust stack pointer to pop 2 items # return n * fact (n - 1) # return to the caller

J-Format

- ❑ To support long jump to a remote procedure address



ASCII representation of characters

- ASCII stands for *American Standard Code for Information Interchange*
- Uses 1 byte to represent a character

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	~	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	*	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	-	111	o	127	DEL



1 Byte Memory Operation

- ASCII demands 1-byte memory operation
- MIPS supports

lb \$t0,0(\$sp) #Reads 1 byte from memory and stores in the lowest (rightmost) byte of \$t0
sb \$t0,0(\$gp) #Reads 1 byte from the lowest (rightmost) byte of \$t0 and stores in the memory

Unsigned Version:

Load Byte: lbu

Store Byte: Not available in MIPS



Dealing with Strings

❑ Three commonly available strategies

- the first position of the string is reserved to give the length of a string
- an accompanying variable has the length of the string
- the last position of a string is indicated by a character used to mark the end of a string



Dealing with String: Example

- ASCII demands 1-byte memory operation
- MIPS supports

lb \$t0,0(\$sp) #Reads 1 byte from memory and stores in the lowest (rightmost) byte of \$t0
sb \$t0,0(\$gp) #Reads 1 byte from the lowest (rightmost) byte of \$t0 and stores in the memory

Unsigned Version:

Load Byte: lbu

Store Byte: Not available in MIPS



Dealing with String: An Example

```
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
}
```

strcpy:	addi \$sp,\$sp,-4 sw \$s0, 0(\$sp) add \$s0,\$zero,\$zero	# adjust stack for 1 more item # save \$s0 # $i = 0 + 0$
L1:	add \$t1,\$s0,\$a1 lbu \$t2, 0(\$t1) add \$t3,\$s0,\$a0 sb \$t2, 0(\$t3) beq \$t2,\$zero,L2 addi \$s0, \$s0,1 j L1	# address of $y[i]$ in \$t1 # $t2 = y[i]$ # address of $x[i]$ in \$t3 # $x[i] = y[i]$ # if $y[i] == 0$, go to L2 # $i = i + 1$ # go to L1. While loop continues
L2:	lw \$s0, 0(\$sp) addi \$sp,\$sp,4 jr \$ra	# End of string. Restore old \$s0 # pop 1 word off stack # return

String Copy Example

- MIPS code:

```
strncpy:  
    addi $sp, $sp, -4      # adjust stack for 1 item  
    sw   $s0, 0($sp)       # save $s0  
    add  $s0, $zero, $zero # i = 0  
L1:  add  $t1, $s0, $a1      # addr of y[i] in $t1  
    lbu $t2, 0($t1)        # $t2 = y[i]  
    add  $t3, $s0, $a0      # addr of x[i] in $t3  
    sb   $t2, 0($t3)        # x[i] = y[i]  
    beq $t2, $zero, L2     # exit loop if y[i] == 0  
    addi $s0, $s0, 1        # i = i + 1  
    j    L1                 # next iteration of loop  
L2:  lw   $s0, 0($sp)       # restore saved $s0  
    addi $sp, $sp, 4        # pop 1 item from stack  
    jr  $ra                 # and return
```

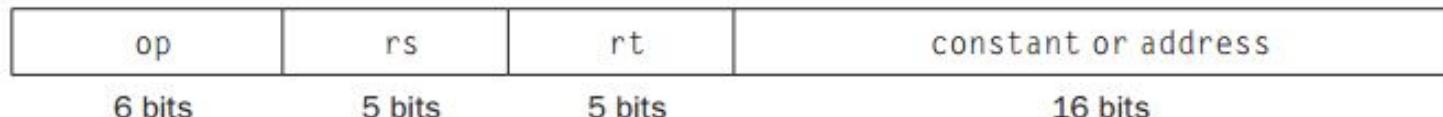
Dealing with Multiple Languages

- ❑ Unicode, a universal encoding, supports alphabets of most human languages
- ❑ Java uses Unicode
- ❑ Requires 16 bits to represent a character
- ❑ Operations required to load and store 16bits (halfwords)
 - Load halfword: lh
 - Load halfword unsigned: lhu
 - Store halfword: sh



Constant Size: Is larger feasible?

- ❑ The I-format is used by both immediate and memory data transfer operations



- ❑ More than 16 bit long constant?

- We have two options in hand
 - Supporting short constant in 1 instruction and deal long constant with additional instructions
 - Supporting long constant in 2 instructions
- We opted for the first option (to exploit the benefit of common case fast)

- ❑ Are we limited to use 16 bit constants (-32,768 to 32,767)?

- ❑ No ☺ Use two instructions to populate a register with 32 bit constant



Manipulating Larger Constant

□ Populating a register with 32 bit constant

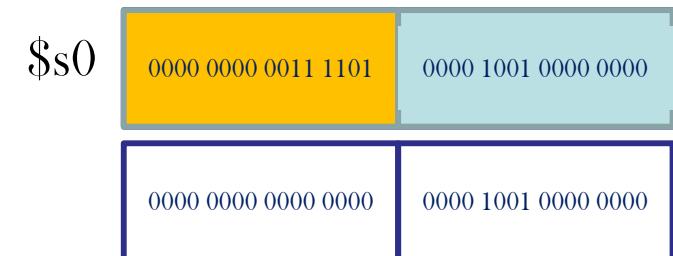
- Load Upper Immediate (lui) : Loads upper 16 bits
- Or Immediate (ori): : Loads lower 16 bits

□ Example: $x=y+4000000$

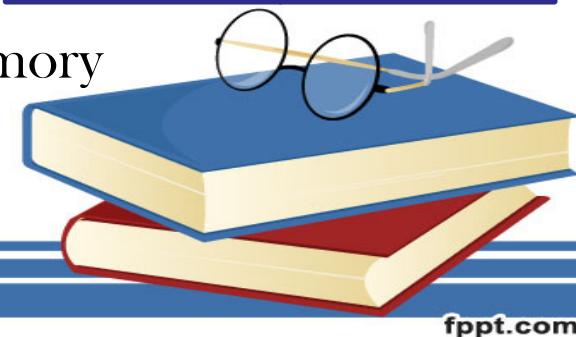
□ $4000000 = 0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000$

```
lui $s0, 61           // 61 = 0000 0000 0011 1101  
ori $s0, 2304         // 2304= 0000 1001 0000 0000  
add $s1, $t0, $s0
```

No sign extension for logical operations but it happens for arithmetic operations



□ Think how to use 32 bit address to access data from memory



Addressing in Branches

- PC is 32 bit but the address in conditional branching (if-else, loop etc.) is 16 bit
bne \$s0,\$s1,Exit #go to Exit if \$s0 ≠ s1

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Forces the program instruction count within 2^{16} instructions (2^{18} Bytes=128 KB)
- Most of the conditional branches jump nearby (whether forward or backward)
- Use PC relative addressing to access forward (PC + relative constant address), or backward (PC - relative constant address) location
- Can support branching up to $\pm 2^{15}$ relative constant address value
- PC normally increments 1 word (4 bytes) after every instruction
- The addressing should be (PC+4 + relative constant address) for forward access and (PC + 4 - relative constant address) for backward access

Addressing in Jumps

- ❑ PC is 32 bit but the address in Jumps is 16 bit

- ❑ PC is 32 bit but the address in Jumps is 16 bit

- Forces the program instruction count within 2^{26} instructions (2^{28})

- ❑ Replaces the 28 rightmost byte of PC

- Known as pseudodirect addressing

- A program is not placed across an address boundary of 256 MB

- Otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register

opcode

Jumping Address

- Forces the program instruction count within 2^{26} instructions (2^{28})



Addressing in Jumps

□ PC is 32 bit but the address in Jumps is 16 bit

□ PC is 32 bit but the address in Jumps is 16 bit

➤ Forces the program instruction count within 2^{26} instructions (2^{28})

□ Replaces the 28 rightmost byte of PC

➤ Known as pseudodirect addressing

➤ A program is not placed across an address boundary of 256 MB

➤ Otherwise, a jump must be replaced by a jump register instruction preceded

by other instructions to load the full 32-bit address into a register

opcode

Jumping Address

➤ Forces the program instruction count within 2^{26} instructions (2^{28})

□ Replaces the 28 rightmost byte of PC

➤ Known as pseudodirect addressing

➤ A program is not placed across an address boundary of 256 MB

➤ Otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register



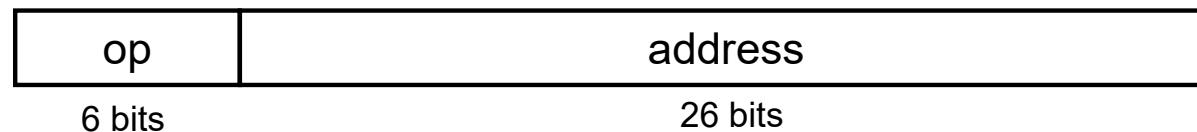
Food for Thought

- ❑ Can you reverse engineer a binary?
- ❑ Is your IP secured?

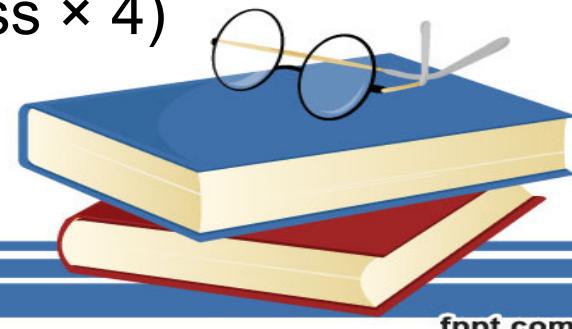


Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
 - Encode full address in instruction



- (Pseudo)Direct jump addressing
 - Target address = $PC_{31\dots 28} : (address \times 4)$

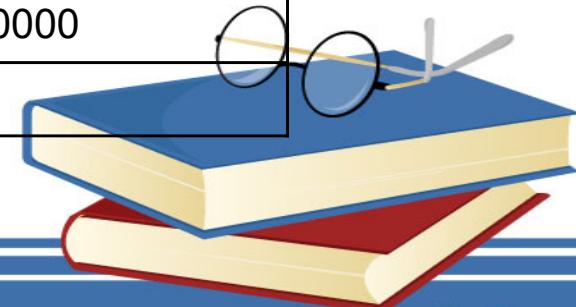


Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

Loop:	sll	\$t1, \$s3, 2	80000
	add	\$t1, \$t1, \$s6	80004
	lw	\$t0, 0(\$t1)	80008
	bne	\$t0, \$s5, Exit	80012
	addi	\$s3, \$s3, 1	80016
	j	Loop	80020
Exit:	...		80024

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8		0	
5	8	21		2	
8	19	19		1	
2			20000		



Branching Far Away

- ❑ If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- ❑ Example

```
beq $s0,$s1, L1  
      ↓  
bne $s0,$s1, L2  
j L1  
L2: ...
```



Addressing Mode Summary

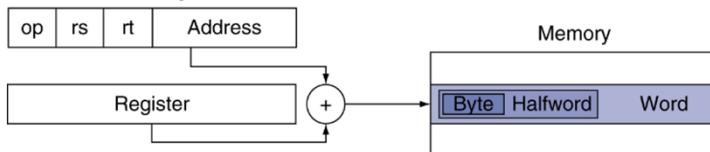
1. Immediate addressing



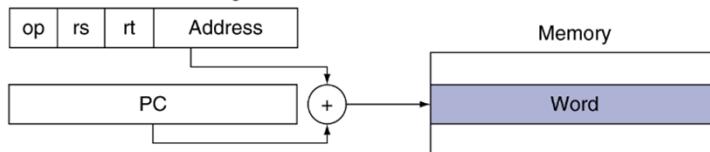
2. Register addressing



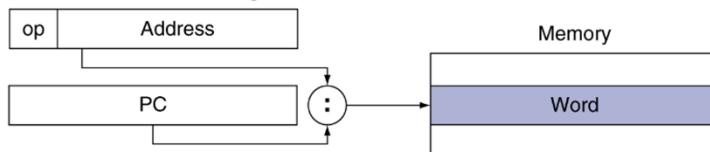
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Synchronization

- ❑ Two processors sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - Result depends of order of accesses
- ❑ Hardware support required
 - Atomic read/write memory operation
 - No other access to the location allowed between the read and write
- ❑ Could be a single instruction
 - E.g., atomic swap of register \leftrightarrow memory
 - Or an atomic pair of instructions



Synchronization in MIPS

- Load linked: **ll rt, offset(rs)**
- Store conditional: **sc rt, offset(rs)**

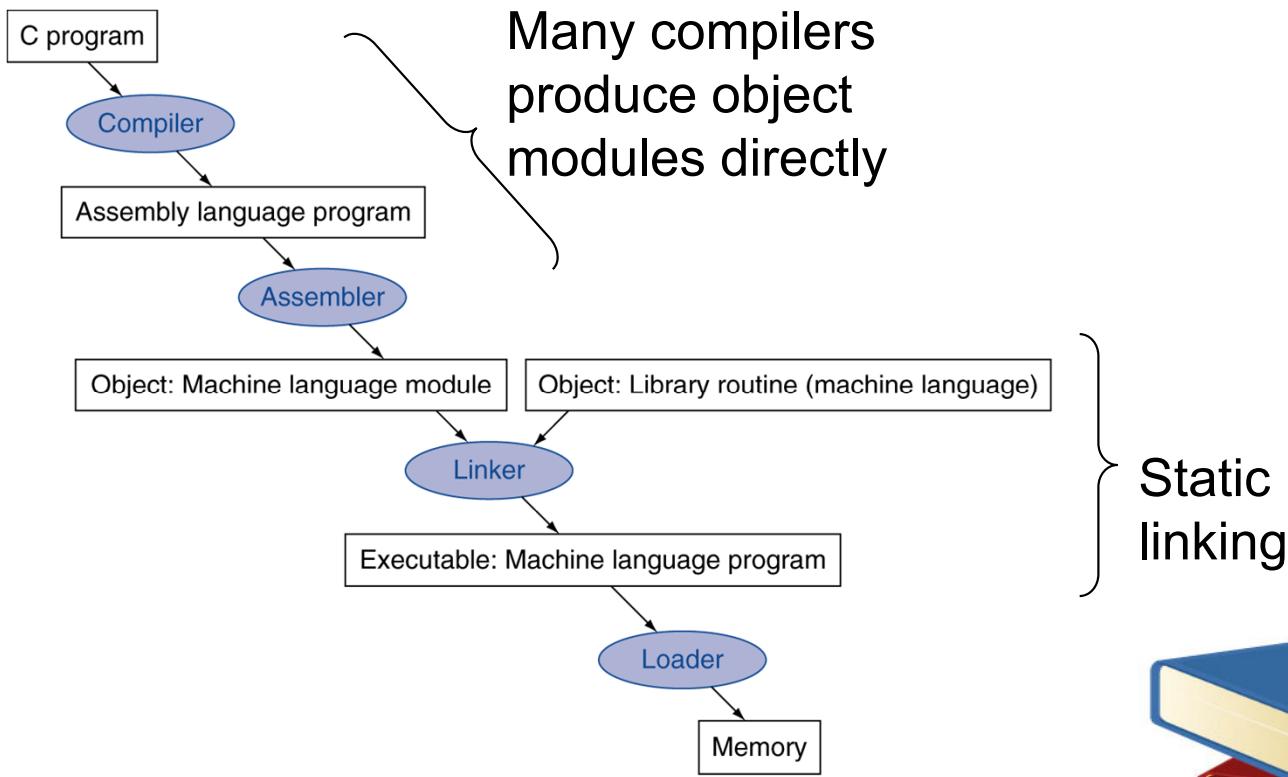
- Succeeds if location not changed since the **ll**
 - Returns 1 in rt
- Fails if location is changed
 - Returns 0 in rt

- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
      ll $t1,0($s1)    ;load linked
      sc $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```



Translation and Startup



Assembler Pseudoinstructions

- ❑ Most assembler instructions represent machine instructions one-to-one
- ❑ Pseudoinstructions: figments of the assembler's imagination

`move $t0, $t1 → add $t0, $zero, $t1`

`blt $t0, $t1, L → slt $at, $t0, $t1
bne $at, $zero, L`

➤ \$at (register 1): assembler temporary



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code



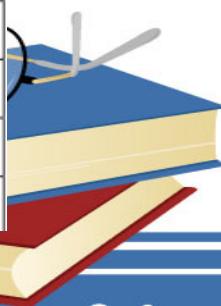
Linking Object Modules

- ❑ Produces an executable image
 1. Merges segments
 2. Resolve labels (determine their addresses)
 3. Patch location-dependent and external refs
- ❑ Could leave location dependencies for fixing by a relocating loader
 - But with virtual memory, no need to do this
 - Program can be loaded into absolute location in virtual memory space



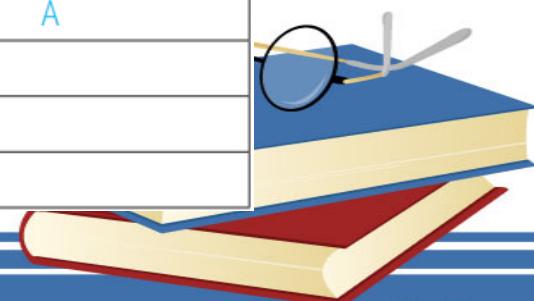
Object 1

Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	-	
	B	-	



Object 2

Object file header			
Text size	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	-	
	A	-	

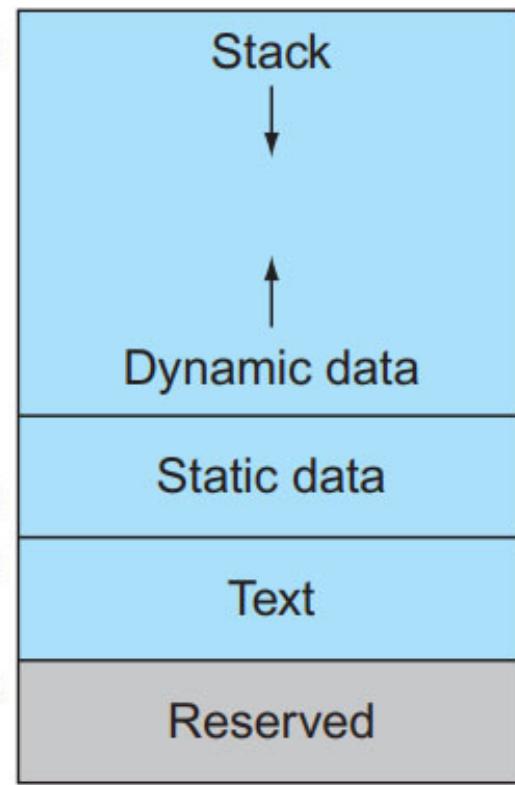


\$sp → 7fff fffc_{hex}

\$gp → 1000 8000_{hex}
1000 0000_{hex}

pc → 0040 0000_{hex}

0



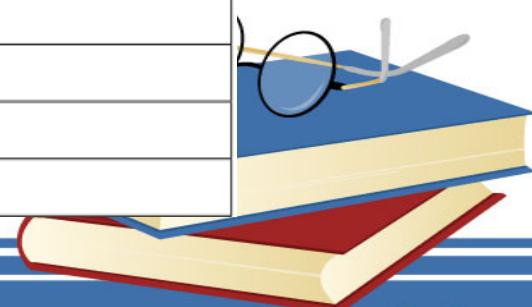
Merged Object

Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

	0040 0100 _{hex}	sw \$a1, 8020 _{hex} (\$gp)
	0040 0104 _{hex}	jal 40 0000 _{hex}

Data segment	Address	
	1000 0000 _{hex}	(X)

	1000 0020 _{hex}	(Y)



Loading a Program

❑ Load from image file on disk into memory

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialized data into memory
 - Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including \$sp, \$fp, \$gp)
6. Jump to startup routine
 - Copies arguments to \$a0, ... and calls main
 - When main returns, do exit syscall



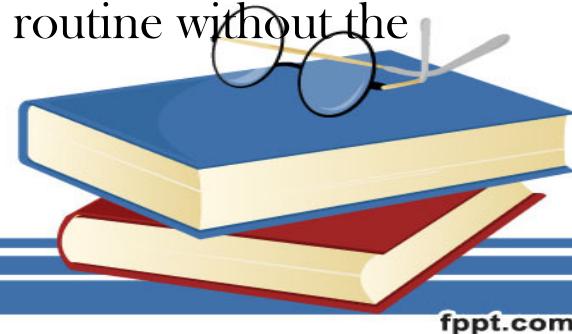
Dynamic Linking

- ❑ Only link/load library procedure when it is called
 - Requires procedure code to be relocatable
 - Avoids image bloat caused by static linking of all (transitively) referenced libraries
 - Automatically picks up new library versions

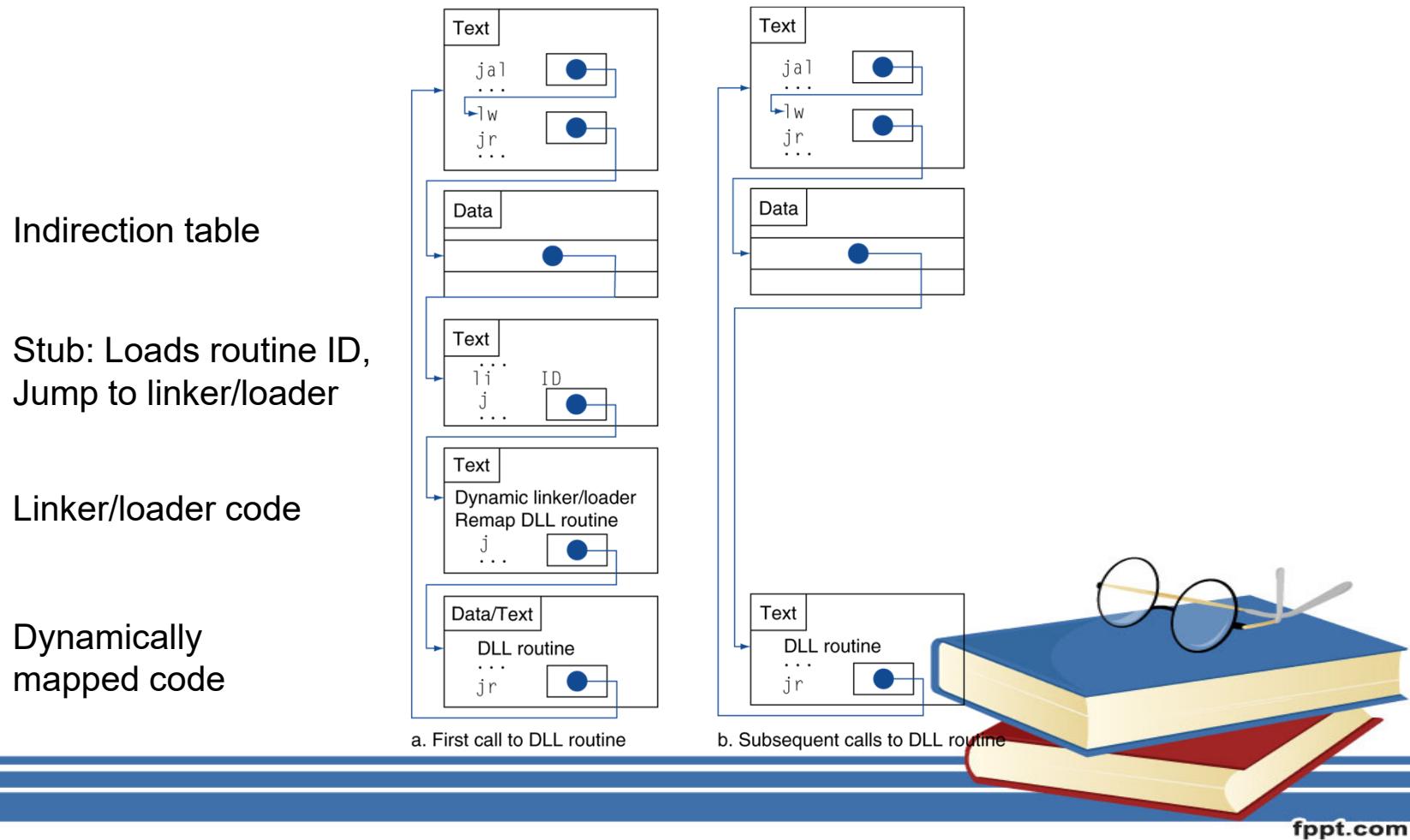


Dynamically Linked Libraries (DLL)

- Library routines that are linked to a program during execution
 - To incorporate the updated version of library routines
- In **lazy procedure linkage**, each routine is linked only when it is called
 - Provides a level of indirection
 - Nonlocal routine calls a set of dummy entries at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect jump
 - The Dynamic Linker/Loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine
 - From then, the call to the library routine jump indirectly to routine without the extra hops



Lazy Linkage

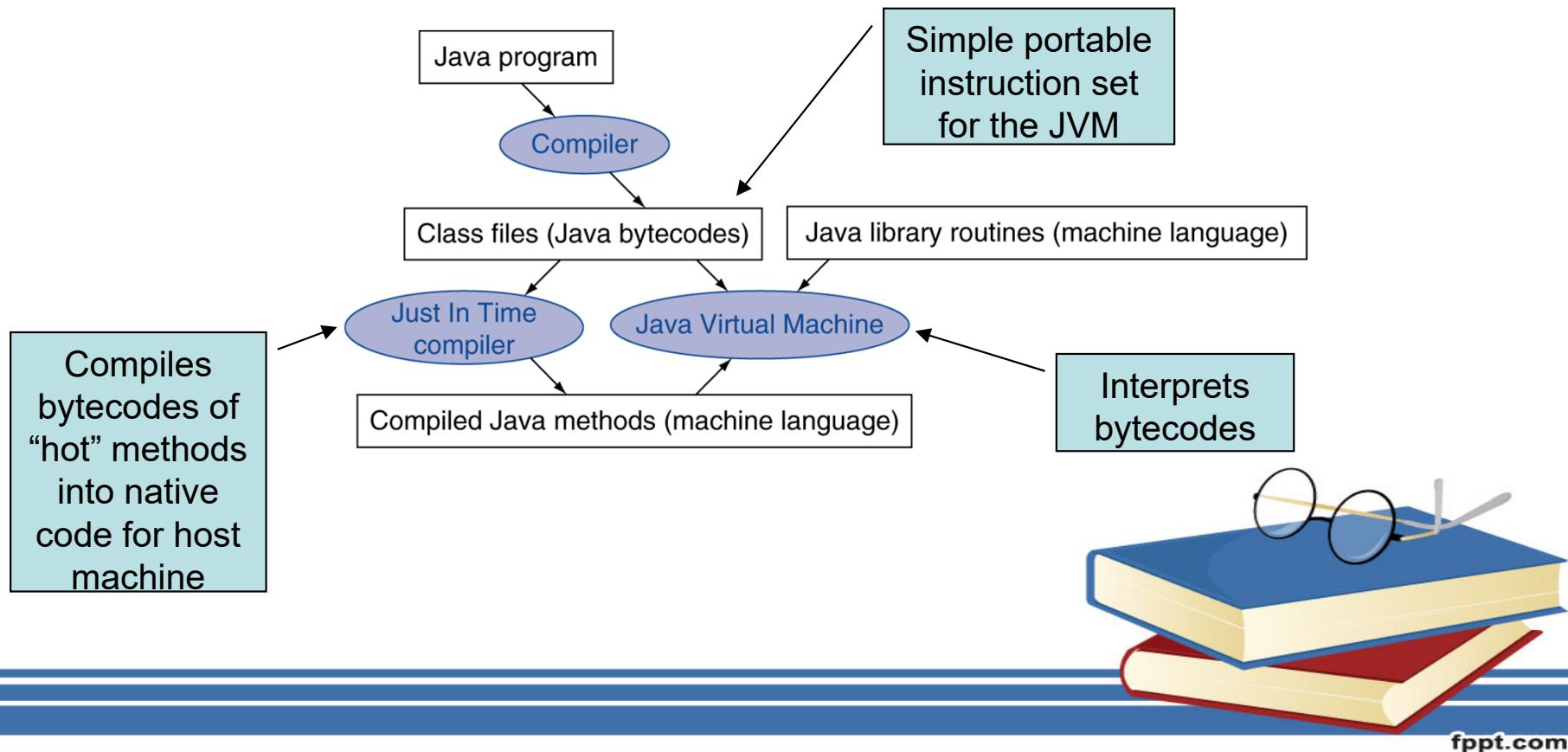


Starting a Java Program

- ❑ Java programs ensure **portability** sacrificing some **performance**
- ❑ Compiled first in to an easy-to-interpret instruction set: **Java bytecode**
- ❑ A software interpreter, called **Java Virtual Machine (JVM)** can execute Java byte code
 - This process is slow
- ❑ **Just In Time Compiler (JIT)** makes it faster
 - Statistically identify the commonly used (hot) methods
 - Compiles these methods into native instruction set



Starting Java Applications





C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

➤ v in \$a0, k in \$a1, temp in \$t0

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                           #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra              # return to calling routine
```



The Sort Procedure in C

- ❑ Non-leaf (calls swap)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            swap(v, j);
        }
    }
}
```

➤ v in \$a0, k in \$a1, i in \$s0, j in \$s1



The Full Procedure

```
sort:    addi $sp,$sp, -20      # make room on stack for 5 registers
        sw $ra, 16($sp)       # save $ra on stack
        sw $s3,12($sp)        # save $s3 on stack
        sw $s2, 8($sp)         # save $s2 on stack
        sw $s1, 4($sp)         # save $s1 on stack
        sw $s0, 0($sp)         # save $s0 on stack
...
...
exit1:   lw $s0, 0($sp)        # restore $s0 from stack
        lw $s1, 4($sp)         # restore $s1 from stack
        lw $s2, 8($sp)         # restore $s2 from stack
        lw $s3,12($sp)        # restore $s3 from stack
        lw $ra,16($sp)        # restore $ra from stack
        addi $sp,$sp, 20       # restore stack pointer
        jr $ra                 # return to calling routine
```



The Procedure Body

```

move $s2, $a0          # save $a0 into $s2
move $s3, $a1          # save $a1 into $s3
move $s0, $zero         # i = 0
for1tst: slt $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
    beq $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
    addi $s1, $s0, -1     # j = i - 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
    bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
    sll $t1, $s1, 2        # $t1 = j * 4
    add $t2, $s2, $t1        # $t2 = v + (j * 4)
    lw $t3, 0($t2)          # $t3 = v[j]
    lw $t4, 4($t2)          # $t4 = v[j + 1]
    slt $t0, $t4, $t3        # $t0 = 0 if $t4 ≥ $t3
    beq $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
move $a0, $s2            # 1st param of swap is v (old $a0)
move $a1, $s1            # 2nd param of swap is j
jal swap                # call swap procedure
addi $s1, $s1, -1        # j -= 1
j for2tst               # jump to test of inner loop
exit2: addi $s0, $s0, 1      # i += 1
j for1tst               # jump to test of outer loop

```

Move
params

Outer loop

Inner loop

Pass
params
& call

Inner loop

Outer loop

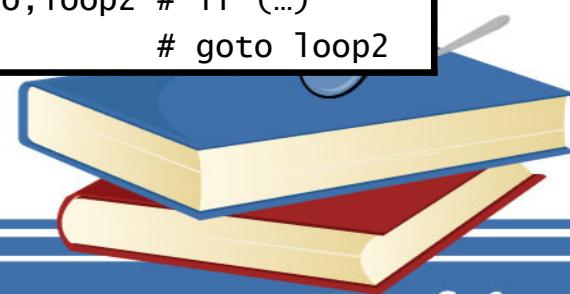
Example: Clearing an Array

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
move $t0,$zero    # i = 0  
loop1: sll $t1,$t0,2    # $t1 = i * 4  
        add $t2,$a0,$t1    # $t2 =  
                            # &array[i]  
        sw $zero, 0($t2)  # array[i] = 0  
        addi $t0,$t0,1    # i = i + 1  
        slt $t3,$t0,$a1    # $t3 =  
                            # (i < size)  
        bne $t3,$zero,loop1 # if (...)  
                            # goto loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
         p = p + 1)  
        *p = 0;  
}
```

```
move $t0,$a0      # p = & array[0]  
sll $t1,$a1,2      # $t1 = size * 4  
add $t2,$a0,$t1    # $t2 =  
                    # &array[size]  
loop2: sw $zero,0($t0)  # Memory[p] = 0  
       addi $t0,$t0,4    # p = p + 4  
       slt $t3,$t0,$t2    # $t3 =  
                           #(p<&array[size])  
       bne $t3,$zero,loop2 # if (...)  
                           # goto loop2
```



Comparison of Array vs. Ptr

- ❑ Multiply “strength reduced” to shift
- ❑ Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- ❑ Compiler can achieve same effect as manual use of pointers
 - Induction variable elimination
 - Better to make program clearer and safer



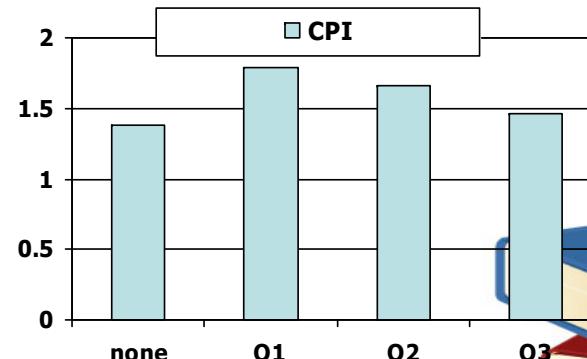
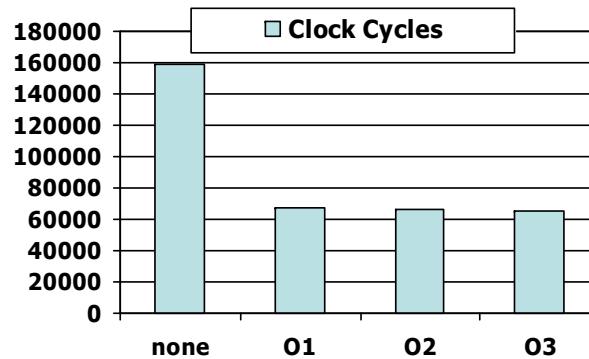
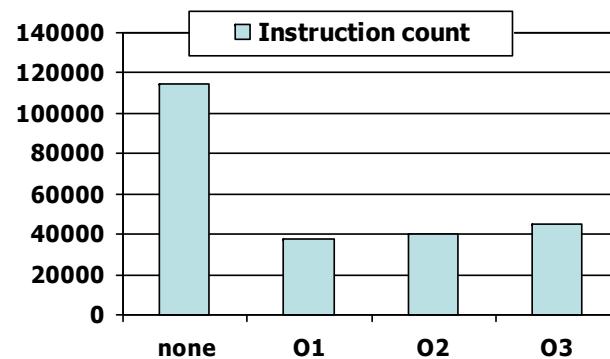
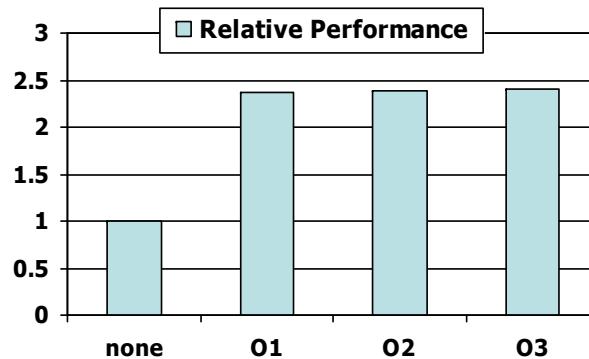
MIPS (RISC) Design Principles In Summary

- ❑ Simplicity favors regularity
 - fixed size instructions
 - small number of instruction formats
 - opcode always the first 6 bits
- ❑ Smaller is faster
 - limited instruction set
 - limited number of registers
 - limited number of addressing modes
- ❑ Good design demands good compromises
 - Three instruction formats
- ❑ Make the common case fast
 - arithmetic operands using the registers
 - Saving the commonly used registers into stack
 - PC-relative addressing
 - allow instructions to contain immediate operands

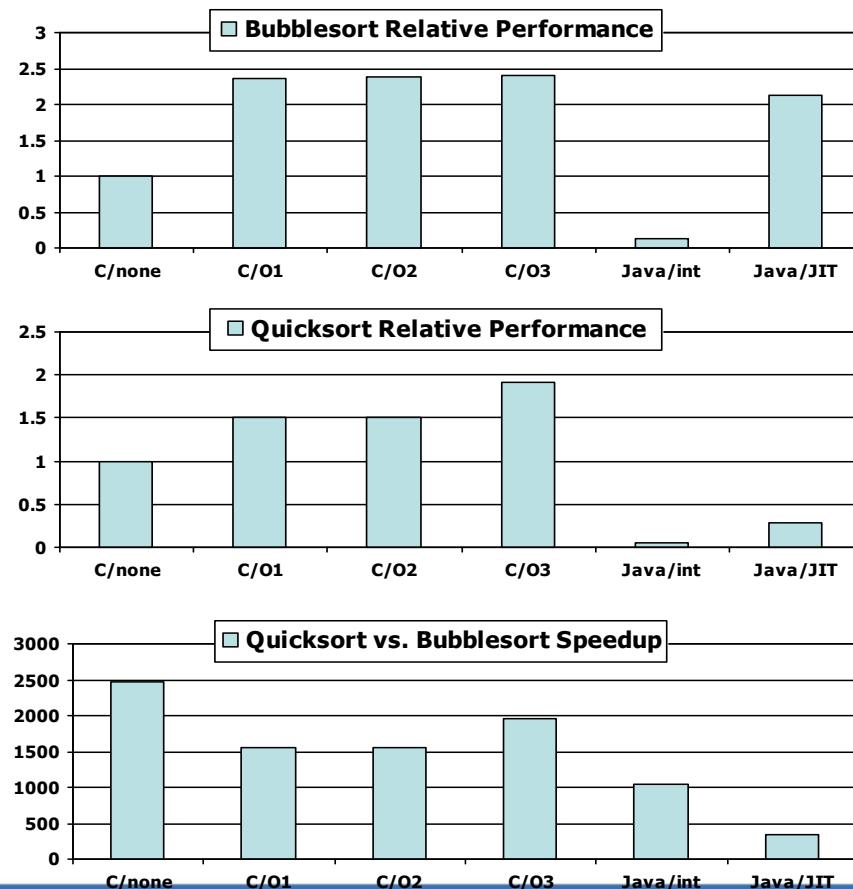


Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux



Effect of Language and Algorithm



Lessons Learnt

- ❑ Instruction count and CPI are not good performance indicators in isolation
- ❑ Compiler optimizations are sensitive to the algorithm
- ❑ Java/JIT compiled code is significantly faster than JVM interpreted
 - Comparable to optimized C in some cases
- ❑ Nothing can fix a dumb algorithm!



Alternative Architectures

□ Design alternative

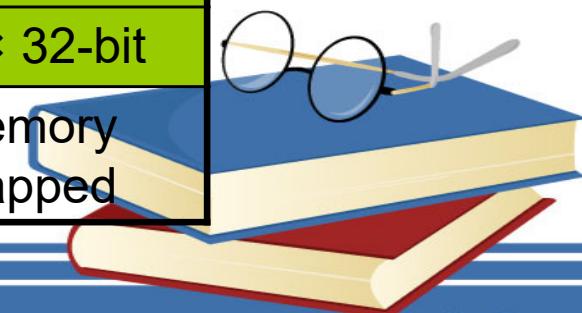
- provide more powerful operations
- goal is to reduce number of instructions executed
- danger is a slower cycle time and/or a higher CPI



ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	$15 \times 32\text{-bit}$	$31 \times 32\text{-bit}$
Input/output	Memory mapped	Memory mapped

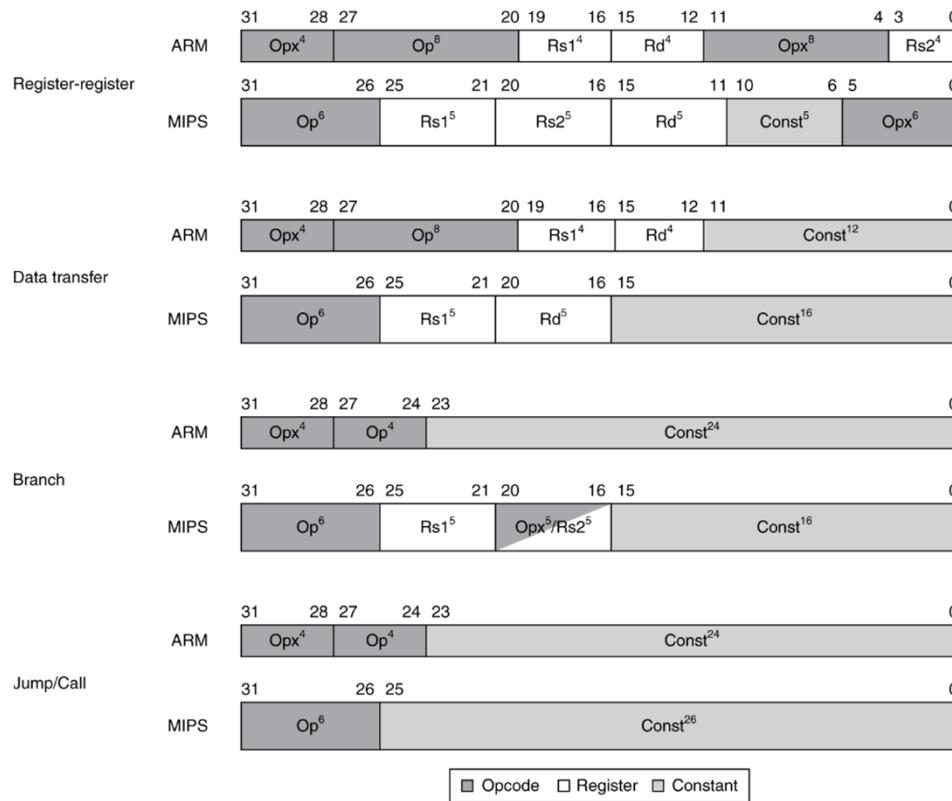


Compare and Branch in ARM

- ❑ Uses condition codes for result of an arithmetic/logical instruction
 - Negative, zero, carry, overflow
 - Compare instructions to set condition codes without keeping the result
- ❑ Each instruction can be conditional
 - Top 4 bits of instruction word: condition value
 - Can avoid branches over single instructions



Instruction Encoding



The Intel x86 ISA

□ Evolution with backward compatibility

- 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
- 8086 (1978): 16-bit extension to 8080
 - Complex instruction set (CISC)
- 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
- 80286 (1982): 24-bit addresses, MMU
 - Segmented memory mapping and protection
- 80386 (1985): 32-bit extension (now IA-32)
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments



The Intel x86 ISA

❑ Further evolution...

- i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
- Pentium (1993): superscalar, 64-bit datapath
 - Later versions added MMX (Multi-Media eXtension) instructions
 - **The infamous FDIV bug**
- Pentium Pro (1995), Pentium II (1997)
 - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- Pentium III (1999)
 - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
 - New microarchitecture
 - Added SSE2 instructions



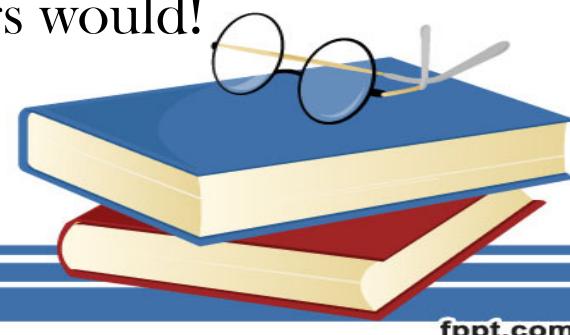
The Intel x86 ISA

❑ And further...

- AMD64 (2003): extended architecture to 64 bits
- EM64T - Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
- Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
- AMD64 (announced 2007): SSE5 instructions
 - Intel declined to follow, instead...
- Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions

❑ If Intel didn't extend with compatibility, its competitors would!

- Technical elegance ≠ market success



Basic x86 Registers

Name	Use
EAX	GPR 0
ECX	GPR 1
EDX	GPR 2
EBX	GPR 3
ESP	GPR 4
EBP	GPR 5
ESI	GPR 6
EDI	GPR 7
CS	Code segment pointer
SS	Stack segment pointer (top of stack)
DS	Data segment pointer 0
ES	Data segment pointer 1
FS	Data segment pointer 2
GS	Data segment pointer 3
EIP	Instruction pointer (PC)
EFLAGS	Condition codes



Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

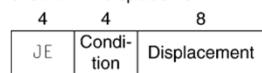
- Memory addressing modes

- Address in register
- Address = $R_{base} + \text{displacement}$
- Address = $R_{base} + 2^{\text{scale}} \times R_{index}$ (scale = 0, 1, 2, or 3)
- Address = $R_{base} + 2^{\text{scale}} \times R_{index} + \text{displacement}$



x86 Instruction Encoding

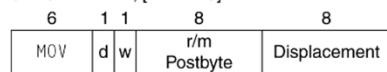
a. JE EIP + displacement



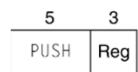
b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



❑ Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
 - Operand length, repetition, locking, ...



Implementing IA-32 (Seg:Off)

- ❑ Complex instruction set makes implementation difficult
 - Hardware translates instructions to simpler microoperations
 - Simple instructions: 1-1
 - Complex instructions: 1-many
 - Microengine similar to RISC
 - Market share makes this economically viable
- ❑ Comparable performance to RISC
 - Compilers avoid complex instructions



ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
 - Changes from v7:
 - No conditional execution field
 - Immediate field is 12-bit constant
 - Dropped load/store multiple
 - PC is no longer a GPR
 - GPR set expanded to 32
 - Addressing modes work for all word sizes
 - Divide instruction
 - Branch if equal/branch if not equal instructions



Fallacies

❑ Powerful instruction \Rightarrow higher performance

- Fewer instructions required
- But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
- Compilers are good at making fast code from simple instructions

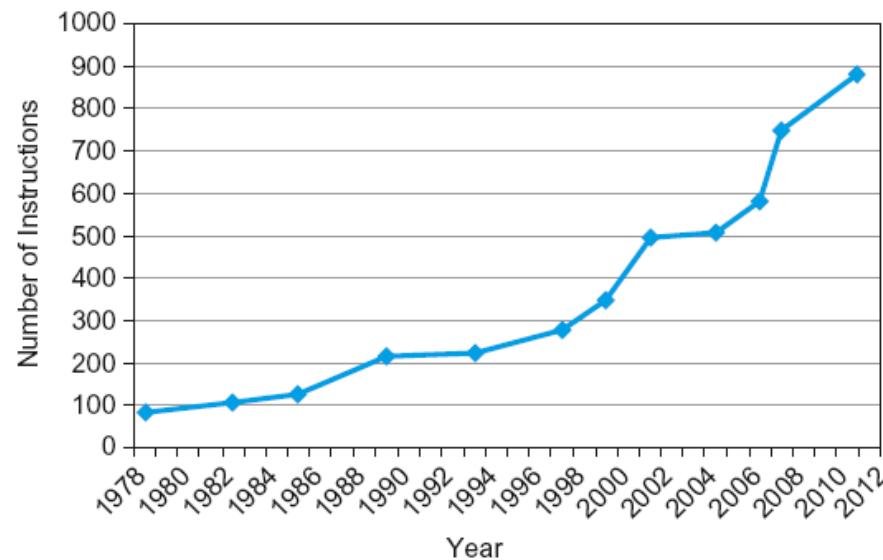
❑ Use assembly code for high performance

- But modern compilers are better at dealing with modern processors
- More lines of code \Rightarrow more errors and less productivity



Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set



Pitfalls

- ❑ Sequential words are not at sequential addresses
 - Increment by 4, not by 1!
- ❑ Keeping a pointer to an automatic variable after procedure returns
 - e.g., passing pointer back via an argument
 - Pointer becomes invalid when stack popped



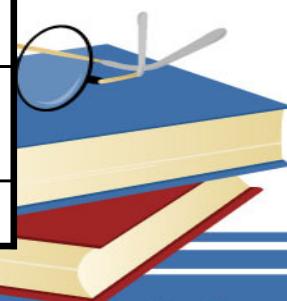
- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
 - c.f. x86



Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
 - Consider making the common case fast
 - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne,slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%



Acknowledgements

- These slides contain material developed and copyright by:
 - Lecture slides by Dr. Tanzima Hashem, Professor, CSE, BUET
 - Lecture slides by Mehnaz Tabassum Mahin, Assistant Professor, CSE, BUET



Thank You ☺





Chapter 3

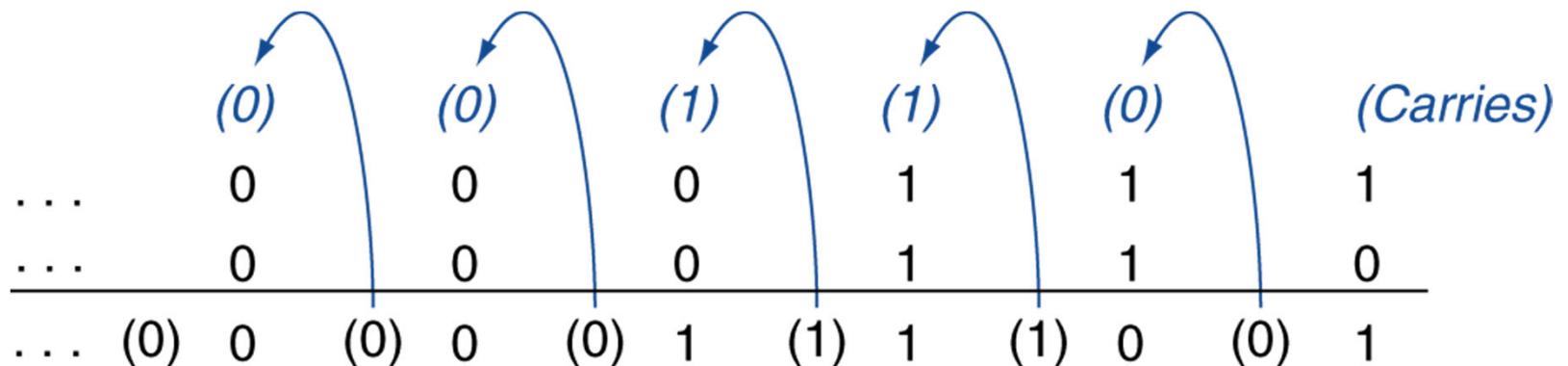
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Multiplication and division
 - Dealing with overflow
- Floating-point real numbers
 - Representation and operations

Integer Addition

Example: $7 + 6$



Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands
 - Overflow if result sign is 1
- Adding two –ve operands
 - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: $7 - 6 = 7 + (-6)$
 - +7: 0000 0000 ... 0000 0111
 - 6: 1111 1111 ... 1111 1010
 -
 - +1: 0000 0000 ... 0000 0001
- Overflow if result out of range
 - Subtracting two +ve or two -ve operands, no overflow
 - Subtracting +ve from -ve operand
 - Overflow if result sign is 0
 - Subtracting -ve from +ve operand
 - Overflow if result sign is 1

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
 - Use MIPS addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
 - Use MIPS add, addi, sub instructions
 - On overflow, invoke exception handler
 - Save PC in exception program counter (EPC) register
 - Jump to predefined handler address
 - mfc0 (move from system control reg) instruction can retrieve EPC value, to return after corrective action

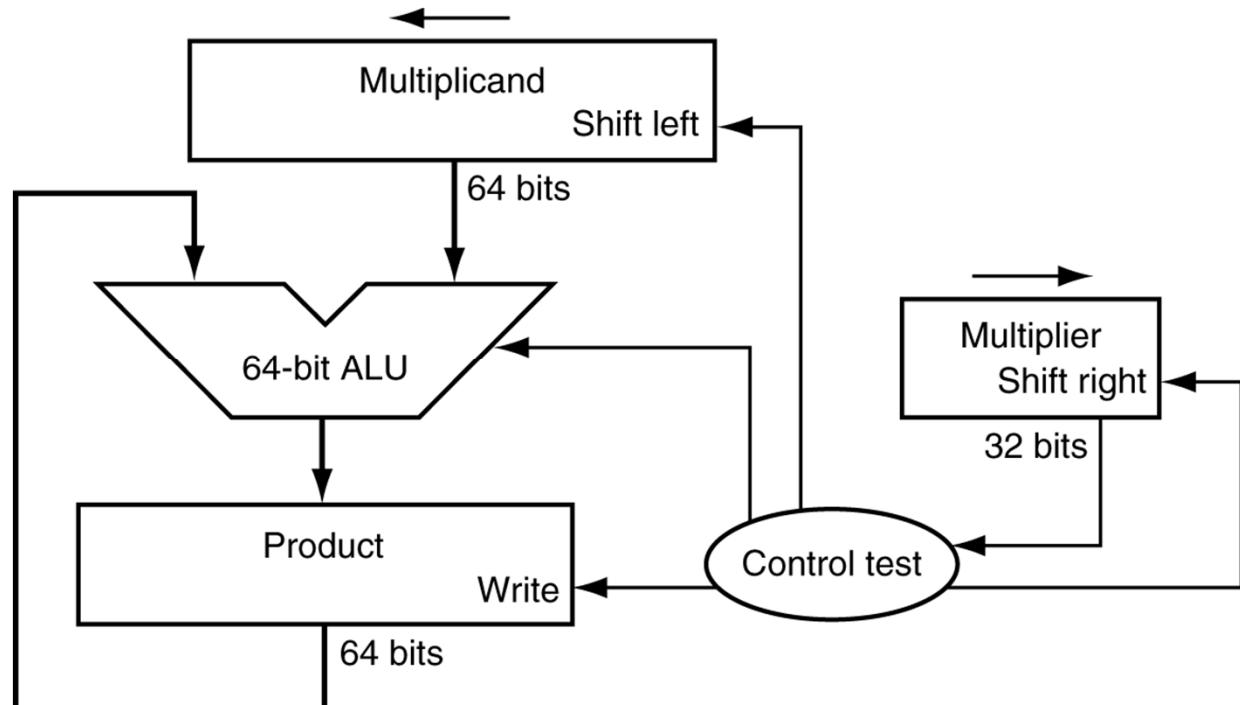


Multiplication

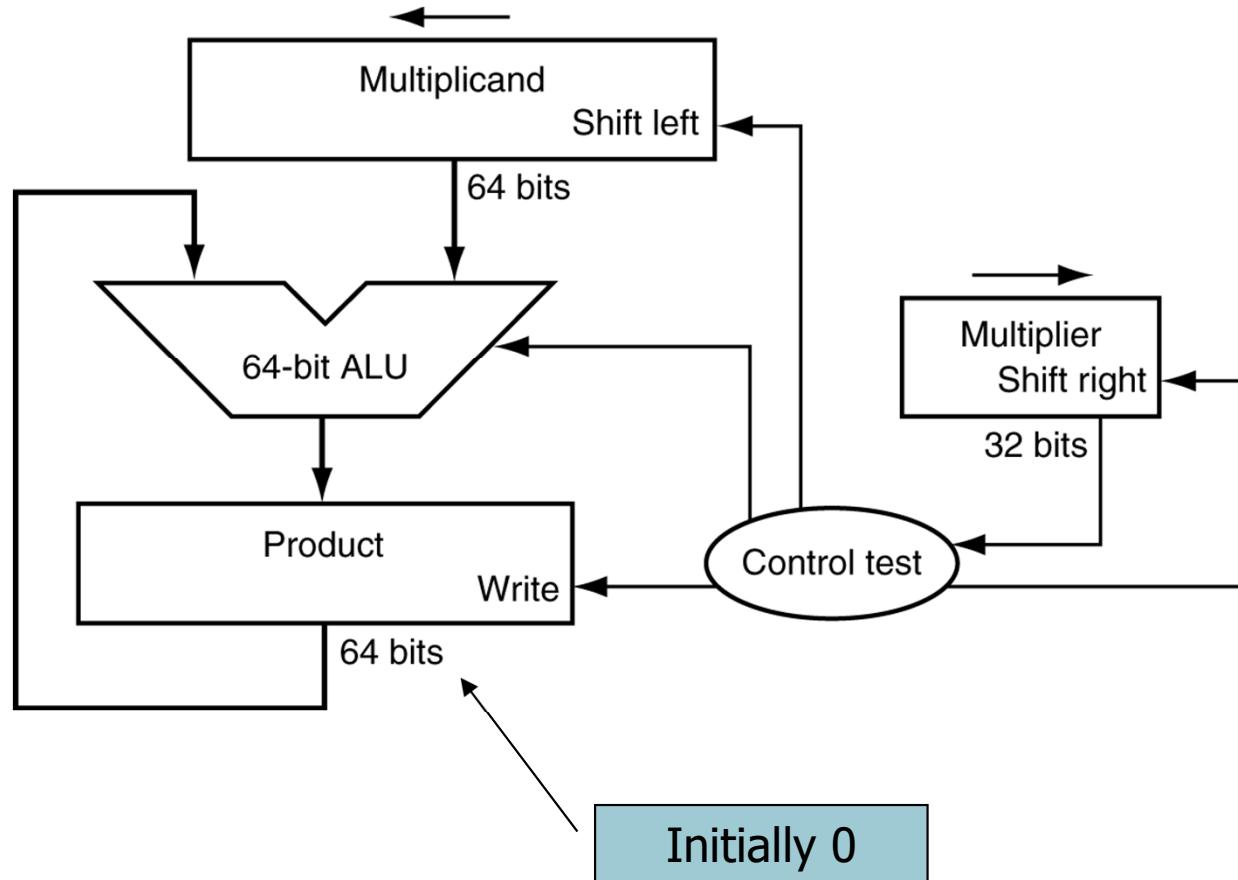
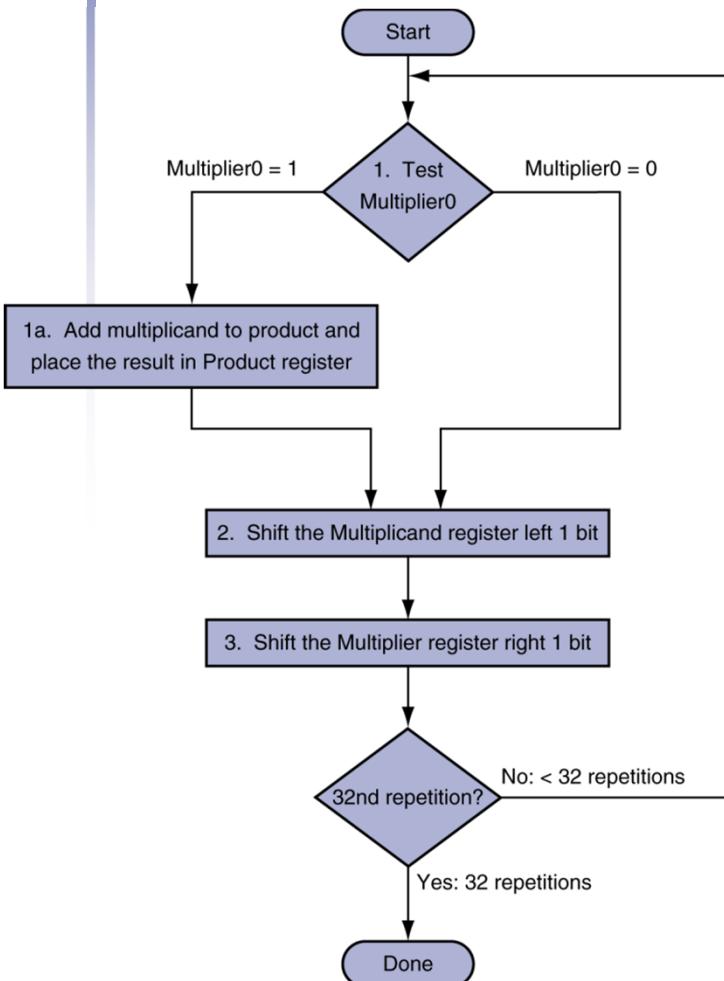
- Start with long-multiplication approach

multiplicand	1000
multiplier	x 1001
	<hr/>
	1000
	0000
	0000
product	1000
	<hr/>
	1001000

Length of product is
the sum of operand
lengths

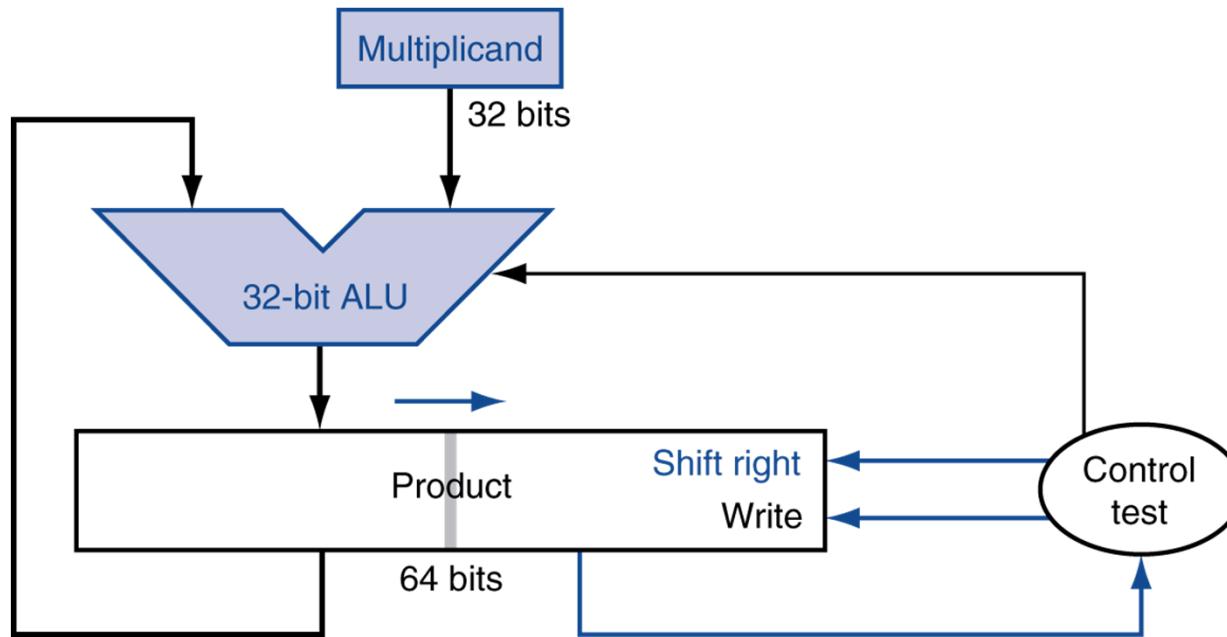


Multiplication Hardware



Optimized Multiplier

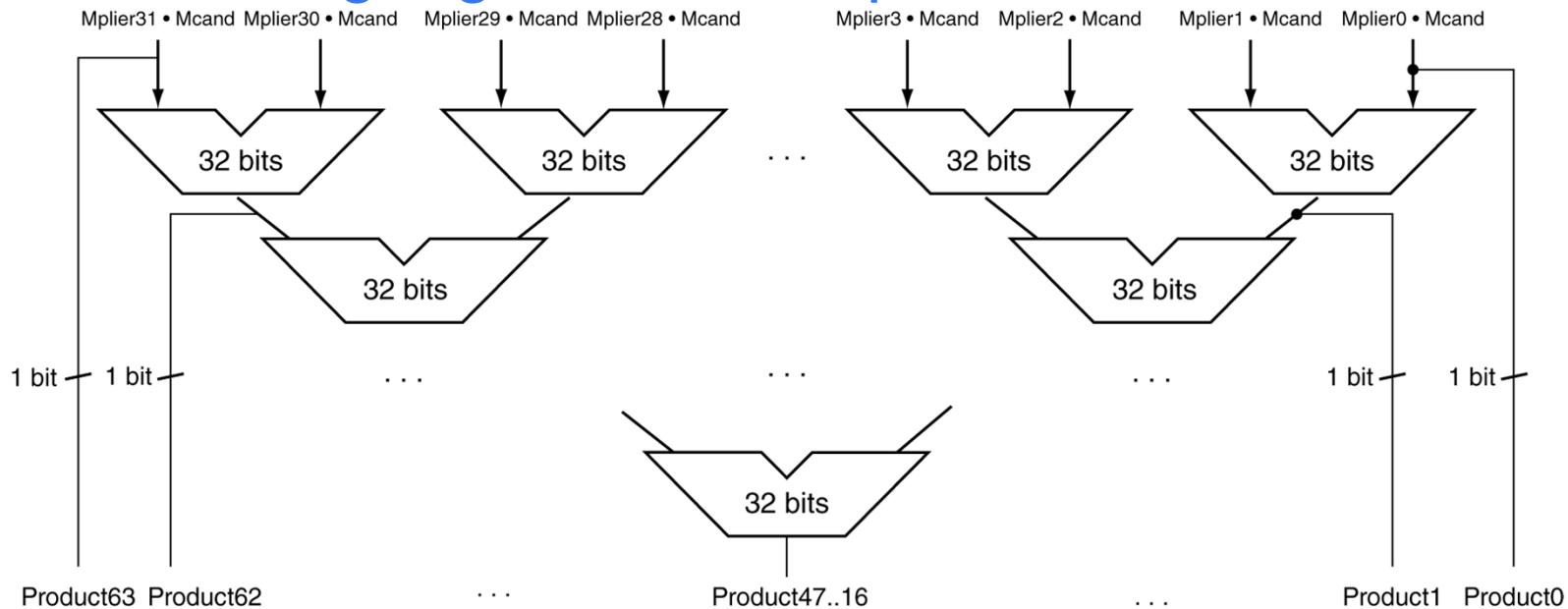
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
 - That's ok, if frequency of multiplications is low

Faster Multiplier

- Uses multiple adders
 - Cost/performance tradeoff
 - Following Figure is incomplete

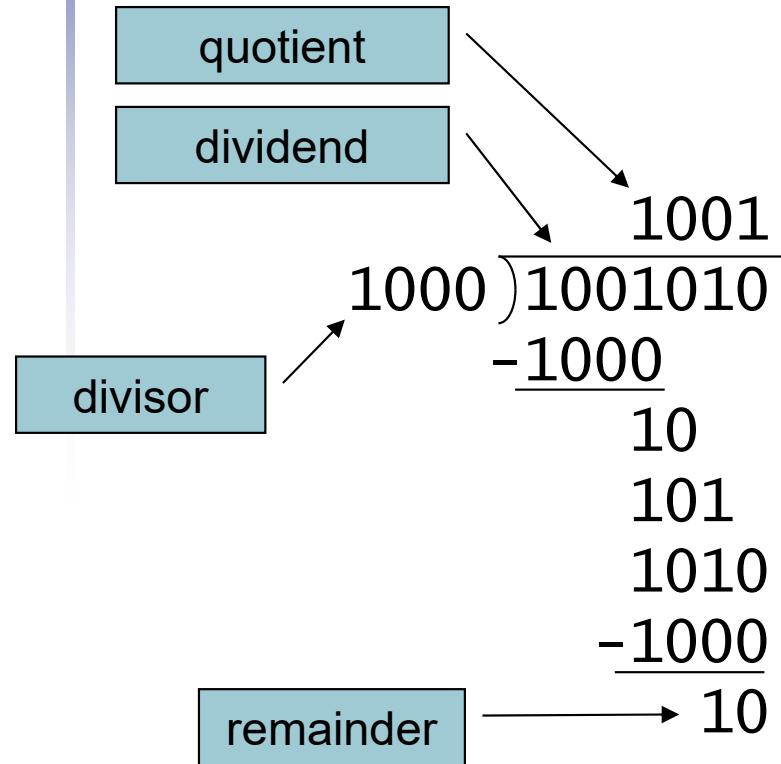


- Can be pipelined
 - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
 - HI: most-significant 32 bits
 - LO: least-significant 32-bits
- Instructions
 - `mult rs, rt / multu rs, rt`
 - 64-bit product in HI/LO
 - `mfhi rd / mflo rd`
 - Move from HI/LO to rd
 - Can test HI value to see if product overflows 32 bits
 - `mul rd, rs, rt`
 - Least-significant 32 bits of product → rd

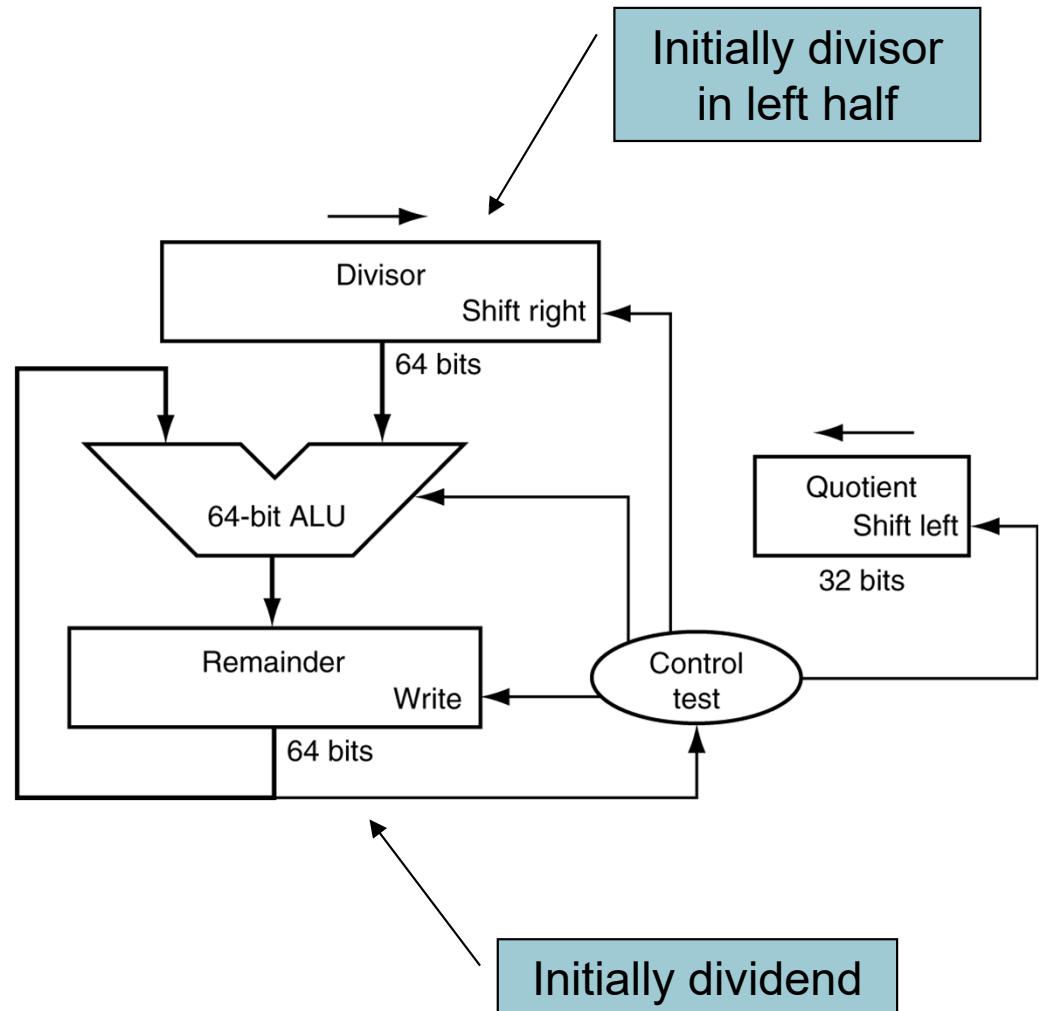
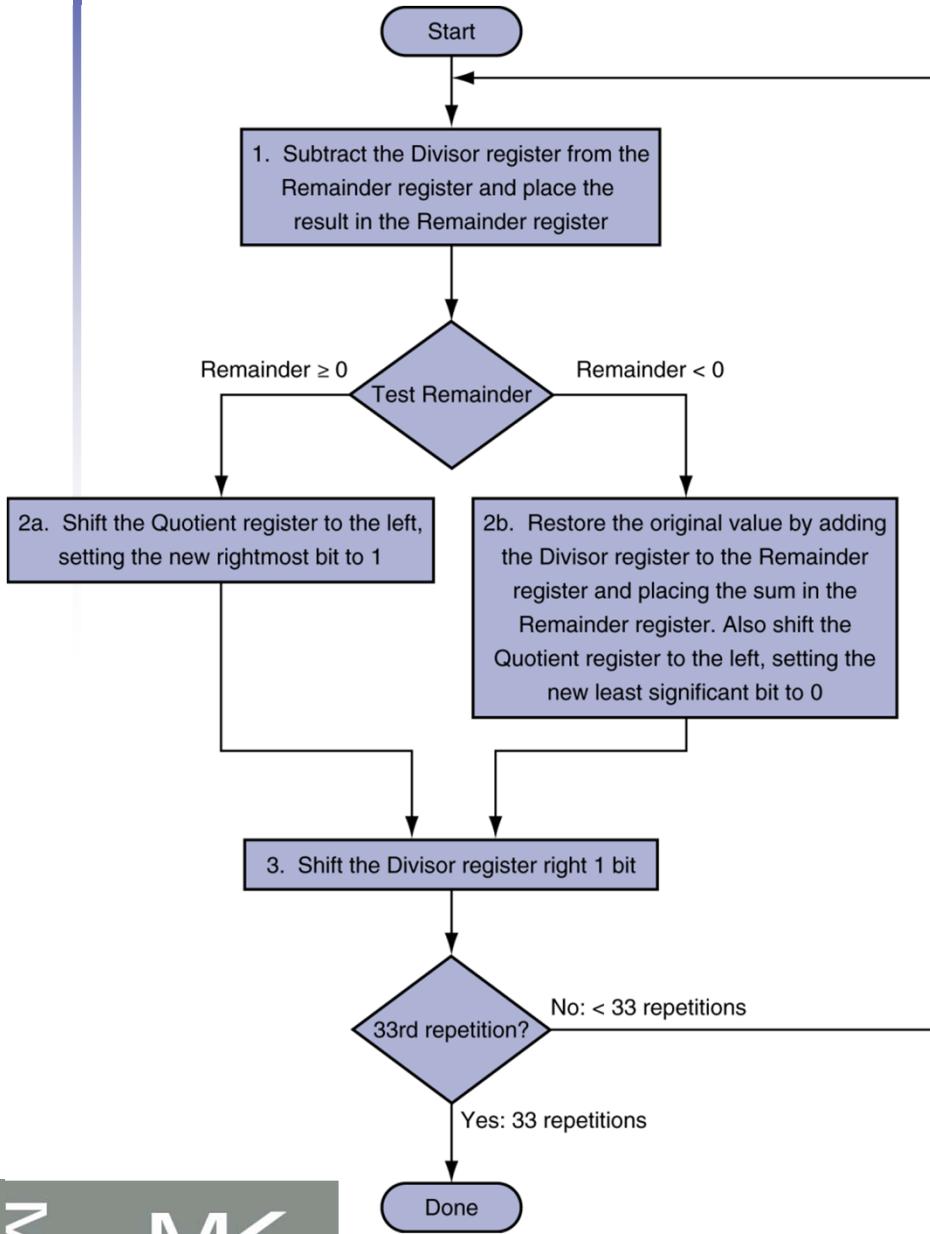
Division



n -bit operands yield n -bit quotient and remainder

- Check for 0 divisor
- Long division approach
 - If divisor \leq dividend bits
 - 1 bit in quotient, subtract
 - Otherwise
 - 0 bit in quotient, bring down next dividend bit
- Restoring division
 - Do the subtract, and if remainder goes < 0 , add divisor back
- Signed division
 - Divide using absolute values
 - Adjust sign of quotient and remainder as required

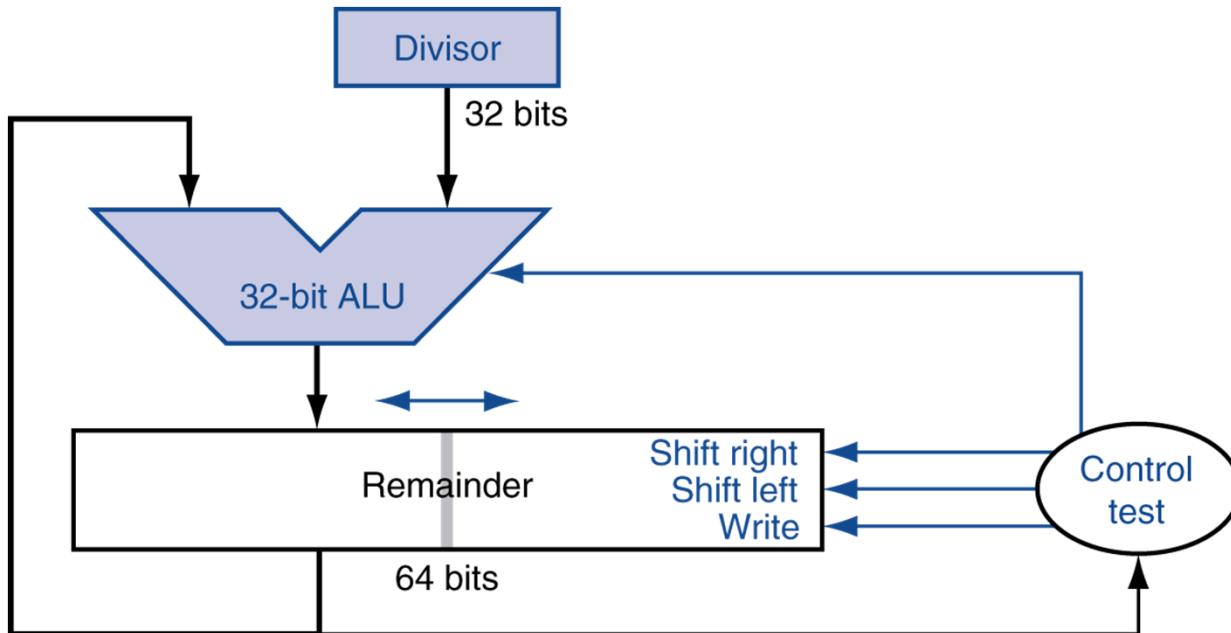
Division Hardware



Division Example

Iteration	Step	Quotient	Divisor	Reminder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	②000 0011
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	②000 0001
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
 - Same hardware can be used for both

Optimized Divider

Iteration	Step	Divisor)	Remainder (R/ Q)	
0	Init	0010	0000	0111
1	Rem = Rem - Sub	0010	Negative!	0111
	Restore Rem Left Shift and Set 0	0010	0000 0000	0111 1110
2	Rem = Rem – Sub	0010	Negative!	1110
	Restore Rem Left Shift and Set 0	0010	0000 0001	1110 1100
3	Rem = Rem – Sub	0010	Negative!	1100
	Restore Rem Left Shift and Set 0	0010	0001 0011	1100 1000
4	Rem = Rem – Sub	0010	0001	1000
	Left Shift and Set 1	0010	0011	0001
5	Rem = Rem – Sub	0010	0001	0001
	Left Shift and Set 1 (special shift)	0010	0001	0011

Optimized Multiplier

Iteration	Step	Multiplicand	Result / Multiplier	
0	Init	0111	0000	0101
1	LSB =1, Add	0111	0111	0101
	RS	0111	0011	1010
2	LSB =0, Pass	0111	0011	1010
	RS	0111	0001	1101
3	LSB =1, Add	0111	1000	1101
	RS	0111	0100	0110
4	LSB =0, Pass	0111	0100	0110
	RS	0111	0010	0011

MIPS Division

- Use HI/LO registers for result
 - HI: 32-bit remainder
 - LO: 32-bit quotient
- Instructions
 - `div rs, rt` / `divu rs, rt`
 - No overflow or divide-by-0 checking
 - Software must perform checks if required
 - Use `mfhi`, `mflo` to access result
 - Q in LO, R in HI



Floating Point

- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation
 - -2.34×10^{56} ← normalized
 - $+0.002 \times 10^{-4}$ ← not normalized
 - $+987.02 \times 10^9$ ← not normalized
- In binary
 - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
 - Portability issues for scientific code
- Now almost universally adopted
- Two representations
 - Single precision (32-bit)
 - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits

double: 11 bits

single: 23 bits

double: 52 bits

S	Exponent	Fraction
---	----------	----------

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 \Rightarrow non-negative, 1 \Rightarrow negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
 - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
 - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
 - Ensures exponent is unsigned
 - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
 - Exponent: 00000001
 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
 - exponent: 11111110
 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Double-Precision Range

- Exponents 0000...00 and 1111...11 reserved
- Smallest value
 - Exponent: 00000000001
 \Rightarrow actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
 - Exponent: 11111111110
 \Rightarrow actual exponent = $2046 - 1023 = +1023$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

■ Relative precision

- all fraction bits are significant
- Single: approx 2^{-23}
 - A float has 23 bits of mantissa, and 2^{23} is 8,388,608. 23 bits let you store all 6 digit numbers or lower, and most of the 7 digit numbers. This means that floating point numbers have between 6 and 7 digits of precision, regardless of exponent.
 - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double: approx 2^{-52}
 - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision



Floating-Point Example

- Represent -0.75
 - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000\dots00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: $101111101000\dots00$
- Double: $1011111111101000\dots00$

Floating-Point Example

- What number is represented by the single-precision float

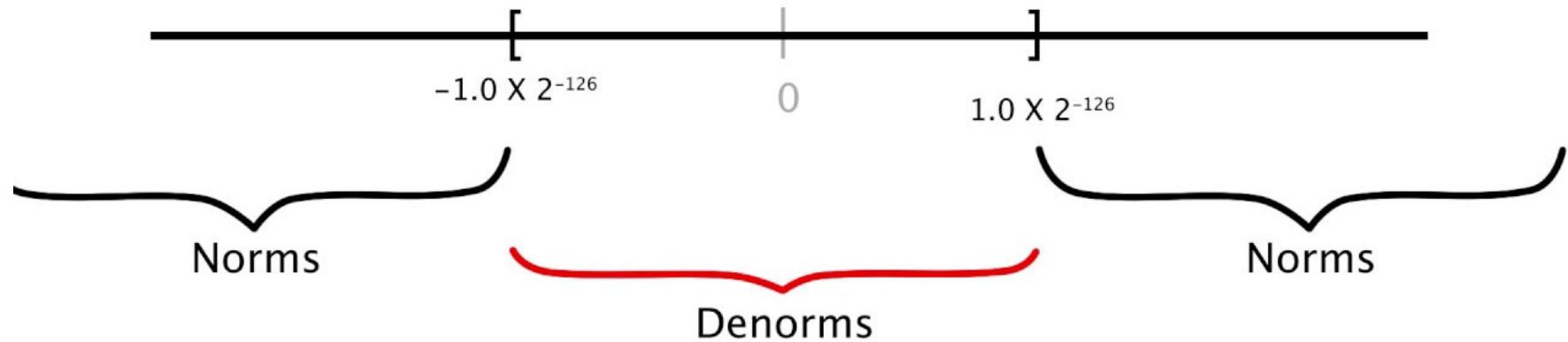
1 100000010100...00

- S = 1
- Fraction = 01000...00₂
- Exponent = 10000001₂ = 129
- $$\begin{aligned}x &= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\&= (-1) \times 1.25 \times 2^2 \\&= -5.0\end{aligned}$$

Denormal Numbers

Denormalized Numbers $\pm 0.B \times 2^e$

Every denormalized number is smaller in absolute value than every normalized number.



Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^s \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
 - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^s \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations
of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
 - $\pm\infty$
 - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction \neq 000...0
 - Not-a-Number (NaN)
 - Indicates illegal or undefined result
 - e.g., $0.0 / 0.0$
 - Can be used in subsequent calculations

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

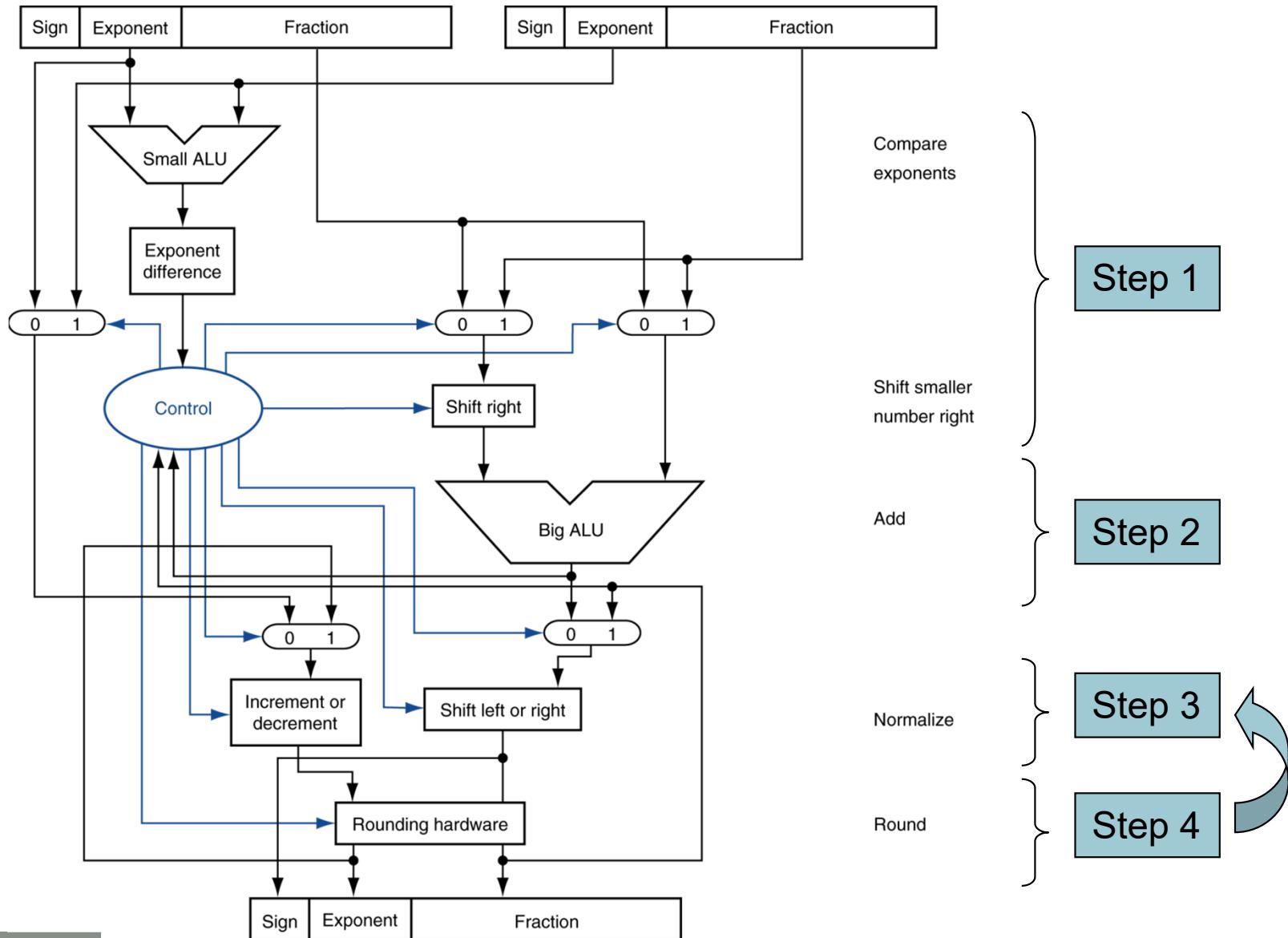
Floating-Point Addition

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ ($0.5 + -0.4375$)
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles
 - Can be pipelined

FP Adder Hardware



Rounding Bits

- Guard and round digits, and sticky bit
- When computing result, assume there are several extra digits available for shifting and computation. This improves accuracy of computation.
- **Guard digit:** first extra digit/bit to the right of mantissa -- used for rounding addition results
- **Round digit:** second extra digit/bit to the right of mantissa -- used for rounding multiplication results
- **Sticky bit:** third extra digit/bit to the right of mantissa – used for resolving ties such as 0.50...00 vs. 0.50...01

Rounding Bits

- Rounding using Guard and round digits, and sticky bit

G	R	S	Action
0	0	0	Truncate
0	0	1	Truncate
0	1	0	Truncate
0	1	1	Truncate
1	0	0	Round to Even
1	0	1	Round Up
1	1	0	Round Up
1	1	1	Round Up

1.100GRS

1.10001 = 1.53125

1.10010 = 1.56250

1.10011 = 1.59375

Floating-Point Multiplication

- Consider a 4-digit decimal example
 - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
 - For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$
- 2. Multiply significands
 - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize result & check for over/underflow
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$



Floating-Point Multiplication

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

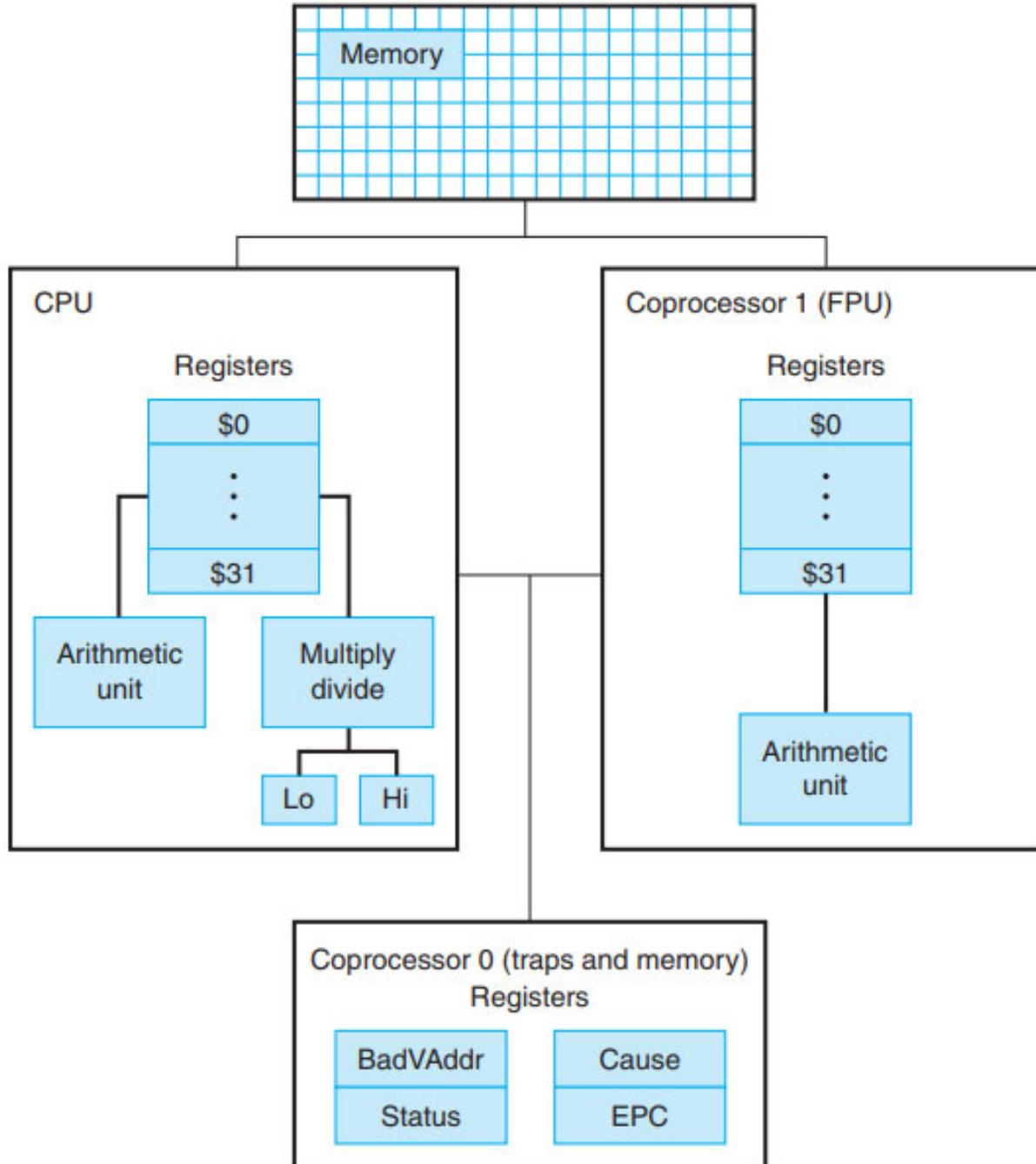
- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root
 - FP \leftrightarrow integer conversion
- Operations usually takes several cycles
 - Can be pipelined

Associativity of FP Addition

- Associativity: $a + (b + c) = (a + b) + c$
- Is FP addition associative?
 - No

FP Instructions in MIPS

- FP hardware is coprocessor 1
 - Adjunct processor that extends the ISA
- Separate FP registers
 - 32 single-precision: \$f0, \$f1, ... \$f31
 - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, ...
 - Release 2 of MIPS ISA supports 32×64
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - lwc1, ldc1, swc1, sdc1
 - e.g., ldc1 \$f8, 32(\$sp)



FP Instructions in MIPS

- Single-precision arithmetic
 - add.s, sub.s, mul.s, div.s
 - e.g., add.s \$f0, \$f1, \$f6
- Double-precision arithmetic
 - add.d, sub.d, mul.d, div.d
 - e.g., mul.d \$f4, \$f4, \$f6
- Single- and double-precision comparison
 - c.xx.s, c.xx.d (xx is eq, lt, le, ...)
 - Sets or clears FP condition-code bit
 - e.g. c.lt.s \$f3, \$f4
- Branch on FP condition code true or false
 - bc1t, bc1f
 - e.g., bc1t TargetLabel

Single and Double Precision operations

- A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.

```
lwcl      $f4,c($sp)  # Load 32-bit F.P. number into F4  
lwcl      $f6,a($sp)  # Load 32-bit F.P. number into F6  
add.s    $f2,$f4,$f6 # F2 = F4 + F6 single precision  
swcl      $f2,b($sp)  # Store 32-bit F.P. number from F2
```

- What if add.d was used?

Floating-Point Instructions in MIPS

- Floating-point *addition*, *single* (add.s) and *addition, double* (add.d)
 - e.g., add.s \$f0, \$f4, \$f6 # \$f2 = \$f4 + \$f6
- Floating-point *subtraction*, *single* (sub.s) and *subtraction, double* (sub.d)
 - e.g., sub.d \$f2, \$f4, \$f6 # \$f2 = \$f4 - \$f6
- Floating-point *multiplication*, *single* (mul.s) and *multiplication, double* (mul.d)
 - e.g., mul.s \$f2, \$f4, \$f6 # \$f2 = \$f4 X \$f6
- Floating-point *division*, *single* (div.s) and *division, double* (div.d)
 - e.g., div.d \$f2, \$f4, \$f6 # \$f2 = \$f4 / \$f6

Floating-Point Instructions in MIPS

- Floating-point *comparison*, *single* (c.x.s) and *comparison*, *double* (c.x.d),
where x may be *equal* (eq), *not equal* (neq), *less than* (lt), *less than or equal* (le), *greater than* (gt), or *greater than or equal* (ge)
 - e.g., c.lt.s \$f2, \$f4 # if (\$f2 < \$f4) cond = 1; else cond = 0
- Floating-point *branch*, *true* (bc1t) and *branch, false* (bc1f)
 - e.g., bc1t 25 # if (cond == 1) go to PC + 4 + 100

Floating-Point Instructions in MIPS

```
if (i==j)
    f = g + h;
else
    f = g - h;
```

c.eq.s \$f2, \$f4
bc1f Else #go to Else if $i \neq j$
add.s \$f6,\$f8,\$f10 # $f = g + h$
j Exit
Else: *sub.s \$f6,\$f8,\$f10* # $f = g - h$
Exit:

MIPS FP Operands

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	\$f0, \$f1, \$f2, . . . , \$f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS FP Operations

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2,\$f4,\$f6	$\$f2 = \$f4 + \$f6$	FP add (single precision)
	FP subtract single	sub.s \$f2,\$f4,\$f6	$\$f2 = \$f4 - \$f6$	FP sub (single precision)
	FP multiply single	mul.s \$f2,\$f4,\$f6	$\$f2 = \$f4 \times \$f6$	FP multiply (single precision)
	FP divide single	div.s \$f2,\$f4,\$f6	$\$f2 = \$f4 / \$f6$	FP divide (single precision)
	FP add double	add.d \$f2,\$f4,\$f6	$\$f2 = \$f4 + \$f6$	FP add (double precision)
	FP subtract double	sub.d \$f2,\$f4,\$f6	$\$f2 = \$f4 - \$f6$	FP sub (double precision)
	FP multiply double	mul.d \$f2,\$f4,\$f6	$\$f2 = \$f4 \times \$f6$	FP multiply (double precision)
	FP divide double	div.d \$f2,\$f4,\$f6	$\$f2 = \$f4 / \$f6$	FP divide (double precision)
Data transfer	load word copr. 1	lwcl \$f1,100(\$s2)	$\$f1 = \text{Memory}[\$s2 + 100]$	32-bit data to FP register
	store word copr. 1	swcl \$f1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$f1$	32-bit data to memory
Conditional branch	branch on FP true	bclt 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if ($\$f2 < \$f4$) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if ($\$f2 < \$f4$) cond = 1; else cond = 0	FP compare less than double precision

MIPS FP Instruction Format

MIPS floating-point machine language

Name	Format	Example							Comments
add.s	R	17	16	6	4	2	0		add.s \$f2,\$f4,\$f6
sub.s	R	17	16	6	4	2	1		sub.s \$f2,\$f4,\$f6
mul.s	R	17	16	6	4	2	2		mul.s \$f2,\$f4,\$f6
div.s	R	17	16	6	4	2	3		div.s \$f2,\$f4,\$f6
add.d	R	17	17	6	4	2	0		add.d \$f2,\$f4,\$f6
sub.d	R	17	17	6	4	2	1		sub.d \$f2,\$f4,\$f6
mul.d	R	17	17	6	4	2	2		mul.d \$f2,\$f4,\$f6
div.d	R	17	17	6	4	2	3		div.d \$f2,\$f4,\$f6
lwcl	I	49	20	2	100				lwcl \$f2,100(\$s4)
swcl	I	57	20	2	100				swcl \$f2,100(\$s4)
bclt	I	17	8	1	25				bclt 25
bclf	I	17	8	0	25				bclf 25
c.lt.s	R	17	16	4	2	0	60		c.lt.s \$f2,\$f4
c.lt.d	R	17	17	4	2	0	60		c.lt.d \$f2,\$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions	32 bits

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- fahr in \$f12, result in \$f0, literals in global memory space

- Compiled MIPS code:

```
f2c:  lwc1  $f16, const5($gp)  
      lwc2  $f18, const9($gp)  
      div.s $f16, $f16, $f18  
      lwc1  $f18, const32($gp)  
      sub.s $f18, $f12, $f18  
      mul.s $f0, $f16, $f18  
      jr    $ra
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
 - Extra bits of precision (guard, round, sticky)
 - Choice of rounding modes
 - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
 - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements



Concluding Remarks

- Bits have no inherent meaning
 - Interpretation depends on the instructions applied
- Computer representations of numbers
 - Finite range and precision
 - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
 - Signed and unsigned integers
 - Floating-point approximation to reals
- Bounded range and precision
 - Operations can overflow and underflow
- MIPS ISA
 - Core instructions: 54 most frequently used
 - 100% of SPECINT, 97% of SPECFP
 - Other instructions: less frequent



Chapter 4

The Processor

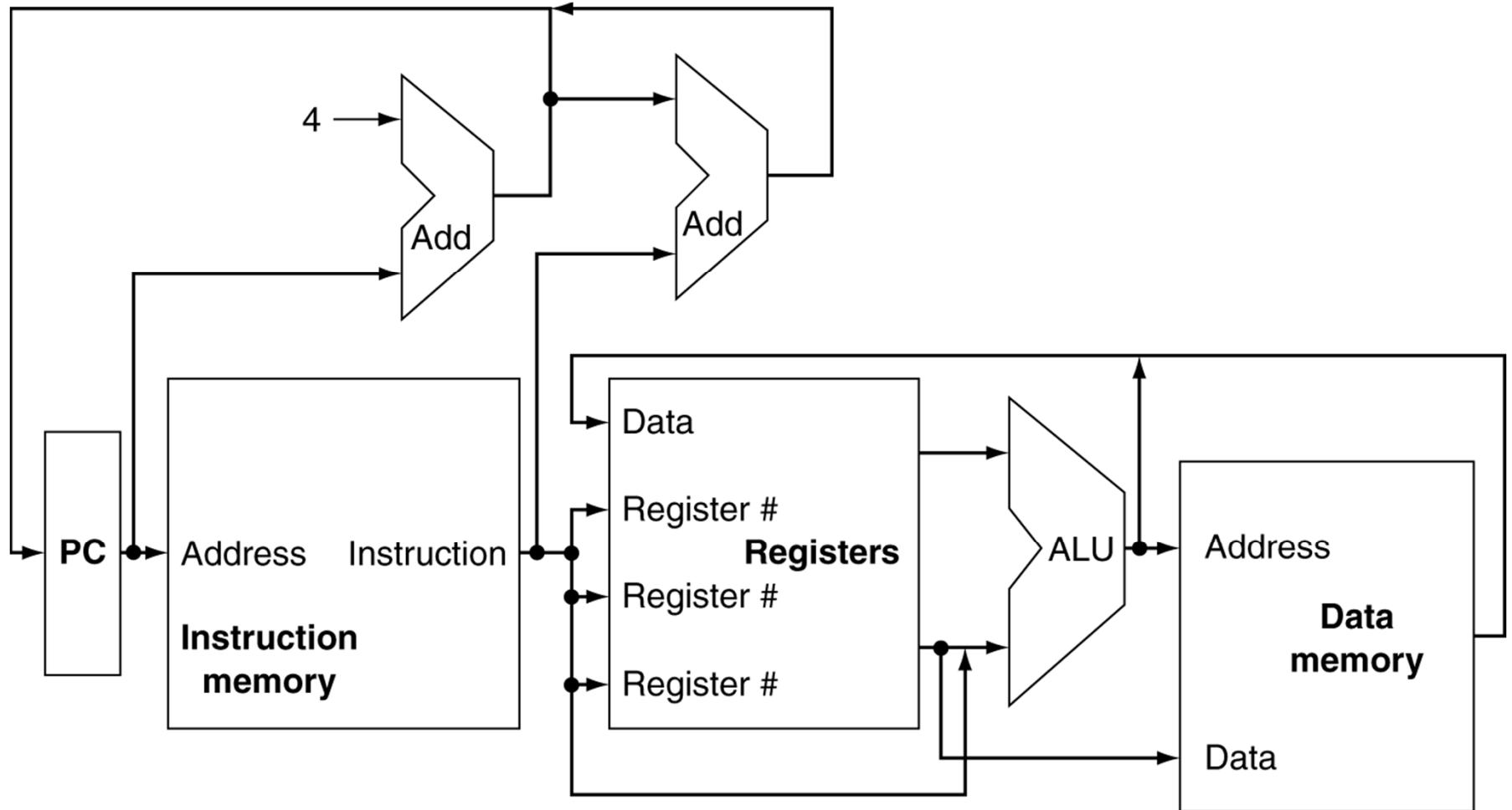
Introduction

- CPU performance factors
 - Instruction count
 - Determined by ISA and compiler
 - CPI and Cycle time
 - Determined by CPU hardware
- We will examine two MIPS implementations
 - A simplified version
 - A more realistic pipelined version
- Simple subset, shows most aspects
 - Memory reference: `lw`, `sw`
 - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
 - Control transfer: `beq`, `j`

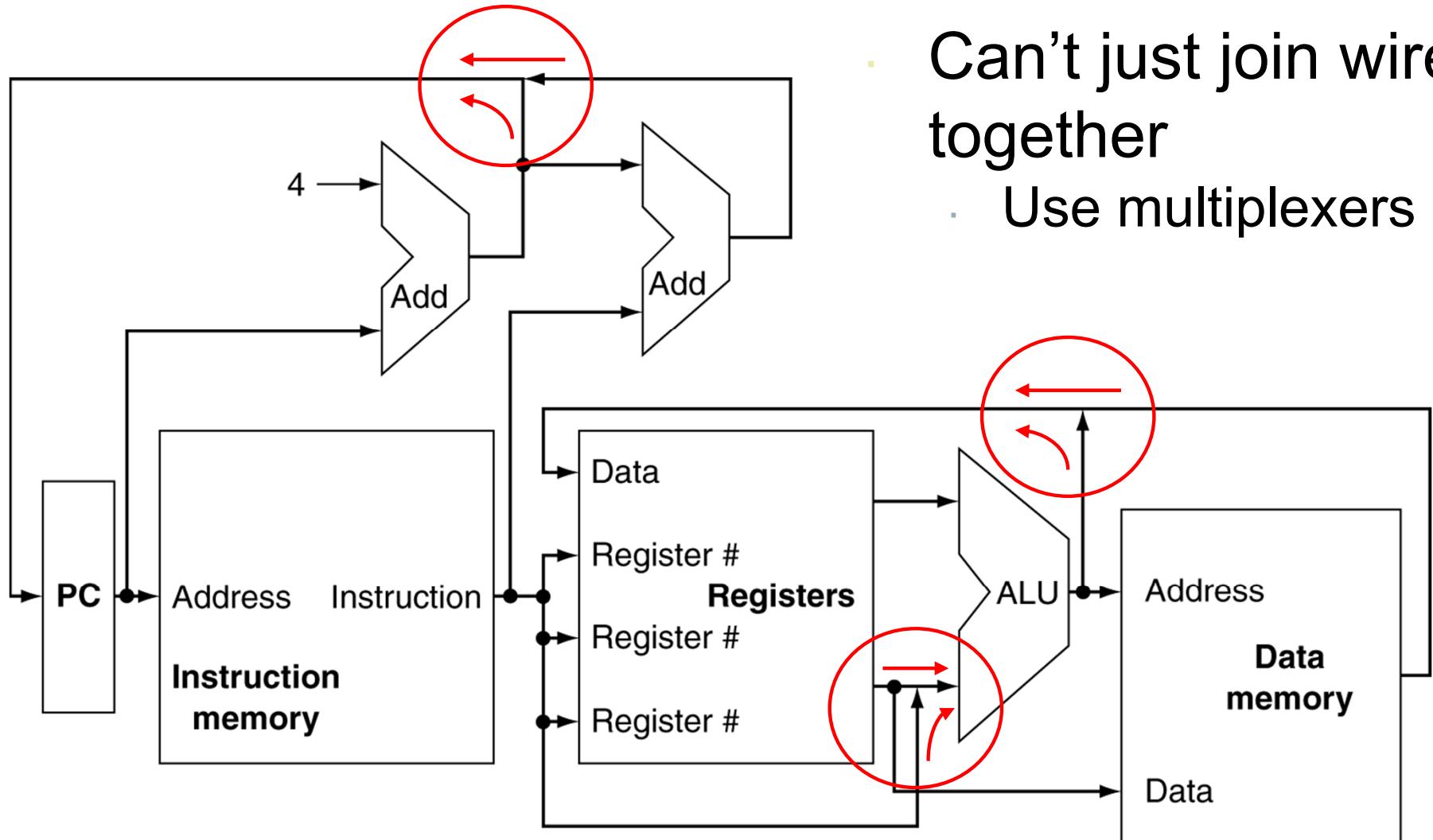
Instruction Execution

- PC \square instruction memory, fetch instruction
- Register numbers \square register file, read registers
- Depending on instruction class
 - Use ALU to calculate
 - Arithmetic result
 - Memory address for load/store
 - Branch target address
 - Access data memory for load/store
 - PC \square target address or PC + 4

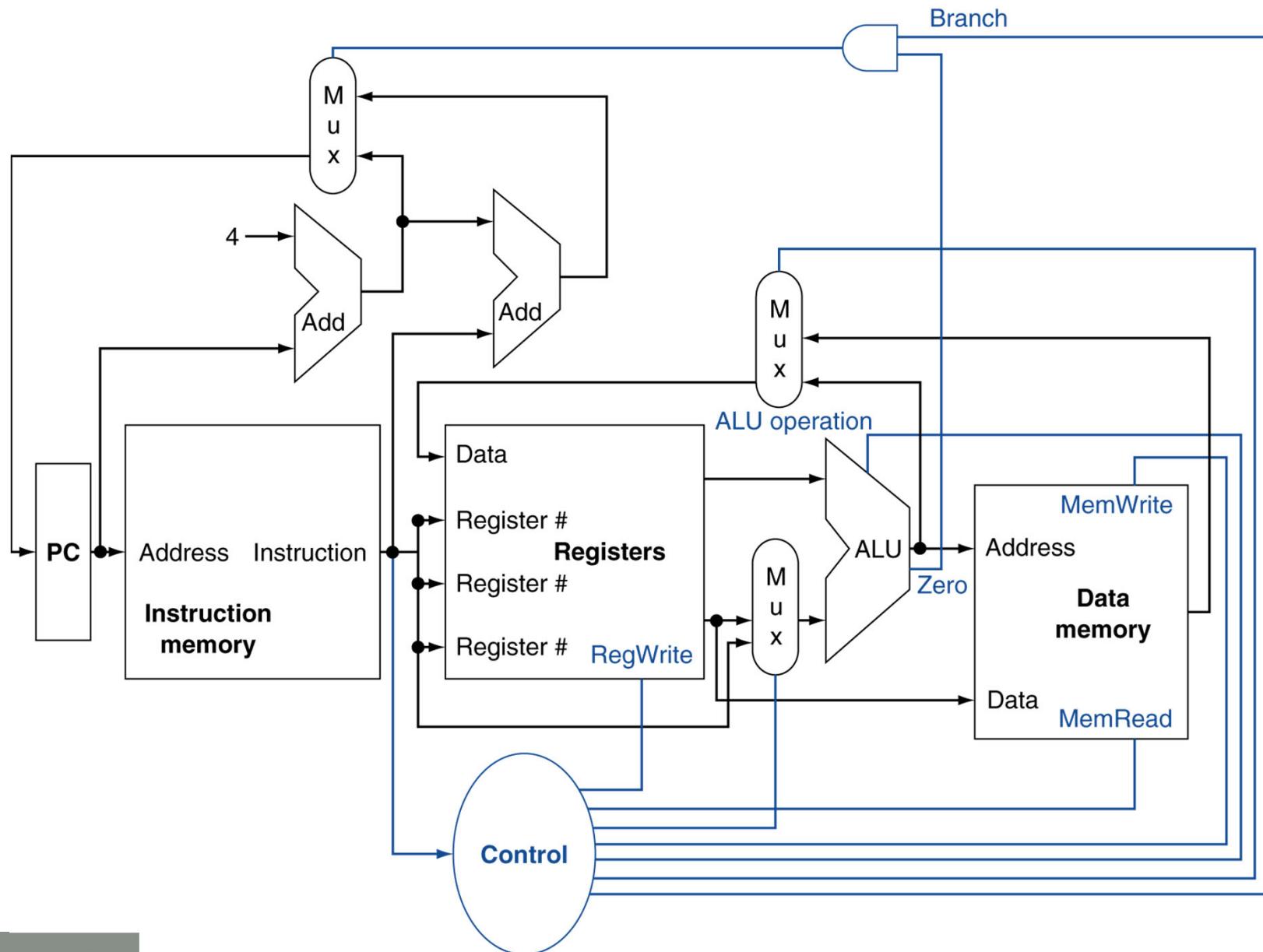
CPU Overview



Multiplexers



Control



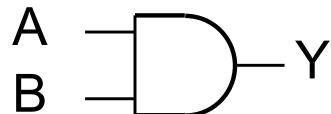
Logic Design Basics

- Information encoded in binary
 - Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses
- Combinational element
 - Operate on data
 - Output is a function of input
- State (sequential) elements
 - Store information

Combinational Elements

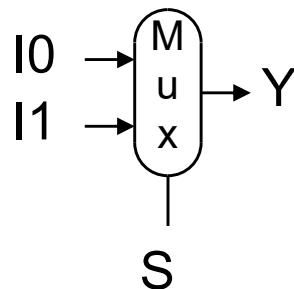
- AND-gate

- $Y = A \& B$



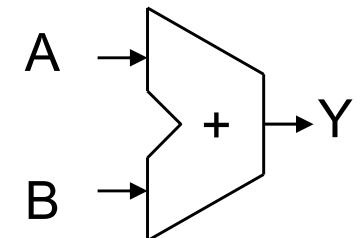
- Multiplexer

- $Y = S ? I_1 : I_0$



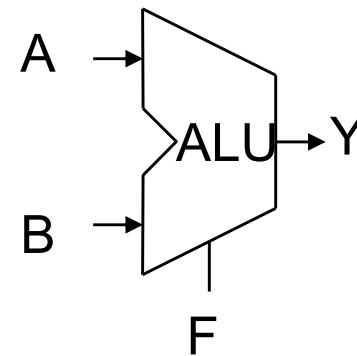
- Adder

- $Y = A + B$



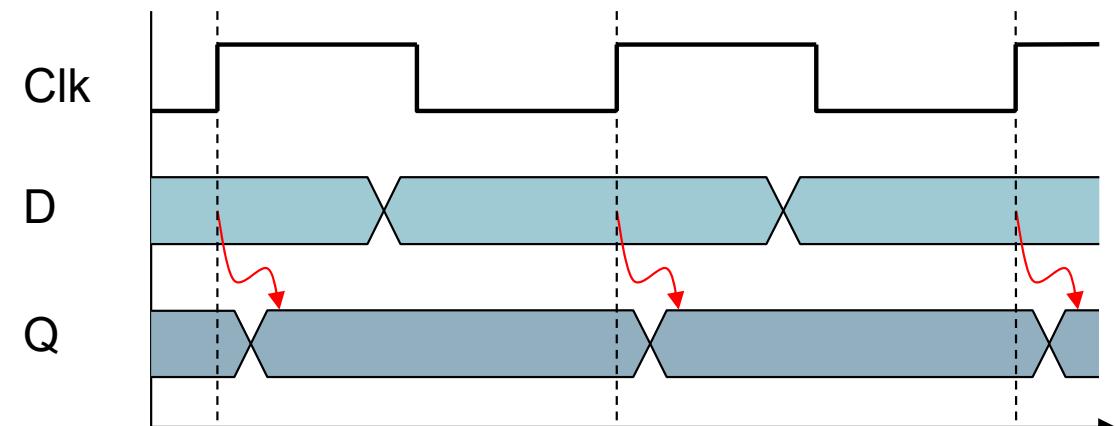
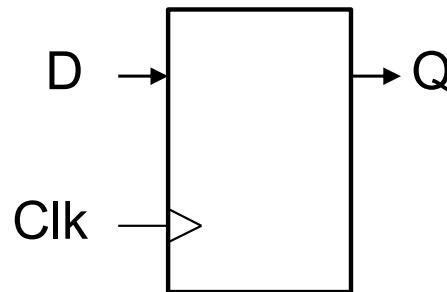
- Arithmetic/Logic Unit

- $Y = F(A, B)$



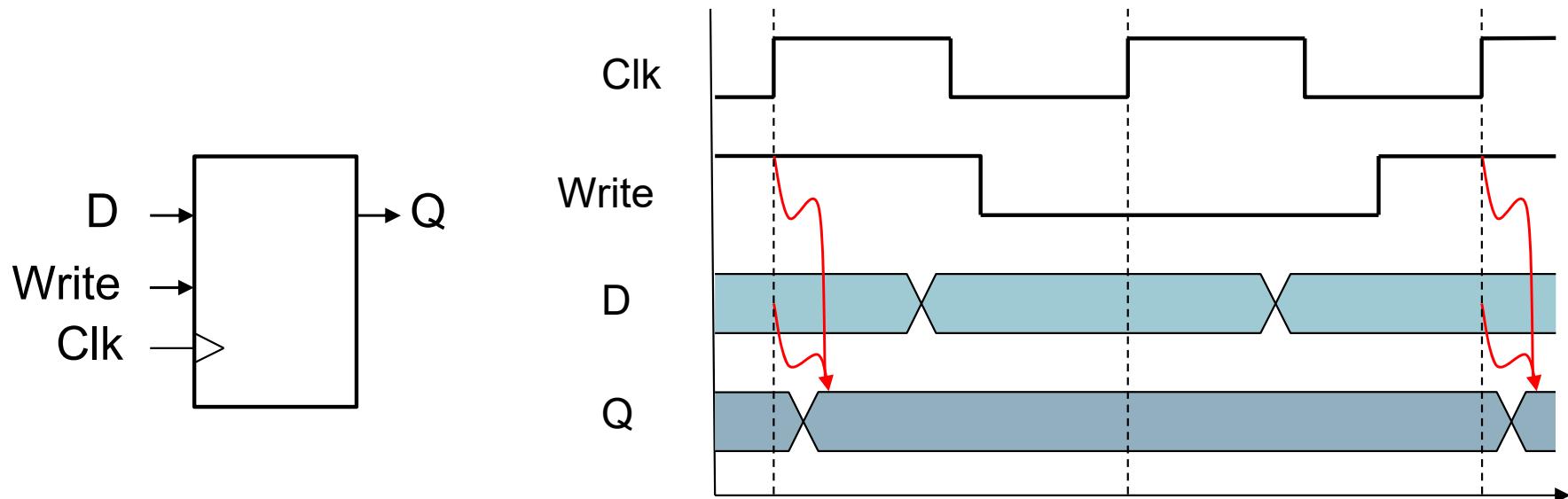
Sequential Elements

- Register: stores data in a circuit
 - Uses a clock signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1



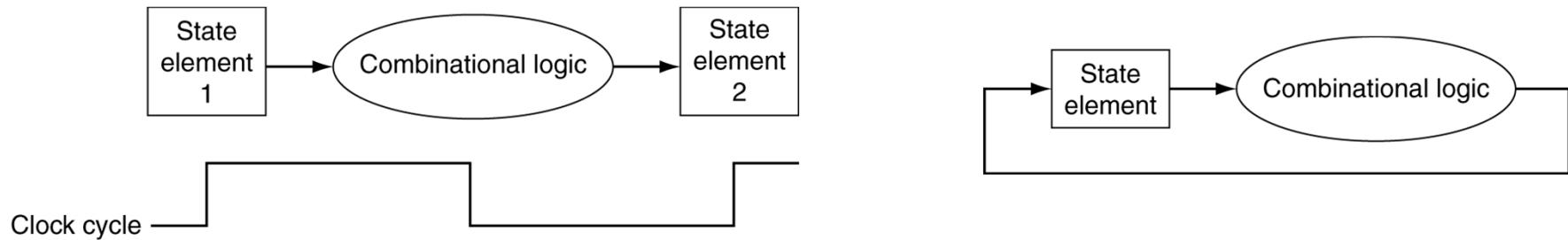
Sequential Elements

- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later



Clocking Methodology

- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements, output to state element
 - Longest delay determines clock period

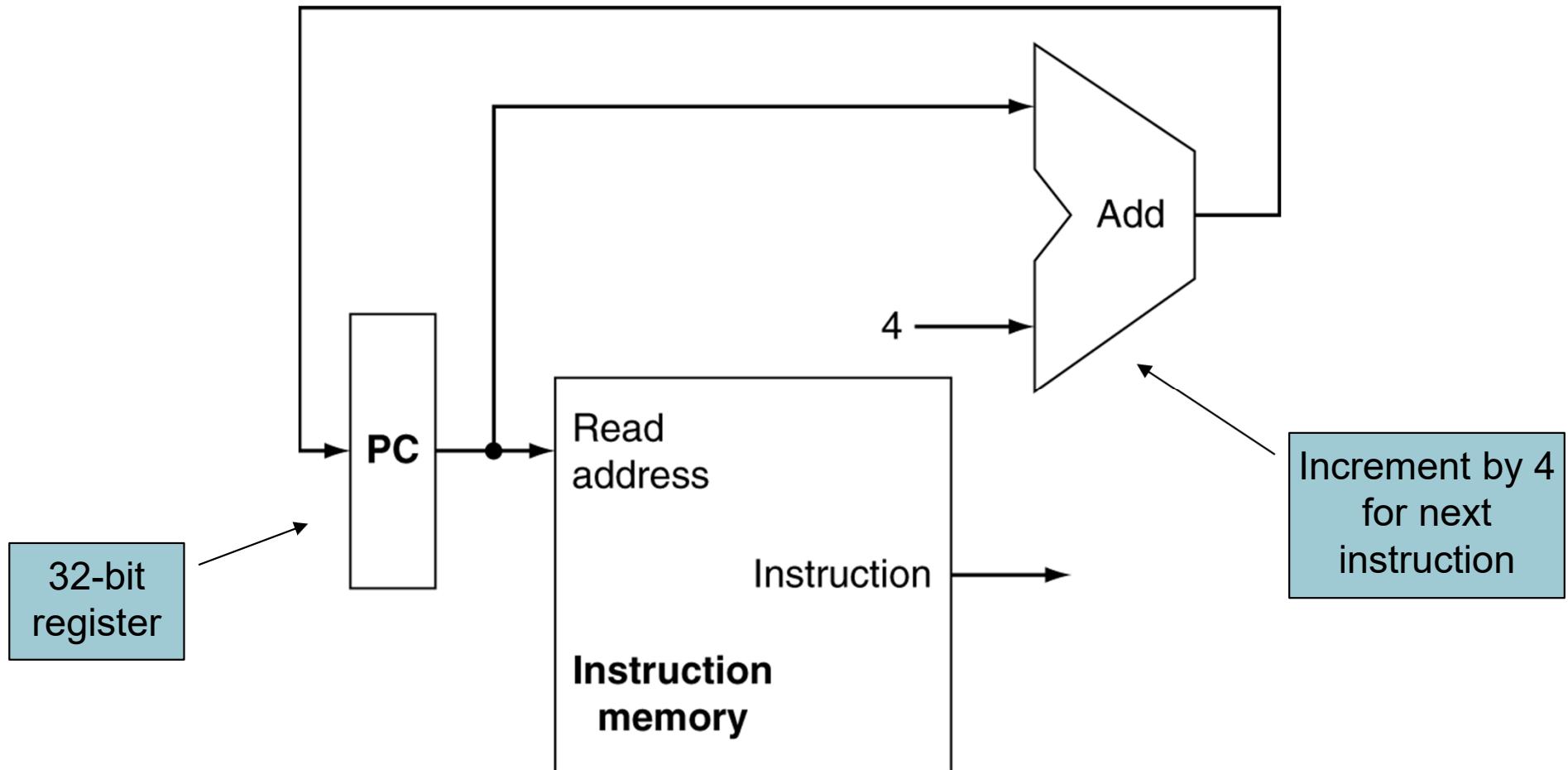


Building a Datapath

- Datapath
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
 - Refining the overview design

Instruction Fetch

Common
for all

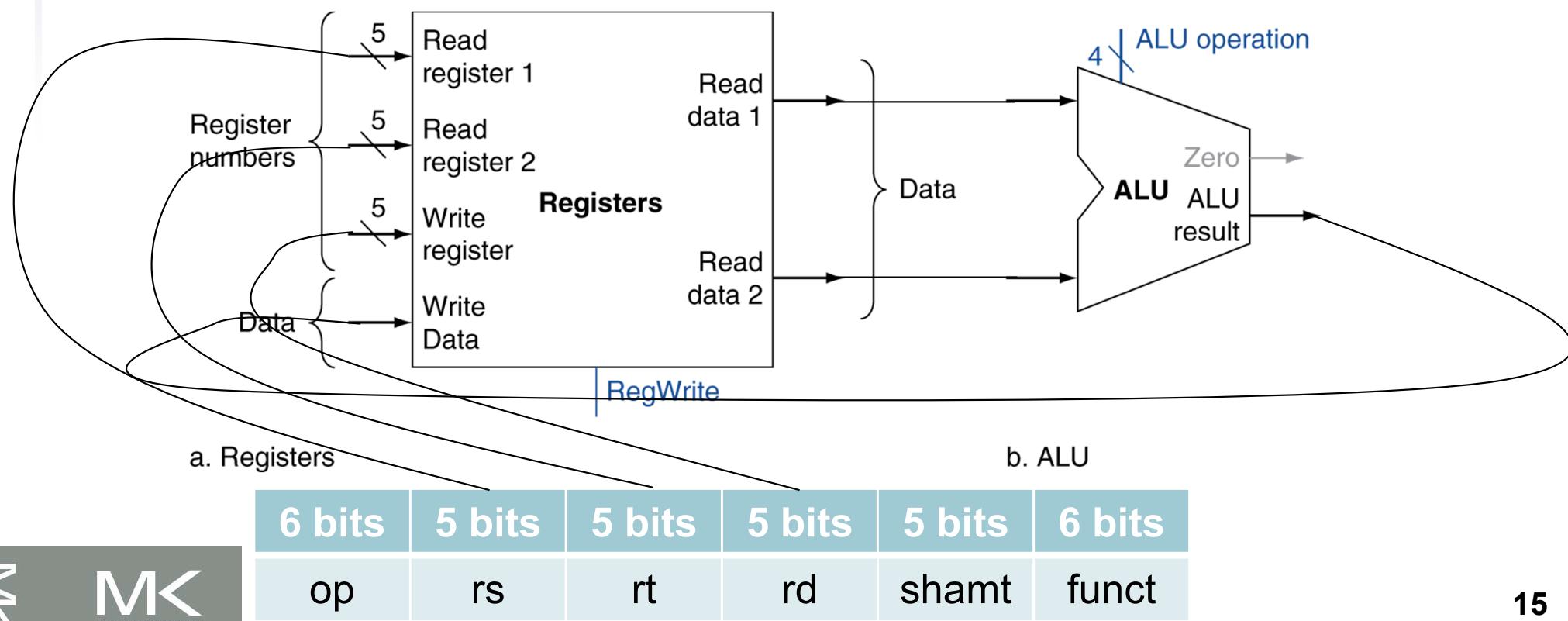


Recall: Instruction Formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

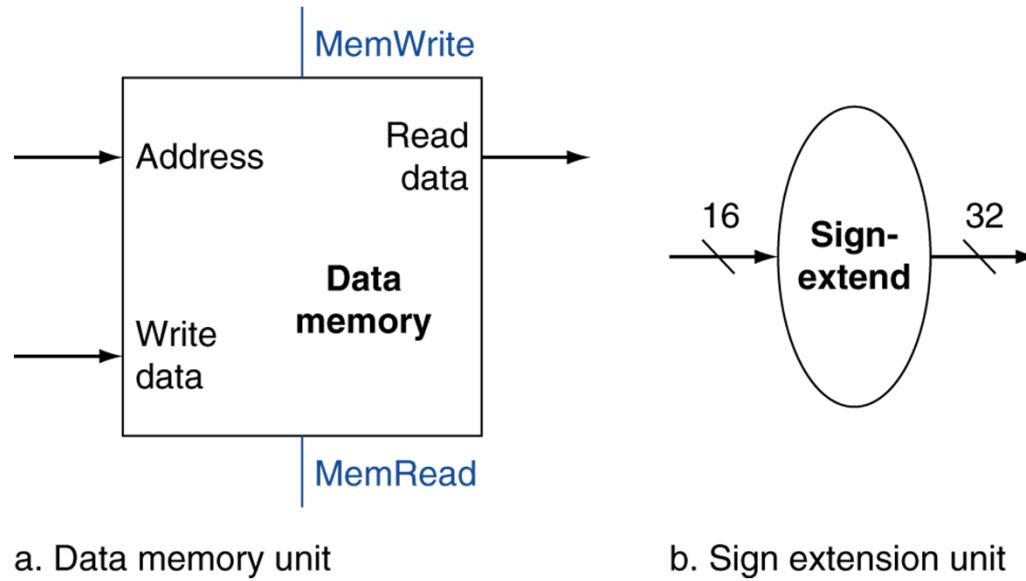
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

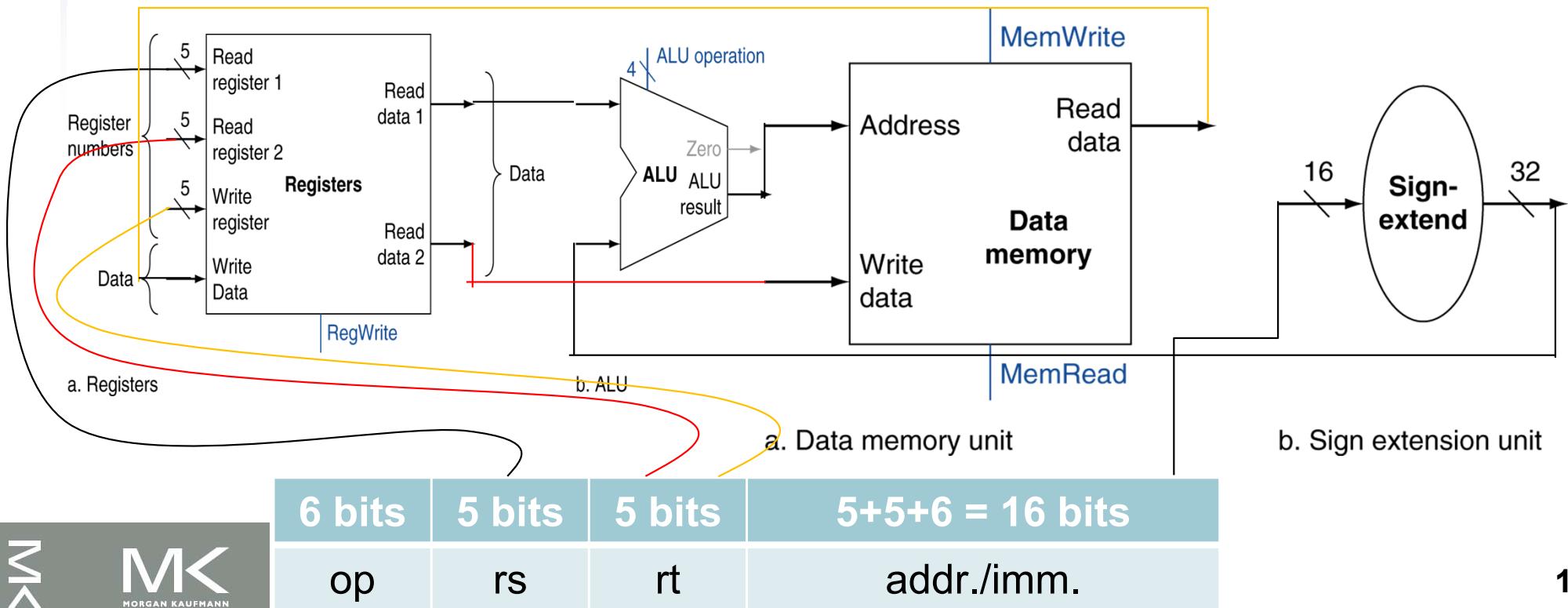


6 bits	5 bits	5 bits	5+5+6 = 16 bits
op	rs	rt	addr./imm.

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

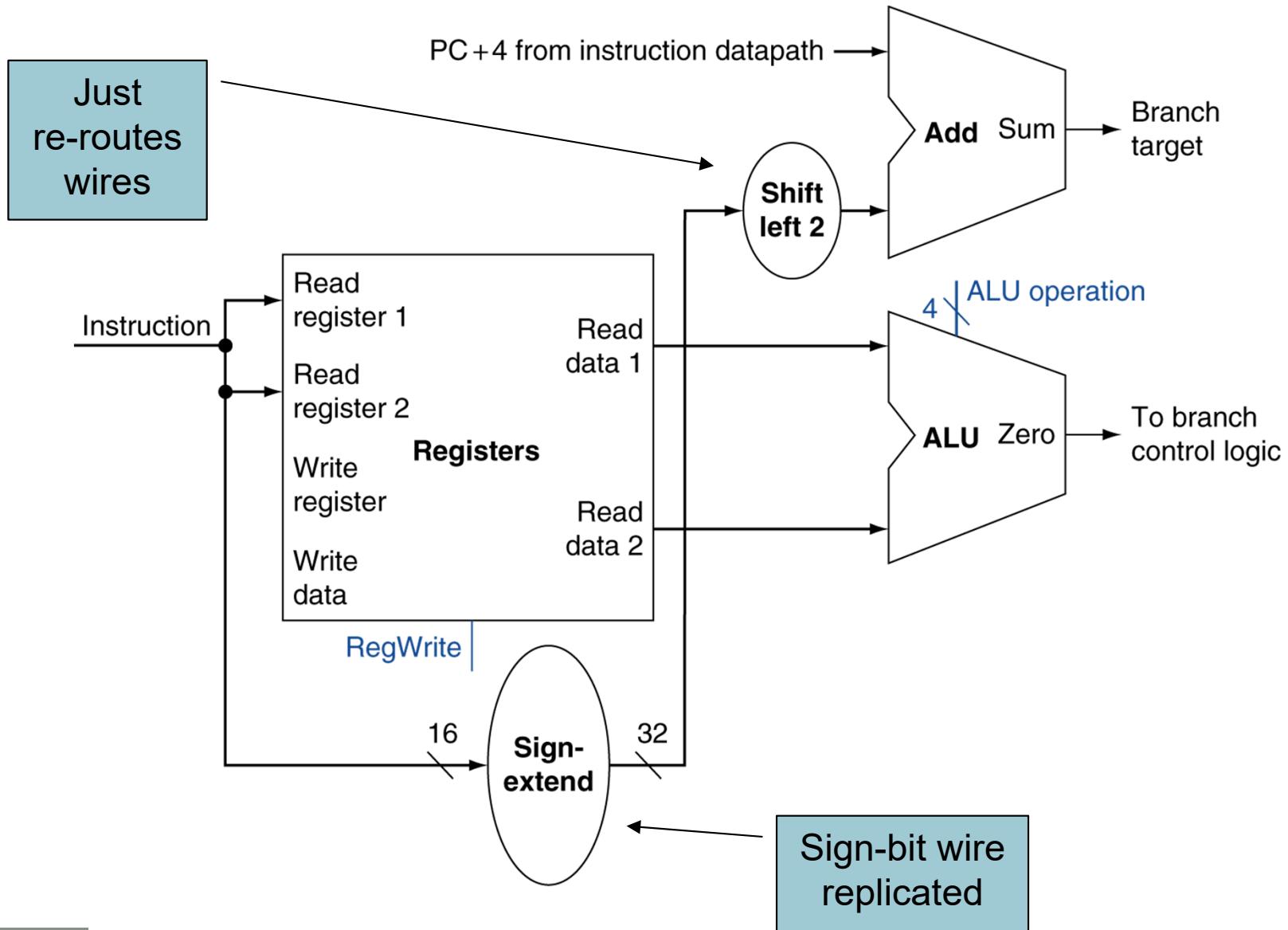
— Store
— Load



Branch Instructions

- Read register operands
- Compare operands
 - Use ALU, subtract and check Zero output
- Calculate target address
 - Sign-extend displacement
 - Shift left 2 places (word displacement)
 - Add to PC + 4
 - Already calculated by instruction fetch

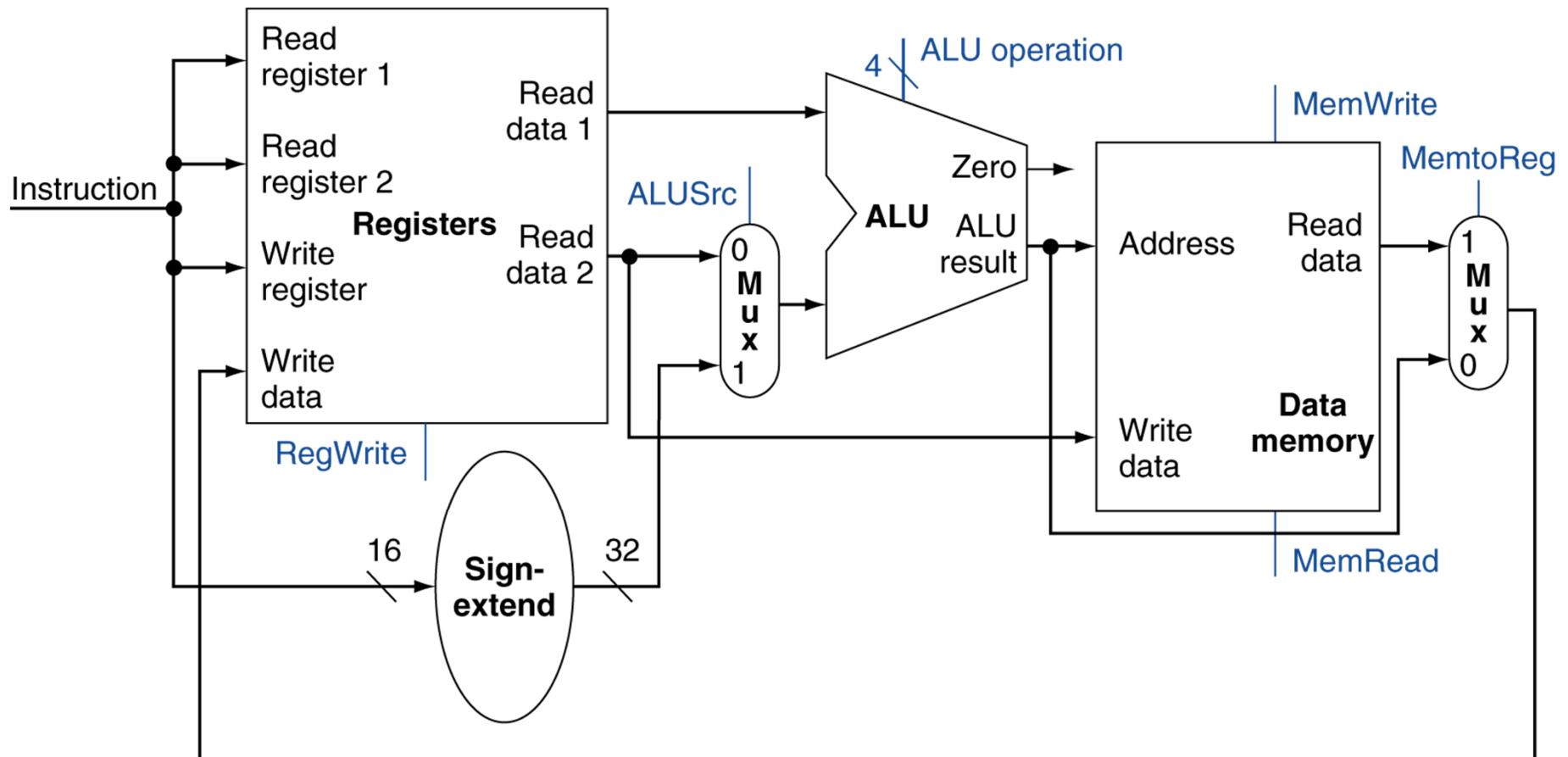
Branch Instructions



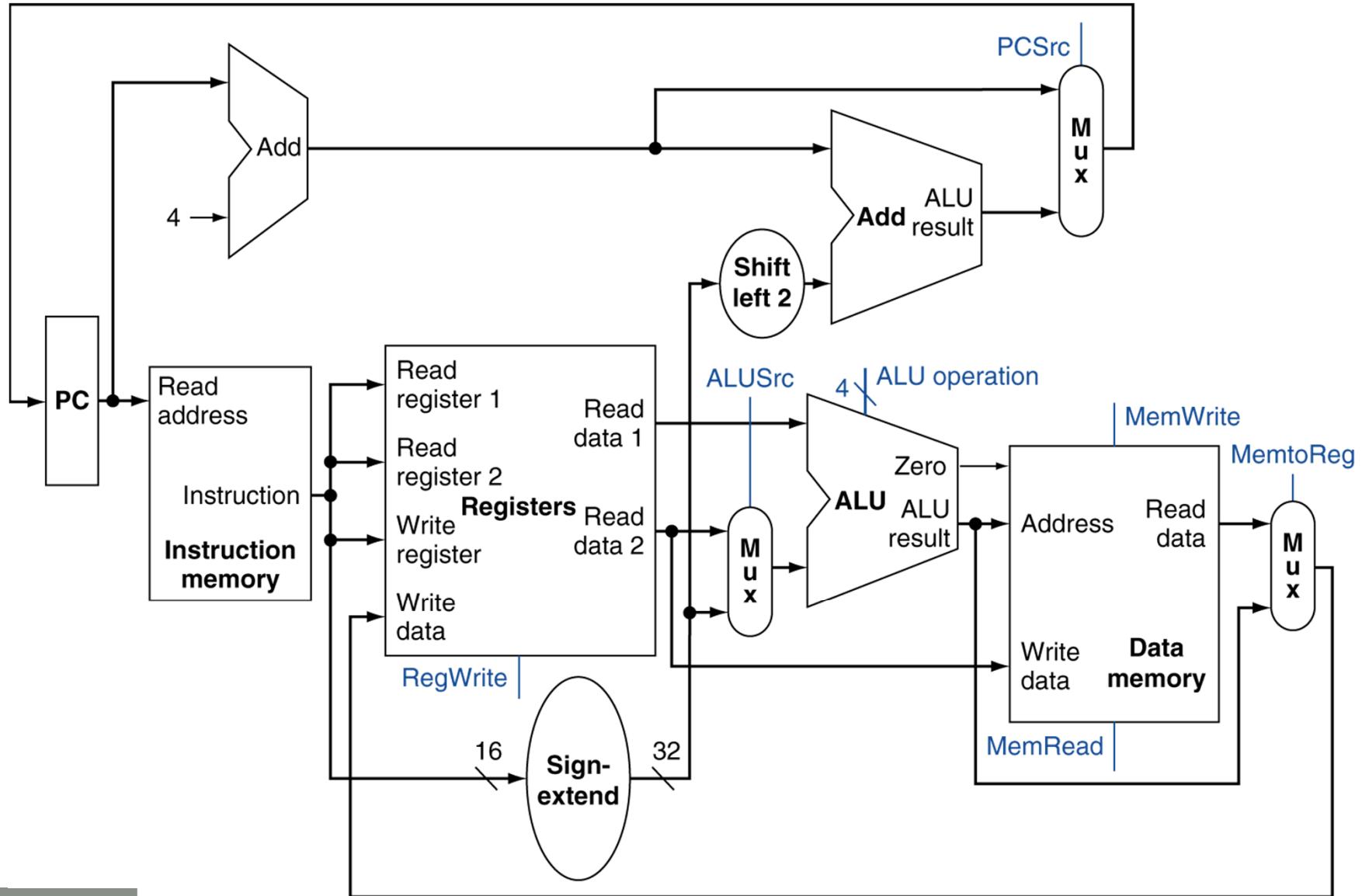
Composing the Elements

- First-cut data path does an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
 - Load/Store: F = add
 - Branch: F = subtract
 - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

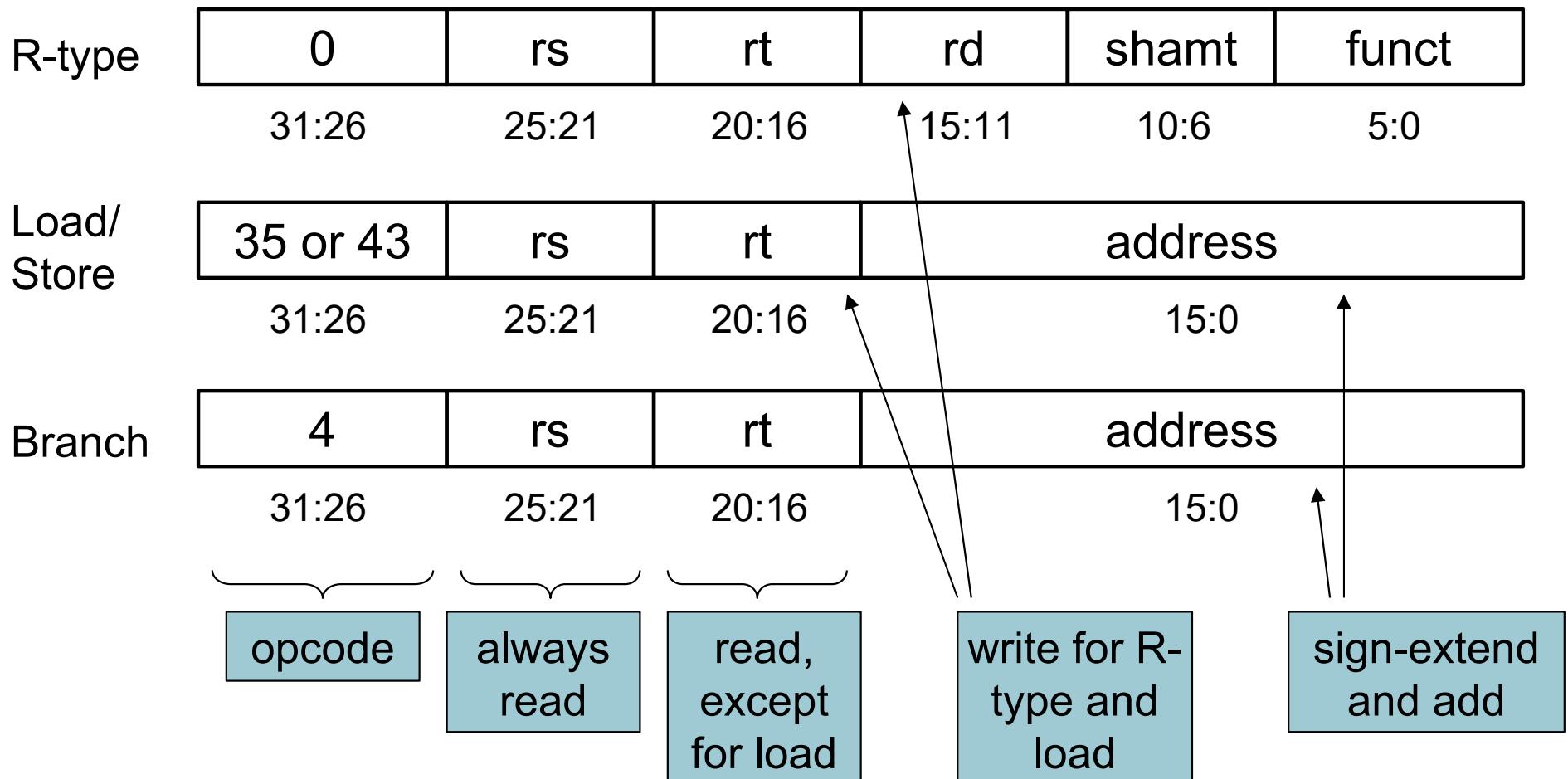
ALU Control

- Assume 2-bit ALUOp derived from opcode
 - Combinational logic derives ALU control

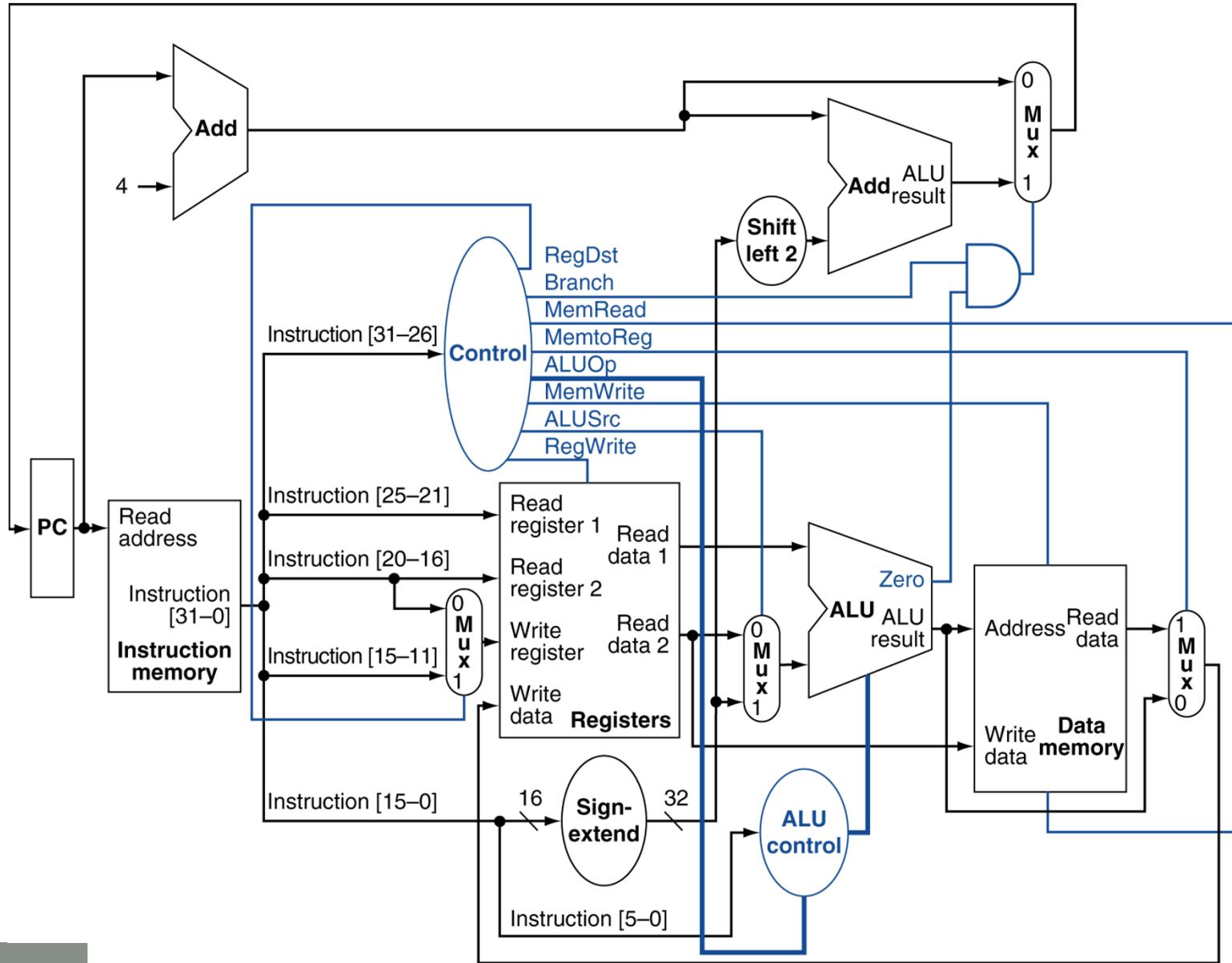
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction



Datapath With Control

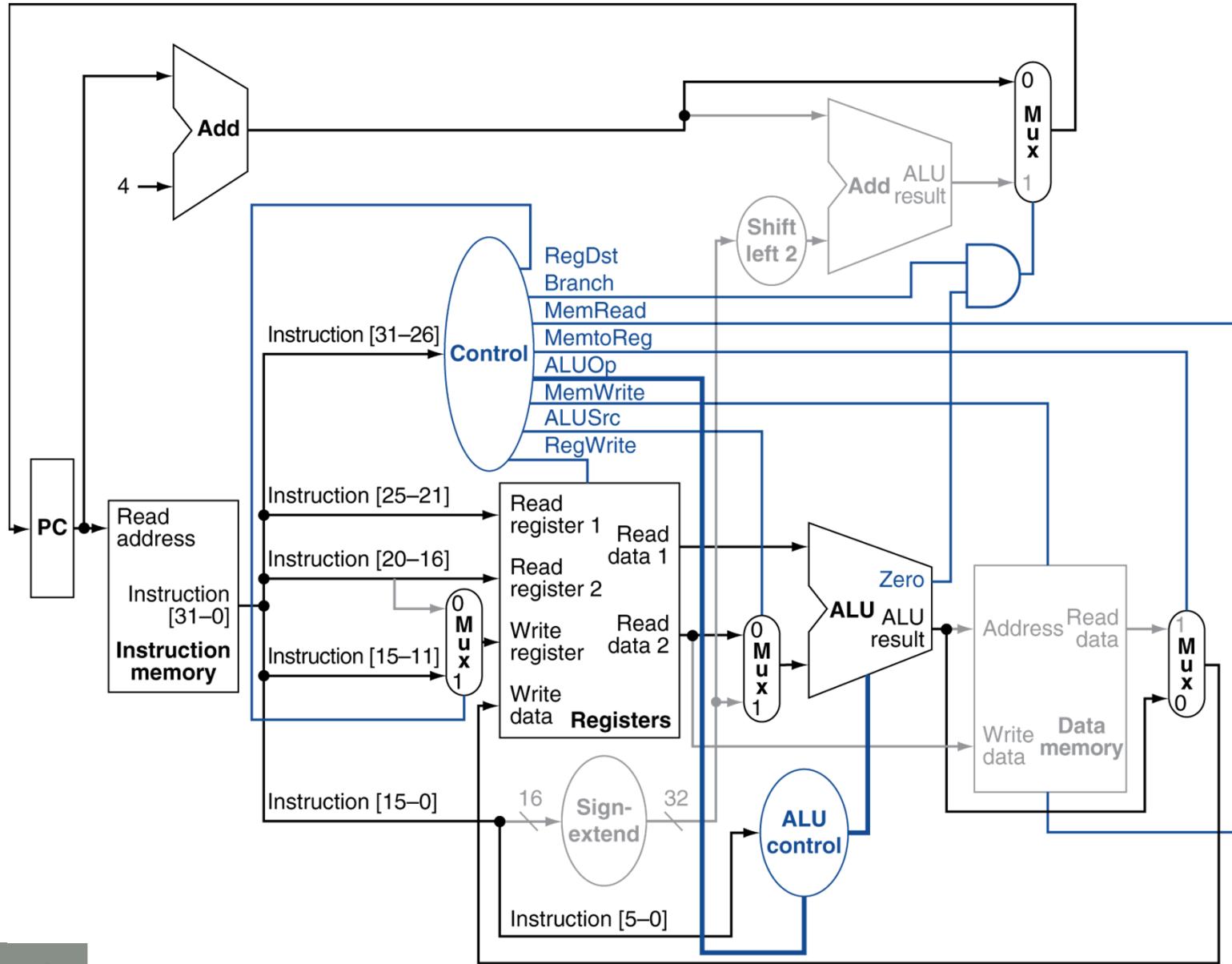


All Control Signals

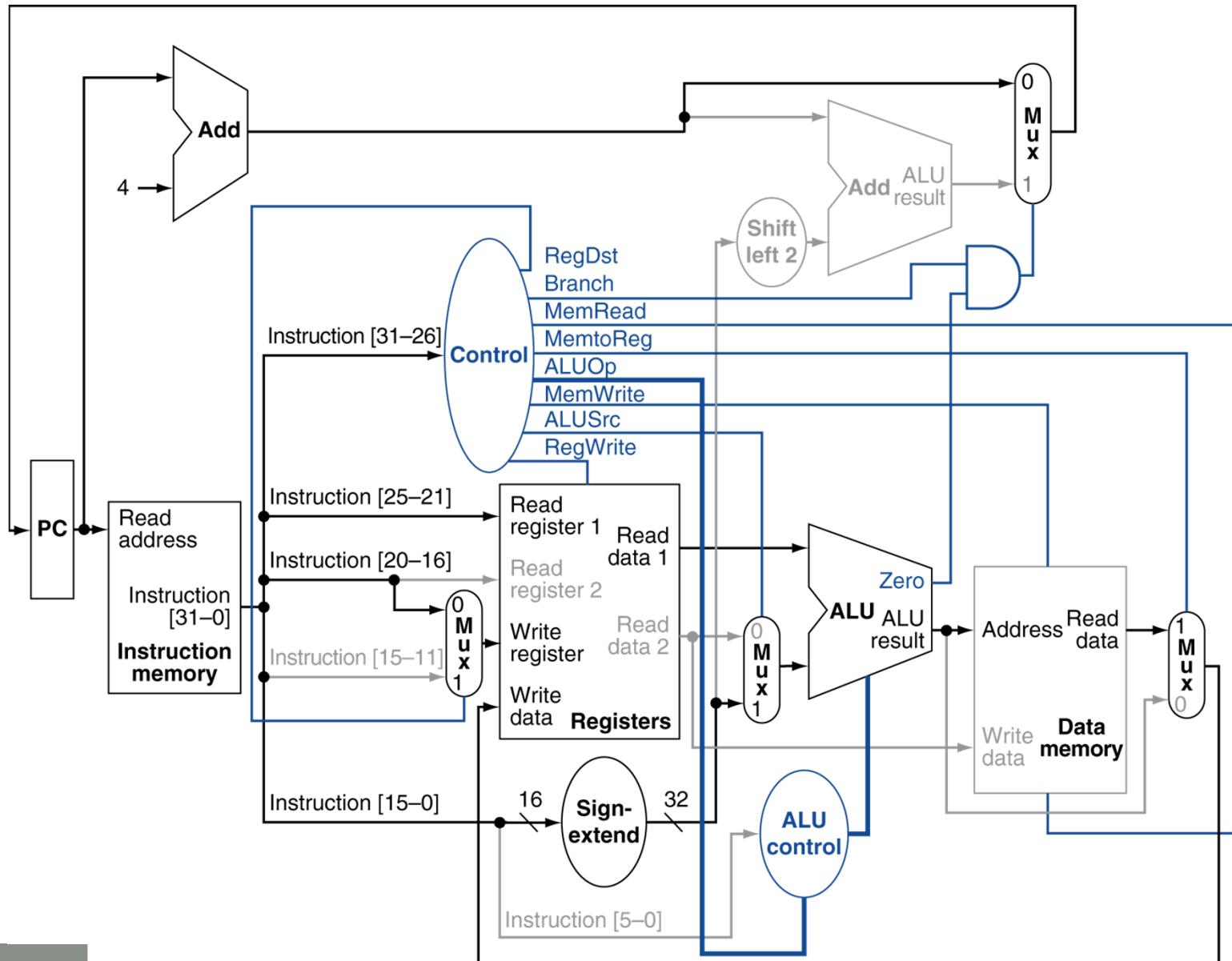
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See  Appendix B for further discussion of this problem.)

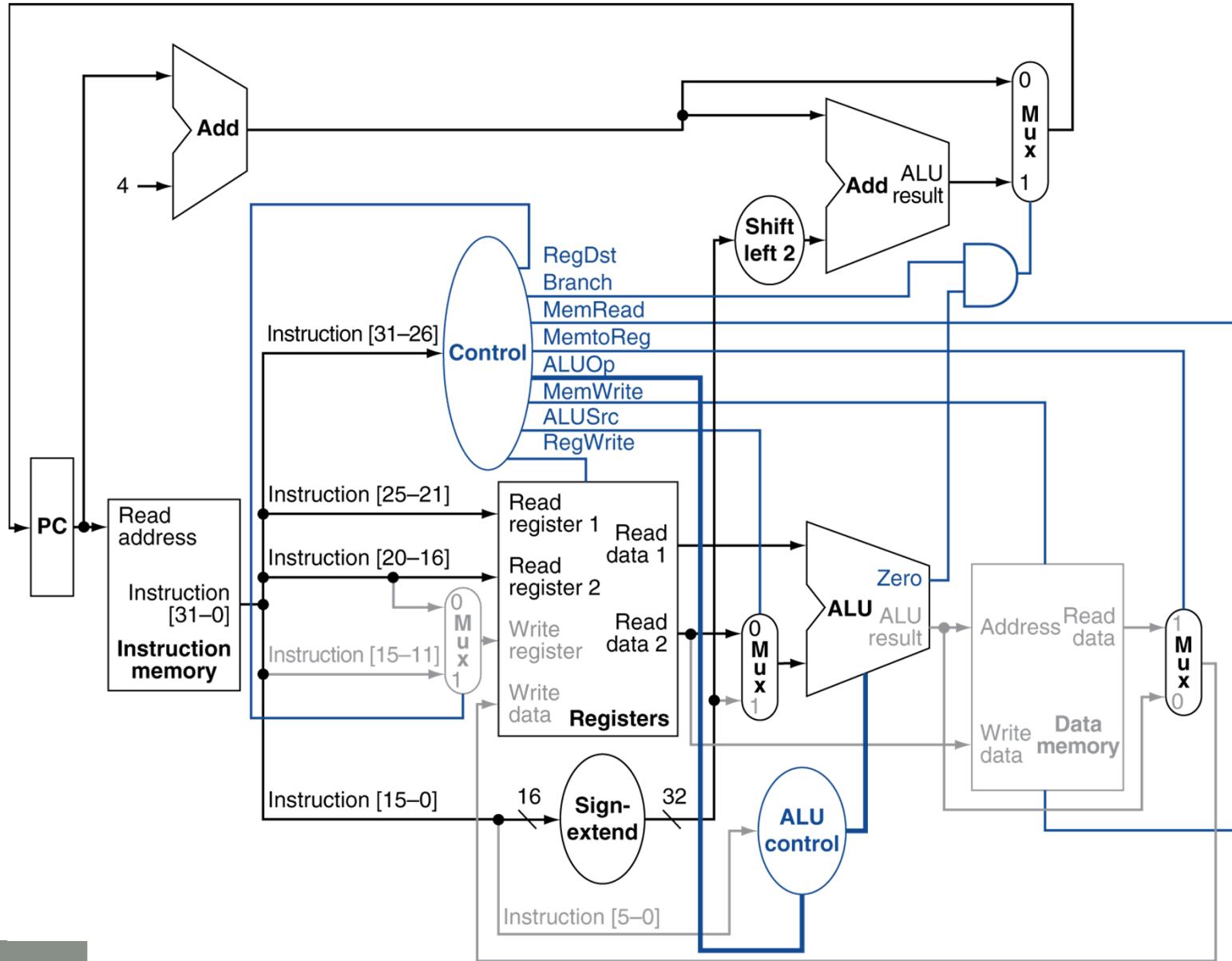
R-Type Instruction



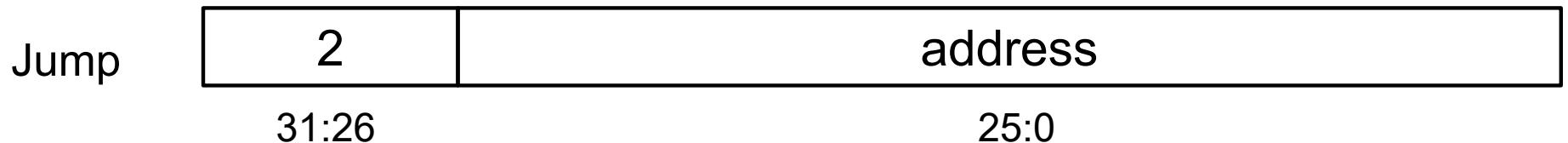
Load Instruction



Branch-on-Equal Instruction

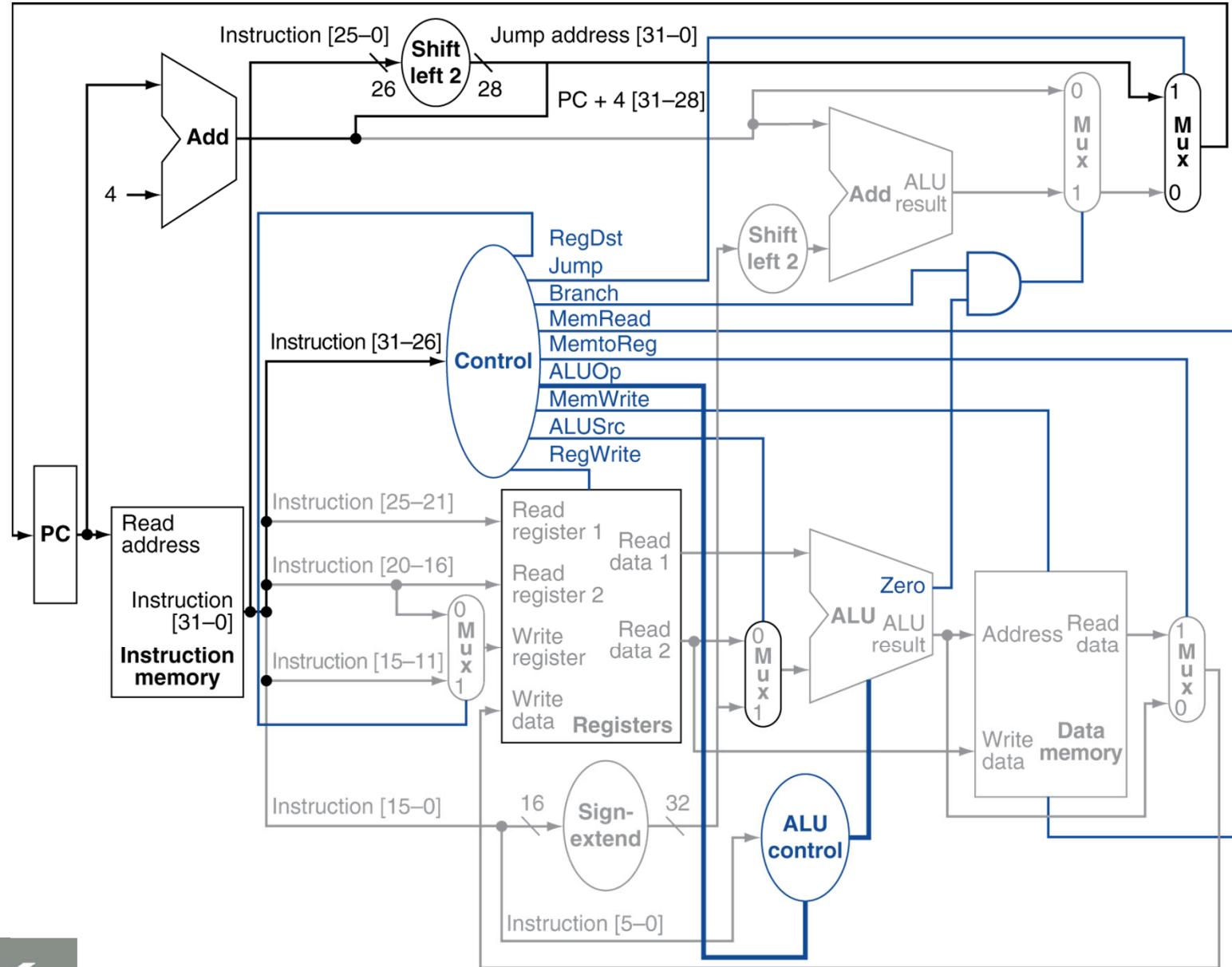


Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
 - Top 4 bits of old PC
 - 26-bit jump address
 - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added

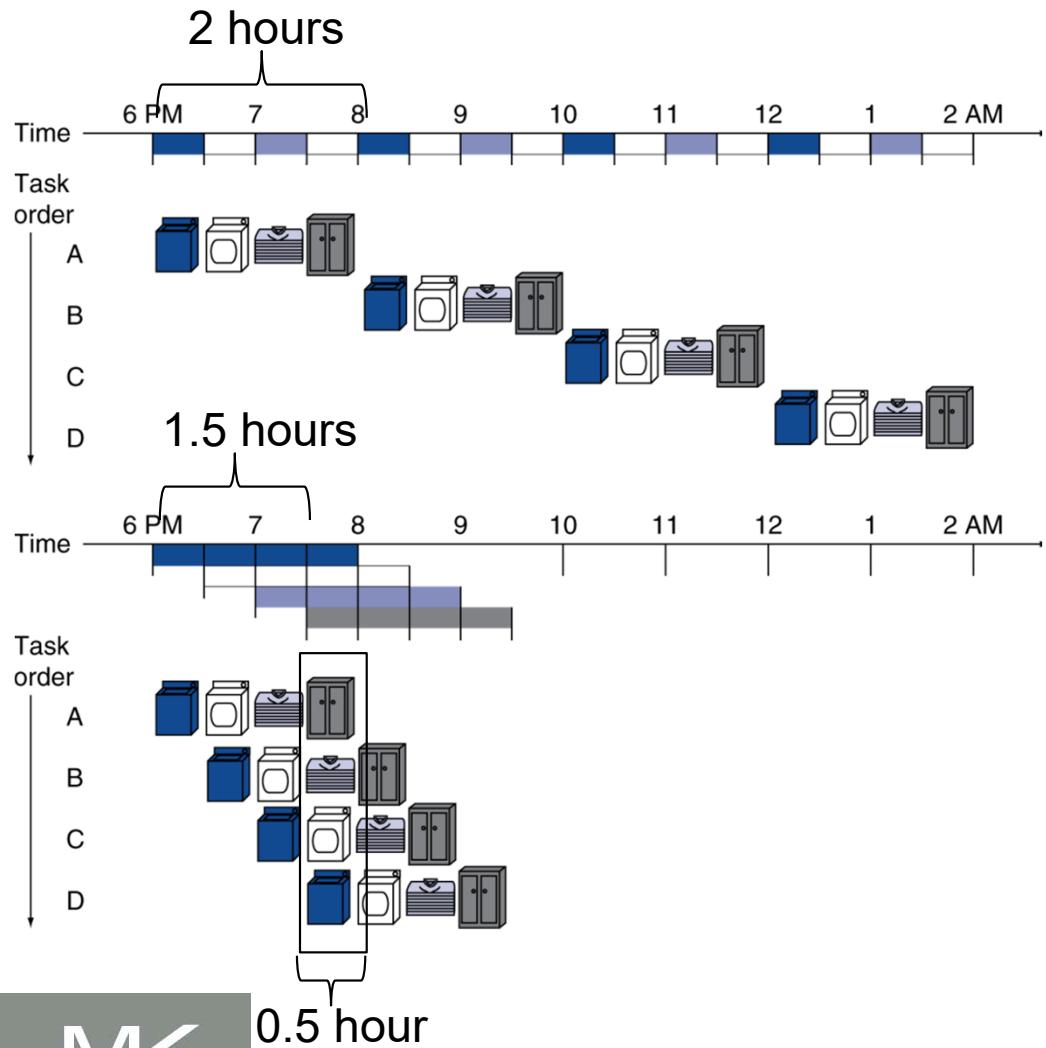


Performance Issues

- Longest delay determines clock period
 - Critical path: load instruction
 - Instruction memory □ register file □ ALU □ data memory □ register file
- Not feasible to vary period for different instructions
- Violates design principle
 - Making the common case fast
- We will improve performance by pipelining

Pipelining Analogy

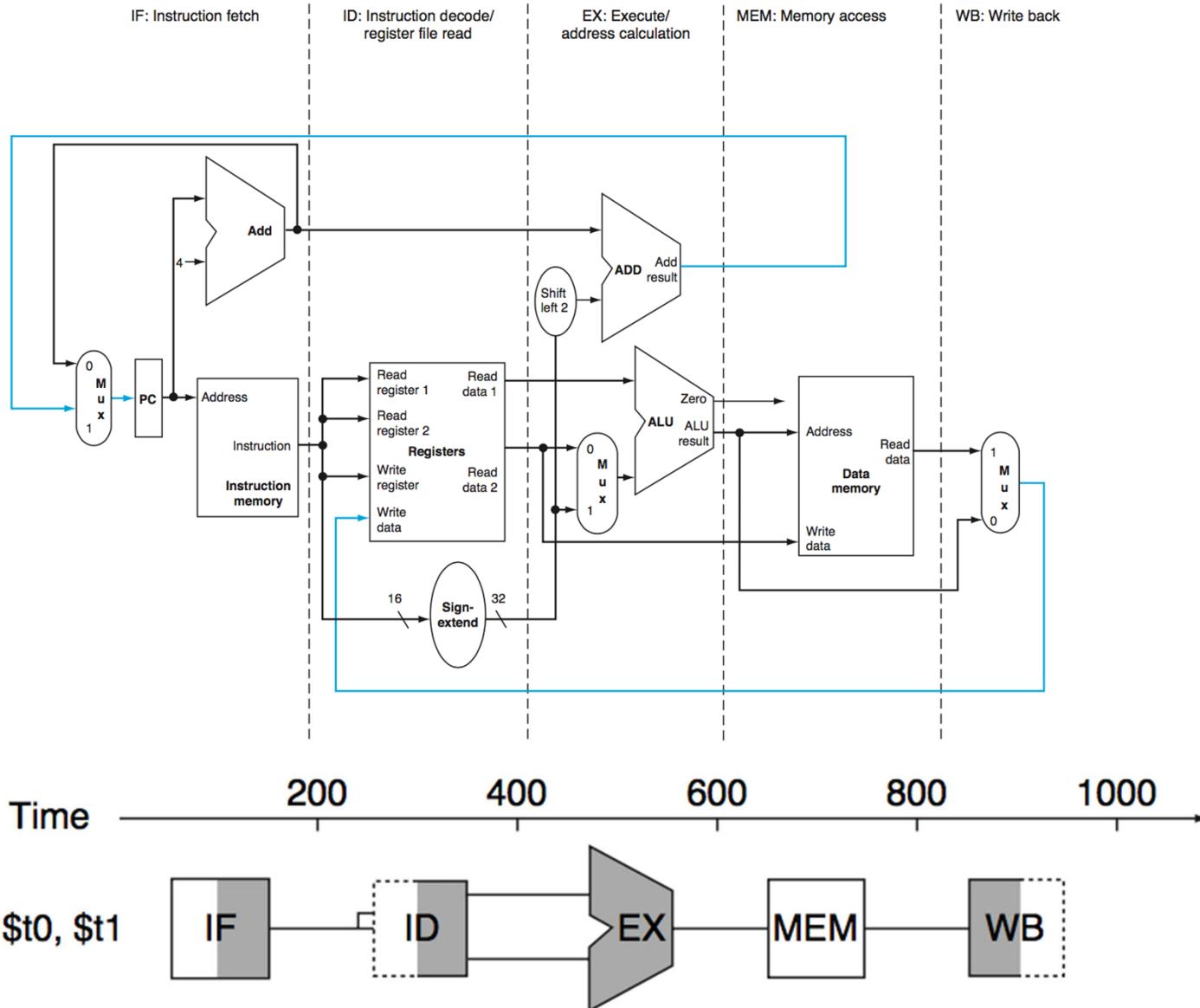
- Pipelined laundry: overlapping execution
 - Parallelism improves performance



MIPS Pipeline

- Five stages, one step per stage
 1. IF: Instruction fetch from memory
 2. ID: Instruction decode & register read
 3. EX: Execute operation or calculate address
 4. MEM: Access memory operand
 5. WB: Write result back to register

MIPS Stages

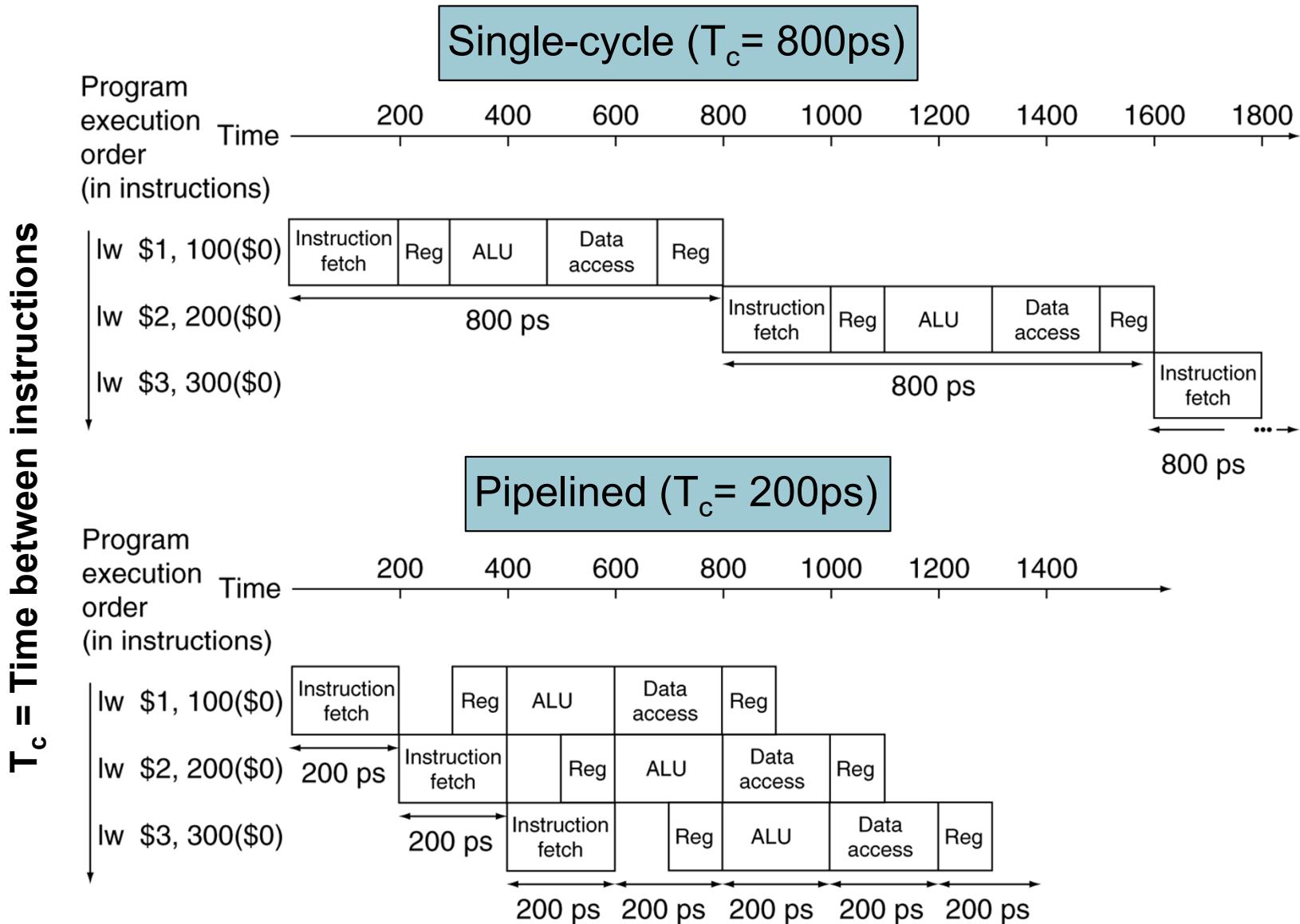


Pipeline Performance

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
 - i.e., all take the same time
 - Time between instructions_{pipelined}
= Time between instructions_{nonpipelined}

$$\frac{\text{Number of stages}}{}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
 - Latency (time for each instruction) does not decrease

Hazards

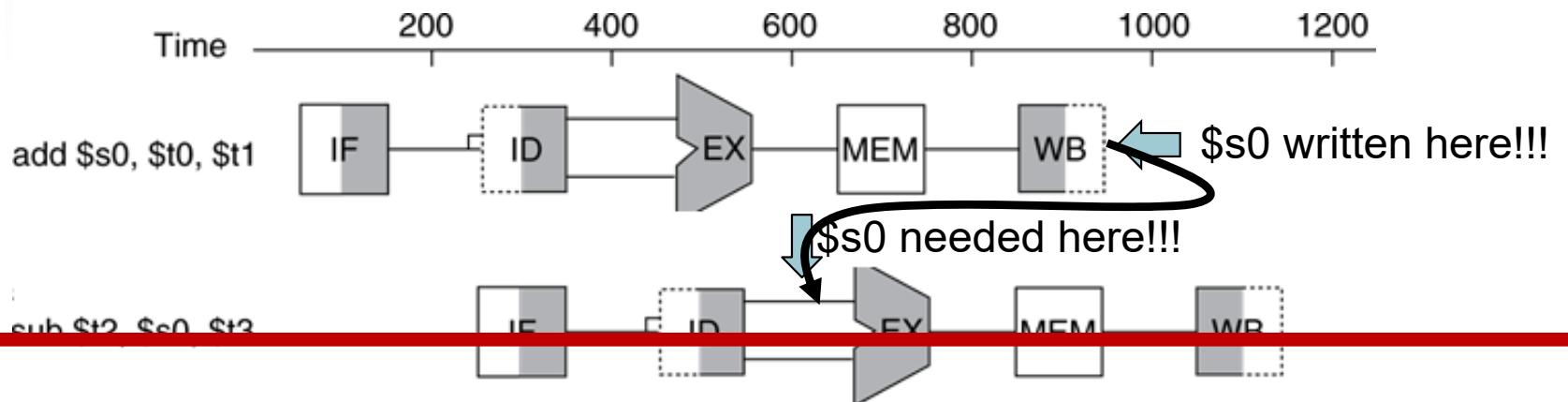
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - A required resource is busy
- Data hazard
 - Need to wait for previous instruction to complete its data read/write
- Control hazard
 - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
 - Load/store requires data access
 - Instruction fetch would have to *stall* for that cycle
 - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches

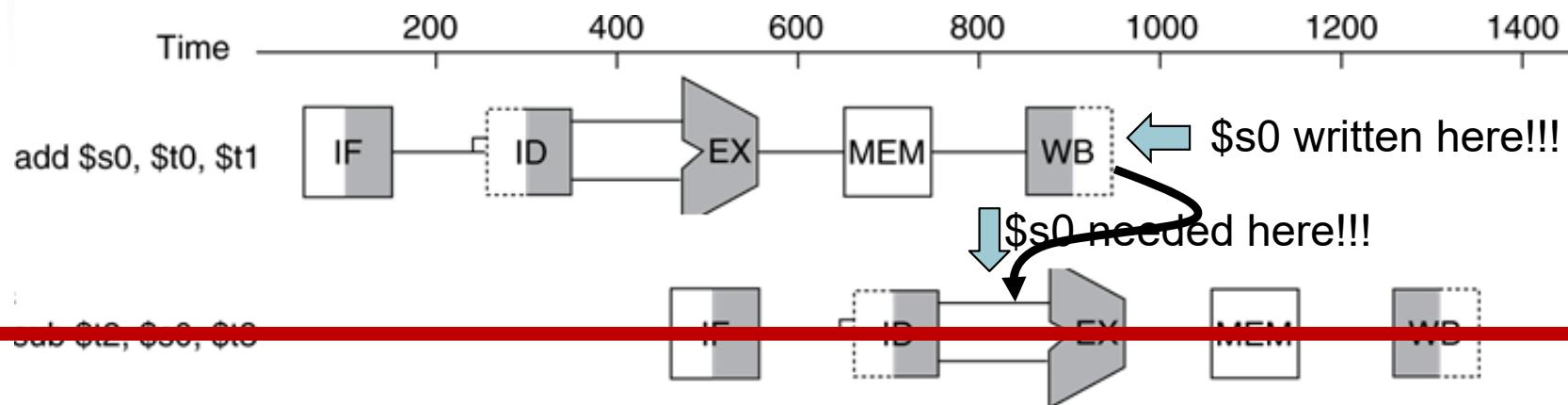
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



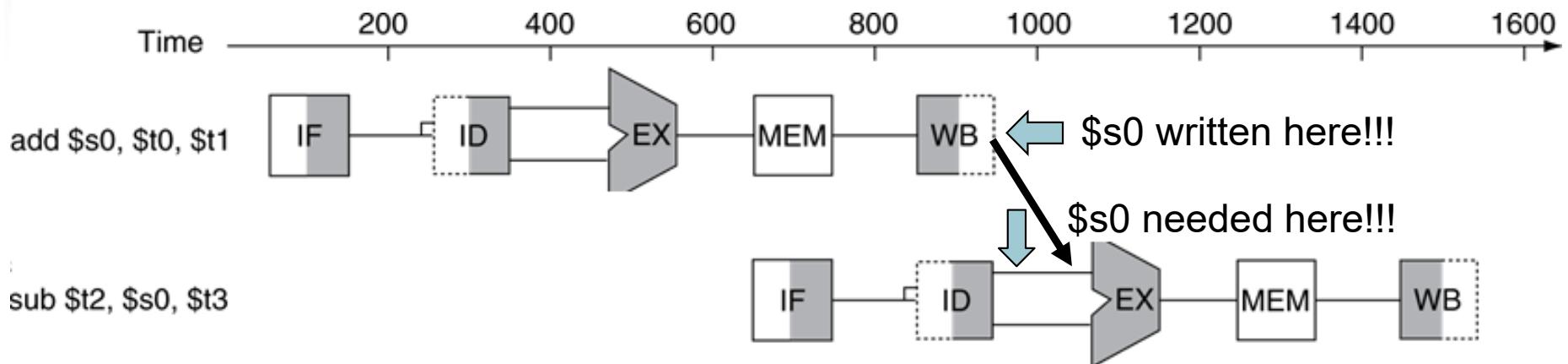
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



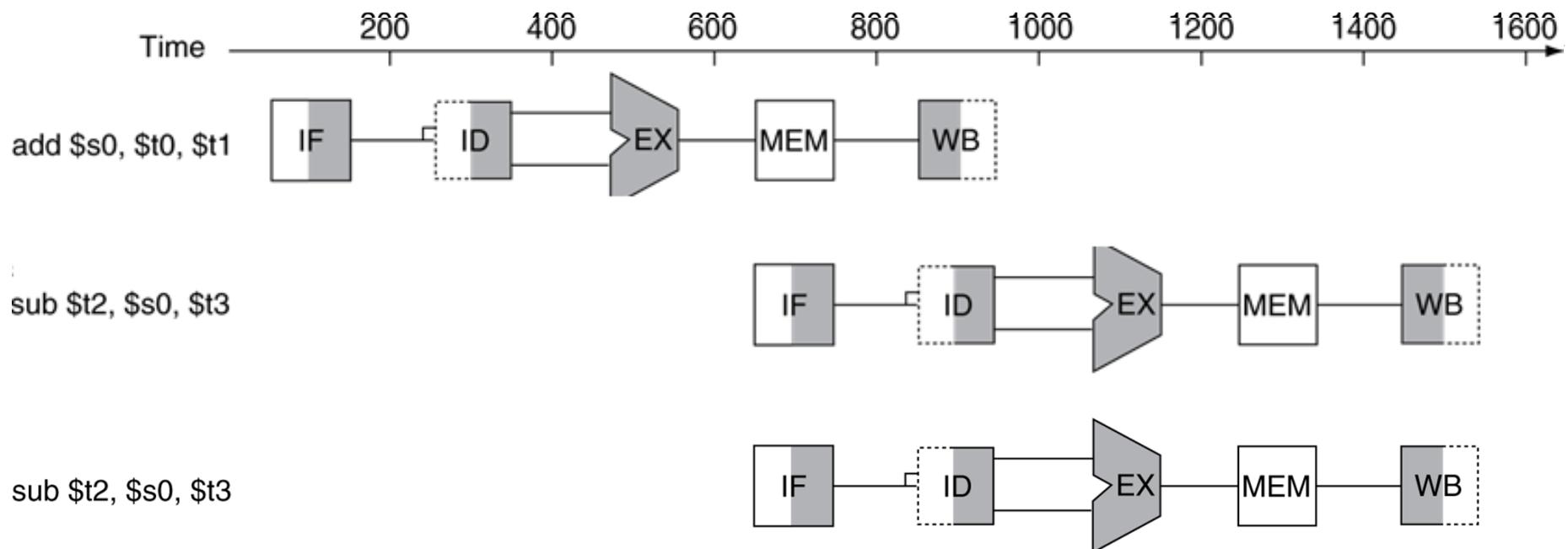
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



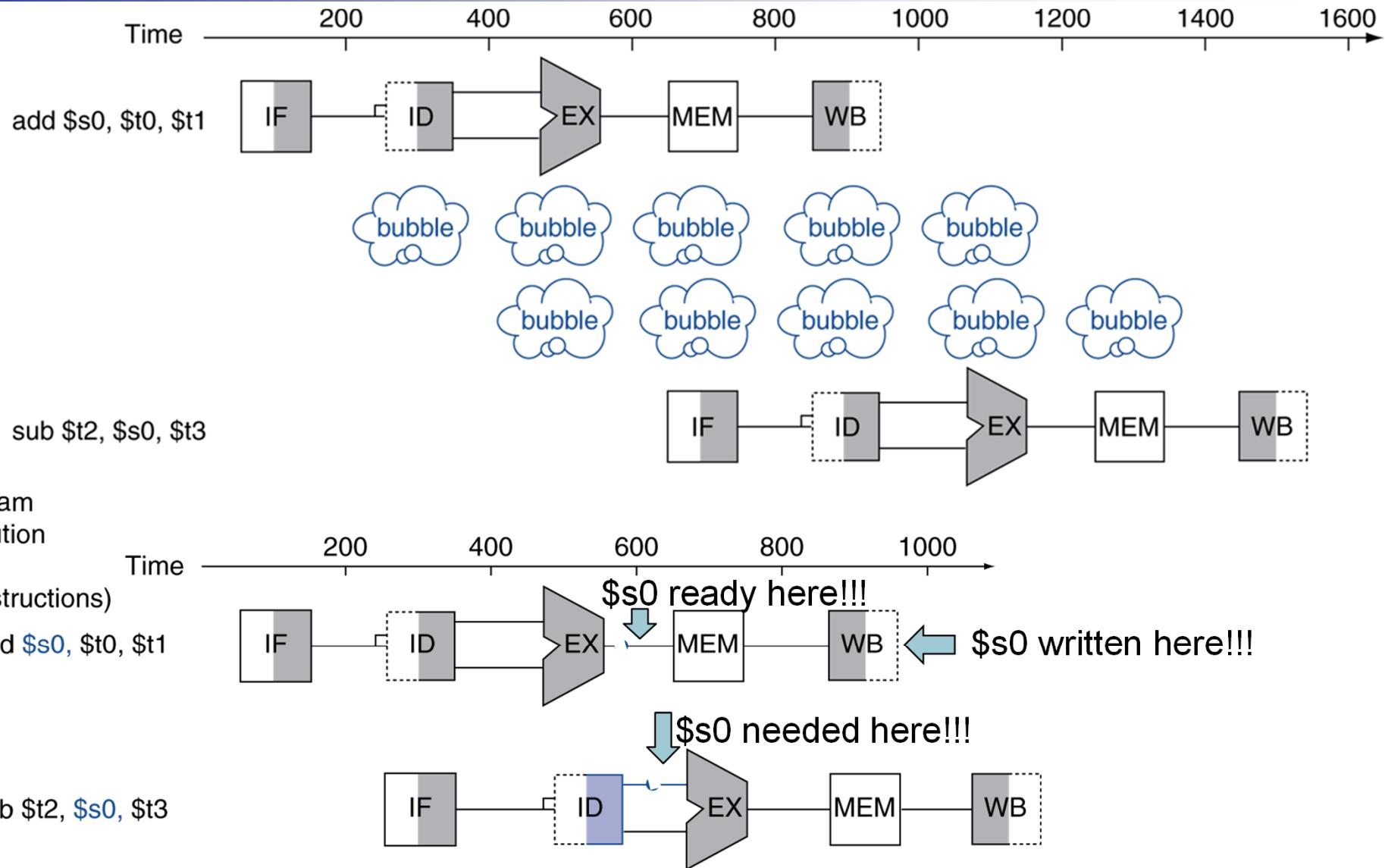
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



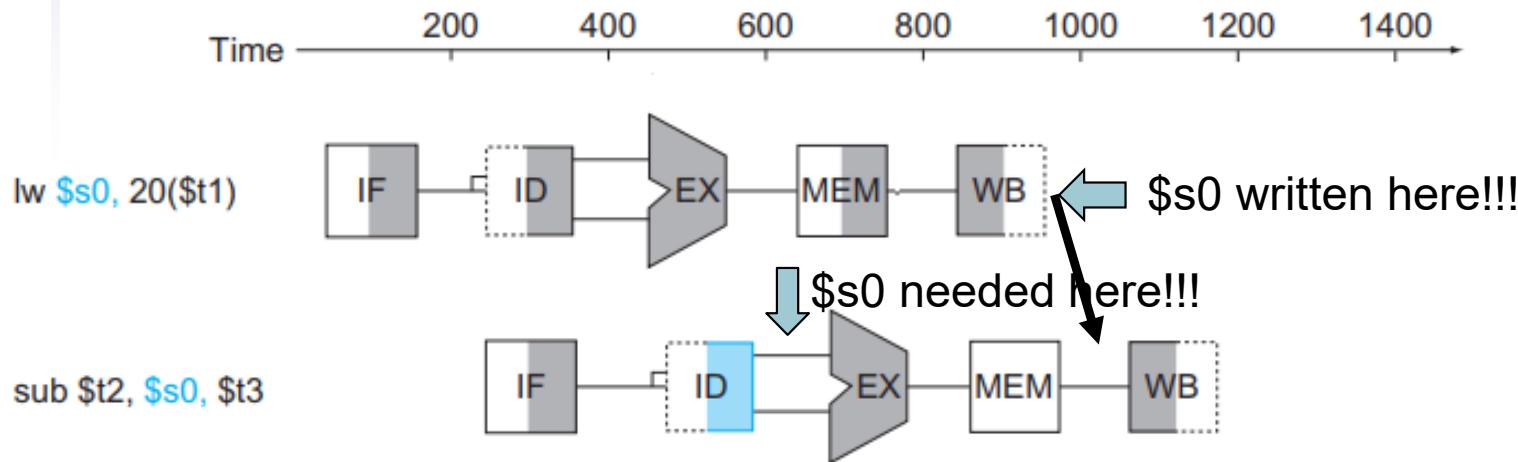
Forwarding (aka Bypassing)

COMPARE



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!



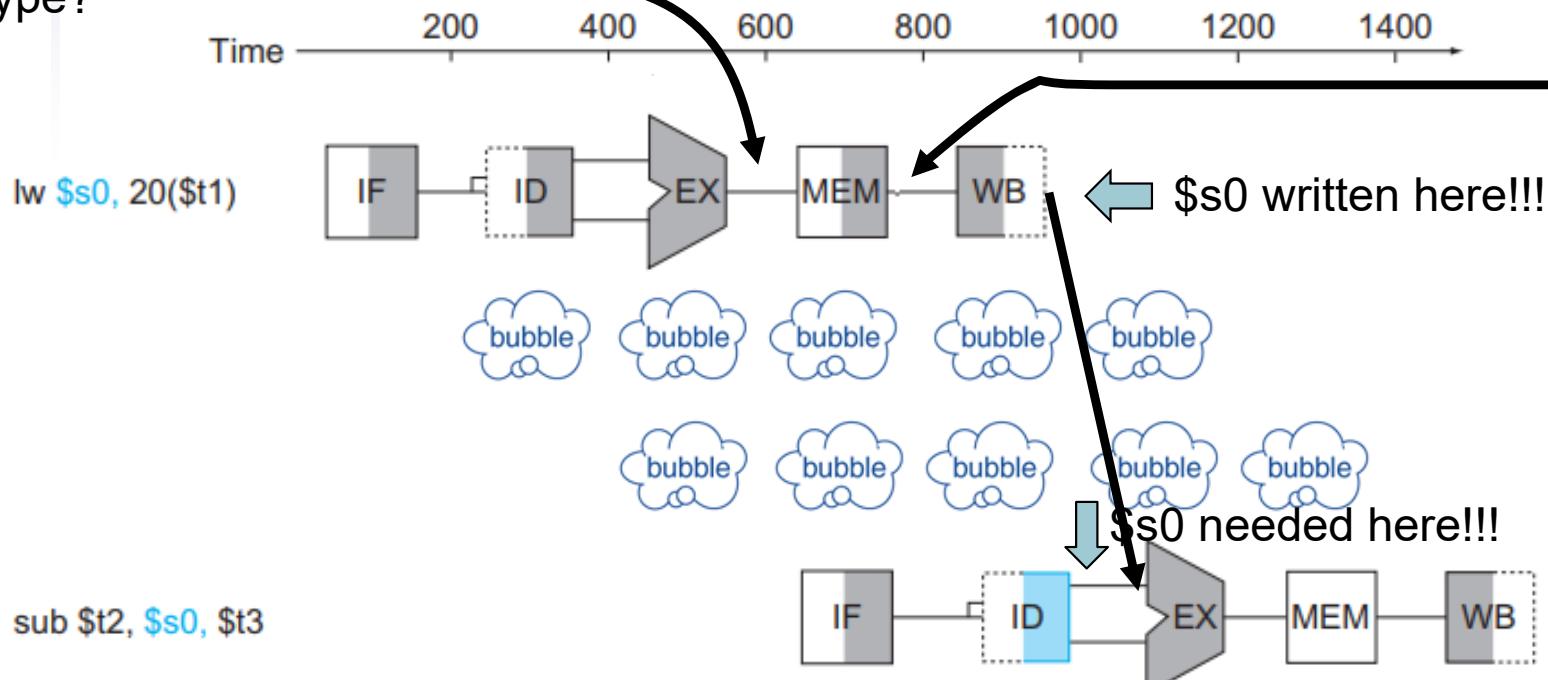
Load-Use Data Hazard

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!

When WAS the
data Ready for R-
type?

Can we solve by
FORWARDING?

When is the data
Ready?



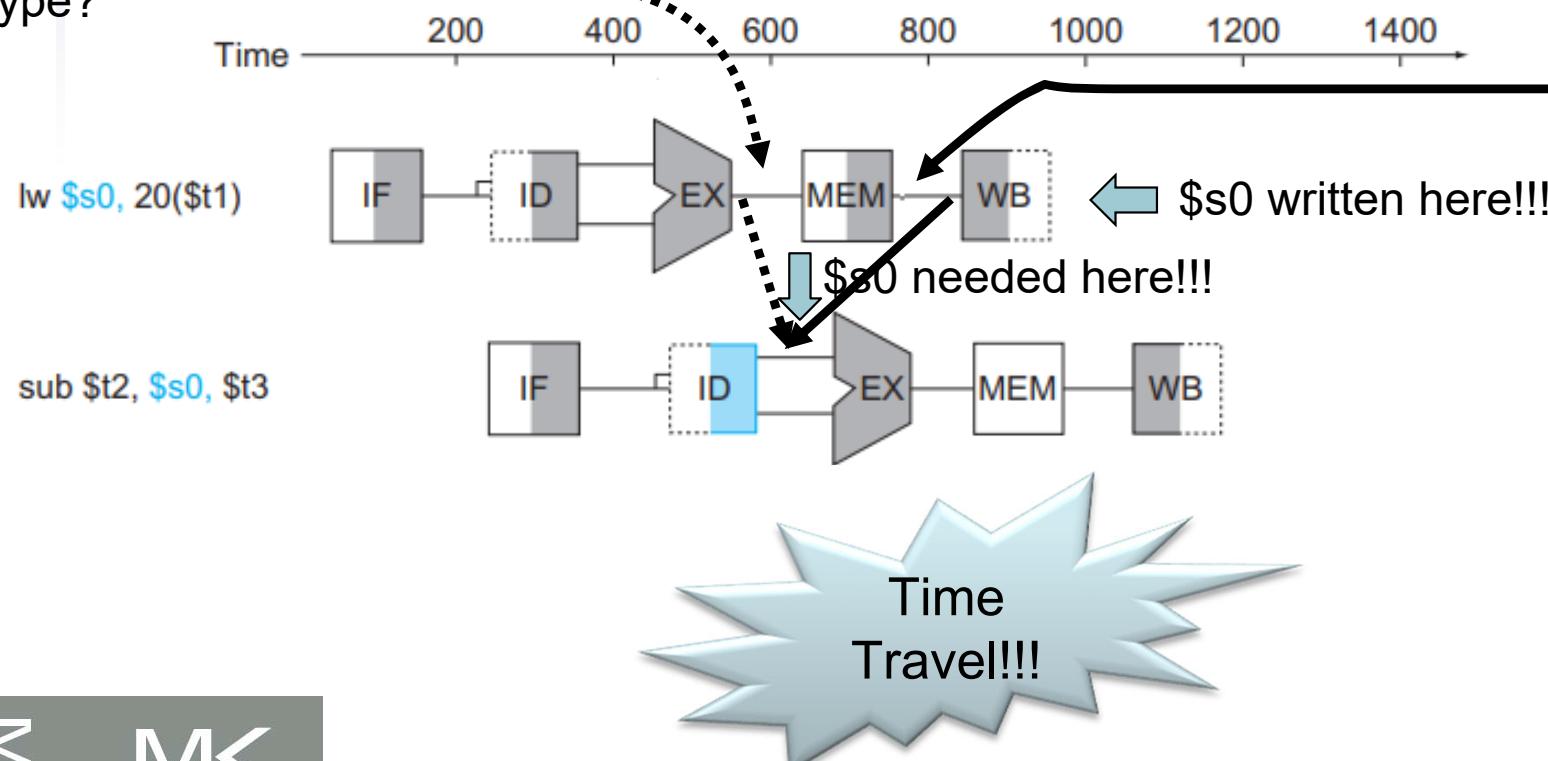
Load-Use Data Hazard

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!

When WAS the
data Ready for R-
type?

Can we solve by
FORWARDING?

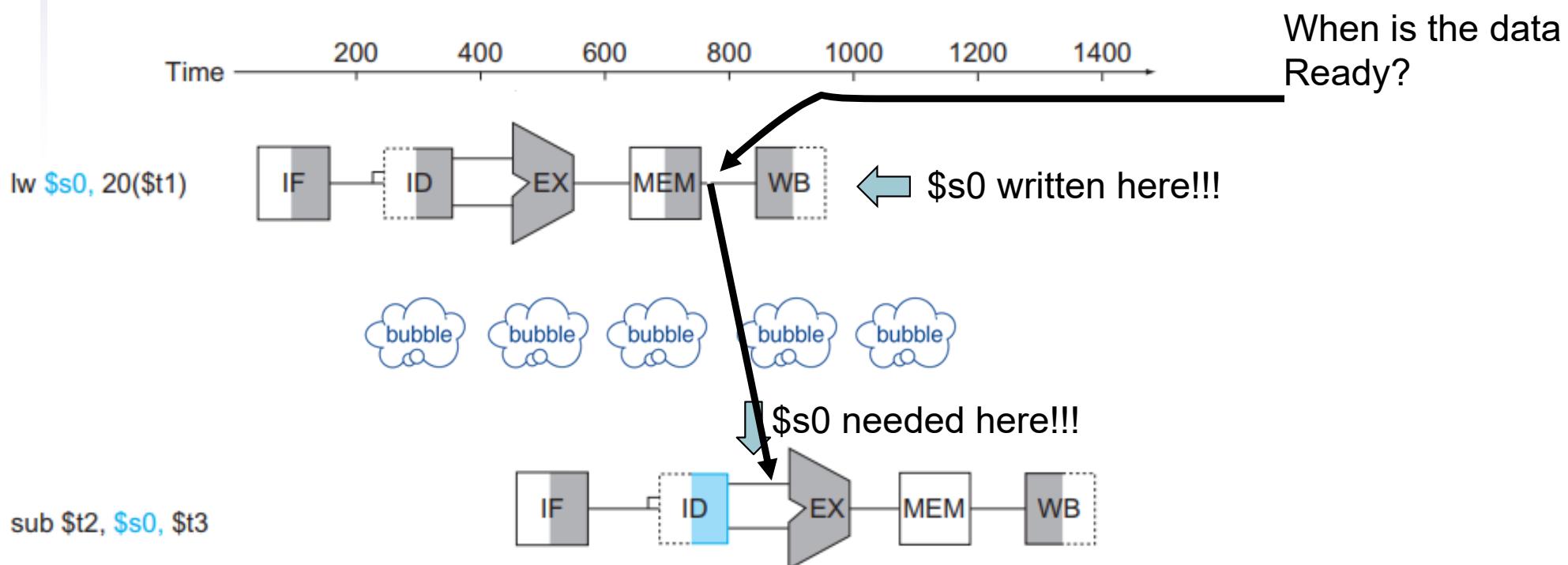
When is the data
Ready?



Load-Use Data Hazard

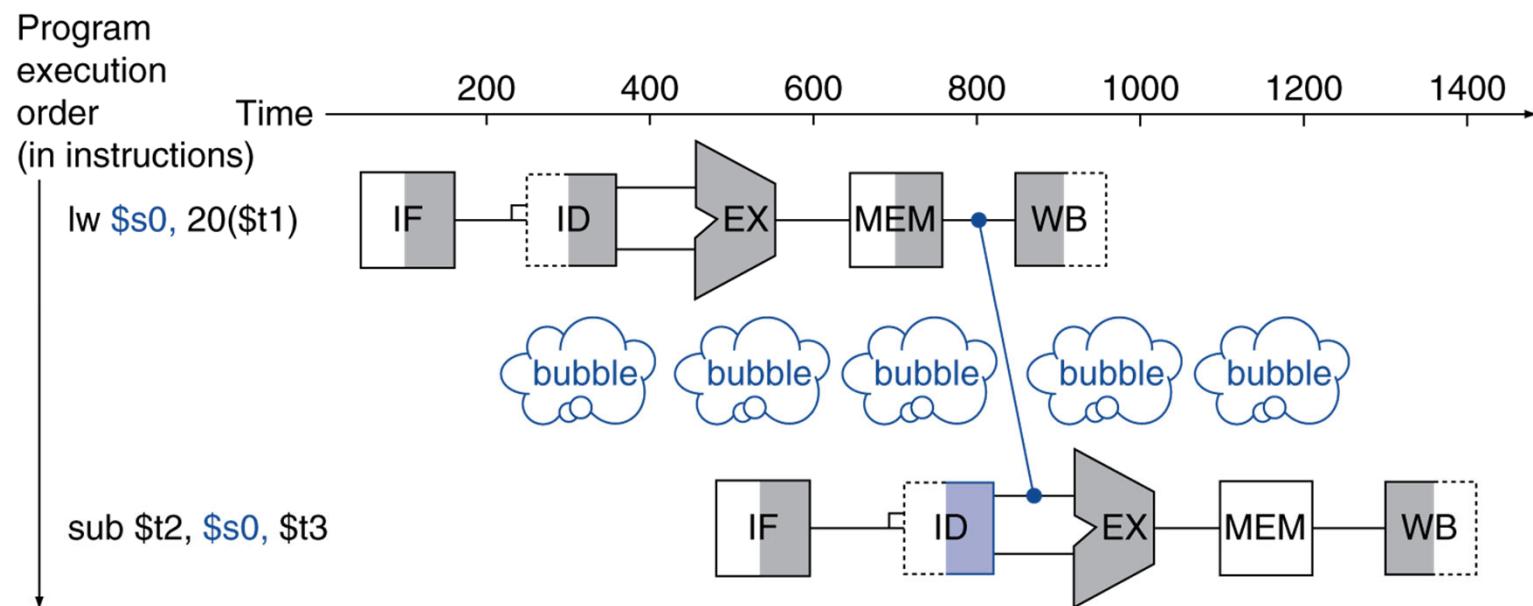
- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!

Can we solve by FORWARDING?



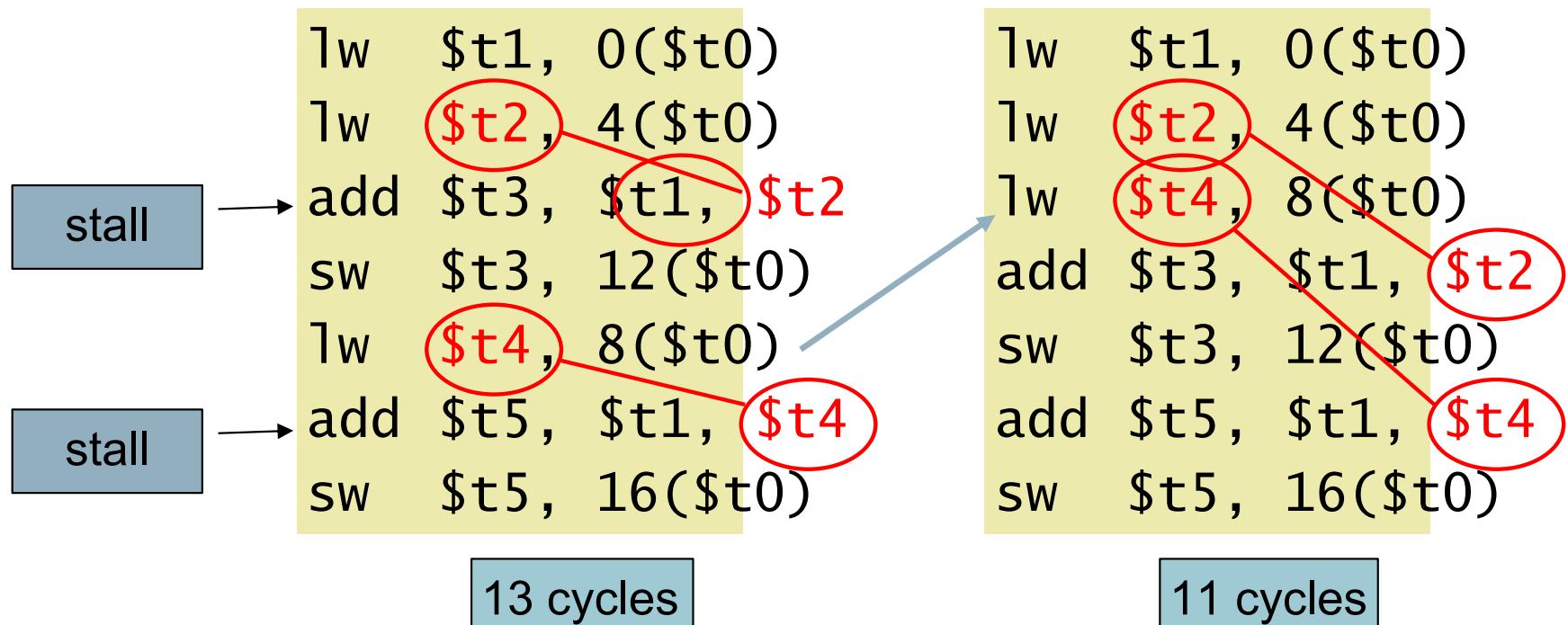
Load-Use Data Hazard

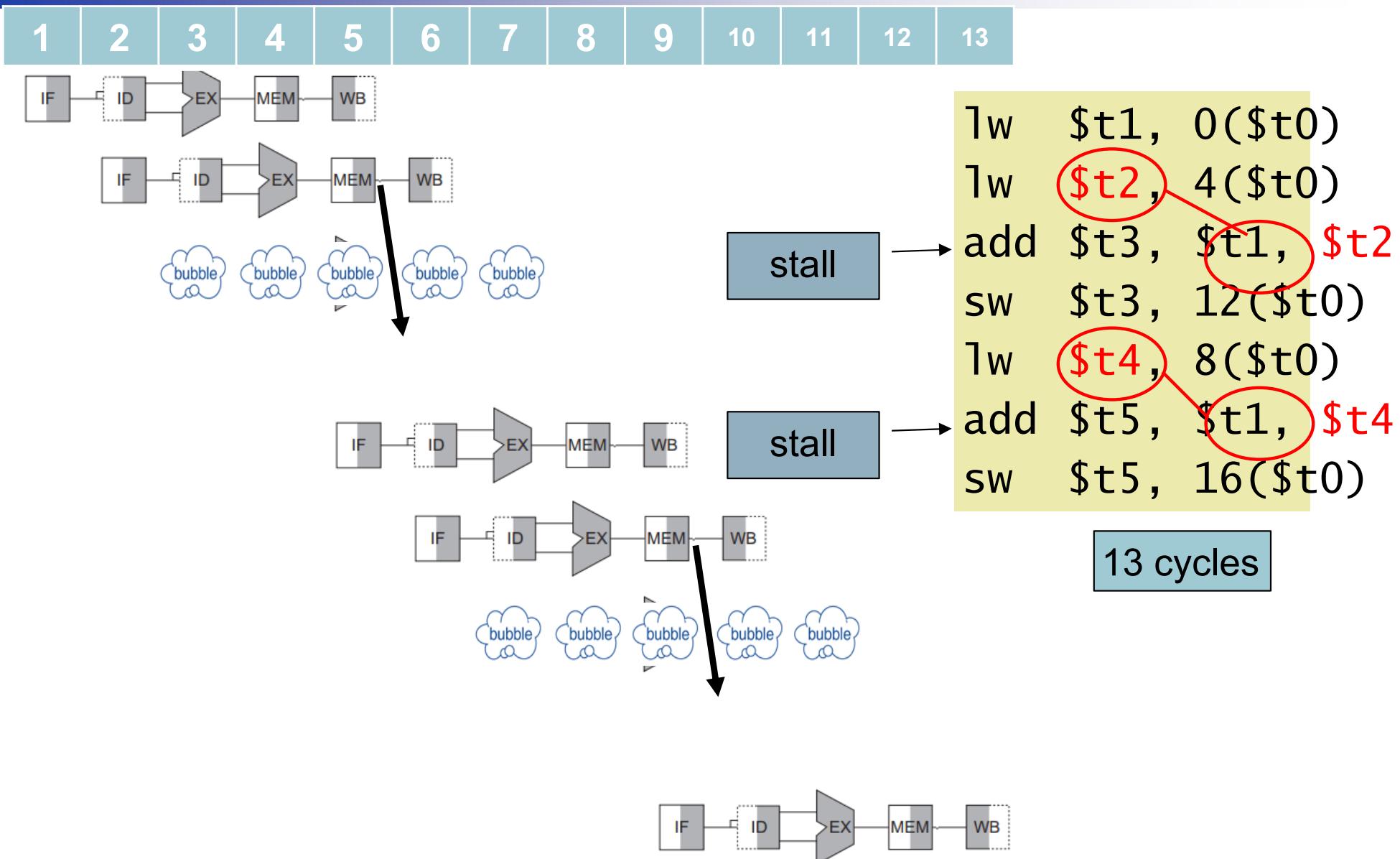
- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!

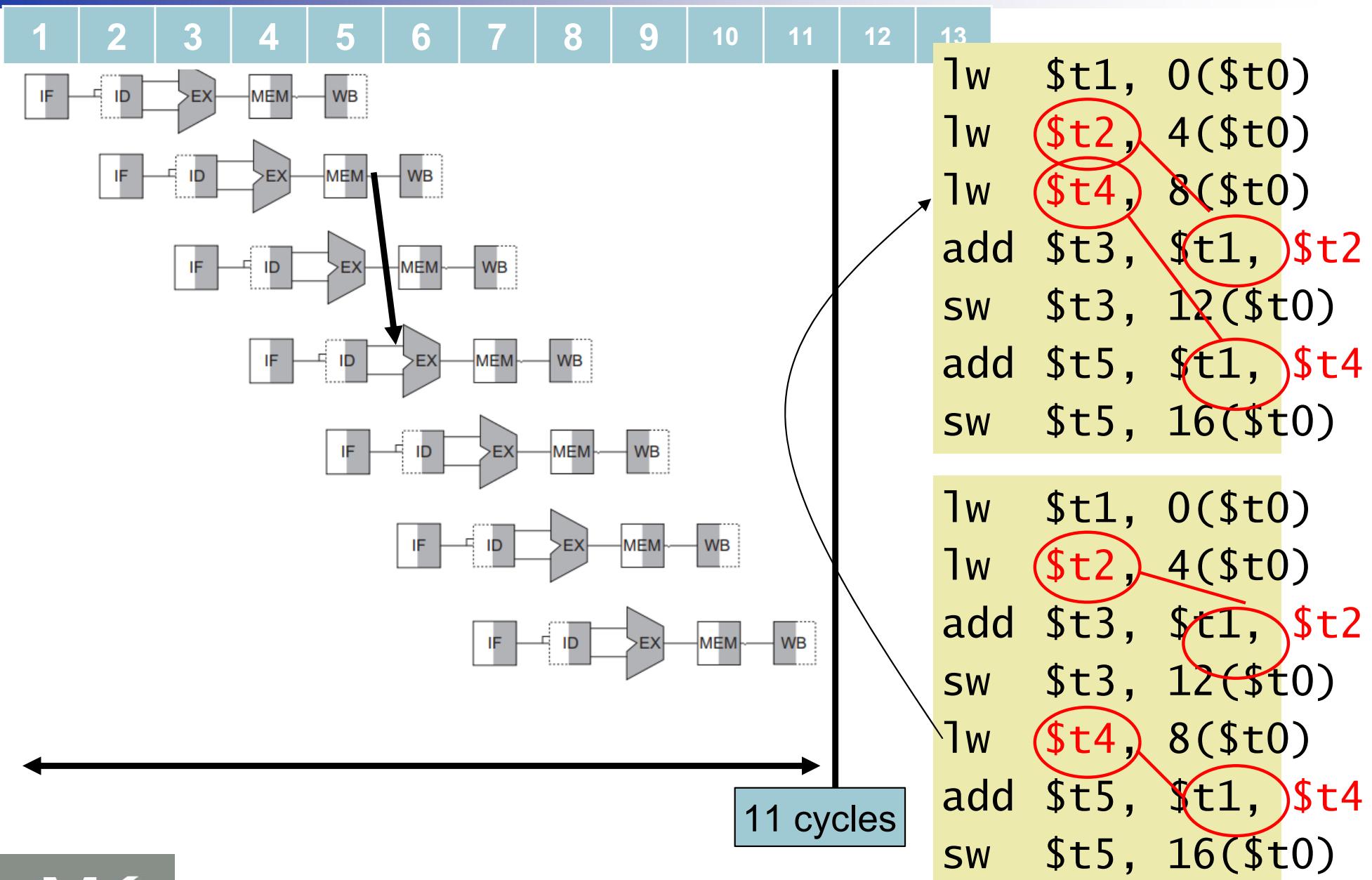


Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E; C = B + F;$



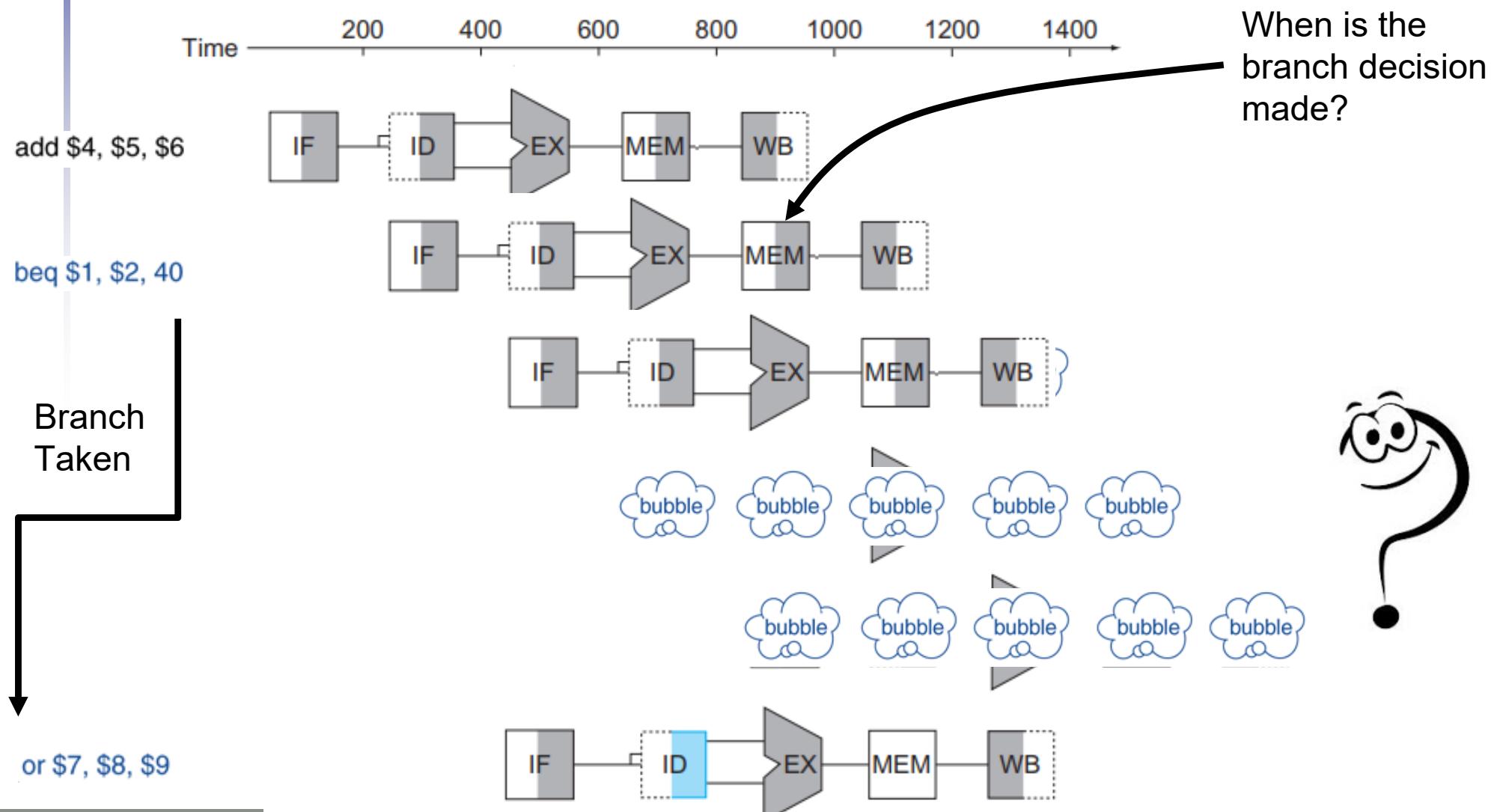




Control Hazards

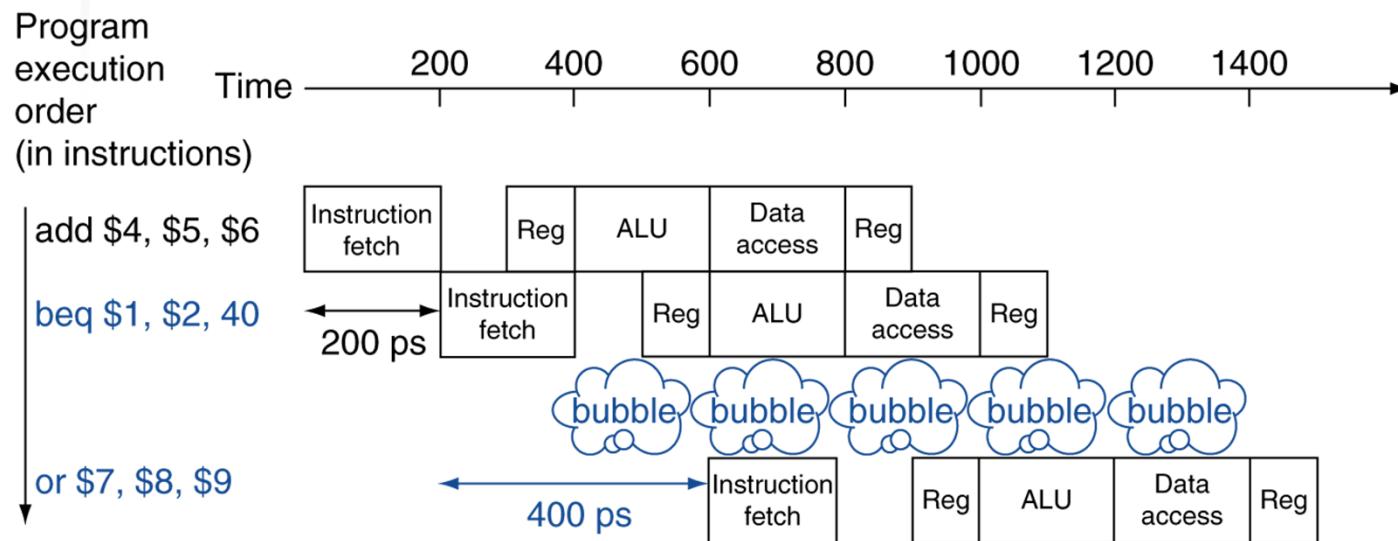
- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- In MIPS pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage

Branch Scenario



Stall on Branch (Improved)

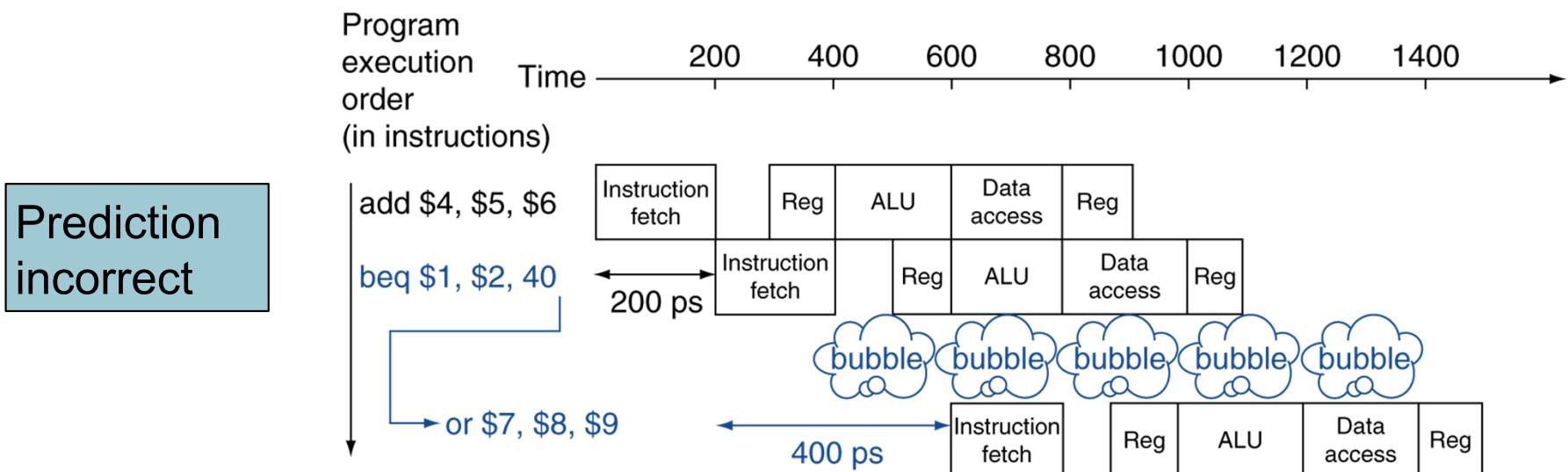
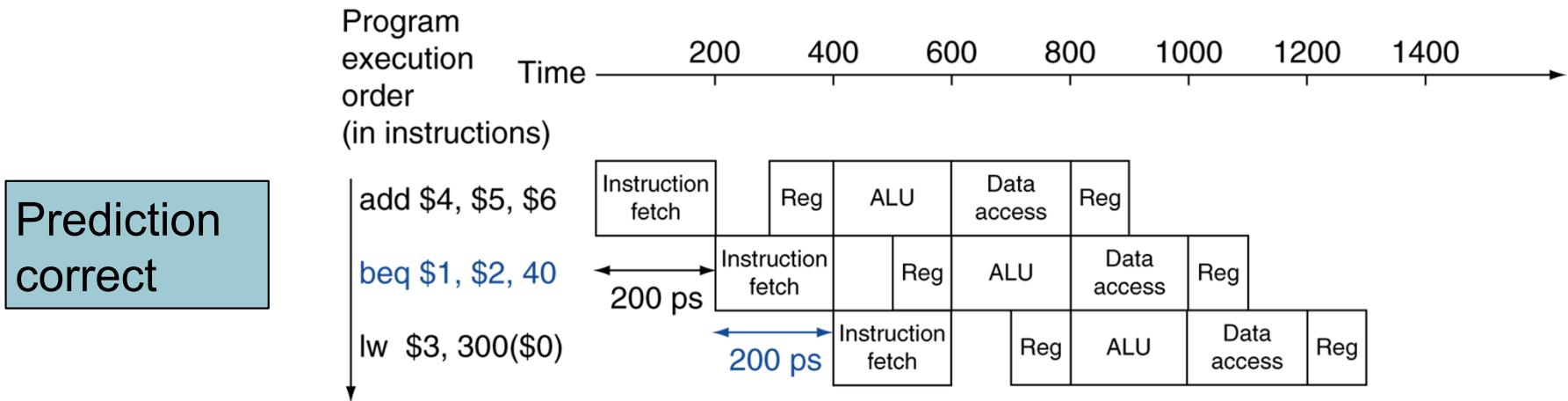
- Assume enough extra hardware to test registers, calculate the branch address, and update the PC during the second stage (ID stage)



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

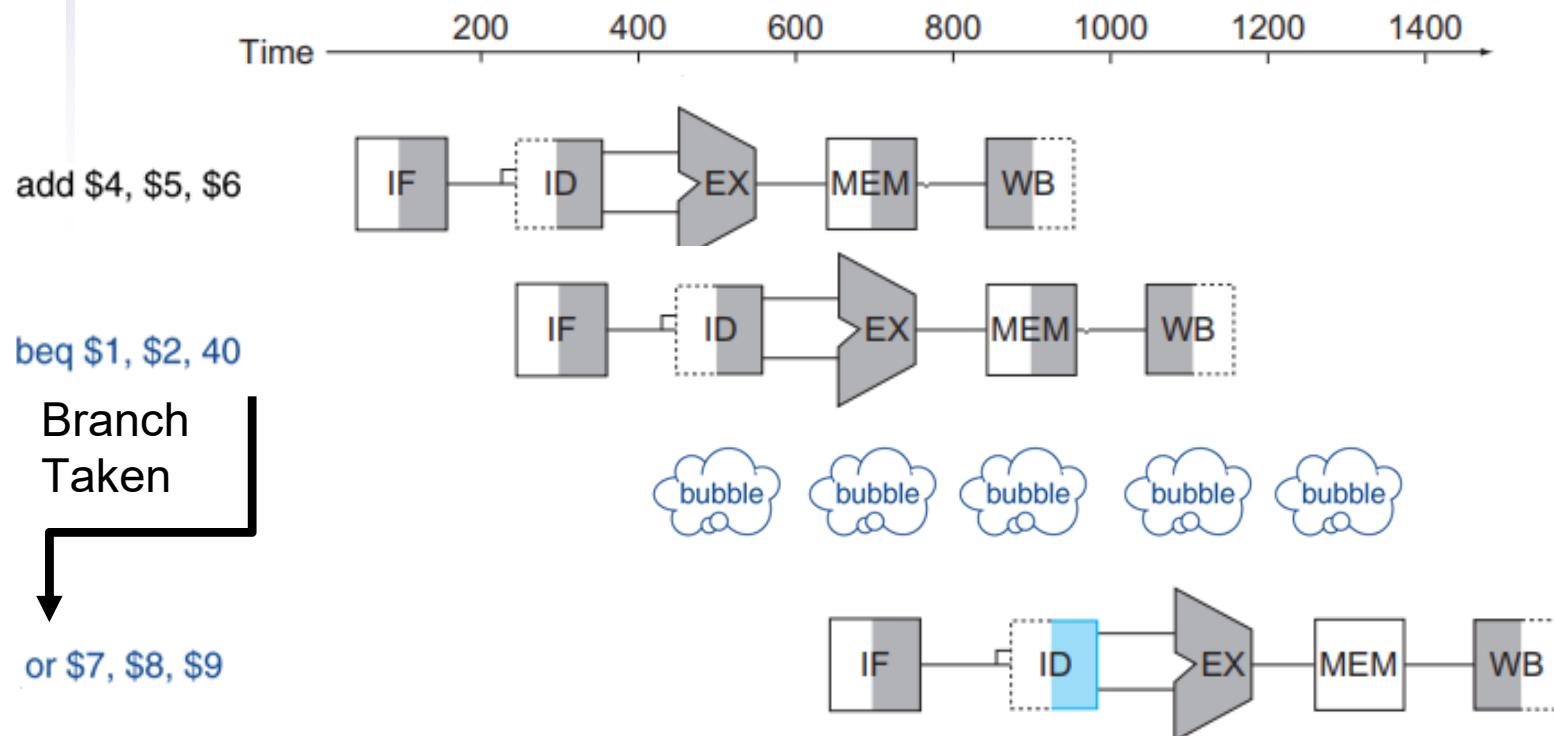


More-Realistic Branch Prediction

- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history

Delayed Branch

- Always execute the next sequential instruction
 - i.e., taken branch is always delayed
 - The delay is hidden by rearrangement



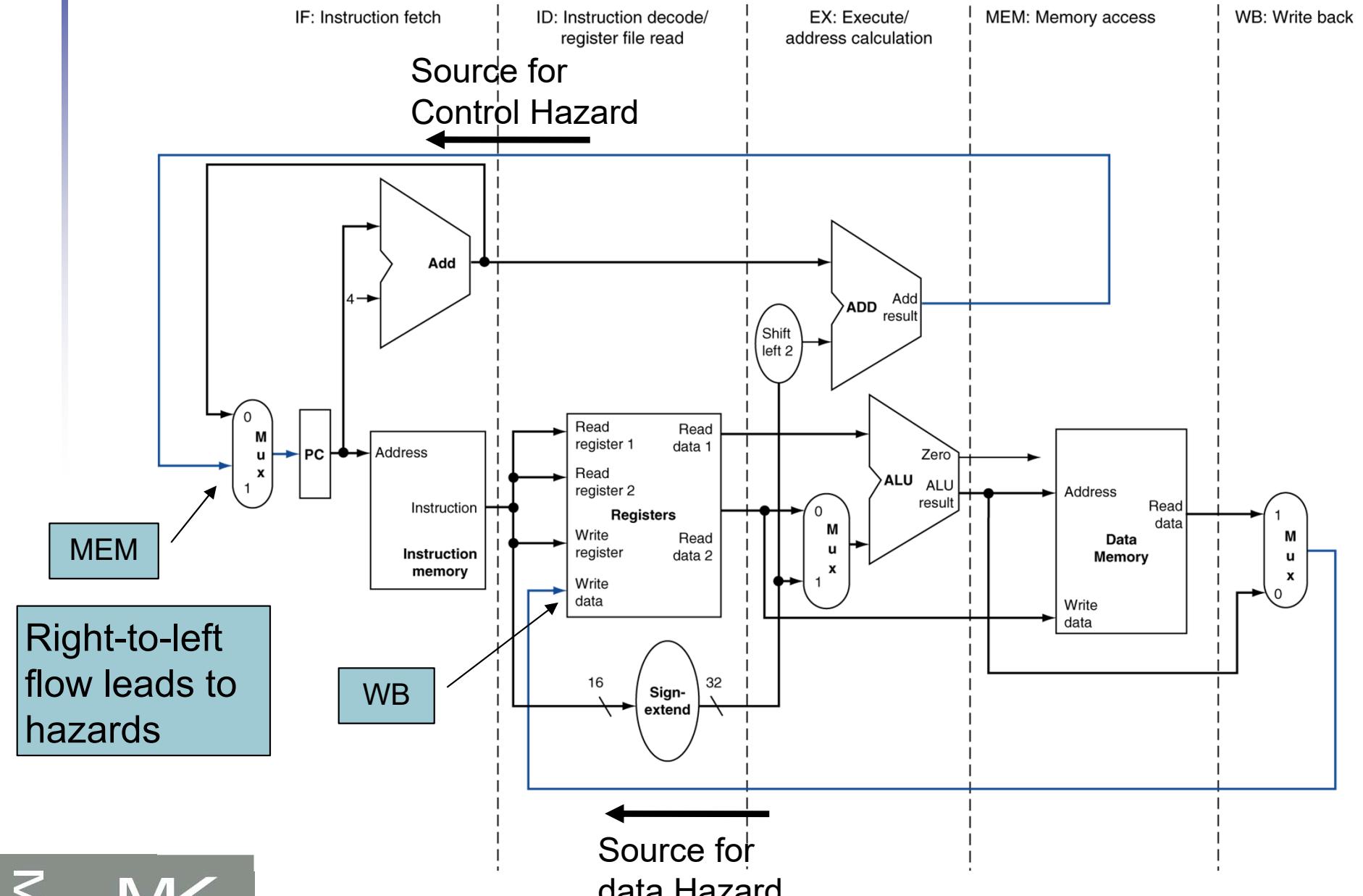
MIPS Solution to Control Hazard

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

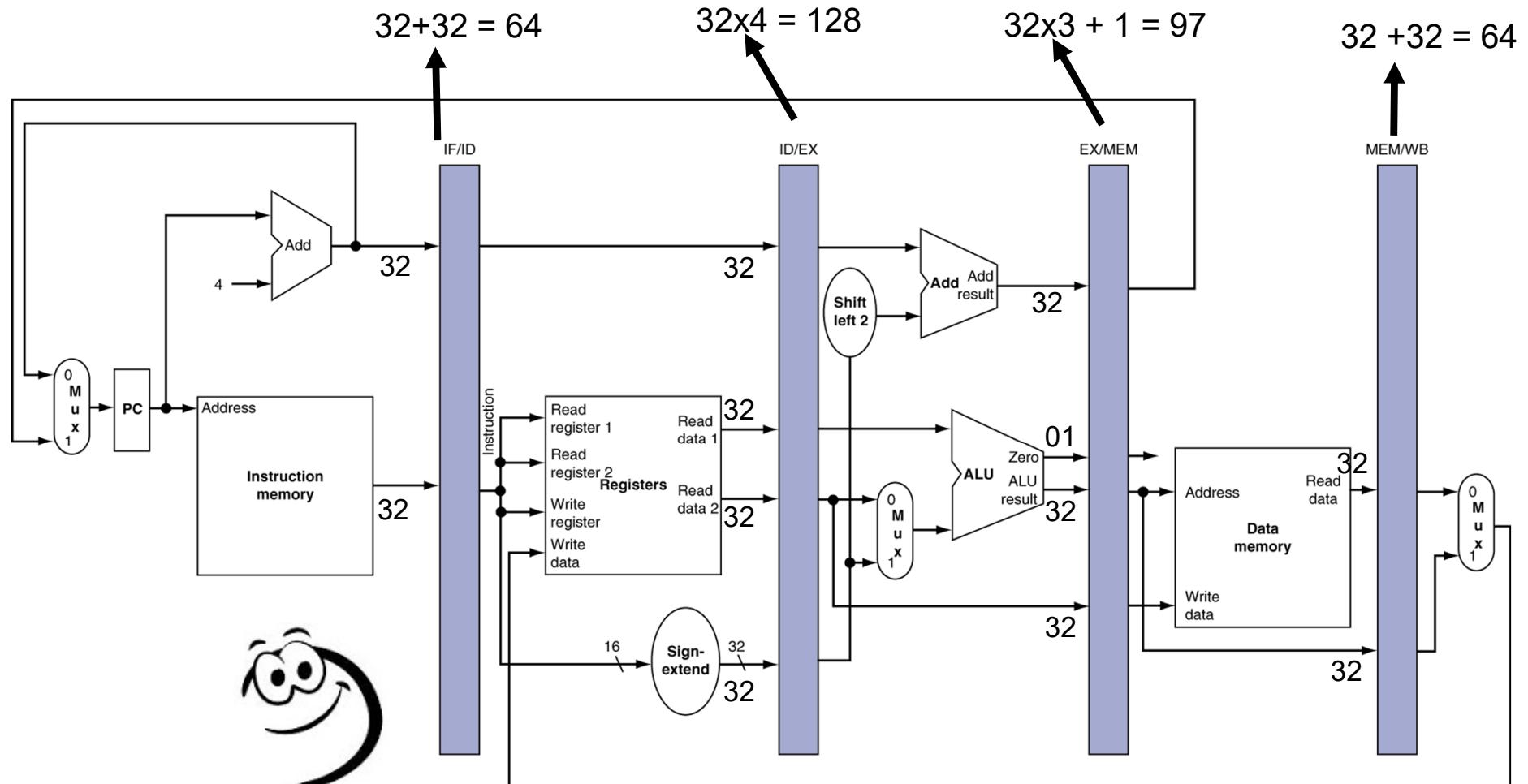
MIPS Pipelined Datapath



Pipeline registers

We may need to update the sizes as we proceed!!!

- Need registers between stages
 - To hold information produced in previous cycle



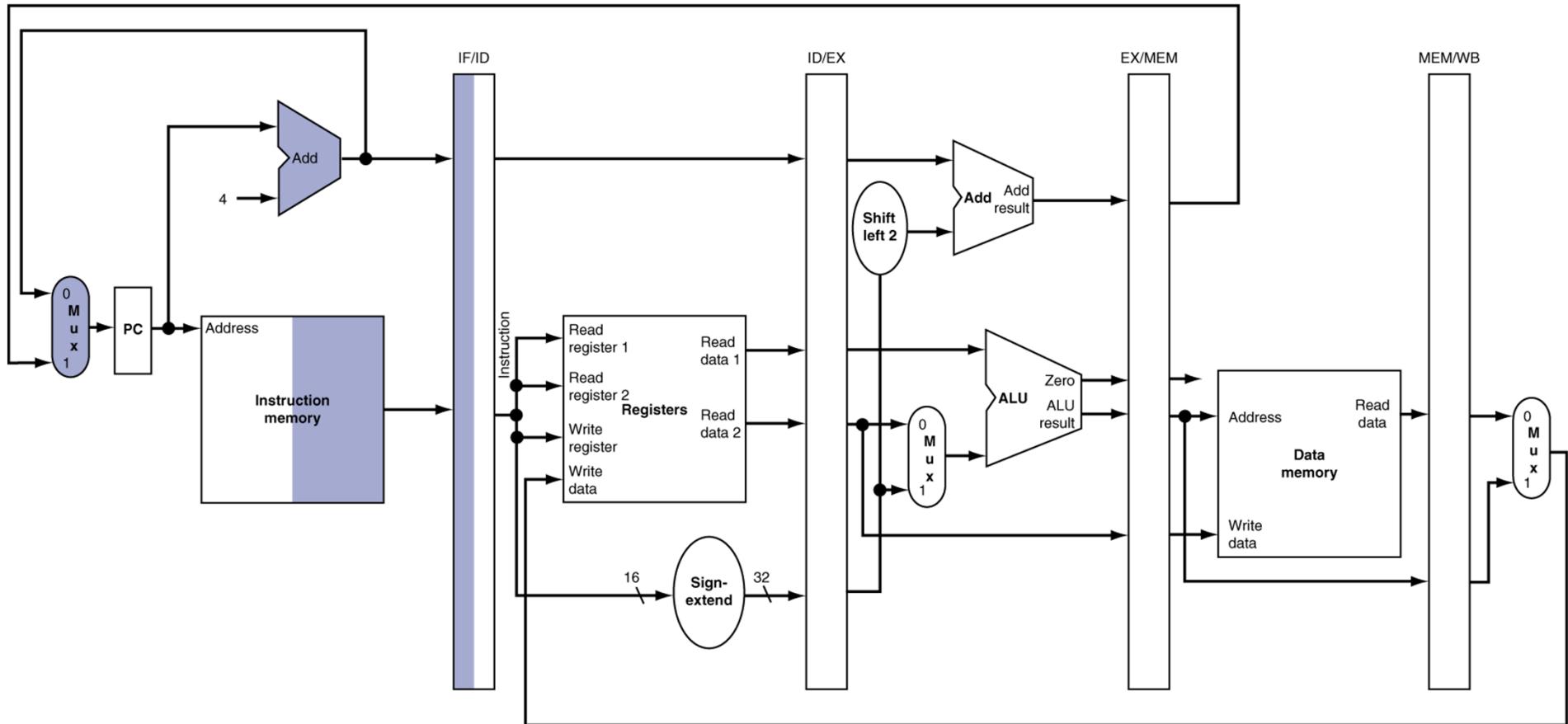
What is the size of the pipeline registers?

Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - c.f. “multi-clock-cycle” diagram
 - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

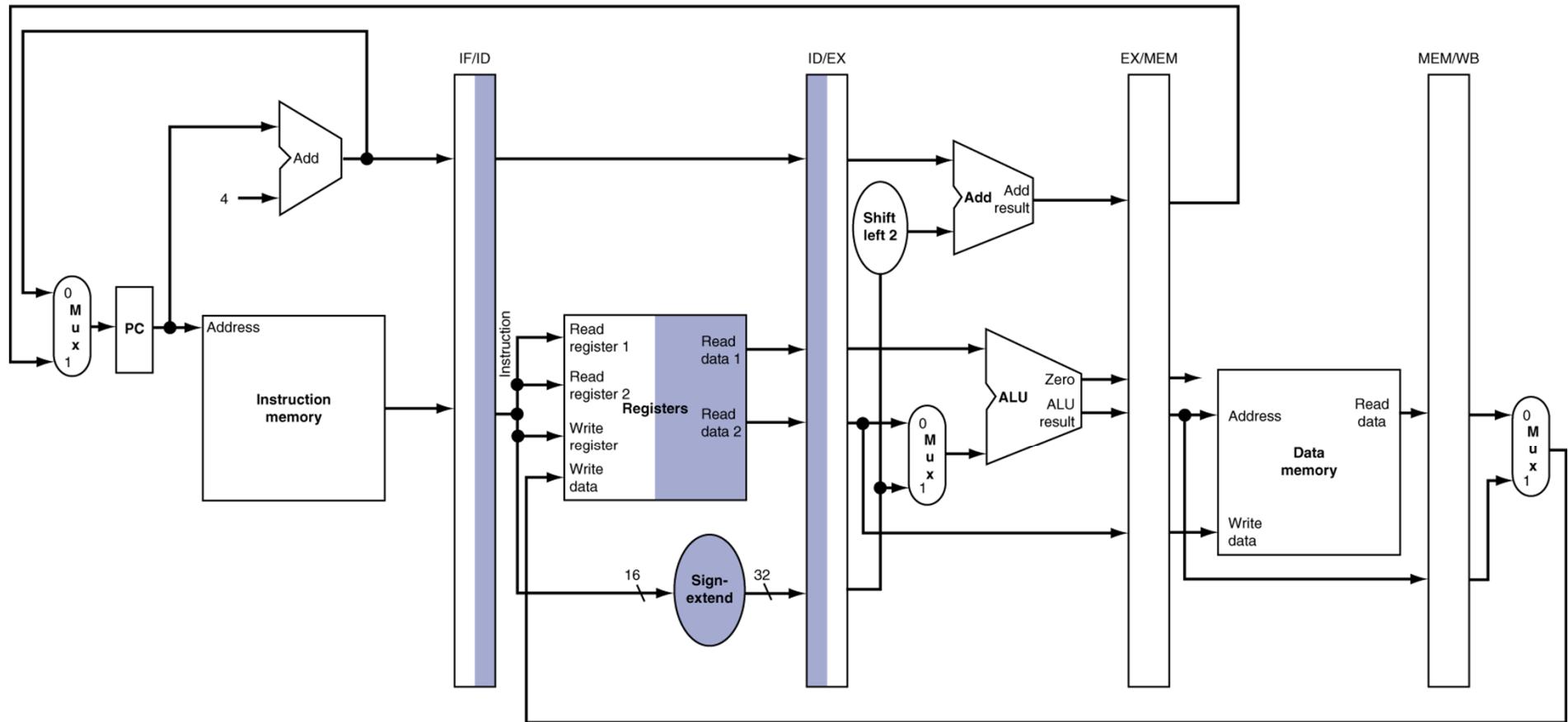
IF for Load

lw
Instruction fetch

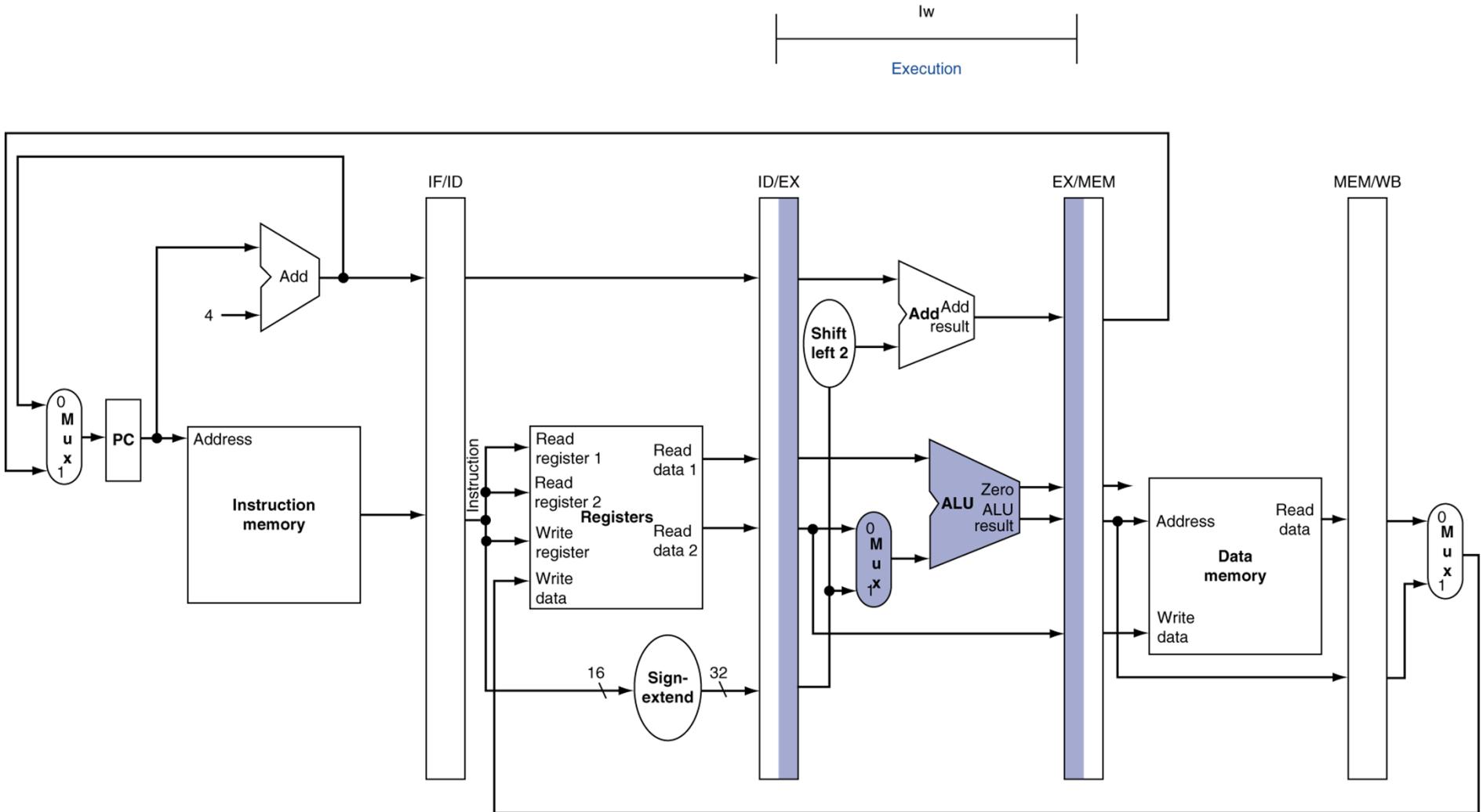


ID for Load

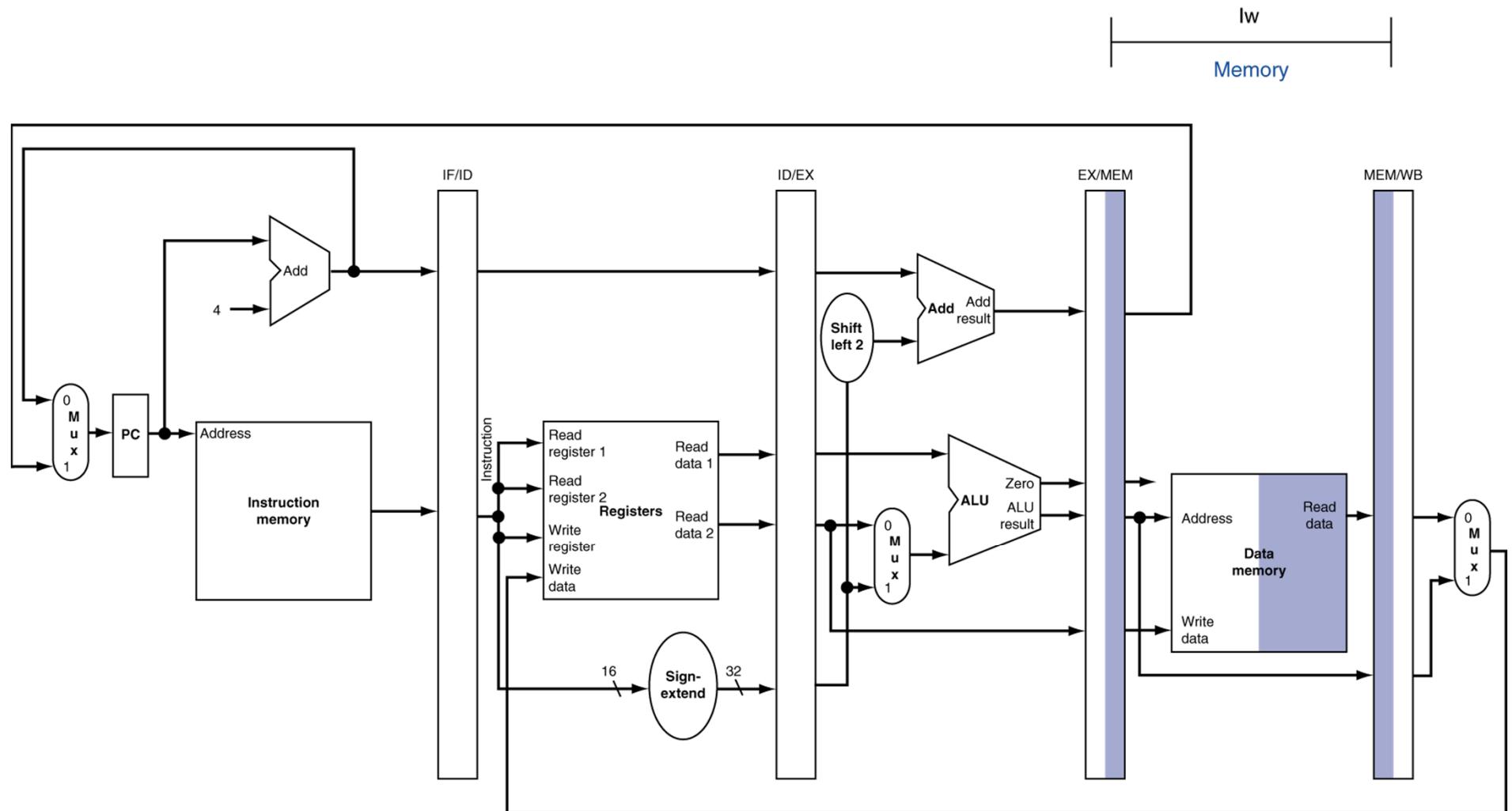
lw
Instruction decode



EX for Load

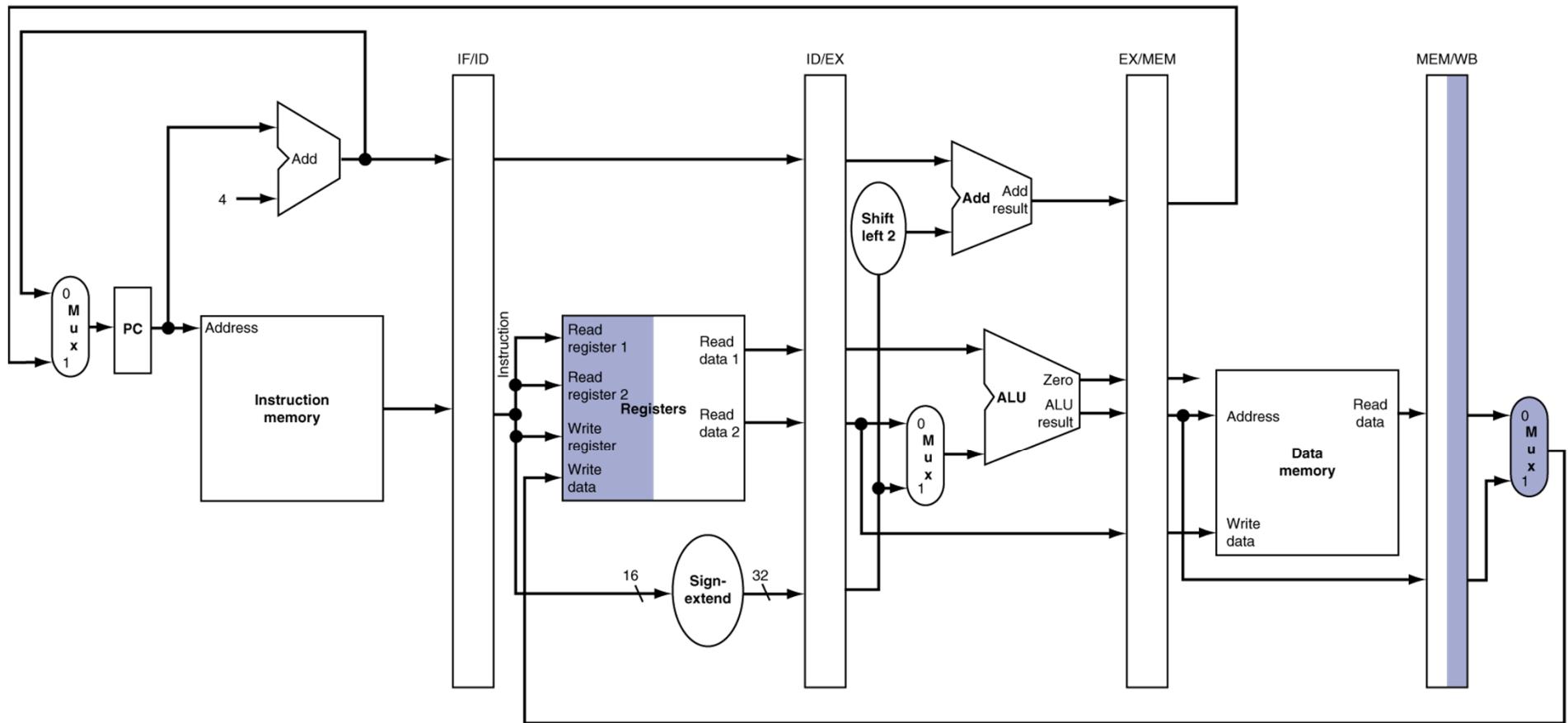


MEM for Load

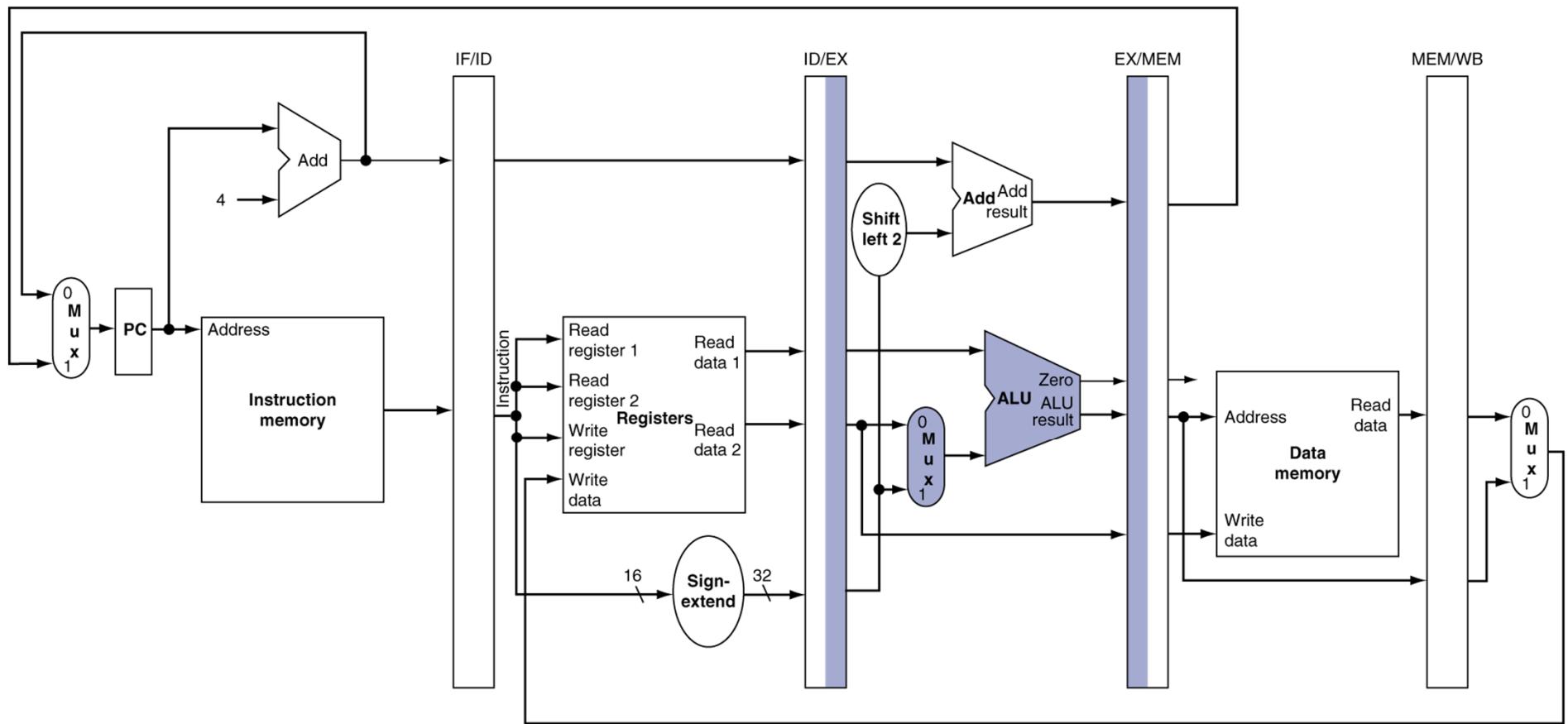


WB for Load

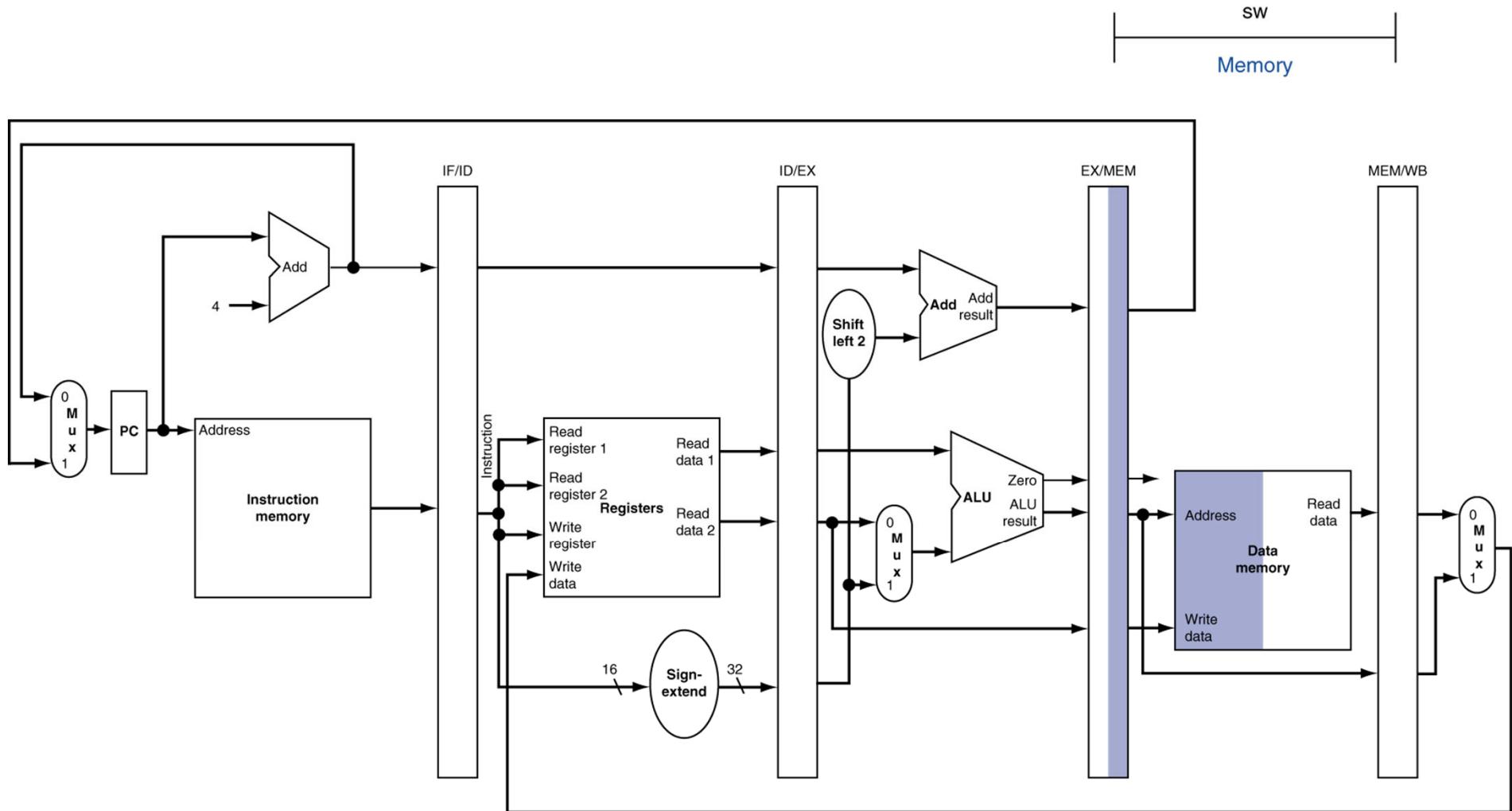
Iw
Write back



EX for Store

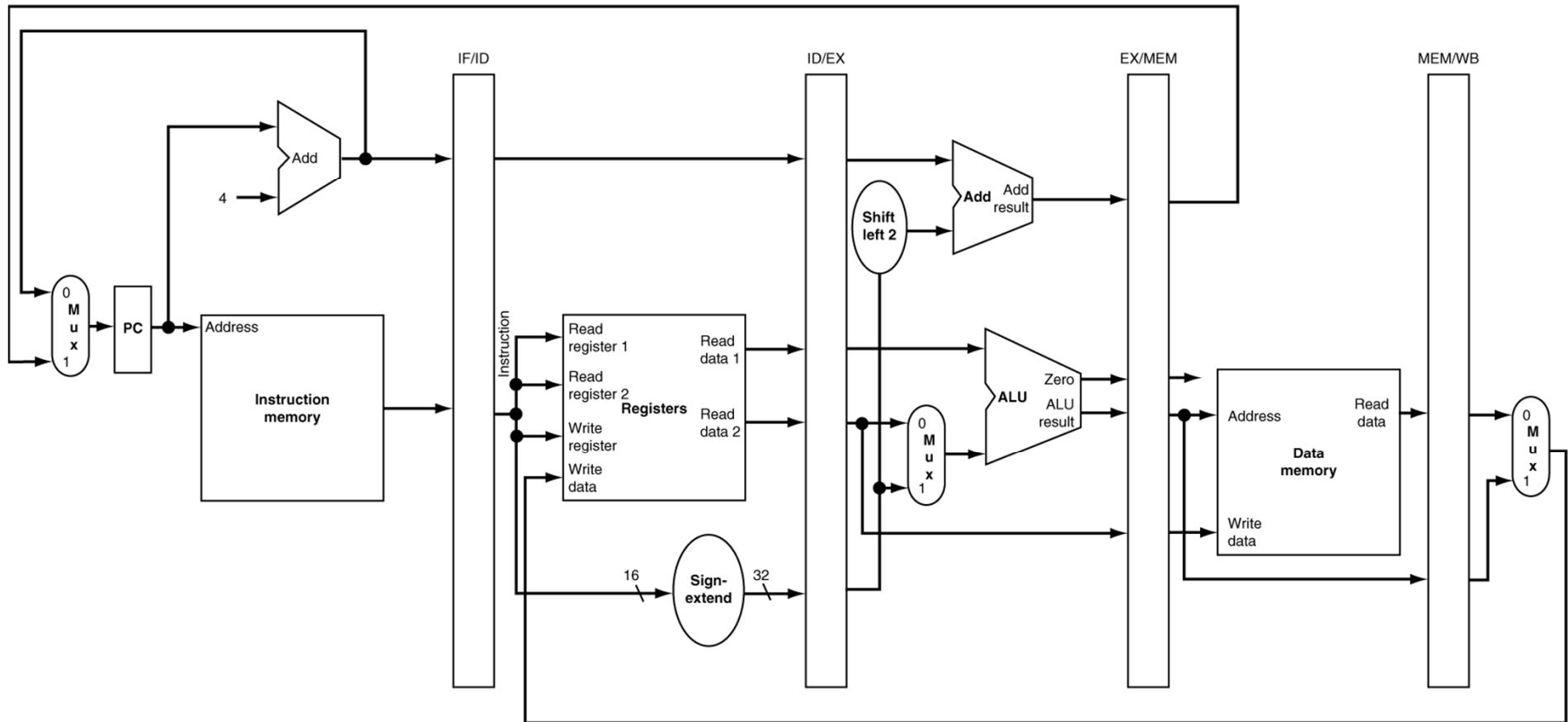


MEM for Store



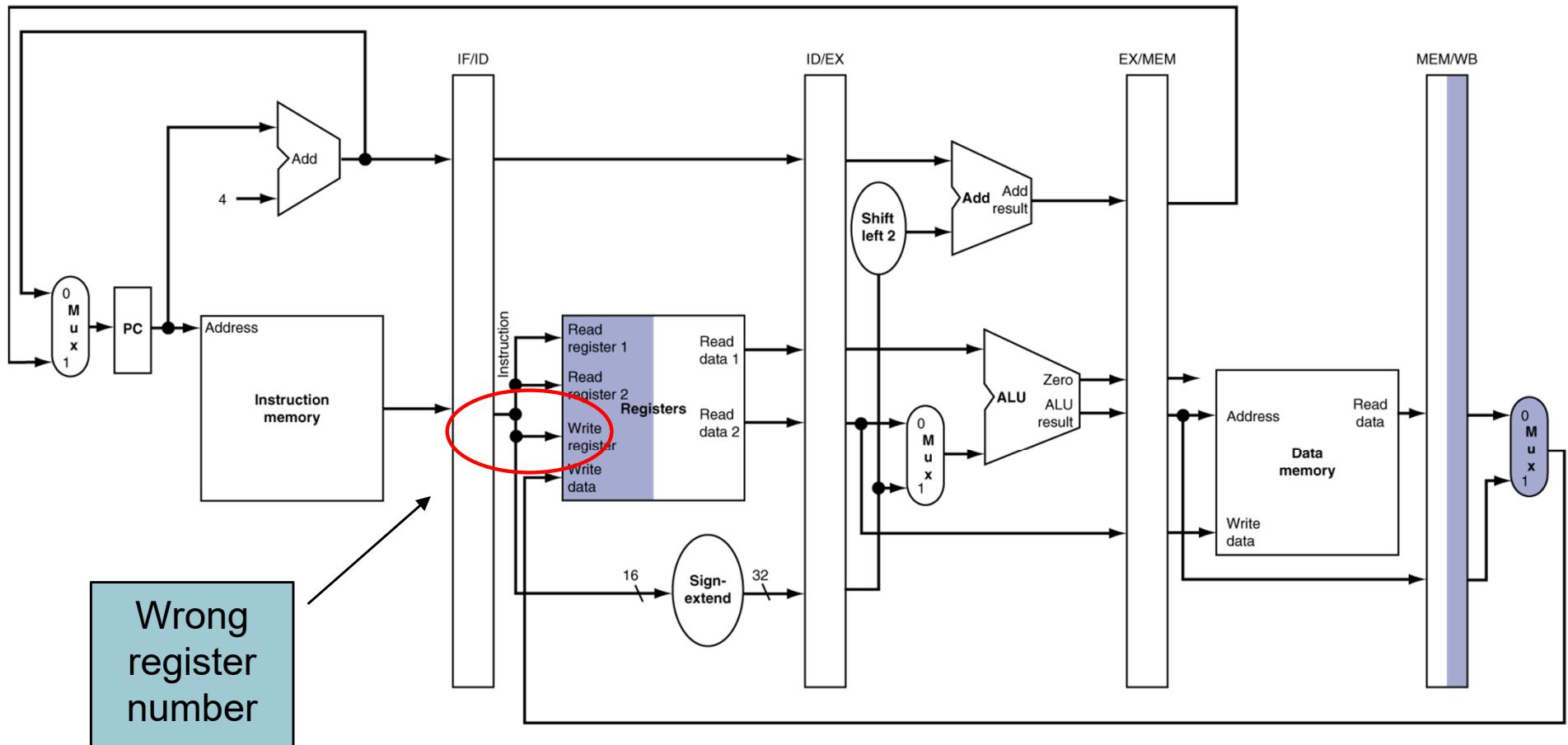
WB for Store

SW
Write-back

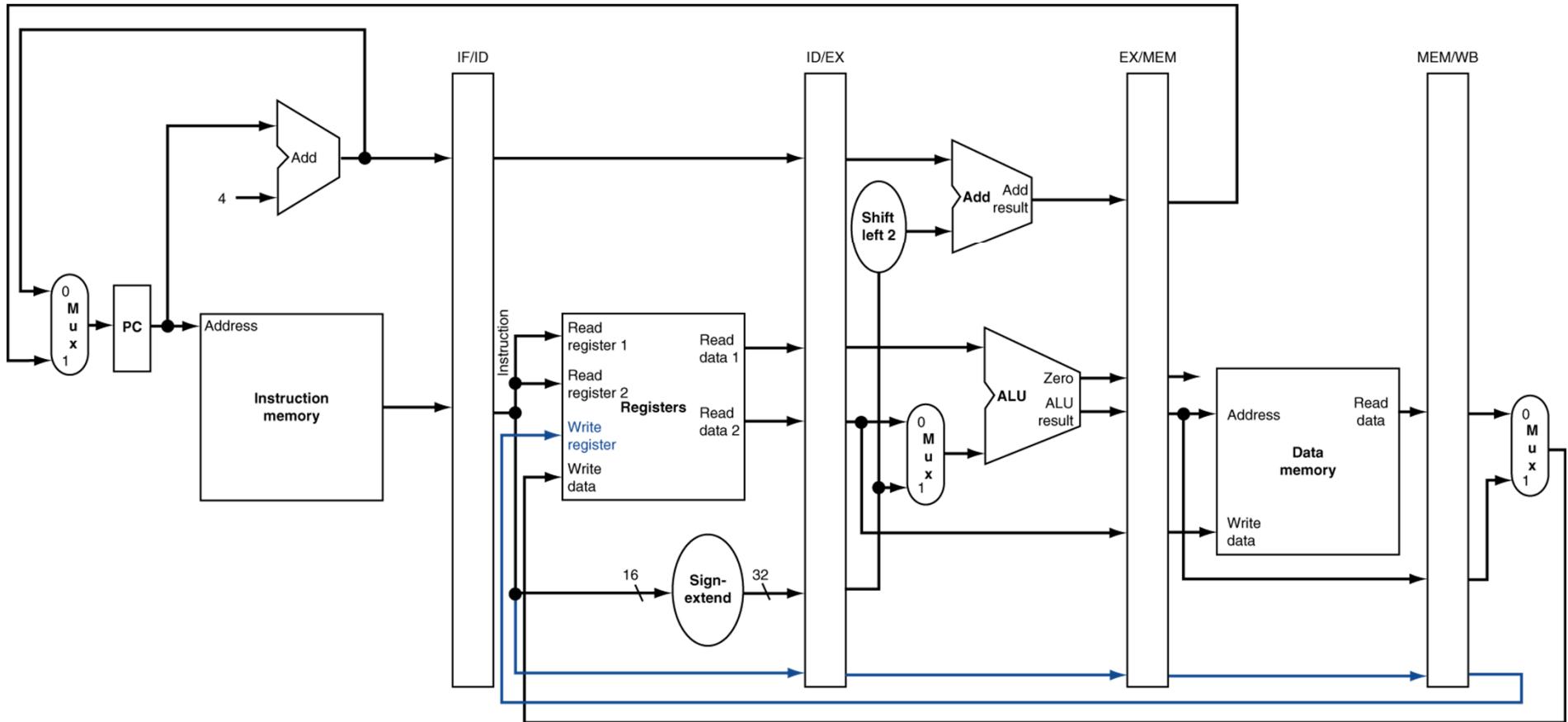


WB for Load Revisited

Iw
Write back

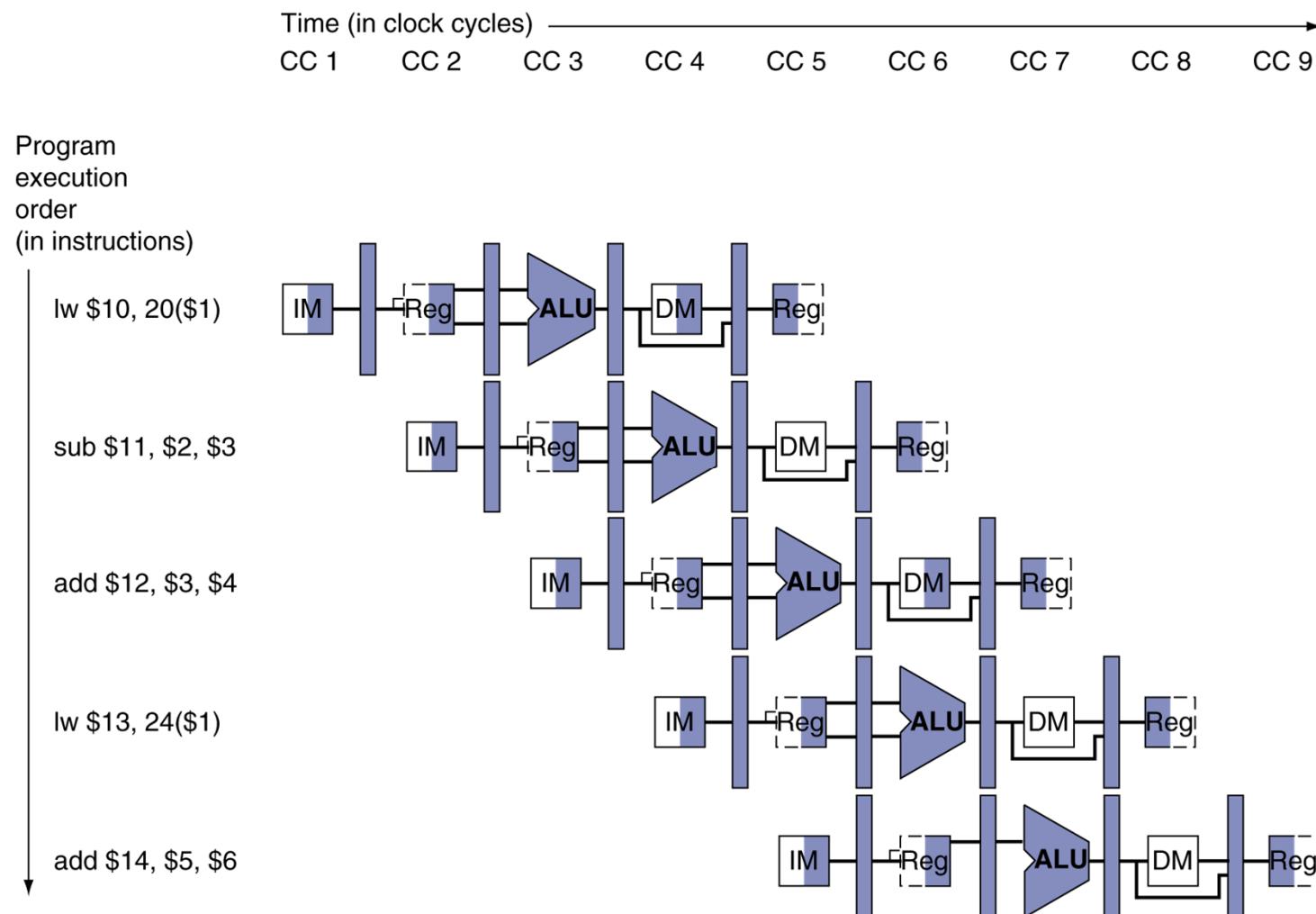


Corrected Datapath for Load



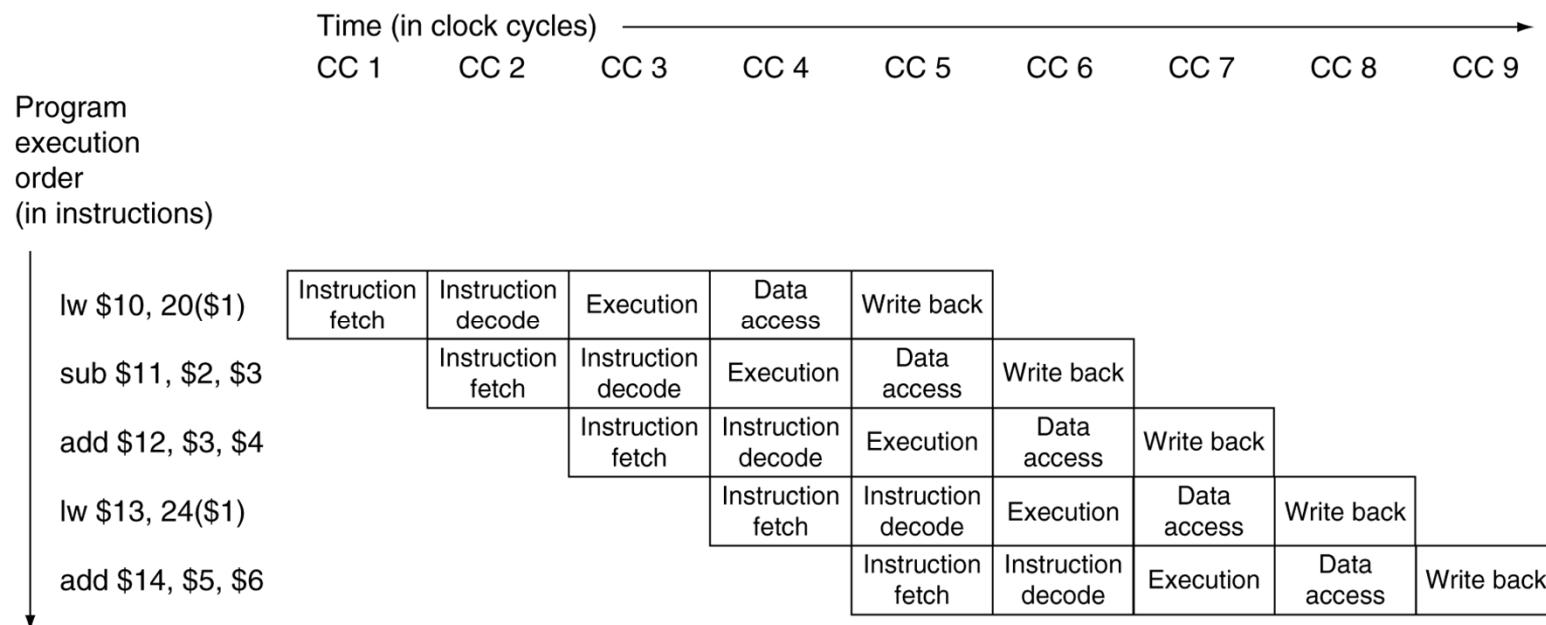
Multi-Cycle Pipeline Diagram

- Form showing resource usage



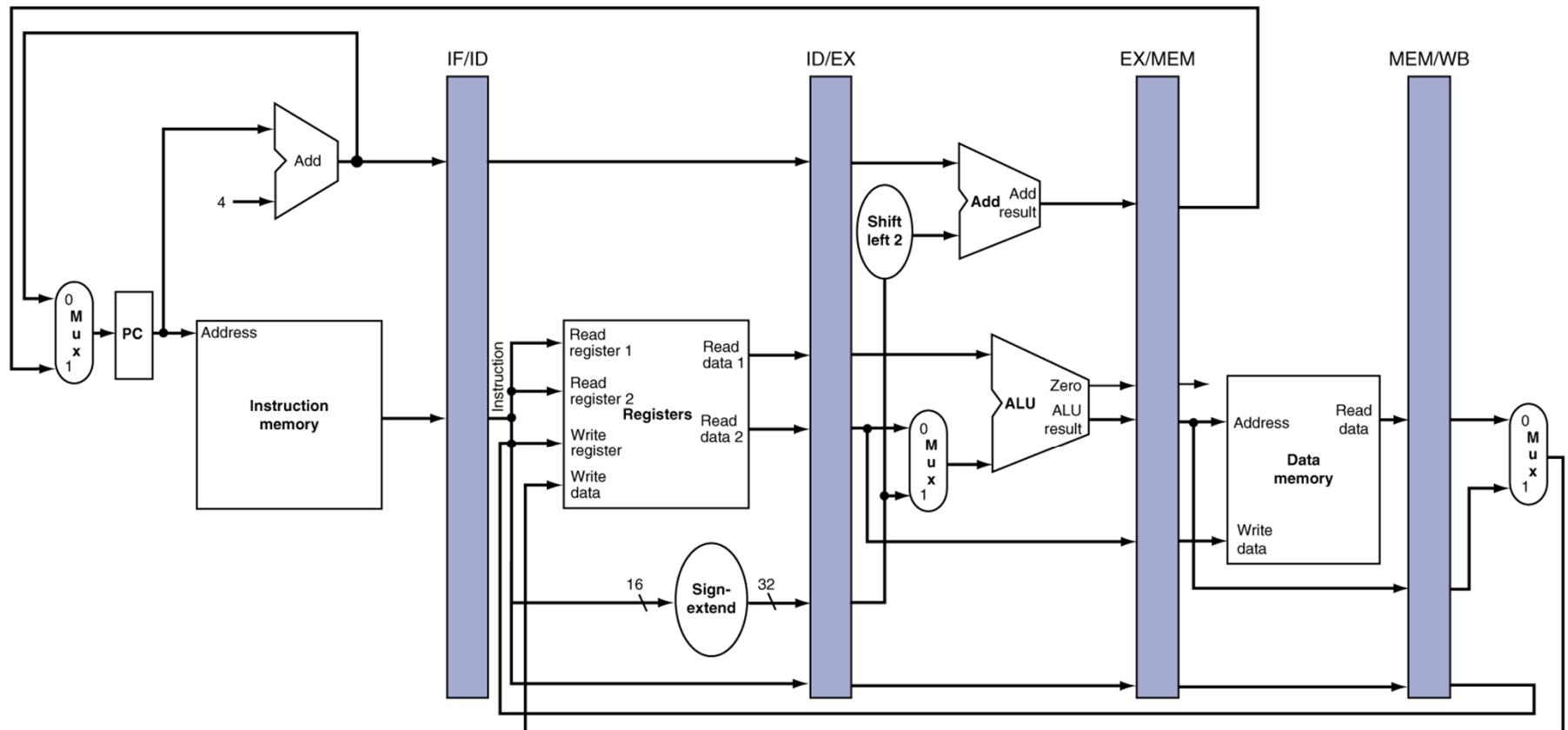
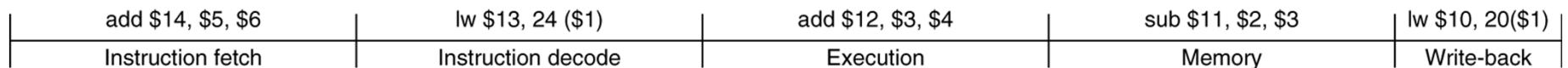
Multi-Cycle Pipeline Diagram

Traditional form



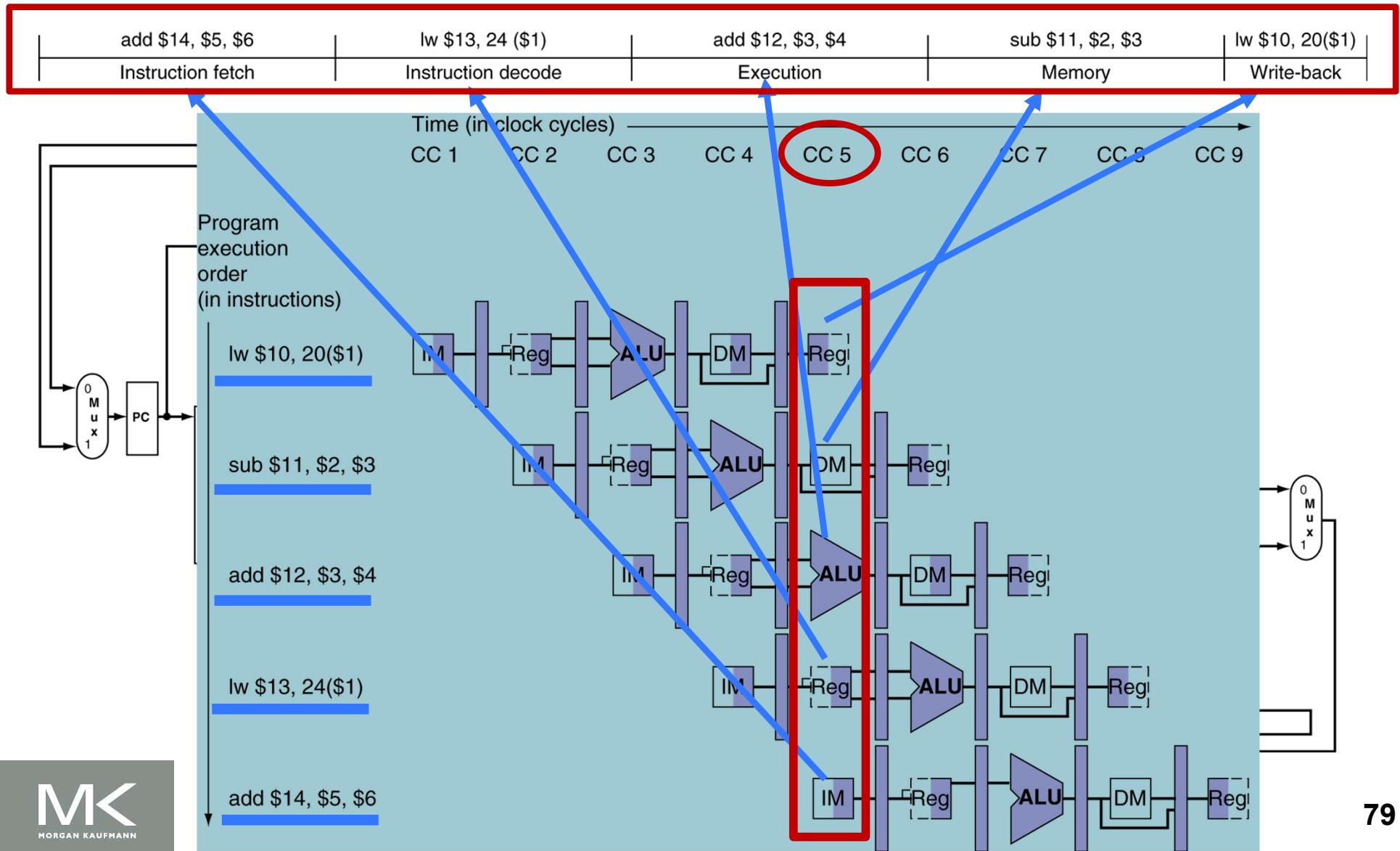
Single-Cycle Pipeline Diagram

State of pipeline in a given cycle

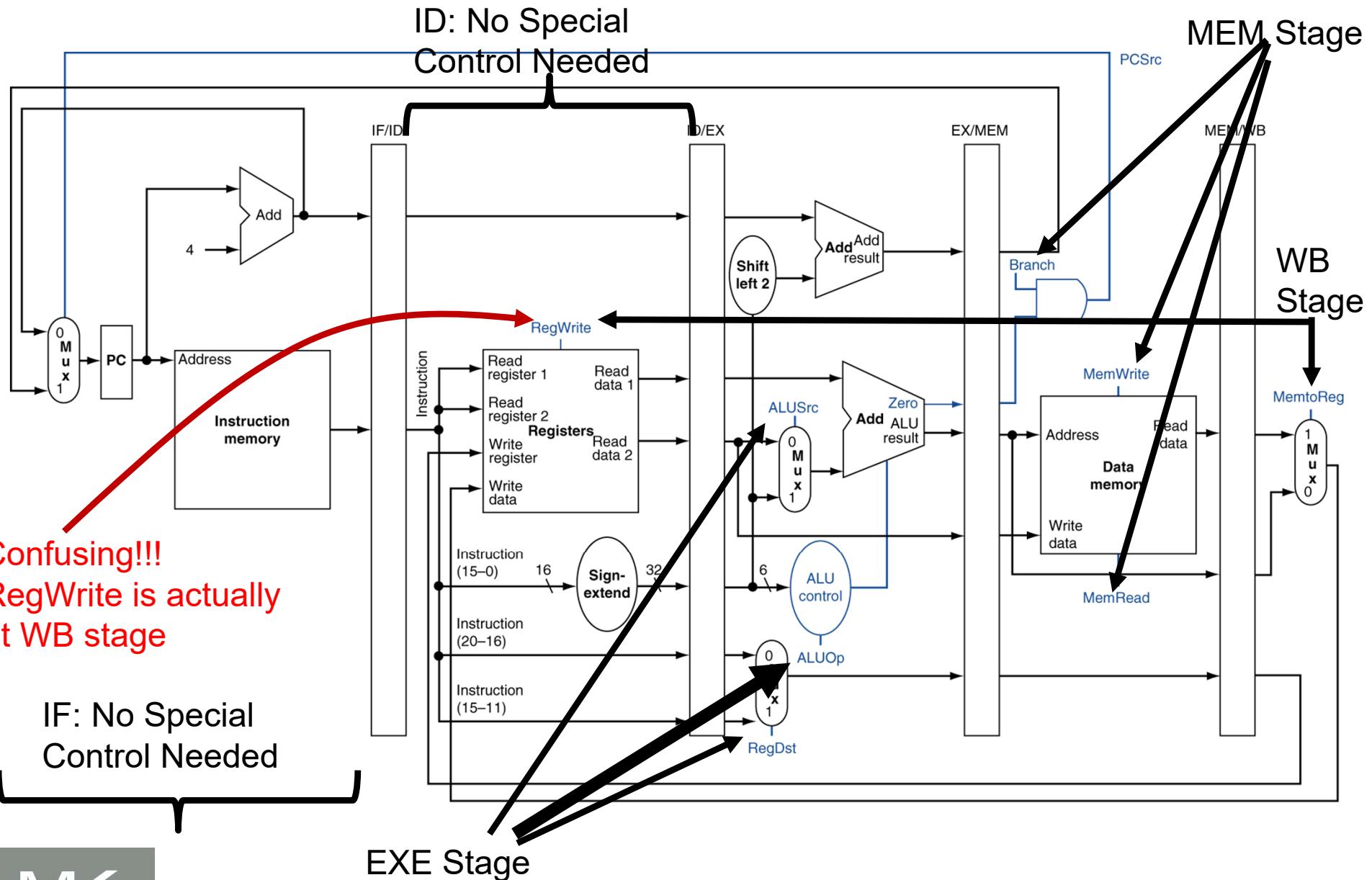


Single-Cycle Pipeline Diagram

State of pipeline in a given cycle

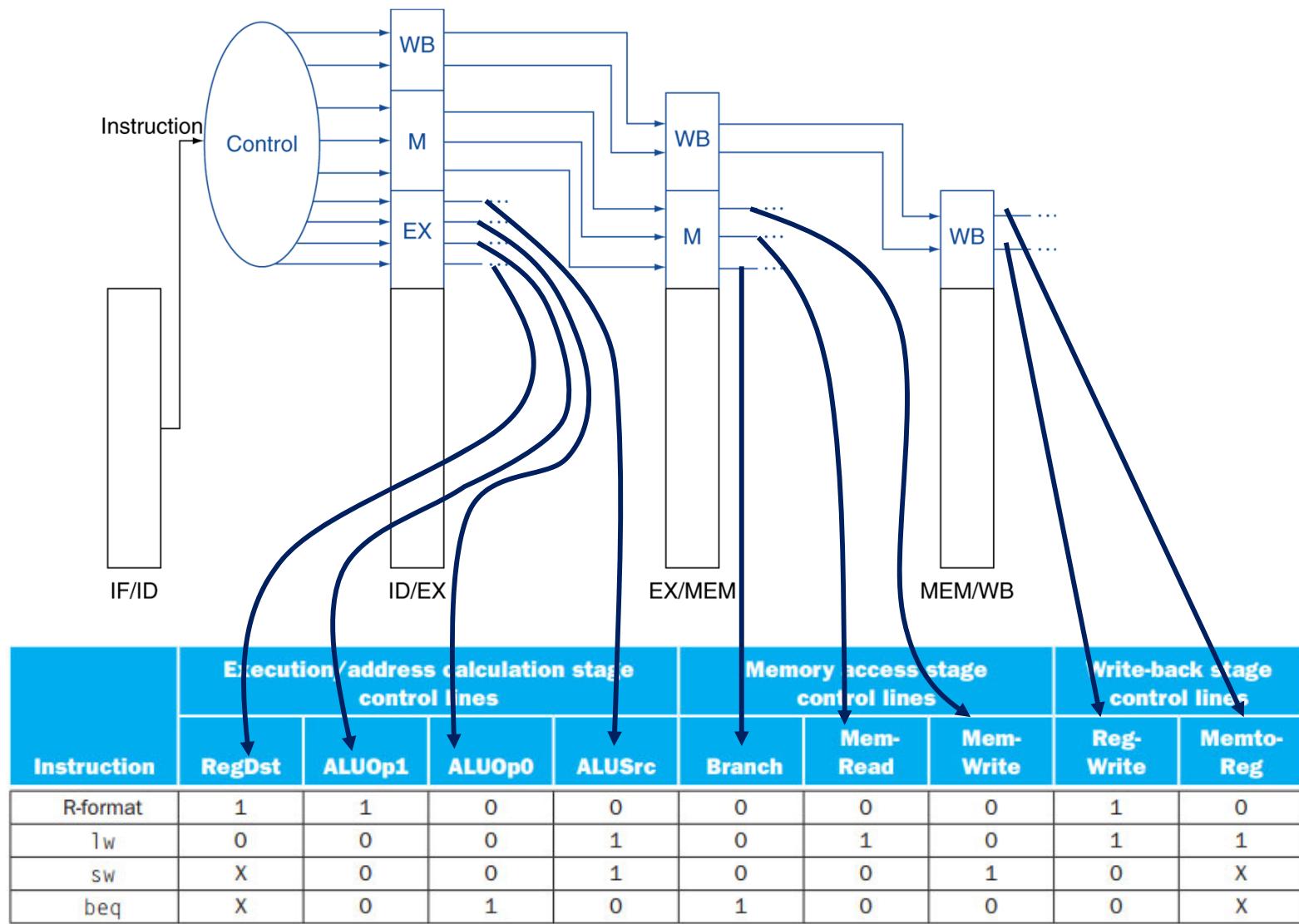


Pipelined Control (Simplified)

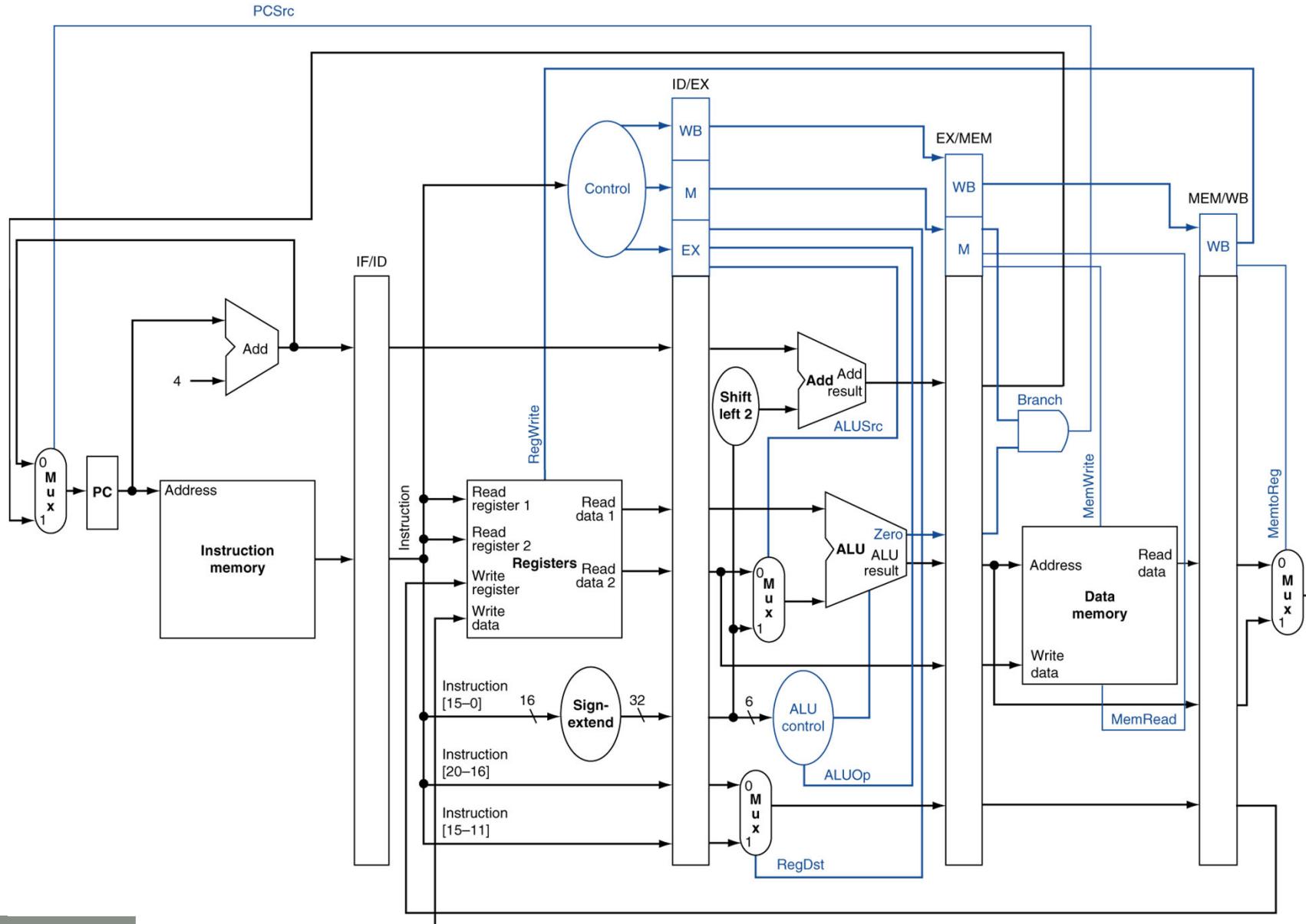


Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation



Pipelined Control



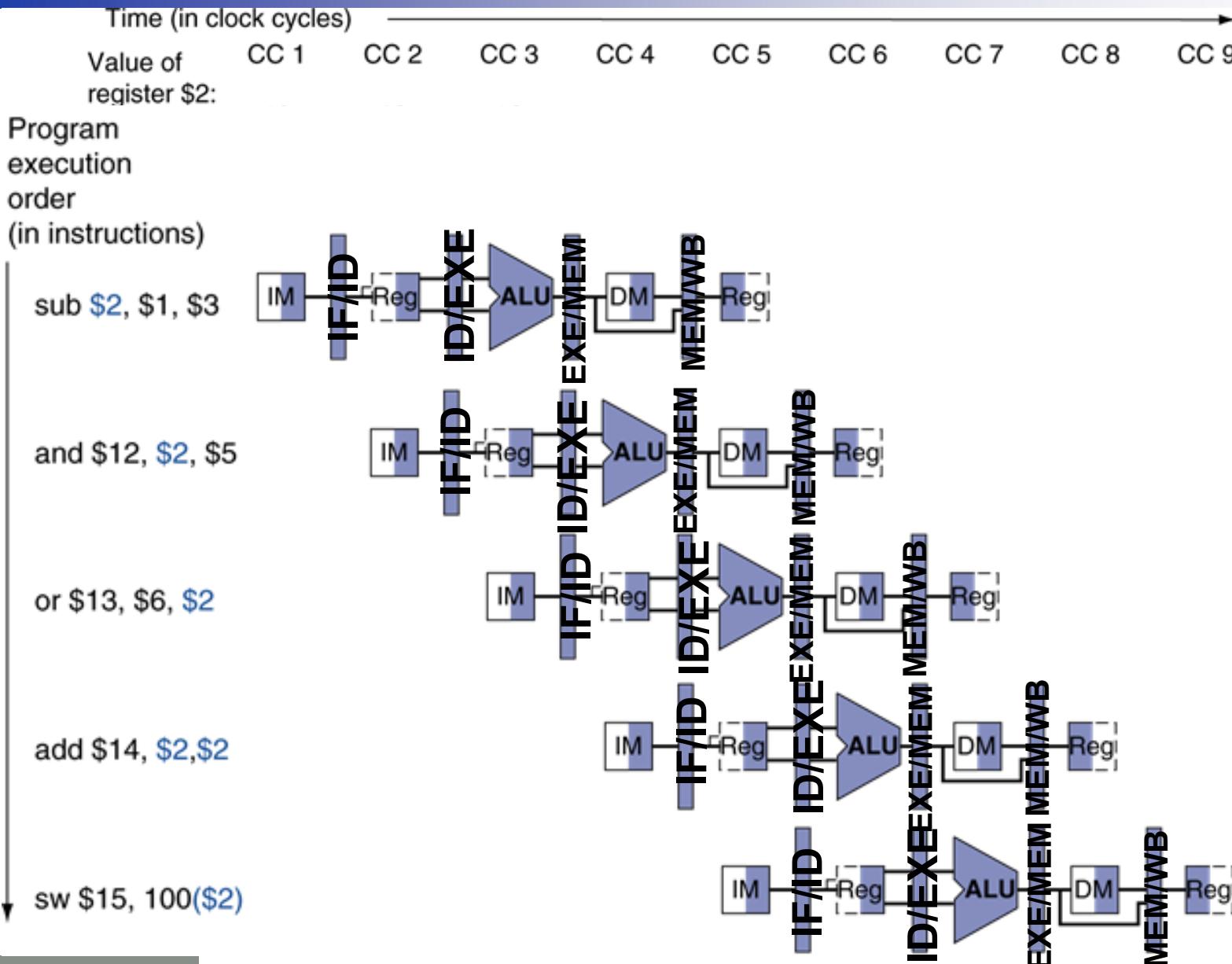
Data Hazards in ALU Instructions

- Consider this sequence:

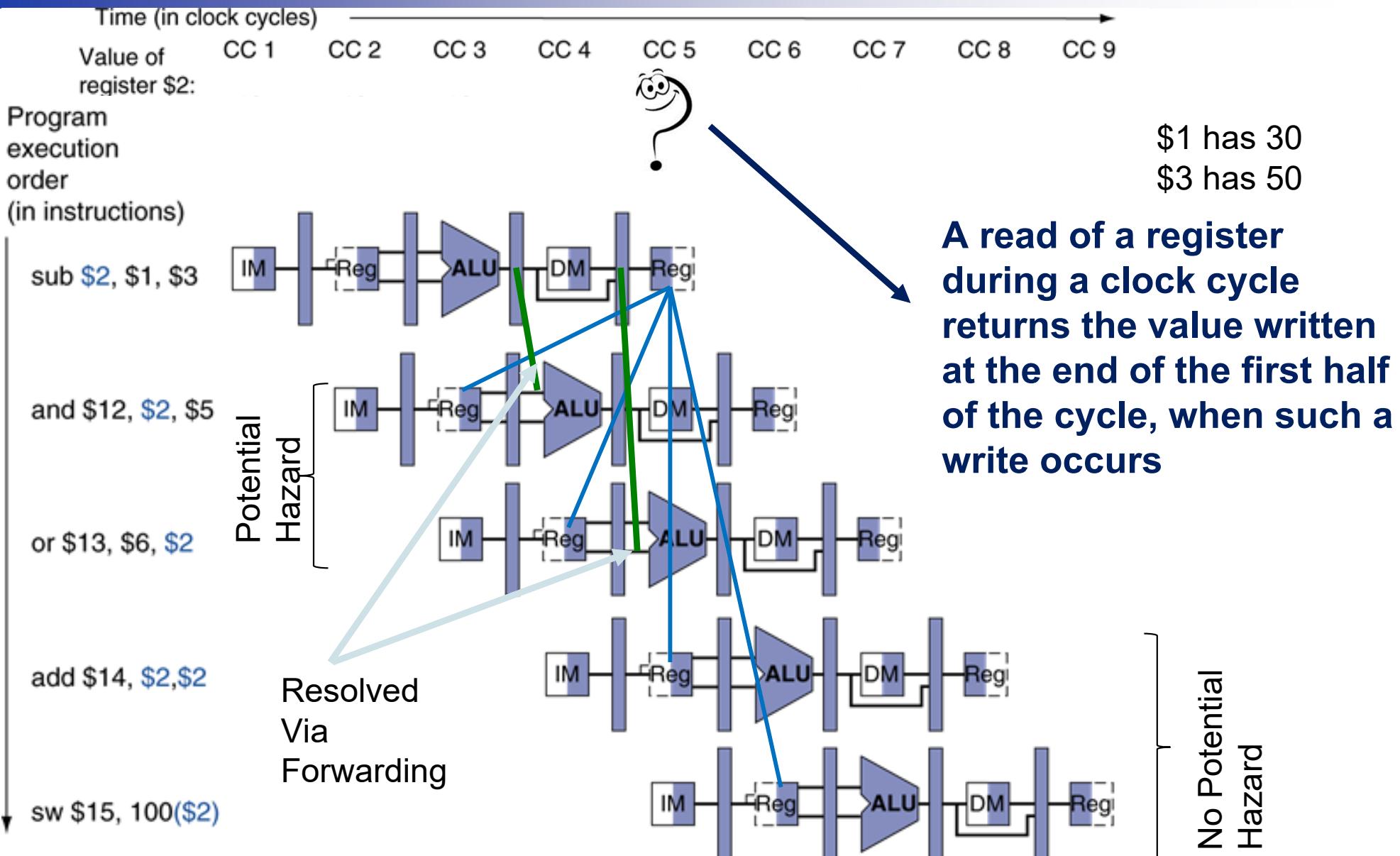
```
sub $2, $1, $3  
and $12, $2, $5  
or $13, $6, $2  
add $14, $2, $2  
sw $15, 100($2)
```

- We can resolve hazards with forwarding
 - How do we detect when to forward?

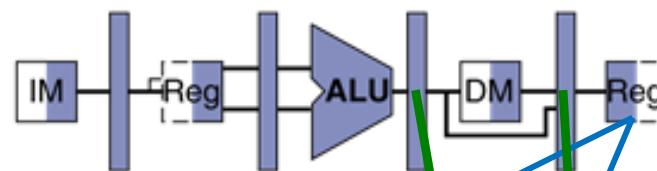
Dependencies & Forwarding



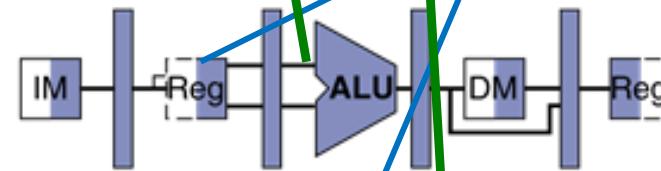
Dependencies & Forwarding



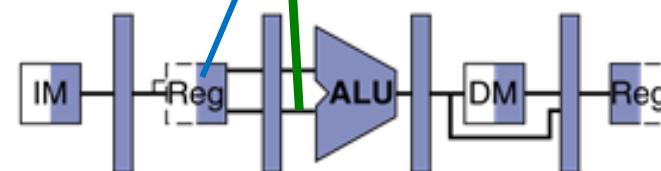
sub \$2, \$1, \$3



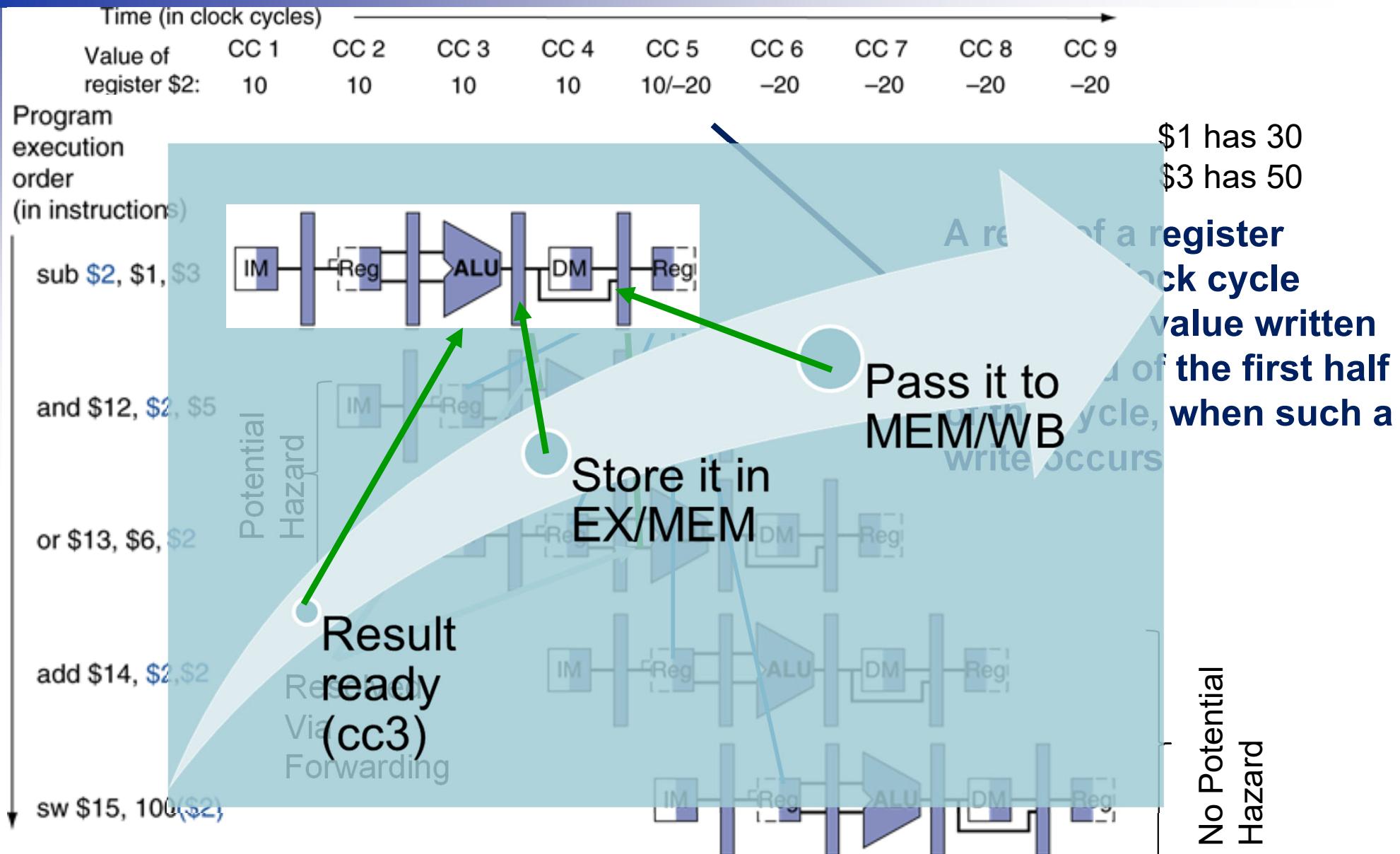
and \$12, \$2, \$5



or \$13, \$6, \$2



Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
 - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
 - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

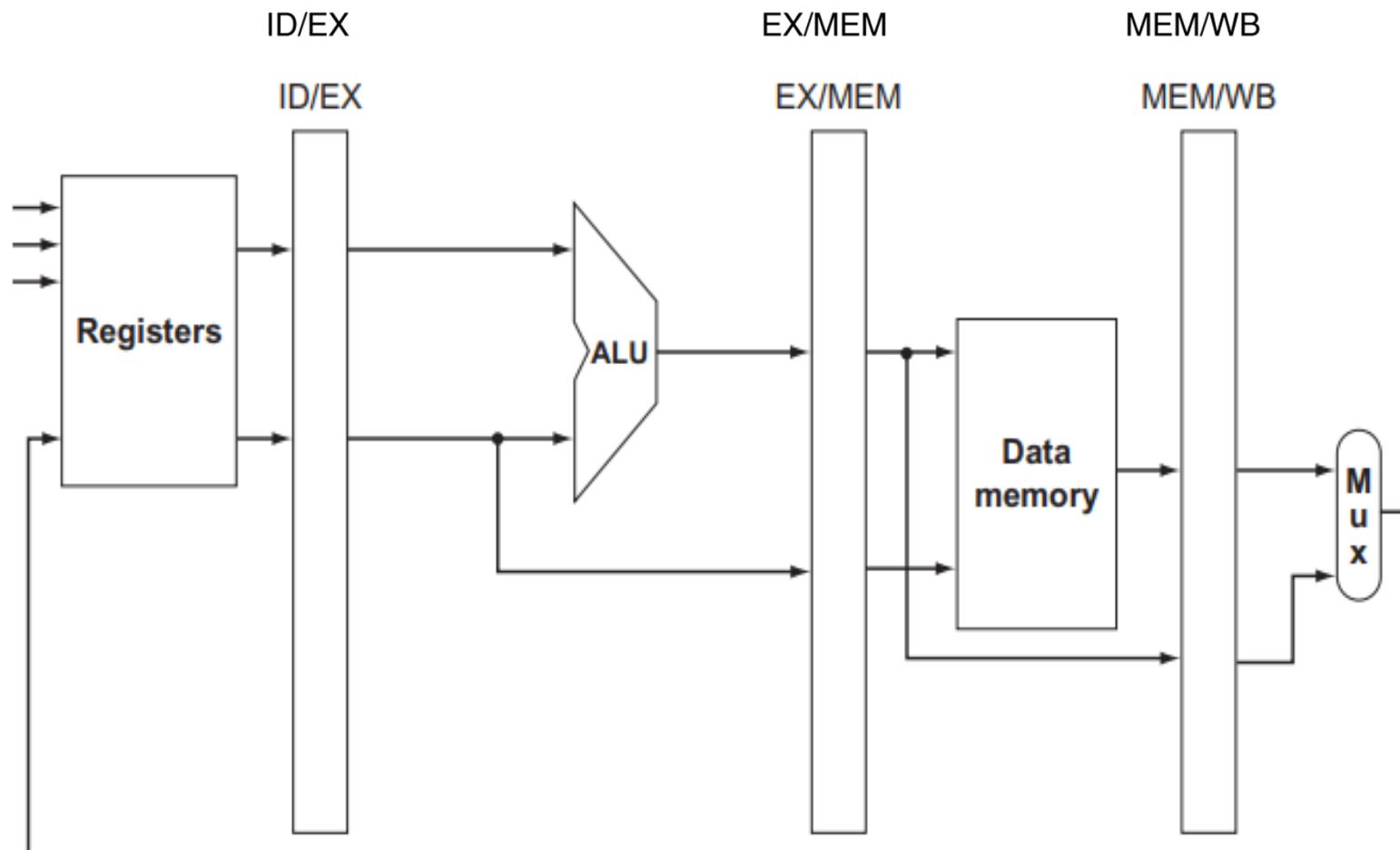
Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

Detecting the Need to Forward

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

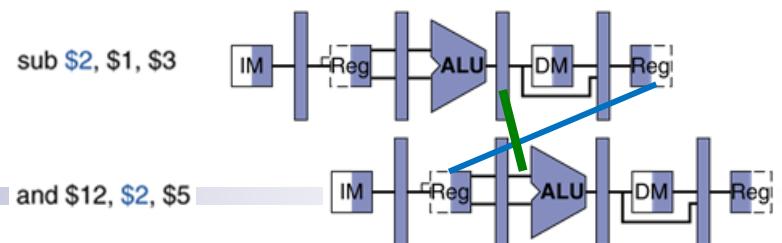
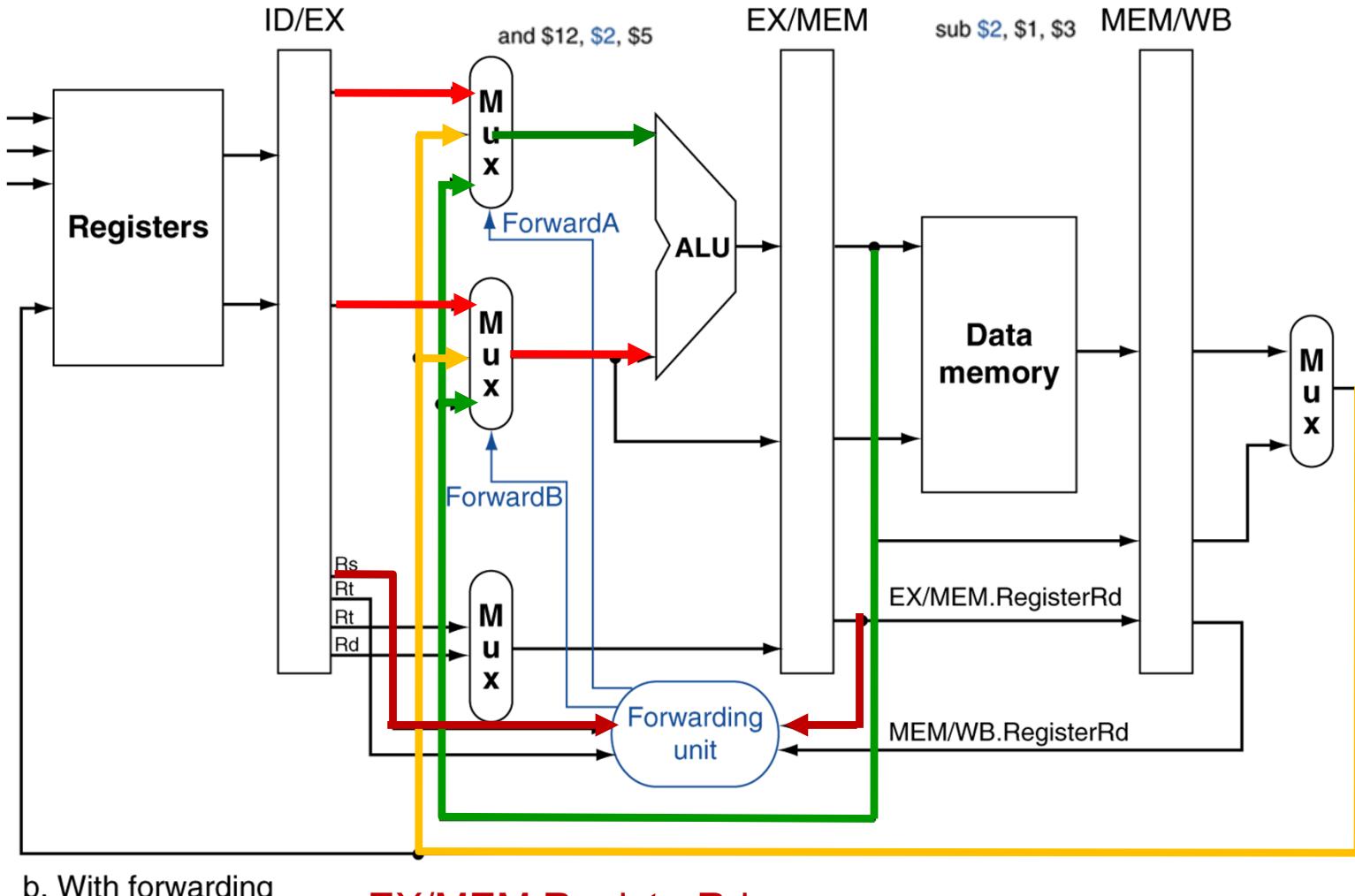
Forwarding Paths



a. No forwarding

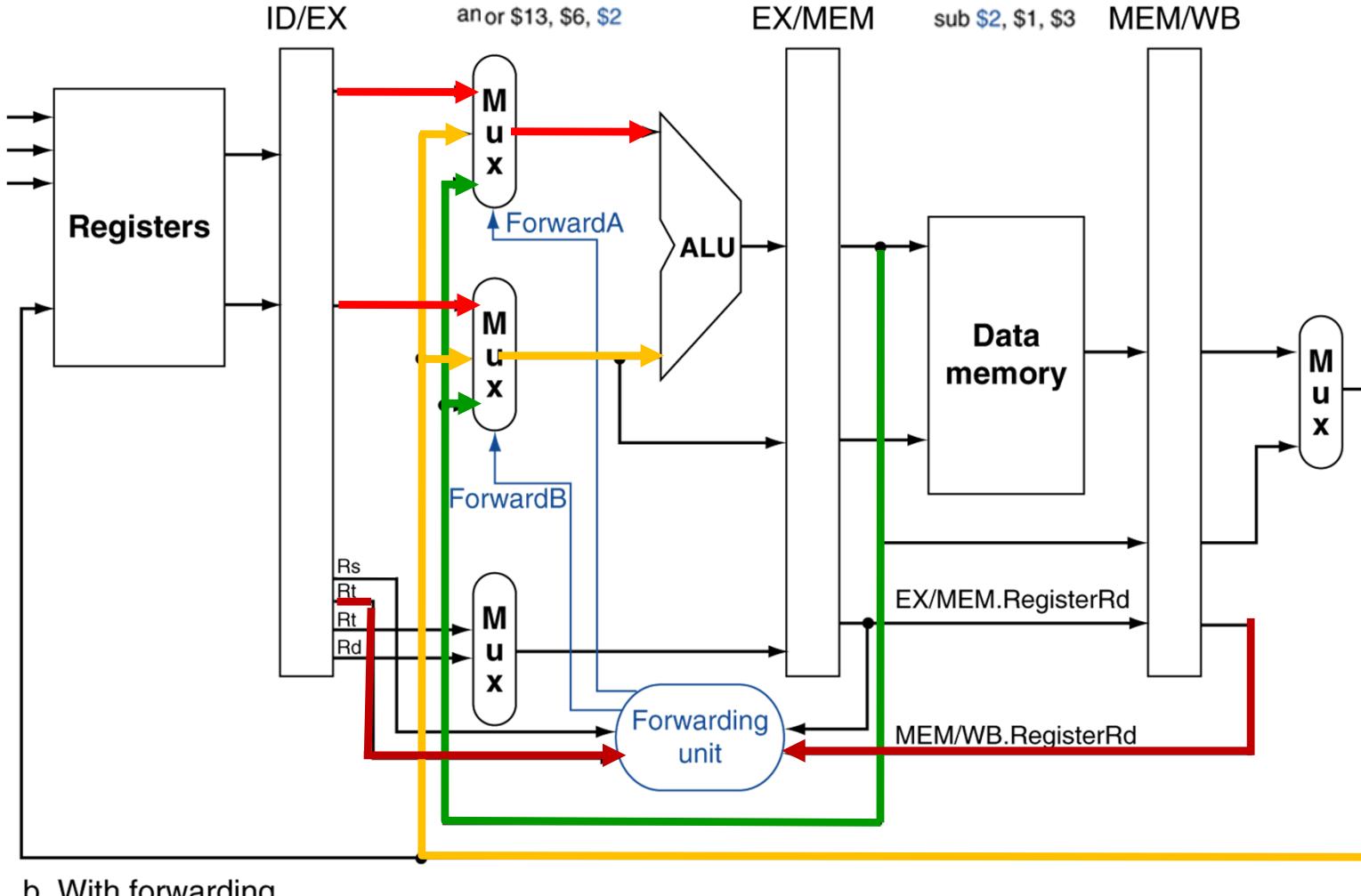
Forwarding Paths

We have not shown regWrite and \$zero checks here.



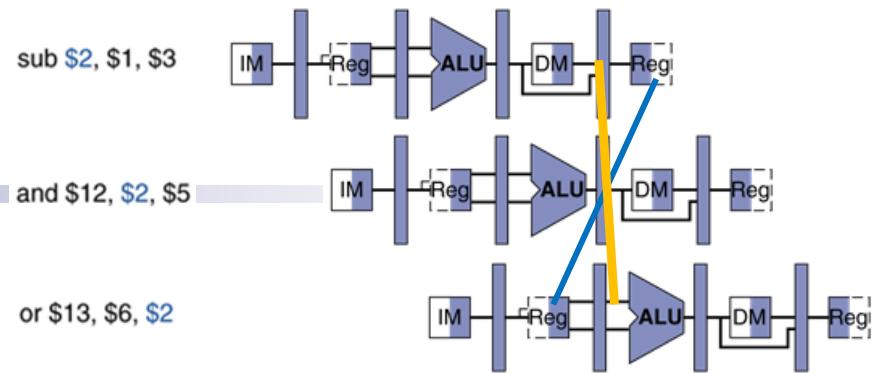
Forwarding Paths

We have not shown regWrite and \$zero checks here.



b. With forwarding

$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}$$



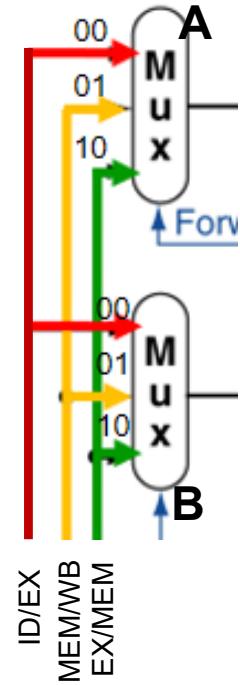
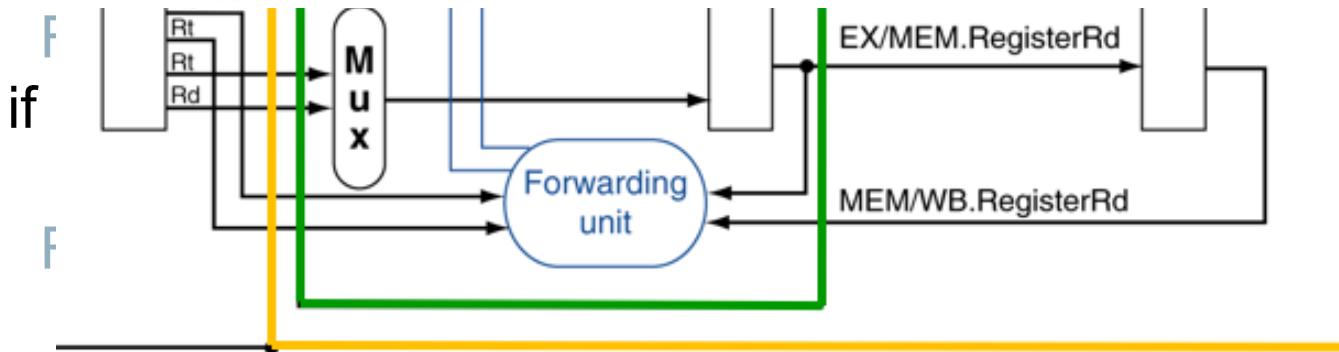
Forwarding Conditions

EX forward

if $F = 1$



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



Double Data Hazard

- Consider the sequence:
 - add \$1, \$1, \$2
 - add \$1, \$1, \$3
 - add \$1, \$1, \$4
- Both hazards occur
 - Want to use the most recent
- Revise MEM hazard condition
 - Only fwd if EX hazard condition isn't true

Double Data Hazard

CC:	cc1	cc2	cc3	cc4	cc5	cc6	cc7
\$2:	30	30	30	30	30	30	30
\$3:	20	20	20	20	20	20	20
\$4:	4	4	4	4	4	4	4
\$1:	10	10	10	10	10/40	4	4

What is expected?

add \$1, \$1, \$2



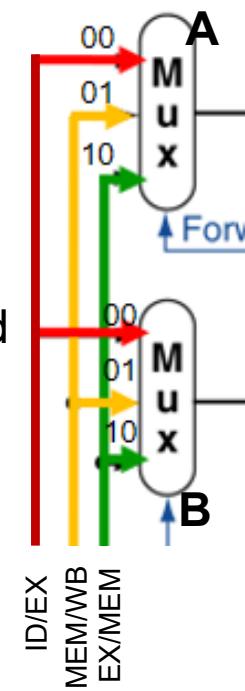
add \$1, \$1, \$3



add \$1, \$1, \$4



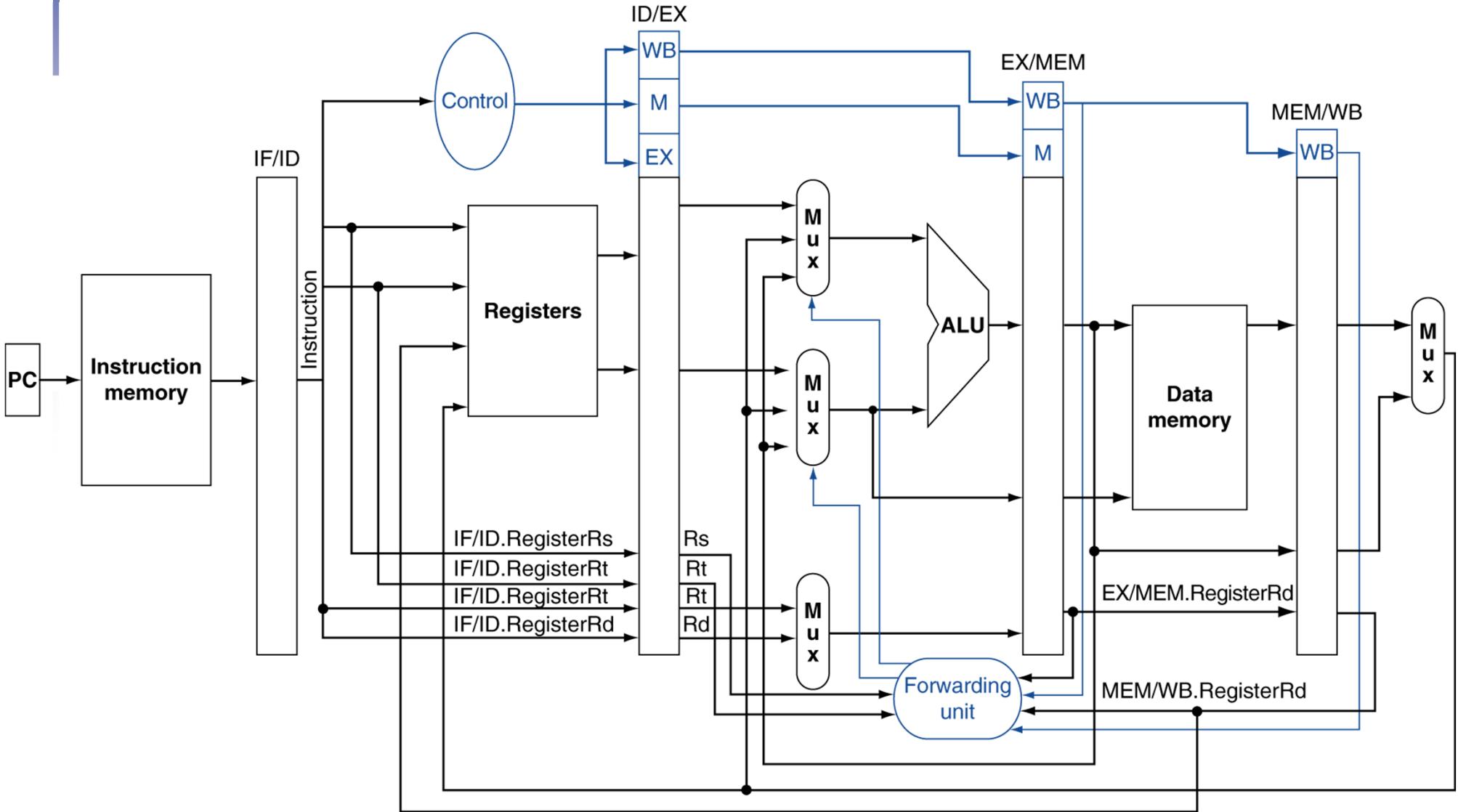
60 is Forwarded
Correct!!!



Revised Forwarding Condition

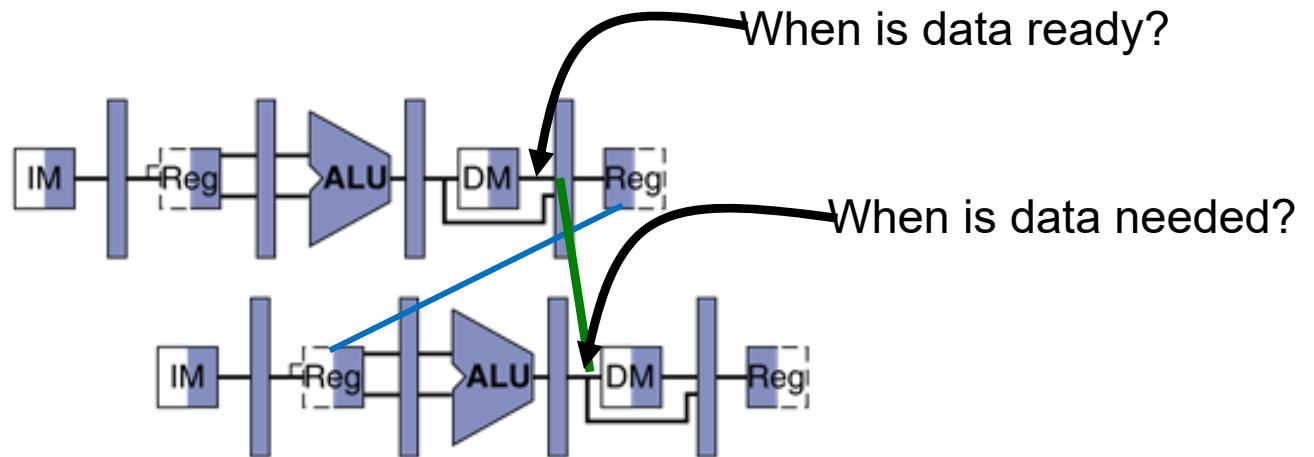
- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

Datapath with Forwarding

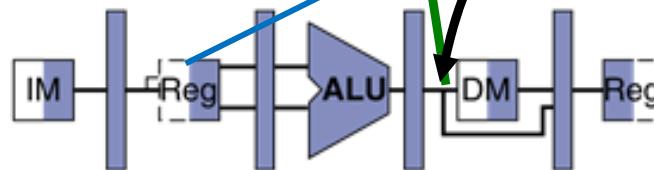


Load -> Store (Mem to Mem Copy)

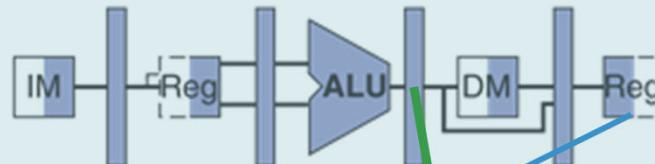
lw \$2, 20(\$1)



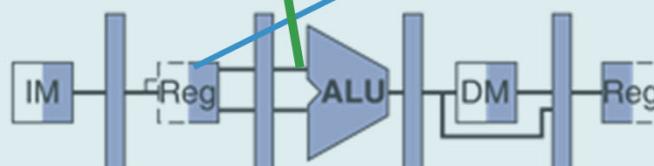
sw \$2, 15(\$3)



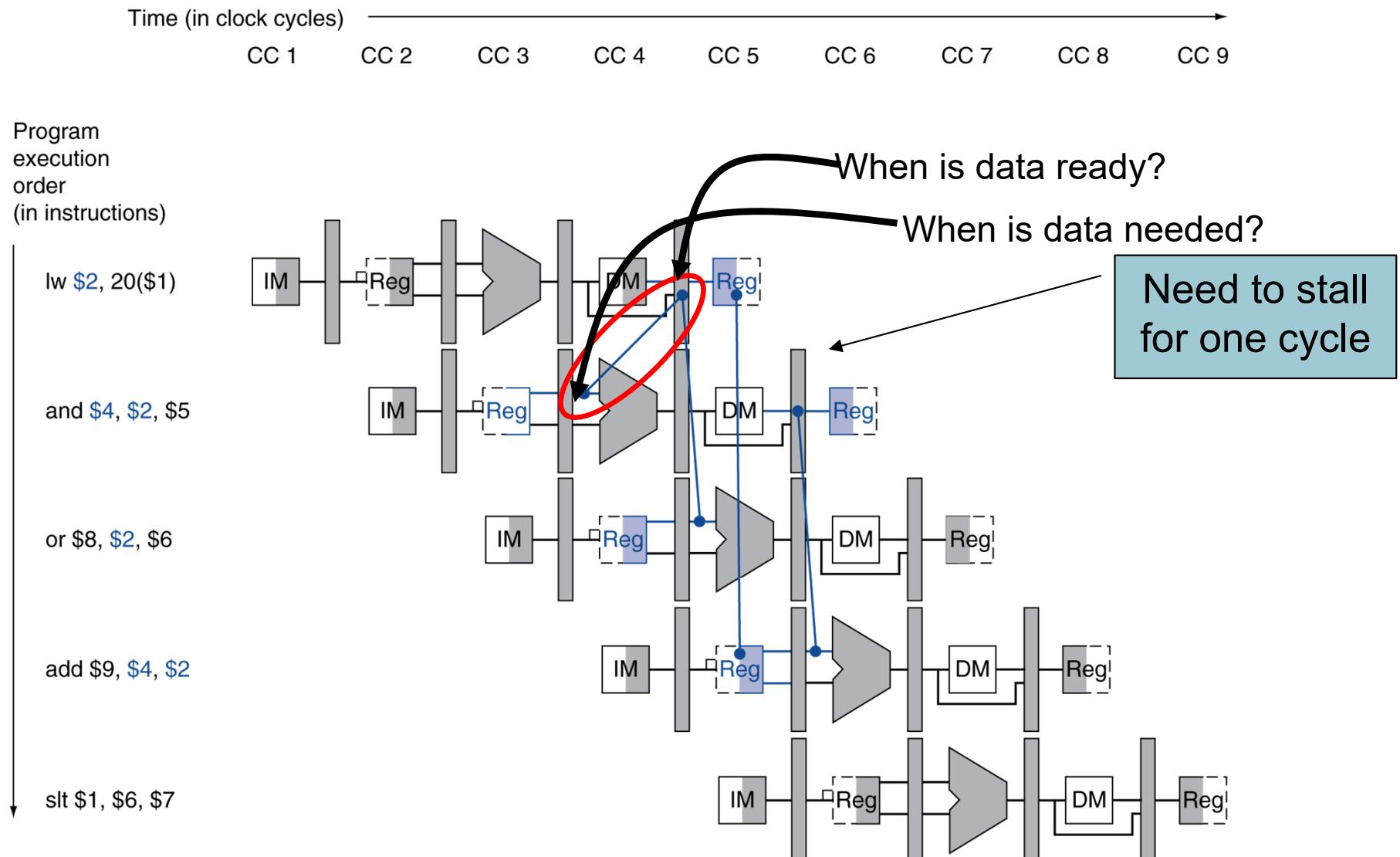
sub \$2, \$1, \$3



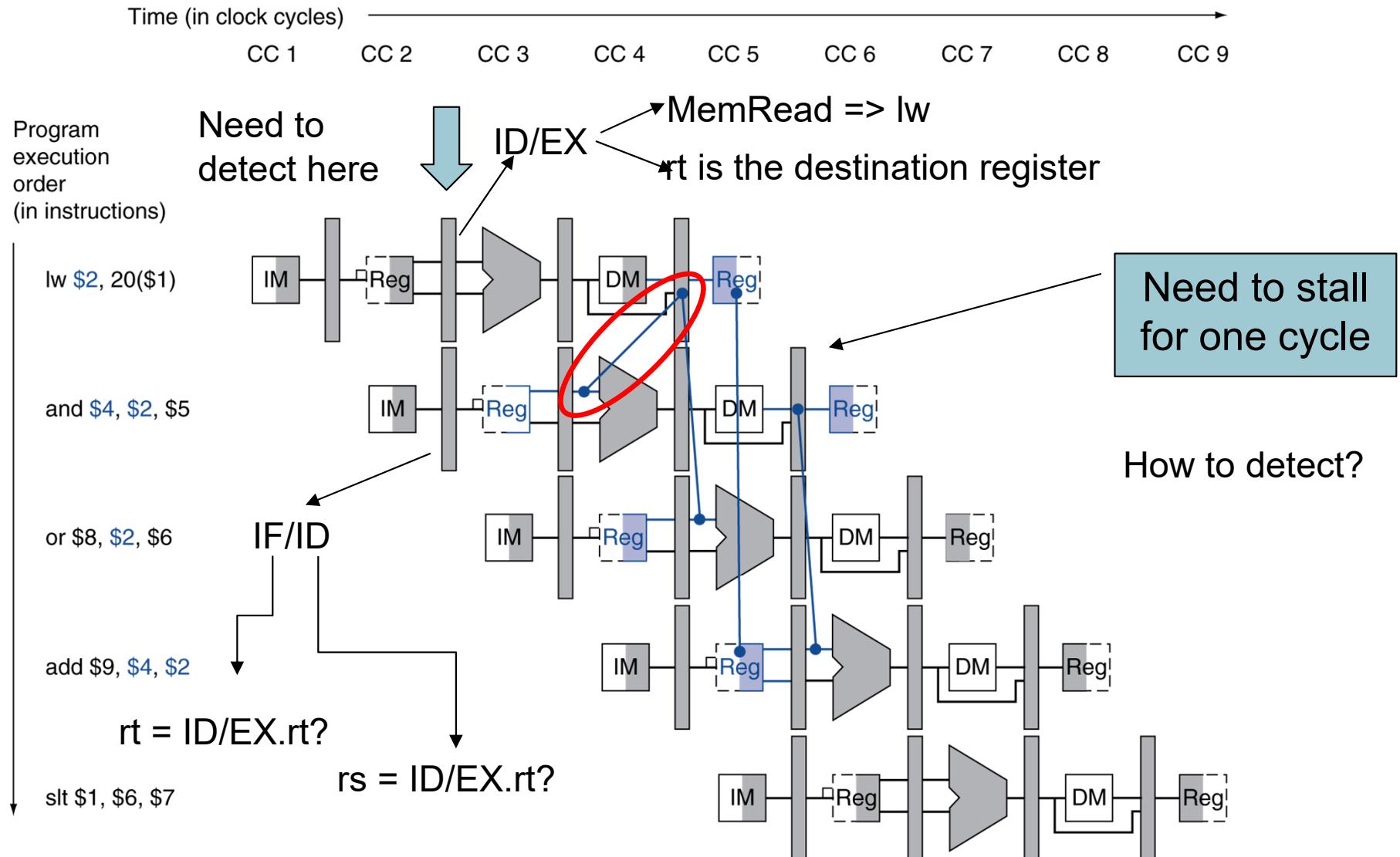
and \$12, \$2, \$5



Load-Use Data Hazard



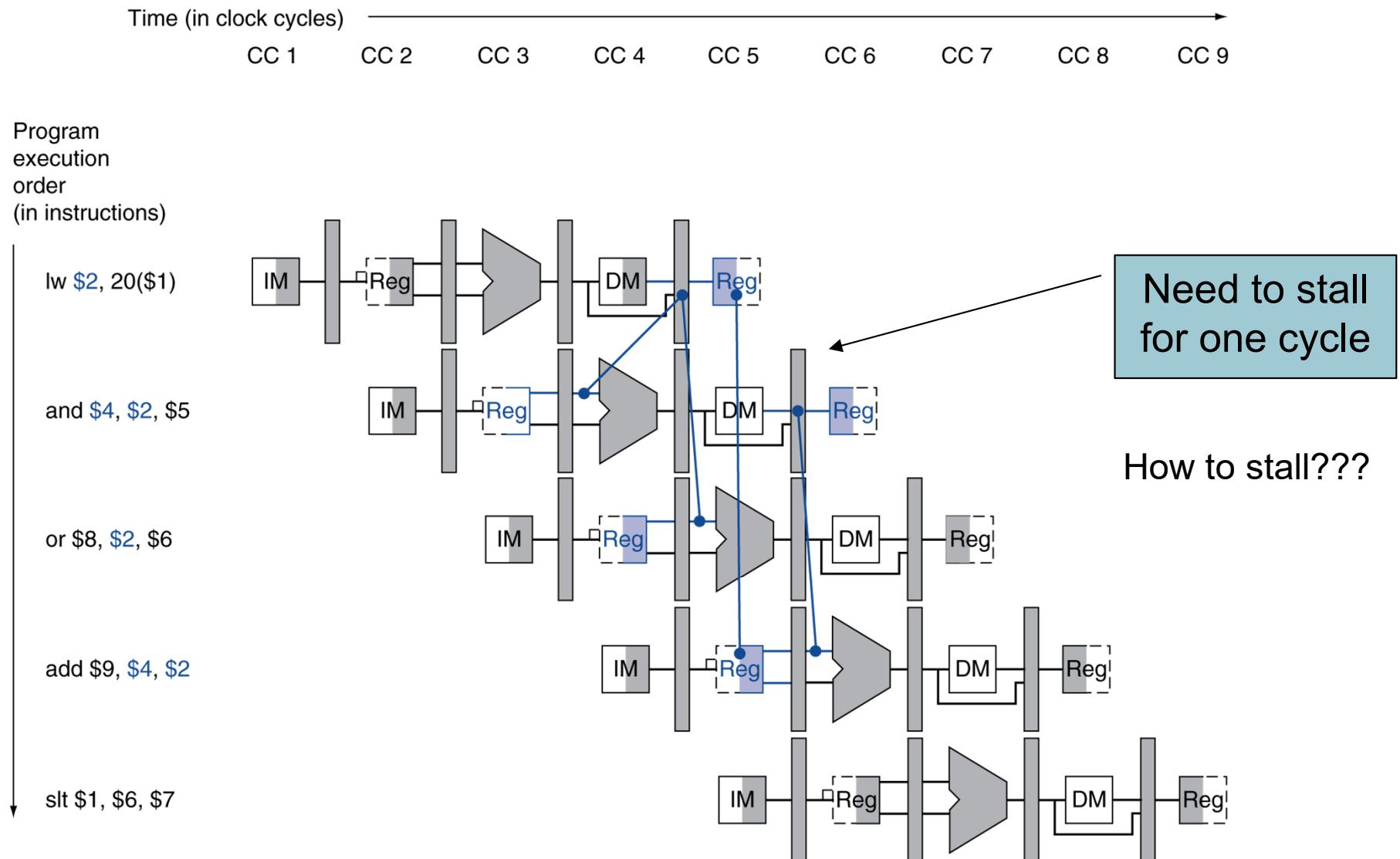
Load-Use Data Hazard



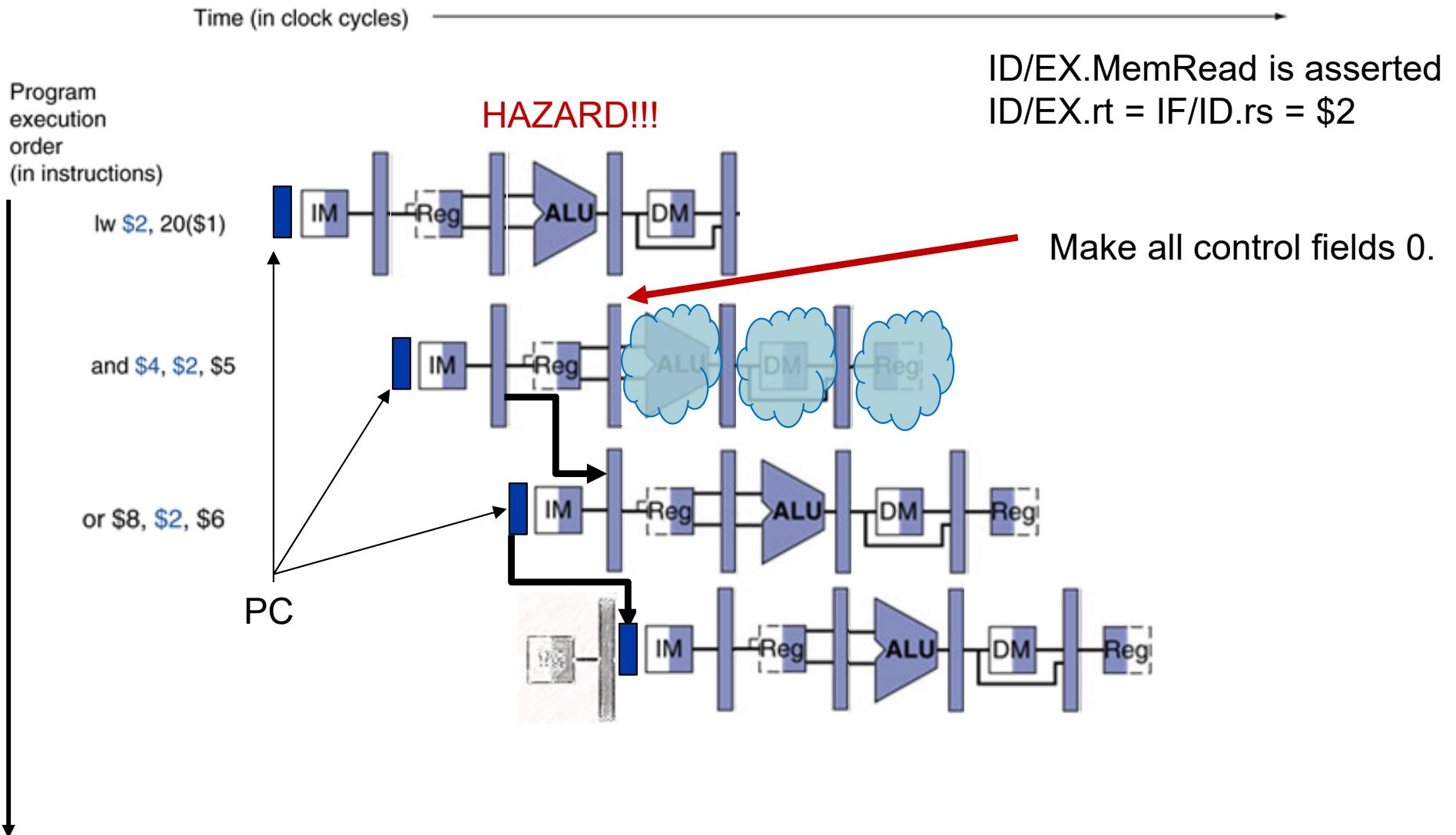
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
 - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
 - ID/EX.MemRead and
$$((ID/EX.RegisterRt = IF/ID.RegisterRs) \text{ or } (ID/EX.RegisterRt = IF/ID.RegisterRt))$$
- If detected, stall and insert bubble

Load-Use Data Hazard



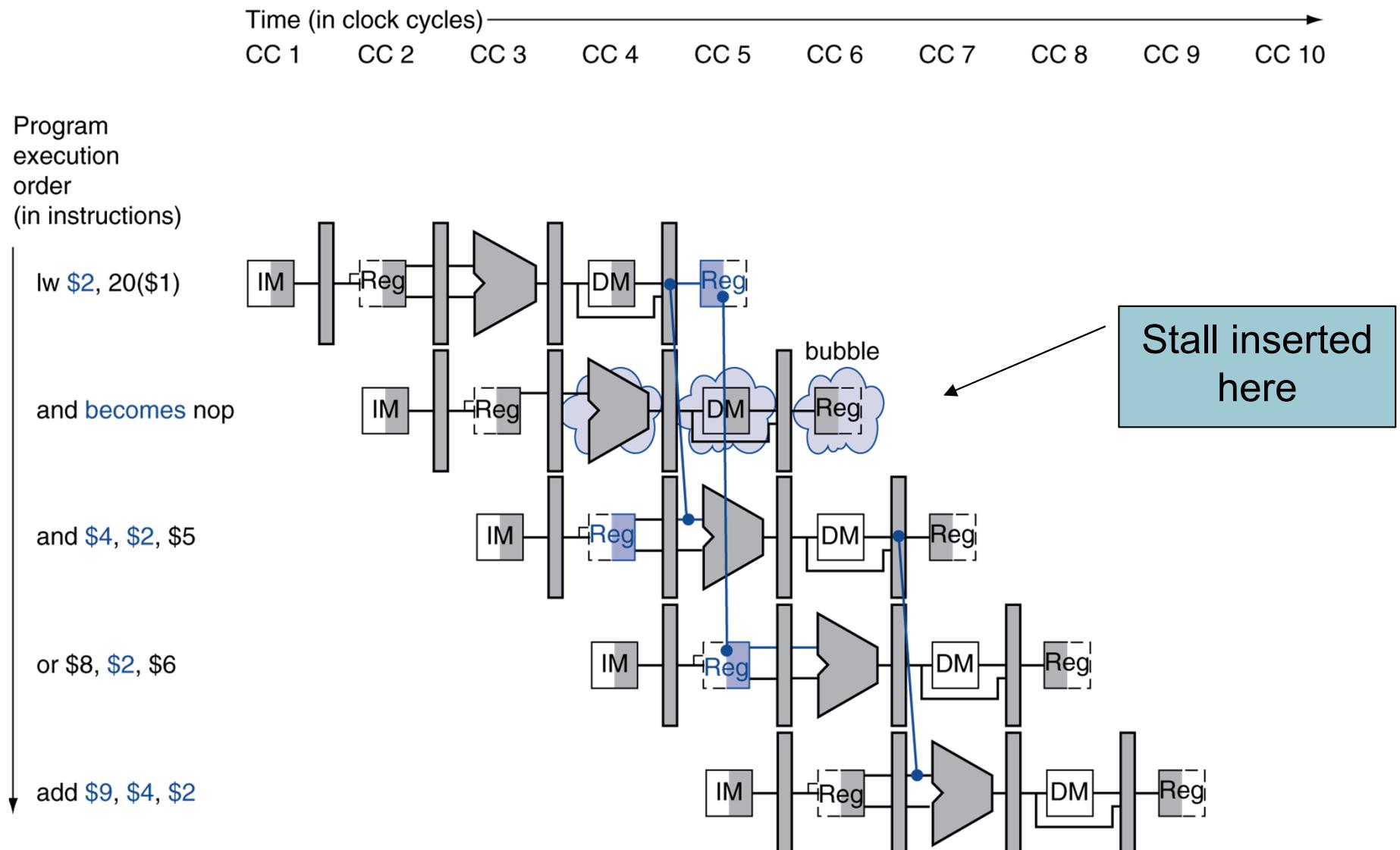
How to Stall



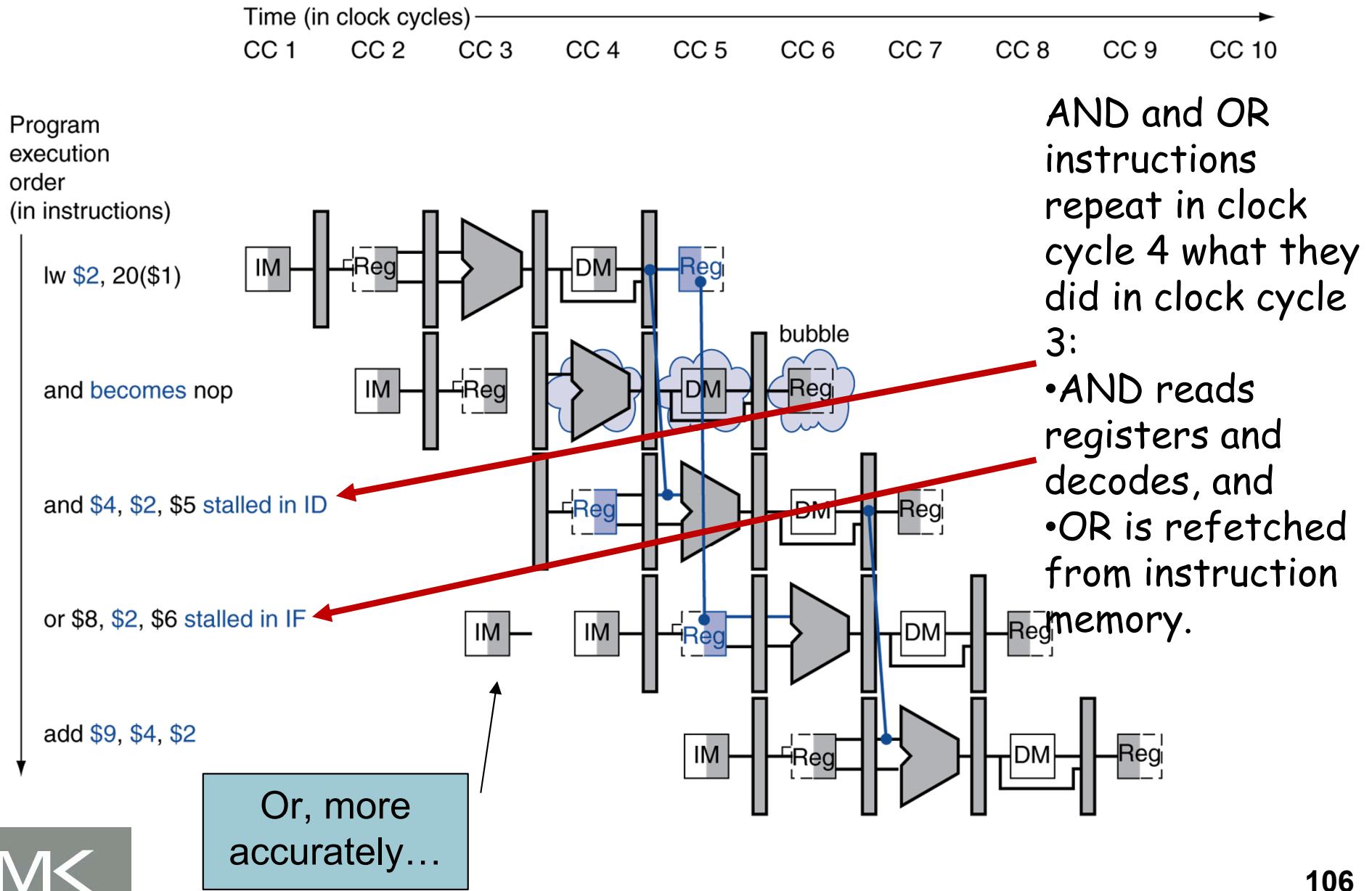
How to Stall the Pipeline

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

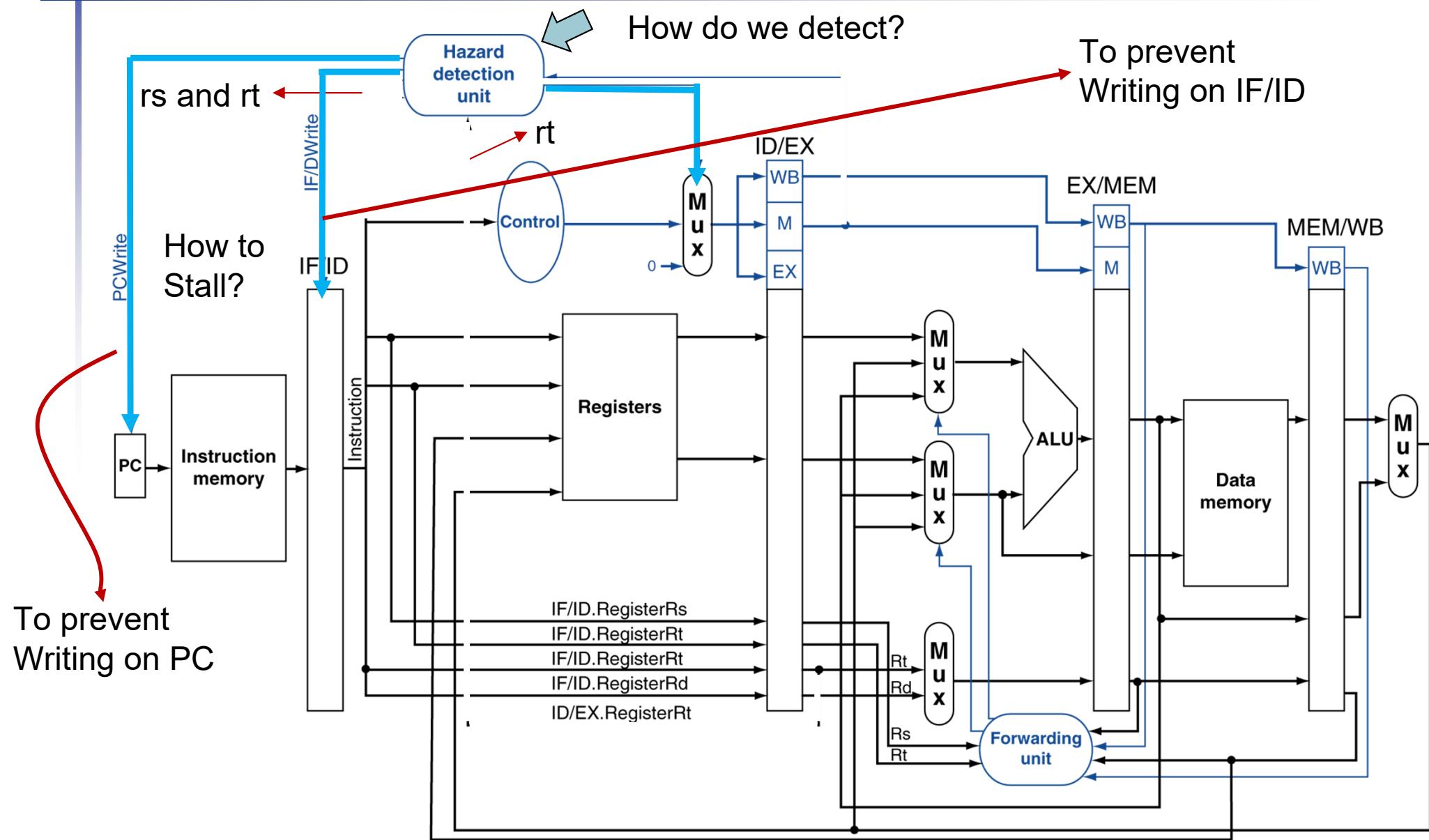
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



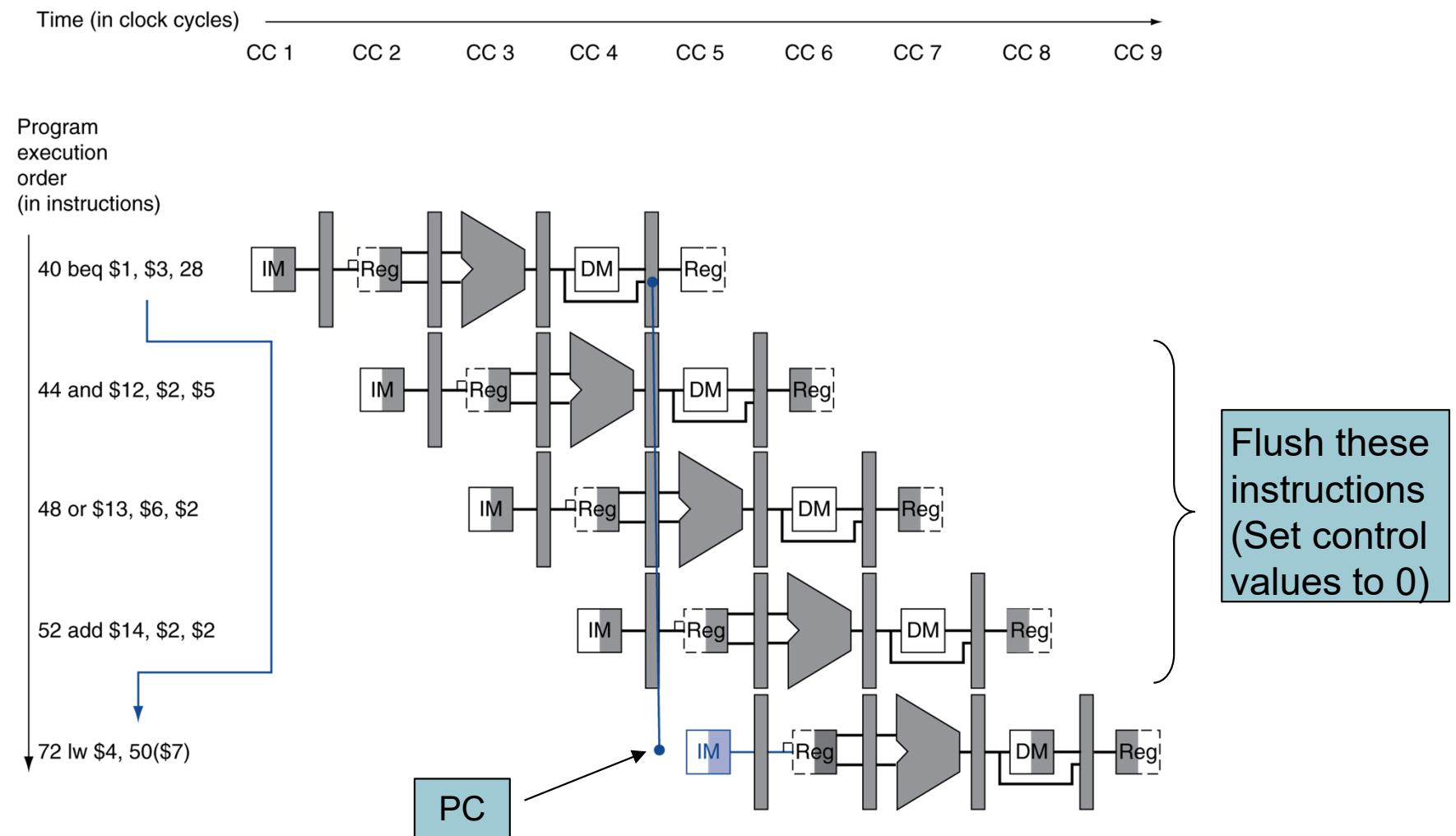
Stalls and Performance

The BIG Picture

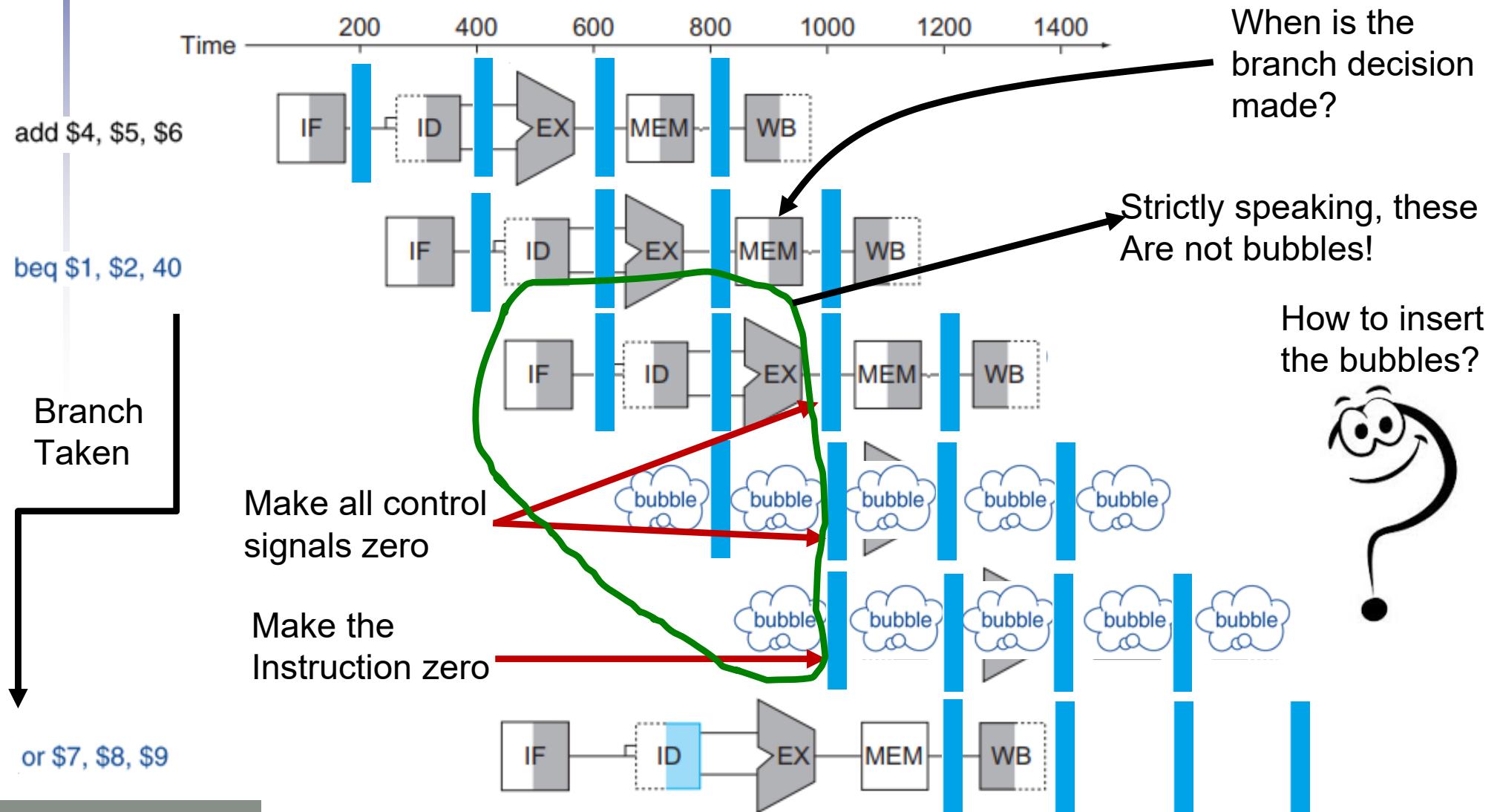
- Stalls reduce performance
 - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
 - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM



Branch Scenario (RECALLING)

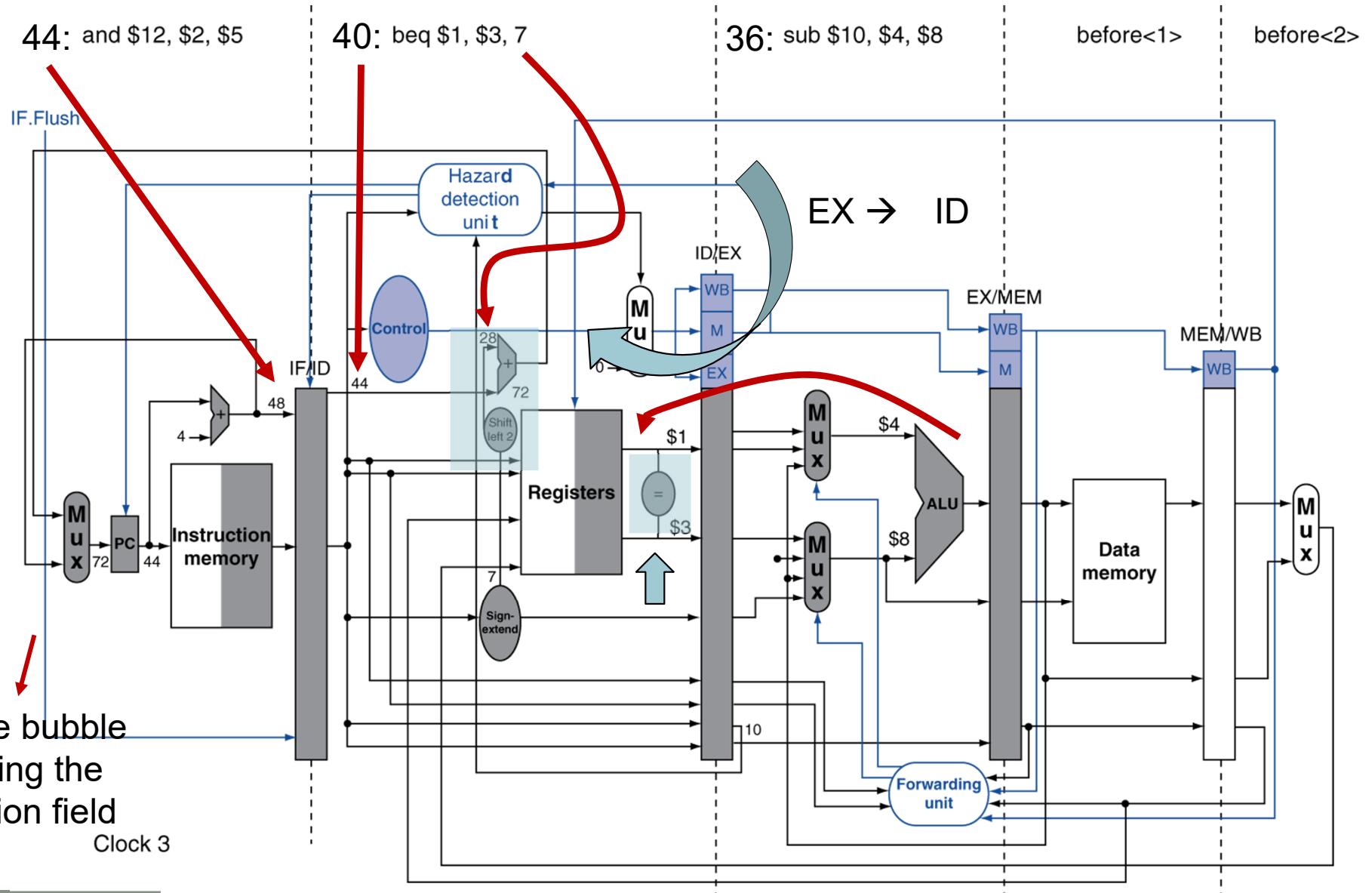


Reducing Branch Delay

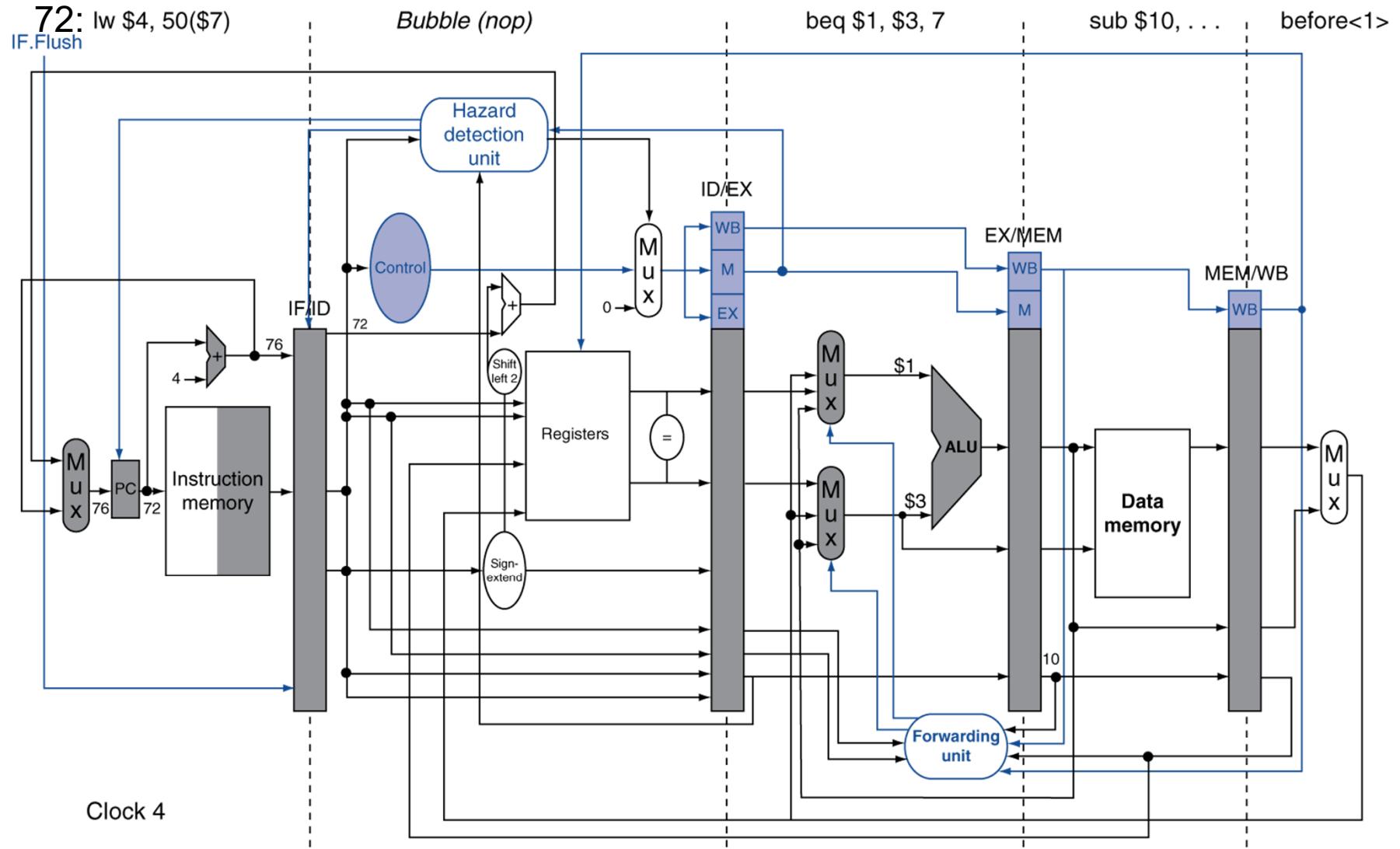
- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken

```
36:    sub   $10, $4, $8  
40:    beq   $1,  $3,  7  
44:    and   $12, $2, $5  
48:    or    $13, $2, $6  
52:    add   $14, $4, $2  
56:    slt   $15, $6, $7  
      ...  
72:    lw    $4, 50($7)
```

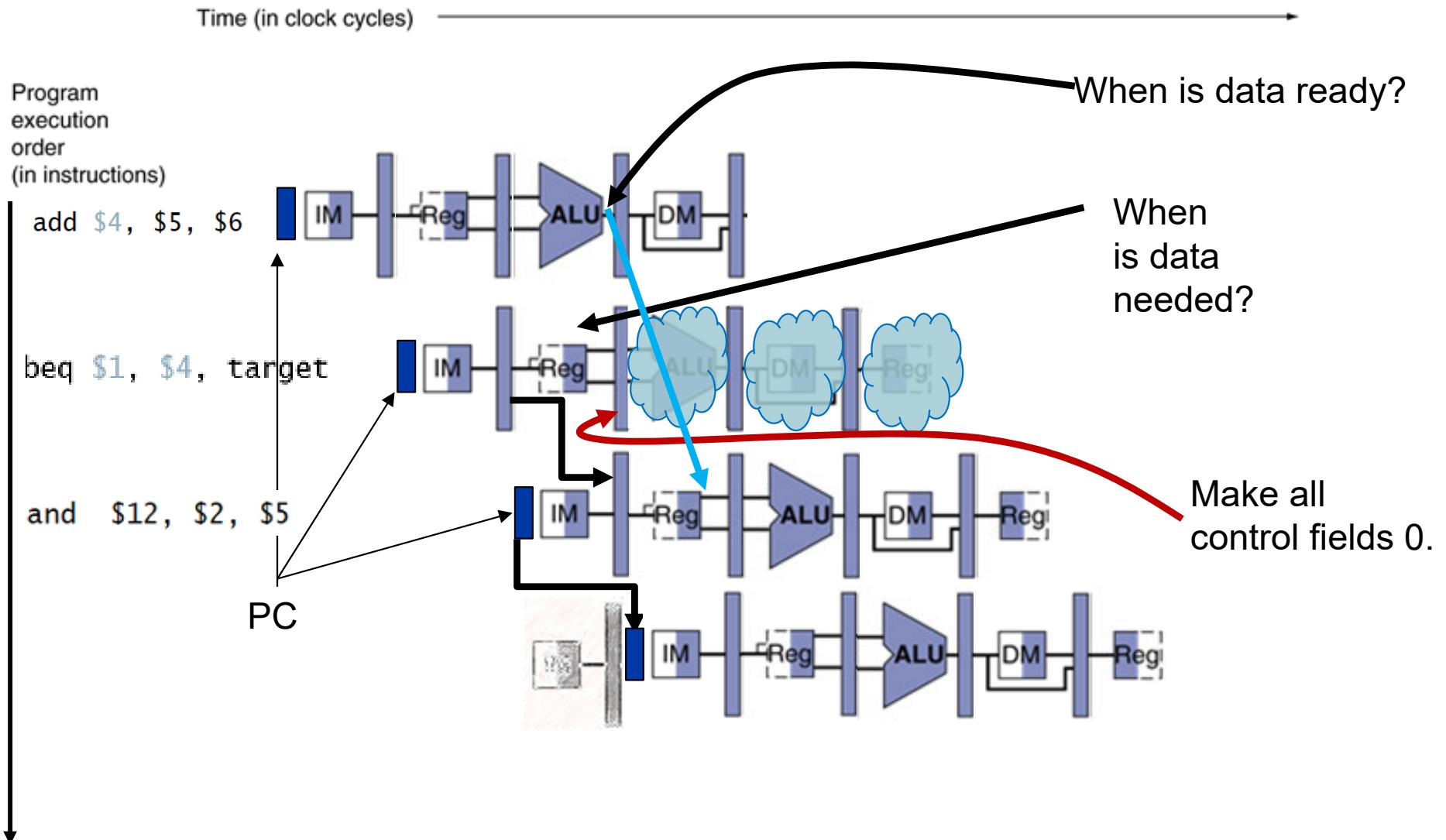
Example: Branch Taken



Example: Branch Taken

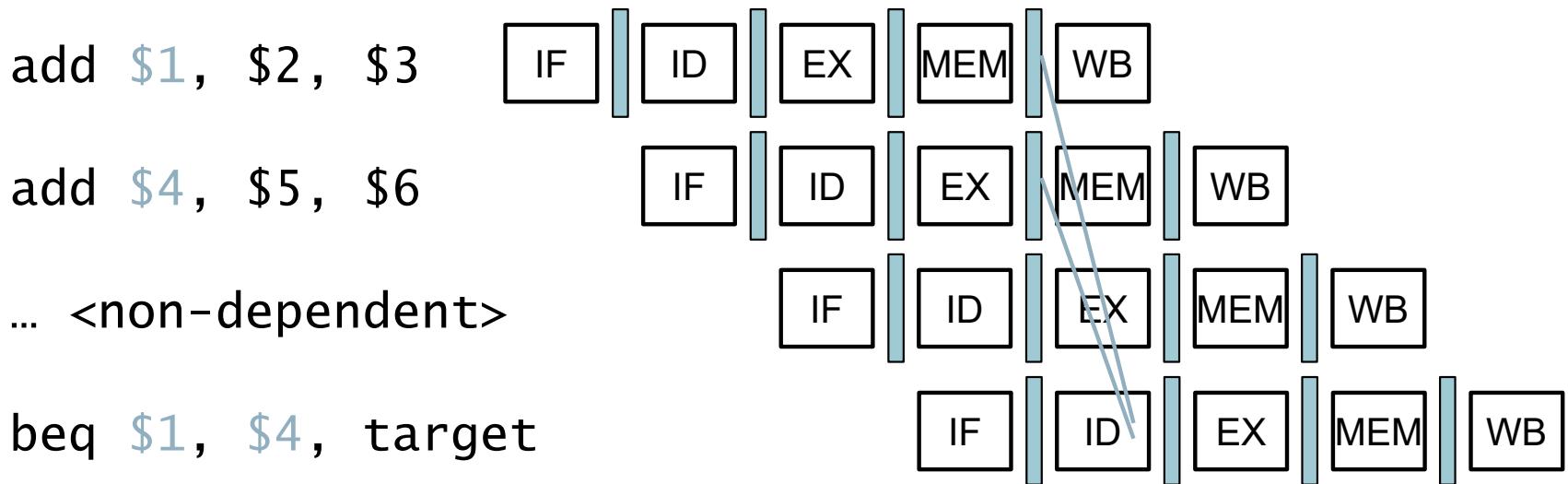


Data Hazards for Branches



Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- Can resolve using forwarding

Data Hazards for Branches

lw \$1, addr

beq \$1, \$4, target

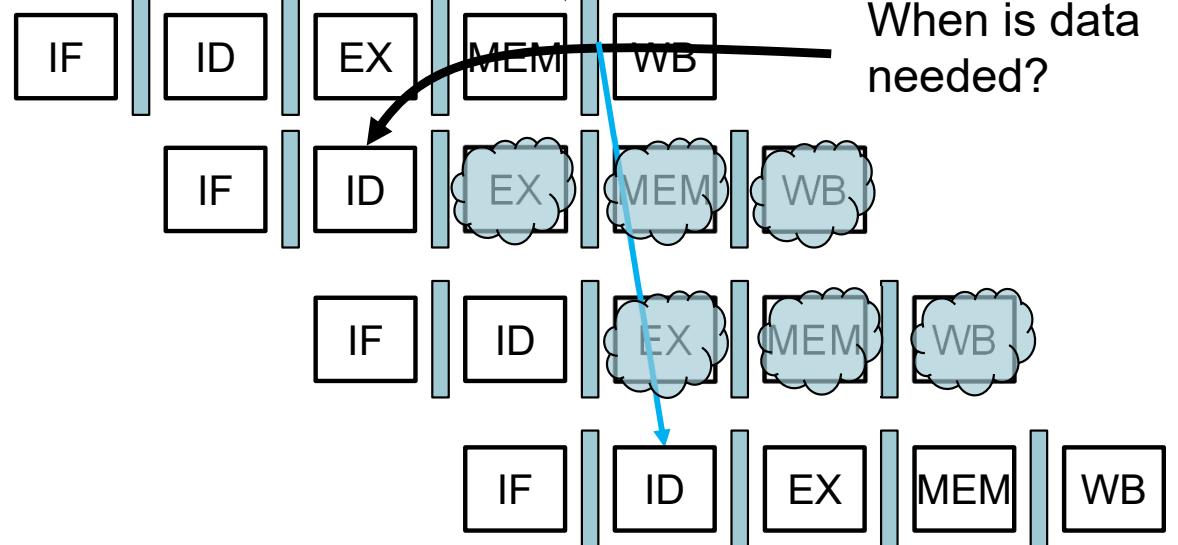
beq stalled

IF.Flush

All controls
made zero

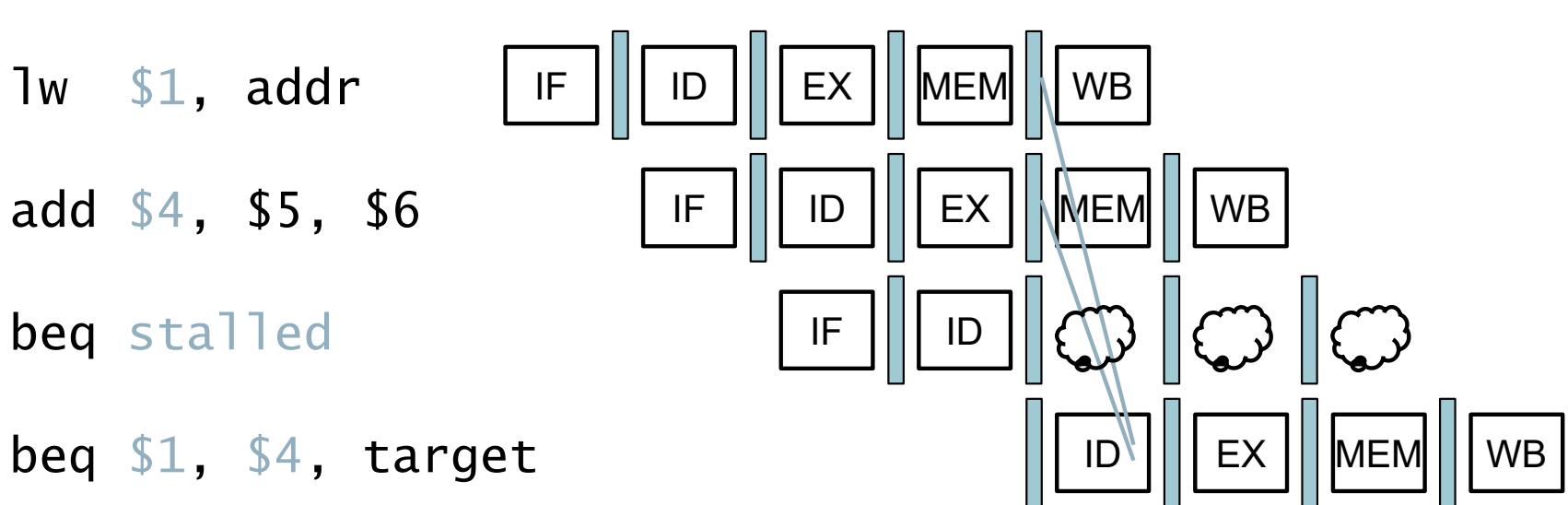
When is data ready?

When is data
needed?



Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
 - Need 1 stall cycle



Dynamic Branch Prediction

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

Calculating the Branch Target

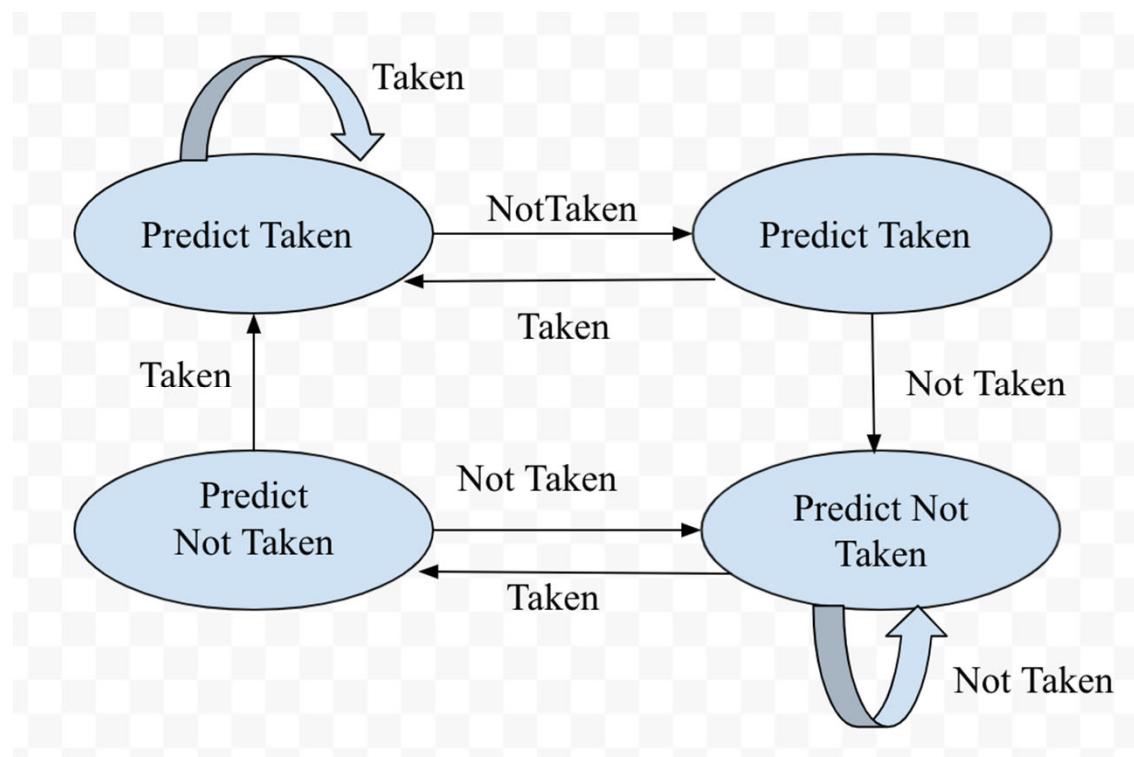
- Even with predictor, still need to calculate the target address
 - 1-cycle penalty for a taken branch
- Branch target buffer
 - Cache of target addresses
 - Indexed by PC when instruction fetched
 - If hit and instruction is branch predicted taken, can fetch target immediately

1-Bit Predictor: Shortcoming

- Consider a loop branch:
 - branches nine times in a row
 - then is not taken once.
 - What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?
 - Mispredict as taken on last iteration of inner loop
 - Then mispredict as not taken on first iteration of inner loop next time around
- 90% time
Taken**
- 80% time
Accurate**

2-Bit Predictor

- Only change prediction on two successive mispredictions



a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once.

Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
 - Different ISAs use the terms differently
- Exception
 - Arises within the CPU
 - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
 - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
 - Deeper pipeline
 - Less work per stage ®shorter clock cycle
 - Multiple issue
 - Replicate pipeline stages ®multiple pipelines
 - Start multiple instructions per clock cycle
 - CPI < 1, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak CPI = 0.25, peak IPC = 4
 - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
 - Compiler groups instructions to be issued together
 - Packages them into “issue slots”
 - Compiler detects and avoids hazards
- Dynamic multiple issue
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - Compiler can help by reordering instructions
 - CPU resolves hazards using advanced techniques at runtime

Speculation

- “Guess” what to do with an instruction
 - Start operation as soon as possible
 - Check whether guess was right
 - If so, complete the operation
 - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
 - Speculate on branch outcome
 - Roll back if path taken is different
 - Speculate on load followed by a store (whether the load is dependent on the store)
 - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
 - Buffer results until it determines they are actually needed
 - Flush buffers on incorrect speculation

Static Multiple Issue

- Compiler groups instructions into “issue packets”
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations
 - ®Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove/reduce/prevent some/all hazards
 - Reorder instructions into issue packets
 - No dependencies within a packet
 - Possibly some dependencies between packets
 - This will be handled by Hardware (forwarding/Stall)
 - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
		IF	ID	EX	MEM	WB		
n	ALU/branch							
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Speculation

- Predict branch and continue issuing
 - Don't commit until branch outcome determined
- Load speculation
 - support speculation on load addresses,
 - allowing load-store reordering,
 - Don't commit load until speculation cleared



Chapter 5

Large and Fast: Exploiting Memory Hierarchy

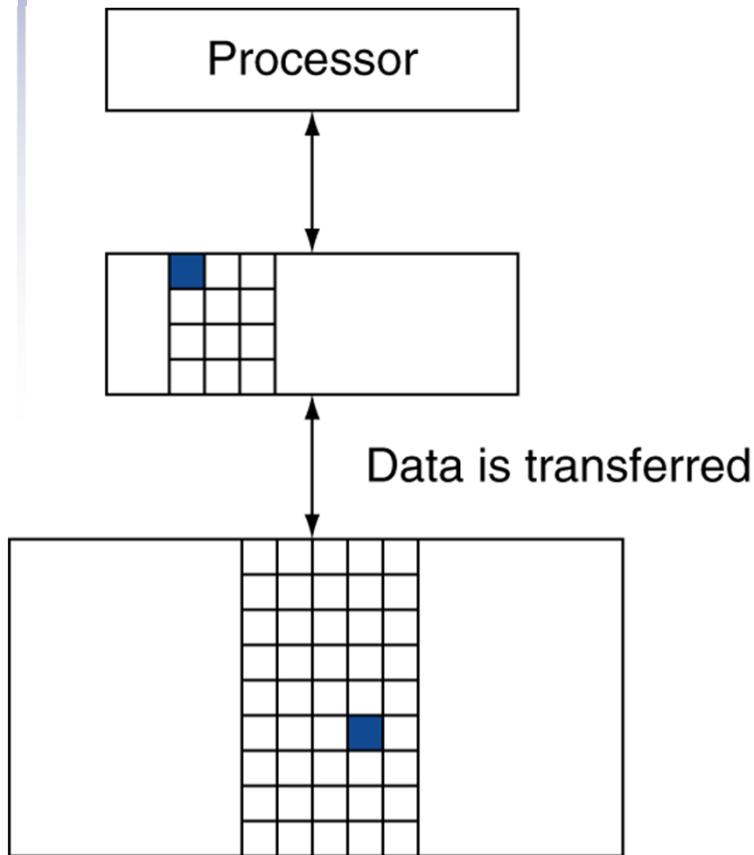
Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
 - Items accessed recently are likely to be accessed again soon
 - e.g., instructions in a loop, induction variables
- Spatial locality
 - Items near those accessed recently are likely to be accessed soon
 - E.g., sequential instruction access, array data

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
 - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
 - Cache memory attached to CPU

Memory Hierarchy Levels



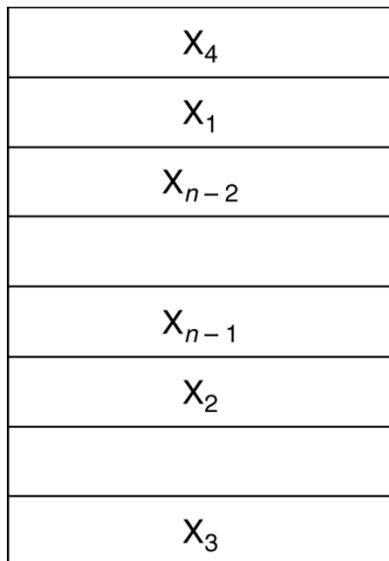
- Block (aka line): unit of copying
 - May be multiple words
- If accessed data is present in upper level
 - Hit: access satisfied by upper level
 - Hit ratio: hits/accesses
- If accessed data is absent
 - Miss: block copied from lower level
 - Time taken: miss penalty
 - Miss ratio: misses/accesses
 $= 1 - \text{hit ratio}$
 - Then accessed data supplied from upper level

Basic Structure of Memory Hierarchy Levels

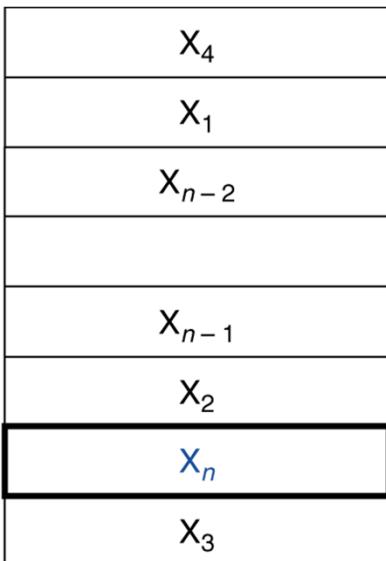
Speed	Processor	Size	Cost (\$/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

Cache Memory

- Cache memory
 - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, \dots, X_{n-1}, X_n



a. Before the reference to X_n

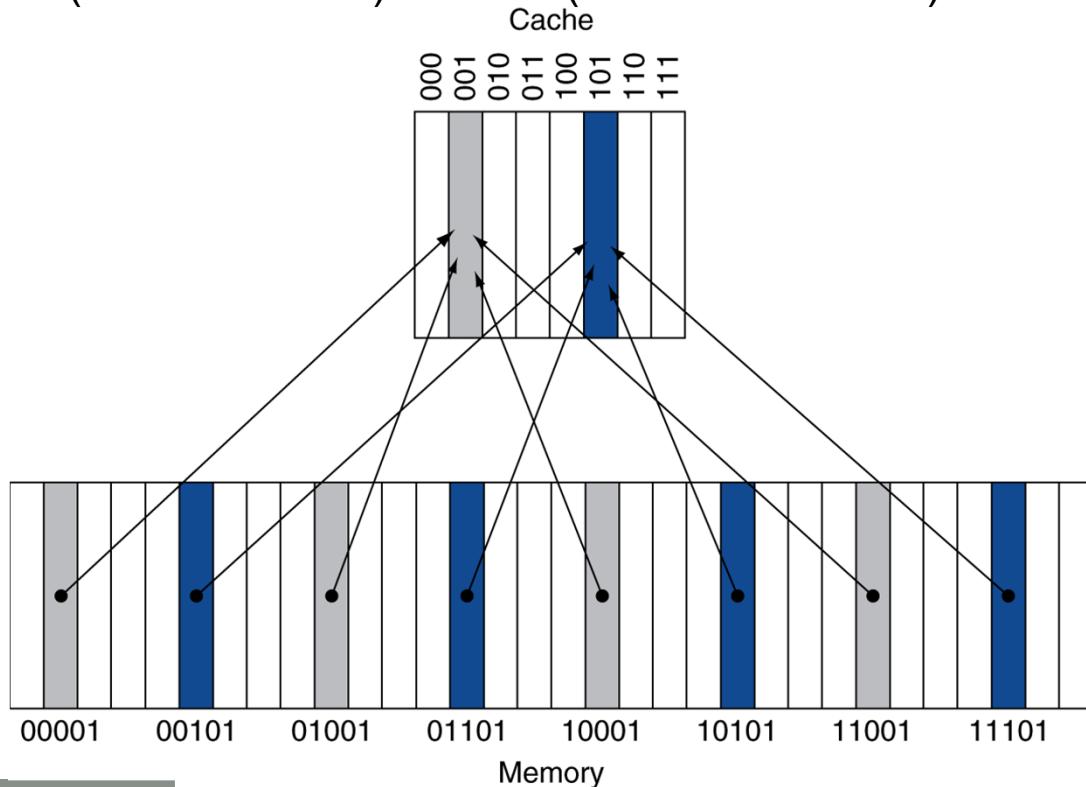


b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by address
- Direct mapped: A cache structure in which each memory location is mapped to exactly one location in the cache.
- Mapping to find a memory block in the cache =
 - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
 - Store block address as well as the data
 - Actually, only need the high-order bits
 - Called the tag
- What if there is no data in a location?
 - Valid bit: 1 = present, 0 = not present
 - Initially 0

Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

22,
26,
22,
26,
16,
3,
16,
18,
16

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110



22,

26,

22,
26,

16,
3,

16,
18,

16

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

22,

26,

22,

26,

16,

3,

16,

18,

16



Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

22,
26,
22,
26,
16,
3,
16,
18,
16



Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

22,

26,

22,

26,

16,

3,

16,

18,

16

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

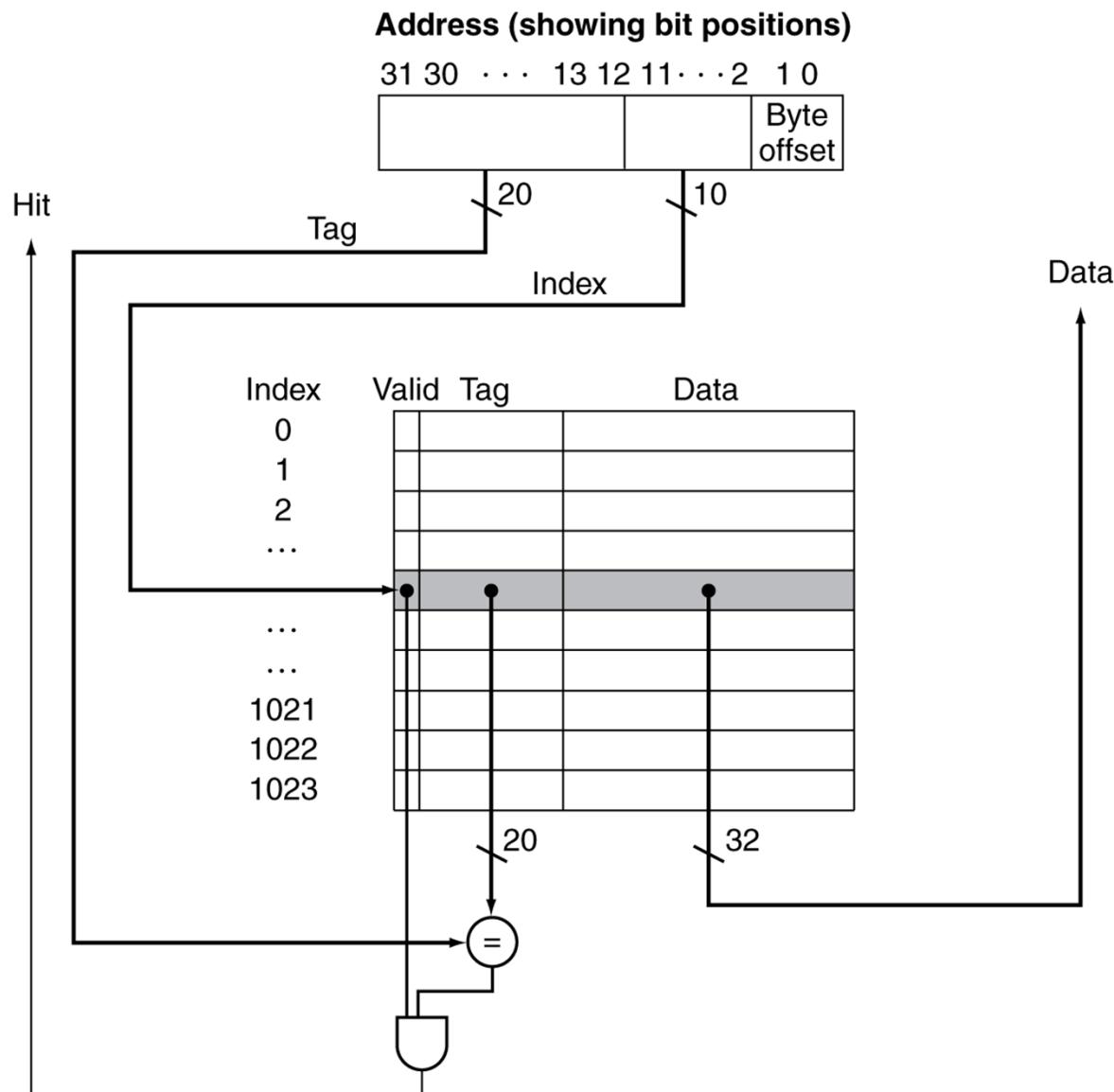
Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

22,
26,
22,
26,
16,
3,
16,
18,
16

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Address Subdivision



Total bits in a cache

- 32-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks
 - so n bits are used for the index
- The block size (b) is 2^m words (2^{m+2} bytes = 2^{m+2+3} bits)
 - m bits are used to access the words within the block
 - two bits are used for the byte part of the address

Total number of bits
(C)

- $C = 2^n \times (b + t + v)$
- $t = 32 - (n + m + 2)$
- $b = 2^{m+5}$

C

$$\begin{aligned} &= 2^n \times (2^{m+5} + 32 - (n + \\ &m + 2) + 1) \\ &= 2^n \times (2^m \times 32 + 32 - n \\ &- m - 1) \end{aligned}$$

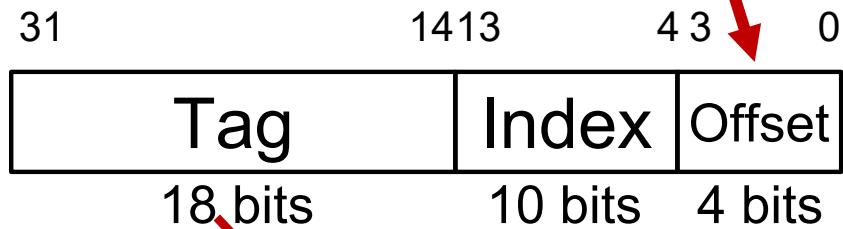
Size of tag field

v is the valid field size, i.e., 1

Block size

Total bits in a cache

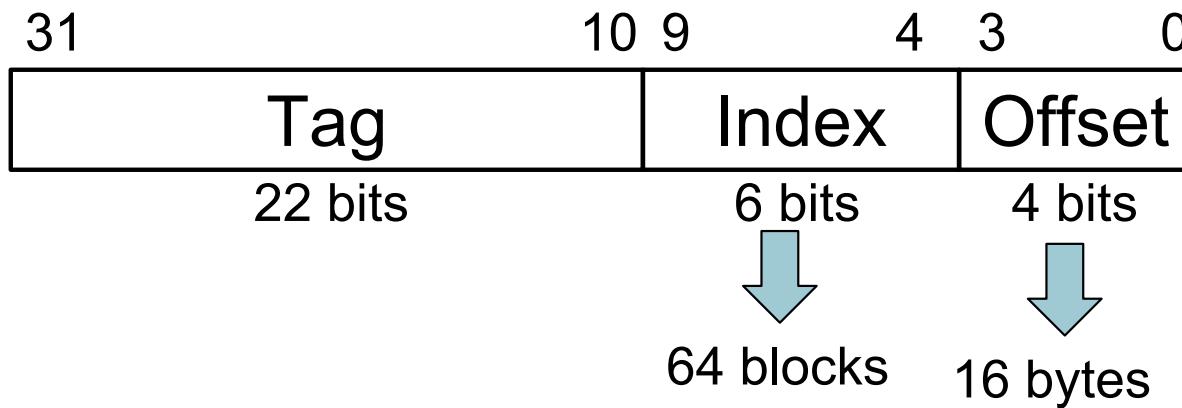
- How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks (i.e., 16 bytes), assuming a 32-bit address?



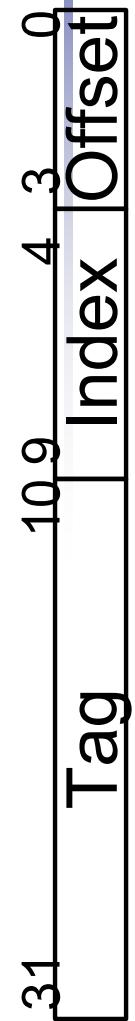
- 16 KiB = 4 Ki Words =
1 Ki Blks = 2^{10} Blks
 $C = 1024 \times (b + t + v)$
 $= 1024 \times (4 \times 32 + t + 1)$
 $= 1024 \times (4 \times 32 + 18 + 1)$
 $= 147$ Kilo bits

Example: Larger Block Size

- 64 blocks, 16 bytes/block
 - To what block number does address 1200 map?
- Block address = $\lfloor \frac{1200}{16} \rfloor = 75$
- Block number = $75 \bmod 64 = 11$



Example: Larger Block Size



64 blocks, 16 bytes/block

- To what block number does address 1200 map?

$$\text{Block address} = \lfloor \frac{1200}{16} \rfloor = 75$$

$$\text{Block number} = 75 \bmod 64 = 11$$

In fact Block 11 maps all addresses between 1200 and 1215.

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor$$

$\times \text{Bytes per block}$

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor$$

$\times \text{Bytes per block} + (\text{Bytes per block} - 1)$

Byte Address
↑



Block Size Considerations

- Larger blocks should reduce miss rate
 - Due to spatial locality
- But in a fixed-sized cache
 - Larger blocks \Rightarrow fewer of them
 - More competition \Rightarrow increased miss rate
- Larger miss penalty
 - Larger blocks \Rightarrow Larger transfer time
 - Can override benefit of reduced miss rate
 - Early restart critical-word-first can help

Early restart

- resume execution as soon as the requested word of the block is returned; Does not wait for the entire block
- For instruction access, it works best
 - Instruction accesses are largely sequential
 - so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time.
 - This technique is usually less effective for data caches because the probability that the processor will need another word from a different cache block before the transfer of the current block completes is high

Critical Word First

- Organizes the memory so that the requested word is transferred from the memory to the cache first.
- The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block.
- Can be slightly faster than early restart
- but it is limited by the same properties that limit early restart.

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
 - Stall the CPU pipeline
 - Fetch block from next level of hierarchy
 - Instruction cache miss
 - Restart instruction fetch
 - Data cache miss
 - Complete data access

Write-Through

A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data is always consistent between the two.

- On data-write hit, could just update the block in cache
 - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
- Solution: write buffer
 - Holds data waiting to be written to memory
 - CPU continues immediately
- Only stalls on write if write buffer is already full

Write-Back

A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

- Alternative: On data-write hit, just update the block in cache
 - Keep track of whether each block is dirty
- When a dirty block is replaced
 - Write it back to memory

Can use a write buffer to allow replacing block to
be read first

Implementing Store for write-through

- we can write the data into the cache first and then read the tag;
- if the tag mismatches, then a miss occurs.
- Because the cache is write-through, the overwriting of the block in the cache is not catastrophic
 - memory has the correct value and we can do it right.
- stores require one cycle when a hit occurs:
 - In the same clock cycle cache is overwritten and the tag is checked.
 - THIS CANNOT BE DONE FOR WRITE BACK

Implementing Store for Write Back

- Read the tag first
- If we have a cache miss, we must first write the block back to memory if the data in the cache is modified. Then the cache is overwritten.
- stores require two cycles:
 - a cycle to check for a hit
 - followed by a cycle to actually perform the write on the cache

Write Back- Write Buffer for hit

Write Buffer: A queue that holds data while the data is waiting to be written to memory.

- effectively allowing the store to take only one cycle by pipelining it:
 - When a store buffer is used, the processor does the cache lookup and places the data in the store buffer during the normal cache access cycle.
 - Assuming a cache hit, the new data is written from the store buffer into the cache on the next unused cache access cycle.

Write Back- Write Buffer for miss

- the modified block (the block that is going to be replaced in the cache) is moved to a write-back buffer in case of a miss
 - while the requested block is read from memory.
 - The write-back buffer is later written back to memory.
- Assuming another miss does not occur immediately, this technique reduces the miss penalty

Measuring Cache Performance

Components of CPU time

- CPU execution cycles
 - Includes cache hit time
- Memory stall cycles
 - Mainly from cache misses

Measuring Cache Performance

- Read-stall cycles = $\frac{\{\text{Reads}\}}{\{\text{Program}\}} \times$
Read miss rate \times Read miss penalty
- Write-stall cycles = $\left(\frac{\{\text{Writes}\}}{\{\text{Program}\}} \times \right.$
Write miss rate \times Write miss penalty $) +$
Write buffer stalls

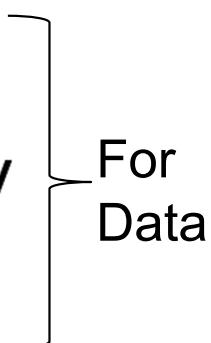
For Write-through Cache

Measuring Cache Performance

With simplifying assumptions:

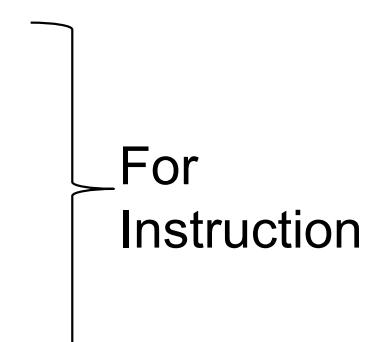
- read and write miss penalties are same
 - In most write-through schemes this is the case
- Write buffer stalls are negligible

Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$


For Data

Memory stall cycles

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$


For Instruction

Cache Performance Example



Given

- miss rate: I-cache = 2% ; D-cache = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- Load & stores are 36% of instructions

How much faster is a processor with a perfect cache?

Say total instruction: I

- I-cache miss cycles: $I \times 0.02 \times 100 = 2.00 \times I$
- D-cache: $I \times 0.36 \times 0.04 \times 100 = 1.44 \times I$

$$\text{Actual CPI} = 2 + 2 + 1.44 = 5.44$$

Ideal CPU is $5.44/2 = 2.72$ times faster

Cache Performance Example 2



Given

- miss rate: I-cache = 2% ; D-cache = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 1
- Load & stores are 36% of instructions

How much faster is a processor with a perfect cache?

Say total instruction: I

- I-cache miss cycles: $I \times 0.02 \times 100 = 2.00 \times I$
- D-cache: $I \times 0.36 \times 0.04 \times 100 = 1.44 \times I$

Actual CPI = $1 + 2 + 1.44 = 4.44$

Ideal CPU is $4.44/1 = 4.44$ times faster

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
 - $AMAT = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$
- Example
 - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
 - $AMAT = 1 + 0.05 \times 20 = 2\text{ns}$
 - 2 cycles per instruction

Performance Summary

- When CPU performance increased
 - Miss penalty becomes more significant
- Decreasing base CPI
 - Greater proportion of time spent on memory stalls
- Increasing clock rate
 - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

Associative Caches

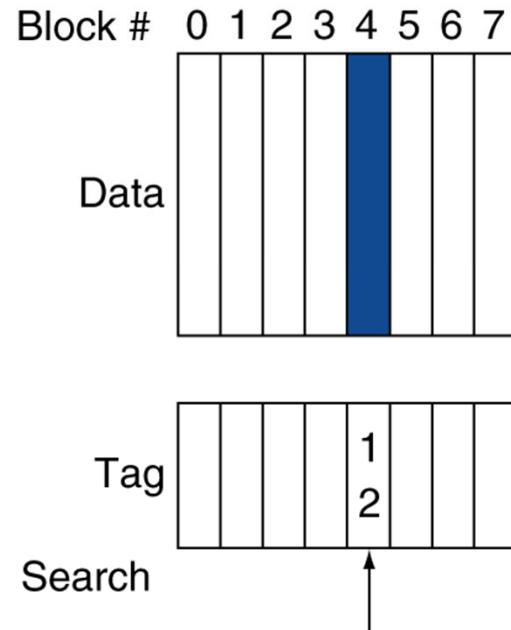
- Fully associative
 - Allow a given block to go in any cache entry
 - Requires all entries to be searched at once
 - Comparator per entry (expensive)
- n -way set associative
 - Each set contains n cache block entries
 - Set containing a memory block is given by
 - (Memory block number) modulo (#Sets in cache)
 - Search all entries in a given set at once
 - n comparators (less expensive)

Number of sets in a cache

- Consider a n -way set associative cache with m cache entries/blocks
 - #of cache blocks in a set = n
 - #of sets in a cache = m/n
- Direct Mapped cache is 1-way set associate
 - #of sets in a cache = $m/1 = m$
 - #of sets in a cache = #of blocks in a cache
 - Set containing a memory block is given by
 - (Memory block number) modulo (#blocks in cache)
- Full Associative cache is m -way set associate
 - #of sets in a cache = $m/m = 1$
 - a memory block can be placed anywhere in the cache

Associative Cache Example

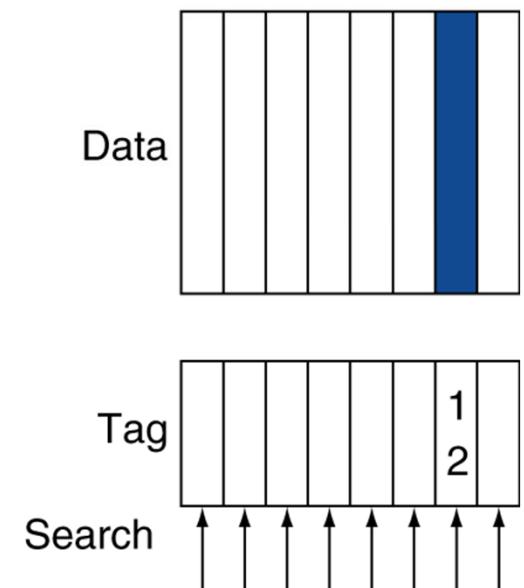
Direct mapped



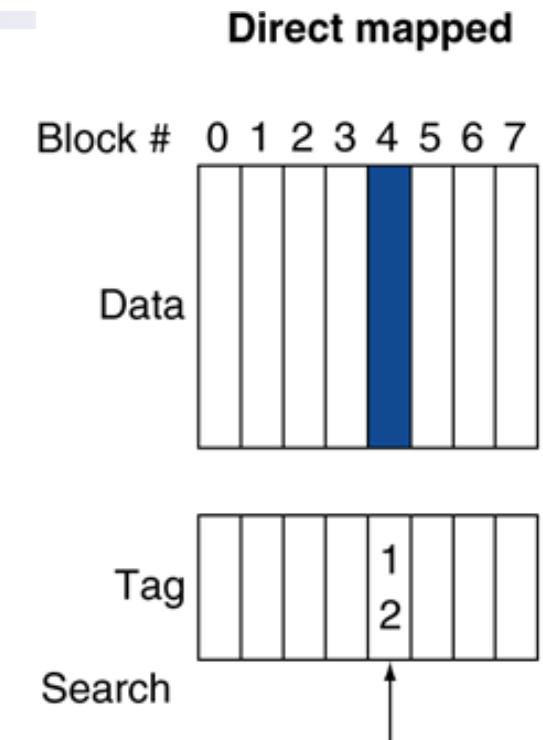
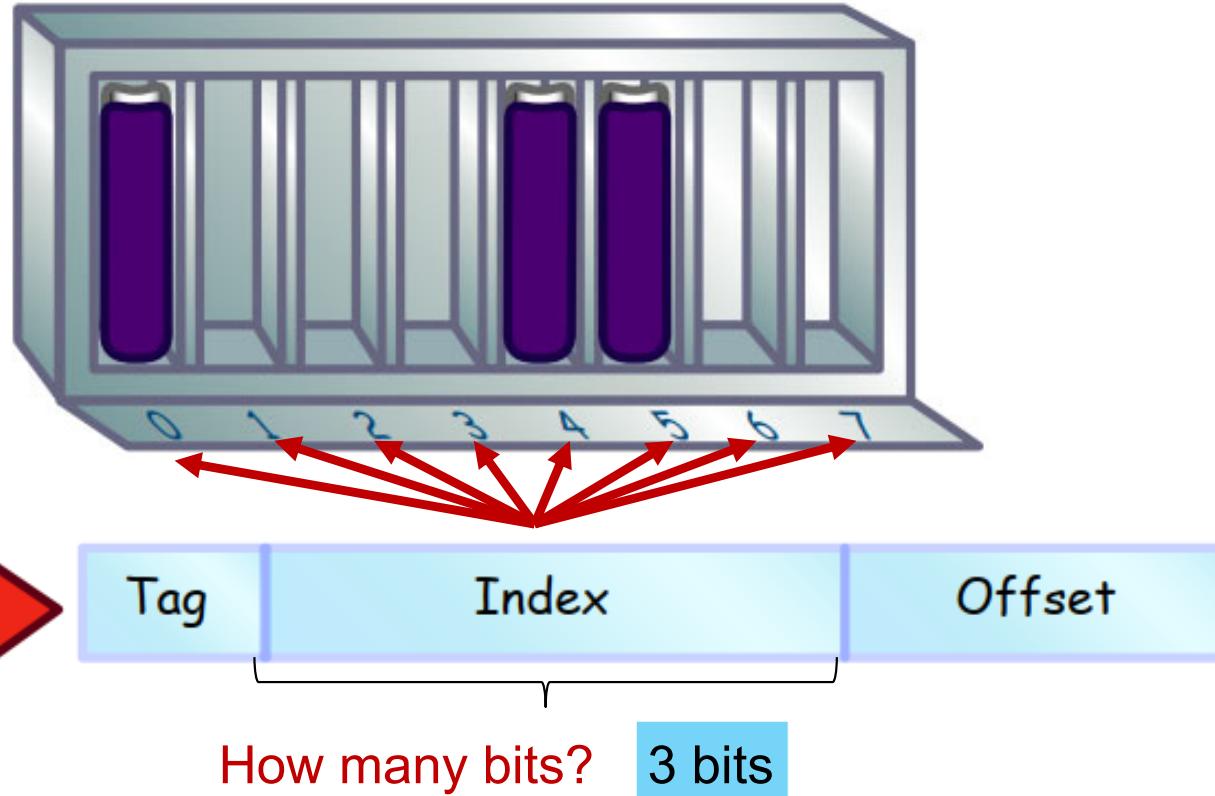
Set associative



Fully associative

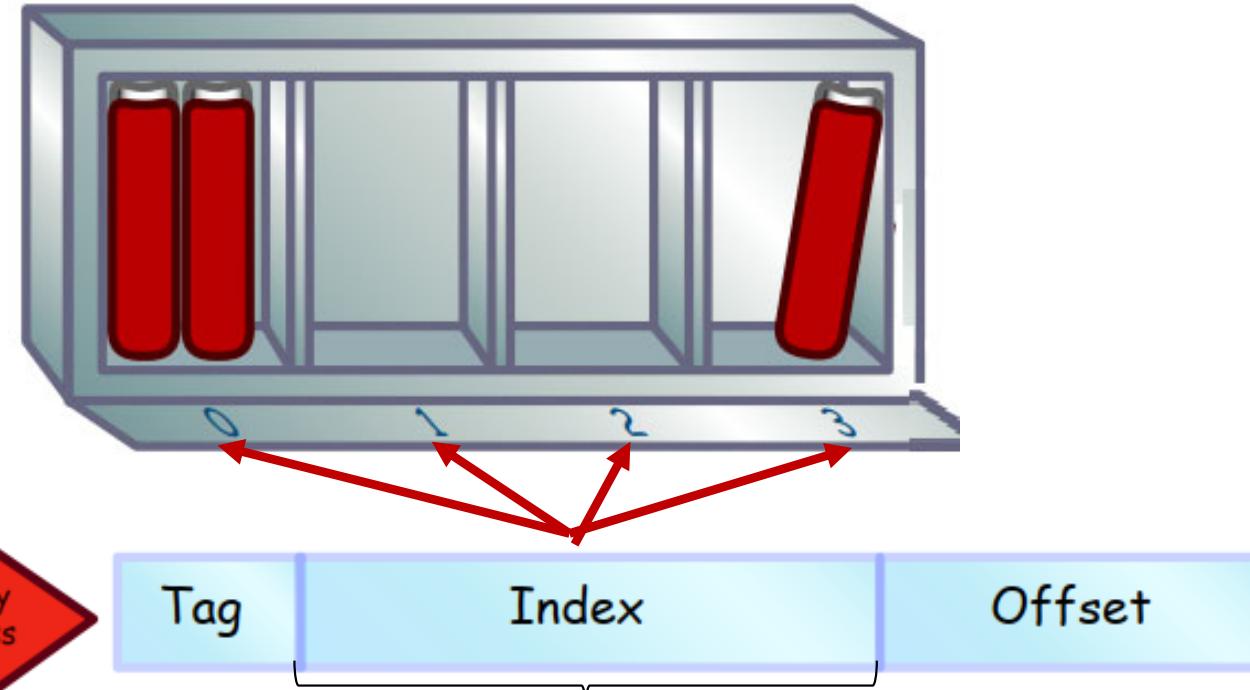


1-Way Set Associative



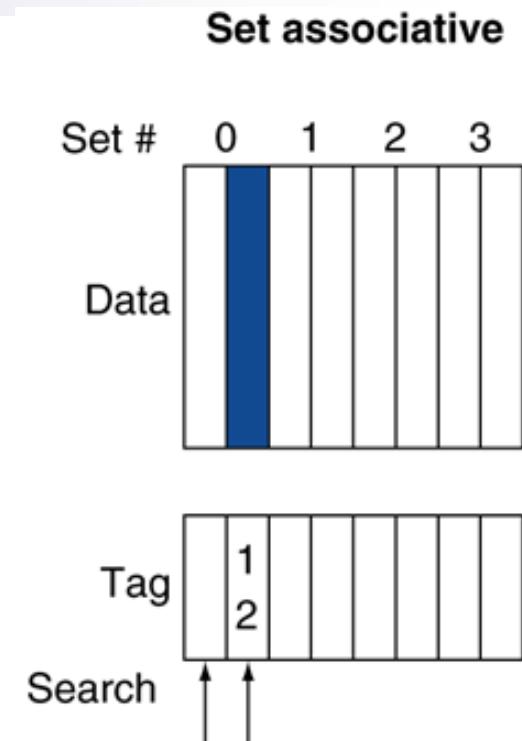
- A cache block can only go in one spot in the cache.
- It makes a cache block very easy to find
- but does not support flexibility in placing a memory block in cache.

2-Way Set Associative

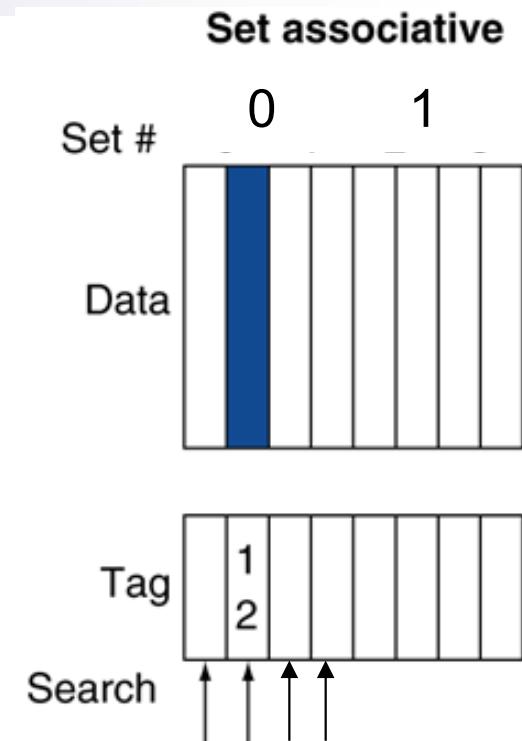
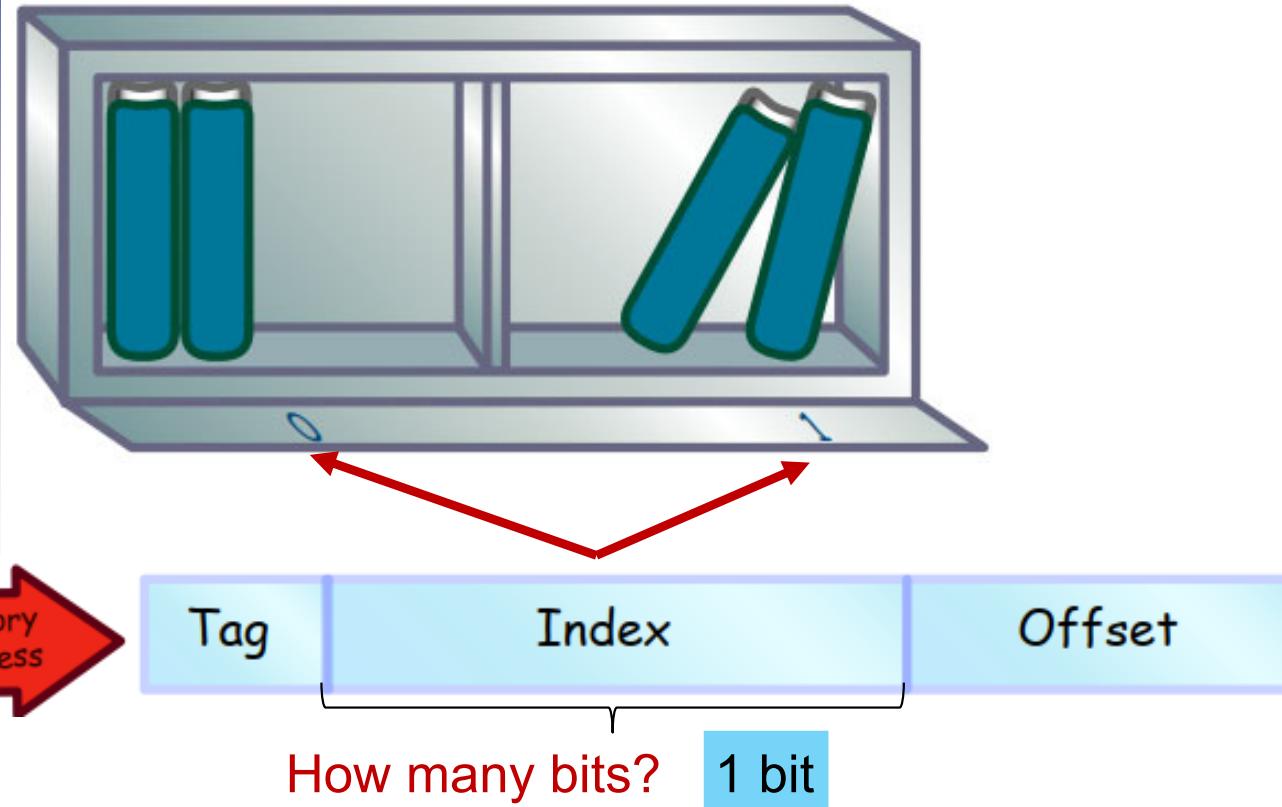


How many bits? 2 bits

- This cache is made up of sets that can fit two blocks each.
- The index is now used to find the set
- The tag helps find the block within the set.
- Need to check 2 tags in a set

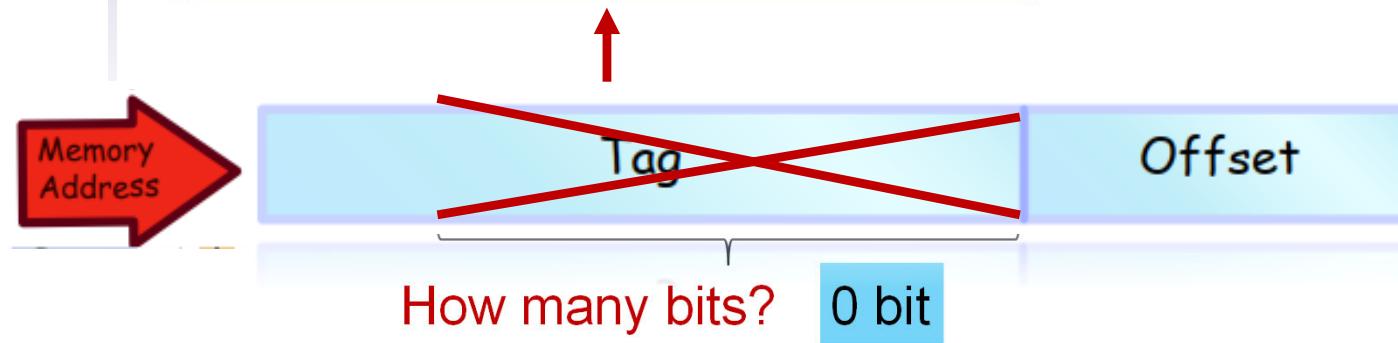
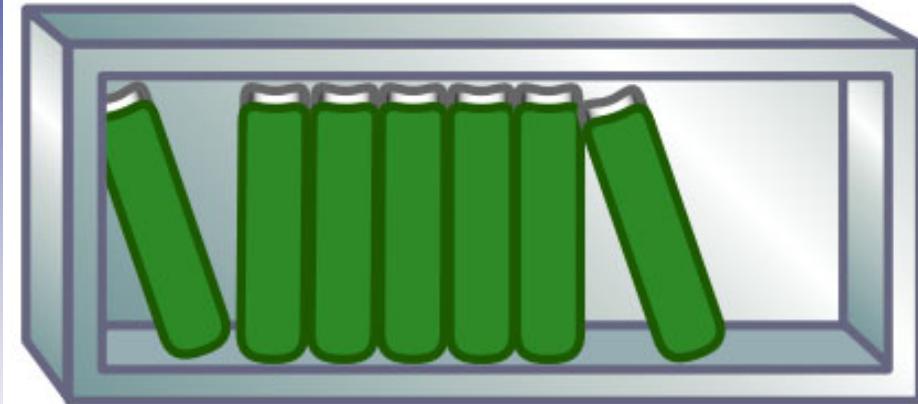


4-Way Set Associative

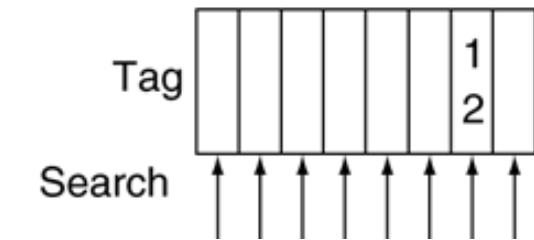


- Each set here fits four blocks,
- So there are fewer sets.
- As such, fewer index are needed.
- Need to check 4 tags within a set

Fully Associative



Fully associative



Search

- No **index** is needed, since a cache block can go anywhere in the cache.
- Every tag must be compared when finding a block in the cache
- but block placement is very flexible!

Spectrum of Associativity

- For a cache with 8 entries

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data														

Associativity Example

- Compare 4-block caches
 - Direct mapped, 2-way set associative, fully associative
 - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0 modulo 4 = 0	0	miss	Mem[0]			

0 modulo 4 = 0
8 modulo 4 = 0
0 modulo 4 = 0
6 modulo 4 = 2
8 modulo 4 = 0

Associativity Example

- Compare 4-block caches
 - Direct mapped, 2-way set associative, fully associative
 - Block access sequence: 0, 8, 0, 6, 8
- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0 modulo 2 = 0	0	miss	Mem[0]	

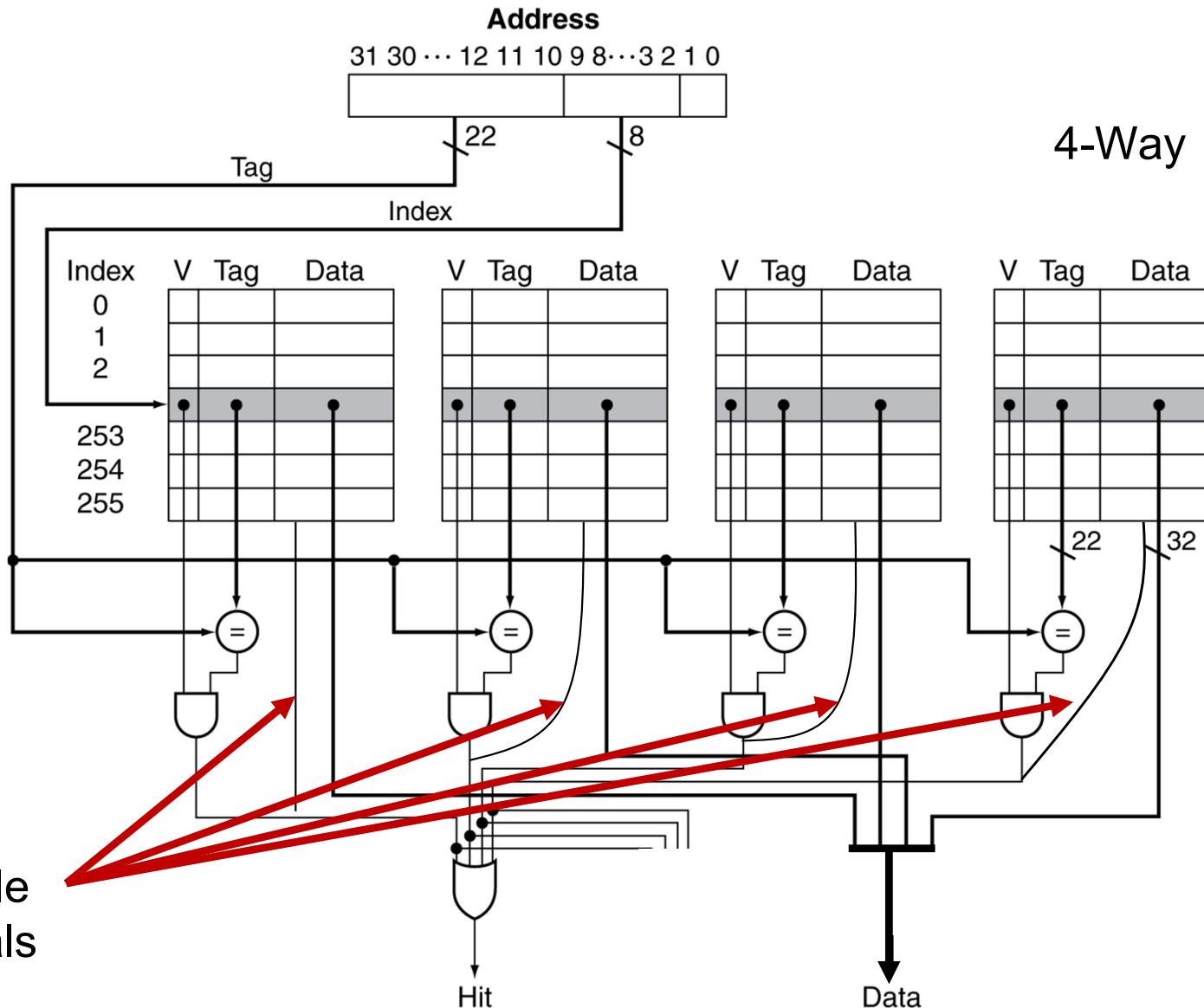
0 modulo 2 = 0
8 modulo 2 = 0
0 modulo 2 = 0
6 modulo 2 = 0
8 modulo 2 = 0

Associativity Example

- Compare 4-block caches
 - Direct mapped, 2-way set associative, fully associative
 - Block access sequence: 0, 8, 0, 6, 8
- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
 - Prefer non-valid entry, if there is one
 - Otherwise, choose among entries in the set
- Least-recently used (LRU)
 - Choose the one unused for the longest time
 - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
 - Gives approximately the same performance as LRU for high associativity

Tags versus Set Associativity

- Cache of 4096 blocks
- a 4-word block size
- 32-bit address,
- Find the total number of sets and the total number of tag bits for caches that are
 - direct mapped
 - two-way set associative
 - four-way set associative
 - fully associative.



Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = $16 = 2^4$ Bytes per block
- So for tag + index $32 - 4 = 28$ bits



- Direct Mapped:
 - $4096 (2^{12})$ 1-way set => 12 bit index
 - 16 bit tag ($28 - 12$)
 - $4096 \text{ entries} \cdot 16 \text{ bit tag} = 64 \text{ K bits tag}$

Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = $16 (2^4)$ Bytes block
- So for tag + index $32 - 4 = 28$ bits



- 2-way set associative :
 - $4096/2 = 2048 (2^{11})$ sets => 11 bit index
 - 17 bit tag ($28 - 11$)
 - $4096 \text{ entries} \cdot 17 \text{ bit tag} = 68 \text{ K bits tag}$

Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = $16 (2^4)$ Bytes block
- So for tag + index $32 - 4 = 28$ bits



- 4-way set associative :
 - $4096/4 = 1024 (2^{10})$ sets => 10 bit index
 - 18 bit tag (28 -10)
 - $4096 \text{ entries} \cdot 18 \text{ bit tag} = 72 \text{ K bits tag}$

Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = $16 (2^4)$ Bytes block
- So for tag + index $32 - 4 = 28$ bits



- Fully set associative :
 - No index
 - 28 bits tag ($28 - 0$)
 - $4096 \text{ entries} \cdot 28 \text{ bit tag} = 112 \text{ K bits tag}$

Tags versus Set Associativity

- 4096 blocks; 4-words block; 32-bit address
- 4-words block = $16 (2^4)$ Bytes block
- So, for tag + index has $32 - 4 = 28$ bits.

28 bits (tag + Index)		4 bits
Associativity	Tag bits (K)	
Direct mapped (1 Way)	64	
2 Way Set Associative	68	
4 Way Set Associative	72	
Fully Associative	112	

Multilevel Caches

- Primary cache attached to CPU
 - Small, but fast
- Level-2 cache services misses from primary cache
 - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - Miss rate/instruction @ primary cache = 2%
 - Main memory access time = 100ns
 - 4 GHz => 0.25ns cycle length
- With just primary cache
 - Miss penalty = $100\text{ns}/0.25\text{ns} = 400$ cycles
 - Effective CPI = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache
 - Access time = 5ns
 - Global miss rate to main memory = 0.5%
 - Primary miss with L-2 hit
 - Penalty = $5\text{ns}/0.25\text{ns} = 20 \text{ cycles}$
 - Primary miss with L-2 miss
 - $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio = $9/3.4 = 2.6$
- primary Miss
rate = 2%
- CPU base CPI = 1,
clock rate = 4GHz

Multilevel Cache Considerations

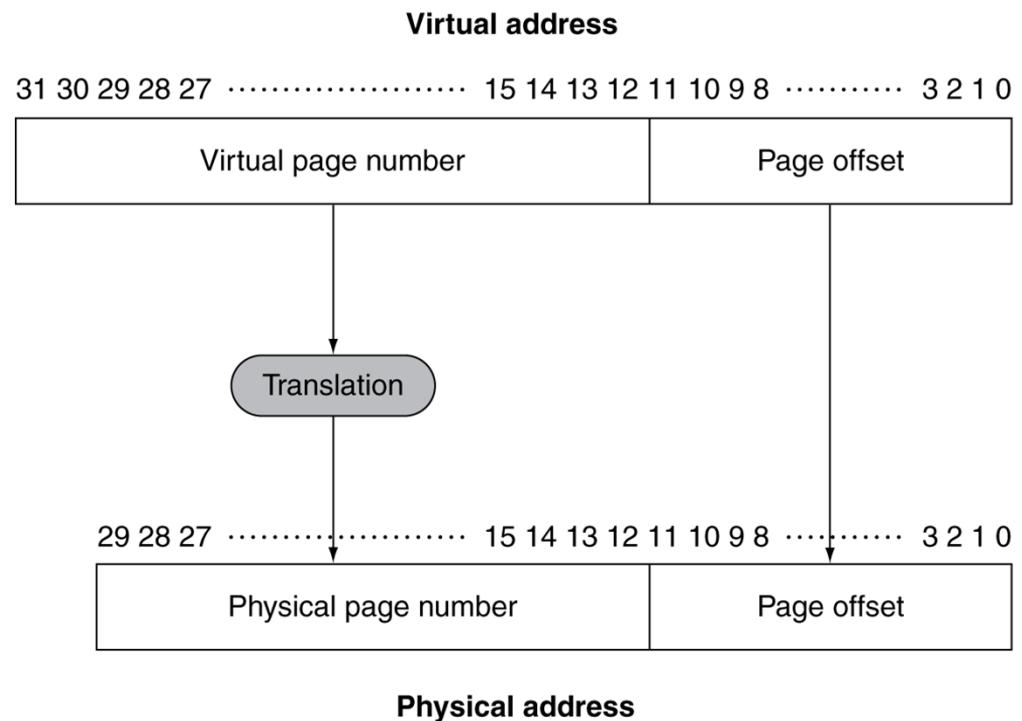
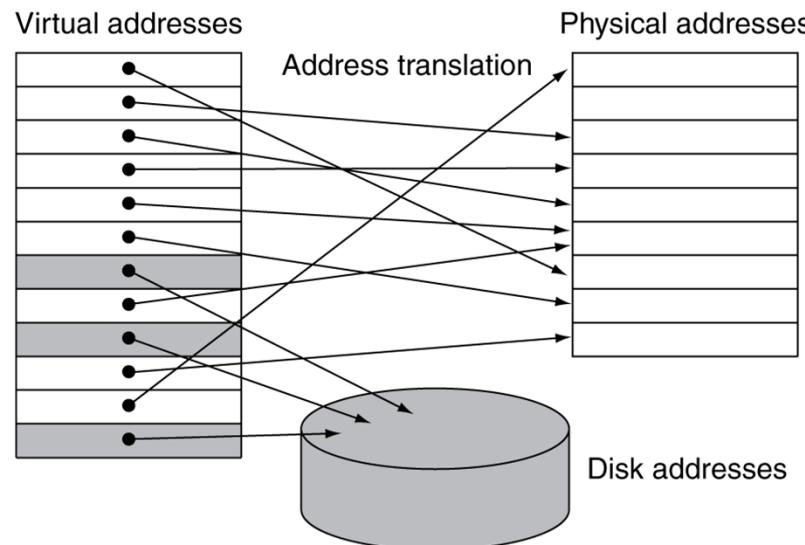
- Primary cache
 - Focus on minimal hit time
- L-2 cache
 - Focus on low miss rate to avoid main memory access
 - Hit time has less overall impact
- Results
 - L-1 cache usually smaller than a single cache
 - L-1 block size smaller than L-2 block size

Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
 - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
 - Each gets a private virtual address space holding its frequently used code and data
 - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
 - VM “block” is called a page
 - VM translation “miss” is called a page fault

Address Translation

- Fixed-size pages (e.g., 4K)



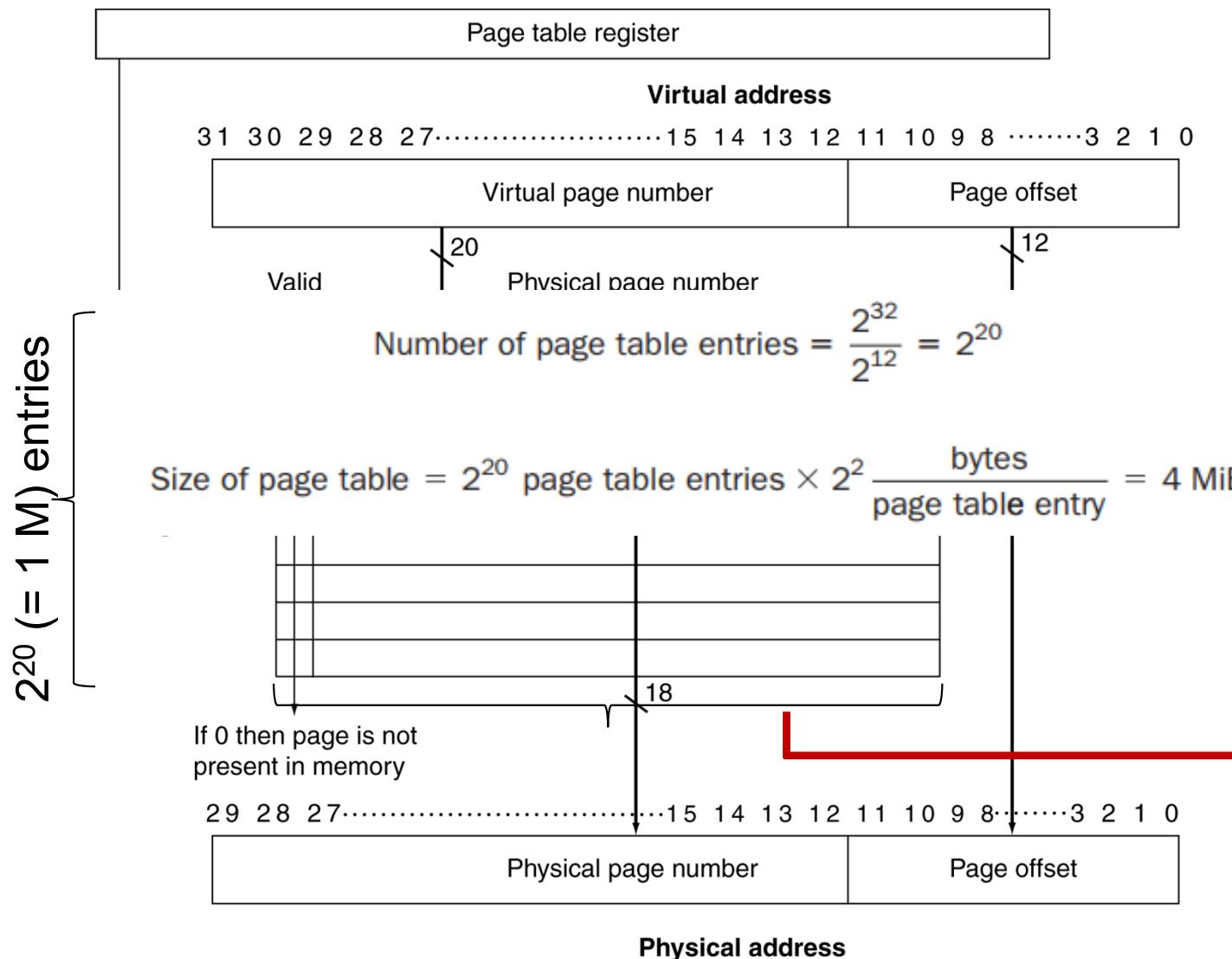
Page Fault Penalty

- On page fault, the page must be fetched from disk
 - Takes millions of clock cycles
 - Handled by OS code
- Try to minimize page fault rate
 - Fully associative placement
 - But this needs costly search!!!
 - Smart replacement algorithms

Page Tables

- Stores placement information
 - Array of page table entries, indexed by virtual page number
 - Page table register in CPU points to page table in physical memory
- If page is present in memory
 - PTE stores the physical page number
 - Plus other status bits (referenced, dirty, ...)
- If page is not present
 - PTE can refer to location in swap space on disk

Translation Using a Page Table



What is
the size
of the
page table?

19 bits;
but 32 bits
wide usually

Page Table Size Issues

- Size of page table => 4 MB (per process)
- What about 100 processes
 - Each with its own page table?
- What will happen if we have 64-bit addresses (according to prev. calc.)?
 - $2^{(64 - 12)} = 2^{52}$ entries!!!

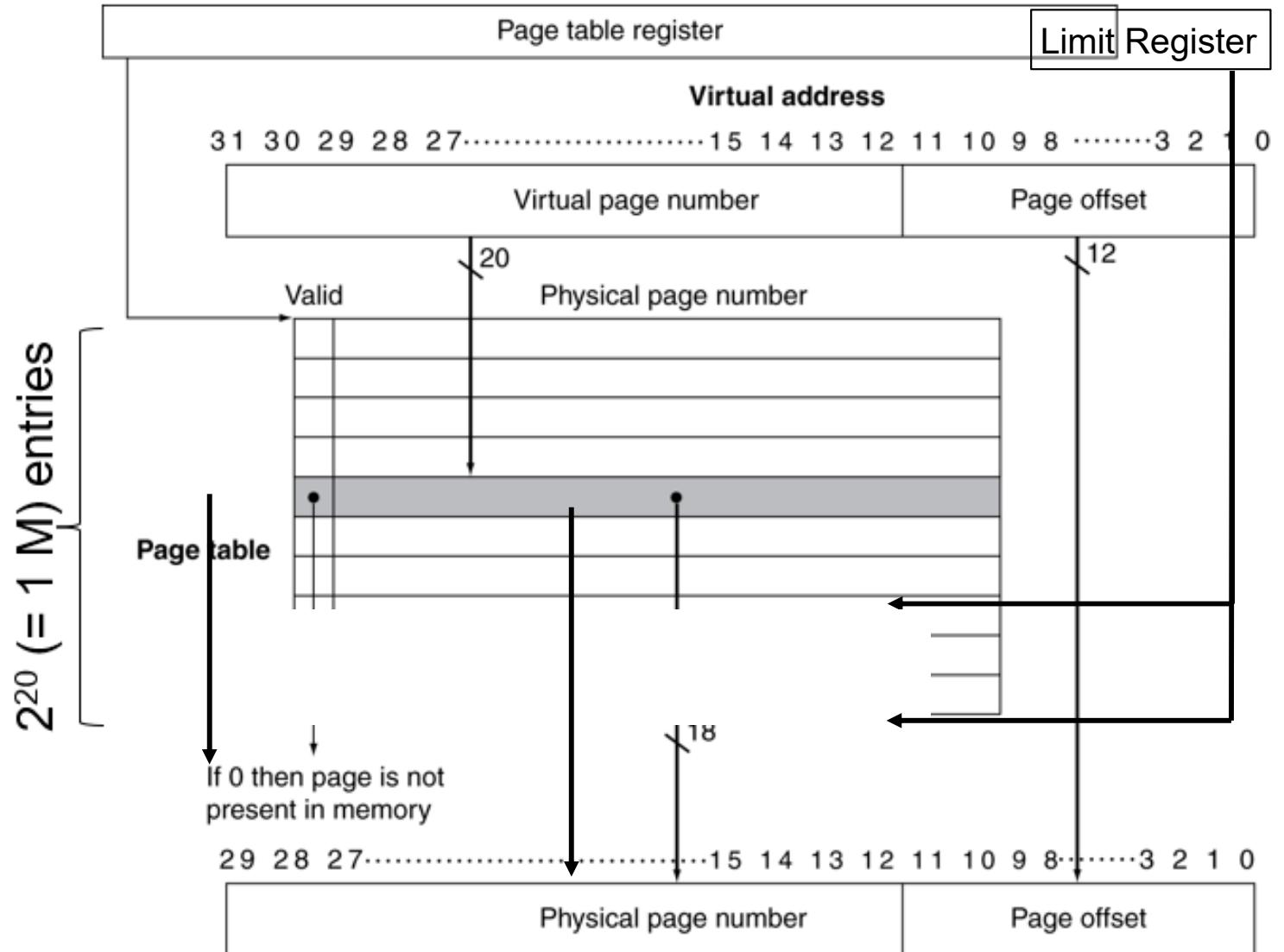


- There are techniques to reduce the amount of storage required for the page table.

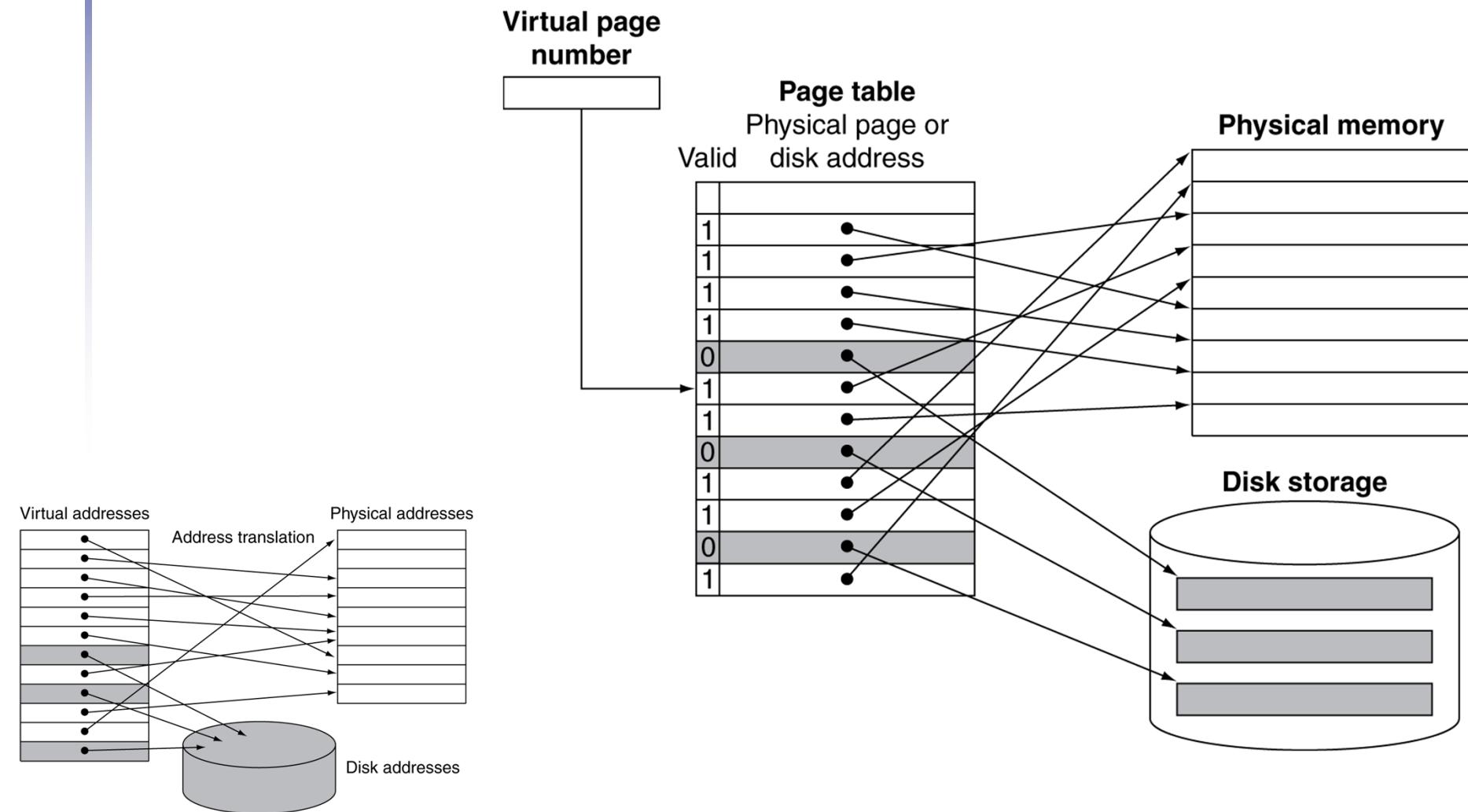
Techniques: Limit Registers

- Keep a limit register that restricts the size of the page table for a given process.
 - If the virtual page number becomes larger than the contents of the limit register, entries must be added to the page table.
 - This technique allows the page table to grow as a process consumes more space.
 - Thus, the page table will only be large if the process is using many pages of virtual address space.
 - This technique requires that the address space expand in only one direction.

Techniques: Limit Registers



Mapping Pages to Storage



Page Fault

- If the valid bit for a virtual page is off, a page fault occurs.
- The operating system must be given control.
 - This transfer is done with the exception mechanism
 - OS must find the page in the next level of the hierarchy
 - and decide where to place the requested page in main memory.

Swap Space

Swap Space

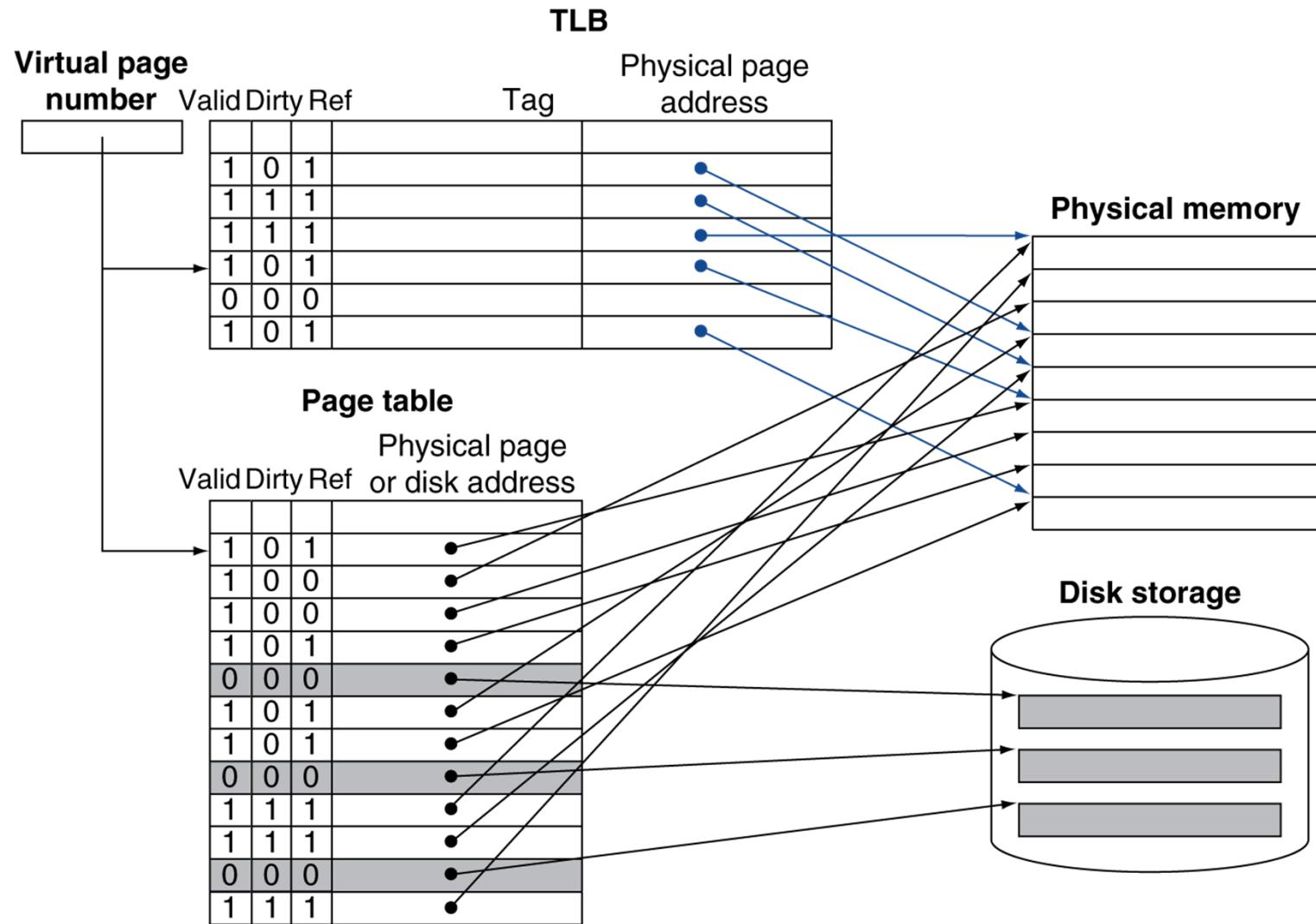
- Virtual addr. alone does not immediately tell us where the page is on disk.
- OS usually creates the space on flash memory/disk for all the pages of a process
- OS creates a data structure to record where each virtual page is stored on disk.
 - may be part of the page table
 - or an auxiliary data structure indexed in the same way as the page table
- OS also creates a data structure that tracks
 - which processes and which virtual addresses use each physical page

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
 - One to access the PTE
 - Then the actual memory access
- But access to page tables has good locality
 - So use a fast cache of PTEs within the CPU
 - Called a Translation Look-aside Buffer (TLB)
 - Typical: 16–512 PTEs
 - Misses could be handled by hardware or software

**Page Tables are
in main memory**

Fast Translation Using a TLB



TLB Misses

- If page is in memory
 - Load the PTE from memory and retry
 - Could be handled in hardware
 - Can get complex for more complicated page table structures
 - Or in software
 - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
 - OS handles fetching the page and updating the page table
 - Then restart the faulting instruction

TLB Miss Handler

- TLB miss indicates
 - Page present, but PTE not in TLB
 - Page not present **True Page Fault**
- Handler copies PTE from memory to TLB
 - Then restarts instruction
 - If page not present, page fault will occur
- The reference and dirty bits may change in TLB
 - So they must also be copied back to PTF for the TLB entry that is replaced **Write-Back Scheme**

Multiprocessor Definitions

multiprocessor A computer system with at least two processors. This is in contrast to a **uniprocessor**, which has one.

multicore microprocessor

A microprocessor containing multiple processors (“cores”) in a single integrated circuit.

job-level parallelism or **process-level parallelism** Utilizing multiple processors by running independent programs simultaneously.

parallel processing program A single program that runs on multiple processors simultaneously.

shared memory multiprocessor (SMP) is a multiprocessor system that offers the programmer a *single physical address space* across all processors. Note that such systems can still run independent jobs/programs in their own virtual address spaces (virtual memory), even if they all share a physical address space (physical memory). Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores.

cluster A set of computers connected over a local area network (LAN) that functions as a single large multiprocessor system. These processors communicate over standard network switches via message-passing. They do not share a single physical address space.

Types of multiprocessors based on Instruction and Data parallelism

instruction-level parallelism The parallelism among instructions.

data-level parallelism Parallelism achieved by operating on independent data.

A conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams.

SISD. Single Instruction stream, single Data stream. A uniprocessor.

MIMD. Multiple Instruction streams, multiple data streams. A multiprocessor.

SIMD. Single Instruction stream, multiple data streams.

A multiprocessor. The same instruction is applied to many data streams, as in a vector processor or array processor.

SPMD. Single Program, multiple Data streams. The conventional MIMD programming model, where a single program runs across all processors.

Reference:

Chapter 7, fourth edition of Computer Organization and Design by Patterson and Hennessy.