

Lecture 9

Programming MIPS

- MIPS is an assembly language
- SPIM is the simulator for MIPS instructions
 - Check the webpage for “Help with SPIM” (<http://www.cs.sunysb.edu/~cse220/helpwithSpim.html>)
 - SPIM is written in C. When we execute SPIM it acts like a MIPS machine
 - It assembles (compiles) and loads your assembly language program into simulated memory
 - Provides 10 system calls for input/output and program exit
- Why use this? Why should we learn this?
 - Best way to learn how a processor functions

Compare C to MIPS code

```
/* hello.c
 * Purpose - print hello
 */
#include <stdio.h>

int main(void){
    printf("Hello, world!\n");

    return 0;
}
```

Note: the code below is available to you on Sparky, in directory `~cse220/examples/spim`

```
## hello.asm - print out "hello world"
##      a0 - points to the string
#####
#      text segment      #
#####
        .text           # tells assembler program code starts here
        .globl main     # defines label for execution start
main:   # execution starts here
        la $a0, str     # put string address into a0
        li $v0, 4       # system call to print
        syscall        # out a string

        li $v0, 10      # Load exit syscall value
        syscall        # Exit
#####
#      data segment      #
#####
        .data          # tells assembler data segment begins here
str:   .asciiz "hello world\n" # declaration of a string
```

	C	MIPS
<i>Comments</i>	// or /* */	#
<i>Main function</i>	int main (void) { }	.text .globl main main:
<i>Variable Declaration</i>	Inside the main function, or global	In own section .data

Format of Assembly Programs

- Every statement/line is divided into 4 fields
 - [Label:] operation [operands] [#comment]
 - Fields in [] are optional
 - Number of operands required depends on the operation (0, 1, 2, or 3 operands)
- *Labels* sequence of alphanumeric characters, underscore(_) and dot(.). Can not begin with a number
 - *End with a colon (:)*
 - After assembly the label refers to the address of where the line of MIPS code (instruction or variable/data) is stored in memory
 - Example: in Hello.asm there are 2 labels
 - *main:* is a required label (you may call it something else)
 - This label denotes the position of the first executable instruction
 - '*label*' for the first line of code and the '*.globl label*' must match
 - *str:* is a label for declaring a string in the program
 - It represents the starting address in memory where the string is stored
- *Operation* field contains the code for actual operation (*opcode*) or assembler directive
 - An operation is a type of instruction for the CPU to execute
 - Example: *la* → load address
li → load immediate
 - An assembler directive begins with a dot (.)
 - It is not an *opcode*, but a command to the assembler
 - Example: Some of directives
 - *.text* → beginning of the text segment
 - *.data* → beginning of data segment
 - *.asciiz* → declares an *ascii* string terminated by NULL
 - *.ascii* → an *ascii* string, not terminated by NULL
 - Strings (*.asciiz* & *.ascii*) are enclosed in "...". They recognize newline ('\n'), tab ('\t'), etc. [Standard C convention]

- *Operand* field contains register names, variable names, labels, and small immediate operands (values)
 - Example: *la \$a0, str*
 - Two operands: register and label (variable)
 - Meaning: load address of *str* in register *\$a0*
 - Example: *li \$v0, 4*
 - Two operands: register and small immediate operand (value)
 - Meaning: load immediate operand, 4, into register *\$v0*
 - *Immediate operand* is a small constant which is stored in the instruction itself. It is not read from memory
- *Comments* are anything after the '#' symbol until the end of the line
 - It is a good idea to include plenty of comments
 - Assembly programs are not easy to debug or maintain

syscall Instruction

- This is an O/S (Operating System) call for doing input/output and exiting the program
- Example: In *hello.asm*
 - Two syscalls: 1 for printing string and one for exiting

Service	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		

- These systems calls are used to read/print integers, float and double precision numbers.
- *syscall* works by loading the call code into register *\$v0*. Also load the argument into the appropriate register shown in the table.
 - Example: Using call code 4/8 we print/read a string.

SPIM & MIPS assembler directives

- SPIM supports a subset of MIPS assembler directives.
 - *.align n* → align the next datum on a 2^n byte boundary
 - *.ascii & .asciiz* → we have already seen
 - *.byte b₁, b₂, ... b_n* → store *n* values in successive bytes of memory
 - *.double d₁, d₂, ... d_n* → store *n* double precision values in successive memory locations
 - *.float f₁, f₂, ... f_n* → store *n* floating point values in successive memory locations

- *.space n* → allocate *n* bytes in memory
- *.word w₁, w₂, ... w_n* → store *n* words in successive memory words
- *.text* , *.data* , *.globl* → we have already seen

MIPS arithmetic instructions

Instruction	Example
add	add \$t1,\$t2,\$t3 # \$t1=\$t2+\$t3
add immediate	addi \$t1,\$t2,50 # \$t1=\$t2+50
subtract	sub \$t1,\$t2,\$t3 # \$t1=\$t2-\$t3
multiply	mult \$t2,\$t3 # Hi,Lo=\$t2x\$t3
divide	div \$t2,\$t3 # Lo=\$t2÷\$t3 # Hi=\$t2 mod \$t3
move from Hi	mghi \$t1 # \$t1 = Hi # get copy of Hi
move from Lo	mflr \$t1 # \$t1 = Lo # get copy of Lo

- *add* , *addi* and *sub* will cause exceptions if there is an overflow
- *addu* , *addiu* and *subu* will not cause exceptions or overflow
 - Why? Because they are unsigned operations
 - C does not detect overflows. A C compiler will make use these instructions
- For multiplication and division, you need to use the *Hi* & *Lo* registers (see past notes on FP)
 - Product is 64 bits long

How to use SPIM

- Instructions on the web site (<http://www.cs.sunysb.edu/~cse220/helpwithSpim.html>)
- You may install the SPIM Simulator (PCSpim or MARS) on your PC and do projects on it
- If using PCSpim:
 - Click on all buttons and see what happens
 - When your program starts running an additional window (console) will open. This is for input/output
 - You can go through your program line by line and see what code has been generated by the assembler by using “Go” (F5), “step” (F10), or “multiple step” (F11)
- If using the MARS MIPS Simulator:
 - See the MARS Simulator web site (<http://courses.missouristate.edu/KenVollmar/MARS/index.htm>)