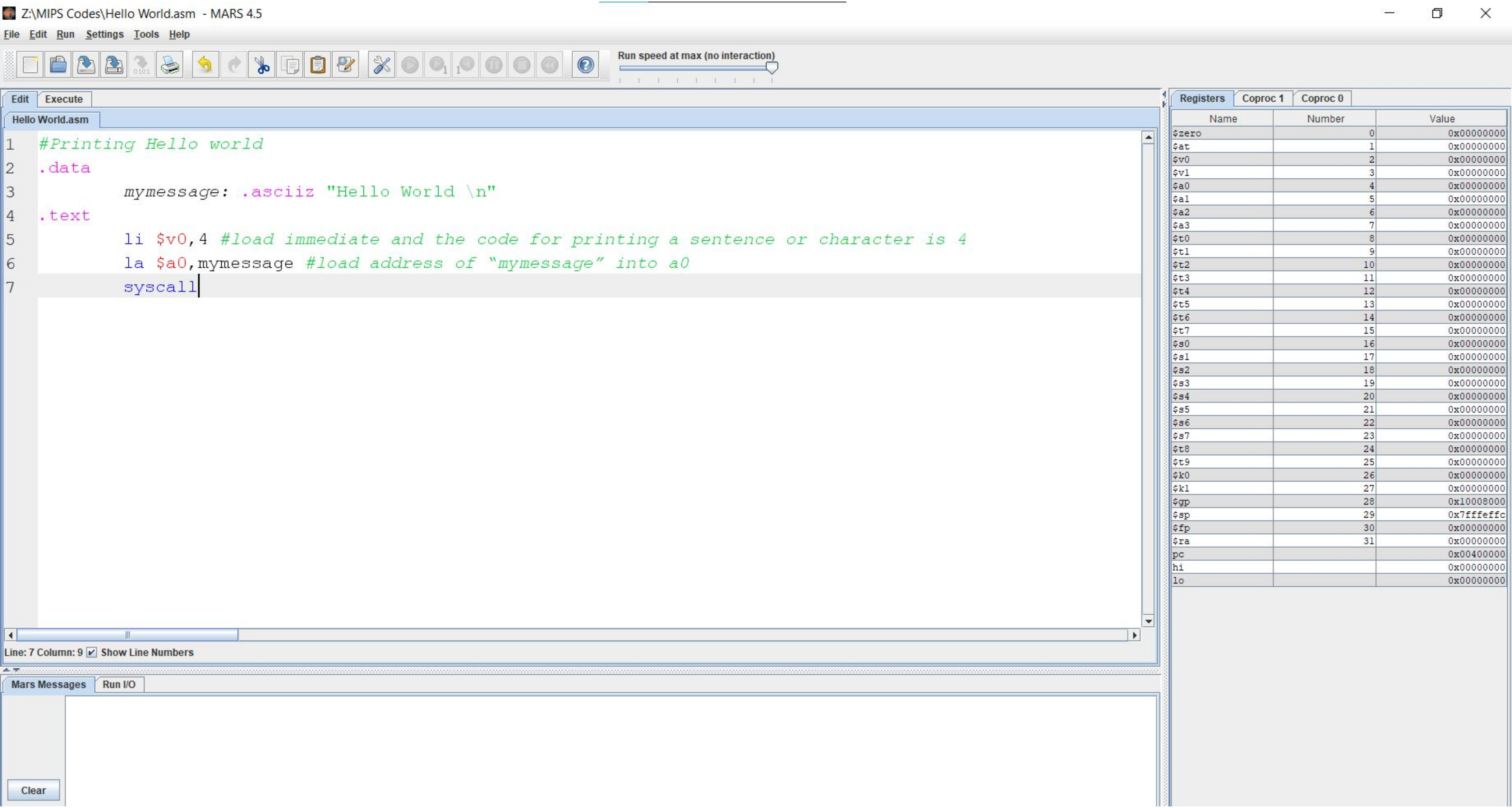
MIPS Assembly Programming

CSF342: Computer Architecture

MIPS Assembly and Simulator

- RISC architecture - more efficient than CISC because it takes less clock cycles and resources. Easier to implement CPU designs such as pipelining.
- If you have an x86 or ARM CPU arch, you need a simulator to run MIPS instruction set arch.
- We use MARS simulator, that runs on a Java VM.
- https://courses.missouristate.edu/KenVollmar/mars/download.htm and download MARS simulator. It's a .jar file. Execute via command line:
  - *java -jar Mars4_5.jar*


Z:\MIPS Codes\Hello World.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Hello World.asm
#Printing Hello world
.data
mymessage: .asciiz "Hello World \n"
.text
li $v0,4 #load immediate and the code for printing a sentence or character is 4
la $a0,mymessage #load address of “mymessage” into a0
syscall
Line: 7 Column: 9 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0x00000000
$at 1 0x00000000
$v0 2 0x00000000
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000000
$t1 9 0x00000000
$t2 10 0x00000000
$t3 11 0x00000000
$t4 12 0x00000000
$t5 13 0x00000000
$t6 14 0x00000000
$t7 15 0x00000000
$s0 16 0x00000000
$s1 17 0x00000000
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10008000
$sp 29 0x7fffeffc
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x00400000
hi 0x00000000
lo 0x00000000

Syscalls

- $v0: determines syscall action
  - 1 - prints integer
  - 2 - prints float
  - 3 - prints double
  - 4 - prints null terminated character/string
  - 5 - inputs an integer
  - 6 - inputs float
  - 7 - inputs double
  - 8 - inputs string
  - 9 - allocate heap memory
  - 10 - exit program

```
.data
	myCharacter: .byte 'm'  #byte is 8-bit value
	myAge: .word 20 # word is 32 bits , an integer requires 32 bits of memory
	myFloat: .float 9.99 #32 bit single precision floating point
	myDouble: .double 5.6655643333344 #64 bit double precision

.text
	#Printing an integer
	li $v0,1 #load immediate and the code for printing an integer is 1
	lw $a0,myAge #load word , loads value of myAge into $a0
	syscall

	#Printing character
	li $v0,4 #load immediate and the code for printing a sentence or character is 4
	la $a0,myCharacter #load address of myCharacter into a0
	syscall

	#Printing float
	li $v0,2 #load immediate and the code for a float is 2
	lwc1 $f12,myFloat
	#load word into co-processor1 since unlike integers floats are in co-processor-1
	syscall

	#Printing double
	li $v0,3 # 3 is the code for printing a double
	ldc1 $f12,myDouble
	#64-bit double value will be stored in the f12 and f13 registers(2 32-bit registers)
	syscall
```

ine: 29 Column: 6 Show Line Numbers

Mars Messages | Run I/O

Clear

```
20m9.995.6655643333344
-- program is finished running (dropped off bottom) --
```

```
.data
        prompt: .asciiz "Enter your age:"
        message: .asciiz " \n Your age is:"
.text
        li $v0,4
        la $a0,prompt
        syscall

        #To get the integer input
        li $v0,5
        syscall

        #move the age into $t0
        move $t0,$v0
        #To print the message
        li $v0,4
        la $a0,message
        syscall
        li $v0,1
        move $a0,$t0
        syscall
```

```
.data
        userInput: .space 20
.text
        #To get the input
        li $v0,8
        la $a0,userInput
        la $a1,20
        syscall
        #To print the input
        li $v0,4
        la $a0,userInput
        syscall
```

Arithmetic Operations

```
.data
        num1: .word 5
        num2: .word 10
.text
        #Addition
        lw $t0,num1 #Get value of num1 into $t0
        lw $t1,num2 #Get value of num2 into $t1
        add $t2,$t0,$t1 #$t2 = $t0 + $t1
        #Printing the result
        li $v0,1
        move $a0,$t2 #move contents of $t2 into $a0
        syscall

        #Subtraction
        lw $t0,num1 #Get value of num1 into $t0
        lw $t1,num2 #Get value of num2 into $t1
        sub $t2,$t0,$t1 #$t2 = $t0 - $t1
        #Printing the result
        li $v0,1
        move $a0,$t2 #move contents of $t2 into $a0
        syscall
```

```
#Multiplication using Mul
mul $t2,$t0,$t1 #mul can multiply only two 16 bit-numbers
#Printing the result
li $v0,1
move $a0,$t2 #move contents of $t2 into $a0
syscall

#Multiplication using Mult
mult $t0,$t1 #$t0 = $s0*$s1
#Printing the result
#Register $lo will contain the lower 32 bits of the result
#Register $hi will contain the higher 32 bits of the result
li $v0,1
mflo $t2 #move contents of lo into t0
move $a0,$t2 #move contents of $t2 into $a0
syscall
```

```
#Division using div
addi $s0,$zero,30
addi $s1,$zero,10
div  $s2,$s0,$s1
#s2 = s0/s1
#Print the result
li $v0,1
add $a0,$zero,$s2 #move contents of $s2 into $a0
syscall

#Another way to do division is by using HI and LO registers
div $s0,$s1
#Here lo will have the quotient and hi will have remainder
mflo $t0
mfhi $t1
#Print the quotient
li $v0,1
add $a0,$zero,$t0 #move contents of $s2 into $a0
syscall
#Print the remainder
li $v0,1
add $a0,$zero,$t1 #move contents of $s2 into $a0
syscall
```

Functions

- Just use labels and *jal* instr.
- But this is only valid for simple functions.
  - What if functions overwrite registers $s0, $s1 … ?
  - What if there is a function call inside a function call? The $ra value will get corrupt.

```
.data

.text
        main:
        # $a regsiters i.e $$a1,$a2... are used for passing values(parameters) to functions
        addi $a1,$zero,10
        addi $a2,$zero,15
        jal addNumbers #Calling the function
        li $v0,1
        addi $a0,$v1,0
        syscall
        #Whenever using functions the following 2 lines are necessary
        #These two lines basically tell the processor to exit program
        li $v0,10
        syscall

        #Function that is going to be called
        addNumbers:
                add $v1,$a1,$a2
                #Generally $v1 registers are used for returning values
                jr $ra #go back to the function that called it
```

```
#Function that is going to be called
increaseNumber:
        #Since we want to work with the value in $s0 register we have to first save
        #Following two lines of code does it
        addi $sp,$sp,-8
        sw $s0,0($sp)
        sw $ra,4($sp)
        #Once we have saved it in the stack we can work with the $s0 register
        addi $s0,$s0,15
        jal printFunction
        #Now we restore the value of $s0 from the stack
        lw $ra,4($sp)
        lw $s0,0($sp)
        addi $sp,$sp,8
        #Generally $v1 registers are used for returning values
        jr $ra #go back to the function that called it
        printFunction:
                li $v0,1
                addi $a0,$s0,0
                syscall
                jr $ra
```

Conditionals & Branching

- *beq, bne, blt, bgt, ble, bge, slt, slti*
- Pseudoinstrs: *bgtz , bltz, b*
- Jump instructions:

jump	`j 1000`
jump register	`jr $1`
jump and link	`jal 1000`

- Branch to Label: "*b someLabel* "

```
.text
    main:
        addi $t0, $zero, 20
        addi $t1, $zero, 20

        beq $t0, $t1, numbersEqual

        # Syscall to end program
        li $v0, 10
        syscall

    numbersEqual:
        li $v0, 4
        la $a0, message
        syscall
```

Looping using conditionals:

```
.text
    main:
        # i = 0
         addi $t0, $zero, 0

         while:
              bgt $t0, 10, exit
              jal printNumber

              addi $t0, $t0, 1        # i++ or i = i + 1

              j while

         exit:
             li $v0, 4
             la $a0, message
             syscall
```

Floating Point Arithmetic

- Coprocessor 1 is used to store floating point numbers. Use *lwc1, ldc1* etc.
- *lwc1* uses just two registers as arguments ($f2 and $f4 here, because floats are 32 bits)
- *ldc1* uses four registers because it needs 64 bits (Specifying $f2 means both $f2 and $f3, specifying $f4 means both $f4 and $f5)

```
.data
    number1:  .float 3.14
    number2:  .float 2.71

.text
   lwc1 $f2, number1
   lwc1 $f4, number2

   add.s $f12, $f2, $f4
```

```
.data
    number1:  .double 3.14
    number2:  .double 2.71

.text
   ldc1 $f2, number1
   ldc1 $f4, number2

   add.d $f12, $f2, $f4

   li $v0, 3
   syscall
```

Floating Point Conditionals

- We cannot use beq, blt etc. on floats and doubles.
- Use coprocessor compare instructions:
- Coproc has an 8-bit flag register.
  - Using *c.eq.s 5, $f0, $f1* makes the 5th flag
  - (out of 8 flags) as true if the regs are equal.
- To check if any flag is true, use *bc1t or bc1f*
  - *Eg.*
    c.eq.s 5, $f0, $f1
    bc1t 5, label

`c.eq.d`	Compare equal double precision
`c.eq.s`	Compare equal single precision
`c.le.d`	Compare less or equal double precision
`c.le.s`	Compare less or equal single precision
`c.lt.d`	Compare less than double precision
`c.lt.s`	Compare less than single precision

```
c.eq.s $f0,$f1
c.eq.s 1,$f0,$f1
```

Bit Manipulation Logical Operations

- *and, or, not, xor* operations available to do bit manipulation.

```
and $a1, $a2, $a3
or $s0, $s1, $s2
```

References

- http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html
- https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf
- "MIPS Assembly Programming Simplified" by Amell Peralta : www.youtube.com/playlist?list=PL5b07qlmA3P6zUdDf-o97ddfpvPFuNa5A