CSE340
Computer Architecture

Chapter 2

Instructions: Language of the Computer

Supplementary Slides

Prepared by Fairoz Nower Khan Miss

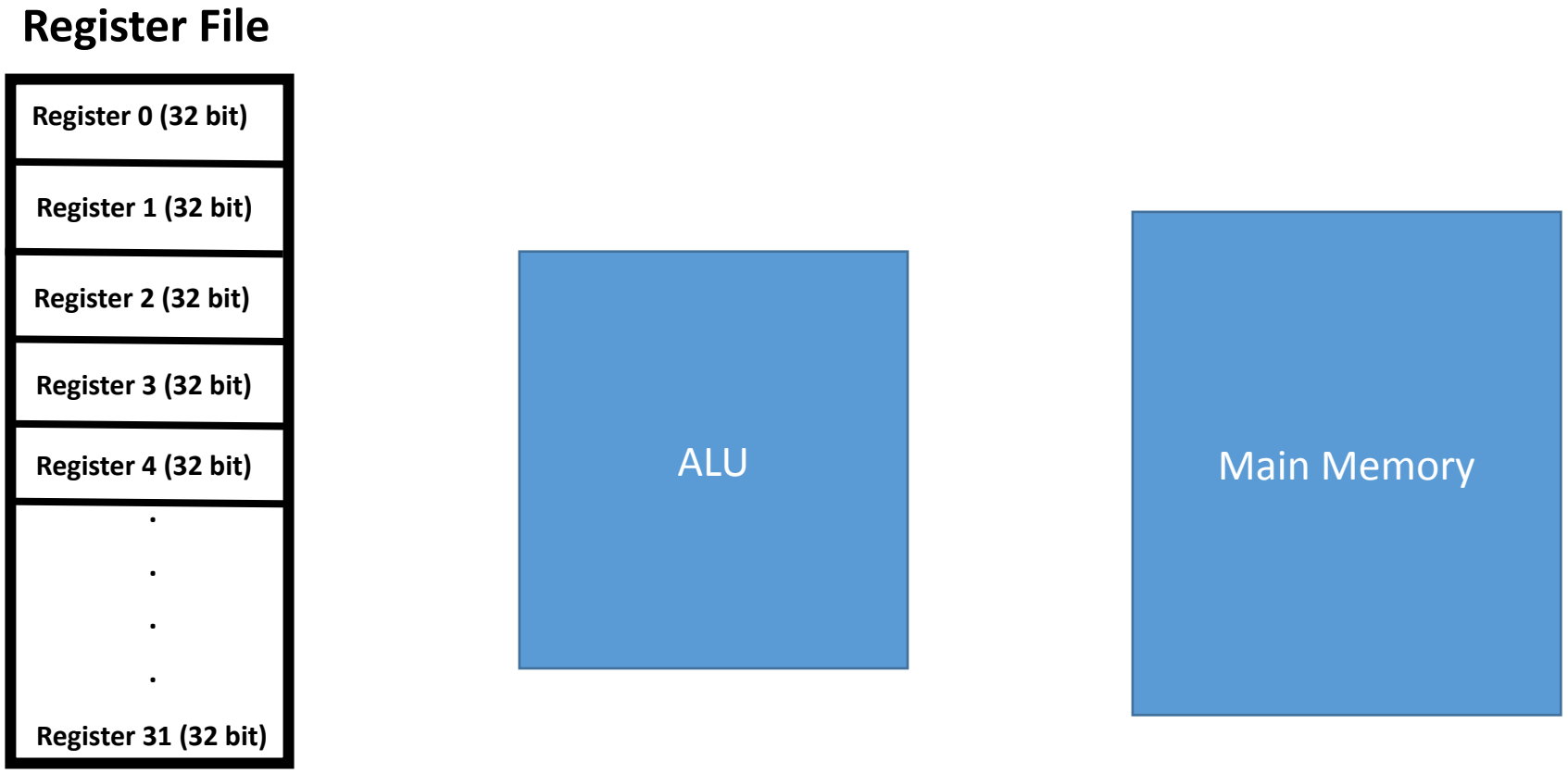
Lecture 8

- Instruction set and MIPS
- MIPS assembly code
- Register operands
- Main memory
- Memory operands

Slides prepared by Fairoz Nower Khan Miss

Computer Architecture

32 bit MIPS Architecture



So, I Hope You Are Here for This

CSE110/111

- How does an assembly program end up executing as digital logic?
- What happens in-between?
- How is a computer designed using logic gates and wires to satisfy specific goals?

CSE260

"Programming language" as a model of computation

Programmer's view of how a computer system works

Architect/microarchitect's view: How to design a computer that meets system design goals. Choices critically affect both the SW programmer and the HW designer

HW designer's view of how a computer system works

Digital logic as a model of computation

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code:

```
add t0, g, h  # temp t0 = g + h
add t1, i, j  # temp t1 = i + j
sub f, t0, t1 # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Register Operand Example

C code:

a = (c + d + e + f) – (r + x)

MIPS Code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
add $t2, $t0, $t1
add $t3, $s5, $s6
sub $s0, $t2, $t3
```

a is in -> $s0
c is in -> $s1
d is in -> $s2
e is in -> $s3
f is in -> $s4
r is in -> $s5
x is in -> $s6

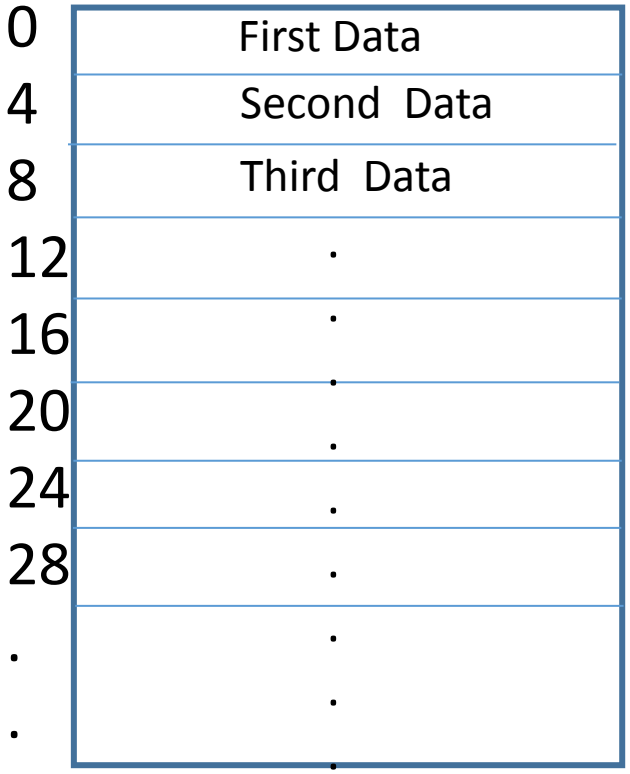
Memory Operands

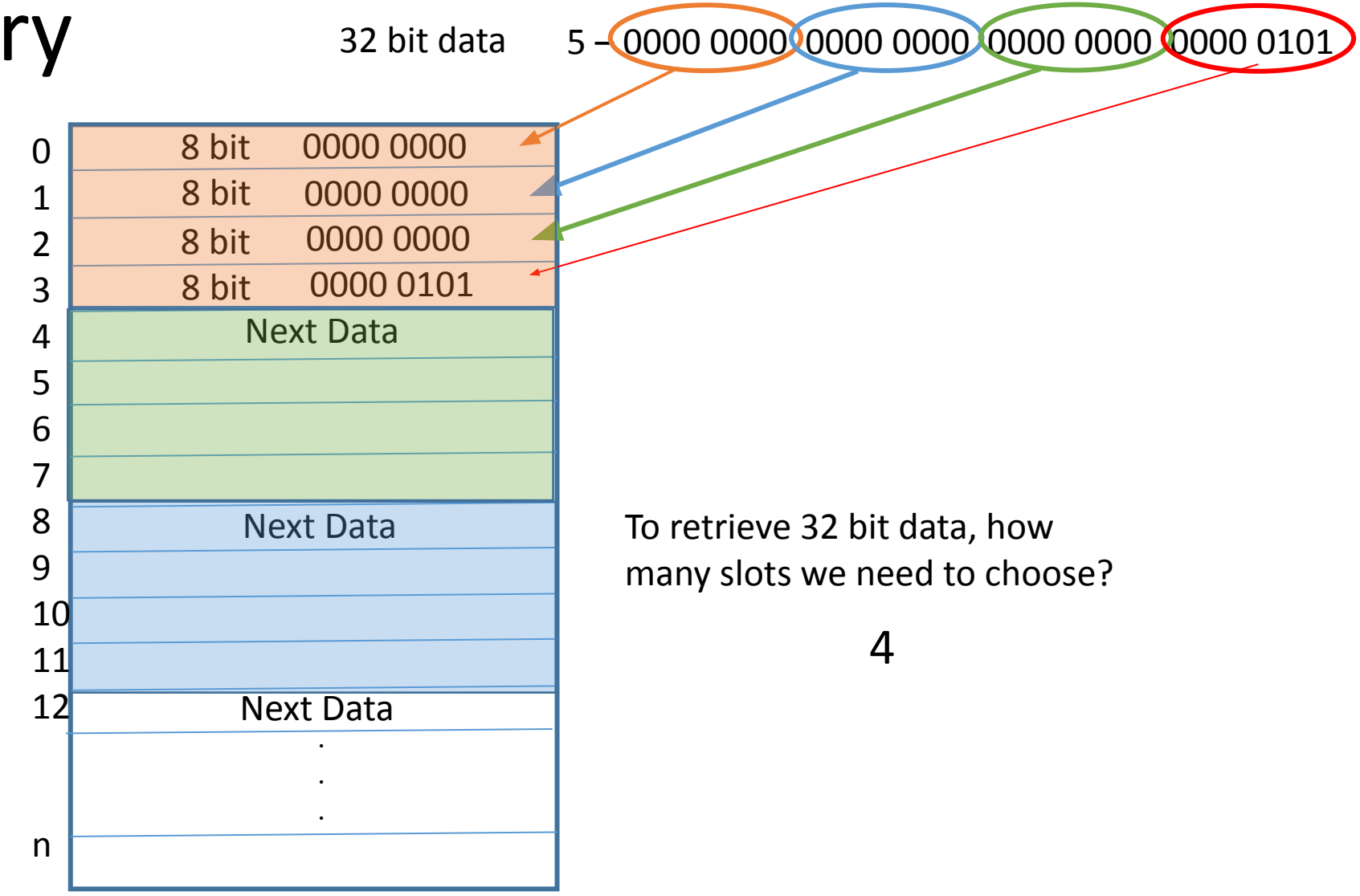
- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address 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

- The MIPS architecture can support up to 32 address lines.
- So an address could be *0xABCD1234* in hexadecimal
- Each memory address or cell is 8-bit or 1 byte
- This results in a 2^32 x 8 RAM, which would be 4 GB of memory.

Main Memory

Each slot contains 8 bit data or 1 byte data





To retrieve 32 bit data, how many slots we need to choose?

4

Main Memory Operand Example 1

C code:

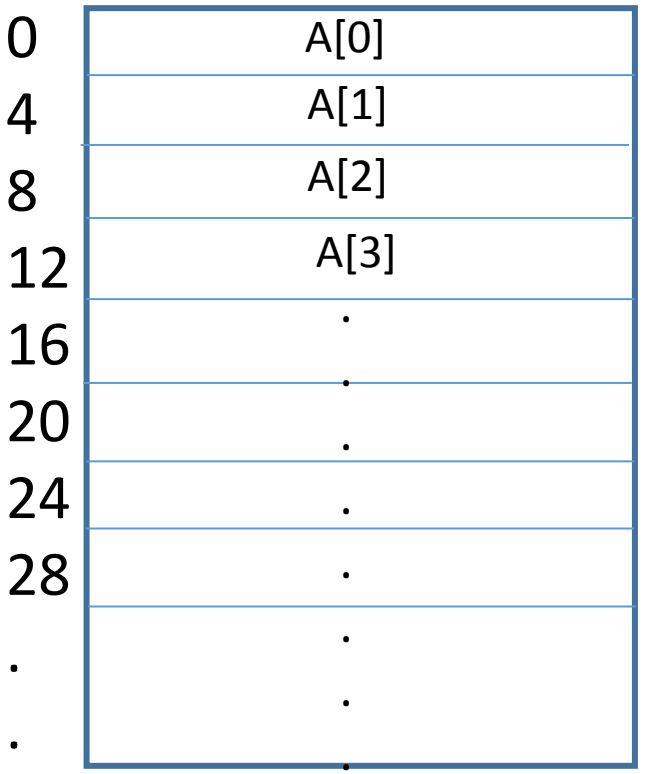
g = h + A[3]

lw $t0, 12($s3)

$s3 + 12

Load word (32 bit)

-> lw



g is in -> $s1

h is in -> $s2

Base Address of A is in -> $s3

Base Address of A means -> where Array A starts

The Address of A[0]

A[index] = 4*index(Base Address)

Main Memory Operand Example 1

C code:

g = h + A[3]

lw $t0, 12($s3)

add $s1, $s2, $t0

g is in -> $s1

h is in -> $s2

Base Address of A is in -> $s3

Main Memory Operand Example 1

- C code:

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

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

```
lw   $t0, 32 ($s3)
add $t1, $s2, $t0
sw   $t1, 48($s3)
```

48 ($s3)

Store word (32 bit)

-> sw

Examples from the book slides…

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
```

Lecture 9

- Registers vs Memory
- Immediate operands
- Zero registers
- Unsigned and signed integers
- MIPS Register file review
- MIPS Instructions
- MIPS R-format Instructions
- MIPS I-format Instructions

Slides prepared by Fairoz Nower Khan Miss

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed

- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

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
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Immediate Operands

- In case of subtraction

  Let's consider $s1 = 12

  And we want to subtract 5 from $s1 and store in $s1

addi $s1, $s1, -5

- Compiler cannot add or subtract two integers

addi $s1, 12, 5 Not Possible

addi $s1, 14, -7

subi does Not Exist

12+ (-5) = 7

$s1= 7

Everything will be stored as 32 bit binary number

addi Destination Register, Source 1 Register, Integer

The Constant Zero or Zero Register

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

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

- Zero Register or $zero always contains 32 bit 0
- The value of $zero always remains 0, cannot change it

Moving values between registers

Suppose we want to move a value from $s2 to $t3. How can we do it?

```
add   $t3,  $s2, $zero
```

Value in $s2 + $zero = Value in $s2

$t3 = Value in $s2

Or if we do not want to use $zero, we can also do this:

```
addi $t3, $s2, 0
```

Unsigned Binary Integers

- Given an n-bit number

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

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

For an 8 bit number then, the range would be:

0 to +((2^8)-1) so

0 to 255

2s-Complement Signed Integers

- Given an n-bit number

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

- 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

For an 8 bit signed number then, the range would be:

-2^(8-1) to +((2^(8-1))-1) so

-128 to +127

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

- Range: -2^{n-1} to $+2^{n-1} - 1$

2s-Complement Signed Integers

- Range: -2^{n-1} to $+2^{n-1} - 1$

Why $-(-2^{n-1})$ or 2^{n-1} cannot be represented?

Suppose for 8 bit representation, n = 8

The highest number in 8 bit signed number=

01111111 = 127

Sign bit

$(2^{8-1}) = 128 = 010000000$

Sign bit

9 bit

Same goes for 32 bit representation as well

Signed Negation

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

$$x + \bar{x} = 1111\ldots111_2 = -1$$

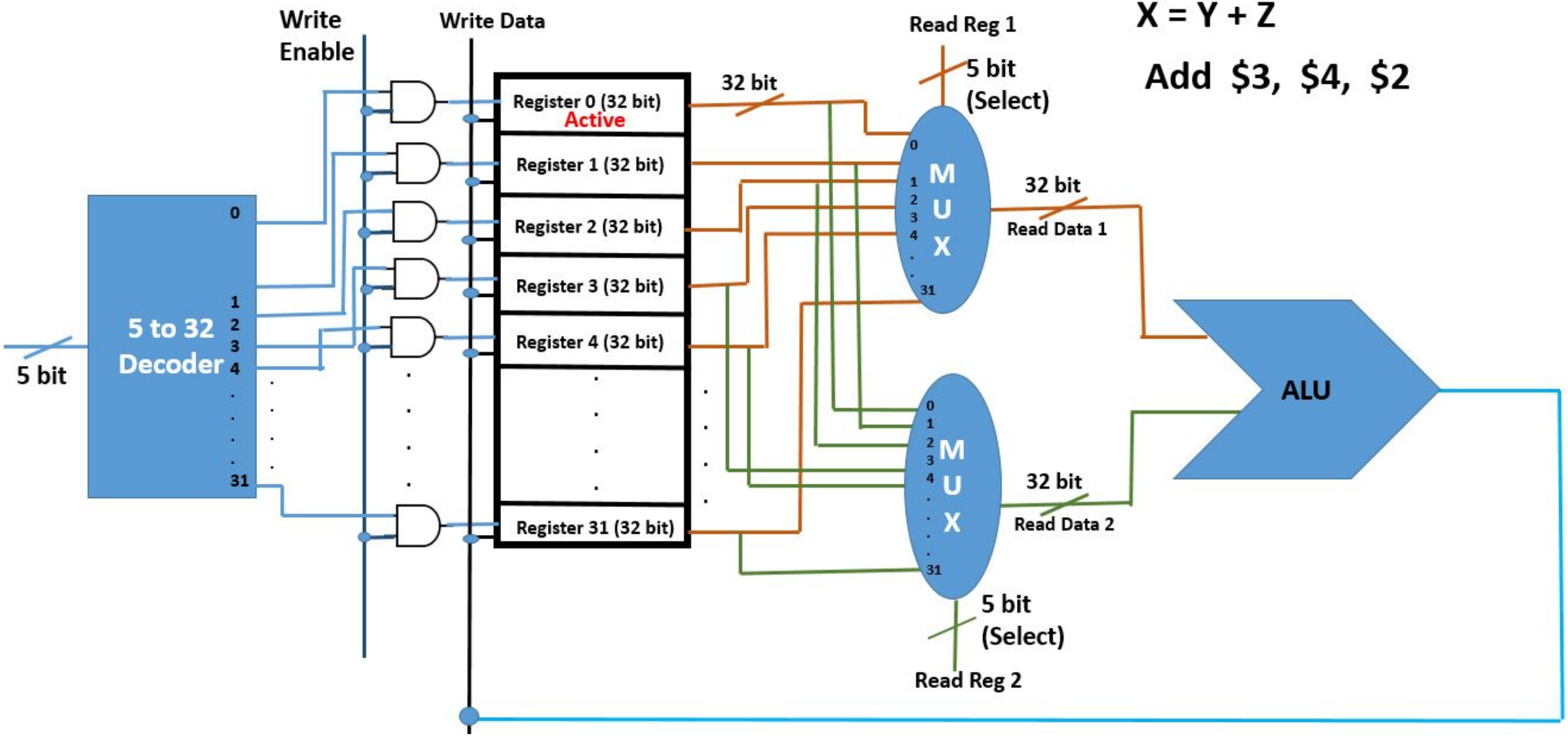
$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

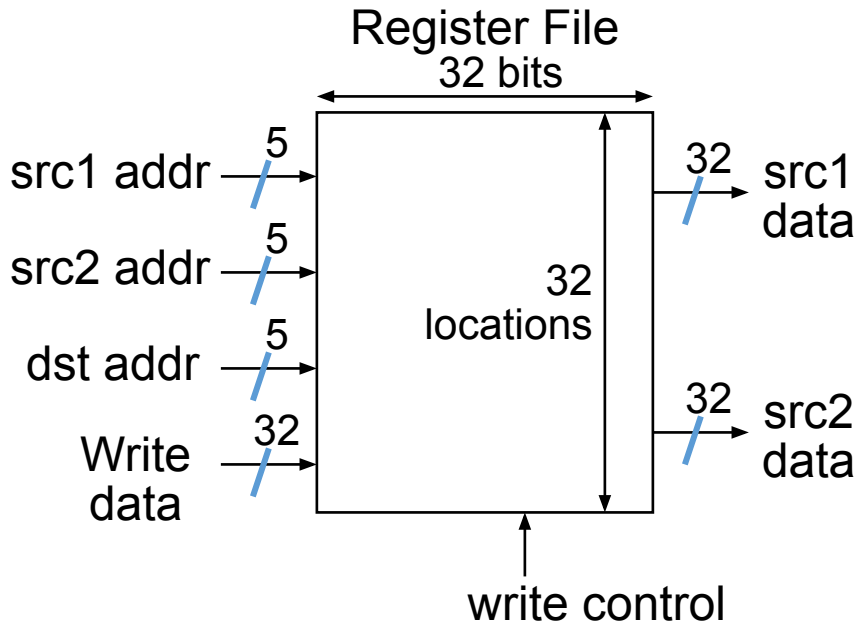
Sign Extension

- 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

Register File or Register Array

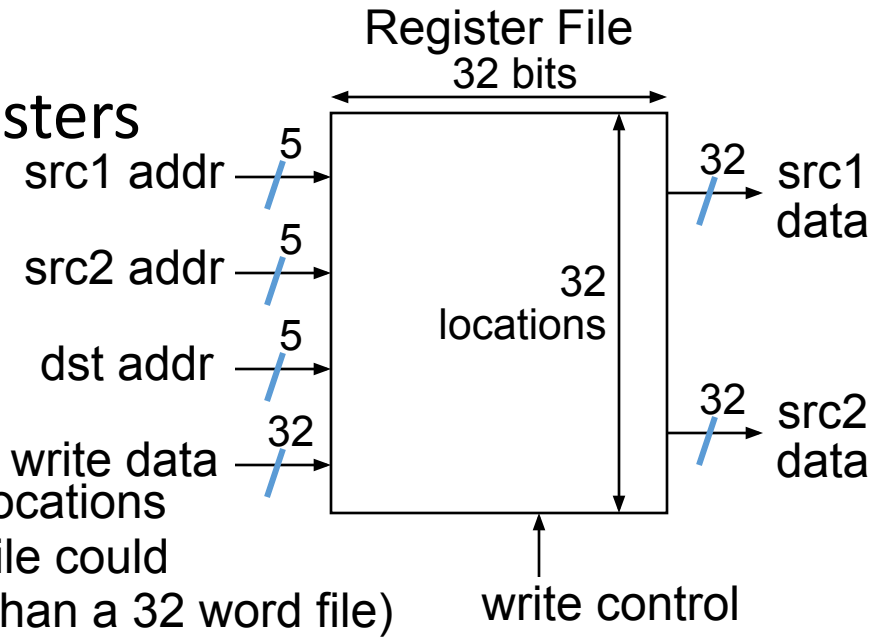


MIPS Register File



MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)



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

Representing Instructions

Register numbers

- $t0 – $t7 are reg's 8 – 15 -> $8 - $15
- $t8 – $t9 are reg's 24 – 25 -> $24 - $25
- $s0 – $s7 are reg's 16 – 23 -> $16 - $23

```
add $s0, $t0, $t1
add $16, $8, $9
```

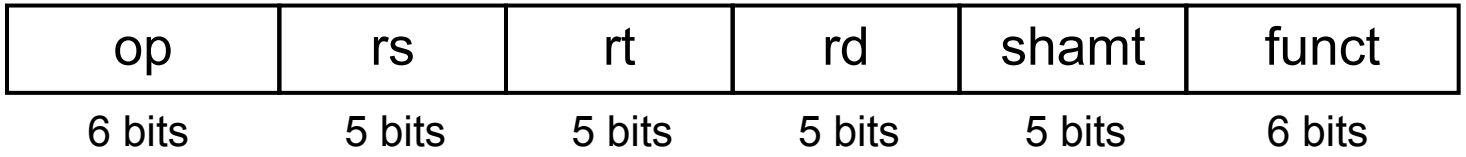
Same instructions

MIPS Instructions

There are Three Types of MIPS Instructions

1. R Type
   - add, sub, and, or, sll, srl (shift) [Arithmatic operations]
2. I Type
   - lw, sw, addi, beq, bne
3. J Type
   - j (jump)

MIPS R-format Instructions



- 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

Register numbers

- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

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

add $t0, $s1, $s2

add $8, $17, $18

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000001000000_2 = 02324020_{16}$$

0 2 3 2 4 0 2 0

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

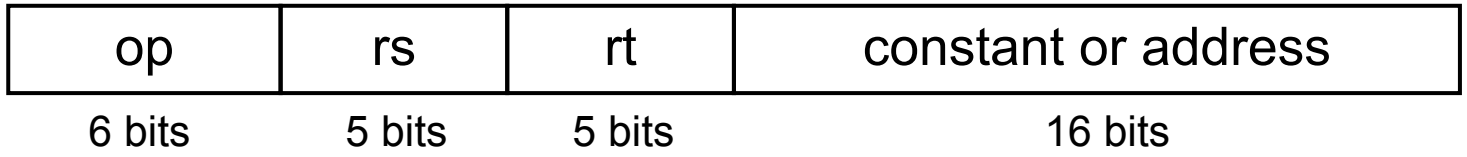
MIPS I-format Instructions

For the instructions which have constants or whole numbers

- addi $s0, $s1, 2
- lw $t0, 12 ($s0)
- sw $t1, 24 ($s1)

MIPS I-format Instructions

- addi $s0, $s1, 2
- lw $t0, 12 ($s0)
- sw $t1, 24 ($s1)

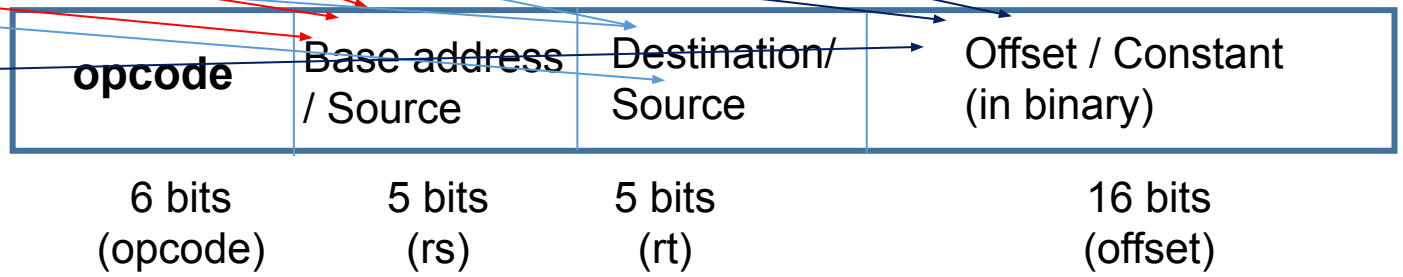


- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit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

MIPS I-format Instructions

For the instructions which have something to do with constants or integers

- addi $s0, $s1, 2
- lw $t0, 12($s0)
- sw $t1, 24($s1)



MIPS I-format Instructions

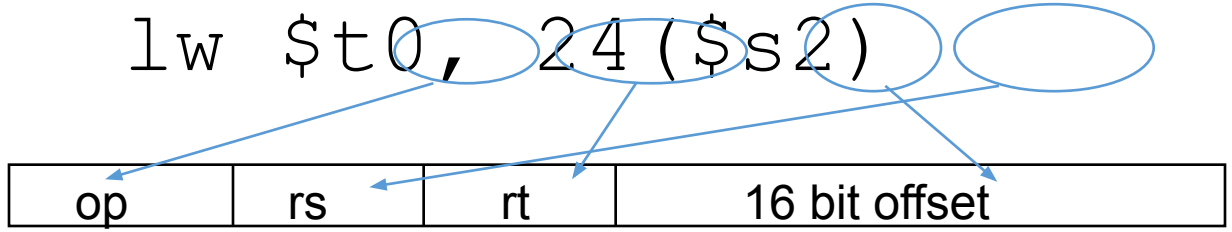
- MIPS has two basic data transfer instructions for accessing memory

```
lw $t0, 4($s3)   #load word from memory
sw $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

MIPS I-format Instructions

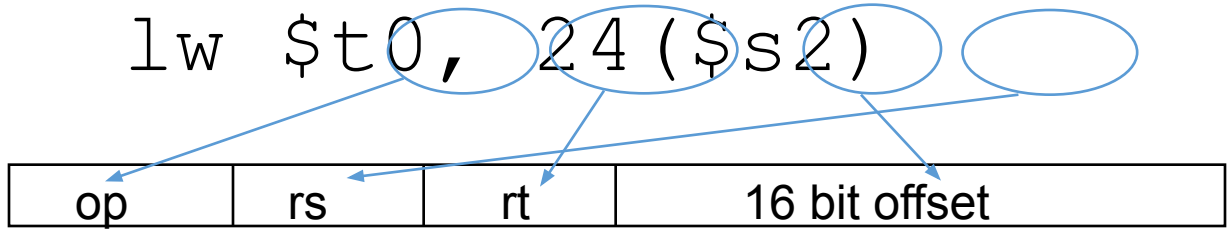
- Load/Store Instruction Format (I format):



For lw and sw Base Address will always be in rs

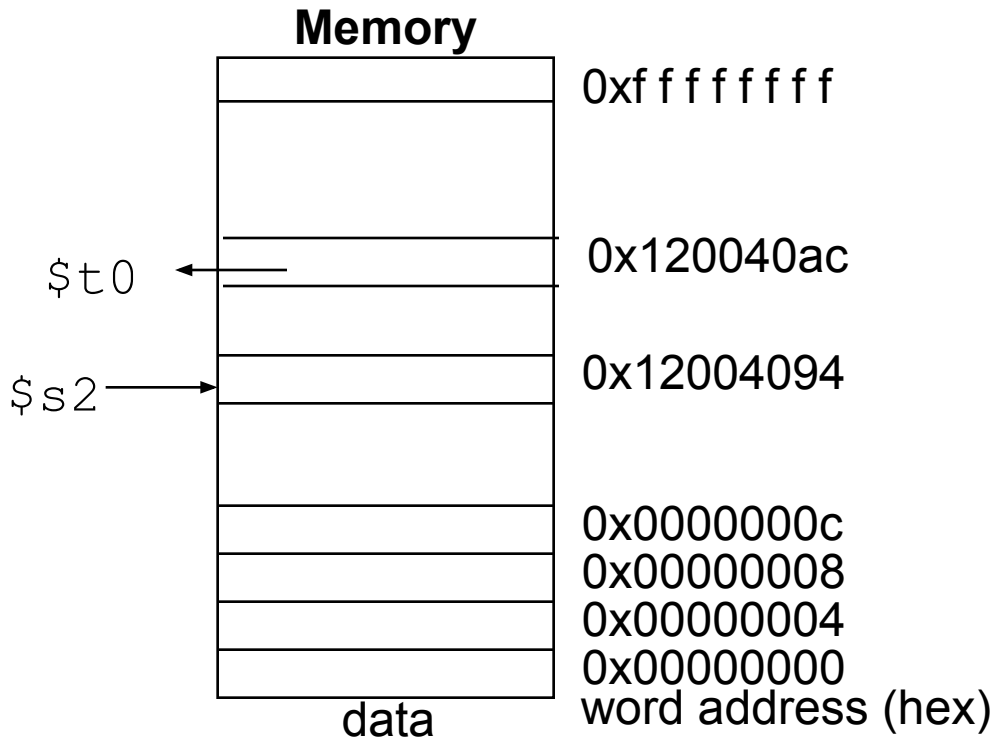
MIPS I-format Instructions

- Load/Store Instruction Format (I format):



24_{10} + $s2 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



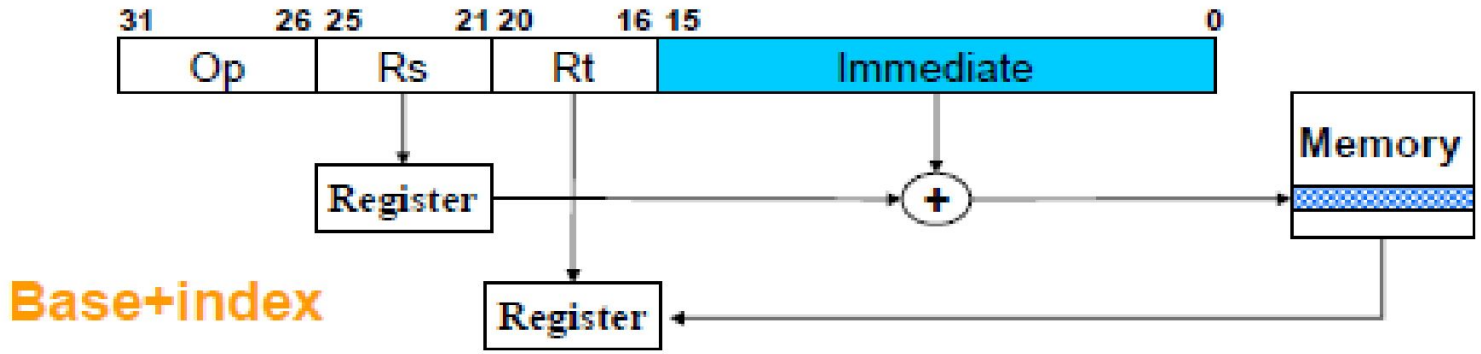
MIPS I-format Instructions

- Load/Store Instruction Format (I format):

Load Word Example

lw $1, 100($2) # $1 = Mem[$2+100]

op	rs	rt	immediate
010011	00010	00001	0000 0000 0110 0100



Lecture 10

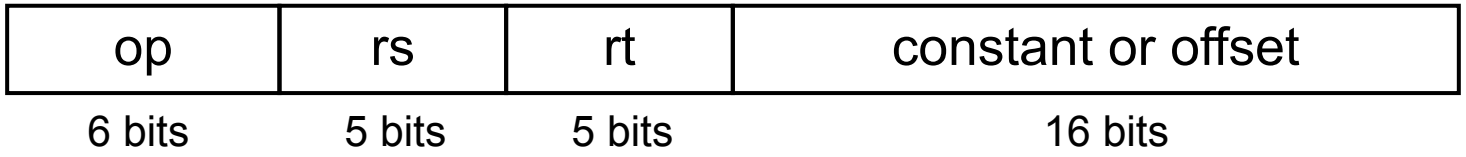
- Conditional operations
- beq, bne
- Program counter
- Calculating branch destination addresses

Slides prepared by Fairoz Nower Khan Miss

MIPS I-format Instructions

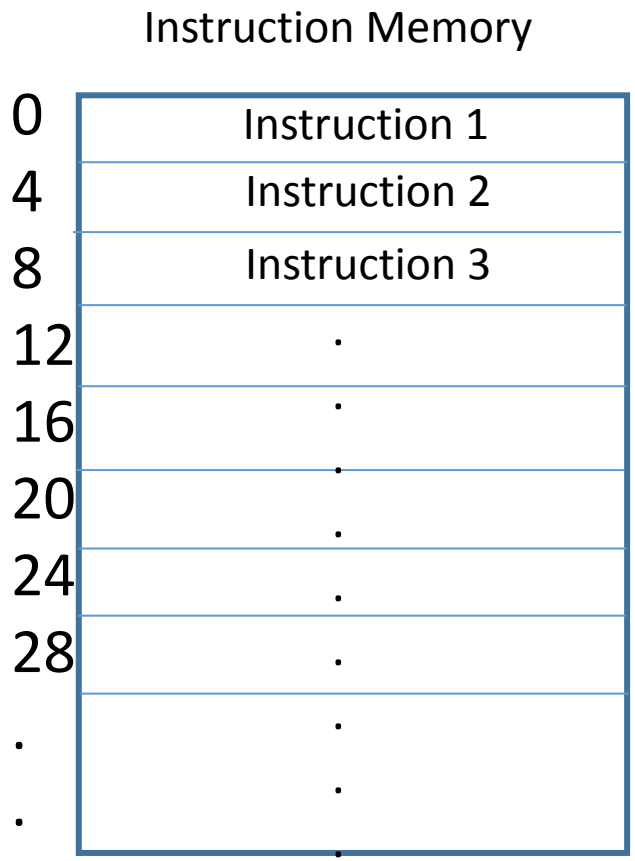
For the instructions which have something to do with constants or integers

- addi $7, $8, 2
- lw $9, 12 ($10)
- sw $17, 24 ($18)
- beq $a, $b, L1
- bne $8, $9, L1



Program Counter (PC)

Holds the address of current instruction



Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- **beq rs, rt, L1** beq □ branch is Equal
  - if (rs == rt) branch to instruction labeled L1;
- **bne rs, rt, L1** bne □ branch is Not Equal
  - if (rs != rt) branch to instruction labeled L1;
- **j L1**
  - unconditional jump to instruction labeled L1

beq

```
                a     b     2
0   beq $s1, $s2, L1
          rs    rt
4   add $s1, $s2, $t1
          a     b     1
8   j Exit

12  L1:
    add $s1, $s2, $t2
          a     b     2
    Exit:
```

if(rs = rt)

Go to L1

Let's consider the code:

```
if (a != b)
    a = b + 1;
else
    a = b + 2;
```

$s1 a

$s2 b

$t1 1

$t2 2

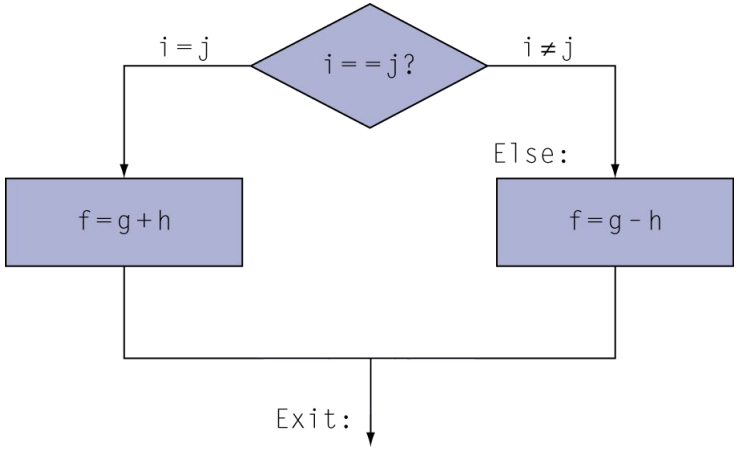
L1 gets converted into the number of instructions to jump over to where L1 is located

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

f □ $s0

g □ $s1

h □ $s2

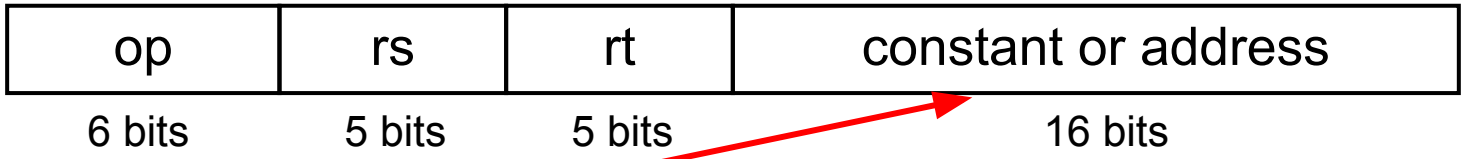
i □ $s3

j □ $s4

MIPS I-format Instructions

For the instructions which have constants or whole numbers

- addi $7, $8, 2
- lw $9, 12 ($10)
- sw $17, 24 ($18)
- beq $a, $b, L1
- bne $8, $9, L1



L1 gets converted into the number of instructions to jump over to where L1 is located

Program Counter

- Program counter (instruction pointer) identifies the current instr.
- Program counter is advanced sequentially except for control transfer instructions

PC (Program Counter) = 0

beq

```
                              2
0   beq $s1, $s2, L1
          rs    rt
4   add $s1, $s2, $t1
8   j Exit
12  L1:
    add $s1, $s2, $t2
    Exit:
```

Shifting

5	000101	001010	010100
	5	10	20

Multiplication by 2^(the number of left shift)

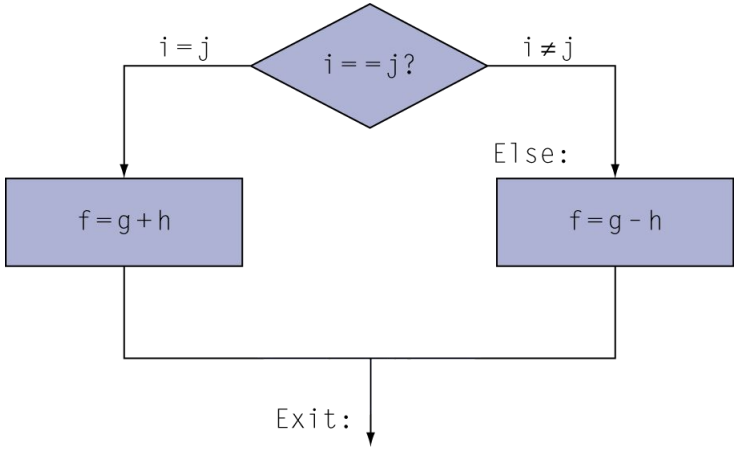
Division by 2^(the number of Right shift)

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

f □ $s0

g □ $s1

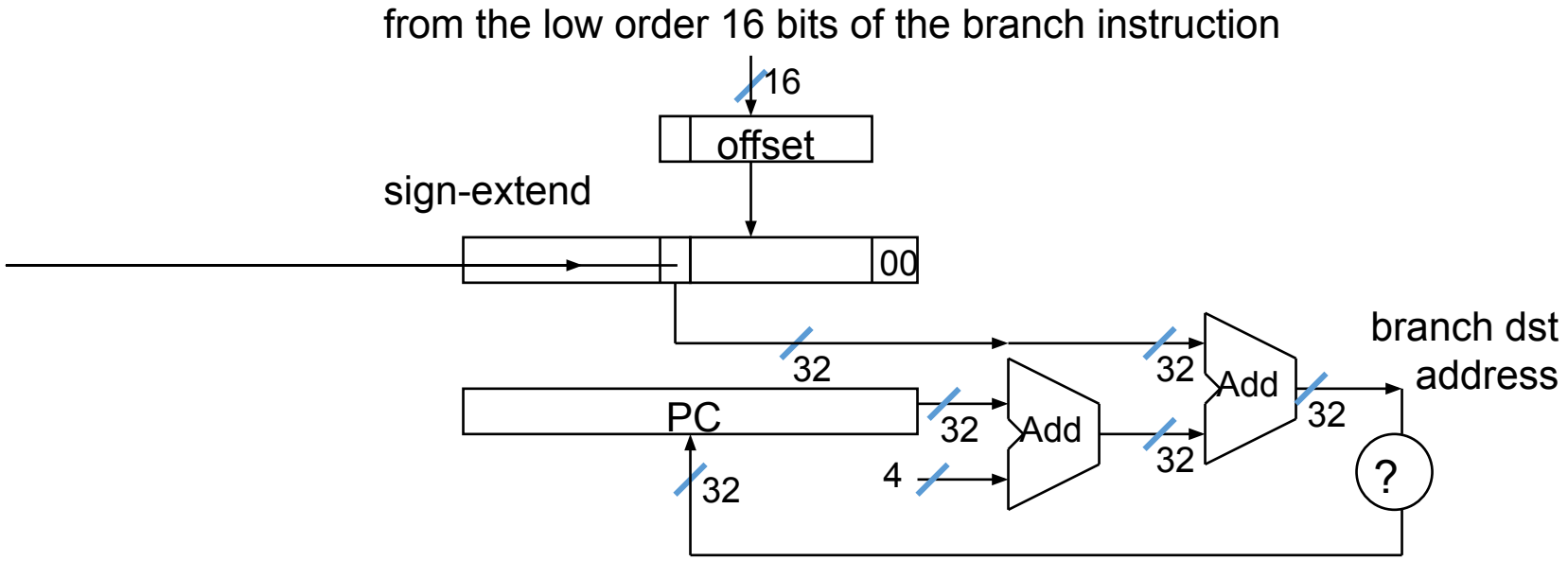
h □ $s2

i □ $s3

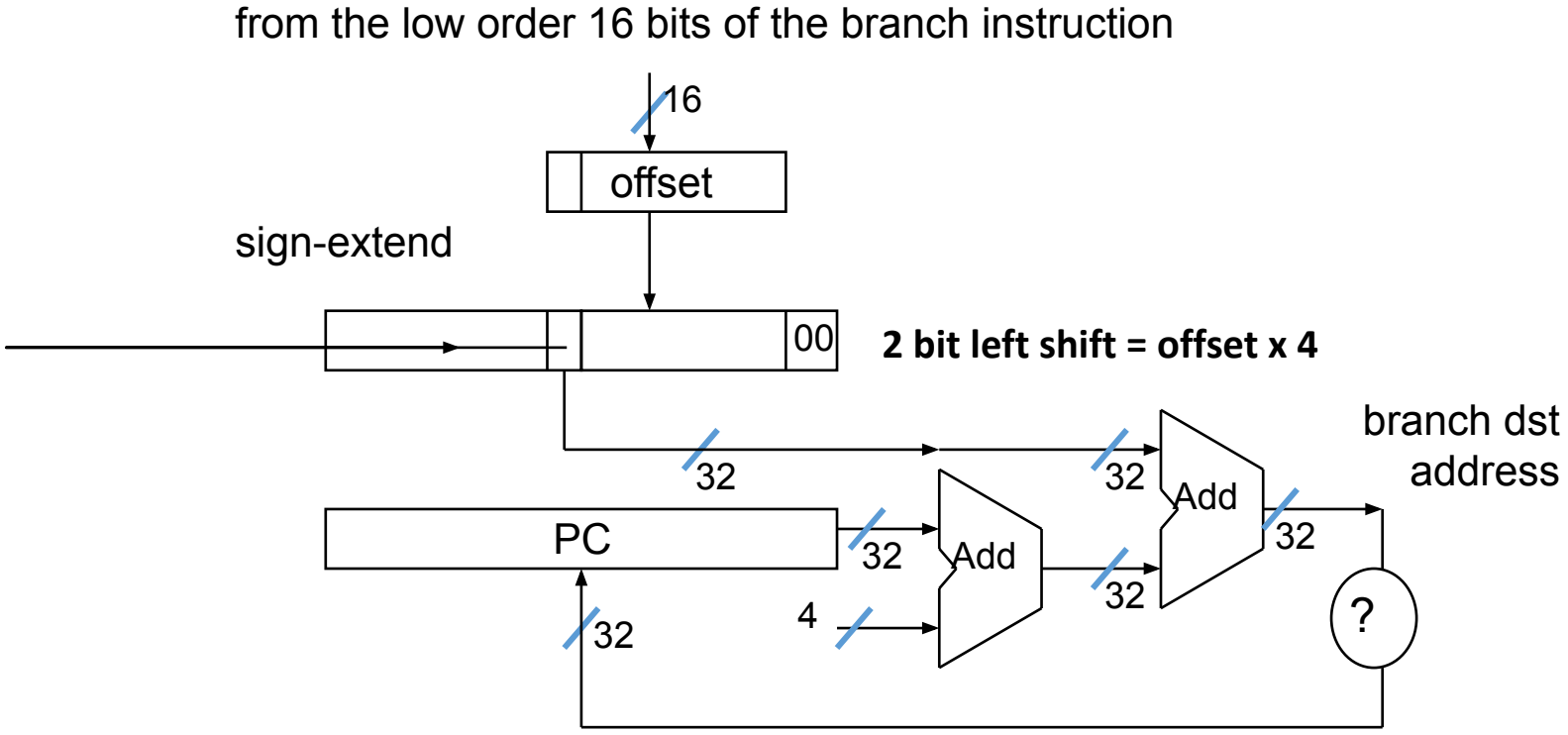
j □ $s4

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



Specifying Branch Destinations



beq

2

0 beq $s1, $s2, L1
rs rt

4 add $s1, $s2, $t1

8 j Exit

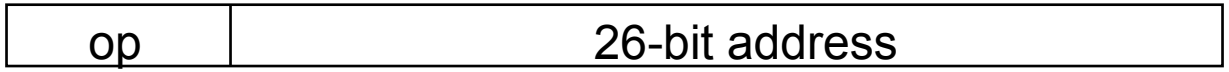
12 L1:
add $s1, $s2, $t2
Exit:

J – Type Instruction

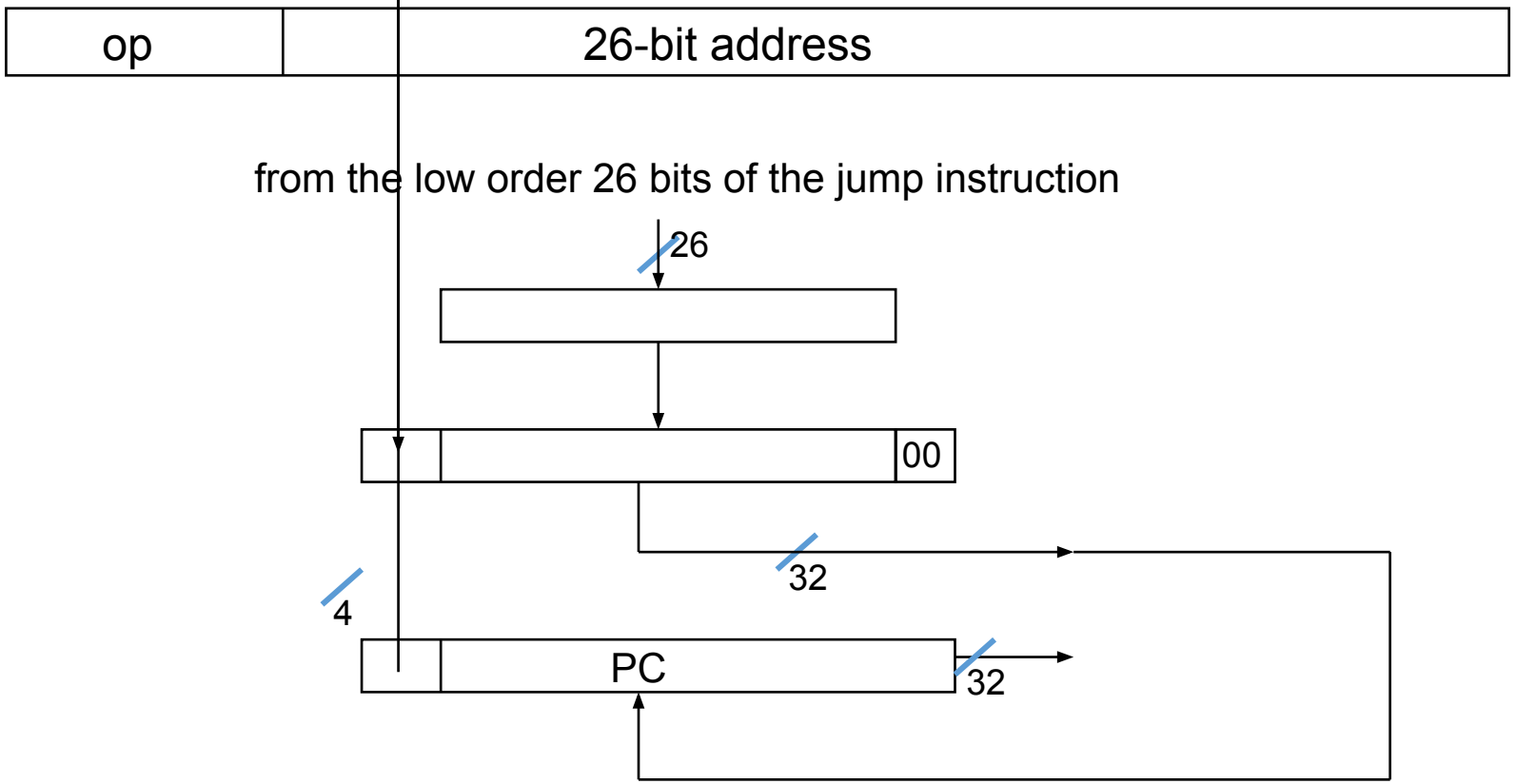
- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label    #go to label
```

- Instruction Format (J Format):



Specifying Branch Destination for Jump



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

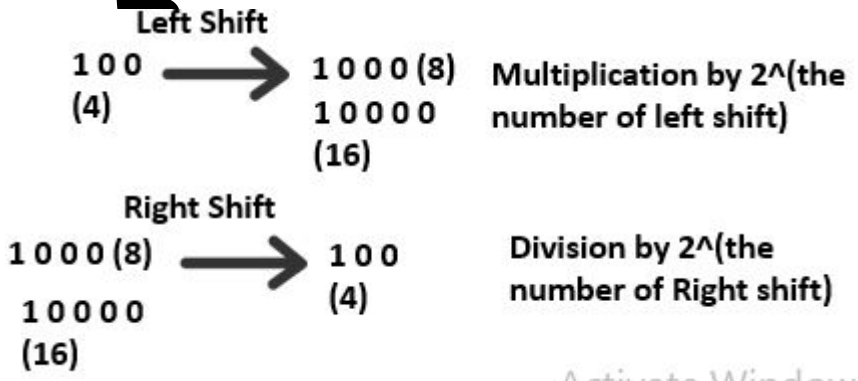
Shift Operations

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

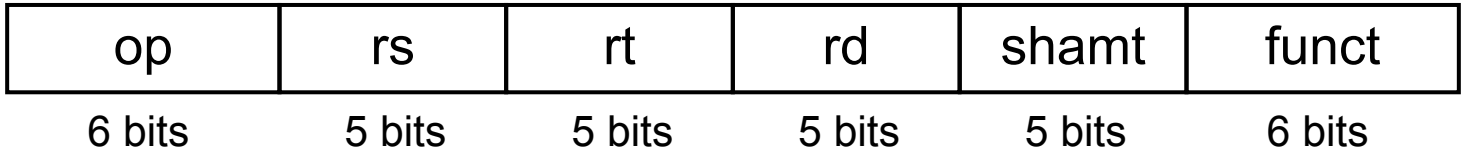
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - **sll** by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - **srl** by *i* bits divides by 2^i (unsigned only)

sll $t1, $s3, 2

slr $t2, $s3, 4

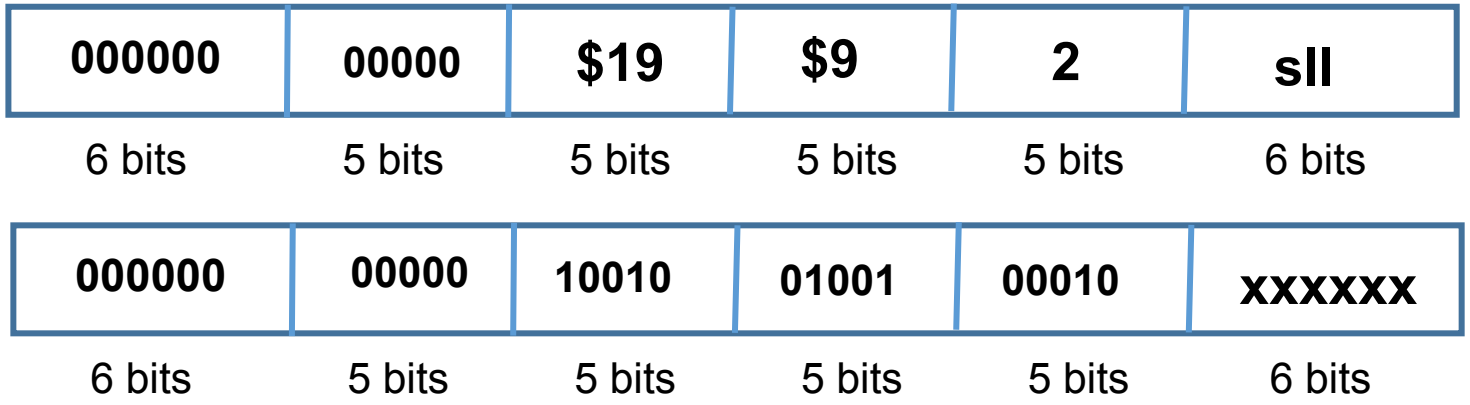


Shift Operations – R Type



sll $t1, $s3, 2

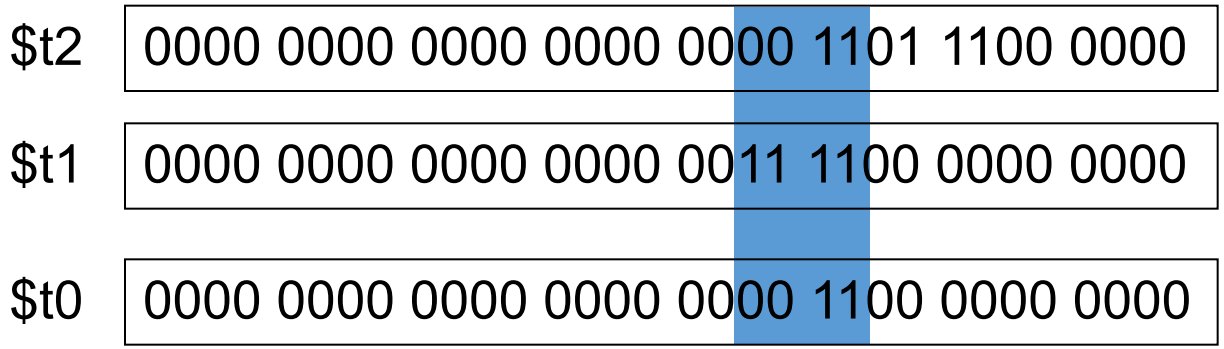
sll $9, $19, 2



AND Operations

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

```
and $t0, $t1, $t2
    rd   rs   rt
```

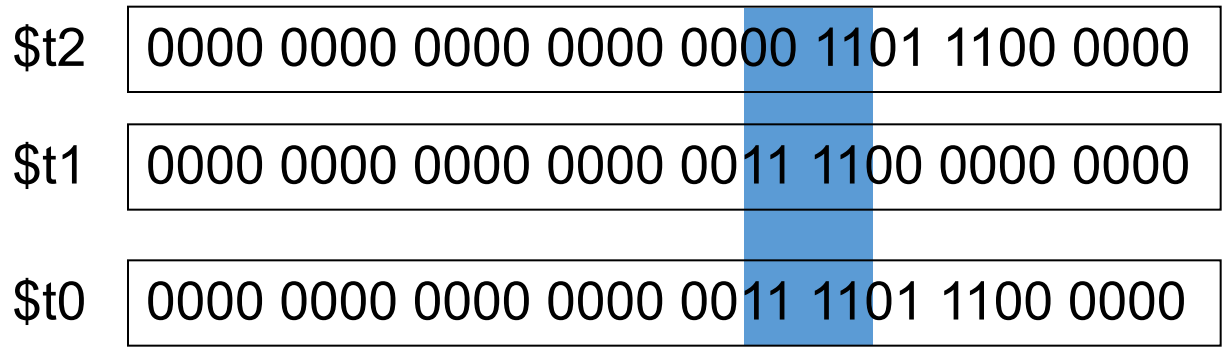


OR Operations

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

or $t0, $t1, $t2

rd rs rt



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, $t2

nor $to, $t1, $zero

rd rs rt

Register 0: always read as zero

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

MIPS Code Practice

- X = 2Y + 65Z - 10

Where X, Y and Z are in $s0, $s1 and $s2 respectively

```
add   $t0, $s1, $s1
sll   $t1, $s2, 6
add   $t1, $t1, $s2
add   $t1, $t0, $t1
addi  $s0, $t1, - 10
```

$t0 □ 2Y

$t1 □ 4Z

$t1 □ 5Z

$t1 □ 2Y+ 5Z

MIPS Code Practice

- B[10] = A[5] + 2

Where base address for B and A are $s1 and $s2 respectively

```
lw   $t0, 20($s2)
addi   $t1,  $t0,  2
sw     $t1,  40($s1)
```

$t0 □ value of A[5]

B □ A[5] + 2

MIPS Code Practice

Base address of A is in $s0

```
if (A[3] != A[6]){
   if(A[3] == 0){
      A[3] = A[3] + 2; }
    else {
     A[6] = A[6] / 16; }
}
else {
    A[6] = A[6] * 8;
}
```

```
lw $t0, 12($s0)        $t0 ← A[3]
lw $t1, 24($s0)        $t1 ← A[6]

beq $t0, $t1, Label1
bne $t0,   $zero, Label2
addi $t7, $t0, 2       $t7 ← $t0 + 2
sw $t7, 12($s0)
j Exit
Label2:
srl $t6, $t1, 4        $t6 ← $t1 / 2^4
sw $t6, 24($s0)
j Exit
Label1:
sll   $t5, $t1, 3      $t5 ← $t1 * 2^3
sw $t5,   24($s0)
Exit
```

More Conditional Operations
slt, slti □ R Type

$s3 = 5
$s4 = 10

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- **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 $t1, $s3, $s4 # $t1 = 1

slti $t2, $s3, 2 # $t2 = 0

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

Conditional Operations

a and b is in $s0 and $s1 respectively

```
if (a>b){
   a= a+1;}
else{
   a=a+2;
}
```

```
                    a      b
slt $t1, $s0, $s1        ($s0>$s1), $t1 = 0
bne $t1, $zero, L1
addi $s0, $s0, 1
j Exit
L1:
addi $s0, $s0, 2
Exit
```

Branch Instruction Design

- Why not **blt**, **bge**, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- **beq** and **bne** are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: **slt**, **slti**
- Unsigned comparison: **sltu**, **sltui**
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - **slt $t0, $s0, $s1 # signed**
    - –1 < +1 ⇒ $t0 = 1
  - **sltu $t0, $s0, $s1 # unsigned**
    - +4,294,967,295 > +1 ⇒ $t0 = 0

MIPS Code Practice

- x = A[i] + 2

Where x, base address of A and i are in $s1, $s2 and $s3 respectively

```
sll   $t0, $s3, 2
add   $t1, $s2, $t0
lw    $t2, 0($t1)
addi  $s1, $t2, 2
```

$t0 □ 4*i

$t1 □ 4*I + $s3

$t2 □ the value in A[i]

Compiling Loop Statements

Java code:

```
while(save[i] == k){
      a= a+2;
      i+=1;
}
```

- Compiled MIPS code:

```
Loop:   sll   $t0, $s3, 2
        add   $t1, $s6, $t0
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        addi  $s4, $s4, 2
        addi  $s3, $s3, 1
        j     Loop
Exit:
```

i= $s3

k= $s5

Base address of save= $s6

a= $s4

Compiling Loop Statements

Java code:

```
for (int i=0; save[i]>k;i++){
      a= a+2; }
```

- Compiled MIPS code:

```
            add  $s3, $zero, $zero
Loop:    sll    $t0, $s3, 2
            add  $t1, $s6, $t0
            lw    $t0, 0($t1)
            slt    $t5, $t0, $s5
            bne   $t5, $zero, Exit
            addi  $s4, $s4, 2
            addi  $s3, $s3, 1
             j       Loop
Exit:
```

i -> $s3

k -> $s5

Base address of save $s6

a -> $s4

Compiling Loop Statements

Java code:

```
for (int i=0; i<15;i++){
      if (A[i+1] !=0){
            sum = sum +1;
      else
            sum = sum -1;
 }
```

i is in $s3

Base address of A is in $s0

Sum is $s1

- Compiled MIPS code:

```
            add  $s3, $zero, $zero
Loop:
            slti   $t0, $s3,    15
            beq  $t0, $zero, Exit
            addi  $t0, $s3,    1
            sll     $t0, $t0, 2
            add   $t1, $s6, $t0
            lw     $t0, 0($t1)
            beq  $t0, $zero, Else
            addi   $s1, $s1,    1
            addi   $s3, $s3, 1
            j       Loop
Else:
            addi   $s1, $s1, -1
            addi   $s3, $s3, 1
            j       Loop
Exit:
```

MIPS Code Practice

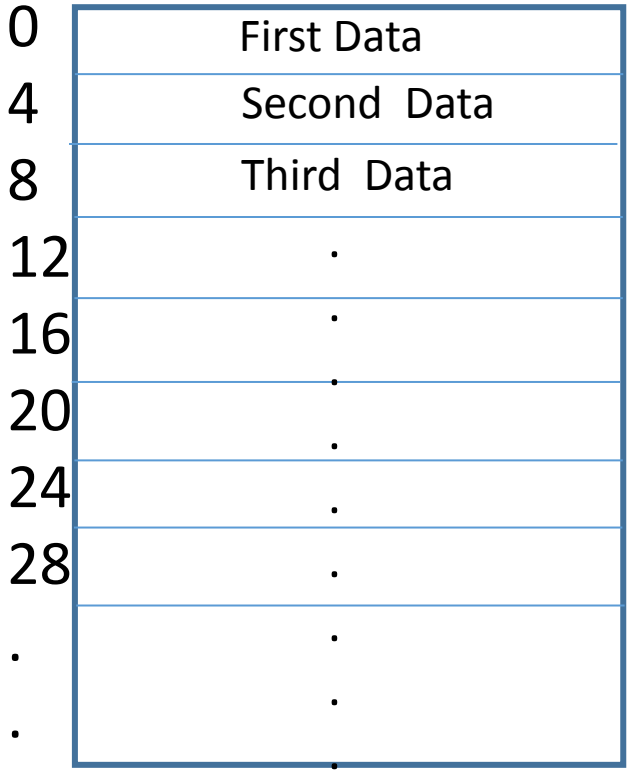
- A[B[i]] = x

Where base address for A and B are $s1 and $s2 respectively and x and i are in $s3 and $s4

```
sll  $t0, $s4, 2
add  $t1, $s2, $t0
lw   $t2, 0($t1)
sll  $t0, $t2, 2
add  $t1, $s1, $t2
sw   $s3, 0($t1)
```

2^n bit Architecture

- 32 bit Architecture – Each register is of 32 bit. Data memory and instruction memory with 8 bit slots. So memory increment by 4 (32/8)



- 64 bit architecture
- 128 bit Architecture
- 256 bit Architecture

Address for 32 bit Architecture

Memory Address for a data in Array = Base Address + Offset x 4

Branch Address = PC+ 4 + Offset x 4

Jump Address = PC (MSB 4 bits)+ Offset x 4

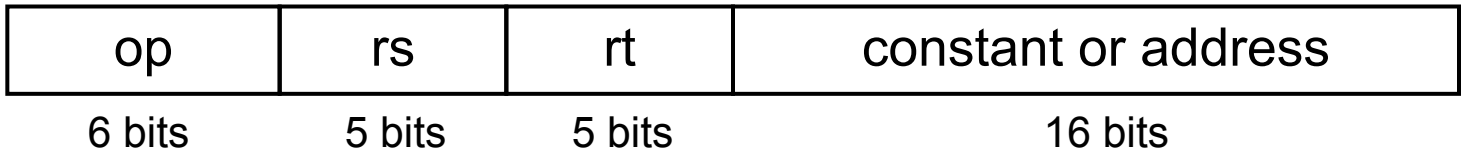
Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Register Usage

- $a0 – $a3: arguments (reg's 4 – 7) **#Function Parameters Registers**
- $v0, $v1: result values (reg's 2 and 3) **#Function Result Registers**
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31) **#Return address of main function**

32-bit Constants



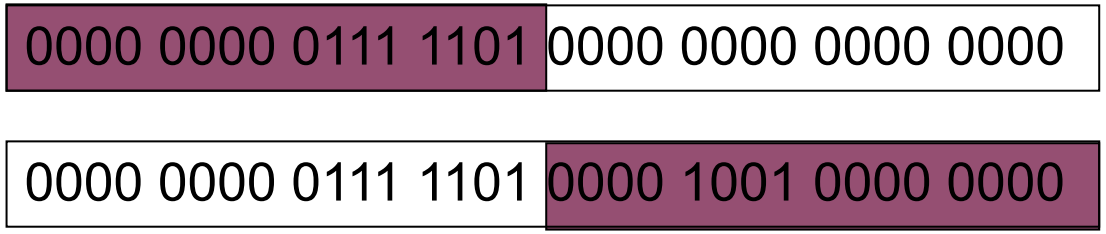
- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

lui rt, constant

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

- addi $s0, $s1, 2

lui $s0, 61
ori $s0, $s0, 2304

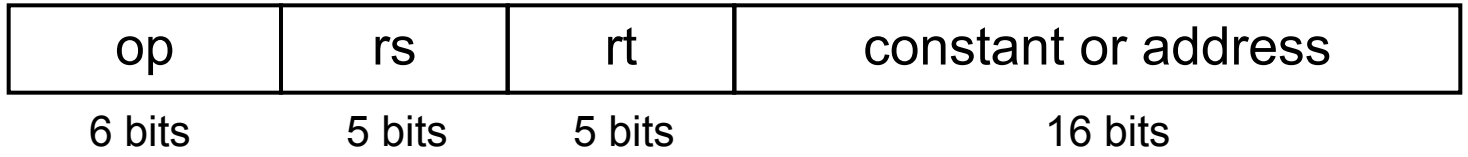


Branch Addressing

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

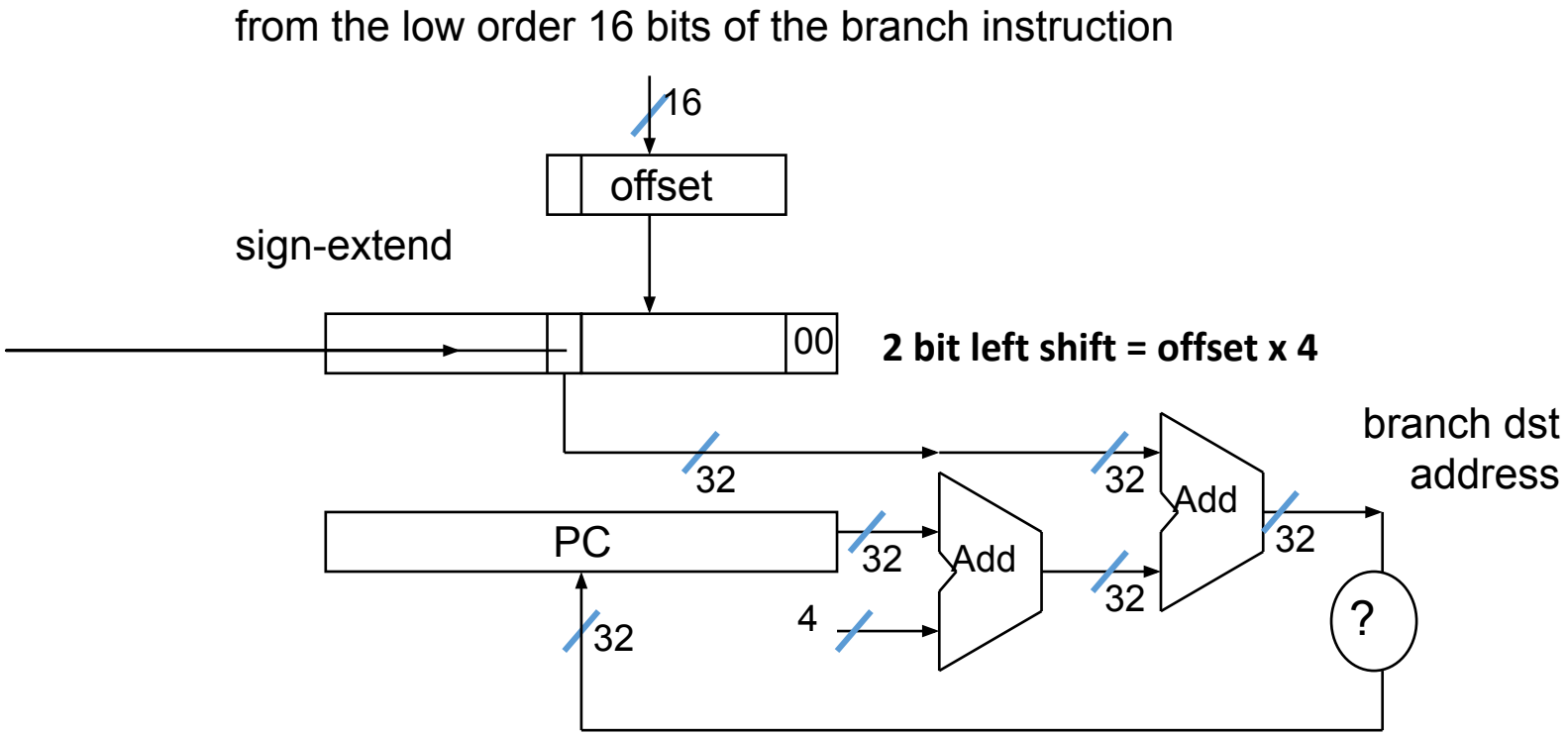
bne/ beq instructions

- beq $a, $b, L1
- bne $8, $9, L1



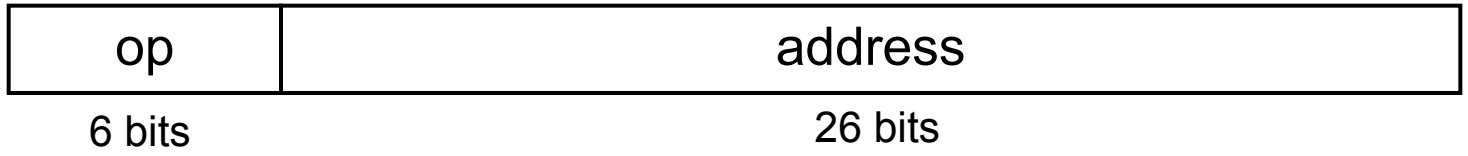
- PC-relative addressing
  - Target address = (PC+4) + offset × 4
  - PC already incremented by 4 by this time

Specifying Branch Destinations



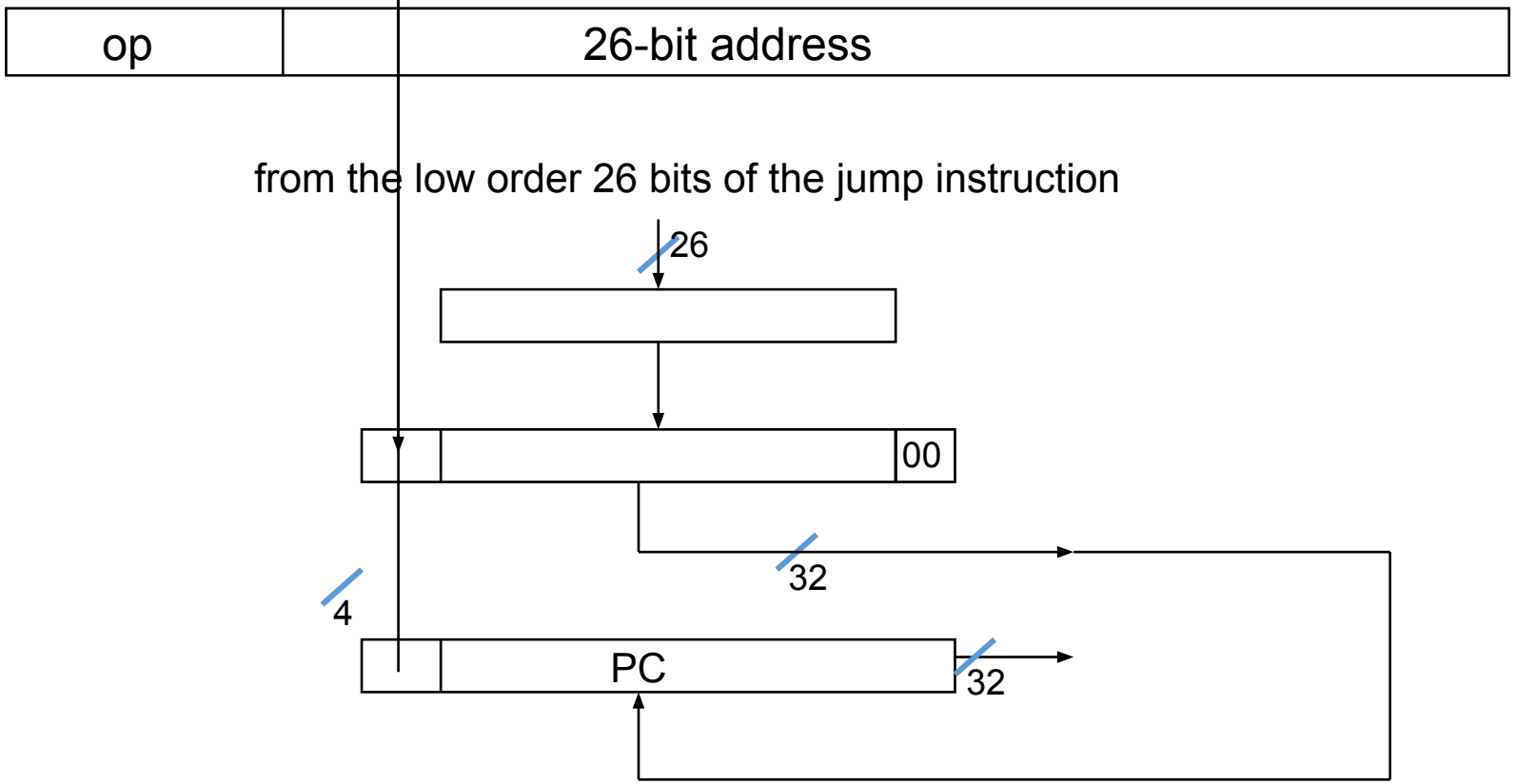
Jump Addressing

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



- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

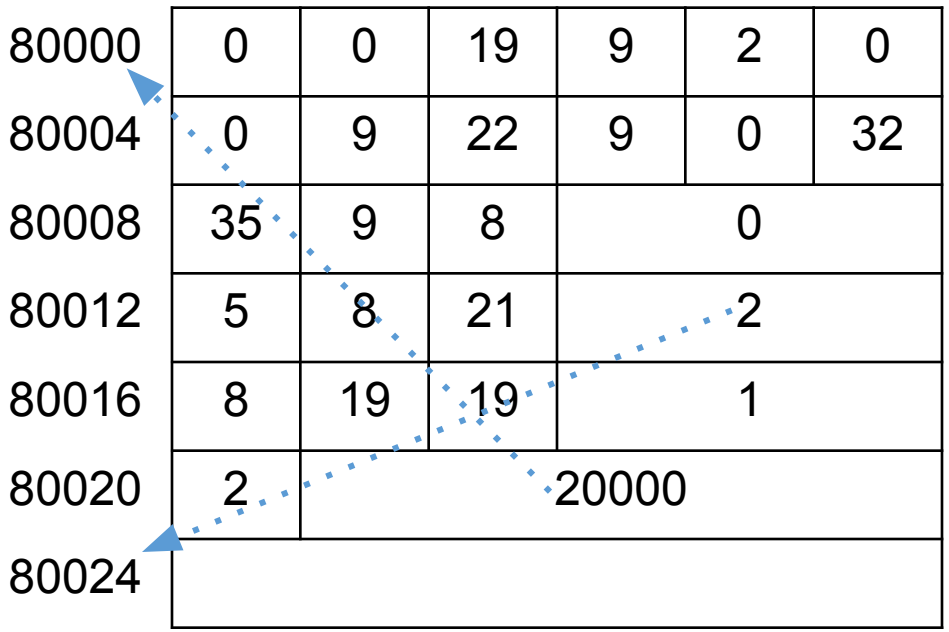
Specifying Branch Destination for Jump



Target Addressing Example

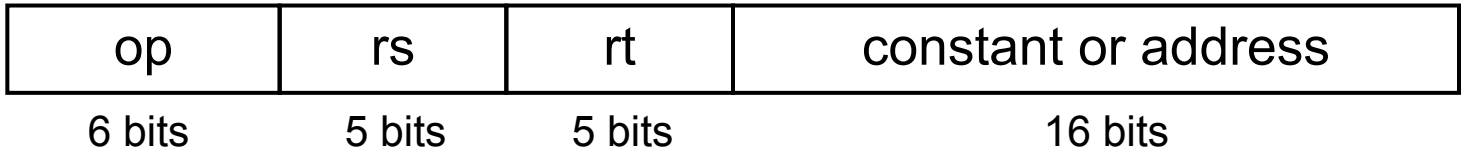
- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j  Loop
Exit: …
```



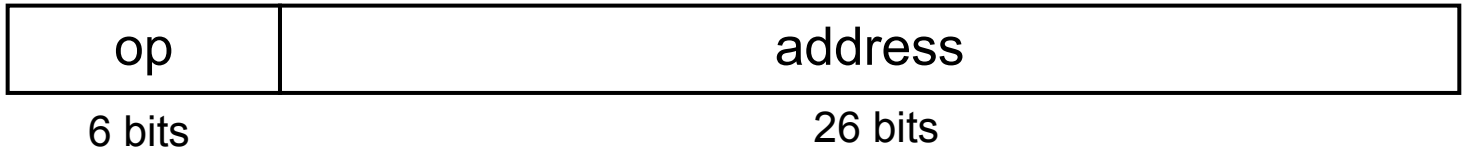
Branching Far Away

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



- Example

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

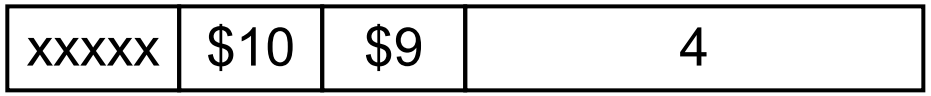


If L1 is more than 16 bits,
The assembler rewrites the code to allocate 26 bits for L1 by using j type instruction

Addressing Mode Summary

1. Immediate addressing

addi $9, $10, 4

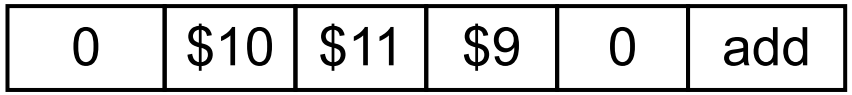


6 bits | 5 bits | 5 bits | 16 bits

2. Register addressing

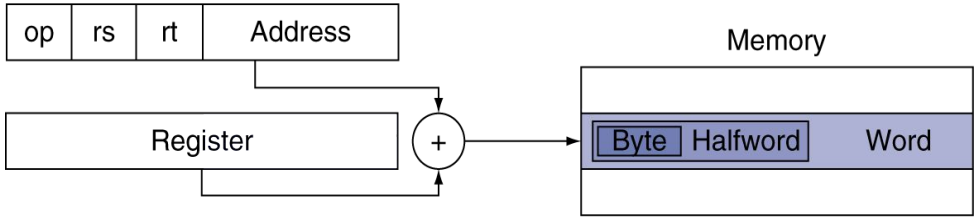
add $9, $10, $11

R Type



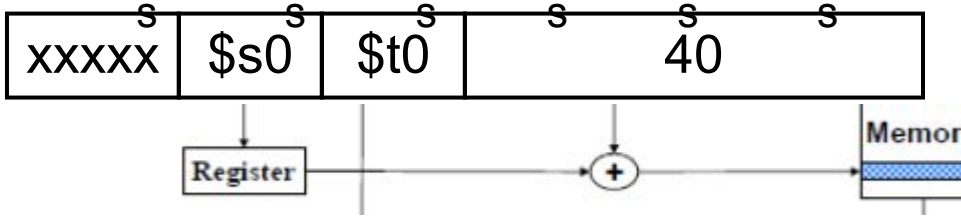
6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits

3. Base addressing



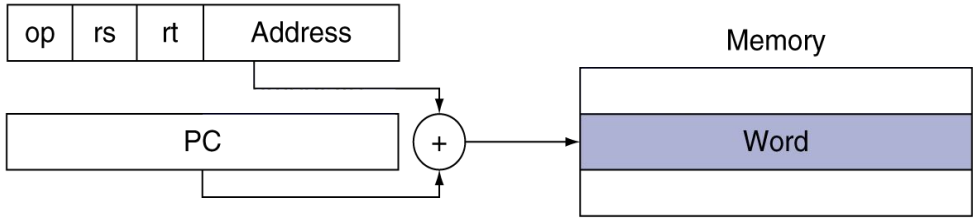
lw $t0, 40($s0)

Load/ Store



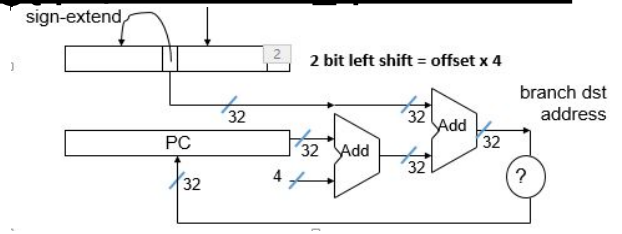
5 bits | 5 bits

4. PC-relative addressing

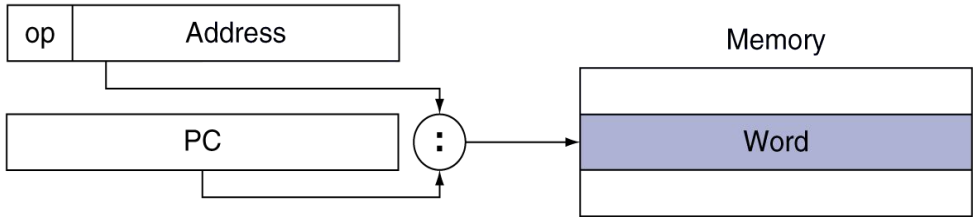


beq $t0, $t1, L1

Beq/ Bne



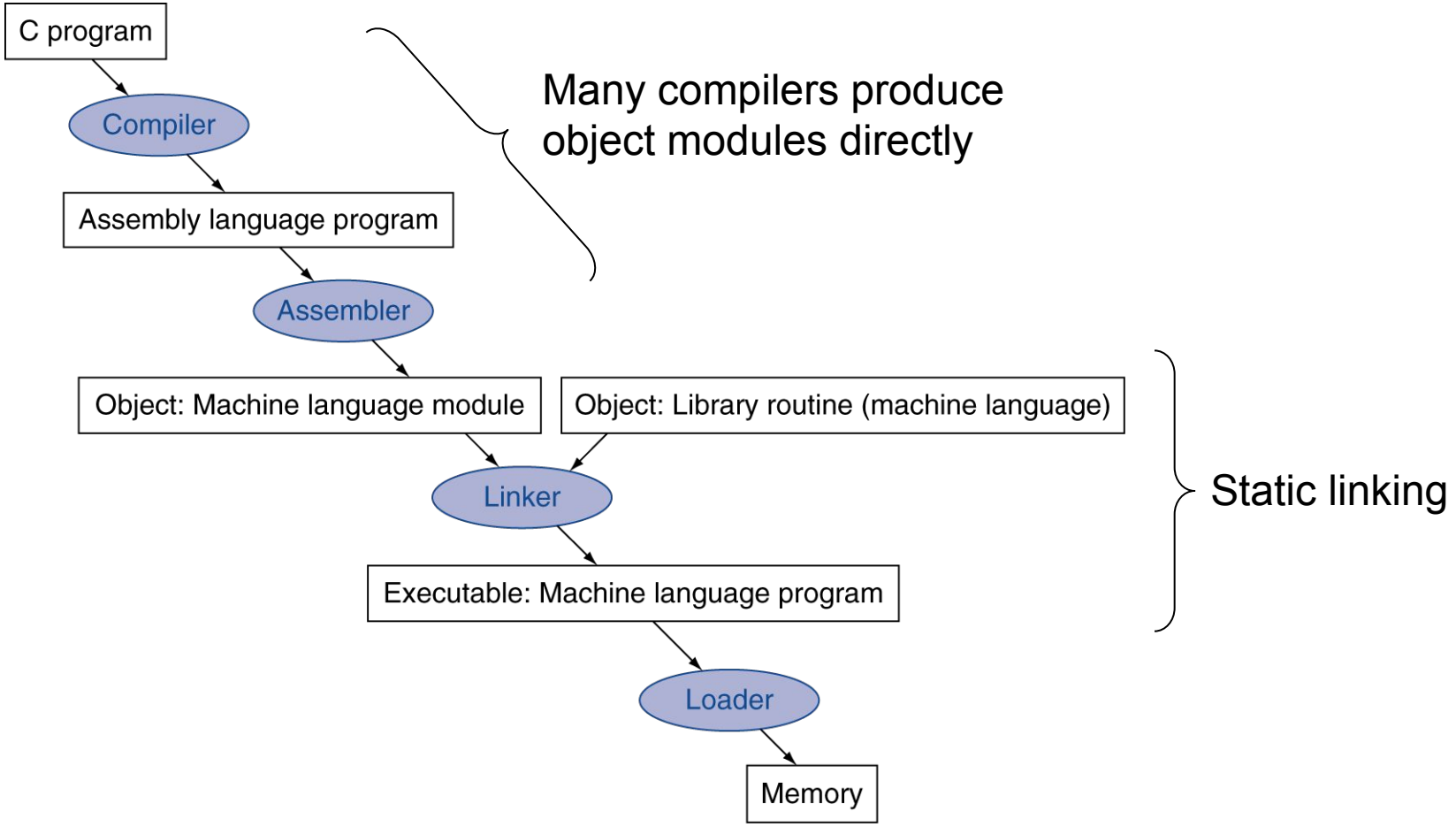
5. Pseudodirect addressing



j 1024

J type

Translation and Startup



Assembler Pseudoinstructions

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

```
move $to, $ti   → add $to, $zero, $ti
blt $to, $ti, L  →  slt $at, $to, $ti
                    bne $at, $zero, L
```

  - $at (register 1): assembler temporary