



EECS 2021

Computer Organization Fall 2015

*The slides are based on the publisher slides
and contribution from Profs Amir Asif and
Peter Lian*

*The slides will be modified, annotated,
explained on the board, and sometimes
corrected in the class*

Based on slides by the author and prof.
Mary Jane Irwin of PSU.

Chapter Summary

- Stored-program concept
- Assembly language
- Number representation
- Instruction representation
- Supporting procedures in hardware
- MIPS addressing
- Some real-world stuff
- Fallacies and Pitfalls




It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature:

***simplicity** of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the **speed** of its handling of those problems.*

Burks, Goldstine, and von Neumann, 1947





Marketing and management will not usually ask for a combination of features, cost, and schedule that are simultaneously realizable – count on that. They will reliably ask for the impossible. ... It's a very short step from using Moore's Law to *check* your road map to using it to *dictate* the road map.

Bob Colwell, *The Pentium Chronicles*



Stored-Program Concept

- Program instructions are stored in the memory.
- Every cycle, an instruction is read from the memory (fetched).
- The instruction is examined to decide what to do (decode)
- Then we perform the operation stated in the instruction (execute)
- Fetch-Decode-Execute cycle.



Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets RISC vs. CISC



The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - See MIPS Reference Data tear-out card, and Appendixes B and E



The Four Design Principles

1. Simplicity favors regularity.
2. Smaller is faster.
3. Make the common case fast.
4. Good design demands good compromises



Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- ***Design Principle 1: Simplicity favors regularity***
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost



Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code: *(almost, this is not really assembly)*

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f,  t0, t1  # f = t0 - t1
```

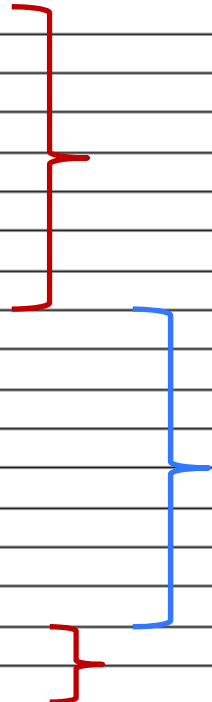


Register Operands

- Arithmetic instructions use **register operands**
- MIPS has a 32 **32-bit register** file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
- ***Design Principle 2: Smaller is faster***
 - c.f. main memory: millions of locations



Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)
\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)



Register Operand Example

- C code:

$f = (g + h) - (i + j);$

- f, \dots, j in $\$s0, \dots, \$s4$

- Compiled MIPS code: (This is a real assembly)

```
add $t0, $s1, $s2
```

```
add $t1, $s3, $s4
```

```
sub $s0, $t0, $t1
```



Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is **Big Endian** (The commercial MIPS, not really, but in this course)
 - Most-significant byte at least address of a word
 - *c.f.* Little Endian: least-significant byte at least address



Memory Access

Another way to put it

Big Endian: leftmost byte is word address

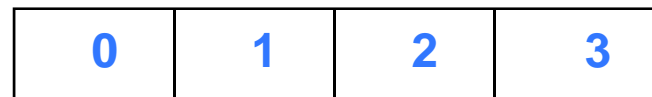
Little Endian: rightmost byte is word address

little endian

LSB

MSB

Bytes address

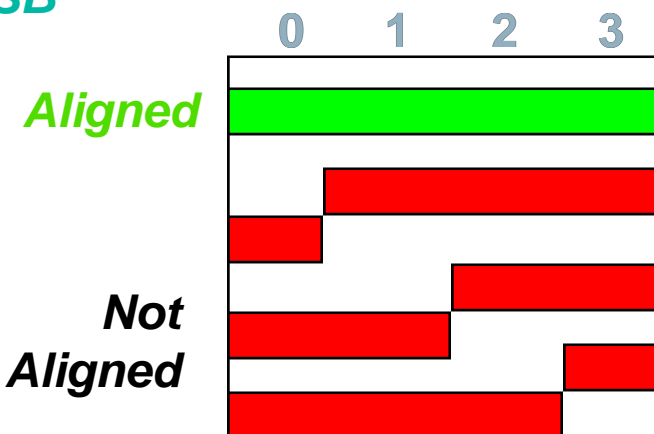


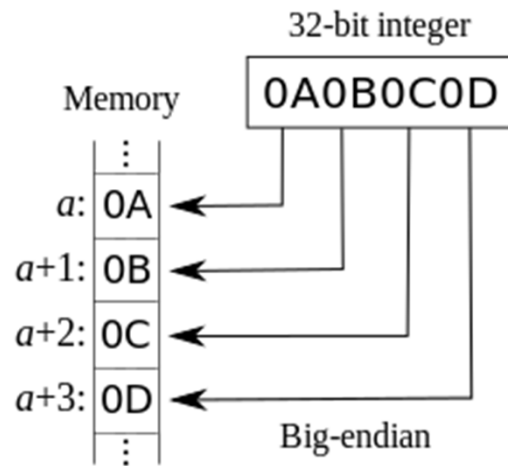
big endian

MSB

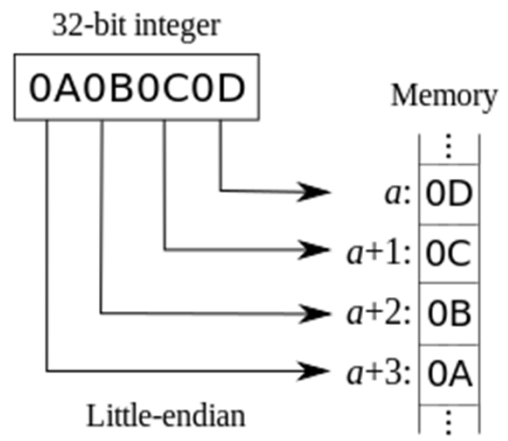
LSB

Alignment restriction: requires that objects fall on address that is multiple of their size





Big-Endian



Little-Endian

"Little-Endian" by R. S. Shaw - Own work. Licensed under Public Domain via Commons - <https://commons.wikimedia.org/wiki/File:Little-Endian.svg#/media/File:Little-Endian.svg>



Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)    #load byte from memory
```

```
sb    $t0, 6($s3)    #store byte to memory
```

- What 8 bits get loaded and stored?
 - load byte places the byte from memory in the rightmost 8 bits of the destination register
 - what happens to the other bits in the register?
 - store byte takes the byte from the rightmost 8 bits of a register and writes it to the byte in memory
 - leaving the other bytes in the memory word unchanged



Example

- Given the following code sequence and memory state what is the state of the memory after executing the code?

```

add    $s3, $zero, $zero
lb     $t0, 1($s3)
sb     $t0, 6($s3)

```

- What value is left in \$t0?

$\$t0 = 0x00000090$

Memory		
24	0x 0 0 0 0 0 0 0 0	24
20	0x 0 0 0 0 0 0 0 0	20
16	0x 0 0 0 0 0 0 0 0	16
12	0x 1 0 0 0 0 0 1 0	12
8	0x 0 1 0 0 0 4 0 2	8
4	0x F F F F F F F F	4
0	0x 0 0 9 0 1 2 A 0	0

- What word is changed in Memory and to what?

$mem(4) = 0xFFFF90FF$

- What if the machine was little Endian?

$\$t0 = 0x00000012$

?

$mem(4) = 0xFF12FFFF$ Value not layout