MIPS Assembly Language

Prof. James L. Frankel
Harvard University

Version of 6:28 PM 1-Oct-2024

Assembler Input

- The assembly language file should have ".s" as its file name extension
- Input contains one instruction or directive per line
  - Assembly Language instructions
  - Pseudo-instructions
  - Assembler directives
  - Lines may be prefixed by a label followed by a colon
  - Comments
    - Comments begin with a pound-sign (#) and continue through the end of the line

- SPIM includes minimal input and output system call facilities using the **syscall** instruction

Usual Assembler Input Format

- If a label is present, it begins in column one and ends with a colon
- Instruction opcodes, pseudo-instruction opcodes, and assembler directives are preceded by a tab (so that they are aligned) and follow a possible label
- If an opcode or directive has any operands, then the opcode or directive is followed by a tab so that the operands are aligned
- Comments may be on lines by themselves or may follow instructions or directives
  - If the comments follow instructions or directives, they are preceded by tabs so that they are aligned

Pseudo-Instructions

- Pseudo-instructions look like real instructions, but extend the hardware instruction set
- Each pseudo-instruction is translated into one or more real assembly language instructions
- The assembler may use register $at in generating code for pseudo-assembly language instructions

- In the documentation included with SPIM (at http://www.cs.wisc.edu/~larus/SPIM/spim_documentation.pdf), all pseudo-assembly language instructions are tagged with a dagger (†)

Examples of Pseudo-Instructions

- Absolute value: abs rdest, rsrc
- Bitwise logical NOT: not rdest, rsrc
- Load immediate: li rdest, immediate
- Set on equal: seq rdest, rsrc1, rsrc2
seq rdest, rsrc, immediate
- Unconditional branch: b label
- Load address: la rdest, label
- Copy contents of register: move rdest, rsrc

Assembler Directives

- Directives tell the assembler how to function
- Groups of directives
  - In which segment should following code or data be placed
  - Externally visible labels
  - Reserve space for data
    - Possibly initialize the values of data

Assembler Segment Directives

- .text
  - Anything that follows is placed in the text segment
  - The text segment is where executable code exists
  - .text may be followed by an address
    - Anything that follows is placed in the text segment beginning at the specified address
  - In SPIM, the text segment may contain only instructions or .word's
- .data
  - Anything that follows is placed in the data segment
  - The data segment is where ***static*** data stored in memory exists
  - .data may be followed by an address
    - Anything that follows is placed in the data segment beginning at the specified address

Externally Visible Label Directive

- .globl label
  - The specified *label* is made visible to other files
  - The *label* must be declared within the current file

- Each executable unit must have the label **`main`** declared and made externally-visible

Assembler Data Value Directives

- .word w1, w2, …
  - The value of each operand (*w1*, *w2*, etc.) is stored in a 32-bit word in memory
  - The words are aligned on word boundaries
- .half h1, h2, …
  - The value of each operand (*h1*, *h2*, etc.) is stored in a 16-bit halfword in memory
  - The halfwords are aligned on halfword boundaries
- .byte b1, b2, …
  - The value of each operand (*b1*, *b2*, etc.) is stored in a 8-bit byte in memory
  - No alignment is performed

Assembler String Value Directives

- .ascii "string"
  - The *"string"* is stored in memory using ASCII values
  - Each character is stored in an 8-bit byte
  - No alignment is performed
- .asciiz "string"
  - The *"string"* is stored in memory using ASCII values with null-termination
  - Each character is stored in an 8-bit byte
  - No alignment is performed

Assembler Data Space Directive

- .space n
  - Reserve *n* uninitialized bytes of space in memory
  - No alignment is performed

Reserving Memory for Global/Static Data

- Space for global/static variables is reserved in the **.data** segment
  - Space may be reserved using the **.word**, **.half**, **.byte**, **.ascii**, **.asciiz**, and **.space** directives

- In the C Programming Language, static variables are initialized to zero
  - Therefore, storage for all static variables should be reserved using the **.word**, **.half**, and **.byte** directives with an initial value of zero
- In the C Programming Language, literal strings are always null terminated
  - Therefore, storage for literal strings should be reserved using the **.asciiz** directive

Minimal Input/Output and Other System Calls

- print_int
- print_string
- read_int
- read_string
- exit

print_int System Call

```
        .text
        .globl  main

main:   li      $v0, 1          # $v0 <- system call code for print_int
        li      $a0, 42         # $a0 <- value of integer to be printed
        syscall                 # output the integer
```

print_string System Call

```
        .data
hello:  .asciiz  "Hello world\n"

        .text
        .globl   main

main:   li       $v0, 4          # $v0 <- system call code for print_string
        la       $a0, hello      # $a0 -> the greeting string
        syscall                  # output the greeting string
```

read_int System Call

```
        .text
        .globl  main

main:   li      $v0, 5          # $v0 <- system call code for read_int
        syscall                 # $v0 <- input integer
```

- read_int reads a complete line including the newline character and returns the value of an integer in register $v0
- Characters following the integer are consumed and ignored

read_string System Call

```
        .data
buffer: .space  256

        .text
        .globl  main

main:   li      $v0, 8          # $v0 <- system call code for read_string
        la      $a0, buffer     # $a0 -> input string buffer
        li      $a1, 256        # $a1 <- buffer length
        syscall                 # read a null-terminated string into buffer
```

- Semantics are same as for Unix/Posix fgets()

exit System Call

```
        .text
        .globl  main

main:   li      $v0, 10         # $v0 <- system call code for exit
        syscall                 # exit from the program
```

Using SPIM

- SPIM is already installed on the **cscie93.dce.harvard.edu** instance
  - You can also install a version of **QtSpim** on a Microsoft Windows, Apple Mac OS X, or Linux computer
  - See https://sourceforge.net/projects/spimsimulator/files/
- Invoke SPIM from the shell by entering "spim"
- At the "(spim) " prompt, load your code by entering

  load "filename.s"
- Run program to completing by entering

  run
- Run a single instruction by entering

  step
- Run a program from the current location to completion without pausing by entering

  continue
- Leave SPIM by entering

  exit
- The previous SPIM command can be repeated by typing simply the Enter key

Stepping a Program Under SPIM

- After entering a “step” command to SPIM, the MIPS instruction that has just completed is displayed
- Here is an example of SPIM instruction display

```
[0x00400024]    0x34080061  ori $8, $0, 97                ; 6: li    $t0,97
```

- “[0x00400024]” is the address of the instruction that just completed
- “0x34080061” is the value of the instruction word
- “ori $8, $0, 97” is the disassembly of the instruction
- “; 6: li $t0,97” is the assembly language input to SPIM added as a comment with its line number in the source file

Displaying Instructions and Data in SPIM

- At the “(spim) ” prompt, display all registers by entering

  print_all_regs
  print_all_regs hex

- Display the value of one register by entering

  print $*n*
  print $*sn*

- Display the contents of memory by entering

  print *address* (such as: print 0x10010000)
  print *label* (such as: print main)

  To be able to use a label in SPIM, it must be declared as a global symbol

- Display all labels by entering

  print_symbols

Additional SPIM Commands

- Clear all registers and memory by entering

    reinitialize

- A breakpoint is a point in the program where execution will pause when running instructions following a “run” or “continue” command
    - Execution will pause before the instruction at the breakpoint
- Set a breakpoint at an address or label by entering

    breakpoint *address*
    breakpoint *label*

- Display all breakpoints by entering

    list

Passing Command-Line Arguments to a MIPS Program Running Under SPIM

- See argcargv.s at on the class website for a program that prints out argc and each argv string
- To pass arguments using command-line version of SPIM:
  - spim "" argcargv.s a b c d
- To pass arguments using QtSpim:
  - (1) First start up qtspim
  - (2) Load the .s file to be run
  - (3) Under "Simulator", click on "Run Parameters" and enter the parameters in the "Command-line arguments to pass to program" text box
  - (4) Run the program

  - Note: qtspim does not do the correct parsing into separate parameters if directories include spaces!