Consultations

Julian García

Thursday: 2 PM to 3 PM

Office 230, 25 Exhibition Walk, Clayton

Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 3
Assembly Programming

FIT 1008
Introduction to Computer Science

Where are we at

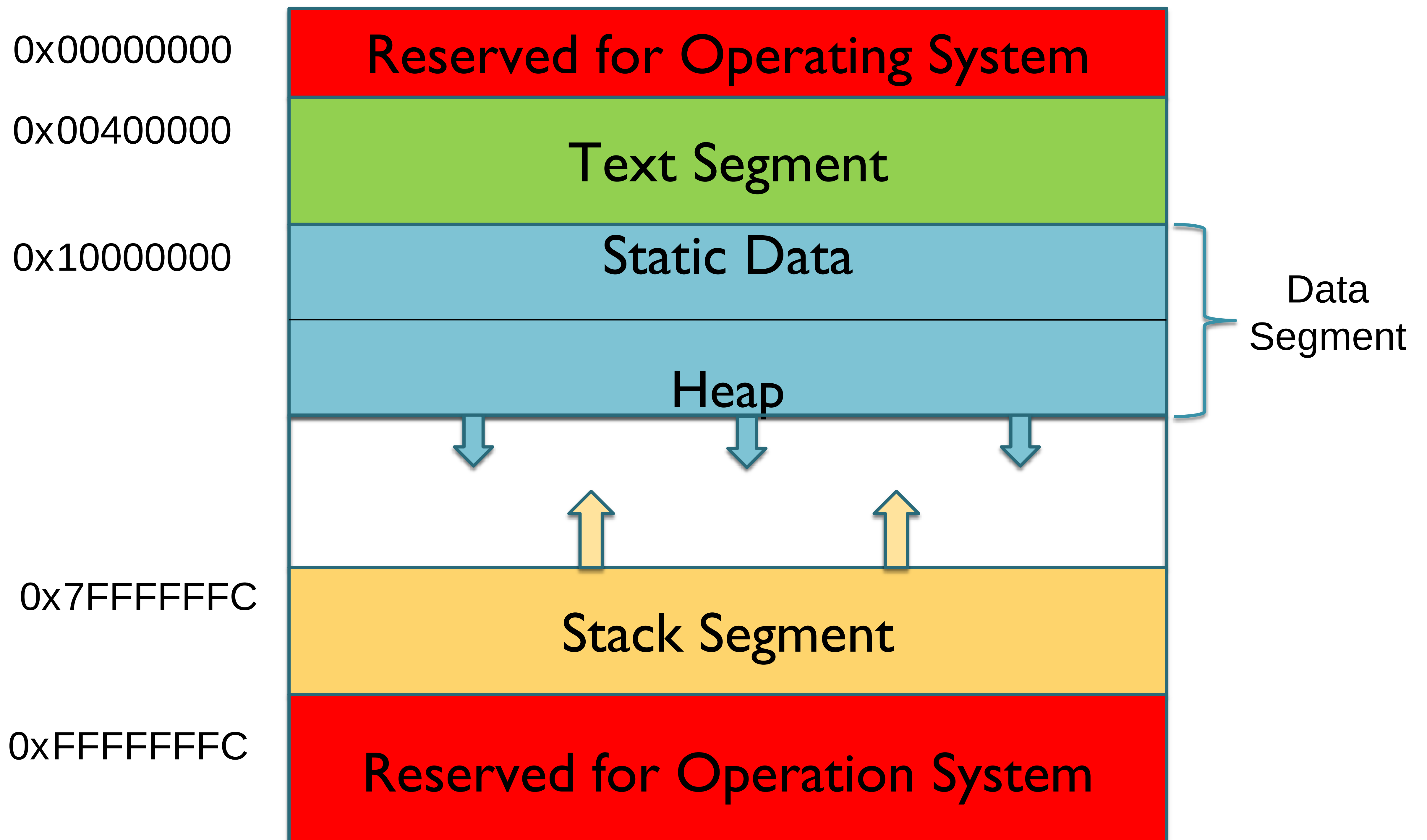
We know the basics of:

- MIPS R2000 architecture (32 GPR, Special registers)
- Memory segments
- Fetch-decode-execute cycle

Objectives

- To understand how to write simple MIPS programs
- To understand how to read and write integers and strings in MIPS.
- To understand how to use the data segment.
- To understand some assembly directives.

MIPS Architecture: Memory



hello_world.py

```
print("Hello World!")
```

Hello World! (MIPS)

label

assembly directives

```
        .data
str:    .asciiz "Hello World!"

        .text
        la      $a0, str          # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10       # exit
        syscall
```

MIPS instructions

System call

comments

General Purpose Registers

used in FIT1008	Register name	Register number	Typical use
	$zero	$0	constant zero, cannot change, read only
	$at	$1	assembler uses for pseudoinstructions
	$v0, $v1	$2, $3	function return values; system call number
	$a0 - $a3	$4 - $7	function and system call arguments
	$t0 - $t7, $t8, $t9	$8 - $15, $24, $25	temporary storage (caller-saved)
	$s0 - $s7	$16 - $23	temporary storage (callee-saved)
	$k0, $k1	$26, $27	reserved for kernel trap handler
	$gp	$28	pointer to global area
	$sp	$29	top-of-stack pointer
	$fp	$30	stack frame pointer
	$ra	$31	function return address

General Purp

```
        .data
str:    .asciiz "Hello World!"

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

Register name	Register number	
$zero	$0	constant zero, cannot change, read only
$v0, $v1	$2, $3	function return values; system call number
$a0 - $a3	$4 - $7	function and system call arguments
$t0 - $t7, $t8, $t9	$8 - $15, $24, $25	temporary storage (caller-saved)
$s0 - $s7	$16 - $23	temporary storage (callee-saved)
$sp	$29	top-of-stack pointer
$fp	$30	stack frame pointer
$ra	$31	function return address

Assembler Directives

```
        .data
str:    .asciiz "Hello World!”

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

- Always start with . (dot)
- Instruct the assembler to allocate space or data, or switch modes.
- Assembler dir
  language instr

Table 3: **Assembler directives**

.data	assemble into data segment
.text	assemble into text (code) segment
.byte b1[, b2, ...]	allocate byte(s), with initial value(s)
.half h1[, h2, ...]	allocate halfword(s), with initial value(s)
.word w1[, w2, ...]	allocate word(s) with initial value(s)
.space n	allocate n bytes of uninitialized, unaligned space
.align n	align the next item to a 2^n-byte boundary
.ascii "string"	allocate ASCII string, do not terminate
.asciiz "string"	allocate ASCII string, terminate with '\0'

Assembler Directives – Switch Mode

```
        .data
str:    .asciiz "Hello World!”

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

.data

- Tells the assembler it is working in the data segment
- Remember: this is where global variables are

.text

- Tells the asse
- Remember: t

Table 3: **Assembler directives**

.data	assemble into data segment
.text	assemble into text (code) segment
.byte b1[, b2, ...]	allocate byte(s), with initial value(s)
.half h1[, h2, ...]	allocate halfword(s), with initial value(s)
.word w1[, w2, ...]	allocate word(s) with initial value(s)
.space n	allocate n bytes of uninitialized, unaligned space
.align n	align the next item to a 2^n-byte boundary
.ascii ”string”	allocate ASCII string, do not terminate
.asciiz ”string”	allocate ASCII string, terminate with '\0'

```
        .data
str:    .asciiz "Hello World!”

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

Assembler Directives
Allocate Space

Allocates memory in the data segment

- **`.space N`**
  allocate N bytes, store nothing

```
        .data
str:    .asciiz "Hello World!"

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

- **`.word w1 [, w2, w3, …]:`**
  allocate 4-byte words and store values in them

- **`.asciiz "string"`**
  allocates the string as a sequence of ASCII values, terminated by a zero byte (null character indicating end of string)

Labels and symbols

- Labels: names for <u>locations in memory</u>
- But they need to be translated into numbers before execution
- The assembler uses a <u>symbol table</u> to help it do this
  - When it sees a label being defined:
    It puts the label name and the current address in the table
  - …abel being used:
    …p in the table to find what address it

```
        .data
str:    .asciiz "Hello World!"

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

How a (simplified) assembler works

1. Build symbol table: Check file for assembler directives and labels

2. Go back to start of file

3. 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

```
        .data
str:    .asciiz "Hello World!”

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

- I/O is complicated. Usually handled by the Operating System (OS)
- MIPS programs request I/O from the OS using a special command called `syscall`

System Services

To make a system call

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

```
        .data
str:    .asciiz "Hello World!”

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

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

Hello MIPS

```
        .data
str:    .asciiz "Hello World!”

        .text
        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall
        addi    $v0, $0, 10     # exit
        syscall
```

li $v0, 4
alternative

Pseudo-instructions

li Rdest, Imm ⋆	Load immediate	Rdest=Imm	-	-

hello.py

```
name = input('Enter name (max 60 chars):  ')

print('Hello ' + name)
```

```
        .data
prompt: .asciiz "Enter name (max 60 chars):  "
str:    .asciiz "Hello "
name:   .space 60

        .text
        la      $a0, prompt     # print prompt
        addi    $v0, $0, 4
        syscall

        la      $a0, name       # read name
        addi    $a1, $0, 60
        addi    $v0, $0, 8
        syscall

        la      $a0, str        # print str
        addi    $v0, $0, 4
        syscall

        la      $a0, name       # print name
        addi    $v0, $0, 4
        syscall

        addi    $v0, $0, 10     # exit
        syscall
```

convert_temp.py

<u>Suggestion</u>: write first
in simple python

```
temp_C = int(input('Enter temperature in Celsius '))

temp_F = int(9*temp_C/5 + 32)

print('Temperature in Fahrenheit is ' + str(temp_F))
```

1. Setup up string constants and global variables
2. Read temp_C
3. Compute temp_F
4. Print temp_C

Setup constant strings and global variables

recall: these labels
hold addresses

```
        .data
prompt: .asciiz "Enter temperature in Celsius:"
str:    .asciiz  "Temperature in Fahrenheit is"
temp_C: .word   0
temp_F: .word   0
```

temp_C and temp_F initialised to 0

Read temp_C

```
.text
la      $a0, prompt     # print prompt
addi    $v0, $0, 4
syscall
addi    $v0, $0, 5      # read int
syscall
sw      $v0, temp_C
```

sw: store what is in $vo, in address temp_C

sb Rsrc2, Addr (or label ⋆)	Store byte	mem8[Addr] = Rsrc2	-	-
sh Rsrc2, Addr (or label ⋆)	Store halfword	mem16[Addr] = Rsrc2	-	-
sw Rsrc2, Addr (or label ⋆)	Store word	mem32[Addr] = Rsrc2	-	-

Arithmetic Instructions

- **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`

Compute temp_F

```
lw      $t0, temp_C
addi    $t1, $0, 9
mult    $t0, $t1
mflo    $t0
addi    $t1, $0, 5
div     $t0, $t1
mflo    $t0
addi    $t1, $0, 32
add     $t0, $t0, $t1
sw      $t0, temp_F
```

temp_F = int(9*temp_C/5 + 32)

Print temp_F

```
la      $a0, str
addi    $v0, $0, 4
syscall
lw      $a0, temp_F
addi    $v0, $0, 1
syscall
addi    $v0, $0, 10
syscall
```

Summary

- Simple MIPS Programming
  - String handling
  - Arithmetic
- Input/Output: system calls
- Data Segment
- Text Segment
- Assembler directives