

Computer Organization and Architecture

Prepared by **Dr. Eng. Gamal Selim**

Lecture 03



Revision on Instruction set

Instruction Set

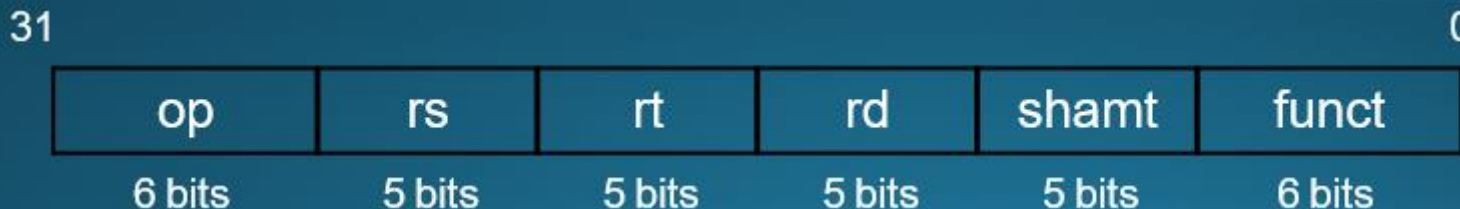
- The collection of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

Instruction Set Architecture

- Anything **programmers** need to know to make a binary machine language program work correctly, including:
 - Registers
 - Organization of programmable storage
 - Data types and data structures: encoding and representations
 - Instruction set
 - Instruction formats
 - Modes of addressing: accessing data items and instructions
 - Exceptional conditions

Arithmetic Operations

- Add and subtract, three **operands**
- Two sources and one destination
- add a, b, c # a gets $b + c$
- Operation, operator, operand
- All arithmetic operations have this form
- *Design Principle 1: Simplicity favors 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 operands from **registers**
- The size of a register in MIPS architecture is 32 bits, which is a **word**
- The **word** is the natural **unit of access** in a computer, usually a group of 32 bits; **corresponds to the size of a register**

Register Operands

- MIPS has a 32×32 -bit **register file**
 - Use for frequently accessed data
 - Cache of **cache**?
- Numbered 0 to 31
- Assembler names
 - \$t0, \$t1, ..., \$t9 for **temporary** values
 - \$s0, \$s1, ..., \$s7 for **saved** variables
 - **\$zero**
- *Design Principle 2: Smaller is faster Cf. main memory: millions of locations*

Register Operand Example

- C code:
 - $f = (g + h) - (i + j);$
 - $f, \dots, j$ in $\$s0, \dots, \$s4$
- Compiled MIPS code:

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

Memory Operands

- **Data transfer instruction:** before arithmetic operations can be done
 - **Load** values from memory into registers
 - **Store** result from register to memory
 - **RISC** (Reduced instruction set computing) vs. CISC
 - Only *load* and *store* instructions access memory
- Memory is **byte addressed**
 - Each address identifies an 8-bit byte
- **Alignment restriction:** words are aligned in memory
 - For lw/sw, the address must be a multiple of 4
 - If either of the **two least-significant bits** of the address are non-zero, an **Address Error exception** occurs

Addressing Alignment Constraints

- MIPS uses byte addressing for halfword, word, and doubleword accesses with the following alignment constraints:
 - Halfword accesses must be aligned on an **even byte boundary** (0, 2, 4...).
 - Word accesses must be aligned on a **byte boundary divisible by four** (0, 4, 8...).
 - Doubleword accesses must be aligned on a **byte boundary divisible by eight** (0, 8, 16...).

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
```

offset

base register

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

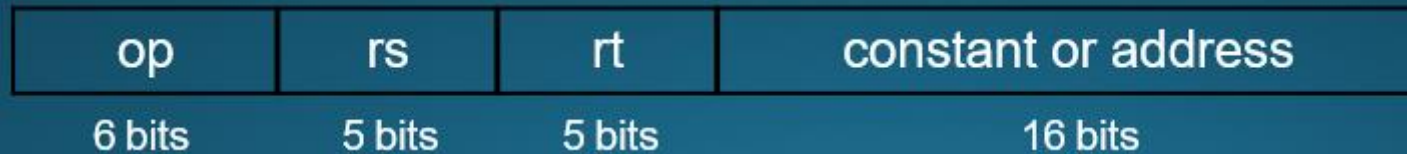
- No** subtract immediate instruction

- Just use a negative constant

addi \$s2, \$s1, -1

- Design Principle 3: Make the common case fast**

- Small constants are common
- Immediate operand **avoids a load** instruction



Unsigned Binary Integers

- Given an n-bit number

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$
- Example
 - 0000 0000 0000 0000 0000 0000 0000 1011₂
 - $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 - $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
 - 0 to +4,294,967,295

Decimal Numeral System - Base-10

Decimal numbers uses digits from 0..9.

These are the regular numbers that we use.

Example:

$$2538_{10} = 2 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 8 \times 10^0$$

Binary Numeral System - Base-2

Binary numbers uses only 0 and 1 digits.

B denotes binary prefix.

Examples:

$$10101_2 = 10101B = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = \\ 16 + 4 + 1 = 21$$

$$10111_2 = 10111B = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = \\ 16 + 4 + 2 + 1 = 23$$

$$100011_2 = 100011B = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ = 32 + 2 + 1 = 35$$

Octal Numeral System - Base-8

Octal numbers uses digits from 0..7.

Examples:

$$27_8 = 2 \times 8^1 + 7 \times 8^0 = 16 + 7 = 23$$

$$30_8 = 3 \times 8^1 + 0 \times 8^0 = 24$$

$$4307_8 = 4 \times 8^3 + 3 \times 8^2 + 0 \times 8^1 + 7 \times 8^0 = 2247$$

Hexadecimal Numeral System - Base-16

Hex numbers uses digits from 0..9 and A..F.

H denotes hex prefix.

Examples:

$$28_{16} = 28H = 2 \times 16^1 + 8 \times 16^0 = 40$$

$$2F_{16} = 2FH = 2 \times 16^1 + 15 \times 16^0 = 47$$

$$BC12_{16} = BC12H = 11 \times 16^3 + 12 \times 16^2 + 1 \times 16^1 + 2 \times 16^0 = 48146$$

2s-Complement Signed Integers

Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

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

Example

$$\begin{aligned} &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

Using 32 bits

$$-2,147,483,648 \text{ to } +2,147,483,647$$

2s-Complement Signed Integers

- Bit 31 (from 0) is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- 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 -2^{n-1}
 - Most-positive: 0111 1111 ... 1111 $+2^{n-1} - 1$
- $-(-2^{n-1})$ can't be represented

Signed Negation

- Complement and add 1

- Complement means $1 \rightarrow 0, 0 \rightarrow 1$

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

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

- Example: negate +2

- $+2 = 0000 \ 0000 \ \dots \ 0010_2$
- $-2 = 1111 \ 1111 \ \dots \ 1101_2 + 1$
 $= 1111 \ 1111 \ \dots \ 1110_2$

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
- **Sign Extension: replicate the sign bit to the left**
 - Cf. 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

MIPS R-format Instructions



- Instruction format
 - The layout of the instruction
- Instruction **fields**
 - op: operation code (opcode)
 - rs: first source **register number**
 - rt: second source **register number**
 - rd: destination **register number**
 - shamt: shift amount (00000, **useless for R-format**)
 - 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

00000010001100100100000000100000₂ = 02324020₁₆

Machine code

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

Logical Operations

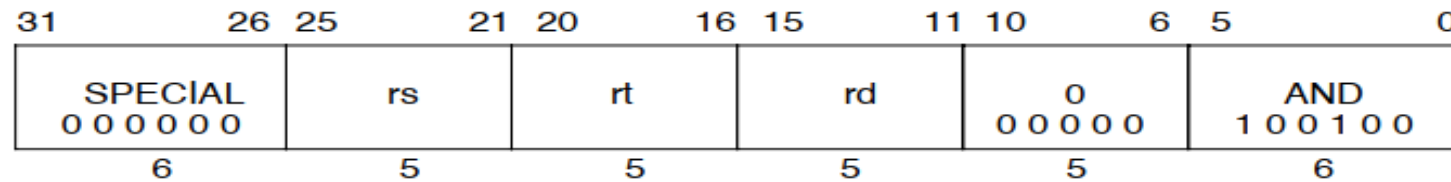
Instructions for bitwise manipulation

Operation	C	Java	MIPS
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

nor

AND Operations

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

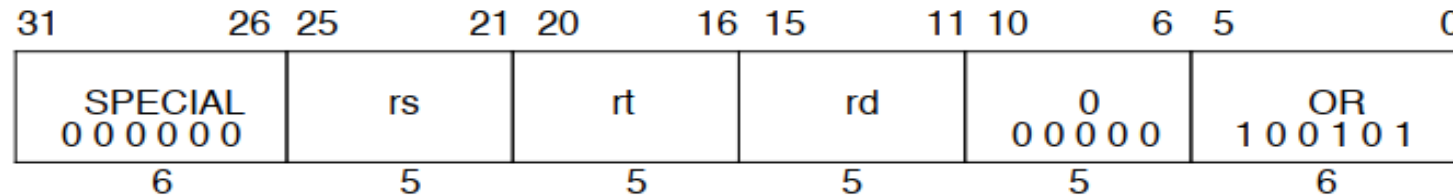


and \$t0, \$t1, \$t2

\$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

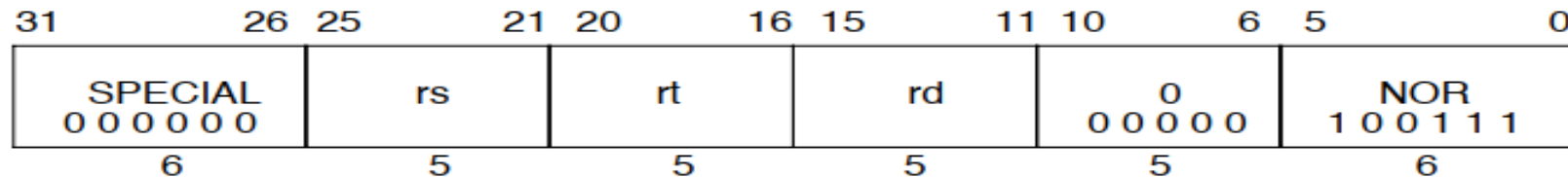


or \$t0, \$t1, \$t2

\$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

NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
 - No NOT operation, but NOR
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$



`nor $t0, $t1, $zero`

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

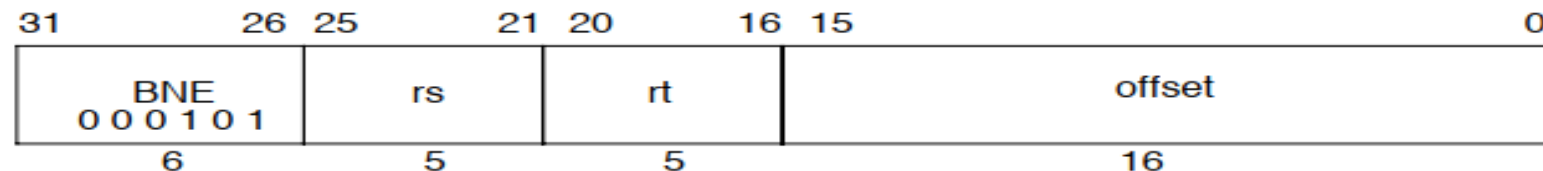
Conditional Operations

- The difference between a computer and a calculator is the ability to **make decisions**
- In high level language
 - *if* statement
 - *go to* statement
- Branch to a **labeled** instruction if a condition is true; **Otherwise**, continue sequentially
- **Conditional** and **unconditional** branches
- **PC-relative** and **absolute/register indirect** branch

Conditional Operations

■ Conditional branches

- `beq rs, rt, L1`
 - if (`rs == rt`) branch to instruction labeled L1;
- `bne rs, rt, L1`
 - if (`rs != rt`) branch to instruction labeled L1;
- Cf. ARM
 - `cmp r1, r2` **#CPSR** $\leq$ `r1-e2` in
 - `beq L1` **#test CPSR**, then..

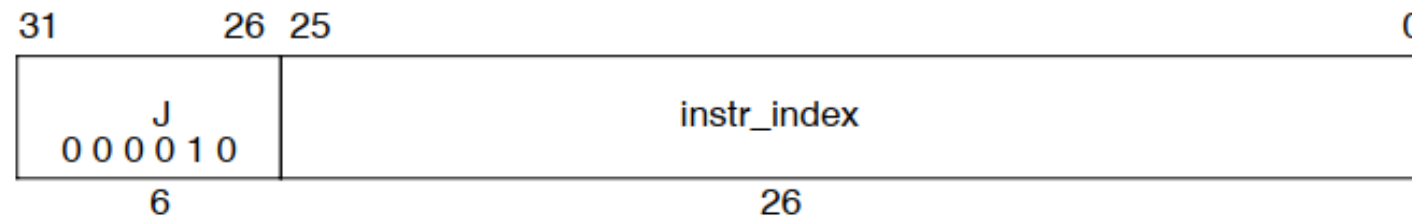


Conditional Operations

■ Unconditional branch

■ j L1

- Unconditional jump to instruction labeled L1
- **PC-region**: 256 MB aligned region



Compiling If Statements

■ C code:

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

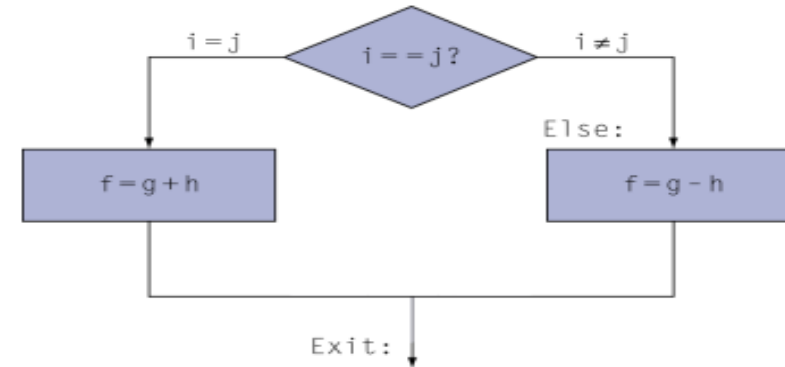
- f, g, ... in \$s0, \$s1, ...

■ Compiled MIPS code:

```

        bne $s3, $s4, Else      # <> ?
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   ...
```

Assembler calculates addresses



Compiling Loop Statements

■ C code:

```
while (save[i] == k) i += 1;    //cf. for
```

- i in \$s3, k in \$s5, base address saved in \$s6

■ Compiled MIPS code:

```

Loop:  sll    $t1, $s3, 2          #word:4B,*4
       add    $t1, $t1, $s6       #base+offset*4
       lw     $t0, 0($t1)         #load save[i]
       bne    $t0, $s5, Exit      # <> k ?
       addi   $s3, $s3, 1         # ==k, i += 1
       j      Loop
Exit:  ...

```

"C" program down to numbers

```
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
```

compiler

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

assembler

```
00000000101000010...  
000000000000110000...  
10001100011000100...  
10001100111100100...  
10101100111100100...  
10101100011000100...  
00000011111000000...
```

Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Multiplication and division
- **ALU: Arithmetic Logic Unit**
 - HW performs addition, subtraction, and usually logical operations such as AND and OR



مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY
BY EVERYONE'S SMART UNIVERSITY



Thank You

