

# Computer Architecture (ENE1004)

## ▼ Lec 4

## Lec 4: Instructions: Language of the Computer 3

### Review: Representing MIPS Instructions

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

- R-type and I-type
- Each type has a fixed, unified format (specific fields with specific sizes)  
(각 유형에는 고정된 통합 형식(특정 크기의 특정 필드)이 있습니다.)
- Opcode is unique for each instruction
- Registers as source/destination operands are specified their numbers

### Review: Example

- Assumption: \$t1 holds the base address of an integer array A, \$s2 corresponds to h
- $A[300] = h + A[300]$ ; is compiled to

```
lw $t0, 1200($t1) # temporary reg $t0 gets A[300]
add $t0, $s2, $t0 # temporary reg $t0 gets h + A[300]
sw $t0, 1200($t1) # store h + A[300] back into A[300]
```

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

## MIPS Instructions for Logical Operations: Shifts

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl

- Moving all the bits in a word to the left or right, filling the emptied bits with 0s (단어의 모든 비트를 왼쪽 또는 오른쪽으로 이동하고 비워진 비트를 0으로 채운다.)
  - shift left (sll): 왼쪽 이동 and shift right (srl): 오른쪽 이동
  - Example of shifting 9 by 4
    - 9 = 0000 0000 0000 0000 0000 0000 0000 1001
    - (To the left) 9 << 4 = 0000 0000 0000 0000 0000 0000 0000 1001  
0000 = 144 (= 9 \* 2^4)
    - (To the right) 9 >> 4 = 0000 0000 0000 0000 0000 0000 0000 0000  
0000 = 0
- Example

```
sll $t2, $s0, 4 # reg $t2 = reg $s0 << 4 bits
```

- The original value is in \$s0 and the result of shifting left by 4 goes into \$t2 (원래 값은 \$s0에 있고 4만큼 왼쪽으로 이동한 결과는 \$t0로 이동한다.)

	op	rs	rt	rd	shamt	funct
R-type	0	0	16	10	4	0

- (op = 0 + funct = 0) for sll , rd = 10 for \$t2, rt = 16 for \$s0, and shamt = 4 (rs is unused and set to 0)

## MIPS Instructions for Logical Operations: AND/OR

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

- Example
  - \$t2 = 0000 0000 0000 0000 0000 1101 1100 0000 = 3,520
  - \$t1 = 0000 0000 0000 0000 0011 1100 0000 0000 = 15,360
- AND: sets 1 in the bit only if both the corresponding bits of the operands are 1 (피연산자의 해당 비트가 모두 1인 경우에만 비트 1을 실행한다.)

```
and $t0, $t1, $t2 # reg $t0 = reg $t1 & reg $t2
```

- \$t0 = 0000 0000 0000 0000 0000 1100 0000 0000 = 3,072
- OR: sets 1 in the bit if either operand bit is a 1 (피연산자 비트가 1이면 비트에 1을 설정한다.)

```
or $t0, $t1, $t2 # reg $t0 = reg $t1 | reg $t2
```

- \$t0 = 0000 0000 0000 0000 0011 1101 1100 0000 = 15,808
- Immediate instructions are supported for AND and OR

MIPS 어셈블리 언어에서 "즉각적이다"라는 표현(영어로 "immediate")은 명령어가 리터럴 값(즉, 고정된 숫자 값)을 직접 포함하고 있다는 의미입니다. MIPS 아키텍처에서는 레지스터 간의 연산뿐만 아니라, 레지스터와 고정된 숫자 값(즉각값) 사이의 연산을 지원하는 명령어들이 있습니다. 이런 명령어들은 "즉각적 연산(immediate operations)"이라고 불립니다.

즉각적 명령어를 사용하면, 데이터를 메모리에서 불러오거나 다른 레지스터에서 전송하지 않고도 고정된 값과의 연산을 빠르게 수행할 수 있습니다. 이는 코드의 효율성을 높이고 실행 시간을 단축시키는 데 도움이 됩니다.

## MIPS Instructions for Logical Operations: NOR

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

- NOT: takes one operand and places a 1 in the bit if one operand bit is a 0, and vice versa (하나의 피연산자를 취하고 피연산자 비트가 0이면 비트에 1을, 그 반대의 경우 반대로 한다.)
  - MIPS designers did their best for keeping with the three-operand format (MIPS 설계자는 세 개의 연산자 형식을 유지하기 위해 최선을 다했다.)
  - MIPS designers decided to include NOR (NOT OR) instead of NOT (MIPS 설계자는 NOT 대신 NOR(NOT OR)을 포함하기로 결정했다.)
  - $A \text{ NOR } 0 = \text{NOT}(A \text{ or } 0) = \text{NOT}(A)$
  - If one operand is zero, NOR is equivalent to NOT (하나의 피연산자가 0인 경우 NOR은 NOT과 동일하다.)
- Example

```
nor $t0, $t1, $t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

- $\$t1 = 0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000 = 15,360$
- $\$t3 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = 0$
- $\$t1 \text{ OR } \$t3 = 0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000 = \$t1$
- $\sim(\$t1 \text{ OR } \$t3) = 1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111 = \text{NOT } \$t1$
- Immediate NOR is not supported, since its main use is to invert the bits (NOT) (즉시 NOR은 비트(NOT)를 반전하는 데 사용되므로 지원되지 않습니다.)

## MIPS Decision-Making Instructions (MIPS 의사 결정 지침)

- Computers have an ability to make decisions (컴퓨터는 의사 결정 능력이 있다.)
  - Based on input data or values created during computation, different instructions are executed (계산 중에 생성된 입력 데이터 또는 값에 따라 다양한 명령어가 실행된다.)

- Decision making is commonly represented in programming languages using if statements (의사 결정은 프로그래밍 언어에서 일반적으로 if문을 사용하여 표현된다.)
- MIPS includes two decision-making instructions (MIPS에는 두 가지 의사 결정 명령어가 포함되어 있다.)
- Branch if equal

```
beq register1, register2, L1
```

- L1 is a label that indicates a statement(instruction) in the code; recall goto statement in C (L1은 코드에서 명령문(명령어)을 나타내는 레이블이다. C에서 goto문을 떠올려라.)
- It jumps to the statement labeled L1 if the value in register1 does not equal the value in register2 (register1의 값이 register2의 값과 같으면 L1이라는 레이블이 붙은 문으로 점프한다.)
- Branch if not equal

```
bne register1, register2, L1
```

- It jumps to the statement labeled L1 if the value in register1 does not equal the value in register2 (register1의 값이 register2의 값과 같지 않으면 L1이라는 레이블이 붙은 문으로 점프한다.)
- We call the two instructions(명령어) "conditional branches"(조건 branches)
- "Unconditional branch"(무조건 branches) jump

```
j L1
```

- There is no condition
- It always jumps to the statement labeled L1 (which is an instruction): 항상 L1(명령어)이라는 레이블이 붙은 문으로 점프한다.

## MIPS Decision-making Instructions: if-else

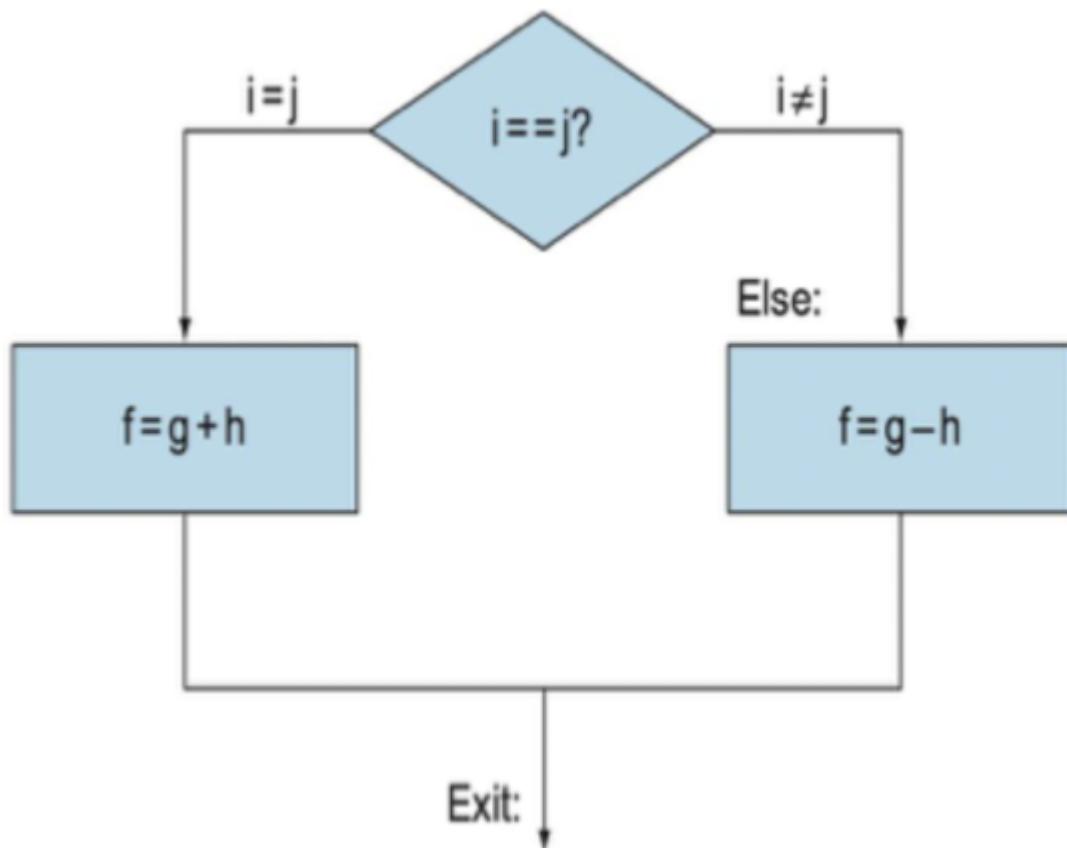
```
if (i == j)
    f = g + h;
```

```
else  
    f = g - h;
```

- f corresponds to \$s0, g to \$s1, h to \$s2, i to \$s3, j to \$s4
- is translated into

```
bne $s3, $s4, Else # go to Else if i != j  
add $s0, $s1, $s2 # f = g + h (skipped if i != j)  
j Exit # go to Exit  
Else: # here is the label Else  
    sub $s0, $s1, $s2 # f = g - h (skipped if i = j)  
Exit: # here is the label Exit
```

- Without the unconditional branch (j Exit), both add and sub are executed (무조건 분기(j Exit)가 없으면 add와 sub가 모두 실행됩니다.)



- An alternative:

```
beq $s3, $s4, Then # go to Then if i = j
```

- If you prefer this, where do you locate the case of  $i \neq j$ ?
- In general, the code will be more efficient if we test for “the opposite condition” to branch over the code that performs the subsequent then part of the if
- Else로 하는 것이 더 편함

## MIPS Decision-making Instructions: while

- Decisions are also important for iterating a computation, which are found in loops (결정은 루프에서 발견되는 계산을 반복하는 데에도 중요하다.)

```
while (save[i] == k)
    i += 1;
```

- i corresponds to \$s3, k to \$s5, the base of the array save is in \$s6
- is translated into

Loop:

```
sll $t1, $s3, 2 # temp reg $t1 = i * 4 (for the index in bytes)
add $t1, $t1, $s6 # $t1 = address of save[i]
lw $t0, 0($t1) # temp reg $t0 = save[i]
bne $t0, $s5, Exit # go to Exit if save[i] != k
addi $s3, $s3, 1 # i = i + 1
j Loop # go to Loop
Exit: # here is the label Exit
```

- Notice: Shift left logical “m” by “n” bits generates  $m * 2^n$
- sll \$t1, \$s3, 2: \$s3에 저장된 i 값을 4배한다. MIPS에서 배열의 index는 바이트 단위로 주소가 계산되므로, 4를 곱해야 원하는 원소의 주소를 얻을 수 있다. 이는 int형 배열에서 각 요소가 4바이트를 차지하기 때문이다.
- add \$t1, \$t1, \$s6: \$s6에 저장된 save 배열의 기본 주소에 \$t1을 더해 save[i]의 실제 메모리 주소를 계산한다.

- `lw $t0, 0($t1)`: 계산된 주소에서 값을 로드하여 임시 레지스터 `$t0`에 저장한다. 이 값은 `save[i]`에 해당한다.
- `bne $t0, $s5, Exit`: 임시 레지스터 `$t0`의 값(`save[i]`)과 `$s5`에 저장된 k값을 비교한다. 만약 같지 않다면 ( $\$t0 \neq \$s5$ ), EXIT 레이블로 branch하여 반복문을 종료한다.
- `addi $s3, $s3, 1`: i의 값을 1 증가시킨다. `$s3` 레지스터에 저장된 현재 값(i)에 1을 더해 다시 `$s3`에 저장한다.
- `j Loop`: 다시 Loop 레이블로 점프하여 반복문을 계속 실행한다.

## MIPS Decision-making Instructions: for

```
for (i = 0; i < 4; i++)
{
    // do something
}
```

- Do you have any idea for handling “`i < 4`”?
- It would be useful to see if a variable is less than another variable
- Two useful instructions, which can tell if a variable is less than another variable
  - Set less than

```
slt reg1, reg2, reg3 # reg1 = 1 if reg2 < reg3
```

- It compares two registers (`reg2` and `reg3`), and sets a register (`reg1`) to 1 if  $\text{reg2} < \text{reg3}$
- Otherwise (if  $\text{reg2} \geq \text{reg3}$ ), it sets `reg1` to 0

<code>slt reg1, reg2, reg3</code>	<code>reg1</code>
<code>reg2 &lt; reg3</code>	1
<code>reg2 ≥ reg3</code>	0

- Set less than immediate: `slti reg1, reg2, const` # `reg1 = 1 if reg2 < const`
  - Immediate version of `slt`

- It compares reg2 and constant, and sets a register (reg1) to 1 if  $\text{reg2} < \text{constant}$
- Do you have any idea for initializing a variable “ $i = 0$ ”?
- MIPS includes a register, named “\$zero”, which always holds a value of 0

```
add $s0, $zero, $zero # $s0 will hold a value of 0
```

register	assembly name	Comment
r0	\$zero	Always 0

```
for (i = 0; i < 4; i++)
{
    // do something
}
```

- $i$  corresponds to  $\$t0$
- is translated into

```

        add $t0, $zero, $zero # i is initialized to 0, $t0 = 0
Loop:
        slti $t1, $t0, 4 # $t1 = 0 if i >= 4
        beq $t1, $zero, Exit # go to Exit if $t1 = 0
        # do something
        addi $t0, $t0, 1 # i++
        j Loop
Exit:
```

## MIPS Decision-making instructions: switch/case

- A straightforward way to implement switch is using a chain of if-then-else statements
  - Try by yourself

```
switch (x) {  
    case 0:  
        // do something for case 0  
        break;  
    case 1:  
        // do something for case 1  
        break;  
    case 2:  
        // do something for case 0  
        break;  
    default:  
        // default action  
}
```

```
# Assume x is in register $t0  
  
# Check if x is equal to 1  
beq $t0, 1, case_1  
  
# Check if x is equal to 2  
beq $t0, 2, case_2  
  
# Check if x is equal to 3  
beq $t0, 3, case_3  
  
# If none of the cases match, jump to the default case  
j default_case  
  
# Case 1  
case_1:  
    # Do something for case 1
```

```
j end_switch

# Case 2
case_2:
    # Do something for case 2
    j end_switch

# Case 3
case_3:
    # Do something for case 3
    j end_switch

# Default case
default_case:
    # Do something if none of the cases match

# End of switch statement
end_switch:
```