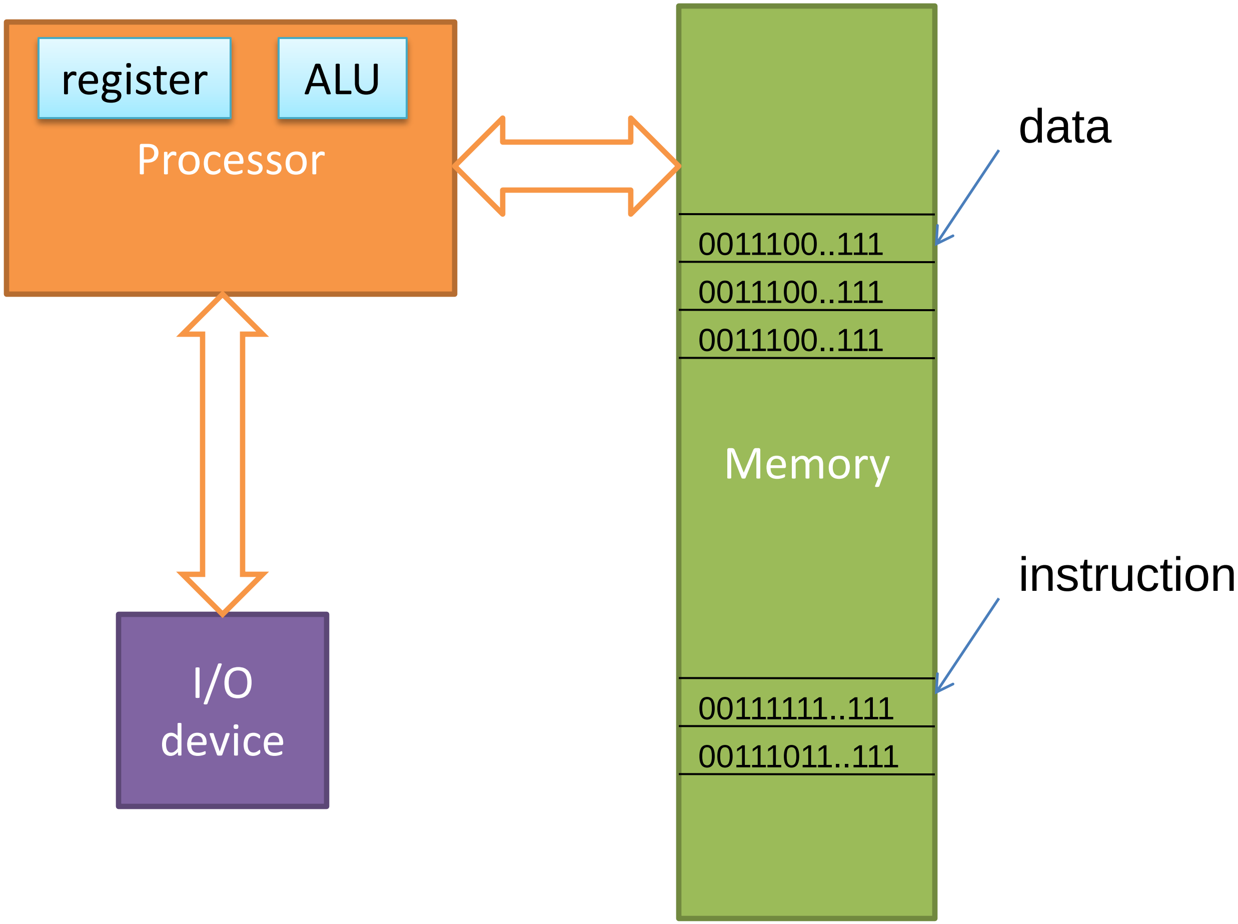
MIPS assembly

Computer Model

- What's in a computer?
- Processor
- Memory
- I/O devices (keyboard, mouse, LCD, video camera, speaker, disk, CD drive, …)

Computer Model



Registers and ALU

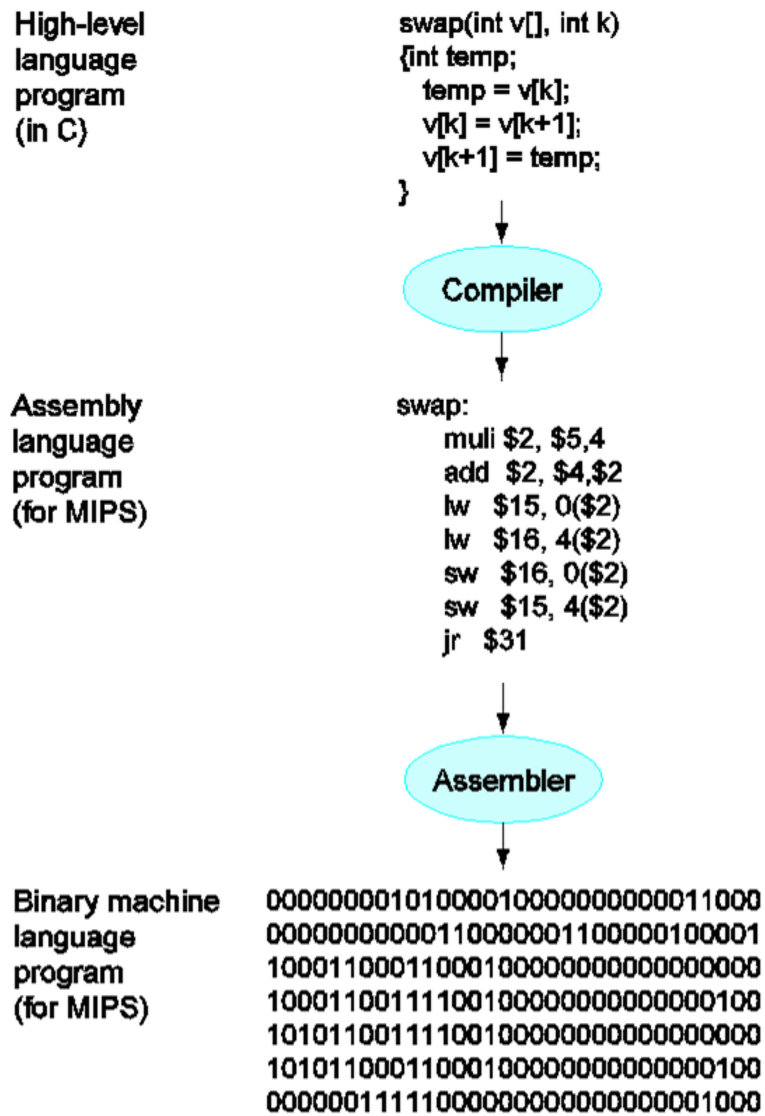
- A processor has **registers** and **ALU**
  - Registers are where you store values (e.g., the value of a variable)
  - The values stored in registers are sent to the ALU to be added, subtracted, anded, ored, xored, …, then the result is stored back in a register. Basically it is the heart of the processor and does the calculation.

Memory

- Memory is modeled as a continuous space from 0 to 0xffff…ffff.
- Every byte in the memory is associated with an index, called **address**.
- We can read and write:
  - Given the address to the memory hardware, we can **read** the content in that byte.
  - Given the address and a byte value, we can **modify** the content in the memory at that addres.

Program and Data

- Programs consist of instructions and data, **both** stored in the memory
- Instructions are also represented as 0's and 1's
- A program is executed **instruction by instruction**



High-level language program (in C)
swap(int v[], int k)
{int temp;
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
}
Compiler
Assembly language program (for MIPS)
swap:
muli $2, $5,4
add $2, $4,$2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
Assembler
Binary machine language program (for MIPS)
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000

GoogleEarth.exe

```
00905a4d 00000003 00000004 0000ffff
000000b8 00000000 00000040 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 000000e8
0eba1f0e cd09b400 4c01b821 685421cd
70207369 72676f72 63206d61 6f6e6e61
65622074 6e757220 206e6920 20534f44
65646f6d 0a0d0d2e 00000024 00000000
bc160841 ef786905 ef786905 ef786905
ef72766a ef786902 ef767586 ef78690b
ef644adf ef786906 ef5d4b36 ef786907
ef796905 ef7869be ef614aff ef786914
ef734a03 ef78690a ef7e6fc2 ef786904
68636952 ef786905 00000000 00000000
00000000 00000000 00004550 0004014c
424219de 00000000 00000000 010f00e0
0006010b 00010a00 0000be00 00000000
```

Linux Kernel

```
ea66 0008 0000 07c0 c88c d88e c08e d08e
00bc fb7c befc 0031 20ac 74c0 b409 bb0e
0007 10cd f2eb c031 16cd 19cd f0ea 00ff
44f0 7269 6365 2074 6f62 746f 6e69 2067
7266 6d6f 6620 6f6c 7070 2079 7369 6e20
206f 6f6c 676e 7265 7320 7075 6f70 7472
6465 0d2e 500a 656c 7361 2065 7375 2065
2061 6f62 746f 6c20 616f 6564 2072 7270
676f 6172 206d 6e69 7473 6165 2e64 0a0d
520a 6d65 766f 2065 6964 6b73 6120 646e
7020 6572 7373 6120 796e 6b20 7965 7420
206f 6572 6f62 746f 2e20 2e20 2e20 0a0d
0000 0000 0000 0000 0000 0000 0000 0000
```

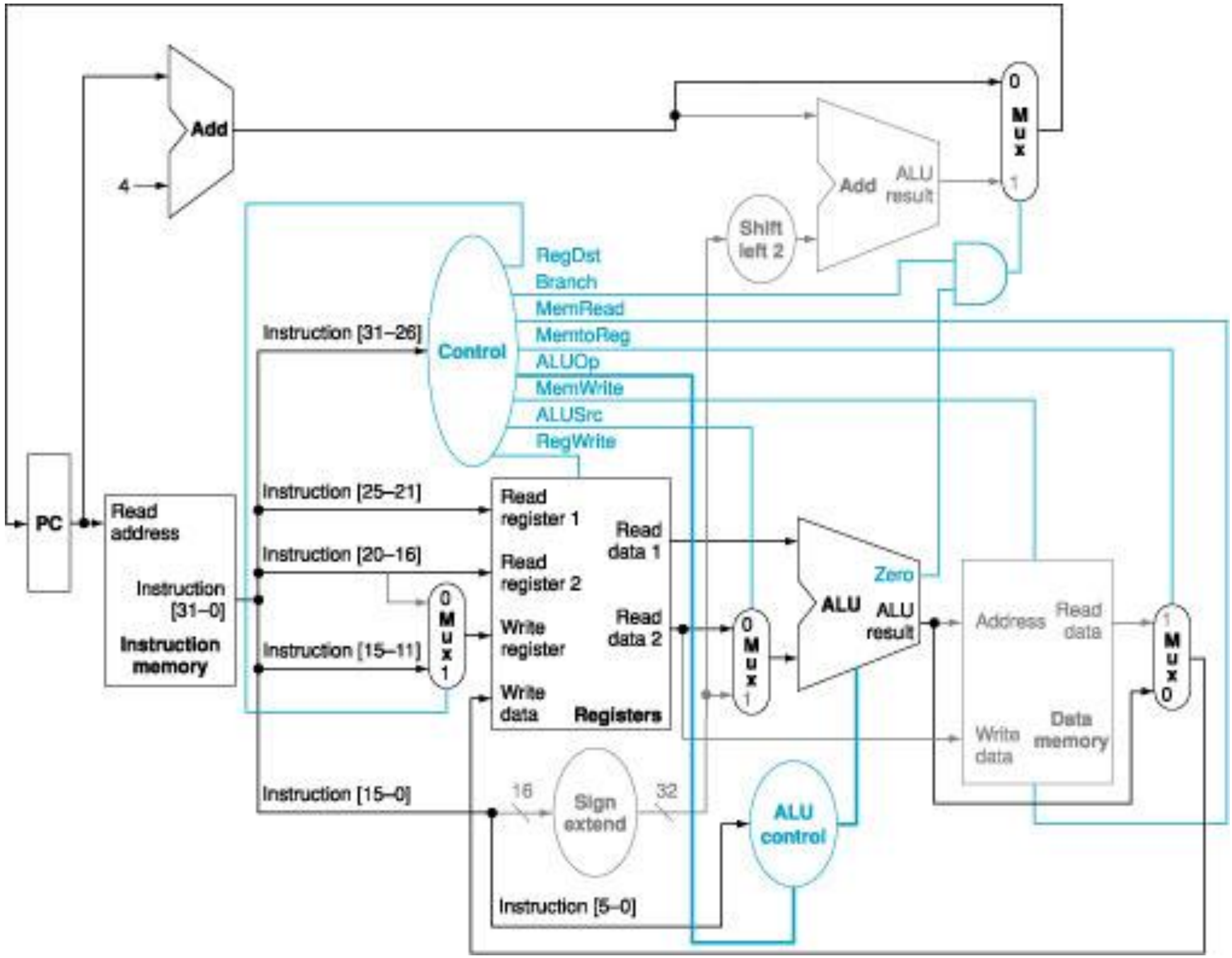
Why are we learning assembly

- Comparing to higher level languages such as C, assembly languages
  - are more difficult to write, read, and debug.
  - have poor portability – Every processor has its own assembly language. The code you wrote for MIPS is NOT going to run on Intel processors.
- Then why are we learning it?
  - After learning the first assembly language, the second will be MUCH easier
  - It brings us closer to the processor, which is the goal of this course.

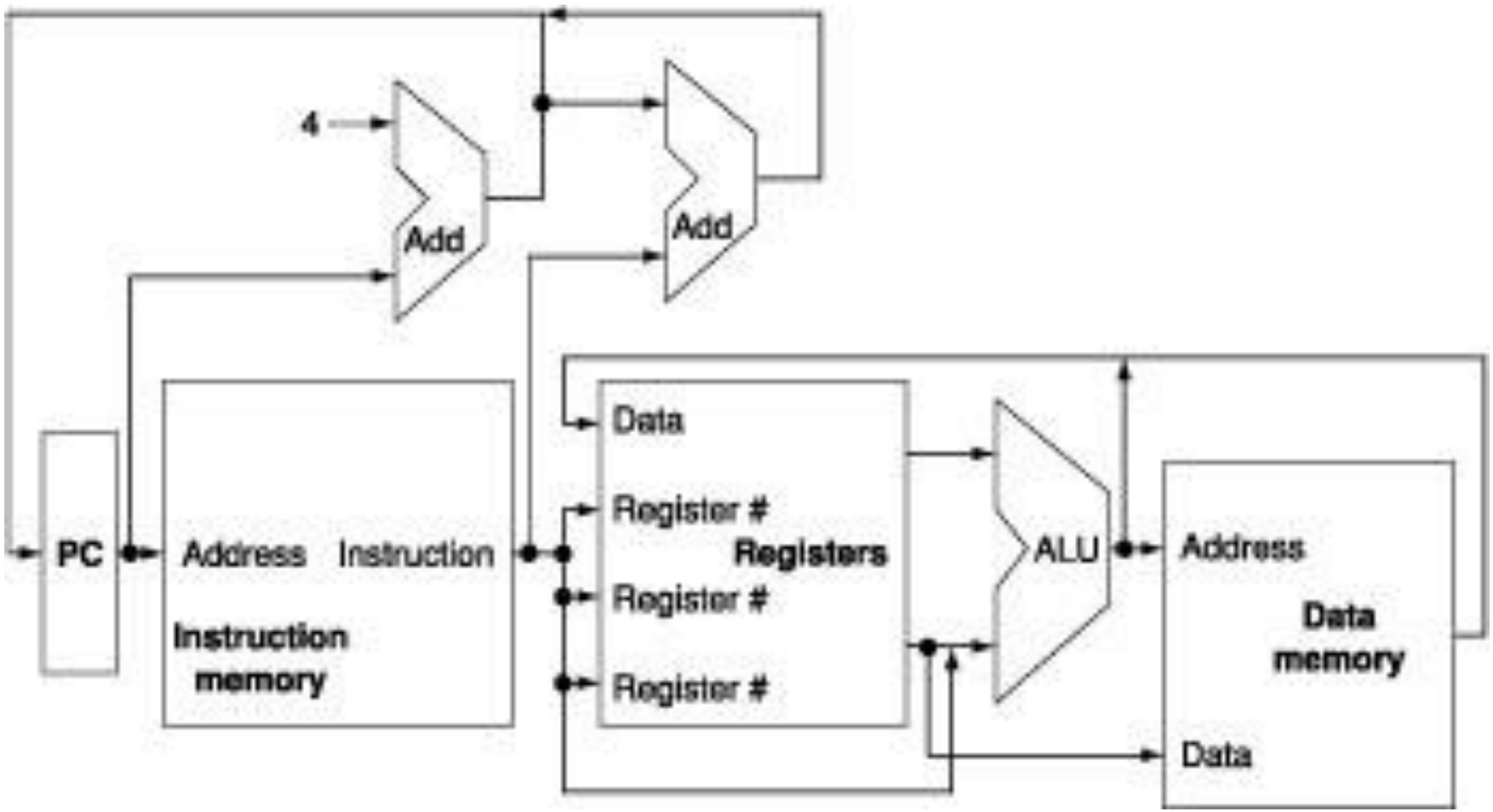
MIPS ISA

- There are many different Instruction Set Architectures designed for different applications with different performance/cost tradeoff
  - Including Intel-32, PowerPC, MIPS, ARM ….
- We focus on MIPS architecture
  - *M*icroprocessor without *I*nterlocked *P*ipeline *S*tages
  - A RISC (reduced instruction set computer) architecture
    - In contrast to CISC (complex instruction set computer)
  - Similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …

A peek into the future…



Abstract View of MIPS Implementation



MIPS Instruction Set

- An instruction is a command that hardware understands
  - Instruction set is the vocabulary of commands understood by a given computer
  - It includes arithmetic instructions, memory access instructions, logical operations, instructions for making decisions

Arithmetic Instructions

- Each MIPS arithmetic instruction performs only one operation
  - Each one must always have exactly three variables

```
add   a, b, c        # a = b + c
```

    - Note that these variables can be the same though
  - If we have a more complex statement, we have to break it into pieces

Arithmetic Instructions

- Example
  - f = (g + h) – (i + j)

Arithmetic Instructions

- Example
  - f = (g + h) – (i + j)

```
add t0, g, h       # temporary variable t0 contains g + h
add t1, i, j       # temporary variable t1 contains i + j
sub f, t0, t1      # f gets t0 - t1
```

Operands of Computer Hardware

- In C, we can define as many as variables as we need
  - In MIPS, operands for arithmetic operations must be from registers
  - MIPS has thirty-two 32-bit registers

MIPS Registers

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
\$0	\$zero	Permanently 0	\$24, \$25	\$t8, \$t9	Temporary
\$1	\$at	Assembler Temporary (reserved)	\$26, \$27	\$k0, \$k1	Kernel (reserved for OS)
\$2, \$3	\$v0, \$v1	Value returned by a subroutine	\$28	\$gp	Global Pointer
\$4–\$7	\$a0–\$a3	Arguments to a subroutine	\$29	\$sp	Stack Pointer
\$8–\$15	\$t0–\$t7	Temporary (not preserved across a function call)	\$30	\$fp	Frame Pointer
\$16–\$23	\$s0–\$s7	Saved registers (preserved across a function call)	\$31	\$ra	Return Address

Arithmetic Instructions

- Example
  - f = (g + h) – (i + j)

    #In MIPS, add can not access variables directly

    #because they are in memory

    # Suppose f, g, h, i, and j are in $s0, $s1, $s2, $s3, $s4 respectively

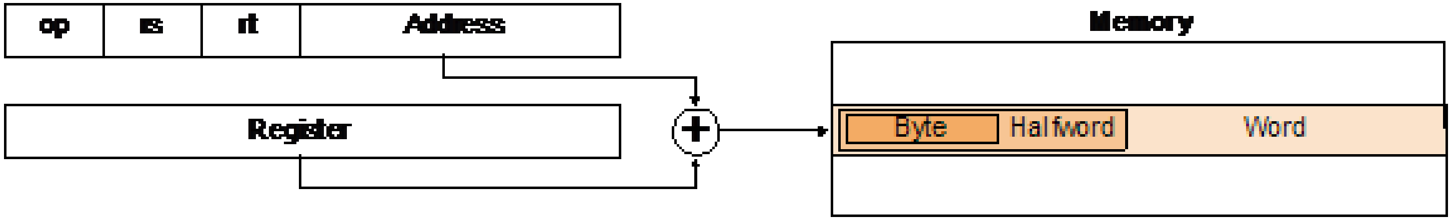
    ```
    add $t0, $s1, $s2     # temporary variable t0 contains g + h
    add $t1, $s3, $s4     # temporary variable t1 contains i + j
    sub $s0, $t0, $t1     # f gets t0 – t1
    ```

Memory Operands

- Since variables (they are data) are initially in memory, we need to have data transfer instructions
  - Note a program (including data (variables)) is loaded from memory
  - We also need to save the results to memory
  - Also when we need more variables than the number of registers we have, we need to use memory to save the registers that are not used at the moment
- Data transfer instructions
  - lw (load word) from memory to a register
  - sw (store word) from register to memory

Using Load and Store

- Memory address in load and store instructions is specified by a base register and offset

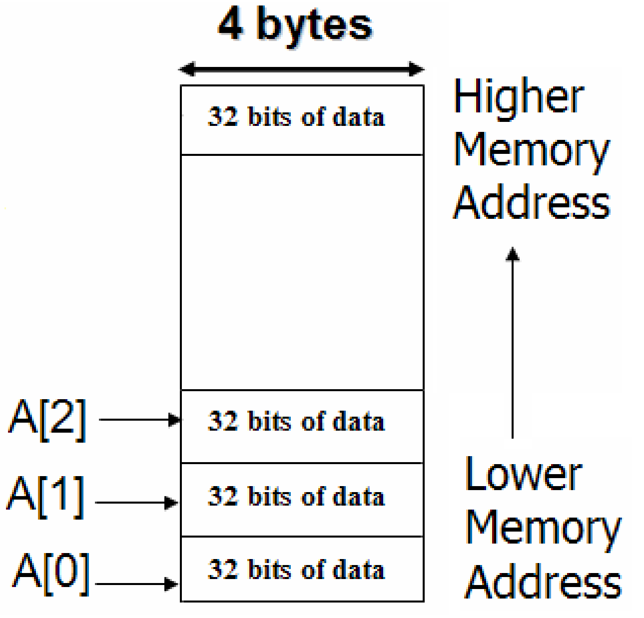


  – This is called base addressing

Using Load and Store

- How to implement the following statement using the MIPS assembly we have so far?
  - Assuming the address of A is in \$s3 and the variable h is in \$s2

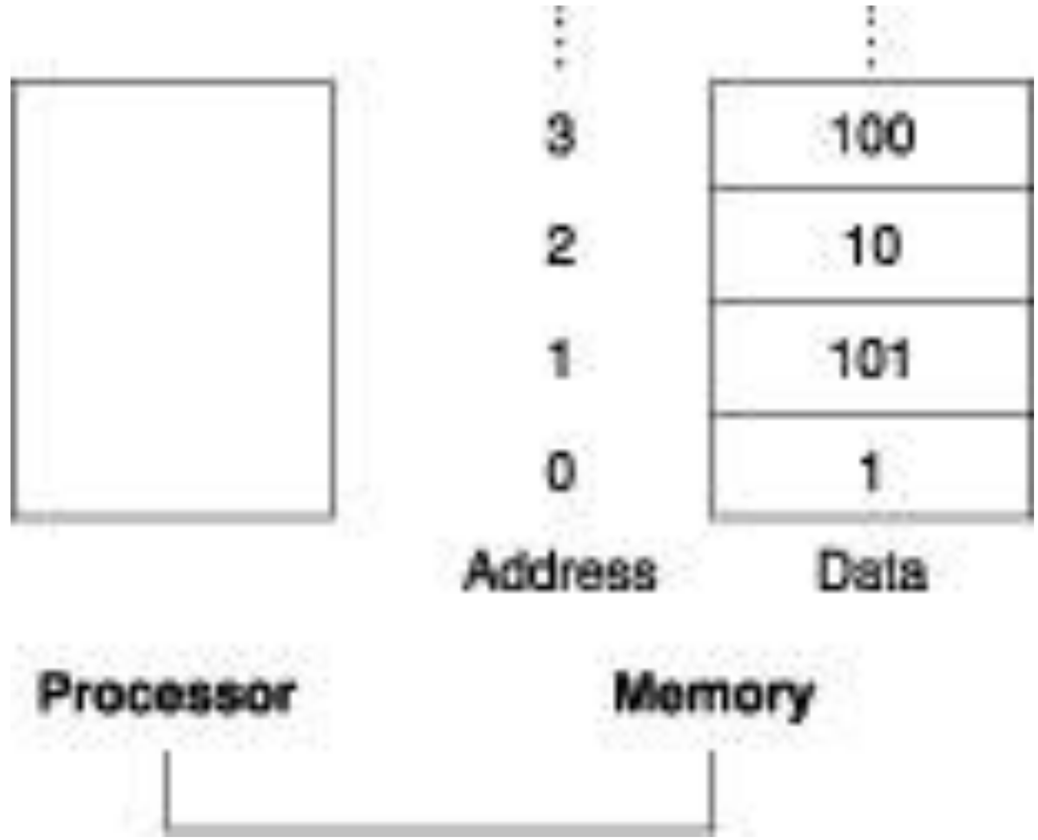
  A[12] = h + A[8]




```
lw      $t0, 32($s3)     #Temporary reg $t0 gets A[8]
add     $t0, $s2, $t0    #Temporary reg $t0 gets h + A[8]
sw      $t0, 48($s3)
```

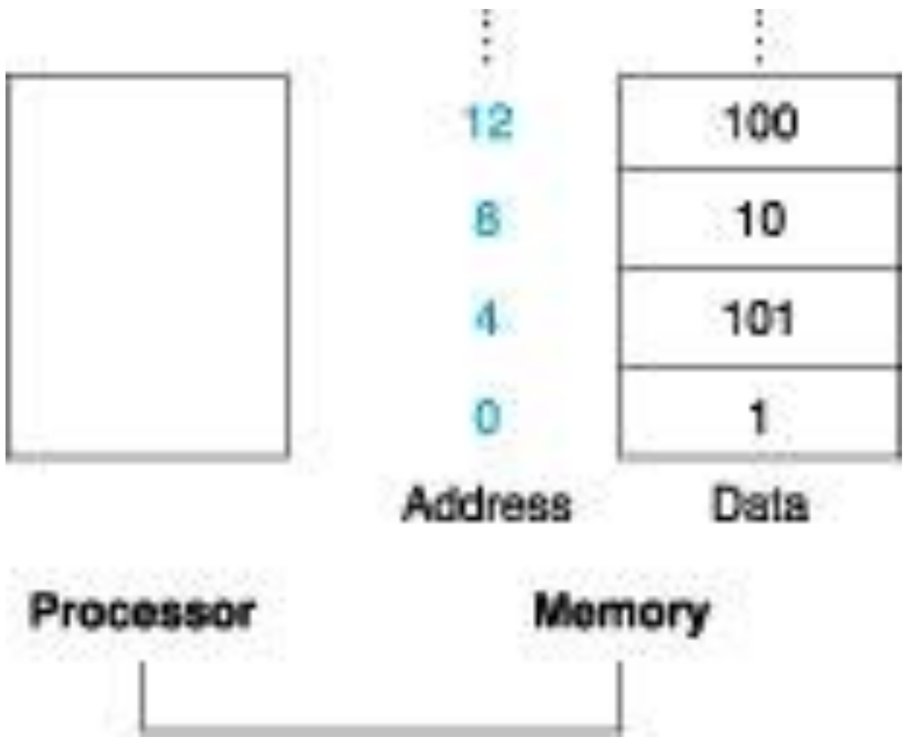
Specifying Memory Address

- Memory is organized as an array of bytes (8 bits)



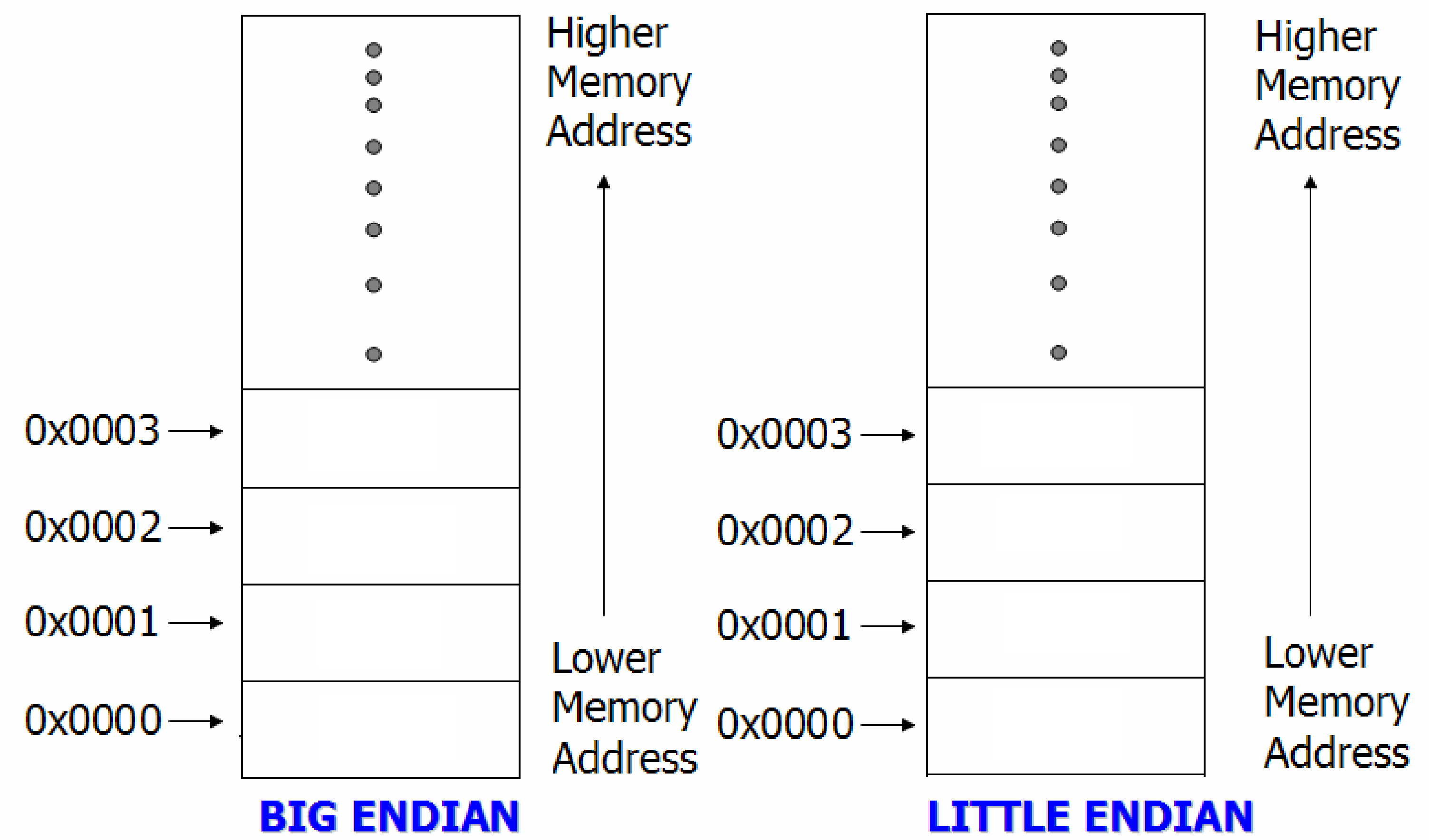
Specifying Memory Address

- MIPS uses words (4 bytes)
  - Each word must start at address that are multiples of 4
  - This is called alignment restriction
  - Big Endian



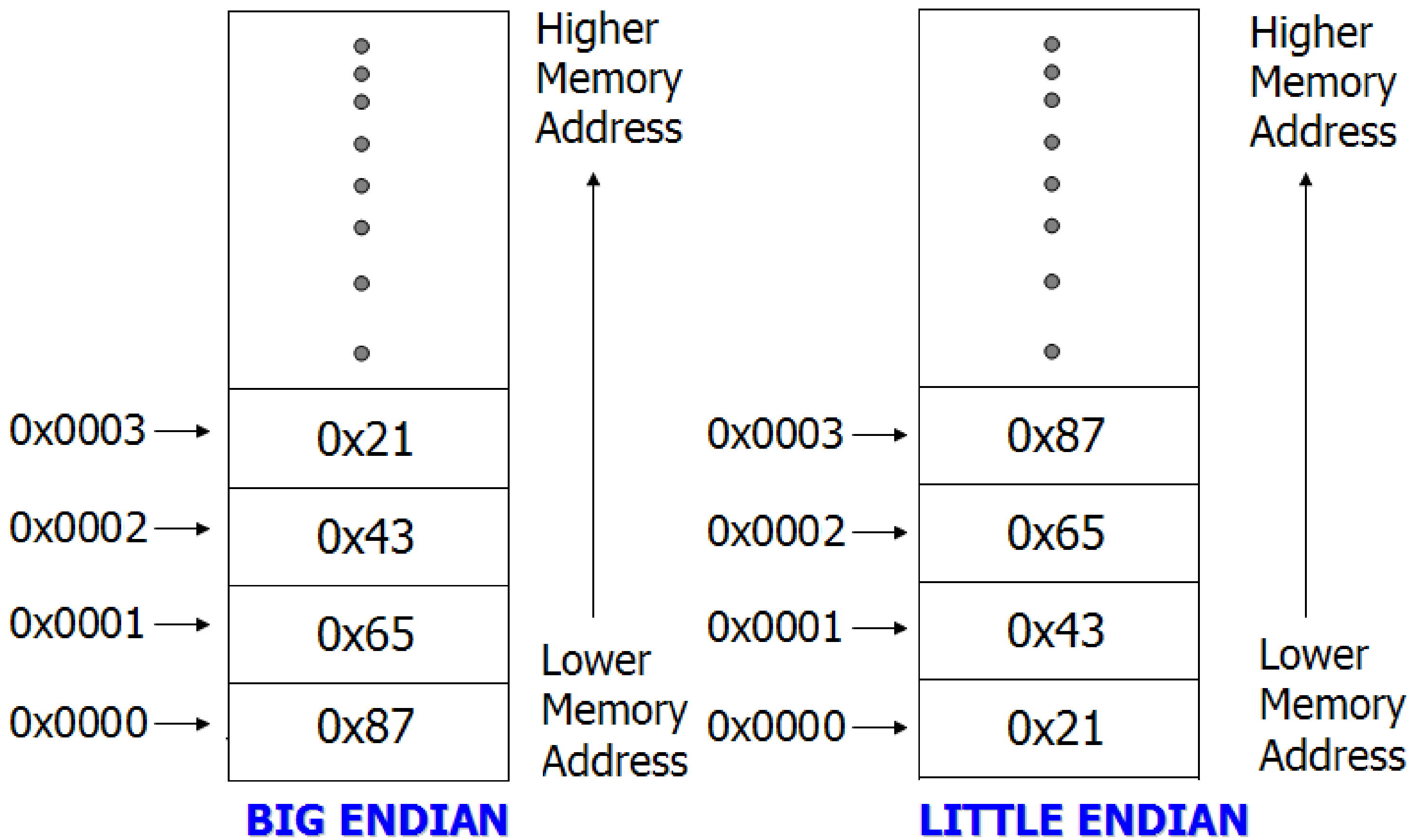
Example of Endianness

- Store 0x87654321 at address 0x0000, byte-addressable



Example of Endianness

- Store 0x87654321 at address 0x0000, byte-addressable



MIPS Assembly Programs

- Consists of MIPS instructions and data
  - Instructions are given in .text segments
    - A MIPS program can have multiple .text segments
  - Data are defined in .data segments using MIPS assembly directives
    - .word, for example, defines the following numbers in successive memory words
  - See Appendix A A.10 (pp. A-45 – A-48) for details

Exercise 1

- Suppose we have an array with starting address stored in $s0. We want to add the content of the first three elements, and put the result in the fourth element?
  - A[3] = A[2] + A[1] + A[0]