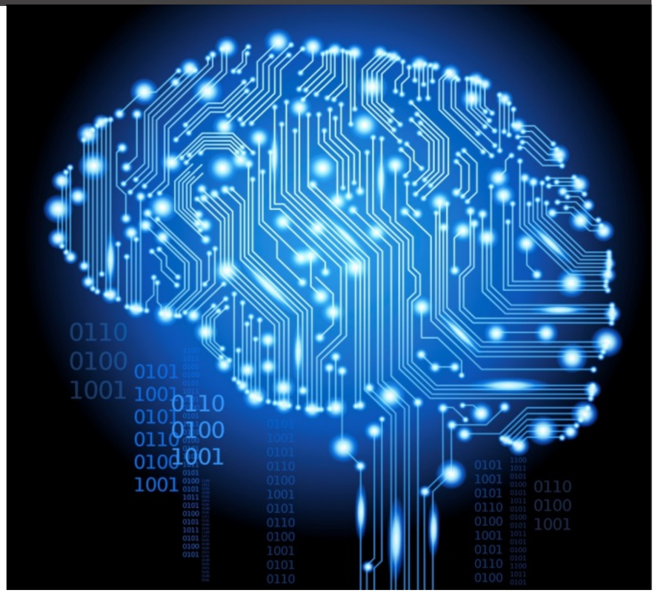
MONASH University

Information Technology

FIT1008/FIT2085 Lecture 3

MIPS Instructions

Prepared by: M. Garcia de la Banda
based on D. Albrecht, J. Garcia



Where are we at

- **We have seen the basics of the MIPS R2000 architecture**
  - CPU registers: 32 GPRs, PC, HI, LO, IR, MAR, MRR, MWR
  - The different memory segments
  - Accessing memory locations
  - Fetch-decode-execute cycle

Objectives

- To learn about the different MIPS R2000 instructions
- To become familiar with MIPS assembly programs
- To be able to write simple MIPS programs (read, write, maths)

Remember: Machine Language

- **The code that runs in the CPU is written in machine language**
  - Not in Java, C, Python, JavaScript, or assembler
- **Machine language programs are stored in memory as bit patterns**
  - In the R2000 the pattern of 32 bits is loaded from memory into IR
- **Each CPU type usually requires a different machine language**
- **Example: the machine language bit patterns to compute the factorial of a number for a “MIPS” R2000 CPU are:**

```
00110100000000100000000000000001
00101000100000010000000000000010
00010100001000000000000000000100
01110000010001000001000000000010
00100000100001001111111111111111
00001000000100000000000000001010
00000011111000000000000000001000
```

Each line is a different machine language instruction

Why is Assembly Language needed?

- **Problem with machine language:**
  - Very difficult to write or even read for humans
- **Need a compromise, a language that:**
  - Supports comments, variable names, line labels, etc
  - Has human-readable versions of machine instructions
  - But is easily converted to machine language
    - Usually a 1-to-1 relationship between each language instruction and its equivalent machine instruction
    - Ideally, this conversion done by a computer program that won't make mistakes
- **Languages of this kind called assembly languages**
  - Recall main distinction: RISC and CISC
- **A program that turns assembly code into machine language is called an assembler (part of a traditional compiler)**

Assembly language

This is what (almost) the same program looks like in assembly language:

Let's look at some features of assembly language that help you read and write it:

```
# Computes factorial of number (n) in $t0
# and returns result ("Res") in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end
        mult    $v0, $t0
        mflo    $v0                 # let Res = Res * n
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

Assembly language

This is what (almost) the same program looks like in assembly language:

Let’s look at some features of assembly language that help you read and write it:

```
# Computes factorial of number (n) in $t0
# and returns result (“Res”) in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end
        mult    $v0, $t0
        mflo    $v0                 # let Res = Res * n
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

The instructions themselves

- One per line
- Human-readable instruction code, e.g. addi
- This one says “add the immediate value 1 to the contents of register zero, and store the result in register s0”
- Translated by the assembler into machine language bit patterns

Assembly language

This is what (almost) the same program looks like in assembly language:

Let's look at some features of assembly language that help you read and write it:

```
# Computes factorial of number (n) in $t0
# and returns result ("Res") in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end

        mult    $v0, $t0            # let Res = Res * n
        mflo    $v0
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

Comments: start with a hash sign (#) and go to end of line.
They are ignored by the assembler

Assembly language

This is what (almost) the same program looks like in assembly language:

Let's look at some features of assembly language that help you read and write it:

```
# Computes factorial of number (n) in $t0
# and returns result ("Res") in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end

        mult    $v0, $t0            # let Res = Res * n
        mflo    $v0
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

Labels identify lines of code so that you can refer to them by name.

They are translated by the assembler into addresses

Assembly language

This is what (almost) the same program looks like in assembly language:

Let's look at some features of assembly language that help you read and write it:

```
# Computes factorial of number (n) in $t0
# and returns result ("Res") in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end
        mult    $v0, $t0
        mflo    $v0                 # let Res = Res * n
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

GPRs: as we saw last lecture,

- begin with $
- can use name (e.g. $v0)
- can use number (e.g. $2)

Assembly language

This is what (almost) the same program looks like in assembly language:

Let’s look at some features of assembly language that help you read and write it:

```
# Computes factorial of number (n) in $t0
# and returns result (“Res”) in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end
        mult    $v0, $t0
        mflo    $v0                 # let Res = Res * n
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

Lines beginning with a dot are *assembler directives*.
They tell the assembler to do something.
This one tells it to put the code in the text segment.

Assembly language

This is what the same program looks like in assembly language:

Let’s look at some features of assembly language that help you read and write it:

Numbers by themselves are *immediate values*; this one is -1.
This instruction decrements $t0 by adding -1 to it.
Immediate values may be in decimal, hexadecimal, or octal.

```
# Computes factorial of number (n) in $t0
# and returns result (“Res”) in $v0

        .text
fact:   ori     $v0, $0, 1          # let Res = 1
        addi    $s0, $0, 1          # let s0 = 1
loop:   slt     $t1, $s0, $t0       # if n <= 1
        bne     $t1, $s0, end       # goto end
        mult    $v0, $t0
        mflo    $v0                 # let Res = Res * n
        addi    $t0, $t0, -1        # let n = n - 1
        j       loop                # goto loop

end:    jr      $ra                 # return
```

Assembler Directives

- **Always start with . (dot)**
- **Assembler directives don't assemble to machine language instructions**
  - Instead, they are interpreted by the assembler
  - Result in the assembler doing something
- **Do what? Depending on the directive:**
  - To allocate space/data
  - To switch modes, tell in which memory segment is working

Assembler Directives – Switch Mode

- **.data**
  - Tells the assembler it is working in the part of the program that will create things (variables) in the data segment
    - From now on, it will find assembler directives that allocate space
- **.text**
  - Tells the assembler it is working in the part of the program that will become machine instructions and reside in the text segment
    - From now on, it will find assembler instructions

Assembler Directives – Allocate Space

- **Allocates memory in the data segment**
  - Thus, they appear under the `.data` directive
- **.space N**
  - allocate N bytes, store nothing
- **.word w1 [, w2, w3, ...]**
  - allocate 4-byte word(s) and store **`wi`** value(s) in it(them)
- **.asciiz "string"**
  - allocates the string as a sequence of ASCII values (1 byte each), terminated by a zero byte (null character indicating end of string)

Labels and symbols

- **We want to use names (labels) for memory locations (addresses)**
- **They might appear under the `.data` and under `.text` directives**
- **They need to be translated into addresses before execution**
- **To do this, the assembler uses a *symbol table***
- **When it sees a label being defined:**
  - It puts the label name and the current address in the table
- **When it sees a label being used:**
  - It looks the name up in the table to find what address it refers to

MIPS Program – Directives Example

```
                .data                  # Sets current addr to 0x10000000

N:              .word 100, 72          # Allocate 4+4=8 bytes
                                       # Set contents to 100 and 72
aString:        .asciiz “Hello!”       # Allocate 7 bytes
                                       # Set contents to “Hello!” + null

                .text                  # Sets current addr to 0x00400000
                lw $t0, N              # Look up N in table, translate instruction
loop:           addi $t0, $t0, 10      # translate instruction
                j loop                 # Look up loop in table, translate instruction
```

Symbol table

N:	0x10000000
aString:	0x10000008
loop:	0x00400004

How a (simplified) assembler works

- **Check file for assembler directives and labels**
  - Handle those if found (build symbol table)
- **Go back to start of file**
- **For each line of assembly language do:**
  - Look up operation in table
    - If valid, set first six bits of instruction to opcode, else output error
  - For each register on the line,
    - Look its number up in table and set the appropriate five bits in the instruction
  - If there is a reference to a label:
    - Look its value up in the symbol table and treat it like an immediate
  - If there is an immediate value on the line
    - Copy it into the last sixteen bits of the instruction

Input/Output

- **Programs often need to communicate with users (I/O)**
- **The operating system manages all peripherals including the console**
- **MIPS programs do I/O by asking the OS using a special command called `syscall`**
- **To make a system call in MIPS you must:**
  1. Work out which service you want
  2. Put service's call code in register `$v0`
  3. Put argument (if any) in registers `$a0, $a1`
  4. Perform the `syscall` instruction
  5. Result (if any) will be returned in register `$v0`

System Services (cont'd)

Service	Call code	Argument	Result
Print integer	1	\$a0 (int to be printed)	n/a
Print string	4	\$a0 (addr of first char of string)	n/a
Read integer	5	n/a	\$v0 (integer)
Read string	8	\$a0 (addr to put string) \$a1 (number of bytes to read)	n/a
Allocate memory	9	\$a0 (number of bytes requested)	\$v0 (addr of allocated memory)
Exit program	10	n/a	n/a

MIPS – I/O Example

Service	Code	Arg	Res
Print integer	1	$a0	n/a
Print string	4	$a0	n/a
Read integer	5	n/a	$v0
Read string	8	$a0 $a1	n/a
Allocate memory	9	$a0	$v0
Exit program	10	n/a	n/a

```
# Program to convert inches to millimetres
# Integer approximation, multiply by 254/10

       # Data used by the program
       .data

       # Program starts from label main (SPIM insists on it)
       .text
main:  addi   $v0, $0, 5        # system call 5 (read integer)
       syscall                  # result is in $v0
       addi   $t1, $0, 254      # put mms per inch in $t1
       mult   $v0, $t1          # multiply
       mflo   $t0               # put result in $t0
       addi   $t2, $0, 10       # put 10 in $t2
       div    $t0, $t2          # divide $t0 by $t2 (integer division)
       mflo   $t3               # put quotient in $t3

       add    $a0, $0, $t3      # print quotient
       addi   $v0, $0, 1        # system call 1 (print integer)
       syscall                  # print
       addi   $v0, $0, 10       # system call 10 (exit)
       syscall                  # exit
```

MIPS Instructions (basic kinds)

- **Arithmetic:** add, addi, sub, mult, div
- **Data movement:** mfhi, mflo
- **Logical:** and, or, xor, nor, andi, ori, xori
- **Shift:** sll, sllv, sra, srav, srl, srlv
- **Load/store:** lw, sw
- **Comparison:** slt, slti
- **Control transfer:** beq, bne **(conditional)**, j, jr, jal **(unconditional)**
- **Sytem calls:** syscall

- **We will not see all today**

Arithmetic Instructions (integer)

- **addition (+)**
  - `add $t0, $t1, $t2  # $t0 = $t1 + $t2`
- **addi – immediate addition (+)**
  - `addi $t0, $t1, 5   # $t0 = $t1 + 5`
- **subtraction (–)**
  - `sub $t0, $t1, $t2  # $t0 = $t1 - $t2`
- **multiplication (*)**
  - `mult $t1, $t2  # LO=$t1*$t2, HI=overflow`
- **division (//)**
  - `div $t1, $t2  # LO=$t1//$t2, HI=remainder`

Data Movement Instructions

- move from HI
  - `mfhi $t0  # $t0 = HI`
- move from LO
  - `mflo $t0  # $t0 = LO`

MIPS Program – Arithmetic Example

```
n = int(input())
n = n*254//10
print(n)
```

```
# Program to convert inches to millimetres
# Integer approximation, multiply by 254/10

        # Data used by the program
        .data

        # Program starts from label main (SPIM insists on it)
        .text
main:   addi  $v0, $0, 5        # system call 5 (read integer n)
        syscall                 # result is in $v0
        addi  $t1, $0, 254      # put mms per inch in $t1 ($t1=254)
        mult  $v0, $t1          # multiply
        mflo  $t0               # put result in $t0 ($t0=n*254)
        addi  $t2, $0, 10       # put 10 in $t2 ($t2=10)
        div   $t0, $t2          # divide $t0 by $t2 (n*254//10)
        mflo  $t3               # put quotient in $t3 ($t3= n*254//10)

        add   $a0, $0, $t3      # print quotient
        addi  $v0, $0, 1        # system call 1 (print integer)
        syscall                 # print
        addi  $v0, $0, 10       # system call 10 (exit)
        syscall                 # exit
```

Load/Store Instructions

- **load (read) word from memory to GPR**
  - `lw $t0, address  # $t0 = cont(address)`
  - Loads the 4 bytes beginning at *address* into `$t0`
- **store (write) word from GPR to memory**
  - `sw $t0, address  # cont(address) = $t0`
  - Stores the content of $t0 into the 4 bytes beginning at *address*

- **Question is, how do we specify an address?**
  - The opcode (**`lw`** or **`sw`**) takes up 6 bits of the IR
  - The destination register (**`$t0`**) takes up another 5
  - This leaves us with 21 bits to indicate the address

Five ways to specify an address

- **Directly (or using a label), e.g.**

```
lw $t1, N  # loads from label N
```

- **Label plus offset, e.g.**

```
lw $t1, N+4      # loads from (label N + 4)
```

- **Using a GPR to store the address, e.g.**

```
lw $t1, ($s0)    # loads from address stored in $s0
```

- **GPR + offset, e.g.**

```
lw $t1, 4($s0)   # loads from (address stored in $s0)+4
```

- **Label, offset, and GPR, e.g.**

```
lw $t1, N+4($s0)# loads from (label N+4)+contents of $s0
```

They are called Addressing Formats

- **Summary of addressing formats in table format:**

Format	Calculation
Imm	**immediate, i.e., actual value**
label	**address of label**
label+[–]Imm	**address of label+[–] Imm**
(register)	**contents of register**
[–]Imm(register)	**contents of register+[–] Imm**
label+[–]Imm(register)	**address of label+ contents of register +[–] Imm**

Load/Store Instructions – Example

```
      .data
var:  .word 3,11
      .text
main: lw    $t0,var
      lw    $t1,var+4
      add   $t0,$t1,$t0
      addi  $t2,$0,12
      sw    $t0,var+-4($t2)
```

label

label+Imm

label+–Imm(register)

$t0 ← 3

$t1 ← 11

$t0 ← 14

$t2 ← 12

(var+12)-4 ← 14

Sneaky assembler tricks

- **Suppose the address of label N is 0x7FFFFFF8**
- **That does not fit in 32 bits with the opcode and register**
  - How does the assembler manage to translate loads/stores from addresses larger than 0xFFFF?
- **Using *pseudoinstructions*: it translates `lw $t0, N` into *two* lines of machine code!**
  - One sets the top 16 bits: `lui $t0, 0x7FFF`
    - "load upper immediate": loads 16-bit value into top 16 bits of register
  - One sets the bottom 16 bits: `ori $t0, 0xFFF8`
    - could use an addi for this

More pseudoinstructions

- `li   $r1, n        # load immediate`
  - puts immediate value into *$r1*
  - translates that line into a `lui` and an `ori` `(or addi)`
- `la $r1, label      # load address`
  - puts address of *label* into *$r1*
  - assembler knows this address, so translates it like `li`

There are many pseudoinstructions in MIPS but you cannot use them for FIT2085. The point is for you to learn to use the basic blocks.

The only one you can use is **`la`**, which is very useful for loading the address of any string we want to print.

Have a look at the MIPS reference sheet to see the instructions you are allowed to use<

Bitwise Logical Instructions

- **Bitwise AND (&)**
  ```
  – and $t0, $t1, $t2          # $t0 = $t1 & $t2
  – andi $t0, $t1, 0xa0b1      # $t0 = $t1 & 0xa0b1
  ```
- **Bitwise OR (|)**
  ```
  – or $t0, $t1, $t2           # $t0 = $t1 | $t2
  – ori $t0, $t1, 5            # $t0 = $t1 | 0x0005
  ```
- **Bitwise exclusive OR (XOR) (^)**
  ```
  – xor $t0, $t1, $t2          # $t0 = $t1 ^ $t2
  – xori $t0, $t1, 5           # $t0 = $t1 ^ 0x0005
  ```
- **Bitwise not-OR (NOR)**
  ```
  – nor $t0, $t1, $t2          # $t0 = ~($t1 | $t2)
  ```

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Using bits to represent other things

Let's think of sequences of bits as sets:

	A	B	C	D	E	F		
Set 1	1	0	1	1	0	0	{A,C,D}	
Set 2	1	1	0	1	0	1	{A,B,D,F}	
AND	1	0	0	1	0	0	{A,D}	Intersection
OR	1	1	1	1	0	1	{A,B,C,D,F}	Union
XOR	0	1	1	0	0	1	{B,C,F}	Difference

- Very fast set operations!

Shift Instructions

Only the lower 5 bits of the $t2 register are used

Immediate value ≤ 31

Same as multiplying by 2^5

- **shift left (logical) (<<)**
  - fill with zero bits
  - `sll $t0, $t1, 5      # $t0=$t1 << 5`
  - `sllv $t0, $t1, $t2 # $t0=$t1 << content($t2)`
- **shift right (logical) (>>> in JS, not provided by Python)**
  - fill with zero bits
  - `srl $t0, $t1, 5      # $t0=$t1 >> 5`
  - `srlv $t0, $t1, $t2 # $t0=$t1 >> content($t2)`
- **shift right (arithmetic) (>>)**
  - fill with copies of MSB
  - `sra $t0, $t1, 5      # $t0=$t1 >> 5`
  - `srav $t0, $t1, $t2 # $t0=$t1 >> content($t2)`

Same as dividing by 2^5

Consider `$t0` being 10111111111111111111111111111101. Then:

```
sll $t0, $t0, 5 #11111111111111111111111110100000
srl $t0, $t0, 5 #00000101111111111111111111111111
sra $t0, $t0, 5 #11111101111111111111111111111111
```

MIPS Program – Shift and Logic Example

```
        .text
main: ori  $t0, $0, 0xa0b1
      ori  $t1, $0, 5
      or   $t2, $t1, $t0
      and  $t3, $t1, $t0
      sll  $t4, $t1, 4
      addi $t5, $0,  -6
      srlv $t6, $t5, $t1
      sra  $t7, $t5, 4
```

```
$t0 = 0000a0b1
$t1 = 00000005
$t2 = 0000a0b5
$t3 = 00000001
$t4 = 00000050
$t5 = fffffffa
$t6 = 07ffffff
$t7 = ffffffff
```

0001 = 1

0101 = 5

1111 = f

0111 = 7

Summary

- **Machine language and Assembly language**
- **Main components (labels, comments, instructions, directives, etc)**
- **MIPS R2000 instructions and assembly directives**
    - Instruction set
    - How MIPS instructions work
    - Assembly directives
    - I/O system calls
    - Addressing formats
    - Pseudoinstructions
    - Simple programs