

# FU05 Computer Architecture

## 3. Assembly Language (アセンブリ言語)

**Ben Abdallah Abderazek**

E-mail: [benab@u-aizu.ac.jp](mailto:benab@u-aizu.ac.jp)

# Levels of Program Code

## ■ High-level language

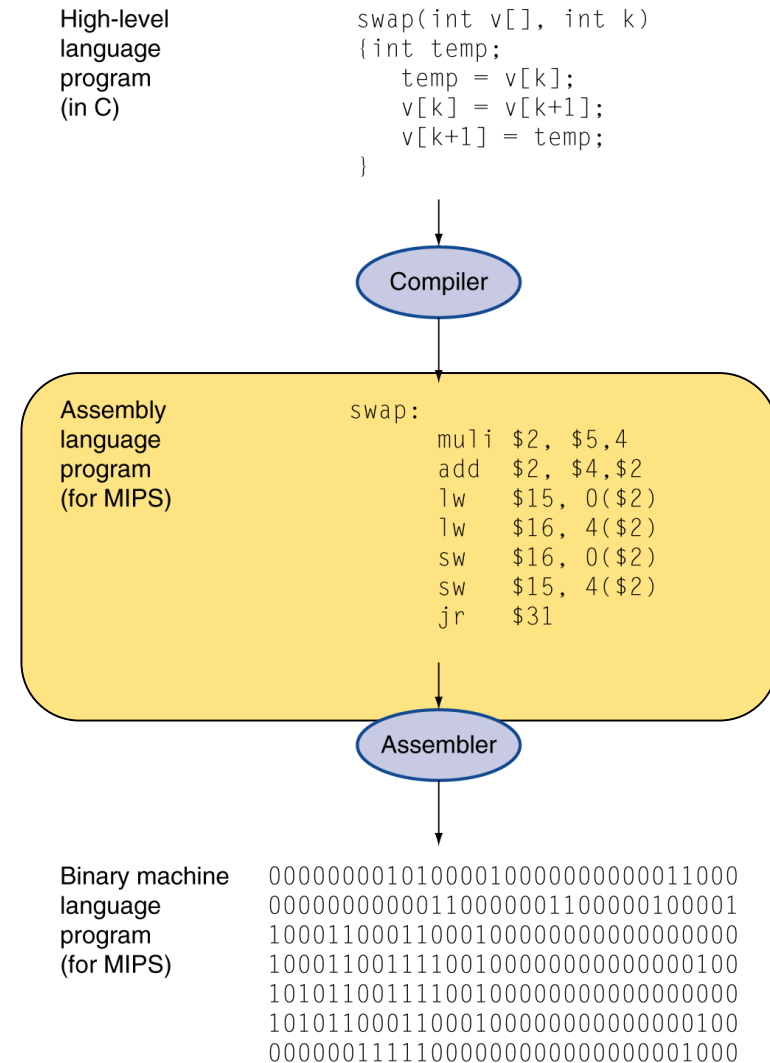
- Level of abstraction closer to problem domain
- Provides for productivity and portability

## ■ Assembly language

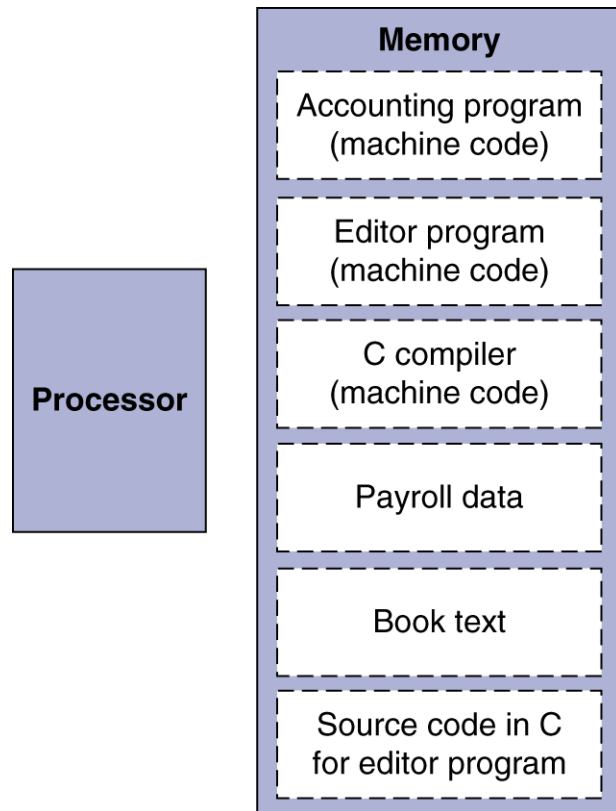
- Textual representation of instructions

## ■ Hardware representation

- Binary digits (bits)
- Encoded instructions and data



# Stored Program Computers



- 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

# Why use Assembly Language Programming ?

- When speed is critical. Maybe use Assembly Language for critical components.
- When no High Level Language compiler is available for a machine.
- To exploit specialized machine capabilities.
- When one wants to debug particularly complex structures.

# Instruction Set

- The language of a computer
- Different computers have different instruction sets
- Early computers had very simple instruction sets
- Many modern computers also have simple instruction sets

# Instruction Set Architecture (ISA)

## ■ Instruction Categories

- Computational
- Load/Store
- Jump and Branch
- Floating Point
  - coprocessor
- Memory Management
- Special

### 3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	sa	funct	R format
OP	rs	rt	immediate			I format
OP	jump target					J format

# Arithmetic Operations

- Add and subtract, three operands (オペランド)
  - 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***

# 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`

# 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 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
```

# 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. **Why?**

**Answer:**

# 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

# MIPS register 0

- MIPS register 0 (\$zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers  
`add $t2, $s1, $zero`



# 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>

# Representing Instructions

- Instructions are encoded in binary
  - Called **machine code** (機械語)
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
- 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

# Conditional Operations (分岐命令)

- 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

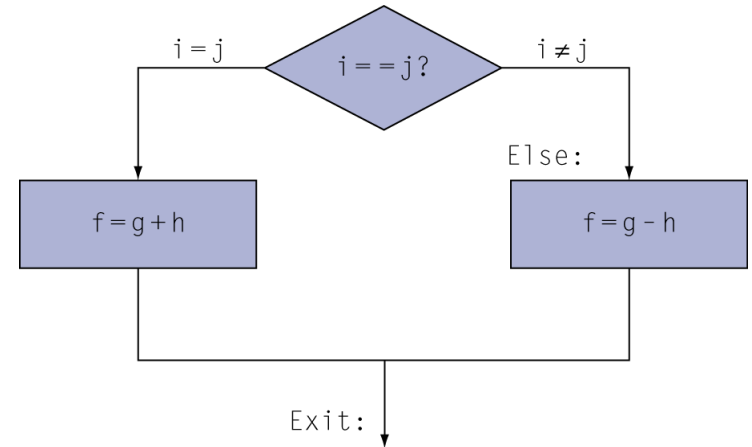
- C code:

```
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:  ...
```

# 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 Addressing

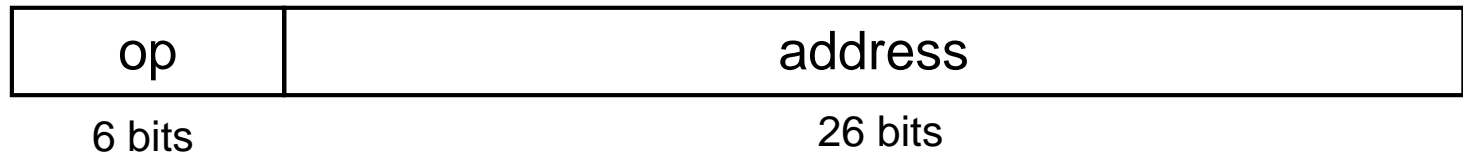
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address =  $PC + \text{offset} \times 4$
  - PC already incremented by 4 by this time

# Jump Addressing

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



- (Pseudo)Direct jump addressing
  - Target address =  $PC_{31...28} : (\text{address} \times 4)$

# Target Addressing Example

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

Loop:	sll	\$t1, \$s3, 2	80000	0	0	19	9	4	0
	add	\$t1, \$t1, \$s6	80004	0	9	22	9	0	32
	lw	\$t0, 0(\$t1)	80008	35	9	8	0		
	bne	\$t0, \$s5, Exit	80012	5	8	21	2		
	addi	\$s3, \$s3, 1	80016	8	19	19	1		
	j	Loop	80020	2	20000				
Exit:	...		80024						

# 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:     ...
```

# Example

Instruction	Example	Meaning
jump	j 2500	go to 10000
jump register	jr \$31	go to \$31
jump and link	jal 2500	\$31 = PC+4; go to 10000
branch on equal	beq \$1, \$2, 25	if(\$1==\$2) go to PC+4+100
branch on not equal	bne \$1, \$2, 25	if(\$1!=\$2) go to PC+4+100
set on less than	slt \$1, \$2, \$3	if(\$2<\$3) \$1=1 else \$1=0
set less than imm.	slti \$1, \$2, 100	if(\$2<100) \$1=1 else \$1=0

# Addressing Mode Summary

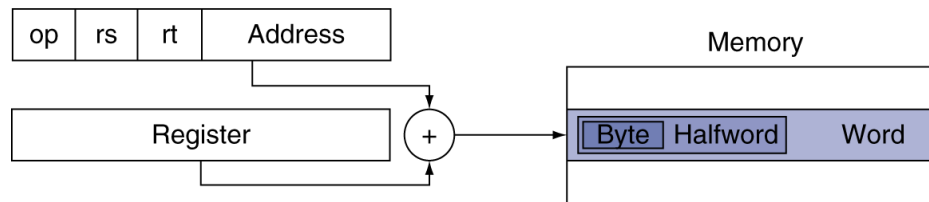
## 1. Immediate addressing



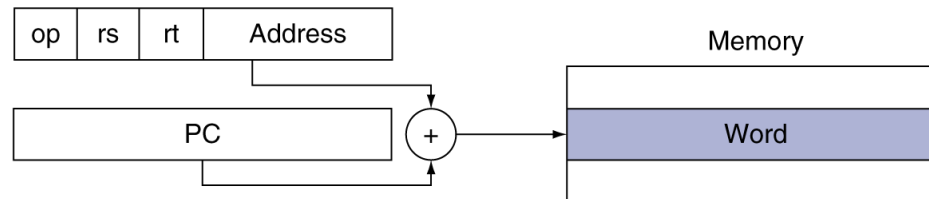
## 2. Register addressing



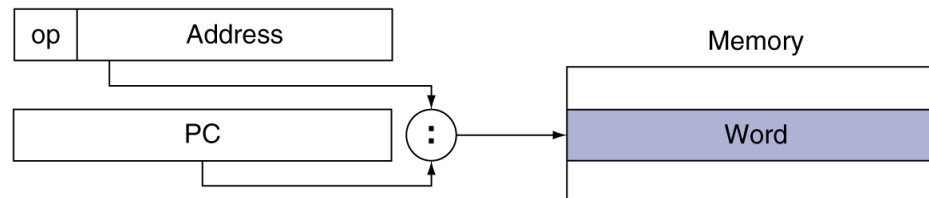
## 3. Base addressing



## 4. PC-relative addressing



## 5. Pseudodirect addressing





# Practice Problem (練習問題)

**2.1** For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables *f*, *g*, *h* and *i* are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.

$f = g + (h - 5);$

# Practice Problem (練習問題)

Solution

**2.1**

# Practice Problem (練習問題)

Solution

**2.1**

addi f, h, -5 (note, no subi)

add f, f, g