INTRODUCTION TO COMPUTER ORGANIZATION

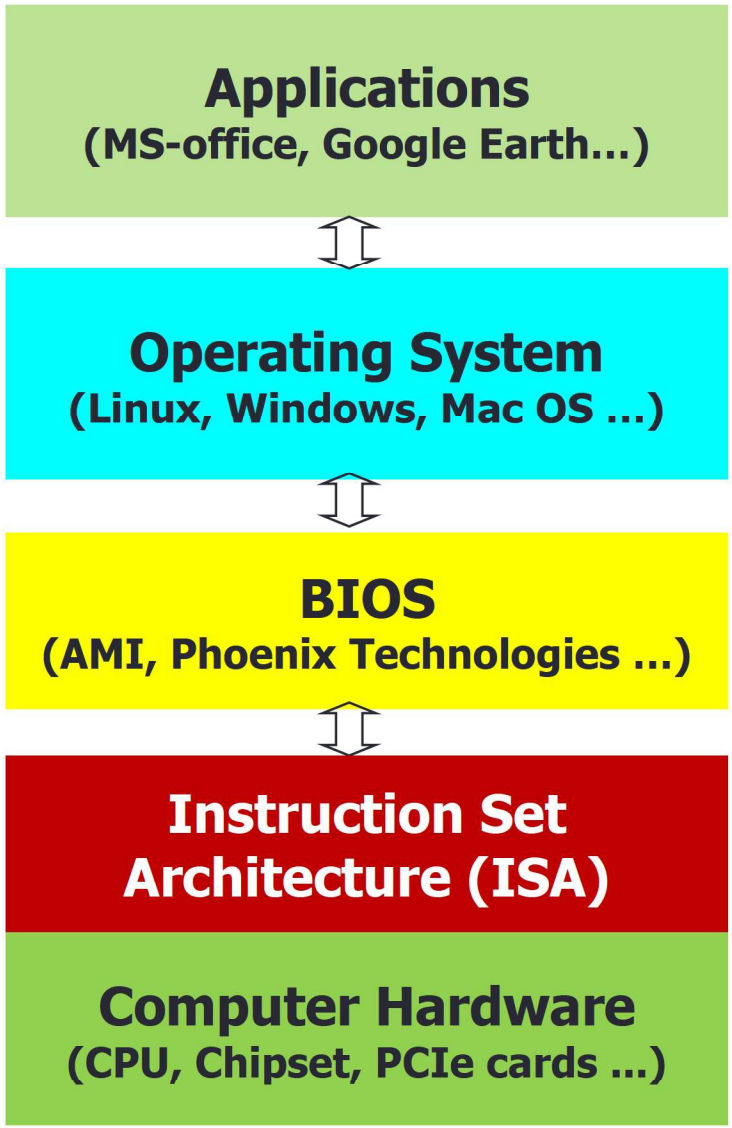
CCIT4026

Chapter 2: Instruction Set Design (Part 1)

Outline

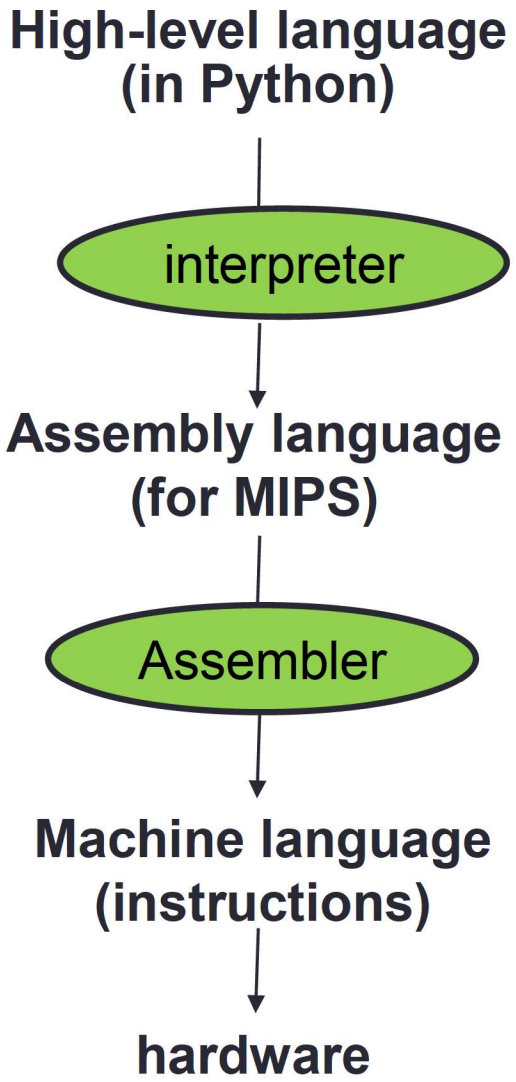
- Hardware/ Software
- Instruction Set Architecture (ISA)
- MIPS architecture (A brief Introduction)

Hardware/Software



- Application software
  - Written in high-level language
- System software
  - Compiler
    - Translates **code written in high-level language** to **machine code**
  - Operating System
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
  - BIOS (Basic Input / Output System)
- ISA
  - Interface between hardware and low-level software
- Hardware
  - Processor, memory, I/O controllers

Hardware/Software

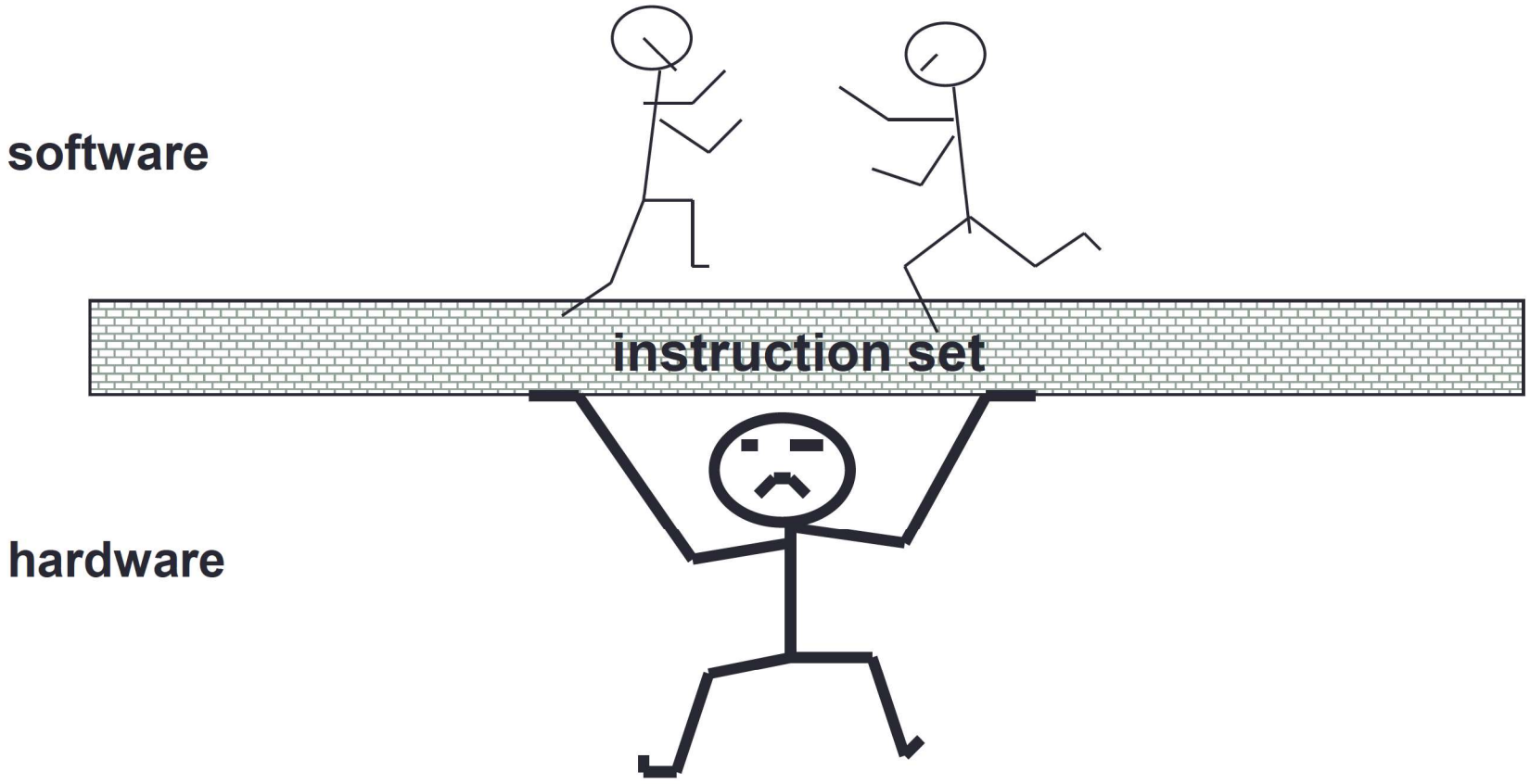


num[300] = num[300] + h

lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)

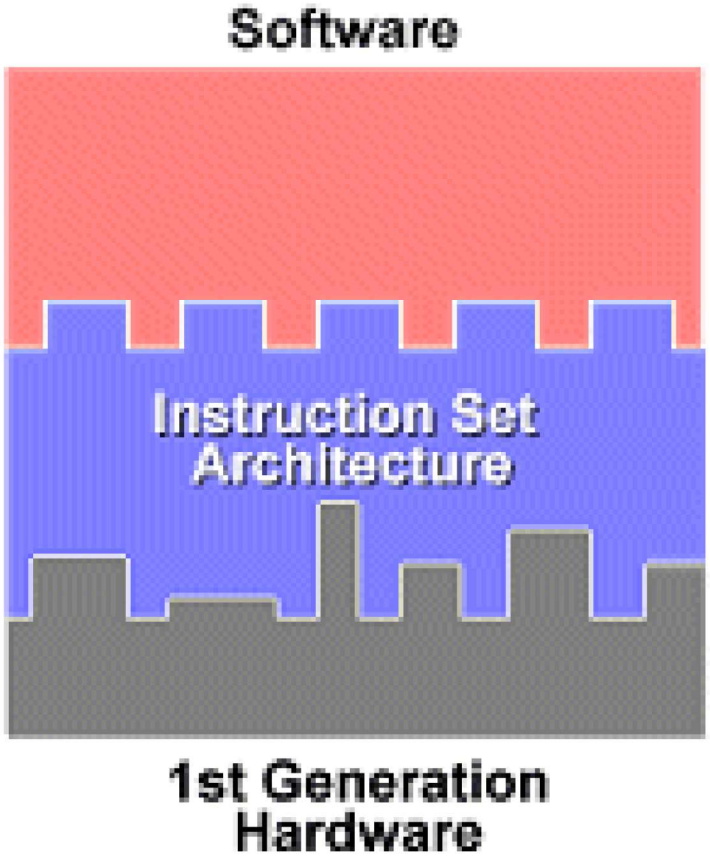
1000110100101000000010010110000
0000001001001000010000000100000
1010110100101000000010010110000

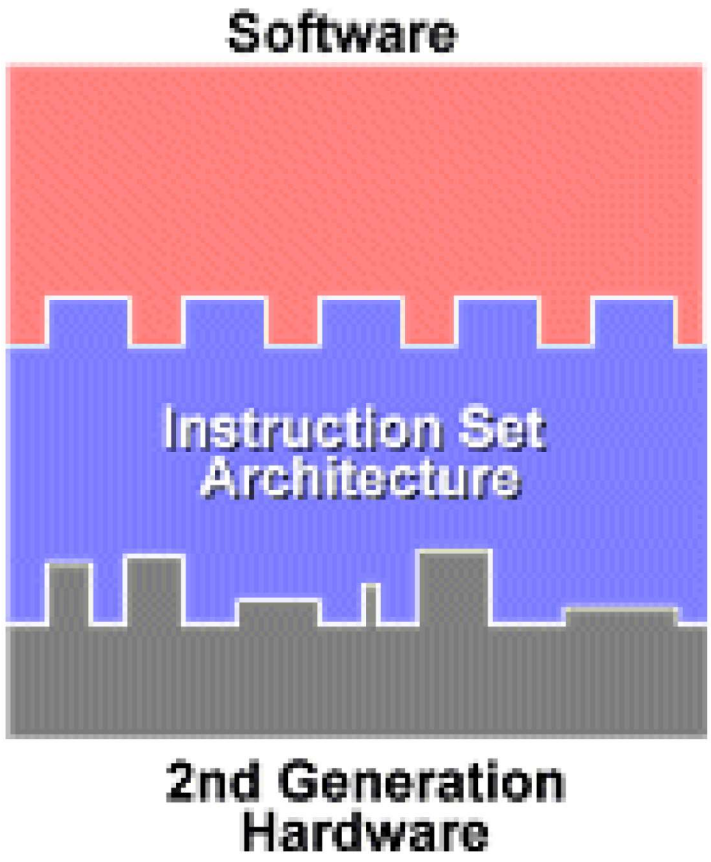
Instruction Set Architecture (ISA)



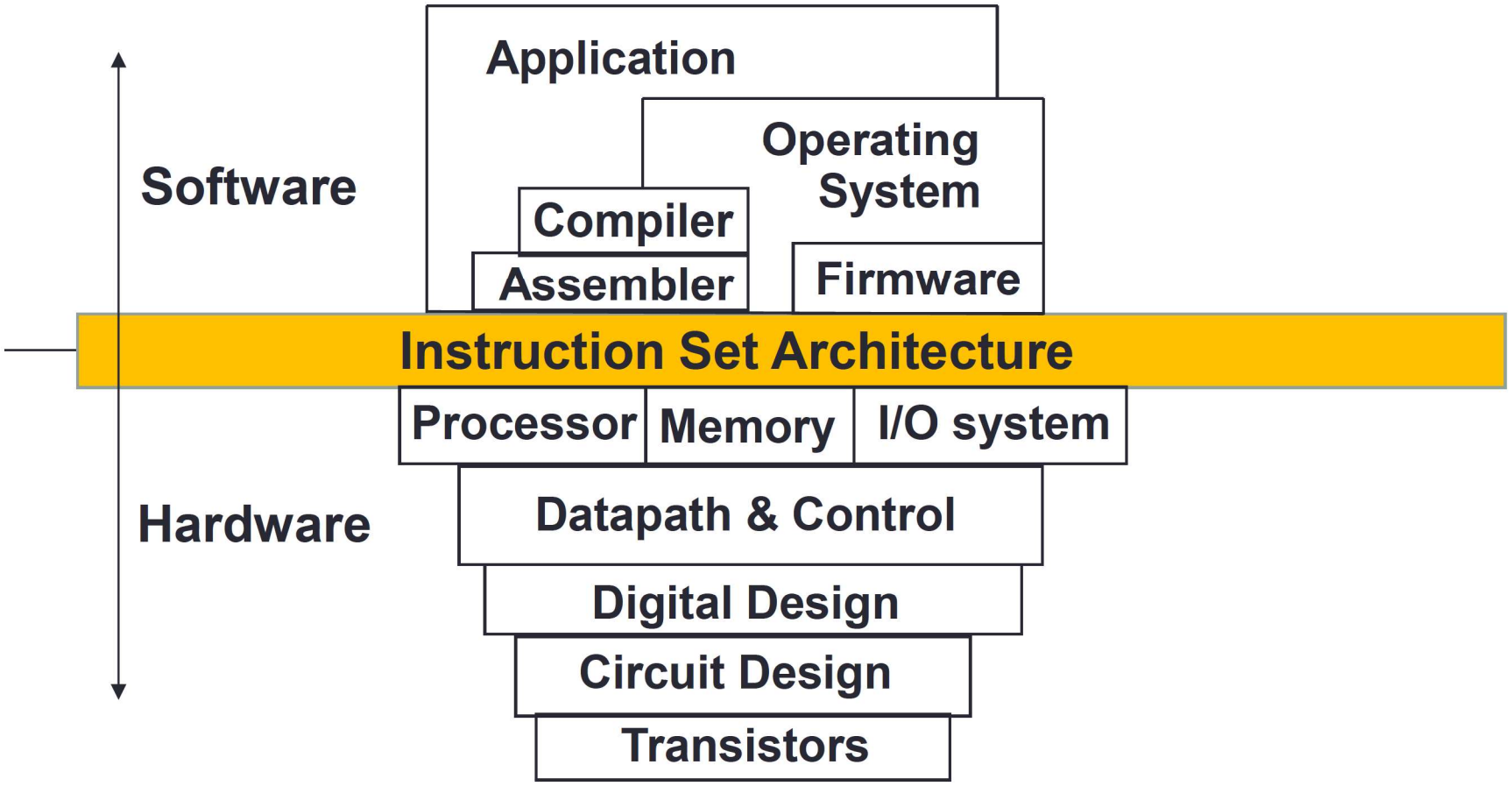
An **instruction set architecture** provides an **abstract interface** between hardware and low-level software.

Instruction Set Architecture (ISA)





Instruction Set Architecture (ISA)



Instruction Set Architecture (ISA)

Instruction

- The low level language that is used by the computer.
- Machine language is built up from discrete statements or called *instructions*.

Instruction set

- The collection of instructions is called instruction set.
- Different processors have different instruction sets, but they can also share a common instruction set. E.g. Intel Pentium and AMD Athlon implement nearly identical versions of x86 instruction set.

Instruction Set Architecture (ISA)

- The ISA serves as the boundary between software and hardware.
- It is part of the computer architecture that includes the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.

Instruction Set Architecture (ISA)

- RISC and CISC

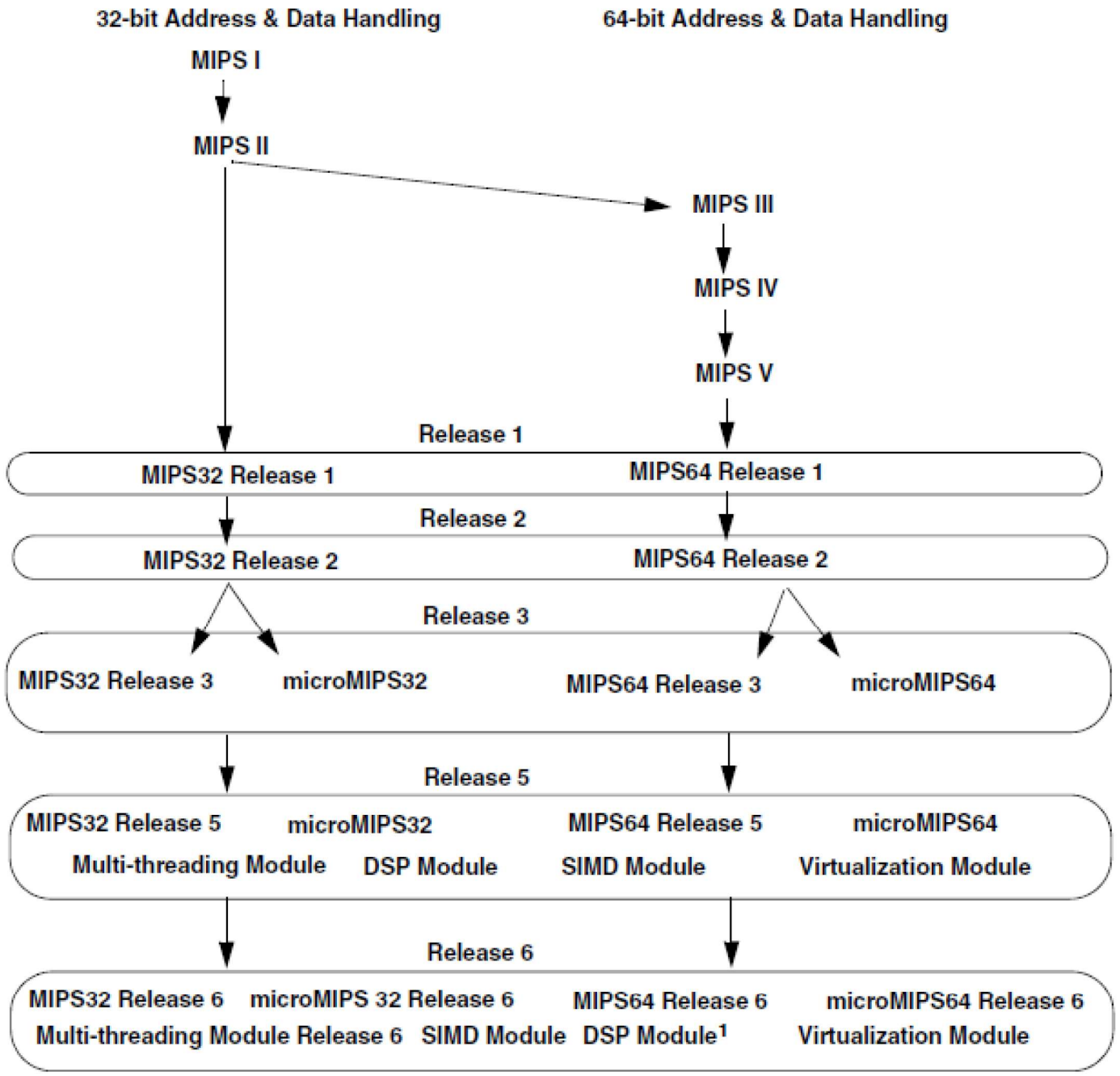
- **Reduced Instruction Set Computer (RISC)**
  - small number of simple instructions
  - example: MIPS

- **Complex Instruction Set Computers (CISC)**
  - large number of instructions
  - example: Intel's x86

MIPS

- **Microprocessor without Interlocked Pipeline Stages** (**MIPS**) is a RISC instruction set architecture (ISA).
- Developed by MIPS Computer Systems, Inc. (MIPS was acquired by Wave Computing in June 2018) (https://wavecomp.ai)
- Multiple revisions of the MIPS instruction set:
  - MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, **MIPS32**, and **MIPS64**
- RISC architecture has been used in:
  - Android-based systems, Apple iPhone, Microsoft Windows Phone, Playstation, SPARC, Fujitsu, IBM's supercomputers, etc.
- Data formats:
  - byte(8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit)

MIPS


32-bit Address & Data Handling
64-bit Address & Data Handling
MIPS I
MIPS II
MIPS III
MIPS IV
MIPS V
Release 1
MIPS32 Release 1
MIPS64 Release 1
Release 2
MIPS32 Release 2
MIPS64 Release 2
Release 3
MIPS32 Release 3
microMIPS32
MIPS64 Release 3
microMIPS64
Release 5
MIPS32 Release 5
microMIPS32
MIPS64 Release 5
microMIPS64
Multi-threading Module
DSP Module
SIMD Module
Virtualization Module
Release 6
MIPS32 Release 6
microMIPS 32 Release 6
MIPS64 Release 6
microMIPS64 Release 6
Multi-threading Module Release 6
SIMD Module
DSP Module[1]
Virtualization Module

CPU data formats

- **Bit** : binary digit, either 0 or 1

- **Byte**: 8-bit
  - Binary 00000000_2 to 11111111_2
  - Decimal 0_{10} to 255_{10}
  - Octal 000_8 to 0377_8 $(011 \mid 111 \mid 111_2)$
  - Hexadecimal 00_{16} to FF_{16}

- **Halfword**: 16-bit

- **Word**: 32-bit

- **Doubleword**: 64-bit

MIPS Instructions

Instruction Categories

- Load and Store
- Computational
- Jump and Branch
- Miscellaneous
- Coprocessor

Instruction Operands

- An instruction needs a physical location from which to retrieve binary operands
- An instruction retrieves operands from:
  - Register
  - Memory
  - Immediate

Register

What are registers?

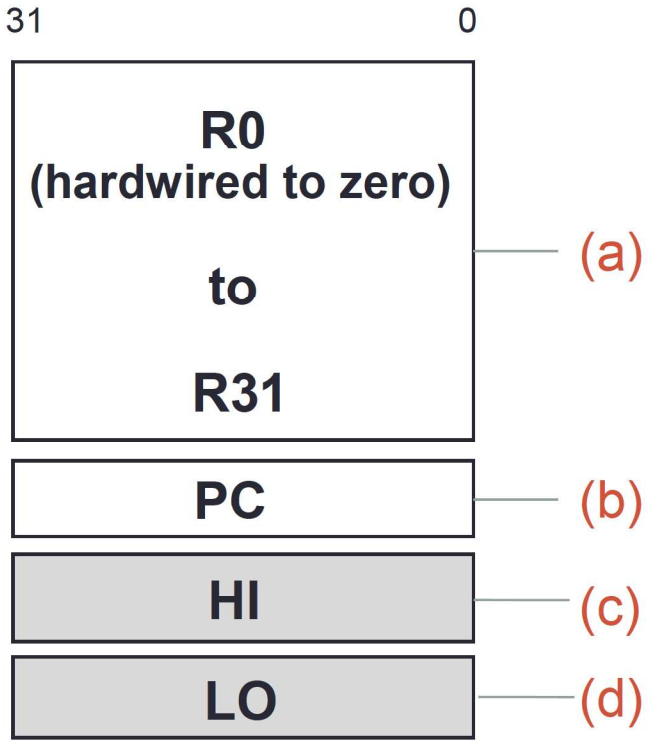
- Fast temporary storage located inside the processor used to hold data, instructions and addresses.
- Size of a register in the MIPS architecture is 32-bit (MIPS32) or 64-bit (MIPS64).

- MIPS instructions are architecturally defined as 32-bit
  - MIPS64 produces 64-bit result by extending from 32-bit to allow 32-bit programs work as expected.

Variable vs. Register

- *Variable* is a **logical** storage, number of variables can be **unlimited**
- *Register* is a **physical** storage, number of registers is **limited**

Register (MIPS32)



a) 32 general purpose registers
b) Program counter register
c) Multiply and Divide register (higher result)
d) Multiply and Divide register (lower result)

Remarks: *HI and LO registers are removed in MIPS64 Release 6*

(Reference: https://www.mips.com/products/architectures/mips64/)

Different Types of Registers in MIPS

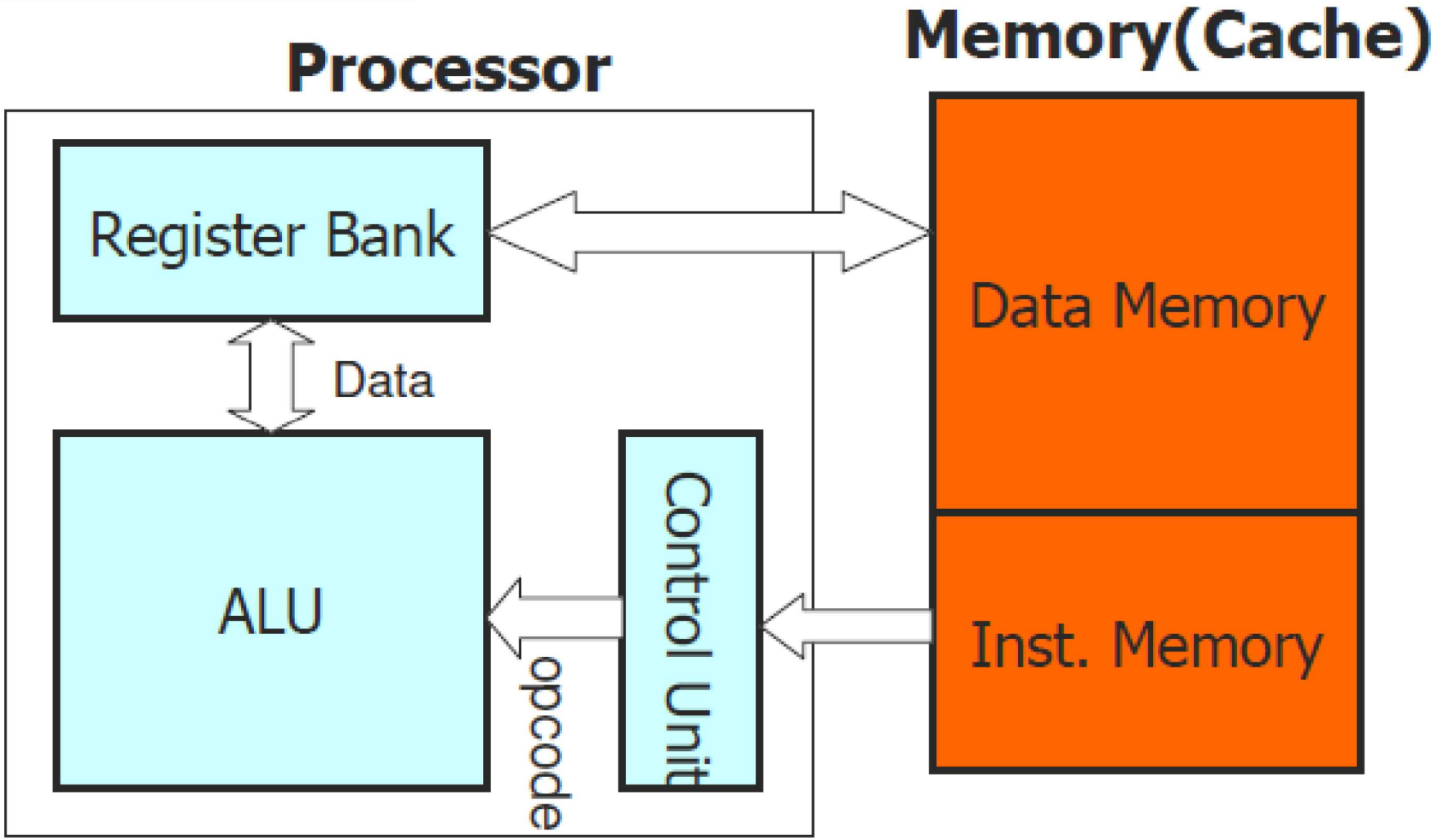
- Registers that correspond to variables in a high-level program are denoted as `$s0`, `$s1`, ... , `$s7`
- Temporary registers needed to compile the program into MIPS instructions are denoted as `$t0`, `$t1`, ... , `$t7`
- `$zero`, a special register holding a constant value 0, read-only (i.e. not modifiable)
- Register `$t0` is the 8th register is also called `$8`
- Others will be introduced later

MIPS registers

Register number	Register name	Usage
0	`zero`	Always zero
1	`$at`	Reserved for the assembler
2 - 3	`$v0 - $v1`	Expression evaluation and procedure return results
4 - 7	`$a0 - $a3`	The first four parameters passed to a procedure. Not preserved across procedure calls
8 - 15	`$t0 - $t7`	Temporary registers. Not preserved across procedure calls
16 - 23	`$s0 - $s7`	Saved values. Preserved across procedure calls
24 - 25	`$t8 - $t9`	Temporary registers. Not preserved across procedure calls
26 - 27	`$k0 - $k1`	Reserved for kernel usage.
28	`$gp`	Global pointer (pointer to global area)
29	`$sp`	Stack pointer
30	`$fp`	Frame pointer[a]
31	`$ra`	Return address

Data Store (Memory and Register)

Overall structure



Operands: Registers

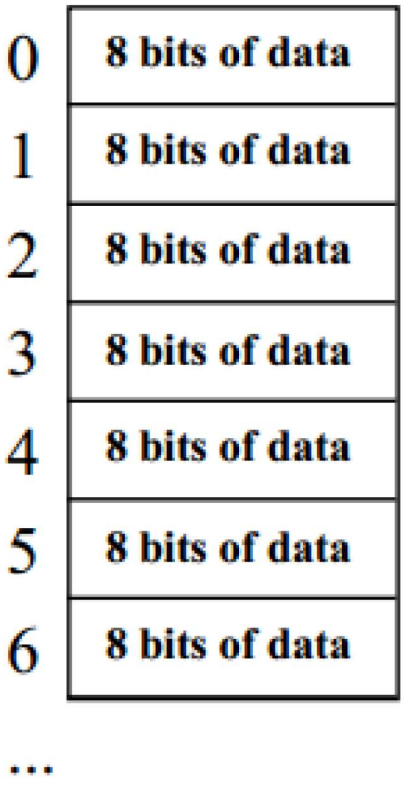
- **Written with a dollar sign ($) before their name**
  - For example, register 0 is written “$0”, pronounced “register zero” or “dollar zero”
- **Certain registers used for specific purposes:**
  - $0 always holds the constant value 0
  - the **saved registers**, $s0-$s7, are used to hold variables
  - the **temporary registers**, $t0 -$t9, are used to hold intermediate values during a larger computation

Operands: Memory

- **Too much data to fit in only 32 registers**
- **Store more data in memory**
  - Memory is large, so it can hold a lot of data
  - But it's also slow
- **Commonly used variables kept in registers**
- **Using a combination of registers and memory, a program can access a large amount of data quickly**

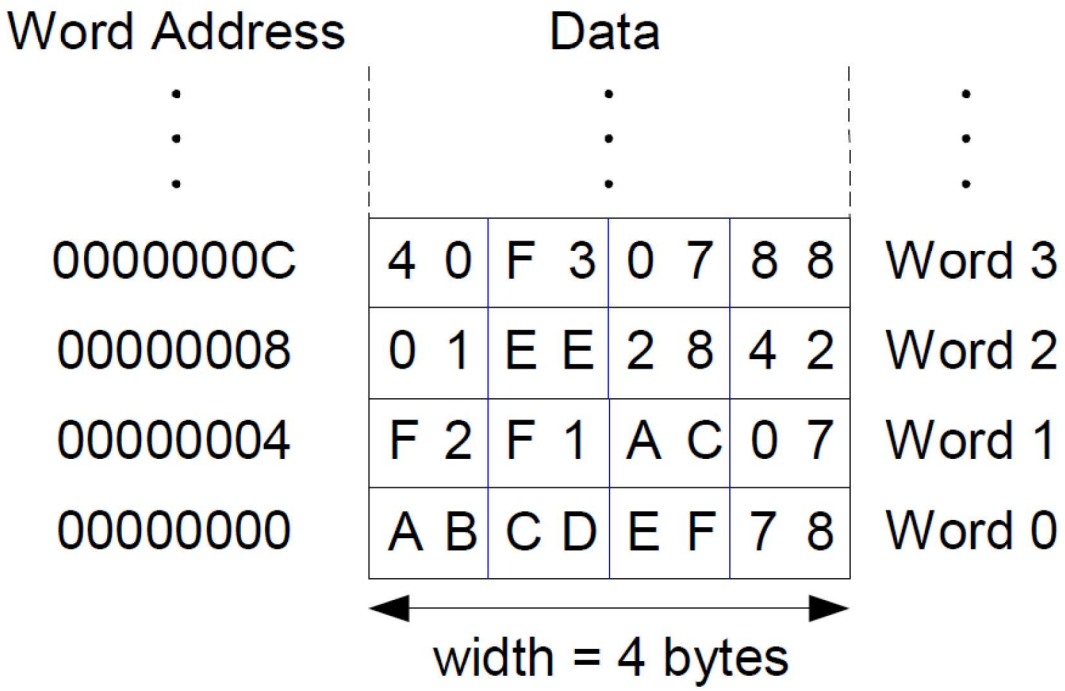
Memory

- Memory is viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- “Byte addressing” is used.



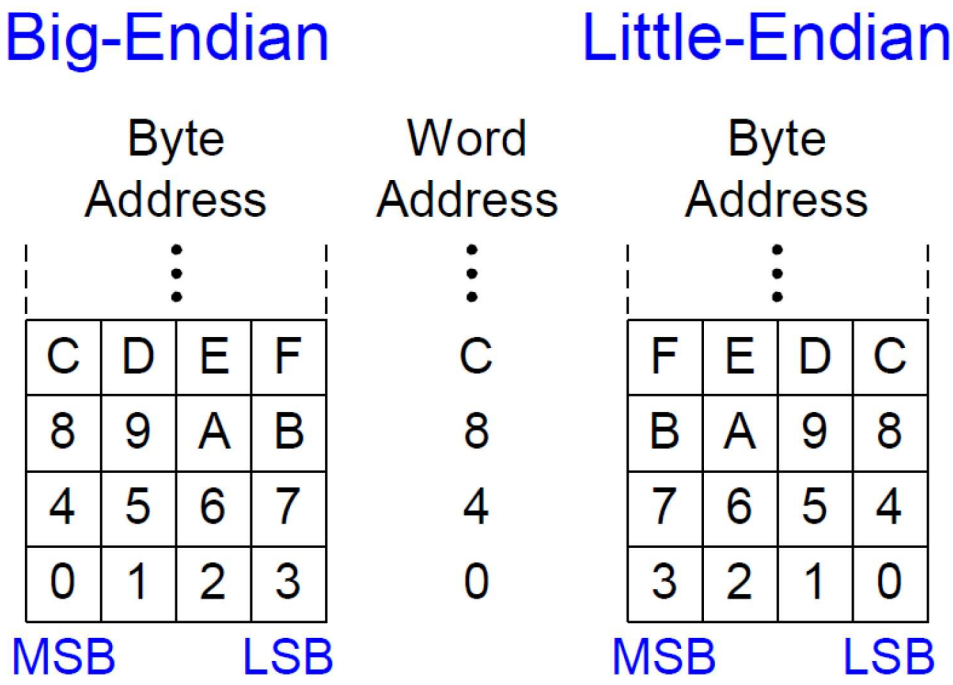
Byte-Addressable Memory

- Each data byte has a unique address
- Each 32-bit words has 4 bytes, so the word address increments by 4.
- MIPS uses byte addressable memory



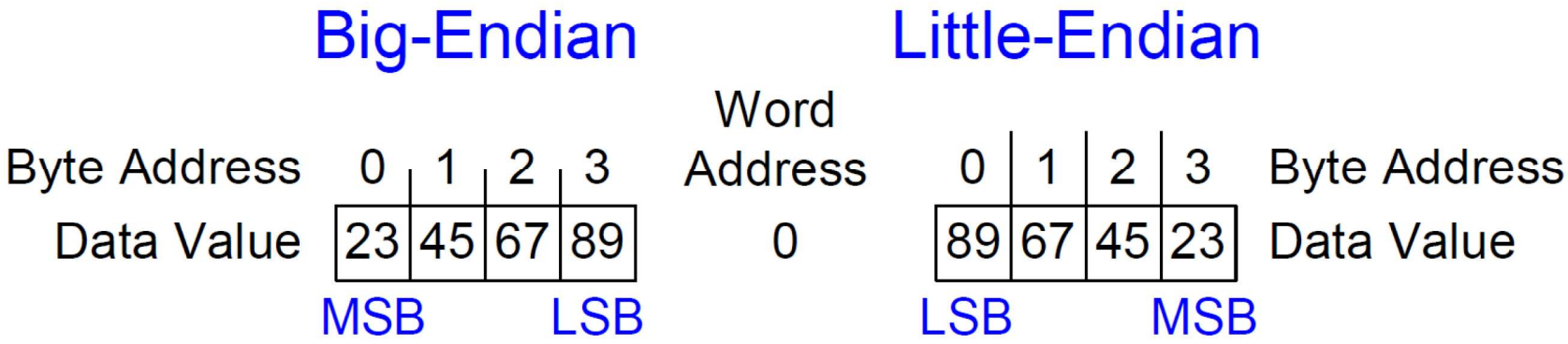
Big-Endian and Little-Endian Memory

- **How to number bytes within a word?**
- **Word address is the same for big-or little-endian**
  - ***Little-endian****:* byte numbers start at the little (least significant) end
  - ***Big-endian:*** byte numbers start at the big (most significant) end



Big-Endian and Little-Endian Memory (Example)

❖ **For a word 0x23456789 stored in memory:**



MIPS Endianness

- **MIPS is in the big endian camp.**
  - **Since the order matters only if it accesses the identical data both as a word and as four bytes $\Longrightarrow$ no need to be aware of the endianness.**
  - **Newer versions of MIPS can support both big and little endian.**
  - **MIPS is classically referred to as big endian.**
- **But for MARS, it is considered little-endian.**

Program Structure and Basic I/O in MIPS

A MIPS program compose of two main parts:

(a) Data Segment (`.data`)

- Declares the "variables" used in the program
- The variables are stored in memory
- Can not perform arithmetic operation directly

(b) Text Segment (`.text`)

- Your real program code (assembly instructions) after the data segment
- Basic operations
  - (a) Memory access
  - (b) Arithmetic operation
  - (c) Procedure call
  - (d) Logic operation
  - (e) more ….

Program Structure and Basic I/O in MIPS

Program Template

```
# Program and description of function
# (anything following '#' would be considered as comments)
#-----Data Segment-----------------
       .data          # variable declarations follow this line
                      # ...

#-----Text Segment-----------------
       .text          # instructions follow this line
                      # ...
                      # ...

# End of program
```

Program Structure and Basic I/O in MIPS

Data Allocation Directives (variable data types)

`.ascii`

- Store the ASCII string in memory, but without a null terminator.

`.asciiz`

- Store the ASCII string in memory with a null terminator (\0: ASCII code 0)
- "z" refers to zero, which is the ASCII code for the null character.
- This is how C-style strings are stored.

`.space`

- Reserves space in memory (in bytes)

Program Structure and Basic I/O in MIPS

Data Allocation Directives (variable data types)

`.word`

Store 32-bit (4 bytes) contiguously in memory, and must start at byte address that is a multiple of 4 bytes (at the word boundary)

Example:

```
.align 2
.word 10
```

`.half`

Store 16-bit (2 bytes) contiguously in memory, and must start at byte address that is a multiple of 2 bytes (at the halfword boundary)

`.byte`

Store 8-bit (1 byte) contiguously in memory

Remarks: Values of word, half and byte can be written in base 10 ,hex or octal, and separated by commas for multiple values

Program Structure and Basic I/O in MIPS

Data declaration

variableName : *data_type* *value(s)*

```
# Program and description of function
#-----Data Segment-----------------
                .data
        age:    .word 30          # 32-bit word initialized with decimal
        var1:   .byte 1,2,3       # similar to char var1[] = {1,2,3};
        var2:   .word 8,7,6,5     # define an array in multiple lines
                .word 4,3,2,1     # int var2[] = {8,7,6,5,4,3,2,1};
        gpa:    .half 0x10        # 16-bit word initialized with hex
        kb:     .space 1024       # reserve 1KB of space
        msg:    .asciiz "Hi"      # null-terminated string
                                  # similar to String msg = "Hi";
#-----Text Segment-----------------
                .text
                                  # instruction statements
                                  # ...
# End of program
```

Program Structure and Basic I/O in MIPS

syscall service	System Call No. in $v0	Arguments	Result	Example
print_int	**1**	**$a0=integer**		**li $v0, 1 li $a0, 100 syscall**
print_float	2	$f12=32-bit float		
print_double	3	$f12=64-bit double		
print_string	**4**	**$a0=start address of the null-terminated string**		
read_int	**5**		**integer (in $v0)**	**li $v0, 5 syscall**
read_float	6		float (in $f0)	
read_double	7		double (in $f0 and $f1)	
read_string	**8**	$a0=address of the string in buffer **$a1=max. length+1**	Input string stores in buffer pointed by $a0	
sbrk	9	$a0=# of bytes	address (in $v0)	
exit	**10**			**li $v0, 10 syscall**

Program Structure and Basic I/O in MIPS

Load/ Store Instructions

- *Load* means load a value from memory, and data is copy from memory into register
- *Store* means store a value to memory, and data is copy from register to memory

Instr	Function	Syntax
`li`	Load immediate value into a register	`li [$rt], [constant value]`
`la`	Load address into a register	`la [$rt], [label]`

`lw`	Load a word from a location in memory to a register	`[instr] [$rt], [label]`
		`[instr] [$rt], [value]`
		`[instr] [$rt], [offset] ([$rs])`
		
		`e.g.`
		`lw $a0, 4($t0)`
		`sw $v0, variable`

`lb`	Load a byte from a location in memory to a register	
`lbu`	Load a byte (unsigned) from a location in memory to a register.	
`sw`	Store a word from a register to a location in memory	
`sb`	Store the least significant byte of a register to a location in memory	

Remarks:

- `$rt` and `$rs` are registers
- value and offset should be16-bit 2's C value (i.e. -32768 to +32767)

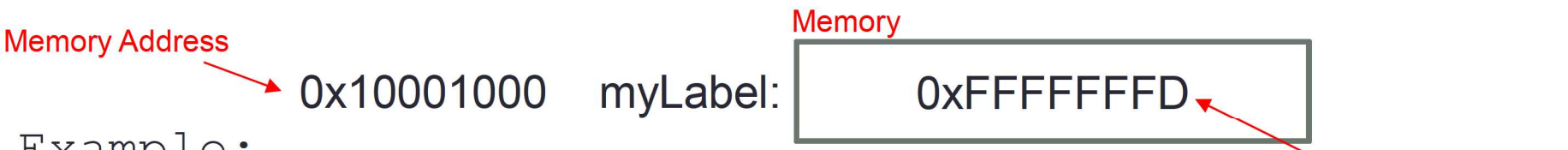
Program Structure and Basic I/O in MIPS

Other Instructions

Instr	Function	Syntax
`move`	Copy value of `$rs` to `$rt`	`move $rt, $rs`
`add`	Add as signed (2's complement) integers	(Will be further explained later)
`addi`	Add immediate	
`mul`	Multiply 32-bit and store 64-bit	
`sub`	Subtraction	
`b`	Unconditional branch to target	
`bgt`	Branch to target if greater than	
`blt`	Branch to target if less than	
`ble`	Branch to target if less than or equal to	
`bge`	Branch to target if greater than or equal to	
`beq`	Branch to target if equal	
`bne`	Branch to target if not equal	
`j`	Unconditional jump to target	
`jr`	Jump to the address contained in the register	
`jal`	Jump to the calculated address and store the return address in $ra	

Program Structure and Basic I/O in MIPS

Given a label `myLabel` is at location [0x10001000] and contains 0xFFFFFFFD,




```
Example:
        la $a0, myLabel    # loads ?
        lw $a0, myLabel    # loads ?
```

Program Structure and Basic I/O in MIPS

Input/ Output Instructions

- Read string input
  1. Use the data segment to create a label to hold the input string, and reserve space (bytes) in memory.

  E.g. `str: .space 65 # reserve 64 characters + 1 for null`

  1. Load the address of the label to `$a0`. E.g. `la $a0, str`
  2. Load the maximum length of the input string to `$a1`

  E.g. `li $a1, 65 # 65=(64+1) which counts max. 64 key inputs`

  (Note - Carriage Return (CR)/Enter key is considered as one character input)

  3. Load the read_string code (= 8) to register `$v0. E.g. li $v0, 8`
  4. Invoke `syscall`
- Read integer input
  1. Load the read_int code to register $v0. E.g. `li $v0, 5`
  2. Invoke `syscall`
  3. (Optional) Store the content of `$v0` to an address or copy to another register

Program Structure and Basic I/O in MIPS

```
# Example 1: Print Hello World
#-----Data Segment-----------------
        .data
msg:    .asciiz "Hello World\n"

#-----Text Segment-----------------
        .text
        .globl __start          # program execution begins
__start:
        # program codes
        li $v0, 4               # load immediate into register
        la $a0, msg             # load address
        syscall                 # call operating system

        # To indicate end of program (Exit from the program)
        li $v0, 10              # system call code for exit = 10
        syscall                 # call operating system
# End of program
```

```
# Example 2: Printing
.data
X: .word 1 2 3
string1: .asciiz "AB"
Y: .byte 67 68 0
Z:
NewLine: .asciiz "\n"

.text
.globl _start
_start:

# Print X
la $t0, X
lw $a0, 0($t0)
li $v0, 1
syscall

lw $a0, 4($t0)
li $v0, 1
syscall

lw $a0, 8($t0)
li $v0, 1
syscall

# Print NewLine
la $a0, NewLine
li $v0, 4
syscall

# Print string1
la $a0, string1
li $v0, 4
syscall

# Print NewLine
la $a0, NewLine
li $v0, 4
syscall

# Print Y
la $a0, Y
li $v0, 4
syscall

# Print NewLine
la $a0, NewLine
li $v0, 4
syscall

# Print Z
la $a0, Z
li $v0, 4
syscall

# Print NewLine
la $a0, NewLine
li $v0, 4
syscall

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

Trace the program carefully !!