

# CENG3420 Computer Organization and Design

## Lab 1-1: MIPS assembly language programming

**Wen Zong**

Department of Computer Science and Engineering  
The Chinese University of Hong Kong

*wzong@cse.cuhk.edu.hk*



香港中文大學  
The Chinese University of Hong Kong

# Overview

## Assembly programming

Programmer view of a MIPS32 machine

Preliminaries of assembly programming

## Using SPIM

## System service in SPIM

## Lab assignment

# Overview

## Assembly programming

Programmer view of a MIPS32 machine

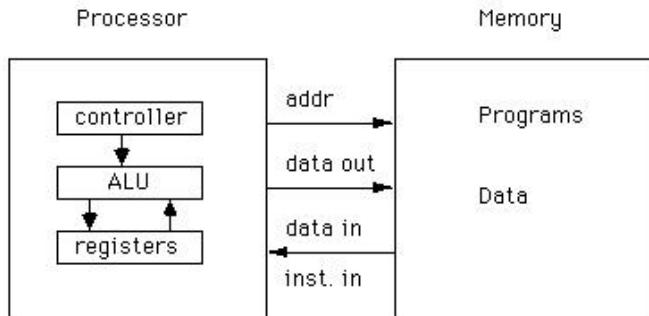
Preliminaries of assembly programming

## Using SPIM

## System service in SPIM

## Lab assignment

# Abstraction of Computer



Question:

1. Where's cache?
2. Why to know programmers' view?

# Registers

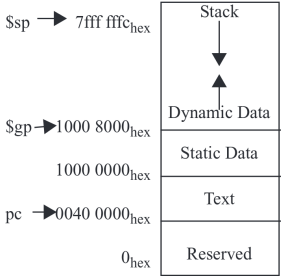
- ▶ 32 general-purpose registers
- ▶ register preceded by \$ in assembly language instruction
- ▶ two formats for addressing:
  - ▶ using register number e.g. \$0 through \$31
  - ▶ using equivalent names e.g. \$t1, \$sp
- ▶ special registers Lo and Hi used to store result of multiplication and division
- ▶ not directly addressable; contents accessed with special instruction mfhi (“move from Hi”) and mflo (“move from Lo”)
- ▶ stack grows from high memory to low memory

# Register Names and Descriptions

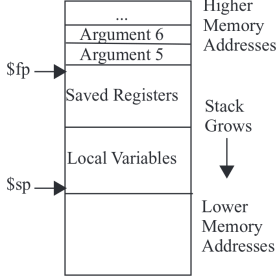
Register Number	Alternative Name	Description
0	zero	the value 0
1	\$at	(assembler temporary) reserved by the assembler
2-3	\$v0 - \$v1	(values) from expression evaluation and function results
4-7	\$a0 - \$a3	(arguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	\$t0 - \$t7	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	\$s0 - \$s7	(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	\$t8 - \$t9	(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to \$t0 - \$t7 above. Not preserved across procedure calls.
26-27	\$k0 - \$k1	reserved for use by the interrupt/trap handler
28	\$gp	<b>g</b> lobal <b>p</b> ointer. Points to the middle of the 64K block of memory in the static data segment.
29	\$sp	<b>s</b> tack <b>p</b> ointer Points to last location on the stack.
30	\$s8/\$fp	saved value / <b>f</b> rame <b>p</b> ointer Preserved across procedure calls
31	\$ra	<b>r</b> eturn <b>a</b> ddress

# Memory Allocation of A Program

## MEMORY ALLOCATION



## STACK FRAME



# Data Types and Literals

## Data types:

- ▶ Instructions are all 32 bits
- ▶ byte(8 bits), halfword (2 bytes), word (4 bytes)
- ▶ a character requires 1 byte of storage
- ▶ an integer requires 1 word (4 bytes) of storage

## Literals:

- ▶ numbers entered as is. e.g. 4
- ▶ characters enclosed in single quotes. e.g. 'b'
- ▶ strings enclosed in double quotes. e.g. "A string"



# Program Structure I

- ▶ Just plain text file with data declarations, program code (name of file should end in suffix `.s` to be used with SPIM simulator)
- ▶ Data declaration section followed by program code section

## Data Declarations

1. placed in section of program identified with assembler directive **.data**.
2. declares variable names used in program; storage allocated in main memory (RAM)

## Code

# Program Structure II

1. placed in section of text identified with assembler directive **.text**
2. contains program code (instructions)
3. starting point for code execution given label **main:**,
4. ending point of main code should use exit system call

## Comments

anything following `#` on a line

The structure of an assembly program looks like this:

## Program outline

# Program Structure III

```
# Comment giving name of program and description
# Template.s
# Bare-bones outline of MIPS assembly language program

    .data        # variable declarations follow this line
                # ...

    .text        # instructions follow this line

main:           # indicates start of code
                # ...

# End of program, leave a blank line afterwards
# to make SPIM happy
```

# An Example Program I

```
# Declare main as a global function
.globl main
# All memory structures are placed after the
# .data assembler directive
.data
# The .word assembler directive reserves space
value: .word 12
msg:   .asciiz "Hello CENG3420!\n"

# All program code is placed after the
# .text assembler directive
# The label 'main' represents the starting point
.text
main:
li $t2, 25 # Load immediate value (25)
lw $t3, value # Load the word stored at label 'value'
add $t4, $t2, $t3 # Add
sub $t5, $t2, $t3 # Subtract
la $a0, msg      # Pointer to string
li $v0, 4        # to use print_string syscall
syscall

# Exit the program by means of a syscall.
# There are many syscalls - pick the desired one
# by placing its code in $v0. The code for exit is "10"
li $v0, 10 # Sets $v0 to "10" to select exit syscall
syscall # Exit
```

## Pseudo instruction I

Some instructions in this example are pseudo instructions which will be translated to MIPS instructions by the assembler. Here's a list of useful pseudo-instructions.

- ▶ *mov \$t0, \$t1*: Copy contents of register t1 to register t0.
- ▶ *li \$s0, immed*: Load immediate into to register s0. The way this is translated depends on whether immed is 16 bits or 32 bits.
- ▶ *la \$s0, addr*: Load address into to register s0.
- ▶ *lw \$t0, address*: Load a word at address into register t0
- ▶ Similar pseudo-instructions exist for *sw*, etc

Translating some pseudoinstructions

- ▶ *mov \$t0, \$s0* → *addi \$t0, \$s0, 0*
- ▶ *li \$rs, small* → *addi \$rs, \$zero, small*
- ▶ *li \$rs, big* → *lui \$rs, upper(big) ori \$rs, \$rs, lower(big)*
- ▶ *la \$rs, big* → *lui \$rs, upper(big) ori \$rs, \$rs, lower(big)*

## Pseudo instruction II

1. where small means a quantity that can be represented using 16 bits, and big means a 32 bit quantity. upper(big) is the upper 16 bits of a 32 bit quantity. lower(big) is the lower 16 bits of the 32 bit quantity.
2. upper( big ) and lower(big) are not real instructions. If you were to do the translation, you'd have to break it up yourself to figure out those quantities.

## More Information

For more information about MIPS instructions and assembly programming you can refer to:

1. Lecture slides and textbook.
2. Google

# Overview

## Assembly programming

Programmer view of a MIPS32 machine

Preliminaries of assembly programming

## Using SPIM

System service in SPIM

Lab assignment



# What is SPIM

- ▶ **SPIM is a MIPS32 simulator.**
- ▶ *Spim* is a self-contained simulator that runs MIPS32 programs.
- ▶ It reads and executes assembly language programs written for this processor.
- ▶ *Spim* also provides a simple debugger and minimal set of operating system services.
- ▶ *Spim* does not execute binary (compiled) programs.

Download it here:

<http://sourceforge.net/projects/spimsimulator/files/>

# SPIM Overview

The screenshot displays the QtSpim MIPS simulator interface. The main window is titled "QtSpim" and contains several panes. At the top, there is a menu bar with "File", "Simulator", "Registers", "Text Segment", "Data Segment", "Window", and "Help". Below the menu bar is a toolbar with various icons for simulation control. The "FP Regs" and "nt Regs [16]" tabs are visible, with "Data" and "Text" tabs selected. The "FP Regs" pane on the left shows floating-point registers F0 through F29, all of which are currently set to 0. The main "Text" pane displays assembly code for two segments: "User Text Segment [00400000]..[00460000]" and "Kernel Text Segment [80000000]..[80010000]". The assembly code includes instructions such as `lw $t0, 0($t3)`, `addiu $t1, $t3, 4`, `ori $t2, $t0, 10`, `syscall`, `add $t4, $t2, $t3`, `sub $t3, $t0, $t1`, `lui $t1, 4097`, `ori $t4, $t1, 4`, `addiu $t2, $t0, $t1`, `sw $t2, 512($t1)`, `mf0 $t3, $t3`, `andi $t4, $t4, 31`, `ori $t2, $t0, 4`, `lui $t4, -28672`, `syscall`, `ori $t2, $t0, 1`, `ori $t4, $t4, 31`, and `syscall`. The instruction `ori $t2, $t0, 10` is highlighted in blue. At the bottom of the window, there are two text blocks containing copyright information for SPIM version 9.1.17 of January 1, 2016, by James R. Larus, and a notice that QtSPIM is linked to the Qt library under the GNU Lesser General Public License version 3 and version 2.1.

What SPIM looks like.

# Register Panel and Memory Panel

The screenshot displays the QtSpim MIPS simulator interface. At the top, there is a menu bar with options: File, Simulator, Registers, Text Segment, Data Segment, Window, and Help. Below the menu bar is a toolbar with various icons for file operations and simulation control. The main window is divided into three panels:

- Register Panel:** Located on the left side, it shows the state of MIPS registers. The 'FP Regs' section is expanded, listing registers from FFR to FFR29. The 'Single Precision' section lists registers from FFD0 to FFD3. The 'Memory Panel' label is overlaid on this panel.
- Memory Panel:** Located in the center, it displays assembly code with comments. The code is organized into segments: 'User Text Segment [00400000]..[00460000]', 'Kernel Text Segment [80000000]..[80010000]', and 'Data Segment [00000000]..[00000000]'. The current instruction being executed is highlighted in blue: `00400044 0000000c syscall`. The 'Memory Panel' label is overlaid on this panel.
- Message Panel:** Located at the bottom, it shows the output of the simulator. It includes copyright information for James H. Larus (1990-2016) and a notice about the BSD license. The 'Message Panel' label is overlaid on this panel.

At the bottom of the window, there is a status bar with navigation icons and a search icon.

There's also a console window.

# Operations

- ▶ Load a source file: File → Reinitialize and Load File
- ▶ Run the code: F5 or Press the green triangle button
- ▶ Single stepping: F10
- ▶ Breakpoint: in Text panel, right click on an address to set a breakpoint there.

# Overview

## Assembly programming

Programmer view of a MIPS32 machine

Preliminaries of assembly programming

## Using SPIM

## System service in SPIM

## Lab assignment

# System calls in SPIM I

SPIM provides a small set of operating system-like services through the system call ( syscall ) instruction.

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$v0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

## System calls in SPIM II

To request a service, a program loads the system call code into register `$v0` and arguments into registers `$a0` - `$a3` (or `$f12` for floating-point values). System calls that return values put their results in register `$v0` (or `$f0` for floating-point results). Like this example:

Using system call

## System calls in SPIM III

```
.data
str: .asciiz "the answer = "
.text

li    $v0, 4    # system call code for print_str
la    $a0, str  # address of string to print
syscall                # print the string
li    $v0, 1    # system call code for print_int
li    $a0, 5    # integer to print
syscall                # print it
```



# Run An Example Program

Download the file from [course website](#) and run it on your computer.

# Overview

## Assembly programming

Programmer view of a MIPS32 machine

Preliminaries of assembly programming

## Using SPIM

## System service in SPIM

## Lab assignment

# Lab Assignment

Finish these two assignments and submit your code (.s file) to elearn system before **Feb. 05** (midnight).

1. Write an assembly program that outputs your student ID.
2. Write an assembly program that outputs the odd digit in your student ID (e.g. sid 1155012345 should output 1155135).  
The SID is required to be declared as an array of word in the data segment.