CS161 – Design and Architecture of Computer Systems

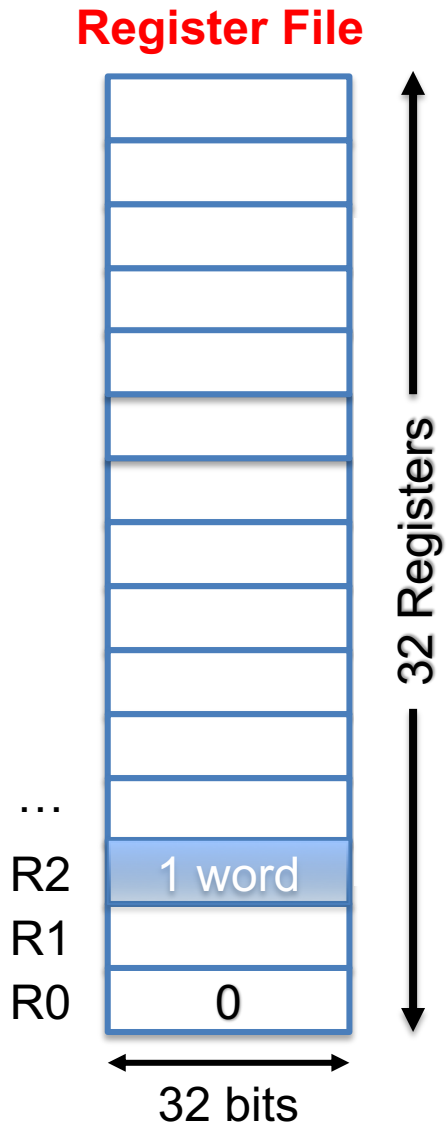
Week 2 - Discussion

*some slides adapted from:
Prof Daniel Wong UCR – EE/CS)

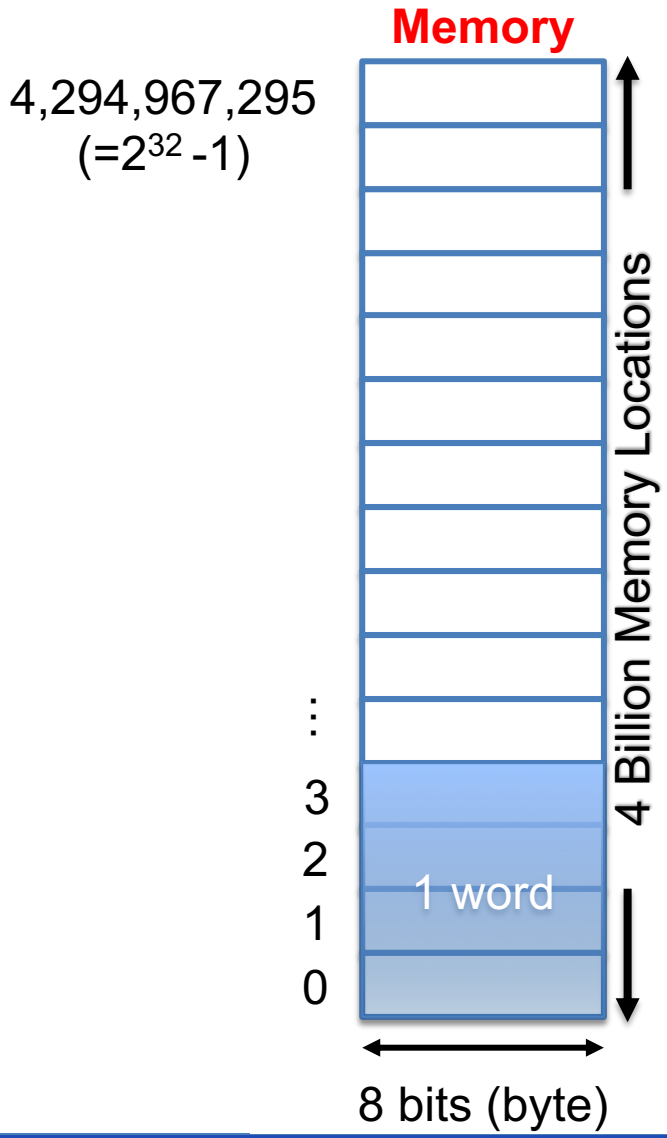
MIPS Registers

- 32 × 32-bit registers (0-31)
- Assembly register names

Name	Register number	Usage
$zero	0	The 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	Saved
$t8–$t9	24–25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

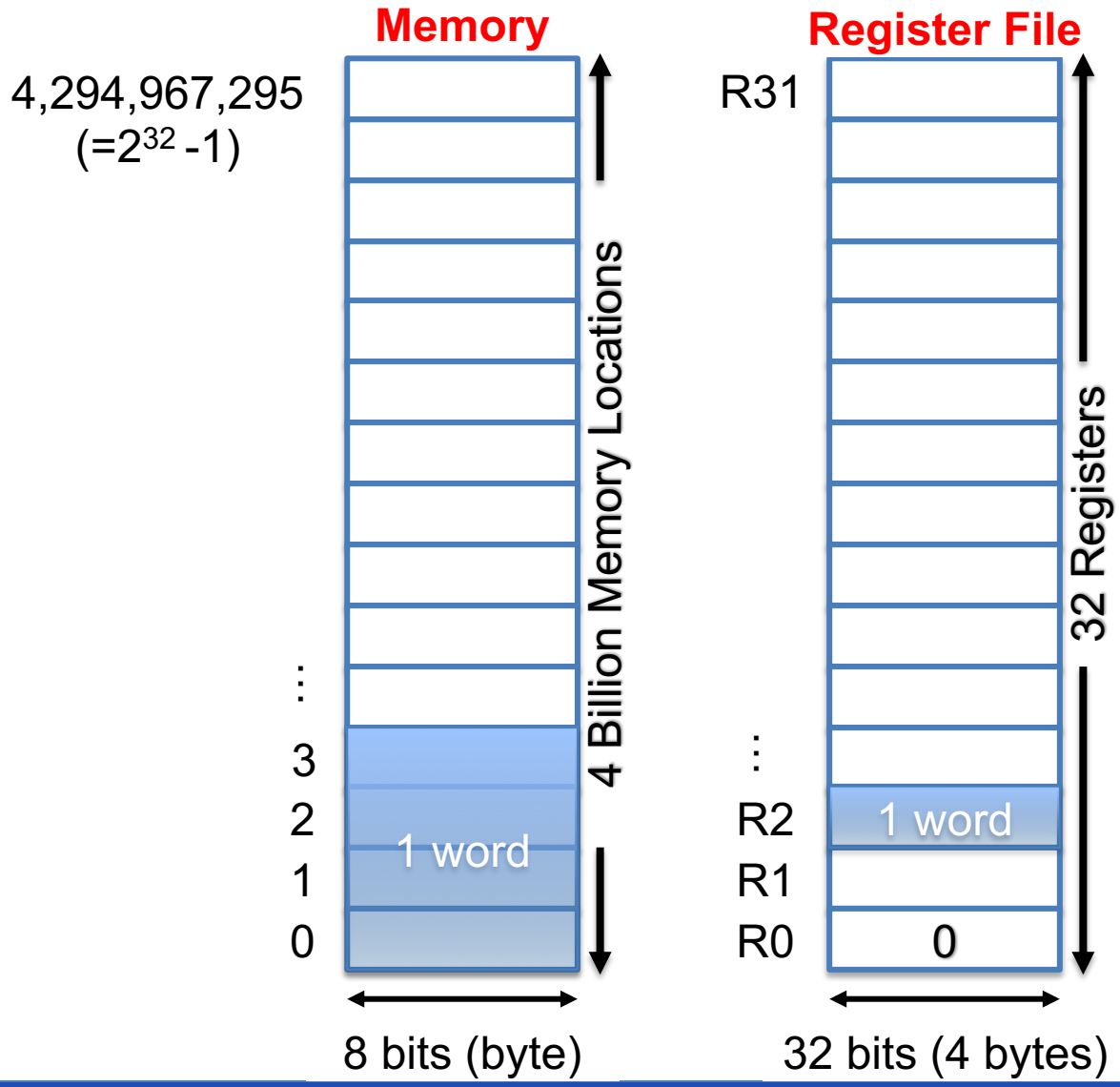


Memory



- 4-byte aligned:
  - each location is one byte
  - 4 bytes (32 bits) = word
  - 4 locations per instruction

Memory and Register File



- only *load/store* access memory
- 32-bit data called a "word"
- 4 bytes/instruction
- 4-byte aligned

Memory Operand Example

- C code:
  - $s2 = h, $s3 = base address of array A

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

- MIPS code:
  - index 8 → offset of 32
  - index 12 → offset of 48

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

MIPS Assembly Instructions

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

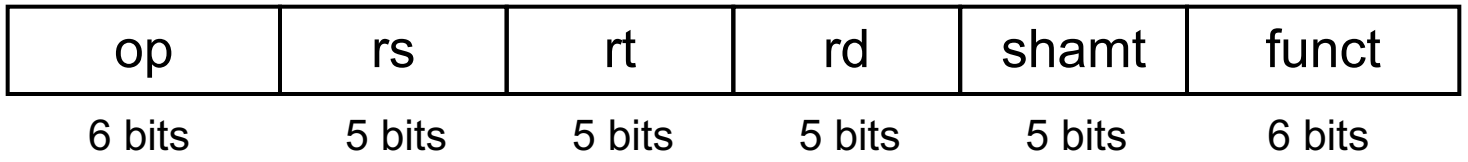
MIPS Instruction Format

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

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

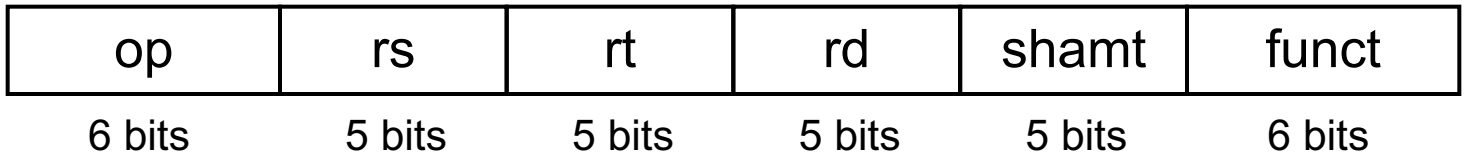
- op: operation code
- rs: src register
- rt: src register (R-type) / dst register (I-type)
- rd: dst register
- shamt: shift amount (shift left logical – sll amount)
- funct: function code (extends opcode)

R-format Example: *add*

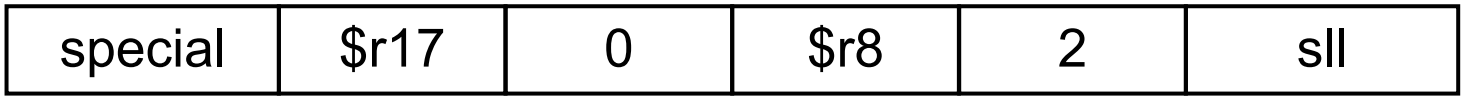



```
add $r8, $r17, $r18 (add rd, rs, rt)
```

R-format Example: *sll*




```
sll $r8, $r17, 2 (sll rd, rs, shamt)
```



- shamt: shift amount (# positions to shift)
- Shift Left Logical (sll)
  - `sll` by *i* bits multiplies by 2 x i
  - i.e., above 2 x 2 =4

sll Example

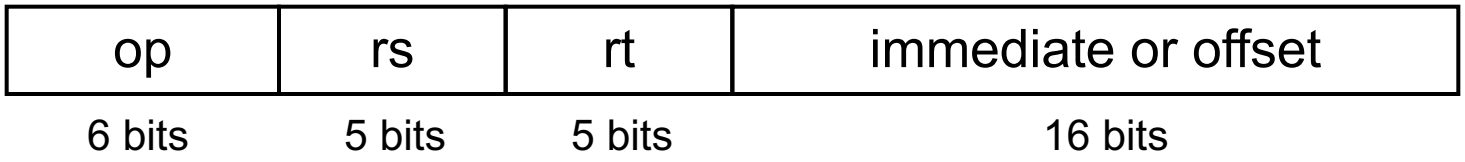
- C code:
  - $s0 = i, $s1 = j, $s2 = base address of array A

```
j = A[i];
```

- MIPS code:
  - index i → offset of i x 4

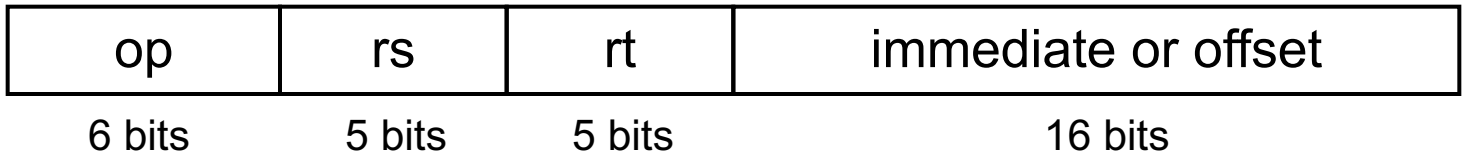
```
sll $t0, $s0, 2     # $t0 = i x 4
add $t0, $t0, $s2   # $t0 = &A[i]
lw  $s1, 0($t0)     # $s1 = A[i]
```

MIPS I-format Instructions

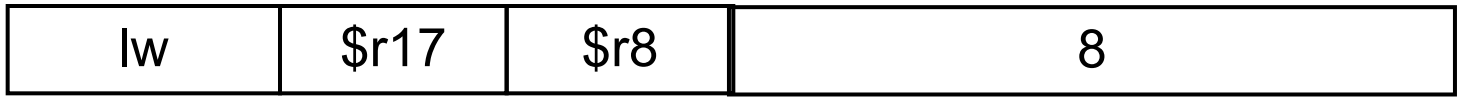


- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to + ($2^{15} - 1$), used as *immediate*
  - Address: offset added to base address in rs

I-format Example: *load*

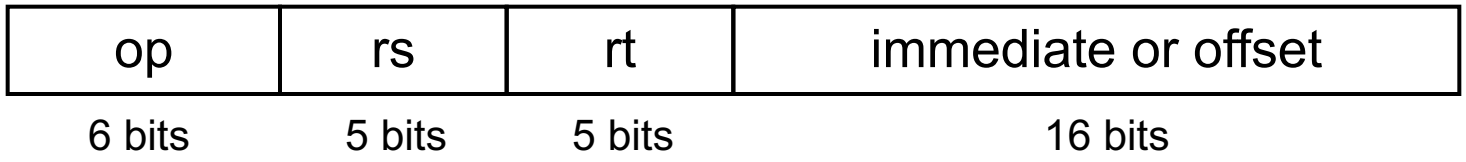



```
lw $r8, 8($r17),  $r8 ← Mem[8+$r17]
```

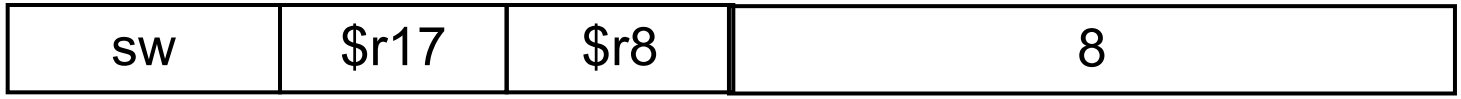


- rs: src register
- rt: dst register

I-format Example: *store*

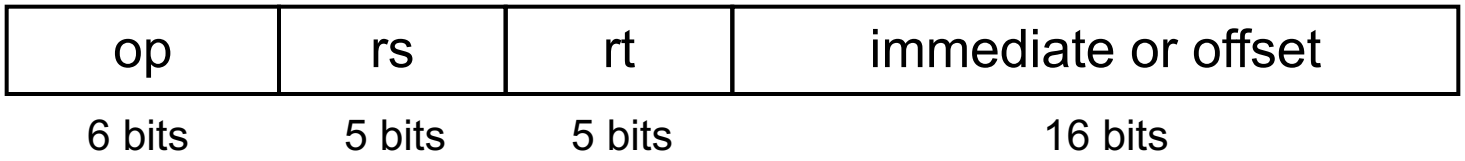



```
sw $r8, 8($r17), Mem[8+$r17] ← $r8
```

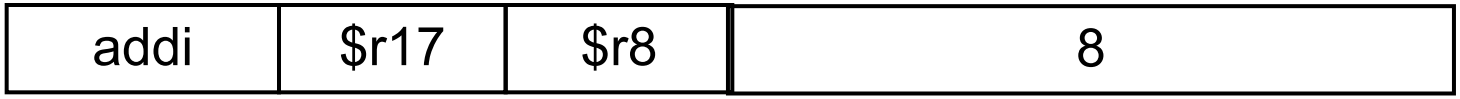


- rs: src register
- rt: src register

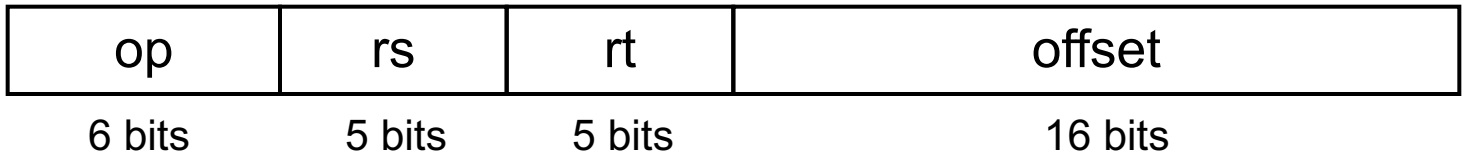
I-format Example: *addi*



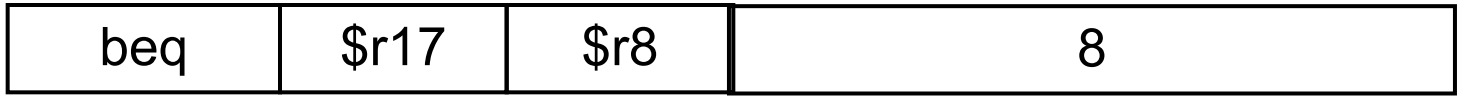

```
addi $r8,$r17, 8    $r8 ← 8+$r17
```



I-format Example: *beq*




```
beq $r8,$r17, 8    if ($r8==$r17)
```
PC ← PC+4 + 8x4



- Most branch targets near branch
- PC-relative addressing
  - Target address = PC + offset x 4
  - (PC already incremented by 4 by this time)

If Statements (branches)

- C code:
  - $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j

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

- MIPS code:

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

More Conditional Operations

- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
L
```

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

$zero	0000 0000 0000 0000 0000 0000 0000 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Amdahl's Law

- One improvement at a time:

$$speedup_{overall} = \frac{1}{(1 - fract_{enh}) + \frac{fract_{enh}}{speedup_{enh}}}$$

- Multiple improvements at once:

$$speedup_{overall} = \frac{1}{\left(1 - fract_{enh(total)}\right) + \frac{fract_{enh(1)}}{speedup_{enh(1)}} + \frac{fract_{enh(2)}}{speedup_{enh(2)}} + \cdots}$$

Instructions Per Cycle (IPC)

- Cycles Per Instruction (CPI)

$$CPI = \frac{cycles}{instruction}$$

- Instructions Per Cycle (IPC)

$$IPC = \frac{instructions}{cycle}$$

- Relationship:

$$IPC = \frac{1}{CPI} \qquad CPI = \frac{1}{IPC}$$

Practice Problems

1) Convert the following C code to MIPS assembly. Assume ‘i’ is held in $s0 and ‘j’ is held in $s1

```
if (i > j) {
        j++;
}
else if (i == j ) {
        i++;
}
else {
         i = j+i
}
```

2) Convert the following MIPS assembly to C code, where $s0, $s1, and $s2 are variables i, j, and k, respectively and $s3 is a pointer to an array A (i.e., the base address of array A)

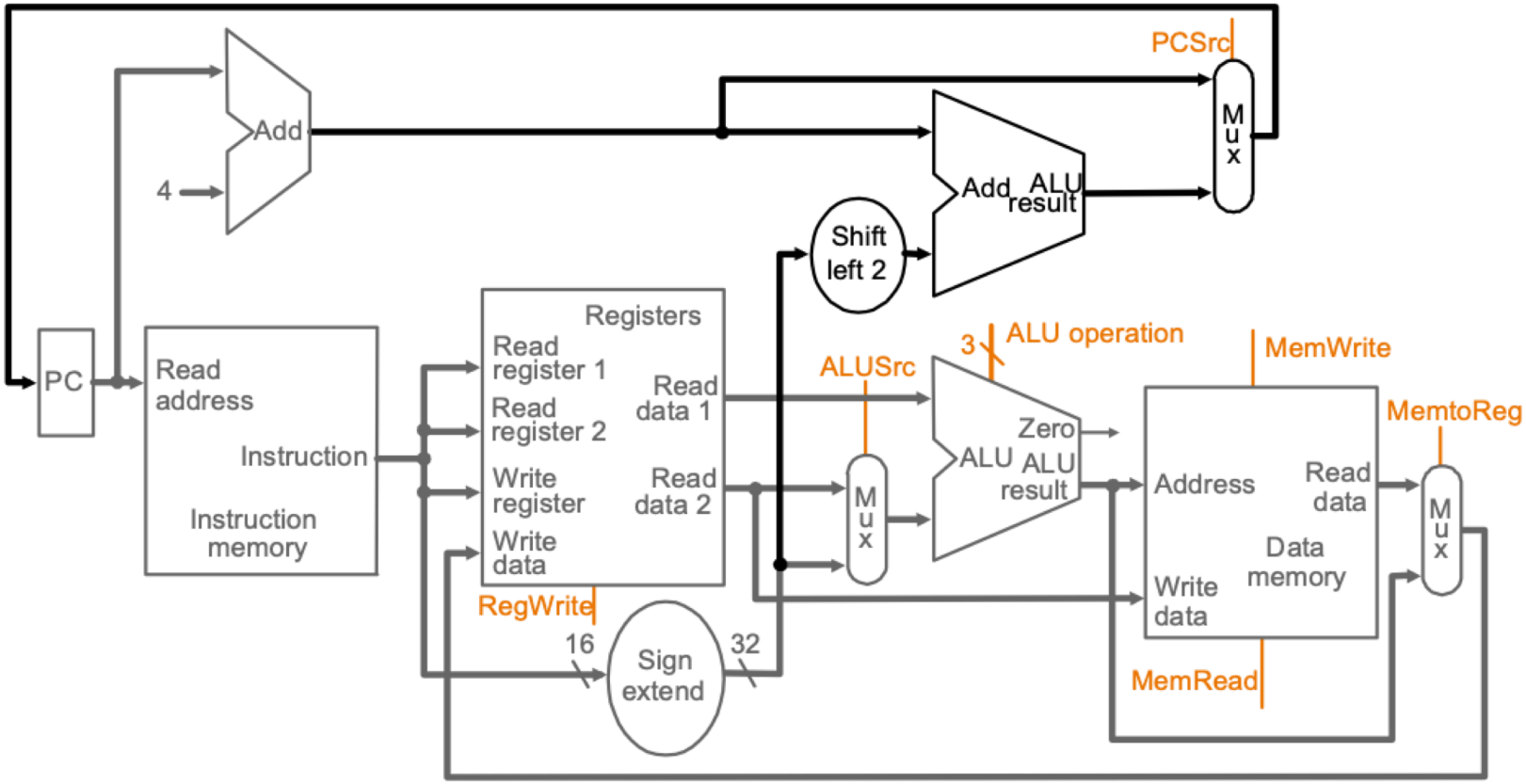
```
sll  $t0, $s0, 2
add  $t0, $t0, $s3
lw   $s1, 0($t0)
addi $s0, $s0, 1
sll  $t0, $s0, 2
add  $t0, $t0, $s3
lw   $s2, 0($t0)
add  $t0, $s1, $s2
sw $t0, 12($s3)
```

CS161 – Design and Architecture of Computer Systems

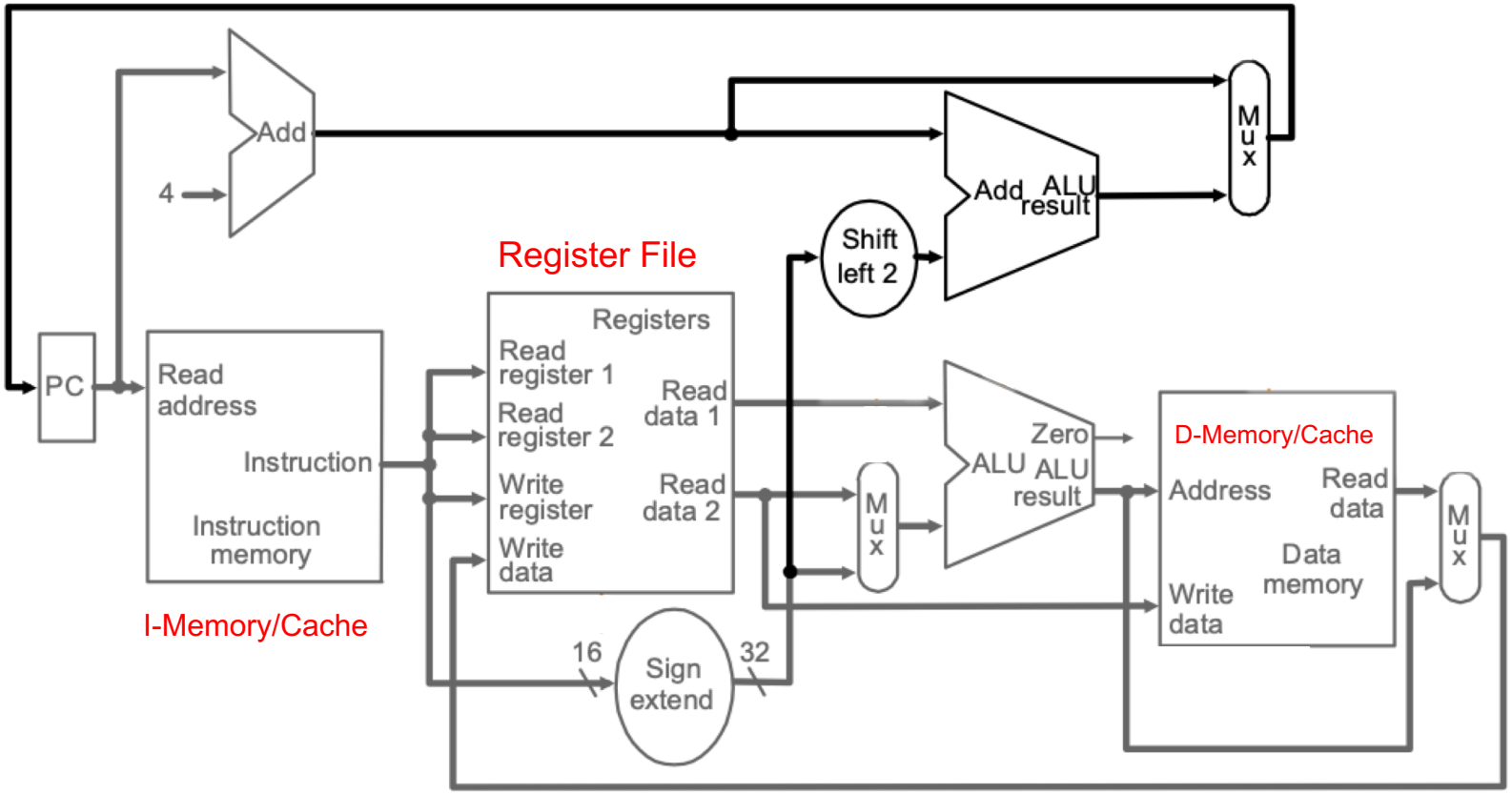
CHP 3

UNIVERSITY OF CALIFORNIA, RIVERSIDE

Datapath

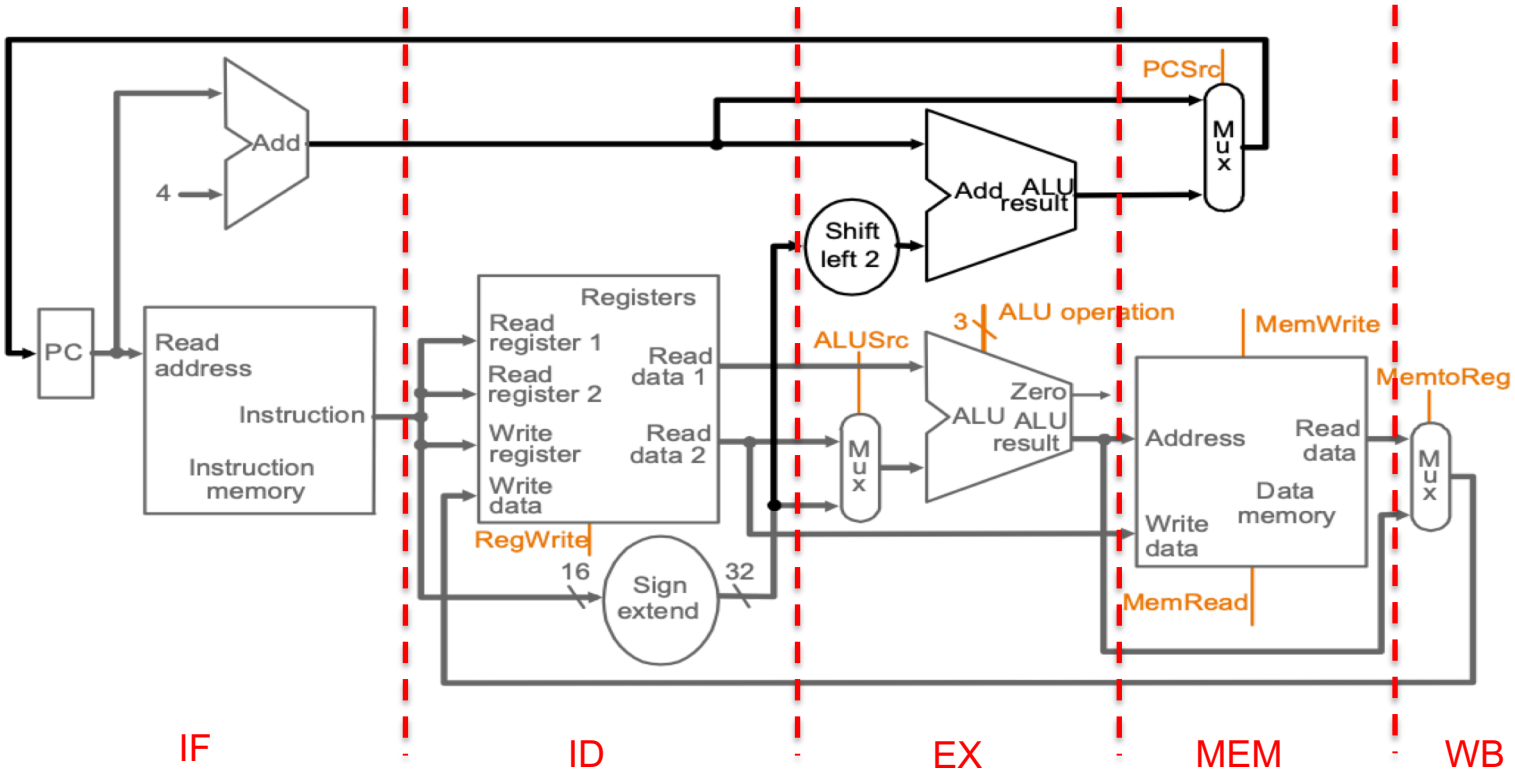

PCSrc
Add
4
M
u
x
Add
ALU result
Shift left 2
Registers
PC
Read address
Instruction
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
RegWrite
ALUSrc
3
ALU operation
MemWrite
MemtoReg
Zero
ALU
ALU result
Address
Read data
Data memory
Write data
MemRead
16
Sign extend
32

Datapath

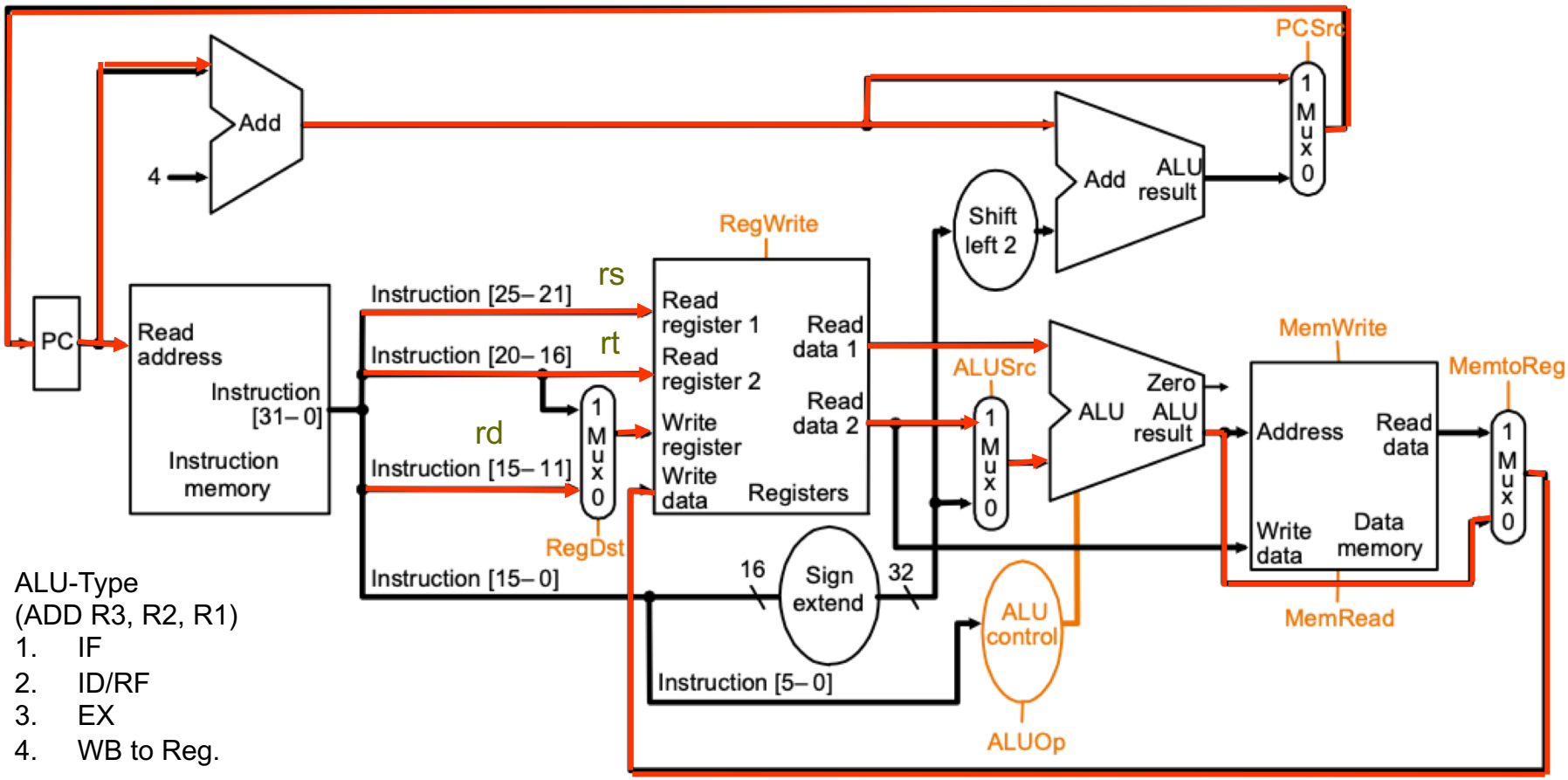

Add
4
M
u
x
Add
ALU result
Shift left 2
Register File
Registers
PC
Read address
Instruction
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
I-Memory/Cache
M
u
x
Zero
ALU
ALU result
D-Memory/Cache
Address
Read data
Data memory
Write data
M
u
x
16
Sign extend
32

Instruction Flow

- IF - Instruction fetch
- ID - Instruction decode and register operand fetch
- EX - Execute/Evaluate memory address
- MEM - Memory operand fetch
- WB - Store/writeback result



R-type Instructions – ADD Instruction



ALU-Type
(ADD R3, R2, R1)

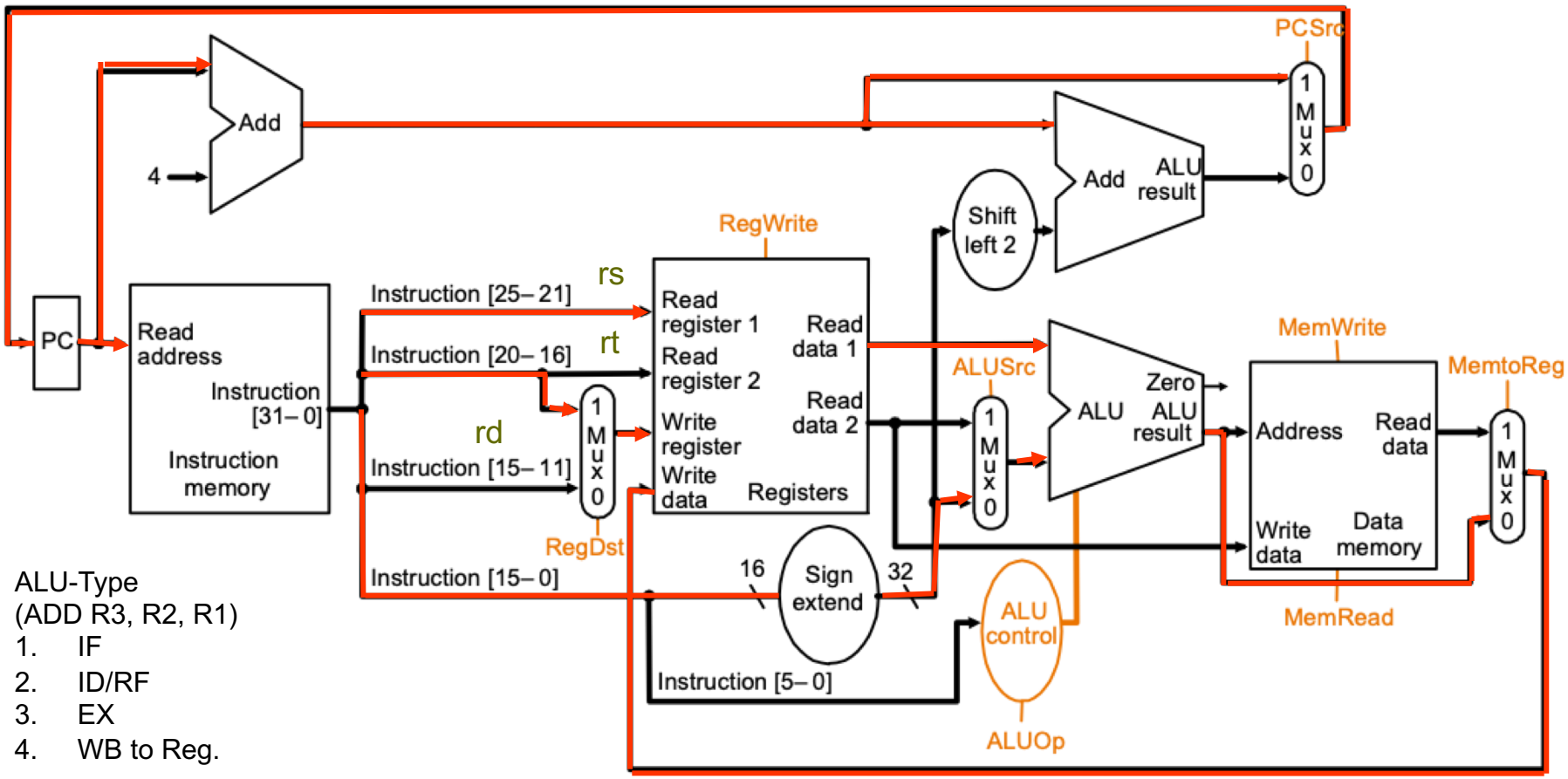
1. IF
2. ID/RF
3. EX
4. WB to Reg.

if MEM[PC] == ADD rd rs rt
GPR[rd] ← GPR[rs] + GPR[rt]
PC ← PC + 4

0	rs	rt	rd	0	ADD
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

ADD rd, rs, rt

I-type Instructions - ADDI Instruction



ALU-Type
(ADD R3, R2, R1)

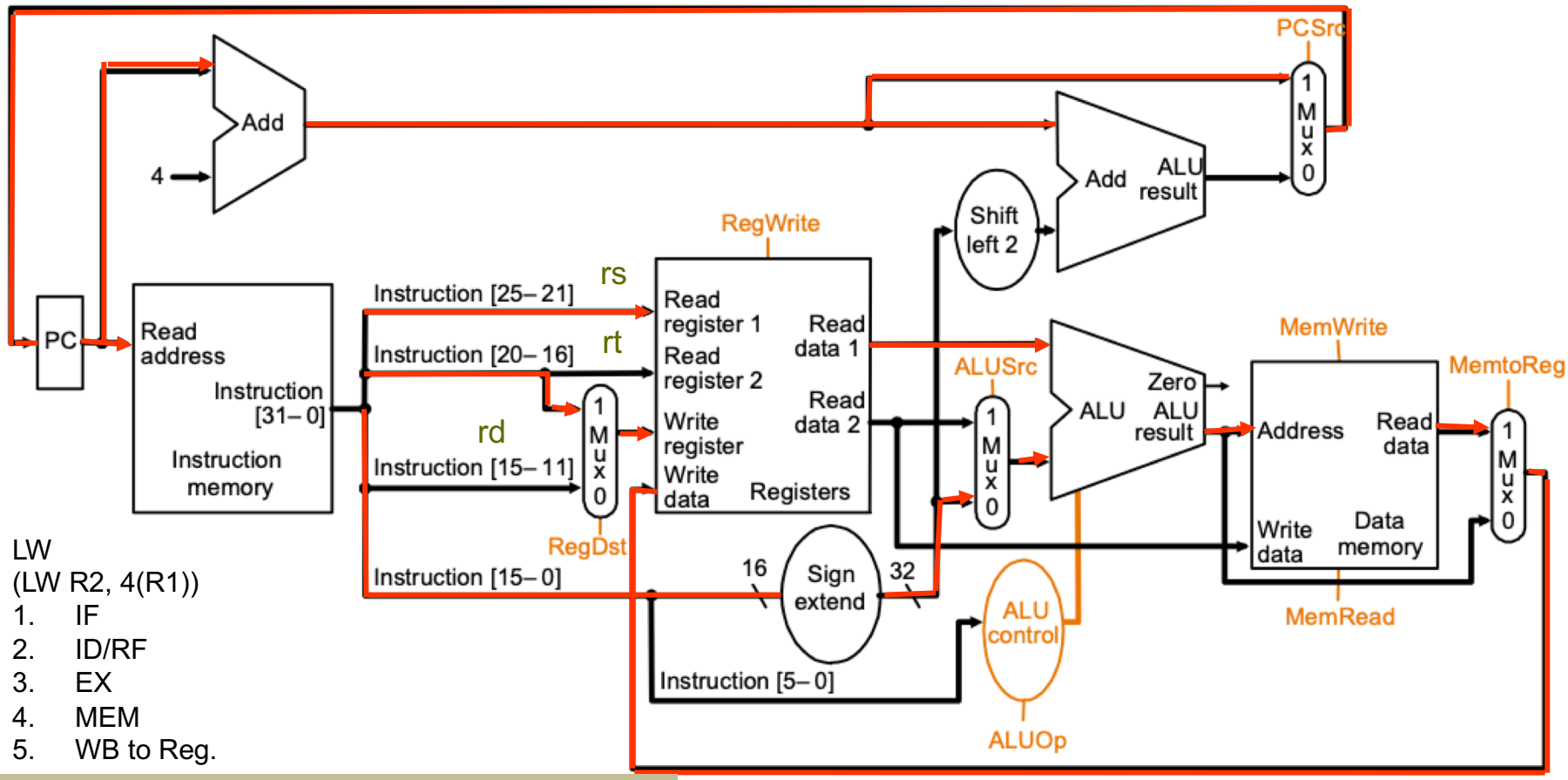
1. IF
2. ID/RF
3. EX
4. WB to Reg.

if MEM[PC] == ADDI rt rs immediate
GPR[rt] ← GPR[rs] + sign-extend (imm)
PC ← PC + 4

ADDI	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

ADDI rt, rs, immediate

I-type Instructions - LW Instruction



LW
(LW R2, 4(R1))

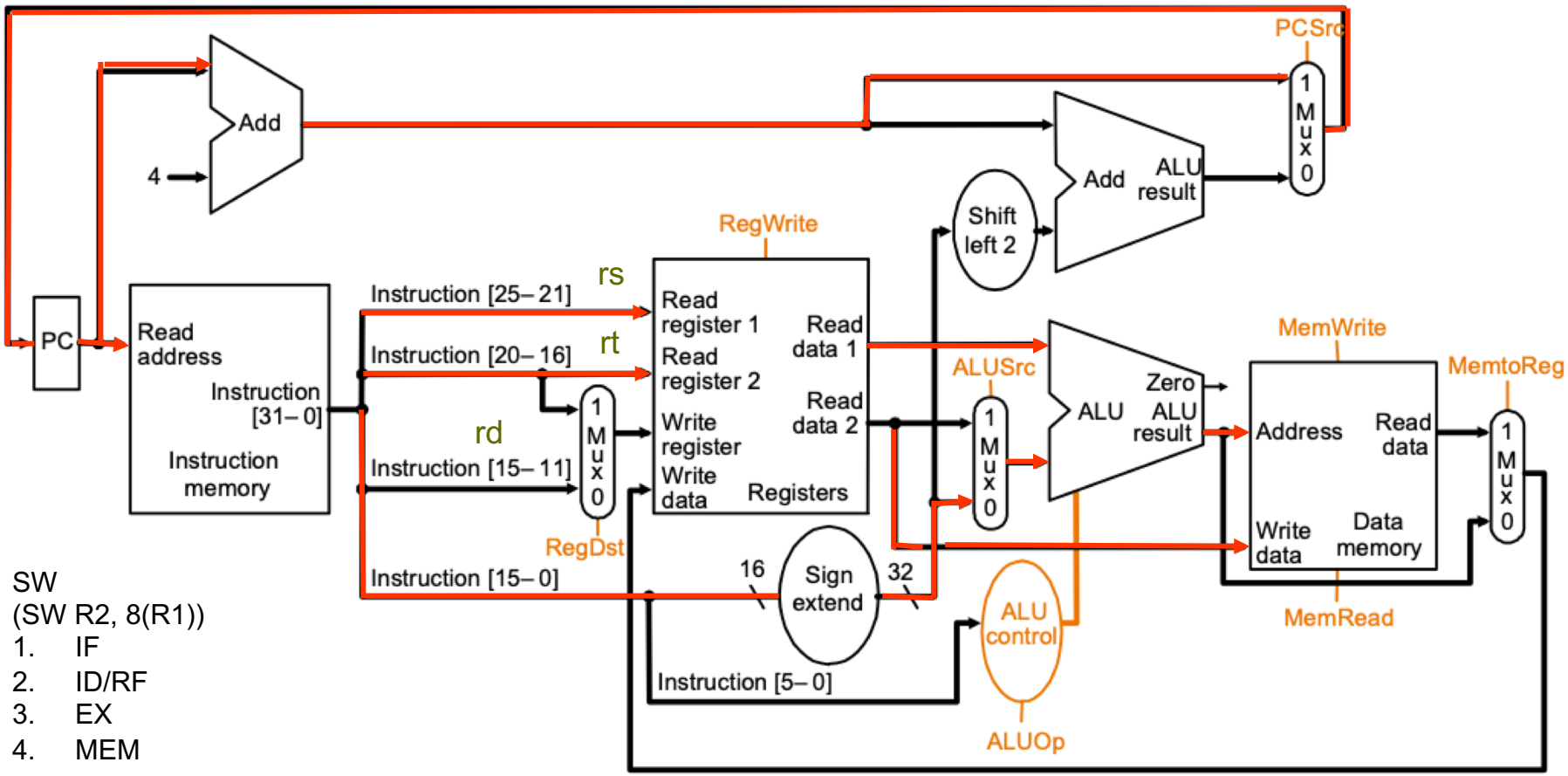
1. IF
2. ID/RF
3. EX
4. MEM
5. WB to Reg.

if MEM[PC]==LW rt offset$_{16}$ (base)
Addr = sign-extend(offset) + GPR[base]
GPR[rt] ← MEM[Addr]
PC ← PC + 4

LW	rs	rt	offset
6-bit	5-bit	5-bit	16-bit

LW rt, offset(rs)

I-type Instructions - SW Instruction



SW
(SW R2, 8(R1))

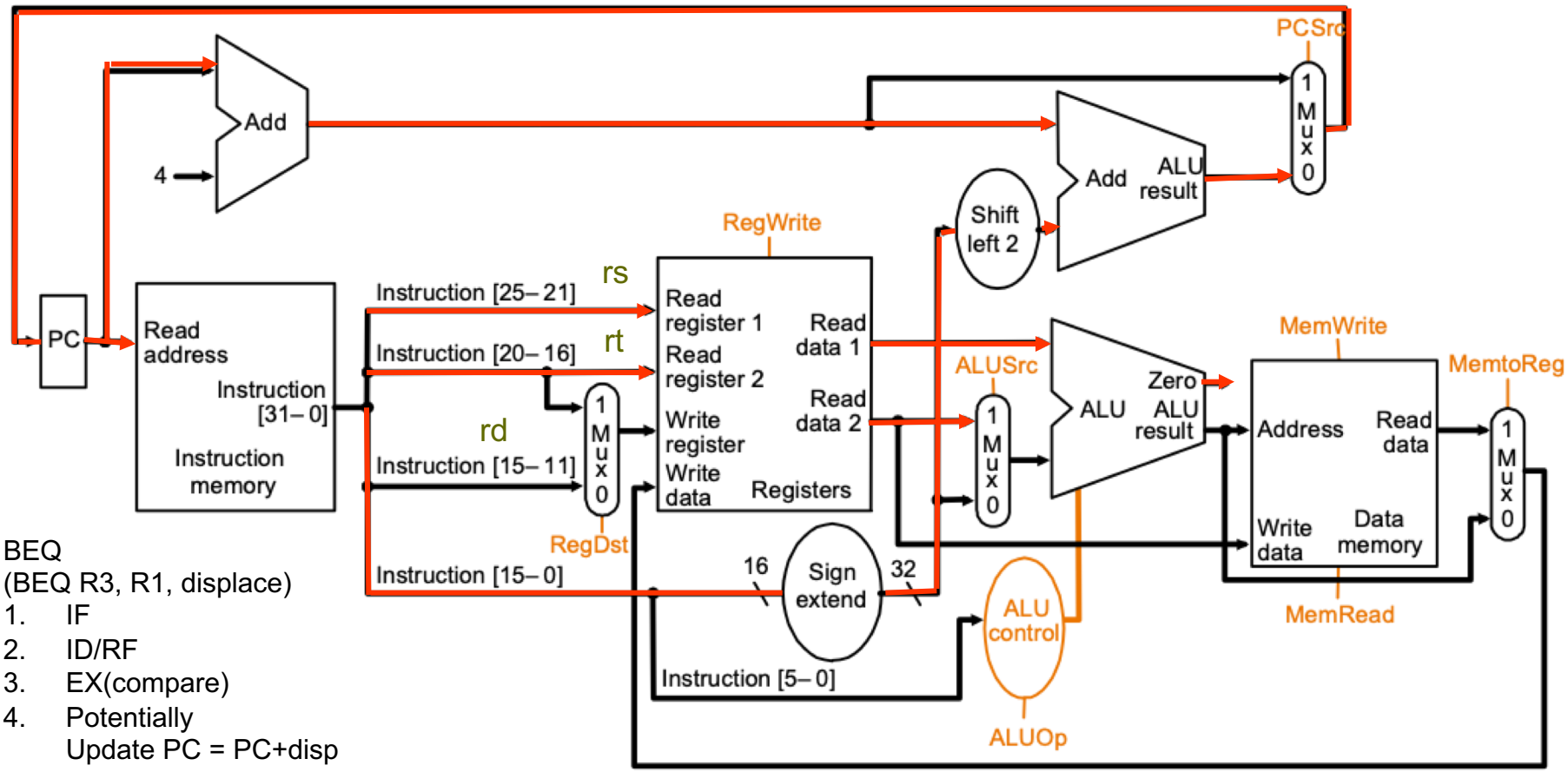
1. IF
2. ID/RF
3. EX
4. MEM

if MEM[PC]==SW rt offset$_{16}$ (base)
Addr = sign-extend(offset) + GPR[base]
MEM[Addr] ← GPR[rt]
PC ← PC + 4

SW	rs	rt	offset
6-bit	5-bit	5-bit	16-bit

SW rt, offset(rs)

I-type Instructions - Branch Instructions



BEQ
(BEQ R3, R1, displace)

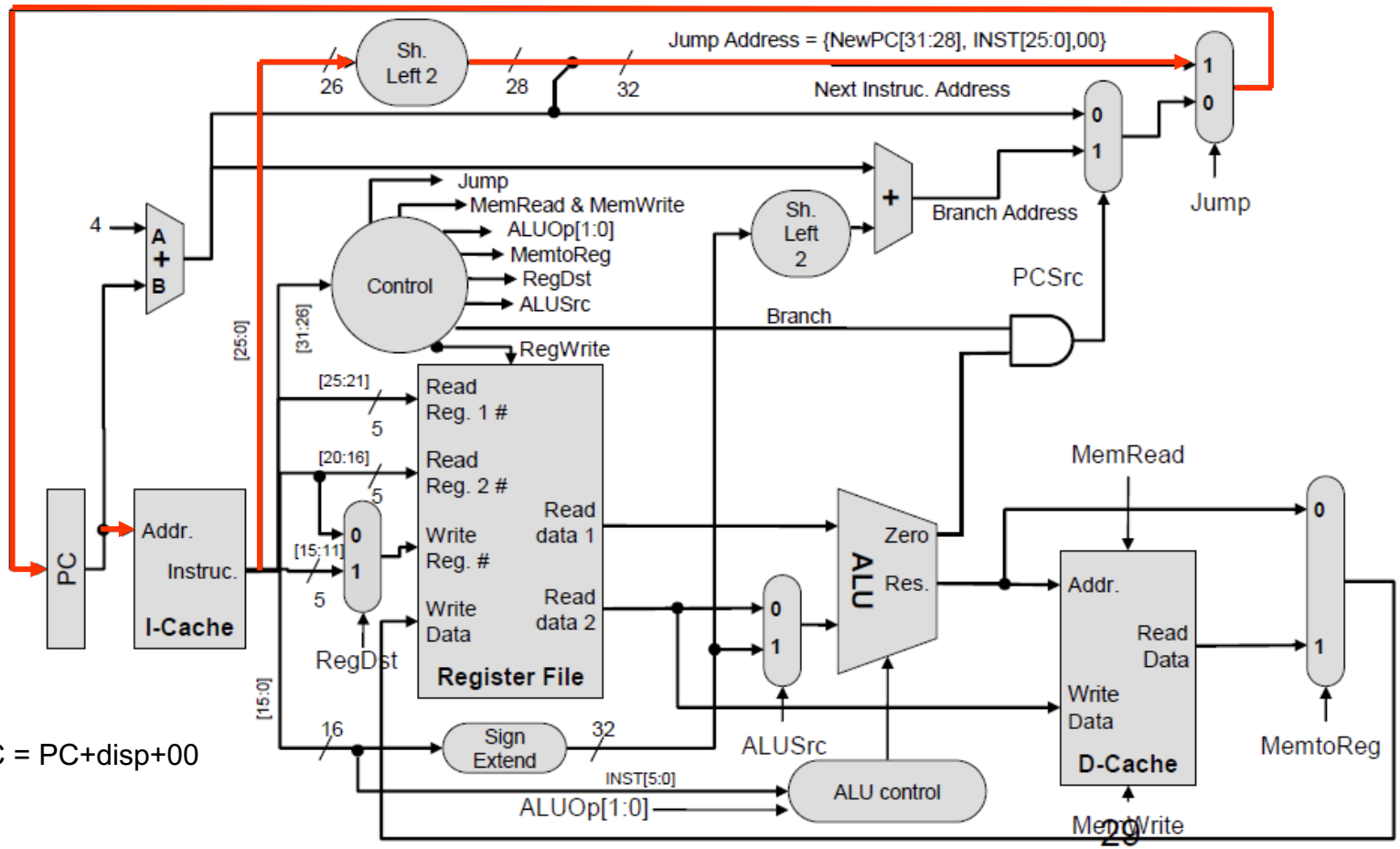
1. IF
2. ID/RF
3. EX(compare)
4. Potentially
   Update PC = PC+disp

if MEM[PC]==BEQ rs rt immediate$_{16}$
target = PC + 4 + sign-extend(immediate) x 4
if (GPR[rs]==GPR[rt]) PC ← target
else PC ← PC + 4

BEQ	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

BEQ rs, rt, immediate

J-type Instructions - Jump Instructions



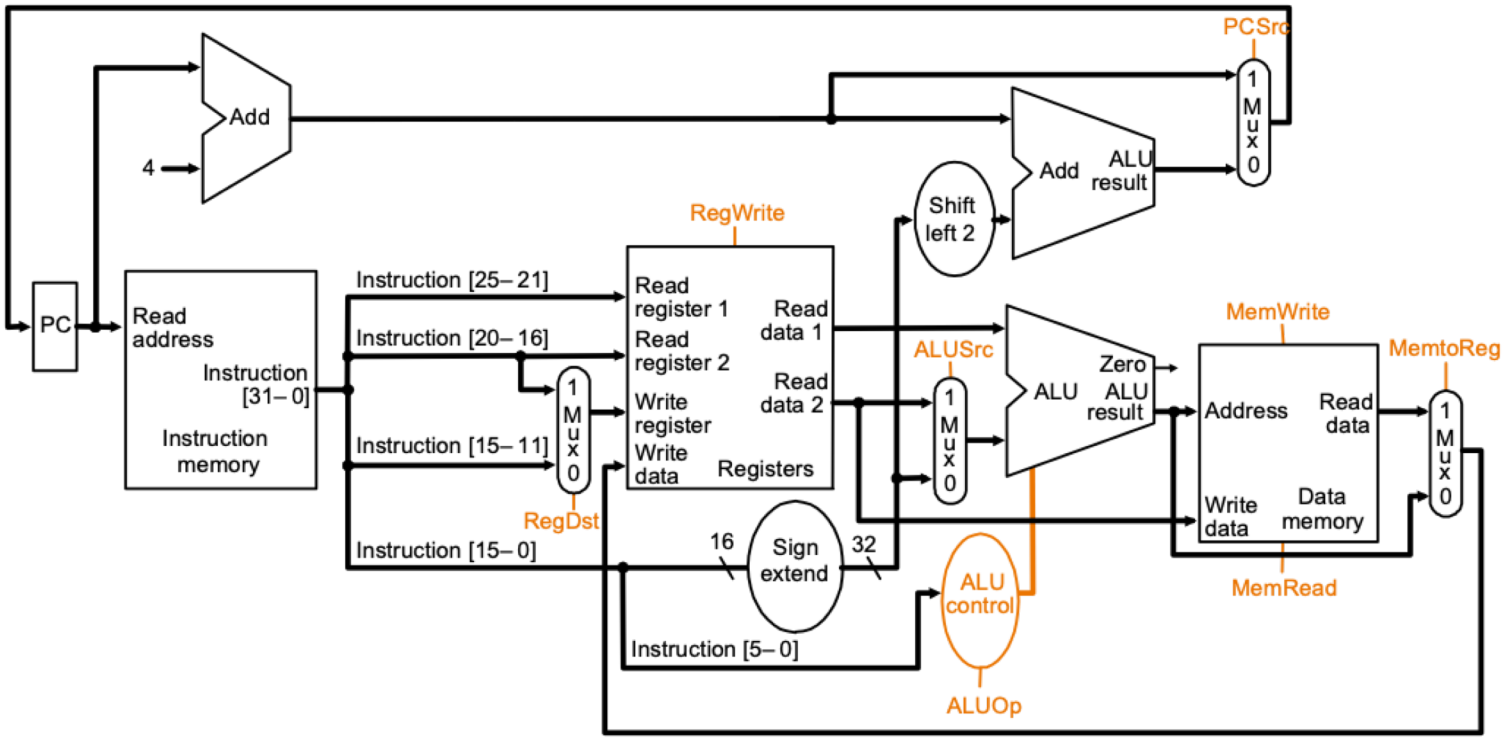
J
(J displace)
1. IF
2. ID/RF
3. Update PC = PC+disp+00

if MEM[PC]==J immediate$_{26}$
target = { PC[31:28], immediate$_{26}$, 2'b00 }
PC ← target

J	immediate
6-bit	26-bit

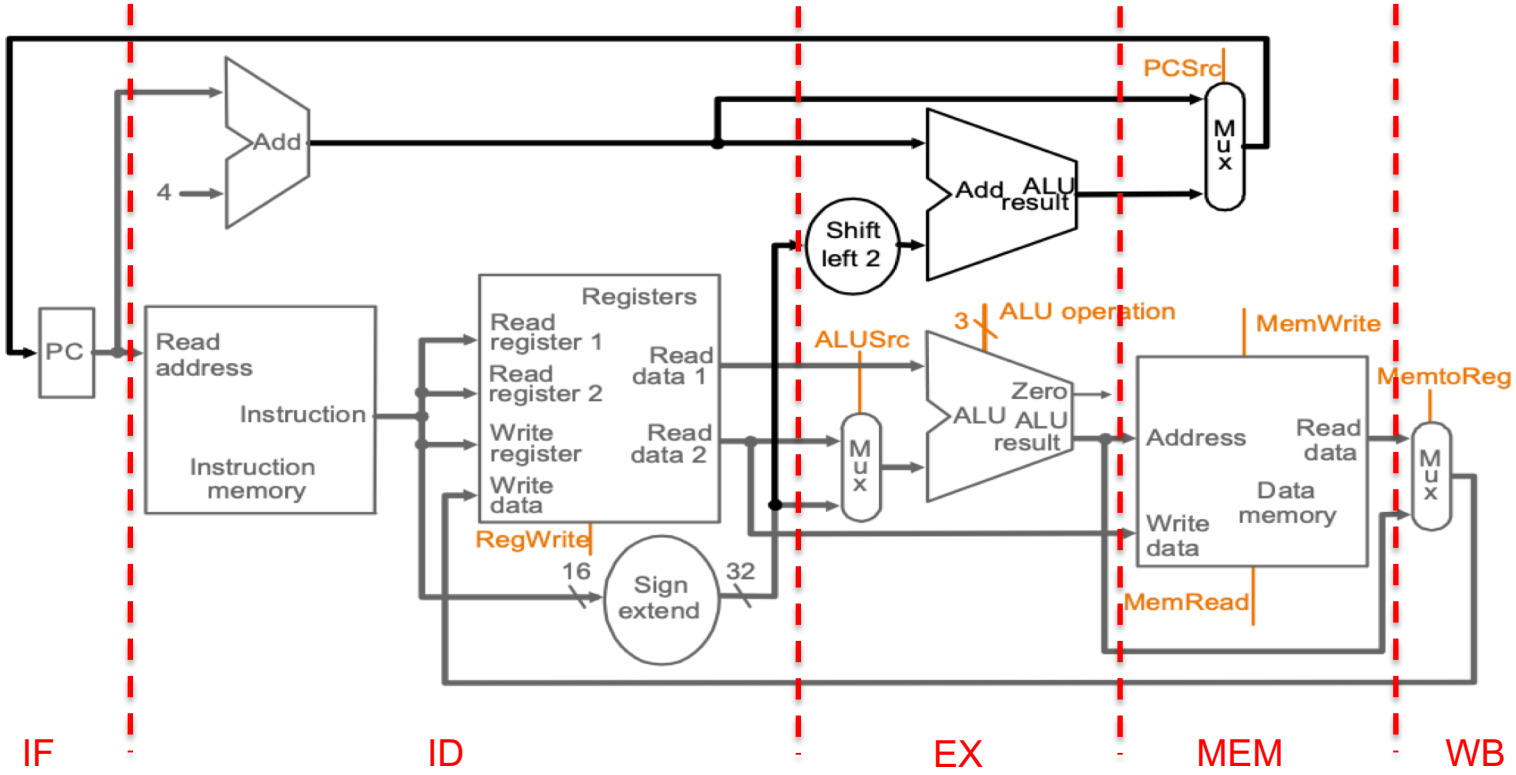
Single Cycle Implementation

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



Instruction Flow

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



Single Cycle – How long is the cycle?

Inst. Type	Inst. Mem.	Reg. File (read)	ALU (s)	Data Mem.	Reg. File (write)	Total	Inst. %
R-type	2	1	2	0	1	6 ns	44
Load	2	1	2	2	1	8 ns	24
Store	2	1	2	2	0	7 ns	12
Branch	2	1	2	0	0	5 ns	18
Jump	2	0	0	0	0	2 ns	2

The cycle time must accommodate the longest operation: *lw*.
Cycle time = 8 ns but the CPI = 1.

If we can accommodate variable number of cycles for each instruction and a cycle time of 1ns.
CPI = 6*44% + 8*24% + 7*12% + 5*18% + 2*2% = 6.3

Control Signals

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111