CS/ECE 5381/7381
Computer Architecture
Spring 2023

Dr. Manikas

Computer Science

Lecture 5: Feb 2, 2023

Assignments

- Quiz 3 – due Sat., Feb. 4 (11:59 pm)
  - Covers concepts from Module 3 (this week)

Quiz 3 Details

- The quiz is open book and open notes.
- You are allowed 90 minutes to take this quiz.
- You are allowed 2 attempts to take this quiz - your highest score will be kept.
  - Note that some questions (e.g., fill in the blank) will need to be graded manually
- Quiz answers will be made available 24 hours after the quiz due date.

Instruction Set Principles

(Appendix A, Hennessy and Patterson)

Note: some course slides adopted from publisher-provided material

Outline

- A.9 MIPS Architecture
  - MIPS Instruction Set
  - MIPS Processor Design

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  – h in $s2, base address of A in $s3

- Compiled MIPS code:
  – Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Immediate Operands

- Constant data specified in an instruction

  `addi $s3, $s3, 4`

- No subtract immediate instruction
  - Just use a negative constant

    `addi $s2, $s1, -1`

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - \$t0 – \$t7 are reg's 8 – 15
  - \$t8 – \$t9 are reg's 24 – 25
  - \$s0 – \$s7 are reg's 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

Example Applications

- How to represent text in computer?
  - ASCII: American Standard Code for Information Interchange
- Case conversions: upper to lower, lower to upper
  - Use AND, OR with bit masks

ASCII: Alphabetic codes

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	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	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

How to convert between cases?

Upper to lower: OR with 20H = 010 0000
Lower to upper: AND with 5FH = 101 1111

Example: A to a:

```
    100 0001
+   010 0000
------------
    110 0001
```

ori $t0, $t1, 0x20

z to Z

```
    111 1010
*   101 1111
------------
    101 1010
```

andi $t0, $t1, 0x5F

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Example A.9-1

Translating C to MIPS.

Assume that C variables are 32-bits and are assigned to MIPS registers as follows: h = $s1

Also assume that the base addresses of C arrays are stored in the following MIPS registers: A: $s2

What is the corresponding MIPS code for the following C statement?

```
A[6] = h;
```

Example A.9-2

What MIPS instruction does this bit string represent?

0010 0001 0000 1000 0000 0000 0000 0001

Example A.9-3

Given the following MIPS instruction:

```
sub $t0, $t1, $t2
```

Show the **binary**, then **hexadecimal**, representation of this instruction

Example A.9-4

Assume we have the following MIPS code, starting at memory address 1000H:

1000H	LOOP:	addi	\$t1, \$t1, 4
1004H		sub	\$t0, \$t3, \$t4
1008H		beq	\$t0, \$zero, NEXT
100CH		addi	\$t2, \$t2, -1
1010H		bne	\$t2, \$zero, LOOP
1014H	DONE:	sw	\$t0,16(\$s0)
1018H	NEXT:	add	\$s0, \$s1, \$t0

If we use PC relative addressing for loop, what is the constant for **NEXT** in the **beq** instruction?

Example A.9-5

Assume we have the following MIPS code, starting at memory address 1000H:

1000H	LOOP:	addi	$t1, $t1, 4
1004H		sub	$t0, $t3, $t4
1008H		beq	$t0, $zero, NEXT
100CH		addi	$t2, $t2, -1
1010H		bne	$t2, $zero, LOOP
1014H	DONE:	sw	$t0,16($s0)
1018H	NEXT:	add	$s0, $s1, $t0

If we use PC relative addressing for loop, what is the constant for **LOOP** in the **bne** instruction?

Outline

- A.9 MIPS Architecture
  - MIPS Instruction Set
  - <u>MIPS Processor Design</u>

Processor Units

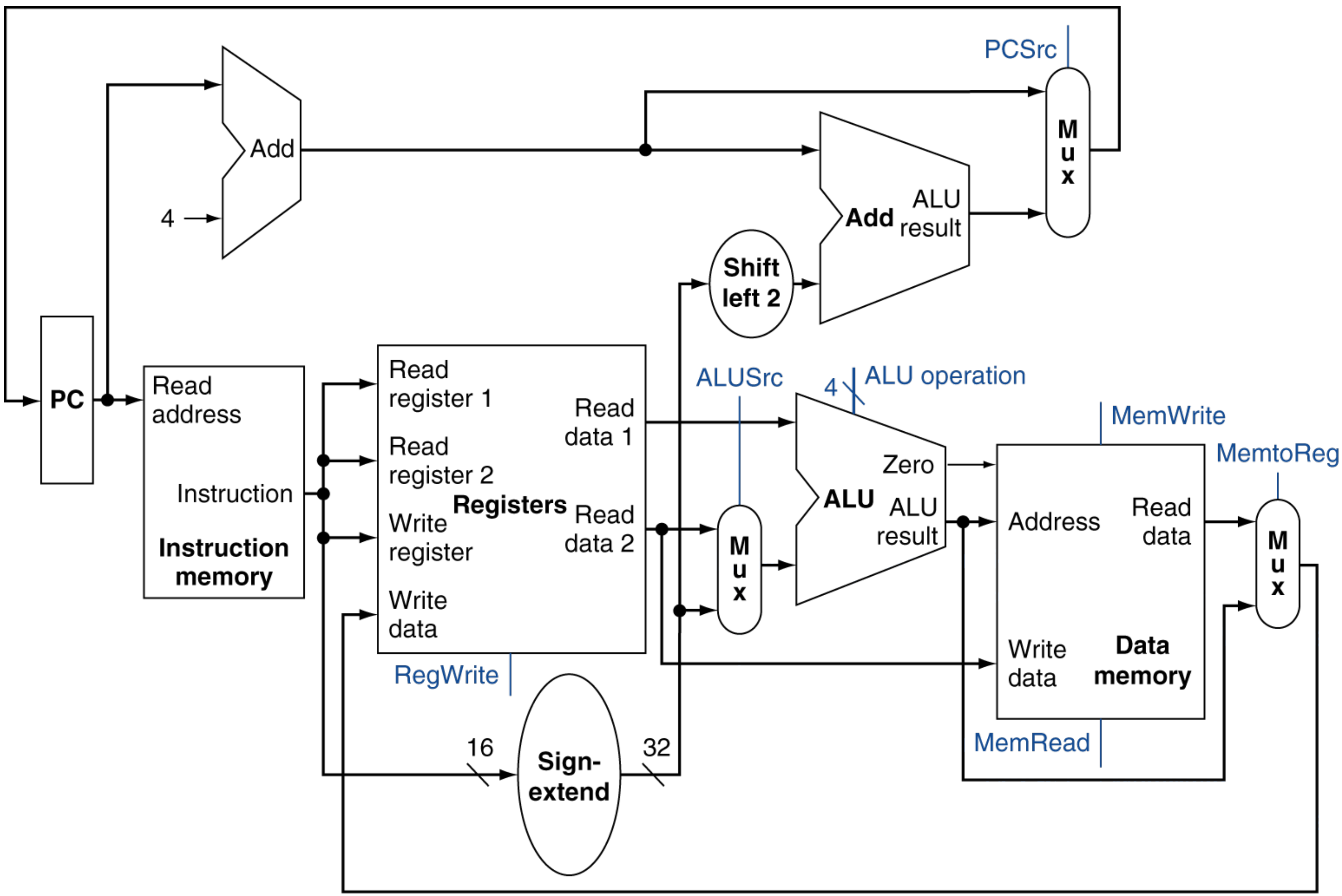
Datapath

Hardware path (components and interconnections) for instructions and data during program execution

Control Unit

Controls operations of datapath units – flow of data in datapath

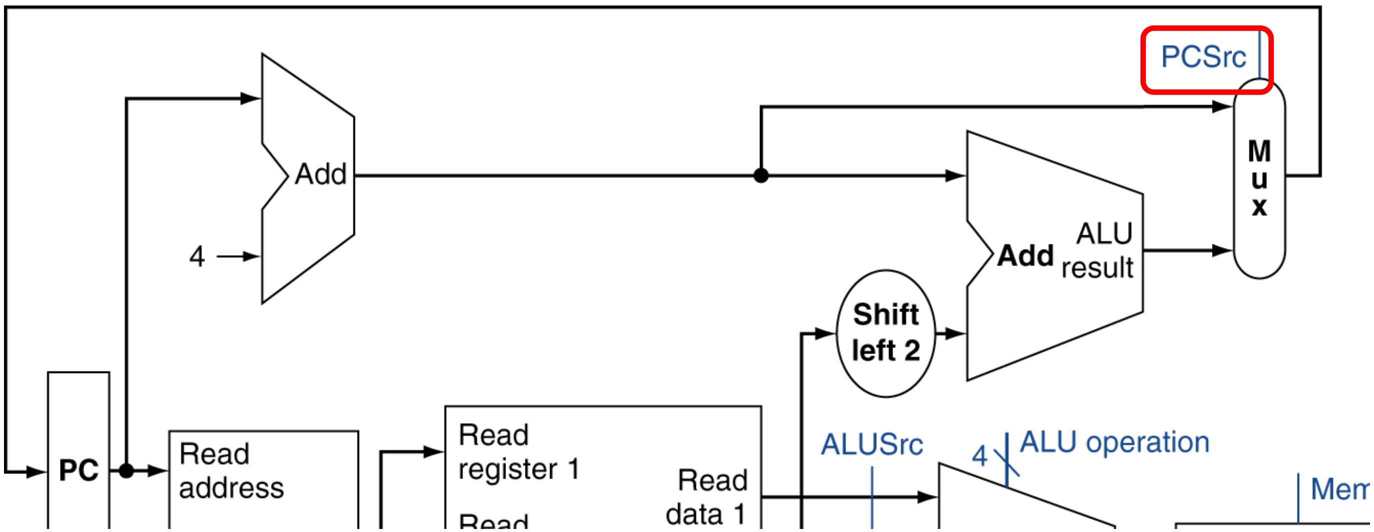
MIPS Datapath



Datapath Control Signals

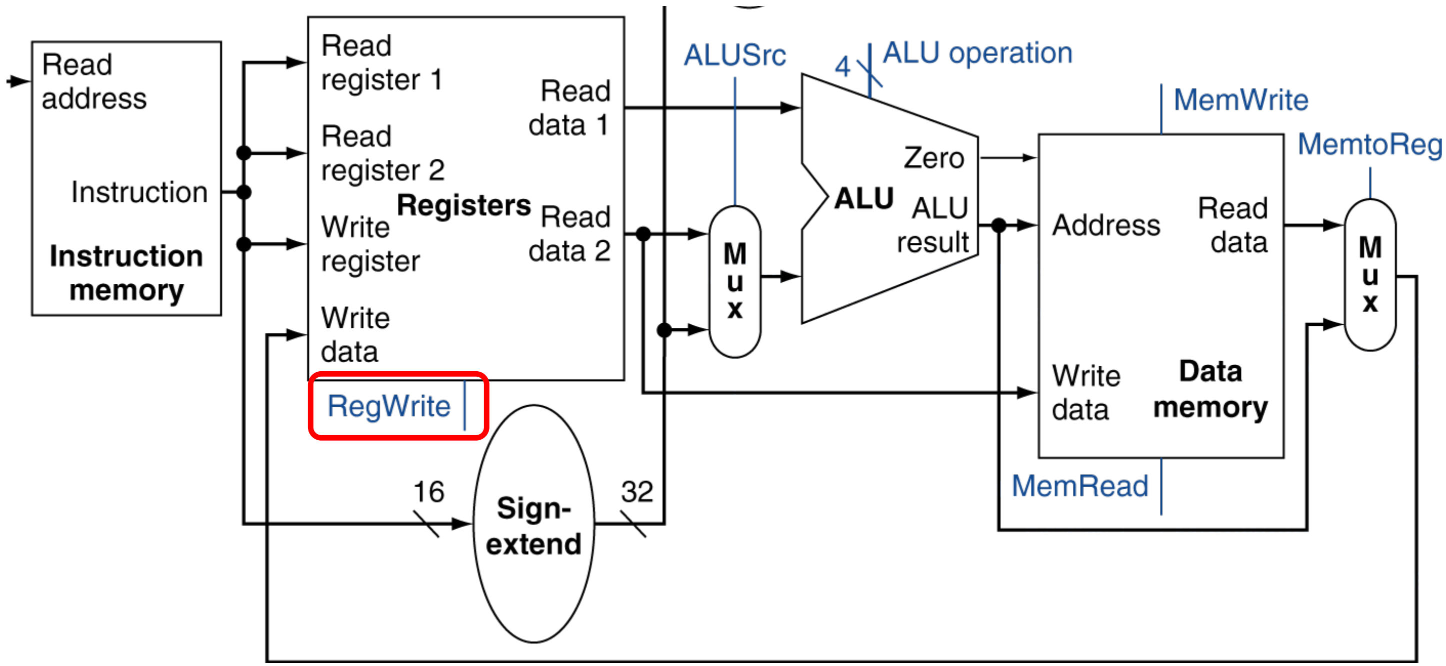
- What are our basic control signals?
  - PCSrc
  - RegWrite
  - ALUSrc
  - MemWrite
  - MemRead
  - MemtoReg
  - ALU operation (4 bits)

PCSrc



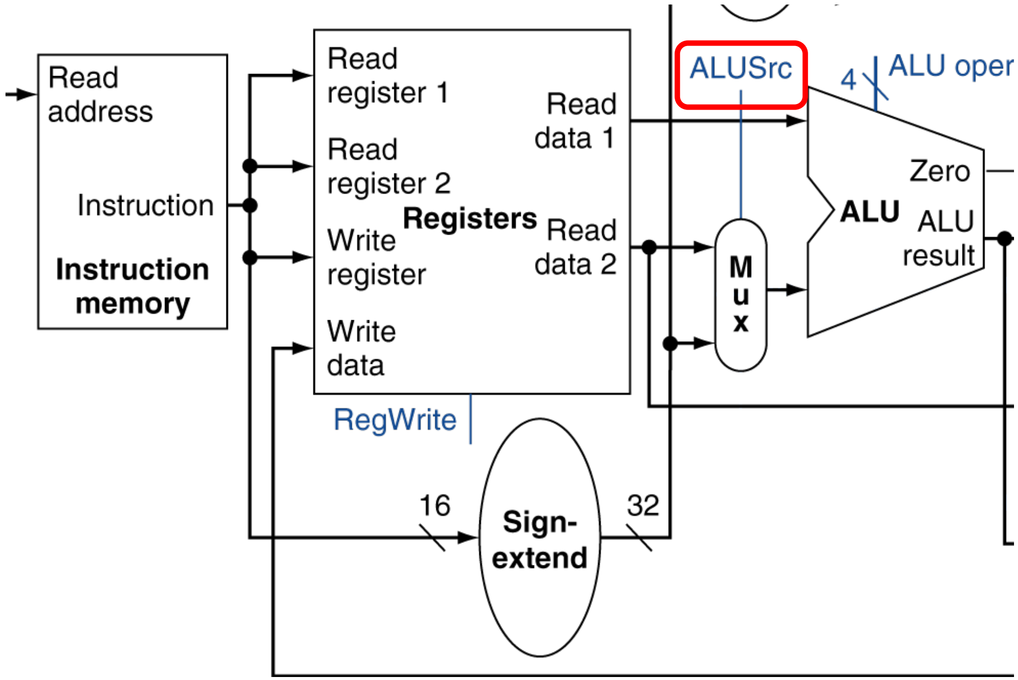
Value	Effect
0	PC <= PC + 4
1	PC <= PC + 4 + offset

RegWrite



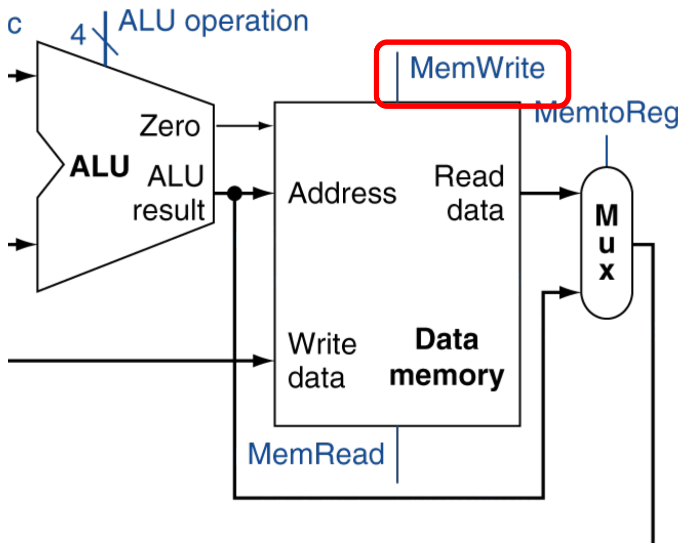
Value	Effect
0	none
1	R[Write register] <= Write data

ALUSrc



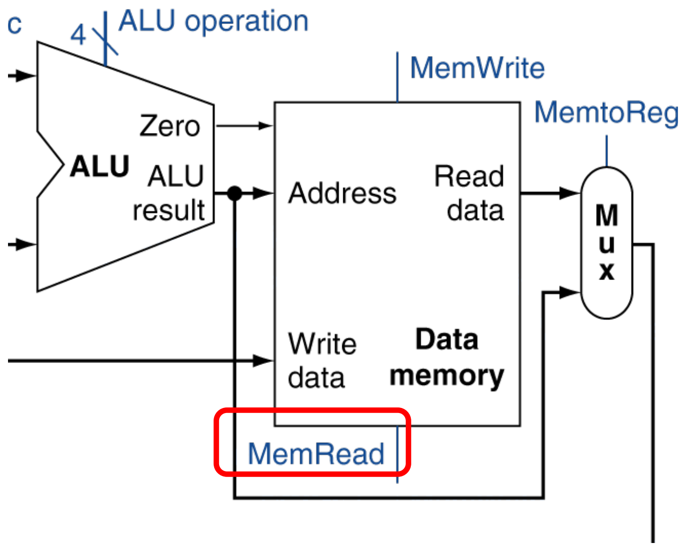
Value	Effect
0	2nd ALU operand <= Read data 2
1	2nd ALU operand <= Sign-extended constant

MemWrite



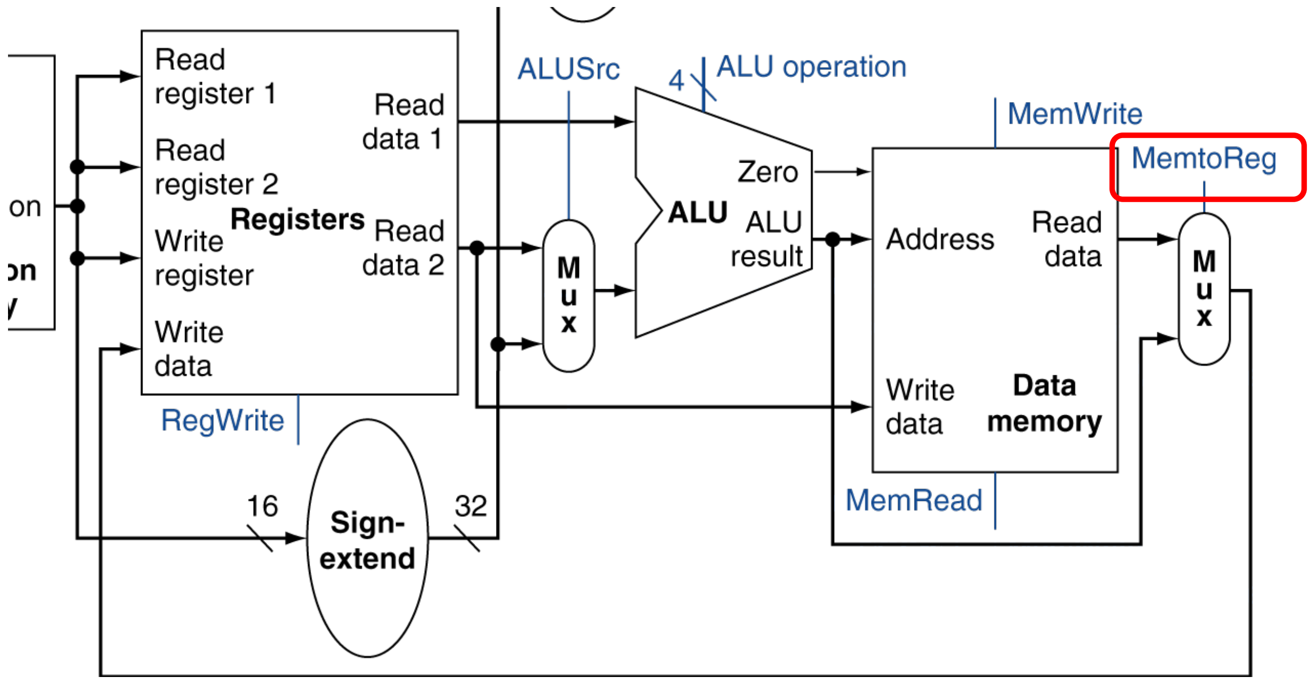
Value	Effect
0	none
1	Mem[Address] <= Write data

MemRead



Value	Effect
0	none
1	Read data <= Mem[Address]

MemtoReg



Value	Effect
1	Write data <= Read data
0	Write data <= ALU result