MIPS Assembly Language

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

(draft)

MIPS Operands and Operations

- A MIPS instruction consists of an **operation** and zero or more **operand**
- **Operation** specifies the function of the instruction
- **Operand** specifies data to use with the instruction
- Example:

Destination Operand

```
add $t0, $t1, $t2
```

operation (addition)

Source Operands

MIPS Operations

- **Arithmetic Operations**: addition, subtraction, ...
- **Logical Operations**: AND, OR, ...
- **Shift**: Moves bits left or right
- **Compare**: equal, less than, etc
- **Load/Store**: load data from memory or store data to memory
- **Branch/Jump**: make decisions or go to a certain instruction
- **System control**

MIPS Operands

- Registers (32-bit general-purpose register): (`$t0`, `$v0`, ...)
- Fixed registers: `Hi` and `Lo` registers
- Memory location (Base Address + Offset)
  - Base Address: from a register
  - Offset: a constant value
- Immediate Value (constant)

MIPS Registers

- In MIPS, there are 32 general-purpose registers (32-bit word each)

Name	Number	Use	Name	Number	Use
$zero	0	The Constant Value 0	$s0	16	Saved Temporary
$at	1	Assembler Temporary	$s1	17	Saved Temporary
$v0	2	Return Value	$s2	18	Saved Temporary
$v1	3	Return Value	$s3	19	Saved Temporary
$a0	4	Argument	$s4	20	Saved Temporary
$a1	5	Argument	$s5	21	Saved Temporary
$a2	6	Argument	$s6	22	Saved Temporary
$a3	7	Argument	$s7	23	Saved Temporary
$t0	8	Temporary	$t8	24	Temporary
$t1	9	Temporary	$t9	25	Temporary
$t2	10	Temporary	$k0	26	Reserved for OS Kernel
$t3	11	Temporary	$k1	27	Reserved for OS Kernel
$t4	12	Temporary	$gp	28	Global Pointer
$t5	13	Temporary	$sp	29	Stack Pointer
$t6	14	Temporary	$fp	30	Frame Pointer
$t7	15	Temporary	$ra	31	Return Address

- `Hi` and `Lo` are used to store result from multiplication operation
  - Multiply two 32-bit number results in a 64-bit number
- `$pc` or program counter
  - Store the memory address of the current or next instruction
  - Cannot directly manipulate by programmer

General-Purpose Registers

- Can be use to store values or as operands in instructions
- Need to follow conventions:
  - `$zero` or `$0`: always 0 and cannot be used to store any other value
  - `$v0` and `$v1`: are used for storing return values
  - `$a0` to `$a3`: are used for storing arguments/parameters for functions
  - `$t0` to `$t9`: can be used freely without any restrictions
  - `$s0` to `$s7`: values in these registers should be stored somewhere (generally stack) before we can use them and restore them back after we finish using them
  - `$at`, `$k0` and `$k1` are reserved for assembler and OS kernel. **Do not use them.**
  - `$gp`, `$sp`, and `$fp` will be explained later. **Do not use them yet.**
  - `$ra` is used for storing return address before calling a function

Instruction Encoding

- Computers only understand binary numbers
- Instructions are encoded in binary number
- Assembler translates an assembly program into a series of binary numbers
- All MIPS instructions are 32-bit wide
- Processor fetch an instruction (32-bit binary number) and decode it to see what it needs to do
- Encoding from assembly program to machine code is simple (done by software)
- Decoding from machine code in hardware is hard
- We need formats of 32-bit instructions which will help processors decode them

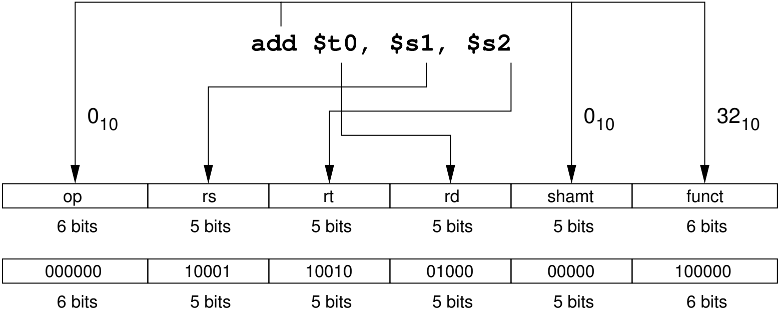
R-Type

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

- **op**: Operation (opcode)
- **rs**: First register source operand
- **rt**: Second register source operand
- **rd**: Destination register operand
- **shamt**: Shift amount
- **funct**: Function code
- For R-type, all operands must be registers

Example (add)

- For the instruciton `add`:
  - **op**: must be 0 (000000_2)
  - **shamt**: must be 0 (00000_2)
  - **funct**: must be 32_{10} (100000_2)

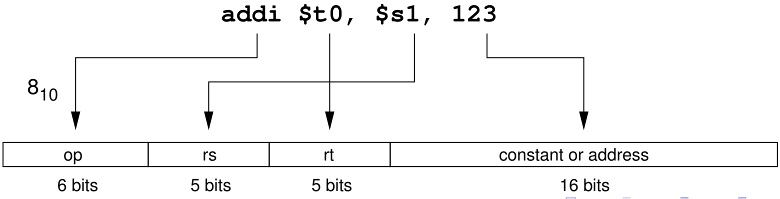


- **Note**: `$t0` is 8 (01000_2), `$s1` is 17 (10001_2), and `$s2` is 18 (10010_2).

I-type (Immediate)

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

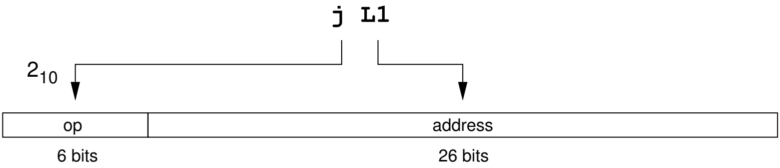
- **op**: Operation (opcode)
- **rs**: First register source operand
- **rt**: Second register source operand **OR** destination register operand
- **Constant or Address**: (Immediate field) constant operand
- For example: `addi`, **op** field must be 8_{10} (01000_2)



J-type (Jump)

op	address
6 bits	26 bits

- **op**: Operation (opcode)
- **address**: the 26-bit **word** address to jump to
- For example, jump instruction (`j`), the **op** must be 2_{10} (000010_2)



MIPS Fields

- R-type:

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

- I-type:

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- J-type:

op	address
6 bits	26 bits

Arithmetic Instructions

- Format:

```
<op>  <Destination>, <First Source>, <Second Source>
```

- All operands must be registers
- MIPS field: R-type
- Examples:

```
add $t0, $t1, $t2     # $t0 = $t1 + $t2
sub $a0, $v0, $s3     # $a0 = $v0 - $s3
```

Logical Instructions

- Big-wise logic operations
- Format:

```
<op> <Destination>, <First Source>, <Second Source>
```

- MIPS Field: R-type
- Examples:

```
and $t0, $t1, $t2    # $t0 = $t1 & $t2
or  $s3, $s4, $t1    # $s3 = $s4 | $t1
nor $s4, $t3, $t4    # $s4 = ~($t3 | $t4)
nor $s6, $v0, $0     # $s6 = ~($v0 | 0) = ~$v0
xor $a2, $t3, $s4    # $a2 = $t3 xor $s4
```

Arithmetic and Logical Instructions with Immediate

- Some operations require a constant
  - x = y + 2;
  - y = y - 1;
  - z = x & 0x1a2b
- **Note**: 0x1a2b represents 4-digit hexadecimal number (16-bit binary)
- Format:

```
<op> <Destination>, <First Source>, <Constant>
```

- MIPS Field: I-type
- Examples:

```
addi $t1, $s1, 35                # $t1 = $s1 + 35
andi $t5, $t6, 0x34ab            # $t5 =
                                 # $t6 & 0011010010101011b
ori  $t2, $v0, 0x0f0f            # $t2 =
                                 # $v0 | 0000111100001111b
```

- **Note**: 0000111100001111b represents 16-bit binary number

Long Immediate Values

- Note that the arithmetic and logical instruction with immediate field have I-type
- The size of constant field is only 16 bit

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- If an instruction requires 32-bit constant:
  - load upper half of the constant into a register
  - or the lower half with that register

Long Immediate Values

- For examle, suppose we need to use the following 32-bit constant:

  0000 1111 0000 1111 0101 0101 0101 0101

  we need to perform the following:

  ```
  lui $t0, 0000111100001111b
  ```

  Now the content of the register `$t0` is

  0000 1111 0000 1111 0000 0000 0000 0000

  then

  ```
  ori $t0, $t0, 0101010101010101b
  ```

  Now the content of the register `$t0` is

  0000 1111 0000 1111 0101 0101 0101 0101

Pseudo-Ops

- Pseudo-ops are instructions that are not implemented in hardware (processor do not understand)
- The purpose of pseudo-ops is to help programmer just like high-level language
- A pseudo-op will be translated to a series of instruction that processor understand
- For example: `li` (load immediate 16/32 bit)

```
li  $t0, 0x00012345
```

may be translated to

```
lui $at, 0x0001
ori $t0, $at, 0x2345
```

Pseudo-Ops

- For example: `move` move the value in one register to another

```
move $t1, $t2   # $t1 = $t2
```

  may be translated to

```
addu $t1, $t2, $0
```

- For example: `la` (load address)

```
la  $a0, L1
```

  may be translated to

```
lui $at, <upper half of L1>
ori $a0, $at, <lower half of L1>
```

Interacting with the OS

- Often times, we need help from the OS
  - display numbers on screen
  - read keyboard input
  - terminate the program
  - open/close/read/write files
- These are called operating system services
- It is a special instruction called `syscall`
  - It is a software interrupt to notify OS to perform actions
  - Need to indicate the service type
  - May need to pass argument value for a certain service

Some Useful `syscall`

- To use `syscall`, we need to perform the following:
  1. Provide arguments (in specific registers) if required
  2. Set the service type ID in the register `$v0`
  3. Call `syscall`
- Useful `syscall`:
  - **Print integer**: Store integer to print in `$a0` and set `$v0` to 1
  - **Read integer**: Set `$v0` to 5, after `syscall`, `$v0` holds the integer read from keyboard
  - **Print string**: Store the memory address of the string print (terminated by `null`) and set `$v0` to 4
  - **Exit (halt)**: Set `$v0` to 10
- See MARS documents for more information about `syscall`

Example: Print an Integer

- Consider the following Java code

```
int x = 5;
int y = 9;

y = x + y;

System.out.println(y);
```

- When we want to show an integer number on the console screen, we need help from the OS

Example: Print an Integer

- The code from previous slide can be translated to assembly code as shown below:

```
.text
    add  $t0, $zero, 5      # $t0 = 5
    addi $t0, $t0, 9        # $t0 = $t0 + 9
    add  $v0, $zero, 1      # Syscall 1: print integer
    add  $a0, $t0, $zero    # $a0 = $t0: integer to be printed
    syscall                 # Print integer
    add  $v0, $zero, 10     # Syscall 10: terminate program
    syscall                 # Terminate program
```

- Note that `.text` defines the text segment of a program (the segment where instructions are stored in the main memory)

Example: Print a String

- Consider the following Java code:

```
System.out.println(``Hello World!'');
```

- The above code can be translated to assembly code as shown below:

```
.data
    msg: .asciiz "Hello World!"
.text
    addi $v0, $zero, 4    # Syscall 4: print string
    la   $a0, msg         # Set $a0 to memory address of msg
    syscall               # Print the string msg
    addi $v0, $zero, 10   # Syscall 10: terminate program
    syscall               # Terminate program
```

- Note that `.data` defines the data segment of a program (the segment where data are stored in the main memory)
- Note that `.asciiz` create an array of characters that is terminated by the `null` character

Example: Keyboard Input

- Consider the following Java code:

```
Scanner s = new Scanner(System.in);

System.out.print("Please enter an integer number: ");
int x = s.nextInt();
System.out.println("Thanks for entering " + x + ".");
```

- We need help from OS to display messages
- We need help from OS to get an input from the keyboard

Example: Keyboard Input

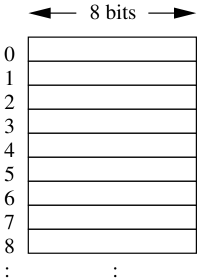
```
.data
    prompt1: .asciiz "Please enter an integer number: "
    msg:     .asciiz "Thanks for entering "
    period:  .asciiz "."
.text
    addi $v0, $zero, 4    # Syscall 4: print string
    la   $a0, prompt1     # Set $a0 to memory address of prompt1
    syscall               # Print the string prompt1
    addi $v0, $zero, 5    # Syscall 5: read integer
    syscall               # Read integer and store it in $v0
    add  $t0, $zero, $v0  # Store the input integer in $t0
    la   $a0, msg         # Set $a0 to memory address of msg
    addi $v0, $zero, 4    # Syscall 4: print string
    syscall               # Print the string msg
    addi $v0, $zero, 1    # Syscall 1: print integer
    add  $a0, $zero, $t0  # Set integer to be printed
    syscall               # Print integer
    addi $v0, $zero, 4    # Syscall 4: print string
    la   $a0, period      # Set $a0 to memory address of period
    syscall               # Print the string period
    addi $v0, $zero, 10   # Syscall 10: terminate program
    syscall               # Terminate program
```

Memory Transfer Instructions

- **Load** Read data from memory
- **Store** data into memory
- The following are memory transfer instructions in MIPS (32-bit architecture):
  - **Word** (32 bits) or `int` in Java
    - `lw`: load word
    - `sw`: store word
  - **Half-word** (16 bits) or `short` in Java
    - `lh`: load half word
    - `sh`: store half word
    - `lhu`: load half word unsigned
  - **Byte** (8 bits) or `byte` in Java
    - `lb`: load byte
    - `sb`: store byte
    - `lbu`: load byte unsigned

Memory View

- Memory is a large one-dimensional 8-bit array



- A memory location can be accessed using index the same as in array
- Notation: `Memory[4]` means the memory location referred by index 4 (the fifth byte)

Effective Address Calculation

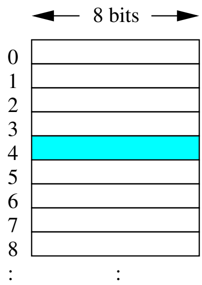
- In MIPS, the effective memory address specified as

$$< \texttt{immediate value} > (< \texttt{\$register} >)$$

  - `$register` stores a base address
  - `immediate value` is use to determine the offset from the base address
  - The `immediate value` is a 16-bit value that can be either positive or negative value
  - The actual location is `immediate value` plus the value stored in the `$register`
- Example:

```
addi $t0, $zero, 32    # Set $t0 to 32
lw   $s0, 12($t0)      # the address is 32 + 12 = 44
sw   $s1, -24($t0)     # the address is 32 - 24 = 8
```

Accessing a Byte



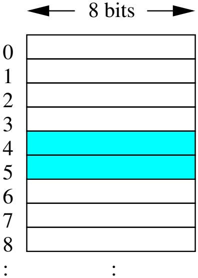
- The highlighted byte (location 4) can be loaded or stored using the following instructions:

```
addi $t0, $zero, 4
lw   $s0, 0($t0)
sw   $s1, 0($t0)
```

or

```
add  $t0, $zero, $zero
lw   $s0, 4($t0)
sw   $s1, 4($t0)
```

Accessing a Half-word



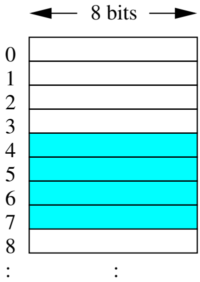
- The highlighted half-word (starting at location 4) can be loaded or stored using the following instructions:

```
addi $t0, $zero, 4
lh   $s0, 0($t0)
sh   $s1, 0($t0)
```

or

```
add  $t0, $zero, $zero
lh   $s0, 4($t0)
sh   $s1, 4($t0)
```

Accessing a Word



- The highlighted word (starting at location 4) can be loaded or stored using the following instructions:

```
addi $t0, $zero, 4
lw   $s0, 0($t0)
sw   $s1, 0($t0)
```

or

```
add  $t0, $zero, $zero
lw   $s0, 4($t0)
sw   $s1, 4($t0)
```

Load Address

- Recall that values stored in variables are actually stored in memory locations
- For example:

```
int x = 5;
```

  - a memory location is assigned to the variable named `x`
  - the value 5 is stored in that memory location
- For example:

```
char[] str = "Hello World!";
```

  - a memory location is assigned to the variable named `str`
  - The character `H` is stored in that memory location
  - The character `e` is stored in the next memory location

Load Address in Assembler

- To create variables in assembler

```
.data
    num:   .word     973
    str:   .asciiz   "Hello'
```

Names Types Values

- To access variables, we need to load their address (`la`) into registers first

```
la  $t0, num
la  $t1, str
```

- Then use load or store instruction to access them

```
lw  $s0, 0($t0)
lb  $s1, 0($t1)
```

- Note that `la` is a pseudo-op which is translated into the following sequence of instructions:

```
lui $at, <upper half of the address>
ori $t0, $at, <lower half of the address>
```

Exercise

- Create a word (integer) variables named `x` with initial value 0, and y with initial value 5

Exercise

- Create a word (integer) variables named `x` with initial value 0, and y with initial value 5

```
.data
        x:  .word
        y:  .word 5
```

- Print the value of `y` on the console

Exercise

- Create a word (integer) variables named x with initial value 0, and y with initial value 5

```
.data
        x:  .word
        y:  .word 5
```

- Print the value of y on the console

```
.text
        la   $s0, y          # Load address of y
        lw   $a0, 0($s0)     # Load data stored in y
        addi $v0, $zero, 1   # Service 1: print integer
        syscall              # Print integer
```

- Terminate the program

Exercise

- Create a word (integer) variables named `x` with initial value 0, and `y` with initial value 5

```
.data
        x: .word
        y: .word 5
```

- Print the value of `y` on the console

```
.text
        la   $s0, y           # Load address of y
        lw   $a0, 0($s0)      # Load data stored in y
        addi $v0, $zero, 1    # Service 1: print integer
        syscall               # Print integer
```

- Terminate the program

```
        addi $v0, $zero, 10  # Service 10: exit
        syscall               # Exit
```

- Note: The directive `.space` (with a number of bytes) should be used when creating a variable without initial value

```
.data
        z: .space 4           # 4 bytes
```

Exercise

- Add 9 to `y` and store it into `x` and print the value of `x` on the console

Exercise

- Add 9 to y and store it into x and print the value of x on the console

```
.data
        x: .word 0             # Variable x
        y: .word 5             # Variable y initialized to 5
.text
        la   $s0, x            # Load the address of x to $s0
        la   $s1, y            # Load the address of y to $s1
        lw   $t0, 0($s1)       # Load value stored in y to $t0
        addi $t0, $t0, 9       # $t0 = $t0 + 9
        sw   $t0, 0($s0)       # Store $t0 into x
        addi $a0, $zero, $t0   # $a0 = $t0
        addi $v0, $zero, 1     # Service 1: print integer
        syscall                # Print integer
        addi $v0, $zero, 10    # Service 10: exit
        syscall                # Exit
```

Example

- Consider the following code:

```
int a[4] = {1,3,2,5};
int temp;

temp = a[1];
a[1] = a[2];
a[2] = temp;
```

- What is the MIPS code corresponds to the above code?

Example

```
.data
        a: .word 1,3,2,4
.text
        la   $s0, a           # $s0 is the address of a[0]
        addi $t0, $zero, 1    # Use $t0 as the index
        sll  $t1, $t0, 2      # Use $t1 as the offset (index * 4)
        add  $t2, $s0, $t1    # $t2 is the address of a[index]
        lw   $t3, 0($t2)      # $t3 = a[1]
        lw   $t4, 4($t2)      # $t4 = a[2]
        sw   $t4, 0($t2)      # a[1] = $t4
        sw   $t3, 4($t2)      # a[2] = $t3
        addi $v0, $zero, 10   # Service 10: exit
        syscall               # Exit
```

Note that we can use effective addresses `0($t2)` and `4($t2)` in this case because they are next to each other.

Memory Organization

- MIPS use byte addressing
  - Each memory location is 8-bit wide (1 byte)
  - Each byte can be accessed directly
  - For example, memory locations 0, 1, 2, ...
- A word in MIPS is 32-bit wide
  - Each word requires 4 bytes
  - Words are aligned in memory
  - Two Least Significant Bits (LSBs) of every word memory addresses are always 0s
  - For example, memory location 0, 4, 8, 12, ...
- Half-word in MIPS is 16-bit wide
  - Each half-word requires 2 bytes
  - Half-words are aligned in memory
  - The LSB of every half-word memory addresses is always 0.
  - For example, 0, 2, 4, 6, ...

Big-Endian vs Little-Endian

- Suppose we have a 32-bit number 0x12345678 (hexadecimal)
- How do we store this number as a word at memory location 0?

Big-Endian vs Little-Endian

- Suppose we have a 32-bit number 0x12345678 (hexadecimal)
- How do we store this number as a word at memory location 0?
- Divide the number into 4 bytes ($12\ 34\ 56\ 78_{16}$)
- **Big-Endian**: most significant byte in the smallest address
  - Location 0: 12_{16}
  - Location 1: 34_{16}
  - Location 2: 56_{16}
  - Location 3: 78_{16}
- **Little-Endian**: least significant byte in the smallest address
  - Location 0: 78_{16}
  - Location 1: 56_{16}
  - Location 2: 34_{16}
  - Location 3: 12_{16}
- What is MIPS in MARS (Big-Endian or Little-Endian)?

Big-Endian vs Little-Endian

```
.data
    x: .word 0x12345678
.text
    la   $t0, x          # Set $t0 to the address of x
    lbu  $t0, 0($t0)     # Load a byte at location x
    addi $v0, $zero, 10  # Service 10: exit
    syscall              # Exit
```

- After `lbu`, check the value of the register `$t0`
  - If `$t0` is 12, it is a Big Endian
  - If `$t0` is 78, it is a Little Endian
- Why `lbu` instead of `lb`?

Shift Instructions

- Change bits position inside a word
- Format:

```
<op> <Destination>, <Source>, <Shift Amount>
```

- MIPS Field: R-type
- Example:

```
sll  $t0, $t1, 6     # $t0 = $t1 << 6
sllv $t1, $t2, $t3   # $t1 = $t2 << $t3 (lower 5 bits)
srl  $s0, $s1, 3     # $s0 = $s1 >> 3
srlv $t1, $t2, $t3   # $t1 = $t2 >> $t3 (lower 5 bits)
```

- Shift Right Arithmetic (`sra`) reserves the sign of the number
  - If MSB is 0, shift in with 0
  - If MSB is 1, shift in with 1

Exercise: Shift Instruction

- Suppose we have four 8-bit numbers 0xaa, 0xbb, 0xcc, and 0xdd stored in registers $t0, $t1, $t2, and $t3 respectively.
- Write a code that generates a 32-bit number 0xaabbccdd from registers $t0 to $t3.

```
.text
        addiu $t0, $zero, 0xaa   # Set $t0 to 0xaa
        addiu $t1, $zero, 0xbb   # Set $t1 t0 0xbb
        addiu $t2, $zero, 0xcc   # Set $t2 to 0xcc
        addiu $t3, $zero, 0xdd   # Set $t3 to 0xdd
        or    $t4, $zero, $t0    # Now $t4 is 0xaa
        sll   $t4, $t4, 8        # Now $t4 is 0xaa00
        or    $t4, $t4, $t1      # Now $t4 is 0xaabb
        sll   $t4, $t4, 8        # Now $t4 is 0xaabb00
        or    $t4, $t4, $t2      # Now $t4 is 0xaabbcc
        sll   $t4, $t4, 8        # Now $t4 is 0xaabbcc00
        or    $t4, $t4, $t3      # Now $t4 is 0xaabbccdd
```

Controls

- In high-level language, we can choose to execute a set of statements or not using the `if` statement.

Line	High-Level Language	Assembly Language
1.	`if(condition)`	`Go to line 5 if the condition is false`
2.	{	
3.	`:`	`Instruction(s) in if body`
4.	}	
5.	`:`	`Instructions after if statement`

Controls

- In high-level language, we can choose to execute a set of statements or another set of statements using the `if-else` statement

Line	High-Level Language	Assembly Language
1.	`if(condition)`	`Go to line 7 if the condition is false`
2.	{	
3.	:	`Instruction(s) in if body`
4.	}	`Go to line 9`
5.	`else`	
6.	{	
7.	:	`Instruction(s) in else body`
8.	}	
9.	:	`Instruction after if-else statement`

Controls

- In high-level language, we can choose to execute among sets of statements using the `else-if` construct

Line	High-Level Language	Assembly Language
1.	`if(condition1)`	`Go to line 5 if the condition1 is false`
2.	{	
3.	:	`Instruction(s) in if body`
4.	}	`Go to line 13`
5.	`else if(condition2)`	`Go to line 11 if the condition2 if false`
6.	{	
7.	:	`Instruction(s) in second if body`
8.	}	`Go to line 13`
9.	`else`	
10.	{	
11.	:	`Instruction(s) in else body`
12.	}	
13.	:	`Instruction after if-else statement`

Branches

- In MIPS, you can jump to a certain instruction (referred by its location) based on either equal or not equal condition
- These are called **conditional branches**

```
beq  $t0, $t1, L1      # If $t0 == $t1 go to L1
bne  $t0, $t1, L2      # If $t0 != $t1 go to L2
```

- MIPS also support **unconditional branches** (jump)

```
j    L3                # Go to L3
```

- In assembly, we use a label to reference to an address in memory
  - The actual location of an instruction in memory can be changed during programming
  - By using labels, we do not have to keep track of the actual locations

Set Less Than

- Conditional branches in MIPS only support equal to or not equal to
- Note that the conditions in higher-level languages include less than, less than or equal to, greater than, and greater than or equal to.
- MIPS has basic instructions called **Set Less Than**
  - `slt $t0, $t1, $t2`:
    - **Set Less Than**: Set `$t0` to 1 if `$t1` is less than `$t2`. Otherwise, `$t0` will be 0.
  - `slti $t0, $t1, 100`
    - **Set Less Than Immediate**: Set `$t0` to 1 if `$t1` is less than 100. Otherwise, `$t0` will be 0.
  - `sltiu $t0, $t1, 100`
    - **Set Less Than Immediate Unsigned**: Treat the value in `$t1` as an unsigned number. Set `$t0` to 1 if `$t1` is less than 100. Otherwise, `$t0` will be 0.
  - `sltu $t0, $t1, $t2`
    - **Set Less Than Unsigned**: Treat values in `$t1` and `$t2` as unsigned numbers. Set `$t0` to 1 if `$t1` is less than `$t2`. Otherwise, `$t0` will be 0.

Using Branches for if Statement

- Equal to:

Line	High-Level Language	Assembly Language
1.	`if($t0 == $t1)`	`bne $t0, $t1, label`
2.	{	
3.	:	`...`
4.	}	
5.	:	`label: ...`

- Not Equal to:

Line	High-Level Language	Assembly Language
1.	`if($t0 != $t1)`	`beq $t0, $t1, label`
2.	{	
3.	:	`...`
4.	}	
5.	:	`label: ...`

Using Branches for if Statement

- Less Than:

Line	High-Level Language	Assembly Language
0.		`slt $t2, $t0, $t1`
1.	`if($t0 < $t1)`	`beq $t2, $zero, label`
2.	{	
3.	:	...
4.	}	
5.	:	`label: ...`

- Less Than or Equal to:

Line	High-Level Language	Assembly Language
0.		`slt $t2, $t1, $t0`
1.	`if($t0 <= $t1)`	`bne $t2, $zero, label`
2.	{	
3.	:	...
4.	}	
5.	:	`label: ...`

- Note that the condition `$t0 <= $t1` is the same as `!($t1 < $t0)`

Using Branches for if-else Statement

Line	High-Level Language	Assembly Language
1.	`if($t0 != $t1)`	`        beq $t0, $t1, else`
2.	{	
3.	:	`        ...`
4.	}	`        j   label`
5.	`else`	
6.	{	
7.	:	`else:   ...`
8.	}	
9.	:	`label:  ...`

Exercise

- Convert the following program into MIPS assembly code:

```
Scanner in = new Scanner(System.in);

System.out.print("Please enter an integer number: ")

int x = in.nextInt();

if(x % 2 == 0)
{
    System.out.println(x + " is an even number.");
}
else
{
    System.out.println(x + " is an odd number.");
}
```

Solution

```
.data
        prompt: .asciiz "Please enter an integer number: "
        evenmsg: .asciiz " is an even number."
        oddmsg: .asciiz " is an odd number."
.text
        la   $a0, prompt           # Set $a0 to prompt
        addi $v0, $zero, 4         # Service 4: print string
        syscall                    # Print string
        addi $v0, $zero, 5         # Service 5: read integer
        syscall                    # Read integer
        add  $s0, $zero, $v0       # Store read integer in $s0
        add  $a0, $zero, $v0       # Set integer to print
        addi $v0, $zero, 1         # Service 1: print integer
        syscall                    # Print integer
        andi $s0, $s0, 1           # Retrive the LSB
        bne  $s0, $zero, odd       # If LSB != 0 go to odd
        la   $a0, evenmsg          # Set $a0 to evenmsg
        j    print                 # Go to print
odd:    la   $a0, oddmsg           # Set $a0 to oddmsg
print:  addi $v0, $zero, 4         # Service 4: print string
        syscall                    # Print string
        addi $v0, $zero, 10        # Service 10: exit
        syscall                    # Exit
```

Loops

- We can use conditional/unconditional branches and set less than instructions to construct loop
- Note that the following code

```
for(initial expression; loop condition; loop expression)
{
   loop body;
}
```

is the same as

```
initial expression;

while(loop condition)
{
   loop body;
   loop expression;
}
```

- Note that converting a `while` loop to assembly code is simpler than converting from a `for` loop.

The `while` Loop

High-Level Language	Assembly Language
`i = 0;`	`add $t0, $zero, $zero`
	`loop: slti $t1, $t0, 10`
`while(i < 10)`	`beq $t1, $zero, exit`
`{`	
`:`	`...`
`i = i + 1;`	`addi $t0, $t0, 1`
`}`	`j loop`
`:`	`exit: ...`

- Can we branch when $t1 is equal to 10?

```
      add  $t0, $zero, $zero
      addi $t5, $zero, 10
loop: beq  $t1, $t5, exit
       :
exit: ...
```

The do Loop

High-Level Language	Assembly Language
`do`	
`{`	
`  :`	`loop:  ...`
`  :`	`       ...`
`}`	`       slt $t2, $t0, $t1`
`while($t0 < $t1)`	`       bne $t2, $zero, loop`
`:`	`       ...`

Exercise

- Consider the following program:

```
char[] str = {'H','e','l','l','o',' ',
              'W','o','r','l','d','!','\0'};

int length = 0;

while(str[length] != '\0')
{
    length = length + 1;
}

System.out.println("The length is " + length);
```

Solution

```
.data
        str: .asciiz "Hello World!"
        msg: .asciiz "The length is "
.text
        la   $s0, str              # Set $s0 to point to str
        add  $t0, $zero, $zero     # Set length to 0
loop:   add  $t1, $t0, $s0         # Add legnth to the address of str
        lbu  $s1, 0($t1)           # Load a character into $s1
        beq  $s1, $zero, exit      # If $s1 == 0, go to exit
        addi $t0, $t0, 1           # Increase length by 1
        j    loop                  # Go to loop
exit:   la   $a0, msg              # Set $a0 to point to msg
        addi $v0, $zero, 4         # Service 4: print string
        syscall                    # Print string
        add  $a0, $zero, $t0       # Set $a0 to length
        addi $v0, $zero, 1         # Service 1: print integer
        syscall                    # Print integer
        addi $v0, $zero, 10        # Service 10: exit
        syscall                    # Exit
```

Address in I-type

- Conditional branches instructions use I-type
- The address field of the I-type instruction is 16-bit wide
  - This is a **signed** two's complement number
  - The address field is used to specify the number of instructions to be skipped (forward and backword)
- This value is used to adjust the program counter ($pc)
- Note that the program counter is 32-bit wide but the address field is only 16-bit wide
  - 16-bit address field is needed to be sign-extended to a 32-bit number
- Note that MIPS uses byte addressing (each byte can be accessed directly)
  - The number of instructions to be skipped must be changed to the number of bytes to be skipped
  - Since all MIPS instructions are 32-bit wide, each instruction takes 4 bytes
  - This can be done by multiplying the number of instructions to be skipped by 4

Address in I-type

- The new address can be calculated by

$$PC + 4 + (\text{sign-extend(16-bit address field)} << 2)$$

- This is called **relative addressing** (address related to program counter in this case)
- Example: Suppose the instruction `beq $t0, $t1, exit` is located at the address 16, and the instruction at the `exit:` label is located at the address 32
  - From the above formula, we need to solve

$$32 = 16 + 4 + (\text{sign-extend(16-bit address field)} << 2)$$

which gives us

$$\text{sign-extend(16-bit address field)} << 2 = 12$$

or

$$\text{sign-extend(16-bit address field)} = 3$$

Thus the actual code for the instruction `beq $t0, $t1, exit` will be `beq $t0, $t1, 3`

Sign Extension

- The purpose of sign extension is to extend a number with lower number of bits to higher number of bits where its value is preserved.
- For example
  - The value 5 in 4-bit representation is 0101_2
  - 0101_2 can be extended to 8-bit representation as 00000101_2
  - Note that the value is still 5
- Another example
  - The value -4 in 4-bit representation is 1100_2
  - 1100_2 can be extended to 8-bit representation as 11111100_2
  - Note that the value is still -4
- Simply extend the most significant bit (signed bit)
- For a positive number, it is easy to see that the value is preserved
- What about negative number?

Recall that the value of a 4-bit representation $x_3x_2x_1x_0$ in two's complement is

$$-(x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Consider the first term, if x_3 is 0, the first term is 0. If x_3 is 1, the first time is -8. Suppose we extend this number to 5-bit representation by copying the value of the most significant bit (x_3), we get $x_3x_3x_2x_1x_0$ and its value is

$$-(x_3 \times 2^4) + (x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Now consider the first two terms. If x_3 is 0, the first two term is 0. If x_3 is 1, the first two terms is -16 + 8 = -8. Notice that it is the same value as the first term before sign extension. Suppose we extend this number to 6-bit representation by copying the value of the most significant bit (x_3, we get $x_3x_3x_3x_2x_1x_0$ and its value is

$$-(x_3 \times 2^5) + (x_3 \times 2^4) + (x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Now consider the first three terms. If x_3 is 0, the first three terms is 0. If x_3 is 1, the first three terms is $-32 + 16 + 8 = -8$. Again, it is the same. Therefore, by extending the most significant bit, the value is preserved.

Jump Addressing

- Instruction jump (`j`) uses J-type (26-bit address field)
- Jump does not use relative addressing
- Since each instruction is a word (32-bit) wide, jump uses word addressing instead of byte addressing
  - Note that for byte addressing to access a word, the last two least significant bit are always 0 any way.
  - This also increase the range of jump
- After multiply by 4 (shift left by 2), we get the byte address
- But it is only $26 + 2 = 28$ bits (we need 32-bit for program counter)
- MIPS takes the most significant four bits from the original program counter to fill it to 32 bits.

Procedures/Functions

- We use procedures/functions (methods) often in high-level programming language
  - The `main` function calls function `insert`
  - The function `insert` may call function `isEmpty`

```
public static void main(String[] args)
{
    :
    insert(...);
    :
}

private static void insert(...)
{
    :
    if(isEmpty())
    :
}

private static boolean isEmpty()
{
    :
}
```

Procedures/Functions

- From previous code:
  - Function `main` calls function `insert`
  - Function `insert` calls function `isEmpty`
  - When function `isEmpty` is done, it goes back to the `if` statement of the function `insert`
  - When function `insert` is done, it goes back to the next statement after the statement `insert(...);`
- To simulate function calls in assembly, we need a way to:
  - call a function
  - return back to caller
  - pass arguments
  - return values

Procedure Call

- Simply jump to the first instruction of the procedure using label
- When the procedure is done, it needs to jump back to the instruction immediately after the call
- Need to tell the procedure where to jump back using the register `$ra` (return address)
- Example: To call a procedure at the label `myFunction`, use

```
jal   myFunction    # Jump and Link to myFunction
```

- The instruction `jal myFunction` performs two things:
  - Set `$ra` to PC + 4 (next instruction after `jal`)
  - Set PC to `PC[31:28]`|`(myFunction << 2)`

Procedure Return

- When a function is finished, it needs to return back to the instruction immediately after the call
- The address of that instruction is stored in `$ra`
- Simply use

```
jr $ra    # Jump back to the address stored in $ra
```

- The instruction `jr $ra` simply sets PC to the value stored in `$ra`

Exercise

- Write a procedure named `_greeting`
  - This procedure prints the string `"Hello "`
- Write the main program that calls the procedure `_greeting` and then prints the string `"How are you?"`

Solution

```
.data
        asking: .asciiz "How are you?"
.text
        jal  _greeting          # Call the procedure _greeting
        la   $a0, asking        # Set $a0 to the string asking
        addi $v0, $zero, 4      # Service 4: print string
        syscall                 # Print string
        addi $v0, $zero, 10     # Service 10: exit
        syscall                 # Exit

_greeting:
.data
        str: .asciiz "Hello. "
.text
        la   $a0, str           # Set $a0 to the string str
        addi $v0, $zero, 4      # Service 4: print string
        syscall                 # Print string
        jr   $ra                # Go back to caller
```

Arguments and Return Values

- Register conventions:
  - `$a0` – `$a3`: Four arguments for passing values to called procedure
  - `$v0` and `$v1`: Two values returned from called procedure
  - `$ra`: return address register (set by caller and used by return)
- Sending Arguments:
  - Caller stores argument values in registers `$a0` to `$a3`
  - Called procedure will know that argument values are stored in those register
- Returning Values:
  - Called procedure stores return values in register `$v0` and `$v1`
  - Caller will know that return values are stored in those registers

Exercise

- Suppose we do not have the multiplication operator (*)
- Create a function named `_multi(int a, int b)` that returns the result of `a` multiplied by `b`
- For simplicity, both `a` and `b` are greater than or equal to 0

Exercise

- Suppose we do not have the multiplication operator (∗)
- Create a function named `_multi(int a, int b)` that returns the result of `a` multiplied by `b`
- For simplicity, both `a` and `b` are greater than or equal to 0
- You may come up with the following Java code:

```
int _multi(int a, int b)
{
   int result = 0;
   int i = 0;

   while(i < b)
   {
      result = result + a;
      i = i + 1;
   }

   return result;
}
```

Solution

```
.text
        addi $a0, $zero, 5      # Set value of the first argument
        addi $a1, $zero, 9      # Set value of the second argument
        jal  _multi             # Call the procedure _multi
        add  $a0, $zero, $v0    # Set $a0 to returned value (from _multi)
        addi $v0, $zero, 1      # Service 1: print integer
        syscall                 # Print integer
        addi $v0, $zero, 10     # Service 10: exit
        syscall                 # Exit

_multi:
        add  $v0, $zero, $zero  # Set result to 0
        add  $t0, $zero, $zero  # Set counter to 0
loop:   slt  $t1, $t0, $a1      # $t1 is 1 if counter < $a1
        beq  $t1, $zero, done   # If $t1 == 0 go to done
        add  $v0, $v0, $a0      # result = result + $a0
        addi $t0, $t0, 1        # Increase counter by 1
        j    loop               # Go back to loop
done:   jr   $ra                # Go back to caller
```

Comments

- It is a good idea to put comment in detail about the procedure
- For example, the procedure `_multi`

```
# _multi - Perform multiplication
#
# Arguments:
#   $a0 - multiplicand
#   $a1 - multiplier
# Return values:
#   $v0 - result of $a0 * $a1
#
# Note: Trash values stored in register $t0 and $t1

_multi:
   :
```

Call Chains

- Imagine the main program calls procedure `X` and procedure `X` calls procedure `Y`
- According to the conventions, we use `$ra` for return address
  - The main program sets `$ra` and calls procedure `X`
  - The procedure `X` sets `$ra` and calls procedure `Y`
  - The previous value stored in `$ra` is destroyed
- If a procedure needs to call another procedure, it must backup `$ra` first
  - Simple solution (for now): Store `$ra` in an unused register
- Leaf procedure (does not make any calls): does not need to save `$ra`

Exercise

- Write a procedure named `_power(int x, int y)` that calculates `x` to the power of `y` (x^y) by using procedure `_multi`

Exercise

- Write a procedure named `_power(int x, int y)` that calculates `x` to the power of `y` (x^y) by using procedure `_multi`
- You may come up with the following Java code:

```
int _power(int x, int y)
{
   int result = 1;
   int i = 0;

   while(i < y)
   {
      result = _multi(result, x);
      i = i + 1;
   }

   return result;
}
```

Solution

```
_power:
        add   $s0, $zero, $a0      # Save $a0 in $s0 (x)
        add   $s1, $zero, $a1      # Save $a1 in $s1 (y)
        add   $s7, $zero, $ra      # Save $ra in $s7
        addi  $s2, $zero, 1        # Set result to 1
        add   $s3, $zero, $zero    # Set counter to 0
ploop:  slt   $s4, $s3, $s1        # $s4 is 1 if counter < $s1
        beq   $s4, $zero, pdone    # If $s4 == 1 go to pdone
        add   $a0, $zero, $s2      # Set $a0 to result
        add   $a1, $zero, $s0      # Set $a1 to x
        jal   _multi               # Call procedure _multi
        add   $s2, $zero, $v0      # Save return value in $s2
        addi  $s3, $s3, 1          # Increase counter by 1
        j     ploop                # Go back to ploop
pdone:  add   $ra, $zero, $s7      # Restore $ra back from $s7
        jr    $ra                  # Go back to caller
```

Sharing Registers

- **Caller**: The procedure that calls other procedure(s)
- **Callee**: The procedure that is called by other procedure(s)
- In MIPS, there are limited number of registers
- Callee may want to use same registers as caller
  - Values stored in those registers were destroyed after the callee use them
  - Usually temporary saved registers (`$s0` to `$s7`)
  - Same situation as in `$ra`
- Caller may want to use same registers as callee
  - Similarly, values are destroyed
  - Usually temporary registers (`$t0` to `$t9`)

Register Usage Conventions

- $t0 – $t9:
  - These registers are temporary registers
  - Caller **should not** expect that values stored in these register will not be destroyed by callee
  - If caller wants to use them, caller must save their values before the call and restore their values back after the call
- $s0 – $s7:
  - These registers are temporary saved registers
  - Callee **must** maintain their values
  - If callee wants to use them, callee must save their values before using them and restore their values before return to caller

Where to Save Values of Registers?

- Simply save in other registers may not be a good idea
  - We have limited number of registers
  - We may run out of registers
- Need memory space to hold saved ("spilled") registers
  - Caller spills `$t0` – `$t9` that must be saved to memory
  - Callee spills `$s0` – `$s7` that are used to memory
  - Other registers (`$a0` – `$a3`, `$v0`, and `$v1`) may also need to be saved
  - Non-leaf caller saves `$ra` to memory before they make another call
- Each procedure needs memory locations to save registers
- We cannot simply allocate a fixed size and fixed memory location of each procedure
  - Hard to keep track
  - Required memory size for recursive procedure cannot be determined in advance
  - Call-chain depth (the number of called procedures) is generally unknown. Need to support undetermined length

Program Stack

- **Program Stack**: Memory locations used by running program
- Generally used for:
  - saving values of spilled registers
  - storing local variables, if we do not have enough registers
  - arguments, if we have more than four arguments
  - return values, if we have more than two return values
  - storing return address (`$ra`)
- Each procedure allocates space called **activation frame**
  - The size of activation frame is fixed for each procedure being called
  - The size can be determined by its purpose
  - The location of the activation frame is unknown until procedure call made

Calling Convention

- Caller saves needed registers, sets up arguments, makes call
  - Saves temporary registers (`$t0` to `$t9`) as needed
  - Sets up arguments (`$a0` to `$a3`) as needed
  - If more arguments are required, put arguments onto the stack
- Callee procedure when entering the procedure
  - Adjust the stack pointer according to activation frame size to hold temporary saved registers, locals, return address (non-left)
  - Save return address to the stack
  - Save any temporary saved registers to the stack
- Callee procedure body
  - Access stack items as needed (additional arguments, locals, etc)
- Callee procedure before leaving the procedure
  - Restore any temporary saved registers
  - Restore return address
  - Restore the stack (deactivate activation frame)
  - Return to caller

Example: Leaf Procedure

- Consider a leaf procedure that needs to use all temporary registers `$t0` to `$t9` and three temporary saved registers `$s0` to `$s2`
- According to the convention, this procedure only needs to save values stored in registers `$s0`, `$s1`, and `$s2` (3 words or 12 bytes)
- When entering the procedure, it must perform the following:

```
addi $sp, $sp, -12      # Allocate activation frame
sw   $s0, 0($sp)        # Store $s0 in memory
sw   $s1, 4($sp)        # Store $s1 in memory
sw   $s2, 8($sp)        # Store $s2 in memory
```

- Before return back to the caller, it must perform the following:

```
lw   $s2, 8($sp)        # Restore $s2 from memory
lw   $s1, 4($sp)        # Restore $s1 from memory
lw   $s0, 0($sp)        # Restore $s0 from memory
addi $sp, $sp, 12       # Deallocate activation frame
```

Example: Non-Leaf Procedure

- Consider a non-left procedure that uses registers `$s0`, `$s1`, `$s2`, `$t0` and `$t1`
- This procedure needs to call another procedure located at the label `_aFunction`
- The size of the activation frame is 6 words including `$ra` (24 bytes)
- When entering the procedure, it must perform the following:

```
addi $sp, $sp, -24      # Allocate activation frame
sw   $ra, 0($sp)        # Store return address in memory
sw   $s0, 4($sp)        # Store $s0 in memory
sw   $s1, 8($sp)        # Store $s1 in memory
sw   $s2, 12($sp)       # Store $s2 in memory
```

Example: Non-Leaf Procedure

- Before and after calling `_aFunction`, it must perform the following:

```
sw   $t0, 16($sp)       # Store $t0 in memory
sw   $t1, 20($sp)       # Store $t1 in memory
jal  _aFunction         # Call function _aFunction
lw   $t1, 20($sp)       # Restore $t1 from memory
lw   $t0, 16($sp)       # Restore $t0 from memory
```

- Before return back to the caller, it must perform the following

```
lw   $s2, 12($sp)       # Restore $s2 from memory
lw   $s1, 8($sp)        # Restore $s1 from memory
lw   $s0, 4($sp)        # Restore $s0 from memory
lw   $ra, 0($sp)        # Restore $ra from memory
addi $sp, $sp, 24       # Deallocate activation frame
jr   $ra                # Return to caller
```

Example: Recursive Multiplication

- Note that the result of $a * b$ is the same as $a + (a * (b - 1))$ which is the same as $a + a + (a * (b - 2))$ and so on.
- A recursive version of a multiplication function can be as follows:

```
int multi(int a, int b)
{
   if(b == 1)
   {
      return a;
   }
   else
   {
      return a + multi(a, b - 1);
   }
}
```

Solution

```
_multi: addi $sp, $sp, -8       # Adjust the stack
        sw   $ra, 0($sp)        # Store $ra in memory
        sw   $s0, 4($sp)        # Store $s0 in memory
        addi $s0, $zero, 1      # Set $s0 to 1
        bne  $a1, $s0, else     # If $a1 != 1 to to else
        add  $v0, $zero, $a0    # Set return value to $a0 (multiplicand)
        j    exit               # Go to exit
else:   addi $a1, $a1, -1       # Reduce multiplier by 1
        jal  _multi             # Call the procedure _multi
        add  $v0, $v0, $a0      # Add return value to mutiplicand
exit:   lw   $s0, 4($sp)        # Restore $s0 from memory
        lw   $ra, 0($sp)        # Restore $ra from memory
        addi $sp, $sp, 8        # Adjust the stack
        jr   $ra                # Go back to caller
```