C335
Computer Structures

MIPS Assembly Programming with SPIM

Dr. David R. Surma
Department of Computer and Information Sciences
Indiana University South Bend
Adapted from Morgan Kaufmann, Dr. L. Zhang and others
Outline

- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System calls
Assembly Language Statements

- Three types of statements in assembly language
  - Typically, one statement should appear on a line

1. Executable Instructions
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do

2. Pseudo-Instructions and Macros
   - Translated by the assembler into real instructions
   - Simplify the programmer task

3. Assembler Directives
   - Provide information to the assembler while translating a program
   - Used to define segments, allocate memory variables, etc.
   - Non-executable: directives are not part of the instruction set
Instructions

- Assembly language instructions have the format:
  
  [label:]  mnemonic  [operands]  [#comment]

- Label: (optional)
  - Marks the address of a memory location, must have a colon
  - Typically appear in data and text segments

- Mnemonic
  - Identifies the operation (e.g. add, sub, etc.)

- Operands
  - Specify the data required by the operation
  - Operands can be registers, memory variables, or constants
  - Most instructions have three operands

L1:  addiu $t0, $t0, 1  #increment $t0
Comments

- Comments are very important!
  - Explain the program's purpose
  - When it was written, revised, and by whom (put in header)
  - Explain data used in the program, input, and output (header)
  - Explain instruction sequences and algorithms used
  - Comments are also required at the beginning of every procedure/function
    - Indicate input parameters and results of a procedure
    - Describe what the procedure does

In the header a General Description of the overall program is required. This is to be written so that the user can read it and known what the program does.
Comments

- Single-line comment
  - Begins with a hash symbol `#` and terminates at end of line
  - *Within the code* the comments explain “what” is being done. If the user wants to know “how” the statement is done he/she can read the actual assembly language code.
- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls
# Title: 
# Author: 
# Description: 
# Input: 
# Output: 

### Data segment ### 
.data 

### Code segment ### 
.text 
.globl main 
main: 

# main program entry 

.DATA, .TEXT, & .GLOBL Directives

- **.data** directive
  - Defines the **data segment** of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables

- **.text** directive
  - Defines the **code segment** of a program containing instructions

- **.globl** directive
  - Declares a symbol as **global**
  - Global symbols can be referenced from other files
  - We use this directive to declare *main* procedure of a program
Assembly Language Statements
Assembly Language Program Template
Defining Data
Memory Alignment and Byte Ordering
System Calls
Procedures
Parameter Passing and the Runtime Stack
Data Definition Statement

- Sets aside storage in memory for a variable
- May optionally assign a name (label) to the data
- Syntax:

```plaintext
[name:] directive initializer [, initializer] . . .
```

```plaintext
    ↓    ↓    ↓
var1: .word 10
```

- All initializers become binary data in memory
Data Directives

- **.byte** Directive
  - Stores the list of values as 8-bit bytes

- **.half** Directive
  - Stores the list as 16-bit values aligned on half-word boundary

- **.word** Directive
  - Stores the list as 32-bit values aligned on a word boundary

- **.word w:n** Directive
  - Stores the 32-bit value \( w \) into \( n \) consecutive words aligned on a word boundary.
Data Directives

- **.float** Directive
  - Stores the listed values as single-precision floating point

- **.double** Directive
  - Stores the listed values as double-precision floating point
String Directives

- **.ascii** Directive
  - Allocates a sequence of bytes for an ASCII string

- **.asciiz** Directive
  - Same as **.ascii** directive, but adds a NULL char at end of string
  - Strings are null-terminated, as in the C programming language

- **.space n** Directive
  - Allocates space of $n$ uninitialized bytes in the data segment

- Special characters in strings follow C convention
  - Newline: \n  - Tab: \t  - Quote: "
### Examples of Data Definitions

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.byte</td>
<td>'A', 'E', 127, -1, '\n'</td>
</tr>
<tr>
<td>.half</td>
<td>-10, 0xffff</td>
</tr>
<tr>
<td>.word</td>
<td>0x12345678</td>
</tr>
<tr>
<td>.word</td>
<td>0:10</td>
</tr>
<tr>
<td>.float</td>
<td>12.3, -0.1</td>
</tr>
<tr>
<td>.double</td>
<td>1.5e-10</td>
</tr>
<tr>
<td>.ascii</td>
<td>&quot;A String\n&quot;</td>
</tr>
<tr>
<td>.asciiz</td>
<td>&quot;NULL Terminated String&quot;</td>
</tr>
</tbody>
</table>

If the initial value exceeds the maximum size, an error is reported by assembler.
Assembly Language Statements
Assembly Language Program Template
Defining Data
Memory Alignment and Byte Ordering
System Calls
Memory Alignment

- Memory is viewed as an array of bytes with addresses
  - Byte Addressing: address points to a byte in memory

- Words occupy 4 consecutive bytes in memory
  - MIPS instructions and integers occupy 4 bytes

- Alignment: address is a multiple of size
  - Word address should be a multiple of 4
    - Least significant 2 bits of address should be 00
  - Halfword address should be a multiple of 2

- .align n directive
  - Aligns the next data definition on a $2^n$ byte boundary
Assembler builds a symbol table for labels (variables)

- Assembler computes the address of each label in data segment

Example

```
.data
var1: .byte 1, 2, 'Z'
str1: .asciiz "My String\n"
var2: .word 0x12345678
.align 3
var3: .half 1000
```

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1</td>
<td>0x10010000</td>
</tr>
<tr>
<td>str1</td>
<td>0x10010003</td>
</tr>
<tr>
<td>var2</td>
<td>0x10010010</td>
</tr>
<tr>
<td>var3</td>
<td>0x10010018</td>
</tr>
</tbody>
</table>

```
0x10010000 1 2 'Z' 'M' 'y' ' ' 'S' 't' 'r' 'i' 'n' 'g' 'n' 0 0 0  Unused
0x10010010 0x12345678 0 0 0 0 1000 0 0 0 0 0 0  Unused
```

var1
str1
var2 (aligned)
var3 (address is multiple of 8)
Processors can order bytes within a word in two ways

- **Little Endian Byte Ordering**
  - Memory address = Address of *least significant* byte
  - Example: Intel x86

- **Big Endian Byte Ordering**
  - Memory address = Address of *most significant* byte
  - Example: Motorola 68k

- MIPS can operate with both byte orderings
Next . . .

- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls
System Calls

- Programs do input/output through system calls

- SPIM provides a special `syscall` instruction
  - To obtain services from the operating system
  - Provided both in the SPIM and MARS simulators

- Using the `syscall` system services
  - Load the service number in register `$v0$
  - Load argument values, if any, in registers `$a0$, `$a1$, etc.
  - Issue the `syscall` instruction
  - Retrieve return values, if any, from result registers
# Syscall Services

<table>
<thead>
<tr>
<th>Service</th>
<th>$v0</th>
<th>Arguments / Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print Integer</td>
<td>1</td>
<td>$a0 = integer value to print</td>
</tr>
<tr>
<td>Print Float</td>
<td>2</td>
<td>$f12 = float value to print</td>
</tr>
<tr>
<td>Print Double</td>
<td>3</td>
<td>$f12 = double value to print</td>
</tr>
<tr>
<td>Print String</td>
<td>4</td>
<td>$a0 = address of null-terminated string</td>
</tr>
<tr>
<td>Read Integer</td>
<td>5</td>
<td>$v0 = integer read</td>
</tr>
<tr>
<td>Read Float</td>
<td>6</td>
<td>$f0 = float read</td>
</tr>
<tr>
<td>Read Double</td>
<td>7</td>
<td>$f0 = double read</td>
</tr>
<tr>
<td>Read String</td>
<td>8</td>
<td>$a0 = address of input buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a1 = length</td>
</tr>
<tr>
<td>Exit Program</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Print Char</td>
<td>11</td>
<td>$a0 = character to print</td>
</tr>
<tr>
<td>Read Char</td>
<td>12</td>
<td>$a0 = character read</td>
</tr>
</tbody>
</table>

Supported by SPIM
Reading and Printing an Integer

_CODE SEGMENT_
.text
.globl main
main:
    li $v0, 5     # main program entry
    syscall      # Read integer
    move $a0, $v0 # $v0 = value read
    li $v0, 1     # $a0 = value to print
    syscall      # Print integer
Reading and Printing a String

```
### data segment ###
.data
.globl hello
hello: .asciiz "\nHello World\n"  # string to print

### code segment ###
.text
.globl main
main:
    li  $v0, 4       # print_str (system call 4)
    la  $a0, hello  # takes the address of
                  # string as an argument
    syscall
```
About Case Sensitiveness

- For SPIM:
  - Labels, Directives, Instructions are all case sensitive

- For QtSpim:
  - Labels, Directives, Instructions are all case sensitive

- For MARS:
  - Labels are case sensitive
  - Directives and Instructions are case insensitive