Learning MIPS & SPIM

- MIPS assembly is a low-level programming language
- The best way to learn any programming language is to write code
- We will get you started by going through a few example programs and explaining the key concepts
- Tip: Start by copying existing programs and modifying them incrementally making sure you understand the behavior at each step
- Tip: The best way to understand and remember a construct or keyword is to experiment with it in code, not by reading about it
MIPS Assembly Code Layout

- Typical Program Layout

```assembly
.text    #code section
.globl main  #starting point: must be global
main:
    # user program code
.data     #data section
    # user program data
```
MIPS Memory Usage as viewed in SPIM

- **Reserved**
  - 0x00000000
  - 0x00400000
  - 0x10010000
  - 0x7fffeffc
  - 0x7fffffff

- **Text segment (instructions)**
- **Data segment**
- **Stack segment**
MIPS Assembler Directives

• Top-level Directives:

  • `.text`
    • indicates that following items are stored in the user text segment, typically instructions

  • `.data`
    • indicates that following data items are stored in the data segment

  • `.globl` sym
    • declare that symbol sym is global and can be referenced from other files
MIPS Assembler Directives

• Common Data Definitions:

  • `.word`  \( w_1, \ldots, w_n \)
    • store \( n \) 32-bit quantities in successive memory words
  • `.half`  \( h_1, \ldots, h_n \)
    • store \( n \) 16-bit quantities in successive memory halfwords
  • `.byte`  \( b_1, \ldots, b_n \)
    • store \( n \) 8-bit quantities in successive memory bytes
  • `.ascii`  \( \text{str} \)
    • store the string in memory but do not null-terminate it
      • strings are represented in double-quotes “\( \text{str} \)”
      • special characters, eg. \( \backslash n, \backslash t \), follow C convention
  • `.asciiz`  \( \text{str} \)
    • store the string in memory and null-terminate it
MIPS Assembler Directives

- **Common Data Definitions:**
  - `.float` f1, …, fn
    - store n floating point single precision numbers in successive memory locations
  - `.double` d1, …, dn
    - store n floating point double precision numbers in successive memory locations
  - `.space` n
    - reserves n successive bytes of space
  - `.align` n
    - align the next datum on a $2^n$ byte boundary.
    - For example, `.align 2` aligns next value on a word boundary.
    - `.align 0` turns off automatic alignment of `.half`, `.word`, etc. till next `.data` directive
MIPS: Software Conventions for Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 results from callee</td>
</tr>
<tr>
<td>3</td>
<td>v1 returned to caller</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments to callee</td>
</tr>
<tr>
<td>5</td>
<td>a1 from caller: caller saves</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra return Address caller saves</td>
</tr>
</tbody>
</table>

*GP: stack pointer*
Pseudoinstructions

- **Pseudoinstructions** do not correspond to real MIPS instructions.
- Instead, the assembler would translate **pseudoinstructions** to real instructions (one or more instructions).
- **Pseudoinstructions** not only make it easier to program, it can also add clarity to the program, by making the intention of the programmer more clear.
Pseudoinstructions

• 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.
Pseudoinstructions

- 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)`

- `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.

- `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.
Pseudoinstructions

- As you look through the branch instructions, you see `beq` and `bne`, but not `bge` (branch on greater than or equal), `bgt` (branch on greater than), `ble` (branch on less than or equal), `blt` (branch on less than). There are no branch instructions for relational operators!
Pseudoinstructions

• Here's the table for translating pseudoinstructions.
  • `bge $t0, $s0, LABEL`   `slt $at, $t0, $s0`  
    `beq $at, $zero, LABEL`
  • `bgt $t0, $s0, LABEL`   `slt $at, $s0, $t0`  
    `bne $at, $zero, LABEL`
  • `ble $t0, $s0, LABEL`   `slt $at, $s0, $t0`  
    `beq $at, $zero, LABEL`
  • `blt $t0, $s0, LABEL`   `slt $at, $t0, $s0`  
    `bne $at, $zero, LABEL`
System Calls

• System Calls (syscall)
  • OS-like services

• Method
  • Load system call code into register $v0
  • Load arguments into registers $a0…$a3
  • call system with SPIM instruction syscall
  • After call, return value is in register $v0

• Frequently used system calls

<table>
<thead>
<tr>
<th>Service</th>
<th>Code($v0)</th>
<th>Arg</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print_int</td>
<td>1</td>
<td>$a1</td>
<td></td>
</tr>
<tr>
<td>Print_string</td>
<td>4</td>
<td>$a0</td>
<td></td>
</tr>
<tr>
<td>Read_int</td>
<td>5</td>
<td></td>
<td>$v0</td>
</tr>
</tbody>
</table>
# System Call Codes

<table>
<thead>
<tr>
<th>Service</th>
<th>Code (put in $v0)</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0=integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12=float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12=double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0=addr. of string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>int in $v0</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>float in $f0</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>double in $f0</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0=buffer, $a1=length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0=amount</td>
<td>addr in $v0</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QtSPIM

• QtSpim is software that will help you to simulate the execution of MIPS assembly programs.
• It does a context and syntax check while loading an assembly program.
• In addition, it adds in necessary overhead instructions as needed, and updates register and memory content as each instruction is executed.
• Download the source from the SourceForge.org link at: http://pages.cs.wisc.edu/~larus/spim.html
• Alternatively, you can go directly to: http://sourceforge.net/projects/spimsimulator/files/
• Versions for Windows, Linux, and Macs are all available
QtSPIM

- QtSPIM window is divided into different sections:
  1. The *Register tabs display the content of all registers.*
  2. Buttons across the top are used to load and run a simulation
     • Functionality is described in Figure 2.
  3. The *Text tab displays the MIPS instructions loaded into memory to be executed.*
     • From left-to-right, the memory address of an instruction, the contents of the address in hex, the actual MIPS instructions where register numbers are used, the MIPS assembly that you wrote, and any comments you made in your code are displayed.
  4. The *Data tab displays memory addresses and their values in the data and stack segments of the memory.*
  5. The *Information Console lists the actions performed by the simulator.*
Reinitialize and load file
Load file
New... Save log
Clear registers
Run simulation
Step through simulation

Reinitialize simulation
Pause, stop simulation (will likely not use)

User code: (a) your comments appear, (b) register name, number appear

Content of integer registers
- Can view as binary, hex, or decimal
- Do not need to consider floating point (FP) register tab
- PC value also included here

Simulator generated code (ignore)

MIPS code

Memory and registers cleared
Loaded: /var/folders/br/8jSg5dpm54L1pYU1hQmKwY8M000Gm/TFspc elf Temp.l7044
SPM Version 4.6 of January 9, 2011
All Rights Reserved.
QtSPIM Program Example

• A Simple Program

```assembly
#sample example 'add two numbers'

.text
.globl main

main:  la $t0, value
      lw $t1, 0($t0)
      lw $t2, 4($t0)
      add $t3, $t1, $t2
      sw $t3, 8($t0)

.data
value: .word 10, 20, 0
```

# text section
# call main by SPIM

# load address 'value' into $t0
# load word 0(value) into $t1
# load word 4(value) into $t2
# add two numbers into $t3
# store word $t3 into 8($t0)

# data section
# data for addition
QtSPIM Example Program

## Program adds 10 and 11

```
.text                   # text section
.globl main            # call main by SPIM

main:
  ori $8,$0,0xA        # load "10" into register 8
  ori $9,$0,0xB        # load "11" into register 9
  add $10,$8,$9        # add registers 8 and 9, put result
                        # in register 10
```

QtSPIM Example Program: swap2memoryWords.asm

```assembly
## Program to swap two memory words

.data          # load data
.word 7
.word 3

.text
.globl main

main:
lui $s0, 0x1001 # load data area start address 0x10010000
lw  $s1, 0($s0)
lw  $s2, 4($s0)
sw  $s2, 0($s0)
sw  $s1, 4($s0)
```
## QtSPIM Example Program: procCallsProg2.asm

### Procedure call to swap two array words

```assembly
.text
.globl main
main:
    la $a0, array
    addi $a1, $0, 0
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal swap
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
.load parameters for swap
    addi $sp, $sp, -4
    sw $ra, 0($sp)
.save return address $ra in stack
    jal swap
.swap contents of elements $a1 and $a1 + 1 of the array that starts at $a0
    lw $t0, 0($t1)
    lw $t2, 4($t1)
    sw $t0, 4($t1)
    sw $t0, 0($t1)
    jr $ra

.data
.array: .word 5, 4, 3, 2, 1
```

### Equivalent C code:

```c
#include <stdio.h>

int swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
    return 0;
}
```

```
# Procedure call to swap two array words

# load parameters for swap
# save return address $ra in stack
# jump and link to swap
# restore return address
# jump to $ra

# equivalent C code:
# swap(int v[], int k)
```
QtSPIM Example Program: systemCalls.asm

```asm
## Enter two integers in
## console window
## Sum is displayed
.text
.globl main

main:
    la $t0, value
    li $v0, 5
    syscall

li $v0, 4
la $a0, msg1
syscall

li $v0, 1
move $a0, $t3
syscall

li $v0, 10
syscall

.data
value: .word 0, 0, 0
msg1:  .asciiz "Sum = "
```

- system call code for read_int
- result returned by call
- system call code for print_string
- argument to print_string call
- system call code for print_int
- argument to print_int call
- system call code for exit