MIPS Architecture and Assembly Language

COE 233

Logic Design and Computer Organization

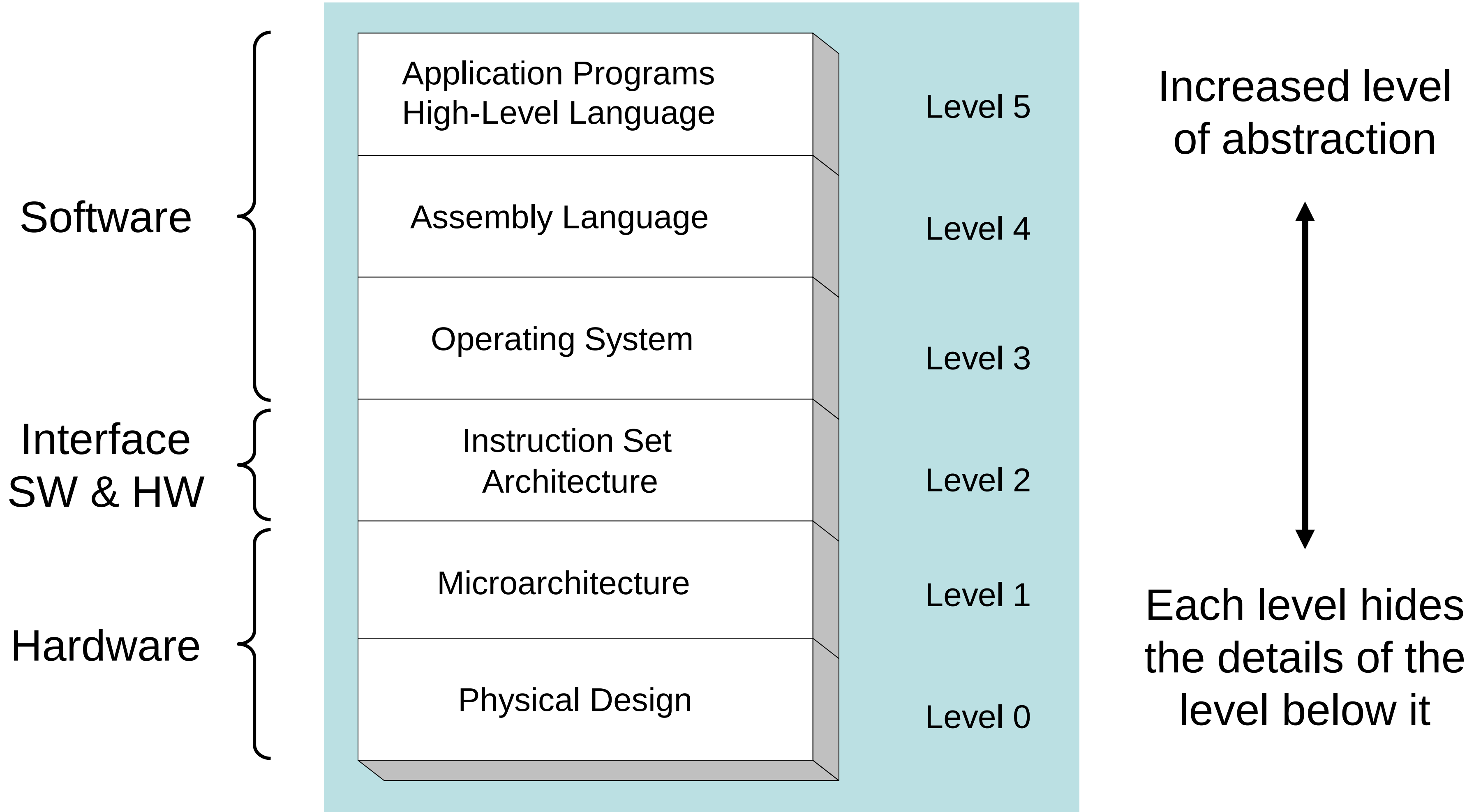
Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- ❖ Programmer's View of a Computer System
- ❖ Assembly, Machine, and High-Level Languages
- ❖ The MIPS Instruction Set Architecture
- ❖ Introduction to Assembly Language Programming
- ❖ Defining Data, Memory Alignment, and Byte Ordering
- ❖ System Calls

Programmer's View of a Computer System



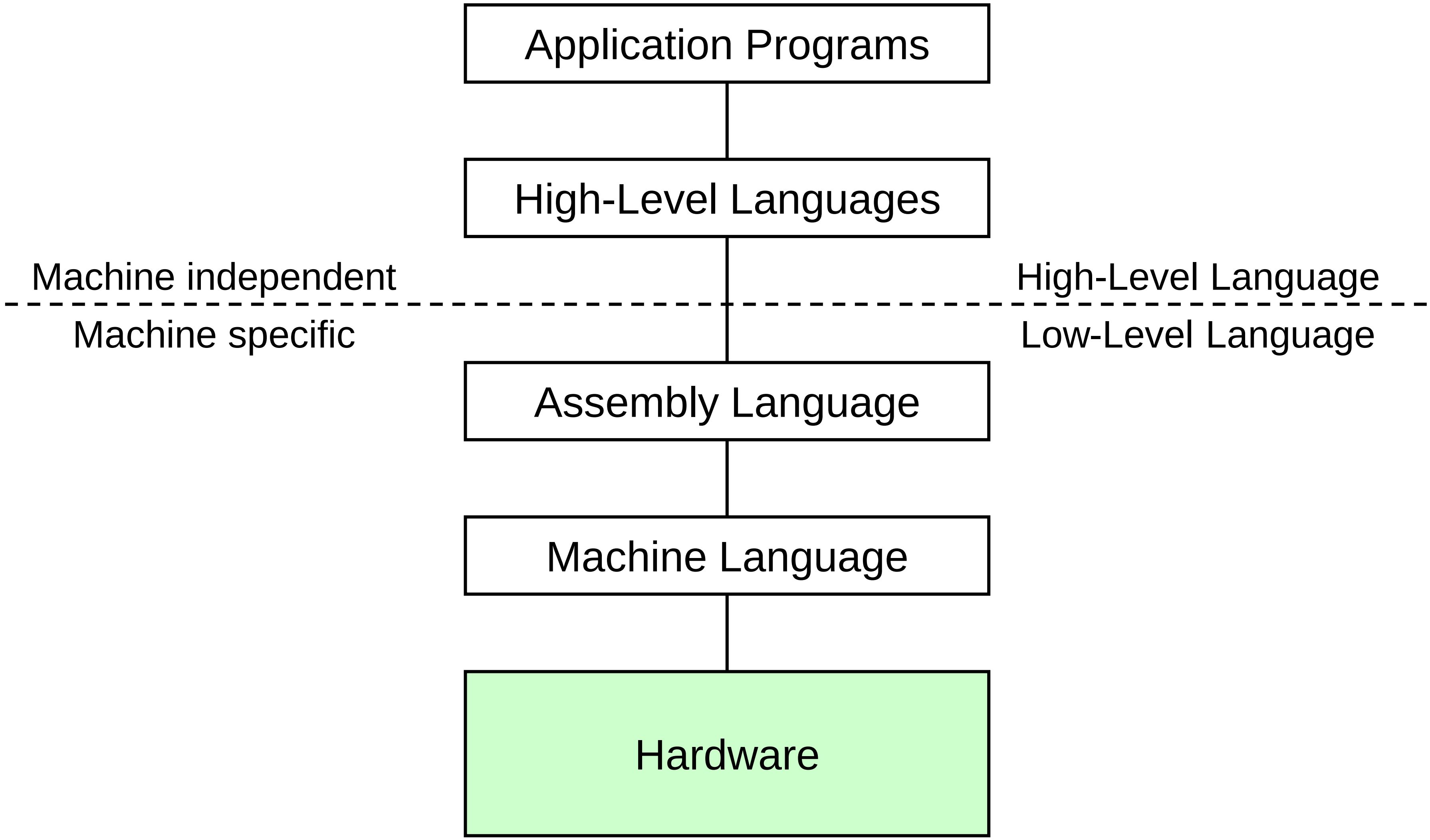
Programmer's View (cont'd)

- Application Programs (Level 5)
  - Written in high-level programming languages
  - Such as Java, C++, Python, . . .
  - Programs are compiled into assembly language level (Level 4)
- Assembly Language (Level 4)
  - Instruction mnemonics (symbols) are used
  - Have one-to-one correspondence to machine language
  - Calls functions written at the operating system level (Level 3)
  - Programs are translated into machine language (Level 2)
- Operating System (Level 3)
  - Provides services to level 4 and 5 programs
  - Translated to run at the machine instruction level (Level 2)

Programmer's View (cont'd)

- Instruction Set Architecture (Level 2)
  - Interface between software and hardware
  - Specifies how a processor functions
  - Machine instructions, registers, and memory are exposed
  - Machine language is executed by Level 1 (microarchitecture)
- Microarchitecture (Level 1)
  - Controls the execution of machine instructions (Level 2)
  - Implemented by digital logic
- Physical Design (Level 0)
  - Implements the microarchitecture at the transistor-level
  - Physical layout of circuits on a chip

A Hierarchy of Languages



What is Assembly Language?

- Low-level programming language for a computer
- One-to-one correspondence with the machine instructions
- Assembly language is specific to a given processor
- Assembler: converts assembly program into machine code
- Assembly language uses:
  - Mnemonics: to represent the names of low-level machine instructions
  - Labels: to represent the names of variables or memory addresses
  - Directives: to define data and constants
  - Macros: to facilitate the inline expansion of text into other code

Translating Languages

Program (C Language):

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

A statement in a high-level language is translated typically into several machine-level instructions

Compiler

MIPS Assembly Language:

```
sll $2,$5, 2
add $2,$4,$2
lw  $15,0($2)
lw  $16,4($2)
sw  $16,0($2)
sw  $15,4($2)
jr  $31
```

Assembler

MIPS Machine Language:

```
00051080
00821020
8C620000
8CF20004
ACF20000
AC620004
03E00008
```

Advantages of High-Level Languages

- Program development is faster
  - High-level statements: fewer instructions to code
- Program maintenance is easier
  - For the same above reasons
- Programs are portable
  - Contain few machine-dependent details
    - Can be used with little or no modification on different machines
  - Compiler translates to the target machine language
  - However, Assembly language programs are not portable

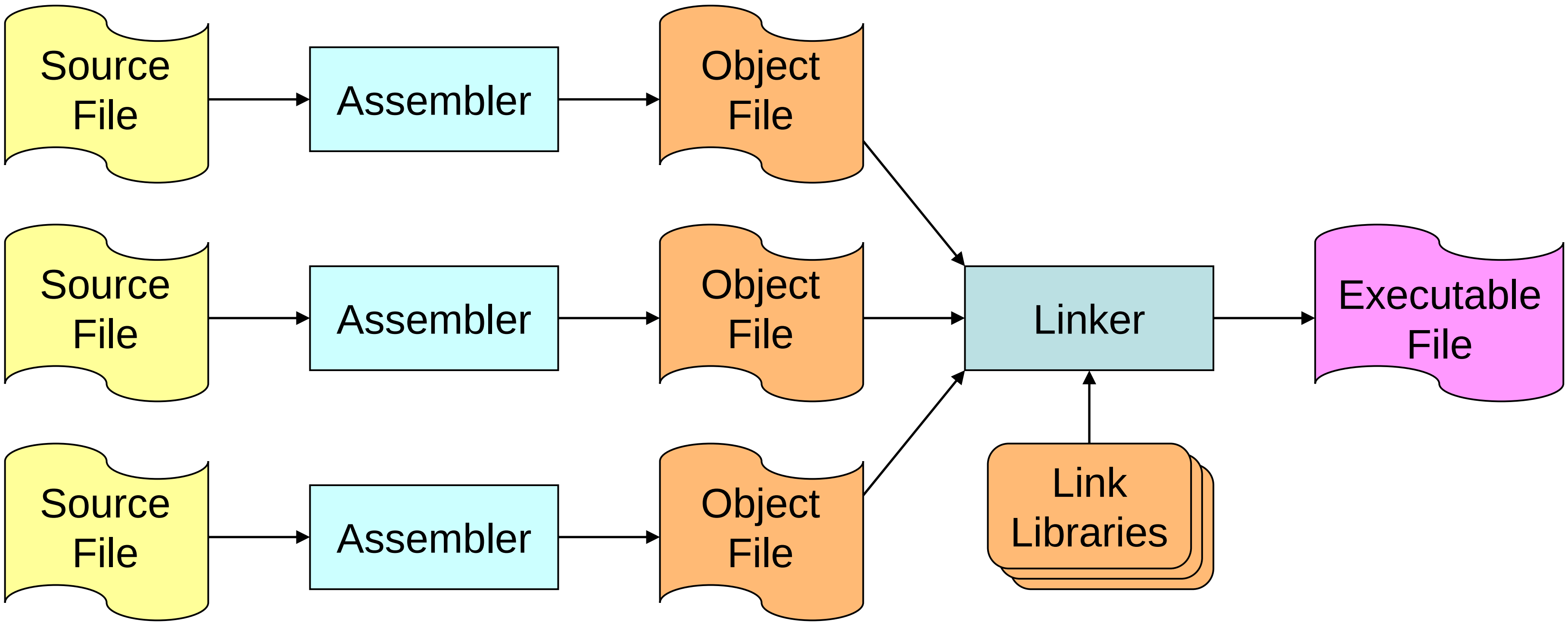
Why Learn Assembly Language?

- Many reasons:
  - Accessibility to system hardware
  - Writing a compiler for a high-level language
  - Deep understanding of the processor architecture
- Accessibility to system hardware
  - Assembly Language is useful for implementing system software
  - Also useful for small embedded system applications
- Programming in Assembly Language is more difficult …
  - Requires deep understanding of the processor architecture
  - However, it is very rewarding to system software designers
  - Adds a new perspective on how programs run on real processors

Assembly Language Programming Tools

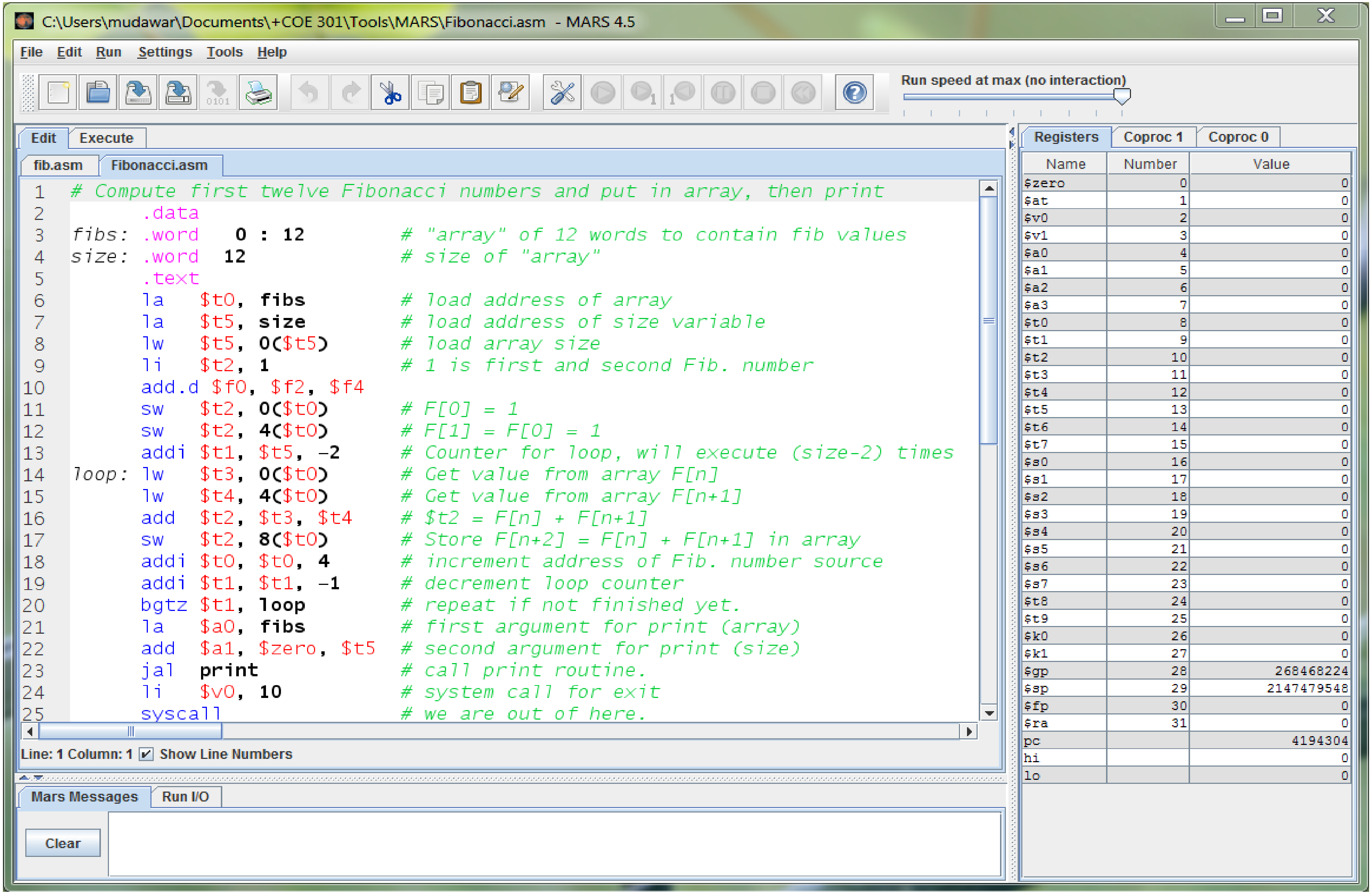
- Editor
  - Allows you to create and edit assembly language source files
- Assembler
  - Converts assembly language programs into object files
  - Object files contain the machine instructions
- Linker
  - Combines object files created by the assembler with link libraries
  - Produces a single executable program
- Debugger
  - Allows you to trace the execution of a program
  - Allows you to view machine instructions, memory, and registers

Assemble and Link Process



- A program may consist of multiple source files
- Assembler translates each source file into an object file
- Linker links all object files together and with link libraries
- The result executable file can run directly on the processor

MARS Assembler and Simulator Tool



MARS Assembler and Simulator Tool

- Simulates the execution of a MIPS program
  - By running the program on the underlying Intel processor
- Editor with color-coded assembly syntax
  - Allows you to create and edit assembly language source files
- Assembler
  - Converts MIPS assembly language programs into object files
- Console and file input/output using system calls
- Debugger
  - Allows you to trace the execution of a program and set breakpoints
  - Allows you to view machine instructions, edit registers and memory
- Easy to use and learn assembly language programming

Next . . .

- ❖ Programmer's View of a Computer System
- ❖ Assembly, Machine, and High-Level Languages
- ❖ The MIPS Instruction Set Architecture
- ❖ Introduction to Assembly Language Programming
- ❖ Defining Data, Memory Alignment, and Byte Ordering
- ❖ System Calls

Instruction Set Architecture (ISA)

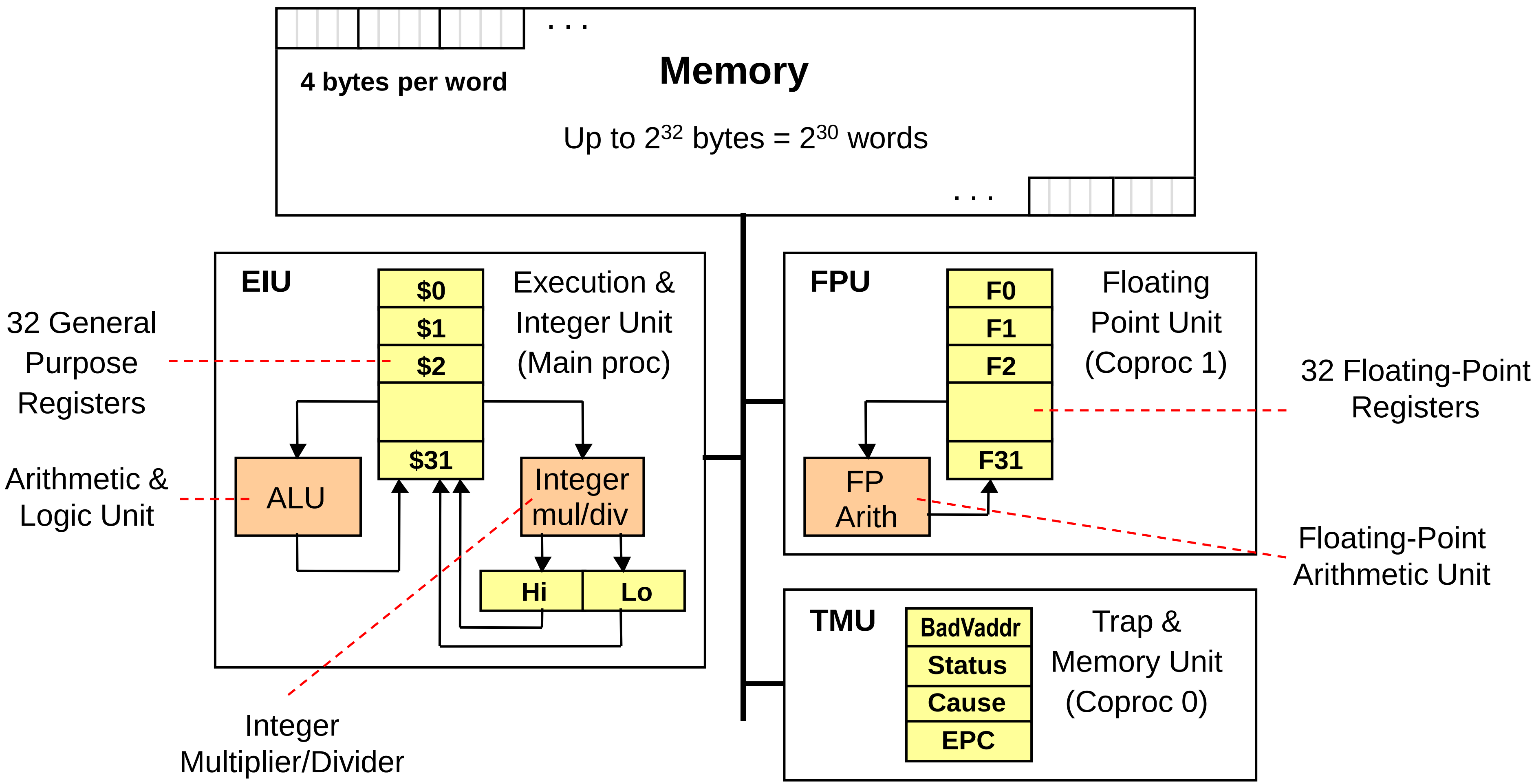
- Interface between software and hardware
- An ISA includes the following …
  - Instructions and Instruction Formats
  - Data Types, Encodings, and Representations
  - Programmable Storage: Registers and Memory
  - Addressing Modes: to address Instructions and Data
  - Handling Exceptional Conditions (like overflow)
- Examples (Versions) Introduced in

Examples	(Versions)	Introduced in
Intel	(8086, 80386, Pentium, Core, ...)	1978
MIPS	(MIPS I, II, …, MIPS32, MIPS64)	1986
ARM	(version 1, 2, …)	1985

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 many RISC architectures, such as ARM and RISC-V
  - Popular, used in many products
    - Silicon Graphics, ATI, Cisco, Sony, etc.
- Alternative to: Intel x86 architecture
  - Known as **Complex Instruction Set Computer (CISC)**

Overview of the MIPS Architecture



MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - All registers are 32-bit wide in the MIPS 32-bit architecture
  - Software defines names for registers to standardize their use
  - Assembler can refer to registers by name or by number ($ notation)

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}

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

Assembly Language 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

- Single-line comment
  - Begins with a hash symbol # and terminates at end of line
- 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

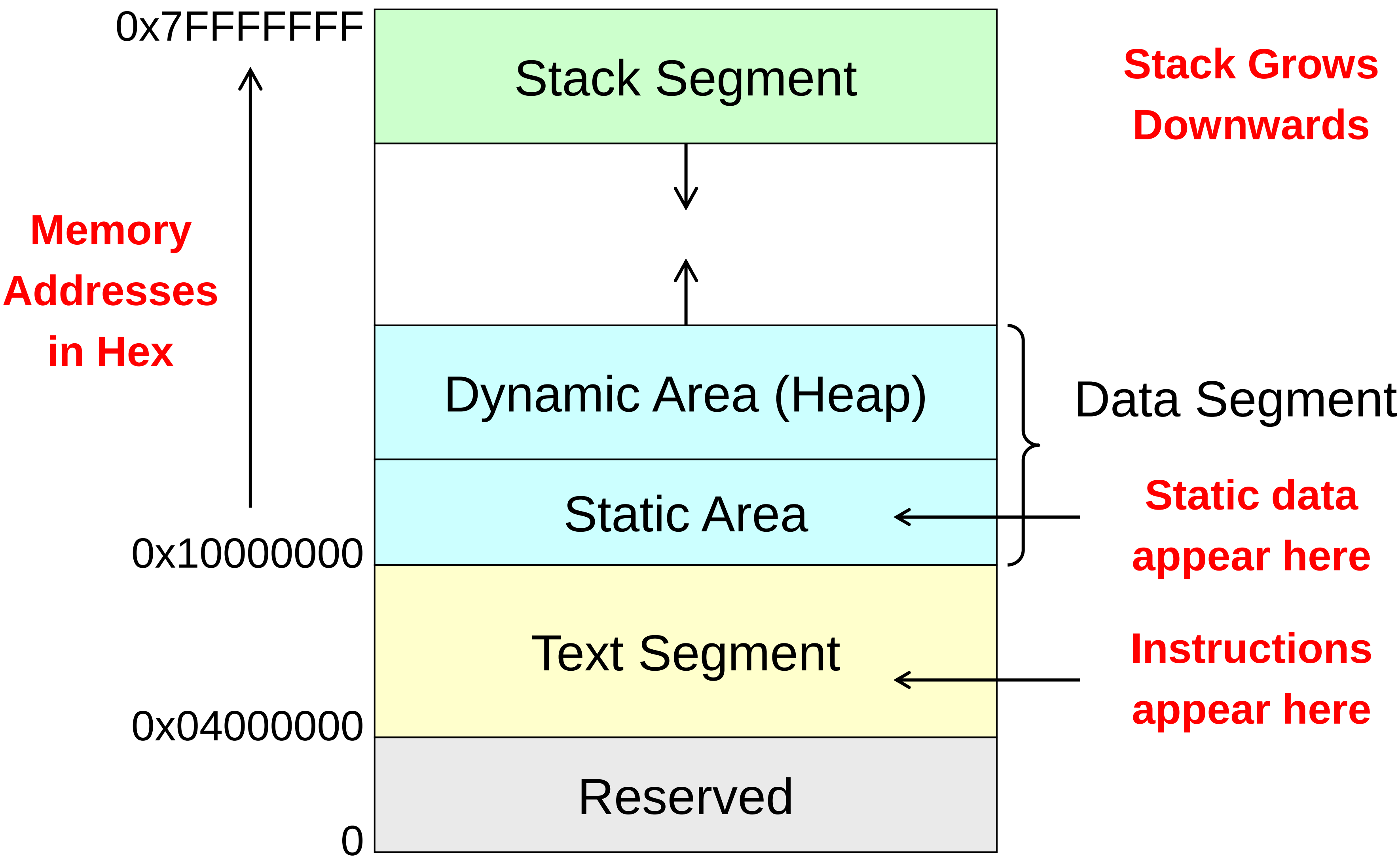
Program Template

```
# 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* function of a program

Layout of a Program in Memory



Next . . .

- ❖ Programmer's View of a Computer System
- ❖ Assembly, Machine, and High-Level Languages
- ❖ The MIPS Instruction Set Architecture
- ❖ Introduction to Assembly Language Programming
- ❖ Defining Data, Memory Alignment, and Byte Ordering
- ❖ System Calls

Data Definition Statement

- The assembler uses directives to define data
- It allocates storage in the static data segment 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

.ASCII and **.ASCIIZ** Directives

- Allocates a sequence of bytes for an ASCII string
- **.ASCIIZ** adds a NULL char (zero byte) at end of string
- Strings are null-terminated, as in the C programming language

.SPACE Directive

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

Examples of Data Definitions

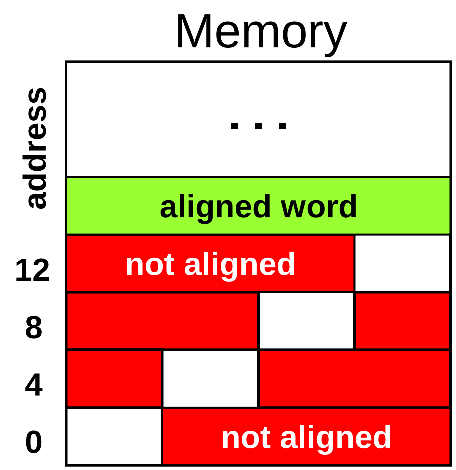
```
.DATA
var1:  .BYTE     'A', 'E', 127, -1, '\n'
var2:  .HALF     -10, 0xffff
var3:  .WORD     0x12345678:100
var4:  .FLOAT    12.3, -0.1
var5:  .DOUBLE   1.5e-10
str1:  .ASCII    "A String\n"
str2:  .ASCIIZ   "NULL Terminated String"
array: .SPACE    100
```

var3: Array of 100 words Initialized with the same value

array: 100 bytes (not initialized)

Memory Alignment

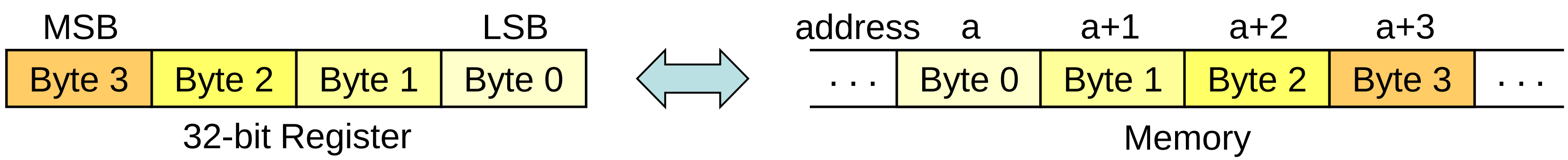
- Memory is viewed as an **addressable array of bytes**
- **Byte Addressing**: address points to a byte in memory
- However, words occupy 4 consecutive bytes in memory
  - MIPS instructions and integers occupy 4 bytes
- **Memory Alignment**:
  - Address must be multiple of size
  - Word address should be a multiple of **4**
  - Double-word address should be a multiple of **8**



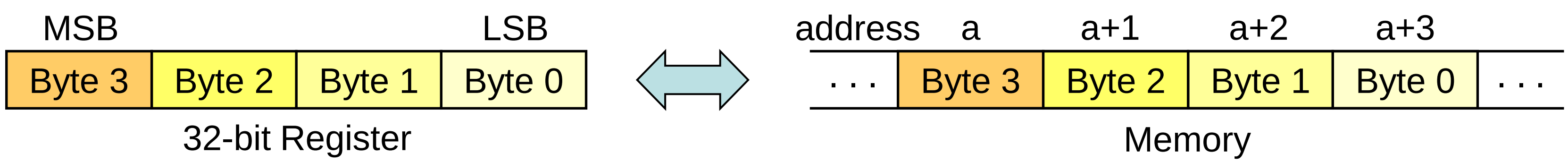
- **`.ALIGN n`** directive
  - Aligns the next data definition on a 2^n byte boundary
  - Forces the address of next data definition to be multiple of 2^n

Byte Ordering (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



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



- MIPS can operate with both byte orderings

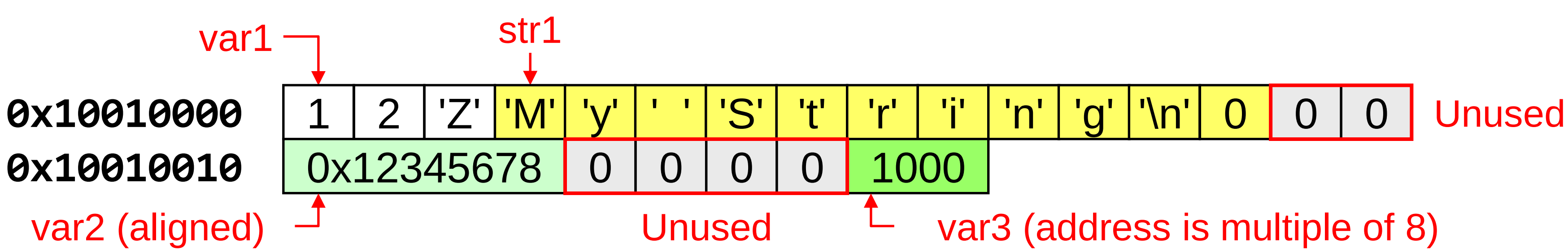
Symbol Table

- Assembler builds a **symbol table** for labels
  - 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



System Calls

- Programs do input/output through system calls
- The MIPS architecture provides a **`syscall`** instruction
  - To obtain services from the operating system
  - The operating system handles all system calls requested by program
- Since MARS is a simulator, it simulates the **`syscall`** services
- To use the **`syscall`** 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-only, 1 = write-only) $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
```

Sum of Three Integers

```
# Sum of three integers
# Objective: Computes the sum of three integers.
# Input: Requests three numbers, Output: 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 - (cont'd)

```
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
```