C335
Computer Structures

MIPS Assembly Programming with SPIM

Dr. Liqiang Zhang

Department of Computer and Information Sciences

Adapted from Morgan Kaufmann 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
  - Explain data used in the program, input, and output
  - Explain instruction sequences and algorithms used
  - Comments are also required at the beginning of every procedure
    - Indicate input parameters and results of a procedure
    - Describe what the procedure does
- Single-line comment
  - Begins with a hash symbol # and terminates at end of line

Next . . .

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

Program Template

```
# Title:                                Filename:
# Author:                               Date:
# 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

Next . . .

- 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:

[*name:*] *directive* *initializer* [, *initializer*] . . .

```
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

```
.data
var1:  .byte    'A', 'E', 127, -1, '\n'
var2:  .half    -10, 0xffff
var3:  .word    0x12345678
Var4:  .word    0:10
var5:  .float   12.3, -0.1
var6:  .double  1.5e-10
str1:  .ascii   "A String\n"
str2:  .asciiz  "NULL Terminated String"
array: .space   100
```

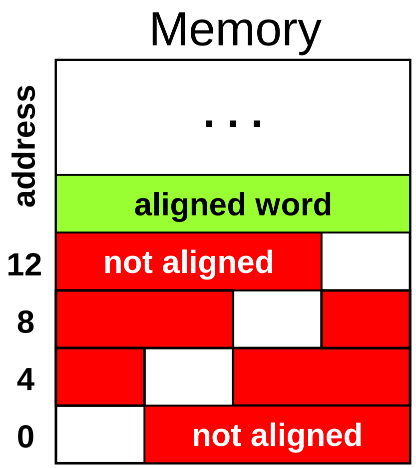
If the initial value exceeds the maximum size, an error is reported by assembler

Next . . .

- 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



Symbol Table

- 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
```

Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

var1 str1

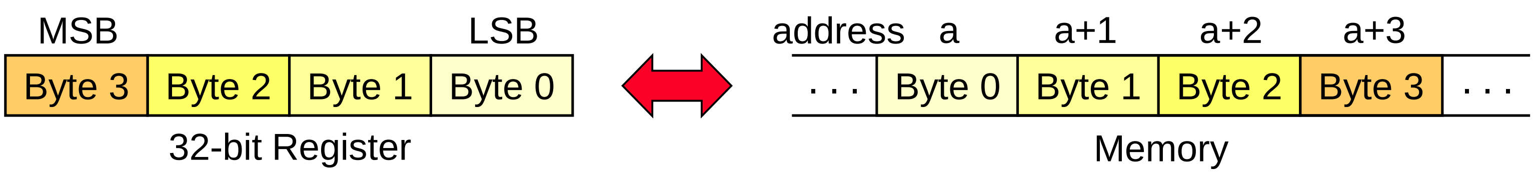
0x10010000	1	2	'Z'	'M'	'y'	' '	'S'	't'	'r'	'i'	'n'	'g'	'\n'	0	0	0
0x10010010	0x12345678				0	0	0	0	1000		0	0	0	0	0	0

Unused

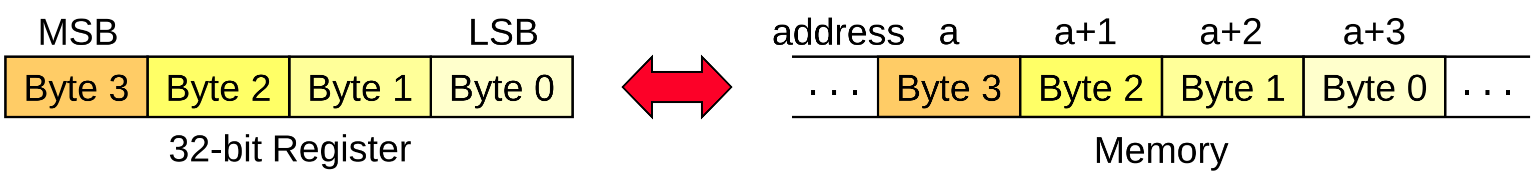
var2 (aligned) Unused var3 (address is multiple of 8)

Byte Ordering and Endianness

- 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

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

Supported by SPIM

Reading and Printing an Integer

```
################## Code segment #####################
.text
.globl main
main:                           # main program entry
  li    $v0, 5                  # Read integer
  syscall                       # $v0 = value read

  move  $a0, $v0                # $a0 = value to print
  li    $v0, 1                  # Print integer
  syscall
```

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
```