Introduction to Assembly Language Programming

Computer Architecture

Riad Bourbia

Computer Sciences department

Guelma University

[Adapted from slides of Dr. A. El-maleh]

Outline

- **The MIPS Instruction Set Architecture**
- Introduction to Assembly Language
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls

Instruction Set Architecture (ISA)

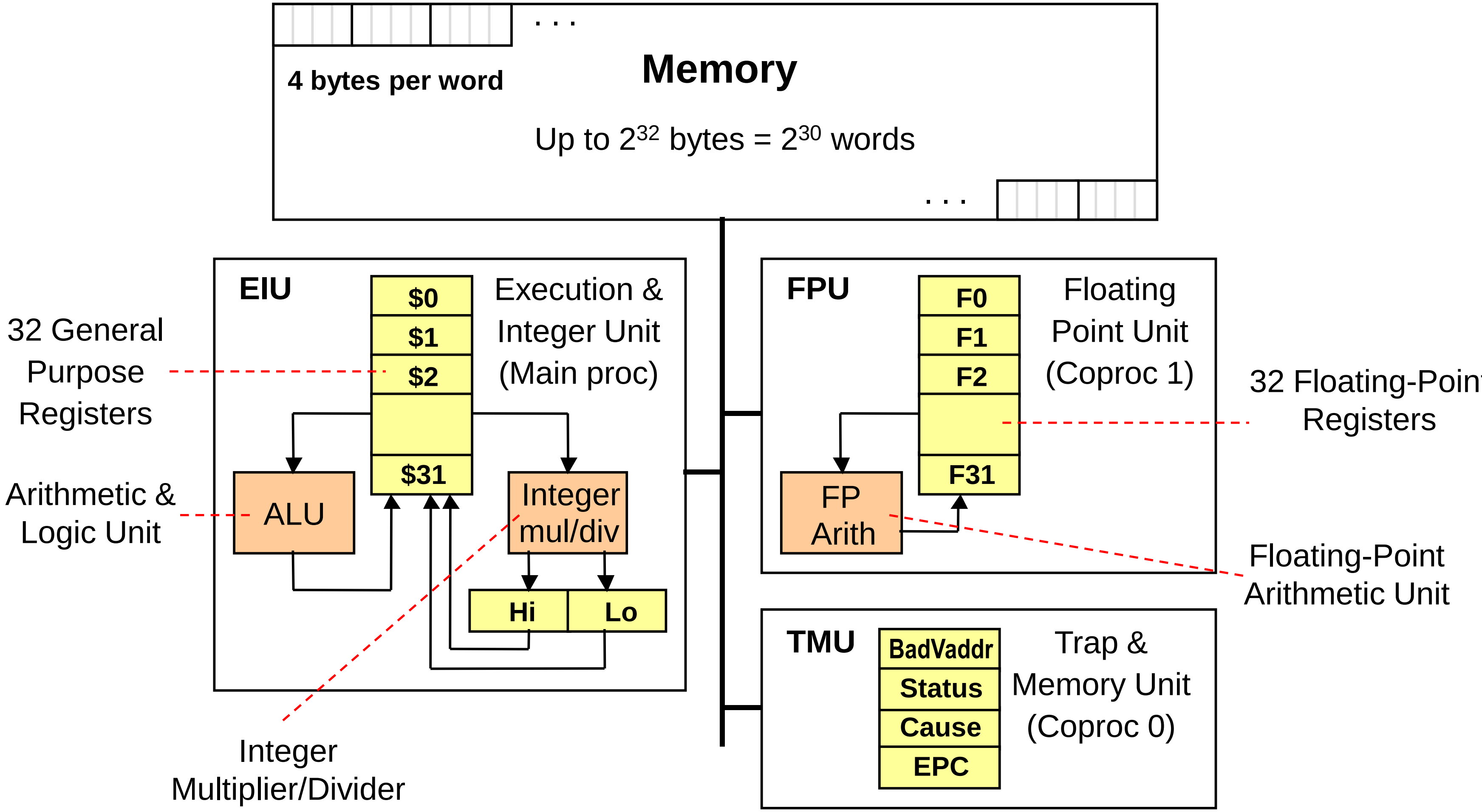
- Critical interface between hardware and software
- An ISA includes the following …
  - Instructions and Instruction Formats
    - Data Types, Encodings, and Representations
    - Addressing Modes: to address Instructions and Data
    - Handling Exceptional Conditions (like division by zero)
  - Programmable Storage: Registers and Memory

Examples	(Versions)	First Introduced in
Intel	(8086, 80386, Pentium, ...)	1978
MIPS	(MIPS I, II, III, IV, V)	1986
PowerPC	(601, 604, …)	1993

Instructions

- Instructions are the language of the machine
- We will study the MIPS instruction set architecture
  - Known as **Reduced Instruction Set Computer (RISC)**
  - Elegant and relatively simple design
  - Similar to RISC architectures developed in mid-1980's and 90's
  - Very popular, used in many products
    - Silicon Graphics, ATI, Cisco, Sony, etc.
  - Comes next in sales after Intel IA-32 processors
    - Almost 100 million MIPS processors sold in 2002 (and increasing)
- Alternative design: Intel IA-32
  - Known as **Complex Instruction Set Computer (CISC)**

Overview of the MIPS Processor



MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - Assembler uses the dollar notation to name registers
    - $0 is register 0, $1 is register 1, …, and $31 is register 31
  - All registers are 32-bit wide in MIPS32
  - Register $0 is always zero
    - Any value written to $0 is discarded
- Software conventions
  - Software defines names to all registers
    - To standardize their use in programs
  - Example: $8 - $15 are called $t0 - $t7
    - Used for temporary values

`$0 = $zero`	`$16 = $s0`
`$1 = $at`	`$17 = $s1`
`$2 = $v0`	`$18 = $s2`
`$3 = $v1`	`$19 = $s3`
`$4 = $a0`	`$20 = $s4`
`$5 = $a1`	`$21 = $s5`
`$6 = $a2`	`$22 = $s6`
`$7 = $a3`	`$23 = $s7`
`$8 = $t0`	`$24 = $t8`
`$9 = $t1`	`$25 = $t9`
`$10 = $t2`	`$26 = $k0`
`$11 = $t3`	`$27 = $k1`
`$12 = $t4`	`$28 = $gp`
`$13 = $t5`	`$29 = $sp`
`$14 = $t6`	`$30 = $fp`
`$15 = $t7`	`$31 = $ra`

MIPS Register Conventions

- Assembler can refer to registers by name or by number
  - It is easier for you to remember registers by name
  - Assembler converts register name to its corresponding number

Name	Register	Usage	
$zero	$0	Always 0	(forced by hardware)
$at	$1	Reserved for assembler use	
$v0 – $v1	$2 – $3	Result values of a function	
$a0 – $a3	$4 – $7	Arguments of a function	
$t0 – $t7	$8 – $15	Temporary Values	
$s0 – $s7	$16 – $23	Saved registers	(preserved across call)
$t8 – $t9	$24 – $25	More temporaries	
$k0 – $k1	$26 – $27	Reserved for OS kernel	
$gp	$28	Global pointer	(points to global data)
$sp	$29	Stack pointer	(points to top of stack)
$fp	$30	Frame pointer	(points to stack frame)
$ra	$31	Return address	(used by jal for function call)

Instruction Formats

- All instructions are 32-bit wide. Three instruction formats:
- Register (R-Type)
  - Register-to-register instructions
  - Op: operation code specifies the format of the instruction

Op^6	Rs^5	Rt^5	Rd^5	sa^5	$funct^6$

- Immediate (I-Type)
  - 16-bit immediate constant is part in the instruction

Op^6	Rs^5	Rt^5	$immediate^{16}$

- Jump (J-Type)
  - Used by jump instructions

Op^6	$immediate^{26}$

Next . . .

- The MIPS Instruction Set Architecture
- **Introduction to Assembly Language**
- 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

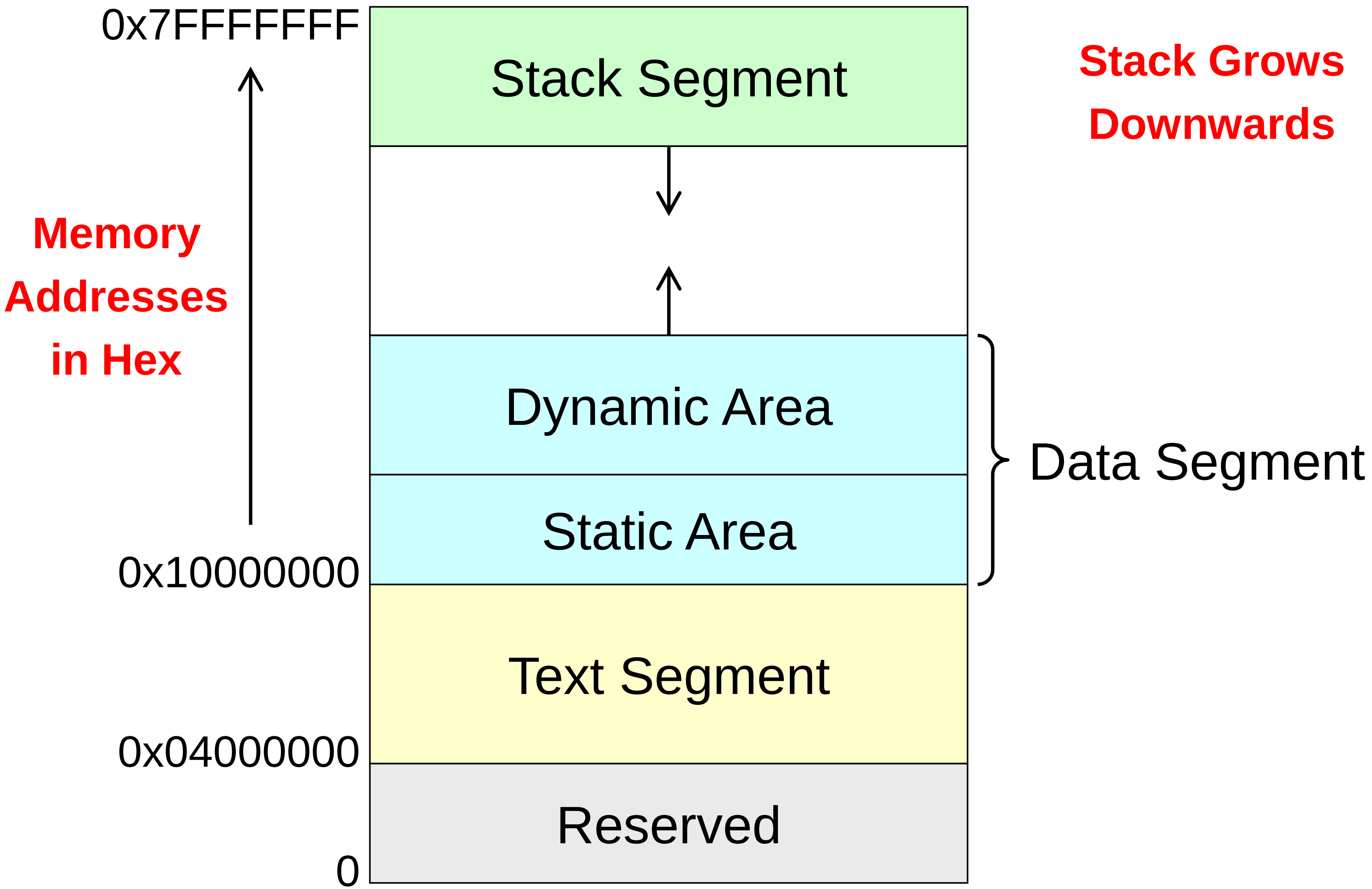
Program Template

```
# Title:                        Filename:
# Author:                       Date:
# Description:
# Input:
# Output:
################# Data segment #####################
.data
  . . .
################# Code segment #####################
.text
.globl main
main:                           # main program entry
  . . .
li $v0, 10                      # Exit program
syscall
```

.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

Layout of a Program in Memory



Next . . .

- The MIPS Instruction Set Architecture
- Introduction to Assembly Language
- **Defining Data**
- Memory Alignment and Byte Ordering
- System Calls

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

- **.HALF w:n** Directive
  - Stores the 16-bit value *w* into *n* consecutive half-words aligned on a half-word boundary .
- **.BYTE w:n** Directive
  - Stores the 8-bit value *w* into *n* consecutive bytes.
- **.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

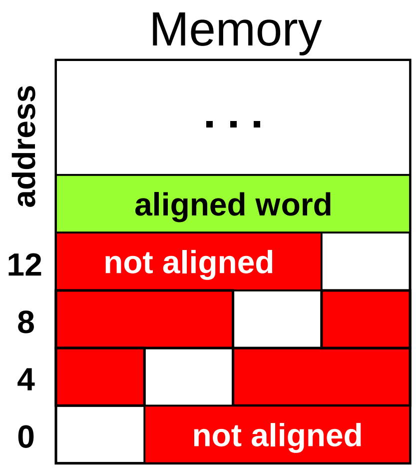
```
.DATA
var1:   .BYTE     'A', 'E', 127, -1, '\n'
var2:   .HALF     -10, 0xffff
var3:   .WORD     0x12345678
Var4:   .WORD     0:10
var5:   .FLOAT    12.3, -0.1
var6:   .DOUBLE   1.5e-10
str1:   .ASCII    "A String\n"
str2:   .ASCIIZ   "NULL Terminated String"
array:  .SPACE    100
```

Next . . .

- The MIPS Instruction Set Architecture
- Introduction to Assembly Language
- 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



Memory Alignment

- .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.
- Example: If the address of X is 0x10010000, then Address of Y is **0x10010002**

```
.align 0
X: .byte 1,2
Y: .word 10
```

- Alignment has to satisfy both the automatic boundary and the boundary given in the align directive
- Example: If the address of X is 0x10010000, then Address of Y is **0x10010004**

```
x: .byte 1
.align 1
y: .word 1
```

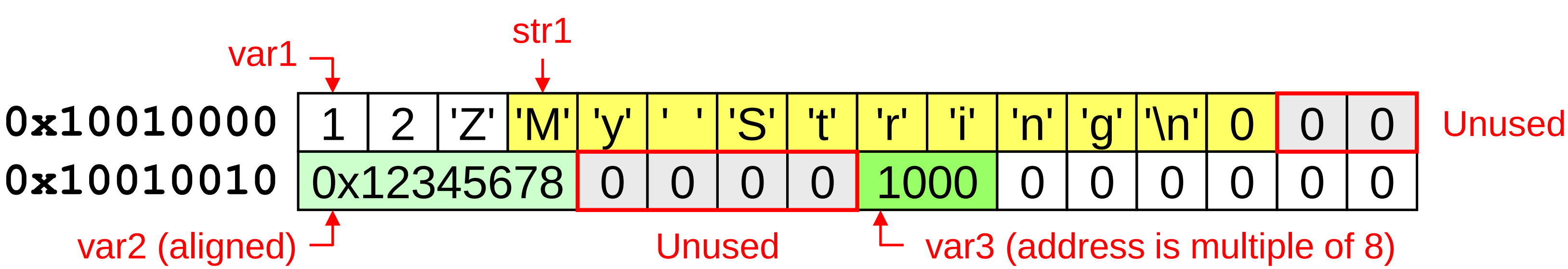
Symbol Table

- 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
```

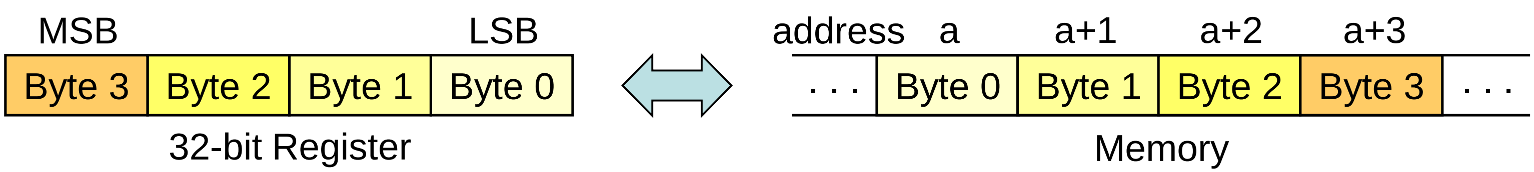
Symbol Table

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

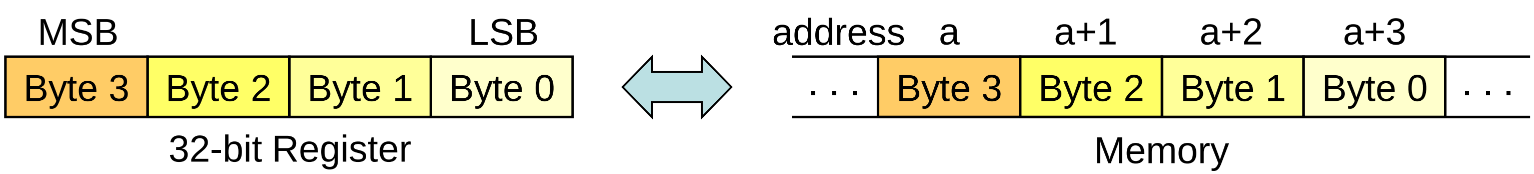


Byte Ordering and Endianness

- Processors can order bytes within a word in two ways
- Little Endian Byte Ordering
  - Memory address = Address of **least significant byte**
  - Example: Intel IA-32, Alpha



- Big Endian Byte Ordering
  - Memory address = Address of **most significant byte**
  - Example: SPARC, PA-RISC



- MIPS can operate with both byte orderings

Next . . .

- The MIPS Instruction Set Architecture
- Introduction to Assembly Language
- Defining Data
- Memory Alignment and Byte Ordering
- **System Calls**

System Calls

- Programs do input/output through system calls
- MIPS provides a special **`syscall`** instruction
  - To obtain services from the operating system
  - Many services are provided 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	Return integer value in $v0
Read Float	6	Return float value in $f0
Read Double	7	Return double value in $f0
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Allocate Heap memory	9	$a0 = number of bytes to allocate Return address of allocated memory in $v0
Exit Program	10	

Syscall Services – Cont'd

Print Char	11	$a0 = character to print
Read Char	12	Return character read in $v0
Open File	13	$a0 = address of null-terminated filename string $a1 = flags (0=read, 1=write, 9=append) $a2 = mode (ignored) Return file descriptor in $v0 (negative if error)
Read from File	14	$a0 = File descriptor $a1 = address of input buffer $a2 = maximum number of characters to read Return number of characters read in $v0
Write to File	15	$a0 = File descriptor $a1 = address of buffer $a2 = number of characters to write Return number of characters written in $v0
Close File	16	$a0 = File descriptor

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

   li     $v0, 10               # Exit program
   syscall
```

Reading and Printing a String

```
################## Data segment ######################
.data
  str: .space  10          # array of 10 bytes
################## Code segment ######################
.text
.globl main
main:                      # main program entry
  la   $a0, str            # $a0 = address of str
  li   $a1, 10             # $a1 = max string length
  li   $v0, 8              # read string
  syscall
  li   $v0, 4              # Print string str
  syscall
  li   $v0, 10             # Exit program
  syscall
```

Program 1: Sum of Three Integers

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#     Input: Requests three numbers.
#    Output: Outputs the sum.
################### Data segment ###################
.data
prompt:   .asciiz     "Please enter three numbers: \n"
sum_msg:  .asciiz     "The sum is: "
################### Code segment ###################
.text
.globl main
main:
      la    $a0,prompt          # display prompt string
      li    $v0,4
      syscall
      li    $v0,5               # read 1st integer into $t0
      syscall
      move  $t0,$v0
```

Sum of Three Integers - Slide 2 of 2

```
li    $v0,5              # read 2nd integer into $t1
syscall
move  $t1,$v0

li    $v0,5              # read 3rd integer into $t2
syscall
move  $t2,$v0

addu  $t0,$t0,$t1        # accumulate the sum
addu  $t0,$t0,$t2

la    $a0,sum_msg        # write sum message
li    $v0,4
syscall

move  $a0,$t0            # output sum
li    $v0,1
syscall

li    $v0,10             # exit
syscall
```

Program 2: Case Conversion

```
# Objective: Convert lowercase letters to uppercase
#     Input: Requests a character string from the user.
#    Output: Prints the input string in uppercase.
################### Data segment ####################
.data
name_prompt:  .asciiz          "Please type your name: "
out_msg:      .asciiz          "Your name in capitals is: "
in_name:      .space 31        # space for input string
################### Code segment ####################
.text
.globl main
main:
      la    $a0,name_prompt   # print prompt string
      li    $v0,4
      syscall
      la    $a0,in_name       # read the input string
      li    $a1,31            # at most 30 chars + 1 null char
      li    $v0,8
      syscall
```

Case Conversion - Slide 2 of 2

```
        la     $a0,out_msg         # write output message
        li     $v0,4
        syscall
        la     $t0,in_name
loop:
        lb     $t1,($t0)
        beqz   $t1,exit_loop       # if NULL, we are done
        blt    $t1,'a',no_change
        bgt    $t1,'z',no_change
        addiu  $t1,$t1,-32         # convert to uppercase: 'A'-'a'=-32
        sb     $t1,($t0)
no_change:
        addiu  $t0,$t0,1           # increment pointer
        j      loop
exit_loop:
        la     $a0,in_name         # output converted string
        li     $v0,4
        syscall
        li     $v0,10              # exit
        syscall
```