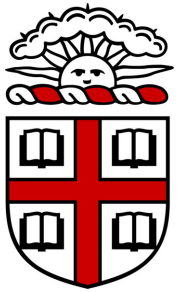
EN164: Design of Computing Systems
Lecture 09: Processor / ISA 2

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2011



[material from Patterson & Hennessy, 4th ed and Harris 1st ed]

MIPS architecture

- Used as the example throughout the course
- 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
- 32 bit and 64 bit available
- Will implement and boot a subset MIPS processor in lab

Arithmetic operations

High-level code

```
a = b + c;
a = b - c;
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
sub $s0, $s1, $s2
```

- add: mnemonic indicates what operation to perform
- $s1, $s1: source operands on which the operation is performed
- $s0: destination operand to which the result is written

- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register operands

- Memory is slow.
- Most architectures have a small set of (fast) registers
- MIPS has a 32 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
  - Written with a $ before their name
- Assembler names
  - $0 always hold value 0
  - $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

MIPS registers

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

Format of R-type instructions

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function*
    together, the opcode and function tell the computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type examples

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

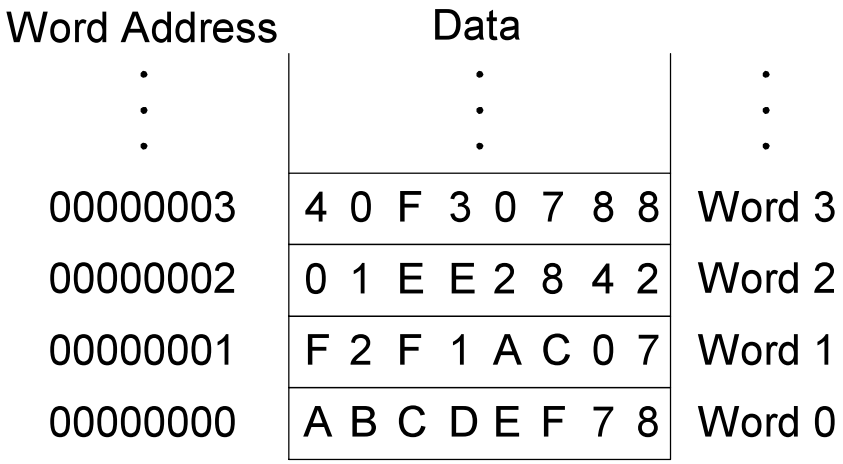
`add rd, rs, rt`

Memory operands

- Too much data to fit in only 32 registers
- Memory is large, but slow
- Commonly used variables kept in registers
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory

Word-addressable memory

- Each 32-bit data word has a unique address

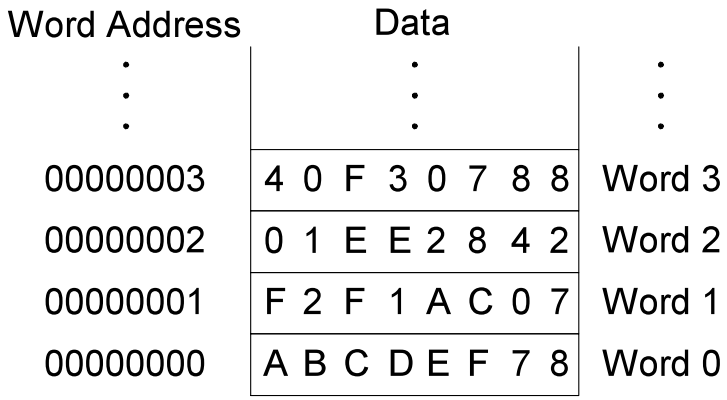


Reading word-addressable memory

- Memory reads are called *loads*
- Mnemonic: *load word* (`lw`)
- Example: read a word of data at memory address 1 into `$s3`
- Memory address calculation:
  - add the *base address* (`$0`) to the *offset* (1)
  - address = ($0 + 1) = 1
- Any register may be used to store the base address.
- `$s3` holds the value 0xF2F1AC07 after the instruction completes.

Assembly code

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

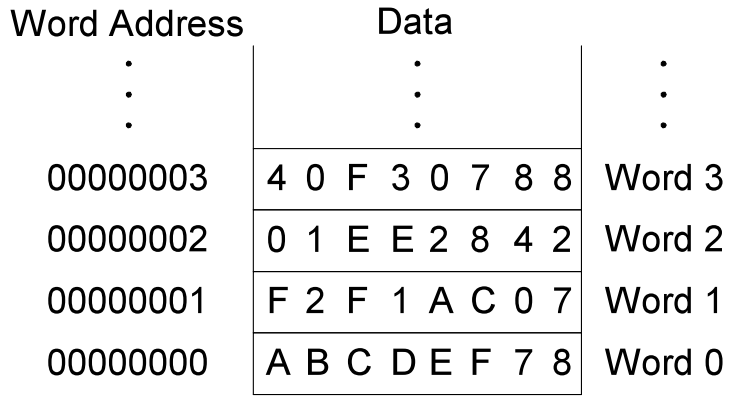


Writing word-addressable memory

- Memory writes are called *stores*
- Mnemonic: *store word* (`sw`)
- Example: Write (store) the value held in `$t4` into memory address 7
- Offset can be written in decimal (default) or hexadecimal
- Memory address calculation:
  - add the base address (`$0`) to the offset (0x7)
  - address: ($0 + 0x7) = 7
- Any register may be used to **store** the base address

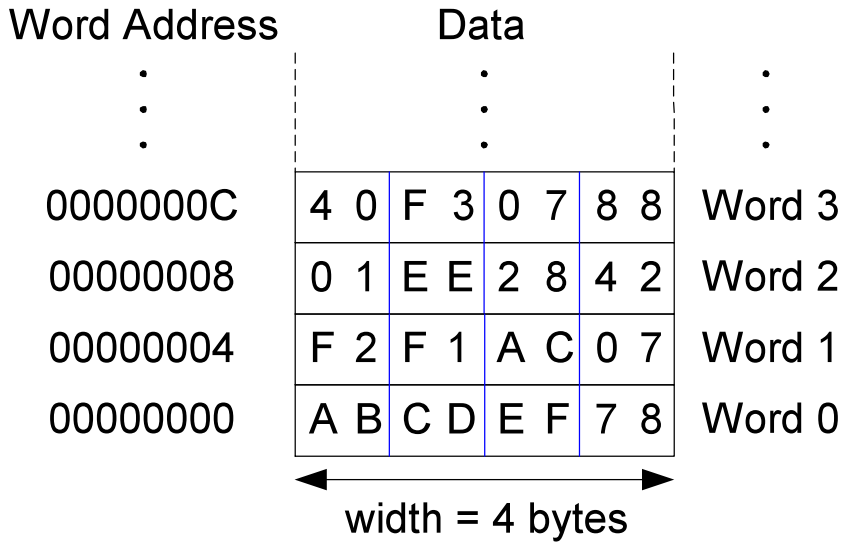
Assembly code

```
sw $t4, 0x7($0)   # write the value in $t4
                  # to memory word 7
```



Byte-addressable memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- Each 32-bit words has 4 bytes, so the word address increments by 4

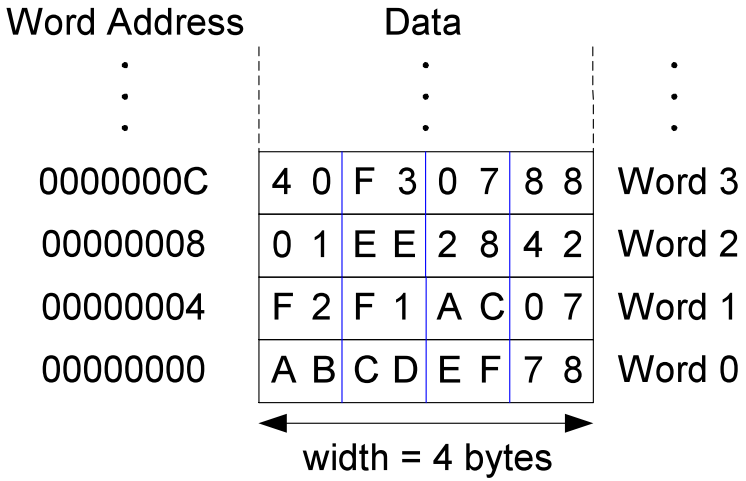


Reading byte-addressable memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)
- Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after the instruction completes.
- MIPS is byte-addressed, not word-addressed. Though in our lab implementation, we can modify it to be word-addressed by customizing the RAM memory blocks

MIPS assembly code

```
lw $s3, 4($0)   # read word at address 4 into $s3
```

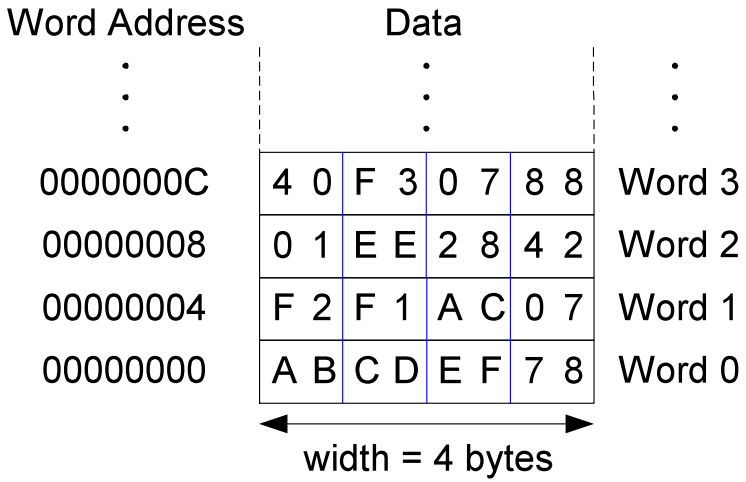


Writing byte-addressed memory

- Example: stores the value held in `$t7` into memory address 0x2C (44)

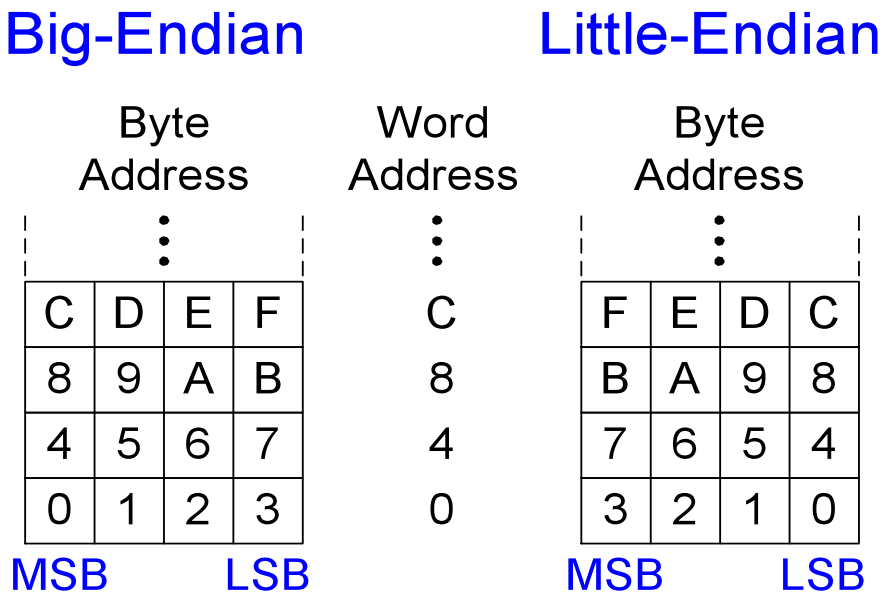
MIPS assembly code

```
sw $t7, 44($0)  # write $t7 into address 44
```



Big-Endian and little-Endian

- 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

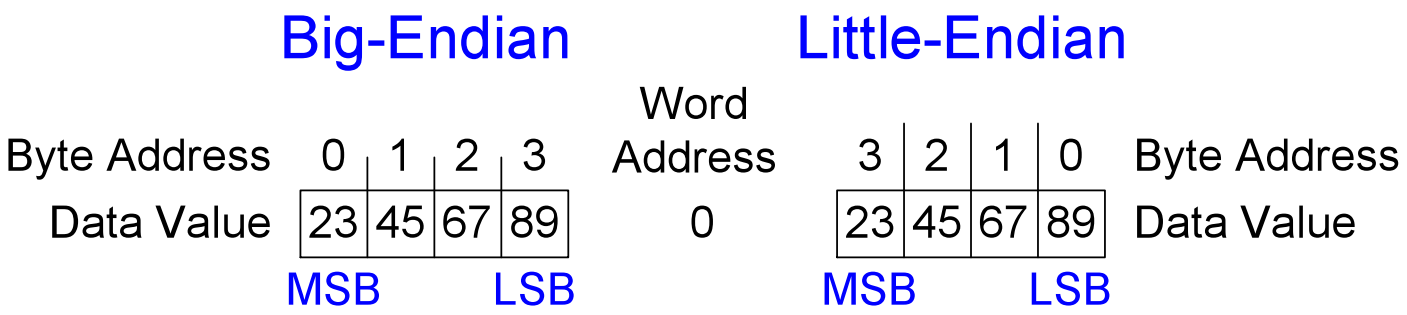


Big- and little- Endian example

- Suppose `$t0` initially contains 0x23456789. After the following program is run on a big-endian system, what value does `$s0` contain? In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Design principle 4 in action

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

Constant/Immediate operands

- lw and sw illustrate the use of constants or *immediates*
- Called immediates because they are *immediate*ly available from the instruction
- Immediates don't require a register or memory access.
- The add immediate (addi) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.
- Is subtract immediate (subi) necessary?

High-level code

```
a = a + 4;
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

I-Type format instructions

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

I-Type

I-Type examples

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Field Values

Note the differing order of registers in the assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	