Assembly Language Laboratory 1

CS/EE 320 Computer Organization and Assembly Language

Shahid Masud

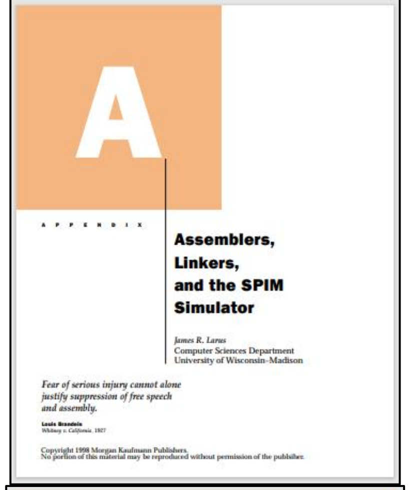
Spring 2025

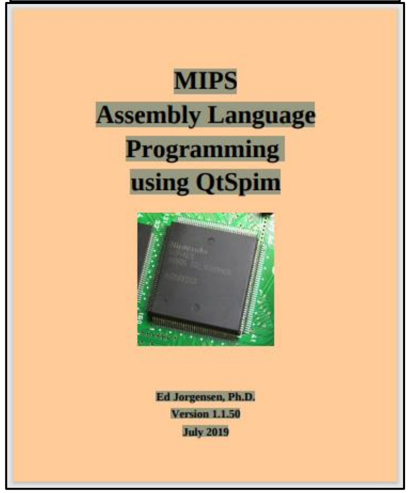
Topics

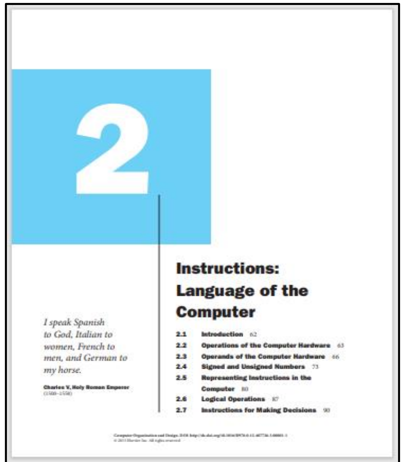
- Resources we will use in lab
- Where does Assembly Language fit in software and hardware?
- What does an Assembler do? and Linker? – The HW / SW Interface
- Some Assemblers and Simulators for MIPS Architecture
- Number and Data Formats in MIPS Assembly
- Program and Data Storage in Memory

Resources We Will use

- Appendix A: Assemblers, Linkers and the SPIM Simulator, written by J. R. Larus, Microsoft Research, part of Patterson and Hennessy book
- Book: MIPS Assembly Language Programming using QtSpim, by Ed Jorgensen, Version 1.1.50, July 2019 (available online)
- Chap 2, P&H Book, “Instructions: Language of the Computer”

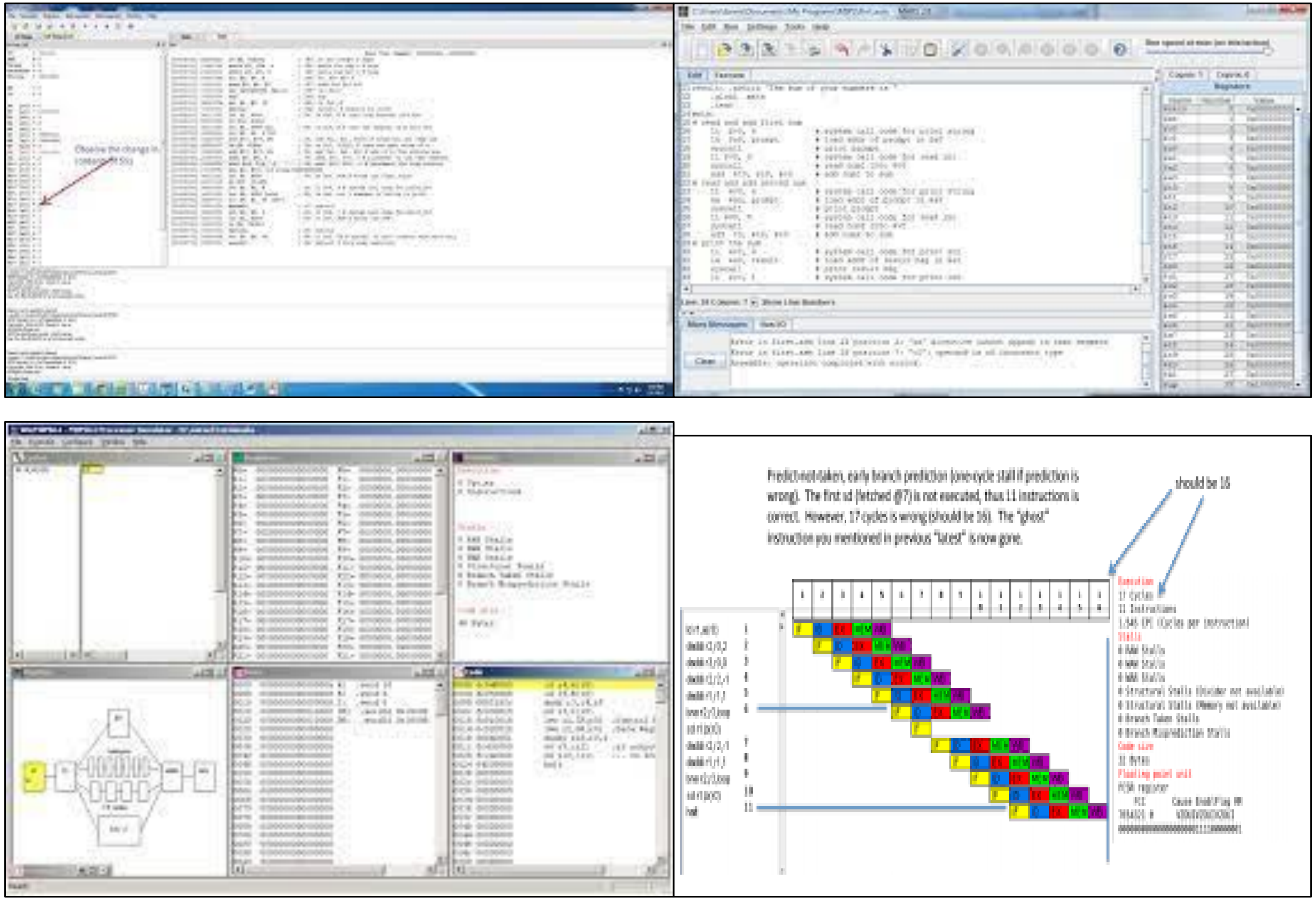




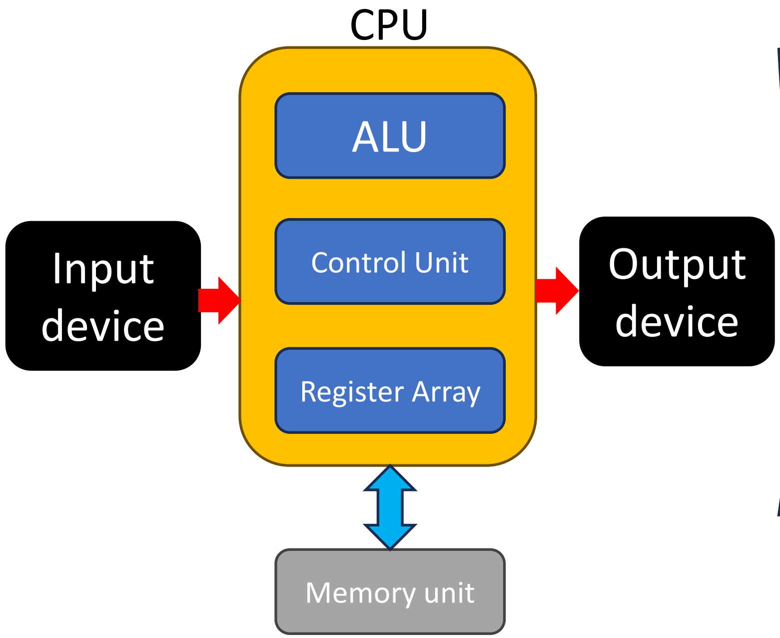


Lab Resources - Software

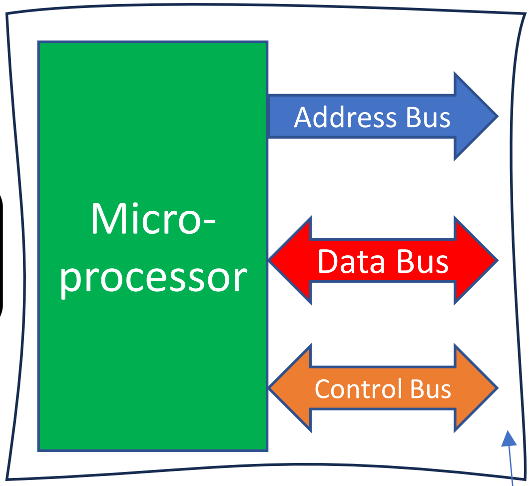
- QTSpim or MARS Assembler for Assembly Language Programming
- Edu64MIPS or WinMIPS64 for Architecture Simulations



How to Talk to a Microprocessor?



Typical Von-Neumann Computer Architecture



µP talks through buses

Look at the direction of buses

- Computers work in Digital World
- Computers using Binary Numbers to represent any information, program, variables, data, etc.
- Binary numbers have only two values ‘0’ and ‘1’ (and a third state ‘Z’ for open circuit)
- Only Binary data ‘1’ and ‘0’ is stored in memory in the form of computer program or data
- Microprocessors ONLY understand Binary numbers
- The program stored in memory in terms of ‘1’ and ‘0’ is called “Machine Language”
- Machine Language is specific to a microprocessor

Mapping Computer Programming Language to Microprocessor

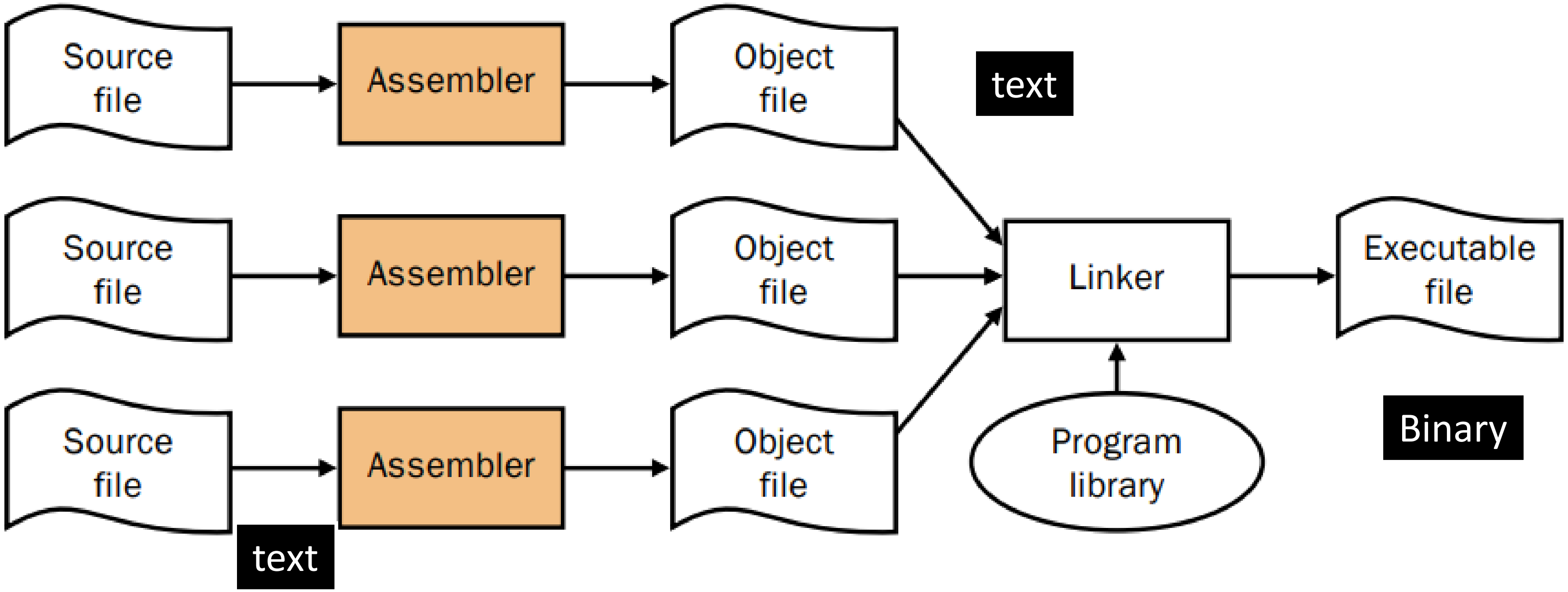
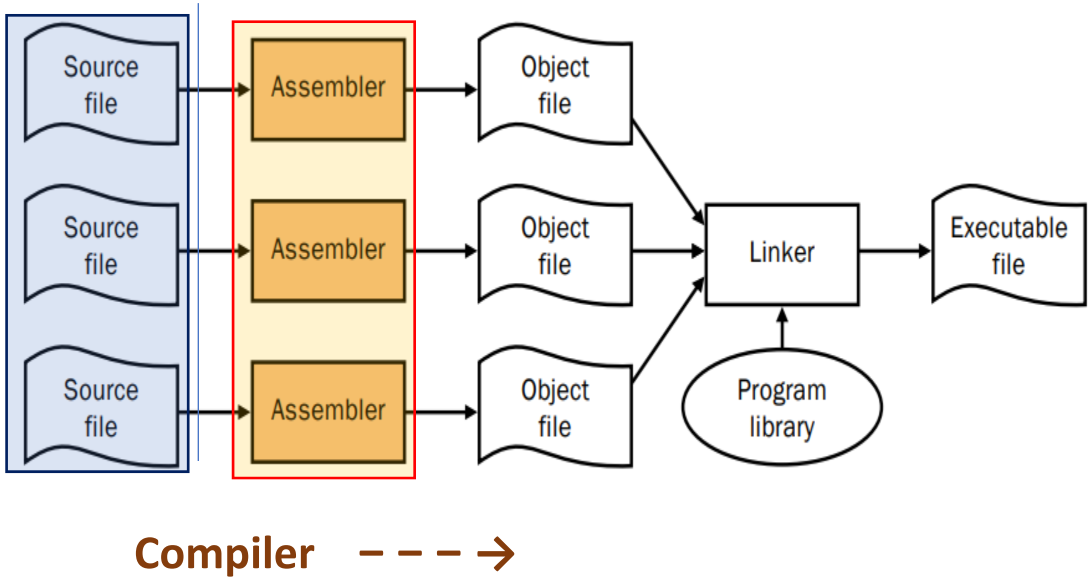


FIGURE A.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Assembly language is used when the (i) execution speed or (ii) memory storage of programs is very important

The Compiler Process



- Source File: In C, Python, Fortran, Pascal, etc. (why not Java?)
- Compiler breaks each line of HLL code to several lines of Assembly Language code
- Assembler: Converts Assembly code programs to Object File
- Linker: Combines all Object Files to produce executable machine code for full program

What does an Assembler do?

- An Assembler translates a file of Assembly language statements into an Object file of binary machine instructions and binary data.
- The translation process has two major parts:
  - The first step is to find memory locations with labels so the relationship between symbolic names and addresses is known when instructions are translated
  - The second step is to translate each assembly statement by combining the numeric equivalents of opcodes, register specifiers, and labels into legal instructions
- The Assembler produces an output file, called an object file, which contains the machine instructions, data, and bookkeeping information
- Program library contains pre-written routines

What does a linker do?

Linker is a tool that merges the object files

Linker performs following tasks:

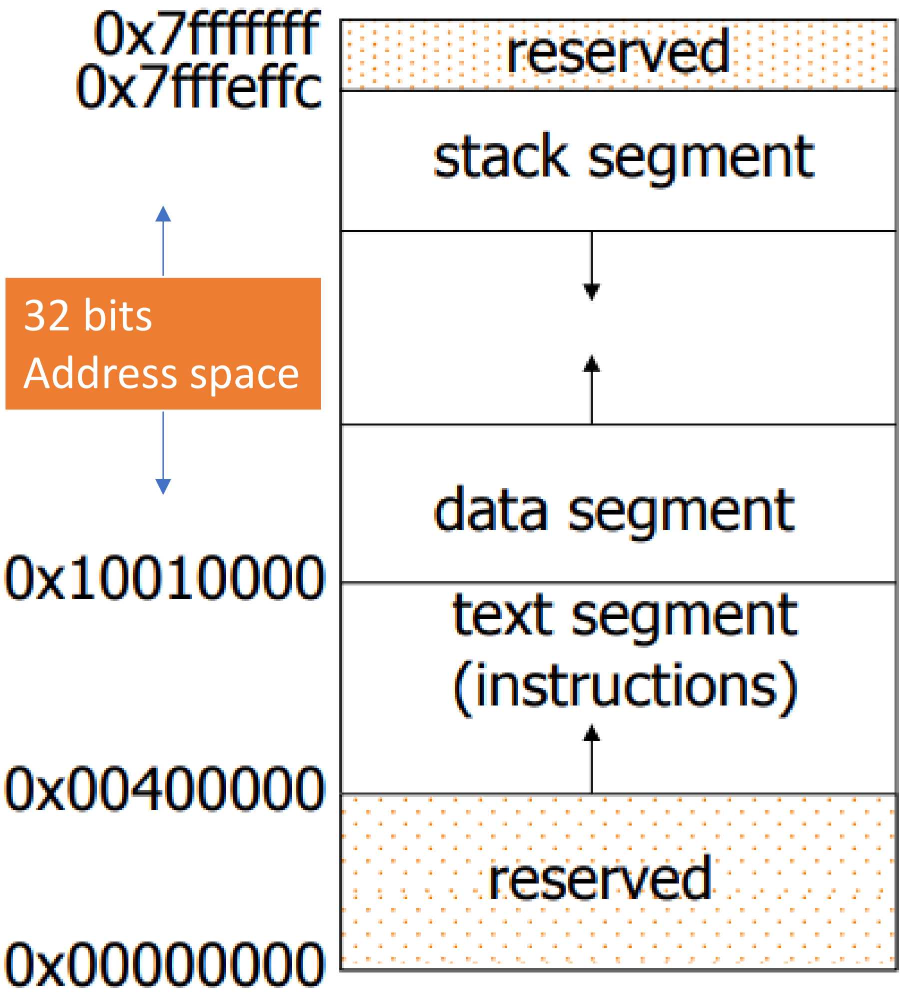
■ Searches the program libraries to find library routines used by the program

■ Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references

■ Resolves references among files

■ Ensures that a program contains no undefined labels

Memory Mapping in MIPS



- **The first part**, near the bottom of the address space (starting at address 400000hex), is the text segment, which holds the program's instructions.
- **The second part**, above the text segment, is the data segment, which is further divided into two parts.
  - Static data (starting at address 10000000hex) contains objects whose size is known to the compiler and whose lifetime—the interval during which a program can access them—is the program's entire execution.
  - Immediately above static data is dynamic data. This data, as its name implies, is allocated by the program as it executes.
- **The third part**, the program stack segment, resides at the top of the virtual address space (starting at address 7fffffffhex). Like dynamic data, the maximum size of a program's stack is not known in advance. As the program pushes values on the stack, the operating system expands the stack segment down, towards the data segment.

Registers in MIPS

a.k.a. Register File

Register Name	Register Number	Register Usage
$zero	$0	Hardware set to 0
$at	$1	Assembler temporary
$v0 - $v1	$2 - $3	Function result (low/high)
$a0 - $a3	$4 - $7	Argument Register 1
$t0 - $t7	$8 - $15	Temporary registers
$s0 - $s7	$16 - $23	Saved registers
$t8 - $t9	$24 - $25	Temporary registers
$k0 - $k1	$26 - $27	Reserved for OS kernel
$gp	$28	Global pointer
$sp	$29	Stack pointer
$fp	$30	Frame pointer
$ra	$31	Return address

$t temporary registers – not preserved across procedure calls
$s saved registers – preserved across system calls
$a, $v – used in arithmetic functions

The MIPS CPU contains **32 general-purpose** registers that are **numbered 0–31**.

- Register **$0 always contains the hardwired value 0**.
- Registers **$at (1), $k0 (26), and $k1 (27) are reserved for the assembler and operating system** and should not be used by user programs or compilers.
- Registers **$a0 – $a3 (4–7)** are used to pass the first four arguments to routines (remaining arguments are passed on the stack).
- Registers **$v0 and $v1 (2, 3)** are used to return values from functions.
- Registers **$t0–$t9 (8–15, 24, 25**) are caller-saved registers that are used to hold temporary quantities that need not be preserved across calls.
- Registers **$s0–$s7 (16–23)** are callee-saved registers that hold long lived values that should be preserved across calls.
- Register **$gp (28)** is a **global pointer** that points to the middle of a 64K block of memory in the static data segment.
- Register **$sp (29)** is **the stack pointer**, which points to the **first free location on the stack**.
- Register **$fp (30) is the frame pointer**.
- **The jal instruction writes register $ra (31), return address from a procedure call**.

Binary, Decimal and Hexadecimal Numbers

Numbering System		
System	Base	Digits
Binary	2	0, 1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Binary to Hex and Back

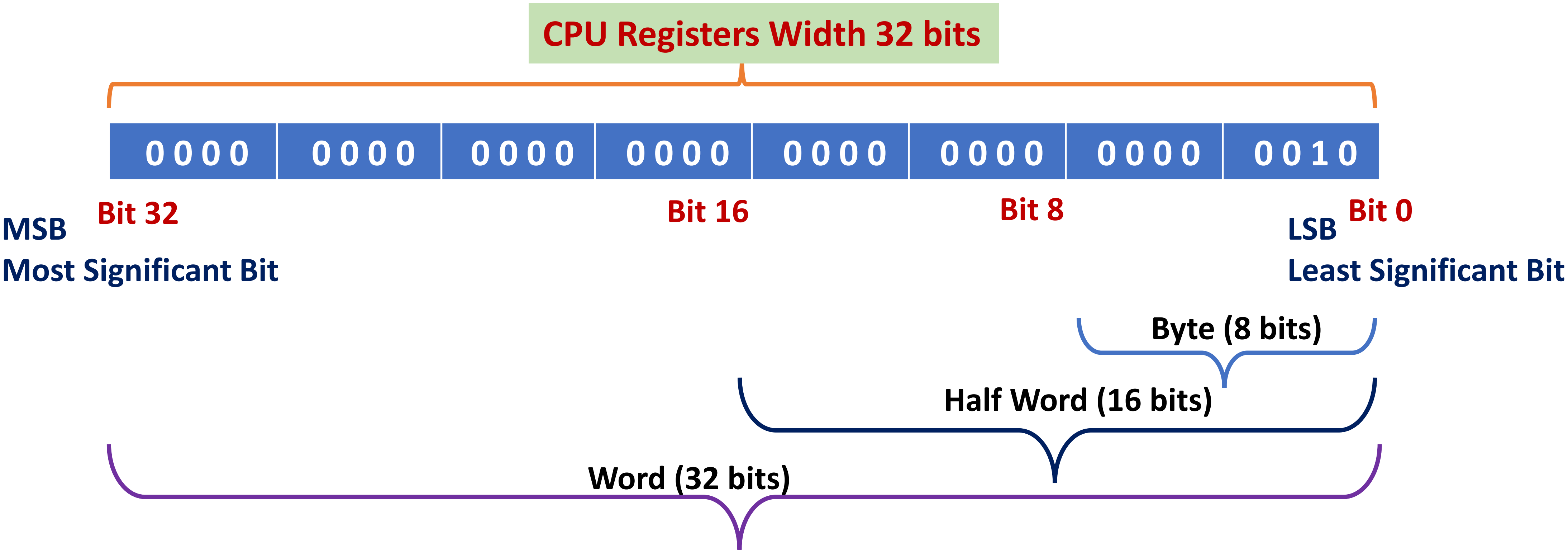
Hex Representation is Compact and Clear

A B C 4 5 6 7 F

Binary Representation is
Cluttered and prone to confusion

1010 1011 1100 0100 0101 0110 0111 1111

Register Usage in MIPS



Size	Size	Unsigned Int Range	Signed Int Range
Byte	2^8	0 to 255	-128 to +127
Half Word	2^{16}	0 to 65,535	-32,768 to +32,767
Word	2^{32}	0 to 4,294,967,295	-2,147,483,648 to +2,147,483,647

The Scale in Computing

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

FIGURE 1.1 The 2^X vs. 10^Y bytes ambiguity was resolved by adding a binary notation for all the common size terms. In the last column we note how much larger the binary term is than its corresponding decimal term, which is compounded as we head down the chart. These prefixes work for bits as well as bytes, so *gigabit* (Gb) is 10^9 bits while *gibibits* (Gib) is 2^{30} bits.

External Memory vs CPU Registers

Remember: blocks

4,294,967,295

Location	Memory Data 8-bits
2^{32}-1	1110 1001
2^{32}-2	1010 0111
….	….
….	….
7	….
6	….
5	….
4	….
3	0100 1001
2	1001 1100
1	0101 1011
0	0000 0010

Word Boundary

Register File

Location	32-bit Registers Data
R0	0x 499C 5B02
R1	0x 025B 9C49
R2	
….	
….	
….	
….	
R29	
R30	
R31	

Byte Order: Big or Little – Endian

is called its *byte order*. MIPS processors can operate with either *big-endian* or *little-endian* byte order. For example, in a big-endian machine, the directive `.byte 0, 1, 2, 3` would result in a memory word containing

Byte #			
0	1	2	3

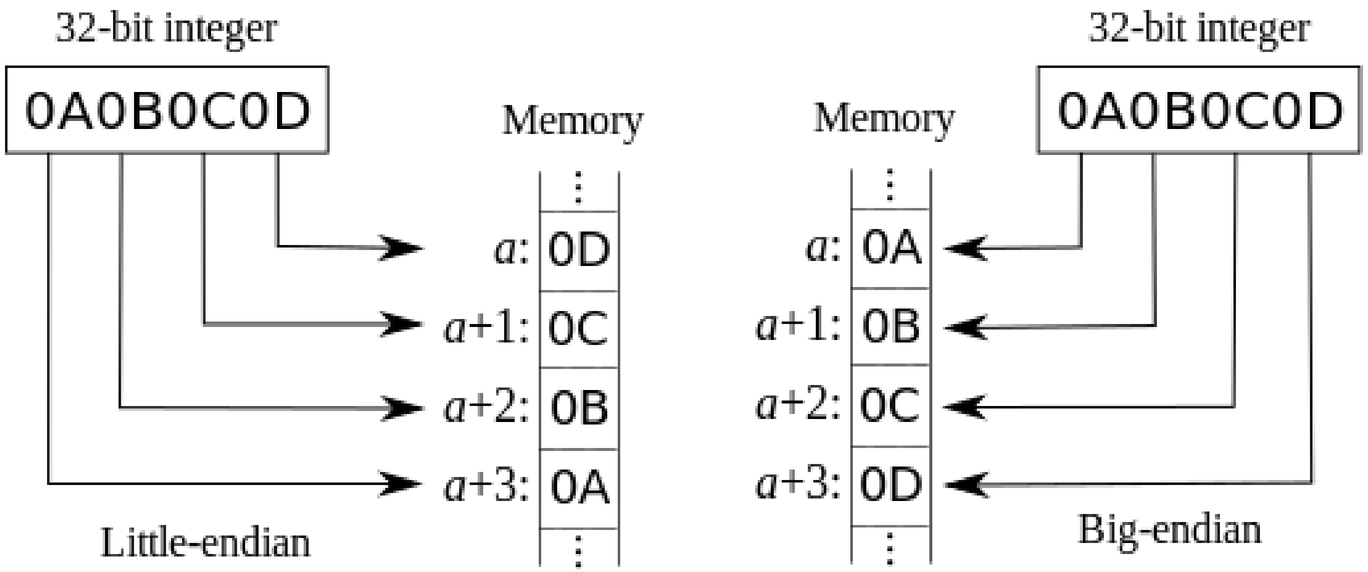
Big Endian => Lowest Significant Byte in Word is
Stored in Highest Byte Location (4)

while in a little-endian machine, the word would contain

Byte #			
3	2	1	0

Little Endian => Lowest Significant Byte in Word is
Stored in Lowest Byte Location (0)

Memory Address in Big Endian and Little Endian



Memory Map in Little Endian

For example, assuming the following declarations:

```
num1:    .word    42
num2:    .word    5000000
```

Recall that 42_{10} in hex, word size, is 0x0000002A and $5{,}000{,}000_{10}$ in hex, word size, is 0x004C4B40.

For a little-endian architecture, the memory picture would be as follows:

variable name	value	address
	?	0x100100C
	00	0x100100B
	4C	0x100100A
	4B	0x1001009
Num2 →	40	0x1001008
	00	0x1001007
	00	0x1001006
	00	0x1001005
Num1 →	2A	0x1001004
	?	0x1001003

Based on the little-endian architecture, the LSB is stored in the lowest memory address and the MSB is stored in the highest memory location.