Computer Architecture
(ENE1004)

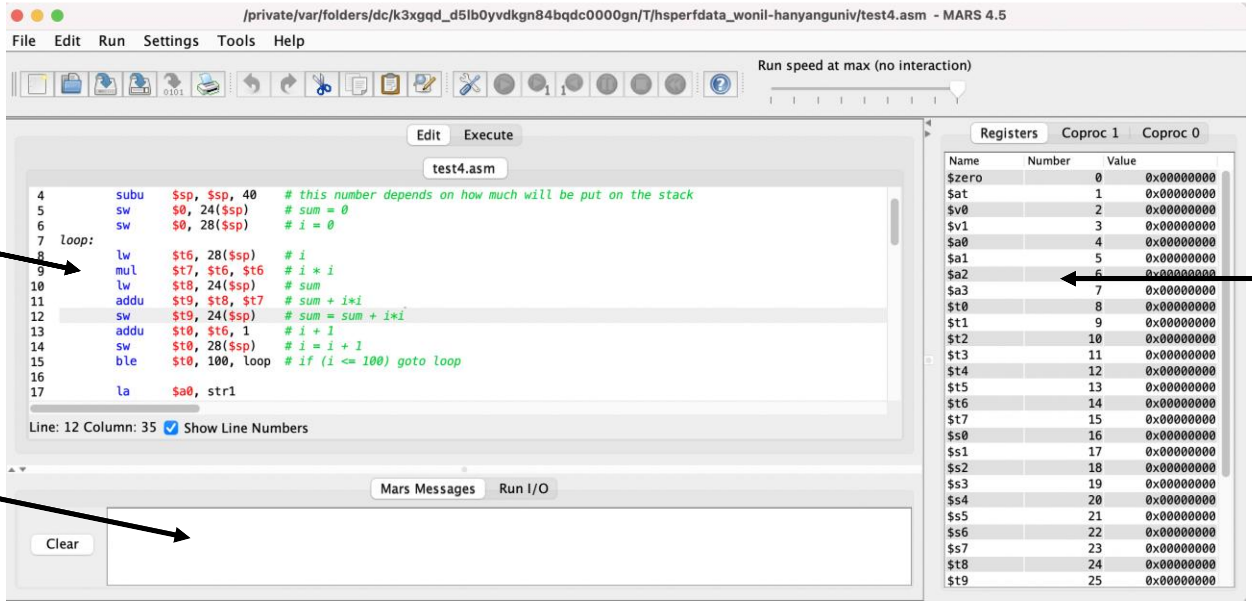
Assignment – 1: MIPS Assembly Language Programming

MARS: A MIPS Simulator

- MARS (MIPS Assembler and Runtime Simulator)
  - Download this from http://courses.missouristate.edu/kenvollmar/mars/
  - MARS is IDE for MIPS assembly programming (you can actually assemble/execute your code)
  - Note that your machine is not a MIPS processor
- MARS is a java program
  - Install a recent version of JDK/JRE from https://www.oracle.com/java/technologies/downloads/
- Start MARS by just double-clicking on its icon

Editor: write your code here

Message: print the code output



Registers: check the values in registers

Self Study is Required!

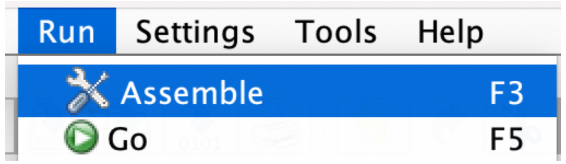
- Google it if you have any question
  - You should be able to address your problems from google as engineer, programmer, or researcher
- About MIPS instruction set
  - https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf
  - You can use pseudo-instructions (provided by assembler, not processor)
- About MARS
  - https://courses.missouristate.edu/kenvollmar/mars/Help/MarsHelpIntro.html
  - You can use system calls such as print and file I/O
- Our goal is NOT to master MIPS assembly language or instruction set
  - Let me introduce various examples, which you can begin with
  - Study only what you need to address your problem (complete your assignment)

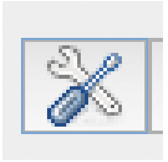
test1.asm

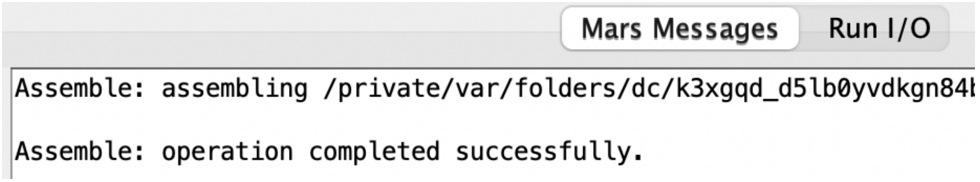
- (1) Start MARS, click on the New File button (or menu File / New), then write your program into the editor window
  - Save your program (e.g., test1.asm)

```
# version 1, do nothing, then exit

# text segment
        .text
main:
        # the main program goes here
        #
        #
        # end the program
        li $v0, 10
        syscall

# data segment
        .data

```

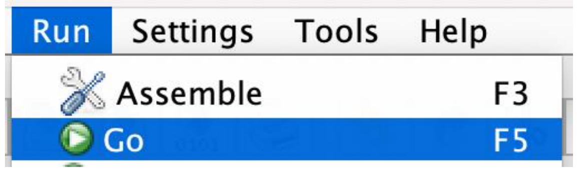
- (2) Assemble your program

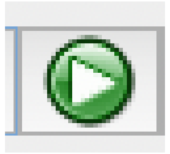


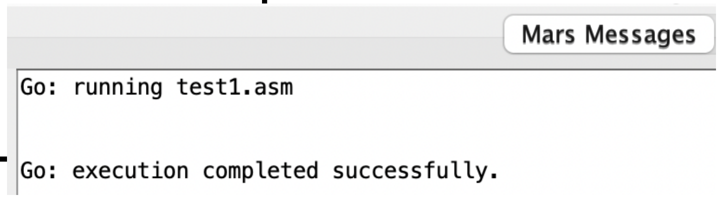


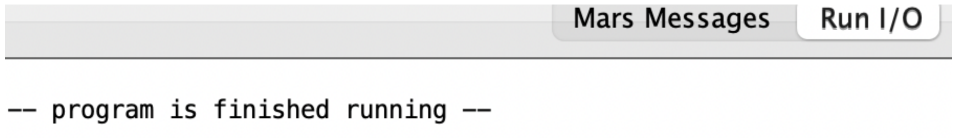


- (3) Run your program









test1.asm

```
# version 1, do nothing, then exit

# text segment
        .text
main:
        # the main program goes here
        #
        #
        # end the program
        li $v0, 10
        syscall

# data segment
        .data

```

- Comments begin with # and go to the end
- Directives begin with a dot, tab-indented
  - **.text** for text segment
  - **.data** for static data segment
- **li (load immediate)** is a pseudo-instruction
  - It is actually **addiu (add immediate unsigned)**
- **syscall** instruction initiates an OS action
  - System call depends on the value in $v0
  - Syscall 10 is like exit() function in C/C++

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x2402000a	addiu $2,$0,0x0000000a	10: li $v0, 10
	0x00400004	0x0000000c	syscall	11: syscall

$v0	2	0x0000000a

pc			0x00400008

test2.asm

```
# version 2, do nothing, then exit

# text segment
        .text
main:
        # the main program goes here

        # call Exit
        jal     Exit

# data segment
        .data


# text segment
        .text

Print_integer:  # print the integer in register a0
        li      $v0, 1
        syscall
        jr      $ra

Print_string:   # print the string whose starting address is in register a0
        li      $v0, 4
        syscall
        jr      $ra

Exit:           # end the program, no explicit return status
        li      $v0, 10
        syscall
        jr      $ra

Exit2:          # end the program, with return status from register a0
        li      $v0, 17
        syscall
        jr      $ra
```

Text Segment

Bkpt	Address	Code	Basic	Source		
	0x00400000	0x0c100007	jal 0x0040001c	9:	jal	Exit
	0x00400004	0x24020001	addiu $2,$0,0x00000001	19:	li	$v0, 1
	0x00400008	0x0000000c	syscall	20:	syscall	
	0x0040000c	0x03e00008	jr $31	21:	jr	$ra
	0x00400010	0x24020004	addiu $2,$0,0x00000004	24:	li	$v0, 4
	0x00400014	0x0000000c	syscall	25:	syscall	
	0x00400018	0x03e00008	jr $31	26:	jr	$ra
	0x0040001c	0x2402000a	addiu $2,$0,0x0000000a	29:	li	$v0, 10

- Add some function definitions, which make it easier to deal with system calls
  - Note that the second **.text** directive for text segment
  - Print_integer (syscall 1), Print_string (syscall 4), Exit (syscall 10), Exit2 (syscall 17)

test3.asm

```
# version 3, print something, then exit

# text segment
        .text
main:
        # the main program goes here
        la      $a0, hello_string
        jal     Print_string

        # call Exit
        jal     Exit

# data segment
        .data
hello_string:
        .asciiz "Hello, world\n"

# text segment
        .text

Print_integer:  # print the integer in register a0
        li      $v0, 1
        syscall
        jr      $ra

Print_string:   # print the string whose starting address is in register a0
        li      $v0, 4
        syscall
        jr      $ra

Exit:           # end the program, no explicit return status
        li      $v0, 10
        syscall
        jr      $ra

Exit2:           # end the program, with return status from register a0
        li      $v0, 17
        syscall
        jr      $ra
```

- Using .data directive, you can define static data segment and store data
  - To store a string, label + .asciiz directive + string

Text Segment

Bkpt	Address	Code	Basic	Source		
	0x00400000	0x3c011001	lui $1,0x00001001	7:	la	$a0, hello_string
	0x00400004	0x34240000	ori $4,$1,0x00000000			
	0x00400008	0x0c100007	jal 0x0040001c	8:	jal	Print_string
	0x0040000c	0x0c10000a	jal 0x00400028	11:	jal	Exit
	0x00400010	0x24020001	addiu $2,$0,0x00000001	22:	li	$v0, 1
	0x00400014	0x0000000c	syscall	23:	syscall	
	0x00400018	0x03e00008	jr $31	24:	jr	$ra
	0x0040001c	0x24020004	addiu $2,$0,0x00000004	27:	li	$v0, 4
	0x00400020	0x0000000c	syscall	28:	syscall	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x6c6c6548	0x77202c6f	0x646c726f	0x0000000a	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

test4.asm

```
        .text

main:
        subu $sp, $sp, 40 # this number depends on how much will be put on the stack
        sw $0, 24($sp) # sum = 0
        sw $0, 28($sp) # i = 0
loop:
        lw $t6, 28($sp) # i
        mul $t7, $t6, $t6 # i * i
        lw $t8, 24($sp) # sum
        addu $t9, $t8, $t7 # sum + i*i
        sw $t9, 24($sp) # sum = sum + i*i
        addu $t0, $t6, 1 # i + 1
        sw $t0, 28($sp) # i = i + 1
        ble $t0, 100, loop # if (i <= 100) goto loop

        la $a0, str1
        jal Print_string # print the string whose starting address is in register a0
        lw $a0, 24($sp) # sum
        jal Print_integer # print the integer in register a0
        la $a0, str2
        jal Print_string # print the string whose starting address is in register a0

        move $a0, $0
        jal Exit2 # end the program, with return status from register a0
```

```

        .data
# .align 0
str1:
        .asciiz "The sum from 0 .. 100 is :"
str2:
        .asciiz "\n"

```

```
# switch to the Text segment
        .text

Print_integer: # print the integer in register a0
        li $v0, 1
        syscall
        jr $ra

Print_string: # print the string whose starting address is in register a0
        li $v0, 4
        syscall
        jr $ra

Exit: # end the program, no explicit return status
        li $v0, 10
        syscall
        jr $ra

Exit2: # end the program, with return
        li $v0, 17
        syscall
        jr $ra
```

```
The sum from 0 .. 100 is :338350

-- program is finished running --
```

- How are the local variables (**i** and **sum**) stored in the stack?
- How is the **for** loop implemented?
- What is the output of program?

Specification

- The goal is to implement a function named “shuffle32” that rearranges the bits of a 32-bit integer
  - Label the bits from 0 (least significant, right end) to 31 (most significant, left end)
  - The function argument is placed in $a0, the function result goes into $v0

```
the "perfect shuffle" on 32-bit units
$a0 (as bits) 11111111111111110000000000000000
$v0 (as bits) 10101010101010101010101010101010
$a0 (as original bit positions)
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
$v0 (as original bit positions)
  31 15 30 14 29 13 28 12 27 11 26 10 25 09 24 08 23 07 22 06 21 05 20 04 19 03 18 02 17 01 16 00
```

- Examples of 4-bit/8-bit shuffle

original

3	2	1	0

final

3	1	2	0

original

7	6	5	4	3	2	1	0

final

7	3	6	2	5	1	4	0

Specification

- A skeleton code (assignment1.asm) is given
- Test cases is also given (find them in assignment1.asm)

```
 i testcase[i]          testcase[i] in binary        shuffled result in binary     result
 0 0xffffffff 11111111111111111111111111111111 11111111111111111111111111111111 0xffffffff
 1 0xffff0000 11111111111111110000000000000000 10101010101010101010101010101010 0xaaaaaaaa
 2 0x0000ffff 00000000000000001111111111111111 01010101010101010101010101010101 0x55555555
 3 0xff00ff00 11111111000000001111111100000000 11111111111111110000000000000000 0xffff0000
 4 0x00ff00ff 00000000111111110000000011111111 00000000000000001111111111111111 0x0000ffff
 5 0xf0f0f0f0 11110000111100001111000011110000 11111111000000001111111100000000 0xff00ff00
 6 0x0f0f0f0f 00001111000011110000111100001111 00000000111111110000000011111111 0x00ff00ff
 7 0xcccccccc 11001100110011001100110011001100 11110000111100001111000011110000 0xf0f0f0f0
 8 0x33333333 00110011001100110011001100110011 00001111000011110000111100001111 0x0f0f0f0f
 9 0xaaaaaaaa 10101010101010101010101010101010 11001100110011001100110011001100 0xcccccccc
10 0x55555555 01010101010101010101010101010101 00110011001100110011001100110011 0x33333333
11 0x00000000 00000000000000000000000000000000 00000000000000000000000000000000 0x00000000
12 0xffff0000 11111111111111110000000000000000 10101010101010101010101010101010 0xaaaaaaaa
13 0xaaaaaaaa 10101010101010101010101010101010 11001100110011001100110011001100 0xcccccccc
14 0xcccccccc 11001100110011001100110011001100 11110000111100001111000011110000 0xf0f0f0f0
15 0xf0f0f0f0 11110000111100001111000011110000 11111111000000001111111100000000 0xff00ff00
16 0xff00ff00 11111111000000001111111100000000 11111111111111110000000000000000 0xffff0000
17 0x12345678 00010010001101000101011001111000 00010011000111000001111101100000 0x131c1f60
All done!
```

- Submission
    - Upload your file, named “name_id.asm”, to LMS
    - Due by May 14 (Sun) at midnight
- Never cheat! If you cheat, you will get an F
    - Do your best; try to submit your best version even if your program does not work well