[Computer Architecture]

Instructions: the Language of Computer

Writing MIPS Assembly with QtSPIM

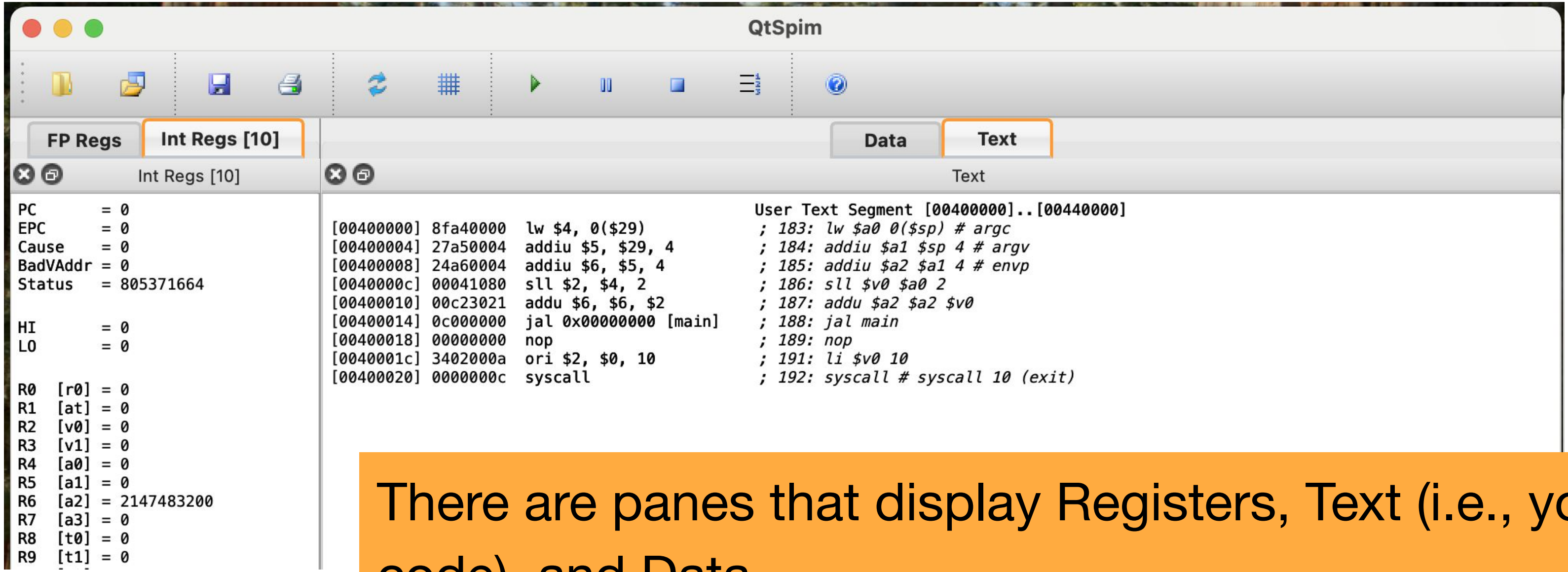
Agenda

- QtSpim overview
- Getting start with QtSpim
- Examples

QtSPIM

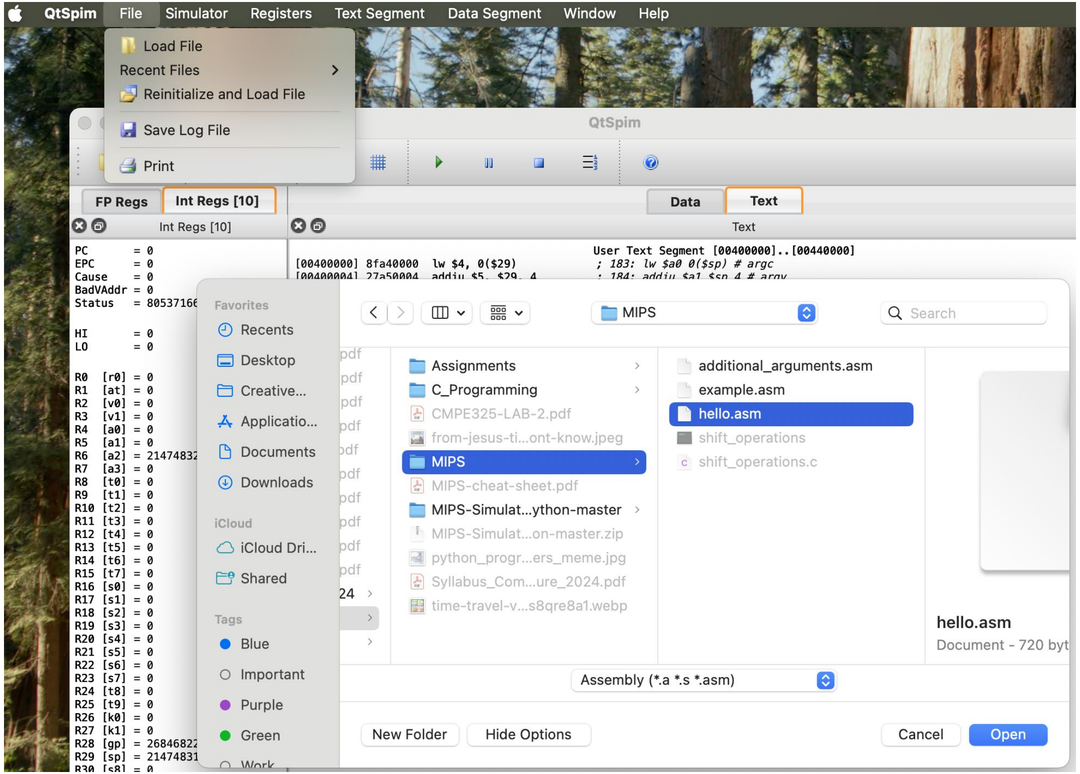
- SPIM is a MIPS simulator, and QtSPIM is a graphical user interface for it.
- QtSPIM is cross platform and runs on Windows, Mac OS, and Linux.
- Please download and install QtSPIM from: https://spimsimulator.sourceforge.net/

Getting start with QtSPIM



There are panes that display Registers, Text (i.e., your code), and Data.

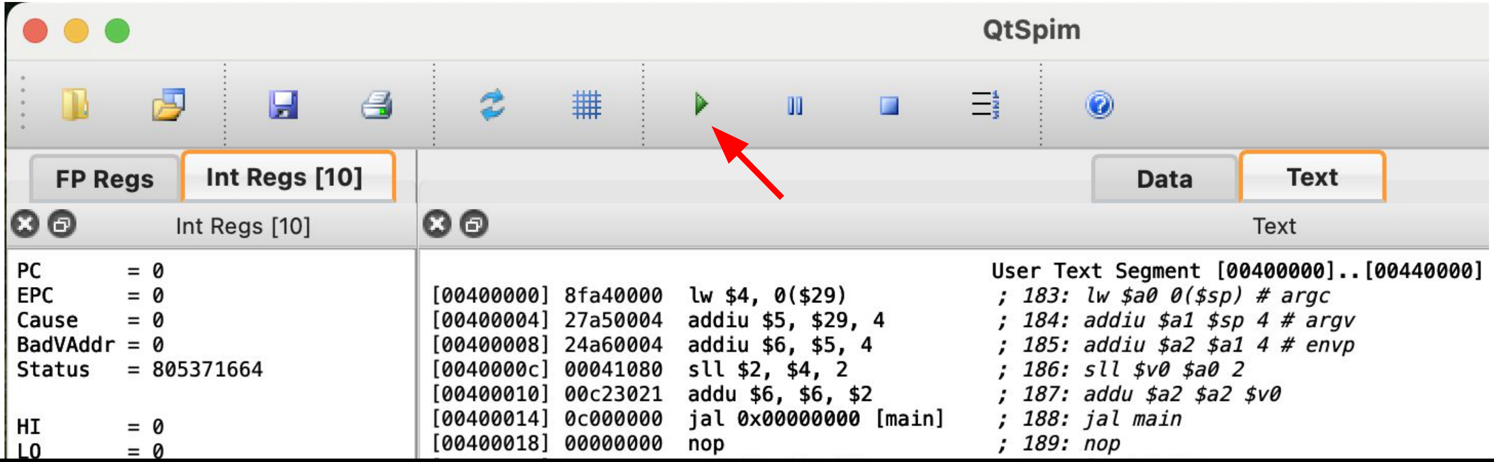
Loading a Program



Your program should be stored in a file.

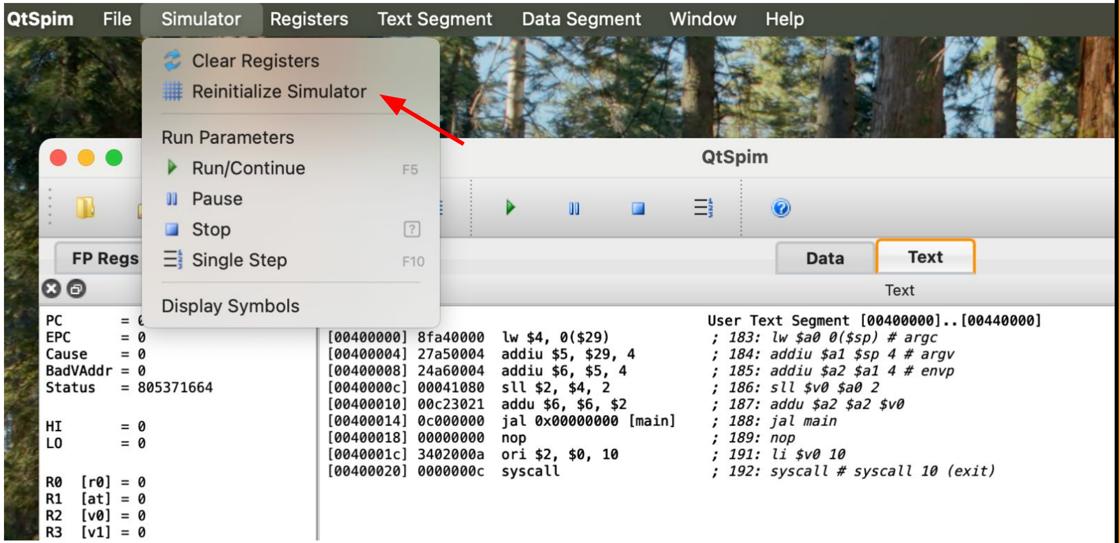
Usually, assembly code files have the extension “.s” or “.ams”.

Running a program



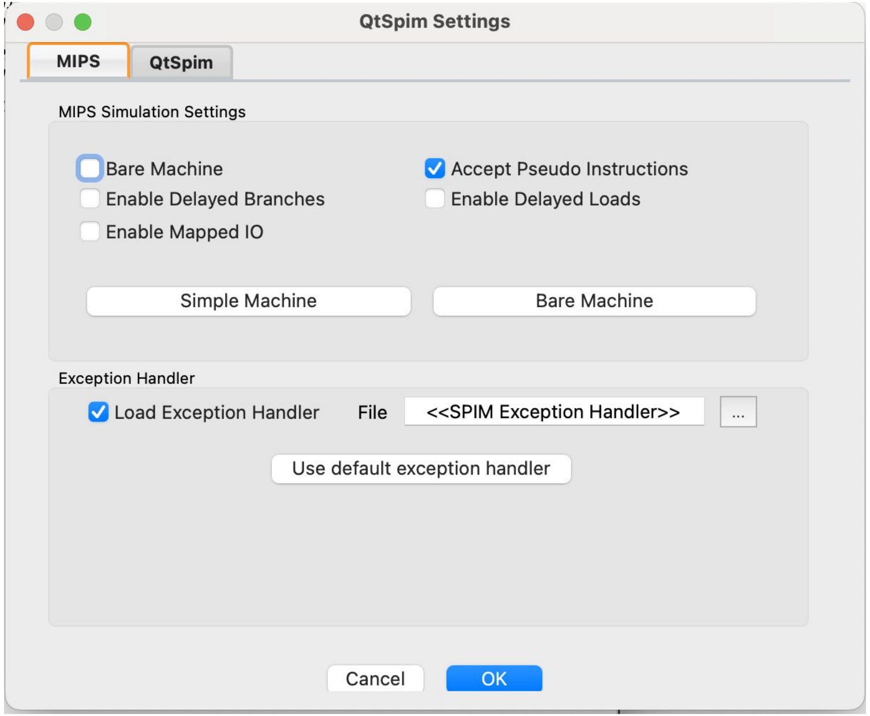
- Similar to other IDEs, QtSPIM provides buttons to run the program.
- You can see changes your program made to the registers and memory.
- The output of your program writes will appear in the Console window.

Reinitialize the simulator



It allows you to reset your simulator, which clears all changes made by a program including the loaded file.

Settings



There are two tabs. One of them allows to change the visual aspects of QtSPIM, such as fonts.

The other one allows you to changes the way that QtSPIM operates. For example, whether the pseudo instructions are allowed.

Writing MIPS Programs

MIPS programs are typically split into two sections:

- Data Segment (.data): Used for declaring and initializing variables
- Text Segment (.text): Contains the actual instructions that the processor will execute.

```
.data
msg: .asciiz "Hello, World!"

.text
.globl main

main:
    li $v0, 4           # Syscall code for printing a string
    la $a0, msg         # Load address of msg into $a0
    syscall             # Make the syscall
    li $v0, 10          # Syscall code for program exit
    syscall             # Exit the program
```

You can use VS Code or any text editor to write the code.

Saving the file with extension “.ams”.

Assembler Directives

- An assembler directive is a message to the assembler that tells it something it needs to know in order to carry out the assembly process. Assembler directives start with a ‘.’. They are required to define the start and end of data declarations and procedures/functions.

```
.data
msg: .asciiz "Hello, World!"

.text
.globl main

main:
    li $v0, 4           # Syscall code for printing a string
    la $a0, msg         # Load address of msg into $a0
    syscall             # Make the syscall
    li $v0, 10          # Syscall code for program exit
    syscall             # Exit the program
```

`.data` and `.text` are assembler directives.

Data Declarations

The data must be declared in the “.data” section. All variables and constants are placed in this section.

```
.data
msg: .asciiz "Hello, World!"

.text
.globl main

main:
    li $v0, 4           # Syscall code for printing a string
    la $a0, msg         # Load address of msg into $a0
    syscall             # Make the syscall
    li $v0, 10          # Syscall code for program exit
    syscall             # Exit the program
```

Variable names must start with a letter and terminated with a “:”

The general format is:
<variableName>: .<dataType> <initialValue>

There are various data types:

- `.asciiz` NULL terminated ASCII string
- `.word` 32-bit variable
- …

Data types

Declaration	
`.byte`	8-bit variable(s)
`.half`	16-bit variable(s)
`.word`	32-bit variable(s)
`.ascii`	ASCII string
`.asciiz`	NULL terminated ASCII string
`.float`	32 bit IEEE floating-point number
`.double`	64 bit IEEE floating-point number
`.space <n>`	<n> bytes of uninitialized memory

Program Code

The code must be preceded by the “.text” directive. The “`main`” is the first procedure to be executed.

```
.data
msg: .asciiz "Hello, World!"

.text
.globl main

main:
    li $v0, 4           # Syscall code for printing a string
    la $a0, msg         # Load address of msg into $a0
    syscall             # Make the syscall
    li $v0, 10          # Syscall code for program exit
    syscall             # Exit the program
```

`.globl` directive is used to make a symbol (e.g., a function or variable) **global**. So, it can be accessed from other files or other parts of the program.

`main` should be global.

Syscalls

Syscalls are a set of predefined services provided by the OS that can be invoked from MIPS assembly programs.

```
.data
msg: .asciiz "Hello, World!"

.text
.globl main

main:
    li $v0, 4           # Syscall code for printing a string
    la $a0, msg         # Load address of msg into $a0
    syscall             # Make the syscall
    li $v0, 10          # Syscall code for program exit
    syscall             # Exit the program
```

In the example, `li` loads 4 in `$v0`, `la` stores the address of msg into `$a0`; `syscall` invokes the execution of syscall numbered as 4, i.e., Print string

To use a syscall in MIPS, you should:

1. Load a service number into register `$v0` to specify which syscall you want to use
2. Pass the required arguments in specific registers (like `$a0`, `$a1`, etc.)
3. Invoke the syscall using the `syscall` instruction
4. If the syscall returns a value, it will typically be in register `$v0`.

In the example, it prints the `msg` into the console.

Common System calls in QtSPIM

Service Name	Call Code	Input	Output
Print Integer (32-bit)	1	**\$a0** : integer to be printed	
Print Float (32-bit)	2	**\$f12** : 32-bit floating-point value to be printed	
Print Double (64-bit)	3	**\$f12** : 64-bit floating-point value to be printed	
Print String	4	**\$a0** : starting address of NULL terminated string to be printed	
Read Integer (32-bit)	5		**\$v0** : 32-bit integer entered by user
Read Float (32-bit)	6		**\$f0** : 32-bit floating-point value entered by user

Functions

```
# Function defined below main
add_numbers:
    addi $sp, $sp, -8           # Adjust stack to save registers
    sw $ra, 4($sp)              # Save return address
    sw $s0, 0($sp)              # Save $s0 if needed

    # Function logic: add two arguments in $a0 and $a1
    add $v0, $a0, $a1           # Store result in $v0

    # Restore the saved registers
    lw $ra, 4($sp)              # Load return address
    lw $s0, 0($sp)              # Restore $s0 if saved
    addi $sp, $sp, 8            # Restore stack pointer

    jr $ra                      # Return to caller
```

Syntax for defining a function in QtSPIM

1. Label: the function name is defined as a label
2. Prologue: save the current state (especially `$ra` for return address and other registers if needed)
3. Function body: the main logic of the function
4. Epilogue: restore saved registers and return to the calling function.

Program structure

Data section

- Contains variables, string, or constants
- You define all the data needed for the program in it.

Text section

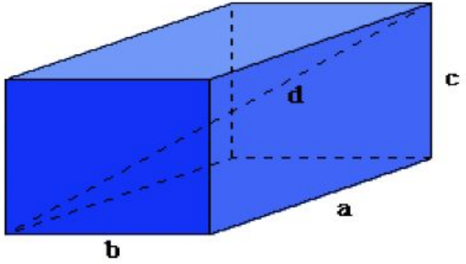
- Contains your code, including the main function and other functions or subroutines.
- Typically, the main function comes first because it is the entry point for the program
- Function definitions follow after main, but you can place them after or before main.

```
.data
    msg: .asciiz "Hello, World!\n"  # Example string

.text
    .globl main                     # Main program starts here
main:
    li $a0, 5                       # First argument
    li $a1, 10                      # Second argument
    jal add_numbers                 # Call the function 'add_numbers'

    # The result from the function is now in $v0
    move $t0, $v0                   # Store result in $t0

    # Print result (optional)
    li $v0, 1                       # Syscall for print integer
    move $a0, $t0                   # Move result to $a0 to print
    syscall

    li $v0, 10                      # Exit the program
    syscall

# Function defined below main
add_numbers:
    addi $sp, $sp, -8               # Adjust stack to save registers
    sw $ra, 4($sp)                  # Save return address
    sw $s0, 0($sp)                  # Save $s0 if needed

    # Function logic: add two arguments in $a0 and $a1
    add $v0, $a0, $a1               # Store result in $v0

    # Restore the saved registers
    lw $ra, 4($sp)                  # Load return address
    lw $s0, 0($sp)                  # Restore $s0 if saved
    addi $sp, $sp, 8                # Restore stack pointer

    jr $ra                          # Return to caller
```

Example 1: rectangular parallelepiped

Compute the volume and surface area of a rectangular parallelepiped.

- Volume = aSide*bSide*cSide
- SurfaceArea = 2(aSide*bSide + aSide*cSide +bSide*cSide)



Solution

```
# Example to compute the volume and surface area
# of a rectangular parallelepiped.
# ---------------------
# Data Declarations
.data

aSide:          .word 73
bSide:          .word 14
cSide:          .word 16

volume:         .word 0
surfaceArea:    .word 0
```

Defining the data and variables

Solution

```
# --------------------
# Text/code section
.text
.globl      main

main:

# --------------------
# Load variables into registers

    lw $t0, aSide
    lw $t1, bSide
    lw $t2, cSide
```

Load values to registers

Solution

```
# --------------------
# Find volume of a rectangular parallelepiped
# volume = aSide * bSide * cSide

    mul $t3, $t0, $t1
    mul $t4, $t3, $t2
    sw  $t4, volume

# --------------------
# Find surface area of a rectangular parallelepiped.
# surfaceArea = 2*(aSide*bSide + aSide*cSide + bSide*cSide)

    mul $t3, $t0, $t1
    mul $t4, $t0, $t2
    mul $t5, $t1, $t2
    add $t6, $t3, $t4
    add $t7, $t6, $t5
    mul $t7, $t7, 2
    sw $t7, surfaceArea
```

Compute the values needed and save them to the addresses

Solution

```
# --------------------
#print the results
    li $v0 1             # call code for print integer
    lw $a0 volume        # load $a0 of volume
    syscall
    li $v0 1             # call code for print integer
    lw $a0 surfaceArea   # load $a0 of surface area
    syscall
# --------------------
# Done, terminate program
    li $v0, 10  # call code for terminate
    syscall     # system call
```

Print the results in the console using system call

End the program using system call

This piece of code is routine. It is always need to terminate at the end of the programs.

Example 2: compute the sum of squares

Given `n`, compute $1^2 + 2^2 + \ldots + n^2$.

```
# Data Declarations
.data
n:                .word 10
sumOfSquares:     .word 0
```

Example 3: find the median of a sorted list

Given a sorted array of integers, find the median of it. The array is even length.

The median is defined as

```
median = (array[len/2] + array[len/2-1])/2
```

```
# Data Declarations
.data
array:  .word 1,   3,  5,  7,  9, 11, 13, 15, 17, 19
        .word 21, 23, 25, 27, 29, 31, 33, 35, 37, 39
        .word 41, 43, 45, 47, 49, 51, 53, 55, 57, 59
length: .word 30
median: .word 0
```

The multiple "`.word`" in the array is a way to separate a large amount of data into multiple lines.

Exercise (from H&H 6.21)

Consider the MIPS assembly code below, func1, func2, and func3 are non-leaf functions. Func4 is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function.

```
0x00401000    func1 : . . .        # func1 uses $s0-$s1
0x00401020            jal func2
  . . .
0x00401100    func2:  . . .        # func2 uses $s2-$s7
0x0040117C            jal func3
  . . .
0x00401400    func3:  . . .        # func3 uses $s1-$s3
0x00401704            jal func4
  . . .
0x00403008    func4:  . . .        # func4 uses no preserved
0x00403118            jr $ra       # registers
```

Questions:

(a) How many words are the stack frames of each function?

(b) Sketch the stack after func4 is called. Clearly indicate which registers are stored where on the stack and mark each of the stack frames. Given values where possible.