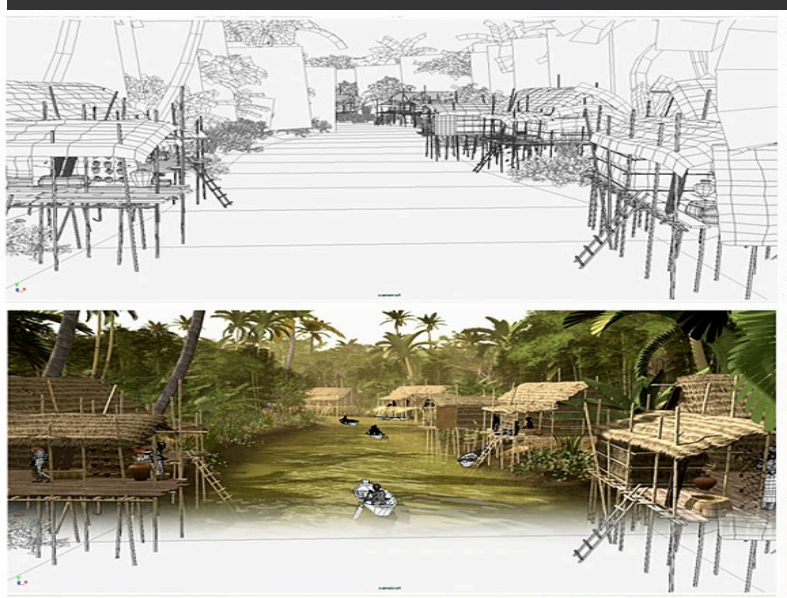
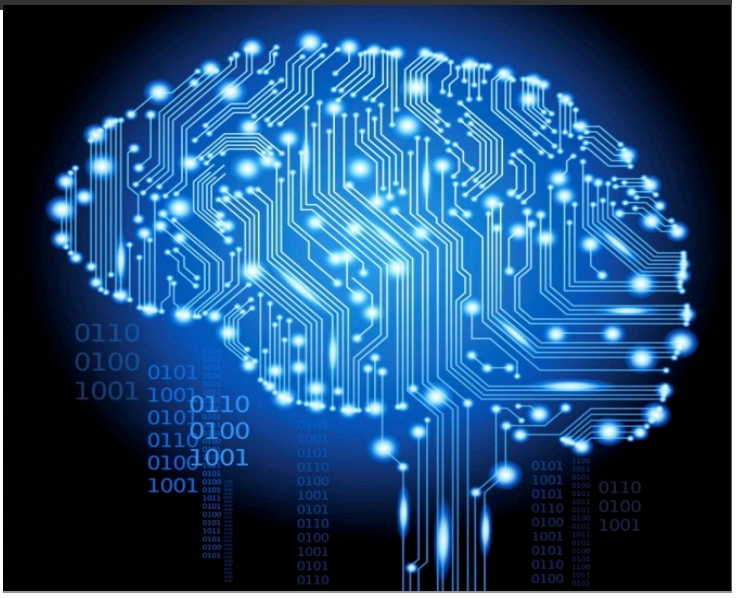
MONASH University

Information Technology

FIT1008/2085
MIPS – Intro to assembly programs

Prepared by:
Maria Garcia de la Banda
Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson





Where are we at

- **We now know why MIPS is useful**
- **We know the basics of MIPS R2000 architecture**
  - CPU: ALU, GPRs, Control, and the special purpose registers
  - Main memory, its segments and their purpose
- **We know how programs are executed in this architecture**
  - The fetch-decode-execute cycle

Learning objectives for this lesson block

- **To become familiar with MIPS assembly programs**
  - Main components (labels, comments, instructions, directives, etc)
- **To understand the role of MIPS assembly directives and to know how to use them in a program**

In the previous lesson we said…

- **The machine code to compute the factorial of a number for a "MIPS" R2000 CPU consists of the following bit patterns:**

```
00110100000000010000000000000001
00101000100000010000000000000010
00010100001000000000000000000100
01110000010001000001000000000010
00100000100001001111111111111111
00001000000100000000000000001010
00000011111000000000000000001000
```

Each line of 32 bits (each word) is a different machine code instruction

(Almost) Same program 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 ("Result") in $v0

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

end:    jr      $ra                 # return
```

(Almost) Same program 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 ("Result") in $v0

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

(Almost) Same program 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 ("Result") in $v0

        .text
fact:   ori     $v0, $0, 1          # let Result = 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                 # Result = Result * n
        addi    $t0, $t0, -1        # 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

(Almost) Same program 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 ("Result") in $v0

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

end:    jr      $ra             # return
```

Defines a label

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

They are translated by the assembler into addresses

Uses a label

(Almost) Same program 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 ("Result") in $v0

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

end:    jr      $ra                 # return
```

GPRs: as we saw last lecture,

- begin with $
- please use name (e.g. $v0)
- can (but don't) use number (e.g. $2) except for $0

(Almost) Same program in Assembly language

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

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

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

end:    jr      $ra                 # return
```

Lines beginning with a dot are *assembler directives.*
They do not become instructions.
Instead, they just tell the assembler to do something.
This one tells it to put the code in the text segment.

(Almost) Same program 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 ("Result") in $v0

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

end:    jr      $ra                 # return
```

Numbers by themselves are *immediate values*; this one is -1.

This instruction decrements `$t0` by adding -1 to it.

Immediate values may appear in decimal, hexadecimal, or octal.

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 (static part of the) data segment
    - From then 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 code instructions and reside in the text segment
    - From then on, it will find assembler instructions

(Almost) Same program 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 ("Result") in $v0

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

Assembler Directives – Allocate Space

- **Allocates memory in the (static part of the) data segment**
  - Thus, they appear under the `.data` directive
- **.space N**
  - Allocates N bytes, stores nothing
- **.word w1 [, w2, w3, ...]**
  - Allocates 4-byte word(s) and stores the `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)
  - So, in total: 1 byte per character in the string + 1 zero byte

Not very used in our unit

Assembler Directives – Example

```
            .data                   # start of data segment
var:        .word 3,11              # word at var address=3
                                    # at var+4 address = 11

prompt:     .asciiz "Hi!"           # starting at address
                                    # prompt, store 4 bytes
                                    # for chars H I ! and Zero byte

            .text                   # start of text segment

main:       lw $t0, var             # first instruction at
                                    # address main
```

Summary

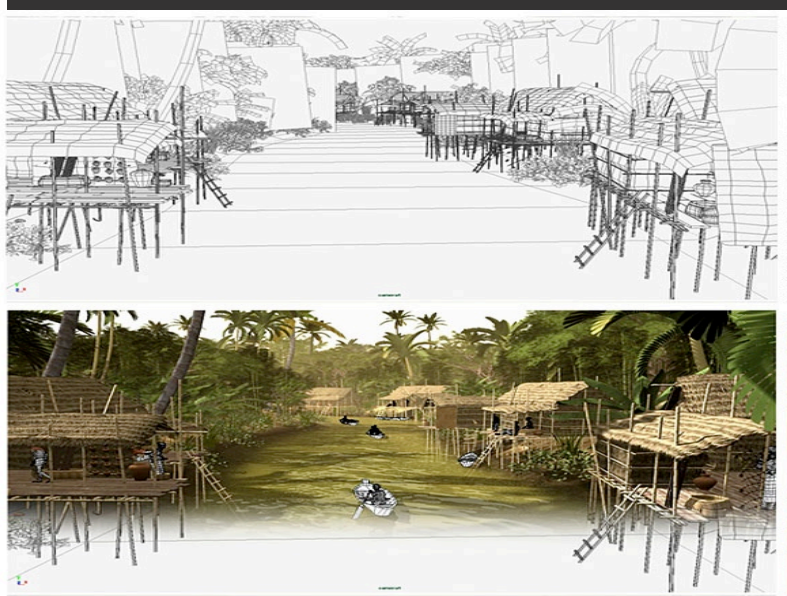
- **Became familiar with MIPS assembly programs**
  - Main components (labels, comments, instructions, directives, etc)
- **Understand the role of MIPS assembly directives:**
  - Switch mode
  - Allocate space
- **Know how to use these in a program:**
  - .data
  - .text
  - .asciiz
  - .word
  - .space N

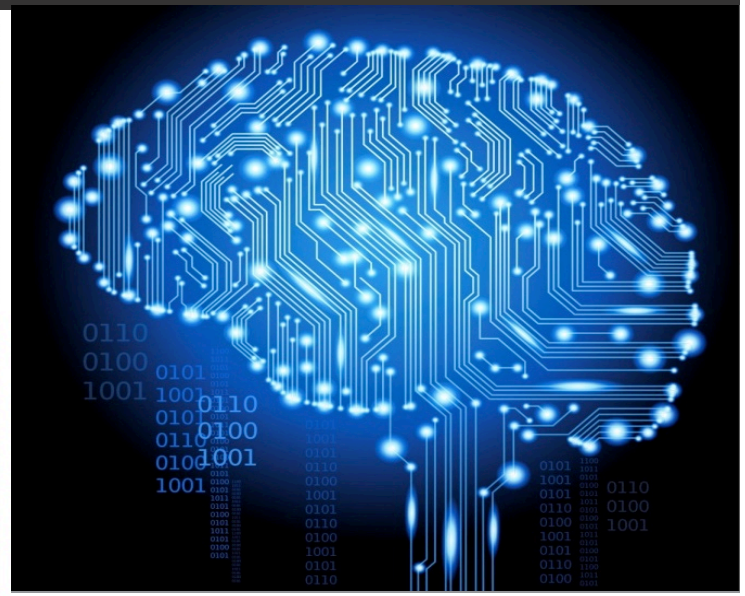
MONASH University

Information Technology

FIT1008/2085
MIPS – Assembly Instruction Set -- I

Prepared by:
Maria Garcia de la Banda
Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson





Where are we at

- **Know the basics of MIPS R2000 architecture**
  - CPU: ALU, GPRs, Control, and the special purpose registers
  - Main memory, its segments and their purpose
- **Understand the fetch-decode-execute cycle**
- **Know the main components of a MIPS program**
  - Labels, comments, instructions, directives, etc
- **Understand the role of assembly directives (switch mode, allocate space)**
- **Know how to use `.data` `.text` `.asciiz` `.word` `.space N` in a program**

Learning objectives for this lesson block

- To become familiar with MIPS instruction set
- To learn about the MIPS instructions we will use
- To be able to use some of them in a program

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 in this lesson**

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`

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

No need to memorise! It is in the MIPS sheet (Moodle Week 1)

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

        # Instructions in the program
        .text

main:   #read integer
        addi   $v0, $0, 5       # put a 5 in $v0
        syscall                 # result is in $v0

        # multiply it by 250 and // by 10
        addi   $t1, $0, 254     # put mms per inch in $t1 ($t1=254)
        mult   $v0, $t1         # multiply, result left in LO
        mflo   $t0              # put result in $t0 ($t0=n*254)
        addi   $t2, $0, 10      # put 10 in $t2
        div    $t0, $t2         # divide $t0 by $t2 (int division: n*254//10)
        mflo   $t3              # put quotient in $t3 ($t3= n*254//10)

        #print result & exit
        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  # $t0 = content(address)`
  - Loads the 4 bytes beginning at *address* into `$t0`
- **store (write) word from GPR to memory**
  - `sw $t0, address  # content(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
  - Addresses are 32 bits…

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

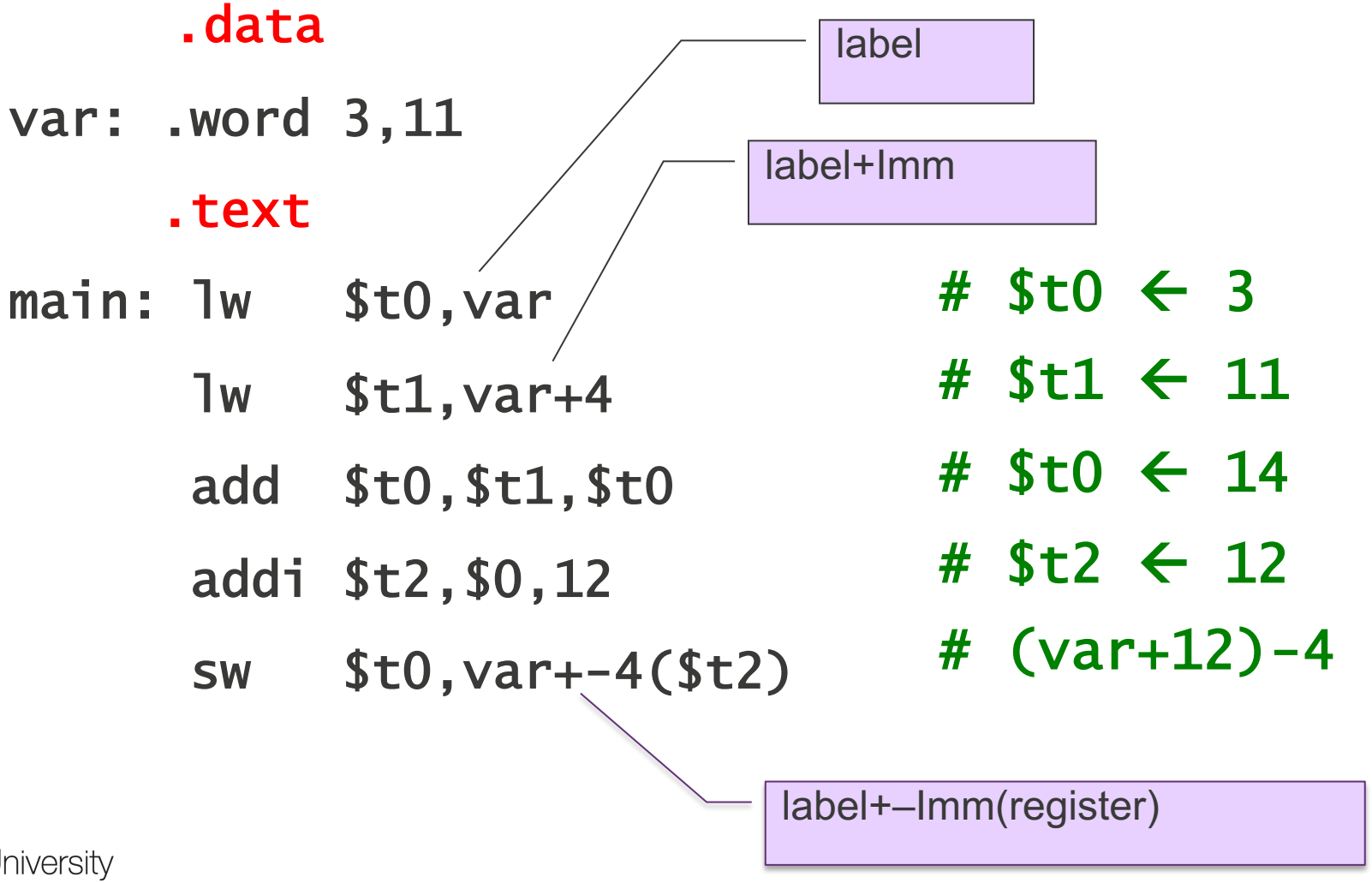
- **Summary of addressing formats in table format:**

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)
```

```
# $t0 ← 3
# $t1 ← 11
# $t0 ← 14
# $t2 ← 12
# (var+12)-4 ← 14
```



Summary

- **Are now familiar with MIPS instruction set**
- **Have learned about the MIPS instruction set we will use**
- **Are able to use some of them in a program:**
  - Arithmetic
  - Data movement
  - Input/output
  - Load/store
- **We know how to use the different addressing formats**

MONASH University

Information Technology

FIT1008/2085
MIPS – Assembly Instruction Set II

Prepared by:
Maria Garcia de la Banda
Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson

Where are we at

- **Know the basics of MIPS R2000 architecture**
- **Understand the fetch-decode-execute cycle**
- **Know the main components of a MIPS program**
- **Can use assembly directives in a program**
- **Know the MIPS instruction set and can use some of them in a program:**
  - Arithmetic
  - Data movement
  - Input/output
  - Load/store
- **Know how to use the different addressing formats**

Learning objectives for this lesson block

- **Continue to learn about other MIPS instructions (bitwise and shift)**

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

Bitwise Logic Example

```
        .text
main: ori  $t0, $0, 0xa0b1      # $t0 = 0000a0b1
      ori  $t1, $0, 5           # $t1 = 00000005
      or   $t2, $t1, $t0        # $t2 = 0000a0b5
      and  $t3, $t1, $t0        # $t3 = 00000001
```

0001 = 1

0101 = 5

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!

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? (i.e., those that do not fit in 16 bits)
- **Using *pseudoinstructions*: it translates `la $t0, N` into *two* lines of machine code! (16 + 16 = 32 bits)**
  - One sets the top 16 bits: **`lui $t0, 0x7FFF`**
    - "load upper immediate": loads 16-bit value into top 16 bits of register
  - The other sets the bottom 16 bits: **`ori $t0, $t0, 0xFFF8`**

You will not be asked to use **`lui`**, so don't worry about the details.

More pseudoinstructions

- `li  $r1, n` **`# load immediate using 32 bits`**
  - puts immediate value into *$r1*
  - translates that line into a `lui` and an `ori` (`or addi`)
- `la $r1, label` **`# load address (again, 32 bits)`**
  - puts address of *label* into *$r1*
  - assembler knows this address, so translates it like `li`

There are many pseudoinstructions in MIPS but you cannot use them for FIT1008/2085. 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 (the one with the blue title!)

Shift Instructions

- **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 (most significant bit – indicates the sign)
  - `sra $t0, $t1, 5     # $t0=$t1 >> 5`
  - `srav $t0, $t1, $t2 # $t0=$t1 >> content($t2)`

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

Immediate value ≤ 31

Same as multiplying by 2^5

Same as dividing by 2^5

Consider `$t0` = 10111111111111111111111111111101. Then `$t2`=:

```
sll $t2, $t0, 5 #11111111111111111111111110100000
srl $t2, $t0, 5 #00000101111111111111111111111111
sra $t2, $t0, 5 #11111101111111111111111111111111
```

Bitwise Shift and Logic Example

```
        .text
main: ori  $t0, $0, 0xa0b1      # $t0 = 0000a0b1
      ori  $t1, $0, 5           # $t1 = 00000005
      or   $t2, $t1, $t0        # $t2 = 0000a0b5
      and  $t3, $t1, $t0        # $t3 = 00000001
      sll  $t4, $t1, 4          # $t4 = 00000050
      addi $t5, $0,  -6         # $t5 = fffffffa
      srlv $t6, $t5, $t1        # $t6 = 07ffffff
      sra  $t7, $t5, 4          # $t7 = ffffffff
```

0001 = 1

0101 = 5

1111 = f

0111 = 7

Summary

- **Are able to use mode MIPS instructions in a program:**
  - Bitwise: `and or xor nor`
  - Shift: `sll slv srl srlv sra srv`
- **We have learned about pseudoinstructions and know we will only use one:**
  - `la` for printing strings

The following 3 slides are not examinable but might be interesting for some

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 performs two passes on every program:**
- **The first builds a *symbol table,* i.e., when it sees a label being defined:**
  - It puts the label name and the current address in the table
- **The second pass uses the table, i.e. 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           # Updates N in table and allocates 4+4=8 bytes
                                        # Set contents to 100 and 72
aString:        .asciiz “Hello!”        # Updates aString and allocates 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       # Update loop in table,  translate instruction
                                        # Look up loop in table, translate instruction
               j loop
```

Symbol table (at the end of the first pass)

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

Dark green happens in the first pass of the assembler, light green in the second pass

How a (simplified) assembler works

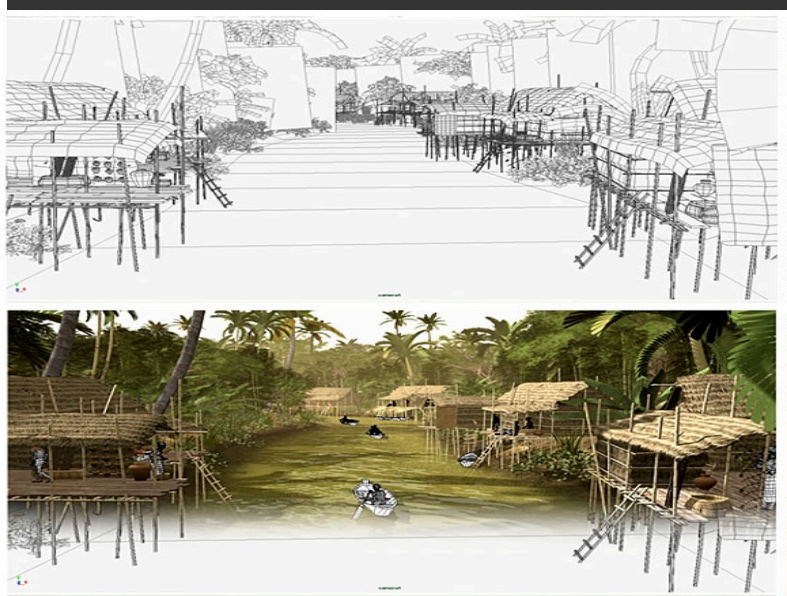
- **First pass: check file for assembler directives and labels**
  - Handle those if found by building a symbol table (see example in next slide)
- **Second: go back to start of file; for each line of assembly language do:**
  - Look up its operation code:
    - If valid, set first six bits of instruction to opcode, else output error
  - For each register in 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

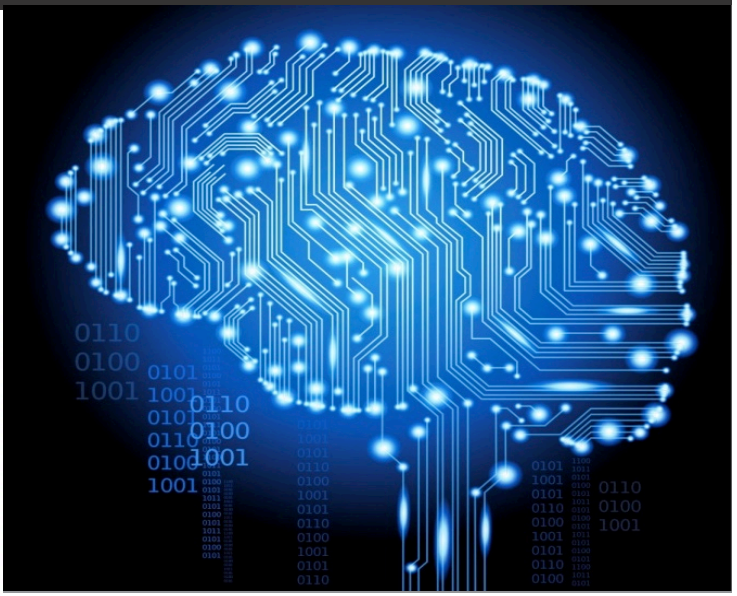
MONASH University

Information Technology

FIT1008/2085
MIPS – Instruction Set III and Formats

Prepared by:
Maria Garcia de la Banda
Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson





Where are we at

- **Know the basics of MIPS R2000 architecture**
- **Understand the fetch-decode-execute cycle**
- **Know the main components of a MIPS program**
- **Can use assembly directives in a program**
- **Know the MIPS instruction set and can use some in a program:**
  - Arithmetic
  - Data movement
  - Input/output
  - Load/store
  - Bitwise
  - Shift
- **Know how to use addressing formats and the pseudoinstruction `la`**

Learning objectives for this lesson block

- To learn how to use unconditional control transfer instructions
- To learn the three different instruction formats

Blast from the past: the `goto` statement

- Remember: a label is an identifier for a memory position (data/code)
- The `goto` statement performs an unconditional jump to its label argument
- It promotes code whose control flow is extremely difficult to understand
- That is why it is not supported by most languages, including Python

If Python had a `goto` statement …

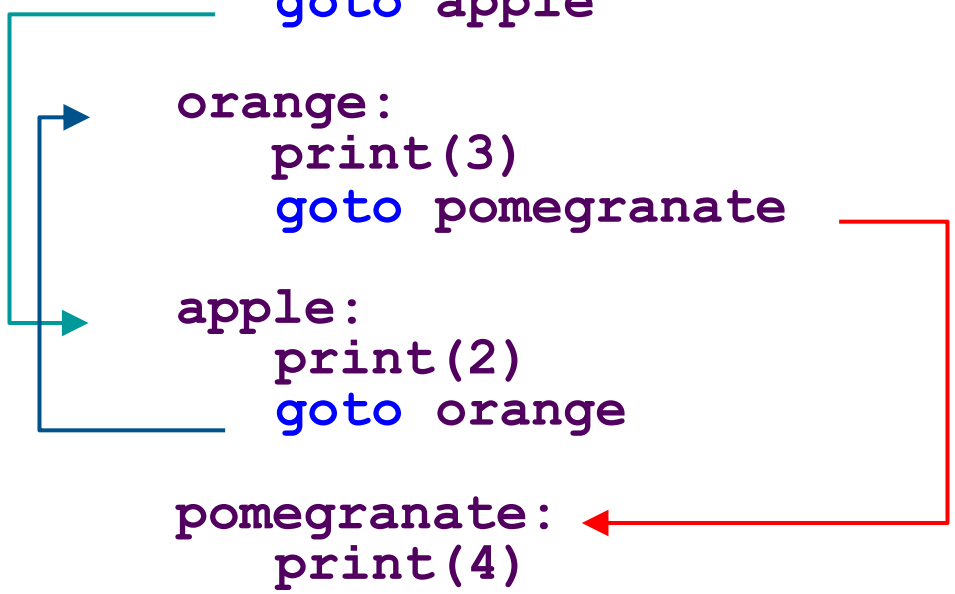
```
# Code could be this ugly!
def main():

    print(1)
    goto apple

orange:
    print(3)
    goto pomegranate

apple:
    print(2)
    goto orange

pomegranate:
    print(4)
```



A very unclear way to print 1 2 3 4!!

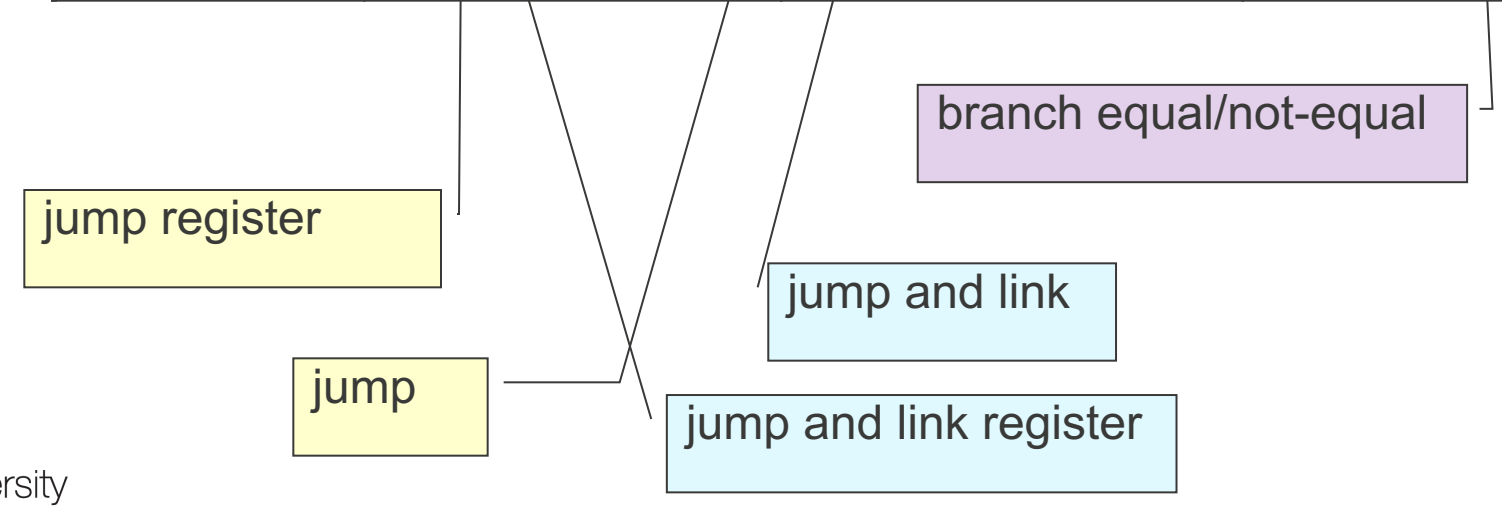
https://crmbusiness.wordpress.com/2015/01/29/why-you-should-write-code-and-customizations-like-a-boy-scout/

MIPS control transfer instructions

- **MIPS does have several kinds of goto instructions:**
  - They are called control transfer instructions
- **They are needed to change the control flow**
- **They replace the value in the Program Counter (PC) with the address value of a specified "destination"**
  - PC = destination address
- **Control is transferred to this new destination point at the next "fetch instruction" phase**
  - The next instruction loaded into IR is the one at the destination
- **They can be conditional and unconditional**
- **And use a register for the address or an immediate (label)**

Control Transfer Instructions

	UNCONDITIONAL		CONDITIONAL
	Register	Immediate	
JUMP	`jr`	`j`	
CALL	`jalr`	`jal`	
BRANCH			`beq/bne`



Jump and call instructions

- jump (go) to label

```
j foo          # set PC = foo
               # so, go to foo
```

- jump to address contained in register

```
jr $t0         # set PC=$t0, so go to the
               # address contained in $t0
```

- jump to label and link (remember origin)

```
jal foo        # $ra = PC+4; PC = foo, same as j foo
               # but setting a return address
```

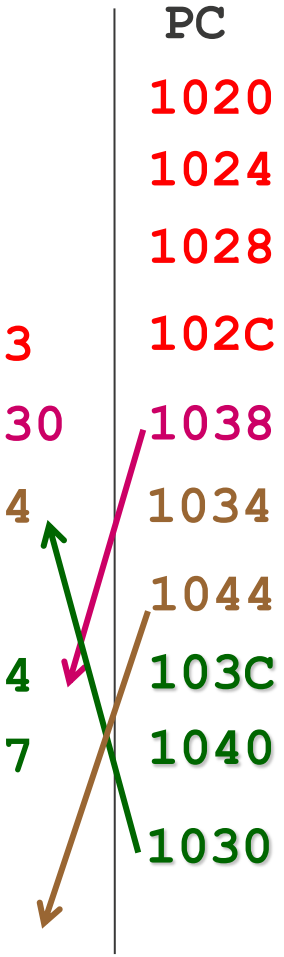
- jump to register and link (remember origin)

```
jalr $t0       # $ra = PC+4; PC = $t0, same as
               # jr $t0 but setting return address
```

Useful to call a function (return)

Jump instructions – Example

```
                .text
1020   main:  addi $t0,$0,5        # $t0 ← 5
1024          addi $t1,$0,9        # $t1 ← 9
1028          addi $t2,$0,-3       # $t2 ← -3
102C          jal  subr            # $ra ←1030
1030          add  $t3,$t3,$t3     # $t3 ← 34
1034          j    end
1038   subr:  add  $t3,$t1,$t0     # $t3 ← 14
103C          sub  $t3,$t3,$t2     # $t3 ← 17
1040          jr   $ra
1044    end:   …
```



MIPS Instruction Format

- **Remember: every MIPS instruction occupies 4 bytes of memory (32 bits)**
- **Remember: each instruction contains**
  - opcode
    - operation code: specifies type of instruction
  - operands
    - values or location to perform operation on
      - registers
      - immediate (constant) numbers
      - labels (addresses of other lines of program)

MIPS Instruction Format

- **Three general assembler formats**

```
sub $t0, $t1, $t2
```

R (for "register") format instruction: three registers

subtract the contents of register $t2 from the contents of register $t1; put the result in register $t0

```
addi $v0, $a2, 742
```

I (for "immediate") format instruction: two registers and one immediate operand

add the immediate number 742 with the contents of register $a2; put the result in register $v0

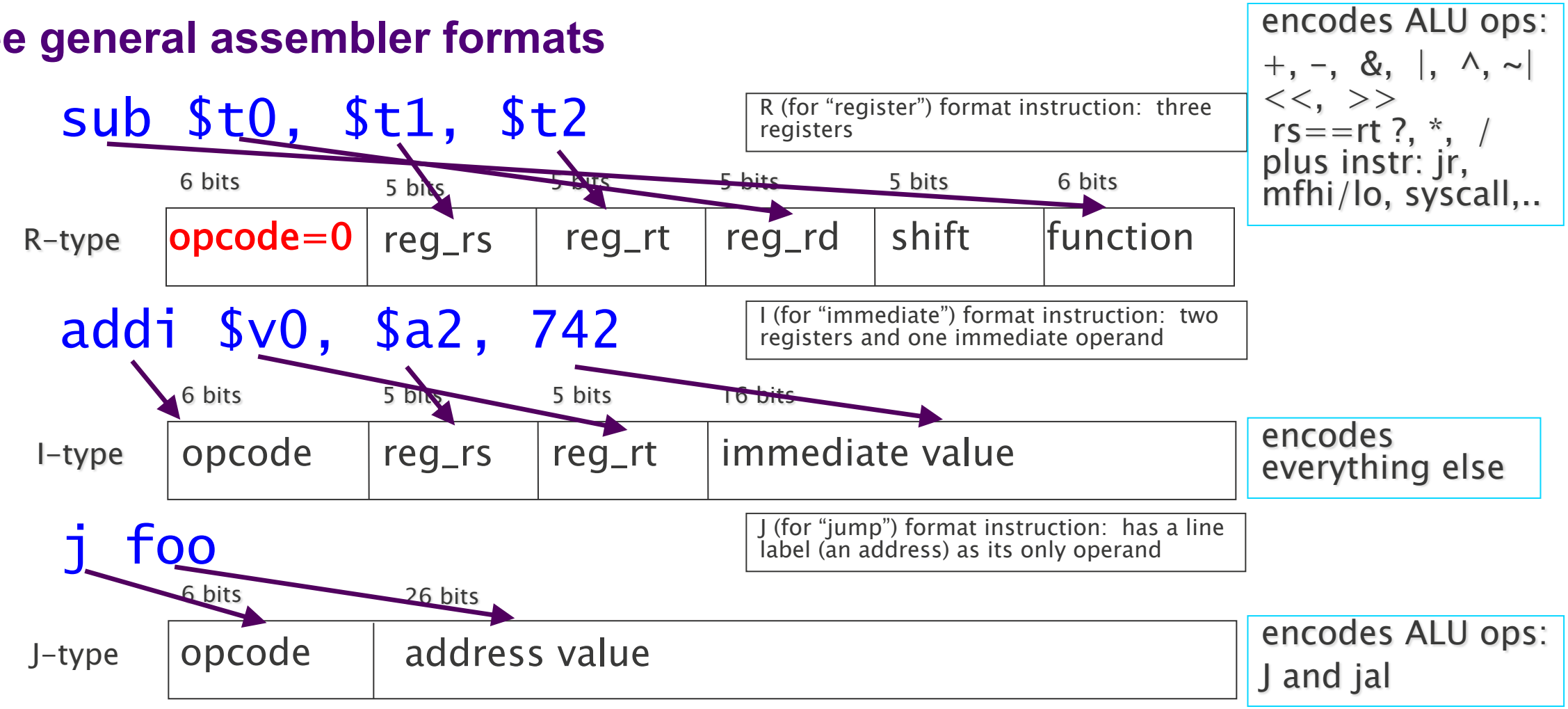
```
j foo
```

J (for "jump") format instruction: has a line label (an address) as its only operand

jump (go) to the line with the label foo and continue running from there

MIPS Instruction Format (cont'd)

- **Three general assembler formats**

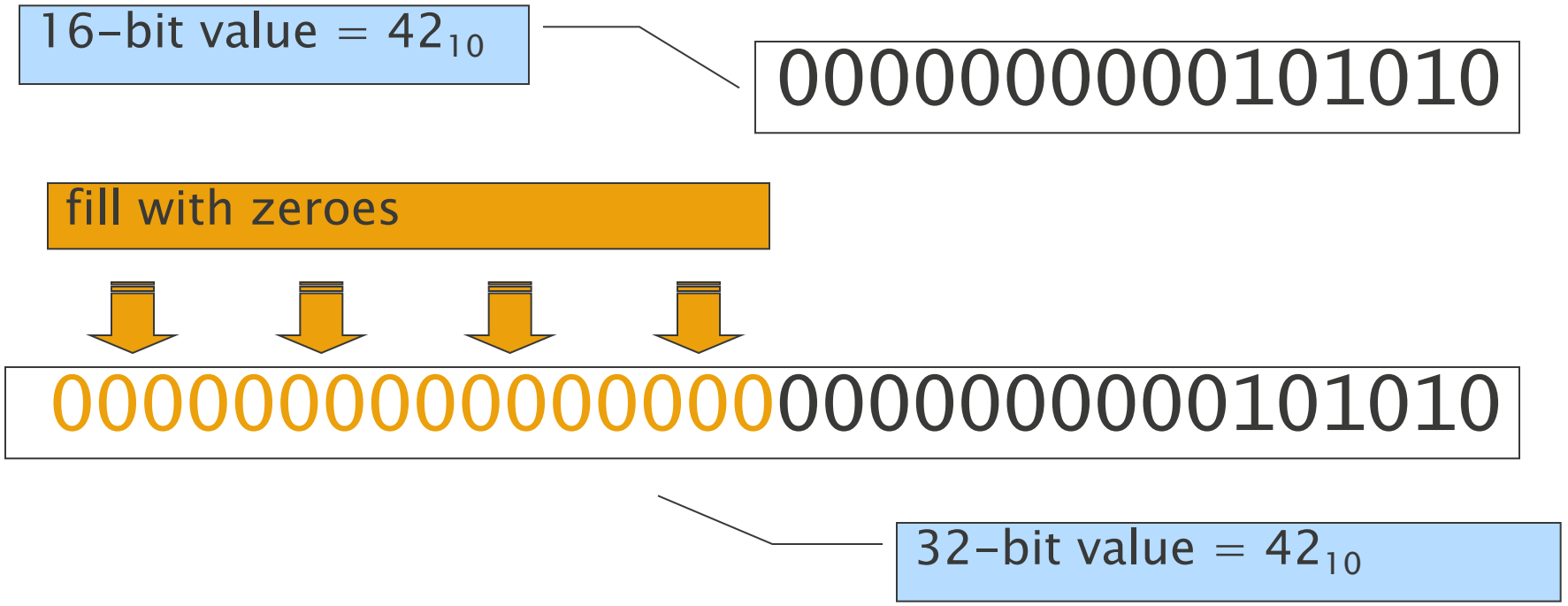


Dealing with immediate values

- **Consider the instruction:**
  - `addi $t0, $t1, 123`
- **Immediate value `123` is represented using 16 bits (I-type instruction)**
- **But it must be put in 32-bit format to go into the ALU**
- **So how do we make a 16 bit value into a 32-bit value?**
  - It depends on the meaning of the immediate
  - In turn, this depends on the actual instruction

Zero Extension

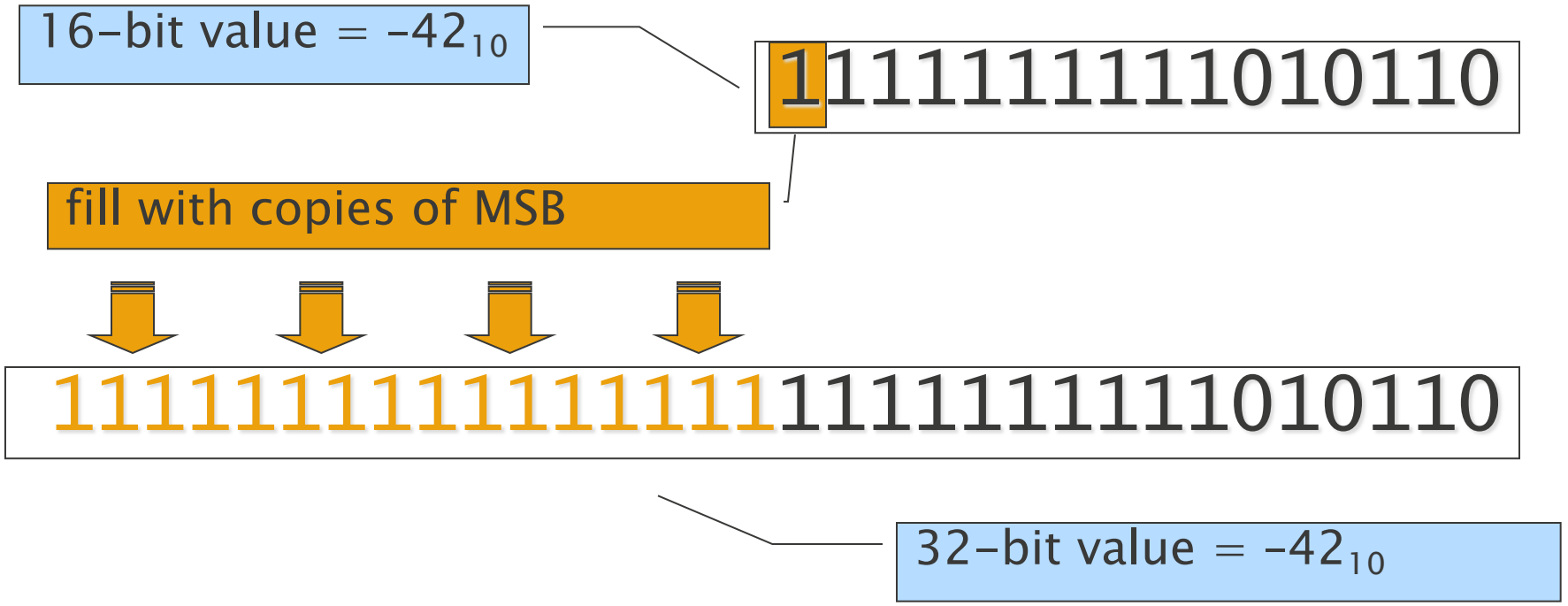
- **Used by** `andi,ori,xori`
- **Zeroes are added in front**



Sign Extension

This is one of the main differences between **`addi`** (sign extended) and **`ori`** (zero extended)

- **Used for `addi,sll,srl,sra,slti`**
- **Sign is extended by duplicating MSB (sign bit)**

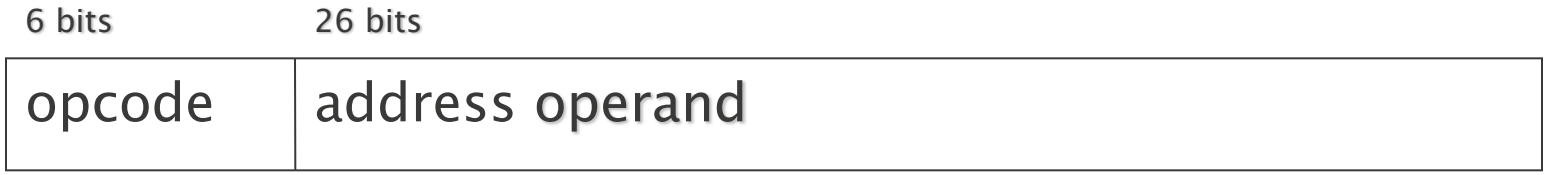


Summary

- **Know how to use unconditional control transfer instructions**
  - `j, jr, jal, jalr`
- **Know the three different instruction formats**

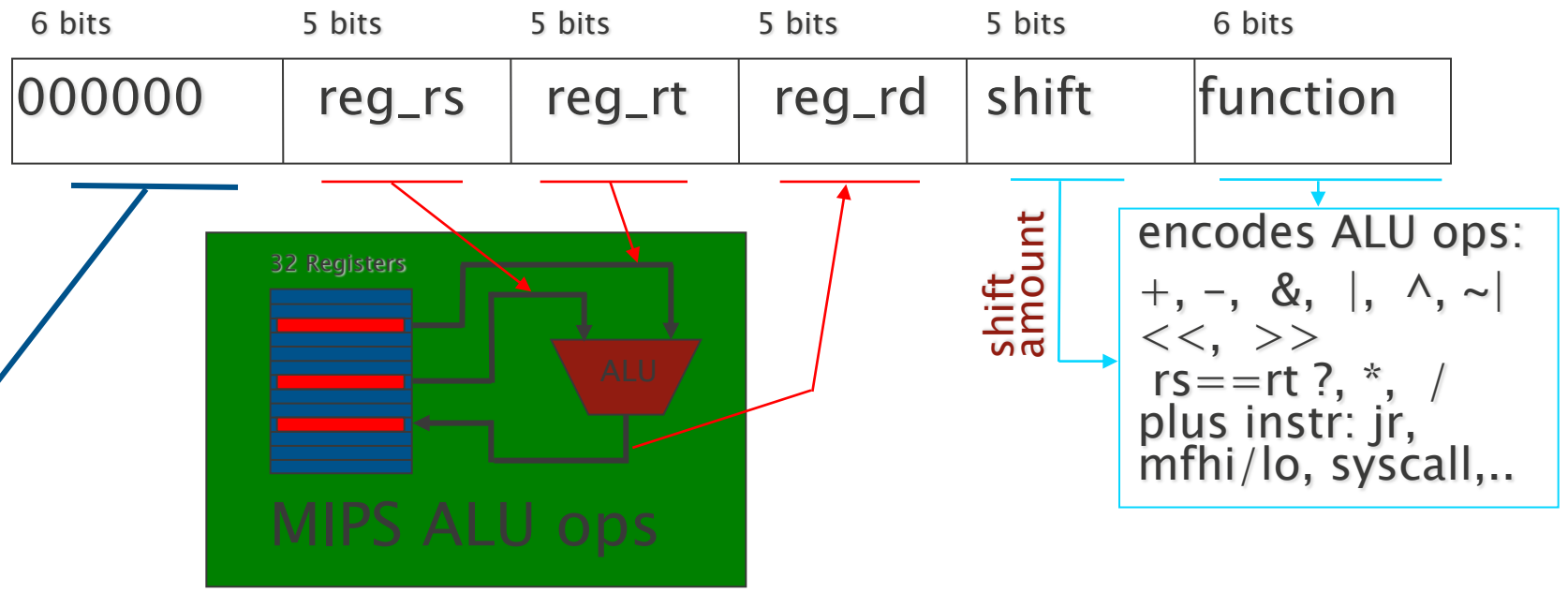
The following 4 slides are not examinable but might be interesting for some

"J-type" instructions



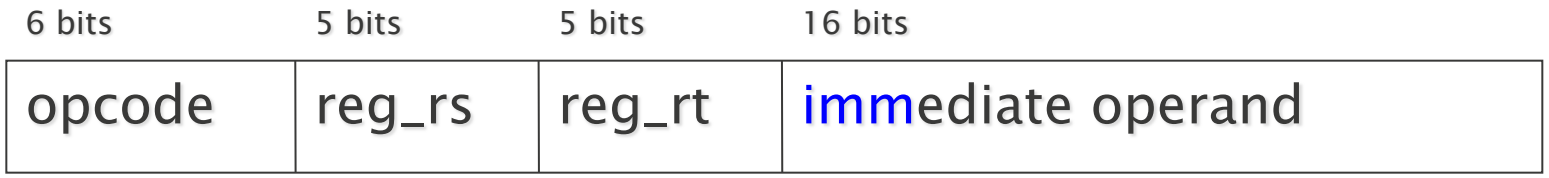
- **Only two MIPS instructions are j-type:**
  `j label` **and** `jal label2` **(opcodes 2 & 3, respectively)**
  - all label addresses in text segment are multiples of 4
  - assembler encodes 26 bit operand = (label addr >> 2)
  - this encoding improves maximum range of a jump
  - during instruction execution the operand is shifted << 2 and used to replace lowest 28 bits of the PC
  - `jal` is used exclusively for function calls
    - before PC is modified, PC+4 (=address of instruction after jal) is saved in register $ra (return address)

“R-type” Instructions



- **All R-type instructions have opcode = 0**
- **They use register operands exclusively**
- **6-bit function field specifies exact action**
  - 28 different instructions encoded by this field

“I-type” Instructions



- **All remaining instructions are I-type**
- **Immediate bit field used in three ways:**
  - In (real) load/store instructions
    - Actual address referenced is sum of reg_rs and imm field
  - In conditional branching instructions
    - imm is the “branch distance” measured in memory words from the address of the following instruction
  - In immediate ALU arithmetic or logic ops
    - Saves time where one operand is a small imm constant

I-type Instruction – Example

- **Instruction's components encoded in binary**