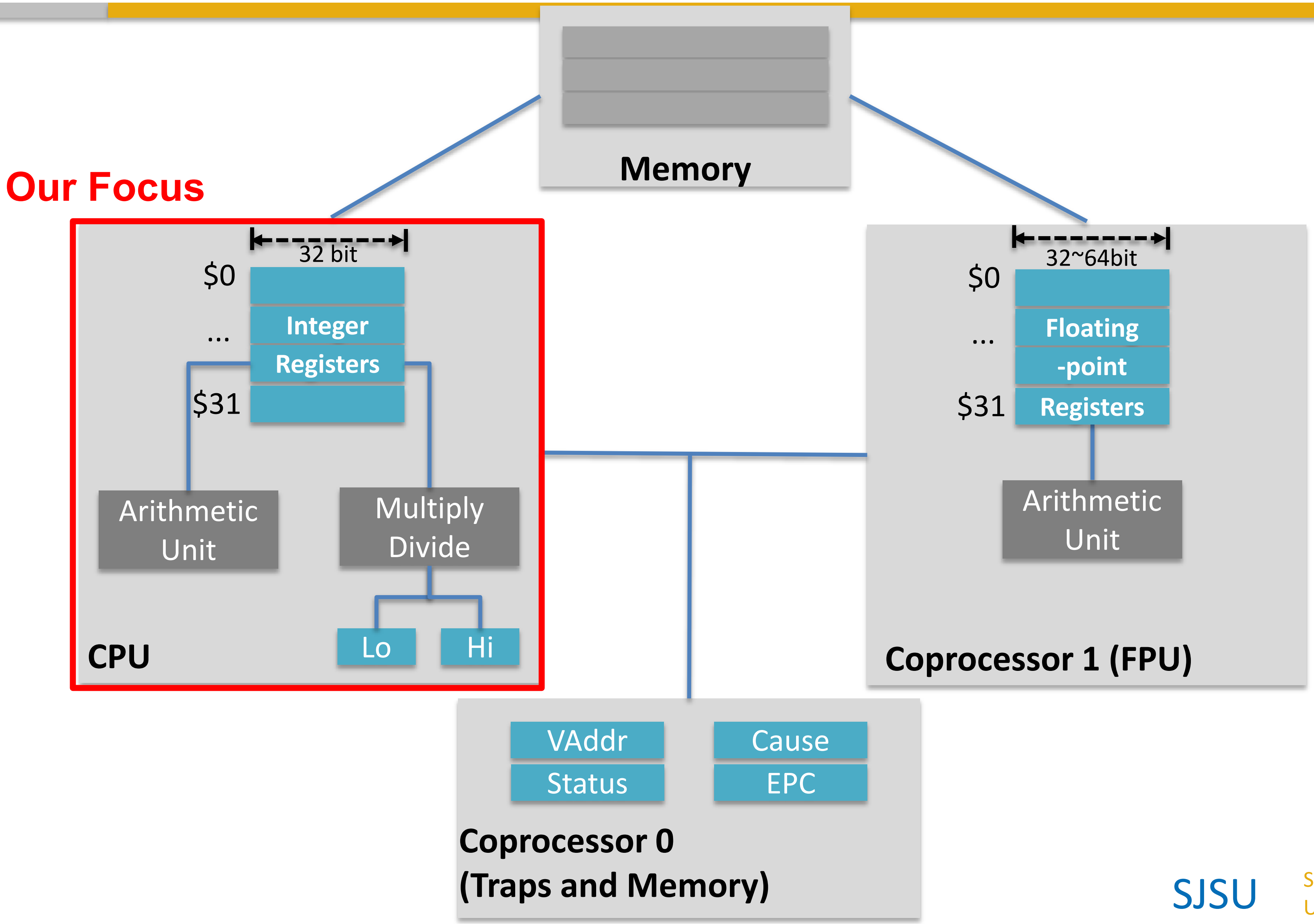
CMPE 200
Computer Architecture & Design

Lecture 2.
Processor Instruction Set Architecture & Language (2)

Haonan Wang

SJSU SAN JOSÉ STATE UNIVERSITY

MIPS Processor Organization



SJSU SAN JOSÉ STATE UNIVERSITY

About MIPS32 Processor

- Instructions are 32-bit wide
- Registers and Computing Logic use 32-bit data
- Memory bus is logically 32-bit wide
- 32 general purpose registers (GPRs) for integer and address values
  - A few special ones (i.e. $zero: constant 0, $fp: frame pointer, $sp: stack pointer..)
- 32 floating point registers for floating point operations (not our focus)

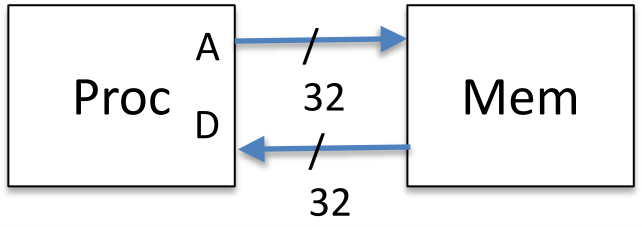
MIPS Data Sizes

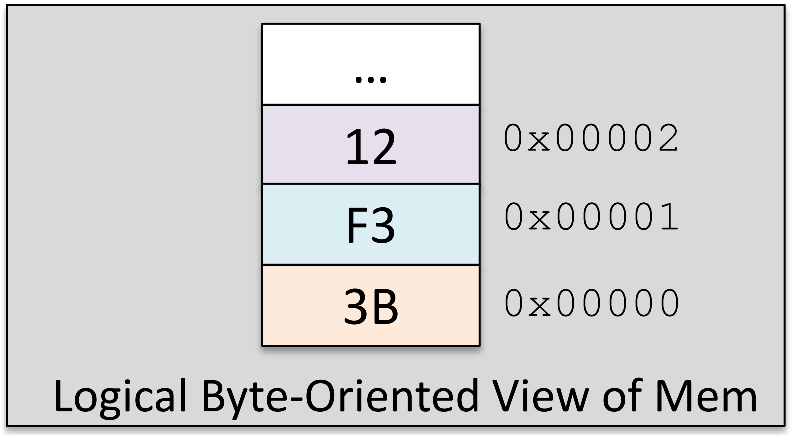
- **Integer:** 3 Sizes Defined
  - **Byte (B)**
    - 8-bits
  - **Halfword (H)**
    - 16-bits = 2 bytes
  - **Word (W)**
    - 32-bits = 4 bytes

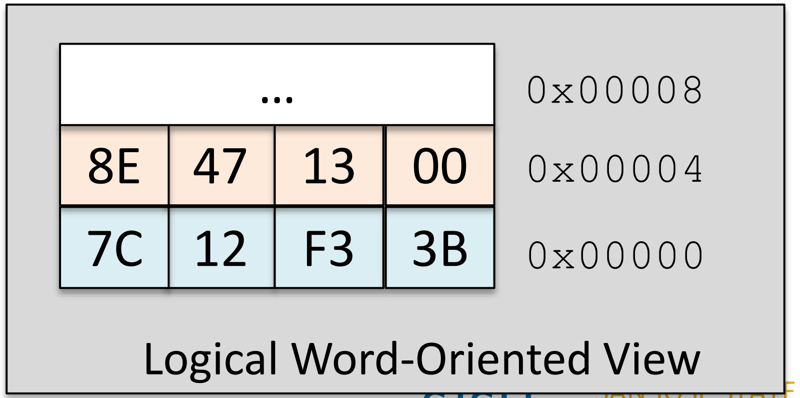
- **Floating-point:** 2 Sizes Defined
  - **Single (S)**
    - 32-bits = 4 bytes
  - **Double (D)**
    - 64-bits = 8 bytes
    - For a 32-bit data bus, a double needs 2 memory reads

Byte-oriented vs. Word-oriented Memory

- **Most processors are byte-oriented**
  - Can access a word from any byte address
- **MIPS: Word-oriented**
  - Words must be aligned to multiples of its size
  - Still byte-addressable!
  - Provides some simplicity in design
- Logical views can be arranged in **rows of 4-bytes** for word-oriented memories



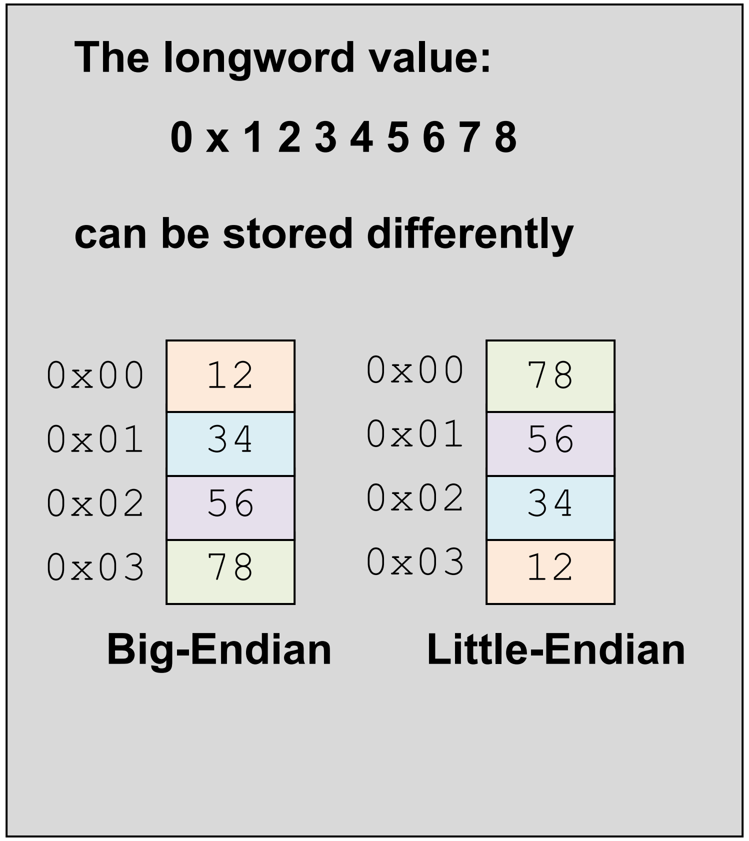




SJSU SAN JOSÉ STATE UNIVERSITY

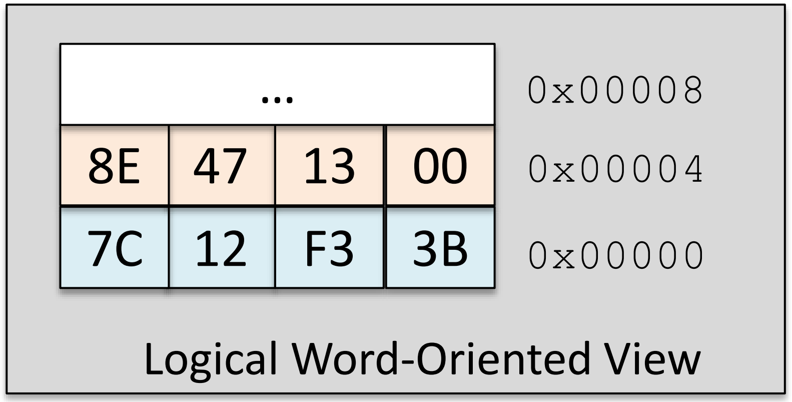
Endian-ness

- **Endian-ness** refers to the two alternate methods of ordering the **bytes** in a larger unit (word, long, etc.)
  - **Big-Endian:** IBM, SPARC, Motorola
    - **Most Significant byte (MSB)** is put at the starting (low) address
  - **Little-Endian:** Intel, DEC
    - **Least Significant byte (LSB)** is put at the starting (low) address
  - Supporting both
    - MIPS, PowerPC, ARM



SJSU SAN JOSÉ STATE UNIVERSITY

Memory Characteristics & Assumptions



- Half-word and Word data are **addressed with lowest byte address** among the bytes in the data
- Addresses from left to right follows the same order as addresses from top to bottom
- We will use Little-Endian for MIPS in this course

Memory Organization Example 1

Example: If the memory layout is given like below

The byte value in address

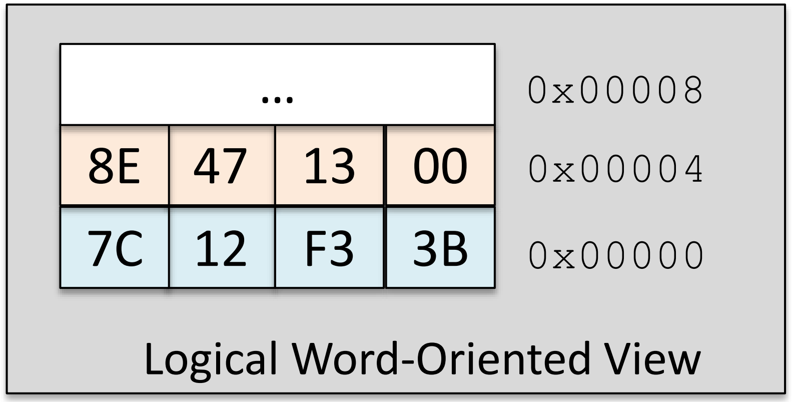
0x0000 is (0x3B)

0x0001 is (0xF3)

0x0002 is (0x12)

0x0003 is (0x7C)

0x0006 is (0x47)



Memory Organization Example 2

Example: If the memory layout in MIPS is given like below

The half-word value in address

0x0000 is (0xF33B)

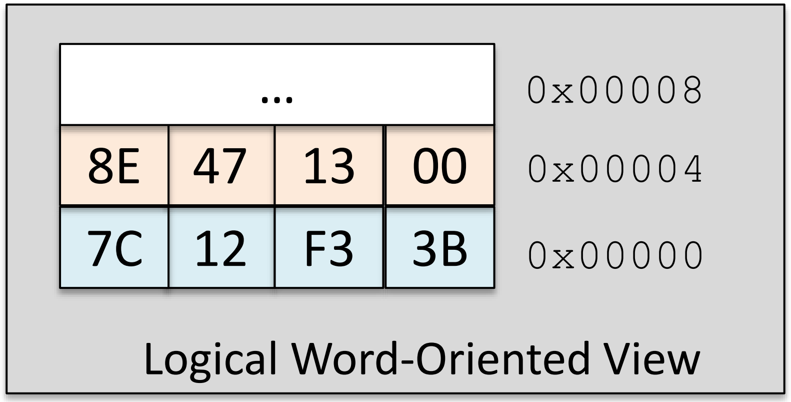
0x0001 is ( ) Incorrect addressing

0x0002 is (0x7C12)

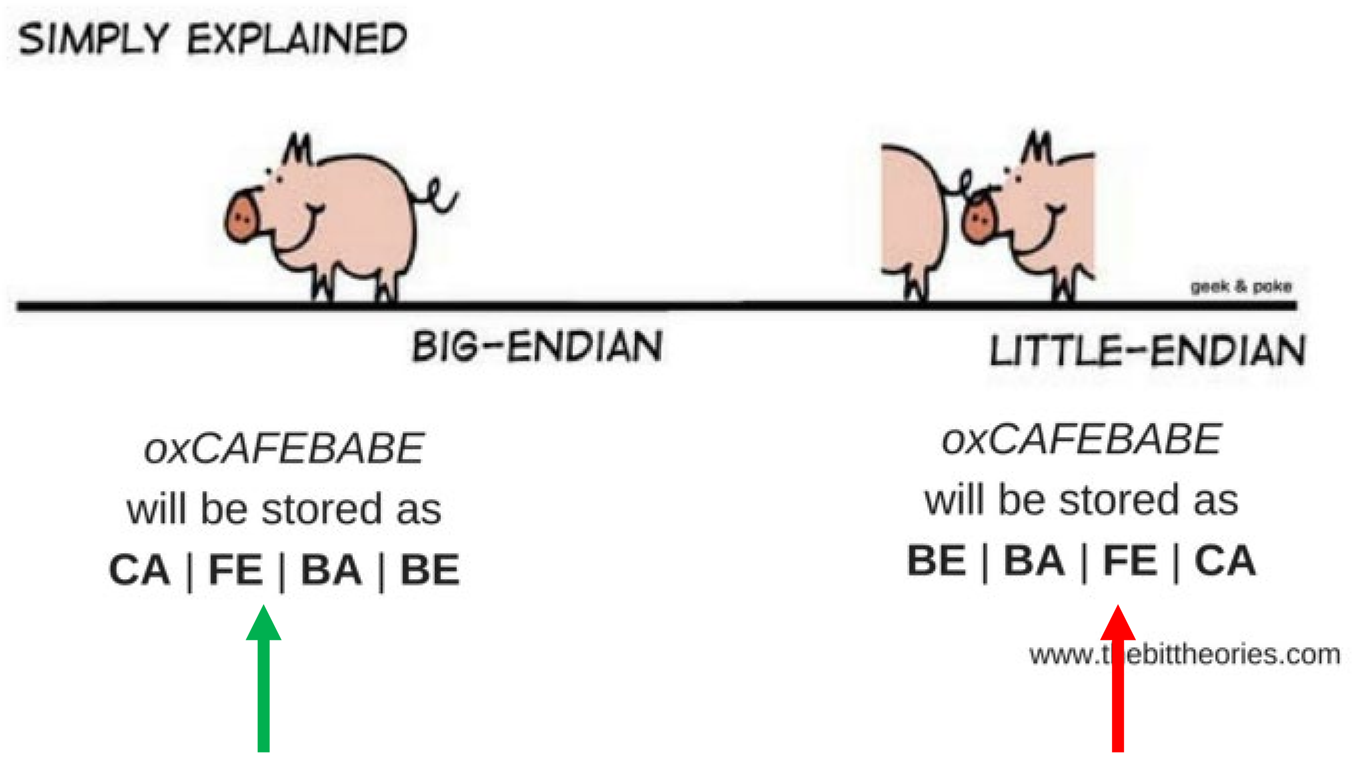
The word value in address

0x0002 is ( ) Incorrect addressing

0x0004 is (0x8E471300)

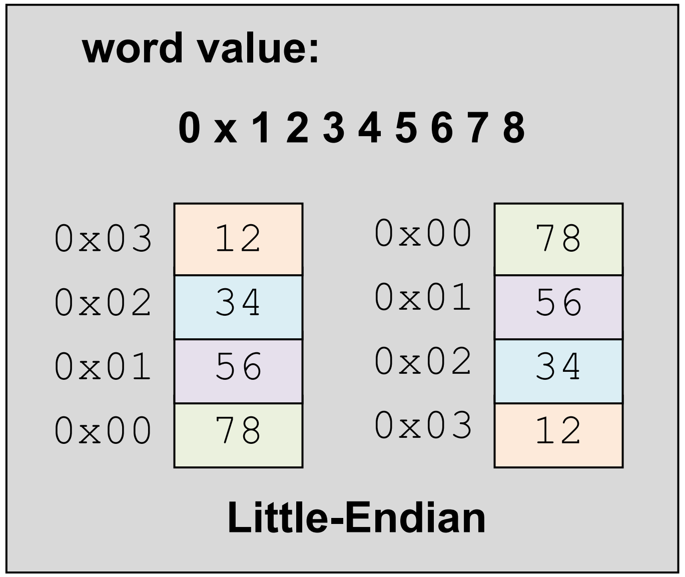


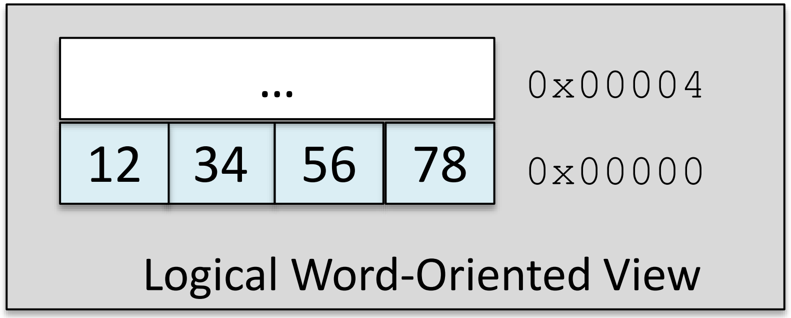
Tips to Remember Endianness



Looks normal (same as writing order) when address is from low to high

Looks strange here
But same as writing order when address is from high to low

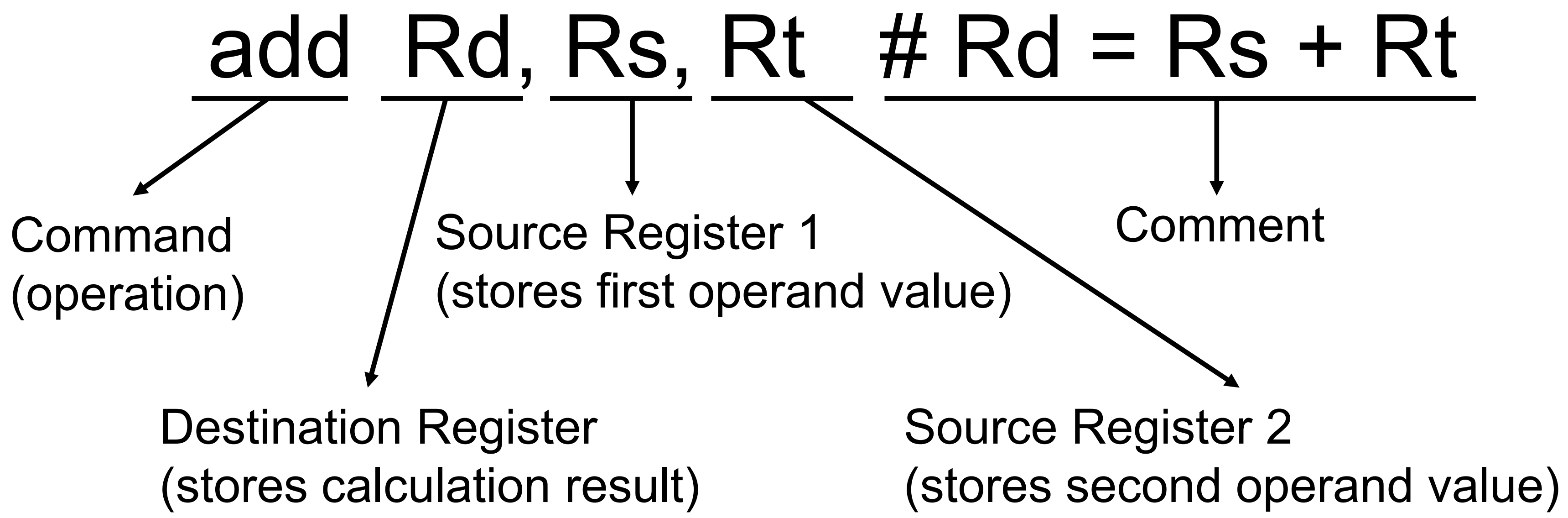




SJSU SAN JOSÉ STATE UNIVERSITY

Basic Instruction Formats

- **Example: Format with 3 registers**


add Rd, Rs, Rt # Rd = Rs + Rt
Command (operation)
Source Register 1 (stores first operand value)
Comment
Destination Register (stores calculation result)
Source Register 2 (stores second operand value)

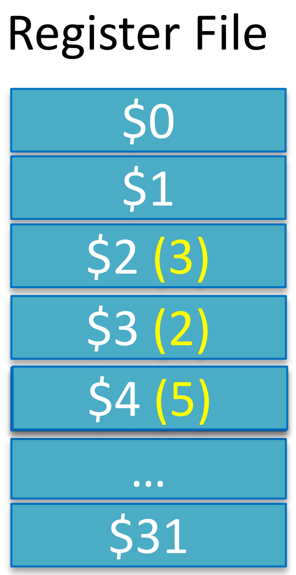
SJSU SAN JOSÉ STATE UNIVERSITY

Basic Instruction Formats

- **Example: Format with 3 registers**

add $4, $3, $2 # $4 = $3 + $2

If $2 and $3 have integer values 3 and 2 each,
$4 will have (5) after executing this instruction



R-Type Instructions

- We call the instructions that use three registers as **R-type** Instructions
  - 2 registers as source, 1 register for result

- **Examples:**
  - **sub** Rd, Rs, Rt # Rd = Rs **–** Rt
  - **mul** Rd, Rs, Rt # Rd = Rs ***** Rt
  - **and** Rd, Rs, Rt # Rd = Rs **&** Rt
  - **or** Rd, Rs, Rt # Rd = Rs **|** Rt
  - **xor** Rd, Rs, Rt # Rd = Rs **^** Rt
  - **slt** Rd, Rs, Rt # **if** (Rs **<** Rt) Rd **= 1**; **else** Rd **= 0**;
  - many more

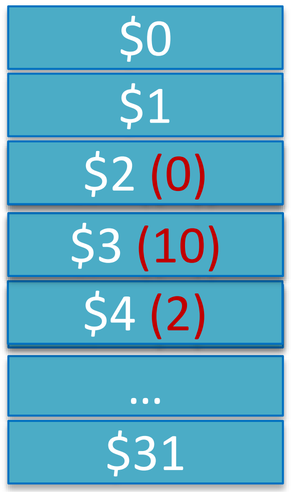
Replace Rd, Rs, Rt
with actual registers

R-Type Instructions

- **Exercise:**
  - Assume that the initial state of register file is like below
  - What is $2 value after executing all three instructions?

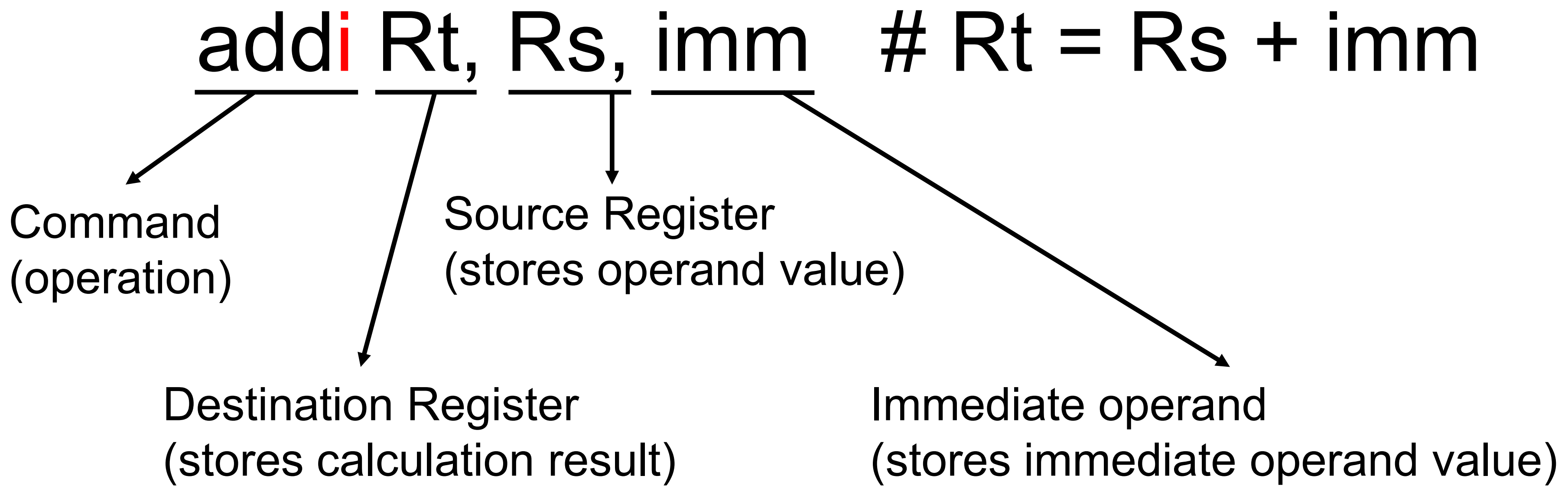
```
sub $4, $2, $3        # 2 = 10 - 8
add $3, $3, $4        # 10 = 8 + 2
slt  $2, $3, $4       # (10 < 2)? →No
                        → $2 = 0
```

Register File



Instructions Using Immediate Value

- What if you want to use an immediate value?
  - E.g. i = i + 1; // add immediate value 1 to i



```
addi Rt, Rs, imm   # Rt = Rs + imm
```

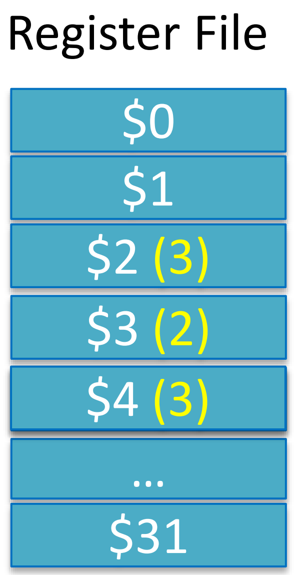
SJSU SAN JOSÉ STATE UNIVERSITY

Instructions Using Immediate Value

- **Example: Instruction format with immediate value**

addi $4, $3, 1 # $4 = $3 + 1

If $3 has integer value 2,
$4 will have (3) after executing this instruction



SJSU SAN JOSÉ STATE UNIVERSITY

I-Type Instructions

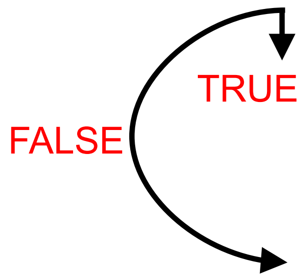
- We call the instructions that use two registers and one immediate value as **I-type** Instructions
  - 1 register and 1 immediate value as source, 1 register for result

- **Examples:**
  - **subi** Rt, Rs, imm # Rt = Rs **–** imm
  - **andi** Rt, Rs, imm # Rt = Rs **&** imm
  - **ori** Rt, Rs, imm # Rt = Rs **|** imm
  - **xori** Rt, Rs, imm # Rt = Rs **^** imm
  - **beq** Rt, Rs, imm # **if** (Rt == Rs) **goto** imm; **else** continue
  - **lw** Rt, imm(Rs) # **load a word** from (imm **+** Rs) address to Rt
  - **slti** Rt, Rs, imm # **if** (Rs **<** imm) Rt **= 1**; **else** Rt **= 0**;
  - many more

I-Type: Branch Instructions

- **Conditional Branches**
  - Branches only if a particular condition is true
    - E.g., Compares Rs, Rt. If EQ/NE, branch to label, else continue

- **Example:**



```
if ( a == b ) {
        a++;
}
else {
        a--;
}
```

Check condition of if statement
Upon the condition result,
you execute either a++ or a--

SJSU SAN JOSÉ STATE UNIVERSITY

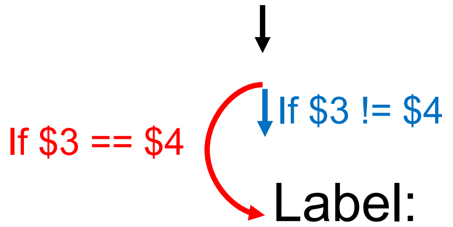
I-Type: Branch Instructions

– **Conditional Branches**
  - Branches only if a particular condition is true
    – E.g., Compares Rs, Rt. If EQ/NE, branch to label, else continue

– **Example:**

```
sub $4, $2, $3
add $3, $3, $4
slt    $2, $3, $4
```

Without branch, instructions are executed sequentially; line by line



```
        sub $4, $2, $3
        beq $3, $4, Label
        slt    $2, $3, $4
Label:  addi $3, $3, 1
```

Branch instruction checks the condition and choose either executing next line or jumping to the specified line

I-Type: Branch Instructions

- **Conditional Branches**
  - Branch if condition satisfies
  - **beq** Rs, Rt, imm # **if** (Rs == Rt) **goto** imm; **else** continue
  - **bne** Rs, Rt, imm # **if** (Rs **!=** Rt) **goto** imm; **else** continue

- **Unconditional Branches**
  - Always branch to label
  - **b** imm # **goto** imm
  - beq $0, $0, imm # **goto** imm

Comparing the same register values
→ always equal

Pseudo-instruction
Instruction "b imm" is not actually supported by the hardware.
Assembler replaces "b imm" to "beq $0, $0, imm"
check more pseudo instructions in textbook

I-Type: Memory Instructions

- **Memory operations use special format to present memory address**

- **Example:**

lw Rt, imm(Rs) # Load a word from address (Rs + imm) to Rt

Command
(operation)

Destination Register
(stores loaded value)

Immediate value
(stores offset value of
the target memory address)

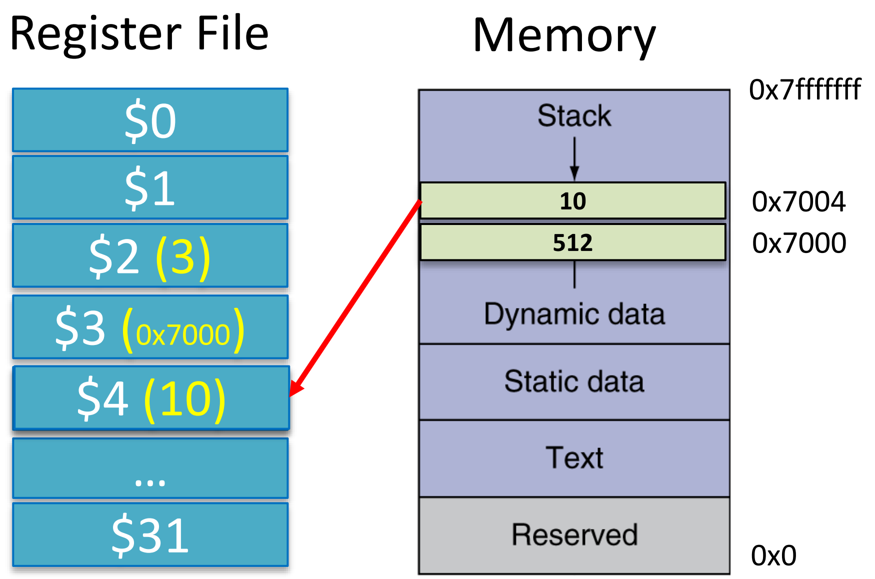
Source operand
(stores base address of
the target memory address)

I-Type: Memory Instructions

- **Memory Instruction Example:**

lw $4, 4($3) # $4 = a word data in ($3 + 4)

If the initial state of register file and memory is like illustrated,
a word in (0x7004) will be loaded to $4 so
$4 will have (10) after executing this instruction



I-Type: Memory Instructions

- **Load: move data from memory to register file**
  - **lb** Rt, imm(Rs) # Rt = 1-byte data in (Rs + imm)
  - **lh** Rt, imm(Rs) # Rt = 2-byte (half-word) data in (Rs + imm)
  - **lw** Rt, imm(Rs) # Rt = 4-byte (word) data in (Rs + imm)

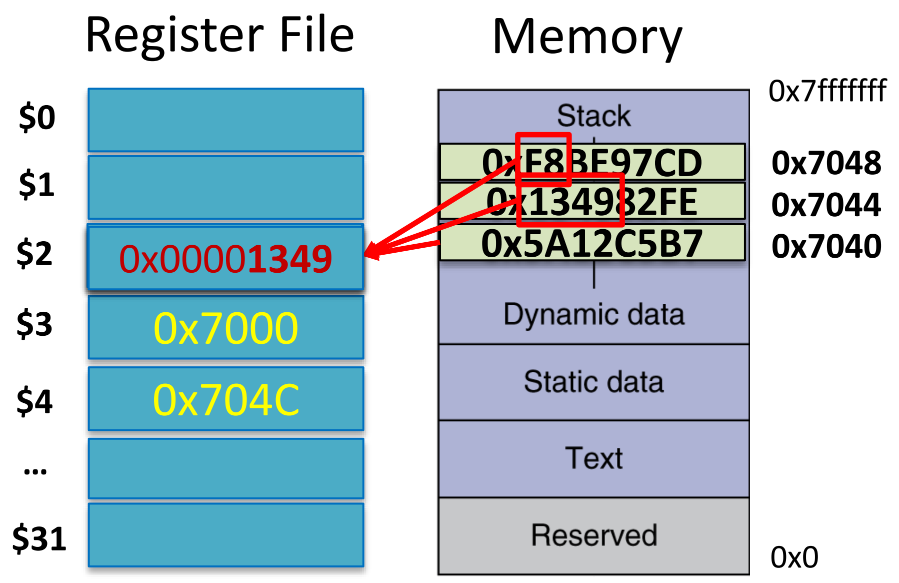
- **Store: move data from register file to memory**
  - **sb** Rt, imm(Rs) # (Rs + imm) = 1-byte data in Rt
  - **sh** Rt, imm(Rs) # (Rs + imm) = 2-byte (half-word) data in Rt
  - **sw** Rt, imm(Rs) # (Rs + imm) = 4-byte (word) data in Rt

I-Type: Memory Instructions

- **Example:**
  - The initial state of register file and memory are like below
  - What is $2 value after executing each of the following instructions?

```
lw    $2, 0x40($3)      # $2 = word in 0x7040
lb    $2, -1($4)        # $2 = byte in 0x704B
lh    $2, -6($4)        # $2 = 2 bytes in 0x7046
```

Note that when executing lb and lh which loads a signed data which is shorter than 4-byte word, the data should be sign-extended to fill 32-bit space of a register.



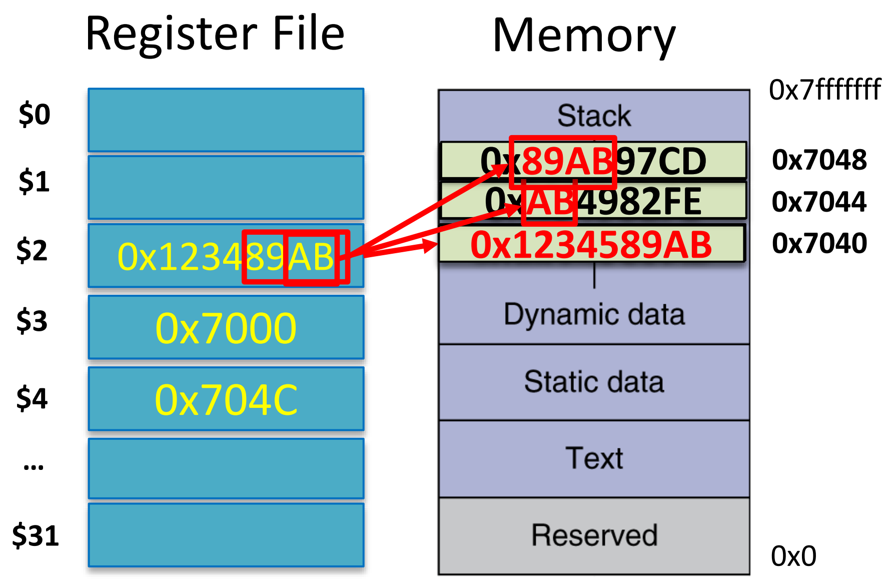
SJSU SAN JOSÉ STATE UNIVERSITY

I-Type: Memory Instructions

- **Example:**
  - The initial state of register file and memory are like below
  - How the memory is updated?

```
sw   $2, 0x40($3)        # 0x7040 = word in $2
sb   $2, -5($4)          # 0x7047 = byte in $2
sh   $2, -2($4)          # 0x704A = 2 bytes in $2
```

When you store 1- to 2- bytes to memory, other values in the same word should remain unchanged



Signed vs. Unsigned Instructions

- **We use sign extension for lb and lh**
  - This is because lb and lh are signed instructions

- **Unsigned instructions**
  - No need to do sign extension because the values are regarded as positive values always; just fill zeros to the remaining bytes
  - lbu $2, -1($4)   # load byte unsigned

    # If lb in the earlier example is lbu,

    # $2 will be updated with **0x000000F8**
  - lhu $2, -6($4) # load half-word unsigned

    # If lh in the earlier example is lhu,

    # $2 will be updated with **0x00001349**

Signed vs. Unsigned Instructions

- **Other instructions also have unsigned versions**
  - addu, addiu, subu, divu, multu, …

- **Example:**
  - Assume that \$2 = 0xFFFFFFFF, \$3 = 0x00000001, what is \$1 in each instruction?

```
–  slt     $1, $2, $3     # signed set less than
                          # -1 < +1 ➔ $1 = 1

–  sltu    $1, $2, $3     # unsigned set less than
                          # +4,294,967,295 > +1 ➔ $1 = 0
```

SJSU SAN JOSÉ STATE UNIVERSITY

Exercise-1

- **Translate the given high-level language code to MIPS assembly.**
  - Assume that the address of integer variables x, y, and z are in 0x400, 0x404, and 0x408 in the memory, respectively
  - $1's initial value is 0x400

High-level language

```
x = y + z;
```

Tasks to do:
1) Load y value to a register
2) Load z value to another register
3) Run add operation
4) Store the result to memory

MIPS

```
lw   $2, 4($1)
lw   $3, 8($1)
add  $2, $2, $3
sw   $2, 0($1)
```

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY