

---

# Instructions: Language of the Computer

---

Computer Architecture

2019 1학기

한양대학교 공과대학 컴퓨터소프트웨어학부  
홍석준

# 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

# Instruction Set Architecture (ISA)

- ISA, or simply architecture – the abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, I/O, ...
  - Enables implementations of varying cost and performance to run identical software
- The combination of the basic instruction set (the ISA) and the operating system interface is called the application binary interface (ABI)
  - ABI – The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

# The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://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  $32 \times 32$ -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:

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

- $f, \dots, j$  in  $\$s0, \dots, \$s4$

- Compiled MIPS code:

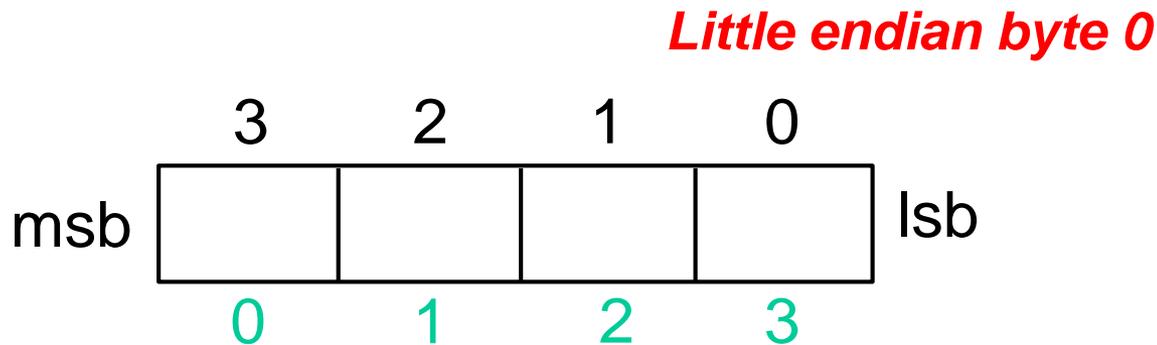
add \$t0, \$s1, \$s2

add \$t1, \$s3, \$s4

sub \$s0, \$t0, \$t1

# Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual **bytes** in memory
  - **Alignment restriction** – the memory address of a **word** must be on natural word boundaries (a multiple of 4 MIPS-32)
- **Big Endian**: leftmost byte is word address
  - IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- **Little Endian** : rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



*big endian byte 0*

# 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 I

- 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

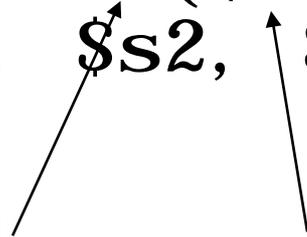
- 4 bytes per word

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

offset

t

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
```

# 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!

# 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 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 registers are named with fewer bits than memory location)



# MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 ( <b>hardware</b> )	n.a.
\$at	1	<b>reserved</b> for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	<b>yes</b>
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	<b>yes</b>
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	<b>yes</b>
\$sp	29	stack pointer	<b>yes</b>
\$fp	30	frame pointer	<b>yes</b>
\$ra	31	return addr ( <b>hardware</b> )	<b>yes</b>

# 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

# 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`

# 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 1011<sub>2</sub>  
= 0 + ... +  $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$   
= 0 + ... + 8 + 0 + 2 + 1 = 11<sub>10</sub>
- Using 32 bits
  - 0 to +4,294,967,295

# 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

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

- 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  $1 \rightarrow 0, 0 \rightarrow 1$

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

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

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

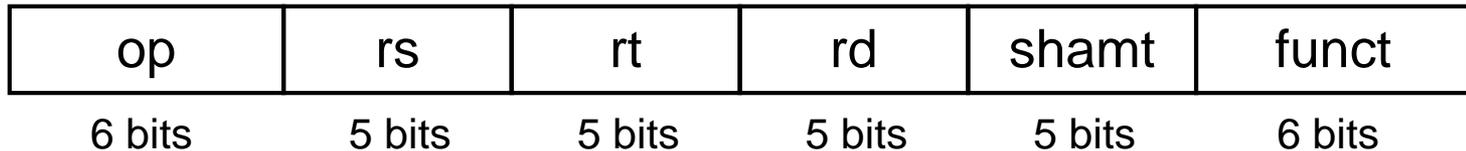
# Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi` : extend immediate value
  - `l b`, `l h`: 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



## ■ 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

$$00000010001100100100000000100000_2 = 02324020_{16}$$

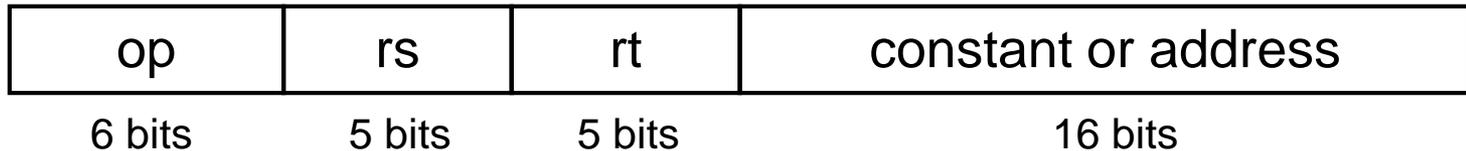
# Hexadecimal

- 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

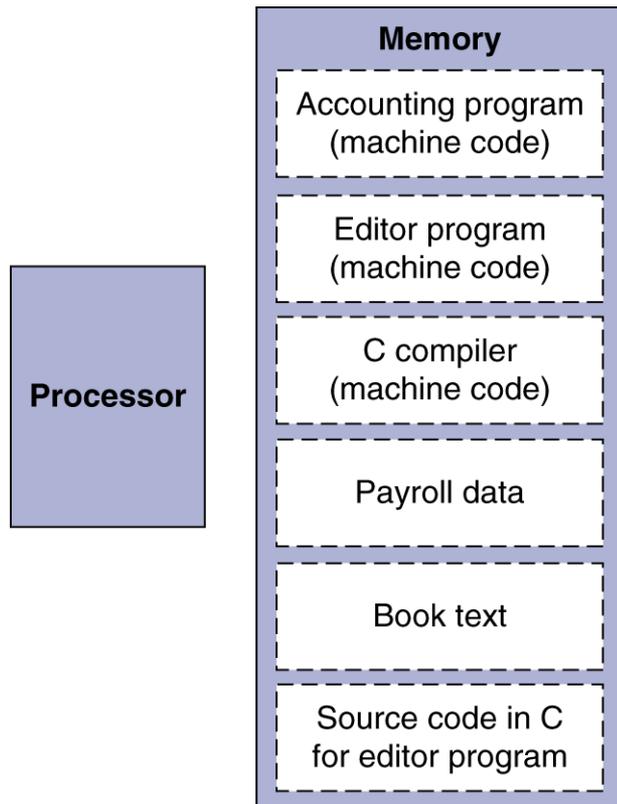
# MIPS I-format Instructions



- 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

# Stored Program Computers

## The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

# 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



- 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

# NOT Operations

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

```
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

# Compiling If Statements

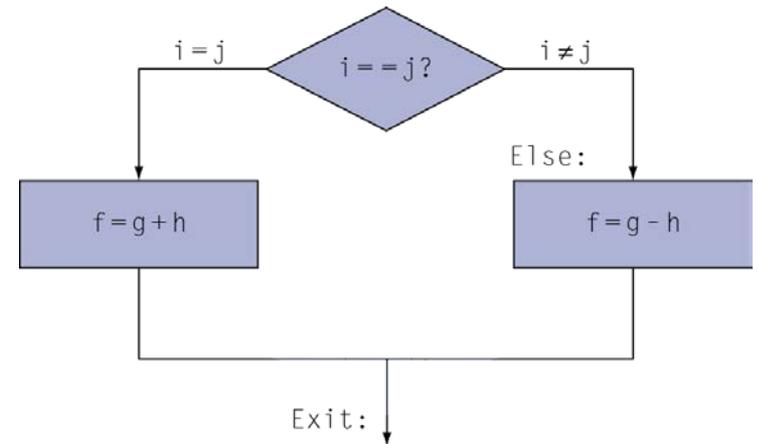
- C code:

```
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;
```

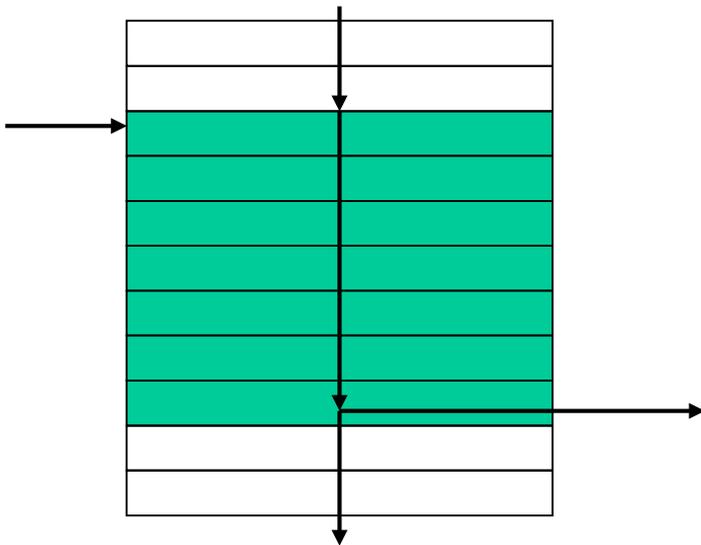
- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

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

# Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

# More Conditional Operations

- 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 ( $rs < \text{constant}$ )  $rt = 1$ ; else  $rt = 0$ ;
- Use in combination with `beq`, `bne`

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

# 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: `sl t, sl ti`
- Unsigned comparison: `sl tu, sl tui`
- Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `sl t $t0, $s0, $s1 # signed`
    - $-1 < +1 \Rightarrow \$t0 = 1$
  - `sl tu $t0, $s0, $s1 # unsigned`
    - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$