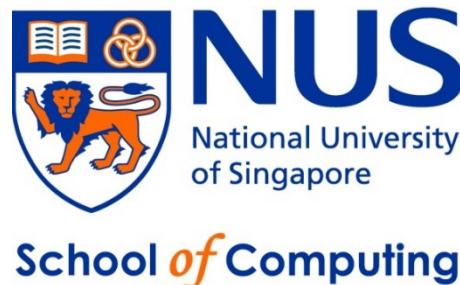


IT5002

Computer Systems and Applications

Introduction

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lecture 1: Introduction

1. **Programming Languages**
2. **Abstraction**
3. **What is a Computer?**
4. **IT5002: It's About Computer Organization and Applications**
5. **What's Next?**

Credit for course notes: A/P Aaron Tan (CS2100)



Programming Languages

Programming language: a formal language that specifies a set of instructions for a computer to implement specific algorithms to solve problems.



Programming Languages

High-level program

Eg: C, Java, Python, ECMAScript

```
int i, a = 0;
for (i=1; i<=10; i++) {
    a = a + i*i;
}
```

```
a = 0
for i in range(1,11):
    a = a + i*i
```

Low-level program

Eg: MIPS (IT5002)

```
addi $t1, $zero, 10
add $t1, $t1, $t1
addi $t2, $zero, 10
Loop: addi $t2, $t2, 10
      addi $t1, $t1, -1
      beq $t1, $zero, Loop
```

Machine code

Computers can execute
only machine code
directly.

```
0010000000010010000000000001010
00000001001010010100100000100000
. . .
```



Programming Languages

- ❖ 1st Generation Languages
- ❖ 2nd Generation Languages
- ❖ 3rd Generation Languages
- ❖ 4th Generation Languages
- ❖ 5th Generation Languages

Machine language.
Directly executable by machine.
Machine dependent.
Efficient code but difficult to write.

Assembly language.
Need to be translated (**assembled**) into machine code for execution.
Efficient code, easier to write than machine code.

Closer to English.
Need to be translated (**compiled** or **interpreted**) into machine code for execution.
Eg: FORTRAN, COBOL, C, BASIC

Require fewer instructions than 3GL.
Used with databases (query languages, report generators, forms designers)
Eg: SQL, PostScript, Mathematica

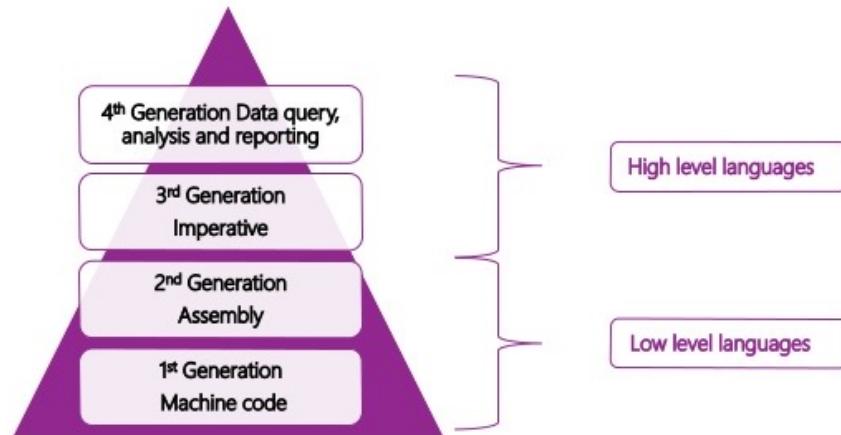
Used mainly in A.I. research.
Declarative languages
Functional languages (eg: Lisp, Scheme, SML)
Logic programming (eg: Prolog)



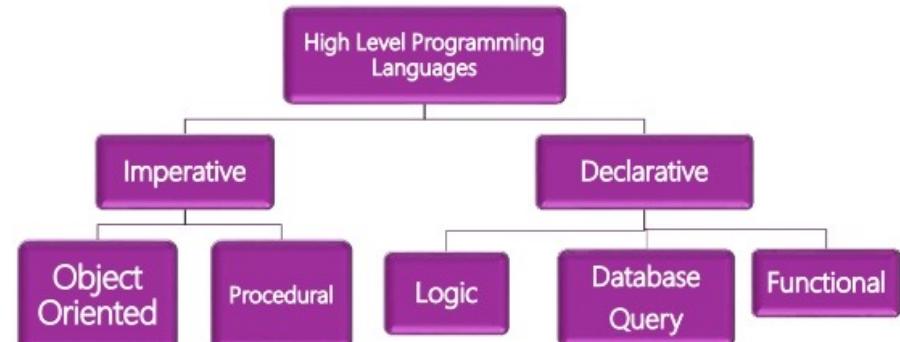
Programming Languages

- “Generational” classification of high level languages (3GL and later) was never fully precise.
- A different classification is based on **paradigm**.

Programming Languages - Generations



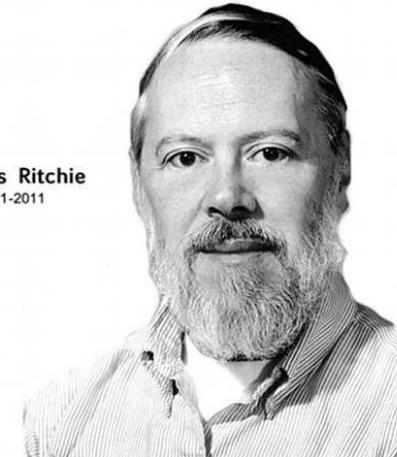
Hierarchy of High Level Languages



The C Programming Language

- Created by Dennis Ritchie (1941 – 2011) at Bell Laboratories in the early 1970s.
- C is an **imperative procedural language**.
- C provides constructs that map efficiently to typical machine instructions.
- C is a high-level language very close to the machine level, hence sometimes it is called “mid-level”.
- UNIX is written in C.

Dennis Ritchie
1941-2011



```
#include <stdio.h>
int main(void) {
    printf("Hello, world\n");
    return 0;
}
```

HelloWorld.c

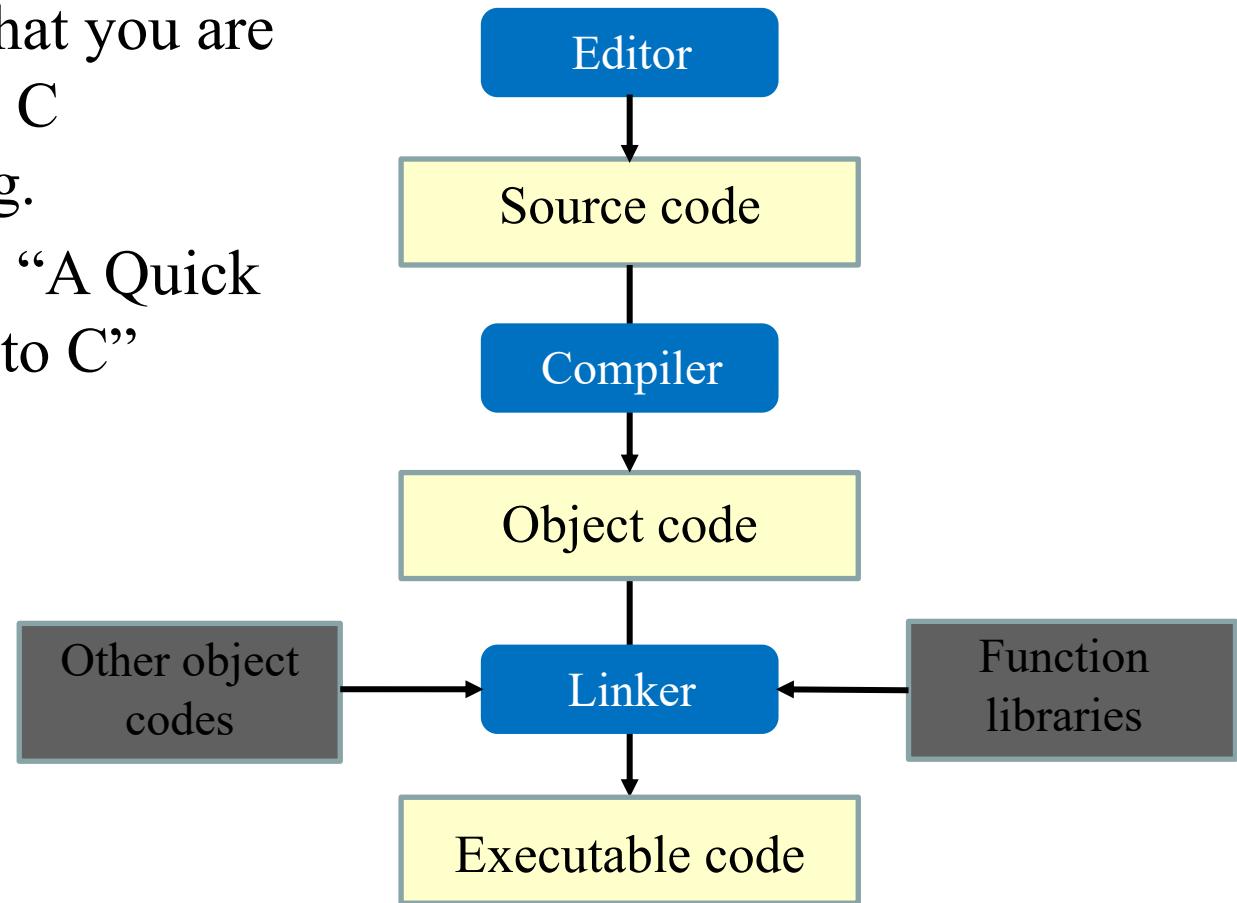
HelloWorld.py

```
print("Hello, world")
```



The C Programming Language

- Creating a C program
 - We assume that you are familiar with C Programming.
 - If not see the “A Quick Introduction to C” document.

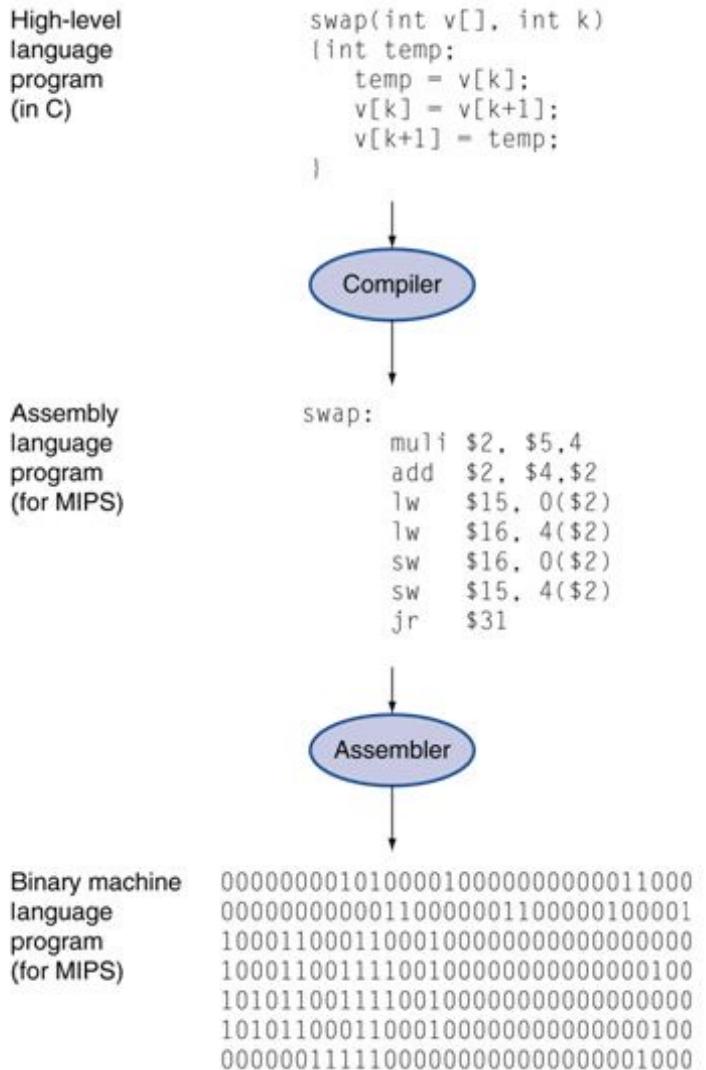


Abstraction

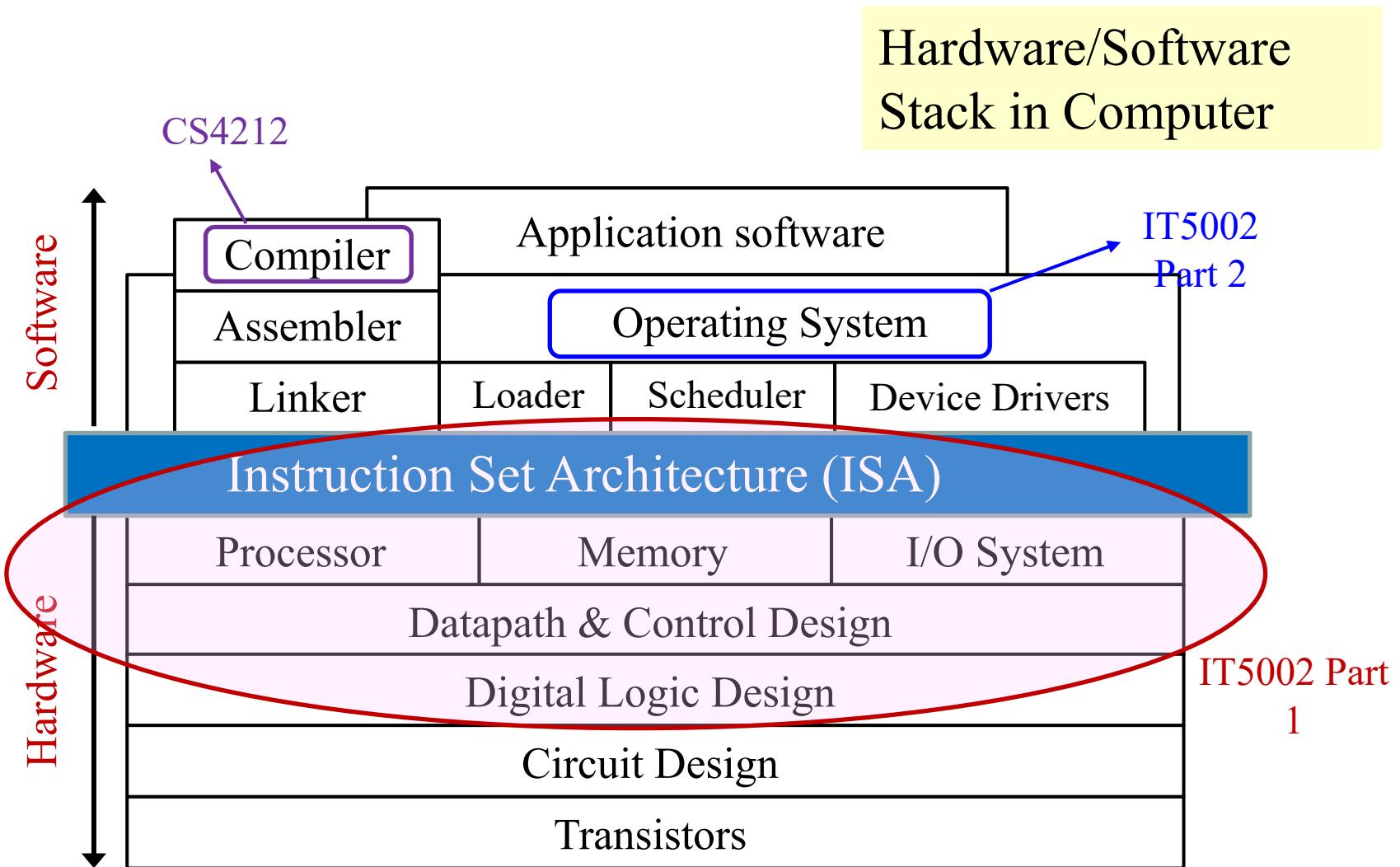
- High-level language
 - Level of abstraction closer to problem domain
 - Provides productivity and portability

- Assembly language
 - Textual and symbolic representation of instructions

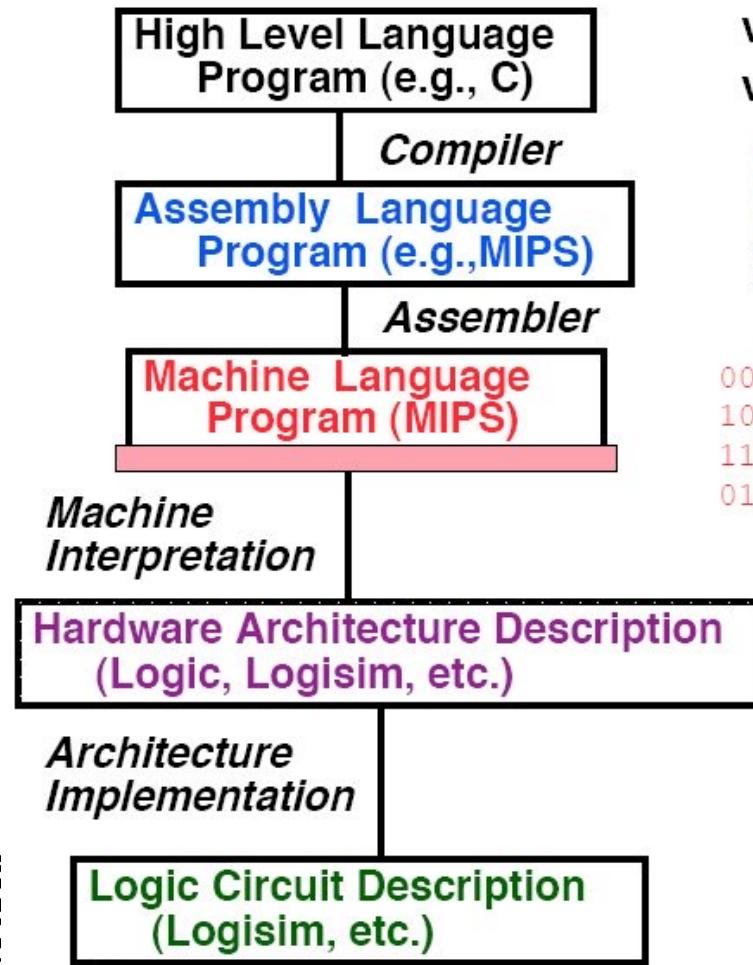
- Machine code (object code or binary)
 - Binary bits of instructions and data



Abstraction



Abstraction

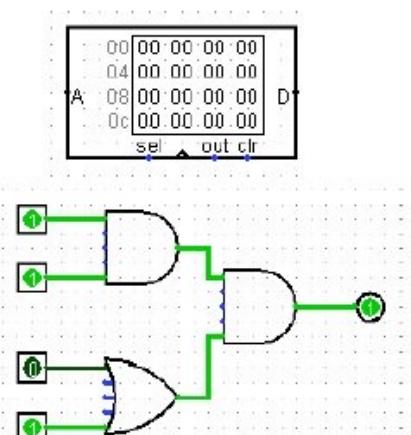


Level of Representation

`temp = v[k];`
`v[k] = v[k+1];`
`v[k+1] = temp;`

`lw $t0, 0($2)`
`lw $t1, 4($2)`
`sw $t1, 0($2)`
`sw $t0, 4($2)`

0000 1001 1100 0110 1010 1111 0101 1000
 1010 1111 0101 1000 0000 1001 1100 0110
 1100 0110 1010 1111 0101 1000 0000 1001
 0101 1000 0000 1001 1100 0110 1010 1111



What is a Computer?

MOST RECENT July 8th, 2012 1 COMMENT »

Most Recent Computer Technology

Written by: admin Tags: breaking



Do you know what is in your computer? Maybe you peeked when the repair technician was installing amazing for you. When you primary open up the CPU and seem inside, a computer is a very intimidating machine. But

SHARE

 0  Digg ↑

once you are acquainted with about the dissimilar parts that make up a total computer it gets a lot easier. Today's computer consists of around eight main devices; some of the advanced computers might have a few additional mechanisms. What are these eight main components and what are they used for? We will start with beginner level facts to get you in progress.

First is the Power Supply. The authority provides is used to provide electrical

1. Power supply
2. Motherboard
3. **Central Processing Unit (CPU)**
4. Random Access Memory (RAM)
5. Hard drive
6. Cooling fan
7. I/O devices

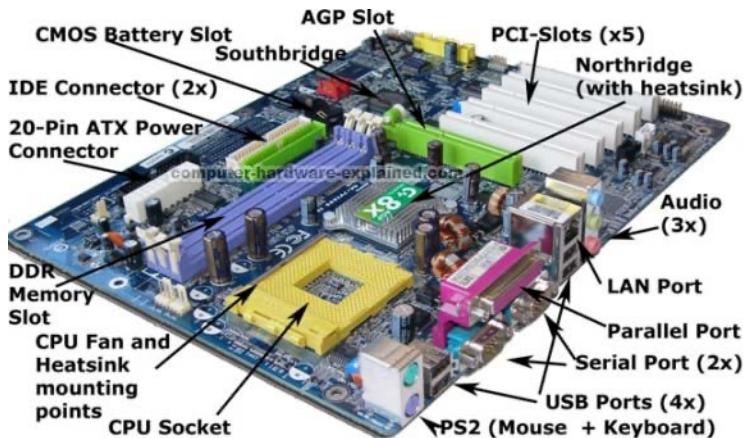


Credit:

http://www.overclock3d.net/reviews/cpu_mainboard/the_computer_council_-_clocked_gamer_quad/1

What is a Computer?

■ PC motherboard

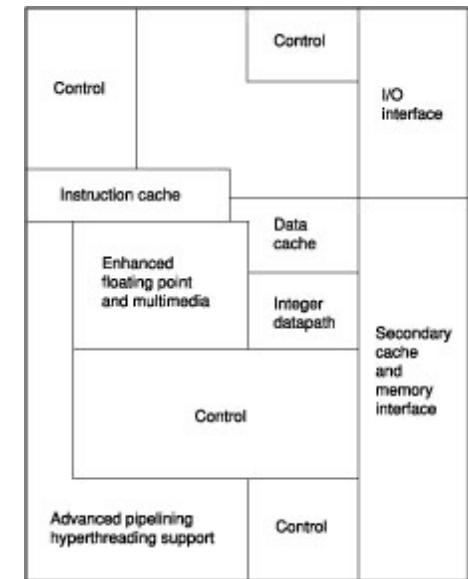
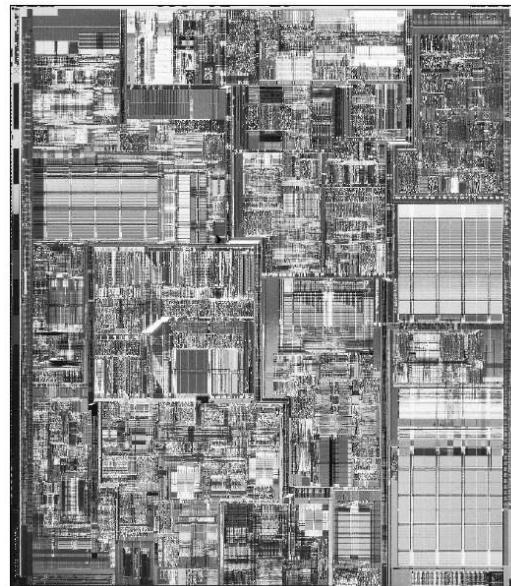


Credit: <http://www.computer-hardware-explained.com/what-is-a-motherboard.html>

■ Pentium processor

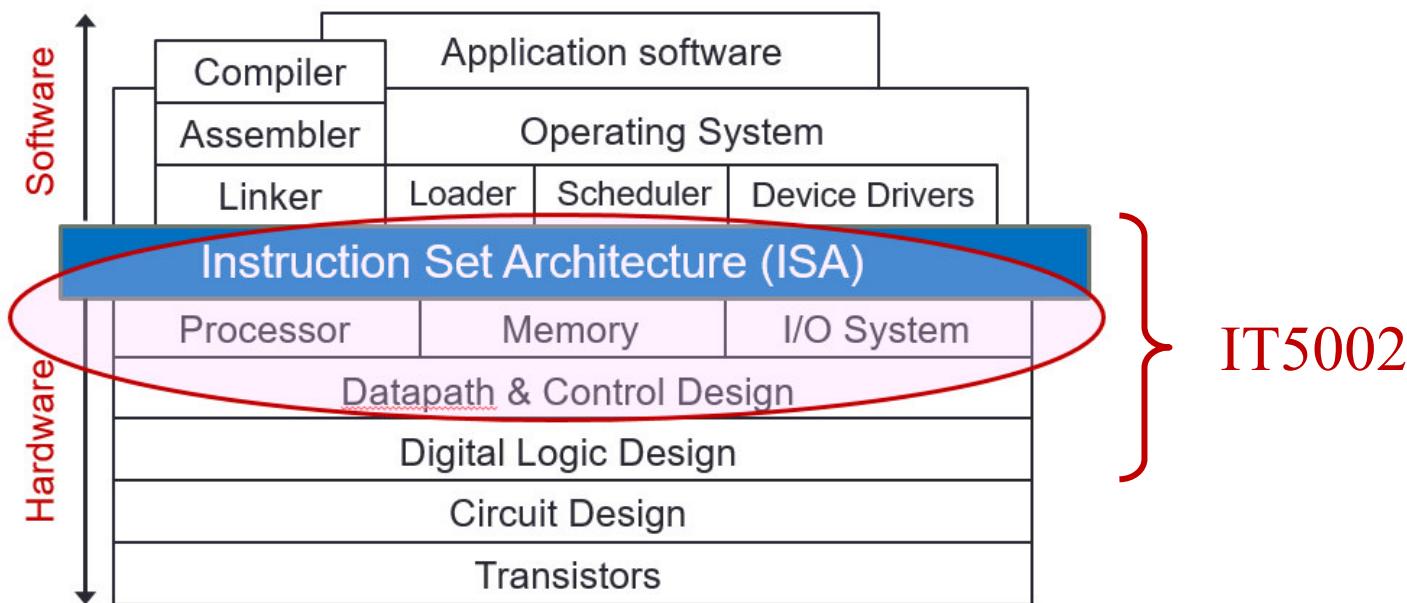


Inside a Pentium chip



IT5002 : It's About Computer Organization

- **Computer organisation** is the study of internal working, structuring and implementation of a computer system.
- It refers to the level of abstraction above the digital logic level, but below the operating system level.



IT5002 : It's About Computer Organization

(From user to builder)

- You want to call yourself a **computer scientist/specialist**.
- You want to **build** software people use.
- You need to make purchasing **decisions**.
- You need to offer “expert” **advice**.
- Hardware and software affect performance
 - Algorithm determines number of source-level statements
 - Language, compiler, and architecture determine machine instructions
(COD chapters 2 and 3)
 - Processor and memory determine how fast instructions are executed
(COD chapters 5, 6 and 7)



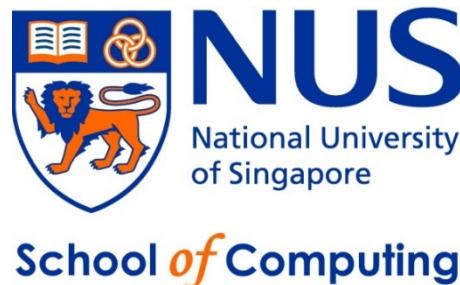
Understanding performance (COD chapter 4)

IT5002

Computer Systems and Applications

Number Systems

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lecture 3: Number Systems

- 1. Data Representation**
- 2. Decimal (base 10) Number System**
- 3. Other Number Systems**
- 4. Base- R to Decimal Conversion**
- 5. Decimal to Binary Conversion**
 - 5.1 Repeated Division-by-2
 - 5.2 Repeated Multiplication-by-2
- 6. Conversion Between Decimal and Other Bases**
- 7. Conversion Between Bases**
- 8. Binary to Octal/Hexadecimal Conversion**



Lecture 3: Number Systems

9. ASCII Code

10. Negative Numbers

- 10.1 Sign-and-Magnitude
- 10.2 1s Complement
- 10.3 2s Complement
- 10.4 Comparisons
- 10.5 Complement on Fractions
- 10.6 2s Complement Addition/Subtraction
- 10.7 1s Complement Addition/Subtraction
- 10.8 Excess Representation

11. Real Numbers

- 11.1 Fixed-Point Representation
- 11.2 Floating-Point Representation



1. Data Representation

Basic data types in C:

int

float

double

char

Variants: short, long

How data is represented depends on its type:

01000110

As an ‘int’, it is 70

As a ‘char’, it is ‘F’

1100000011010000000000000000000000000000

As an ‘int’, it is -1060110336

As an ‘float’, it is -6.5



1. Data Representation

- Data are internally represented as sequence of **bits** (**b**inary **i**ts). A bit is either 0 or 1.
- Other units
 - **Byte**: 8 bits
 - Nibble: 4 bits (rarely used now)
 - **Word**: Multiple of bytes (eg: 1 byte, 2 bytes, 4 bytes, etc.) depending on the computer architecture
- N bits can represent up to 2^N values
 - Eg: 2 bits represent up to 4 values (00, 01, 10, 11);
4 bits represent up to 16 values (0000, 0001, 0010, ..., 1111)
- To represent M values, $\lceil \log_2 M \rceil$ bits required
 - Eg: 32 values require 5 bits; 1000 values require 10 bits



2. Base 10 Number System

- A **weighted-positional** number system.
- **Base** (also called **radix**) is 10
- Symbols/digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Each position has a weight of power of 10
 - Eg: $(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times 10^{-2})$

$$(a_n a_{n-1} \dots a_0 \cdot f_1 f_2 \dots f_m)_{10} = \\ (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \dots + (a_0 \times 10^0) + \\ (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \dots + (f_m \times 10^{-m})$$



3. Other Number Systems

- A **weighted-positional** number system.
- **Base** (also called **radix**) is 10
- Symbols/digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Each position has a weight of power of 10
 - Eg: $(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times 10^{-2})$

$$\begin{aligned}(a_n a_{n-1} \dots a_0 \cdot f_1 f_2 \dots f_m)_{10} = \\ (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \dots + (a_0 \times 10^0) + \\ (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \dots + (f_m \times 10^{-m})\end{aligned}$$



3. Other Number Systems

- In some programming languages/software, special notations are used to represent numbers in certain bases
 - In programming language C
 - Prefix 0 for octal. Eg: 032 represents the octal number $(32)_8$
 - Prefix 0x for hexadecimal. Eg: 0x32 represents the hexadecimal number $(32)_{16}$
 - In QTSpim (a MIPS simulator you will use)
 - Prefix 0x for hexadecimal. Eg: 0x100 represents the hexadecimal number $(100)_{16}$
 - In Verilog, the following values are the same
 - 8'b11110000: an 8-bit binary value 11110000
 - 8'hF0: an 8-bit binary value represented in hexadecimal F0
 - 8'd240: an 8-bit binary value represented in decimal 240



4. Base-R to Decimal Conversion

- Easy!

- $1101.101_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3}$
 $= 8 + 4 + 1 + 0.5 + 0.125 = \mathbf{13.625}_{10}$
- $572.6_8 = 5 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 + 6 \times 8^{-1}$
 $= 320 + 56 + 2 + 0.75 = \mathbf{378.75}_{10}$
- $2A.8_{16} = 2 \times 16^1 + 10 \times 16^0 + 8 \times 16^{-1}$
 $= 32 + 10 + 0.5 = \mathbf{42.5}_{10}$
- $341.24_5 = 3 \times 5^2 + 4 \times 5^1 + 1 \times 5^0 + 2 \times 5^{-1} + 4 \times 5^{-2}$
 $= 75 + 20 + 1 + 0.4 + 0.16 = \mathbf{96.56}_{10}$



5. Decimal to Binary (Base-2) Conversion

- For whole numbers
 - Repeated Division-by-2 Method
- For fractions
 - Repeated Multiplication-by-2 Method



5.1 Decimal to Binary (Base-2) Conversion: Repeated Divide

- To convert a **whole number** to binary, use **successive division by 2** until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$$(43)_{10} = (\quad ? \quad)_2$$



5.1 Decimal to Binary (Base-2) Conversion: Repeated Divide

- To convert a **whole number** to binary, use **successive division by 2** until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$$(43)_{10} = (\textcolor{red}{101011})_2$$

2	43	
2	21 rem 1	← LSB
2	10 rem 1	
2	5 rem 0	
2	2 rem 1	
2	1 rem 0	
0	rem 1	← MSB



5.2 Decimal to Binary (Base-2) Conversion: Repeated Multiply

- To convert **decimal fractions** to binary, **repeated multiplication by 2** is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$$(0.3125)_{10} = (?)_2$$



5.2 Decimal to Binary (Base-2) Conversion: Repeated Multiply

- To convert decimal fractions to binary, **repeated multiplication by 2** is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$$(0.3125)_{10} = (.0101)_2$$

Carry
0 ←MSB
1
0
1 ←LSB

$0.3125 \times 2 = 0.625$

$0.625 \times 2 = 1.25$

$0.25 \times 2 = 0.50$

$0.5 \times 2 = 1.00$



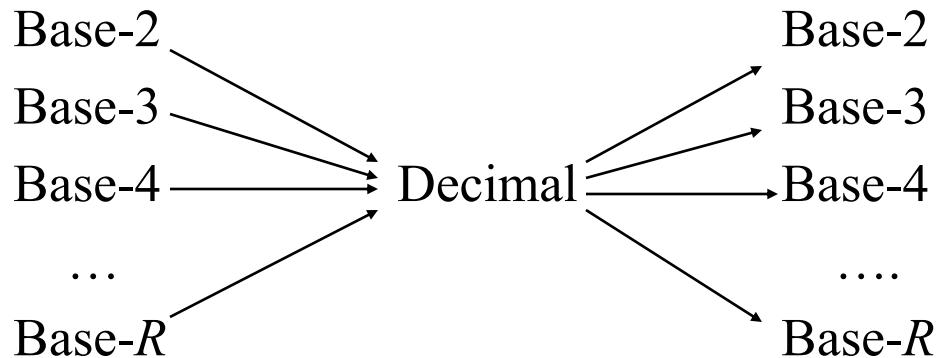
6. Conversion from Decimal to Other Bases

- **Base- R to decimal:** multiply digits with their corresponding weights
- **Decimal to binary (base 2)**
 - Whole numbers: repeated division-by-2
 - Fractions: repeated multiplication-by-2
- **Decimal to base- R**
 - Whole numbers: repeated division-by- R
 - Fractions: repeated multiplication-by- R



7. Conversion Between Bases

- In general, conversion between bases can be done via decimal:



- Shortcuts for conversion between bases 2, 4, 8, 16 (see next slide)



8. Binary to Octal / Hexadecimal Conversion

- **Binary → Octal:** partition in groups of 3
 - $(10\ 111\ 011\ 001\ .\ 101\ 110)_2 = \text{(2731.56)}_8$
- **Octal → Binary:** reverse
 - $(2731.56)_8 = \text{(10 111 011 001 . 101 110)}_2$
- **Binary → Hexadecimal:** partition in groups of 4
 - $(101\ 1101\ 1001\ .\ 1011\ 1000)_2 = \text{(5D9.B8)}_{16}$
- **Hexadecimal → Binary:** reverse
 - $(5D9.B8)_{16} = \text{(101 1101 1001 . 1011 1000)}_2$





9. ASCII Code

- **ASCII code and Unicode** are used to represent characters ('a', 'C', '?', '\0', etc.)
- ASCII
 - American Standard Code for Information Interchange
 - 7 bits, plus 1 parity bit (odd or even parity)

Character	ASCII Code
0	0110000
1	0110001
...	...
9	0111001
:	0111010
A	1000001
B	1000010
...	...
Z	1011010
[1011011
\	1011100



9. ASCII Code

ASCII table

LSBs	MSBs								
	000	001	010	011	100	101	110	111	
0000	NUL	DLE	SP	0	@	P	`	p	
0001	SOH	DC ₁	!	1	A	Q	a	q	
0010	STX	DC ₂	"	2	B	R	b	r	
0011	ETX	DC ₃	#	3	C	S	c	s	
0100	EOT	DC ₄	\$	4	D	T	d	t	
0101	ENQ	NAK	%	5	E	U	e	u	
0110	ACK	SYN	&	6	F	V	f	v	
0111	BEL	ETB	'	7	G	W	g	w	
1000	BS	CAN	(8	H	X	h	x	
1001	HT	EM)	9	I	Y	i	y	
1010	LF	SUB	*	:	J	Z	j	z	
1011	VT	ESC	+	;	K	[k	{	
1100	FF	FS	,	<	L	\	l		
1101	CR	GS	-	=	M]	m	}	
1110	O	RS	.	>	N	^	n	~	
1111	SI	US	/	?	O	_	o	DEL	

‘A’: 1000001
 (or 65₁₀)



9. ASCII Code

01000110

As an ‘int’, it is 70

As a ‘char’, it is ‘F’

- Integers (0 to 127) and characters are ‘somewhat’ interchangeable in C

```
int num = 65;
char ch = 'F';
```

CharAndInt.c

```
printf("num (in %%d) = %d\n", num);
printf("num (in %%c) = %c\n", num);
printf("\n");
```

```
num (in %d) = 65
num (in %c) = A

ch (in %c) = F
ch (in %d) = 70
```

```
printf("ch (in %%c) = %c\n", ch);
printf("ch (in %%d) = %d\n", ch);
```





PAST YEAR QUESTION

PastYearQn.c

```
int i, n = 2147483640;
for (i=1; i<=10; i++) {
    n = n + 1;
}
printf("n = %d\n", n);
```

- What is the output of the above code when run on sunfire?
- Is it 2147483650?



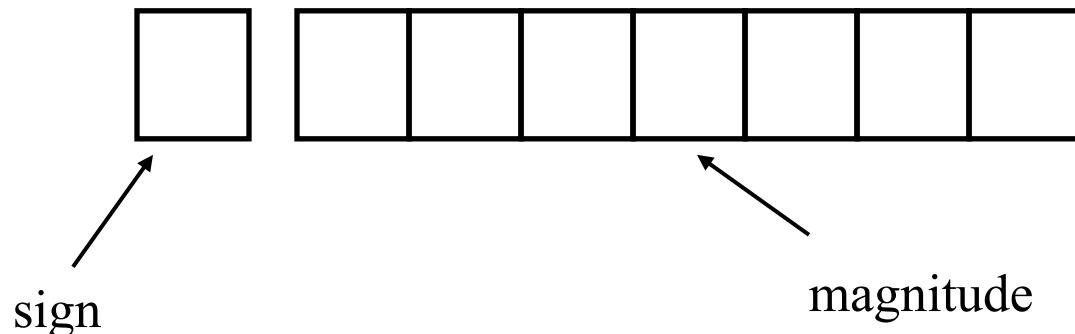
10. Negative Numbers

- Unsigned numbers: only non-negative values
- Signed numbers: include all values (positive and negative)
- There are 3 common representations for signed binary numbers:
 - Sign-and-Magnitude
 - 1s Complement
 - 2s Complement



10.1 Negative Numbers: Sign and Magnitude

- The sign is represented by a ‘sign bit’
 - 0 for +
 - 1 for -
- Eg: a 1-bit sign and 7-bit magnitude format.



- 00110100 → $+110100_2 = +52_{10}$
- 10010011 → $-10011_2 = -19_{10}$



10.1 Negative Numbers: Sign and Magnitude

- Largest value: $01111111 = +127_{10}$
- Smallest value: $11111111 = -127_{10}$
- Zeros:
 - $00000000 = +0_{10}$
 - $10000000 = -0_{10}$
- Range (for 8-bit): -127_{10} to $+127_{10}$
- Question:
 - For an n -bit sign-and-magnitude representation, what is the range of values that can be represented?



10.1 Negative Numbers: Sign and Magnitude

- To negate a number, just invert the sign bit.
- Examples:
 - How to negate 00100001_{sm} (decimal 33)?
Answer: 10100001_{sm} (decimal -33)
 - How to negate 10000101_{sm} (decimal -5)?
Answer: 00000101_{sm} (decimal +5)



10.2 Negative Numbers: 1's Complement

- Given a number x which can be expressed as an n -bit binary number, its negated value can be obtained in **1s-complement** representation using:

$$-x = 2^n - x - 1$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

$$\begin{aligned}-00001100_2 &= 2^8 - 12 - 1 \text{ (calculation done in decimal)} \\ &= 243 \\ &= 11110011_{1s}\end{aligned}$$

(This means that -12_{10} is written as 11110011 in 1s-complement representation.)



10.2 Negative Numbers: 1's Complement

- Technique to negate a value: **invert all the bits.**
- Largest value: $01111111 = +127_{10}$
- Smallest value: $10000000 = -127_{10}$
- Zeros: $00000000 = +0_{10}$
 $11111111 = -0_{10}$
- Range (for 8 bits): -127_{10} to $+127_{10}$
- Range (for n bits): $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- The **most significant bit (MSB)** still represents the sign: 0 for positive, 1 for negative.



10.2 Negative Numbers: 1's Complement

- Examples (assuming 8-bit):

$$(14)_{10} = (00001110)_2 = (00001110)_{1s}$$

$$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$$

$$-(80)_{10} = -(?)_2 = (?)_{1s}$$



10.3 Negative Numbers: 2's Complement

- Given a number x which can be expressed as an n -bit binary number, its negated value can be obtained in **2s-complement** representation using:

$$-x = 2^n - x$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 2s-complement is:

$$\begin{aligned}-00001100_2 &= 2^8 - 12 \text{ (calculation done in decimal)} \\ &= 244 \\ &= 11110100_{2s}\end{aligned}$$

(This means that -12_{10} is written as 11110100 in 2s-complement representation.)



10.3 Negative Numbers: 2's Complement

- Technique to negate a value: **invert all the bits, then add 1.**
- Largest value: $0111111 = +127_{10}$
- Smallest value: $10000000 = -128_{10}$
- Zero: $00000000 = +0_{10}$
- Range (for 8 bits): -128_{10} to $+127_{10}$
- Range (for n bits): -2^{n-1} to $2^{n-1} - 1$
- The **most significant bit (MSB)** still represents the sign: 0 for positive, 1 for negative.



10.3 Negative Numbers: 2's Complement

- Examples (assuming 8-bit):

$$(14)_{10} = (00001110)_2 = (00001110)_{2s}$$

$$-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$$

$$-(80)_{10} = -(?)_2 = (?)_{2s}$$

Compare with slide 29.

- 1s complement:

$$(14)_{10} = (00001110)_2 = (00001110)_{1s}$$

$$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$$



10.4 Negative Numbers: Comparison

4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000



Past-Year's Exam Question! (Answer)

PastYearQn.c

```
int i, n = 2147483640;
for (i=1; i<=10; i++) {
    n = n + 1;
}
printf("n = %d\n", n);
```

- **int** type in sunfire takes up 4 bytes (32 bits) and uses 2s complement
- Largest positive integer = $2^{31} - 1 = 2147483647$

- What is the output of the above code when run on sunfire?
- Is it 2147483650? **X**

1st iteration: n = 2147483641

7th iteration: n = 2147483647

01111 1111111111
+ 1

10000.....0000000000

8th iteration: n = -2147483648

9th iteration: n = -2147483647

10th iteration: n = -2147483646



10.5 Negative Numbers: Complements on Fractions

- We can extend the idea of complement on fractions.
- Examples:
 - Negate 0101.01 in 1s-complement
Answer: 1010.10
 - Negate 111000.101 in 1s-complement
Answer: 000111.010
 - Negate 0101.01 in 2s-complement
Answer: 1010.11



10.6 Negative Numbers: 2's Complements on Additions and Subtractions

- **Algorithm for addition of integers, A + B:**
 1. Perform binary addition on the two numbers.
 2. Ignore the carry out of the MSB.
 3. **Check for overflow.** Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.
- **Algorithm for subtraction of integers, A – B:**
$$A - B = A + (-B)$$
 1. Take 2s-complement of B.
 2. Add the 2s-complement of B to A.



10.6 Negative Numbers: Overflow

- Signed numbers are of a fixed range.
- If the result of addition/subtraction goes beyond this range, an **overflow** occurs.
- Overflow can be easily detected:
 - *positive add positive → negative*
 - *negative add negative → positive*
- Example: 4-bit 2s-complement system
 - Range of value: -8_{10} to 7_{10}
 - $0101_{2s} + 0110_{2s} = 1011_{2s}$
 $5_{10} + 6_{10} = -5_{10}$?! (**overflow!**)
 - $1001_{2s} + 1101_{2s} = \underline{10110}_{2s}$ (discard end-carry) = 0110_{2s}
 $-7_{10} + -3_{10} = 6_{10}$?! (**overflow!**)



10.6 Negative Numbers: Overflow

- Examples: 4-bit system

+3	0011
+ 4	+ 0100
-----	-----
+7	0111
-----	-----

No overflow

+6	0110
+ -3	+ 1101
-----	-----
+3	10011
-----	-----

No overflow

-3	1101
+ -6	+ 1010
-----	-----
-9	10111
-----	-----

Overflow!

-2	1110
+ -6	+ 1010
-----	-----
-8	11000
-----	-----

No overflow

+4	0100
+ -7	+ 1001
-----	-----
-3	1101
-----	-----

No overflow

+5	0101
+ +6	+ 0110
-----	-----
+11	1011
-----	-----

Overflow!

- Which of the above is/are overflow(s)?



10.6 Negative Numbers: Overflow

- Examples: 4-bit system

- $4 - 7$
- Convert it to $4 + (-7)$
- $6 - 1$
- Convert it to $6 + (-1)$
- $-5 - 4$
- Convert it to $-5 + (-4)$

$$\begin{array}{r}
 +4 & 0100 \\
 + -7 & + 1001 \\
 \hline
 -3 & 1101
 \end{array}$$

No overflow

$$\begin{array}{r}
 +6 & 0110 \\
 + -1 & + 1111 \\
 \hline
 +5 & 10101
 \end{array}$$

No overflow

$$\begin{array}{r}
 -5 & 1011 \\
 + -4 & + 1100 \\
 \hline
 -9 & 10111
 \end{array}$$

Overflow!



Which of the above is/are overflow(s)?

10.7 Negative Numbers: 1's Complement on Additions and Subtractions.

- **Algorithm for addition of integers, A + B:**
 1. Perform binary addition on the two numbers.
 2. If there is a carry out of the MSB, add 1 to the result.
 3. Check for overflow. Overflow occurs if result is opposite sign of A and B.
- **Algorithm for subtraction of integers, A – B:**
$$A - B = A + (-B)$$
 1. Take 1s-complement of B.
 2. Add the 1s-complement of B to A.



10.7 Negative Numbers: 1's Complement Addition

- Examples: 4-bit system

$$\begin{array}{r}
 +3 & 0011 \\
 + +4 & + 0100 \\
 \hline
 - & \hline \\
 +7 & 0111 \\
 \hline
 - & \hline
 \end{array}$$

No overflow

$$\begin{array}{r}
 +5 & 0101 \\
 + -5 & + 1010 \\
 \hline
 - & \hline \\
 -0 & 1111 \\
 \hline
 - & \hline
 \end{array}$$

No overflow

$$\begin{array}{r}
 -2 & 1101 \\
 + -5 & + 1010 \\
 \hline
 - & \hline \\
 -7 & \textcolor{red}{1}0111 \\
 \hline
 - & + 1 \\
 \hline
 & \hline \\
 1000 & \\
 \hline
 - & \hline
 \end{array}$$

No overflow

$$\begin{array}{r}
 -3 & 1100 \\
 + -7 & + 1000 \\
 \hline
 - & \hline \\
 -10 & \textcolor{red}{1}0100 \\
 \hline
 - & + 1 \\
 \hline
 & \hline \\
 0101 & \\
 \hline
 - & \hline
 \end{array}$$

Overflow!



Any overflow?

DLD page 42 – 43 Quick Review Questions
 Questions 2-13 to 2-18.

10.8 Negative Numbers: Excess Notation

- Besides sign-and-magnitude and complement schemes, the **excess representation** is another scheme.
- It allows the range of values to be distributed evenly between the positive and negative values, by a simple translation (addition/subtraction).
- Example: Excess-4 representation on 3-bit numbers. See table on the right.

<i>Excess-4 Representation</i>	<i>Value</i>
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3



Questions: What if we use Excess-2 on 3-bit numbers? Or Excess-7?

10.8 Negative Numbers: Excess Notation

- Example: For 4-bit numbers, we may use excess-7 or excess-8. Excess-8 is shown below.

<i>Excess-8 Representation</i>	<i>Value</i>
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

<i>Excess-8 Representation</i>	<i>Value</i>
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7



11 Real Numbers

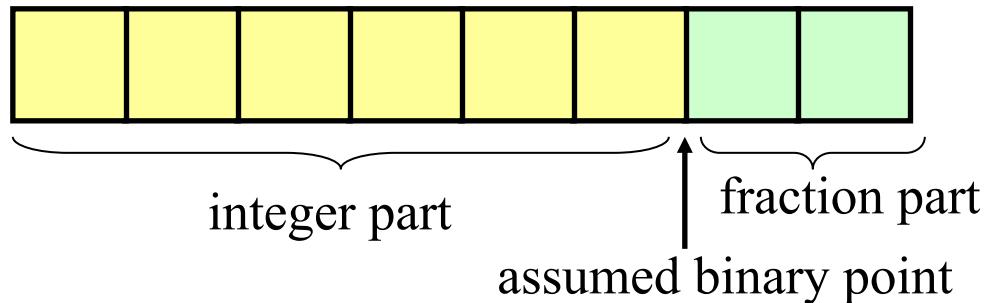
- Many applications involve computations not only on integers but also on real numbers.
- How are real numbers represented in a computer system?
- Due to the finite number of bits, real number are often represented in their approximate values.



11.1 Real Numbers

Fixed Point Representation

- In **fixed-point representation**, the number of bits allocated for the whole number part and fractional part are fixed.
- For example, given an 8-bit representation, 6 bits are for whole number part and 2 bits for fractional parts.



- If 2s complement is used, we can represent values like:

$$011010.11_{2s} = 26.75_{10}$$

$$111110.11_{2s} = -000001.01_2 = -1.25_{10}$$



11.2 Real Numbers

Floating Point Representation

- Fixed-point representation has limited range.
- Alternative: **Floating point numbers** allow us to represent very large or very small numbers.
- Examples:

0.23×10^{23} (very large positive number)

0.5×10^{-37} (very small positive number)

-0.2397×10^{-18} (very small negative number)



11.2 Real Numbers

Floating Point Representation

- 3 components: **sign**, **exponent** and **mantissa (fraction)**



- The base (radix) is assumed to be 2.
- Two formats:
 - Single-precision (32 bits):** 1-bit sign, 8-bit exponent with bias 127 (excess-127), 23-bit mantissa
 - Double-precision (64 bits):** 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), and 52-bit mantissa
- We will focus on the single-precision format



11.2 Real Numbers

Floating Point Representation

- 3 components: **sign**, **exponent** and **mantissa (fraction)**

sign	exponent	mantissa
------	----------	----------

- Sign bit: 0 for positive, 1 for negative.
- Mantissa is **normalised** with an implicit leading bit 1
 - $110.1_2 \rightarrow \text{normalised} \rightarrow 1.\underline{101}_2 \times 2^2 \rightarrow$ only **101** is stored in the mantissa field
 - $0.00101101_2 \rightarrow \text{normalised} \rightarrow 1.\underline{01101}_2 \times 2^{-3} \rightarrow$ only **01101** is stored in the mantissa field



11.2 Real Numbers

Floating Point Representation

- Example: How is -6.5_{10} represented in IEEE 754 single-precision floating-point format?

$$-6.5_{10} = -110.1_2 = \textcolor{blue}{-}1.\boxed{101}_2 \times 2^{\textcolor{green}{2}\textcolor{blue}{0}}$$

$$\text{Exponent} = 2 + 127 = 129 = 10000001_2$$

1	10000001	1010000000000000000000000000
sign	exponent (excess-127)	mantissa

- We may write the 32-bit representation in hexadecimal:

$$1\ 10000001\ 1010000000000000000000000000_2 = \textcolor{red}{C0D00000}_{16}$$

(Slide 4)

11000000110100000000000000000000

 As an ‘int’, it is **-1060110336**

 As an ‘float’, it is **-6.5**


IT5002

Computer Systems and Applications

MIPS Assembly I

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lesson Plan

1. Welcome – This Lecture
2. Introduction – Basic Ideas
3. Number Systems – Everything is numbers!
4. C and Hardware
- 5. MIPS Assembly Language**
6. Instruction Set Architecture Design
7. Datapath Design
8. Processor Control
9. Boolean Algebra
10. Simplification Methods
11. Combinational Circuit Design
12. MSI Components
13. Sequential Circuit Design



Lecture #7: MIPS Part 1: Introduction (1/2)

1. Instruction Set Architecture
2. Machine Code vs Assembly Language
3. Walkthrough
4. General Purpose Registers
5. MIPS Assembly Language
 - 5.1 General Instruction Syntax
 - 5.2 Arithmetic Operation: Addition
 - 5.3 Arithmetic Operation: Subtraction
 - 5.4 Complex Expression
 - 5.5 Constant/Immediate Operands
 - 5.6 Register Zero (\$0 or \$zero)



Lecture #7: MIPS Part 1: Introduction (1/2)

5. MIPS Assembly Language

- 5.7 Logical Operations: Overview
- 5.8 Logical Operations: Shifting
- 5.9 Logical Operations: Bitwise AND
- 5.10 Logical Operations: Bitwise OR
- 5.11 Logical Operations: Bitwise NOR
- 5.12 Logical Operations: Bitwise XOR

6. Large Constant: Case Study

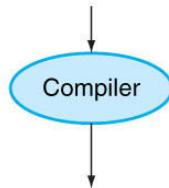
7. MIPS Basic Instructions Checklist



Recap

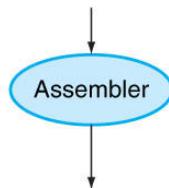
High-level
language
program
(in C)

```
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:  
    muli $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
```



- You write programs in high level programming languages, e.g., C/C++, Java:

$$\mathbf{A} + \mathbf{B}$$

- Compiler translates this into assembly language statement:

add A, B

- Assembler translates this statement into machine language instructions that the processor can execute:

1000 1100 1010 0000

1. Instruction Set Architecture (1/2)

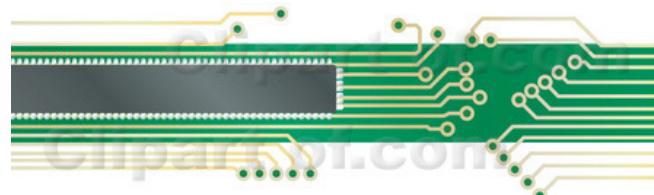
- **Instruction Set Architecture (ISA):**
 - An abstraction on the **interface** between the hardware and the low-level software.

Software
(to be translated to the instruction set)



Instruction Set Architecture

Hardware
(implementing the instruction set)



1. Instruction Set Architecture (2/2)

- Instruction Set Architecture
 - Includes everything programmers need to know to make the machine code work correctly
 - Allows computer designers to talk about functions independently from the hardware that performs them
- This abstraction allows many implementations of varying cost and performance to run identical software.
 - Example: Intel x86/IA-32 ISA has been implemented by a range of processors starting from 80386 (1985) to Pentium 4 (2005)
 - Other companies such as AMD and Transmeta have implemented IA-32 ISA as well
 - A program compiled for IA-32 ISA can execute on any of these implementations



2. Machine Code vs Assembly Language

■ Machine code

- Instructions are represented in binary
- **1000110010100000** is an instruction that tells one computer to add two numbers
- Hard and tedious for programmer

■ Assembly language

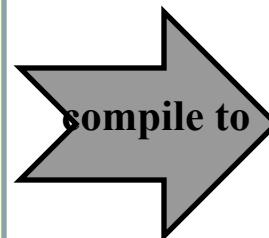
- Symbolic version of machine code
- Human readable
- **add A, B** is equivalent to **1000110010100000**
- **Assembler** translates from assembly language to machine code
- Assembly can provide ‘**pseudo-instructions**’ as syntactic sugar
- When considering performance, only real instructions are counted.



3. Walkthrough: An Example Code (1/15)

- Let us take a journey with the execution of a simple code:
 - Discover the components in a typical computer
 - Learn the type of instructions required to control the processor
 - Simplified to highlight the important concepts ☺

```
// assume res is 0 initially
for (i=1; i<10; i++) {
    res = res + i;
}
```



```
res ← res + i
i ← i + 1
if i < 10, repeat
```

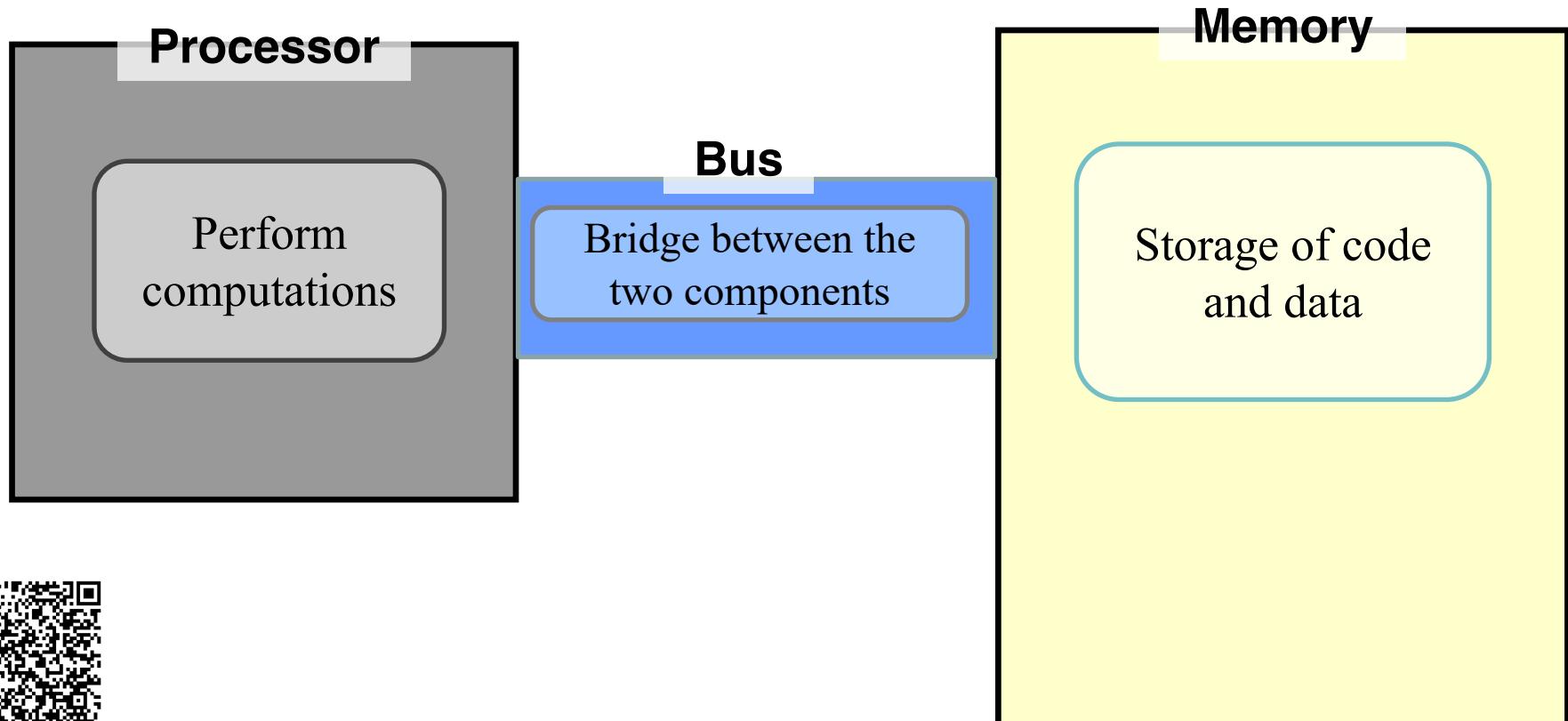
C-like code
fragment

“Assembly”
Code



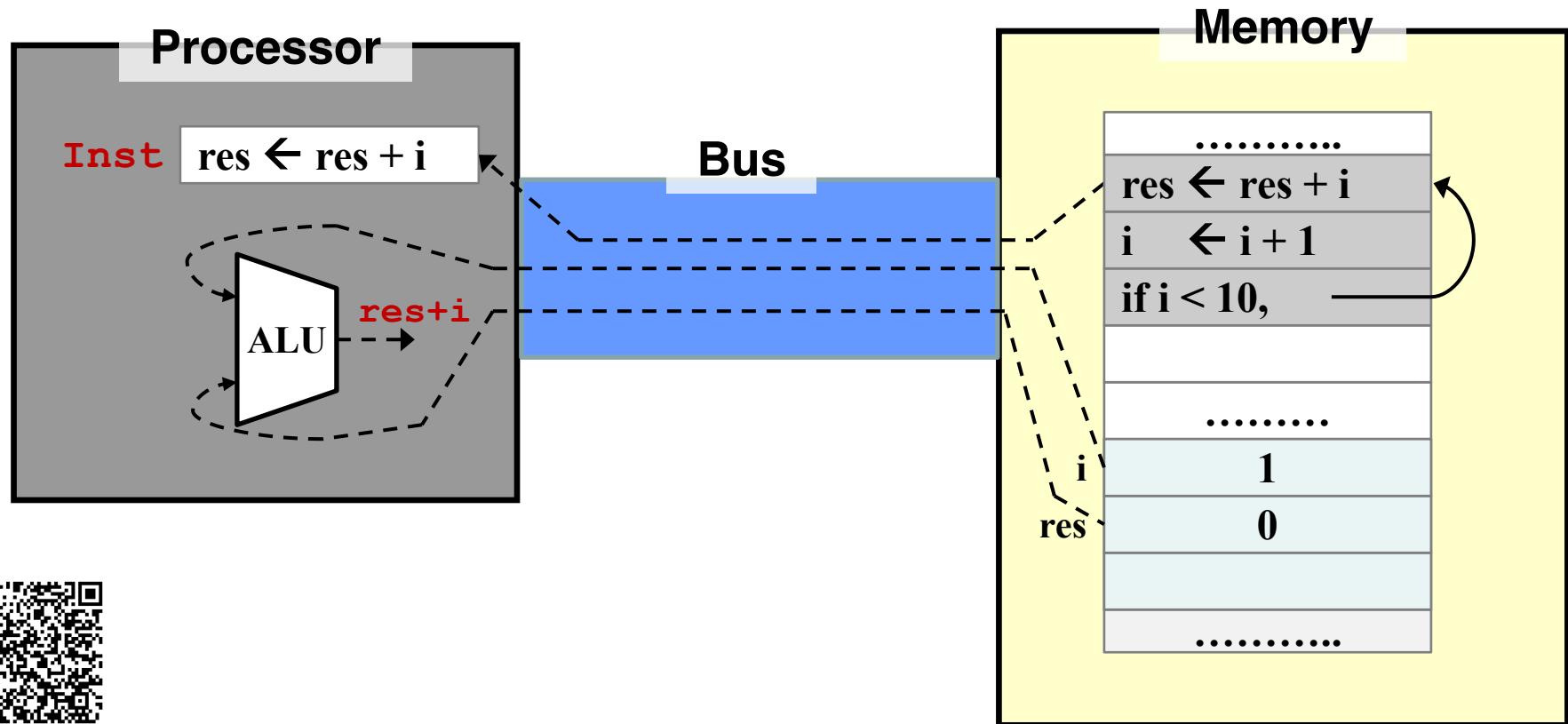
3. Walkthrough: The Components (2/15)

- **The two major components in a computer**
 - **Processor** and **Memory**
 - Input/Output devices omitted in this example



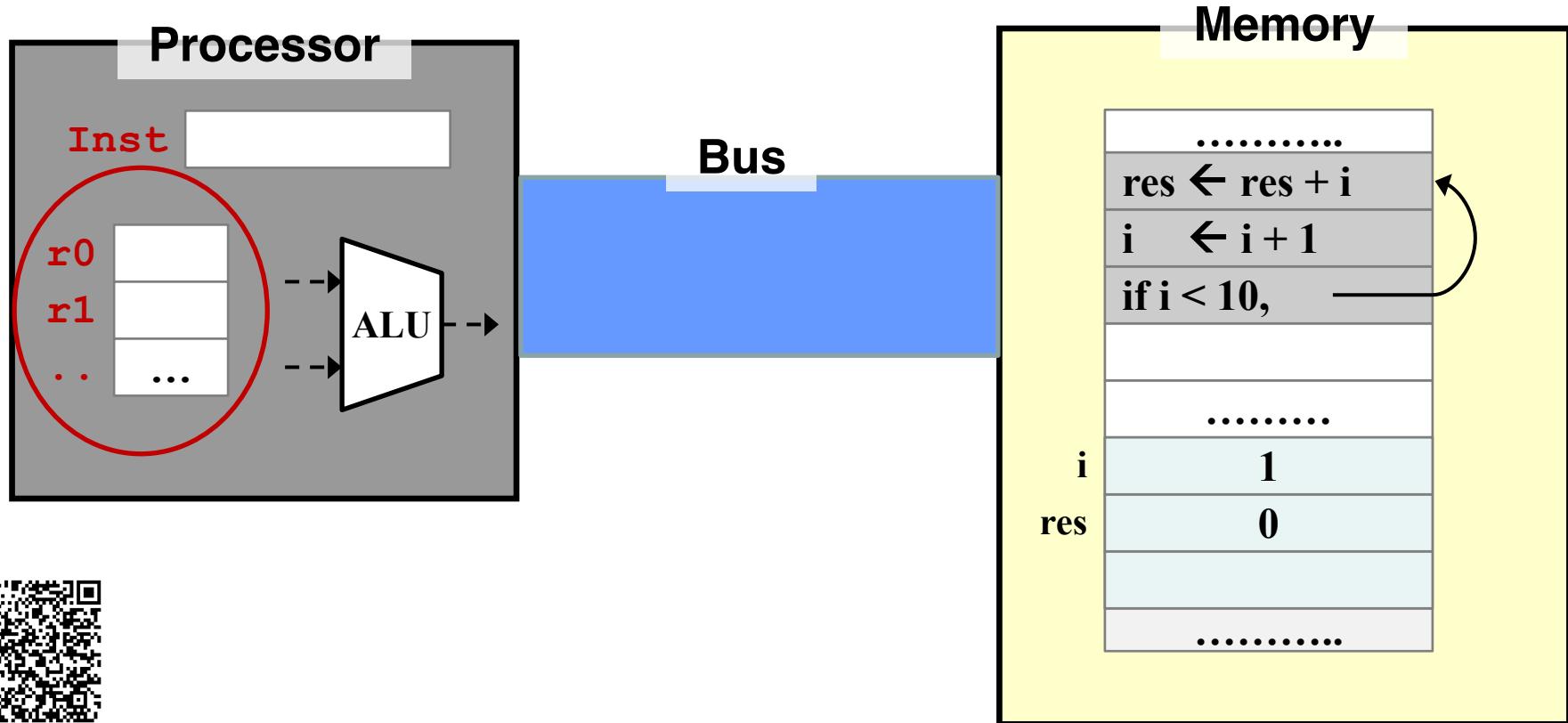
3. Walkthrough: The Code in Action (3/15)

- **The code and data reside in memory**
 - Transferred into the processor during execution



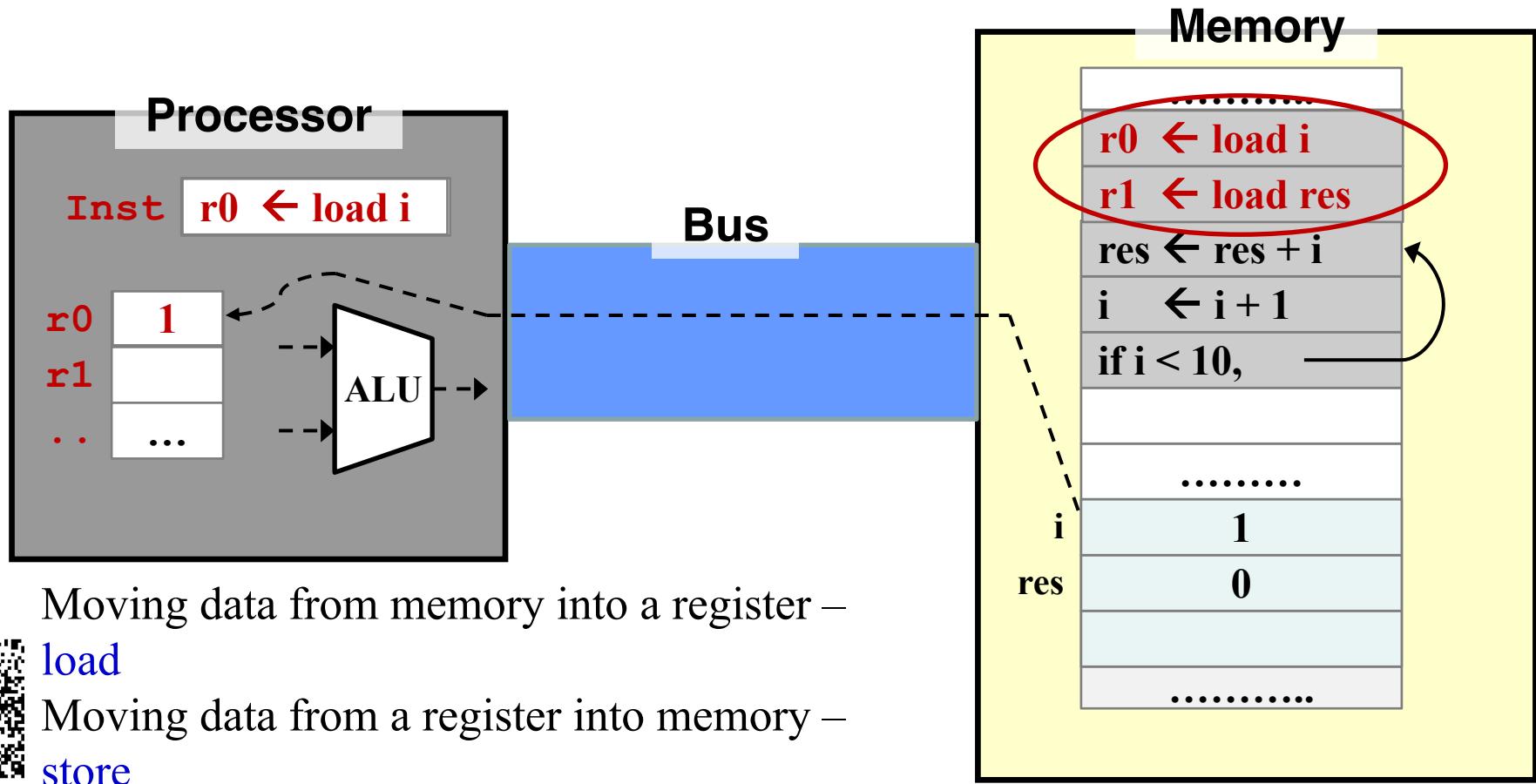
3. Walkthrough: Memory Access is Slow! (4/15)

- To avoid frequent access of memory
 - Provide temporary storage for values in the processor (known as **registers**)



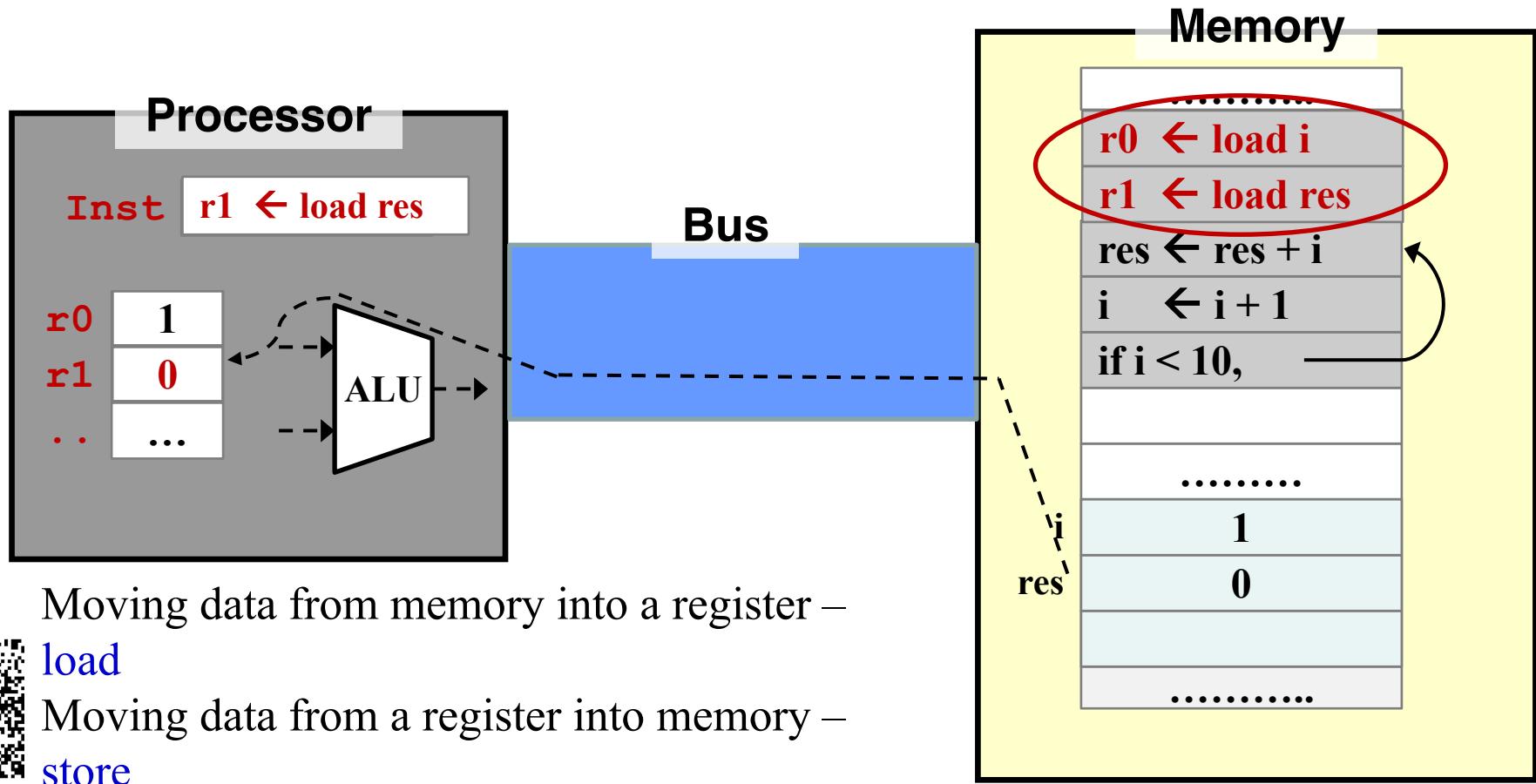
3. Walkthrough: Memory Instruction (5/15)

- Need instruction to move data into registers
 - Also to move data from registers to memory later



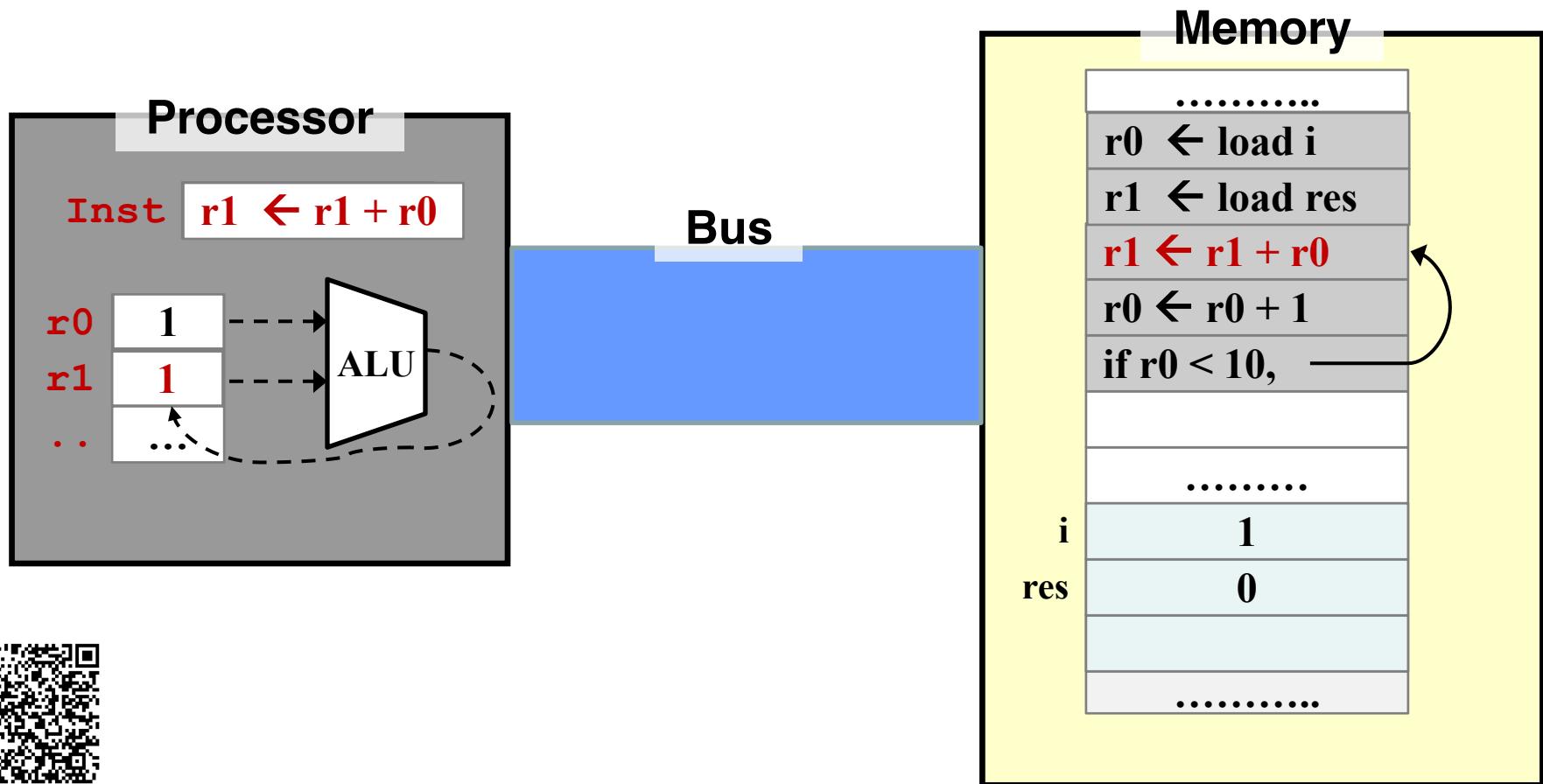
3. Walkthrough: Memory Instruction (6/15)

- Need instruction to move data into registers
 - Also to move data from registers to memory later



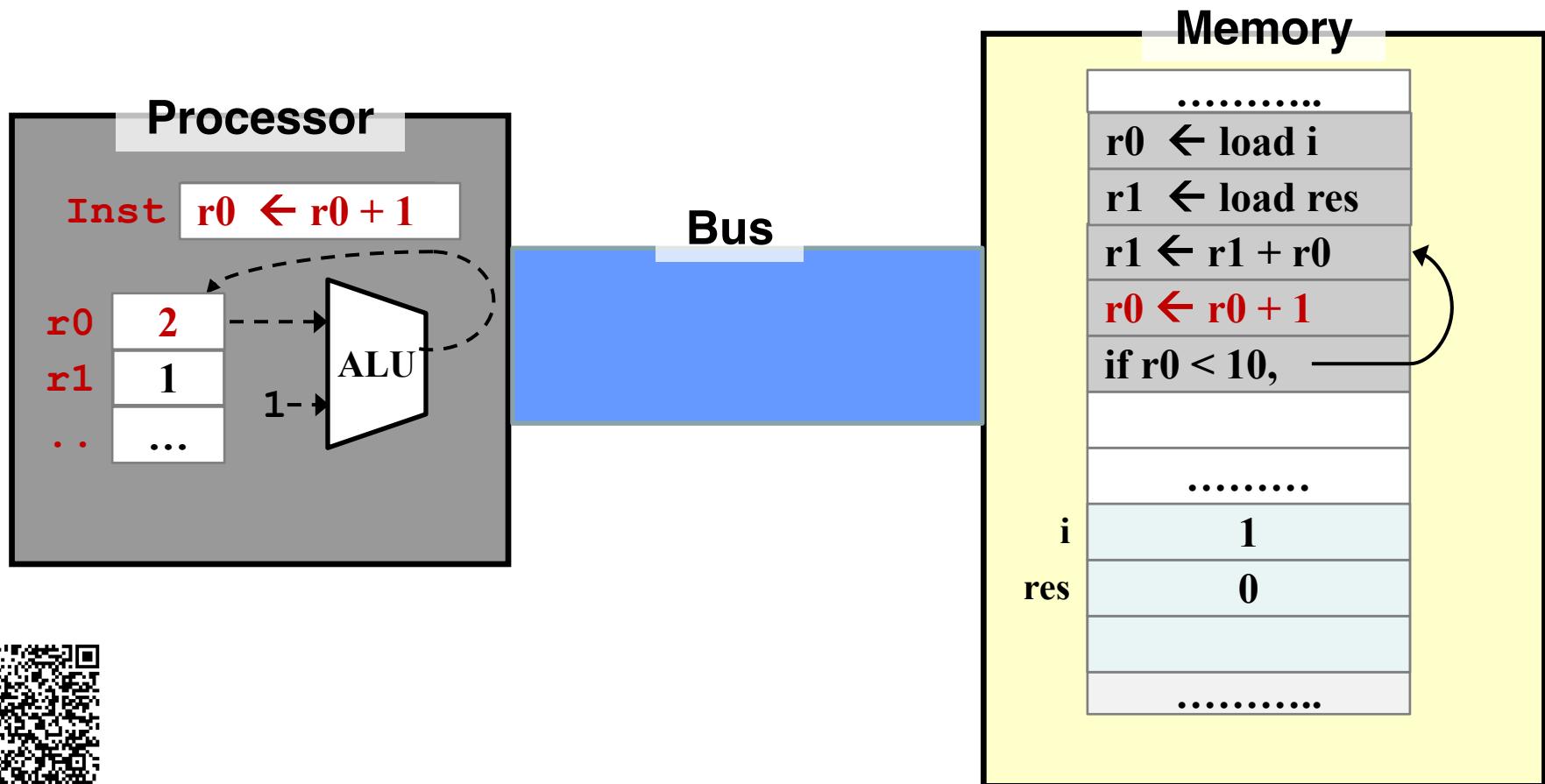
3. Walkthrough: Reg-to-Reg Arithmetic (7/15)

- Arithmetic operations can now work directly on registers only (much faster!)



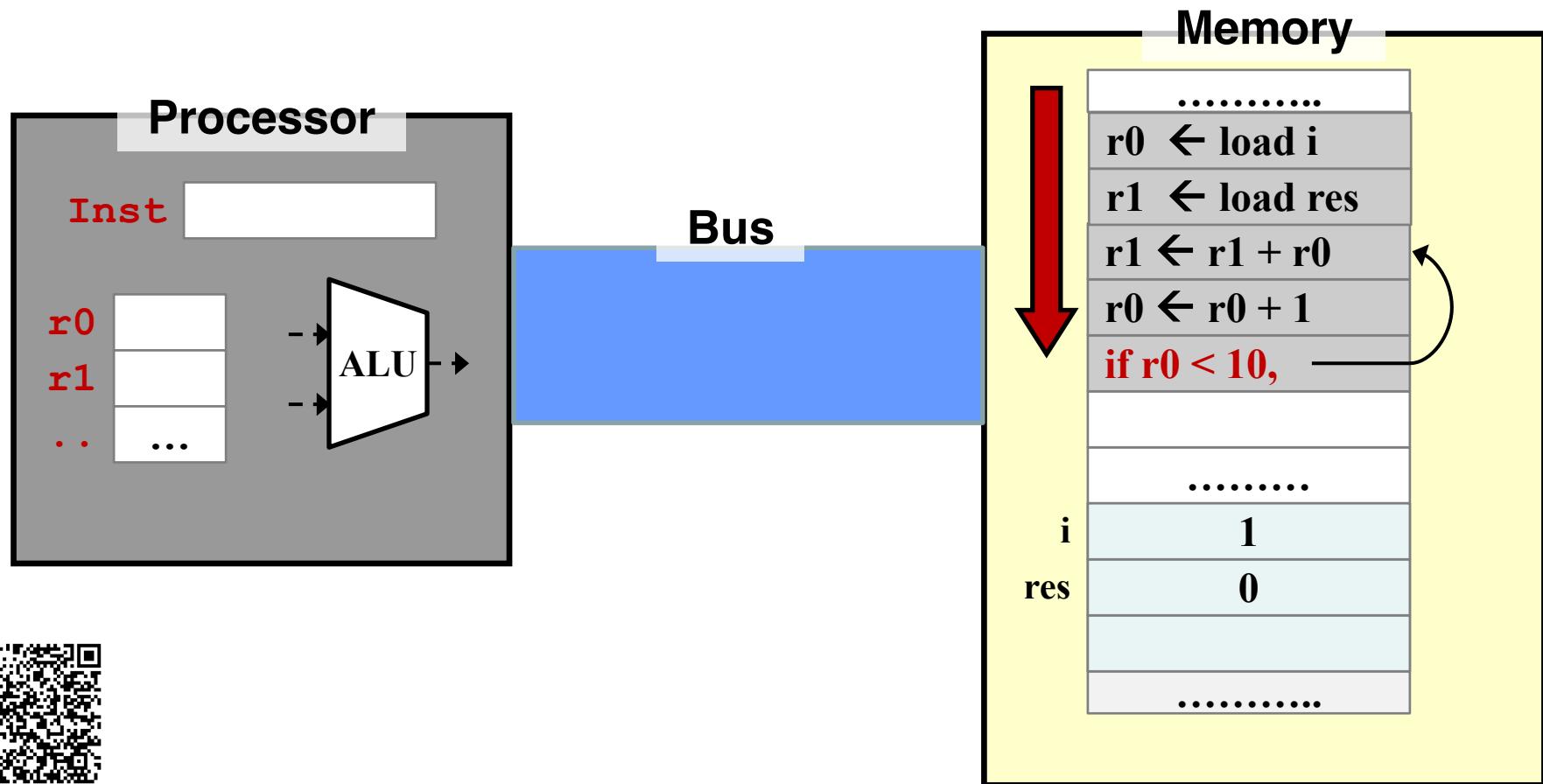
3. Walkthrough: Reg-to-Reg Arithmetic (8/15)

- Sometimes, arithmetic operation uses a **constant value instead of register value**



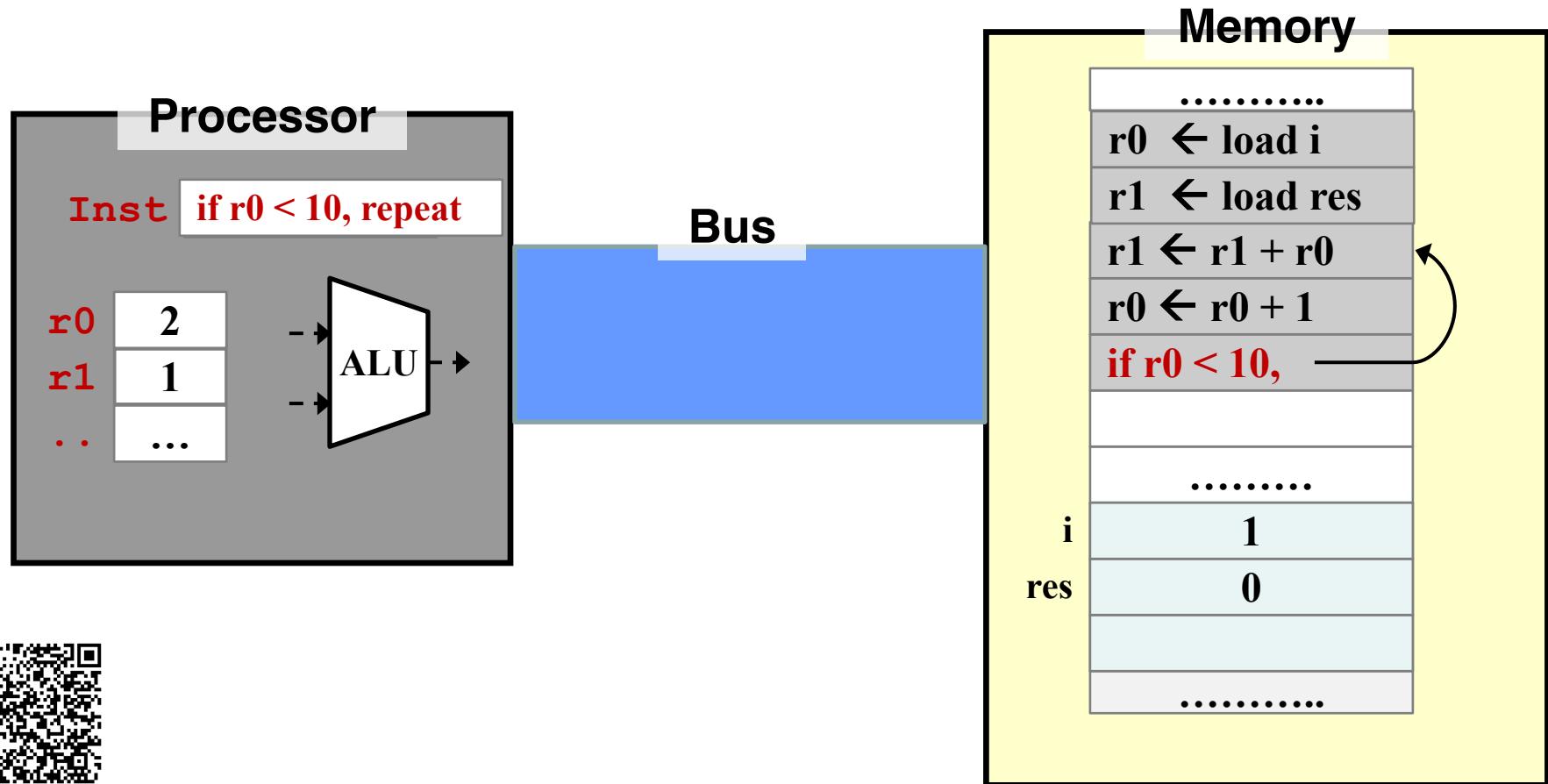
3. Walkthrough: Execution Sequence (9/15)

- Instruction is executed sequentially by default
 - How do we “repeat” or “make a choice”?



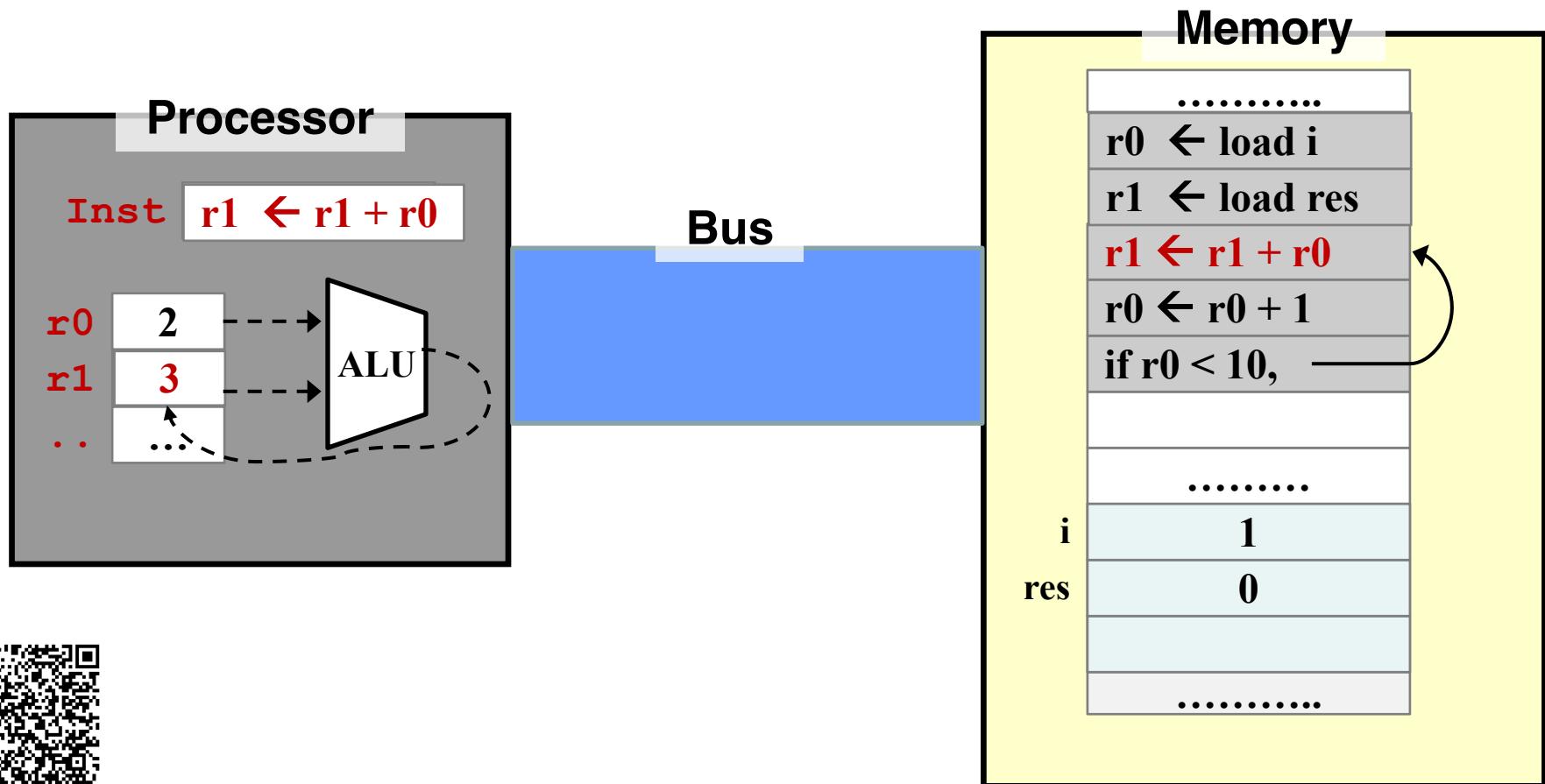
3. Walkthrough: Control Flow (10/15)

- We need instructions to change the control flow based on condition:
 - Repetition (loop) and Selection (if-else) can both be supported



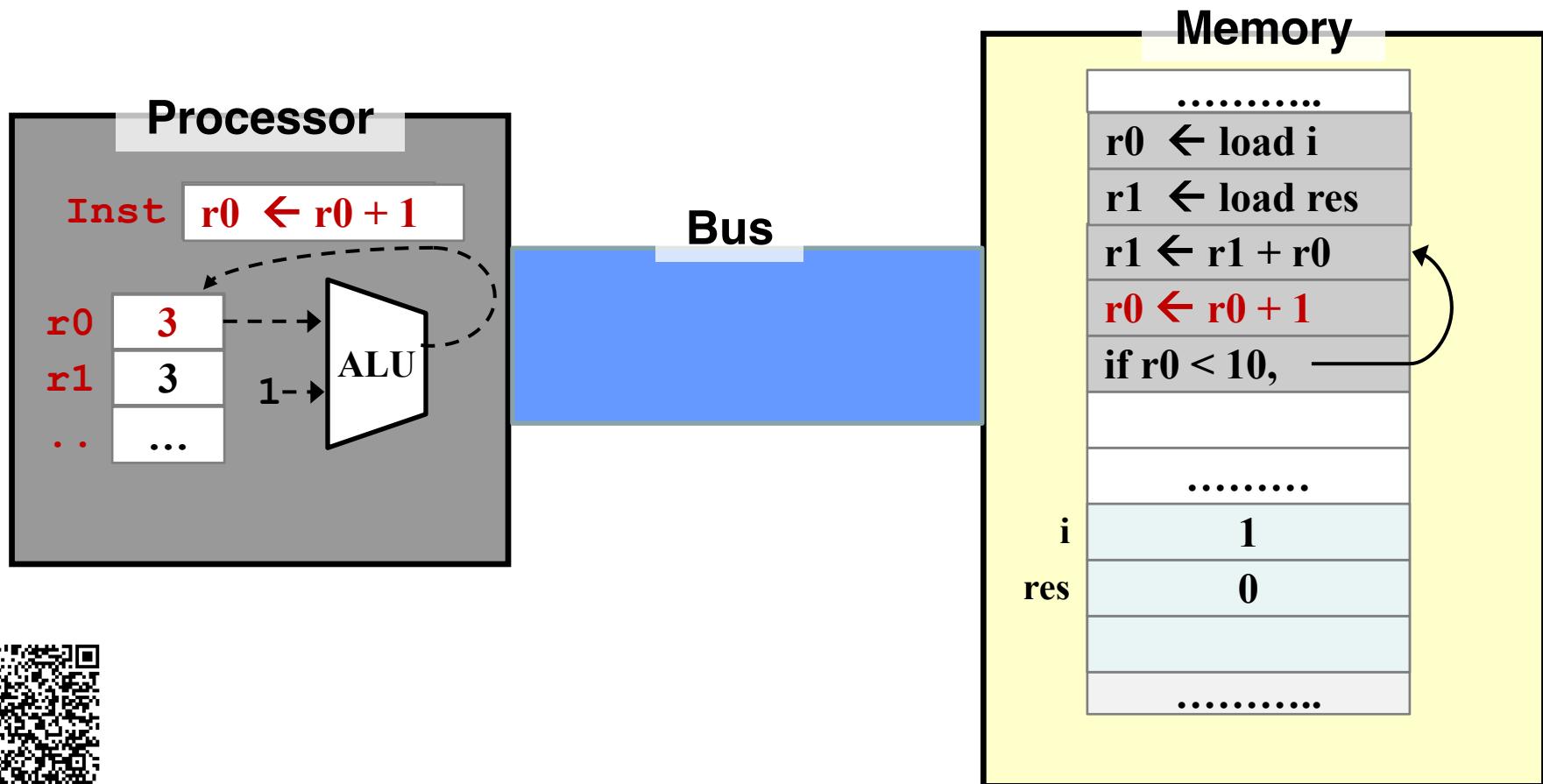
3. Walkthrough: Looping! (11/15)

- Since the condition succeeded, execution will repeat from the indicated position



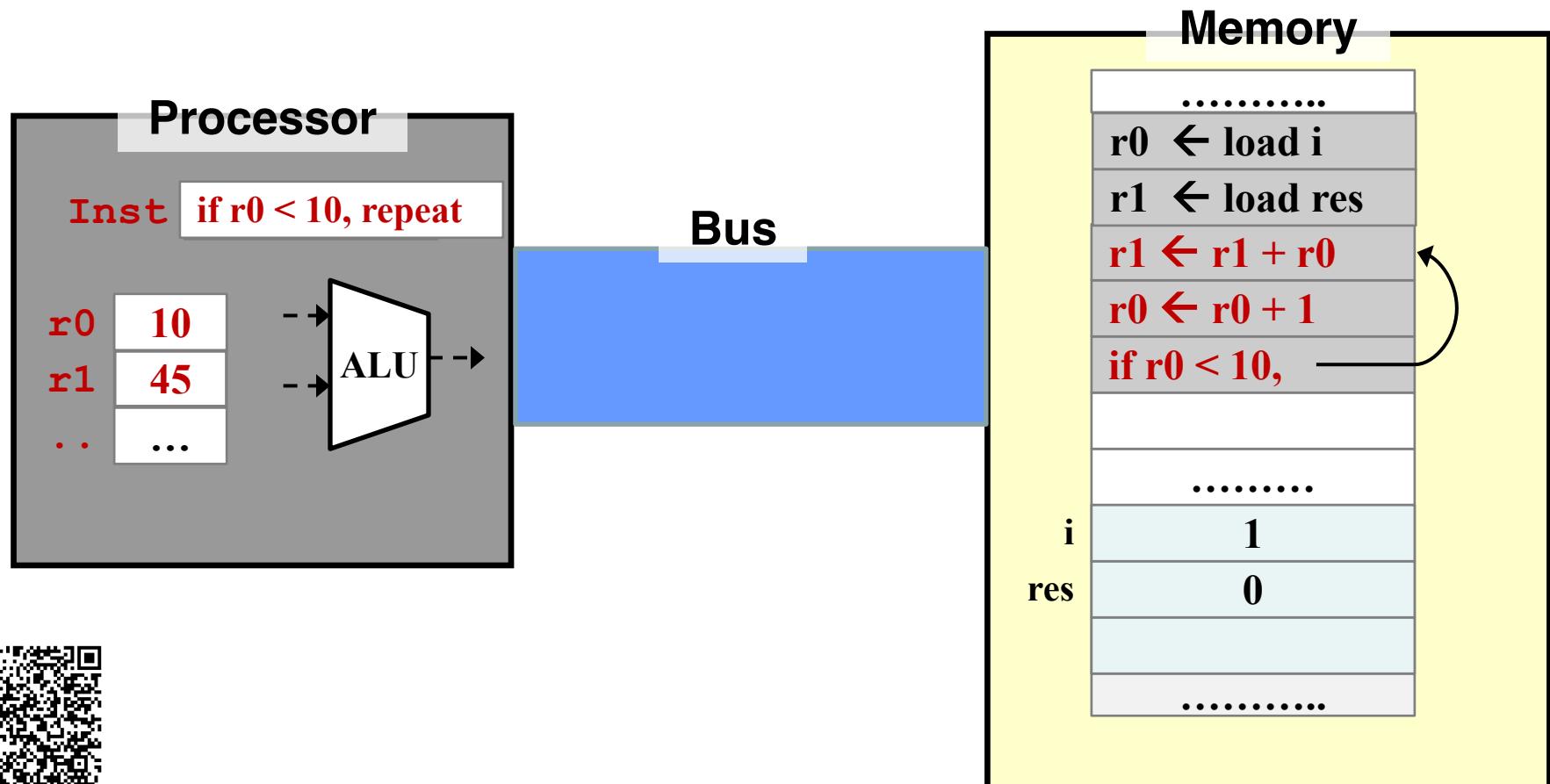
3. Walkthrough: Looping! (12/15)

- Execution will continue sequentially
 - Until we see another control flow instruction



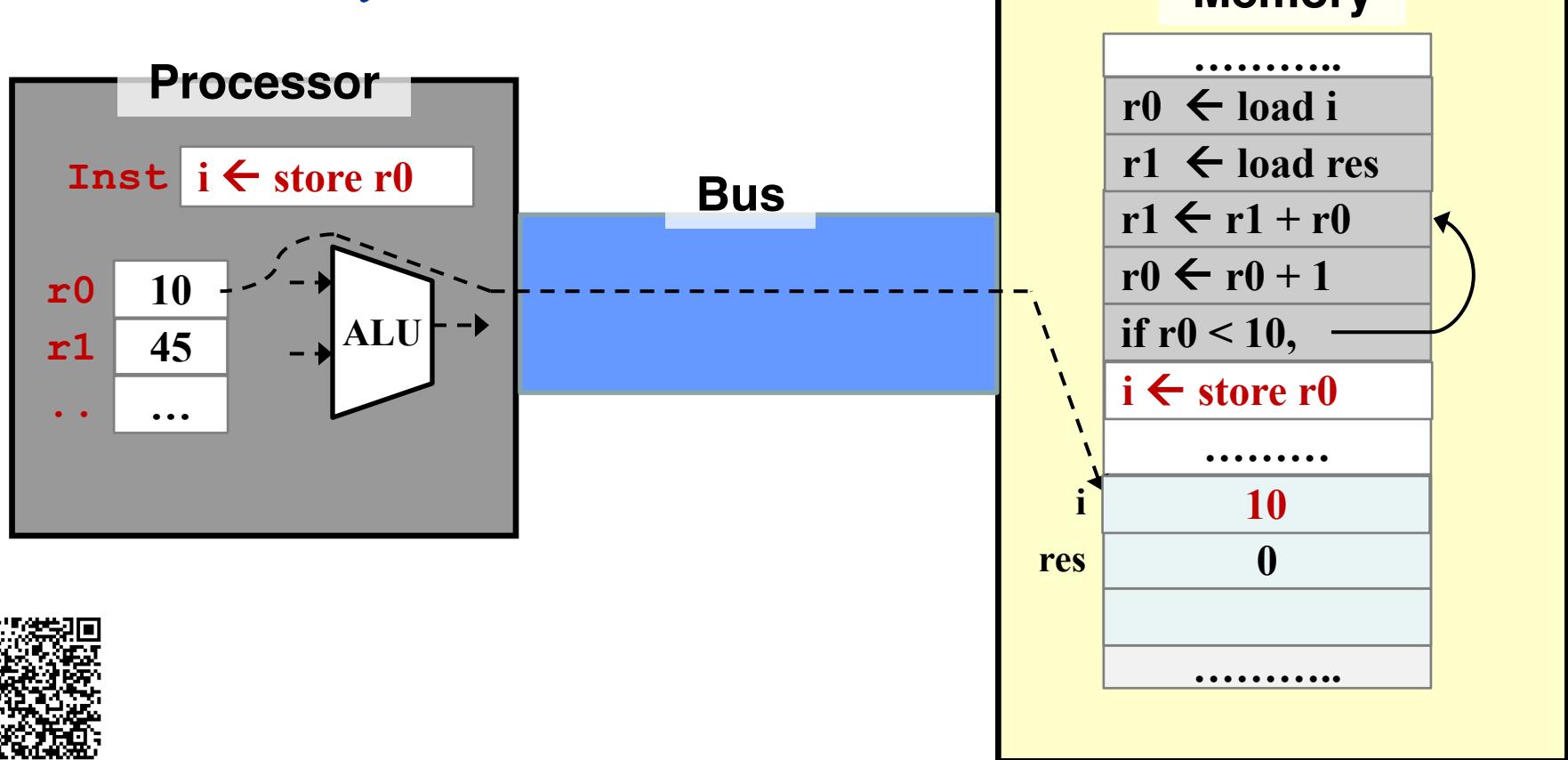
3. Walkthrough: Looping! (13/15)

- The three instructions will be repeated until the condition fails



3. Walkthrough: Memory Instruction (14/15)

- We can now move back the values from register to their “home” in memory
 - Similarly for “r1” to “res”



3. Walkthrough: Summary (15/15)

- **The stored-memory concept:**
 - Both **instruction** and **data** are stored in memory
- **The load-store model:**
 - Limit memory operations and relies on registers for storage during execution
- **The major types of assembly instruction:**
 - **Memory:** Move values between memory and registers
 - **Calculation:** Arithmetic and other operations
 - **Control flow:** Change the sequential execution



4. General Purpose Registers (1/2)

- Fast memories in the processor:
 - Data are transferred from memory to registers for faster processing
- Limited in number:
 - A typical architecture has 16 to 32 registers
 - Compiler associates variables in program with registers
- Registers have **no data type**
 - Unlike program variables!
 - Machine/Assembly instruction assumes the data stored in the register is of the correct type



4. General Purpose Registers (2/2)

- There are **32 registers** in **MIPS** assembly language:
 - Can be referred by a number (**\$0, \$1, ..., \$31**) OR
 - Referred by a name (eg: **\$a0, \$t1**)

Name	Register number	Usage
\$zero	0	Constant value 0
\$v0-\$v1	2-3	Values for results and expression evaluation
\$a0-\$a3	4-7	Arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Program variables

Name	Register number	Usage
\$t8-\$t9	24-25	More temporaries
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address

\$at (register 1) is reserved for the assembler.

\$k0-\$k1 (registers 26-27) are reserved for the operation system.



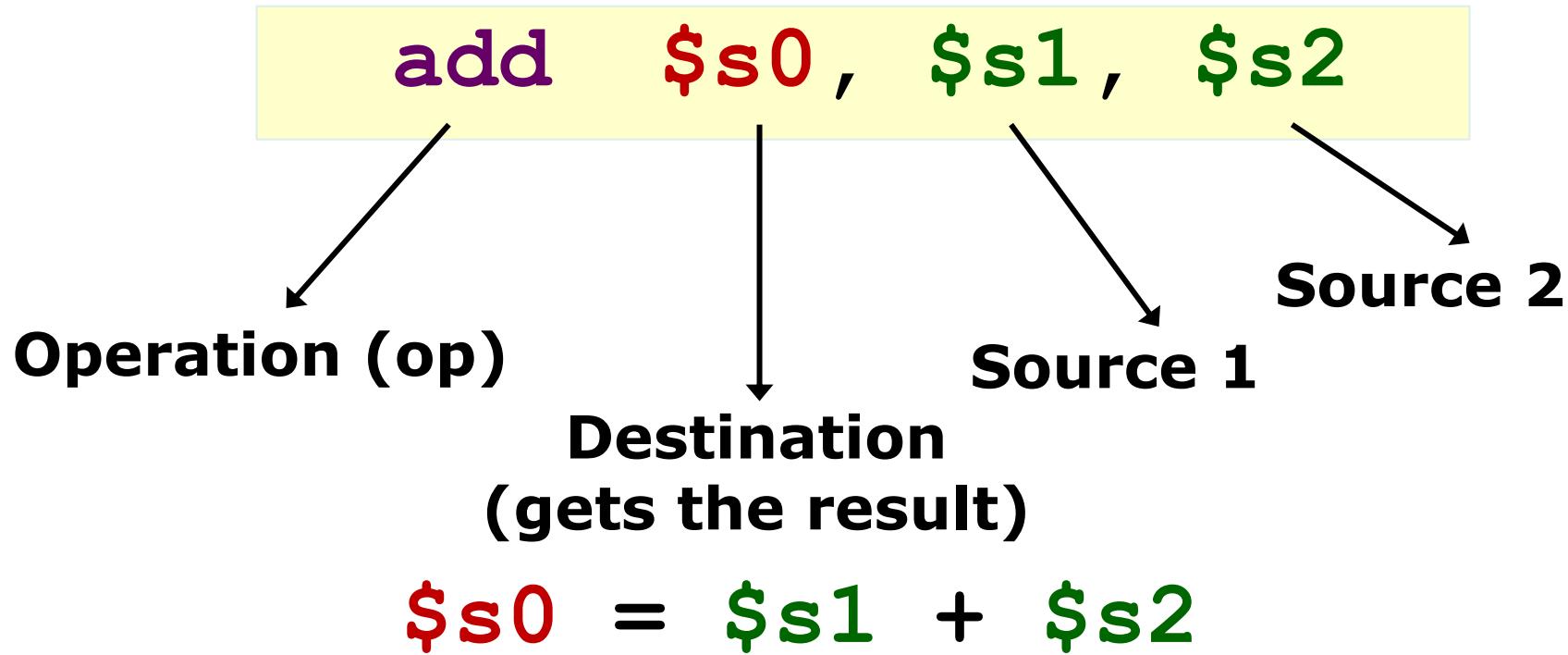
5. MIPS Assembly Language

- Each instruction executes a simple command
 - Usually has a counterpart in high level programming languages like C/C++, Java etc
- Each line of assembly code contains at most 1 instruction
- **#** (hex-sign) is used for comments
 - Anything from **#** to end of line is a comment and will be ignored by the assembler

```
add $t0, $s1, $s2    # $t0 ← $s1 + $s2
sub $s0, $t0, $s3    # $s0 ← $t0 - $s3
```



5.1 General Instruction Syntax



Naturally, most of the MIPS arithmetic/logic operations have three operands: **2 sources** and **1 destination**



5.2 Arithmetic Operation: Addition

C Statement	MIPS Assembly Code
a = b + c;	add \$s0, \$s1, \$s2

- We assume the values of "**a**", "**b**" and "**c**" are loaded into registers "**\$s0**", "**\$s1**" and "**\$s2**"
 - Known as **variable mapping**
 - Actual code to perform the loading will be shown later in ***memory instruction***
- **Important concept:**
 - MIPS arithmetic operations are mainly **register-to-register**



5.3 Arithmetic Operation: Subtraction

C Statement	MIPS Assembly Code
a = b - c;	sub \$s0, \$s1, \$s2 \$s0 → variable a \$s1 → variable b \$s2 → variable c

- Positions of **\$s1** and **\$s2** (i.e., source1 and source2) are important for subtraction



5.4 Complex Expression (1/3)

C Statement	MIPS Assembly Code
<code>a = b + c - d;</code>	<code>??? ??? ???</code> $\$s0 \rightarrow \text{variable } a$ $\$s1 \rightarrow \text{variable } b$ $\$s2 \rightarrow \text{variable } c$ $\$s3 \rightarrow \text{variable } d$

- A single MIPS instruction can handle at most two source operands
- Need to break a complex statement into multiple MIPS instructions

MIPS Assembly Code
<code>add \$t0, \$s1, \$s2 # tmp = b + c</code>
<code>sub \$s0, \$t0, \$s3 # a = tmp - d</code>

Use temporary registers
 $\$t0$ to $\$t7$ for intermediate results



5.4 Complex Expression: Example (2/3)

C Statement	Variable Mappings
<code>f = (g + h) - (i + j);</code>	$\$s0 \rightarrow \text{variable } f$ $\$s1 \rightarrow \text{variable } g$ $\$s2 \rightarrow \text{variable } h$ $\$s3 \rightarrow \text{variable } i$ $\$s4 \rightarrow \text{variable } j$

- Break it up into multiple instructions
 - Use two temporary registers $\$t0, \$t1$

```

add $t0, $s1, $s2 # tmp0 = g + h
add $t1, $s3, $s4 # tmp1 = i + j
sub $s0, $t0, $t1 # f = tmp0 - tmp1

```



5.4 Complex Expression: Exercise (3/3)

C Statement	Variable Mappings
<pre> z = a + b + c + d; add \$s4, \$s0, \$s1 add \$s4, \$s4, \$s2 add \$s4, \$s4, \$s3 </pre>	\$ _{s0} → variable a \$ _{s1} → variable b \$ _{s2} → variable c \$ _{s3} → variable d \$ _{s4} → variable z

C Statement	Variable Mappings
<pre> z = (a - b) + c; sub \$s3, \$s0, \$s1 add \$s3, \$s3, \$s2 </pre>	\$ _{s0} → variable a \$ _{s1} → variable b \$ _{s2} → variable c \$ _{s3} → variable z



5.5 Constant/Immediate Operands

C Statement	MIPS Assembly Code
a = a + 4;	addi \$s0, \$s0, 4

- **Immediate** values are numerical constants
 - Frequently used in operations
 - MIPS supplies a set of operations specially for them
- “Add immediate” (**addi**)
 - Syntax is similar to **add** instruction;
but source2 is a constant instead of register
 - **The constant ranges from [-2¹⁵ to 2¹⁵-1]**

Can you guess what number system is used?

There's no **subi**. Why?

Answer: 16-bit 2s complement number system

Answer: Use **addi** with negative constant

5.6 Register Zero (\$0 or \$zero)

- The number zero (0), appears very often in code
 - Provide register zero (**\$0** or **\$zero**) which always have the **value 0**

C Statement	MIPS Assembly Code
f = g;	add \$s0, \$s1, \$zero \$s0 → variable f \$s1 → variable g

- The above assignment is so common that MIPS has an equivalent **pseudo instruction (move)**:

MIPS Assembly Code
move \$s0, \$s1

Pseudo-Instruction
 "Fake" instruction that gets translated to corresponding MIPS instruction(s). Provided for convenience in coding only.



5.7 Logical Operations: Overview (1/2)

- Arithmetic instructions view the content of a register as a single quantity (signed or unsigned integer)
- **New perspective:**
 - View register as 32 raw bits rather than as a single 32-bit number
 - ➔ Possible to operate on individual bits or bytes within a word

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	sll
Shift right	>>	>>, >>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT*	~	~	nor
Bitwise XOR	^	^	xor, xori



5.7 Logical Operations: Overview (2/2)

- Truth tables of logical operations
 - 0 represents false; 1 represents true

AND

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

NOR

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

XOR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

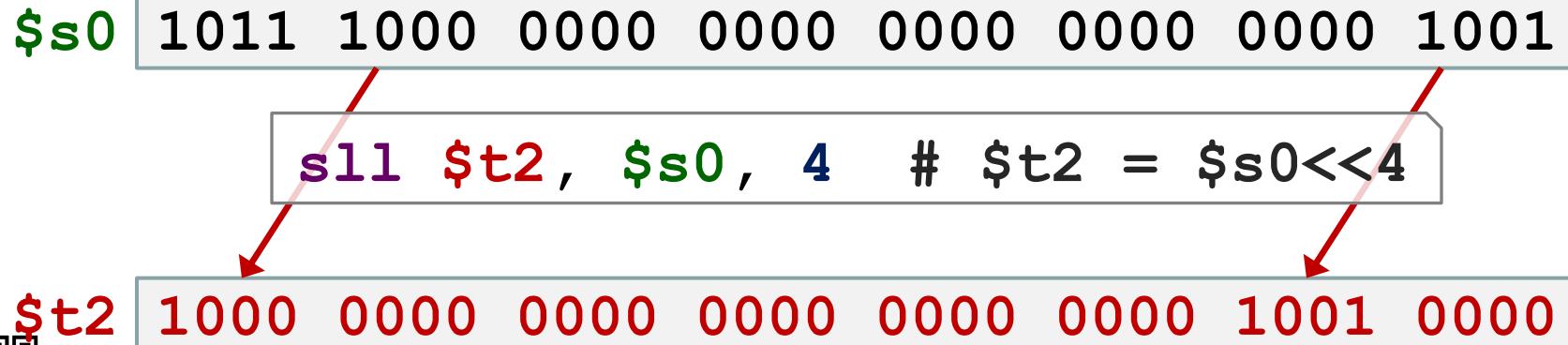


5.8 Logical Operations: Shifting (1/2)

Opcode: **sll** (shift left logical)

Move all the bits in a word to the left by a number of positions; fill the emptied positions with zeroes.

- E.g. Shift bits in **\$s0** to the left by 4 positions



5.8 Logical Operations: Shifting (2/2)

Opcode: **srl** (shift right logical)

Shifts right and fills emptied positions with zeroes.

- What is the equivalent math operations for shifting left/right n bits? Answer: **Multiply/divide by 2^n**
- Shifting is faster than multiplication/division
 - Good compiler translates such multiplication/division into shift instructions

C Statement	MIPS Assembly Code
a = a * 8;	sll \$s0, \$s0, 3



5.9 Logical Operations: Bitwise AND

Opcode: **and** (bitwise AND)

Bitwise operation that leaves a 1 only if both the bits of the operands are 1

- E.g.: **and \$t0, \$t1, \$t2**

\$t1	0110 0011 0010 1111 00	00 1101 0101 1001
mask	0000 0000 0000 0000 00	11 1100 0000 0000
\$t0	0000 0000 0000 0000 00	00 1100 0000 0000

- and** can be used for **masking** operation:
 - Place **0s** into the positions to be ignored → bits will turn into 0s
 - Place **1s** for interested positions → bits will remain the same as the original.



5.9 Exercise: Bitwise AND

- We are interested in the last 12 bits of the word in register **\$t1**. Result to be stored in **\$t0**.
 - Q: What's the mask to use?

\$t1	0000	1001	1100	0011	0101	1101	1001	1100
mask	0000	0000	0000	0000	0000	1111	1111	1111
\$t0	0000	0000	0000	0000	0000	1101	1001	1100

Notes:

The **and** instruction has an immediate version, **andi**



5.10 Logical Operations: Bitwise OR

Opcode: **or** (bitwise OR)

Bitwise operation that places a 1 in the result if either operand bit is 1

Example: **or \$t0, \$t1, \$t2**

- The **or** instruction has an immediate version **ori**
- Can be used to force certain bits to 1s
- E.g.: **ori \$t0, \$t1, 0xFFFF**

\$t1	0000 1001 1100 0011 0101	1101 1001 1100
0xFFFF	0000 0000 0000 0000 0000	1111 1111 1111
\$t0	0000 1001 1100 0011 0101	1111 1111 1111



5.11 Logical Operations: Bitwise NOR

- Strange fact 1:
 - There is no **NOT** instruction in MIPS to toggle the bits ($1 \rightarrow 0, 0 \rightarrow 1$)
 - However, a **NOR** instruction is provided:

Opcode: **nor** (bitwise NOR)

Example: **nor \$t0, \$t1, \$t2**

- Question: How do we get a NOT operation?

nor \$t0, \$t0, \$zero

- Question: Why do you think is the reason for not providing a NOT instruction?

One of design principles: Keep the instruction set small.



5.12 Logical Operations: Bitwise XOR

Opcode: **xor** (bitwise XOR)

Example: **xor \$t0, \$t1, \$t2**

- Question: Can we also get **NOT** operation from **XOR**?
Yes, let **\$t2** contain all 1s.
xor \$t0, \$t0, \$t2
- Strange Fact 2:
 - There is no **NORI**, but there is **XORI** in MIPS
 - Why?

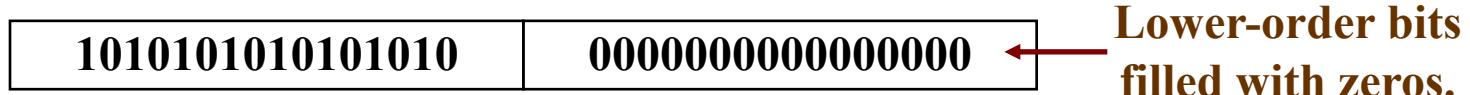


6. Large Constant: Case Study

- Question: How to load a 32-bit constant into a register? e.g
10101010 10101010 11110000 11110000

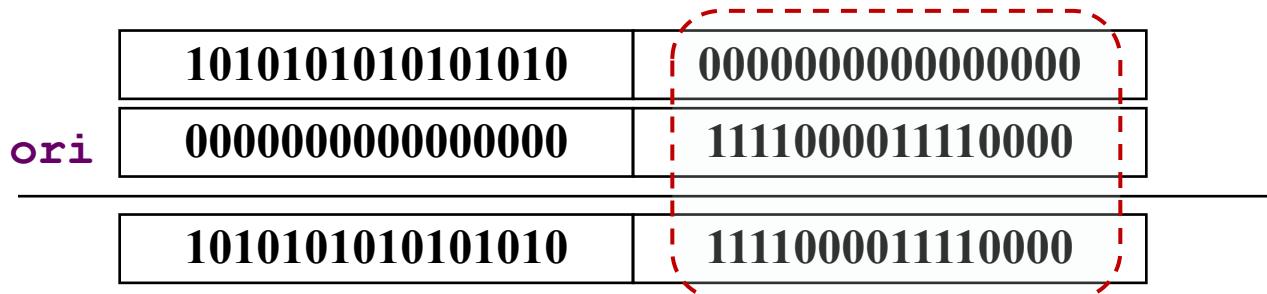
-
1. Use “load upper immediate” (**lui**) to set the upper 16-bit:

lui \$t0, 0xAAAA #1010101010101010



2. Use “or immediate” (**ori**) to set the lower-order bits:

ori \$t0, \$t0, 0xF0F0 #1111000011110000



7. MIPS Basic Instructions Checklist

Operation	Opcode in MIPS	Meaning
Addition	add \$s0, \$s1, \$s2	\$s0 = \$s1 + \$s2
	addi \$s0, \$s1, C16 _{2s}	\$s0 = \$s1 + C16 _{2s}
Subtraction	sub \$s0, \$s1, \$s2	\$s0 = \$s1 - \$s2
Shift left logical	sll \$s0, \$s1, C5	\$s0 = \$s1 << C5
Shift right logical	srl \$s0, \$s1, C5	\$s0 = \$s1 >> C5
AND bitwise	and \$s0, \$s1, \$s2	\$s0 = \$s1 & \$s2
	andi \$s0, \$s1, C16	\$s0 = \$s1 & C16
OR bitwise	or \$s0, \$s1, \$s2	\$s0 = \$s1 \$s2
	ori \$s0, \$s1, C16	\$s0 = \$s1 C16
NOR bitwise	nor \$s0, \$s1, \$s2	\$s0 = \$s1 ↓ \$s2
XOR bitwise	xor \$s0, \$s1, \$s2	\$s0 = \$s1 ⊕ \$s2
	xori \$s0, \$s1, C16	\$s0 = \$s1 ⊕ C16



is [0 to 2⁵-1]

C16_{2s} is [-2¹⁵ to 2¹⁵-1]

C16 is a 16-bit pattern

Lesson Plan

1. Welcome – This Lecture
2. Introduction – Basic Ideas
3. Number Systems – Everything is numbers!
4. C and Hardware
- 5. MIPS Assembly Language**
6. Instruction Set Architecture Design
7. Datapath Design
8. Processor Control
9. Boolean Algebra
10. Simplification Methods
11. Combinational Circuit Design
12. MSI Components
13. Sequential Circuit Design



IT5002

Computer Systems and Applications

MIPS Assembly II

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lecture #8: MIPS Part 2: More Instructions

1. Memory Organisation (General)

- 1.1 Memory: Transfer Unit
- 1.2 Memory: Word Alignment

2. MIPS Memory Instructions

- 2.1 Memory Instruction: Load Word
- 2.2 Memory Instruction: Store Word
- 2.3 Load and Store Instructions
- 2.4 Memory Instruction: Others
- 2.5 Example: Array
- 2.6 Common Questions
- 2.7 Example: Swapping Elements



Lecture #8: MIPS Part 2: More Instructions

3. Making Decisions

- 3.1 Conditional Branch: beq and bne
- 3.2 Unconditional Jump: j
- 3.3 IF statement
- 3.4 Exercise #1: IF statement

4. Loops

- 4.1 Exercise #2: FOR loop
- 4.2 Inequalities

5. Array and Loop

6. Exercises



1. Memory Organisation (General)

- The main memory can be viewed as a large, single-dimension array of memory locations.
- Each location of the memory has an **address**, which is an index into the array.
 - Given a k -bit address, the address space is of size 2^k .
- The memory map on the right contains one byte (8 bits) in every location/address.
 - This is called **byte addressing**

<i>Address</i>	<i>Content</i>
0	8 bits
1	8 bits
2	8 bits
3	8 bits
4	8 bits
5	8 bits
6	8 bits
7	8 bits
8	8 bits
9	8 bits
10	8 bits
11	8 bits

:



1.1 Memory: Transfer Unit

- Using distinct memory address, we can access:
 - a single **byte (byte addressable)** or
 - a single **word (word addressable)**
- **Word** is:
 - Usually 2^n bytes
 - The common unit of transfer between processor and memory
 - Also commonly coincide with the register size, the integer size and instruction size in most architectures

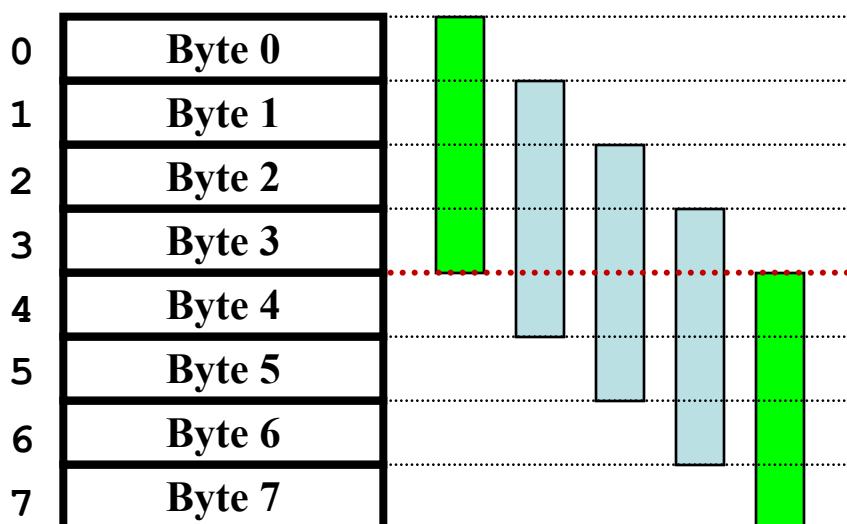


1.2 Memory: Word Alignment

- **Word alignment:**
 - Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.

Example: If a word consists of 4 bytes, then:

Address



Question:

How do we quickly check whether a given memory address is word-aligned or not?



2. MIPS Memory Instructions

- MIPS is a load-store register architecture
 - 32 registers, each 32-bit (4 bytes) long
 - Each word contains 32 bits (4 bytes)
 - Memory addresses are 32-bit long

Why?

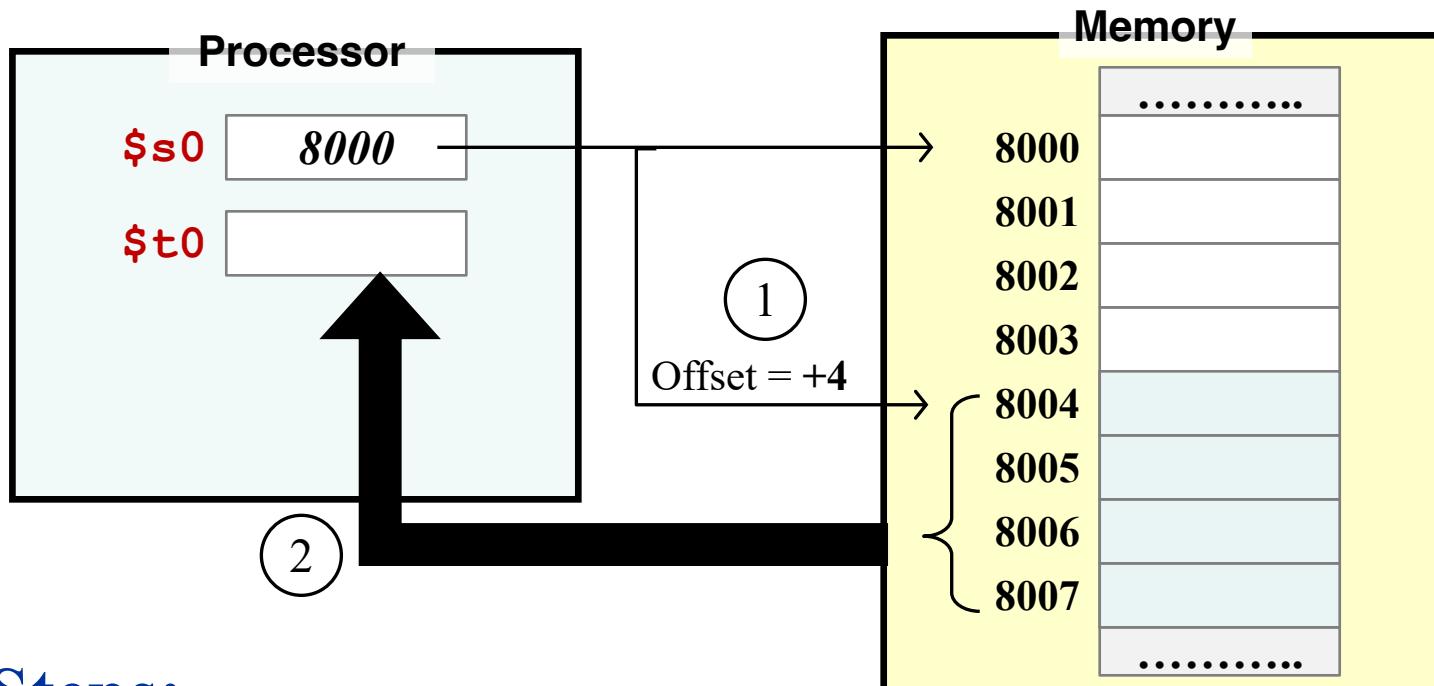
Answer: So that they fit into registers

Name	Examples	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast processor storage for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Mem[0], Mem[4], ..., Mem[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so consecutive words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.



2.1 Memory Instruction: Load Word

- Example: **lw \$t0, 4(\$s0)**

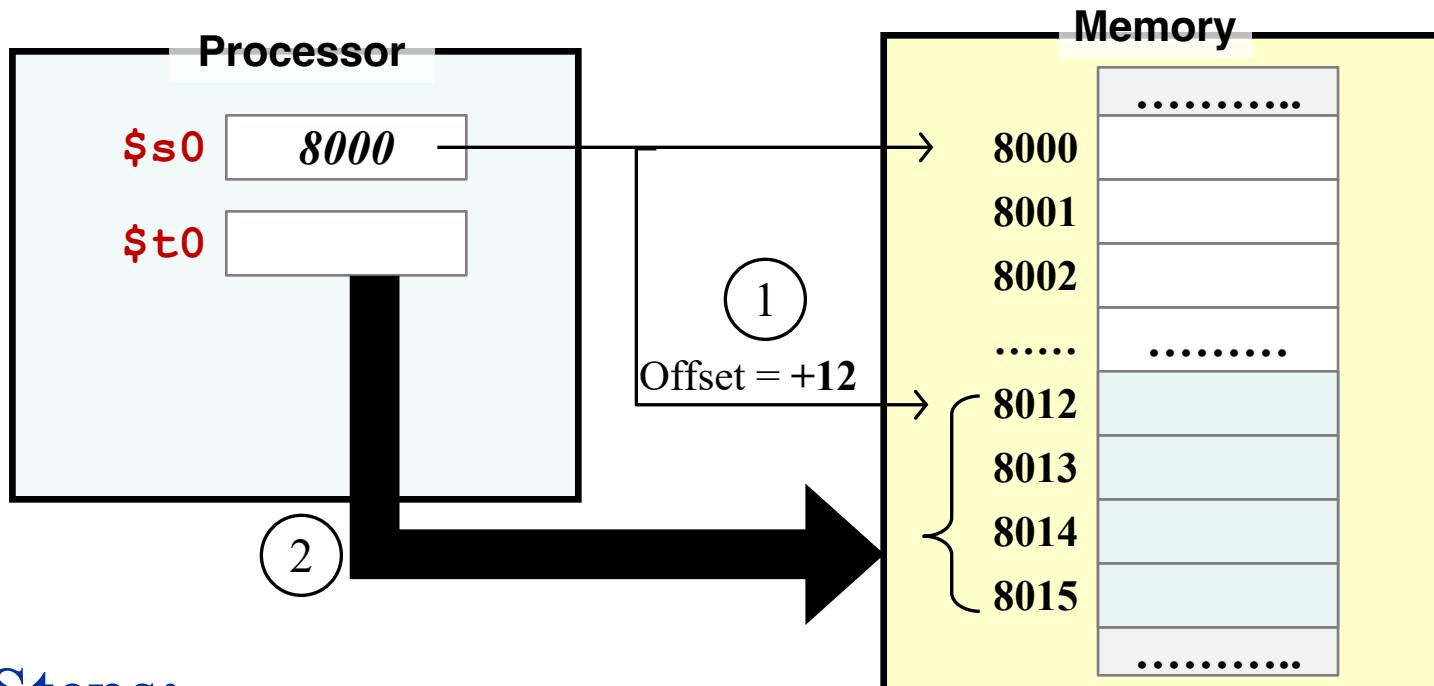


- Steps:
 - Memory Address = **\$s0 + 4** = **8000 + 4 = 8004**
 - Memory word at **Mem[8004]** is loaded into **\$t0**



2.2 Memory Instruction: Store Word

- Example: **sw \$t0 , 12(\$s0)**



- Steps:
 - Memory Address = **\$s0 + 12 = 8000 + 12 = 8012**
 - Content of **\$t0** is stored into word at **Mem[8012]**



2.3 Load and Store Instructions

- Only **load** and **store** instructions can access data in memory.
- Example: Each array element occupies a word.

C Code	MIPS Code
A[7] = h + A[10];	lw \$t0, 40(\$s3) add \$t0, \$s2, \$t0 sw \$t0, 28(\$s3)

- Each array element occupies a word (4 bytes).
 - \$s3** contains the **base address** (address of first element, A[0]) of array A. Variable **h** is mapped to **\$s2**.
- Remember arithmetic operands (for **add**) are registers, not memory!



2.4 Memory Instructions: Others (1/2)

- Other than load word (**lw**) and store word (**sw**), there are other variants, example:
 - load byte (**lb**)
 - store byte (**sb**)
- Similar in format:

```
lb $t1, 12($s3)
sb $t2, 13($s3)
```
- Similar in working except that one byte, instead of one word, is loaded or stored
 - Note that the offset no longer needs to be a multiple of 4



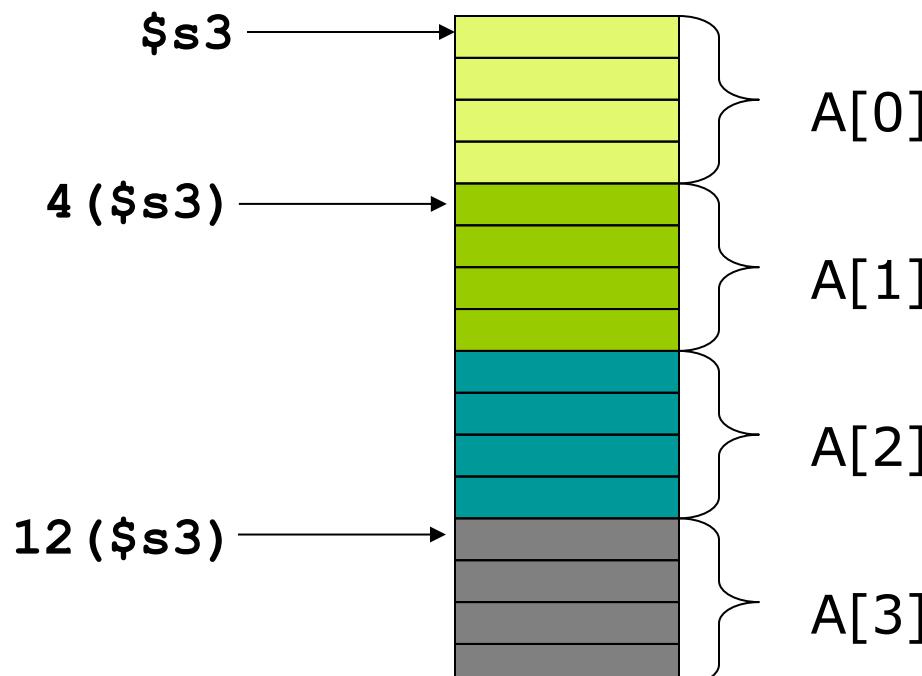
2.4 Memory Instructions: Others (2/2)

- MIPS disallows loading/storing unaligned word using **lw/sw**:
 - Pseudo-Instructions *unaligned load word* (**ulw**) and *unaligned store word* (**usw**) are provided for this purpose
- Other memory instructions:
 - **lh** and **sh**: load halfword and store halfword
 - **lwl**, **lwr**, **swl**, **swr**: load word left / right, store word left / right.
 - etc...



2.5 Example: Array (assume 4 bytes per element)

C Statement to translate	Variables Mapping
$A[3] = h + A[1];$	$h \rightarrow \$s2$ base of $A[] \rightarrow \$s3$



```

lw $t0, 4($s3)
add $t0, $s2, $t0
sw $t0, 12($s3)
  
```



2.6 Common Questions: Address vs Value

Key concept:

Registers do NOT have types

- A register can hold any 32-bit number:
 - The number has no implicit data type and is interpreted according to the instruction that uses it
- Examples:
 - **add \$t2, \$t1, \$t0**
→ \$t0 and \$t1 should contain data values
 - **lw \$t2, 0(\$t0)**
→ \$t0 should contain a memory address



2.6 Common Questions: Byte vs Word

Important:

Consecutive **word addresses** in machines with
byte-addressing do not differ by 1

- Common error:
 - Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes
- For both **lw** and **sw**:
 - The sum of base address and offset must be a multiple of 4 (i.e. to adhere to word boundary)



2.7 Example: Swapping Elements

C Statement to translate	Variables Mapping
<pre>swap(int v[], int k) { int temp; temp = v[k] v[k] = v[k+1]; v[k+1] = temp; }</pre>	<p>k → \$5 Base address of v[] → \$4 temp → \$15</p>
<pre>swap: sll \$2, \$5, 2 add \$2, \$4, \$2 lw \$15, 0(\$2) lw \$16, 4(\$2) sw \$16, 0(\$2) sw \$15, 4(\$2)</pre>	<p>Example: k = 3; to swap v[3] with v[4]. Assume base address of v is 2000.</p> <p>\$5 (k) ← 3 \$4 (base addr. of v) ← 2000</p> <p>\$2 ← 12 \$2 ← 2012 \$15 ← content of mem. addr. 2012 (v[3]) \$16 ← content of mem. addr. 2016 (v[4]) content of mem. addr. 2012 (v[3]) ← \$16 content of mem. addr. 2016 (v[4]) ← \$15</p>

Note: This is simplified and may not be a direct translation of the C code.



3. Making Decisions (1/2)

- We cover only sequential execution so far:
 - Instruction is executed in program order
- To perform general computing tasks, we need to:
 - Make decisions
 - Perform iterations (in later section)
- Decisions making in high-level language:
 - **if** and **goto** statements
 - MIPS decision making instructions are similar to **if** statement with a **goto**
 - **goto** is discouraged in high-level languages but necessary in assembly ☺



3. Making Decisions (2/2)

- Decision-making instructions
 - Alter the control flow of the program
 - Change the next instruction to be executed
- Two types of decision-making statements in MIPS
 - **Conditional** (branch)
`bne $t0, $t1, label`
`beq $t0, $t1, label`
 - **Unconditional** (jump)
`j label`
- A label is an “anchor” in the assembly code to indicate point of interest, usually as branch target
 - Labels are NOT instructions!



3.1 Conditional Branch: **beq** and **bne**

- Processor follows the branch only when the condition is satisfied (true)
- **beq \$r1, \$r2, L1**
 - Go to statement labeled **L1** if the value in register **\$r1** equals the value in register **\$r2**
 - **beq** is “branch if equal”
 - C code: **if (a == b) goto L1**
- **bne \$r1, \$r2, L1**
 - Go to statement labeled **L1** if the value in register **\$r1** does not equal the value in register **\$r2**
 - **bne** is “branch if not equal”
 - C code: **if (a != b) goto L1**



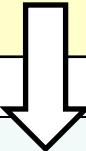
3.2 Unconditional Jump: **j**

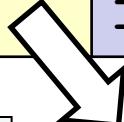
- Processor **always** follows the branch
- **j L1**
 - Jump to label **L1** unconditionally
 - C code: **goto L1**
- Technically equivalent to such statement
beq \$s0, \$s0, L1



3.3 IF statement (1/2)

C Statement to translate	Variables Mapping
<pre>if (i == j) f = g + h;</pre>	<p>f → \$s0 g → \$s1 h → \$s2 i → \$s3 j → \$s4</p>





```
bne $s3, $s4, L1
j Exit
L1: add $s0, $s1, $s2
Exit:
```

```
bne $s3, $s4, Exit
add $s0, $s1, $s2
Exit:
```

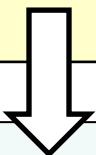
- Two equivalent translations:
 - The one on the right is more efficient

Common technique: Invert the condition for shorter code



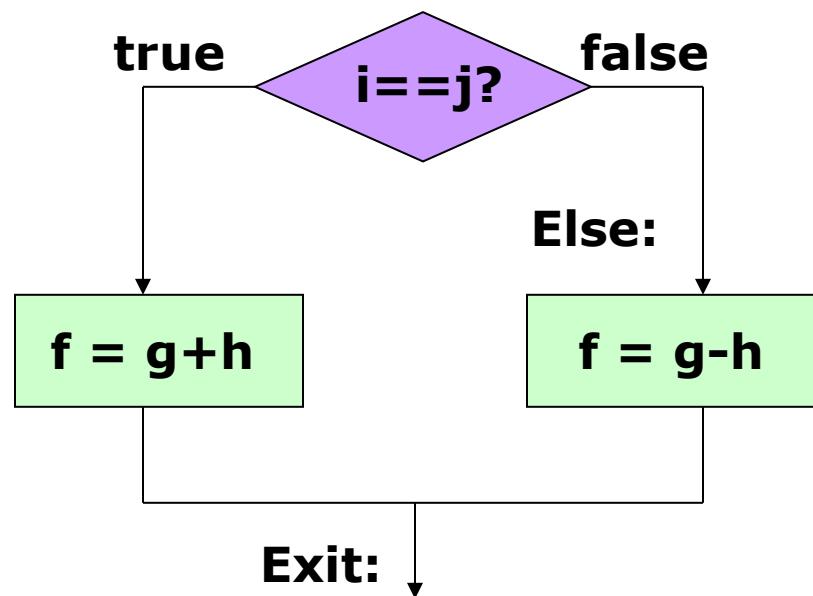
3.3 IF statement (2/2)

C Statement to translate	Variables Mapping
<pre> if (i == j) f = g + h; else f = g - h; </pre>	<p>f → \$s0 g → \$s1 h → \$s2 i → \$s3 j → \$s4</p>



```

bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else: sub $s0, $s1, $s2
Exit:
  
```



- Question: Rewrite with **beq**?



3.4 Exercise #1: IF statement

MIPS code to translate into C	Variables Mapping
<pre>beq \$s1, \$s2, Exit add \$s0, \$zero, \$zero Exit:</pre>	<p>f → \$s0 i → \$s1 j → \$s2</p>

- What is the corresponding high-level statement?

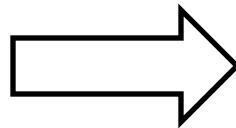
```
if (i != j) {
    f = 0;
}
```



4. Loops (1/2)

- C while-loop:
- Rewritten with goto

```
while (j == k)
    i = i + 1;
```



```
Loop: if (j != k)
      goto Exit;
      i = i+1;
      goto Loop;
```

```
Exit:
```

Key concept:

Any form of loop can be written in assembly with the help of conditional branches and jumps.



4. Loops (2/2)

C Statement to translate	Variables Mapping
Loop: if (j != k) goto Exit; i = i+1; goto Loop;	i → \$s3 j → \$s4 k → \$s5
Exit:	

- What is the corresponding MIPS code?

```

Loop: bne $s4, $s5, Exit    # if (j != k) Exit
        addi $s3, $s3, 1
        j     Loop                 # repeat loop

Exit:
  
```



4.1 Exercise #2: FOR loop

- Write the following loop statement in MIPS

C Statement to translate	Variables Mapping
for (i=0; i<10; i++) a = a + 5;	i → \$s0 a → \$s2

```
add $s0, $zero, $zero
addi $s1, $zero, 10
Loop: beq $s0, $s1, Exit
      addi $s2, $s2, 5
      addi $s0, $s0, 1
      j Loop
Exit:
```



4.2 Inequalities (1/2)

- We have **beq** and **bne**, what about branch-if-less-than?
 - There is no real **blt** instruction in MIPS
- Use **slt** (set on less than) or **slti**.

```
slt $t0, $s1, $s2
```

=

```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```



4.2 Inequalities (2/2)

- To build a “**blt \$s1, \$s2, L**” instruction:

```
slt $t0, $s1, $s2  
bne $t0, $zero, L
```

=

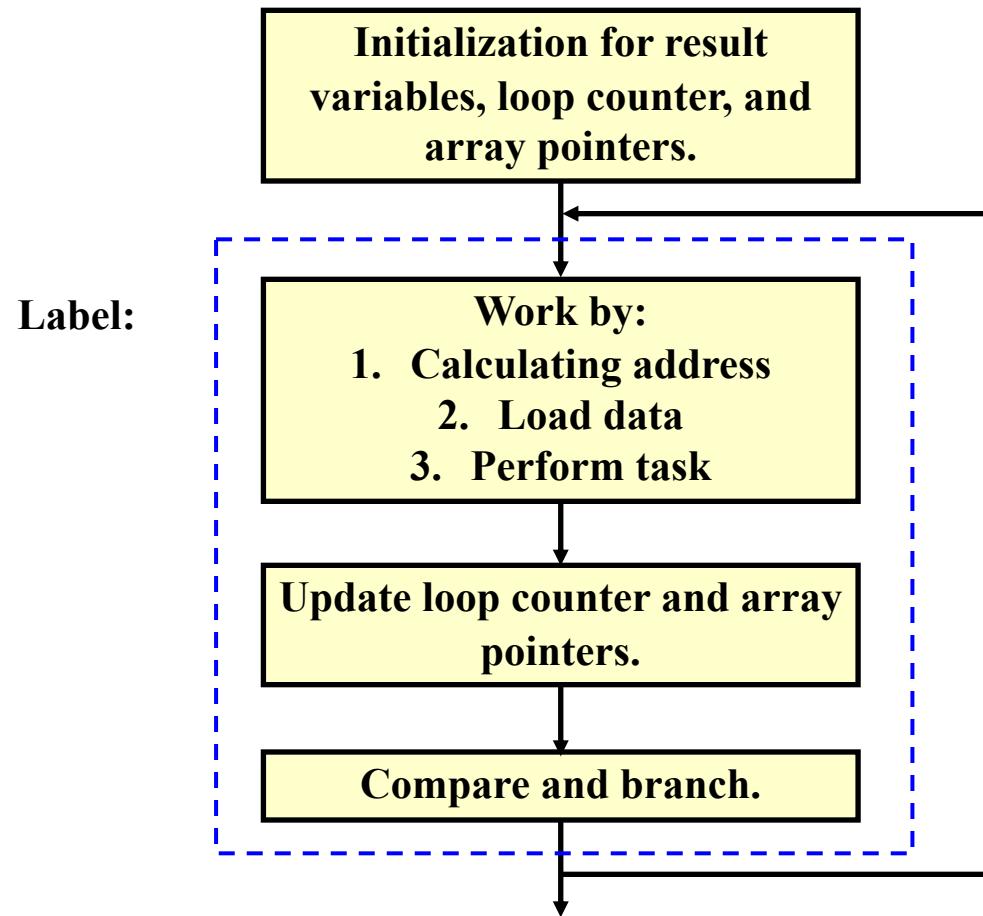
```
if ($s1 < $s2)  
    goto L;
```

- This is another example of pseudo-instruction:
 - Assembler translates (**blt**) instruction in an assembly program into the equivalent MIPS (two) instructions



5. Array and Loop

- Typical example of accessing array elements in a loop:



5. Array and Loop: Question

Count the number of zeros in an Array A

- A is word array with 40 elements
- Address of A[] → **\$t0**, Result → **\$t8**

Simple C Code

```
result = 0;  
  
i = 0;  
  
while ( i < 40 ) {  
    if ( A[i] == 0 )  
        result++;  
    i++;  
}
```

- Think about:
 - How to perform the right comparison
 - How to translate A[i] correctly



5. Array and Loop: Version 1.0

Address of A[] → \$t0 Result → \$t8 i → \$t1	Comments
<pre> addi \$t8, \$zero, 0 addi \$t1, \$zero, 0 addi \$t2, \$zero, 40 # end point loop: bge \$t1, \$t2, end sll \$t3, \$t1, 2 # i * 4 add \$t4, \$t0, \$t3 # &A[i] lw \$t5, 0(\$t4) # \$t3 ← A[i] bne \$t5, \$zero, skip addi \$t8, \$t8, 1 # result++ skip: addi \$t1, \$t1, 1 # i++ j loop end: </pre>	



5. Array and Loop: Version 2.0

Address of A[] → \$t0 Result → \$t8 &A[i] → \$t1	Comments
<pre> addi \$t8, \$zero, 0 addi \$t1, \$t0, 0 addi \$t2, \$t0, 160 loop: bge \$t1, \$t2, end lw \$t3, 0(\$t1) bne \$t3, \$zero, skip addi \$t8, \$t8, 1 skip: addi \$t1, \$t1, 4 j loop end:</pre>	<pre> # addr of current item # &A[40] # comparing address! # \$t3 ← A[i] # result++ # move to next item</pre>

Use of “pointers” can produce more efficient code!



6.1 Exercise #3: Simple Loop

- Given the following MIPS code:

```
addi $t1, $zero, 10
add $t1, $t1, $t1
addi $t2, $zero, 10
Loop: addi $t2, $t2, 10
       addi $t1, $t1, -1
       beq $t1, $zero, Loop
```

- How many instructions are executed? **Answer: (a)**
(a) 6 (b) 30 (c) 33 (d) 36 (e) None of the above
- What is the final value in **\$t2**? **Answer: (b)**
(a) 10 (b) 20 (c) 300 (d) 310 (e) None of the above



6.2 Exercise #4: Simple Loop II

- Given the following MIPS code:

```
add $t0, $zero, $zero
add $t1, $t0, $t0
addi $t2, $t1, 4
Again: add $t1, $t1, $t0
       addi $t0, $t0, 1
       bne $t2, $t0, Again
```

- How many instructions are executed? **Answer: (c)**
(a) 6 (b) 12 (c) 15 (d) 18 (e) None of the above
- What is the final value in **\$t1**? **Answer: (c)**
(a) 0 (b) 4 (c) 6 (d) 10 (e) None of the above



6.3 Exercise #5: Simple Loop III (1/2)

- Given the following MIPS code accessing a word array of elements in memory with the starting address in **\$t0**.

```
addi $t1, $t0, 10
add $t2, $zero, $zero
Loop: ulw $t3, 0($t1) # ulw: unaligned lw
      add $t2, $t2, $t3
      addi $t1, $t1, -1
      bne $t1, $t0, Loop
```

- How many times is the **bne** instruction executed?
(a) 1 (b) 3 (c) 9 (d) 10 (e) 11 **Answer: (d)**
- How many times does the **bne** instruction actually branch to the label **Loop**?
(a) 1 (b) 8 (c) 9 (d) 10 (e) 11 **Answer: (c)**



6.3 Exercise #5: Simple Loop III (2/2)

- Given the following MIPS code accessing a word array of elements in memory with the starting address in **\$t0**.

```
addi $t1, $t0, 10
add $t2, $zero, $zero
Loop: ulw $t3, 0($t1) # ulw: unaligned lw
      add $t2, $t2, $t3
      addi $t1, $t1, -1
      bne $t1, $t0, Loop
```

- iii. How many instructions are executed?
(a) 6 (b) 12 (c) 41 (d) 42 (e) 46 **Answer: (d)**
- iv. How many **unique** bytes of data are read from the memory?
(a) 4 (b) 10 (c) 11 (d) 13 (e) 40 **Answer: (d)**



Summary: Focus of IT5002

- Basic MIPS programming
 - Arithmetic: among registers only
 - Handling of large constants
 - Memory accesses: load/store
 - Control flow: branch and jump
 - Accessing array elements
 - System calls (covered in labs)
- Things we are not going to cover
 - Support for procedures
 - Linkers, loaders, memory layout
 - Stacks, frames, recursion
 - Interrupts and exceptions



IT5002

Computer Systems and Applications

MIPS Instruction Formats

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lecture #6: MIPS Part 3: Instruction Formats

- 1. Overview and Motivation**
- 2. MIPS Encoding: Basics**
- 3. MIPS Instruction Classification**
- 4. MIPS Registers (Recap)**
- 5. R-Format**
 - 5.1 R-Format: Example**
 - 5.2 Try It Yourself #1**



Lecture #9: MIPS Part 3: Instruction Formats

6. I-Format

- 6.1 I-Format: Example
- 6.2 Try It Yourself #2
- 6.3 Instruction Address: Overview
- 6.4 Branch: PC-Relative Addressing
- 6.5 Branch: Example
- 6.6 Try It Yourself #3

7. J-Format

- 7.1 J-Format: Example9
- 7.2 Branching Far Away: Challenge



8. Addressing Modes

1. Overview and Motivation

- **Recap: Assembly instructions will be translated to machine code for actual execution**
 - This section shows how to translate MIPS assembly code into binary patterns
- **Explains some of the “strange facts” from earlier:**
 - Why is *immediate* limited to 16 bits?
 - Why is *shift* amount only 5 bits?
 - etc.
- **Prepare us to “build” a MIPS processor in later lectures!**



2. MIPS Encoding: Basics

- **Each MIPS instruction has a fixed-length of 32 bits**
 - ➔ All relevant information for an operation must be encoded with these bits!
- **Additional challenge:**
 - To reduce the complexity of processor design, the instruction encodings should be as regular as possible
 - ➔ Small number of formats, i.e. as few variations as possible



3. MIPS Instruction Classification

- Instructions are classified according to their operands:
→ Instructions with same operand types have same encoding

R-format (Register format: **op \$r1, \$r2, \$r3**)

- Instructions which use 2 source registers and 1 destination register
 - e.g. **add, sub, and, or, nor, slt, etc**
 - Special cases: **srl, sll**, etc.

I-format (Immediate format: **op \$r1, \$r2, Immd**)

- Instructions which use 1 source register, 1 immediate value and 1 destination register
 - e.g. **addi, andi, ori, slti, lw, sw, beq, bne**, etc.

J-format (Jump format: **op Immd**)

- j** instruction uses only one immediate value



4. MIPS Registers (Recap)

- For simplicity, register numbers (**\$0, \$1, ..., \$31**) will be used in examples here instead of register names

Name	Register number	Usage
\$zero	0	Constant value 0
\$v0-\$v1	2-3	Values for results and expression evaluation
\$a0-\$a3	4-7	Arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Program variables

Name	Register number	Usage
\$t8-\$t9	24-25	More temporaries
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address

\$at (register 1) is reserved for the assembler.

\$k0-\$k1 (registers 26-27) are reserved for the operation system.



5. R-Format (1/2)

- Define fields with the following number of bits each:
 - $6 + 5 + 5 + 5 + 5 + 6 = 32$ bits



- Each field has a name:



- Each field is an independent 5- or 6-bit unsigned integer
 - A 5-bit field can represent any number 0 – 31
 - A 6-bit field can represent any number 0 – 63



5. R-Format (2/2)

Fields	Meaning
opcode	<ul style="list-style-type: none"> - Partially specifies the instruction - Equal to 0 for all R-Format instructions
funct	<ul style="list-style-type: none"> - Combined with opcode exactly specifies the instruction
rs (Source Register)	<ul style="list-style-type: none"> - Specify register containing first operand
rt (Target Register)	<ul style="list-style-type: none"> - Specify register containing second operand
rd (Destination Register)	<ul style="list-style-type: none"> - Specify register which will receive result of computation
shamt	<ul style="list-style-type: none"> - Amount a shift instruction will shift by - 5 bits (i.e. 0 to 31) - Set to 0 in all non-shift instructions



5.1 R-Format: Example (1/3)

MIPS instruction

add \$8, \$9, \$10

R-Format Fields	Value	Remarks
opcode	0	(textbook pg 94 - 101)
funct	32	(textbook pg 94 - 101)
rd	8	(destination register)
rs	9	(first operand)
rt	10	(second operand)
shamt	0	(not a shift instruction)



5.1 R-Format: Example (2/3)

MIPS instruction

add \$8, \$9, \$10



Note the ordering of
the 3 registers

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	32

Field representation in binary:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

Split into 4-bit groups for hexadecimal conversion:

0000	0001	0010	1010	0100	0000	0010	0000
------	------	------	------	------	------	------	------

0_{16}

1_{16}

2_{16}

A_{16}

4_{16}

0_{16}

2_{16}

0_{16}



5.1 R-Format: Example (3/3)

MIPS instruction

sll \$8, \$9, 4



Note the placement of the source register

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	0	9	8	4	0

Field representation in binary:

000000	000000	01001	01000	00100	000000
--------	--------	-------	-------	-------	--------

Split into 4-bit groups for hexadecimal conversion:

0000	0000	0000	1001	0100	0001	0000	0000
------	------	------	------	------	------	------	------

0_{16}

0_{16}

0_{16}

9_{16}

4_{16}

1_{16}

0_{16}

0_{16}



5.2 Try It Yourself #1

MIPS instruction

add \$10, \$7, \$5

Field representation in decimal:

opcode	rs	rt	rd	shamt	funct
0	7	5	10	0	32

Field representation in binary:

000000	00111	00101	01010	00000	100000
--------	-------	-------	-------	-------	--------

Hexadecimal representation of instruction:

0 0 E 5 5 0 2 0₁₆



6. I-format (1/4)

- What about instructions with immediate values?
 - 5-bit **shamt** field can only represent **0 to 31**
 - Immediates may be much larger than this
 - e.g. **lw**, **sw** instructions require bigger offset
- **Compromise:** Define a new instruction format partially consistent with R-format:
 - If instruction has immediate, then it uses at most 2 registers



6. I-format (2/4)

- Define fields with the following number of bits each:
 - $6 + 5 + 5 + 16 = 32$ bits



- Again, each field has a name:



- Only one field is inconsistent with R-format.
 - opcode**, **rs**, and **rt** are still in the same locations.



6. I-format (3/4)

■ **opcode**

- Since there is no **funct** field, **opcode** uniquely specifies an instruction

■ **rs**

- specifies the source register operand (if any)

■ **rt**

- specifies register to receive result
- **note the difference from R-format instructions**

■ Continue on next slide.....



6. I-format (4/4)

■ **immediate:**

- Treated as a *signed integer*
- 16 bits → can be used to represent a constant up to 2^{16} different values
- Large enough to handle:
 - The offset in a typical **lw** or **sw**
 - Most of the values used in the **addi**, **subi**, **slti** instructions



6.1 I-format: Example (1/2)

MIPS instruction
addi \$21, \$22, -50

I-Format Fields	Value	Remarks
opcode	8	(textbook pg 94 - 101)
rs	22	(the only source register)
rt	21	(target register)
immediate	-50	(in base 10)



6.1 I-format: Example (2/2)

MIPS instruction

addi \$21, \$22, -50

Field representation in decimal:

8	22	21	-50
---	----	----	-----

Field representation in binary:

001000	10110	10101	111111111001110
--------	-------	-------	-----------------

Hexadecimal representation of instruction:

2 2 D 5 F F C E₁₆



6.2 Try It Yourself #2

MIPS instruction

lw \$9, 12(\$8)

Field representation in decimal:

opcode	rs	rt	immediate
35	8	9	12

Field representation in binary:

100011	01000	01001	0000000000001100
--------	-------	-------	------------------

Hexadecimal representation of instruction:

8 D 0 9 0 0 0 C₁₆



6.3 Instruction Address: Overview

- As instructions are stored in memory, they too have addresses
 - Control flow instructions uses these addresses
 - E.g. **beq**, **bne**, **j**
- As instructions are 32-bit long, instruction addresses are word-aligned as well
- **Program Counter (PC)**
 - A special register that keeps address of instruction being executed in the processor



6.4 Branch: PC-Relative Addressing (1/5)

- Use I-Format

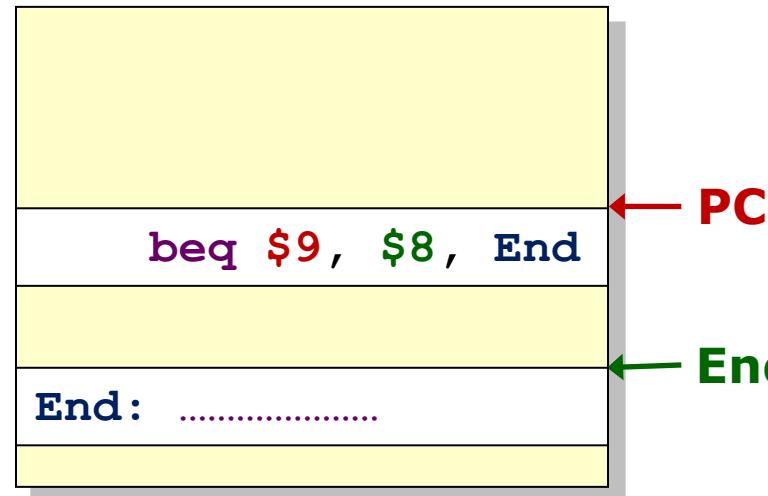
opcode	rs	rt	immediate
--------	----	----	-----------

- **opcode** specifies **beq, bne**
- **rs** and **rt** specify registers to compare
- What can **immediate** specify?
 - **Immediate** is only 16 bits
 - Memory address is 32 bits
 - → **immediate** is not enough to specify the entire target address!



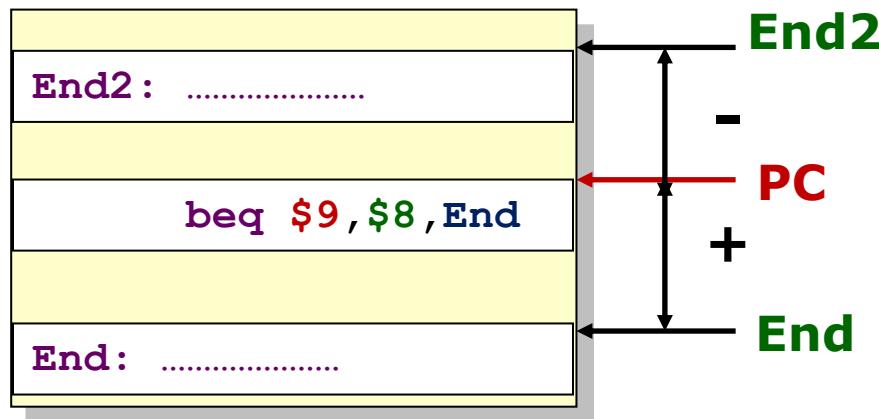
6.4 Branch: PC-Relative Addressing (2/5)

- How do we usually use branches?
 - Answer: **if-else, while, for**
 - Loops are generally **small**:
 - Typically up to 50 instructions
 - Unconditional jumps are done using jump instructions (**j**), not the branches
- **Conclusion:** A branch often changes **PC** by a small amount



6.4 Branch: PC-Relative Addressing (3/5)

- **Solution:**
 - Specify target address **relative to the PC**
 - Target address is generated as:
 - PC + the 16-bit **immediate** field
 - The **immediate** field is a signed two's complement integer
- Can branch to $\pm 2^{15}$ bytes from the PC:
- Should be enough to cover most loops



6.4 Branch: PC-Relative Addressing (4/5)

- Can the branch target range be enlarged?
- **Observation:** Instructions are word-aligned
 - Number of bytes to add to the PC will always be a multiple of 4.
 - ➔ Interpret the **immediate** as number of words, i.e. automatically multiplied by 4_{10} (100_2)
- ➔ Can branch to $\pm 2^{15}$ **words** from the PC
 - i.e. $\pm 2^{17}$ bytes from the PC
 - We can now branch 4 times farther!



6.4 Branch: PC-Relative Addressing (5/5)

- Branch calculation:

If the branch is **not taken**:

$$\text{PC} = \text{PC} + 4$$

(**PC** + 4 is address of next instruction)

If the branch is **taken**:

$$\text{PC} = (\text{PC} + 4) + (\text{immediate} \times 4)$$

- Observations:

- immediate** field specifies the number of words to jump, which is the same as the number of instructions to “skip over”
- immediate** field can be positive or negative
- Due to hardware design, add **immediate** to (PC+4), not to PC (more in later topic)



6.5 Branch: Example (1/3)

Loop:	beq \$9, \$0, End	# rlt addr: 0
	add \$8, \$8, \$10	# rlt addr: 4
	addi \$9, \$9, -1	# rlt addr: 8
	j Loop	# rlt addr: 12
End:		# rlt addr: 16

- **beq** is an I-Format instruction →

I-Format Fields	Value	Remarks
opcode	4	
rs	9	(first operand)
rt	0	(second operand)
immediate	???	(in base 10)



6.5 Branch: Example (2/3)

Loop:	beq \$9, \$0, End	# rlt addr: 0
	add \$8, \$8, \$10	# rlt addr: 4
	addi \$9, \$9, -1	# rlt addr: 8
	j Loop	# rlt addr: 12
End:		# rlt addr: 16

- **immediate** field:
 - Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch
 - In **beq** case, **immediate = 3**
 - **End = (PC + 4) + (immediate × 4)**



6.5 Branch: Example (3/3)

```

Loop:    beq $9, $0, End      # rlt addr: 0
          add $8, $8, $10      # rlt addr: 4
          addi $9, $9, -1       # rlt addr: 8
          j   Loop              # rlt addr: 12
End:      # rlt addr: 16
  
```

Field representation in decimal:

opcode	rs	rt	immediate
4	9	0	3

Field representation in binary:

000100	01001	00000	0000000000000011
--------	-------	-------	------------------



6.6 Try It Yourself #3

```
Loop:    beq    $9,  $0,  End    # rlt addr: 0
          add    $8,  $8,  $10   # rlt addr: 4
          addi   $9,  $9,  -1    # rlt addr: 8
          beq    $0,  $0,  Loop  # rlt addr: 12
End:      # rlt addr: 16
```

- What would be the **immediate** value for the second **beq** instruction?

Answer: – 4



7. J-Format (1/5)

- For branches, PC-relative addressing was used:
 - Because we do not need to branch too far
- For general jumps (**j**):
 - We may jump to anywhere in memory!
- The ideal case is to specify a 32-bit memory address to jump to
 - Unfortunately, we can't (⊗ why?)



7. J-Format (2/5)

- Define fields of the following number of bits each:

6 bits	26 bits
--------	---------

- As usual, each field has a name:

opcode	target address
--------	----------------

- Keep **opcode** field identical to R-format and I-format for consistency
- Combine all other fields to make room for larger target address



7. J-Format (3/5)

- We can only specify 26 bits of 32-bit address
 - **Optimization:**
 - Just like with branches, jumps will only jump to word-aligned addresses, so last two bits are always **00**
 - So, let's assume the address ends with '**00**' and leave them out
- Now we can specify **28 bits** of 32-bit address



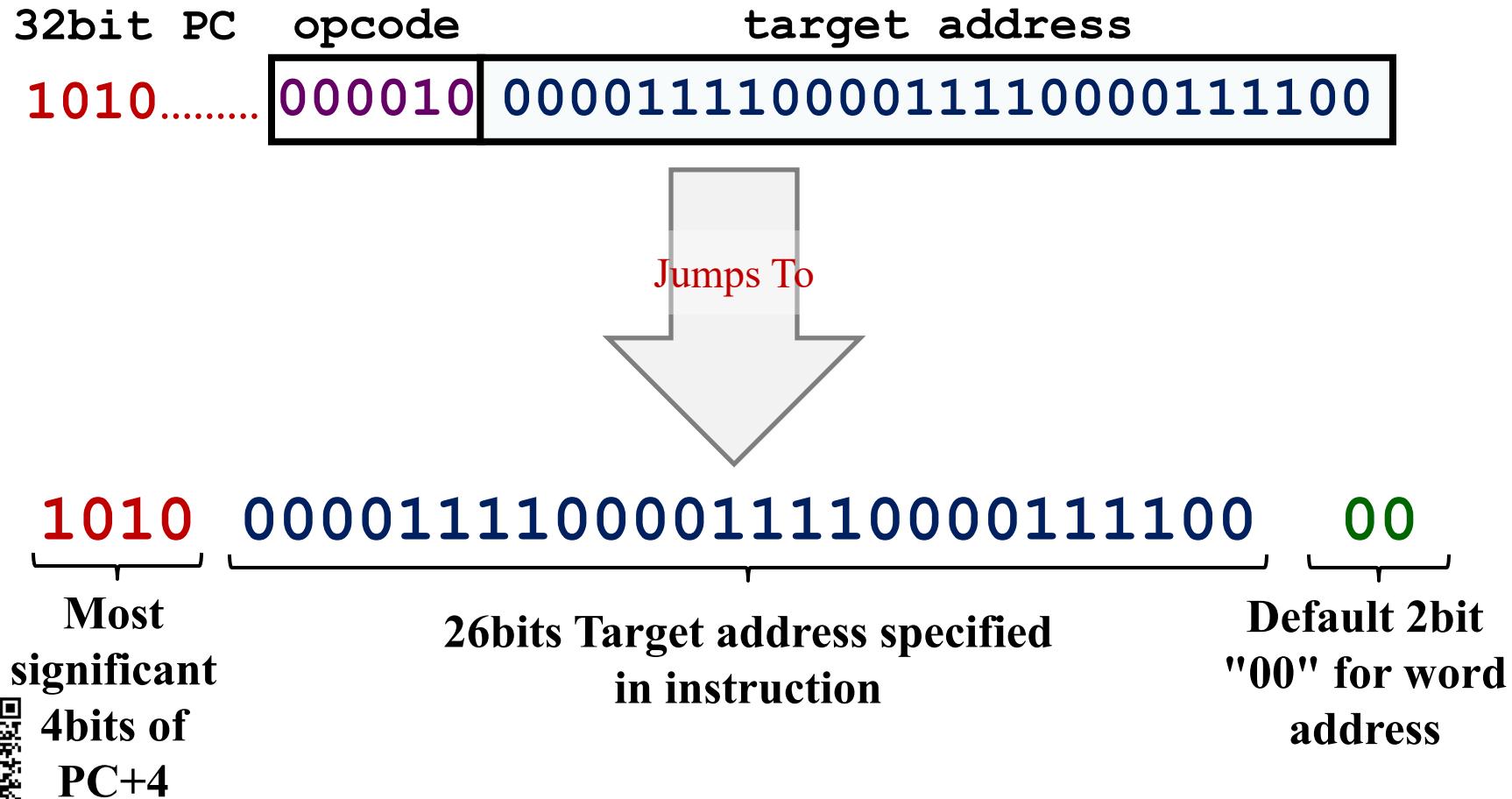
7. J-Format (4/5)

- Where do we get the other 4 bits?
 - MIPS choose to take the **4 most significant bits from PC+4** (the next instruction after the jump instruction)
→ This means that we cannot jump to anywhere in memory, but it should be sufficient ***most of the time***
- Question:
 - What is the **maximum jump range?** 256MB boundary
 - Special instruction if the program straddles 256MB boundary
 - Look up **jr** instruction if you are interested
 - Target address is specified through a register



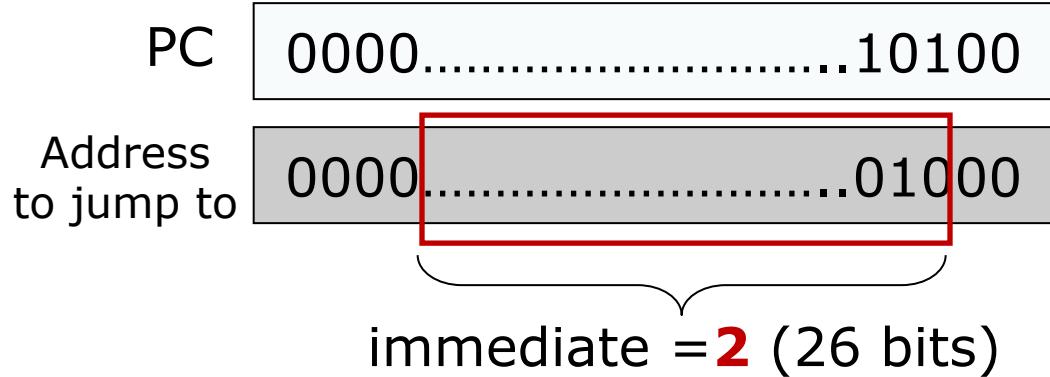
7. J-Format (5/5)

- **Summary:** Given a **Jump** instruction



7.1 J-Format: Example

Loop:	beq \$9, \$0, End #	addr: 8 ←	jump target
	add \$8, \$8, \$10 #	addr: 12	
	addi \$9, \$9, -1 #	addr: 16	
	j Loop #	addr: 20 ← PC	
End:		# addr: 24	



Check your understanding by constructing the new PC value

opcode	target address
000010	0000000000000000000000000010

7.2 Branching Far Way

- Given the instruction

beq \$s0, \$s1, L1

Assume that the address **L1** is farther away from the PC than can be supported by **beq** and **bne** instructions

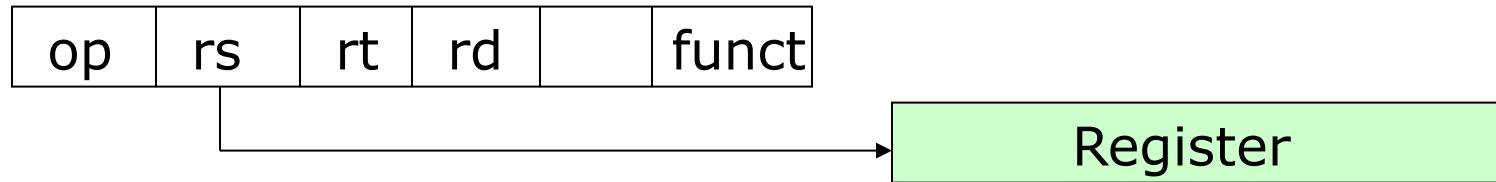
- Challenge:**

- Construct an equivalent code sequence with the help of unconditional (**j**) and conditional branch (**beq**, **bne**) instructions to accomplish this far away branching



8. Addressing Modes (1/3)

- **Register addressing:** operand is a register

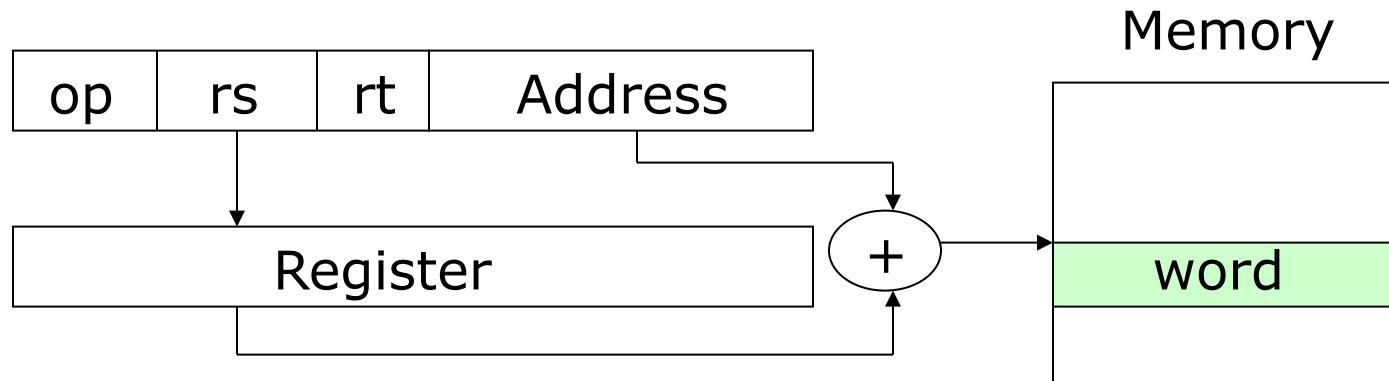


- **Immediate addressing:** operand is a constant within the instruction itself (**addi**, **andi**, **ori**, **slti**)



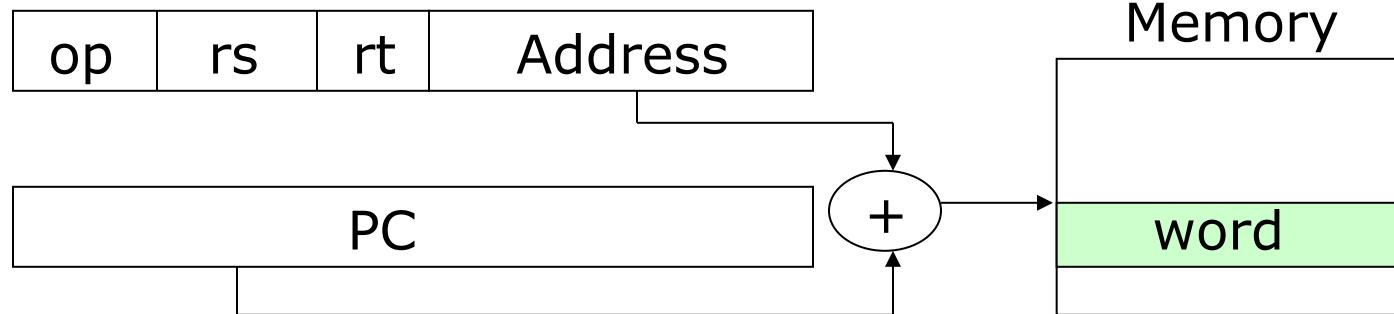
8. Addressing Modes (2/3)

- **Base addressing (displacement addressing):**
operand is at the memory location whose address is sum of a register and a constant in the instruction
(lw, sw)

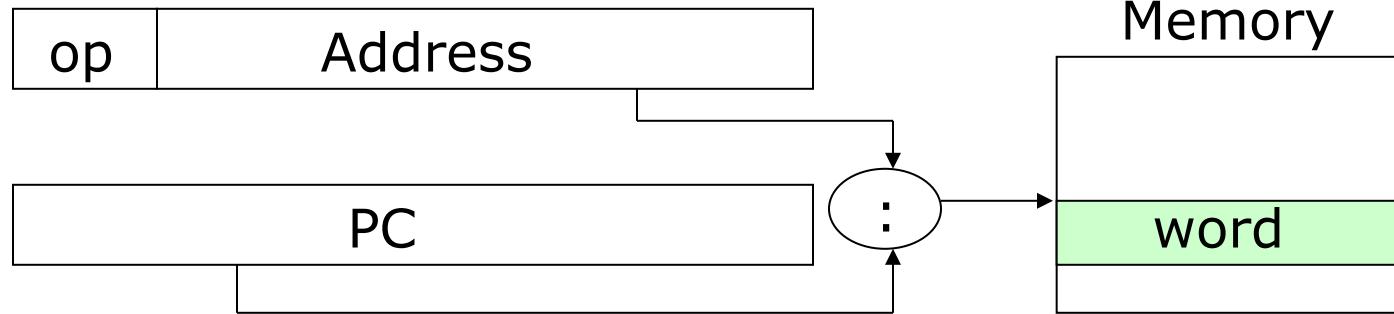


8. Addressing Modes (3/3)

- PC-relative addressing:** address is sum of PC and constant in the instruction (**beq**, **bne**)



- Pseudo-direct addressing:** 26-bit of instruction concatenated with upper 4-bits of PC (**j**)



Summary (1/2)

- MIPS Instruction:

32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct	
I	opcode	rs	rt	immediate			
J	opcode	target address					

- Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
- Branches use PC-relative addressing; jumps use pseudo-direct addressing
- Shifts use R-format, but other immediate instructions (**addi**, **andi**, **ori**) use I-format



Summary (2/2)

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
	branch on equal	beq \$s1, \$s2, 25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional 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 less than immediate	slti \$s1, \$s2, 100	if ($\$s2 < 100$) \$s1 = 1; else \$s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For sw itch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

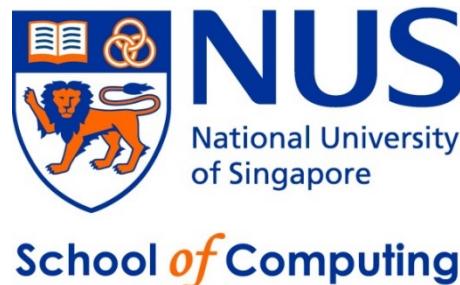


IT5002

Computer Systems and Applications

Datapath Design

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lecture #7: Datapath Design

- 1. Building a Processor: Datapath & Control**
- 2. MIPS Processor: Implementation**
- 3. Instruction Execution Cycle (Recap)**
- 4. MIPS Instruction Execution**
- 5. Let's Build a MIPS Processor**
 - 5.1 Fetch Stage
 - 5.2 Decode Stage
 - 5.3 ALU Stage
 - 5.4 Memory Stage
 - 5.5 Register Write Stage
- 6. The Complete Datapath!**



1. Building a Processor: Datapath & Control

- **Two major components for a processor**

Datapath

- Collection of components that process data
- Performs the arithmetic, logical and memory operations

Control

- Tells the datapath, memory and I/O devices what to do according to program instructions

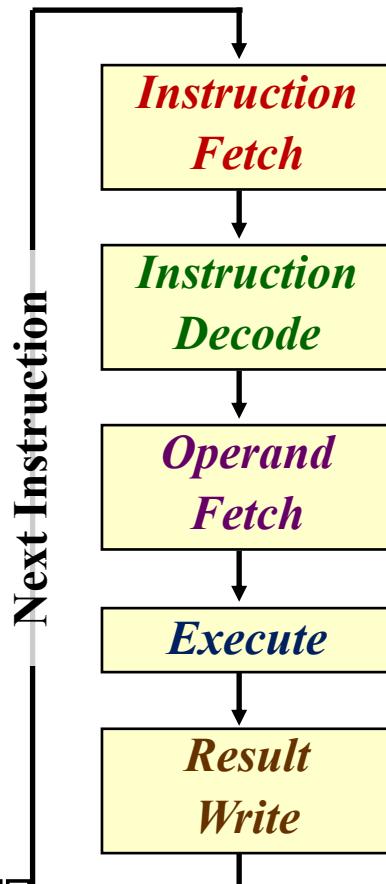


2. MIPS Processor: Implementation

- Simplest possible implementation of a subset of the core MIPS ISA:
 - **Arithmetic and Logical operations**
 - `add`, `sub`, `and`, `or`, `addi`, `andi`, `ori`, `slt`
 - **Data transfer instructions**
 - `lw`, `sw`
 - **Branches**
 - `beq`, `bne`
- Shift instructions (`sll`, `srl`) and J-type instructions (`j`) will not be discussed:
 - Left as exercises ☺



3. Instruction Execution Cycle (Basic)



1.

Fetch:

- Get instruction from memory
- Address is in Program Counter (PC) Register

2.

Decode:

- Find out the operation required

3.

Operand Fetch:

- Get operand(s) needed for operation

4.

Execute:

- Perform the required operation

5.

Result Write (Store):

- Store the result of the operation

4. MIPS Instruction Execution (1/2)

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stages not shown:
 - The standard steps are performed

	add \$3, \$1, \$2	lw \$3, 20(\$1)	beq \$1, \$2, ofst
Fetch	<i>standard</i>	<i>standard</i>	<i>standard</i>
Decode			
Operand Fetch	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Use 20 as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i>
Execute	$Result = opr1 + opr2$	<ul style="list-style-type: none"> ○ $MemAddr = opr1 + opr2$ ○ Use <i>MemAddr</i> to read from memory 	$Taken = (opr1 == opr2) ?$ $Target = (\text{PC}+4) + \text{ofst} \times 4$
Result Write	<i>Result stored in \$3</i>	<i>Memory data stored in \$3</i>	$\text{if } (Taken)$ $\quad \text{PC} = \text{Target}$


 opr = operand

MemAddr = Memory Address

■ ofst = offset

4. MIPS Instruction Execution (2/2)

- **Design changes:**
 - Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
 - Split *Execute* into *ALU* (Calculation) and *Memory Access*

	add \$3, \$1, \$2	lw \$3, 20(\$1)	beq \$1, \$2, ofst
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Use 20 as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i>
ALU	$Result = opr1 + opr2$	$MemAddr = opr1 + opr2$	$Taken = (opr1 == opr2) ?$ $Target = (\text{PC}+4) + ofst \times 4$
Memory Access		Use <i>MemAddr</i> to read from memory	
Result Write	<i>Result</i> stored in \$3	<i>Memory</i> data stored in \$3	$\text{if } (Taken)$ $\text{PC} = \text{Target}$



5. Let's Build a MIPS Processor

- **What we are going to do:**
 - Look at each stage closely, figure out the requirements and processes
 - Sketch a high level block diagram, then zoom in for each elements
 - With the simple starting design, check whether different type of instructions can be handled:
 - **Add modifications when needed**
- ➔ **Study the design from the viewpoint of a designer, instead of a "tourist" ☺**



5.1 Fetch Stage: Requirements

- Instruction Fetch Stage:

1. Use the **Program Counter (PC)** to fetch the instruction from **memory**
 - PC is implemented as a special register in the processor

2. **Increment** the PC by 4 to get the address of the next instruction:

- How do we know the next instruction is at $\text{PC}+4$?
- Note the exception when branch/jump instruction is executed

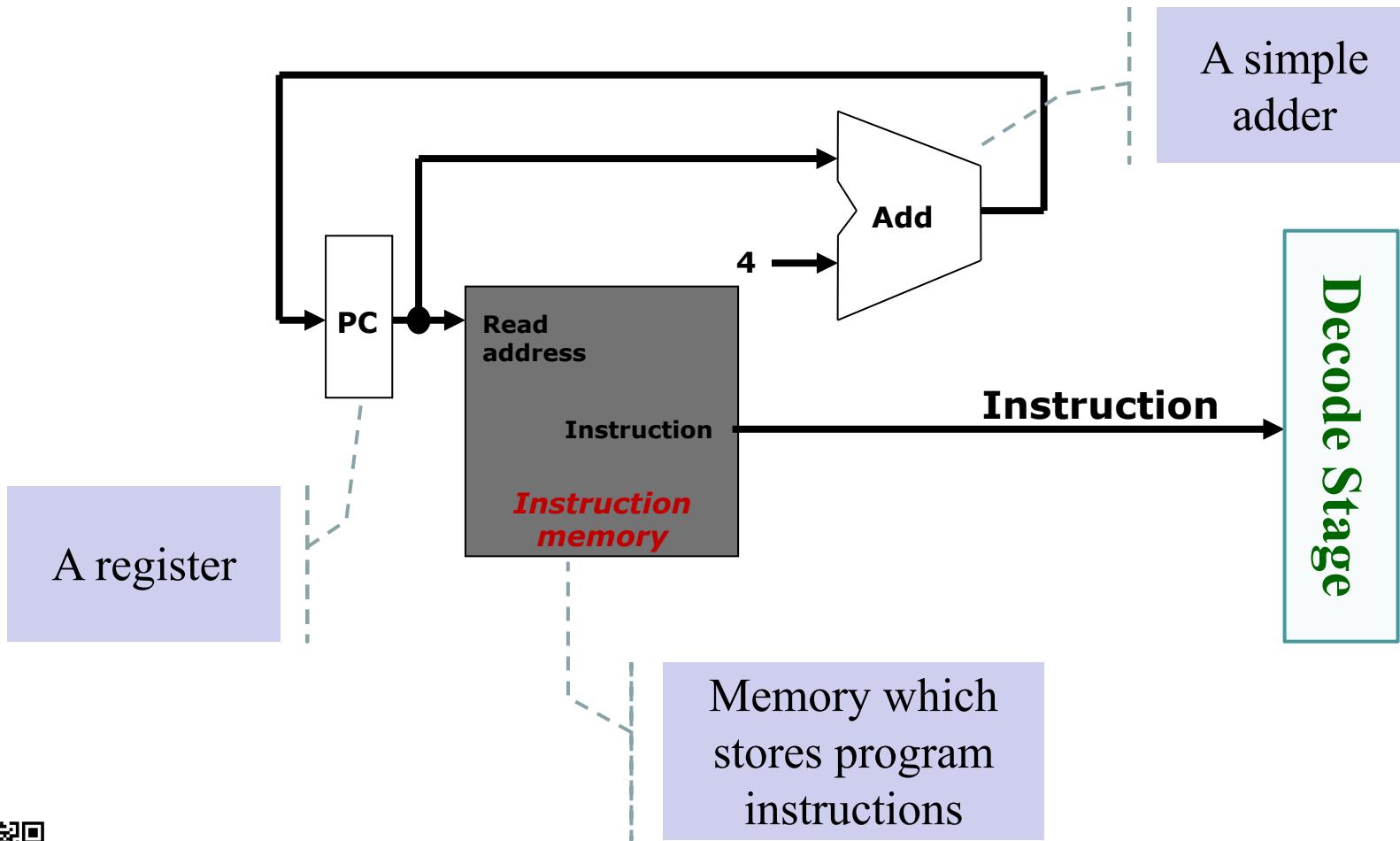
- Output to the next stage (**Decode**):

- The instruction to be executed

1. **Fetch**
2. Decode
3. ALU
4. Memory
5. RegWrite

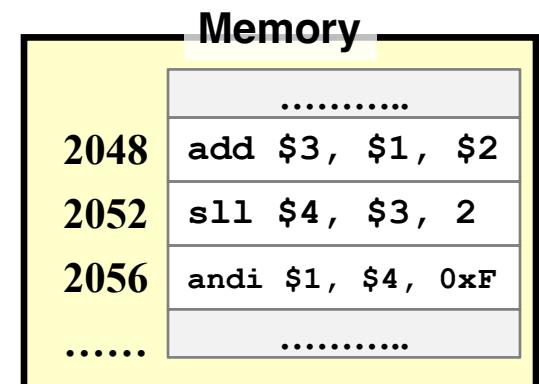
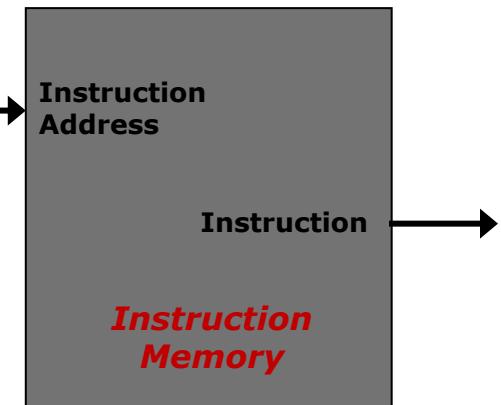


5.1 Fetch Stage: Block Diagram



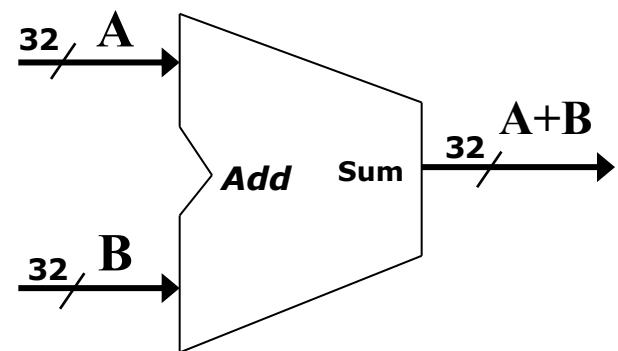
5.1 Element: Instruction Memory

- Storage element for the instructions
 - It is a **sequential circuit** (to be covered later) →
 - Has an internal state that stores information
 - Clock signal is assumed and not shown
- Supply instruction given the address
 - Given instruction address M as input, the memory outputs the content at address M
 - Conceptual diagram of the memory layout is given on the right →



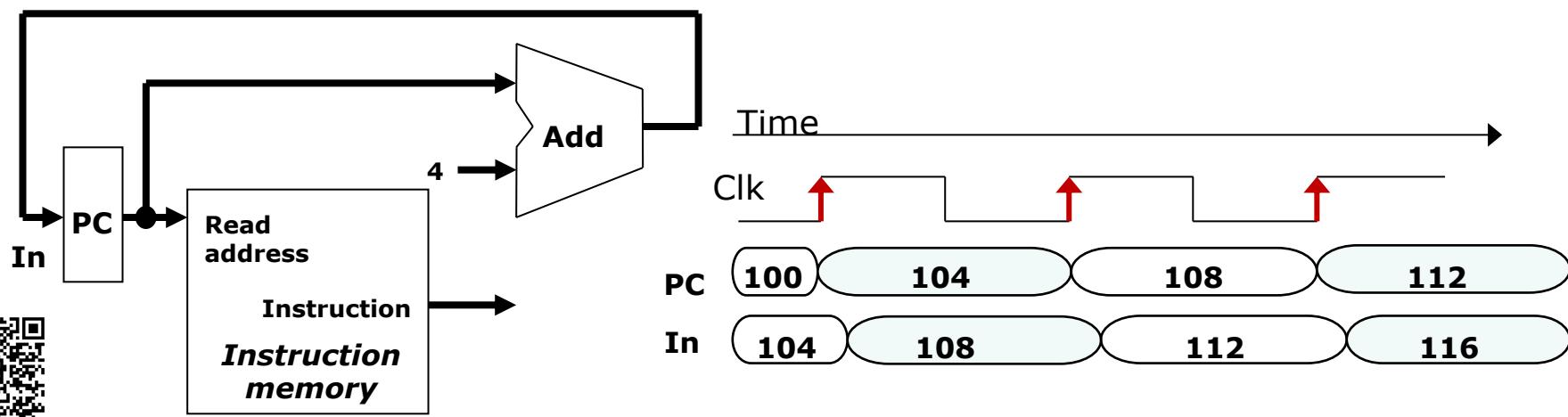
5.1 Element: Adder

- Combinational logic to implement the addition of two numbers
- **Inputs:**
 - Two 32-bit numbers A, B
- **Output:**
 - Sum of the input numbers, $A + B$



5.1 The Idea of Clocking

- It seems that we are reading and updating the PC at the same time:
 - How can it work properly?
- **Magic of clock:**
 - PC is read during the first half of the clock period and it is updated with $\text{PC}+4$ at the **next rising clock edge**



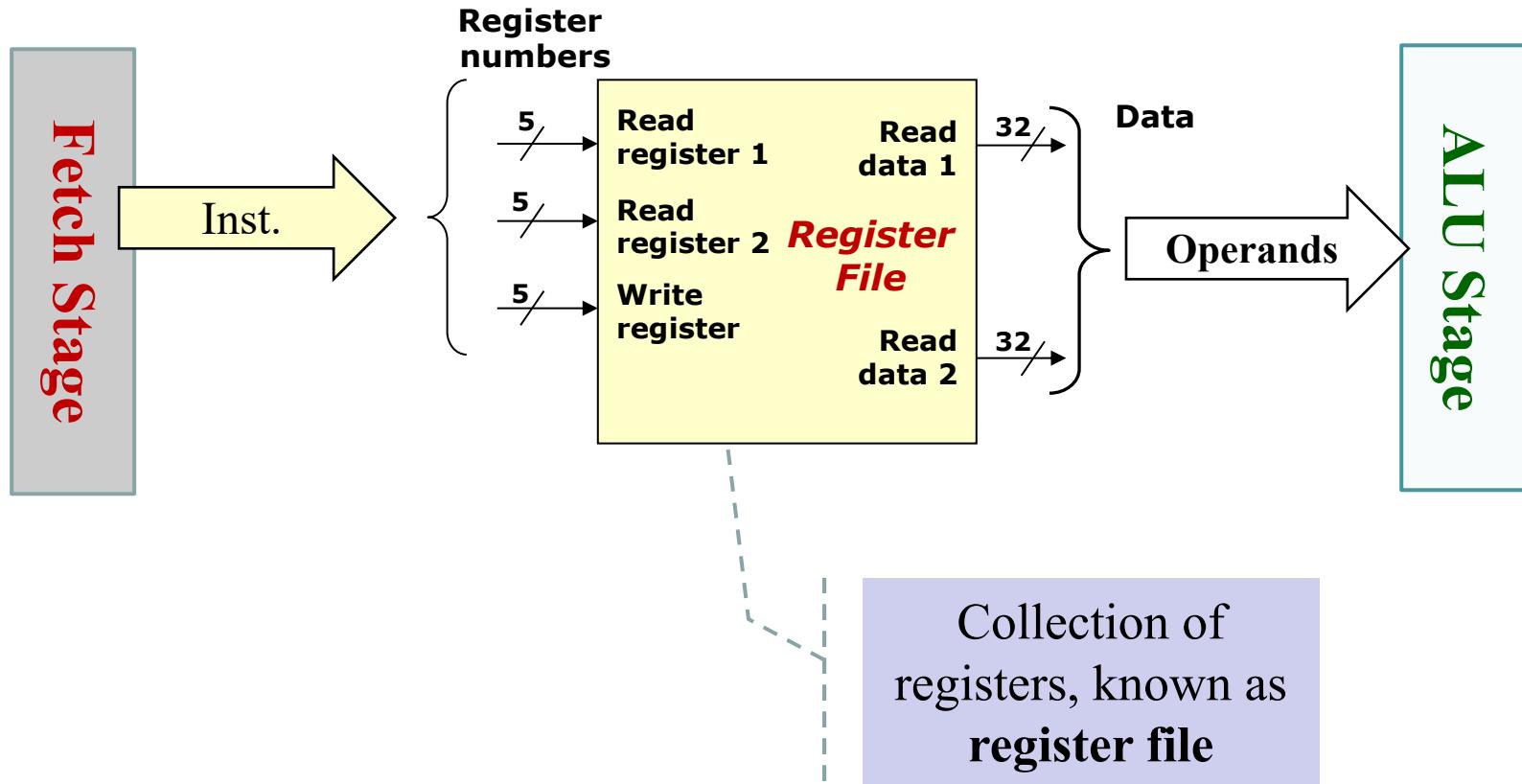
5.2 Decode Stage: Requirements

- **Instruction Decode Stage:**
 - Gather data from the instruction fields:
 1. Read the **opcode** to determine instruction type and field lengths
 2. Read data from all necessary registers
 - **Can be two (e.g. add), one (e.g. addi) or zero (e.g. j)**
- **Input from previous stage (Fetch):**
 - Instruction to be executed
- **Output to the next stage (ALU):**
 - Operation and the necessary operands

1. Fetch
2. **Decode**
3. ALU
4. Memory
5. RegWrite

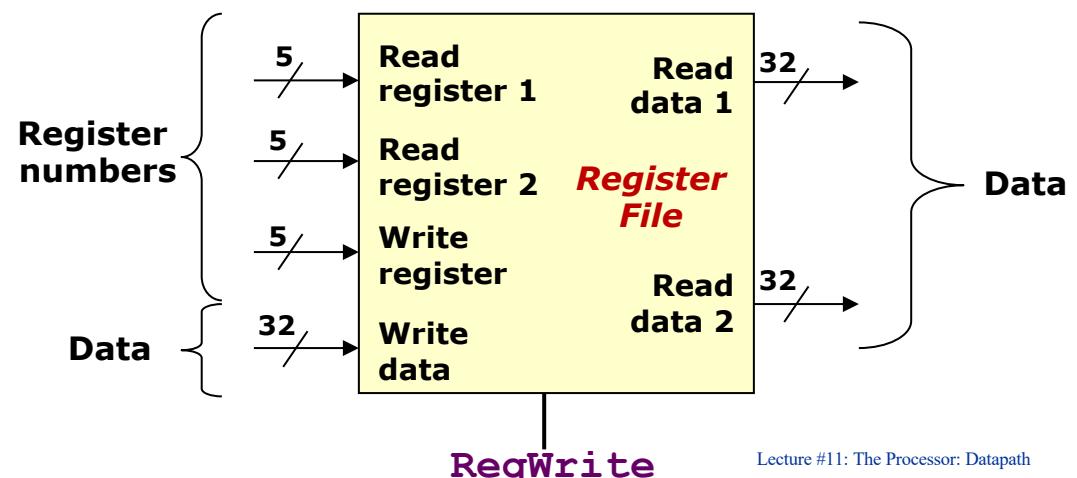


5.2 Decode Stage: Block Diagram

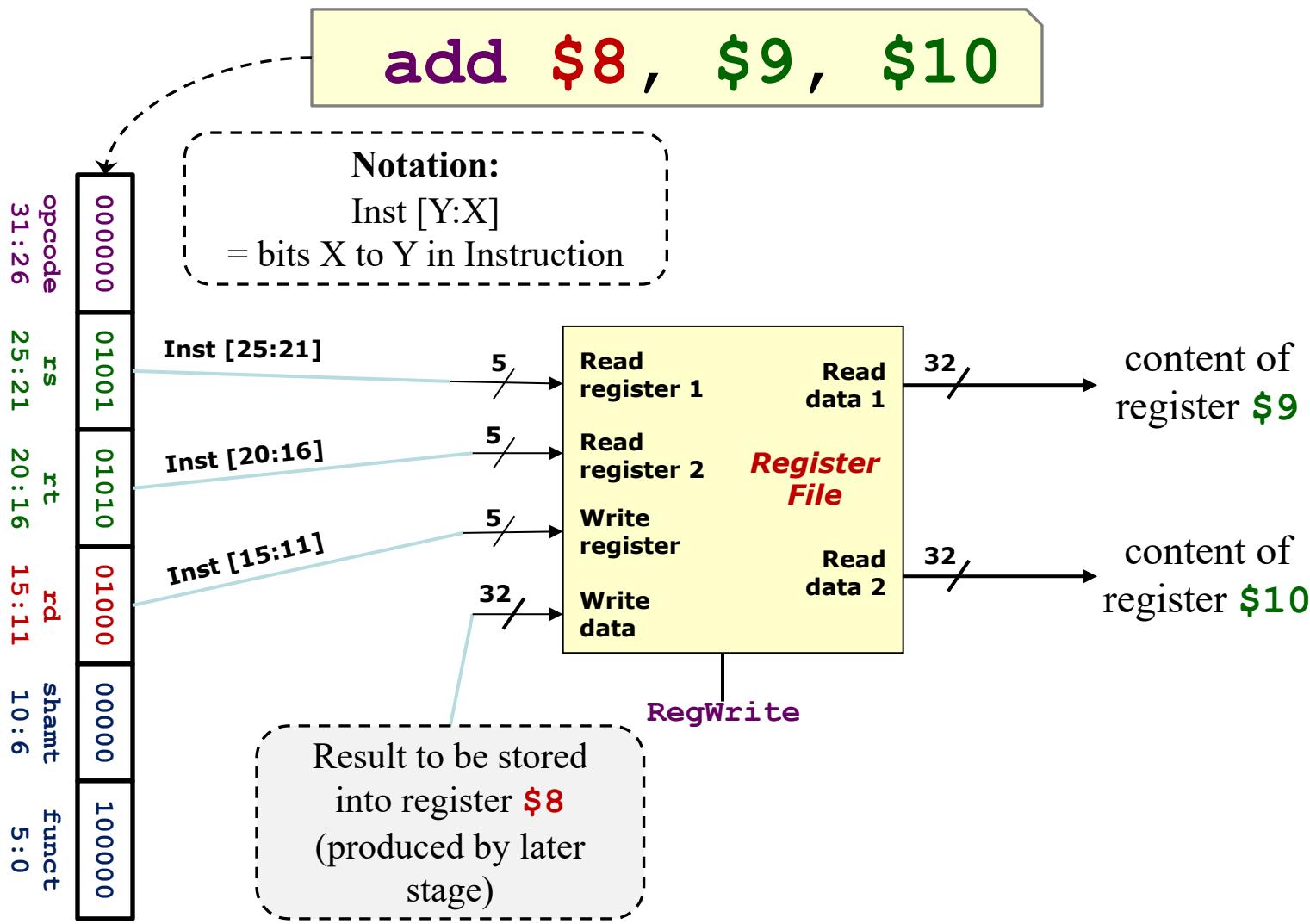


5.2 Element: Register File

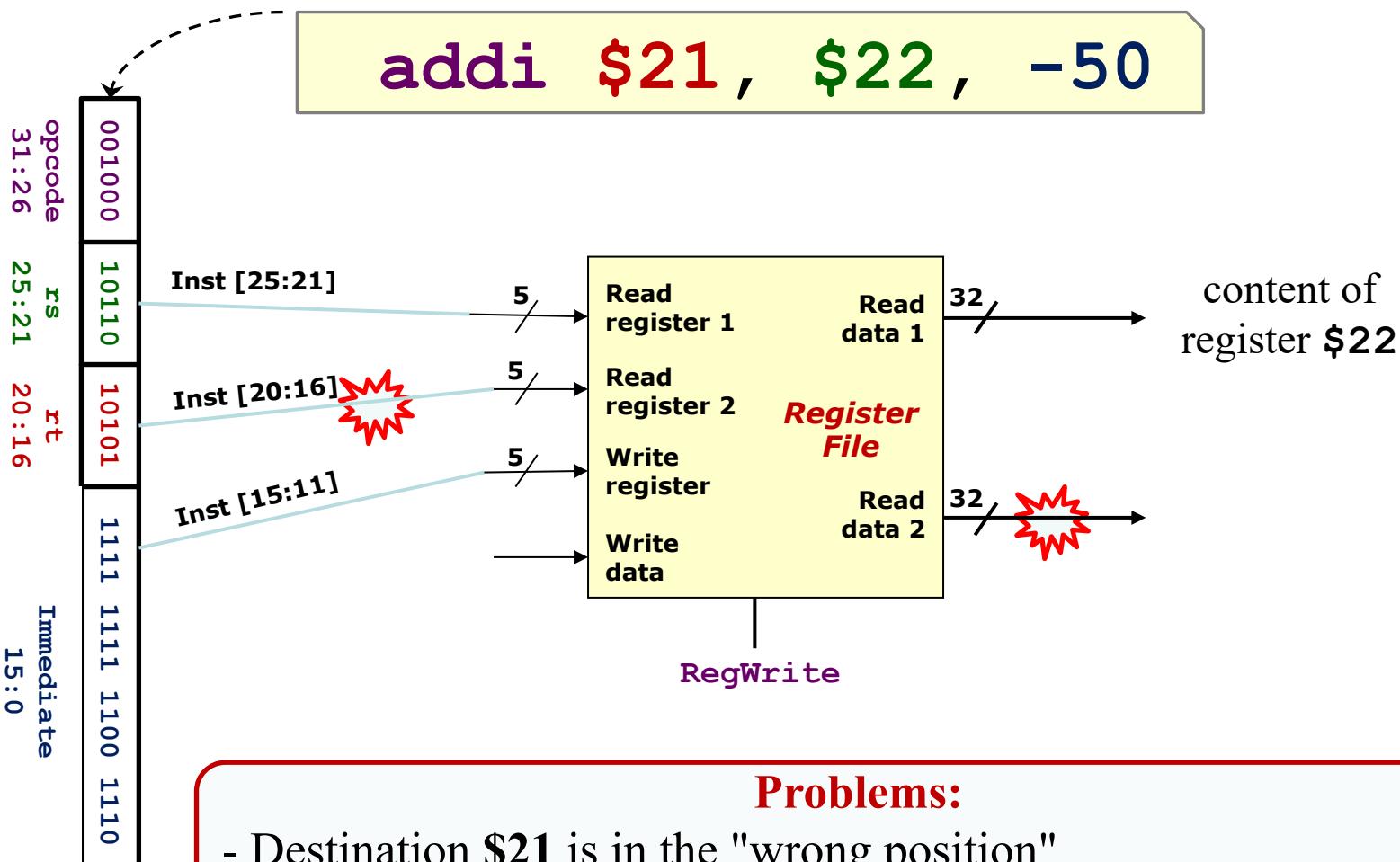
- A collection of 32 registers:
 - Each 32-bit wide; can be read/written by specifying register number
 - Read at most two registers per instruction
 - Write at most one register per instruction
- **RegWrite** is a control signal to indicate:
 - Writing of register
 - 1(True) = Write, 0 (False) = No Write



5.2 Decode Stage: R-Format Instruction



5.2 Decode Stage: I-Format Instruction

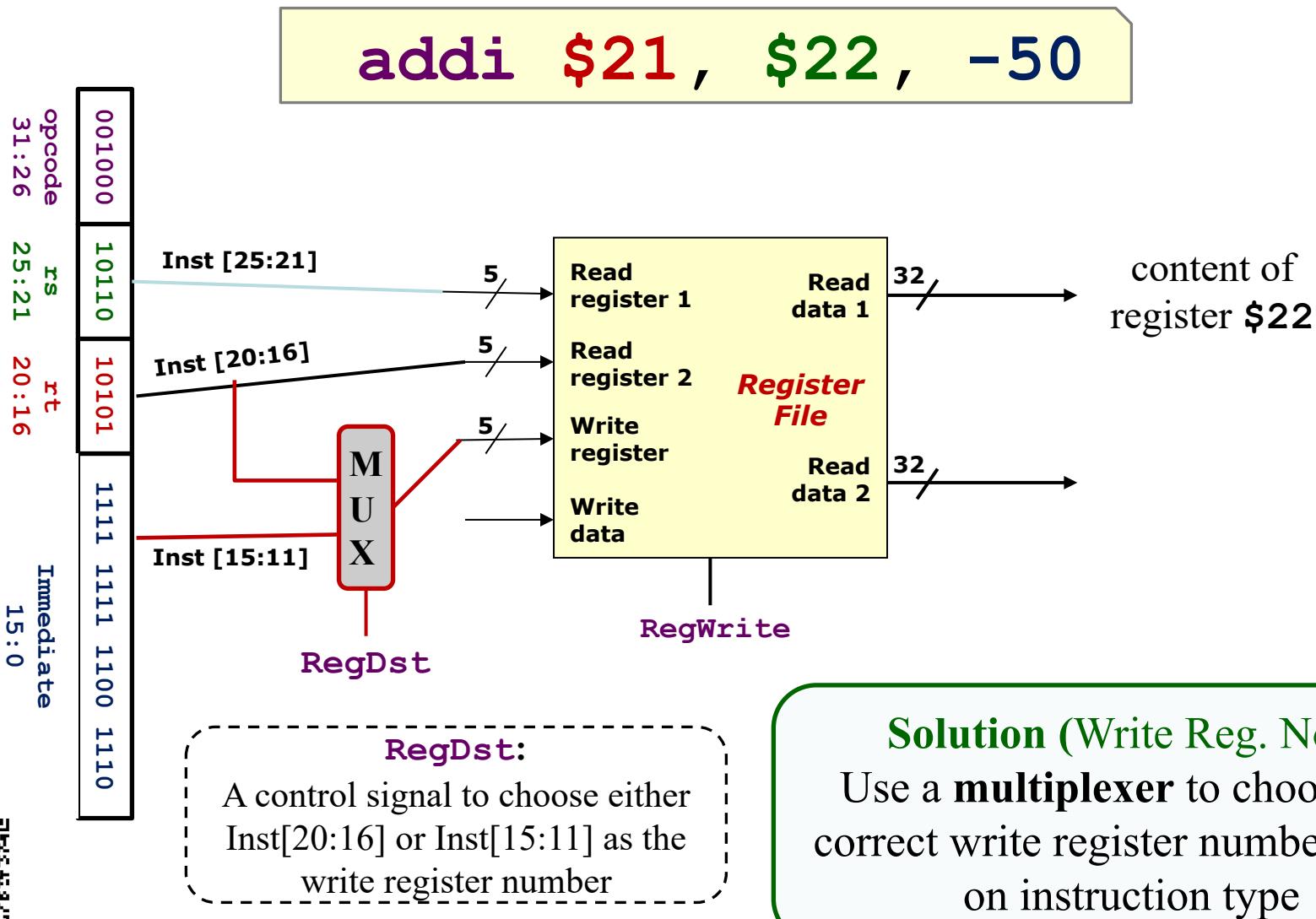


Problems:

- Destination **\$21** is in the "wrong position"
- **Read Data 2** is an immediate value, not from register



5.2 Decode Stage: Choice in Destination



5.2 Multiplexer

Function:

- Selects one input from multiple input lines

Inputs:

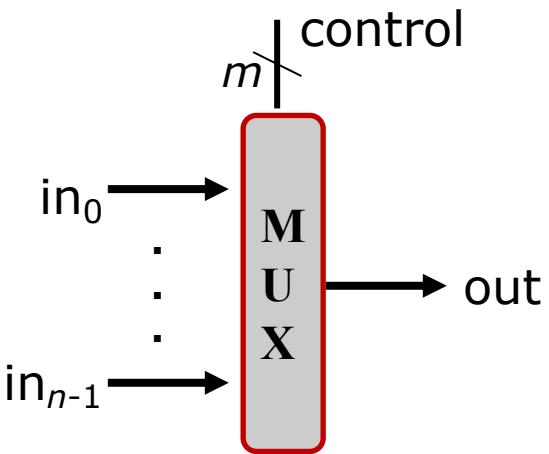
- n lines of same width

Control:

- m bits where $n = 2^m$

Output:

- Select i^{th} input line if control = i

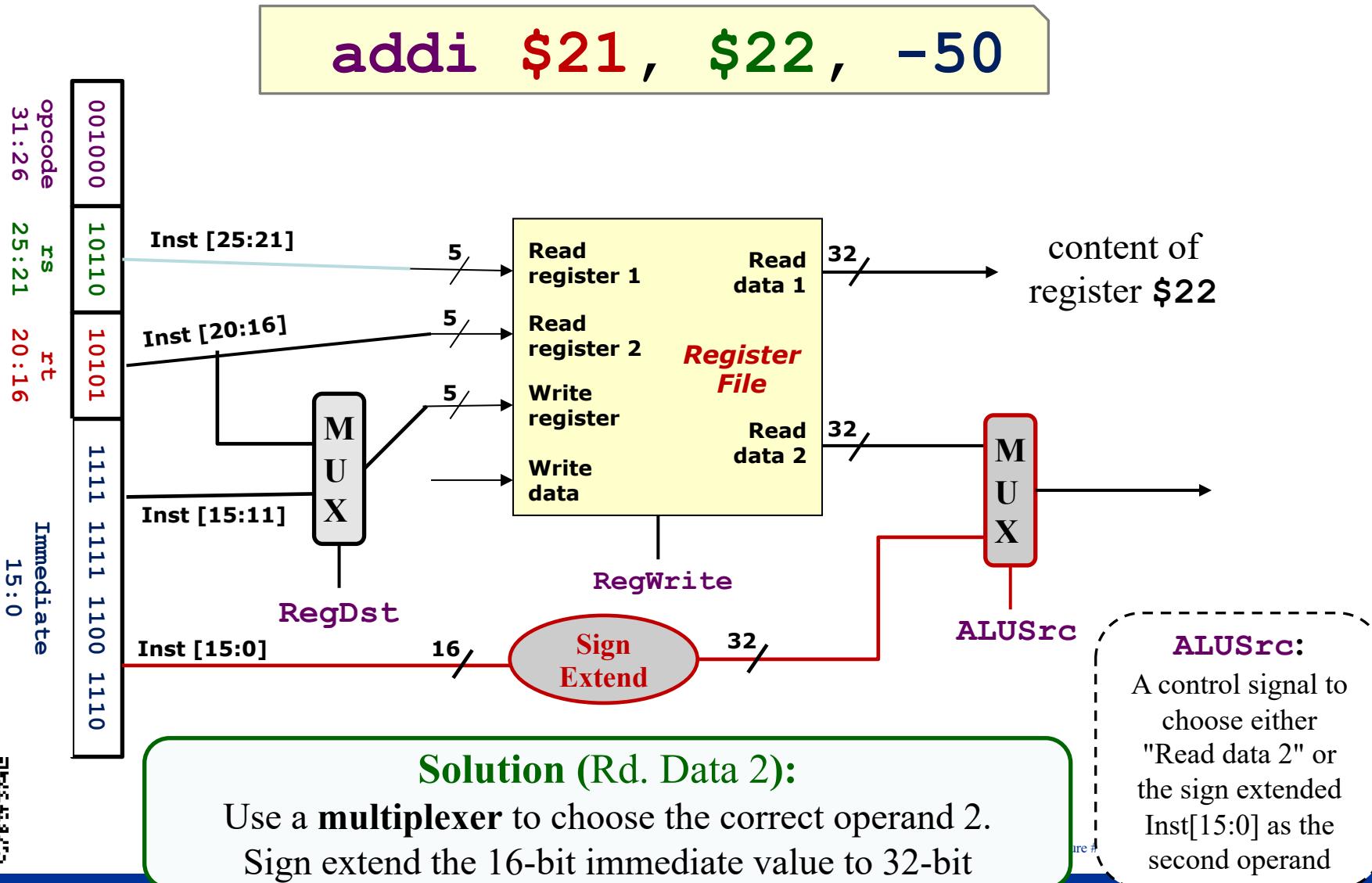


Control=0 → select in_0 to out

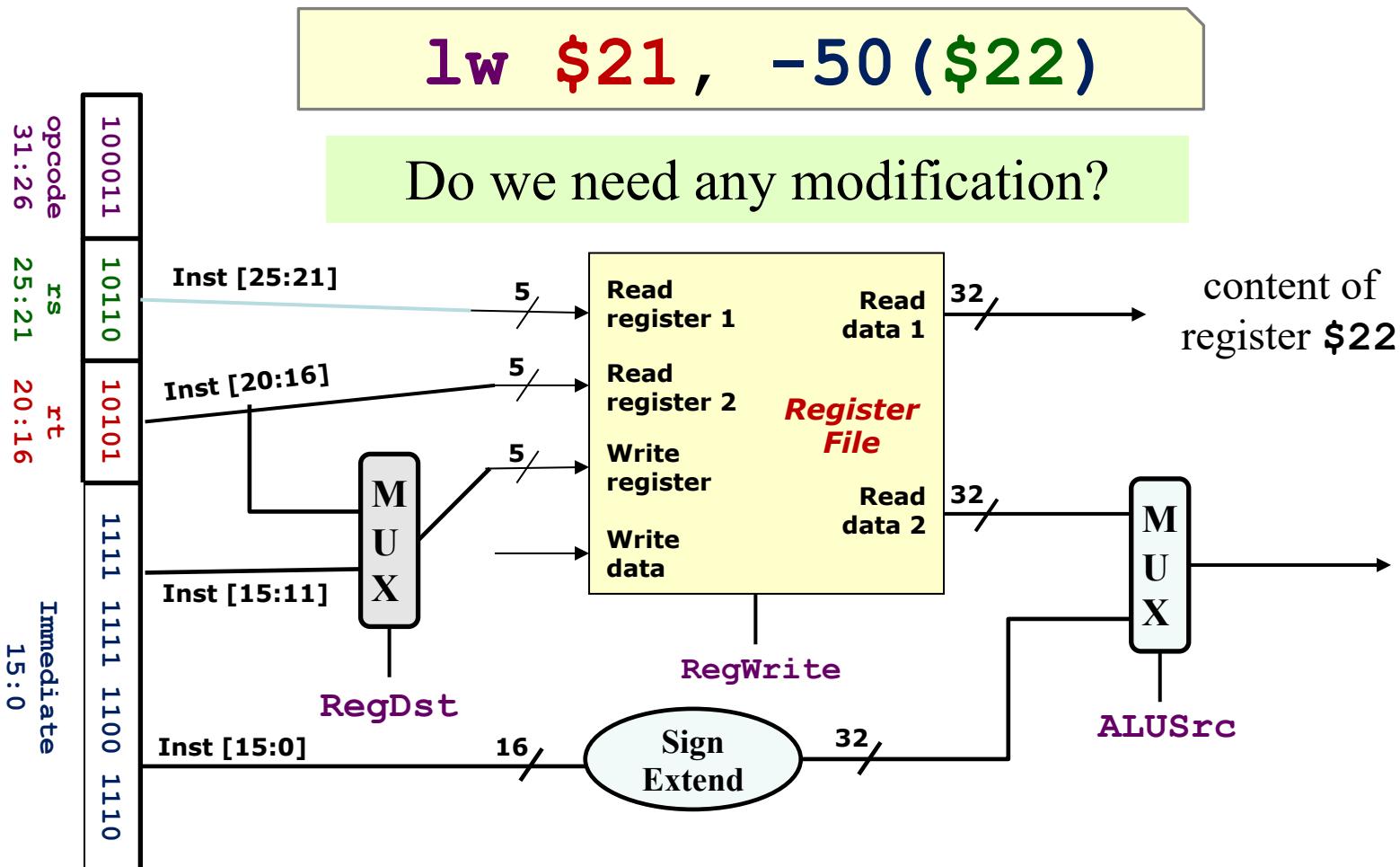
Control=3 → select in_3 to out



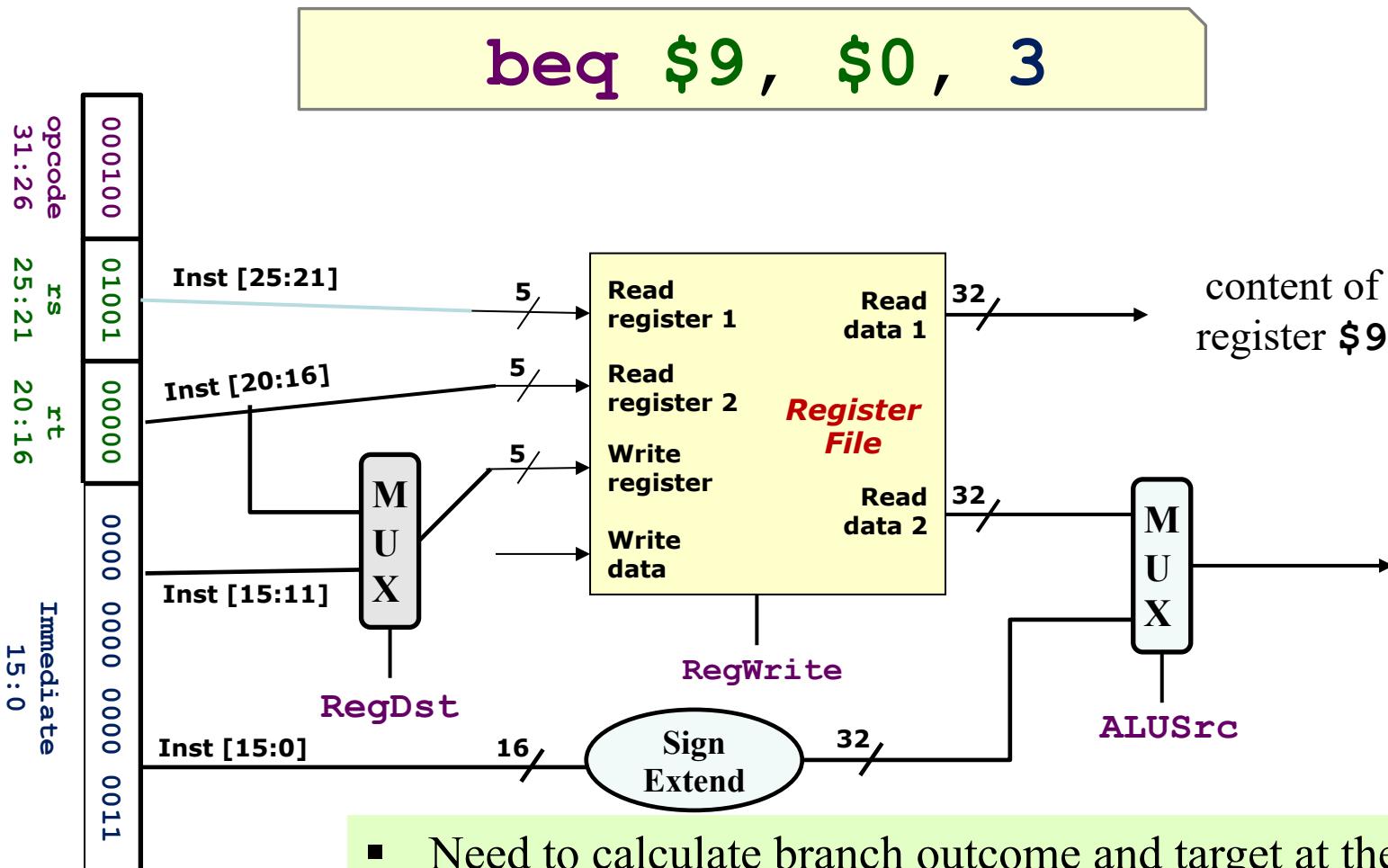
5.2 Decode Stage: Choice in Data 2



5.2 Decode Stage: Load Word Instruction



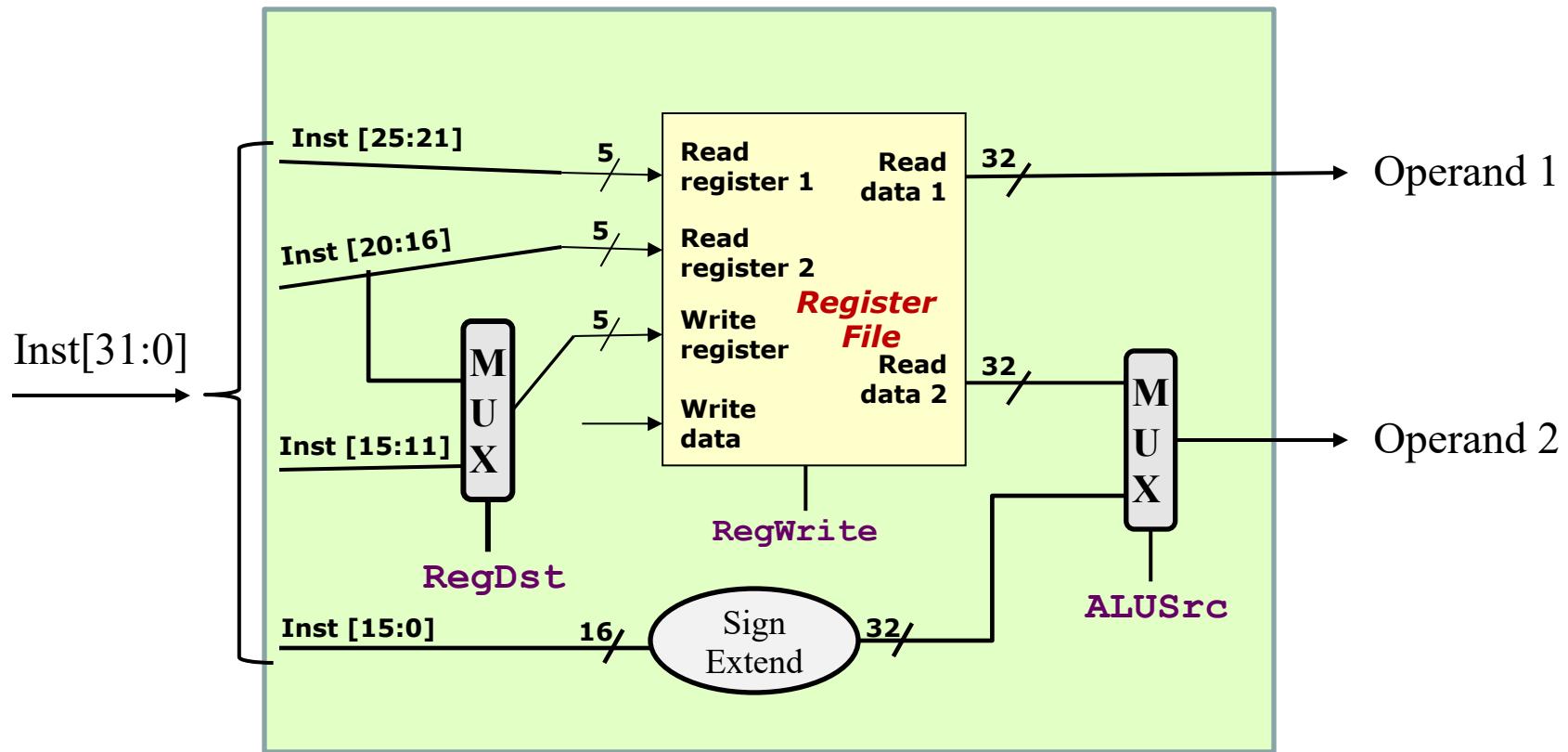
5.2 Decode Stage: Branch Instruction



- Need to calculate branch outcome and target at the same time!
- We will tackle this problem at the ALU stage



5.2 Decode Stage: Summary



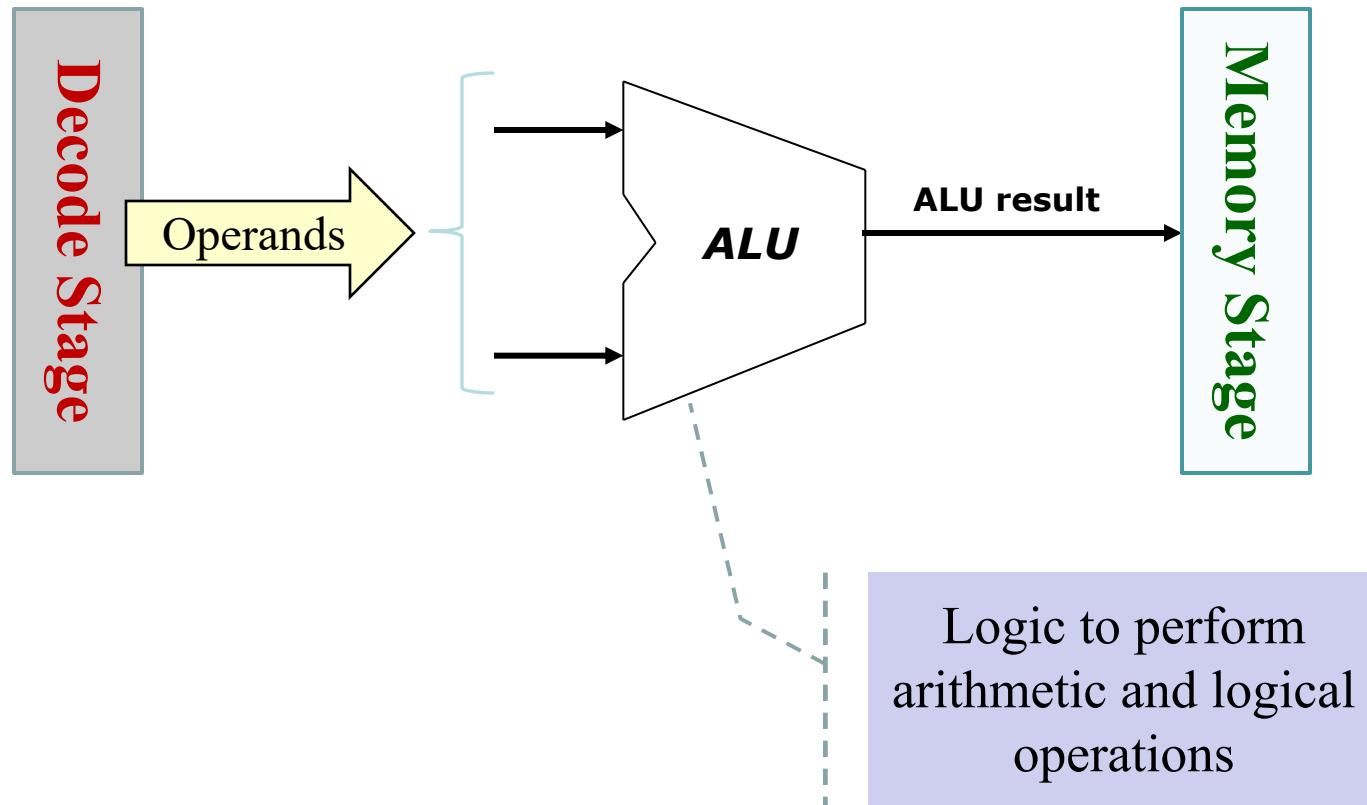
5.3 ALU Stage: Requirements

- **Instruction ALU Stage:**
 - ALU = Arithmetic-Logic Unit
 - Also called the **Execution stage**
 - Perform the real work for most instructions here
 - Arithmetic (e.g. **add**, **sub**), Shifting (e.g. **sll**), Logical (e.g. **and**, **or**)
 - Memory operation (e.g. **lw**, **sw**): Address calculation
 - Branch operation (e.g. **bne**, **beq**): Perform register comparison and target address calculation
- **Input from previous stage (Decode):**
 - Operation and Operand(s)
- **Output to the next stage (Memory):**
 - Calculation result

1. Fetch
2. Decode
- 3. ALU**
4. Memory
5. RegWrite



5.3 ALU Stage: Block Diagram



5.3 Element: Arithmetic Logic Unit

■ ALU (Arithmetic Logic Unit)

- Combinational logic to implement arithmetic and logical operations

■ Inputs:

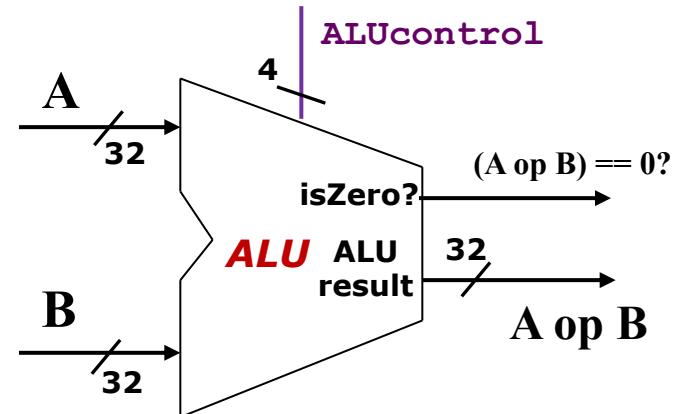
- Two 32-bit numbers

■ Control:

- 4-bit to decide the particular operation

■ Outputs:

- Result of arithmetic/logical operation
- A 1-bit signal to indicate whether result is zero

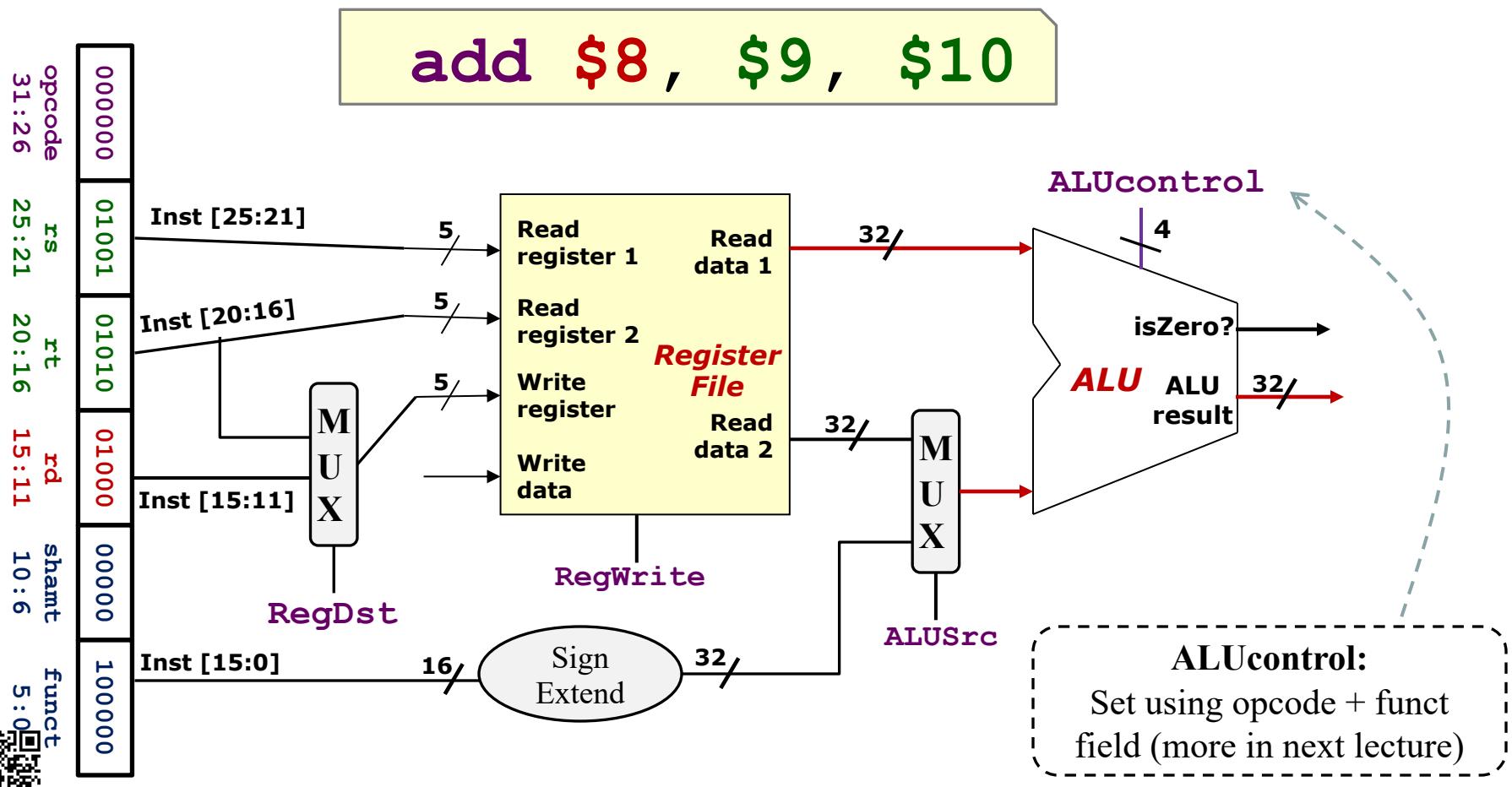


ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR



5.3 ALU Stage: Non-Branch Instructions

- We can handle non-branch instructions easily:



5.3 ALU Stage: Branch Instructions

- Branch instruction is harder as we need to perform two calculations:
- Example: "**beq \$9, \$0, 3**"

1. Branch Outcome:

- Use ALU to compare the register
- The 1-bit "isZero?" signal is enough to handle equal/not equal check (how?)

2. Branch Target Address:

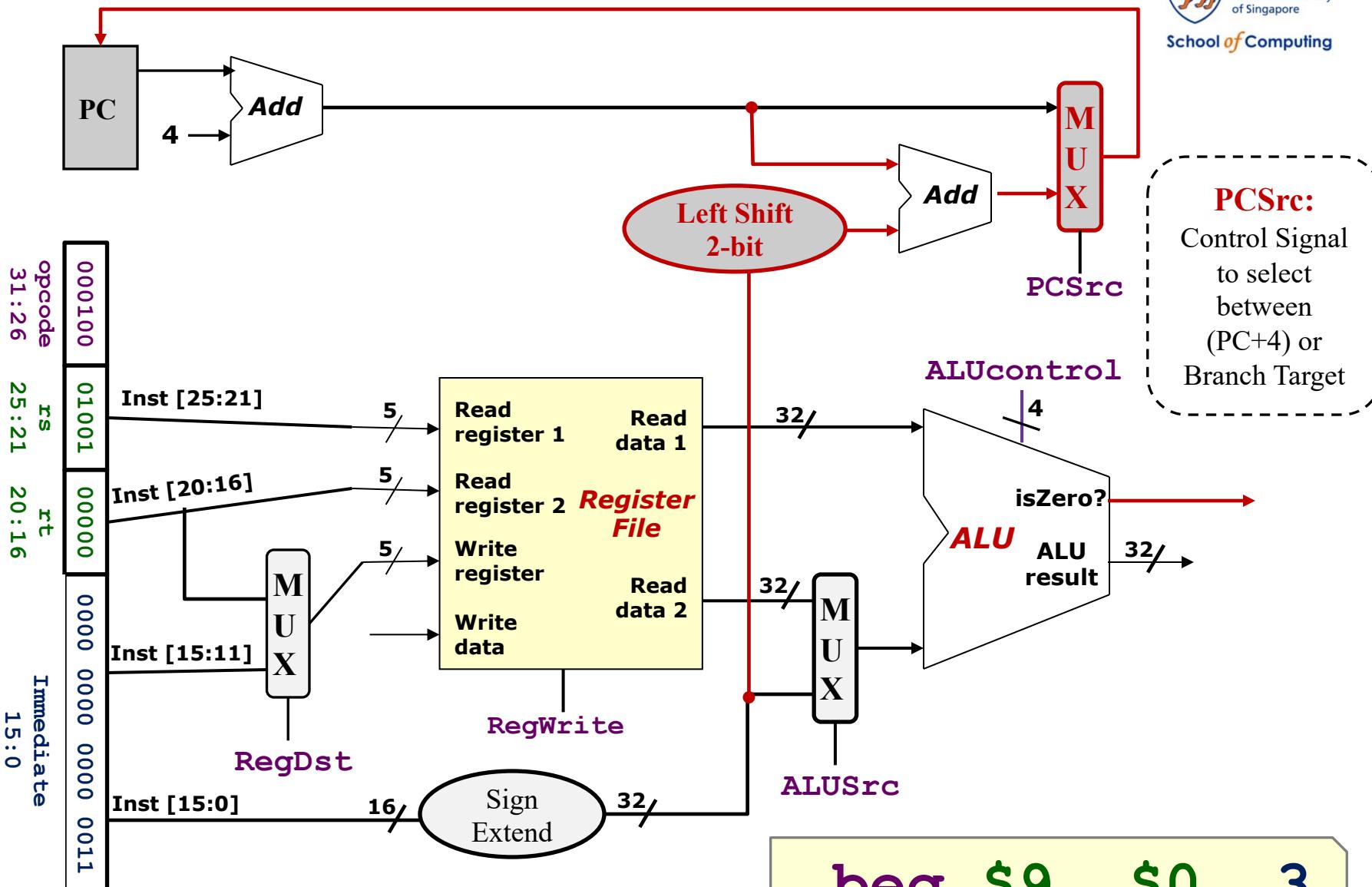
- Introduce additional logic to calculate the address
- Need PC (from Fetch Stage)
- Need Offset (from Decode Stage)



Complete ALU Stage

EE5002 Lecture 7 Datapath Design

Page: 31



beq \$9, \$0, 3

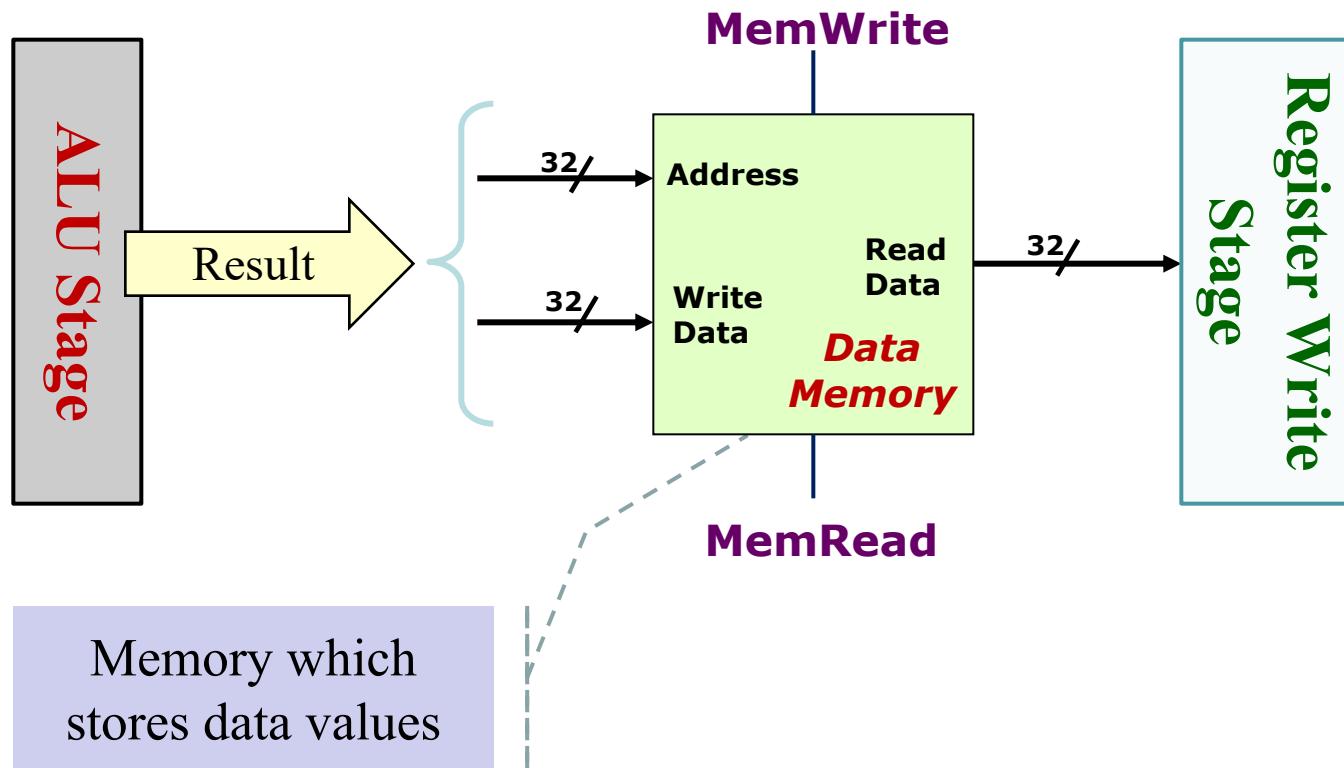
5.4 Memory Stage: Requirements

- **Instruction Memory Access Stage:**
 - Only the **load** and **store** instructions need to perform operation in this stage:
 - Use **memory address calculated by ALU Stage**
 - **Read from or write to data memory**
 - All other instructions remain idle
 - **Result from ALU Stage will pass through to be used in Register Write stage (see section 5.5) if applicable**
- **Input from previous stage (ALU):**
 - Computation result to be used as memory address (if applicable)
- **Output to the next stage (Register Write):**
 - Result to be stored (if applicable)

1. Fetch
2. Decode
3. ALU
- 4. Memory**
5. RegWrite



5.4 Memory Stage: Block Diagram



5.4 Element: Data Memory

- Storage element for the data of a program

- **Inputs:**

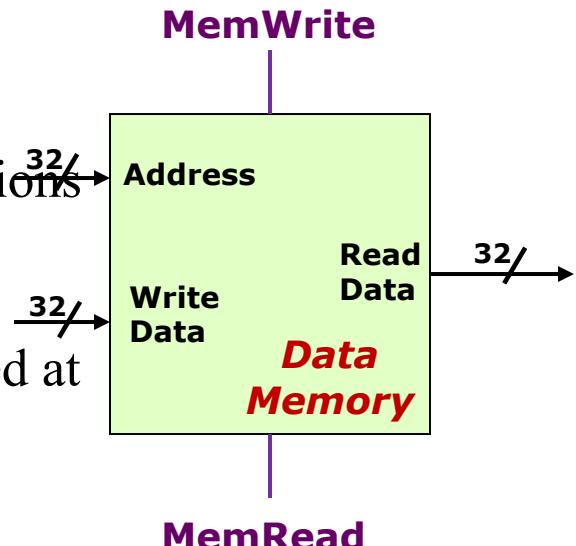
- Memory Address
 - Data to be written (Write Data) for store instructions

- **Control:**

- Read and Write controls; only one can be asserted at any point of time

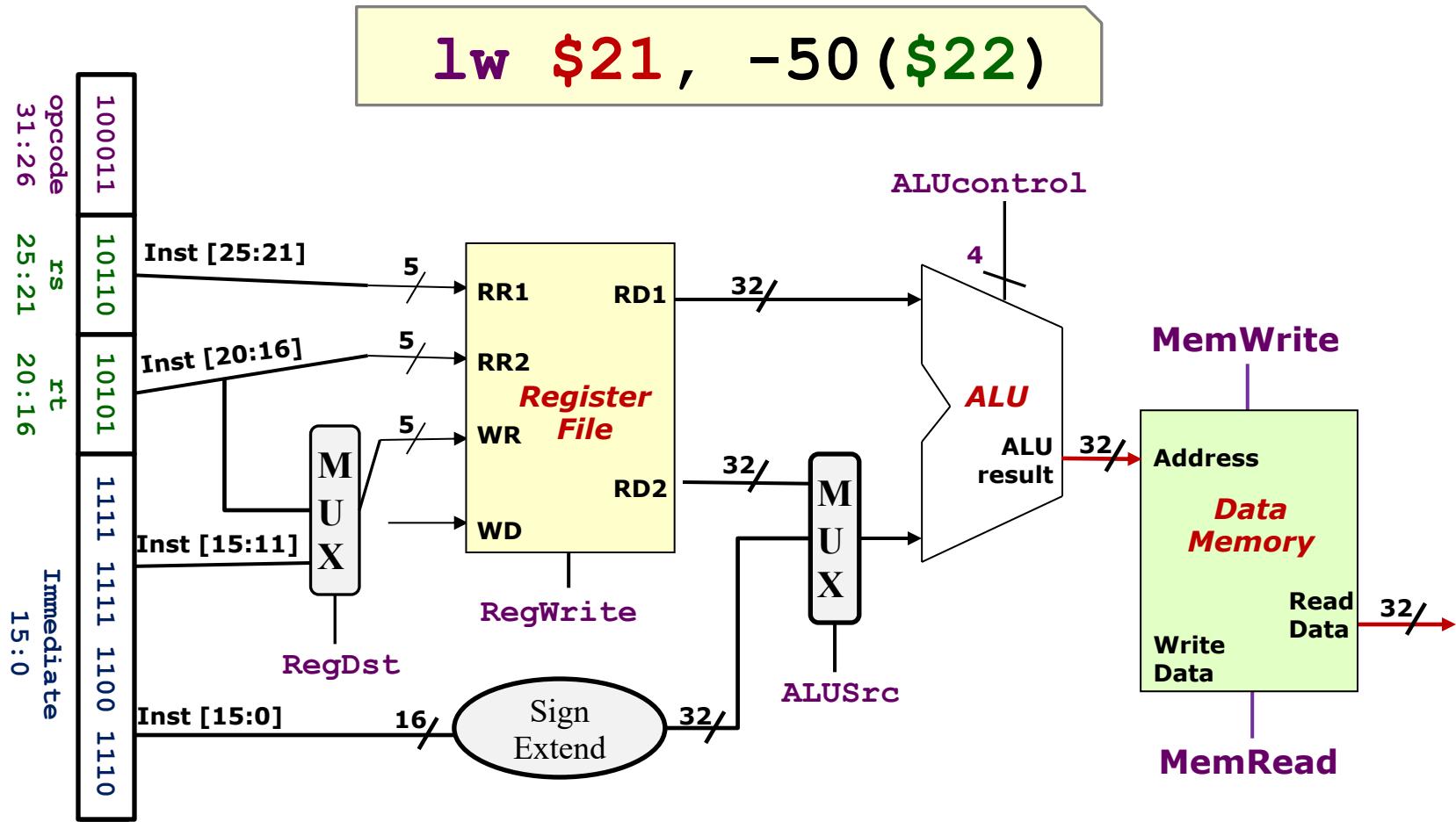
- **Output:**

- Data read from memory (Read Data) for load instructions



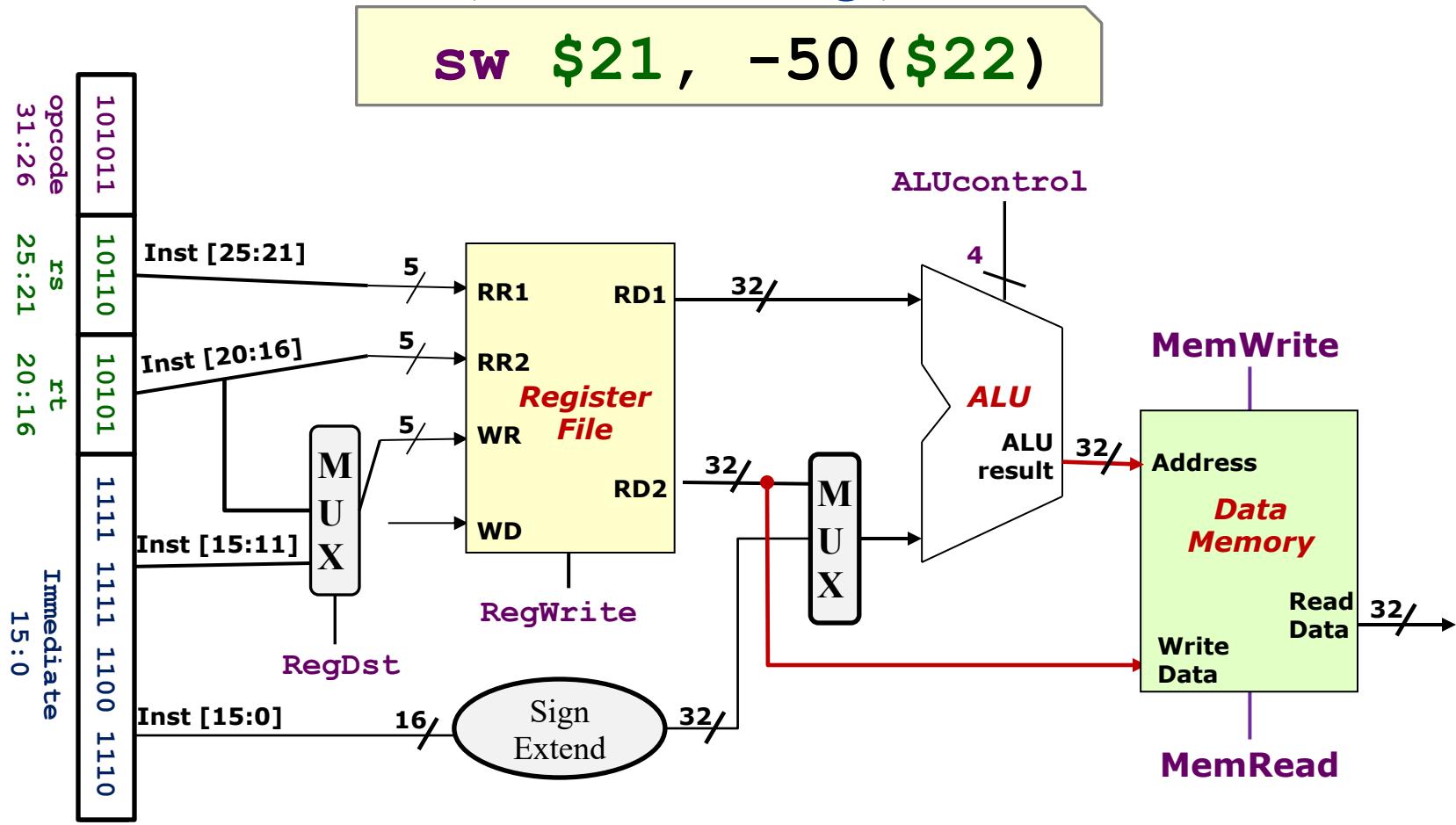
5.4 Memory Stage: Load Instruction

- Only relevant parts of Decode and ALU Stages are shown



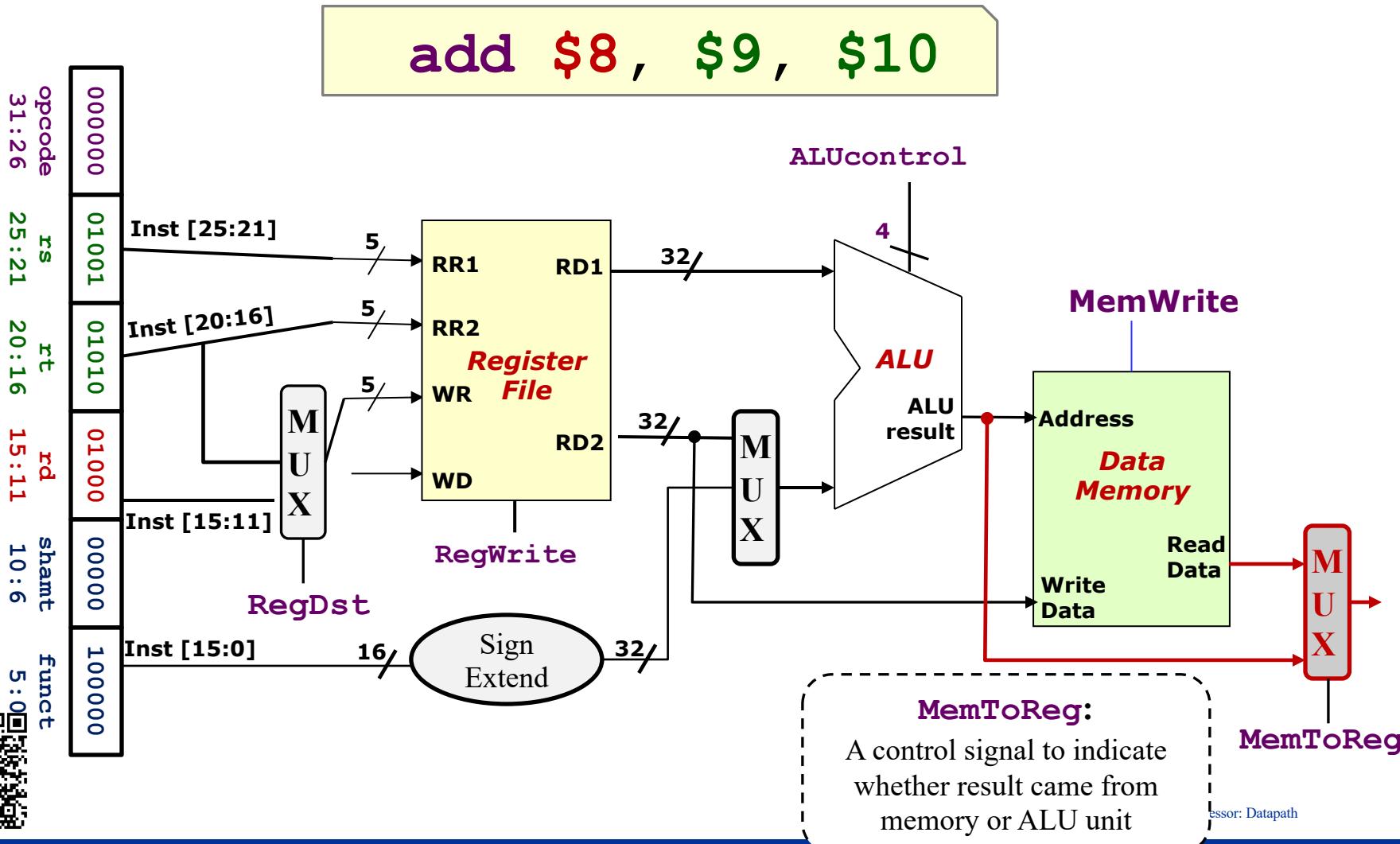
5.4 Memory Stage: Store Instruction

- Need *Read Data 2* (from Decode stage) as the *Write Data*



5.4 Memory Stage: Non-Memory Inst.

- Add a multiplexer to choose the result to be stored



5.5 Register Write Stage: Requirements

■ Instruction Register Write Stage:

1. Fetch
2. Decode
3. ALU
4. Memory
5. **RegWrite**

- Most instructions write the result of some computation into a register

- Examples: arithmetic, logical, shifts, loads, set-less-than
 - Need destination register number and computation result

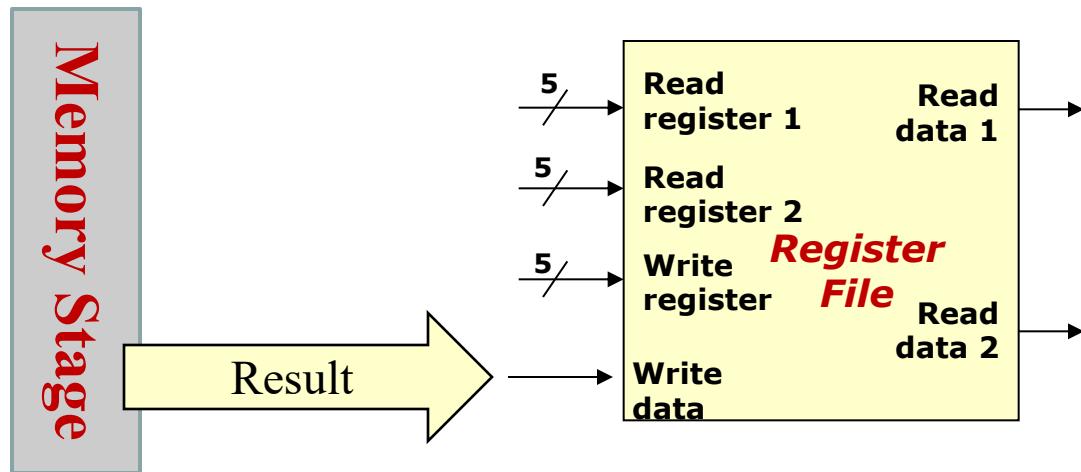
- Exceptions are stores, branches, jumps:
 - There are no results to be written
 - These instructions remain idle in this stage

■ Input from previous stage (Memory):

- Computation result either from memory or ALU



5.5 Register Write Stage: Block Diagram

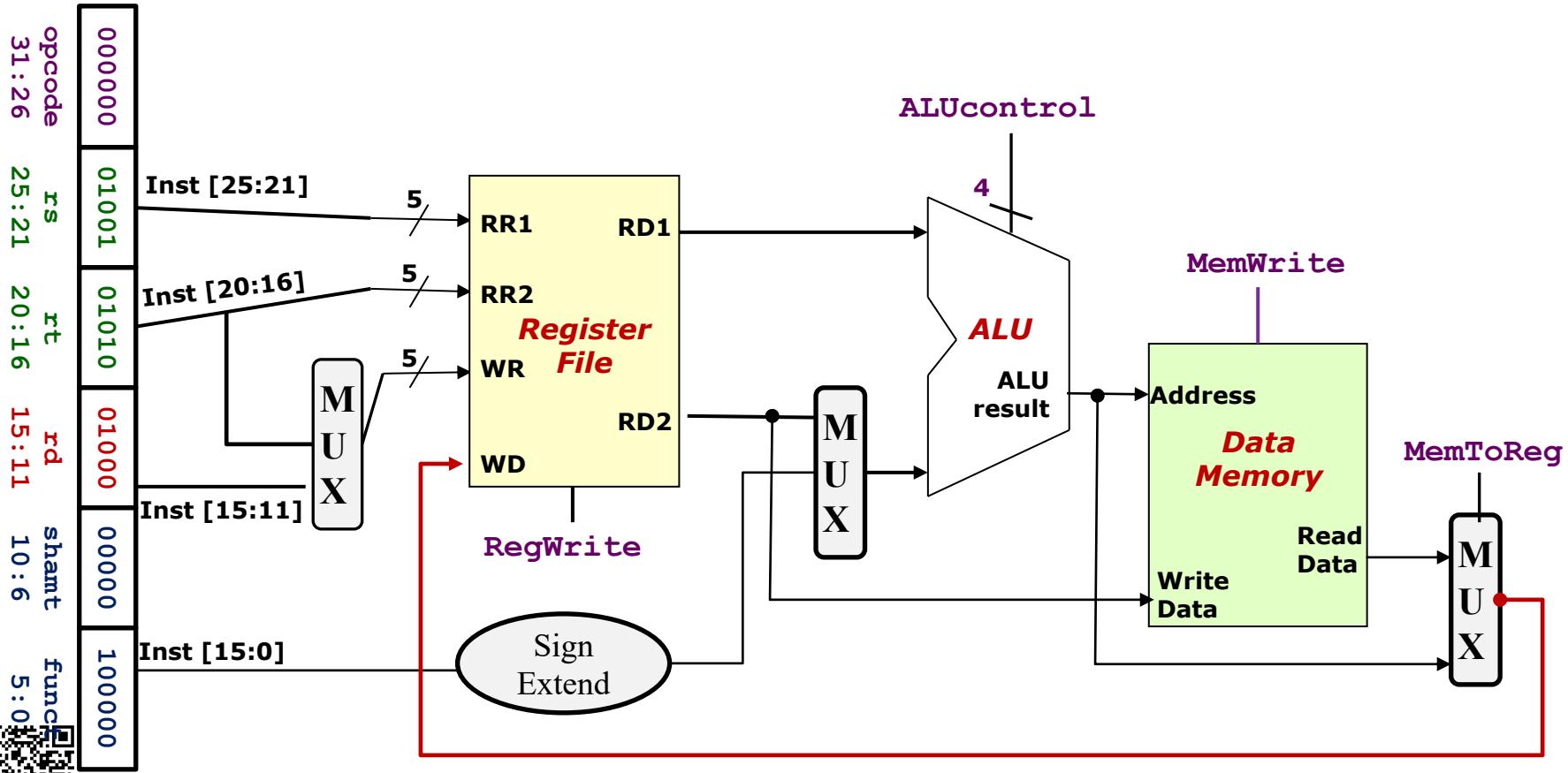


- **Result Write stage has no additional element:**
 - Basically just route the correct result into register file
 - The *Write Register* number is generated way back in the **Decode Stage**



5.5 Register Write Stage: Routing

add \$8, \$9, \$10

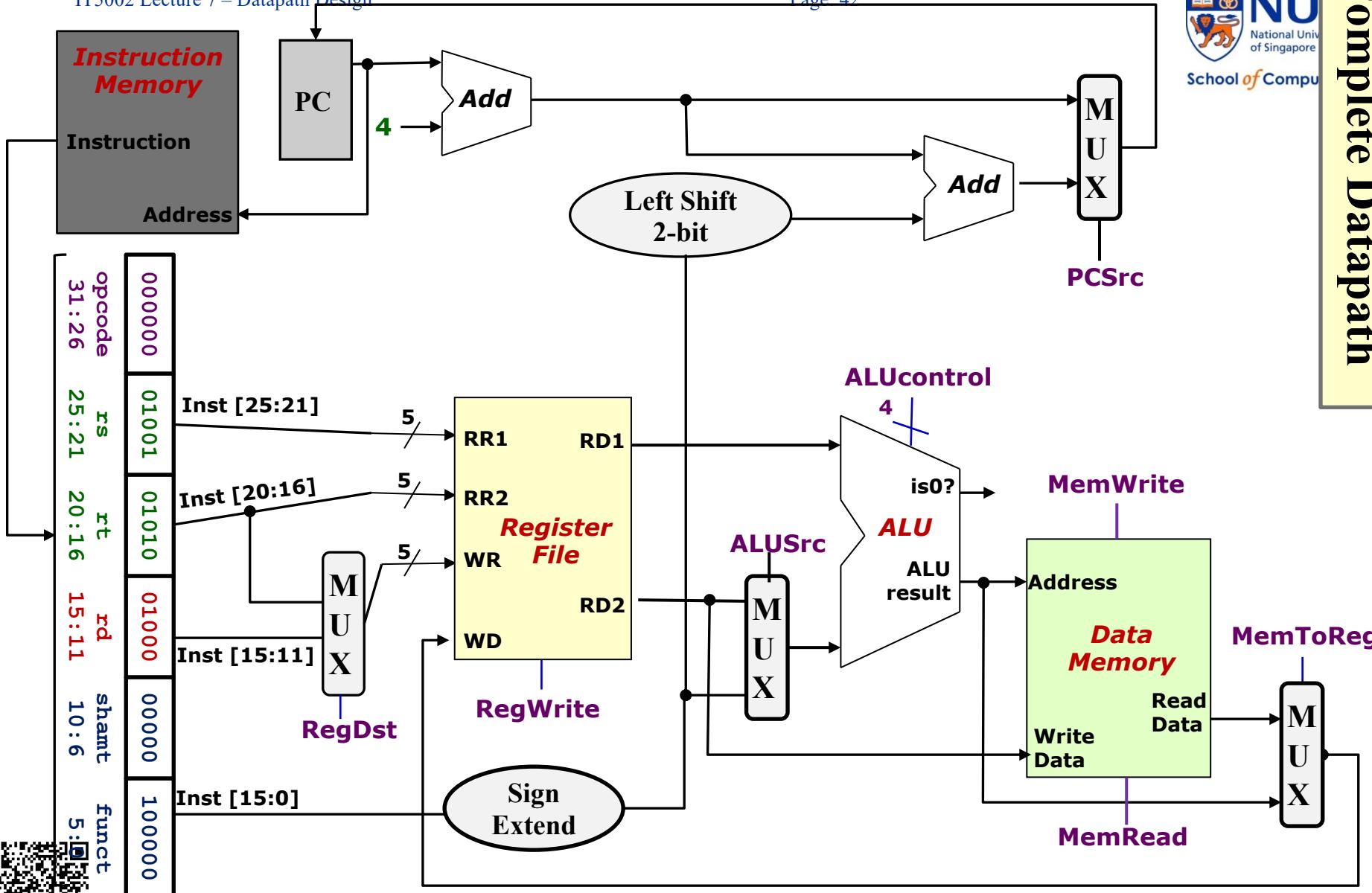


6. The Complete Datapath!

- We have just finished “designing” the datapath for a subset of MIPS instructions:
 - Shifting and Jump are not supported
- Check your understanding:
 - Take the complete datapath and play the role of controller:
 - See how supported instructions are executed
 - Figure out the correct control signals for the datapath elements
- Coming up next: Control



Complete Datapath

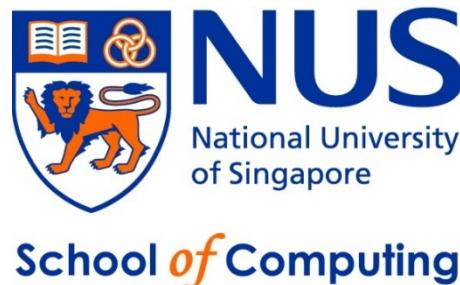


IT5002

Computer Systems and Applications

Datapath Design

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- Please ask questions at
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

Lecture #7: Datapath Design

1. Building a Processor: Datapath & Control
2. MIPS Processor: Implementation
3. Instruction Execution Cycle (Recap)
4. MIPS Instruction Execution
5. Let's Build a MIPS Processor
 - 5.1 Fetch Stage
 - 5.2 Decode Stage
 - 5.3 ALU Stage
 - 5.4 Memory Stage
 - 5.5 Register Write Stage
6. The Complete Datapath!



1. Building a Processor: Datapath & Control

■ Two major components for a processor

Datapath

- Collection of components that process data
- Performs the arithmetic, logical and memory operations

Control

- Tells the datapath, memory and I/O devices what to do according to program instructions

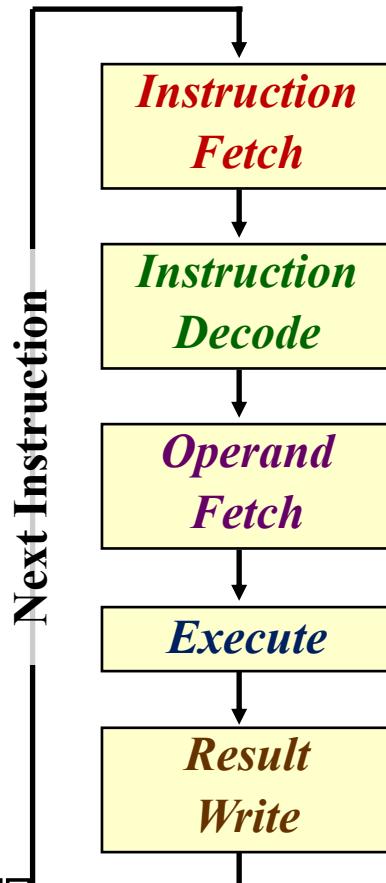


2. MIPS Processor: Implementation

- Simplest possible implementation of a subset of the core MIPS ISA:
 - **Arithmetic and Logical operations**
 - `add`, `sub`, `and`, `or`, `addi`, `andi`, `ori`, `slt`
 - **Data transfer instructions**
 - `lw`, `sw`
 - **Branches**
 - `beq`, `bne`
- Shift instructions (`sll`, `srl`) and J-type instructions (`j`) will not be discussed:
 - Left as exercises ☺



3. Instruction Execution Cycle (Basic)



1.

Fetch:

- Get instruction from memory
- Address is in Program Counter (PC) Register

2.

Decode:

- Find out the operation required

3.

Operand Fetch:

- Get operand(s) needed for operation

4.

Execute:

- Perform the required operation

5.

Result Write (Store):

- Store the result of the operation

4. MIPS Instruction Execution (1/2)

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stages not shown:
 - The standard steps are performed

	add \$3, \$1, \$2	lw \$3, 20(\$1)	beq \$1, \$2, ofst
Fetch	<i>standard</i>	<i>standard</i>	<i>standard</i>
Decode			
Operand Fetch	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Use 20 as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i>
Execute	$Result = opr1 + opr2$	<ul style="list-style-type: none"> ○ $MemAddr = opr1 + opr2$ ○ Use <i>MemAddr</i> to read from memory 	$Taken = (opr1 == opr2) ?$ $Target = (\text{PC} + 4) + \text{ofst} \times 4$
Result Write	<i>Result stored in \$3</i>	<i>Memory data stored in \$3</i>	$\text{if } (Taken)$ $\text{PC} = \text{Target}$


 opr = operand

MemAddr = Memory Address

■ ofst = offset

4. MIPS Instruction Execution (2/2)

- **Design changes:**
 - Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
 - Split *Execute* into *ALU* (Calculation) and *Memory Access*

	add \$3, \$1, \$2	lw \$3, 20(\$1)	beq \$1, \$2, ofst
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Use 20 as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i>
ALU	$Result = opr1 + opr2$	$MemAddr = opr1 + opr2$	$Taken = (opr1 == opr2) ?$ $Target = (\text{PC}+4) + ofst \times 4$
Memory Access		Use <i>MemAddr</i> to read from memory	
Result Write	<i>Result</i> stored in \$3	<i>Memory</i> data stored in \$3	$\text{if } (Taken)$ $\text{PC} = \text{Target}$



5. Let's Build a MIPS Processor

- **What we are going to do:**
 - Look at each stage closely, figure out the requirements and processes
 - Sketch a high level block diagram, then zoom in for each elements
 - With the simple starting design, check whether different type of instructions can be handled:
 - **Add modifications when needed**
- ➔ **Study the design from the viewpoint of a designer, instead of a "tourist" ☺**



5.1 Fetch Stage: Requirements

- Instruction Fetch Stage:

1. Use the **Program Counter (PC)** to fetch the instruction from **memory**
 - PC is implemented as a special register in the processor

2. **Increment** the PC by 4 to get the address of the next instruction:

- How do we know the next instruction is at $\text{PC}+4$?
- Note the exception when branch/jump instruction is executed

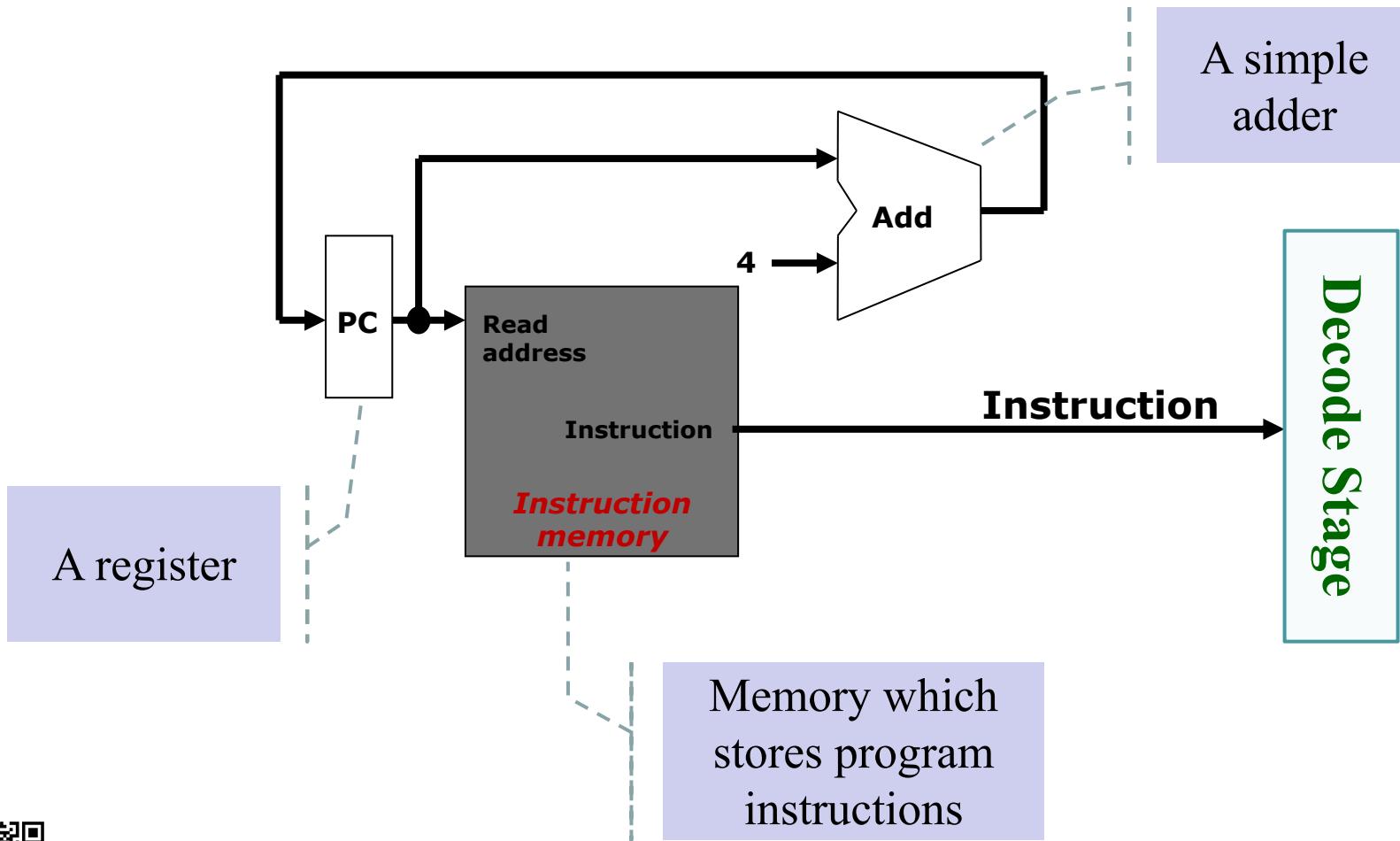
- Output to the next stage (**Decode**):

- The instruction to be executed

1. **Fetch**
2. Decode
3. ALU
4. Memory
5. RegWrite

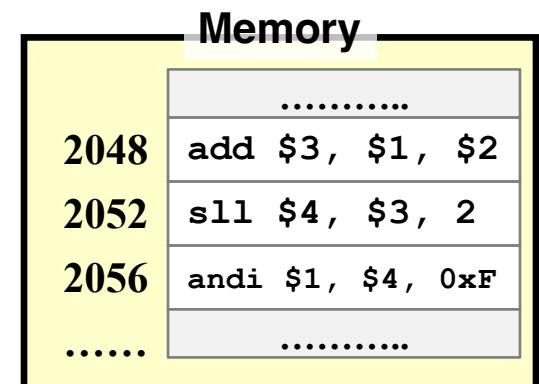
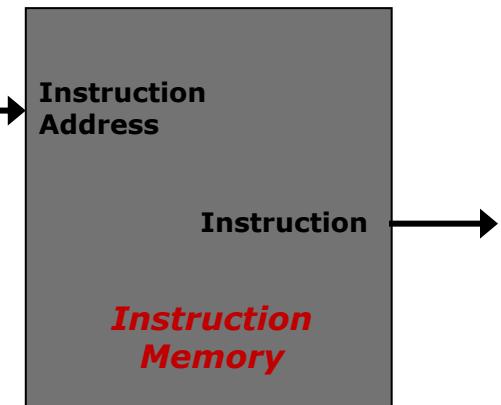


5.1 Fetch Stage: Block Diagram



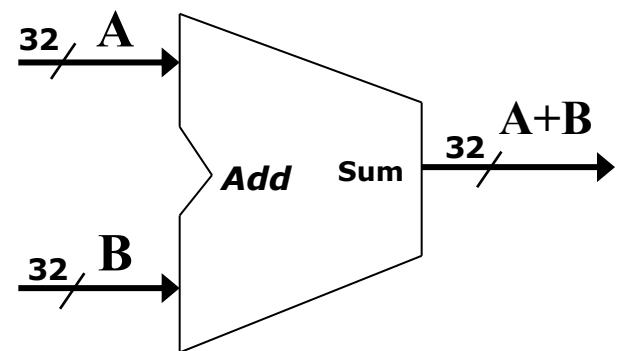
5.1 Element: Instruction Memory

- Storage element for the instructions
 - It is a **sequential circuit** (to be covered later) →
 - Has an internal state that stores information
 - Clock signal is assumed and not shown
- Supply instruction given the address
 - Given instruction address M as input, the memory outputs the content at address M
 - Conceptual diagram of the memory layout is given on the right →



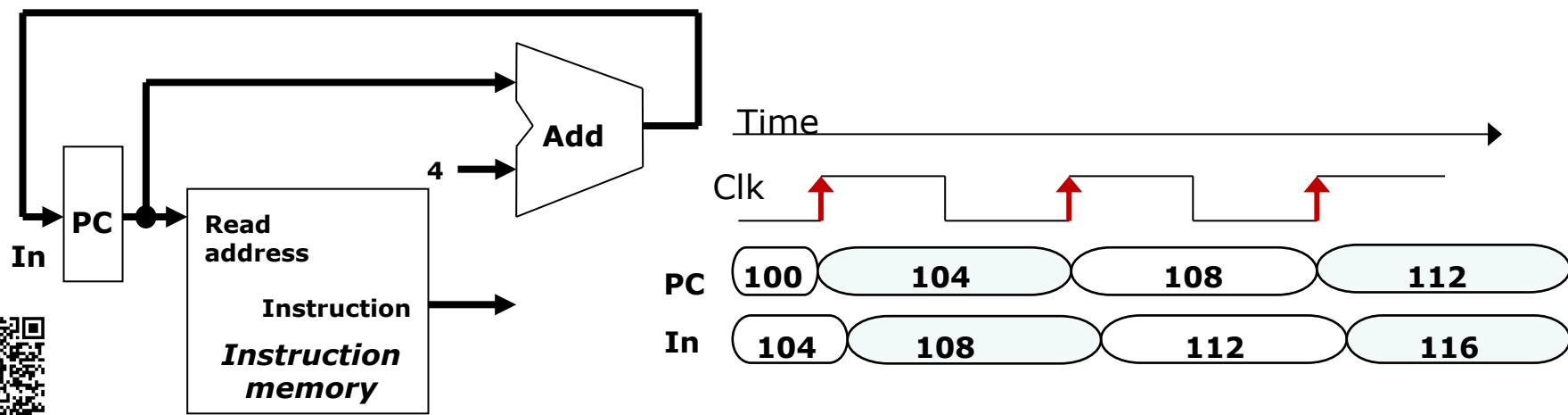
5.1 Element: Adder

- Combinational logic to implement the addition of two numbers
- **Inputs:**
 - Two 32-bit numbers A, B
- **Output:**
 - Sum of the input numbers, $A + B$



5.1 The Idea of Clocking

- It seems that we are reading and updating the PC at the same time:
 - How can it work properly?
- **Magic of clock:**
 - PC is read during the first half of the clock period and it is updated with $\text{PC}+4$ at the **next rising clock edge**



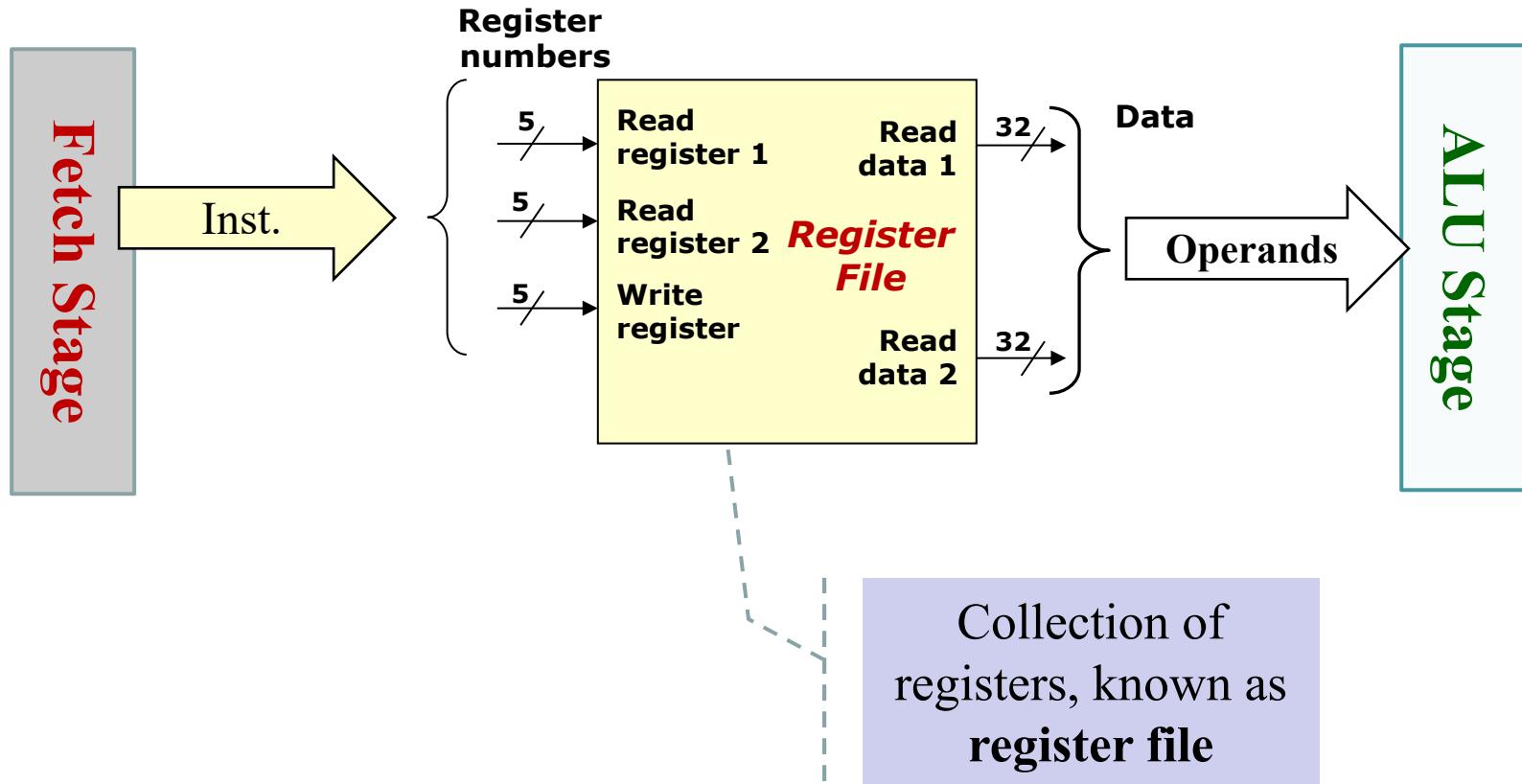
5.2 Decode Stage: Requirements

- **Instruction Decode Stage:**
 - Gather data from the instruction fields:
 1. Read the **opcode** to determine instruction type and field lengths
 2. Read data from all necessary registers
 - **Can be two (e.g. add), one (e.g. addi) or zero (e.g. j)**
- **Input from previous stage (Fetch):**
 - Instruction to be executed
- **Output to the next stage (ALU):**
 - Operation and the necessary operands

1. Fetch
2. **Decode**
3. ALU
4. Memory
5. RegWrite

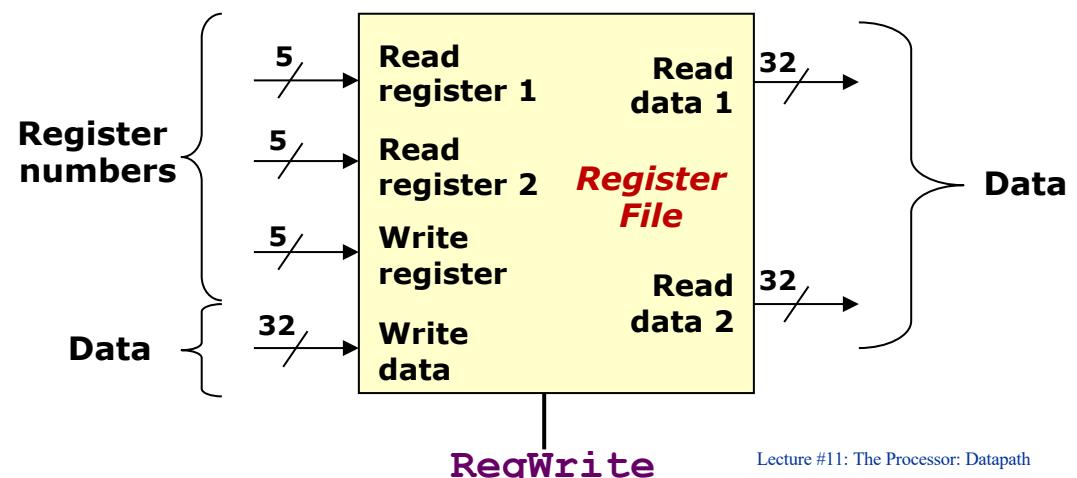


5.2 Decode Stage: Block Diagram

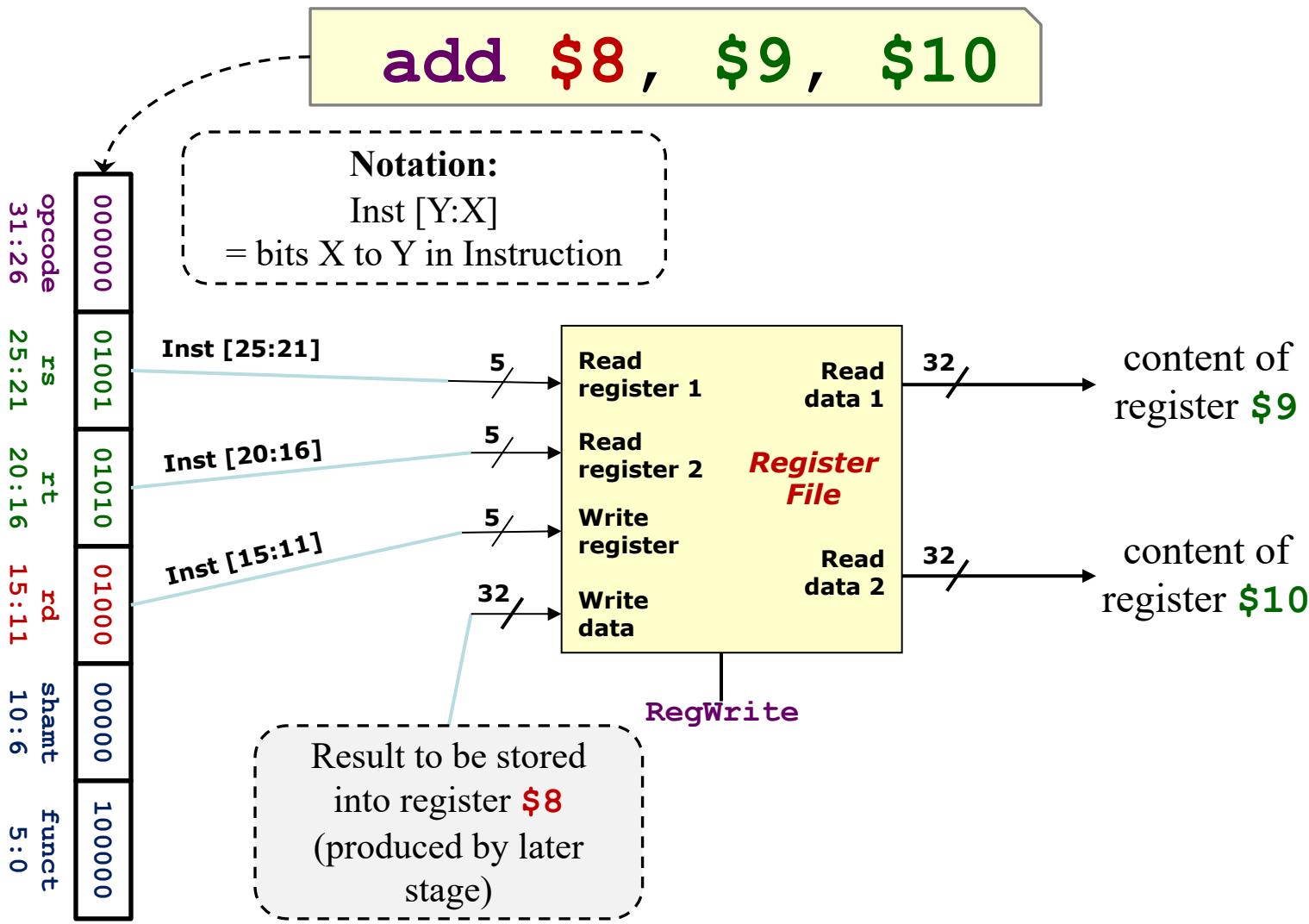


5.2 Element: Register File

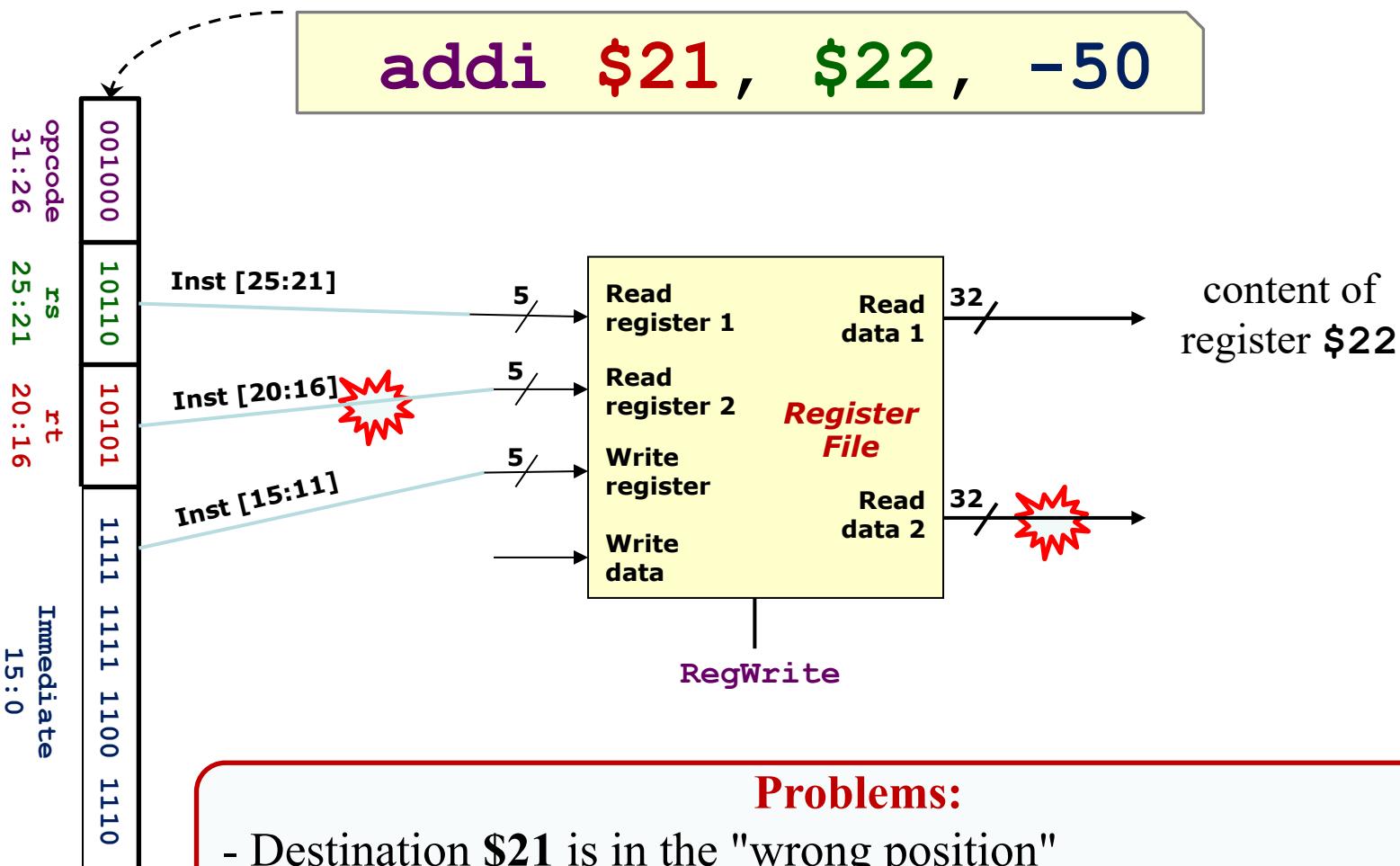
- A collection of 32 registers:
 - Each 32-bit wide; can be read/written by specifying register number
 - Read at most two registers per instruction
 - Write at most one register per instruction
- **RegWrite** is a control signal to indicate:
 - Writing of register
 - 1(True) = Write, 0 (False) = No Write



5.2 Decode Stage: R-Format Instruction



5.2 Decode Stage: I-Format Instruction

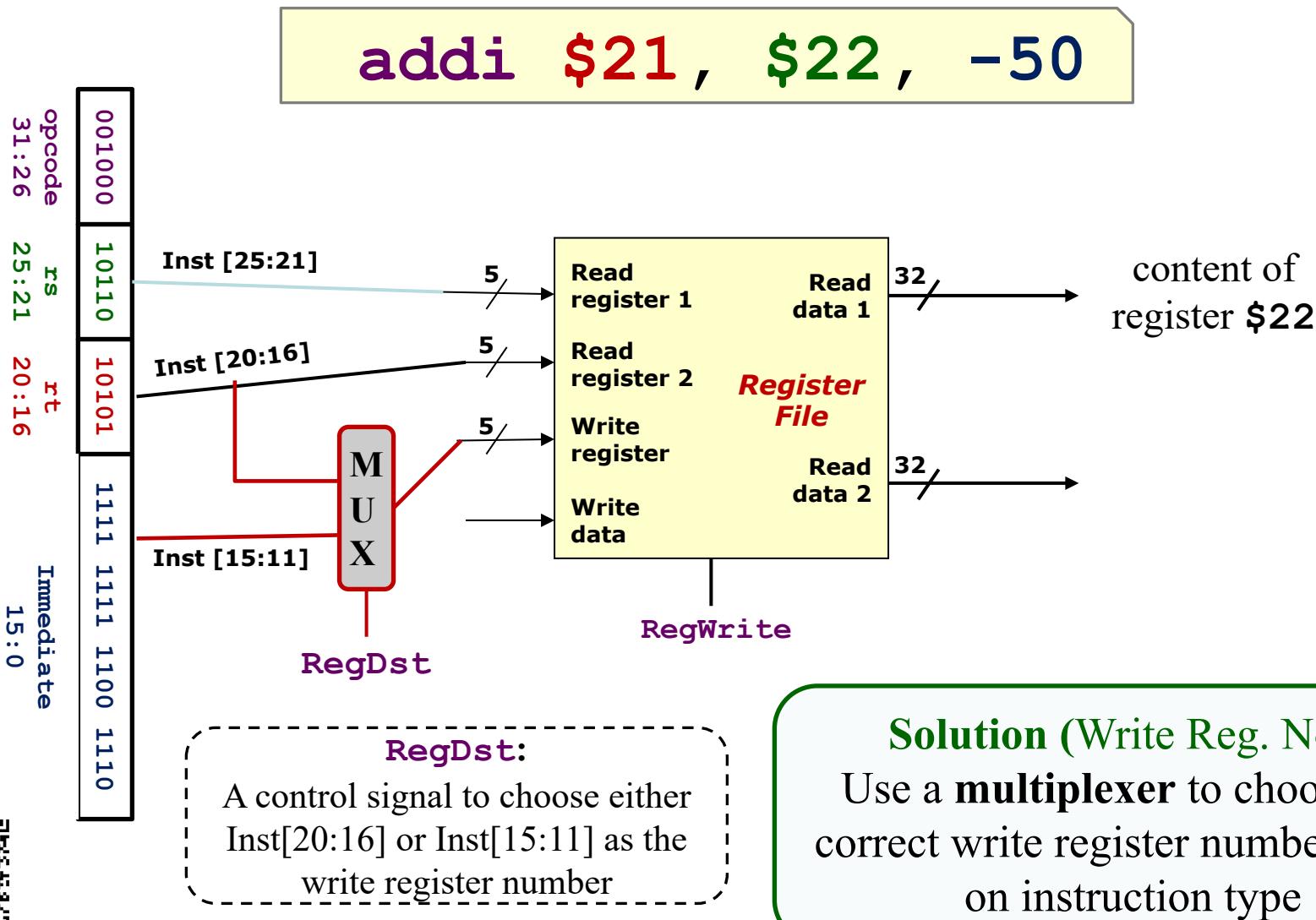


Problems:

- Destination **\$21** is in the "wrong position"
- **Read Data 2** is an immediate value, not from register



5.2 Decode Stage: Choice in Destination



5.2 Multiplexer

Function:

- Selects one input from multiple input lines

Inputs:

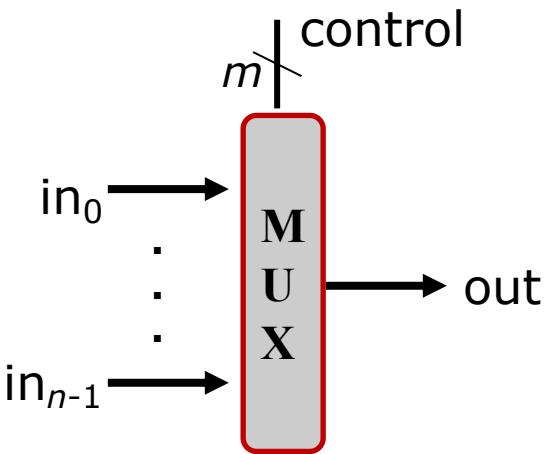
- n lines of same width

Control:

- m bits where $n = 2^m$

Output:

- Select i^{th} input line if control = i

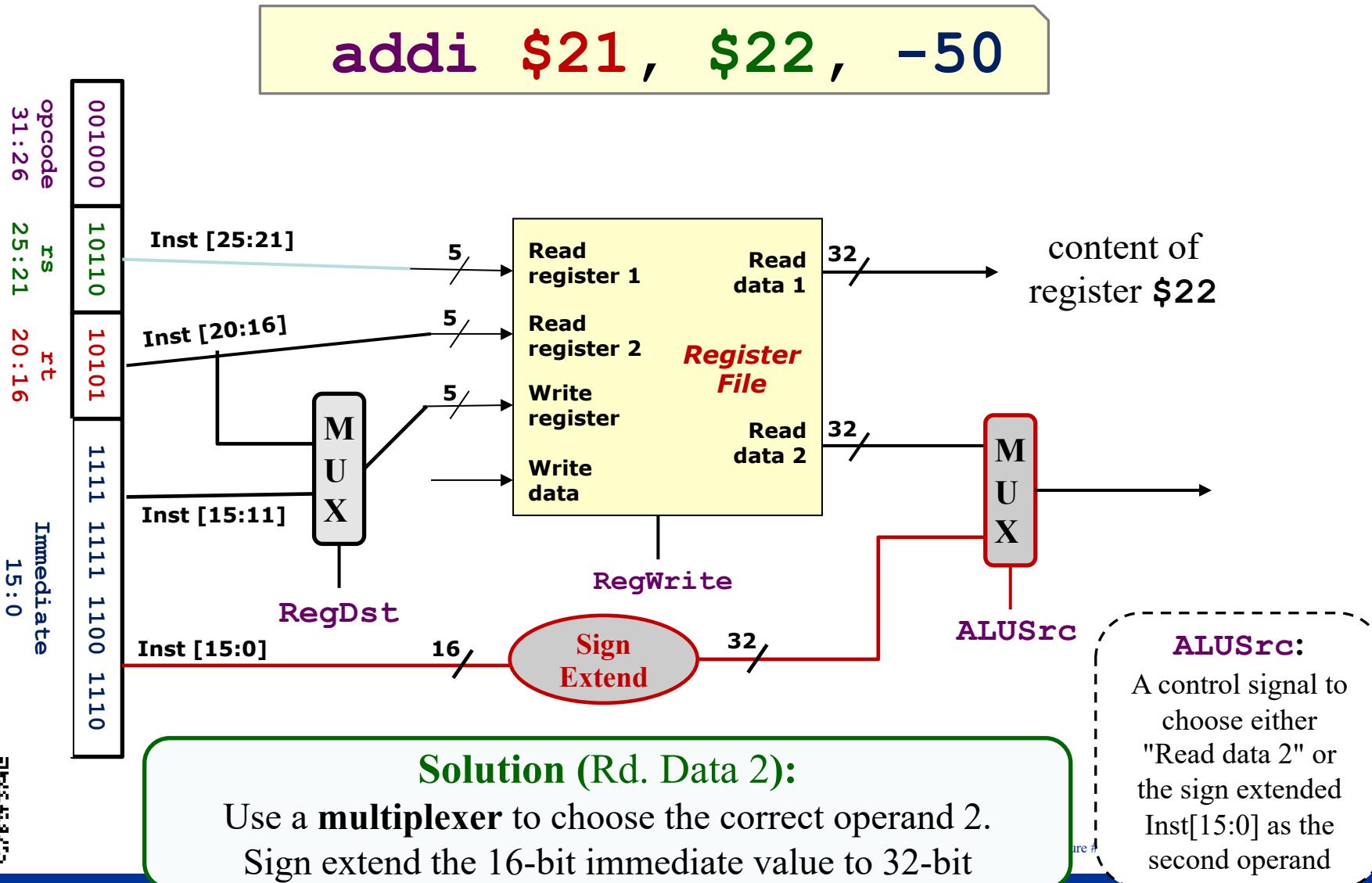


Control=0 → select in_0 to out

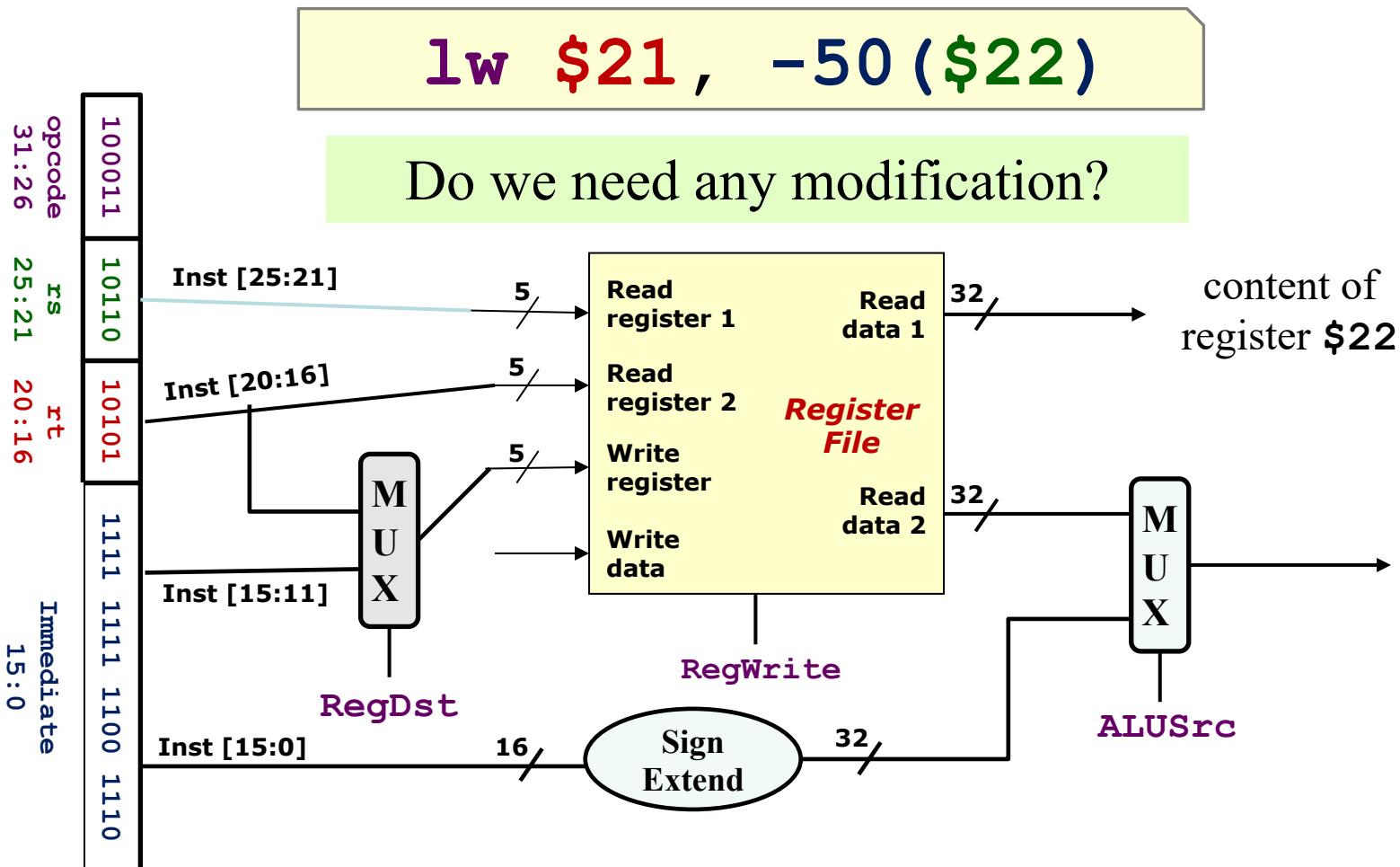
Control=3 → select in_3 to out



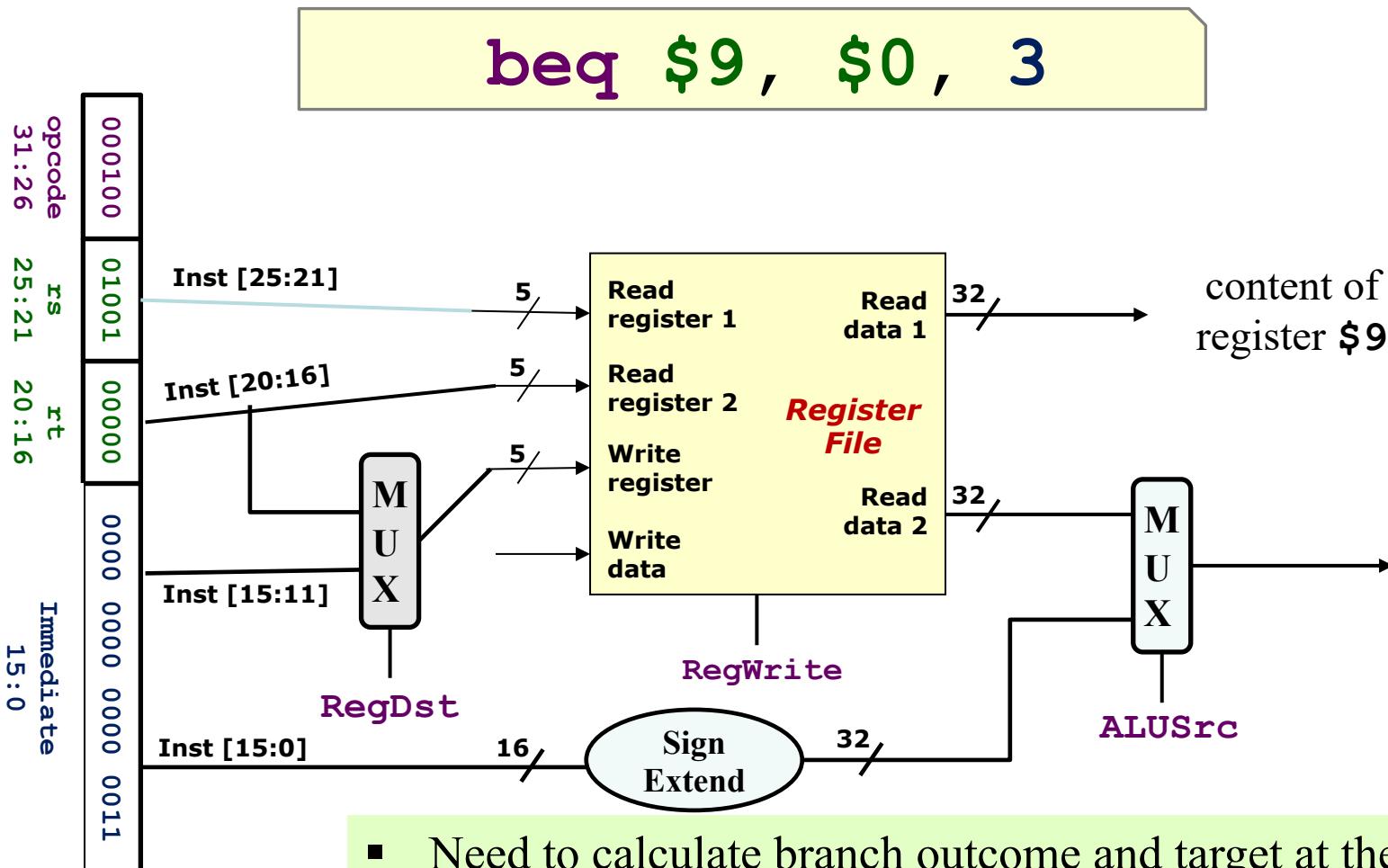
5.2 Decode Stage: Choice in Data 2



5.2 Decode Stage: Load Word Instruction



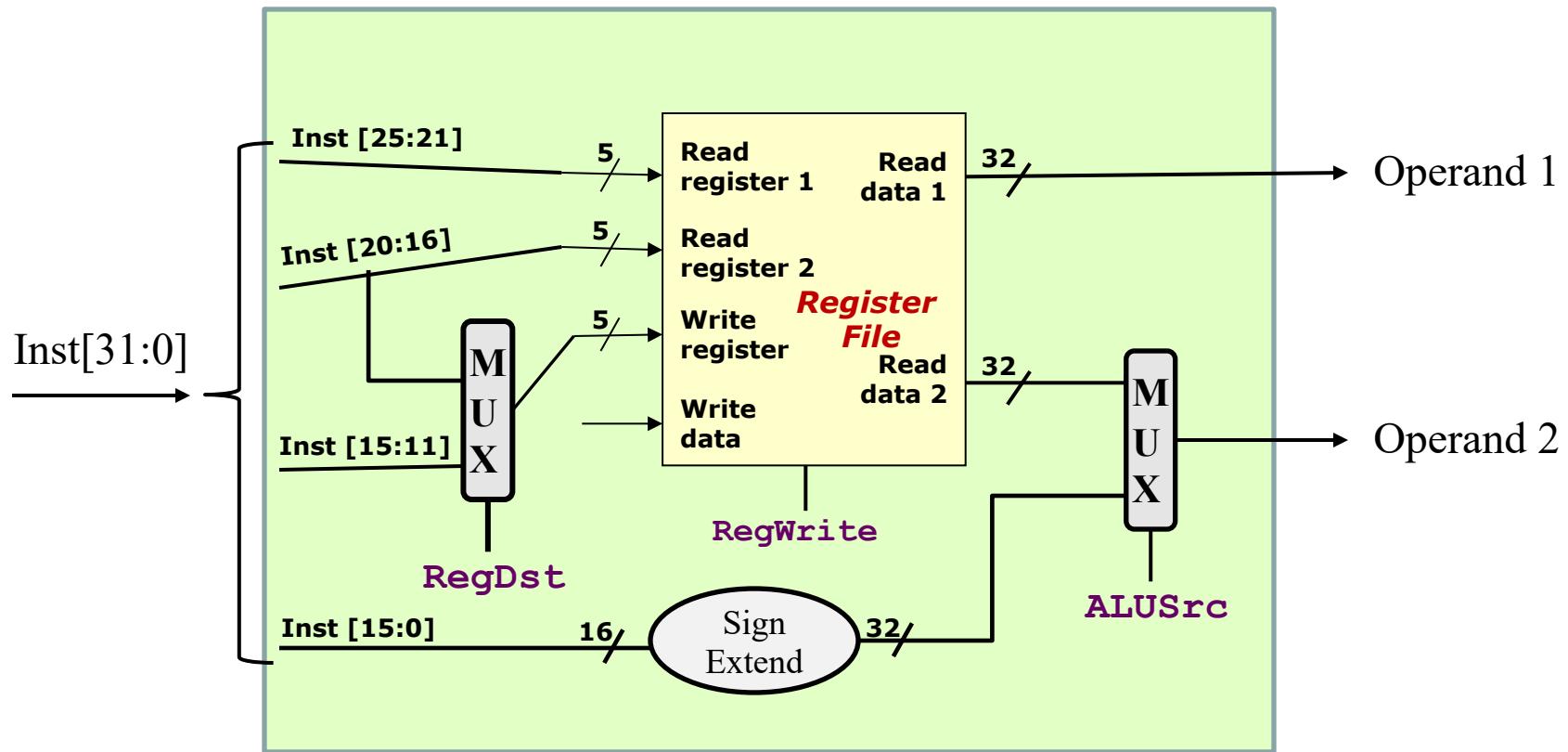
5.2 Decode Stage: Branch Instruction



- Need to calculate branch outcome and target at the same time!
- We will tackle this problem at the ALU stage



5.2 Decode Stage: Summary



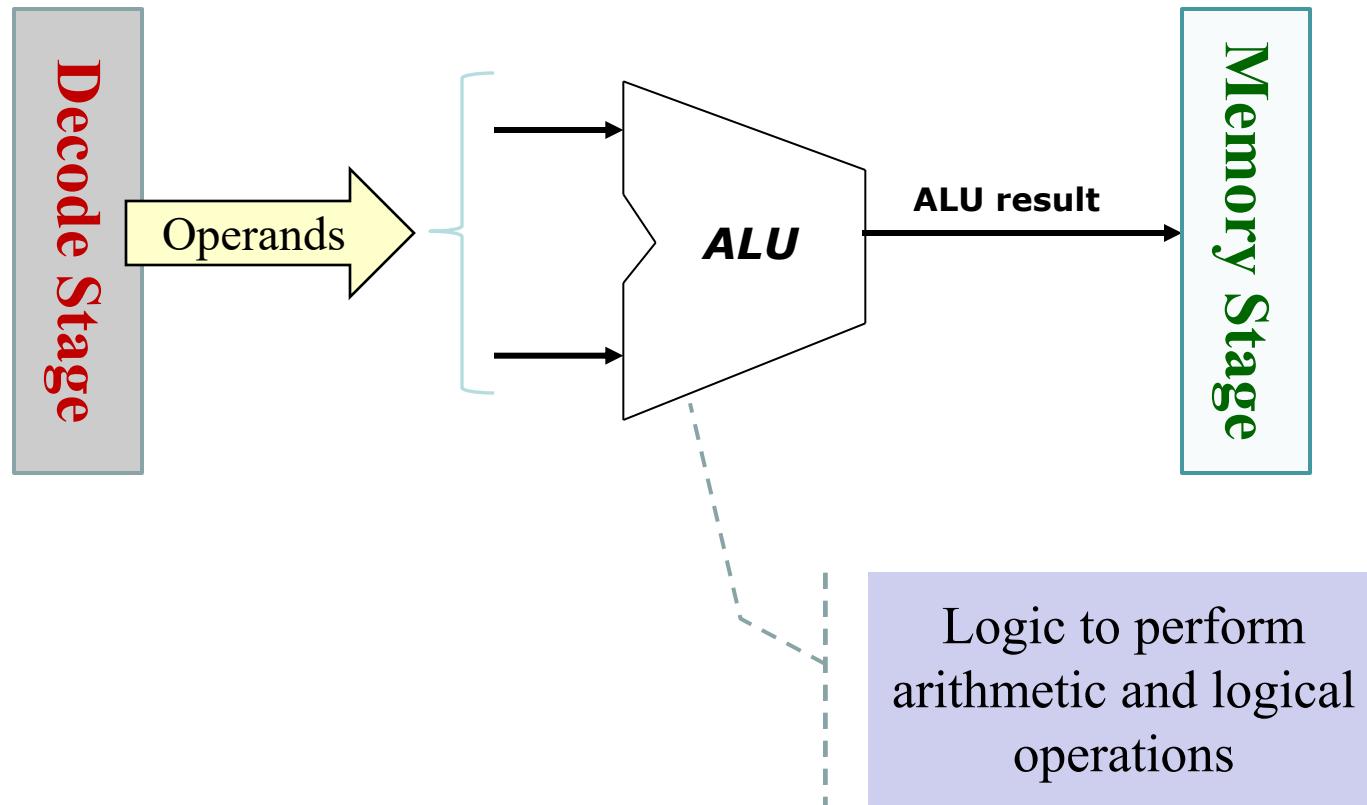
5.3 ALU Stage: Requirements

- **Instruction ALU Stage:**
 - ALU = Arithmetic-Logic Unit
 - Also called the **Execution stage**
 - Perform the real work for most instructions here
 - Arithmetic (e.g. **add**, **sub**), Shifting (e.g. **sll**), Logical (e.g. **and**, **or**)
 - Memory operation (e.g. **lw**, **sw**): Address calculation
 - Branch operation (e.g. **bne**, **beq**): Perform register comparison and target address calculation
- **Input from previous stage (Decode):**
 - Operation and Operand(s)
- **Output to the next stage (Memory):**
 - Calculation result

1. Fetch
2. Decode
- 3. ALU**
4. Memory
5. RegWrite



5.3 ALU Stage: Block Diagram



5.3 Element: Arithmetic Logic Unit

■ ALU (Arithmetic Logic Unit)

- Combinational logic to implement arithmetic and logical operations

■ Inputs:

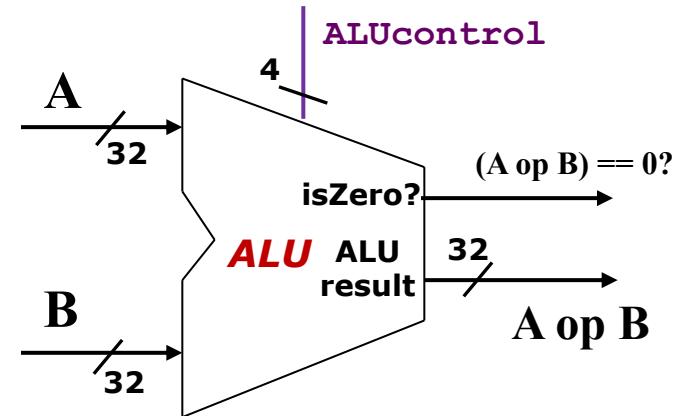
- Two 32-bit numbers

■ Control:

- 4-bit to decide the particular operation

■ Outputs:

- Result of arithmetic/logical operation
- A 1-bit signal to indicate whether result is zero

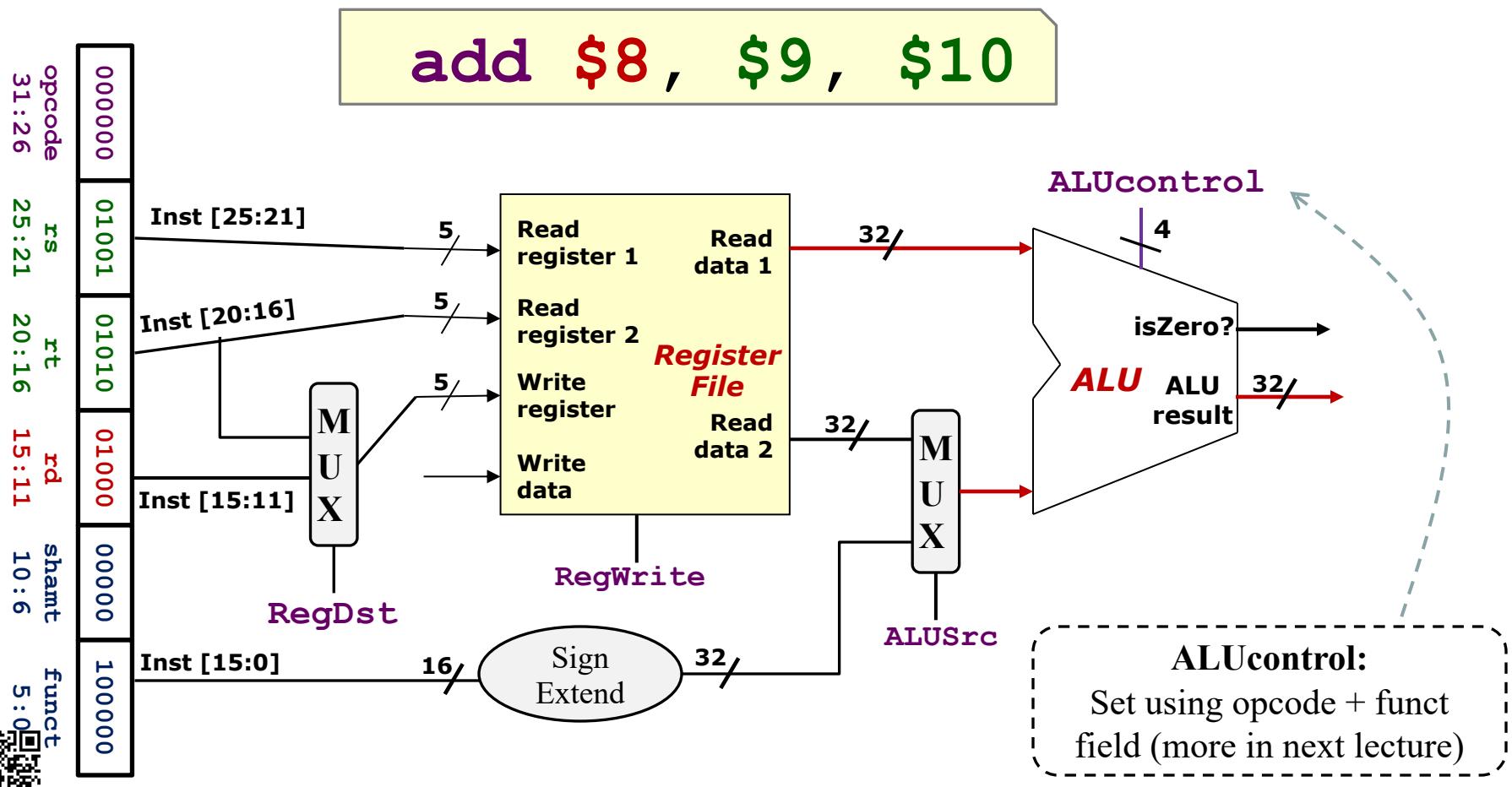


ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR



5.3 ALU Stage: Non-Branch Instructions

- We can handle non-branch instructions easily:



5.3 ALU Stage: Branch Instructions

- Branch instruction is harder as we need to perform two calculations:
- Example: "**beq \$9, \$0, 3**"

1. Branch Outcome:

- Use ALU to compare the register
- The 1-bit "isZero?" signal is enough to handle equal/not equal check (how?)

2. Branch Target Address:

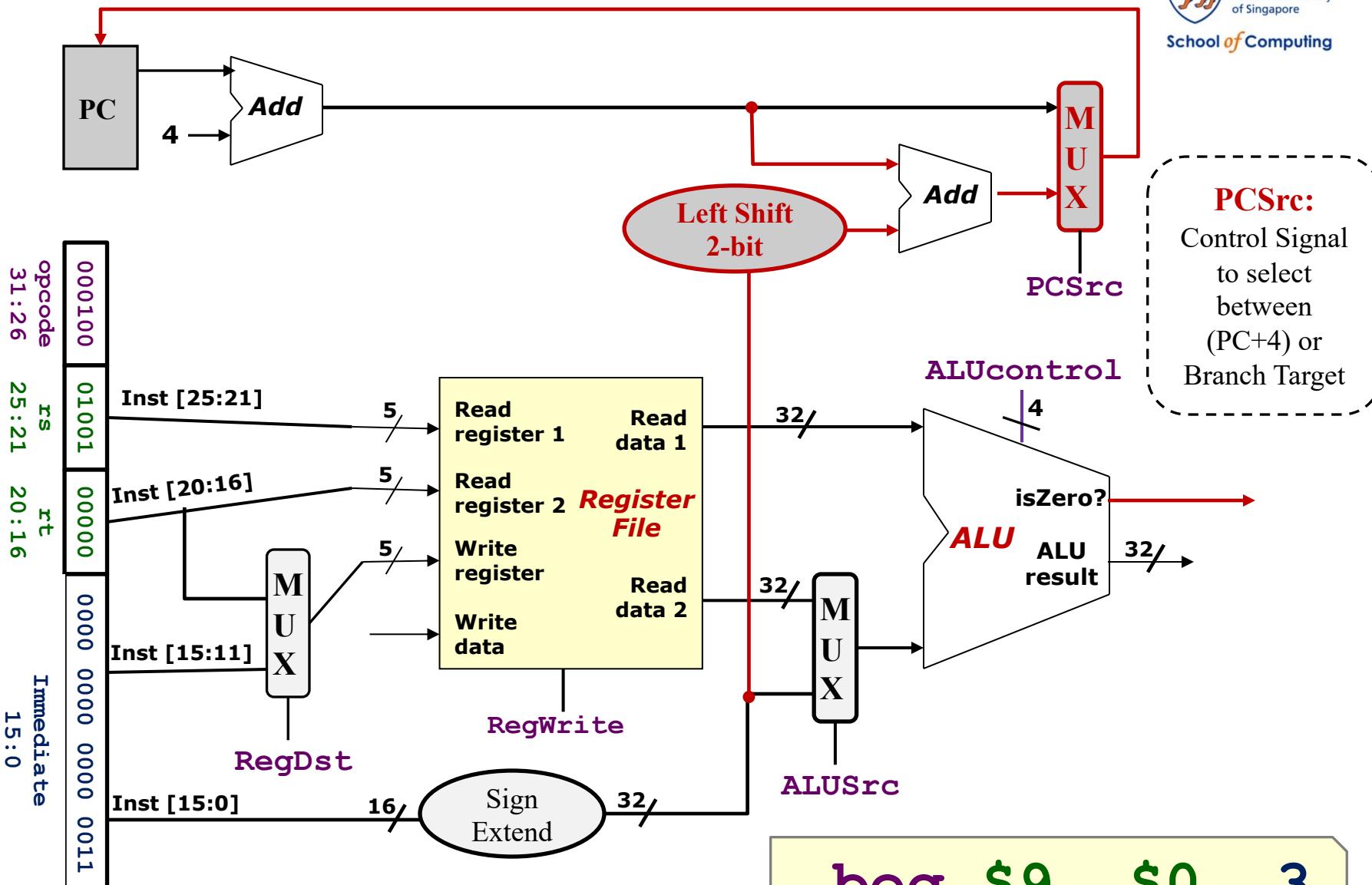
- Introduce additional logic to calculate the address
- Need PC (from Fetch Stage)
- Need Offset (from Decode Stage)



Complete ALU Stage

EE5002 Lecture 7 Datapath Design

Page: 31



`beq $9, $0, 3`

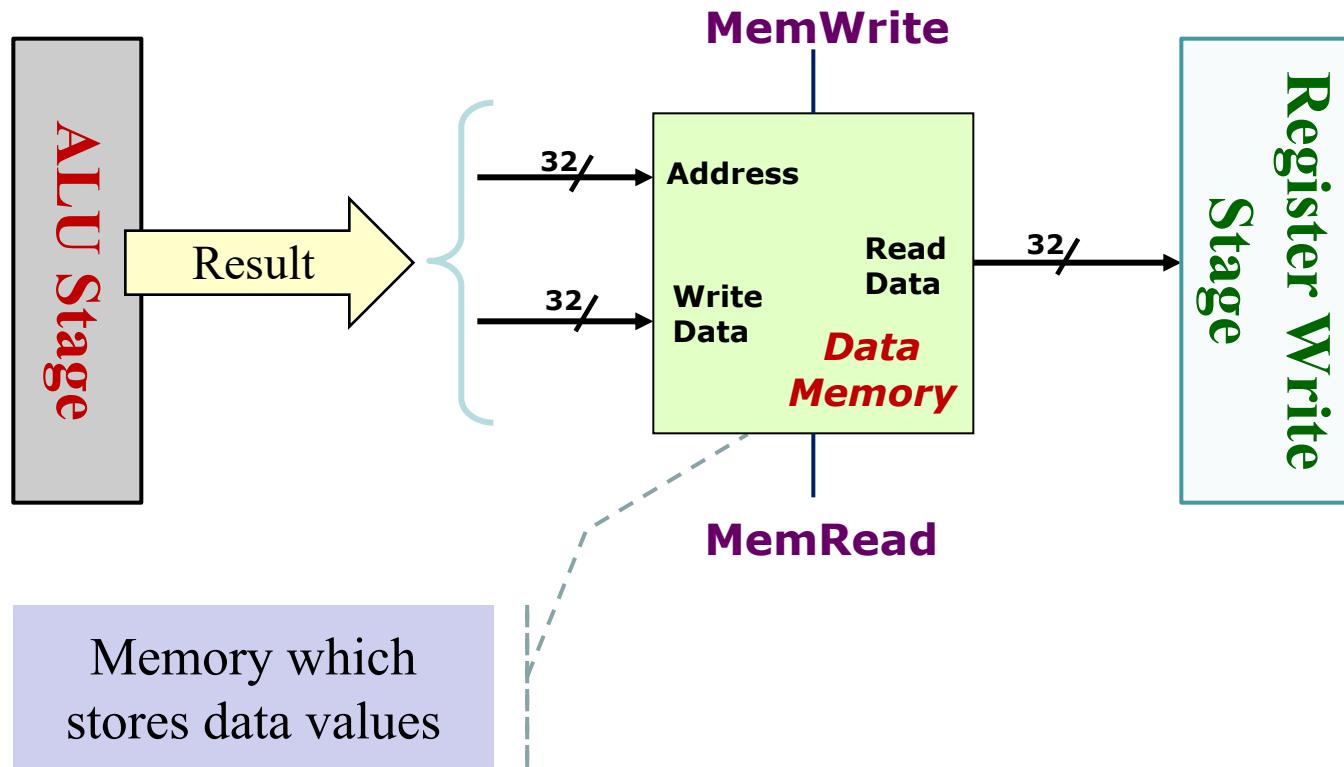
5.4 Memory Stage: Requirements

- **Instruction Memory Access Stage:**
 - Only the **load** and **store** instructions need to perform operation in this stage:
 - Use **memory address calculated by ALU Stage**
 - **Read from or write to data memory**
 - All other instructions remain idle
 - **Result from ALU Stage will pass through to be used in Register Write stage (see section 5.5) if applicable**
- **Input from previous stage (ALU):**
 - Computation result to be used as memory address (if applicable)
- **Output to the next stage (Register Write):**
 - Result to be stored (if applicable)

1. Fetch
2. Decode
3. ALU
- 4. Memory**
5. RegWrite



5.4 Memory Stage: Block Diagram



5.4 Element: Data Memory

- Storage element for the data of a program

- **Inputs:**

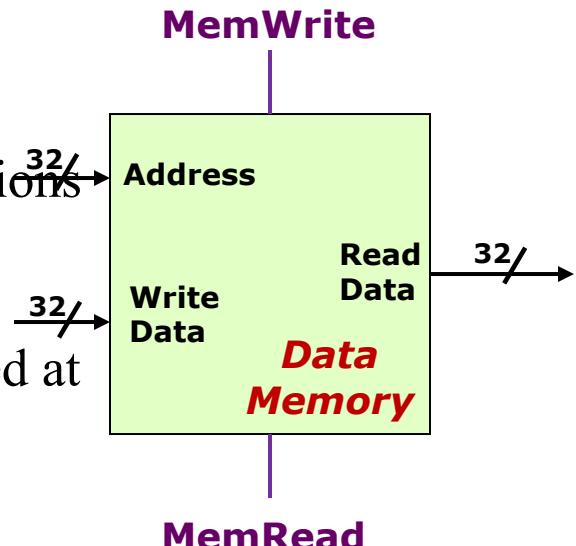
- Memory Address
 - Data to be written (Write Data) for store instructions

- **Control:**

- Read and Write controls; only one can be asserted at any point of time

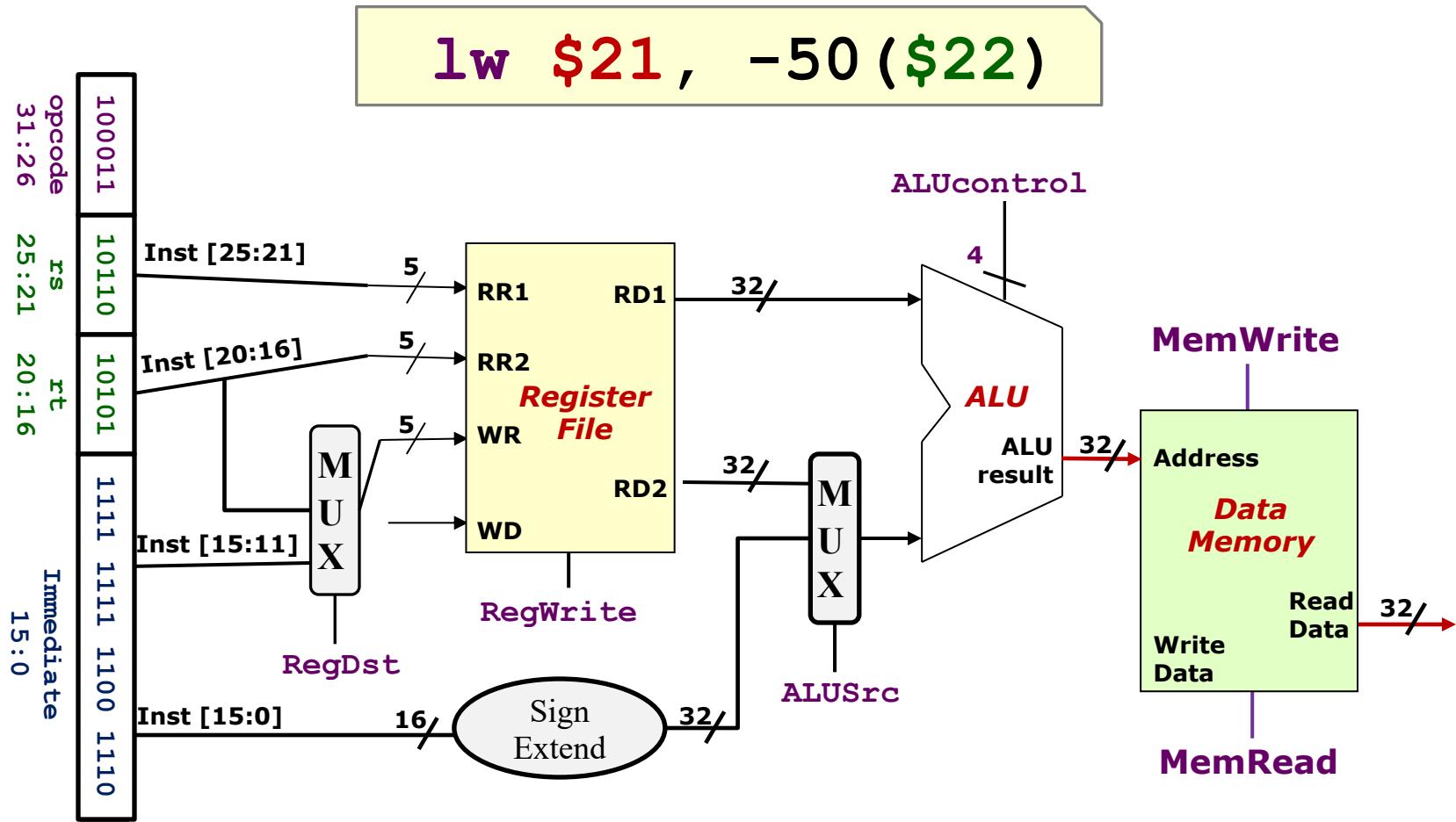
- **Output:**

- Data read from memory (Read Data) for load instructions



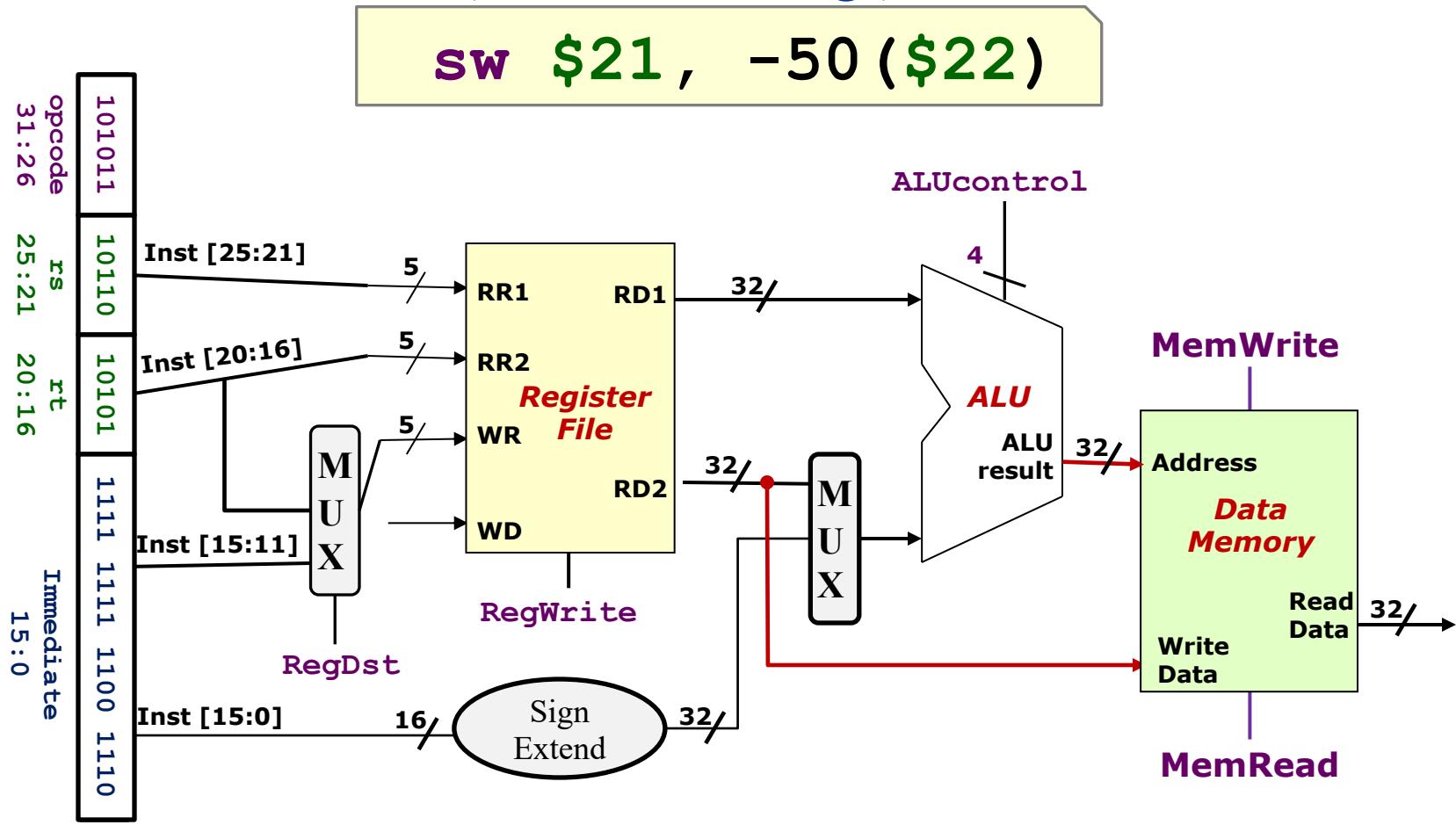
5.4 Memory Stage: Load Instruction

- Only relevant parts of Decode and ALU Stages are shown



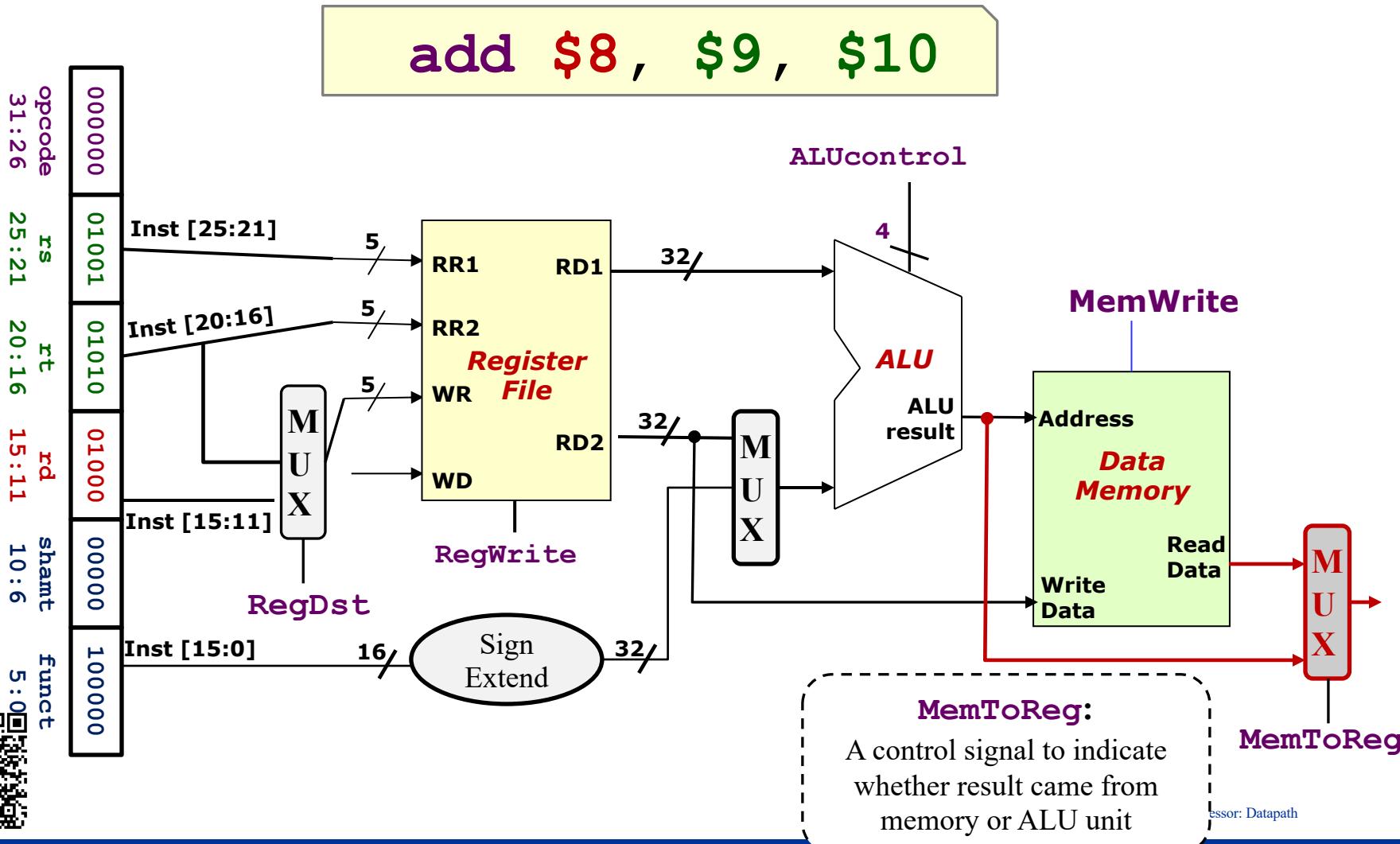
5.4 Memory Stage: Store Instruction

- Need *Read Data 2* (from Decode stage) as the *Write Data*



5.4 Memory Stage: Non-Memory Inst.

- Add a multiplexer to choose the result to be stored



5.5 Register Write Stage: Requirements

■ Instruction Register Write Stage:

1. Fetch
2. Decode
3. ALU
4. Memory
5. **RegWrite**

- Most instructions write the result of some computation into a register

- Examples: arithmetic, logical, shifts, loads, set-less-than
 - Need destination register number and computation result

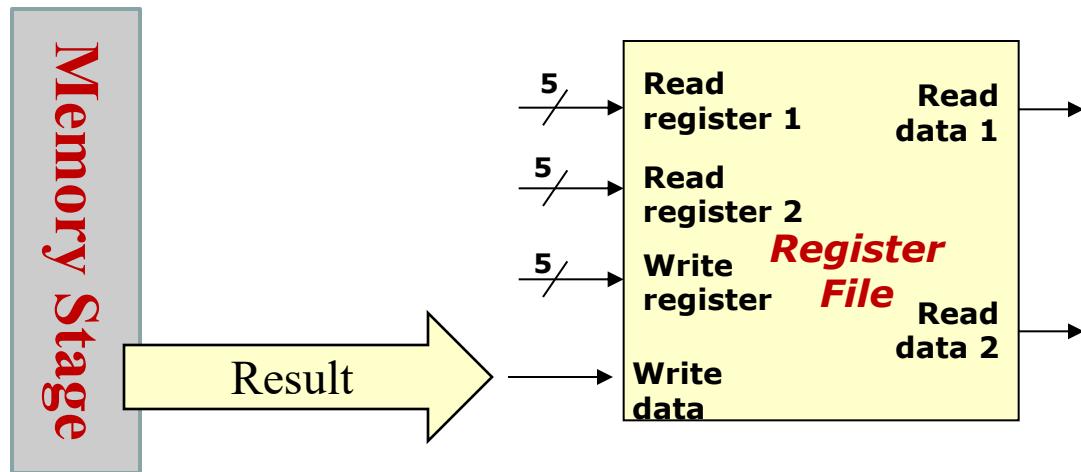
- Exceptions are stores, branches, jumps:
 - There are no results to be written
 - These instructions remain idle in this stage

■ Input from previous stage (Memory):

- Computation result either from memory or ALU



5.5 Register Write Stage: Block Diagram

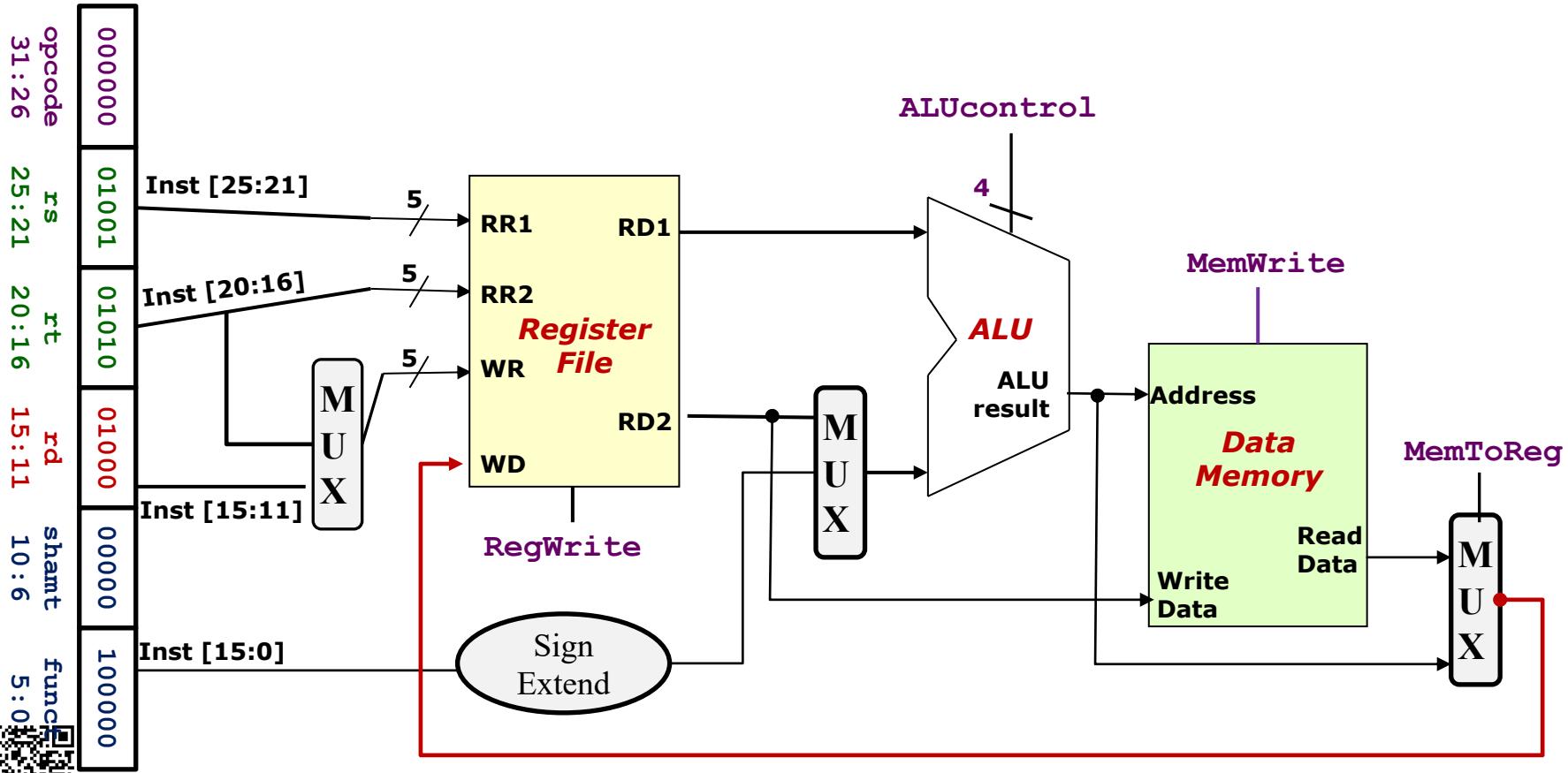


- **Result Write stage has no additional element:**
 - Basically just route the correct result into register file
 - The *Write Register* number is generated way back in the **Decode Stage**



5.5 Register Write Stage: Routing

add \$8, \$9, \$10

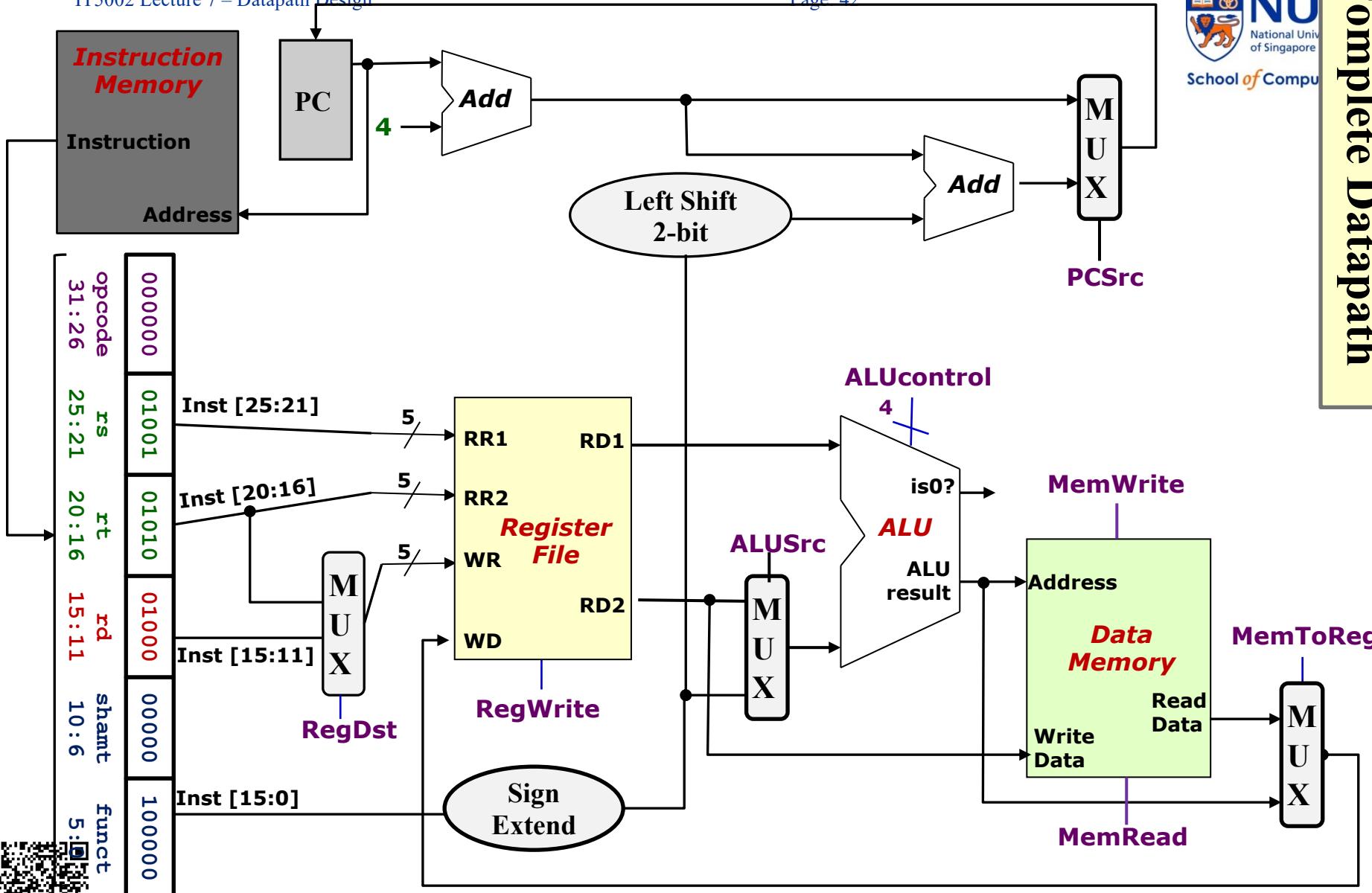


6. The Complete Datapath!

- We have just finished “designing” the datapath for a subset of MIPS instructions:
 - Shifting and Jump are not supported
- Check your understanding:
 - Take the complete datapath and play the role of controller:
 - See how supported instructions are executed
 - Figure out the correct control signals for the datapath elements
- Coming up next: Control



Complete Datapath



Lecture #7a

The Processor: Datapath

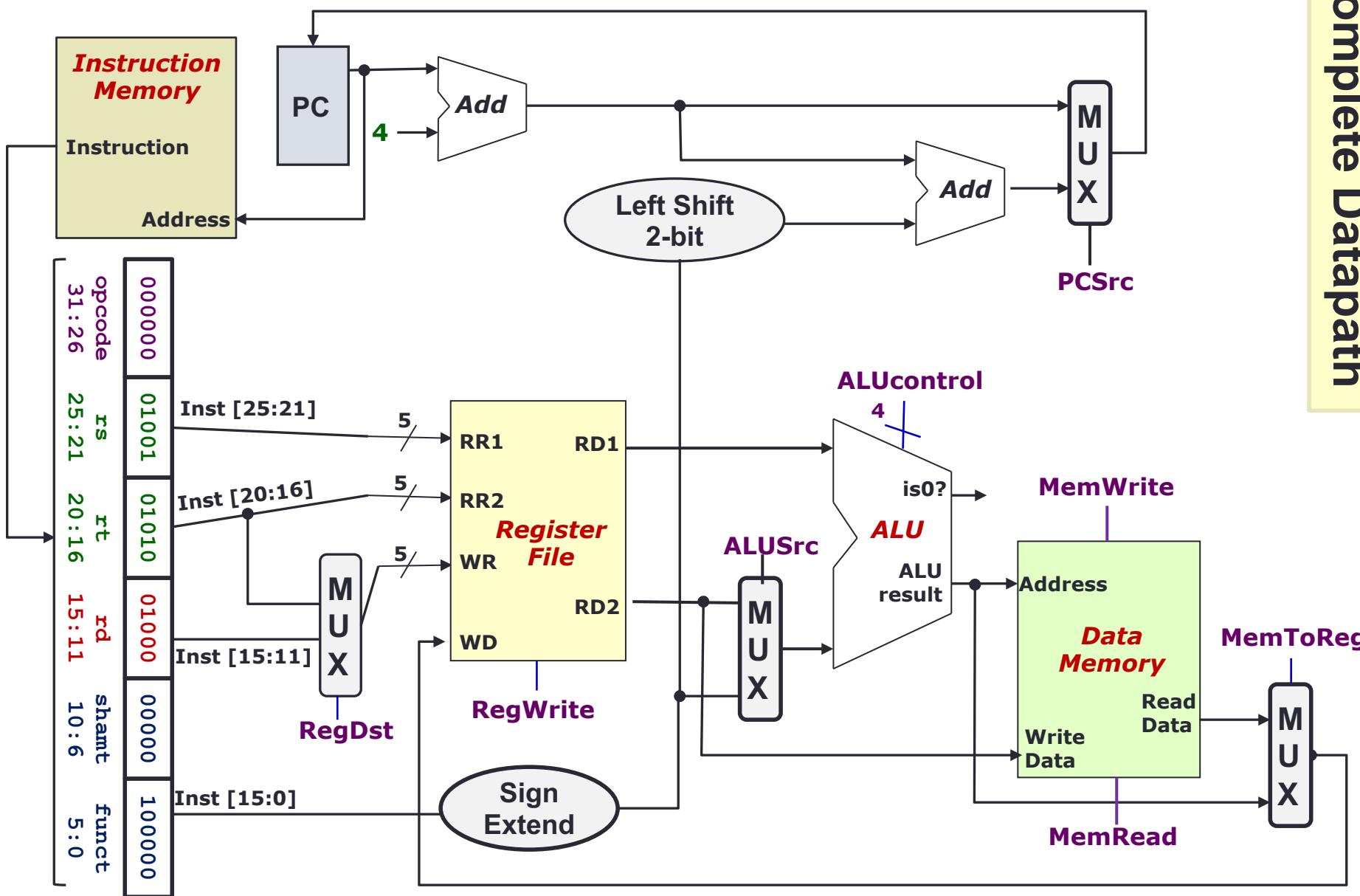


Lecture #7a: Processor: Datapath

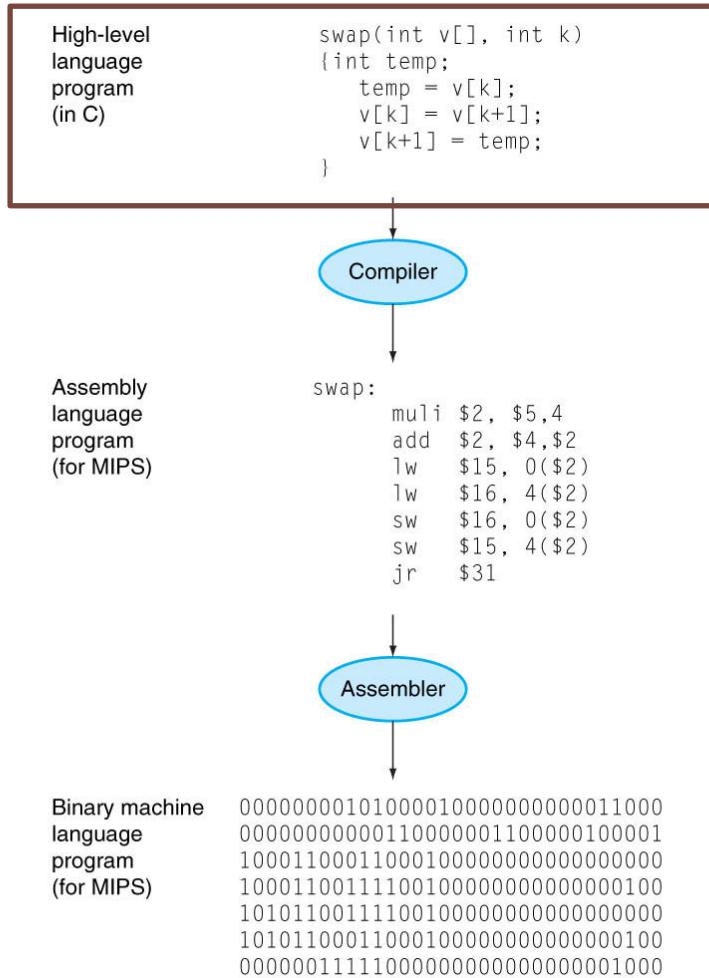
1. The Complete Datapath!
2. Brief Recap
3. From C to Execution
 - 3.1 Writing C Program
 - 3.2 Compiling to MIPS
 - 3.3 Assembling to Binaries
 - 3.4 Execution (Datapath)

Taken from CS2100 Lecture 11a.

Complete Datapath



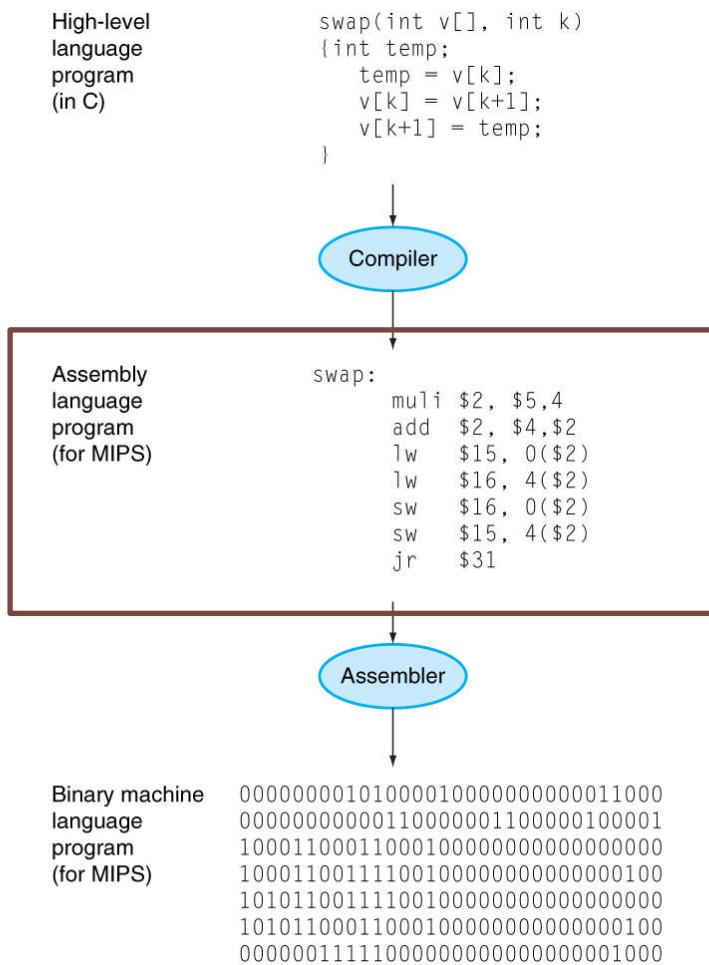
2. Brief Recap



Write program in high-level language (e.g., C)

```
if(x != 0) {  
    a[0] = a[1] + x;  
}
```

2. Brief Recap

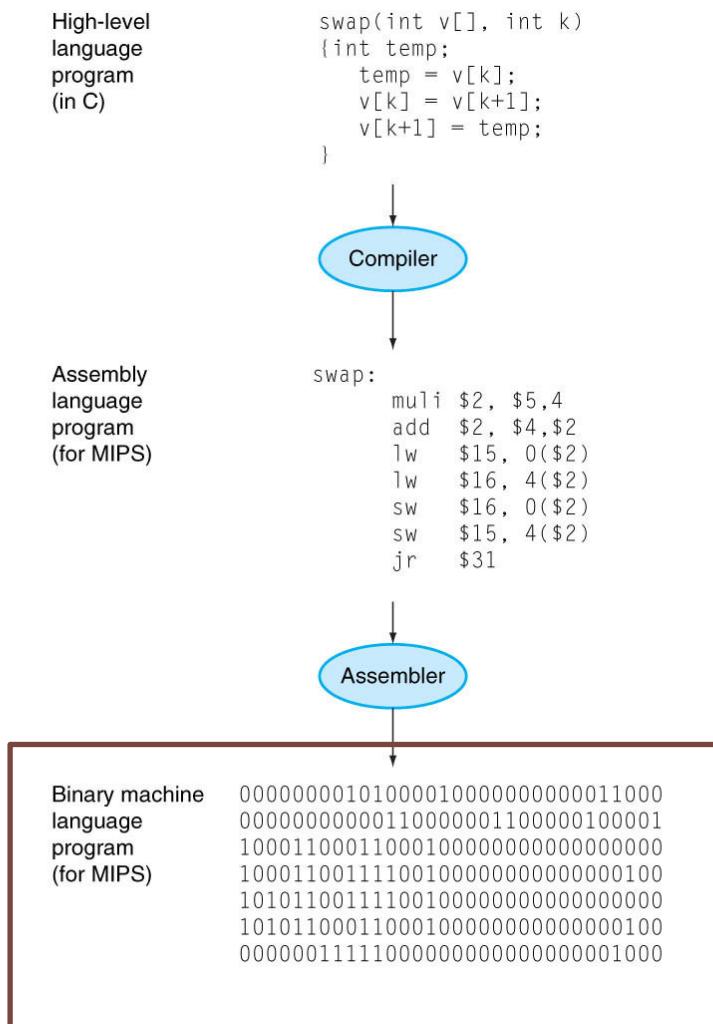


Compiler translates to assembly language (e.g., **MIPS**)

```
beq $16, $0, Else  
lw  $8, 4($17)  
add $8, $8, $16  
sw  $8, 0($17)
```

Else:

2. Brief Recap



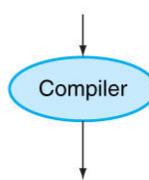
Assembler translates to machine code (i.e., **binaries**)

0001	0010	0000	0000
0000	0000	0000	0011
1000	1110	0010	1000
0000	0000	0000	0100
0000	0010	0000	1000
0100	0000	0001	0100
1010	1110	0010	1000
0000	0000	0000	0000

2. Brief Recap

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;
}
```

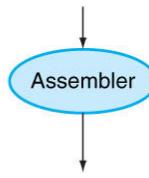


Assembly language program (for MIPS)

```

swap:    muli $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

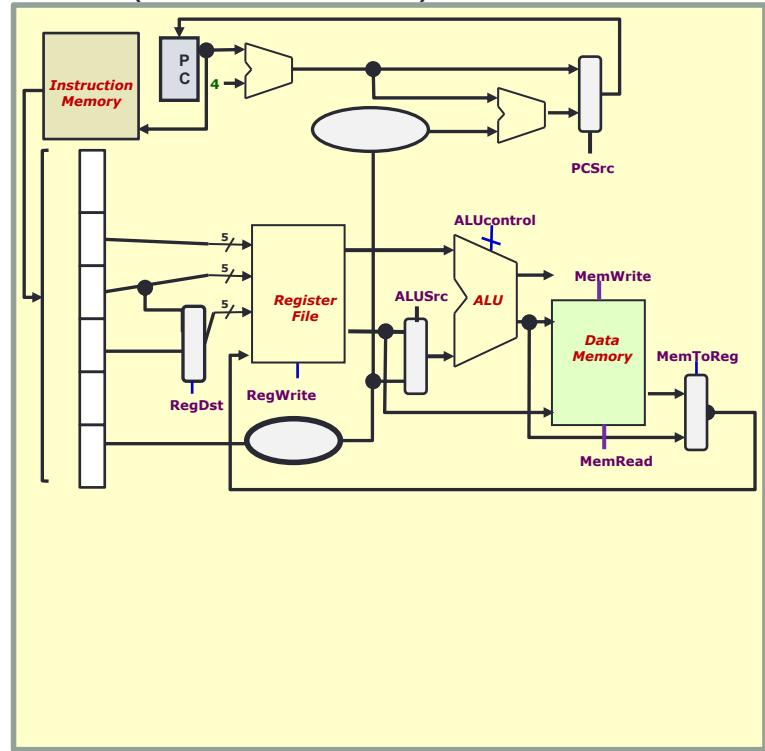
```



Binary machine language program (for MIPS)

```
000000001010000100000000000011000  
00000000000110000001100000100001  
100011000110001000000000000000000  
100011001111001000000000000000000  
101011001111001000000000000000000  
101011000110001000000000000000000  
0000001111100000000000000000000000
```

Processor executes the machine code (i.e., **binaries**)



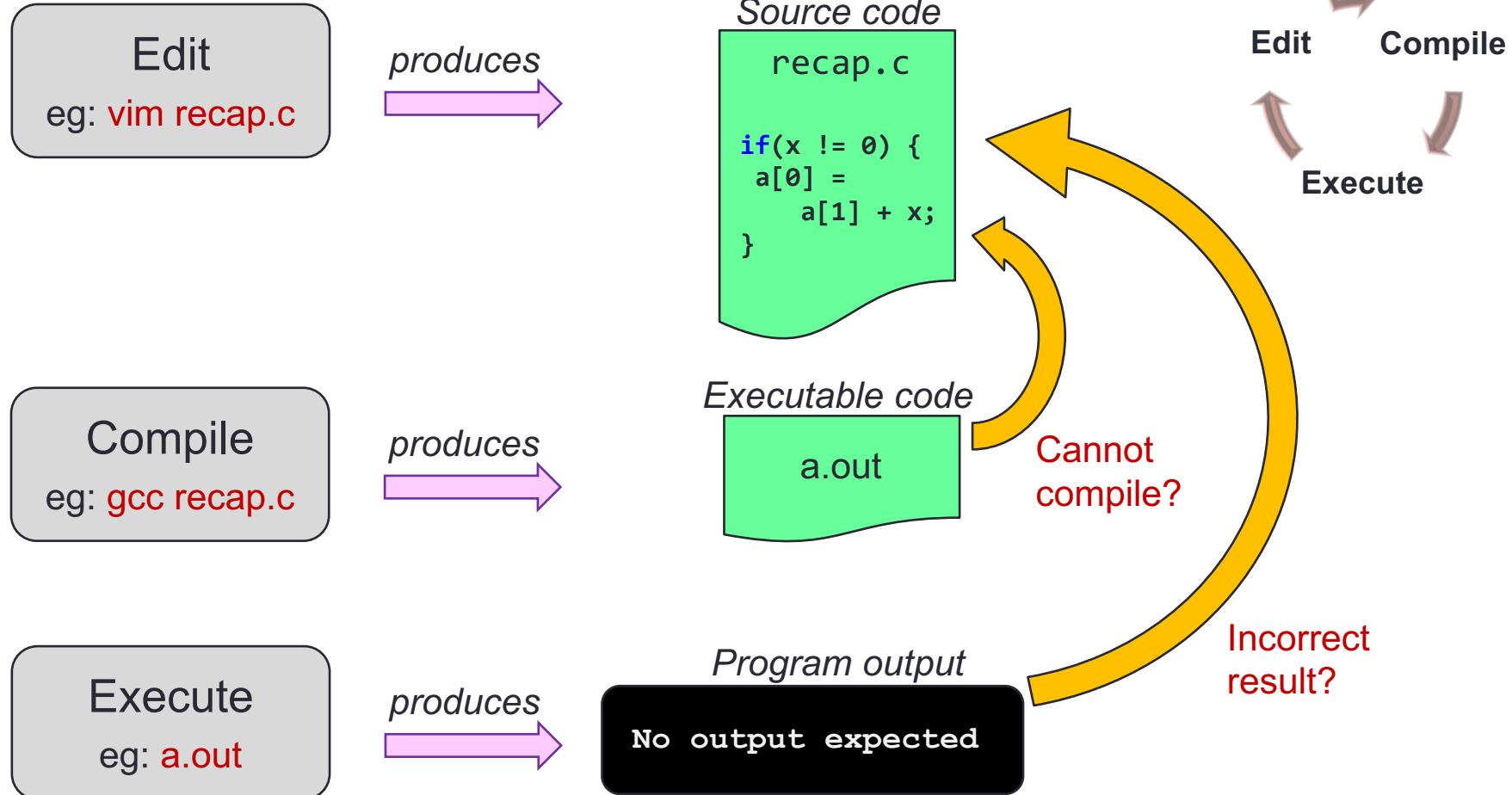
3. From C to Execution

- We play the role of **Programmer, Compiler, Assembler, and Processor**
 - Program:

```
if(x != 0) {  
    a[0] = a[1] + x;  
}
```
 - Programmer:
 - Show the workflow of compiling, assembling, and executing C program
 - Compiler:
 - Show how the program is compiled into MIPS
 - Assembler:
 - Show how the MIPS is translated into binaries
 - Processor:
 - Show how the datapath is activated in the processor

3.1 Writing C Program

- Edit, Compile, Execute: Lecture #2, Slide 5



3.2 Compiling to MIPS

- Key Idea #1:

Compilation is a *structured process*

```
if(x != 0) {
    a[0] = a[1] + x;
}
```

- Each structure can be compiled *independently*

Inner Structure

```
a[0] = a[1] + x;
```

Outer Structure

```
if (x != 0) {
```

```
}
```

recap.c

```
if(x != 0) {
    a[0] =
        a[1] + x;
}
```

recap.c

```
if(x != 0) {  
    a[0] =  
        a[1] + x;  
}
```

3.2 Compiling to MIPS

- Key Idea #2:
Variable-to-Register Mapping

```
if(x != 0) {  
    a[0] = a[1] + x;  
}
```

- Let the mapping be:

Variable	Register Name	Register Number
x	\$s0	\$16
a	\$s1	\$17

3.2 Compiling to MIPS

- Common Technique #1:
Invert the condition for shorter code
(Lecture #8, Slide 22)

Mapping:

x: \$16
a: \$17

recap.c

```
if(x != 0) {
    a[0] =
        a[1] + x;
}
```

Outer Structure

```
if(x != 0) {
}
}
```

Outer MIPS Code

beq \$16, \$0, Else

Inner Structure

Else:

3.2 Compiling to MIPS

- Common Technique #2:
Break complex operations, use temp register
(Lecture #7, Slide 29)

Inner Structure

```
a[0] = a[1] + x;
```

Simplified Inner Structure

```
$t1 = a[1];  
$t1 = $t1 + x;  
a[0] = $t1;
```

Mapping:

x: \$16
a: \$17
\$t1: \$8

recap.c

```
if(x != 0) {  
    a[0] =  
        a[1] + x;  
}
```

3.2 Compiling to MIPS

- Common Technique #3:
Array access is **lw**, array update is **sw**
(Lecture #8, Slide 13)

Mapping:

x: \$16
a: \$17
\$t1: \$8

recap.c

```
if(x != 0) {
    a[0] =
        a[1] + x;
}
```

Simplified Inner Structure

```
$t1 = a[1];
$t1 = $t1 + x;
a[0] = $t1;
```

Inner MIPS Code

```
lw $8, 4($17)
add $8, $8, $16
sw $8, 0($17)
```

3.2 Compiling to MIPS

- Common Error #1:

Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes

- Example:

- `$t1 = a[1];`

is translated to

`lw $8, 4($17)`

instead of

`lw $8, 1($17)`

Mapping:

x: \$16
a: \$17
\$t1: \$8

recap.c

```
if(x != 0) {  
    a[0] =  
        a[1] + x;  
}
```

3.2 Compiling to MIPS

- Last Step:
Combine the two structures logically

Inner MIPS Code

```
lw $8, 4($17)
add $8, $8, $16
sw $8, 0($17)
```

Outer MIPS Code

```
beq $16, $0, Else
# Inner Structure
Else:
```

Combined MIPS Code

```
beq $16, $0, Else
lw $8, 4($17)
add $8, $8, $16
sw $8, 0($17)
```

Else:

Mapping:

```
x: $16
a: $17
$t1: $8
```

recap.c

```
if(x != 0) {
    a[0] =
        a[1] + x;
}
```

recap.mips

```

beq $16, $0, Else
lw $8, 4($17)
add $8, $8, $16
sw $8, 0($17)

```

Else:

3.3 Assembling to Binary

■ Instruction Types Used:

1. R-Format: (Lecture #9, Slide 8)

- **opcode** \$rd, \$rs, \$rt



2. I-Format: (Lecture #9, Slide 14)

- **opcode** \$rt, \$rs, **immediate**



3. Branch: (Lecture #9, Slide 22)

- Uses I-Format
- **PC = (PC + 4) + (immediate × 4)**

recap.mips

```

beq $16, $0, Else
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)

```

Else:

3.3 Assembling to Binary

- **beq \$16, \$0, Else**

- Compute immediate value
(Lecture #9, Slide 27)

- **immediate = 3**

- Fill in fields (*refer to MIPS Reference Data*)

6	5	5	16
4	16	0	3

- Convert to binary

000100	10000	00000	00000000000000011
--------	-------	-------	-------------------

beq \$16, \$0, Else

lw \$8, 4(\$17)

add \$8, \$8, \$16

sw \$8, 0(\$17)

+3

Else:

recap.mips

```

beq $16, $0, Else
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)

```

Else:

3.3 Assembling to Binary

- lw \$8, 4(\$17)**

- Fill in fields (*refer to MIPS Reference Data*)

6	5	5	16
35	17	8	4

- Convert to binary

100011	10001	01000	00000000000000100
--------	-------	-------	-------------------

```
0001 0010 0000 0000 0000 0000 0000 0000 0011
```

lw \$8, 4(\$17)

add \$8, \$8, \$16

sw \$8, 0(\$17)

Else:

recap.mips

```

beq $16, $0, Else
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)

```

Else:

3.3 Assembling to Binary

- add \$8, \$8, \$16**

- Fill in fields (*refer to MIPS Reference Data*)

6	5	5	5	5	6
0	8	16	8	0	32

- Convert to binary

000000	01000	10000	01000	00000	100000
--------	-------	-------	-------	-------	--------

0001 0010 0000 0000 0000 0000 0000 0011

1000 1110 0010 1000 0000 0000 0000 0100

add \$8, \$8, \$16

sw \$8, 0(\$17)

Else:

recap.mips

```

beq $16, $0, Else
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)

```

Else:

3.3 Assembling to Binary

- sw \$8, 0(\$17)**

- Fill in fields (*refer to MIPS Reference Data*)

6	5	5	16
43	17	8	0

- Convert to binary

101011	10001	01000	0000000000000000
--------	-------	-------	------------------

```

0001 0010 0000 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0000 0100
0000 0001 0001 0000 0100 0000 0010 0000
      sw $8, 0($17)

```

Else:

recap.mips

```
beq $16, $0, Else
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)
```

Else:

3.3 Assembling to Binary

- Final Binary
 - Hard to read?
 - Don't worry, this is intended for machine not for human!

0001	0010	0000	0000	0000	0000	0000	0000	0011
1000	1110	0010	1000	0000	0000	0000	0100	
0000	0001	0001	0000	0100	0000	0010	0000	
1010	1110	0010	1000	0000	0000	0000	0000	

3.4 Execution (Datapath)

- Given the binary

- Assume two possible executions:

- 1. $\$16 == \0 (*shorter*)

- 2. $\$16 != \0 (*Longer*)

- Convention:

Fetch: 

Memory: 

Decode: 

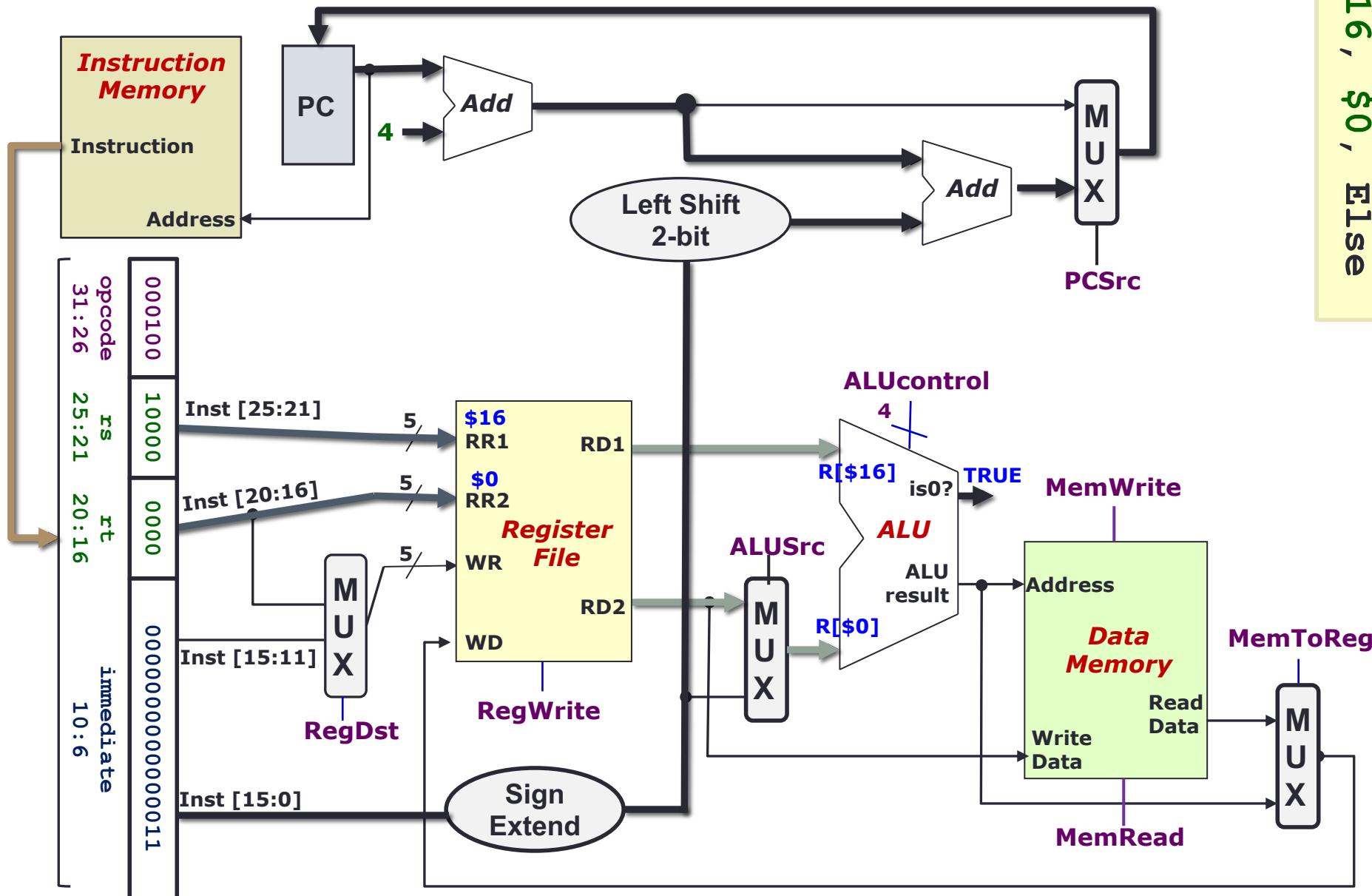
Reg Write: 

ALU: 

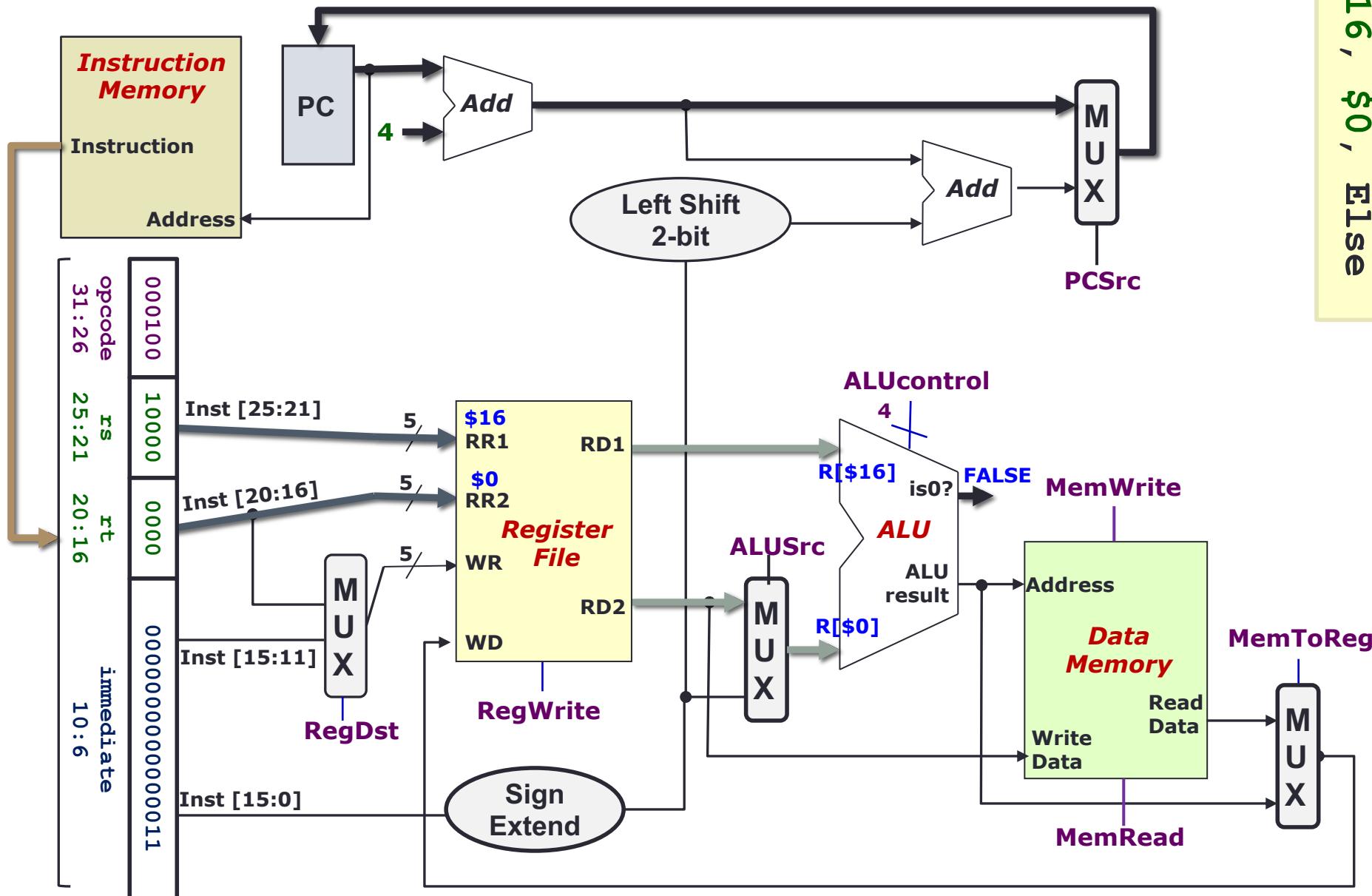
Other: 

0001	0010	0000	0000	0000	0000	0000	0000	0011
1000	1110	0010	1000	0000	0000	0000	0100	
0000	0001	0001	0000	0100	0000	0010	0000	
1010	1110	0010	1000	0000	0000	0000	0000	

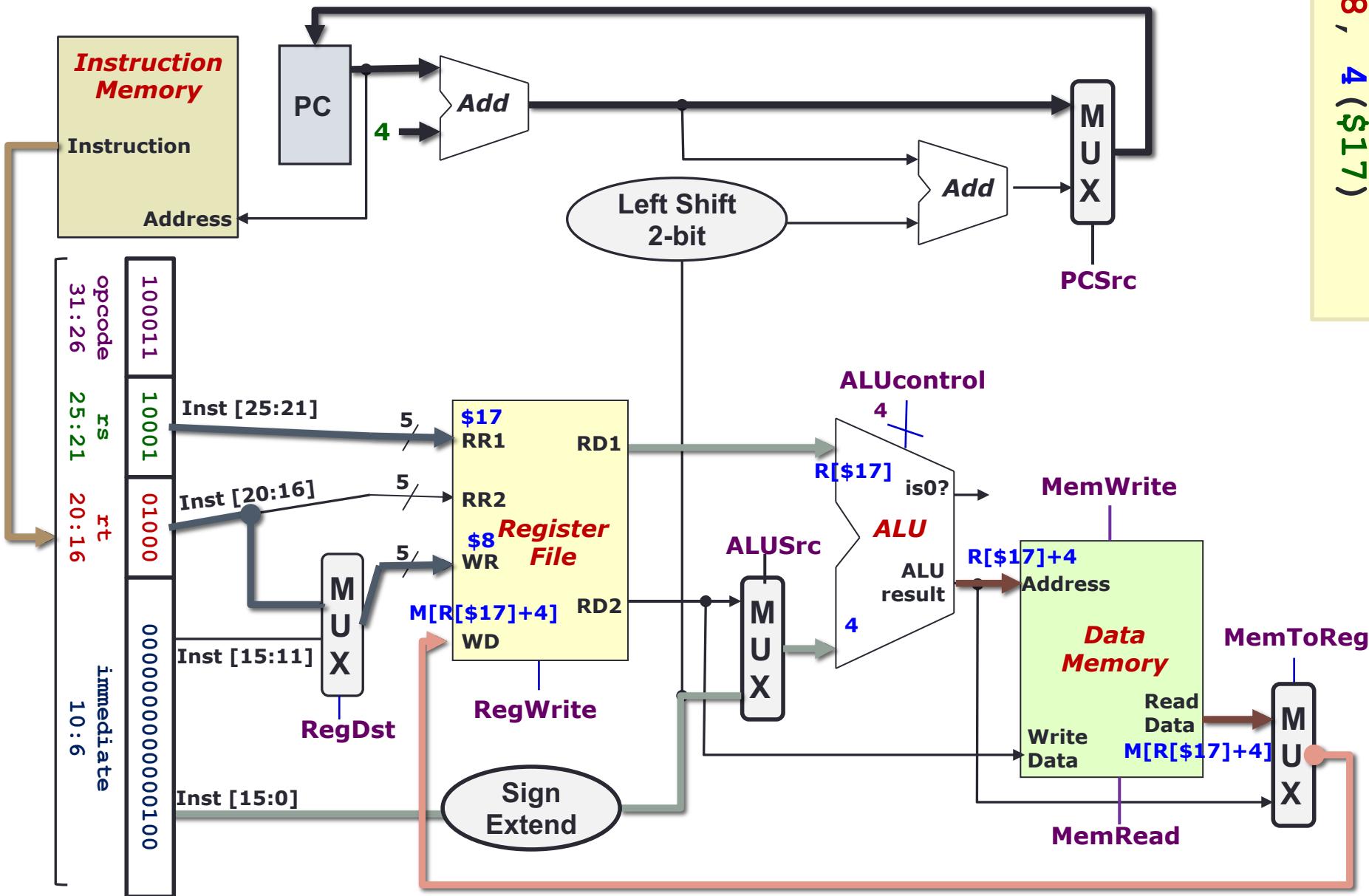
- Assume \$16 == \$0



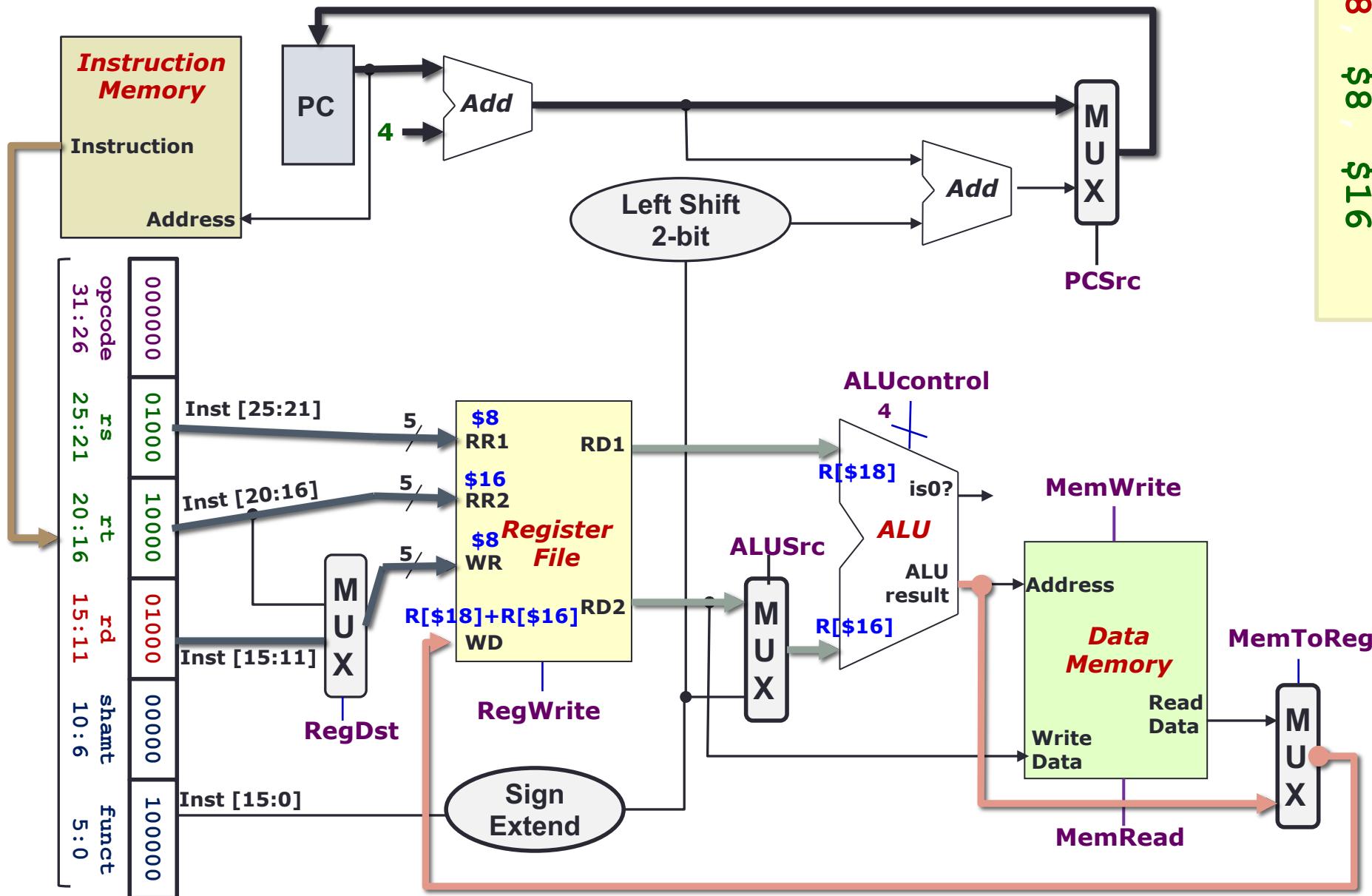
- Assume $\$16 \neq \0



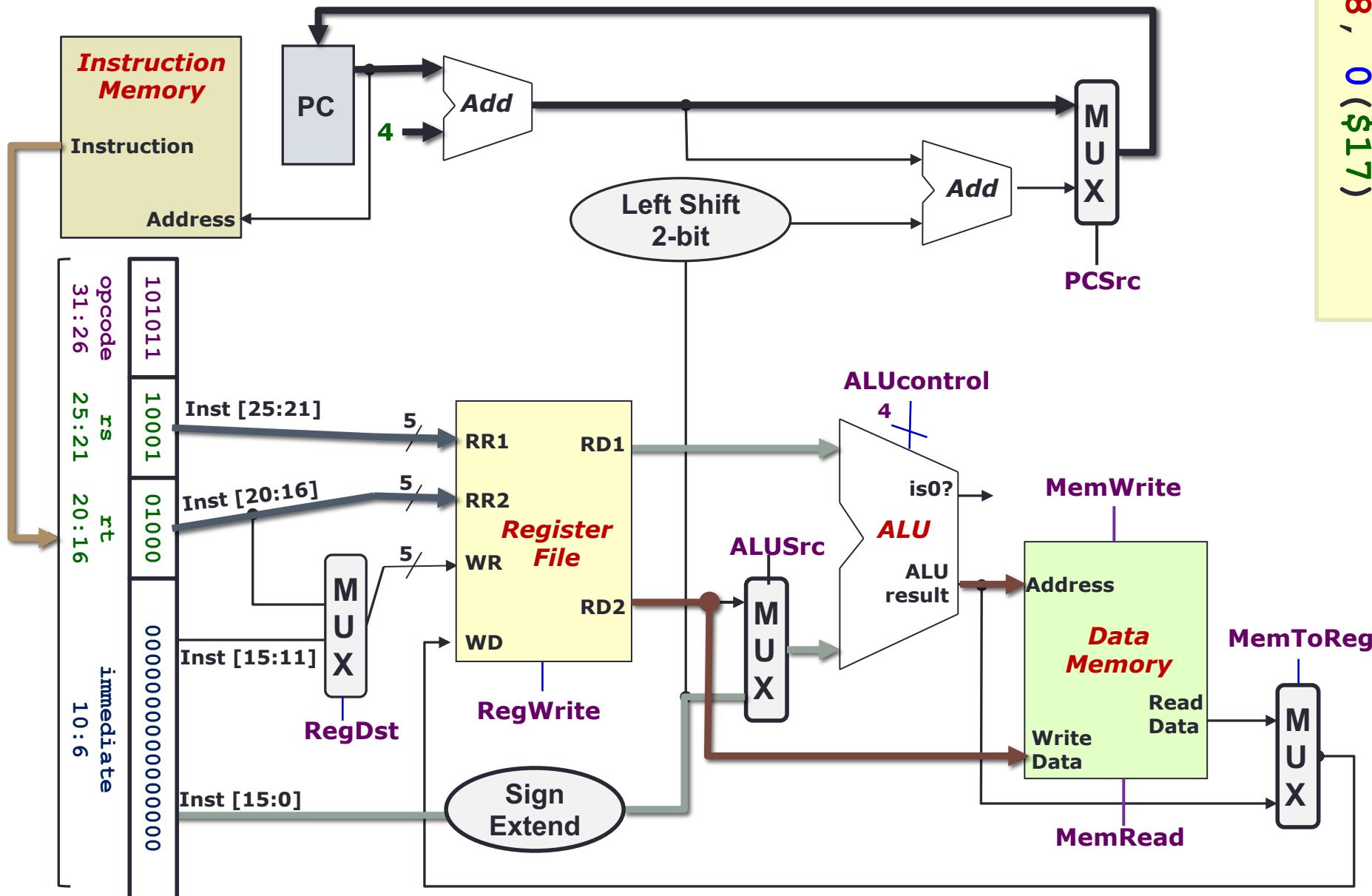
- Assume $\$16 \neq \0



- Assume $\$16 \neq \0



- Assume $\$16 \neq \0



End of File

IT5002

Computer Systems and Applications

CPU Control

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



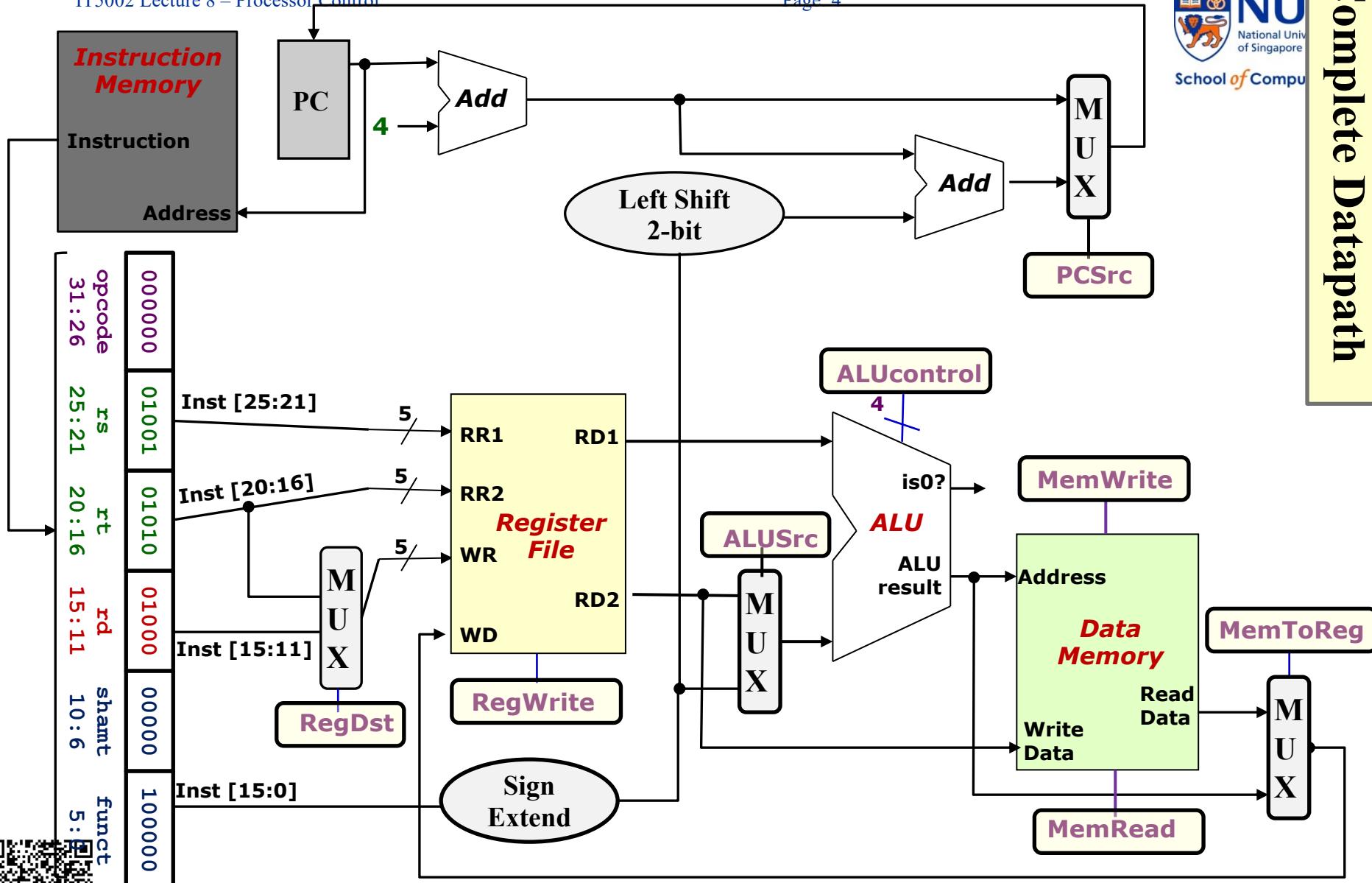
OR scan this QR code (may be obscured on some slides)

Lecture #8: Processor: Control

- 1. Identified Control Signals**
- 2. Generating Control Signals: Idea**
- 3. The Control Unit**
- 4. Control Signals**
- 5. ALU Control Signal**
- 6. Instruction Execution**



Complete Datapath



1. Identified Control Signals

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2 nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

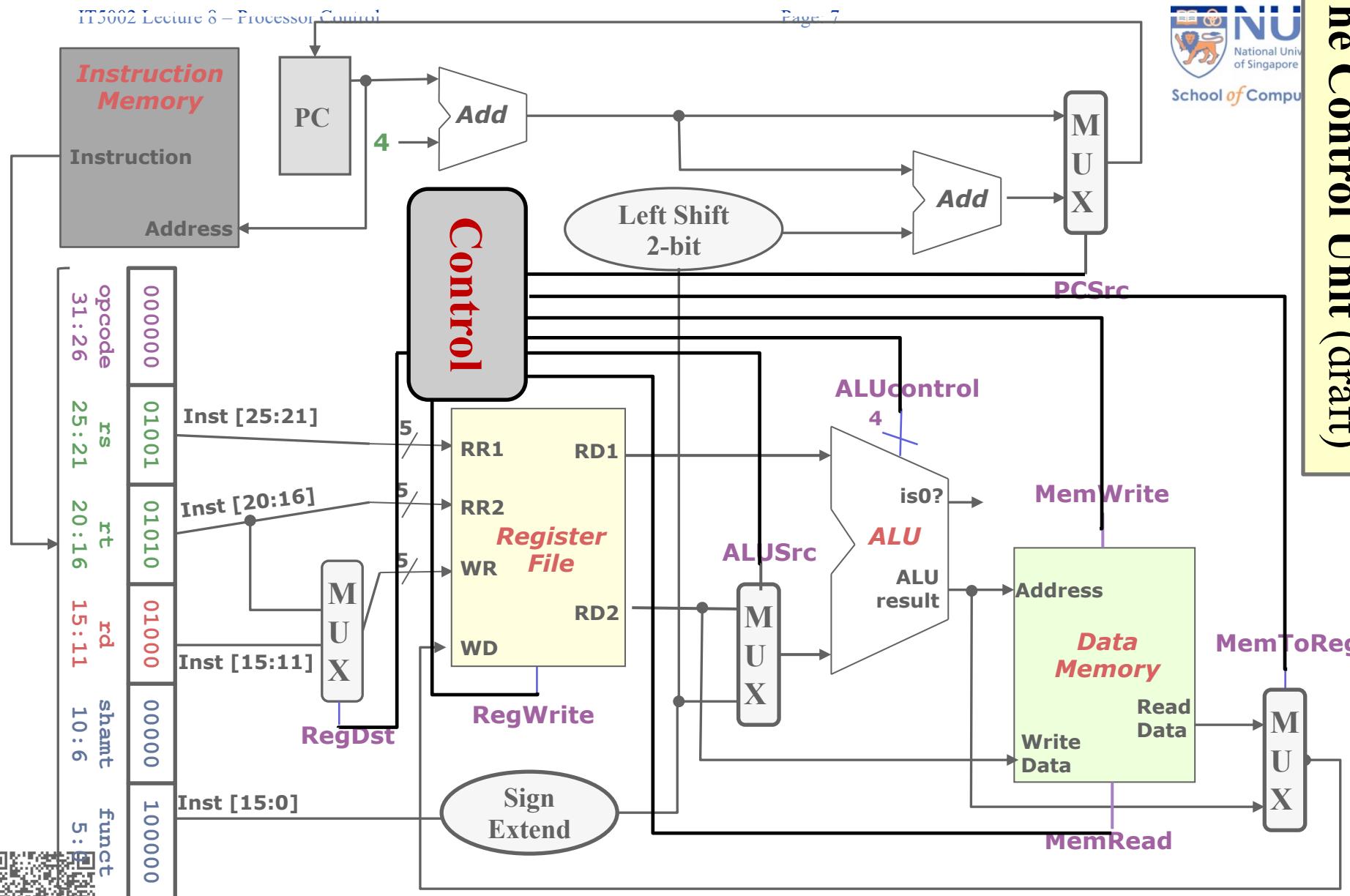


2. Generating Control Signals: Idea

- The control signals are generated based on the instruction to be executed:
 - Opcode → Instruction Format
 - Example:
 - R-Format instruction → $\text{RegDst} = 1$ (use $\text{Inst}[15:11]$)
 - R-Type instruction has additional information:
 - The 6-bit "funct" (function code, $\text{Inst}[5:0]$) field
 - Idea:
 - Design a combinatorial circuit to generate these signals based on Opcode and possibly Function code
 - A control unit is needed (a draft design is shown next)



The Control Unit (draft)



3. Let's Implement the Control Unit!

■ Approach:

- Take note of the instruction subset to be implemented:
 - **Opcode and Function Code (if applicable)**
- Go through each signal:
 - **Observe how the signal is generated based on the instruction opcode and/or function code**
- Construct truth table
- Design the control unit using logic gates



3. MIPS Instruction Subset (Review)

	opcode			shamt	funct	
	31	25	20	15	10	5
add	0_{16}	rs	rt	rd	0	20_{16}
sub	0_{16}	rs	rt	rd	0	22_{16}
and	0_{16}	rs	rt	rd	0	24_{16}
or	0_{16}	rs	rt	rd	0	25_{16}
slt	0_{16}	rs	rt	rd	0	$2A_{16}$

R-type

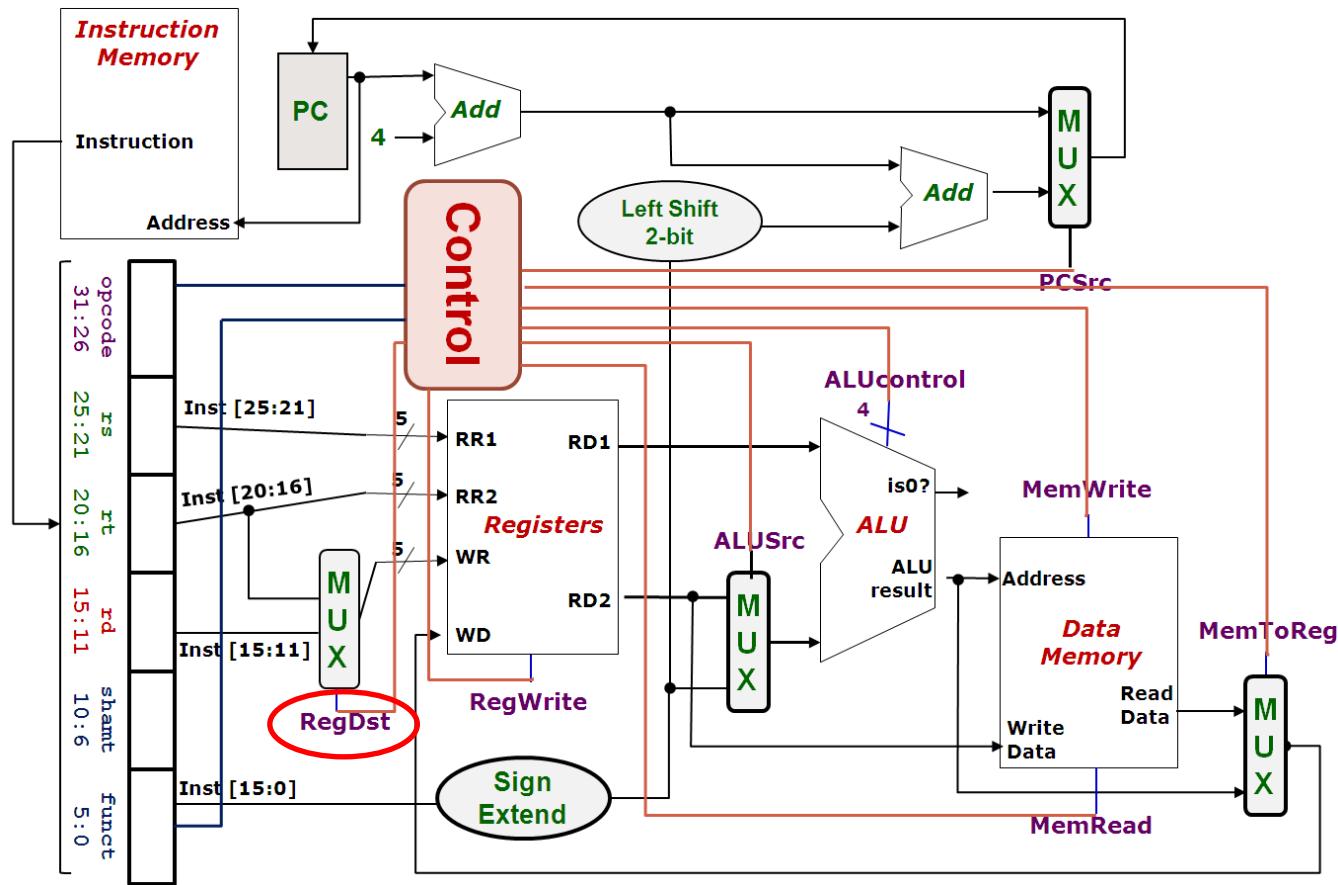
	31	25	20	15		
lw	23_{16}	rs	rd		offset	
sw	$2B_{16}$	rs	rd		offset	
beq	4_{16}	rs	rd		offset	

I-type



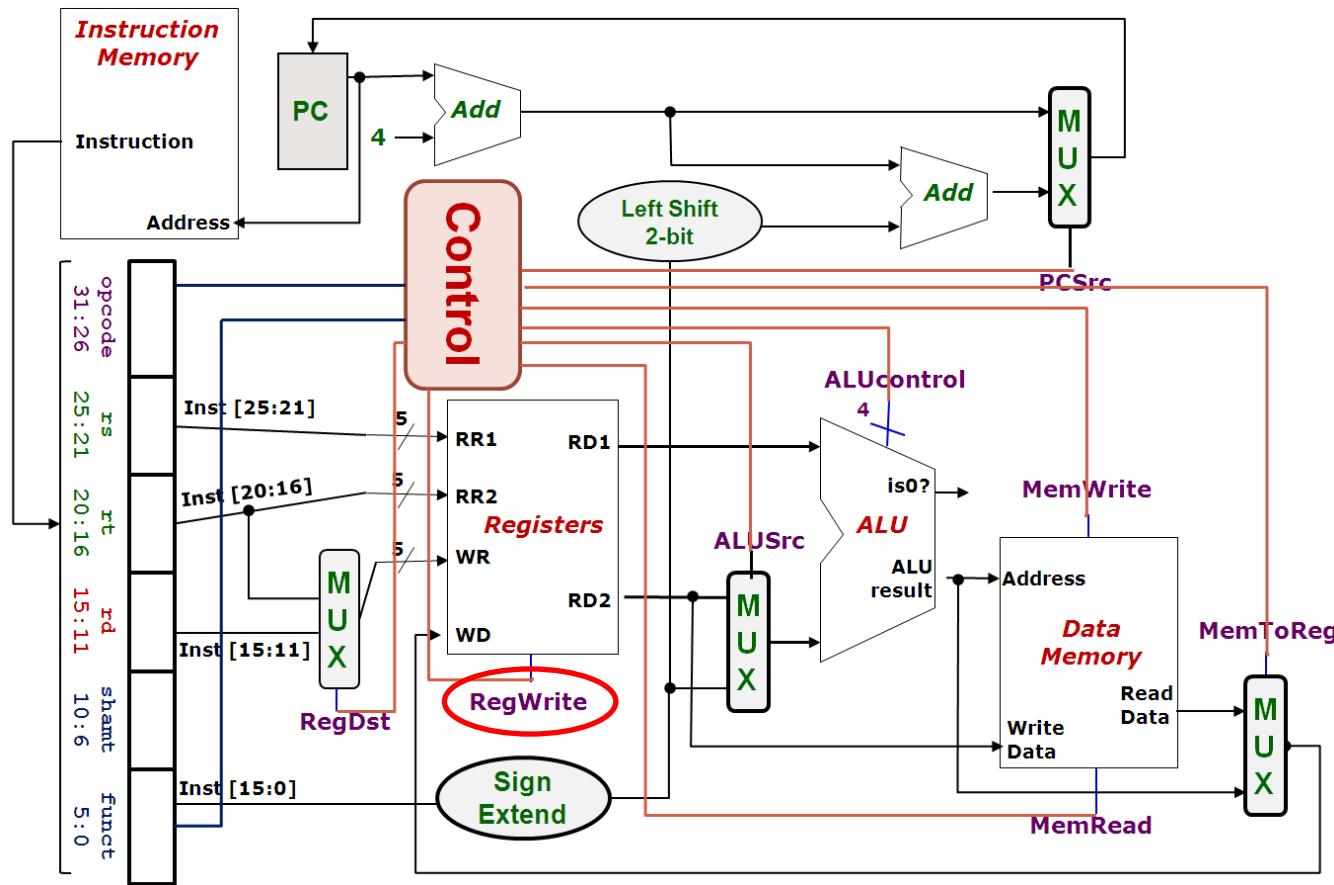
4. Control Signal: **RegDst**

- **False (0):** Write register = **Inst** [20:16]
 - **True (1):** Write register = **Inst** [15:11]



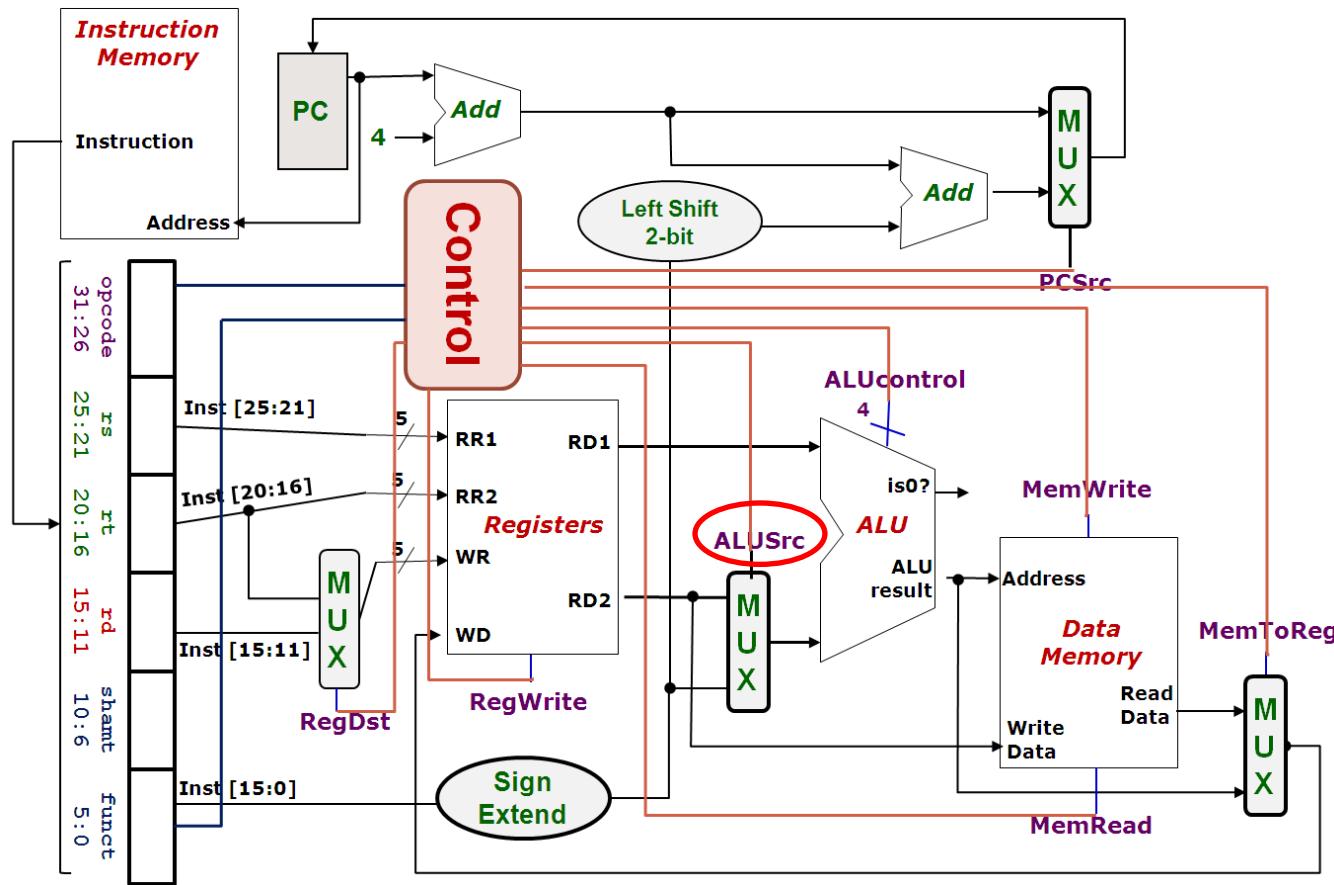
4. Control Signal: RegWrite

- False (0): No register write
- True (1): New value will be written



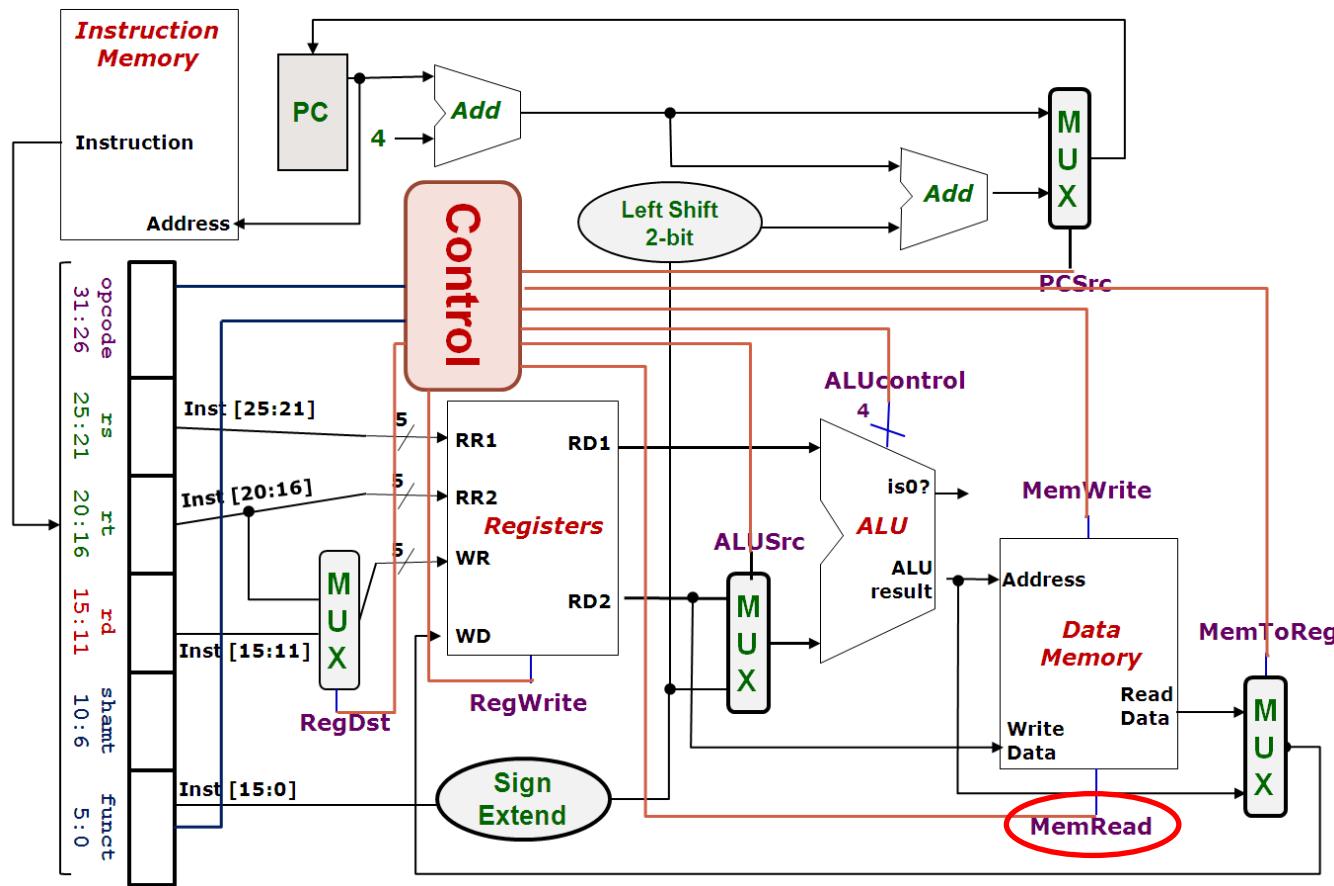
4. Control Signal: ALUSrc

- False (0): Operand2 = Register Read Data 2
- True (1): Operand2 = SignExt(**Inst** [15 : 0])



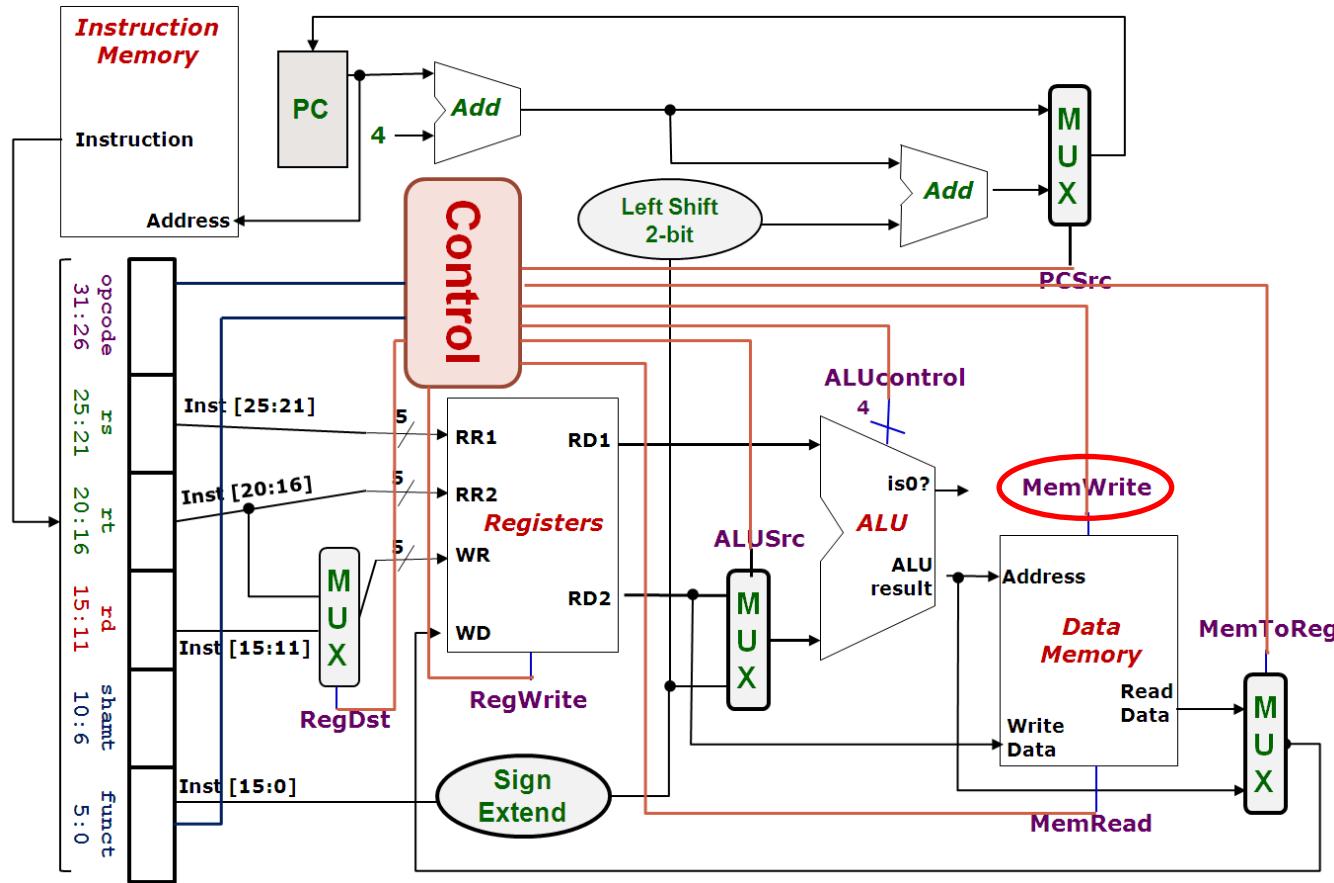
4. Control Signal: MemRead

- False (0): Not performing memory read access
- True (1): Read memory using *Address*



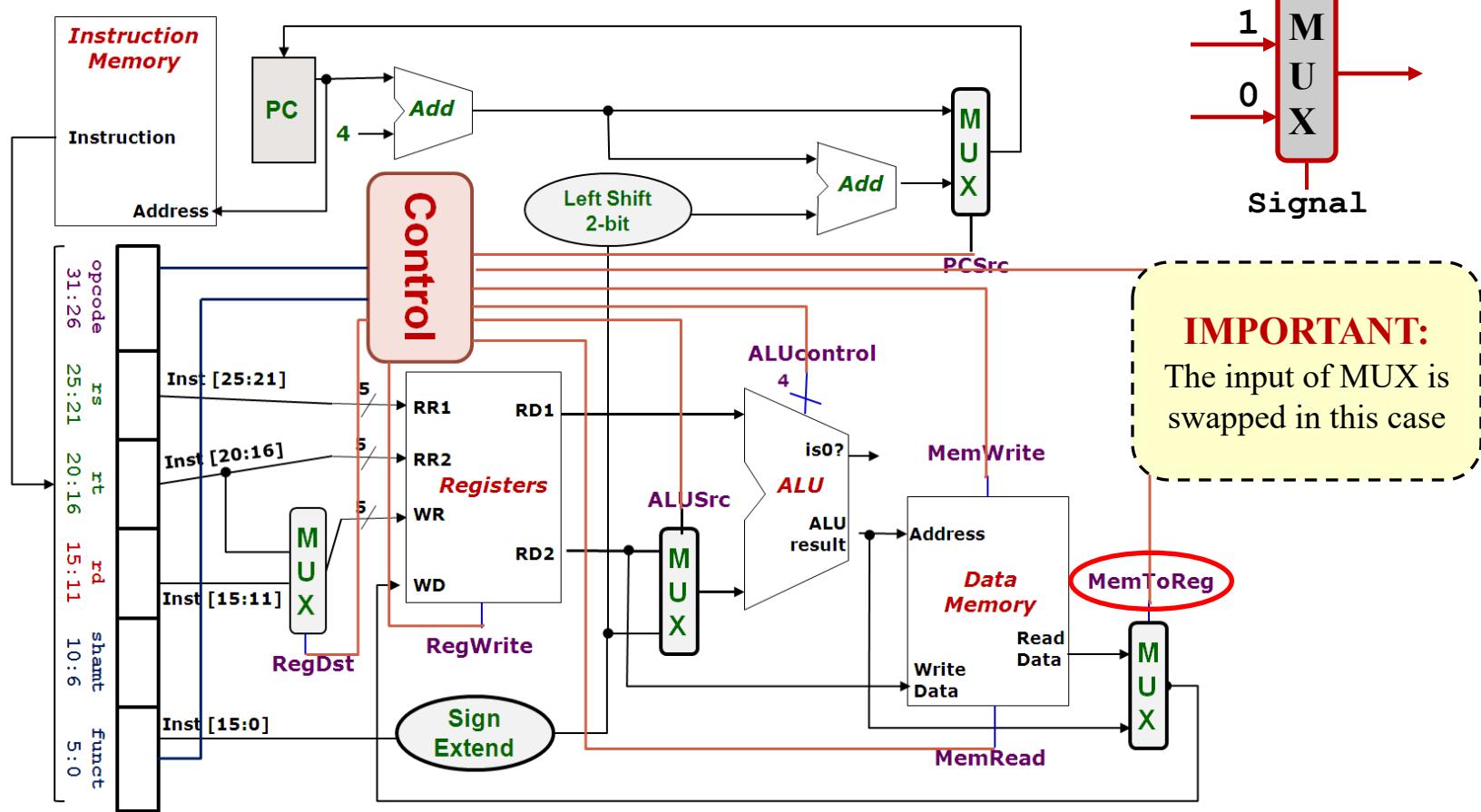
4. Control Signal: MemWrite

- False (0): Not performing memory write operation
- True (1): $\text{memory}[\text{Address}] \leftarrow \text{Register Read Data 2}$



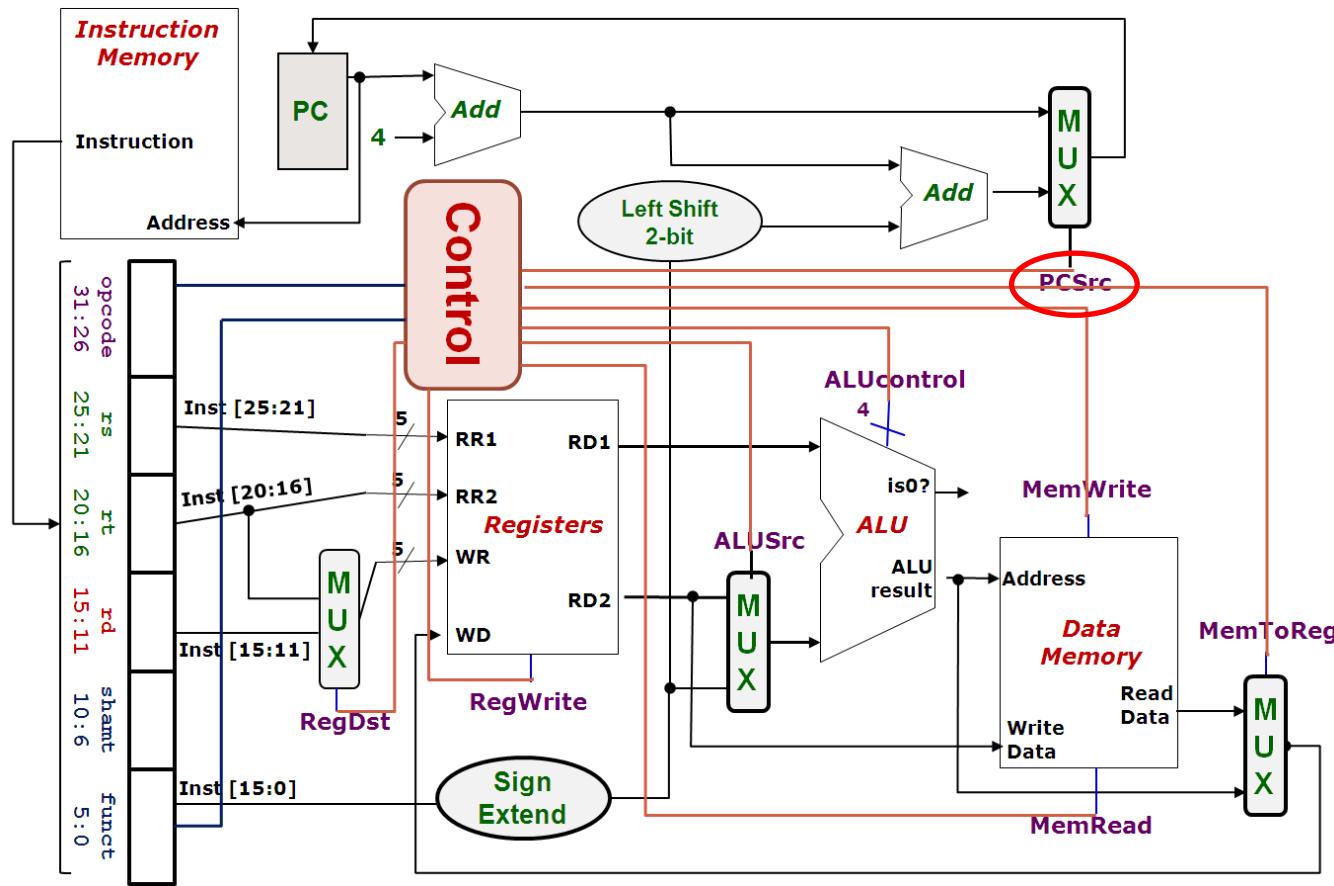
4. Control Signal: MemToReg

- True (1): Register write data = Memory read data
- False (0): Register write data = ALU result



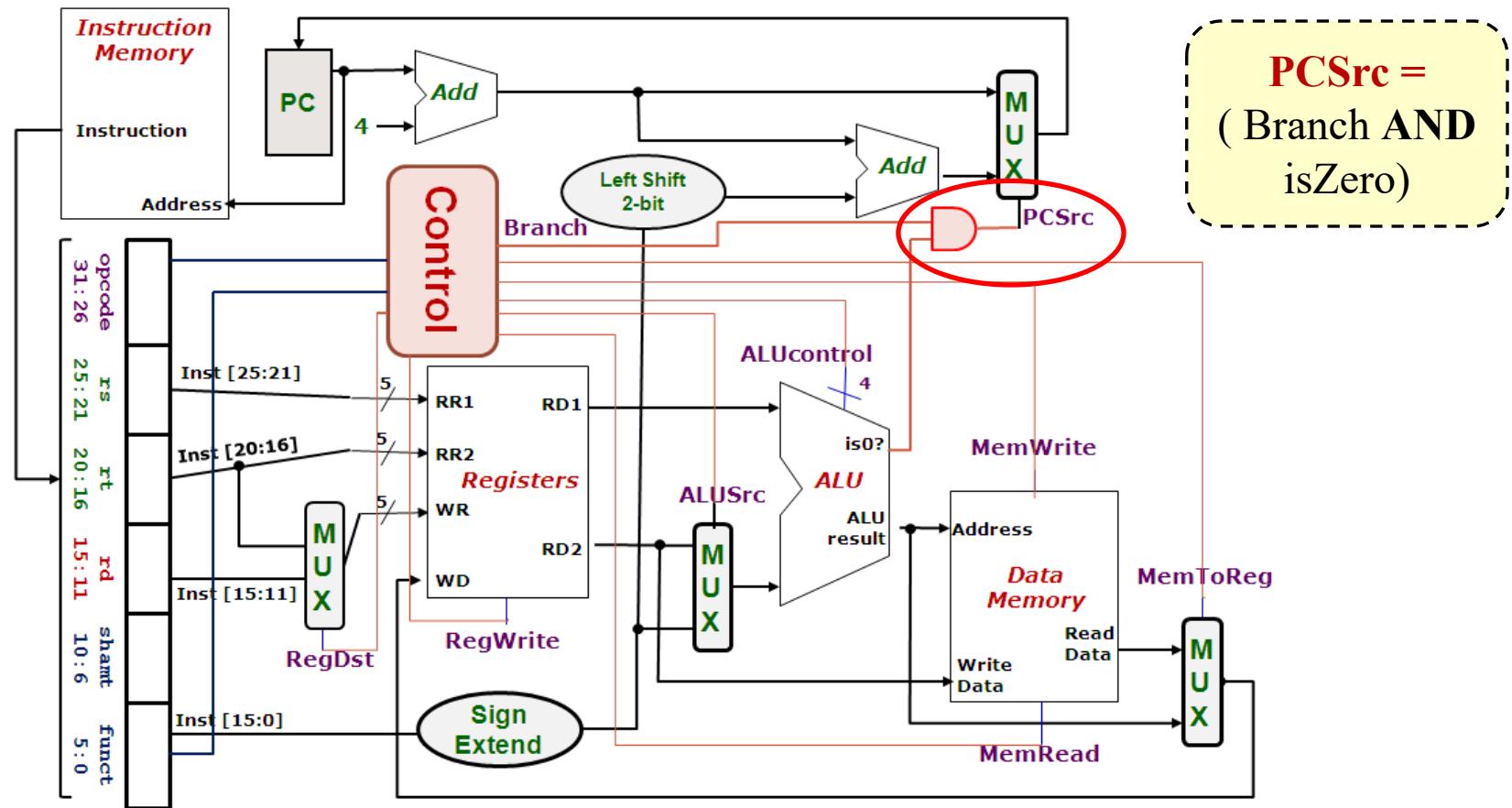
4. Control Signal: PCSrc (1/2)

- The "isZero?" signal from the ALU gives us the actual branch outcome (taken/not taken)
- Idea:** “If instruction is a branch **AND** taken, then...”



4. Control Signal: PCSrc (2/2)

- False (0): Next PC = PC + 4
- True (1): Next PC = SignExt(**Inst** [15:0]) << 2 + (PC + 4)



4. Midpoint Check

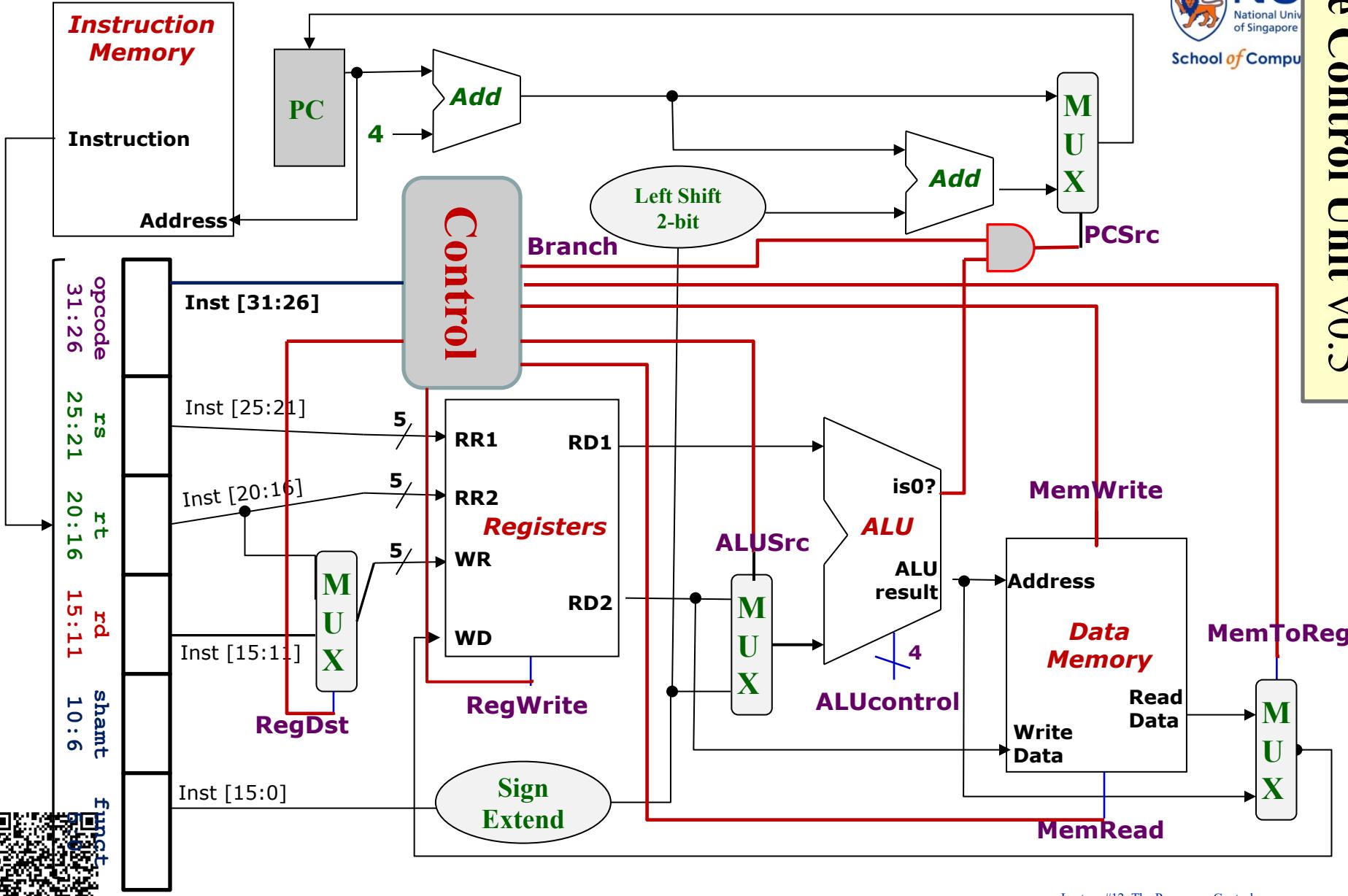
- We have gone through almost all of the signals:
 - Left with the more challenging **ALUControl** signal

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2 nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

- Observation so far:
 - The signals discussed so far can be generated by *opcode* directly
 - Function code is not needed up to this point
- A major part of the controller can be built based on *opcode* alone



The Control Unit v0.5



5. Closer Look at ALU

- The ALU is a combinatorial circuit:
 - Capable of performing several arithmetic operations
- In Lecture #11:
 - We noted the required operations for the MIPS subset

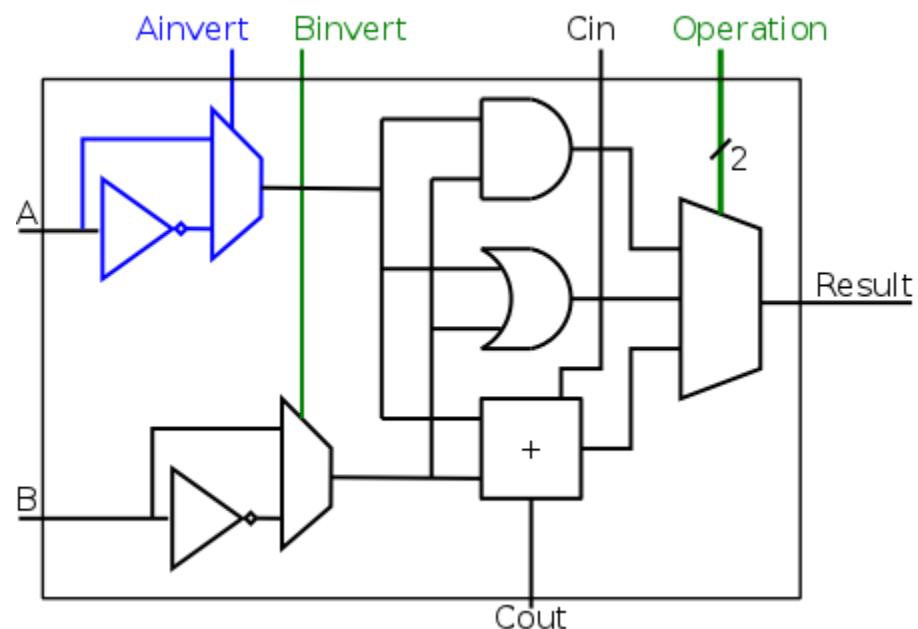
- Question:
 - How is the **ALUcontrol** signal designed?

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR



5. One Bit At A Time

- A simplified 1-bit MIPS ALU can be implemented as follows:
 - 4 control bits are needed:
 - **Ainvert:**
 - 1 to invert input A
 - **Binvert:**
 - 1 to invert input B
 - **Operation (2-bit)**
 - To select one of the 3 results





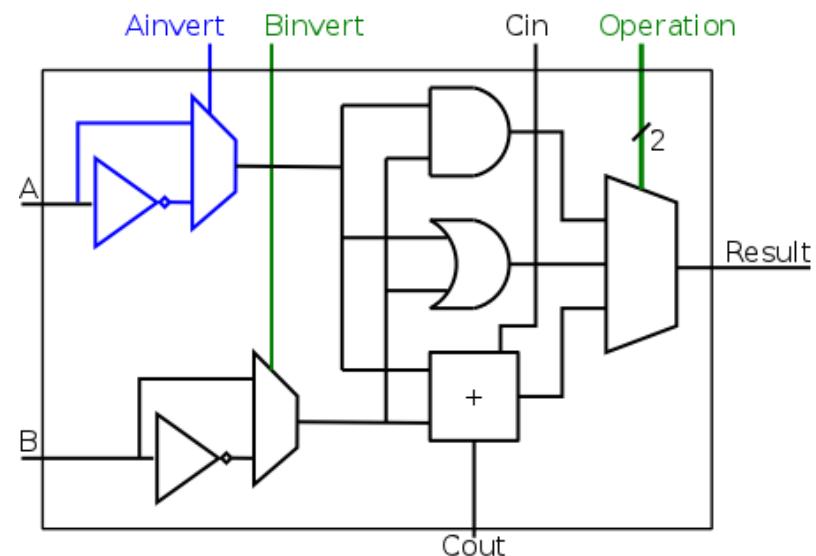




5. One Bit At A Time (Aha!)

- Can you see how the **ALUcontrol** (4-bit) signal controls the ALU?
 - Note: implementation for **slt** not shown

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR



5. Multilevel Decoding

- Now we can start to design for **ALUcontrol** signal, which depends on:
 - Opcode (6-bit) field **and** Function Code (6-bit) field
- **Brute Force approach:**
 - Use Opcode and Function Code directly, i.e. finding expressions with 12 variables
- **Multilevel Decoding approach:**
 - Use some of the input to reduce the cases, then generate the full output
 - Simplify the design process, reduce the size of the main controller, potentially speedup the circuit



5. Intermediate Signal: ALUop

- **Basic Idea:**

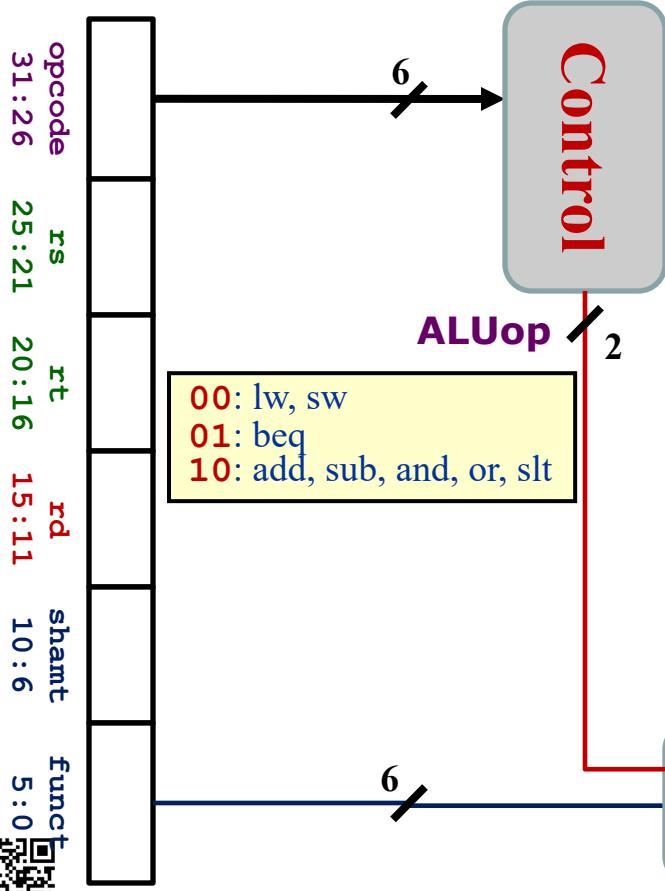
1. Use Opcode to generate a 2-bit **ALUop** signal
 - Represents classification of the instructions:

Instruction type	ALUop
Iw / sw	00
beq	01
R-type	10

2. Use **ALUop** signal and Function Code field (for R-type instructions) to generate the 4-bit **ALUcontrol** signal



5. Two-level Implementation



Step 1.

Generate 2-bit **ALUop** signal from 6-bit opcode.

Step 2.

Generate **ALUcontrol** signal from **ALUop** and optionally 6-bit **Funct** field.

0000	: and
0001	: or
0010	: add
0110	: sub
0111	: set on less than



5. Generating ALUcontrol Signal

Opcode	ALUop	Instruction Operation	Funct field	ALU action	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	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Generation of 2-bit **ALUop** signal
 will be discussed later



5. Design of ALU Control Unit (1/2)

- Input: 6-bit **Funct** field and 2-bit **ALUop**

ALUcontrol3 = 0

- Output: 4-bit **ALUcontrol**

ALUcontrol2 = ?

- Find the simplified expressions

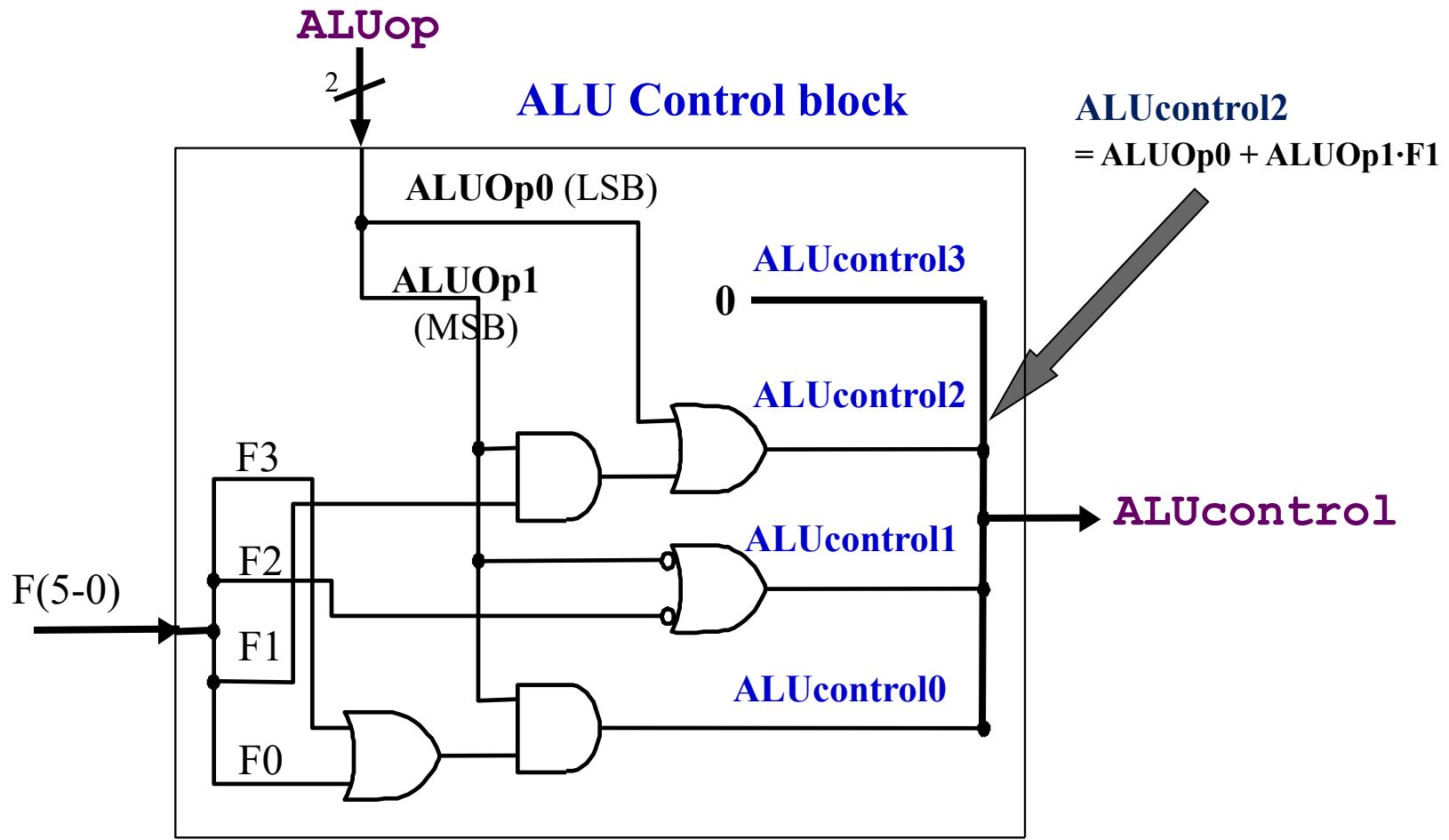
ALUop0 + ALUop1· F1

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	0 X	1	X	X	X	X	X	X	0 1 1 0
add	1	0 X	1 X	0 X	0	0	0	0	0 0 1 0
sub	1	0 X	1 X	0 X	0	0	1	0	0 1 1 0
and	1	0 X	1 X	0 X	0	1	0	0	0 0 0 0
or	1	0 X	1 X	0 X	0	1	0	1	0 0 0 1
slt	1	0 X	1 X	0 X	1	0	1	0	0 1 1 1



5. Design of ALU Control Unit (2/2)

- **Simple combinational logic**



5. Finale: Control Design

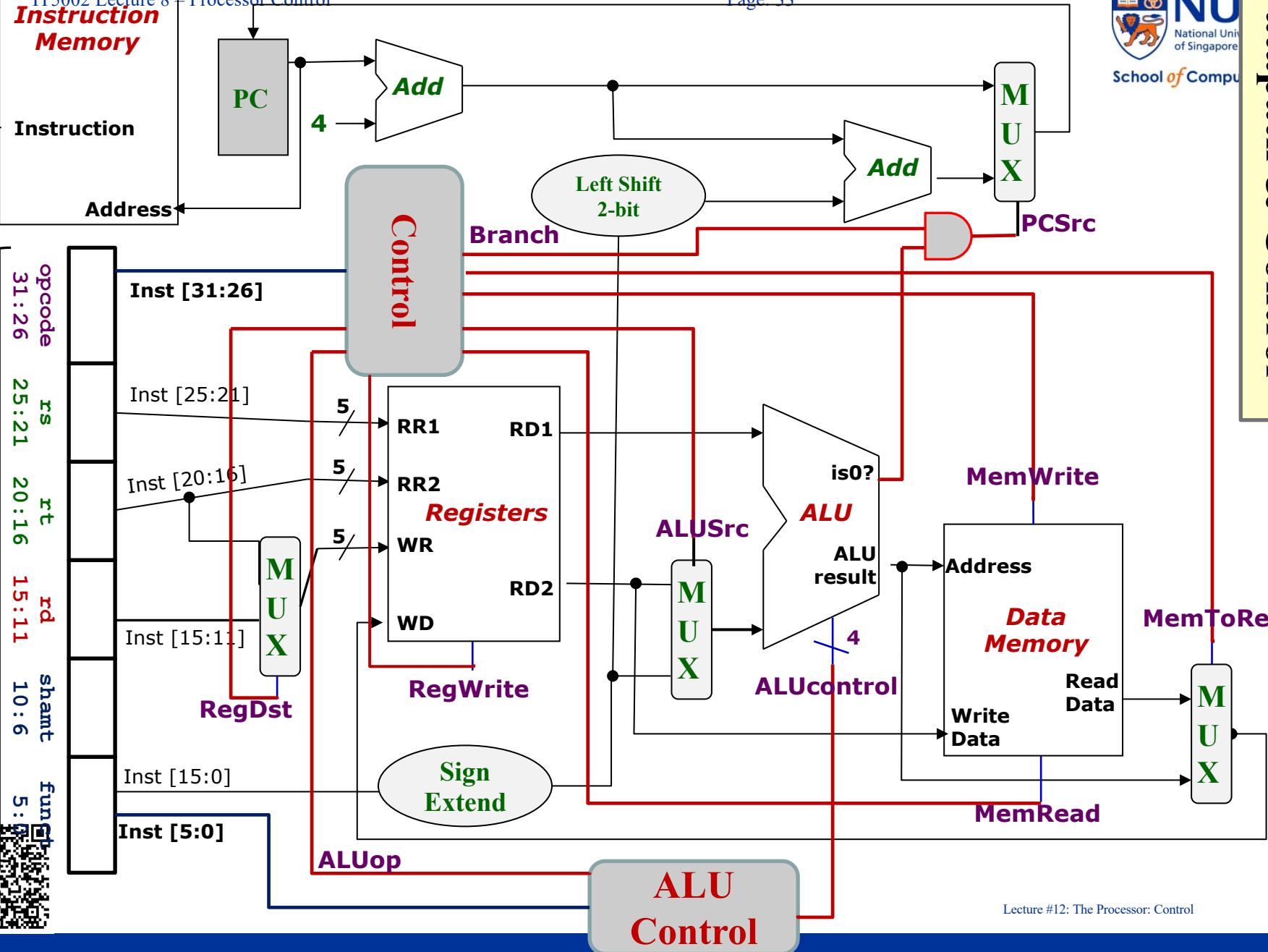
- We have now considered all individual signals and their expected values
 - ➔ Ready to design the controller itself
- Typical digital design steps:
 - Fill in truth table
 - **Input: Opcode**
 - **Output: Various control signals as discussed**
 - Derive simplified expression for each signal



Datapath & Control

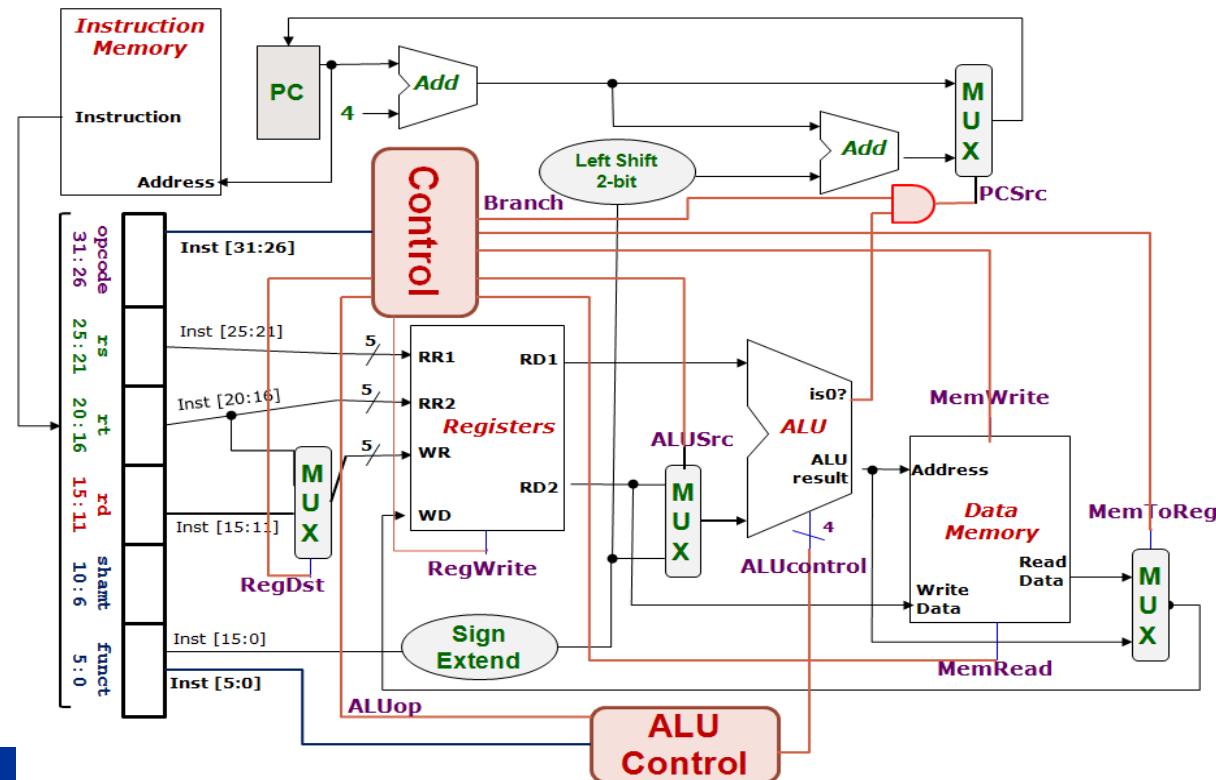
Instruction

Processor Control



5. Control Design: Outputs

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



5. Control Design: Inputs

	Opcode (Op[5:0] == Inst[31:26])						Value in Hexadecimal
	Op5	Op4	Op3	Op2	Op1	Op0	
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

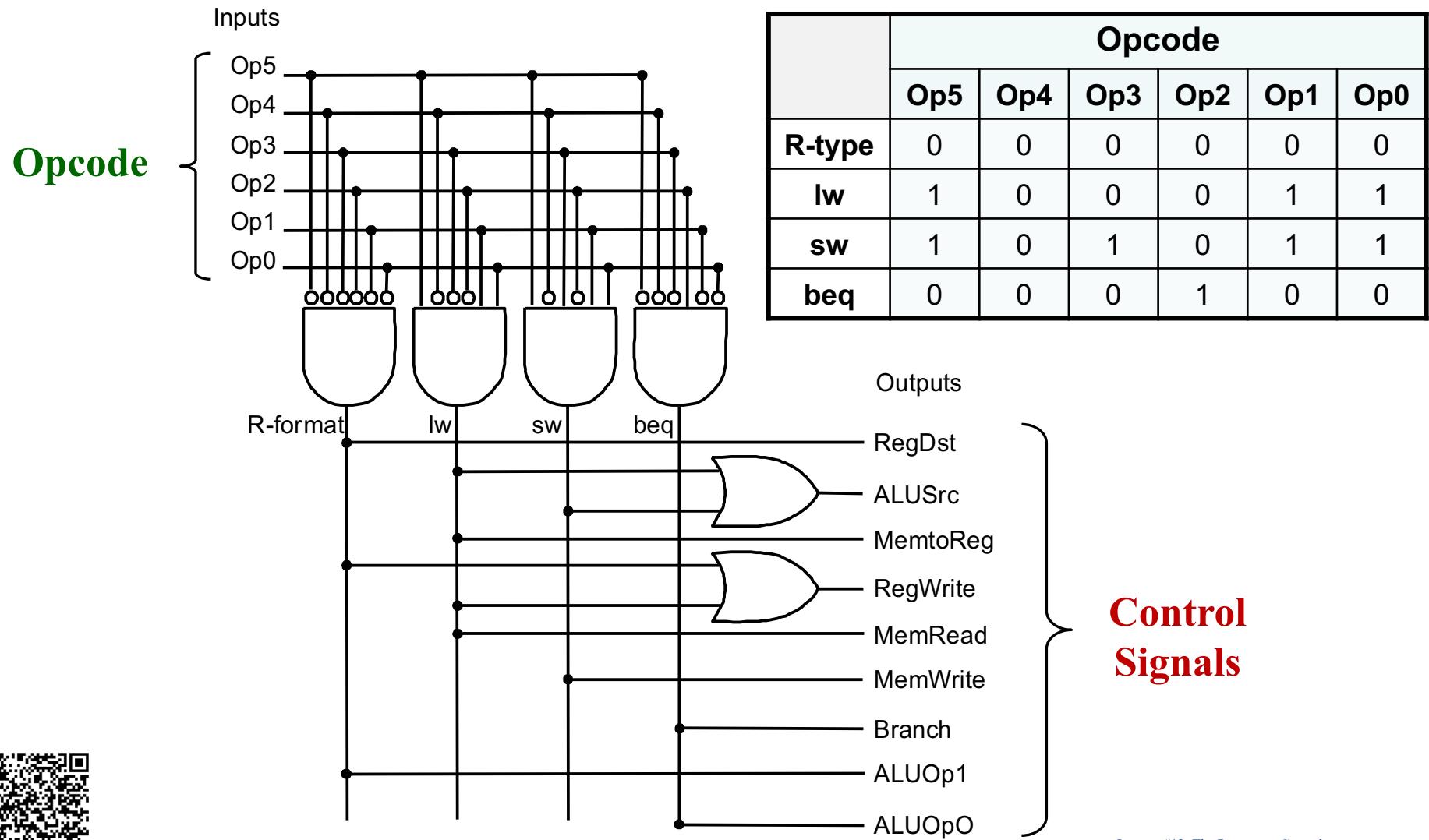
- With the input (opcode) and output (control signals), let's design the circuit



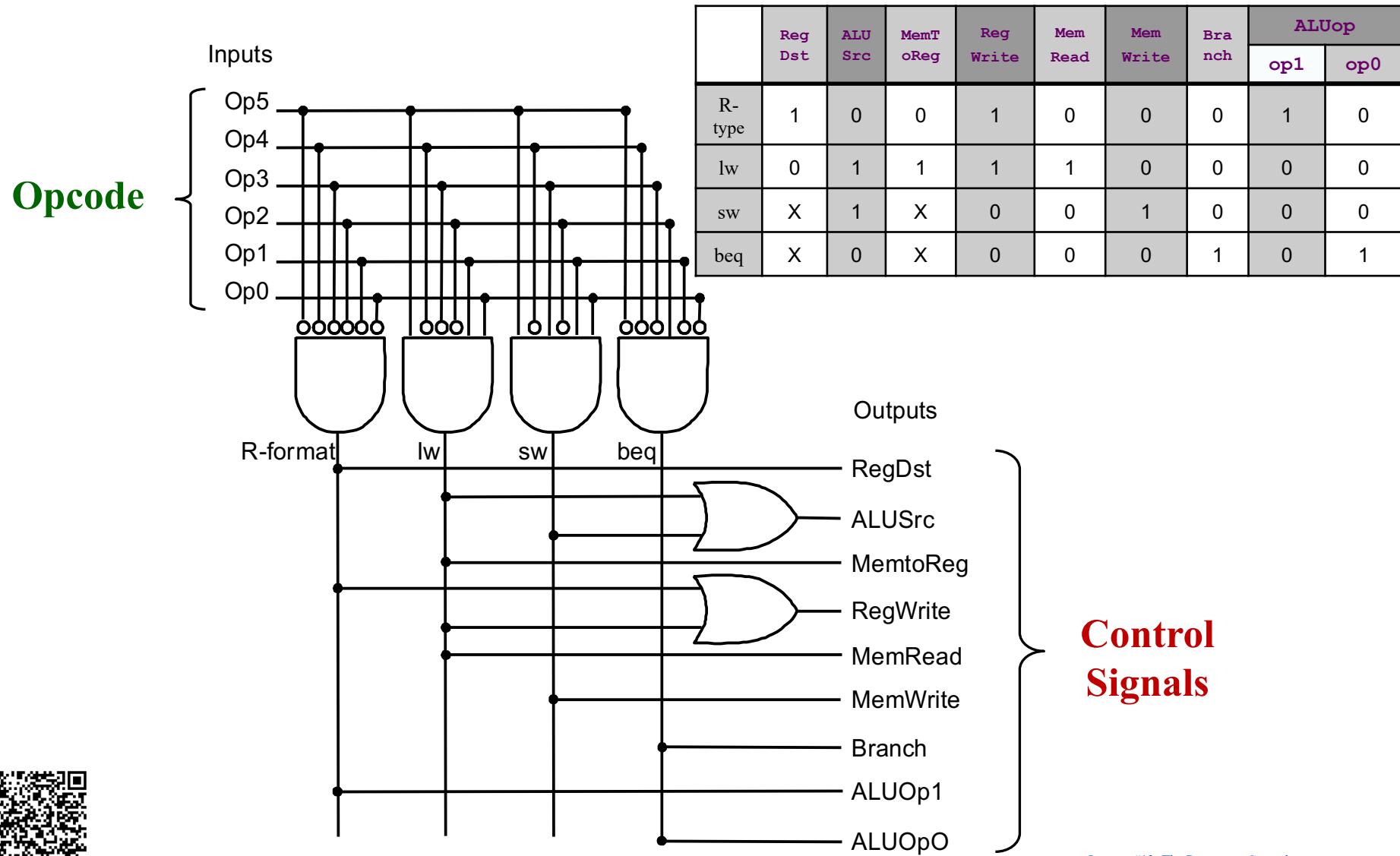




5. Combinational Circuit Implementation

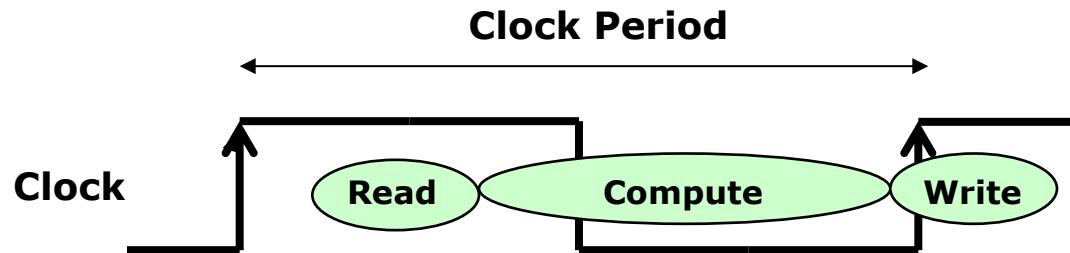


5. Combinational Circuit Implementation



6. Big Picture: Instruction Execution

- Instruction Execution =
 1. Read contents of one or more storage elements (register/memory)
 2. Perform computation through some combinational logic
 3. Write results to one or more storage elements (register/memory)
- All these performed **within a clock period**



Don't want to read a storage element when it is being written.



6. Single Cycle Implementation: Shortcoming

- Calculate cycle time assuming negligible delays: memory (2ns), ALU/adders (2ns), register file access (1ns)

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

- All instructions take as much time as the slowest one (i.e., load)
→ Long cycle time for each instruction



6. Solution #1: Multicycle Implementation

- Break up the instructions into execution steps:
 1. Instruction fetch
 2. Instruction decode and register read
 3. ALU operation
 4. Memory read/write
 5. Register write
- Each execution step **takes one clock cycle**
→ Cycle time is much shorter, i.e., clock frequency is much higher
- Instructions take variable number of clock cycles to complete execution
- Not covered in class:
 - See Section 5.5 of COD if interested



6. Solution #2: Pipelining

- Break up the instructions into execution steps one per clock cycle
- Allow different instructions to be in different execution steps simultaneously
- Covered in a later lecture



Summary

- A very simple implementation of MIPS datapath and control for a subset of its instructions
- Concepts:
 - An instruction executes in a single clock cycle
 - Read storage elements, compute, write to storage elements
 - Datapath is shared among different instruction types using MUXs and control signals
 - Control signals are generated from the machine language encoding of instructions

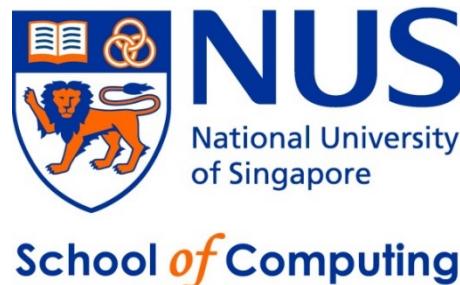


IT5002

Computer Systems and Applications

Pipelining

colintan@nus.edu.sg



Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

IT5002 Midterms

- **IT5002 Midterms:**
 - Saturday 1 October 2.30 pm to 3.30 pm
 - Tutorial Groups 1 – 5, I3 Auditorium
 - Tutorial Groups 6 – 8, LT15
 - Coverage: Lectures 1 to 10



Lecture #14: Pipelining

- 1. Introduction**
- 2. MIPS Pipeline Stages**
- 3. Pipeline Datapath**
- 4. Pipeline Control**
- 5. Pipeline Performance**



1. Introduction: Inspiration

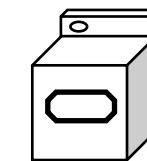
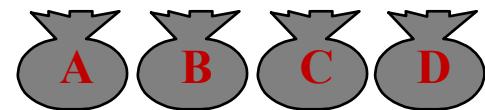
Assembly Line

Simpler station tasks → more cars per hour.
Simple tasks take less time, clock is faster.

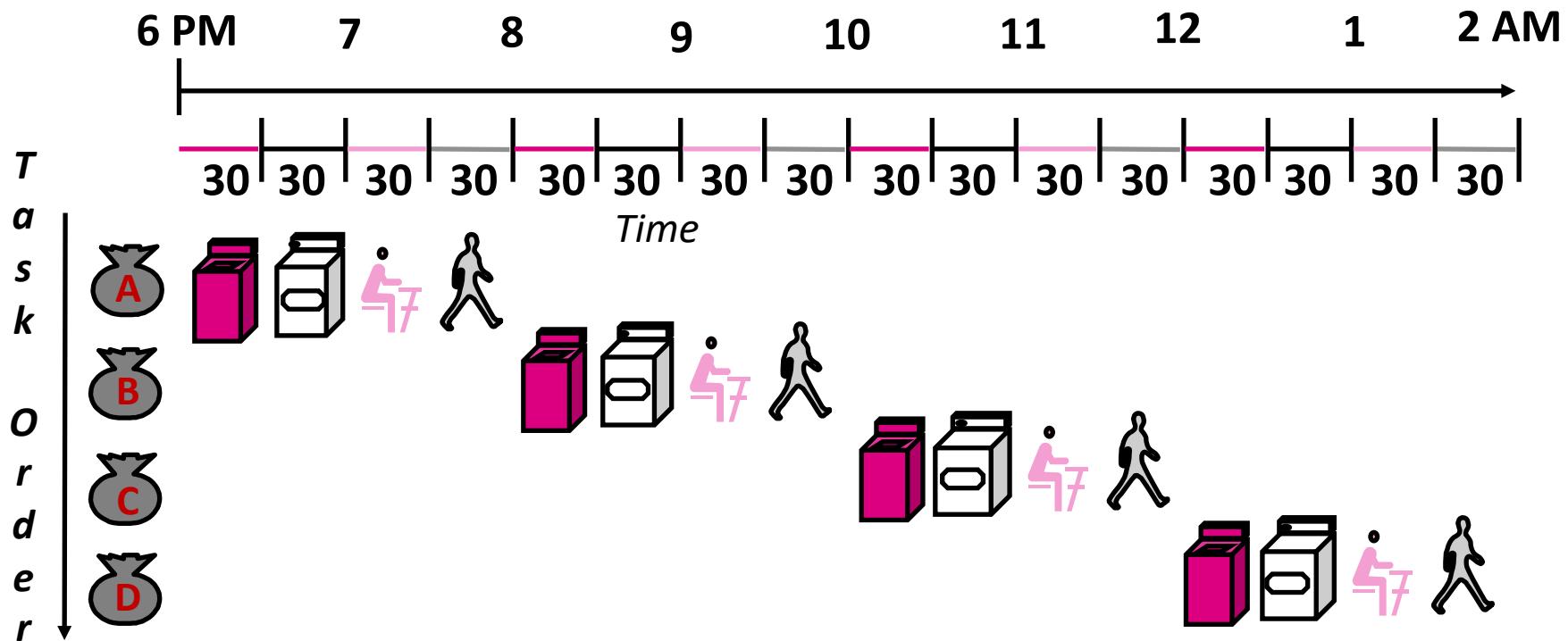


1. Introduction: Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold and stash
- Washer** takes 30 minutes
- Dryer** takes 30 minutes
- “**Folder**” takes 30 minutes
- “**Stasher**” takes 30 minutes to put clothes into drawers



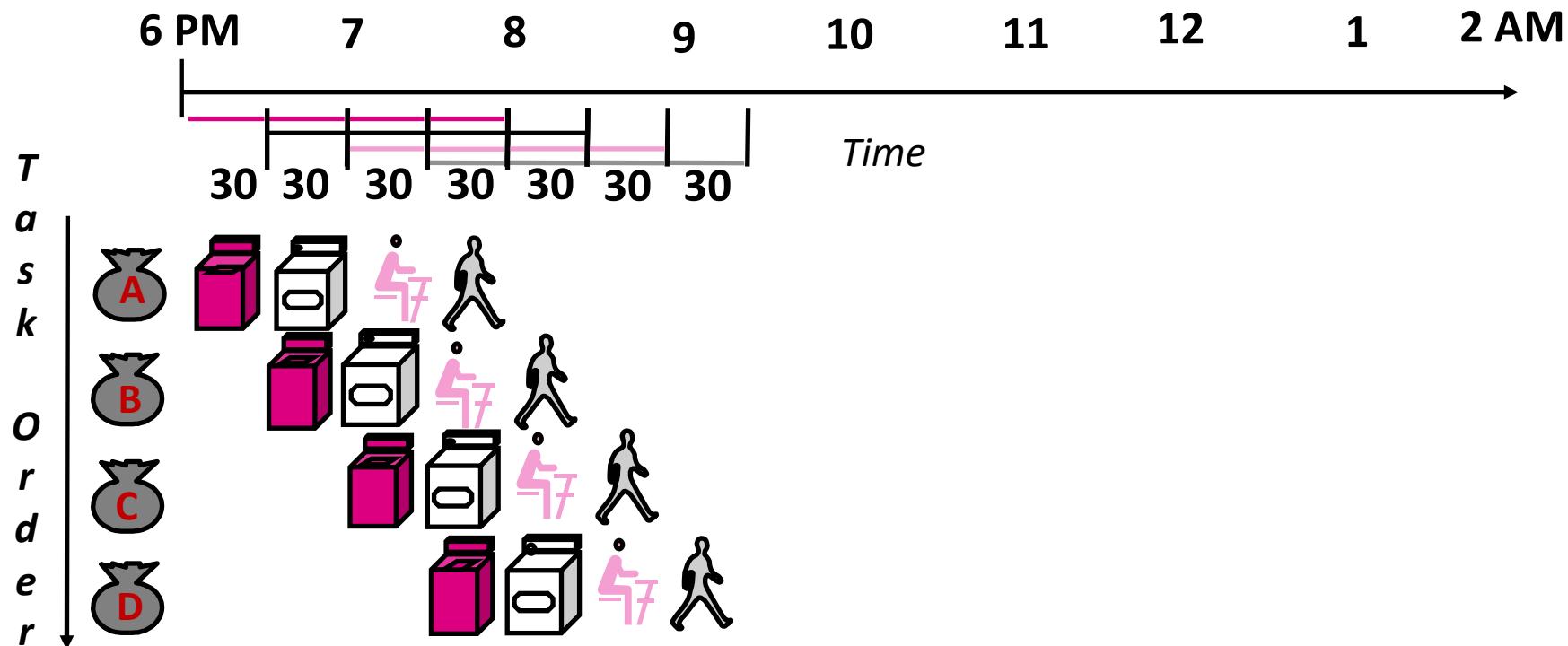
1. Introduction: Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads
- Steady state: 1 load every 2 hours
- If they learned **pipelining**, how long would laundry take?



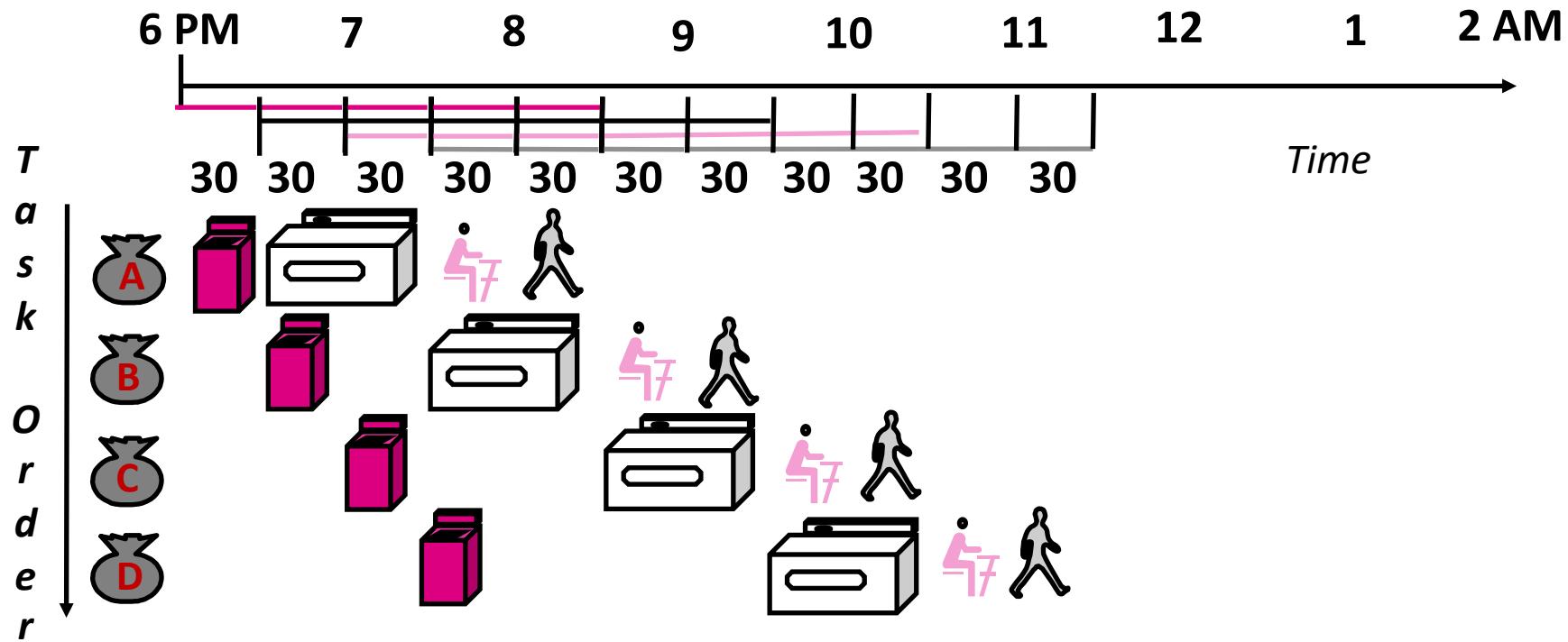
1. Introduction: Pipelined Laundry



- Pipelined laundry takes 3.5 hours for 4 loads!
- Steady state: 1 load every 30 minutes
- Potential speedup = $2 \text{ hr} / 30 \text{ min} = 4$ (no. of stages)
- Time to fill pipeline takes 2 hours → speedup ↓



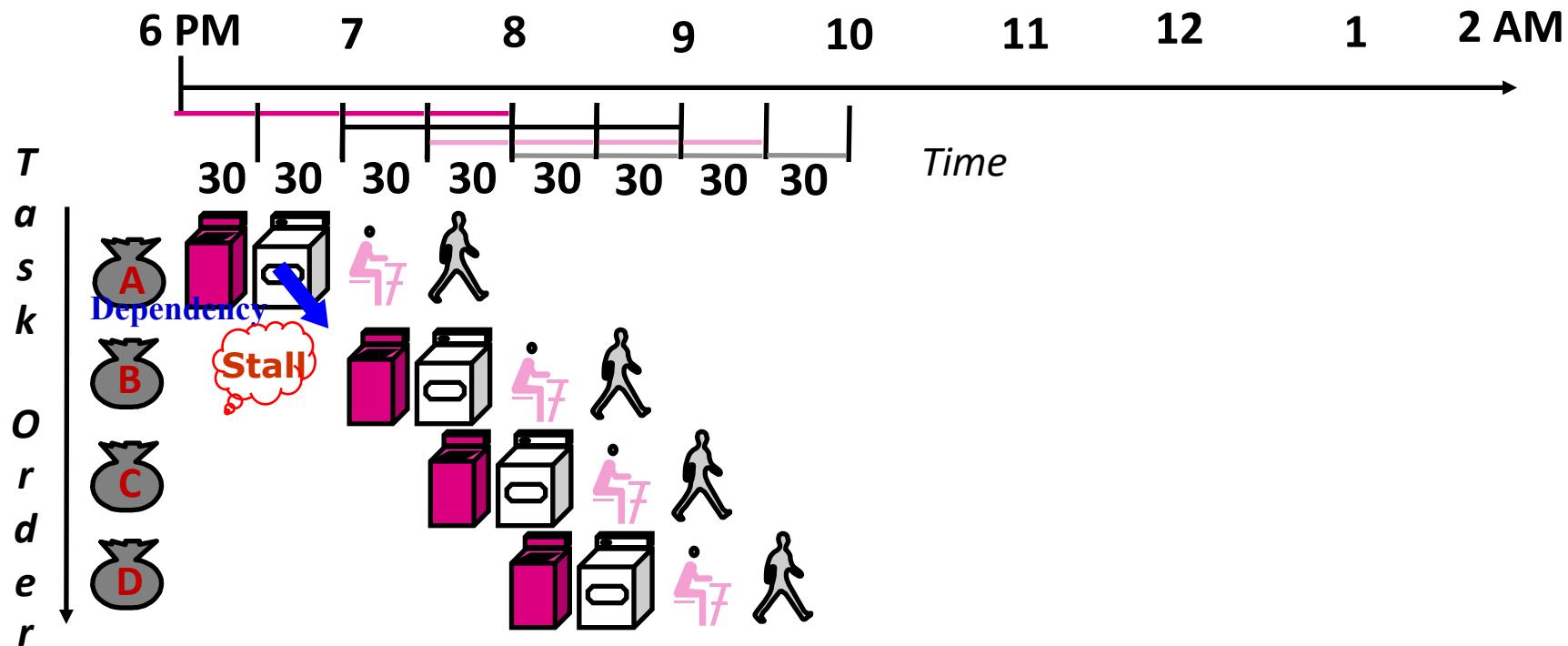
1. Introduction: What If: Slow Dryer



- Pipelined laundry now takes 5.5 hours!
- Steady state: One load every 1 hour (dryer speed)

 **Pipeline rate is limited by the slowest stage**

1. Introduction: What If: Dependency



- Brian is using the laundry for the first time; he wants to see the outcome of one wash + dry cycle first before putting in his clothes

Pipelined laundry now takes 4 hours



1. Introduction: Pipelining Lessons

- Pipelining doesn't help latency of single task:
 - It helps the **throughput** of entire workload
- Multiple tasks operating simultaneously using different resources
- Possible delays:
 - Pipeline rate limited by **slowest** pipeline stage
 - Stall for **dependencies**

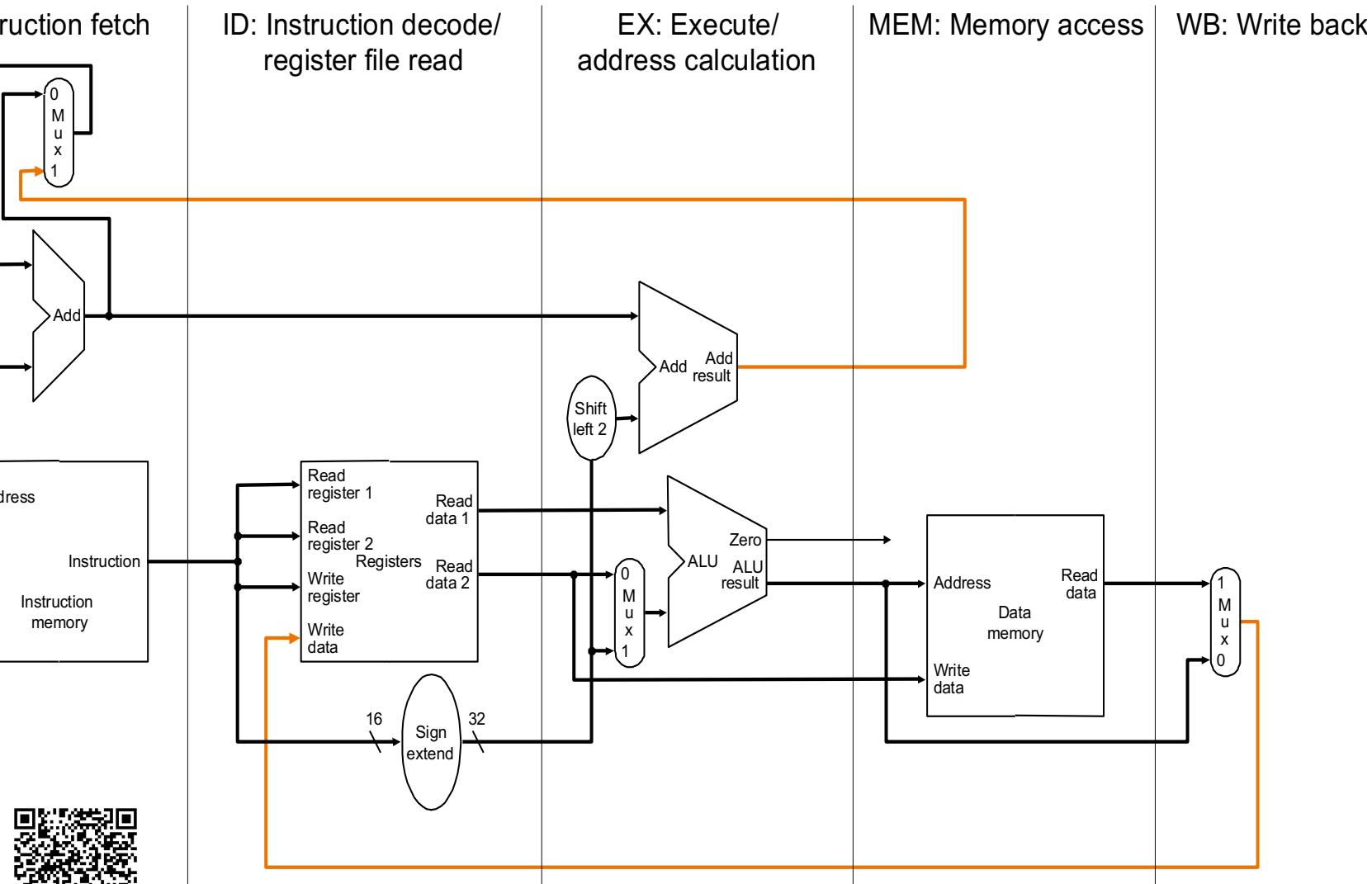


2. MIPS Pipeline Stages (1/2)

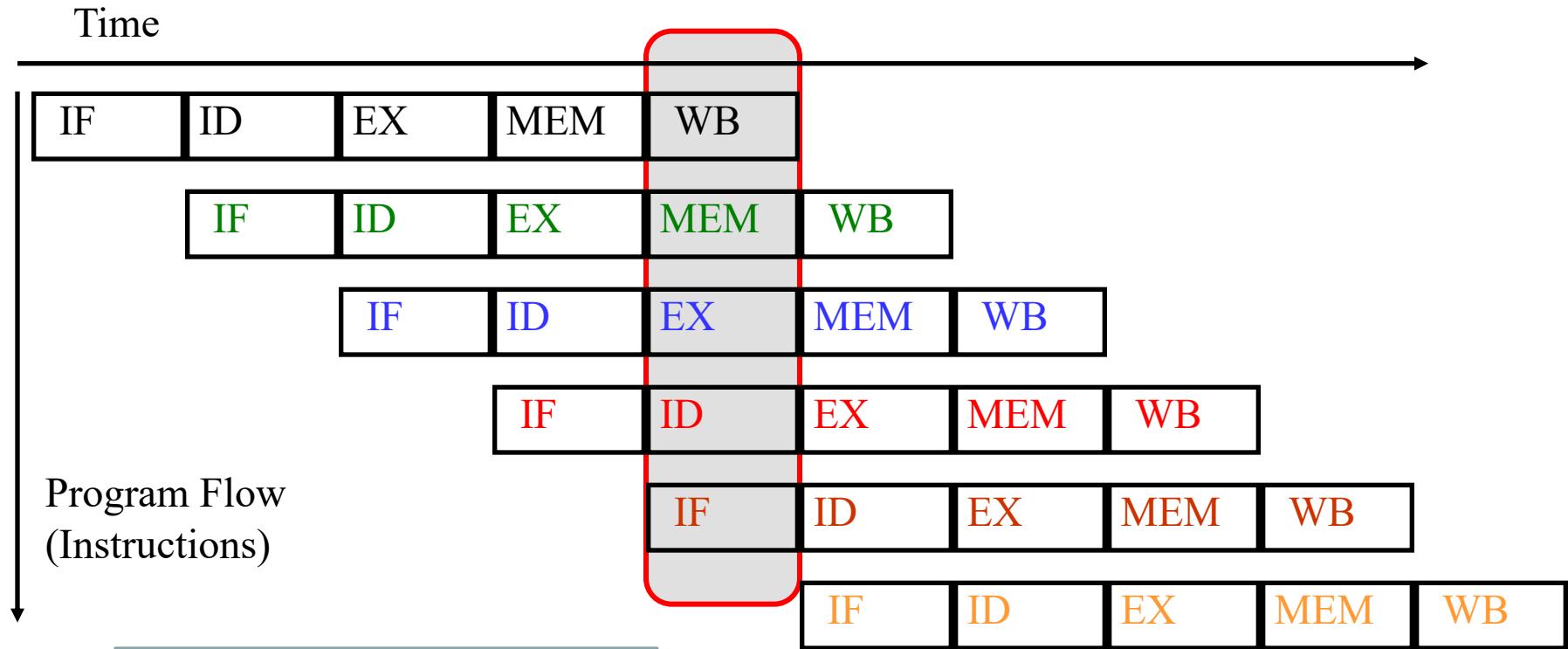
- **Five Execution Stages**
 - **IF**: Instruction Fetch
 - **ID**: Instruction Decode and Register Read
 - **EX**: Execute an operation or calculate an address
 - **MEM**: Access an operand in data memory
 - **WB**: Write back the result into a register
- **Idea:**
 - **Each execution stage takes 1 clock cycle**
 - General flow of data is from one stage to the next
- **Exceptions:**
 - Update of PC and write back of register file – more about this later...



2. MIPS Pipeline Stages (2/2)



2. Pipelined Execution: Illustration



Several instructions
are in the pipeline
simultaneously!



3. MIPS Pipeline: Datapath (1/3)

- Single-cycle implementation:
 - Update all state elements (**PC**, register file, data memory) at the end of a clock cycle
- Pipelined implementation:
 - One cycle per pipeline stage
 - Data required for each stage needs to be stored separately (why?)

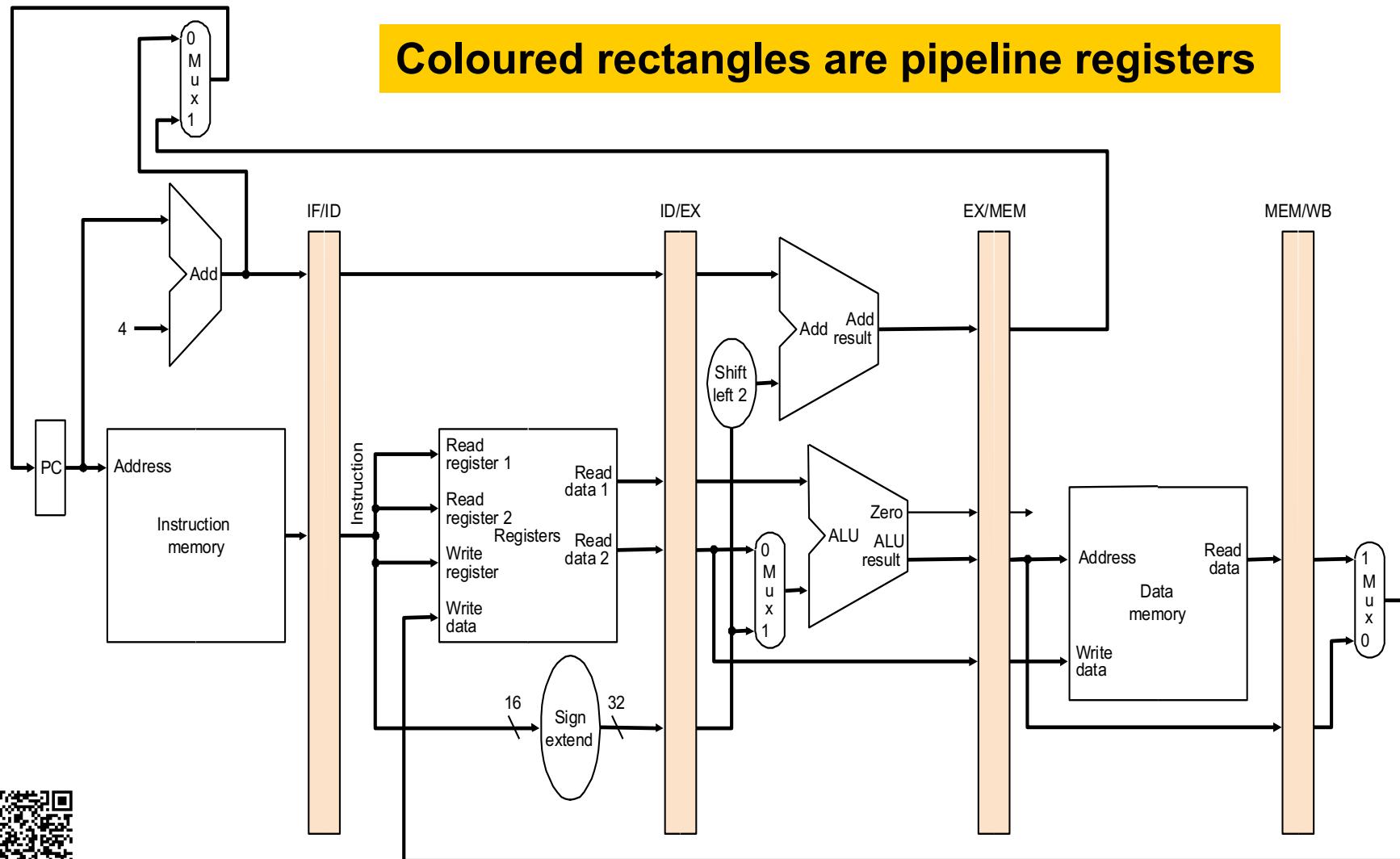


3. MIPS Pipeline: Datapath (2/3)

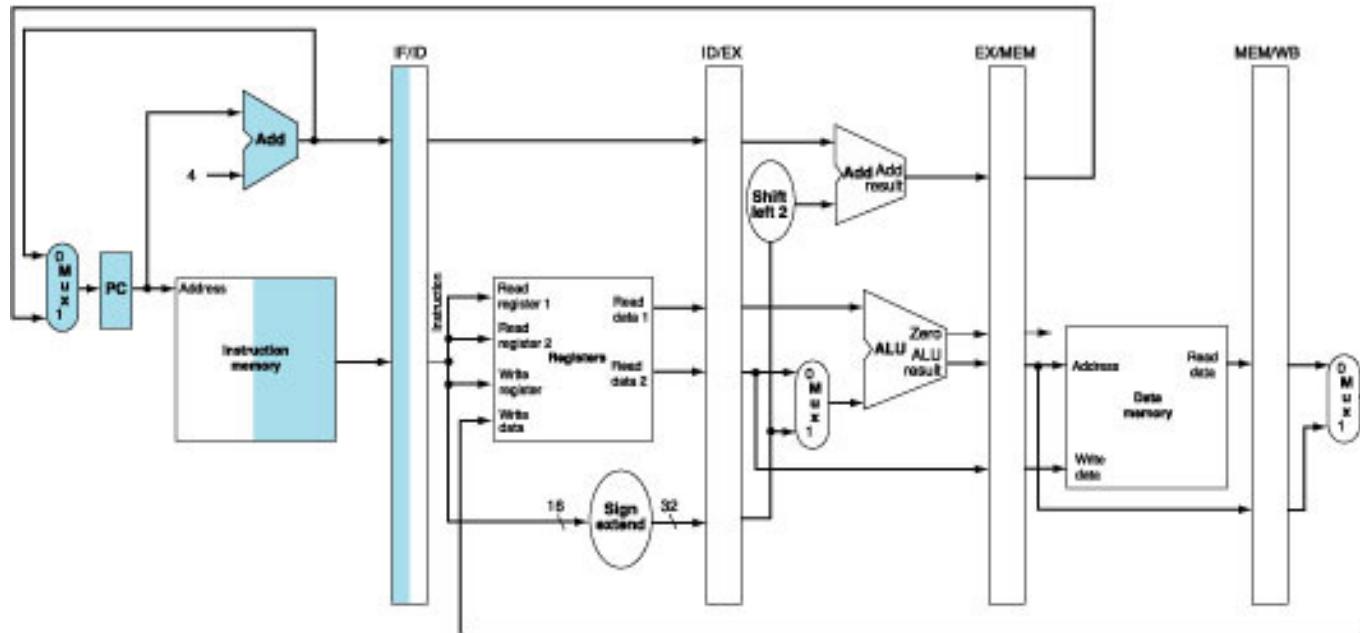
- Data used by **subsequent instructions**:
 - Store in programmer-visible state elements: **PC**, register file and memory
- Data used by **same instruction** in later pipeline stages:
 - Additional registers in datapath called **pipeline registers**
 - **IF/ID**: register between **IF** and **ID**
 - **ID/EX**: register between **ID** and **EX**
 - **EX/MEM**: register between **EX** and **MEM**
 - **MEM/WB**: register between **MEM** and **WB**
- Why no register at the end of **WB** stage?



3. MIPS Pipeline: Datapath (3/3)



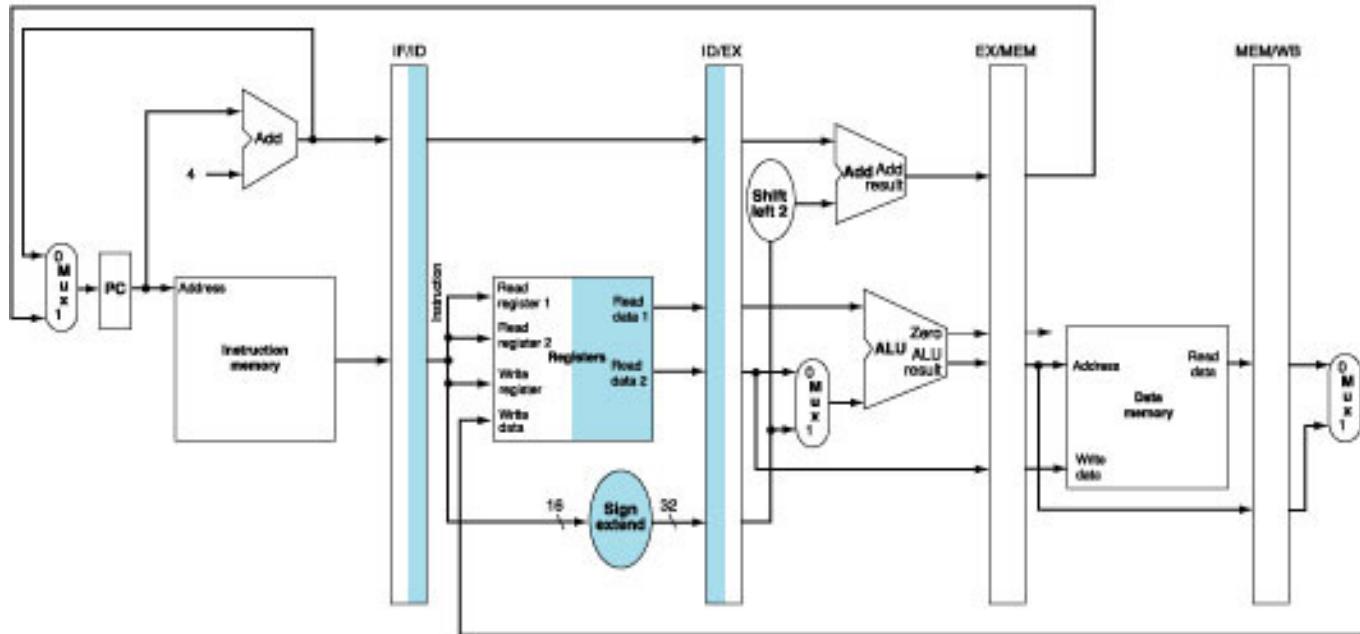
3. Pipeline Datapath: IF Stage



- At the end of a cycle, **IF/ID** receives (stores):
 - Instruction read from InstructionMemory[PC]
 - PC + 4
- PC + 4
 - Also connected to one of the MUX's inputs (another coming later)



3. Pipeline Datapath: ID Stage



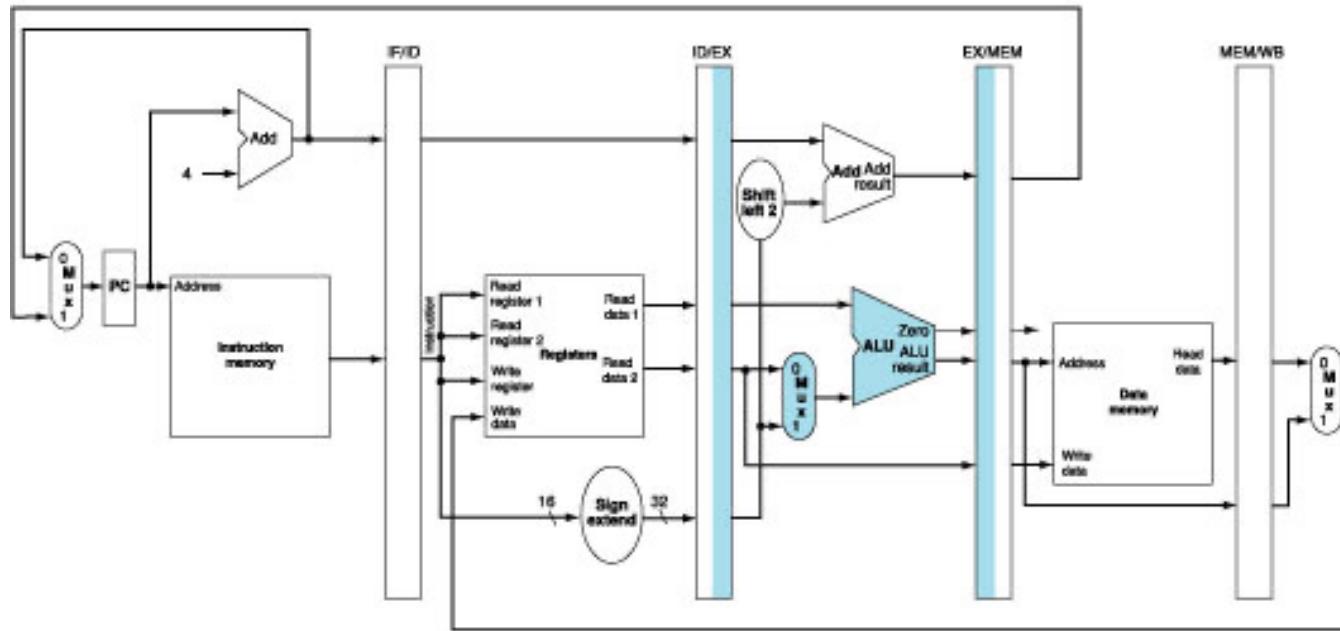
**At the beginning of a cycle
IF/ID register supplies:**

- ❖ Register numbers for reading two registers
- ❖ 16-bit offset to be sign-extended to 32-bit

**At the end of a cycle
ID/EX receives:**

- ❖ Data values read from register file
- ❖ 32-bit immediate value
- ❖ PC + 4

3. Pipeline Datapath: EX Stage



At the beginning of a cycle
ID/EX register supplies:

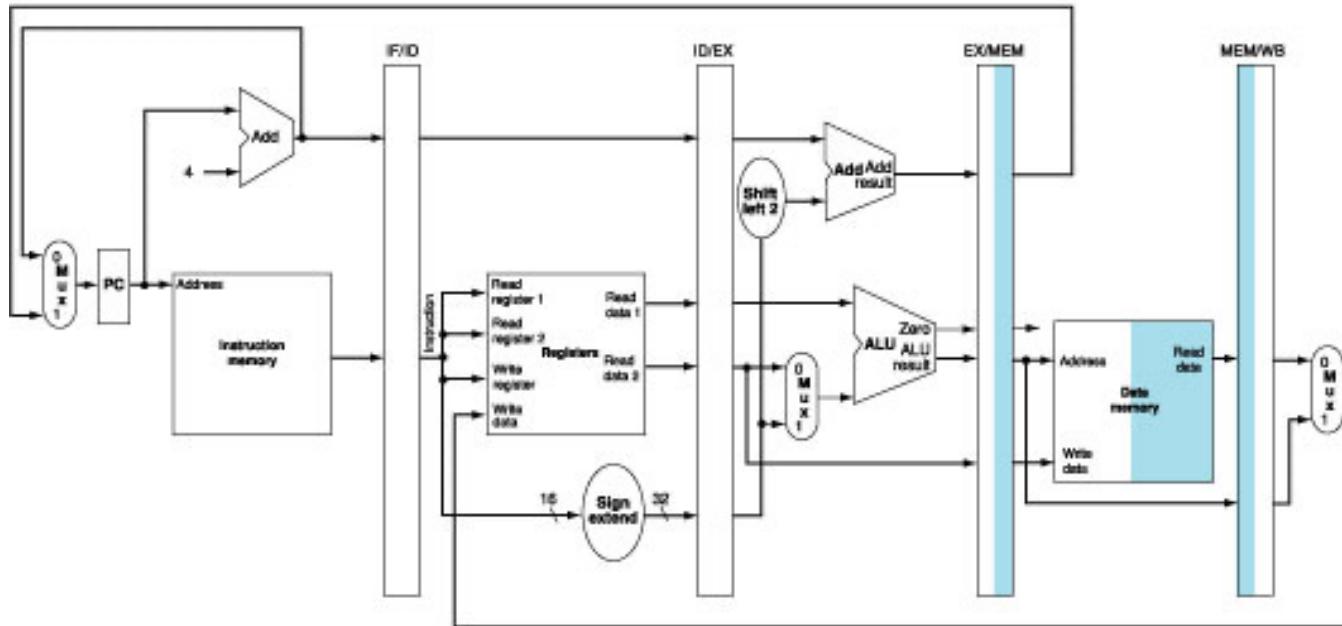
- ❖ Data values read from register file
- ❖ 32-bit immediate value
- ❖ **PC + 4**

At the end of a cycle
EX/MEM receives:

- ❖ $(PC + 4) + (\text{Immediate} \times 4)$
- ❖ ALU result
- ❖ **isZero?** signal
- ❖ Data Read 2 from register file



3. Pipeline Datapath: MEM Stage



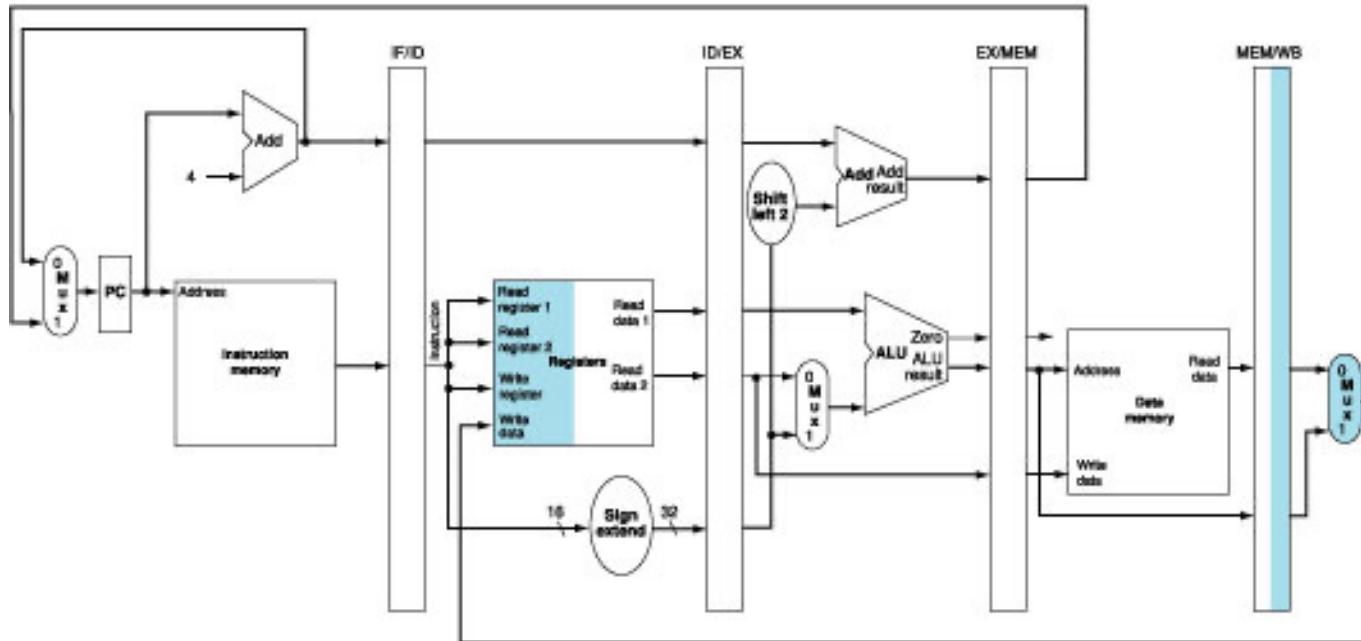
At the beginning of a cycle
EX/MEM register supplies:

- ❖ $(PC + 4) + (\text{Immediate} \times 4)$
- ❖ ALU result
- ❖ **isZero?** signal
- ❖ Data Read 2 from register file

At the end of a cycle
MEM/WB receives:

- ❖ ALU result
- ❖ Memory read data

3. Pipeline Datapath: WB Stage



At the beginning of a cycle
MEM/WB register supplies:

- ❖ ALU result
- ❖ Memory read data

At the end of a cycle

- ❖ Result is written back to register file (if applicable)
- ❖ **There is a bug here.....**

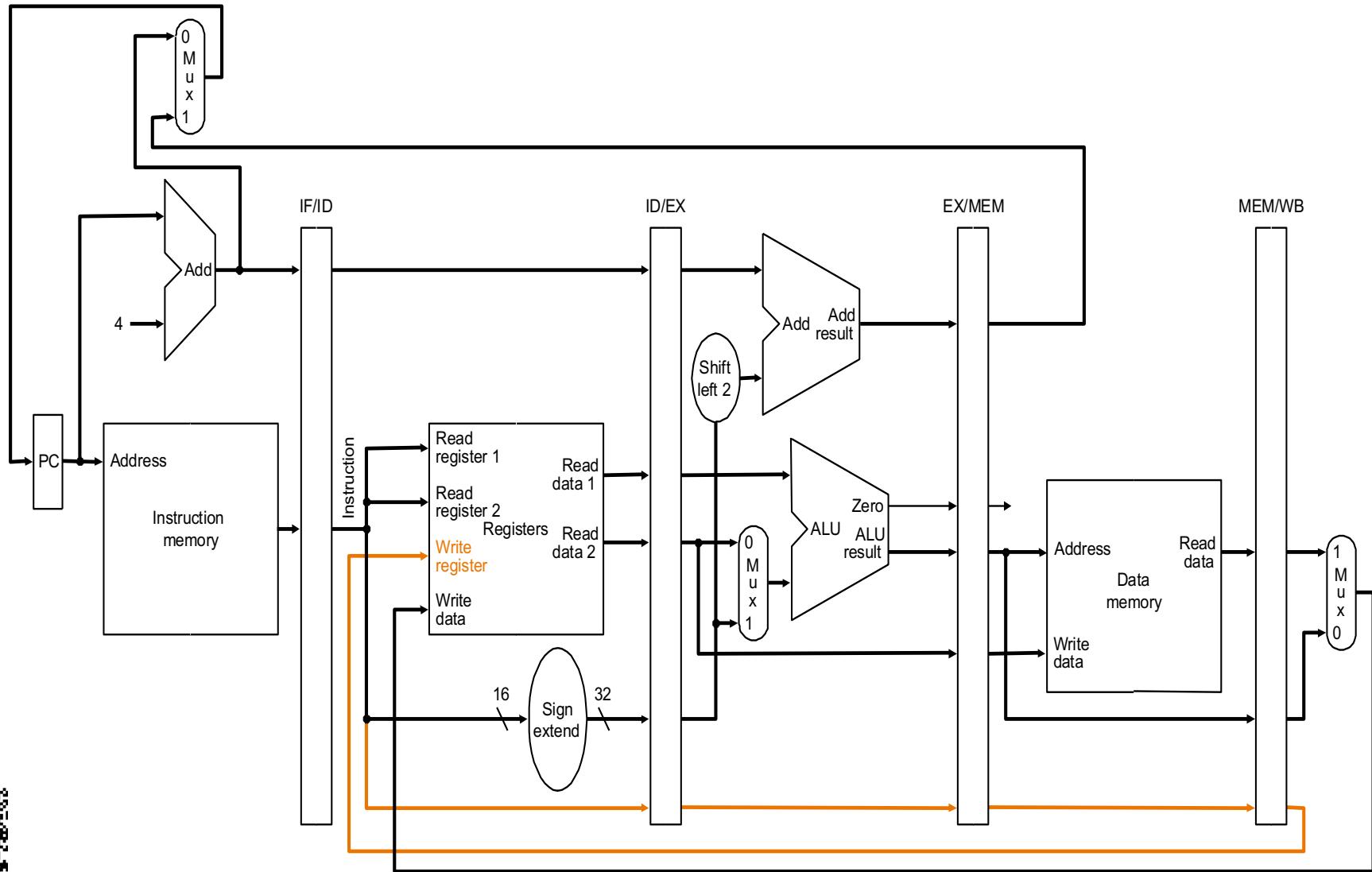


3. Corrected Datapath (1/2)

- Observe the “Write register” number
 - Supplied by the **IF/ID** pipeline register
 - ➔ It is NOT the correct write register for the instruction now in **WB** stage!
- **Solution:**
 - Pass “Write register” number from **ID/EX** through **EX/MEM** to **MEM/WB** pipeline register for use in **WB** stage
 - i.e. let the "Write register" number follows the instruction through the pipeline until it is needed in WB stage

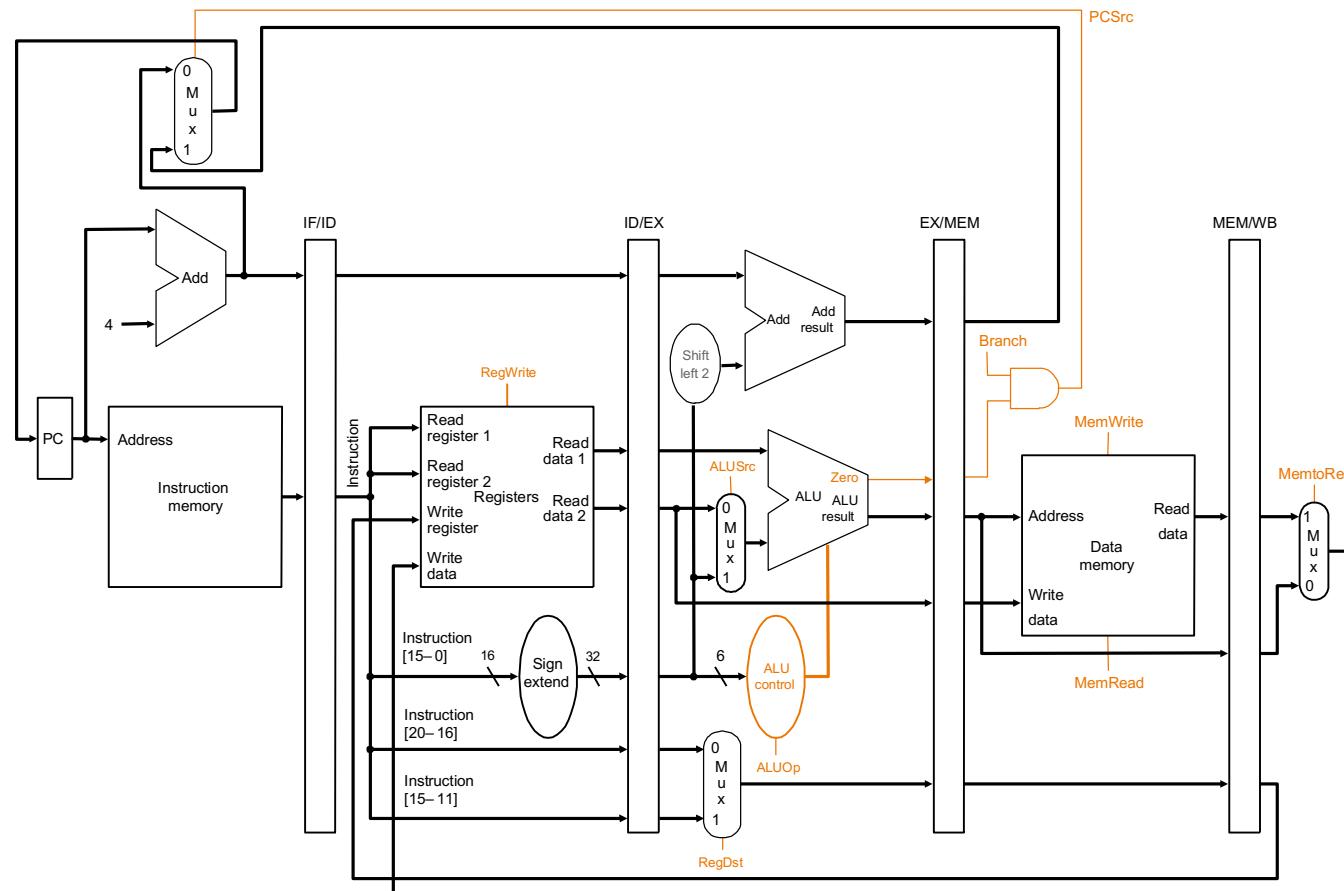


3. Corrected Datapath (2/2)

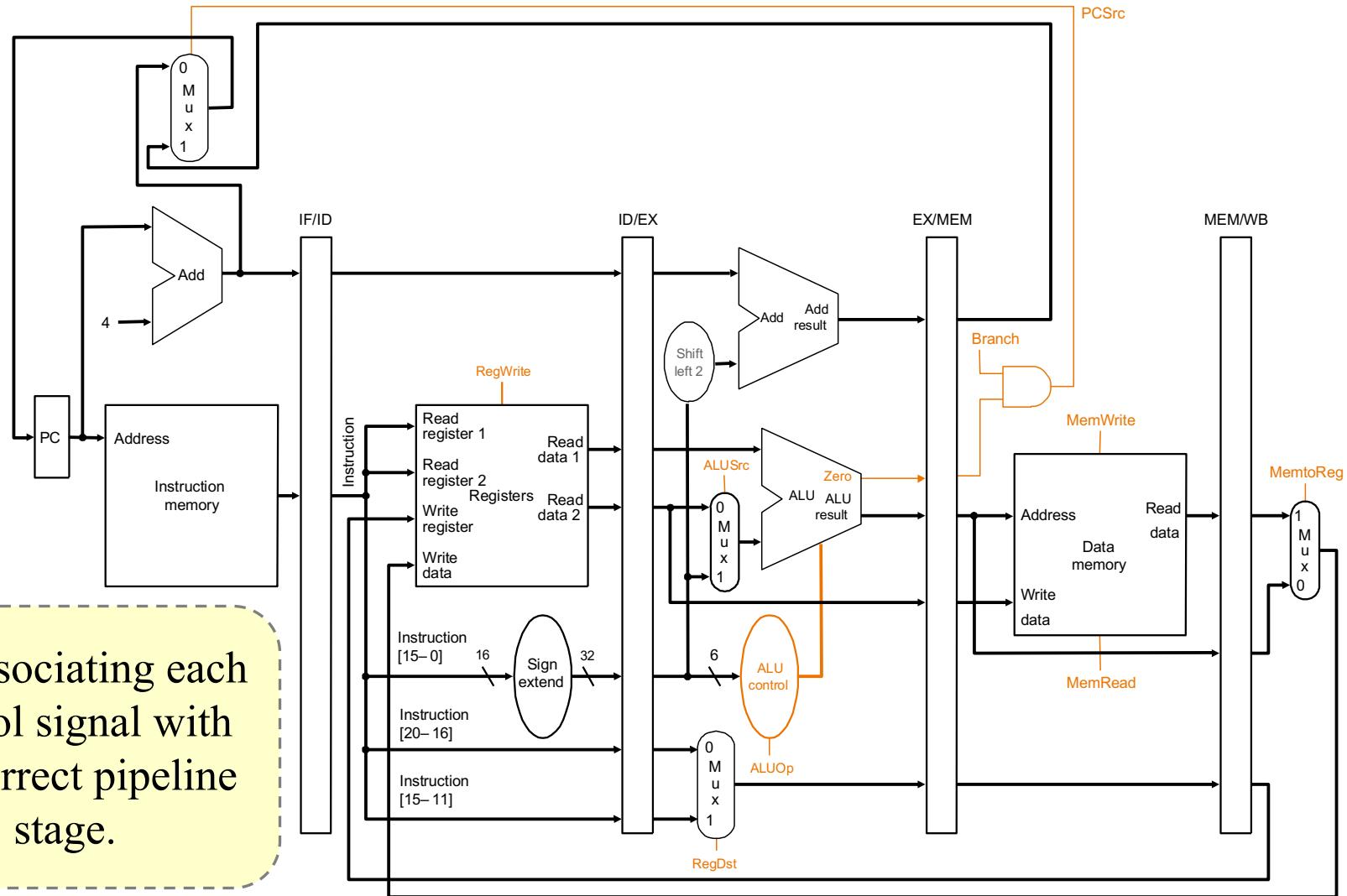


4. Pipeline Control: Main Idea

- Same control signals as single-cycle datapath
- Difference:** Each control signal belongs to a particular pipeline stage



4. Pipeline Control: Try it!



4. Pipeline Control: Grouping

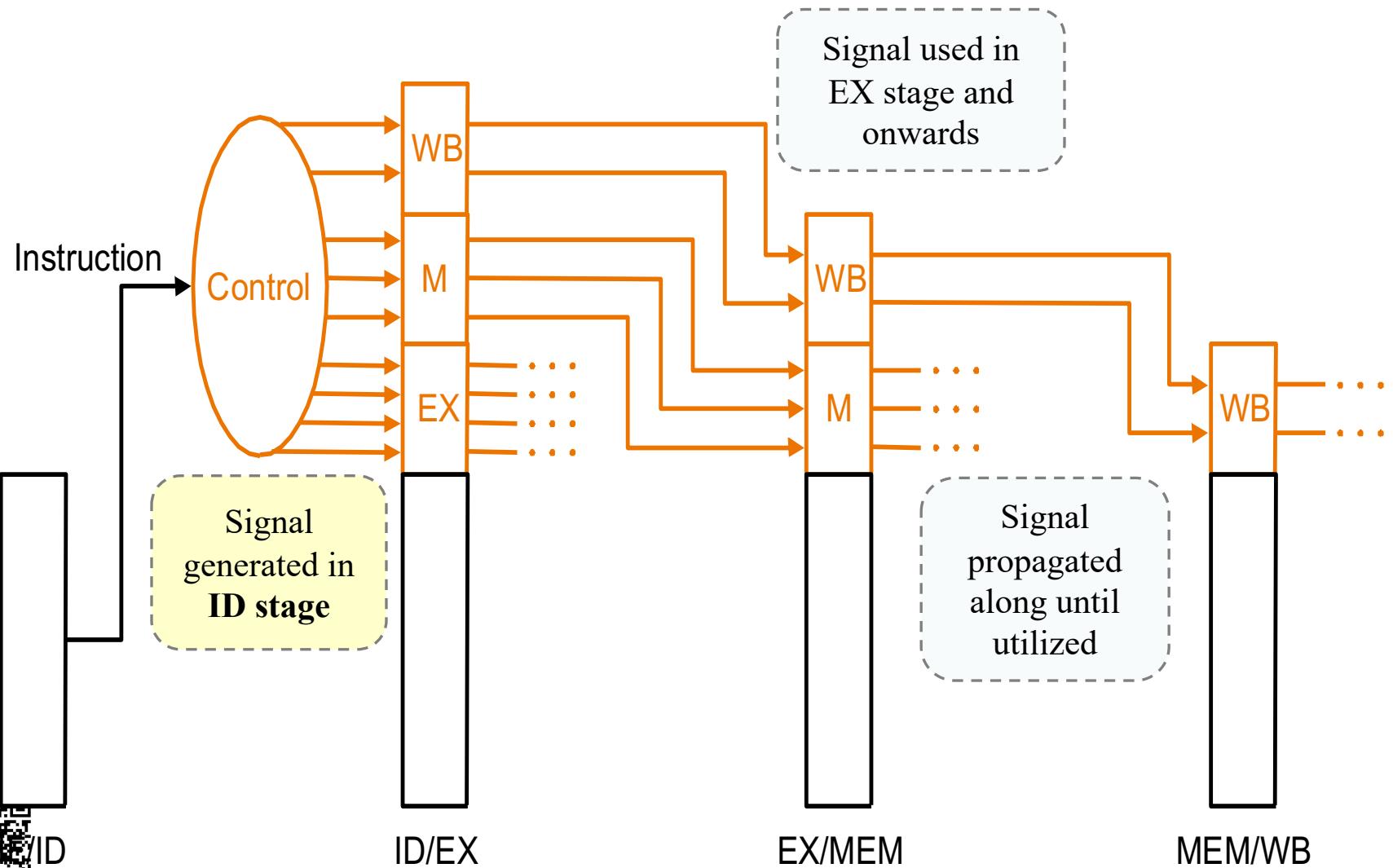
- Group control signals according to pipeline stage

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

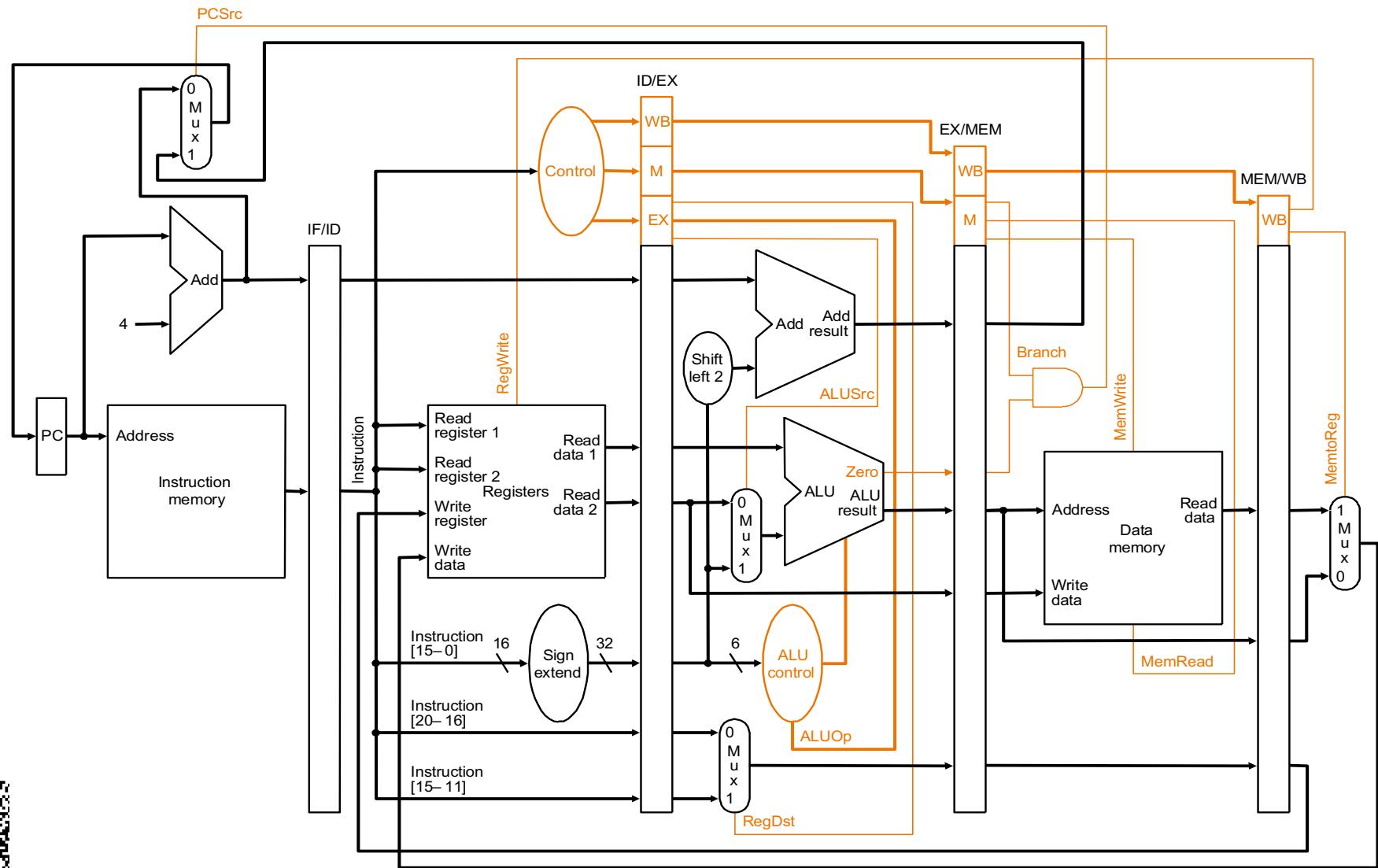
	EX Stage				MEM Stage				WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write	
			op1	op0						
R-type	1	0	1	0	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1	1
sw	X	1	0	0	0	1	0	X	0	0
beq	X	0	0	1	0	0	1	X	0	0



4. Pipeline Control: Implementation

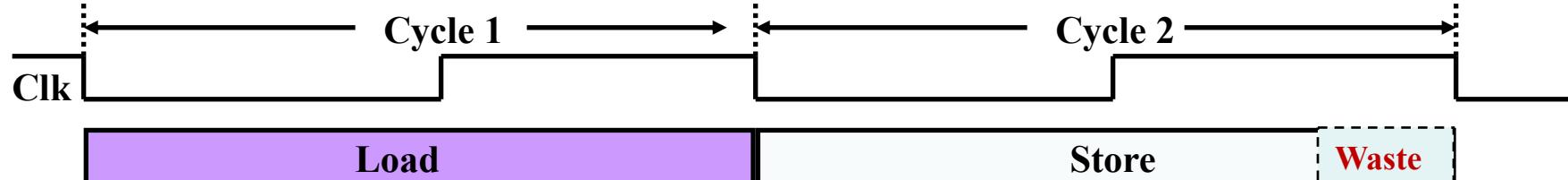


4. Pipeline Control: Datapath and Control

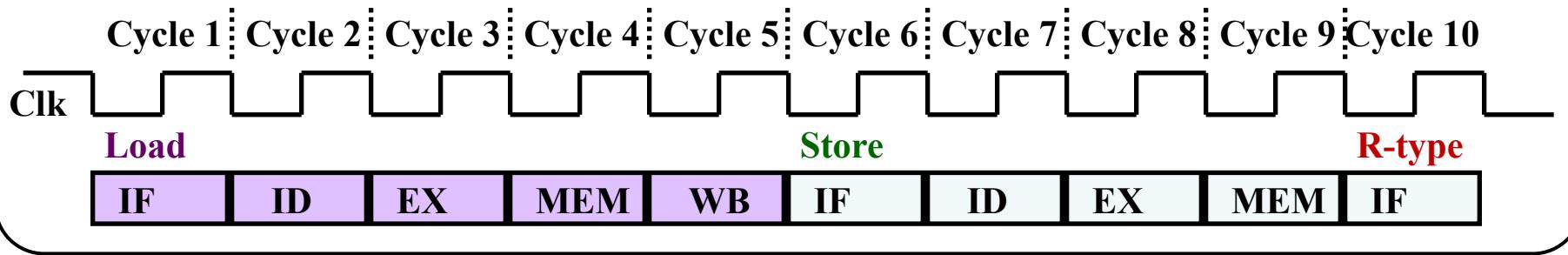


5. Different Implementations

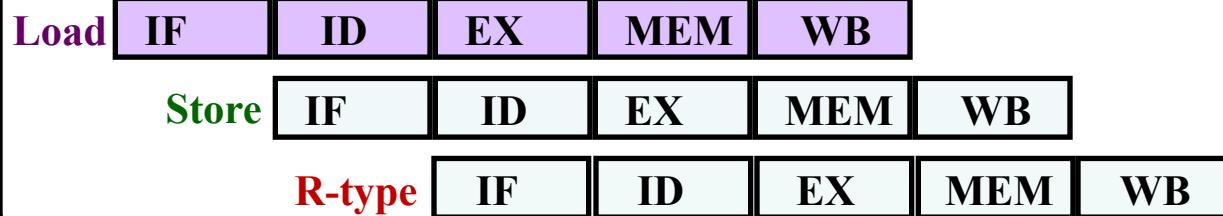
Single-Cycle



Multi-Cycle



Pipeline



5. Single Cycle Processor: Performance

- **Cycle time:**
 - $CT_{seq} = \sum_{k=1}^N T_k$
 - T_k = Time for operation in stage k
 - N = Number of stages
- **Total Execution Time for I instructions:**
 - $Time_{seq} = \text{Cycles} \times \text{CycleTime}$
 $= I \times CT_{seq} = I \times \sum_{k=1}^N T_k$



5. Single Cycle Processor: Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

- **Cycle Time**
 - Choose the longest total time = **8ns**
- To execute **100 instructions**:
 - **$100 \times 8\text{ns} = 800\text{ns}$**



5. Multi-Cycle Processor: Performance

- **Cycle time:**
 - $CT_{multi} = \max(T_k)$
 - $\max(T_k) =$ longest stage duration among the N stages
- **Total Execution Time for I instructions:**
 - $Time_{multi} = \text{Cycles} \times \text{CycleTime}$
 $= I \times \text{Average CPI} \times CT_{multi}$
 - Average CPI is needed because each instruction takes different number of cycles to finish



5. Multi-Cycle Processor: Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

- Cycle Time
 - Choose the longest stage time = **2ns**
- To execute **100 instructions**, with a given **average CPI of 4.6**
 - $100 \times 4.6 \times 2\text{ns} = 920\text{ns}$



5. Pipeline Processor: Performance

- **Cycle Time:**

- $CT_{pipeline} = \max(T_k) + T_d$
- $\max(T_k)$ = longest time among the N stages
- T_d = Overhead for pipelining, e.g. pipeline register

- **Cycles needed for I instructions:**

- $I + N - 1$
- $N - 1$ is the cycles wasted in filling up the pipeline

- **Total Time needed for I instructions :**

- $$\begin{aligned} Time_{pipeline} &= Cycle \times CT_{pipeline} \\ &= (I + N - 1) \times (\max(T_k) + T_d) \end{aligned}$$



5. Pipeline Processor: Example

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

- Cycle Time
 - assume pipeline register delay of **0.5ns**
 - longest stage time + overhead = **$2 + 0.5 = 2.5\text{ns}$**
- To execute **100 instructions**:
 - **$(100 + 5 - 1) \times 2.5\text{ns} = 260\text{ns}$**



5. Pipeline Processor: Ideal Speedup (1/2)

- **Assumptions for ideal case:**
 - Every stage takes the same amount of time:
 $\rightarrow \sum_{k=1}^N T_k = N \times T_k$
 - No pipeline overhead $\rightarrow T_d = 0$
 - Number of instructions I , is much larger than number of stages, N
- **Note: The above also shows how pipeline processor loses performance**



5. Pipeline Processor: Ideal Speedup (2/2)

$$\blacksquare \text{Speedup}_{\text{pipeline}} = \frac{\text{Time}_{\text{seq}}}{\text{Time}_{\text{pipeline}}}$$

$$= \frac{I \times \sum_{k=1}^N T_k}{(I+N-1) \times (\max(T_k) + T_d)}$$

$$= \frac{I \times N \times T_k}{(I+N-1) \times T_k}$$

$$\approx \frac{I \times N \times T_k}{I \times T_k}$$

$$\approx N$$

Conclusion:

Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages



Review Question

- Given this code:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

- a) 4 cycles
- b) $4/(100 \times 10^6) = 40 \text{ ns}$
- c) $4 + 4 = 8 \text{ cycles}$
- d) $8/(500 \times 10^6) = 16 \text{ ns}$

- a) How many cycles will it take to execute the code on a single-cycle datapath?
- b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock?
- c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline?
- d) How long will it take to execute the code on a 5-stage MIPS pipeline, assuming a 500 MHz clock?

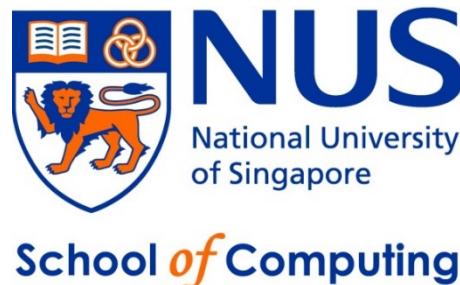


IT5002

Computer Systems and Applications

Caches

colintan@nus.edu.sg



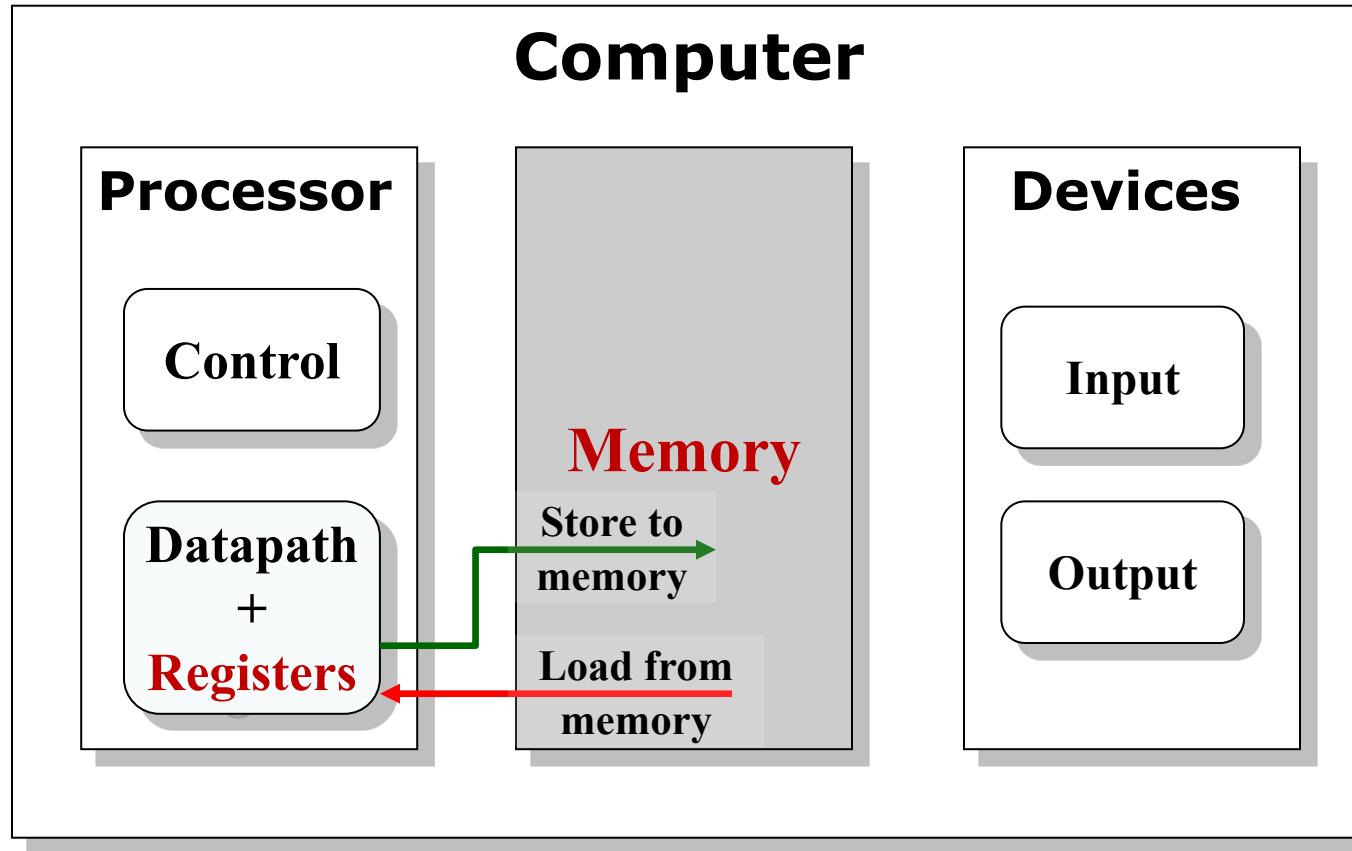
Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at**
<https://sets.netlify.app/module/61597486a7805d9fb1b4accd>



OR scan this QR code (may be obscured on some slides)

1. Data Transfer: The Big Picture



Registers are in the datapath of the processor. If operands are in memory we have to **load** them to processor (registers), operate on them, and **store** them back to memory.

1. Memory Technology: 1950s



**1948: Maurice Wilkes examining EDSAC's delay line memory tubes
16-tubes each storing 32 17-bit words**

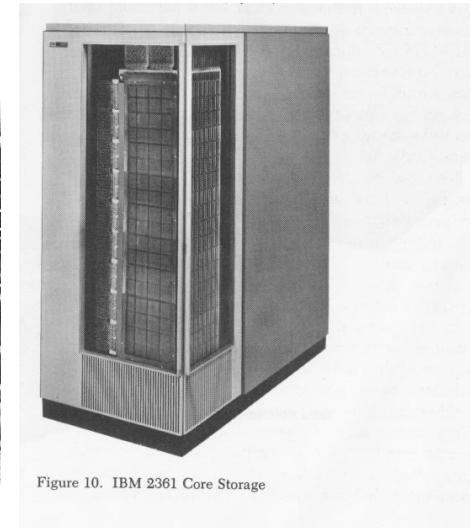
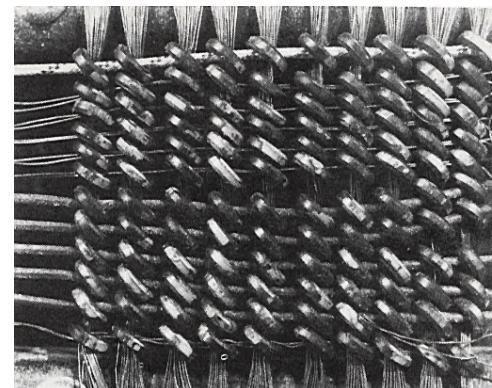


Figure 10. IBM 2361 Core Storage



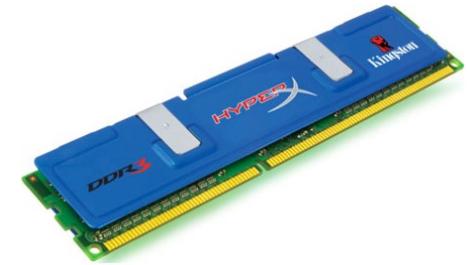
Maurice Wilkes: 2005

1952: IBM 2361 16KB magnetic core memory

1. Memory Technology Today: DRAM

■ DDR SDRAM

- Double Data Rate
 - Synchronous Dynamic RAM
- The dominant memory technology in PC market
- Delivers memory on the positive and negative edge of a clock (double rate)
- Generations:
 - DDR (MemClkFreq x 2(double rate) x 8 words)
 - DDR2 (MemClkFreq x 2(multiplier) x 2 x 8 words)
 - DDR3 (MemClkFreq x 4(multiplier) x 2 x 8 words)
 - DDR4 (Lower power consumption, higher bandwidth)

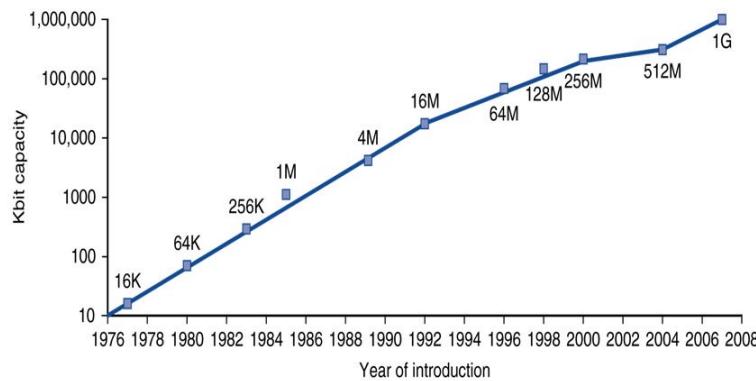


1. DRAM Capacity Growth

Growth of Capacity per DRAM Chip

❖ DRAM capacity quadrupled almost every 3 years

◊ 60% increase per year, for 20 years

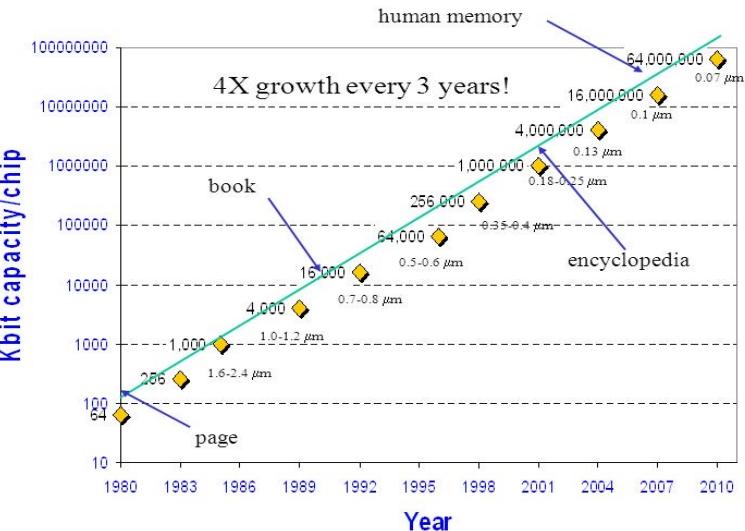


Introduction

ICS 233 – Computer Architecture and Assembly Language – KFUPM

© Muhamed Mudawar – slide 41

DRAM Chip Capacity



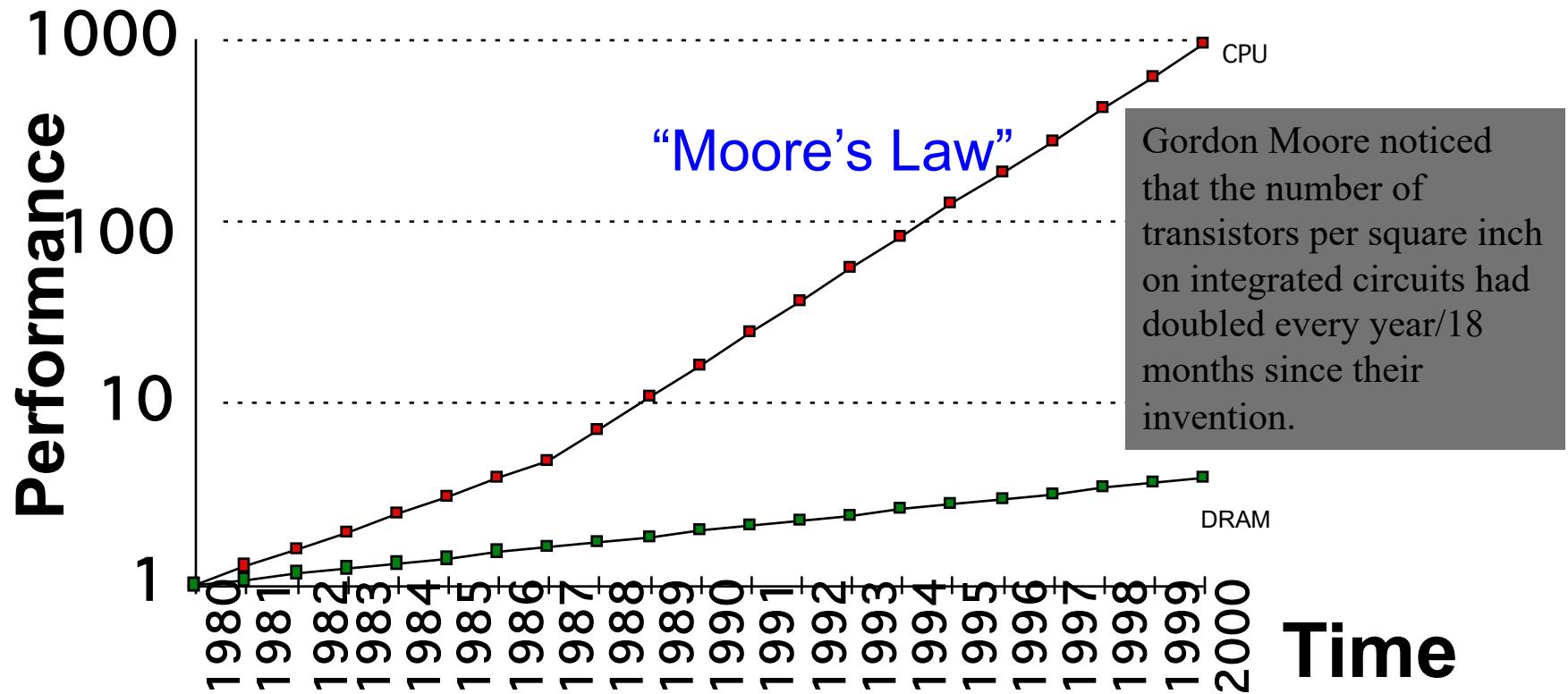
- Unprecedented growth in density, but we still have a problem

1. Processor-DRAM Performance Gap

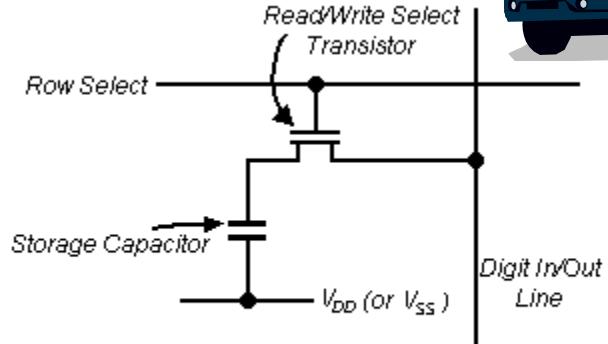
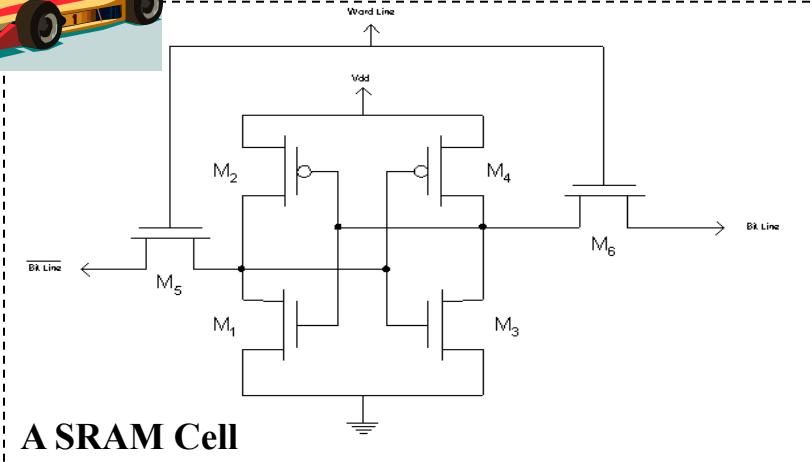
Memory Wall:

1GHz Processor → 1 ns per clock cycle

50ns for DRAM access → 50 processor clock cycles per memory access!



1. Faster Memory Technology: SRAM



SRAM

6 transistors per memory cell

→ **Low density**

Fast access latency of 0.5 – 5 ns

More costly

Uses flip-flops

DRAM

1 transistor per memory cell

→ **High density**

Slow access latency of 50-70ns

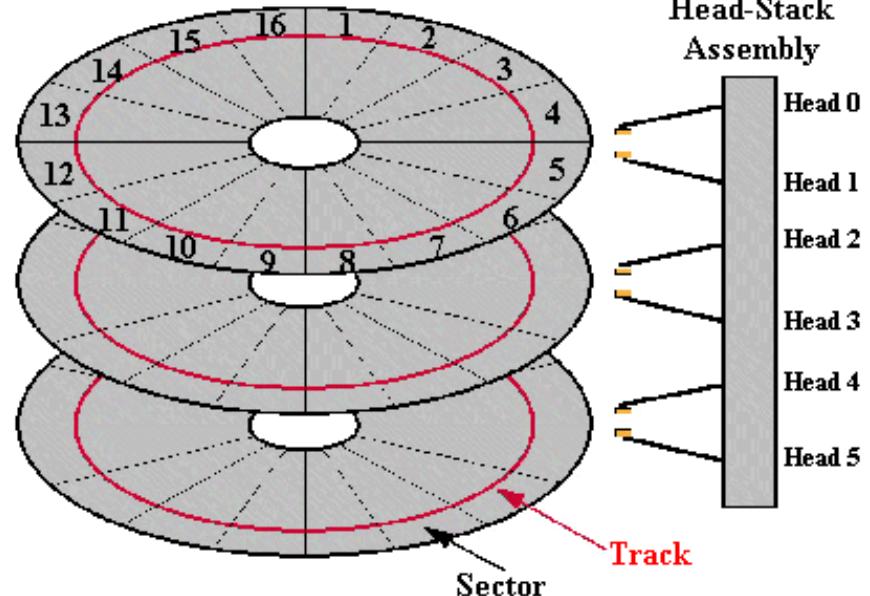
Less costly

Used in main memory

1. Slow Memory Technology: Magnetic Disk



Drive Physical and Logical Organization

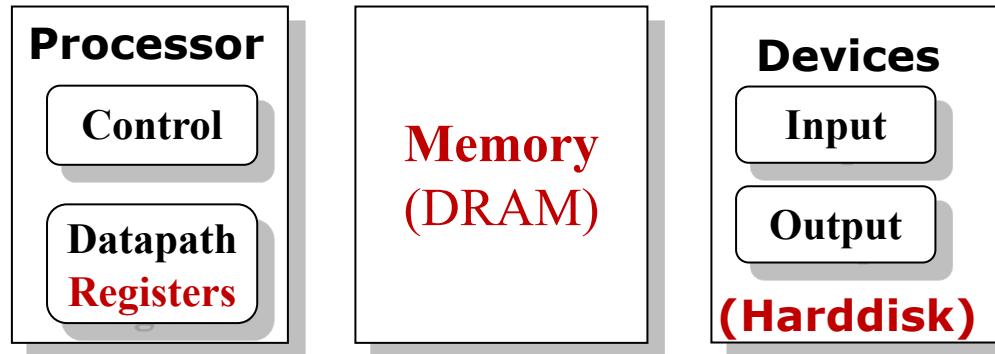


Typical high-end hard disk:

Average Latency: 4 - 10 ms

Capacity: 500-2000GB

1. Quality vs Quantity




	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	\$\$\$\$
SRAM	100s KB	0.5-5 ns	\$\$\$
DRAM	100s MB	50-70 ns	\$
Hard Disk	100s GB	5-20 ms	Cents
Ideal	1 GB	1 ns	Cheap

1. Best of Both Worlds

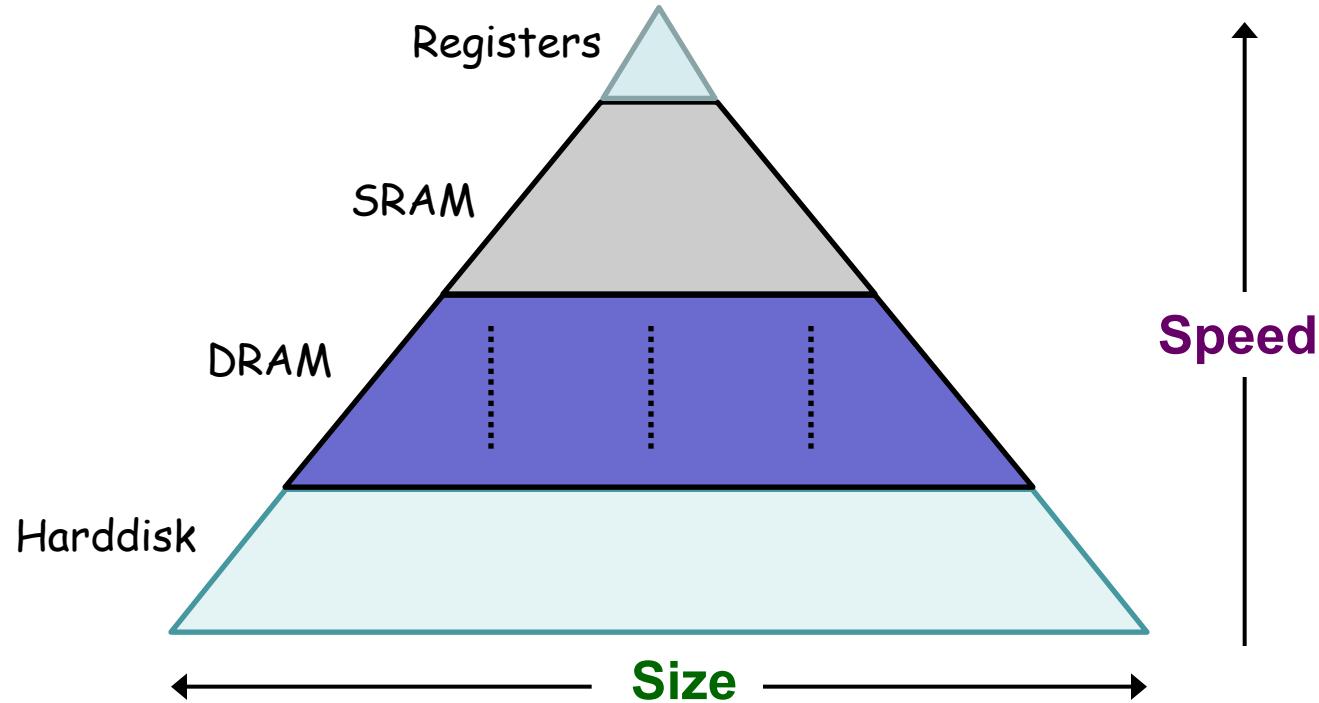
- What we want:
 - A **BIG** and **FAST** memory
 - Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory

Key concept:

Use a **hierarchy** of memory technologies:

- ❖ Small but fast memory near CPU
- ❖ Large but slow memory farther away from CPU

1. Memory Hierarchy



2. Cache: The Library Analogy



Imagine you are forced to put back a book to its bookshelf before taking another book.....

2. Solution: Book on the Desk!



What if you are allowed to take the books that are **likely to be needed soon** with you and place them nearby on the desk?

2. Cache: The Basic Idea

- **Keep the frequently and recently used data in smaller but faster memory**
- **Refer to bigger and slower memory:**
 - Only when you cannot find data/instruction in the faster memory
- **Why does it work?**

Principle of Locality

Program accesses only a small portion of the memory address space within a small time interval

2.1 Cache: Types of Locality

Temporal locality

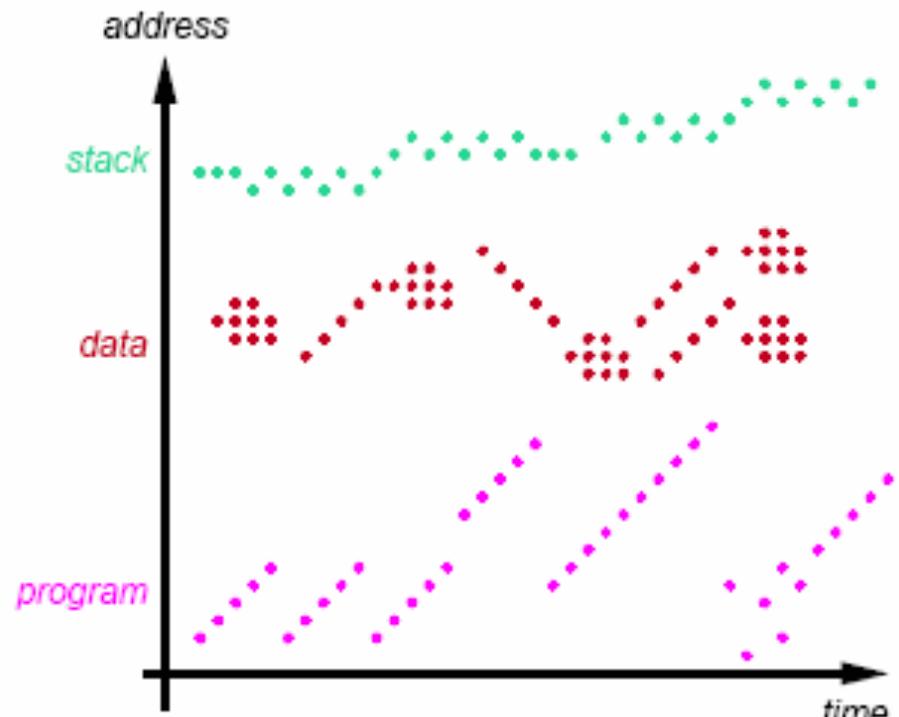
- If an item is referenced, it will tend to be referenced again soon

Spatial locality

- If an item is referenced, nearby items will tend to be referenced soon

Different locality for

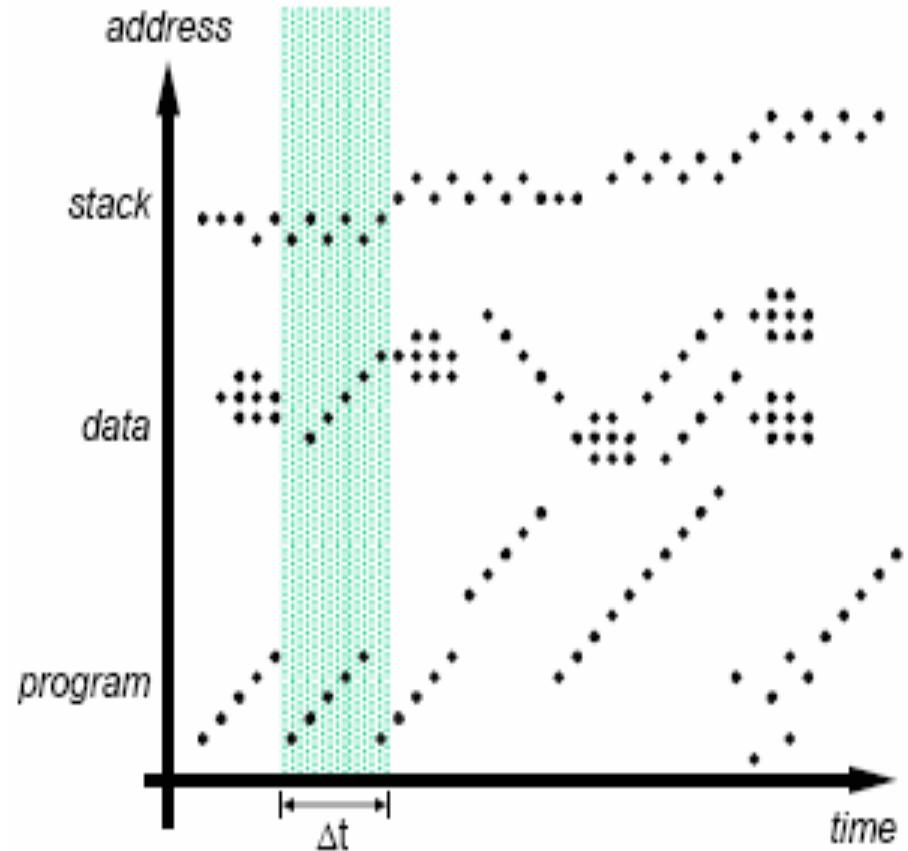
- Instructions
- Data



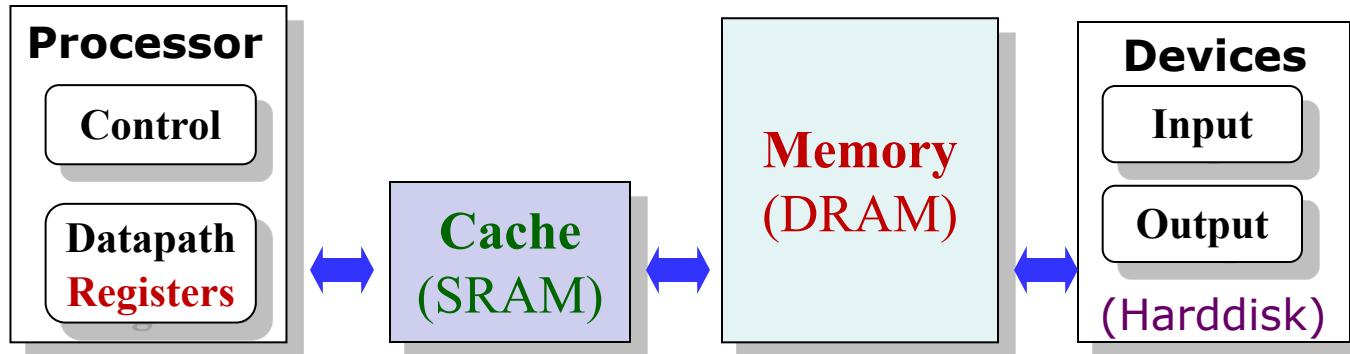
2.1 Working Set: Definition

- Set of locations accessed during Δt
- Different phases of execution may use different working sets

Our aim is to **capture the working set and keep it in the memory closest to CPU**

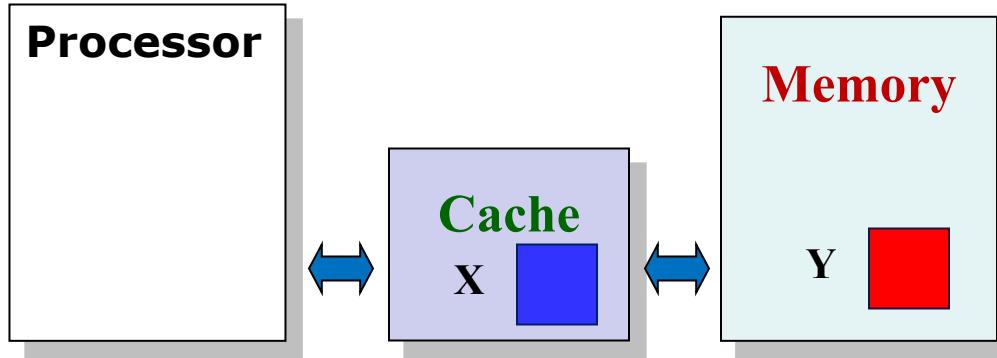


2.2 Two Aspects of Memory Access



- How to make SLOW main memory appear faster?
 - **Cache** – a small but fast SRAM near CPU
 - **Hardware managed:** Transparent to programmer
- How to make SMALL main memory appear bigger than it is?
 - **Virtual memory**
 - **OS managed:** Transparent to programmer
 - Not in the scope of this module (covered in CS2106)

2.2 Memory Access Time: Terminology



- **Hit: Data is in cache (e.g., X)**
 - Hit rate: Fraction of memory accesses that hit
 - Hit time: Time to access cache
- **Miss: Data is not in cache (e.g., Y)**
 - Miss rate = $1 - \text{Hit rate}$
 - Miss penalty: Time to replace cache block + hit time
- **Hit time < Miss penalty**

2.2 Memory Access Time: Formula

Average Access Time

$$= \text{Hit rate} \times \text{Hit Time} + (1 - \text{Hit rate}) \times \text{Miss penalty}$$

Example:

- Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM (main memory) we can get has an access time of **10ns**. **How high a hit rate** do we need to sustain an average access time of **1ns**?

Let h be the desired hit rate.

$$1 = 0.8h + (1 - h) \times (10 + 0.8)$$

$$= 0.8h + 10.8 - 10.8h$$

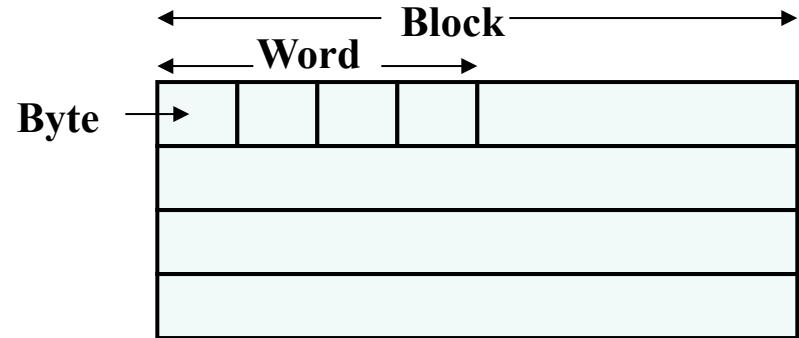
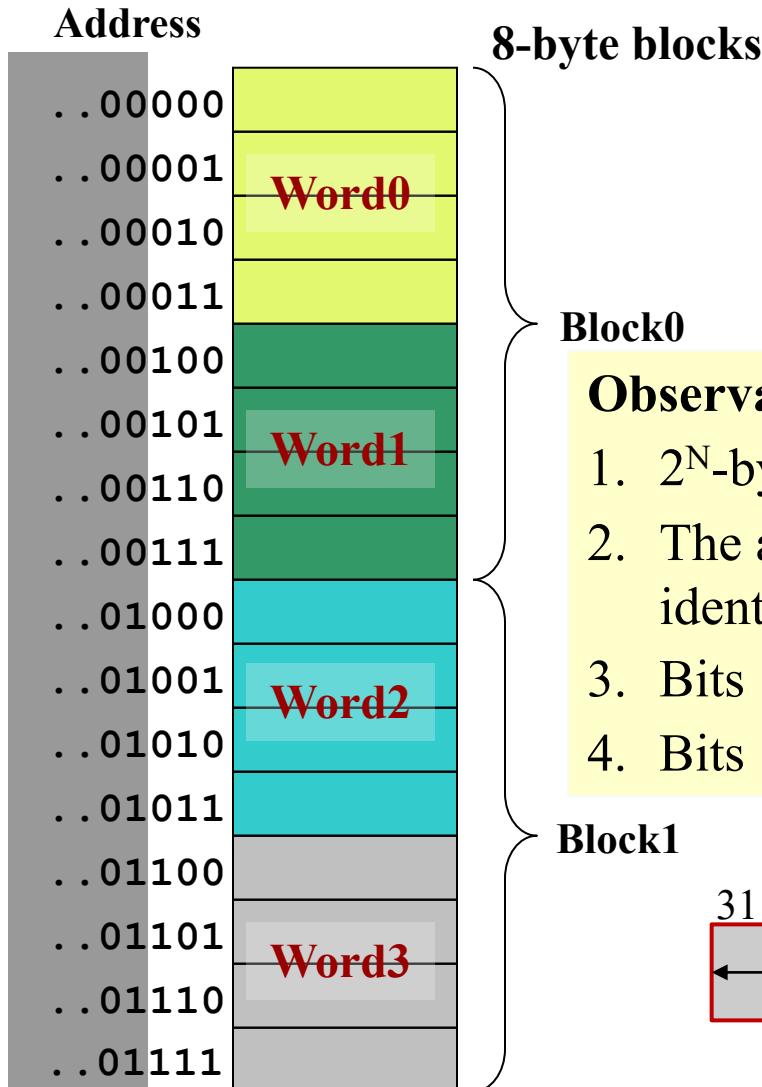
$$10h = 9.8 \rightarrow h = 0.98$$

Hence we need a hit rate of **98%**.

3. Memory to Cache Mapping (1/2)

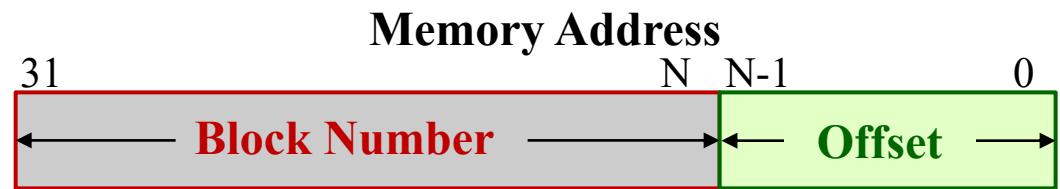
- **Cache Block/Line:**
 - Unit of transfer between memory and cache
- Block size is typically one or more words
 - e.g.: 16-byte block \cong 4-word block
 - 32-byte block \cong 8-word block
- Why is the block size bigger than word size?

3. Memory to Cache Mapping (2/2)



Observations:

1. 2^N -byte blocks are aligned at 2^N -byte boundaries
2. The addresses of words within a 2^N -byte block have identical (32-N) most significant bits (MSB).
3. Bits [31:N] → the **block number**
4. Bits [N-1:0] → the **byte offset** within a block



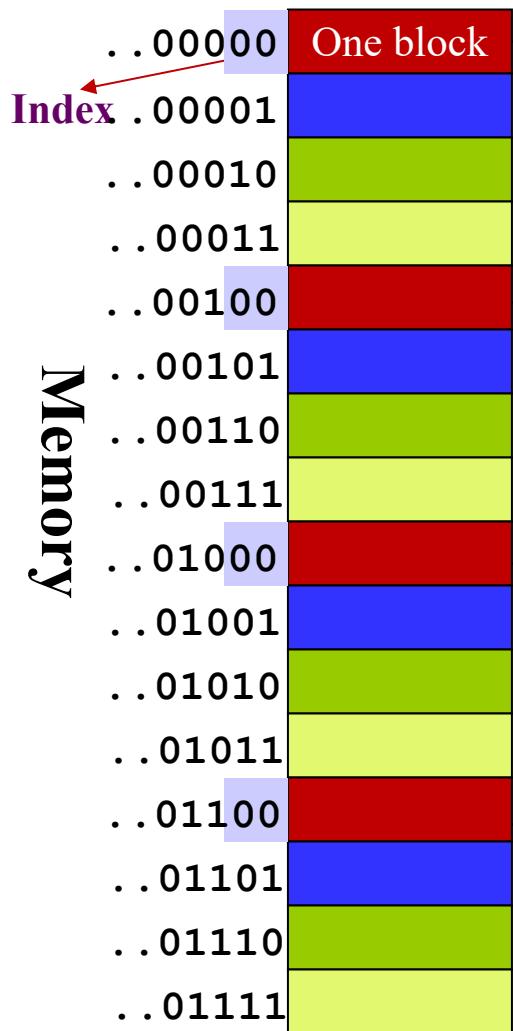
4. Direct Mapping Analogy



Imagine there are 26 “locations” on the desk to store books. A book’s location is determined by the first letter of its title.
→ Each book **has exactly one location.**

4. Direct Mapped Cache: Cache Index

Block Number (Not Address!)



Mapping Function:
Cache Index
 $= (\text{BlockNumber}) \bmod (\text{NumberOfCacheBlocks})$

Observation:

If Number of Cache Blocks = 2^M

→ the last M bits of the block number is the **cache index**

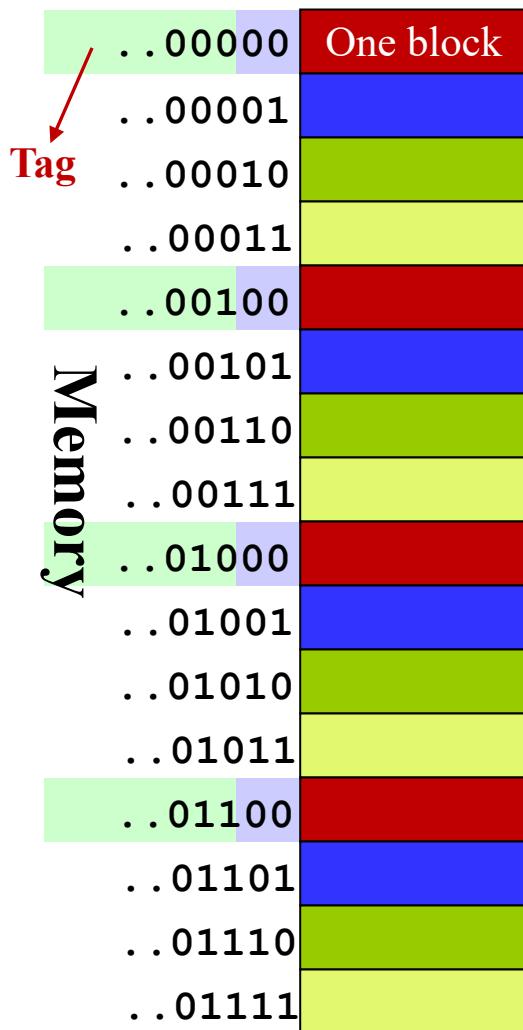
Example:

Cache has $2^2 = 4$ blocks

→ last 2 bits of the block number is the cache index.

4. Direct Mapped Cache: Cache Tag

Block Number (Not Address!)



Mapping Function:
Cache Index
 $= (\text{BlockNumber}) \bmod (\text{NumberOfCacheBlocks})$

Cache Index



Cache

Observation:

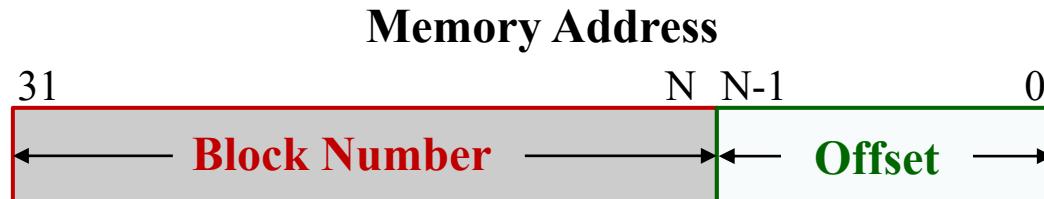
Multiple memory blocks can map to the same cache block

→ Same Cache Index

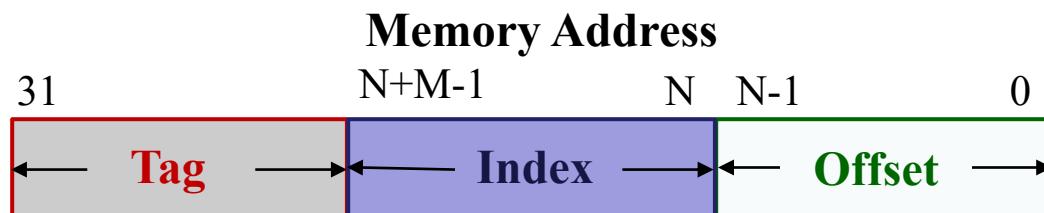
However, they have unique **tag number**:

Tag = Block number / Number of Cache Blocks

4. Direct Mapped Cache: Mapping



Cache Block size = 2^N bytes



Cache Block size = 2^N bytes

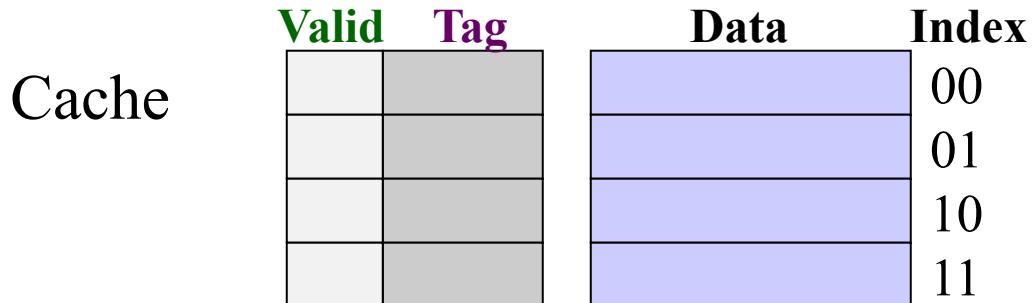
Number of cache blocks = 2^M

Offset = **N bits**

Index = **M bits**

Tag = **32 – (N + M) bits**

4. Direct Mapped Cache: Cache Structure



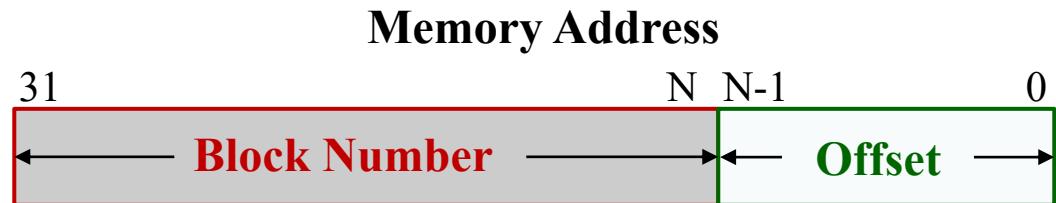
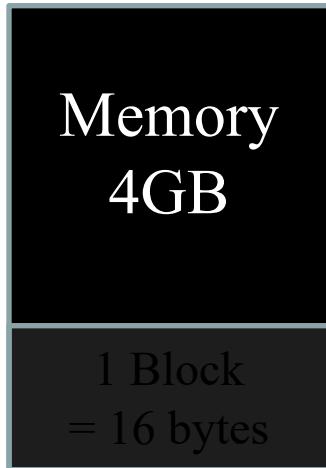
Along with a data block (line), cache also contains the following administrative information (overheads):

1. **Tag** of the memory block
2. **Valid bit** indicating whether the cache line contains valid data

When is there a cache hit?

(Valid[index] = TRUE) AND
 (Tag[index] = Tag[memory address])

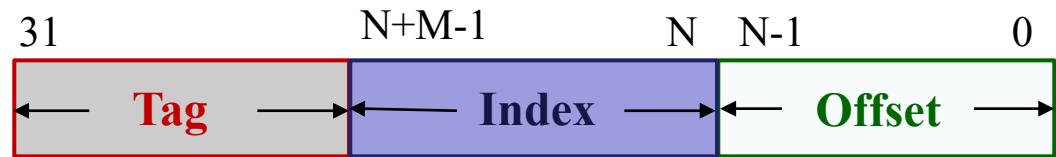
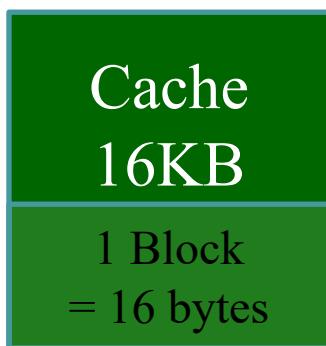
4. Cache Mapping: Example



Offset, N = 4 bits

Block Number = $32 - 4 = 28$ bits

Check: Number of Blocks = 2^{28}



Number of Cache Blocks

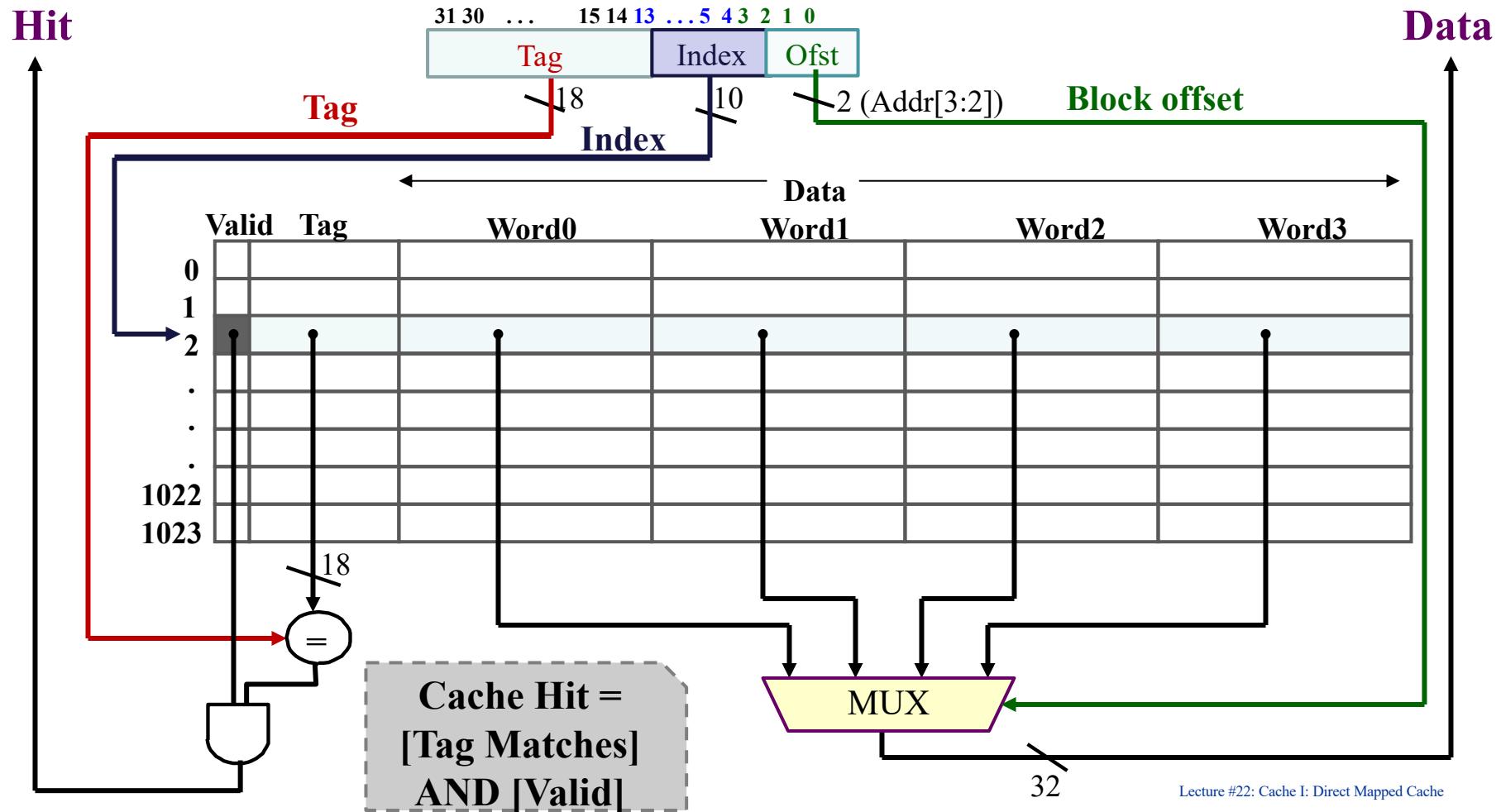
= $16\text{KB} / 16\text{bytes} = 1024 = 2^{10}$

Cache Index, M = 10bits

Cache Tag = $32 - 10 - 4 = 18$ bits

4. Cache Circuitry: Example

16-KB cache:
4-word (16-byte) blocks



5. Reading Data: Setup

- Given a direct mapped 16KB cache:
- 16-byte blocks x 1024 cache blocks
- Trace the following memory accesses:

Tag	Index	Offset
31 00000000000000000000000000000000	14 13 00000000001	4 3 0 0100
00000000000000000000000000000000	00000000001	1100
00000000000000000000000000000000	00000000011	0100
00000000000000000000000000000010	00000000001	1000
00000000000000000000000000000000	00000000001	0000

5. Reading Data: Initial State

- Initially cache is empty
 - All *valid* bits are zeroes (false)

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					

5. Reading Data: Load #1-1

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000001	0100

Step 1. Check Cache Block at index 1

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #1-2

Tag**Index****Offset**

- Load from

00000000000000000000	0000000001	0100
----------------------	------------	------

Step 2. Data in block 1 is invalid [Cold/Compulsory Miss]

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					

5. Reading Data: Load #1-3

Tag**Index****Offset**

- Load from

00000000000000000000	0000000001	0100
----------------------	------------	------

Step 3. Load 16 bytes from memory; Set Tag and Valid bit

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					

5. Reading Data: Load #1-4

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000001	0100

Step 4. Return Word1 (byte offset = 4) to Register

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #2-1

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000001	1100

Step 1. Check Cache Block at index 1

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #2-2

Tag**Index****Offset**

- Load from

00000000000000000000	0000000001	1100
----------------------	------------	------

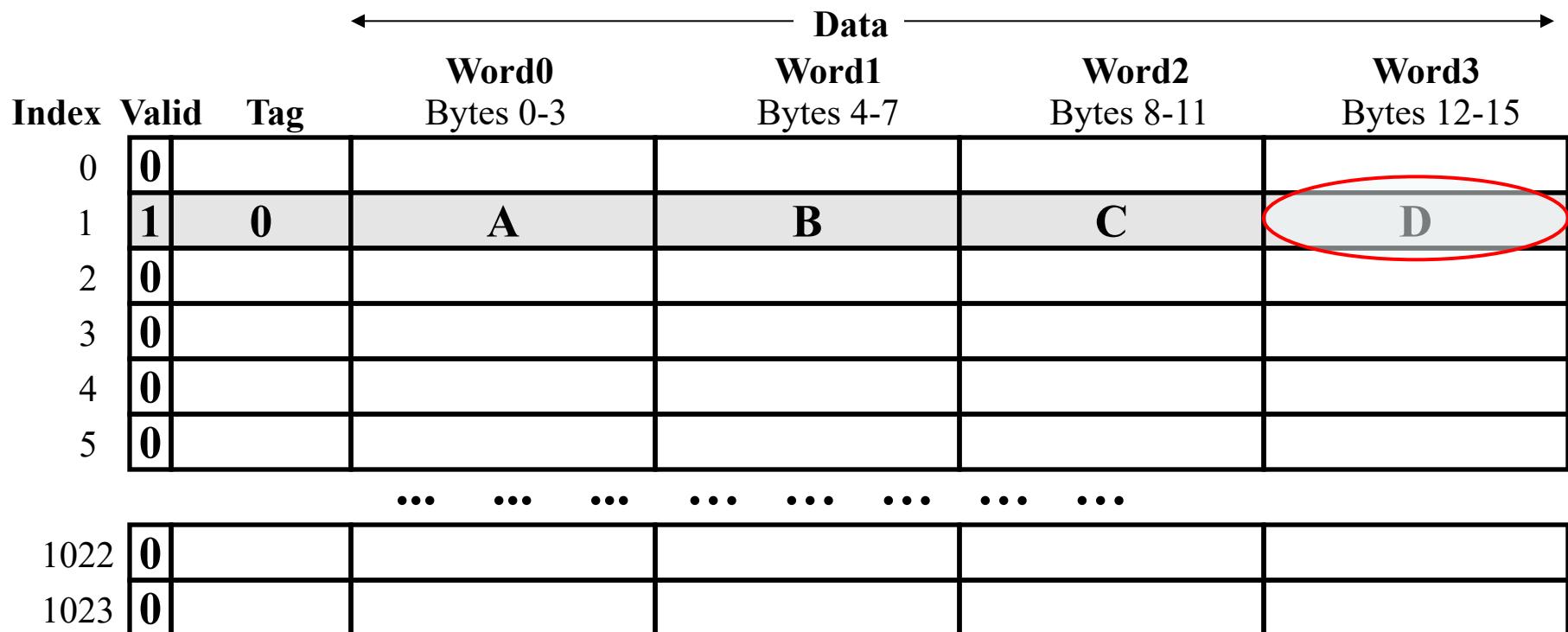
Step 2. [Cache Block is Valid] AND [Tags match] → Cache hit!

Index	Data		
	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11
0			
1	0	A	B
2			C
3			D
4			
5			
...			
1022			
1023			

5. Reading Data: Load #2-3

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000001	1100

Step 3. Return Word3 (byte offset = 12) to Register [Spatial Locality]



5. Reading Data: Load #3-1

	Tag	Index	Offset
■ Load from	00000000000000000000	00000000011	0100

Step 1. Check Cache Block at index 3

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #3-2

Tag**Index****Offset**

- Load from

00000000000000000000	00000000011	0100

Step 2. Data in block 3 is invalid [Cold/Compulsory Miss]

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					

5. Reading Data: Load #3-3

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000011	0100

Step 3. Load 16 bytes from memory; Set Tag and Valid bit

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #3-4

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000011	0100

Step 4. Return Word1 (byte offset = 4) to Register

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #4-1

	Tag	Index	Offset
■ Load from	000000000000000010	0000000001	1000

Step 1. Check Cache Block at index 1

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #4-2

	Tag	Index	Offset
■ Load from	000000000000000010	0000000001	1000

Step 2. Cache block is Valid but Tags mismatch [Cold miss]

Index	Data			
	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0			
1	0	A	B	C
2	0			
3	0	I	J	K
4	0			
5	0			
... 				
1022	0			
1023	0			

5. Reading Data: Load #4-3

Tag**Index****Offset**

- Load from

000000000000000010	0000000001	1000
--------------------	------------	------

Step 3. Replace block 1 with new data; Set Tag

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					

5. Reading Data: Load #4-4



Step 4. Return Word2 (byte offset = 8) to Register

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Load #5-1

Tag**Index****Offset**

- Load from

00000000000000000000	0000000001	0000
----------------------	------------	------



Step 1. Check Cache Block at index 1

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					

5. Reading Data: Load #5-2

	Tag	Index	Offset
■ Load from	00000000000000000000	0000000001	0000

Step 2. Cache block is Valid but Tags mismatch [Cold miss]

Index	Data			
	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0			
1	2	E	F	G
2	0			
3	1	I	J	K
4	0			
5	0			
... 				
1022	0			
1023	0			

5. Reading Data: Load #5-3

Tag**Index****Offset**

- Load from

00000000000000000000	0000000001	0000
----------------------	------------	------

Step 3. Replace block 1 with new data; Set Tag

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					

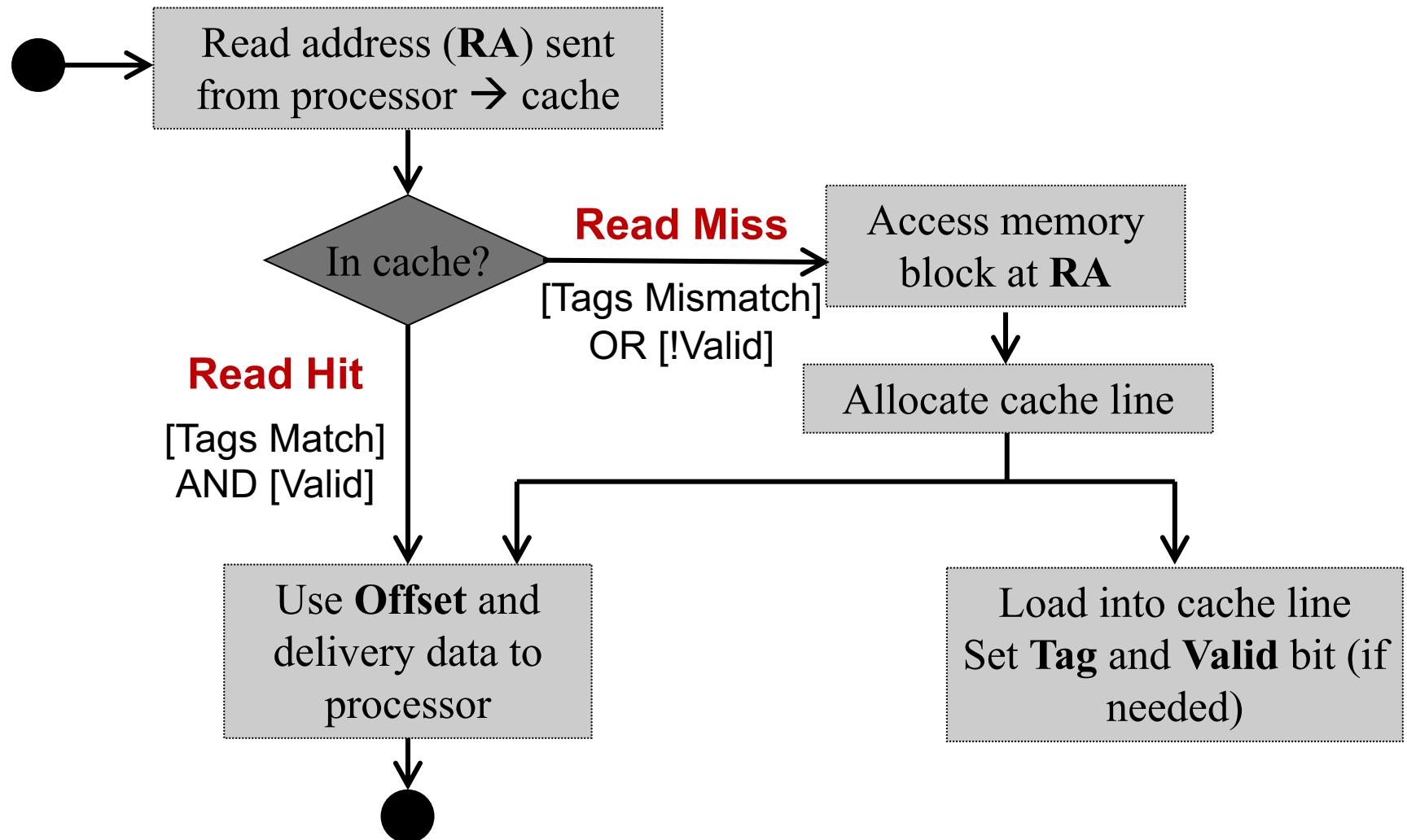
5. Reading Data: Load #5-4



Step 4. Return Word0 (byte offset = 0) to Register

Index	Valid	Tag	Data			
			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... 						
1022	0					
1023	0					

5. Reading Data: Summary



6. Types of Cache Misses

■ Compulsory misses

- On the first access to a block; the block must be brought into the cache
- Also called cold start misses or first reference misses

■ Conflict misses

- Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
- Also called collision misses or interference misses

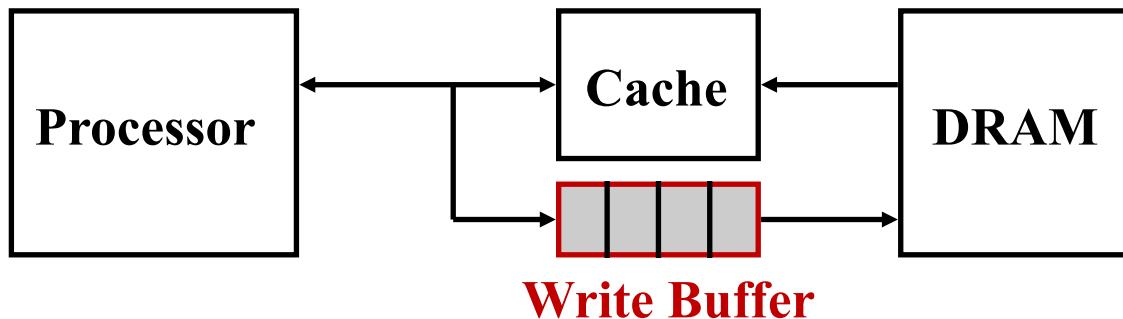
■ Capacity misses

- Occur when blocks are discarded from cache as cache cannot contain all blocks needed

7. Changing Cache Content: Write Policy

- Cache and main memory are inconsistent
 - Modified data only in cache, not in memory!
- **Solution 1: Write-through** cache
 - Write data both to cache and to main memory
- **Solution 2: Write-back** cache
 - Only write to cache
 - Write to main memory only when cache block is replaced (evicted)

7. Write-Through Cache



- **Problem:**

- Write will operate at the speed of main memory!

- **Solution:**

- Put a write buffer between cache and main memory
 - Processor: writes data to cache + write buffer
 - Memory controller: write contents of the buffer to memory

7. Write-Back Cache

- **Problem:**

- Quite wasteful if we write back every evicted cache blocks

- **Solution:**

- Add an additional bit (**Dirty bit**) to each cache block
 - Write operation will change dirty bit to 1
 - Only cache block is updated, no write to memory
 - When a cache block is replaced:
 - Only write back to memory if dirty bit is 1

7. Handling Cache Misses

- On a **Read Miss**:

- Data loaded into cache and then load from there to register

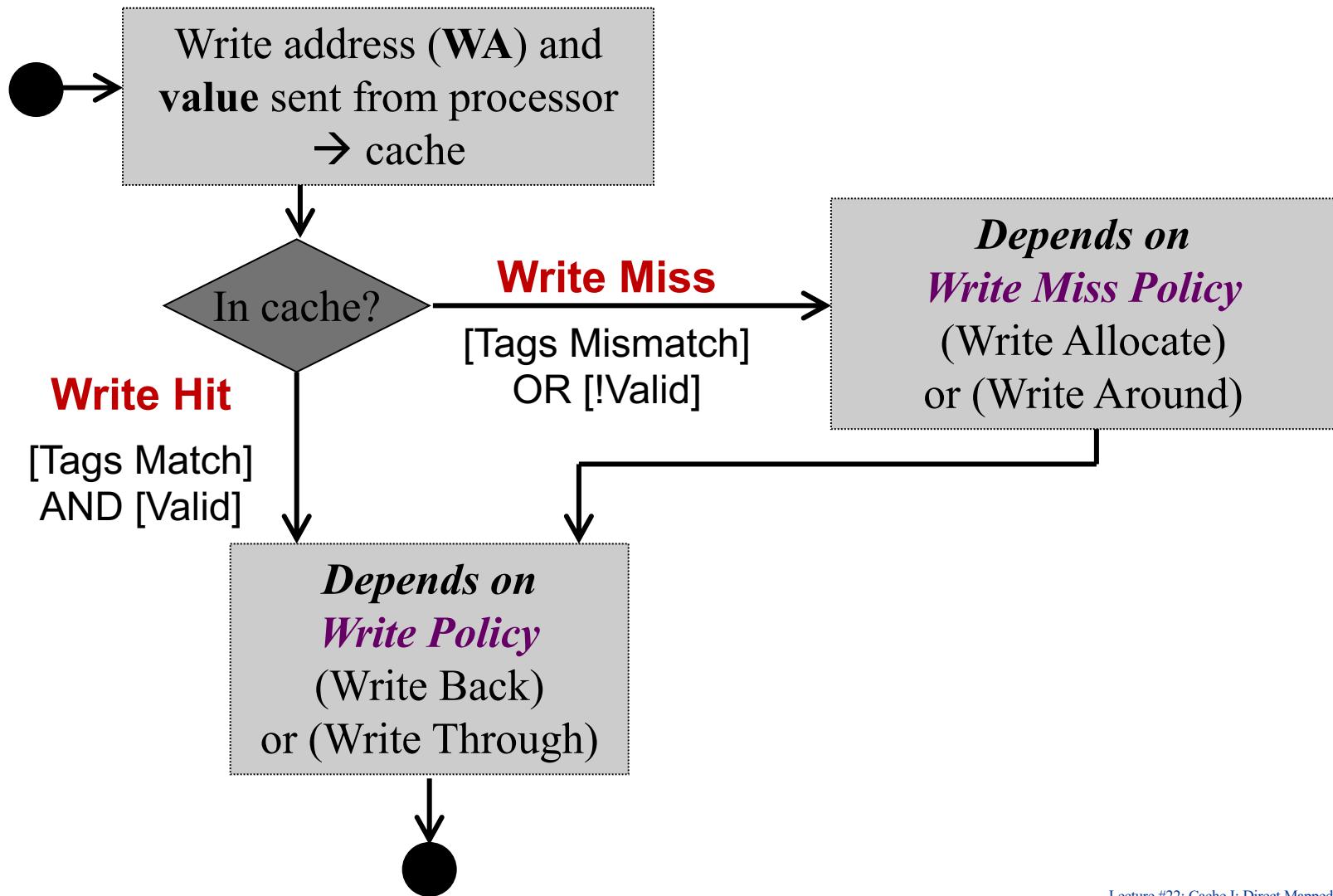
- Write Miss** option 1: **Write allocate**

- Load the complete block into cache
 - Change only the required word in cache
 - Write to main memory depends on write policy

- **Write Miss** option 2: **Write around**

- Do not load the block to cache
 - Write directly to **main memory only**

7. Writing Data: Summary



8. Set Associative (SA) Cache

■ Compulsory misses

- On the first access to a block, it must be brought into the cache
- Also called cold start misses

Solution:

Set Associative Cache

■ Conflict misses

- Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
- Also called **collision misses** or **interference misses**

■ Capacity misses

- Occur when blocks are discarded from cache as cache cannot contain all blocks needed

8. Set Associative Cache: Analogy



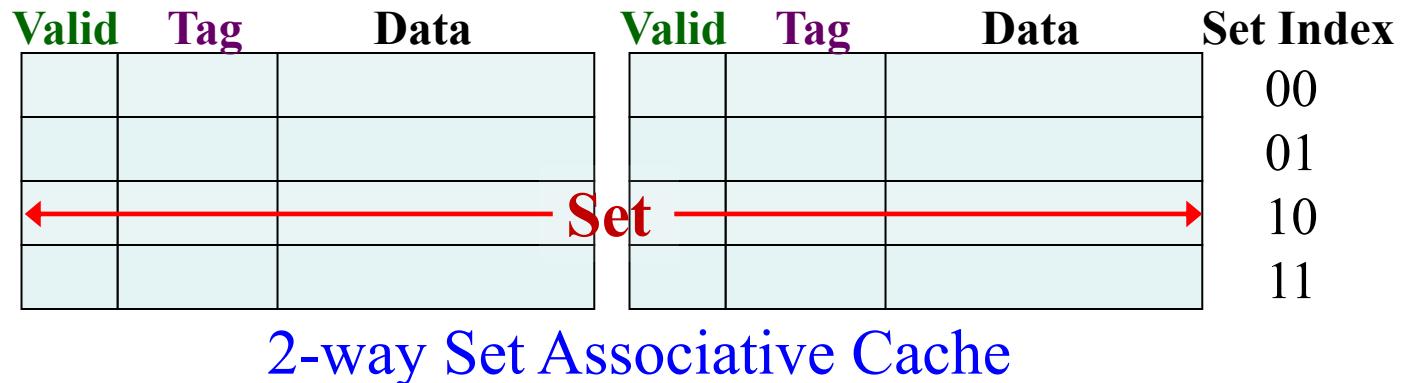
Many book titles start with “T”
→ Too many conflicts!

Hmm... how about we give more slots per letter, 2 books start with “A”, 2 books start with “B”, etc?

8. Set Associative (SA) Cache

- **N-way Set Associative Cache**
 - A memory block can be placed in a fixed number (**N**) of locations in the cache, where **N > 1**
- **Key Idea:**
 - Cache consists of a number of sets:
 - **Each set contains N cache blocks**
 - Each memory block maps to a unique cache set
 - Within the set, a memory block can be placed in **any** of the **N** cache blocks in the set

8. Set Associative Cache: Structure



- An example of 2-way set associative cache
 - Each set has two cache blocks
- A memory block maps to a unique set
 - In the set, the memory block can be placed in either of the cache blocks
- ➔ Need to search both to look for the memory block

8. Set Associative Cache: Mapping

Memory Address



Cache Block size = 2^N bytes

Cache Set Index

= (BlockNumber) modulo (NumberOfCacheSets)



Cache Block size = 2^N bytes

Number of cache sets = 2^M

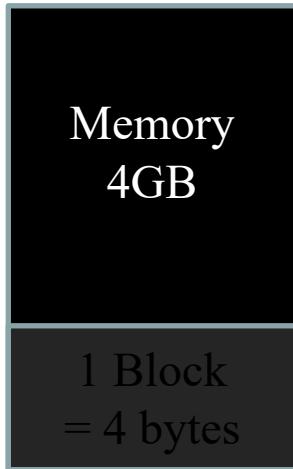
Offset = **N bits**

Set Index = **M bits**

Tag = **32 – (N + M) bits**

Observation:
It is essentially unchanged from the direct-mapping formula

8. Set Associative Cache: Example



The diagram illustrates the structure of a memory address. At the top, the text "Memory Address" is centered. Below it, a horizontal line represents the address field, divided into three main sections: "Block Number" (highlighted in red), "Offset" (highlighted in green), and a numerical range from 31 to 0. The "Block Number" section spans from index 31 down to index N, indicated by a red bracket below the line. The "Offset" section spans from index N-1 down to index 0, indicated by a green bracket below the line.

Offset, N = 2 bits

Block Number = $32 - 2 = 30$ bits

Check: Number of Blocks = **2³⁰**



Number of Cache Blocks

$$= 4\text{KB} / 4\text{bytes} = 1024 = 2^{10}$$

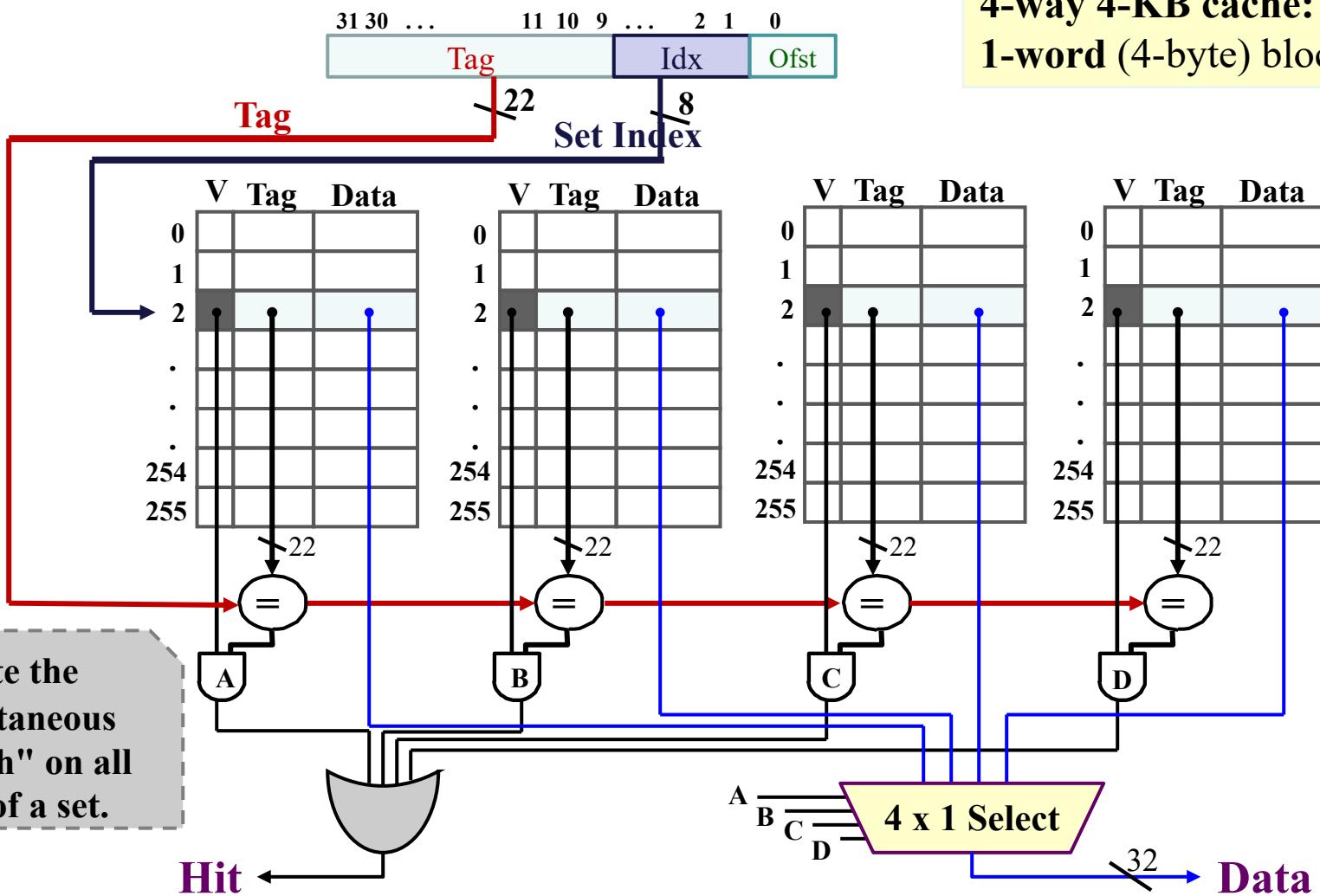
4-way associative, number of sets

$$= 1024 / 4 = 256 = 2^8$$

Set Index, M = 8 bits

$$\text{Cache Tag} = 32 - 8 - 2 = \mathbf{22 \text{ bits}}$$

8. Set Associative Cache: Circuitry



8. SA Cache Example: Setup

- Given:

- Memory access sequence: **4, 0, 8, 36, 0**
- 2-way set-associative cache with a total of four 8-byte blocks → **total of 2 sets**
- Indicate hit/miss for each access



Offset, N = 3 bits

Block Number = $32 - 3 = 29$ bits

2-way associative, number of sets = $2 = 2^1$

Set Index, M = 1 bits

Cache Tag = $32 - 3 - 1 = 28$ bits

8. SA Cache Example: Load #1

Miss

4, 0 , 8, 36, 0

Tag Index Offset

- Load from 4 →

00000000000000000000000000000000

0 100

Check: Both blocks in Set 0 are invalid [**Cold Miss**]

Result: Load from memory and place in Set 0 - Block 0

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	✓ 1	0	M[0]	M[4]	0			
1	0				0			

8. SA Cache Example: Load #2

 Miss Hit

4, 0, 8, 36, 0

Tag Index Offset

- Load from 0 →

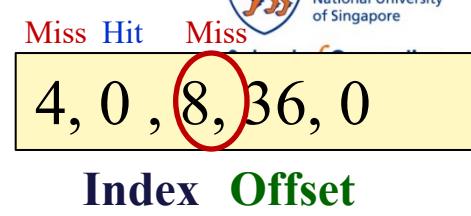
00000000000000000000000000000000 0 000

Result:

[Valid and Tags match] in Set 0-Block 0 [Spatial Locality]

Block 0				Block 1				
Set Index	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	0				0			

8. SA Cache Example: Load #3



- Load from 8 →



Check: Both blocks in Set 1 are invalid [**Cold Miss**]

Result: Load from memory and place in Set 1 - Block 0

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	✓ 1	0	M[8] (circled)	M[12]	0			

8. SA Cache Example: Load #4

Miss Hit Miss Miss

4, 0 , 8, 36, 0

Tag Index Offset

- Load from 36 →

000000000000000000000000000010

0 100

Check: [Valid but tag mismatch] Set 0 - Block 0

[Invalid] Set 0 - Block1 [Cold Miss]

Result: Load from memory and place in Set 0 - Block 1

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0	1	2	M[32]
1	1	0	M[8]	M[12]	0			M[36]

8. SA Cache Example: Load #5

Miss Hit Miss Miss Hit

4, 0 , 8, 36, 0

Tag Index Offset

- Load from 0 →

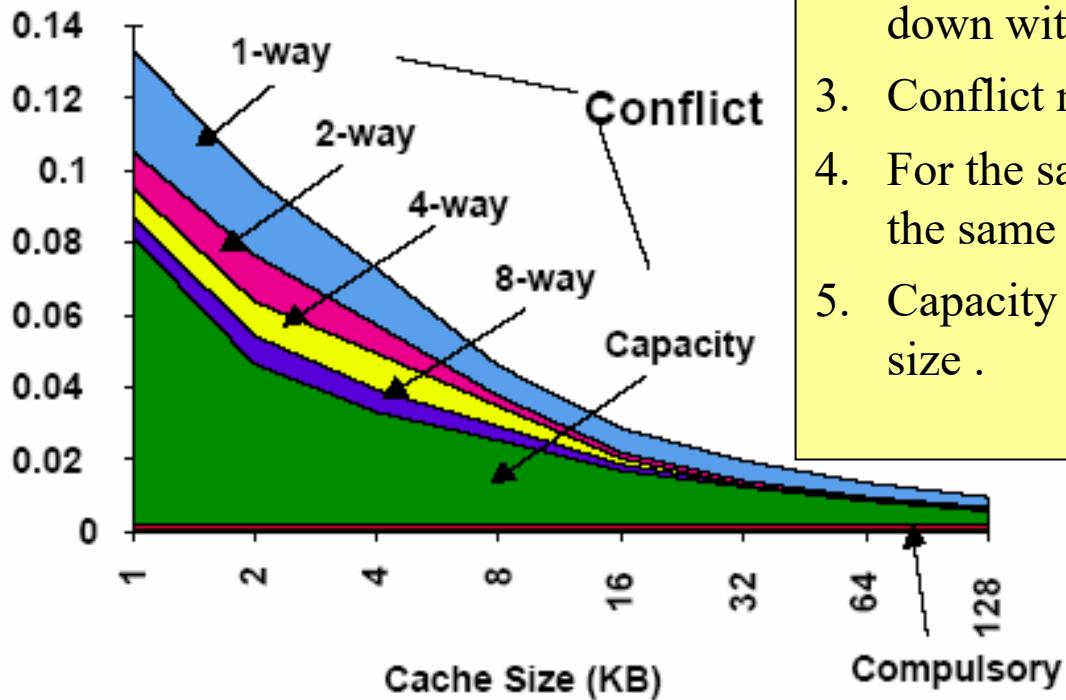
00000000000000000000000000000000 | 0 000

Check: [Valid and tags match] Set 0-Block 0
 [Valid but tags mismatch] Set 0-Block1
 [Temporal Locality]

	Block 0				Block 1			
Set Idx	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	1	2	M[32]	M[36]
1	1	0	M[8]	M[12]	0			

9. Cache Performance

Identical block size



Observations:

1. Cold/compulsory miss remains the same irrespective of cache size/associativity.
2. For the same cache size, conflict miss goes down with increasing associativity.
3. Conflict miss is 0 for FA caches.
4. For the same cache size, capacity miss remains the same irrespective of associativity.
5. Capacity miss decreases with increasing cache size .

$$\text{Total Miss} = \text{Cold miss} + \text{Conflict miss} + \text{Capacity miss}$$

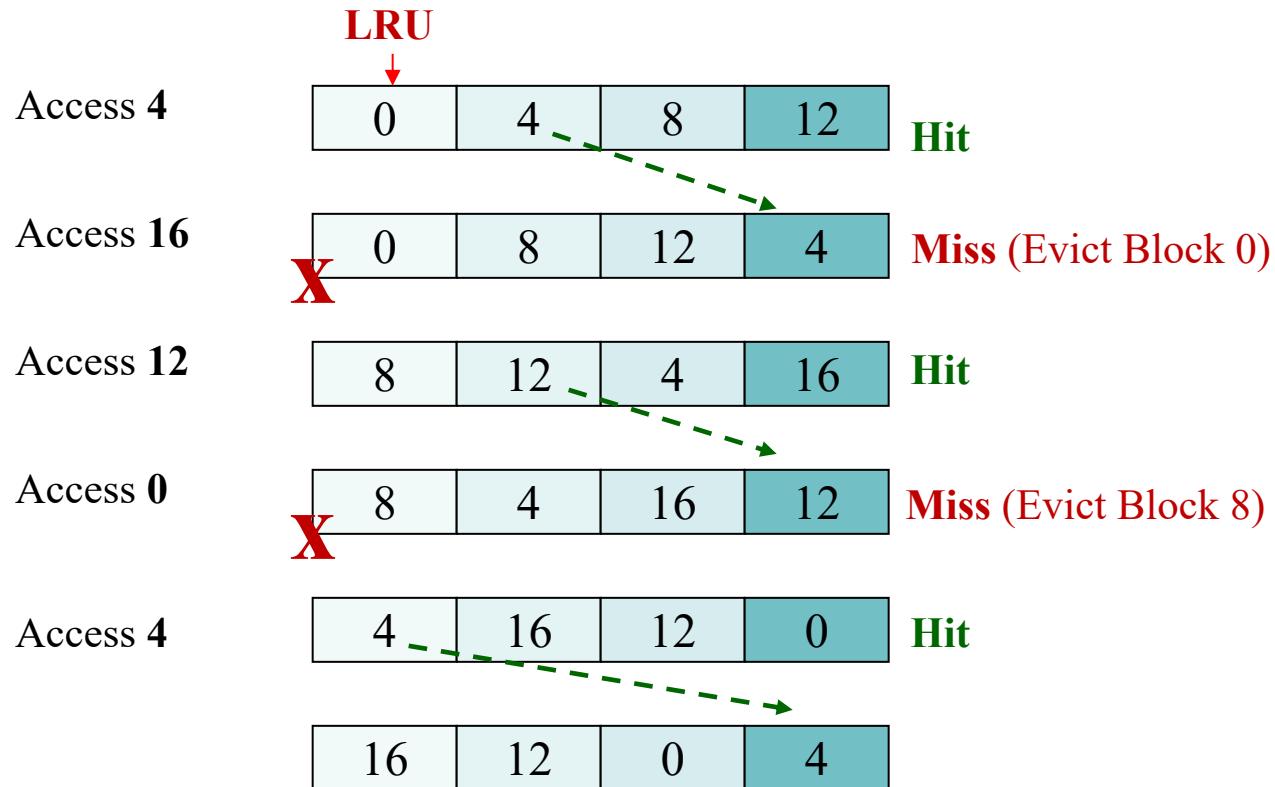
$$\text{Capacity miss (FA)} = \text{Total miss (FA)} - \text{Cold miss (FA)}, \text{ when Conflict Miss} \rightarrow 0$$

10. Block Replacement Policy (1/3)

- Set Associative or Fully Associative Cache:
 - Can choose where to place a memory block
 - Potentially replacing another cache block if full
 - Need **block replacement policy**
- Least Recently Used (LRU)
 - **How:** For cache hit, record the cache block that was accessed
 - When replacing a block, choose one which has not been accessed for the longest time
 - **Why:** Temporal locality

10. Block Replacement Policy (2/3)

- Least Recently Used policy in action:
 - 4-way SA cache
 - Memory accesses: **0 4 8 12 4 16 12 0 4**



10. Block Replacement Policy (3/3)

- Drawback for LRU
 - Hard to keep track if there are many choices
- Other replacement policies:
 - First in first out (FIFO)
 - Random replacement (RR)
 - Least frequently used (LFU)

11. Summary: Cache Organizations

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	Ta	Data	Tag	Data	Tag	Dat	Tag	Data
0	g					a		
1								

11. Summary: Cache Framework (1/2)

Block Placement: Where can a block be placed in cache?

Direct Mapped:

- Only one block defined by index

N-way Set-Associative:

- Any one of the **N** blocks within the set defined by index

Block Identification: How is a block found if it is in the cache?

Direct Mapped:

- Tag match with only one block

N-way Set Associative:

- Tag match for all the blocks within the set

11. Summary: Cache Framework (2/2)

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:

- No Choice

N-way Set-Associative:

- Based on replacement policy

Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

Write Miss Policy: Write allocate vs write no allocate

12. Exploration: Improving Cache Penalty

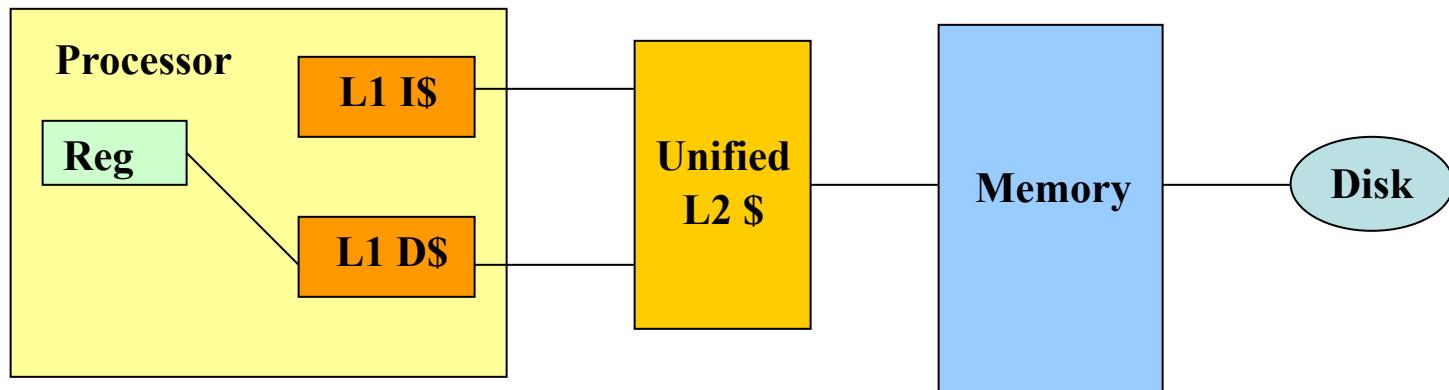
Average Access Time

$$= \text{Hit rate} \times \text{Hit Time} + (1 - \text{Hit rate}) \times \text{Miss penalty}$$

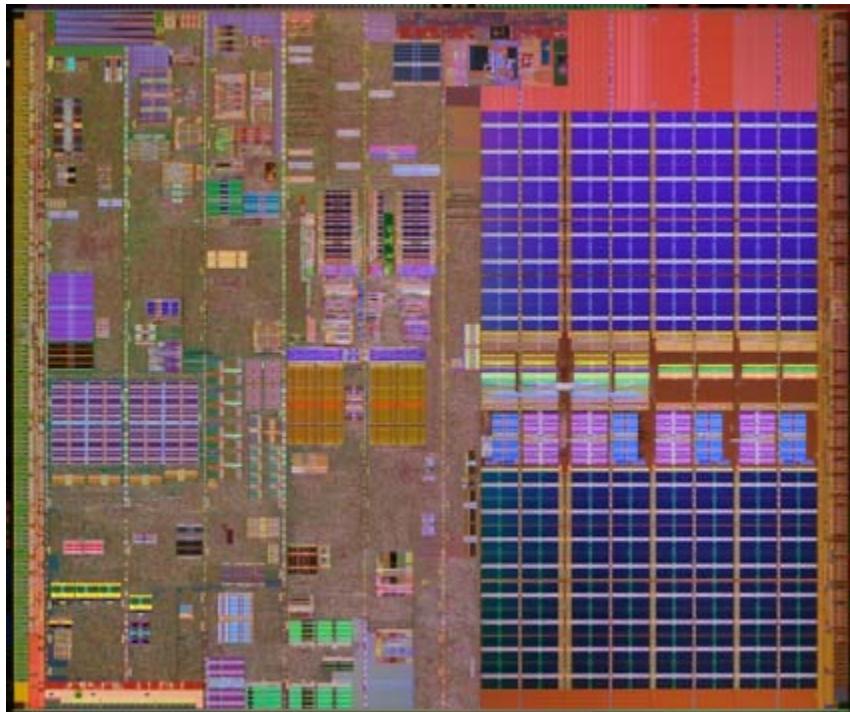
- So far, we tried to improve Miss Rate:
 - Larger block size
 - Larger Cache
 - Higher Associativity
- What about Miss Penalty?

12. Exploration: Multilevel Cache

- Options:
 - Separate data and instruction caches, or a unified cache
- Sample sizes:
 - **L1**: 32KB, 32-byte block, 4-way set associative
 - **L2**: 256KB, 128-byte block, 8-way associative
 - **L3**: 4MB, 256-byte block, Direct mapped



12. Exploration: Intel Processors

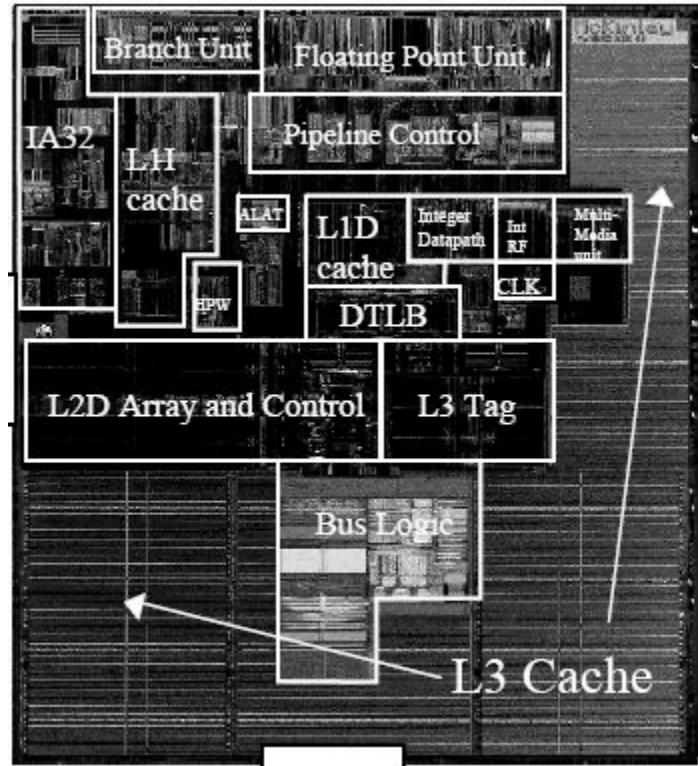


Pentium 4 Extreme Edition

L1: 12KB I\$ + 8KB D\$

L2: 256KB

L3: 2MB



Itanium 2 McKinley

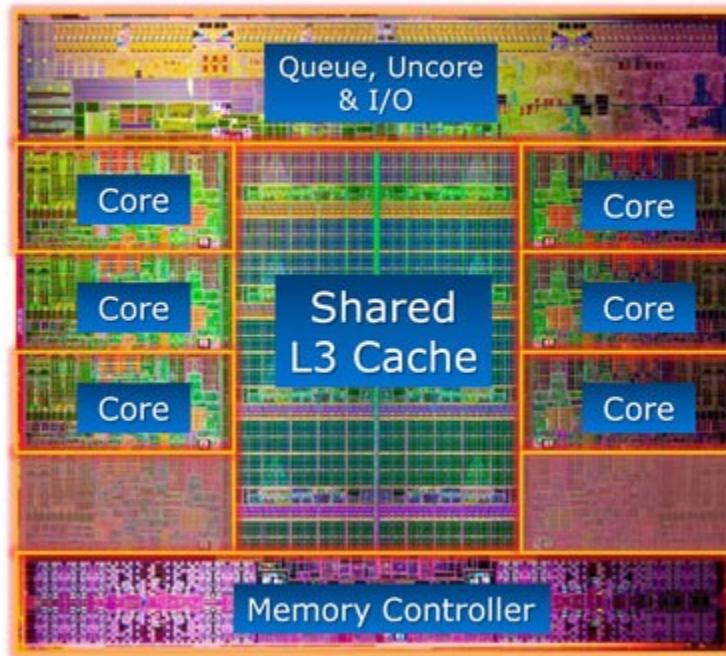
L1: 16KB I\$ + 16KB D\$

L2: 256KB

L3: 1.5MB – 9MB

12. Exploration: Trend: Intel Core i7-3960K

Intel® Core™ i7-3960X Processor Die Detail



Intel Core i7-3960K

per die:

- 2.27 billion transistors

- 15MB shared Inst/Data Cache (LLC)

per Core:

- 32KB L1 Inst Cache

- 32KB L1 Data Cache

- 256KB L2 Inst/Data Cache

- up to 2.5MB LLC