CMPE 200
Computer Architecture & Design

Lecture 2.
Processor Instruction Set Architecture & Language

Haonan Wang

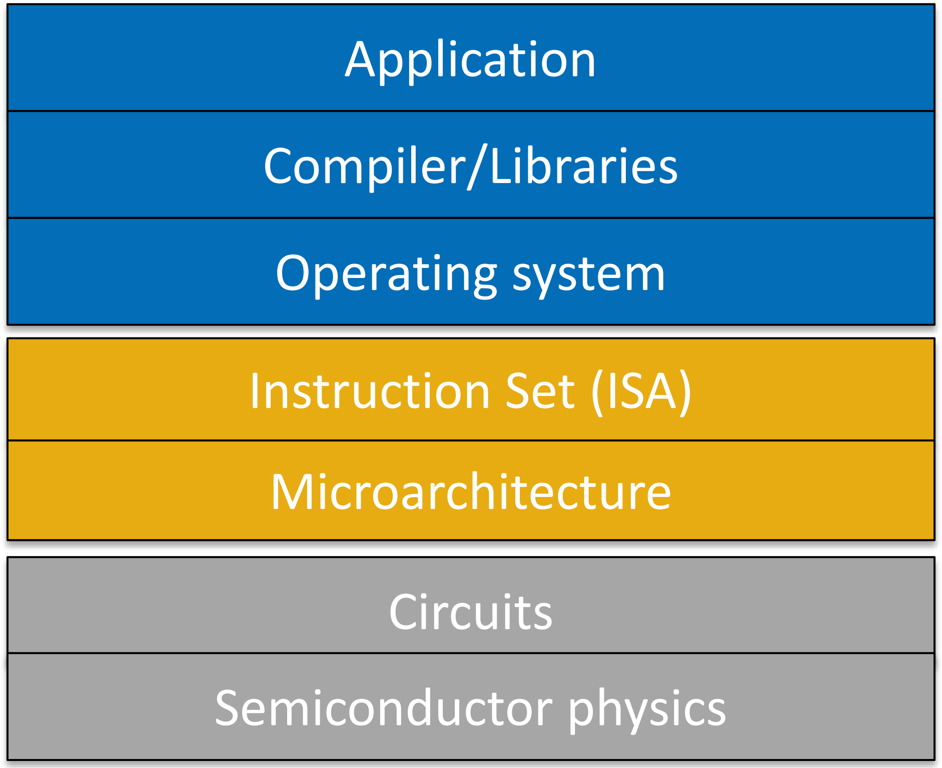
SJSU

SAN JOSÉ STATE
UNIVERSITY

Instruction Set Architecture & Microarchitecture

- **Instruction Set Architecture**
  - the programmer’s view of the computer, defined by a set of instructions that each specifies an operation and its operand(s)

- **Microarchitecture**
  - the way that the ISA is implemented in hardware



Application

Compiler/Libraries

Operating system

Instruction Set (ISA)

Microarchitecture

Circuits

Semiconductor physics

What Does the ISA Deal With Specifically?

Example: a C program that reads two integer values from “file.txt” file and prints the sum of them.

```
#include <stdio.h>
#include <string.h>

int numbers[2];

void myfunction(void)
{
   FILE *fp;
   int size = 2;
   int sum = 0;

   /* Open file for reading */
   fp = fopen(“mynumbers.txt", “r");

   /* Read and display data */
   fread(numbers, sizeof(int), size, fp);
   fclose(fp);
   sum = numbers[0] + numbers[1];
   printf(“Sum = %d\n”, sum);
}

int main (void) {
   myfunction();
   return(0);
}
```

Processor (CPU)

Understands and executes each line of the code.

Uses fast on-chip memories

Memory (DRAM)

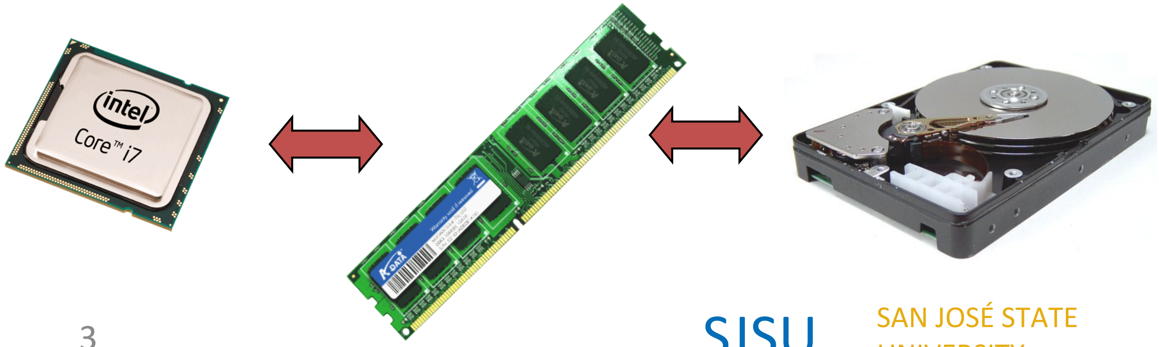
Provides operands to CPU

(*fp, size, sum, numbers[2])

Storage (HDD)

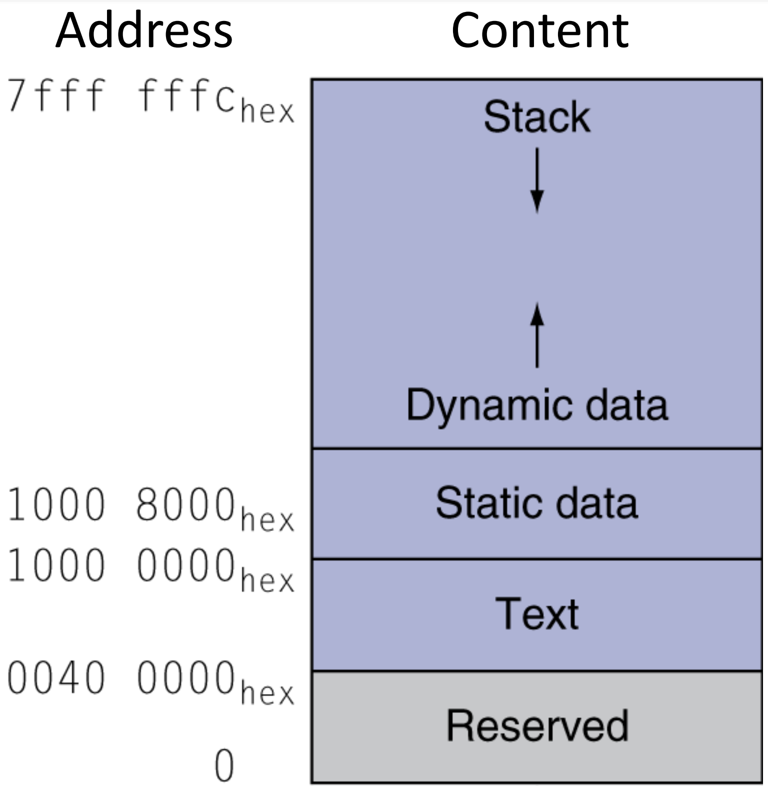
Provides file inputs and program code

(file.txt)



Memory Layout

- **Text:** program code
- **Static data:** global variables
  - e.g., static variables in C, constant arrays and strings
- **Dynamic data:** heap
  - e.g., malloc in C, new in Java
  - Grows from bottom (lower address) to top (higher address)
- **Stack:** temporal storage for functions
  - e.g., return address of sub-functions, local variables
  - Grows from top to bottom



Address

Content

$7fff\ fffc_{hex}$

Stack

Dynamic data

$1000\ 8000_{hex}$

Static data

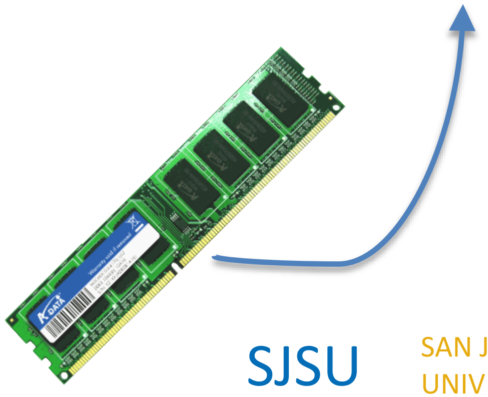
$1000\ 0000_{hex}$

Text

$0040\ 0000_{hex}$

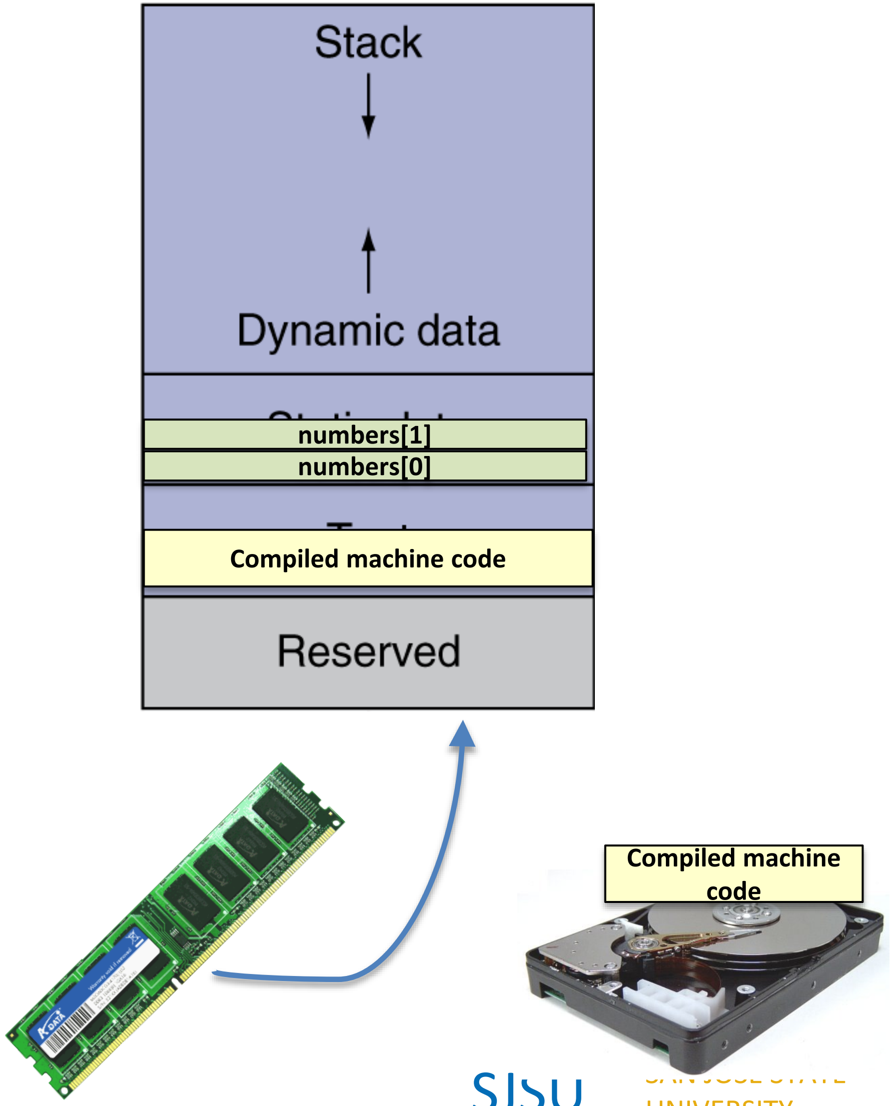
Reserved

0



Memory Layout

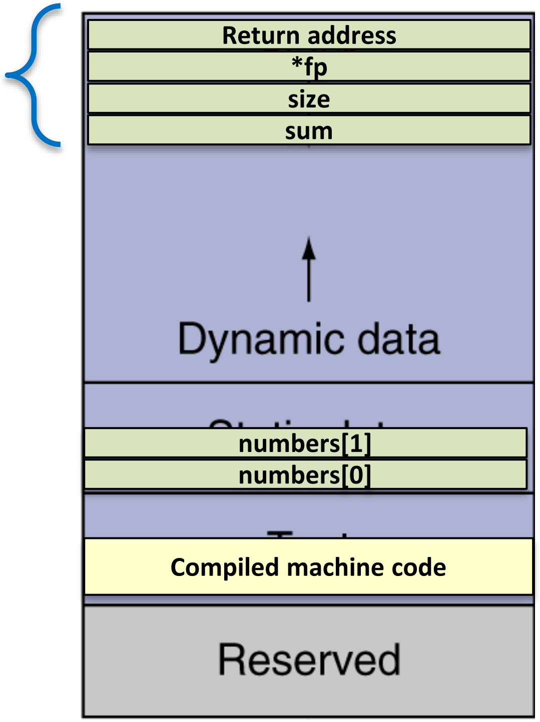
```
#include <stdio.h>
#include <string.h>

int numbers[2];

void myfunction(void)
{
   FILE *fp;
   int size = 2;
   int sum = 0;

   /* Open file for reading */
   fp = fopen(“mynumbers.txt", “r");

   /* Read and display data */
   fread(numbers, sizeof(int), size, fp);
   fclose(fp);
   sum = numbers[0] + numbers[1];
   printf(“Sum = %d\n”, sum);
}

int main (void) {
   myfunction();
   return(0);
}
```

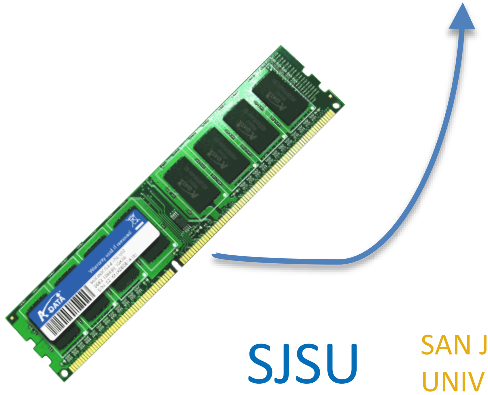


Memory Layout

```
#include <stdio.h>
#include <string.h>

int numbers[2];

void myfunction(void)
{
   FILE *fp;
   int size = 2;
   int sum = 0;

   /* Open file for reading */
   fp = fopen(“mynumbers.txt", “r");

   /* Read and display data */
   fread(numbers, sizeof(int), size, fp);
   fclose(fp);
   sum = numbers[0] + numbers[1];
   printf(“Sum = %d\n”, sum);
}

int main (void) {
   myfunction();
   return(0);
}
```

Stack region for myfunction

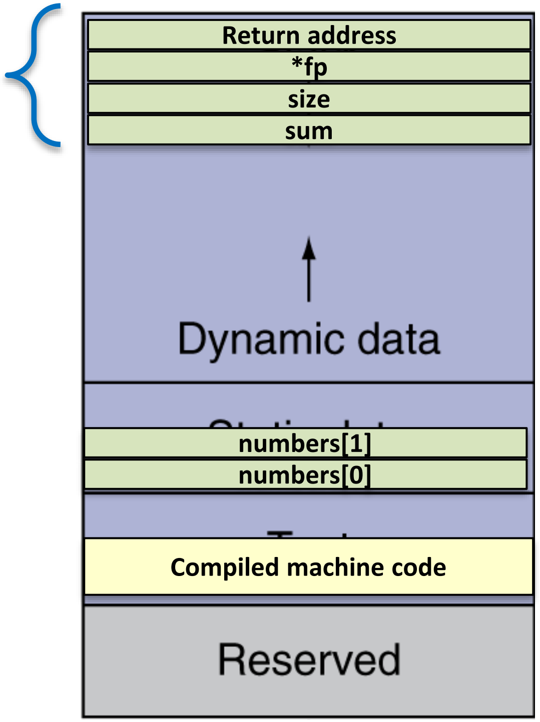


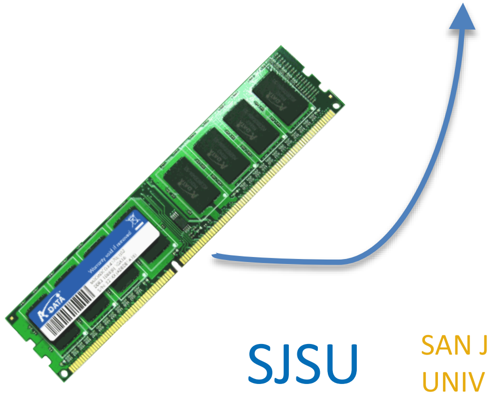


Memory Layout

```
#include <stdio.h>
#include <string.h>

int numbers[2];

void myfunction(void)
{
   FILE *fp;
   int size = 2;
   int sum = 0;

   /* Open file for reading */
   fp = fopen(“mynumbers.txt", “r");

   /* Read and display data */
   fread(numbers, sizeof(int), size, fp);
   fclose(fp);
   sum = numbers[0] + numbers[1];
   printf(“Sum = %d\n”, sum);
}

int main (void) {
   myfunction();
   return(0);
}
```

Stack region for myfunction

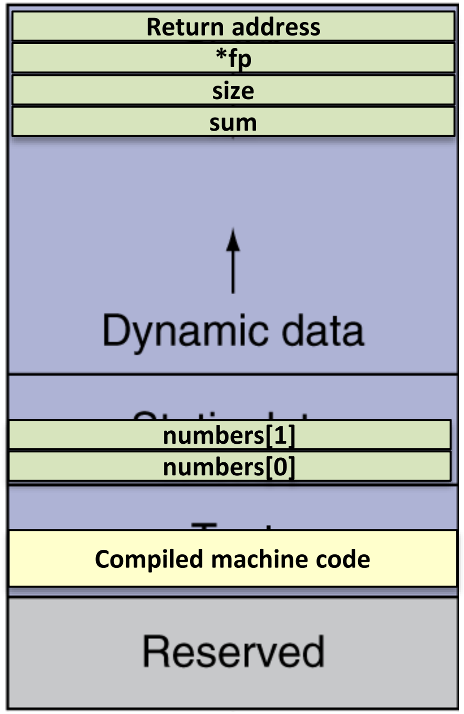


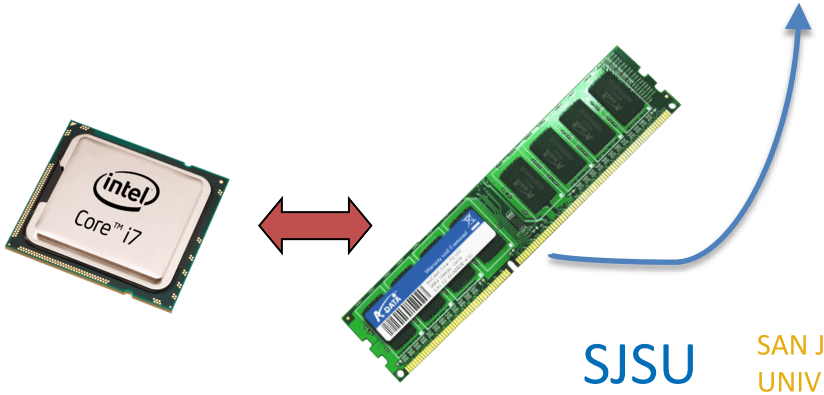


Memory Access Is Slow

- Variables are all stored in Memory, which is **outside of CPU**
  - Slow..

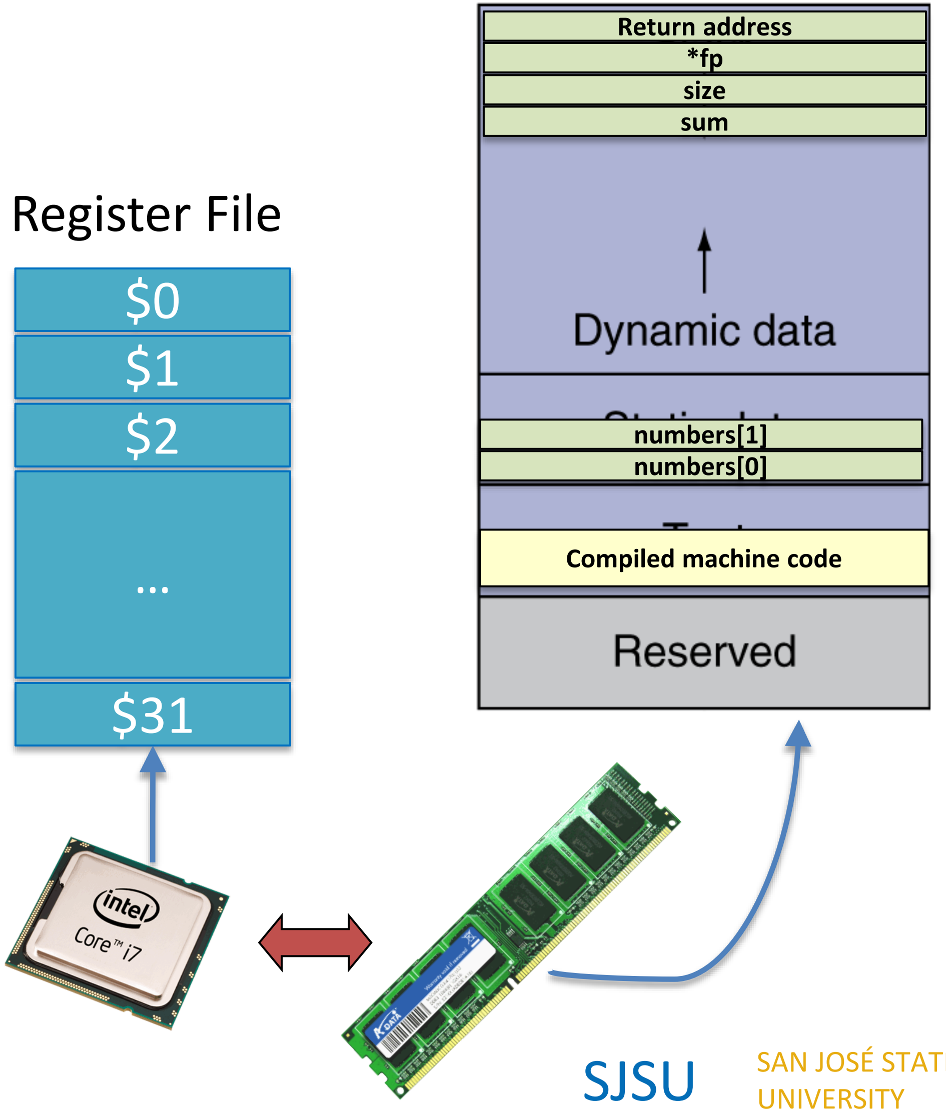
- How can we execute the operations faster?
  - Use **on-chip memory**





Register File

- **Register File**
  - On-chip memory that stores “Registers”
- **Registers**
  - Temporal space to maintain operand values and calculation results before storing back to memory
  - Presented with “$” + “register id” in MIPS processor
    - i.e. $0 : 0th register,
      $1 : 1st register

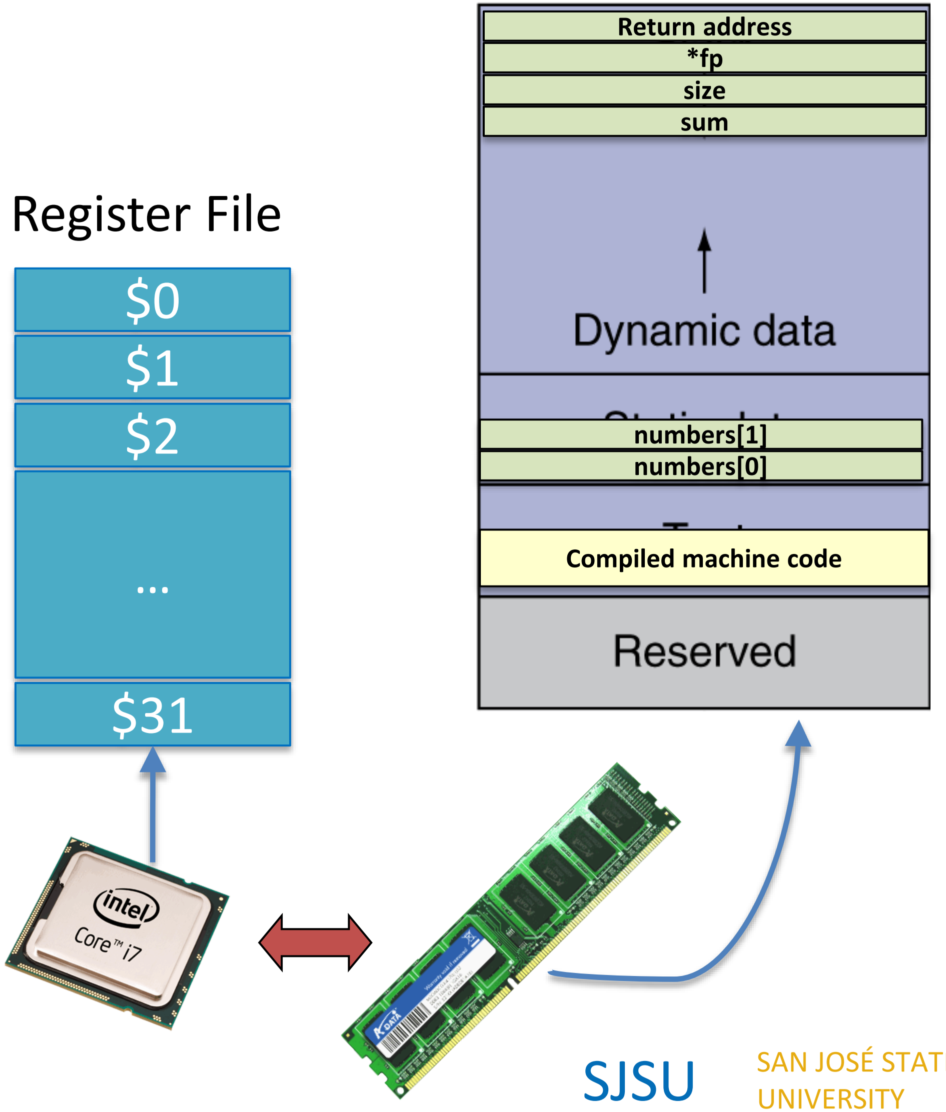


Typical Execution Steps

1. Load operand values from memory to registers

2. Do computation on registers

3. Move the results from register to memory



Register File

$0

$1

$2

…

$31

Return address

*fp

size

sum

Dynamic data

numbers[1]

numbers[0]

Compiled machine code

Reserved

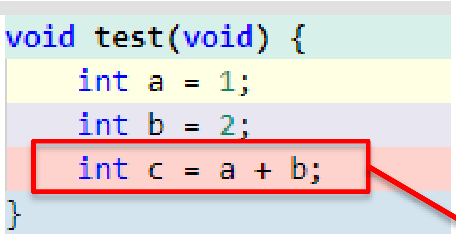
Operations in Hardware

- **Hardware can do one operation at a time**
  - Arithmetic operations
    - add, sub, mult, div, …
  - Data movement
    - move, load data, store data, …
  - Logical operations
    - shift, and, or, xor, …
  - Conditional operations
    - jump, branch on condition, …

- Format of assembly instructions that uses register operands
  - ***Command Result, Operand 1, Operand 2***
  - E.g. **C = A + B → Add C, A, B** (A, B, C should be replaced by register id)

Code Example: Memory & Registers

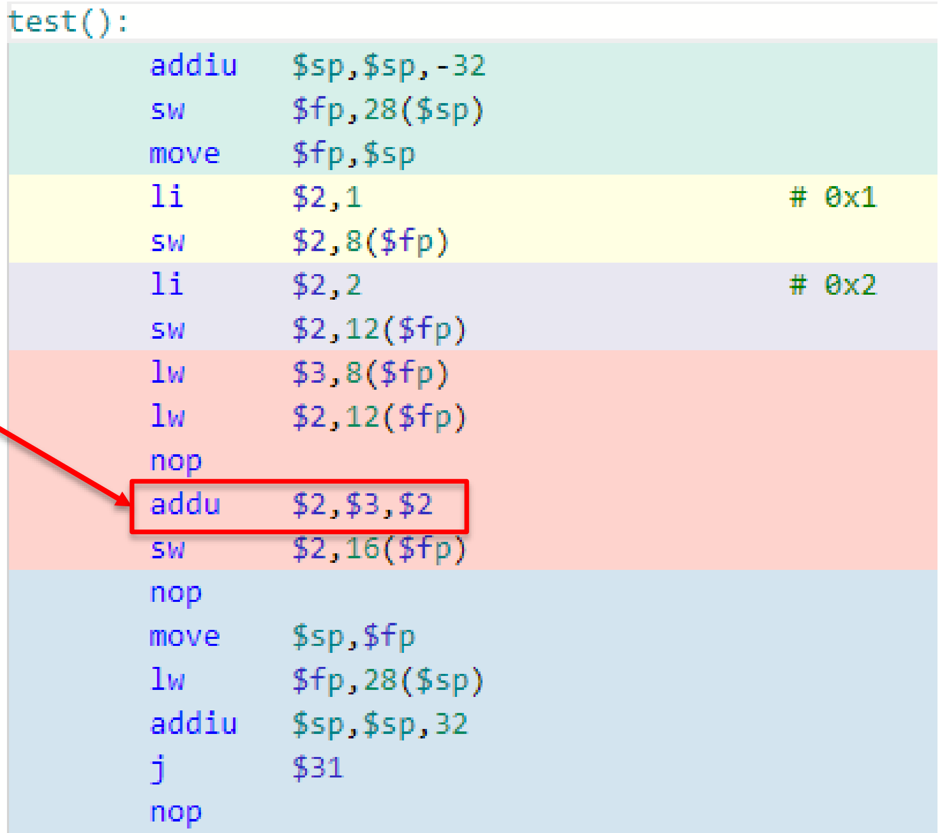
C code



```
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Compile

MIPS Assembly code



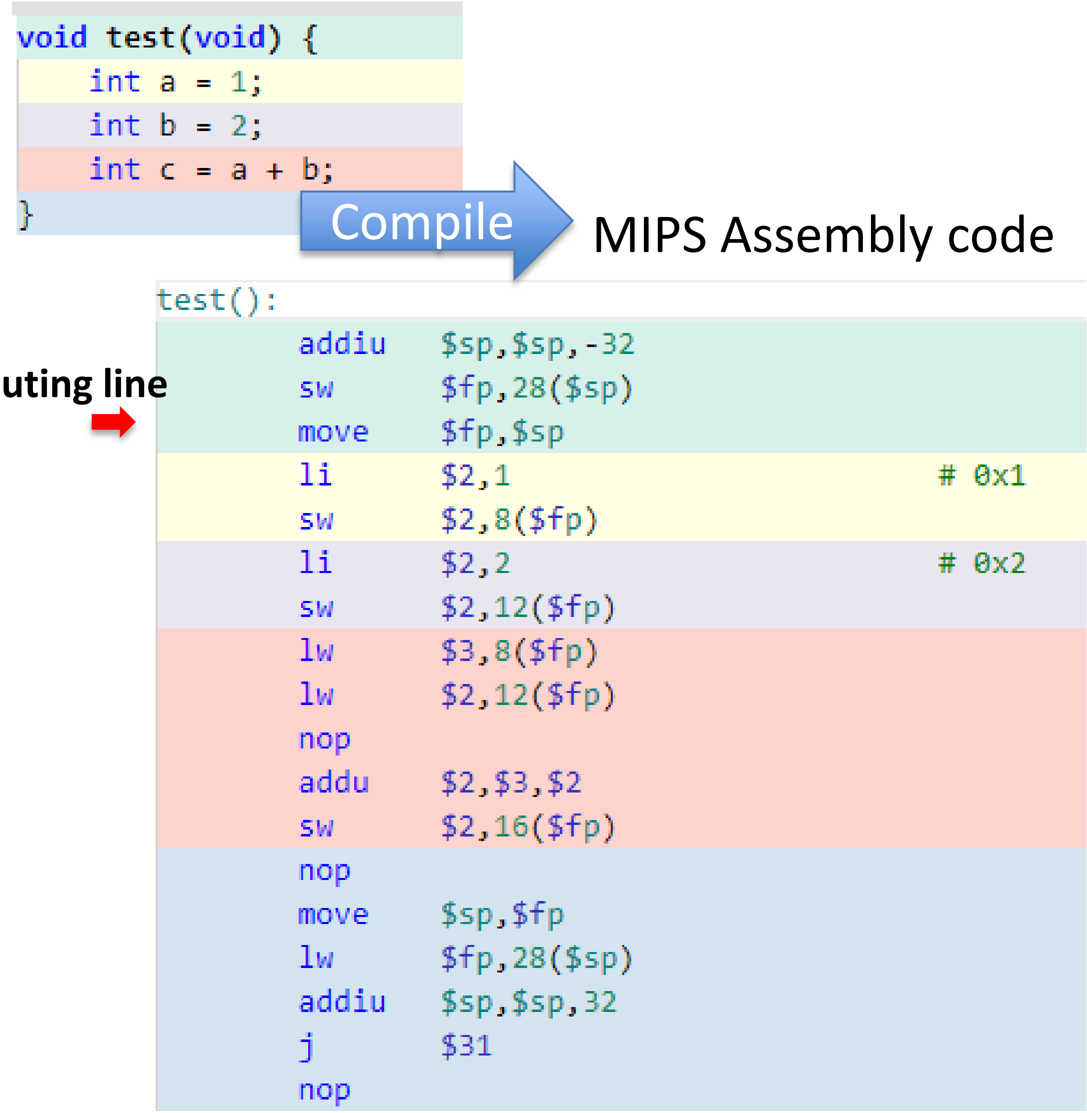
```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                    # 0x1
        sw      $2,8($fp)
        li      $2,2                    # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

Operations involved:

- Value of ‘a’ is loaded from mem to $3
- Value of ‘b’ is loaded from mem to $2
- **Add** operation done on $2 and $3
- Value of ‘c’ is stored to mem

Code Example: Memory & Registers

C code

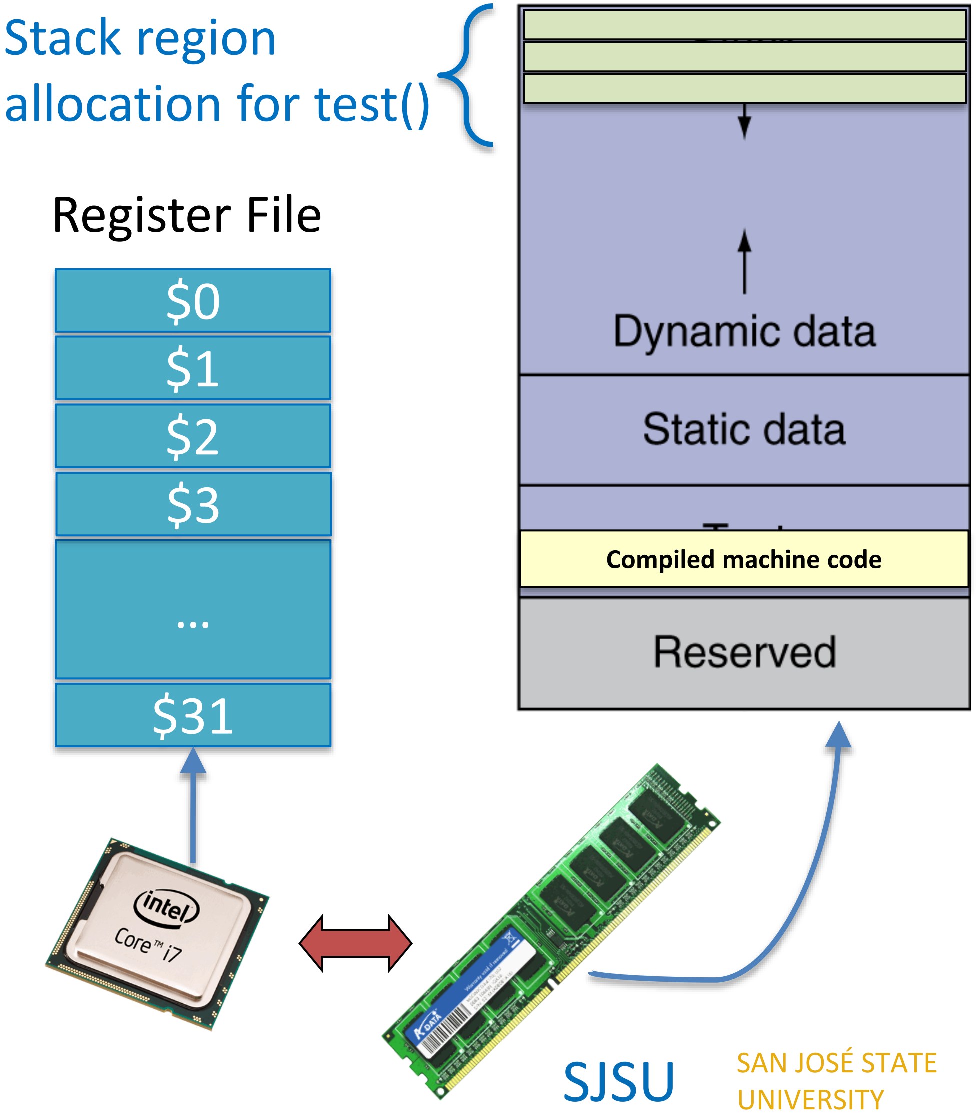


```c
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

MIPS Assembly code

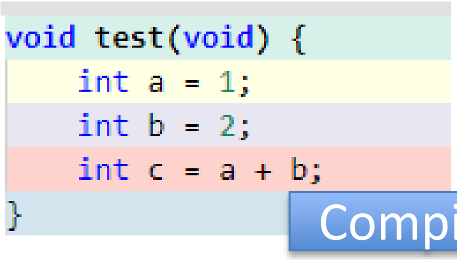
```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                        # 0x1
        sw      $2,8($fp)
        li      $2,2                        # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

executing line



Code Example: Memory & Registers

C code

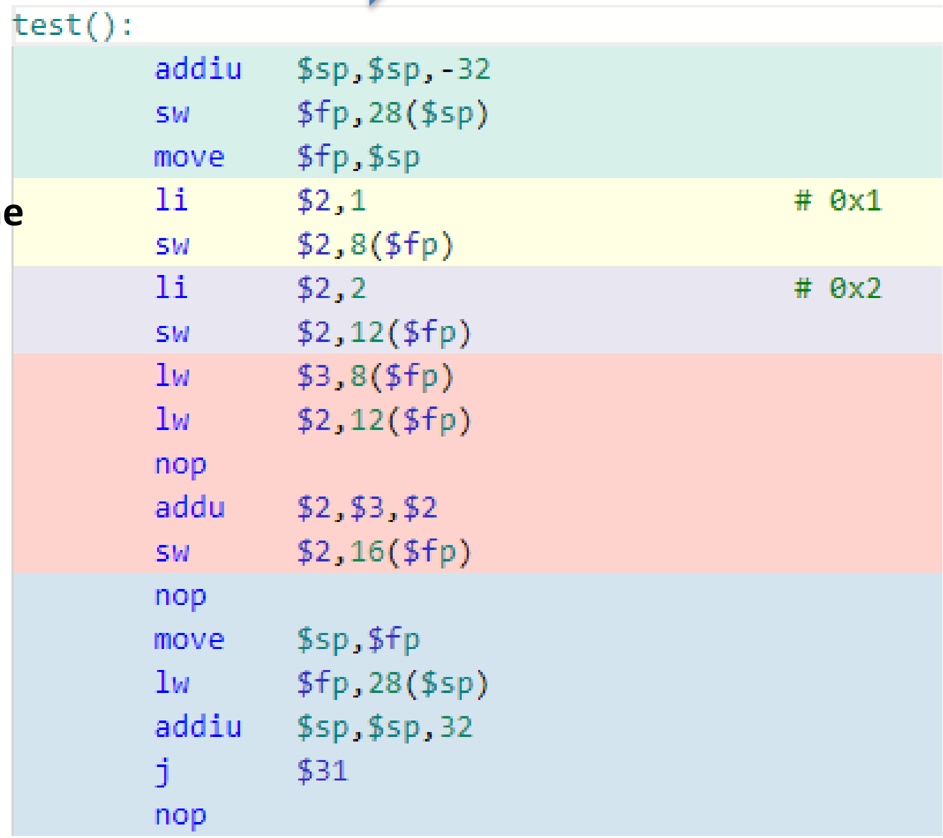


```
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Compile

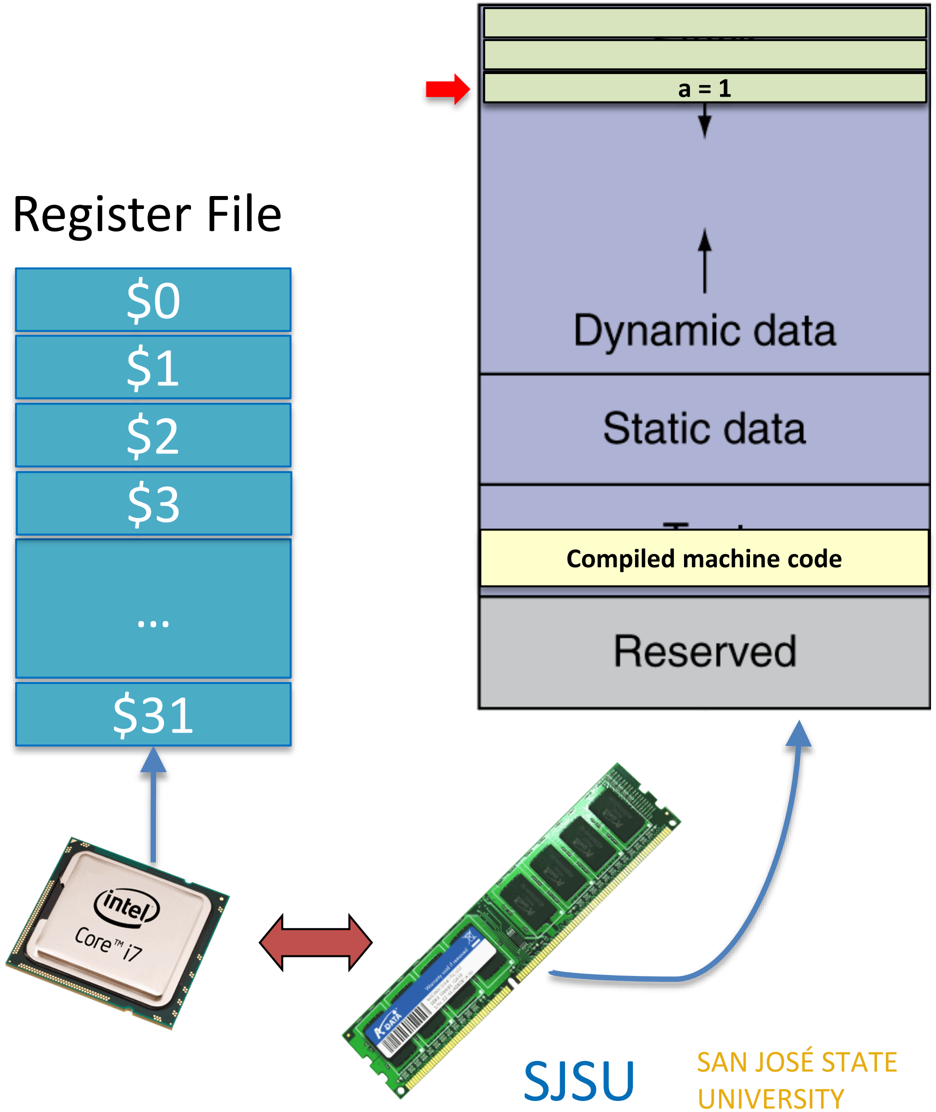
MIPS Assembly code

executing line



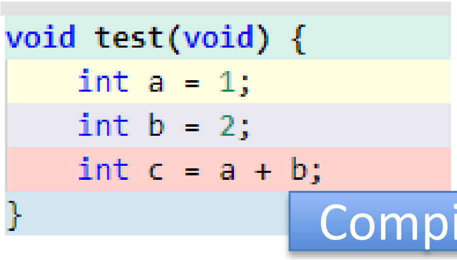
```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                        # 0x1
        sw      $2,8($fp)
        li      $2,2                        # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

Register File



Code Example: Memory & Registers

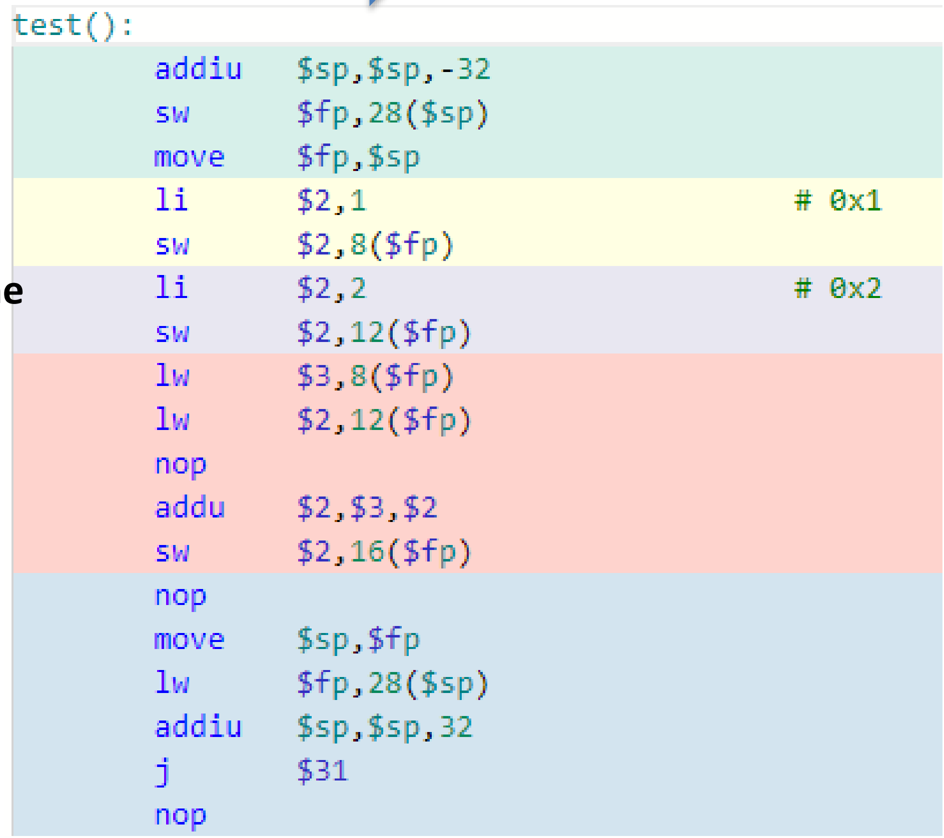
C code



```
void test(void) {
     int a = 1;
     int b = 2;
     int c = a + b;
}
```

Compile

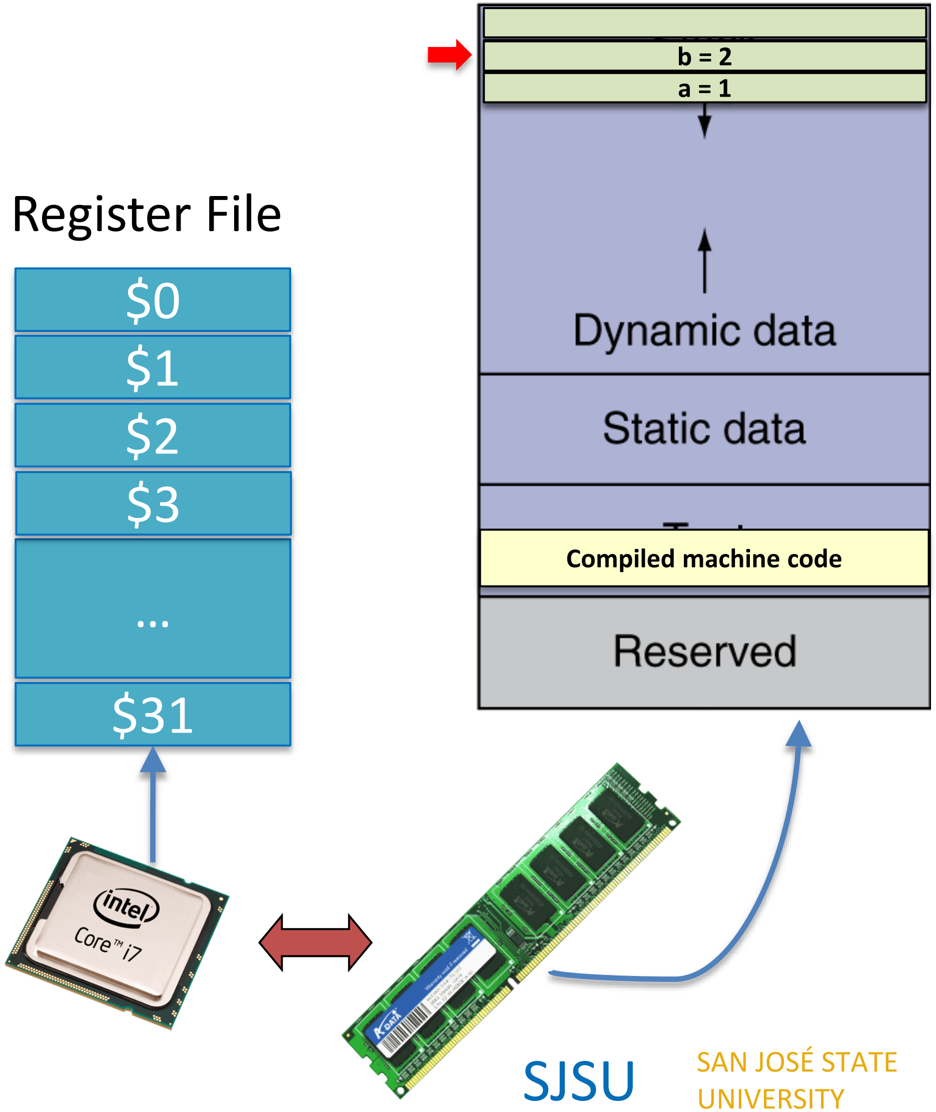
MIPS Assembly code



```
test():
          addiu    $sp,$sp,-32
          sw       $fp,28($sp)
          move     $fp,$sp
          li       $2,1                         # 0x1
          sw       $2,8($fp)
          li       $2,2                         # 0x2
          sw       $2,12($fp)
          lw       $3,8($fp)
          lw       $2,12($fp)
          nop
          addu     $2,$3,$2
          sw       $2,16($fp)
          nop
          move     $sp,$fp
          lw       $fp,28($sp)
          addiu    $sp,$sp,32
          j        $31
          nop
```

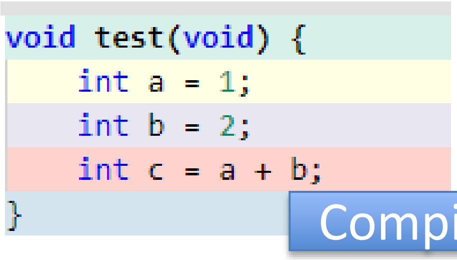
executing line ➡

Register File



Code Example: Memory & Registers

C code

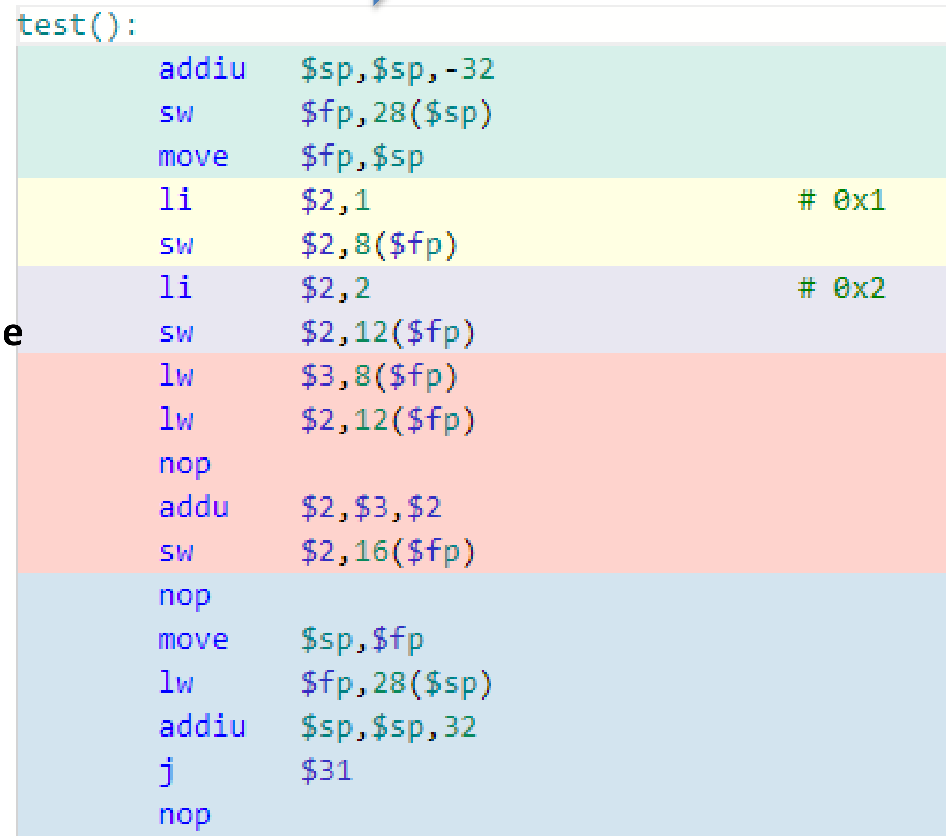


```
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Compile

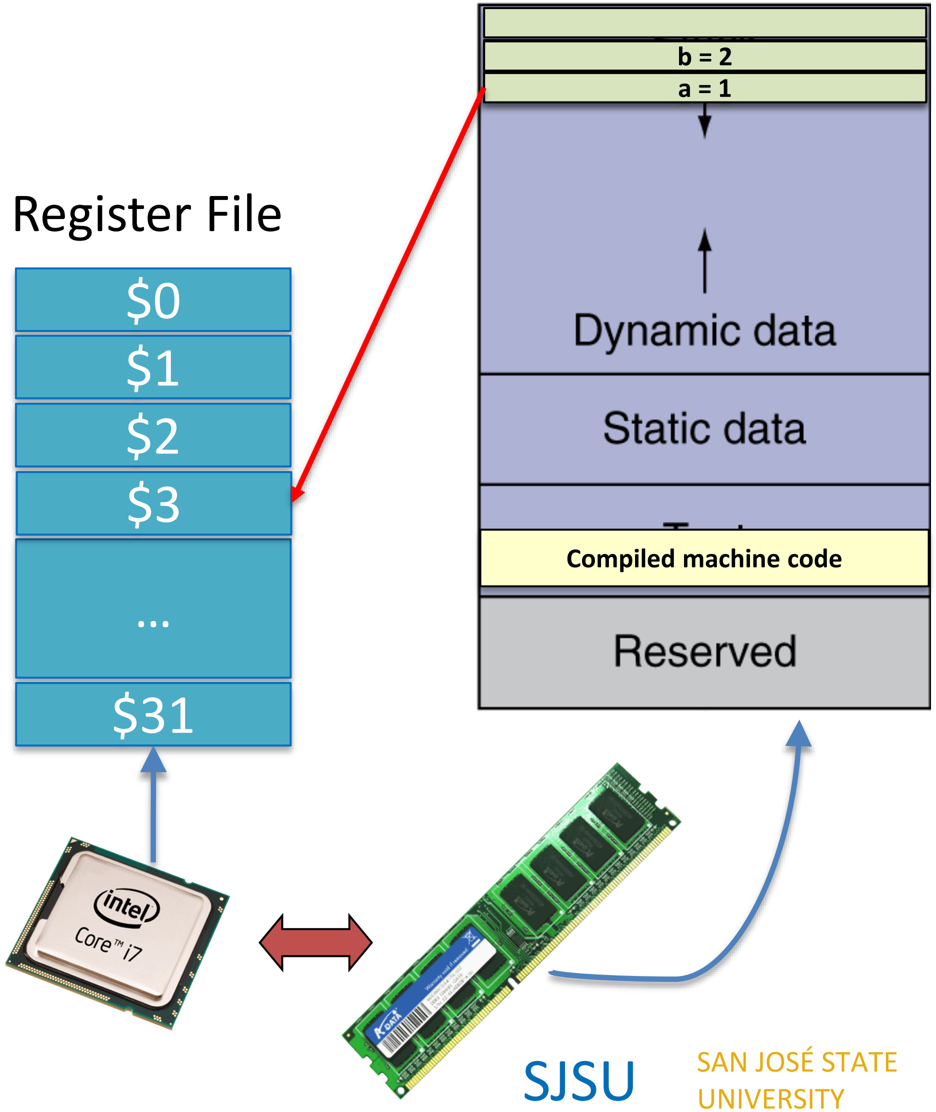
MIPS Assembly code

executing line



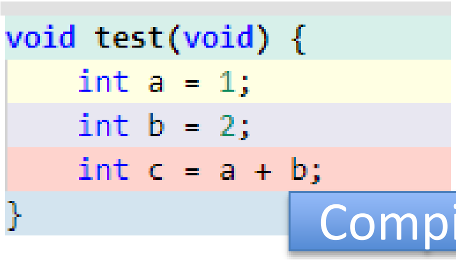
```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                    # 0x1
        sw      $2,8($fp)
        li      $2,2                    # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

Register File



Code Example: Memory & Registers

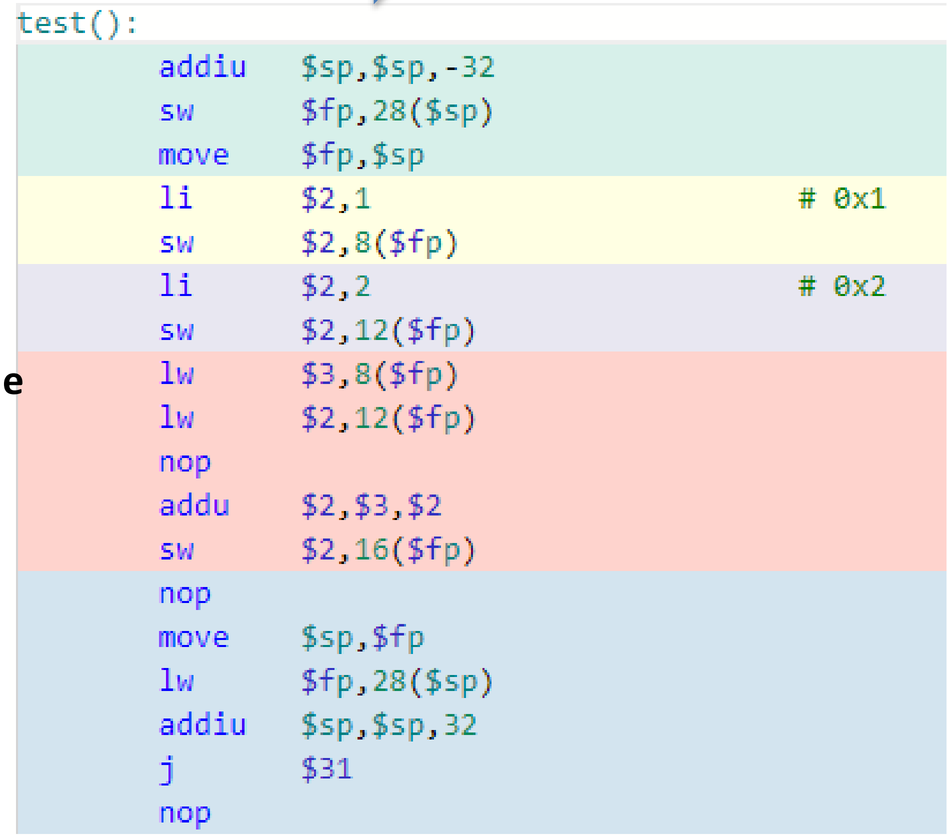
C code



```c
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Compile

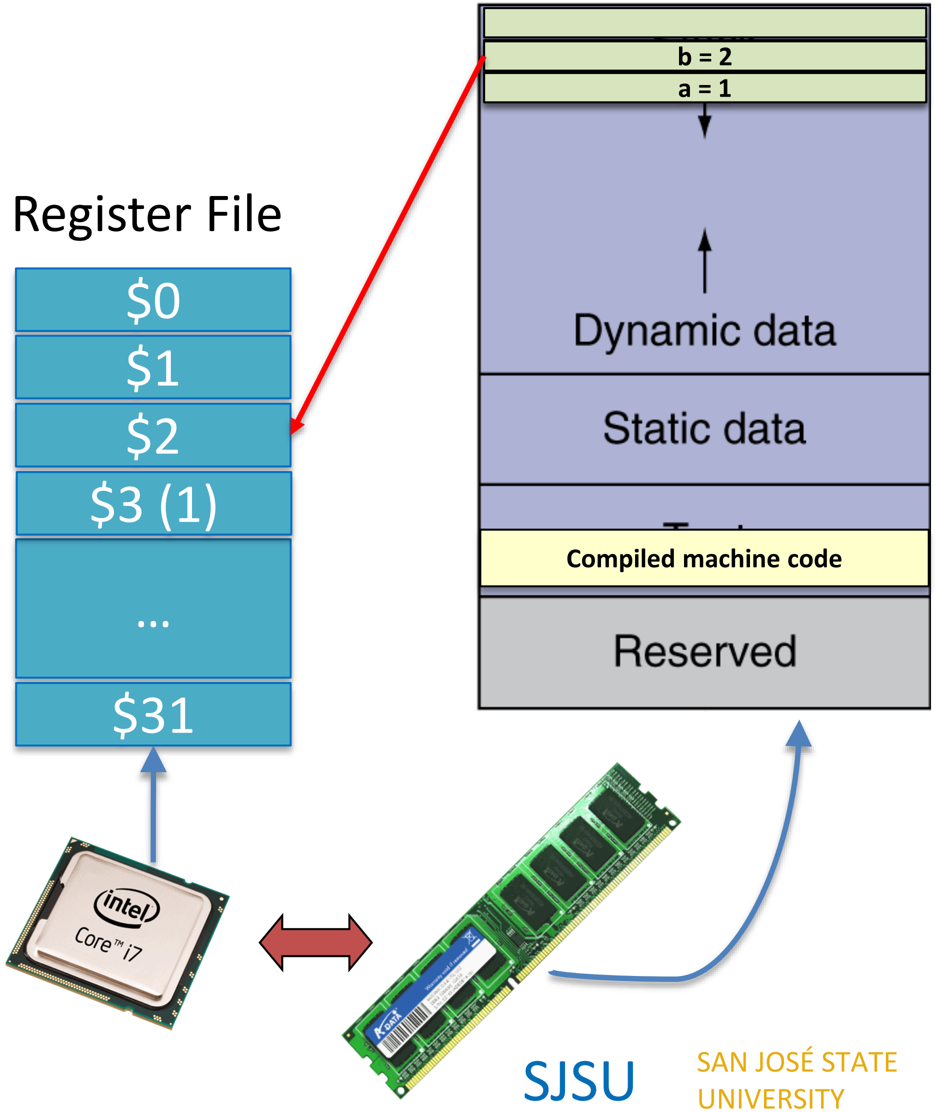
MIPS Assembly code



```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                      # 0x1
        sw      $2,8($fp)
        li      $2,2                      # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

executing line

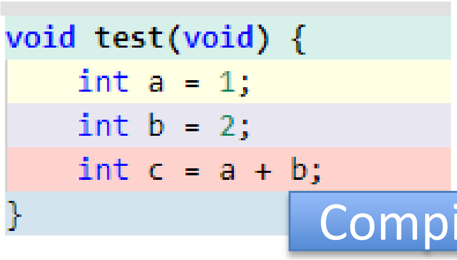
Register File



SJSU SAN JOSÉ STATE UNIVERSITY

Code Example: Memory & Registers

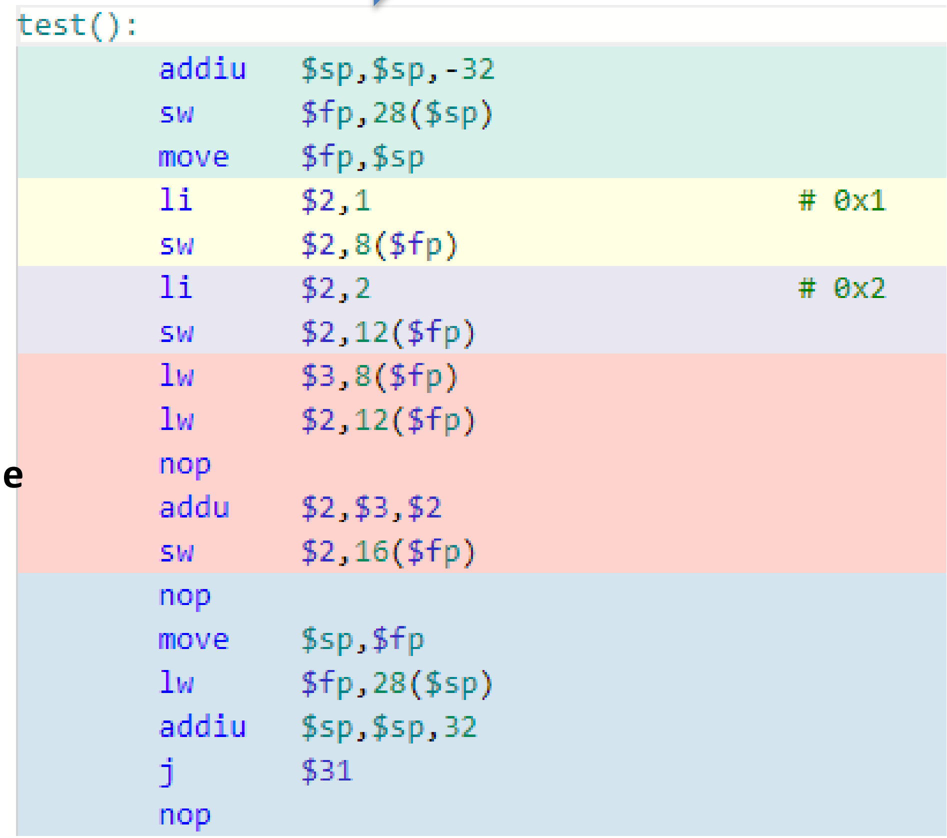
C code



```
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Compile

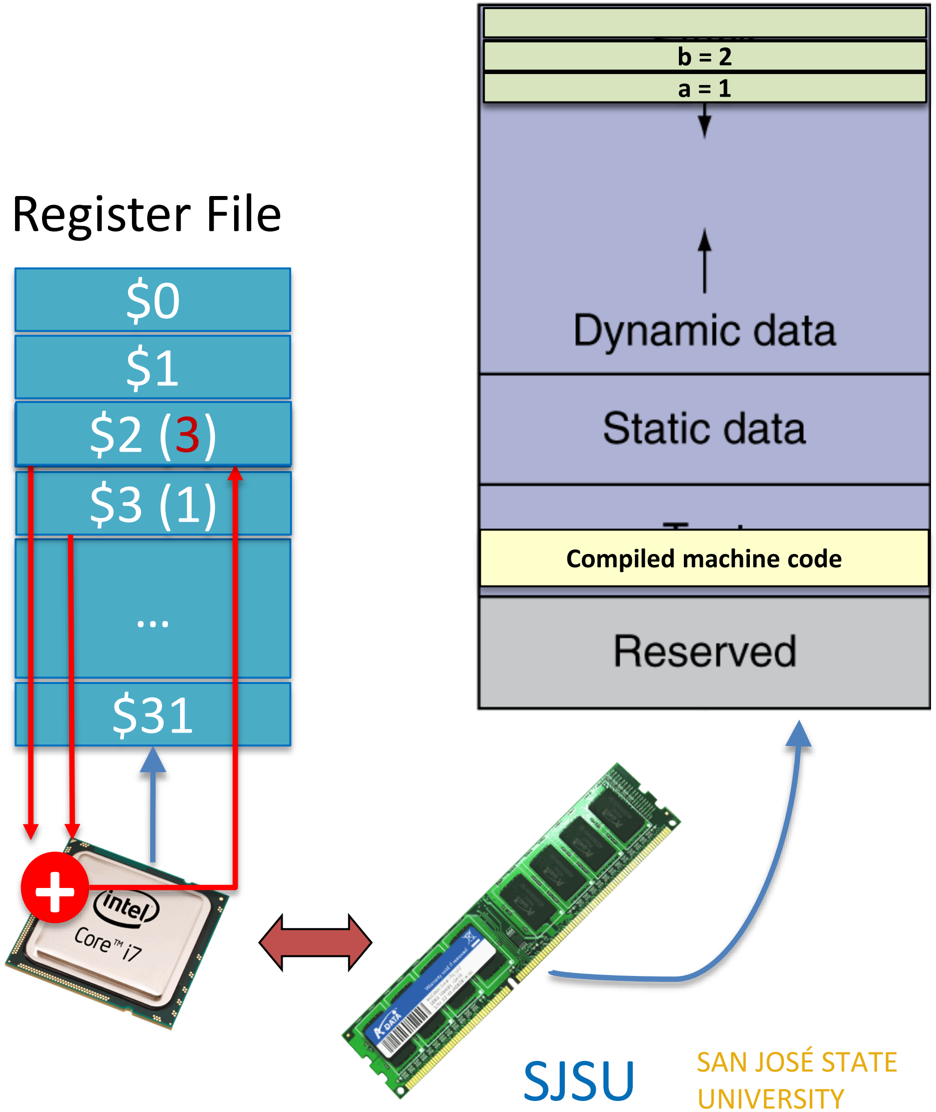
MIPS Assembly code



```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                    # 0x1
        sw      $2,8($fp)
        li      $2,2                    # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

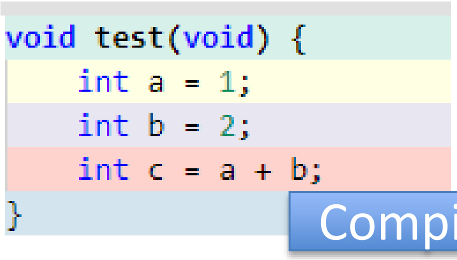
executing line

Register File



Code Example: Memory & Registers

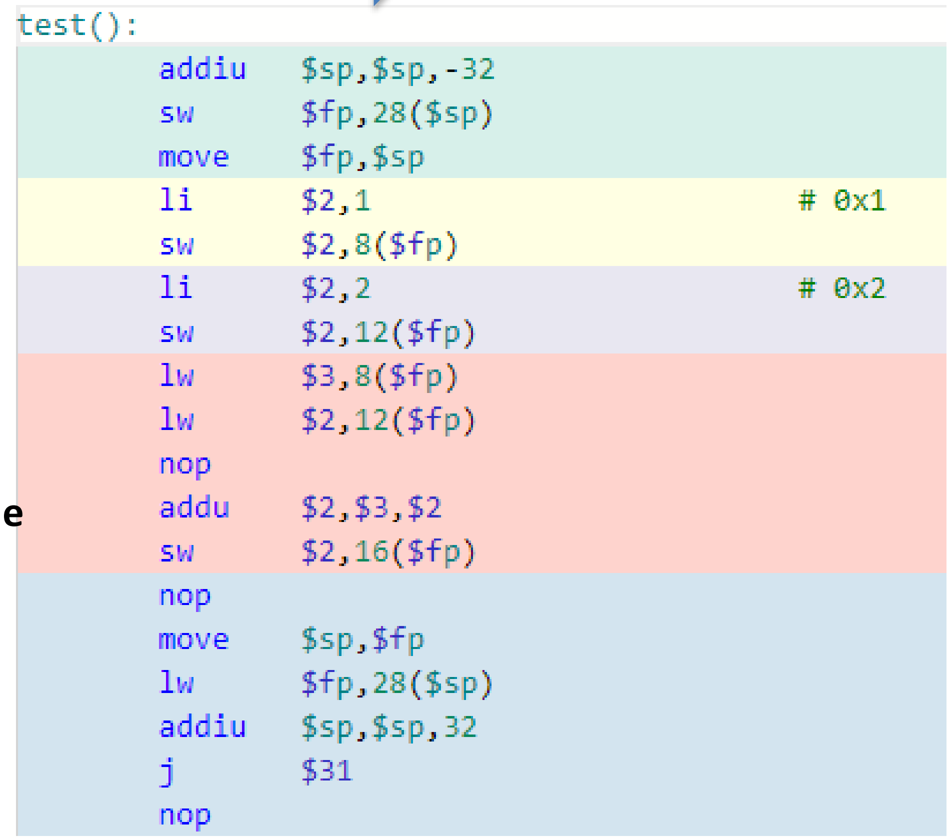
C code



```
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

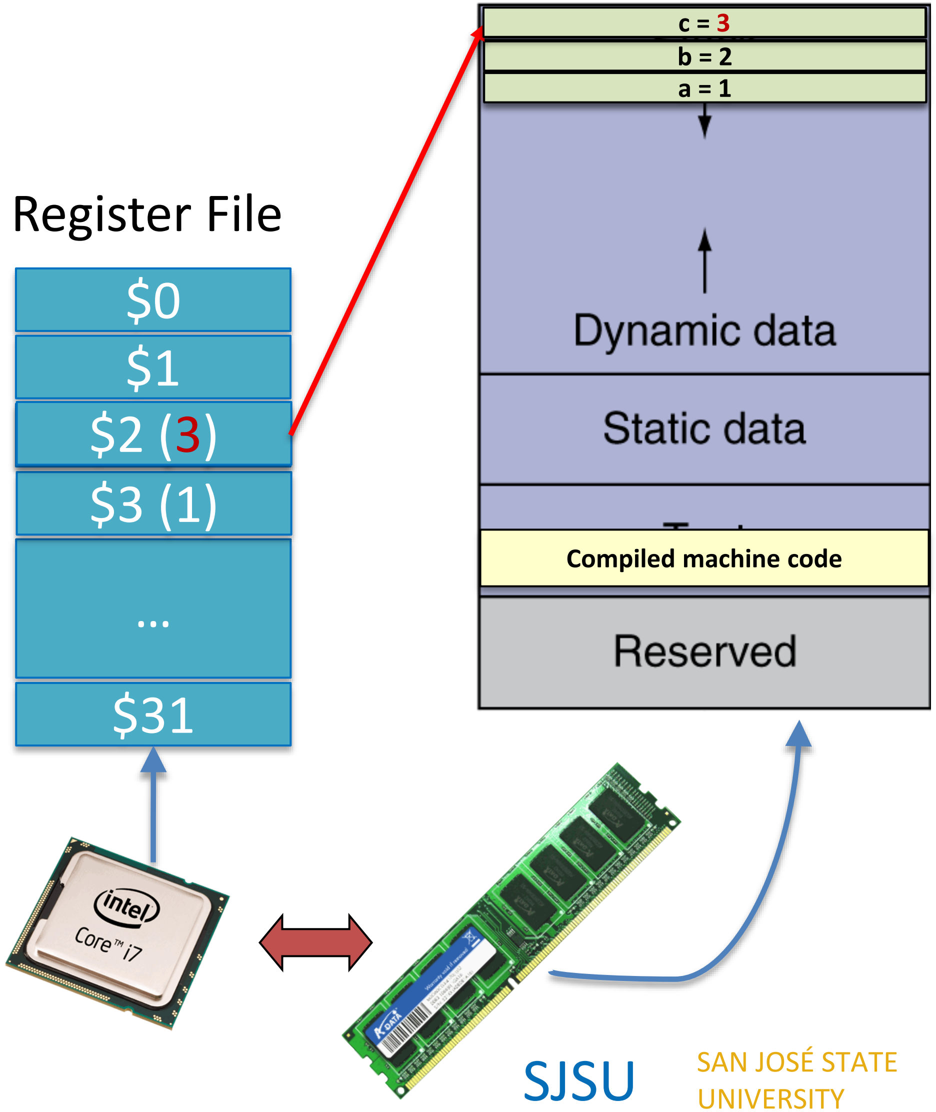
Compile

MIPS Assembly code



```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                    # 0x1
        sw      $2,8($fp)
        li      $2,2                    # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

executing line



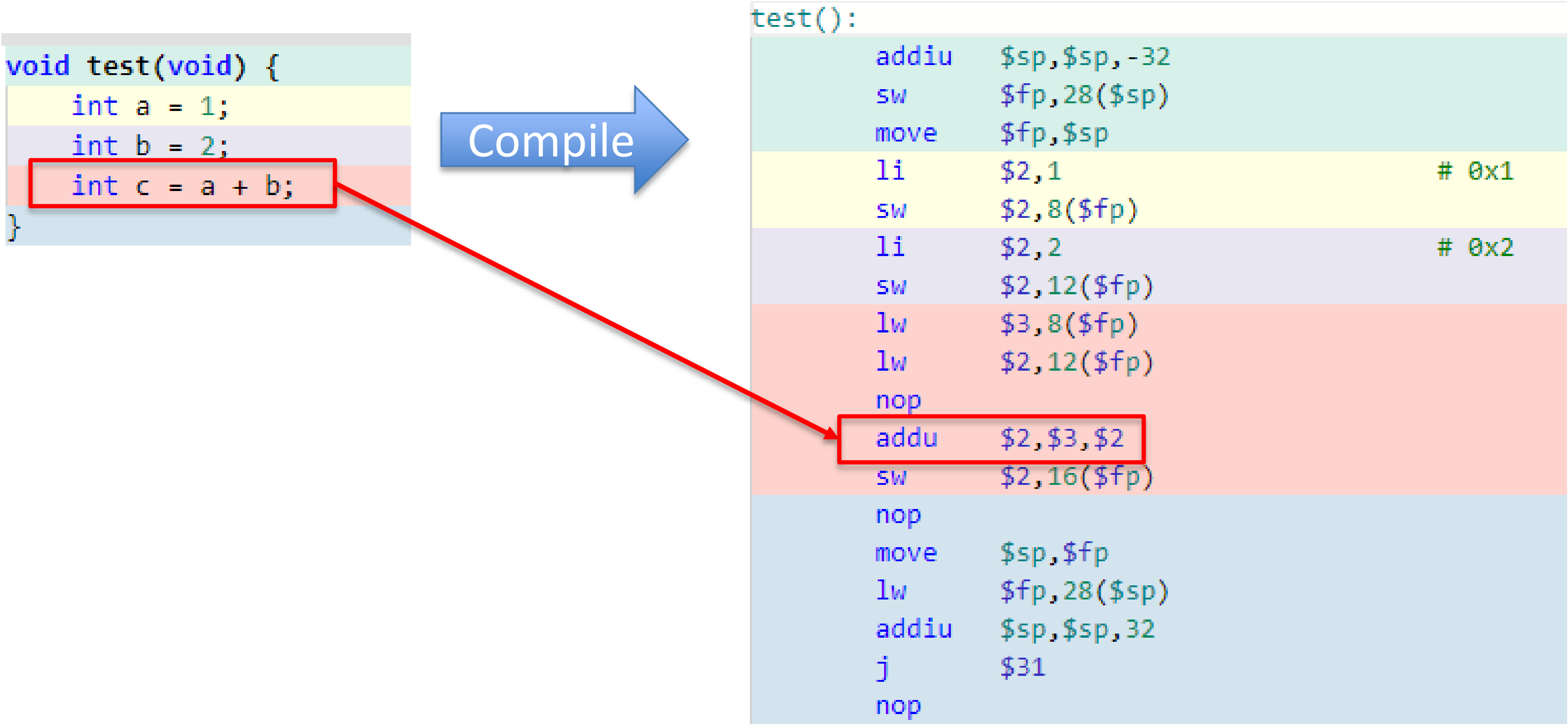
Registers of MIPS CPUs

Assembler Name	Register Number	Description
$zero	$0	Constant 0 value
$at	$1	Assembler temporary
$v0-$v1	$2-$3	Function return values
$a0-$a3	$4-$7	Function Arguments
$t0-$t7	$8-$15	Temporaries
$s0-$s7	$16-$23	Saved Temporaries
$t8-$t9	$24-$25	Temporaries
$k0-$k1	$26-$27	Reserved for OS kernel
$gp	$28	Global Pointer (Global and static variables/data)
$sp	$29	Stack Pointer
$fp	$30	Frame Pointer
$ra	$31	Return Address

HLL vs. Hardware Operations

- HLL are designed to be understood and programmed easily by programmers

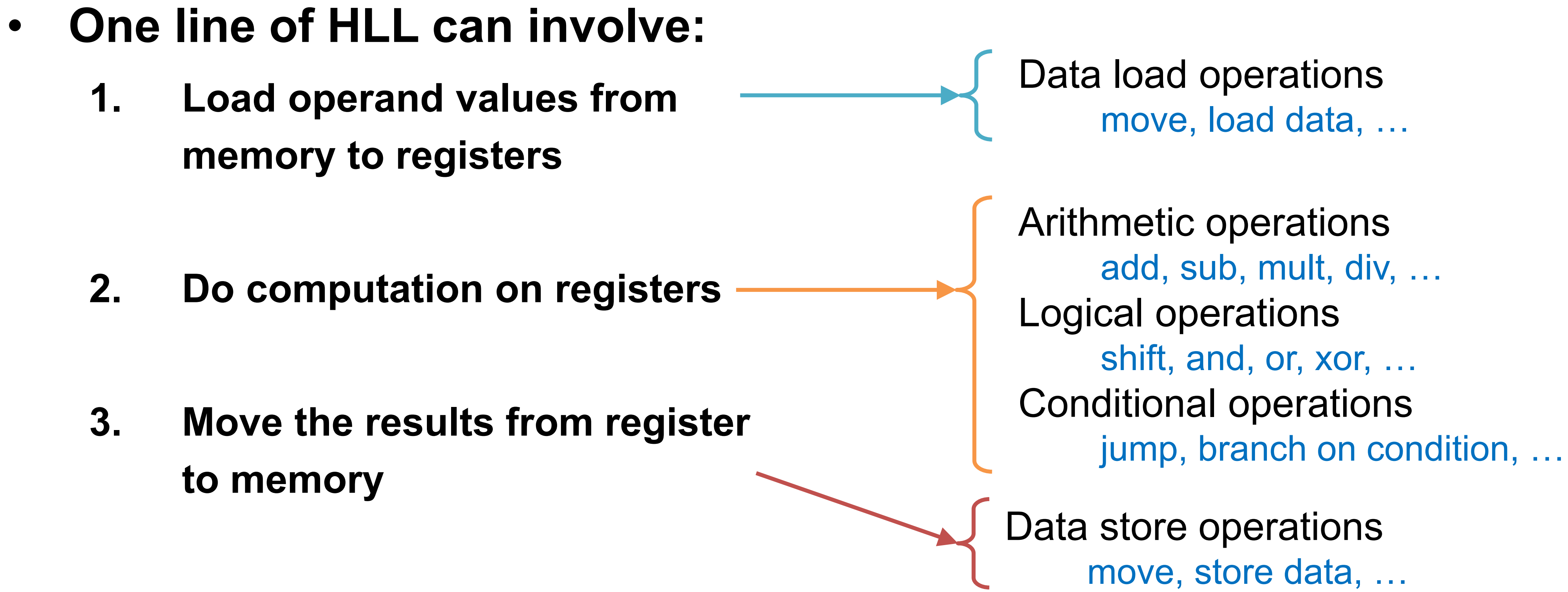
- A HLL line may be a combination of multiple assembly instructions



```
void test(void) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

```
test():
        addiu   $sp,$sp,-32
        sw      $fp,28($sp)
        move    $fp,$sp
        li      $2,1                    # 0x1
        sw      $2,8($fp)
        li      $2,2                    # 0x2
        sw      $2,12($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,16($fp)
        nop
        move    $sp,$fp
        lw      $fp,28($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

A Brief Review



- **One line of HLL can involve:**
  1. **Load operand values from memory to registers**
  2. **Do computation on registers**
  3. **Move the results from register to memory**

Data load operations
move, load data, …

Arithmetic operations
add, sub, mult, div, …
Logical operations
shift, and, or, xor, …
Conditional operations
jump, branch on condition, …

Data store operations
move, store data, …

Operations of MIPS CPUs

C operator	Assembly Operations	Comments
C = A + B	**add** C, A, B	Add two values
C = A - B	**sub** C, A, B	Subtract one from another
C = A * B	**mul** C, A, B	Multiply two values
C = A & B	**and** C, A, B	Logical AND operation
C = A | B	**or** C, A, B	Logical OR operation
C = A ^ B	**xor** C, A, B	Logical XOR operation
C = A << shamt	**sll** C, A, shamt	Shift left by shamt
C = A >> shamt	**srl** C, A, shamt	Shift right by shamt
If (A < B) C = 1	**slt** C, A, B	Set if less than
C = Memory	**lw** C, Memory address	Load value from memory
Memory = C	**sw** C, Memory address	Store value to memory
If (A == B) go to Addr	**beq** A, B, address	Jump if A == B
	Many more …	

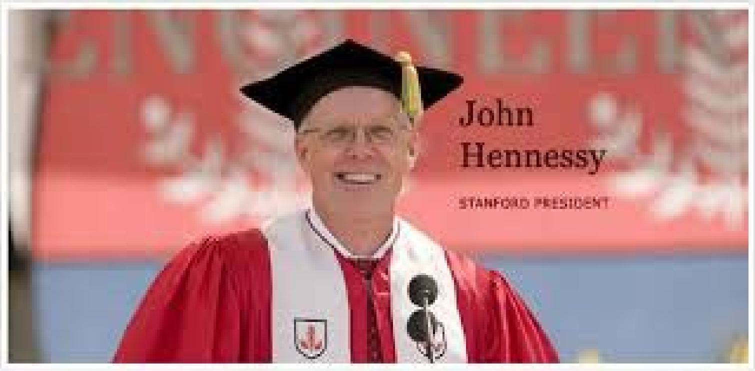
- Note: A, B, C in the table should be replaced by proper register ids

About MIPS Assembly

- **CPUs use their own assembly languages**
  - Assembly language of ARM, Pentium, Opteron… are all different

- **MIPS Assembly Features**
  - Very similar to ARM Assembly
  - One of the earliest RISC architectures that used pipelined instruction processing

- **Developed by MIPS Technologies**
  - Now maintained by Wave Computing
  - Various generations
    - MIPS I~V
    - MIPS32 (32-bit processor)
    - MIPS64 (64-bit processor)
    - microMIPS..



Two types of Computers

- **Reduced Instruction Set Computer (RISC):** MIPS, ARM, …

  - Operands are in registers

  - Each instruction can do only one operation: Simple but longer codes

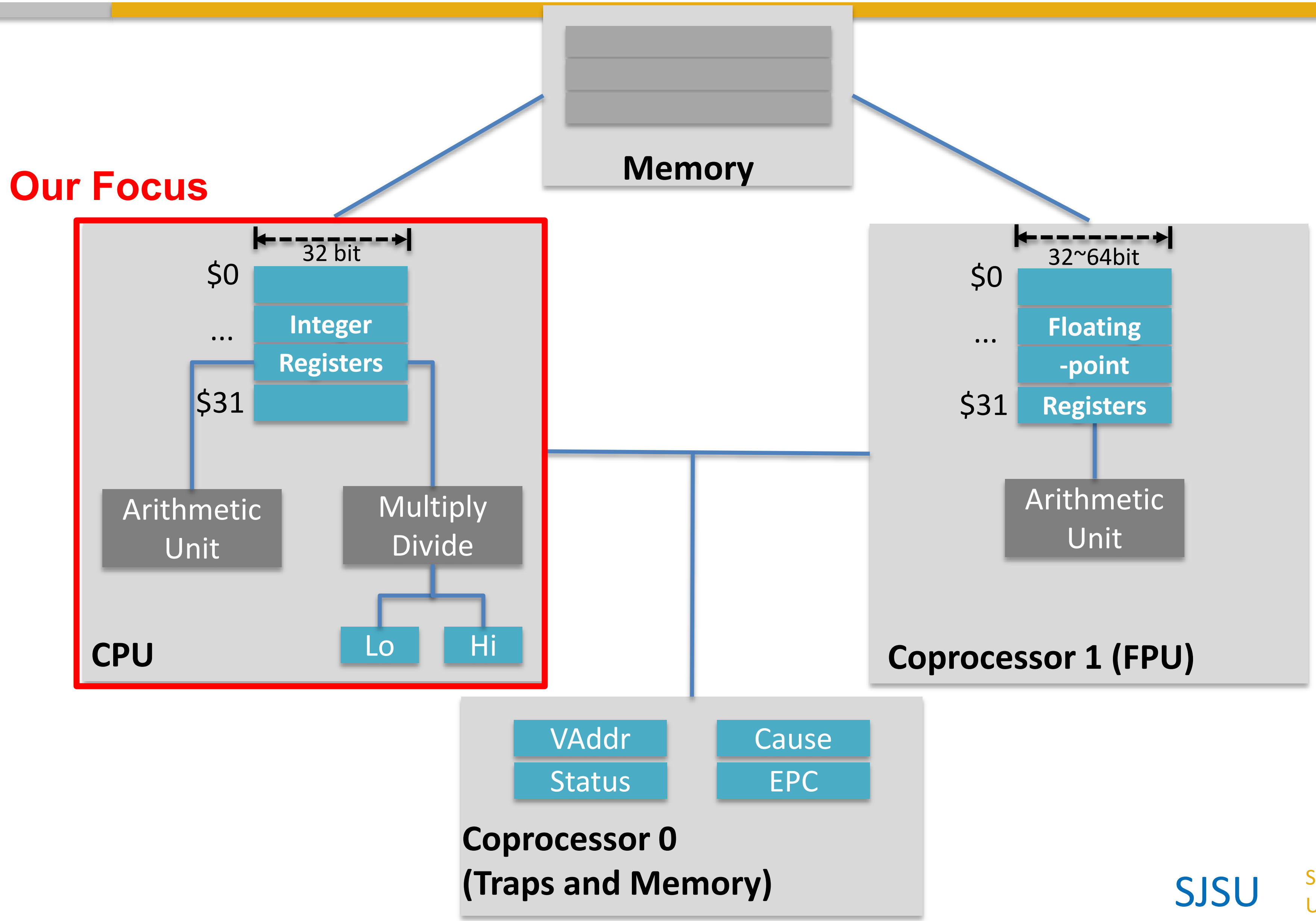
- **Complex Instruction Set Computer (CISC):** Intel, AMD, …

  - Operands can be in registers, stacks, accumulators, in memories

  - Each instruction can do multiple operations: Complex but shorter codes

  - Preferred when memory was very expensive

MIPS Processor Organization



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

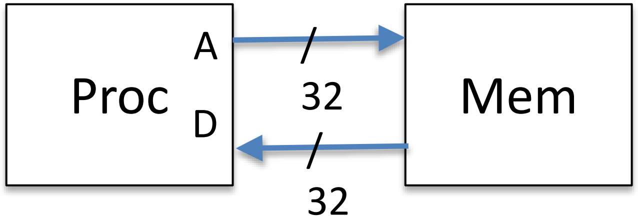
- **Floating-point:** 2 Sizes Defined
  - **Single (S)**
    - 32-bits = 4 bytes
  - **Double (D)**
    - 64-bits = 8 bytes
    - For a 32-bit data bus, a double needs 2 memory reads

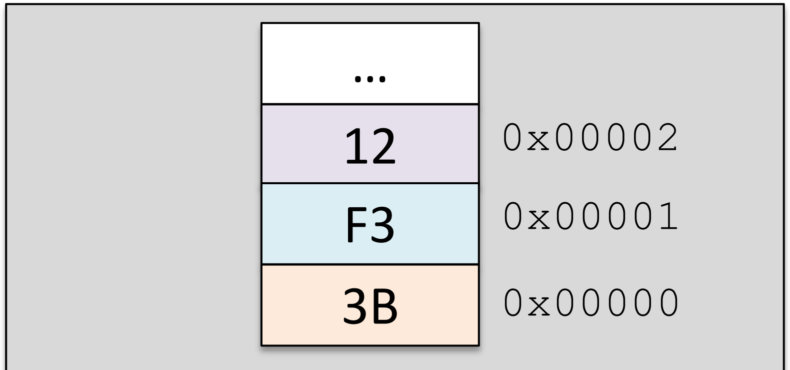
Byte-oriented vs. Word-oriented Memory

- **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

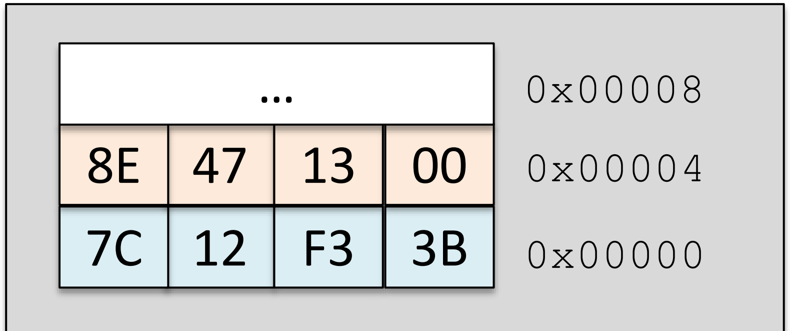
- Logical views can be arranged in **rows of 4-bytes** for word-oriented memories





…	
12	0x00002
F3	0x00001
3B	0x00000

Logical Byte-Oriented View of Mem



…				0x00008
8E	47	13	00	0x00004
7C	12	F3	3B	0x00000

Logical Word-Oriented View

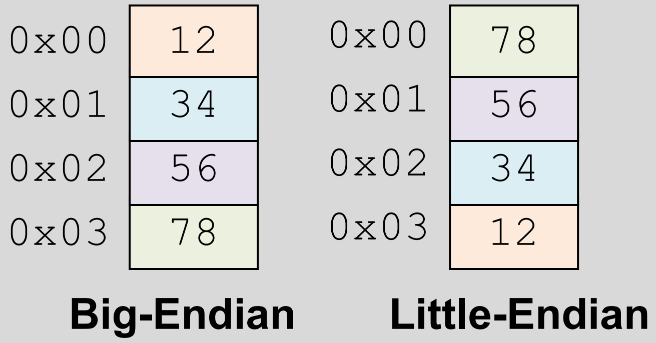
Endian-ness

- **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

The longword value:

0 x 1 2 3 4 5 6 7 8

can be stored differently



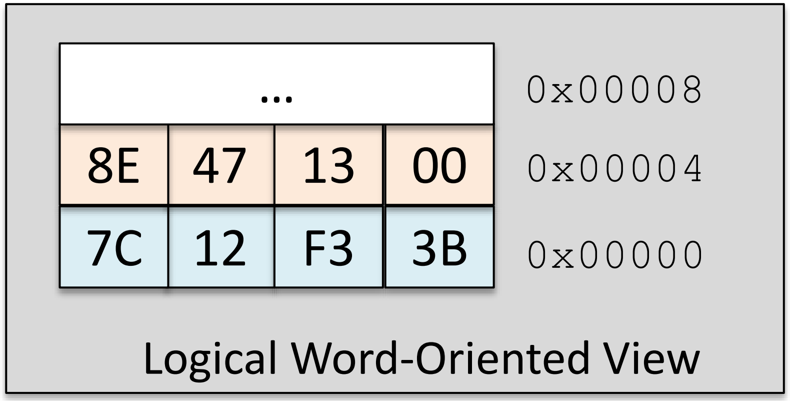
0x00	12
0x01	34
0x02	56
0x03	78

Big-Endian

0x00	78
0x01	56
0x02	34
0x03	12

Little-Endian

Memory Characteristics & Assumptions



…				0x00008
8E	47	13	00	0x00004
7C	12	F3	3B	0x00000

Logical Word-Oriented View

- 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

Tips to Remember Endianness

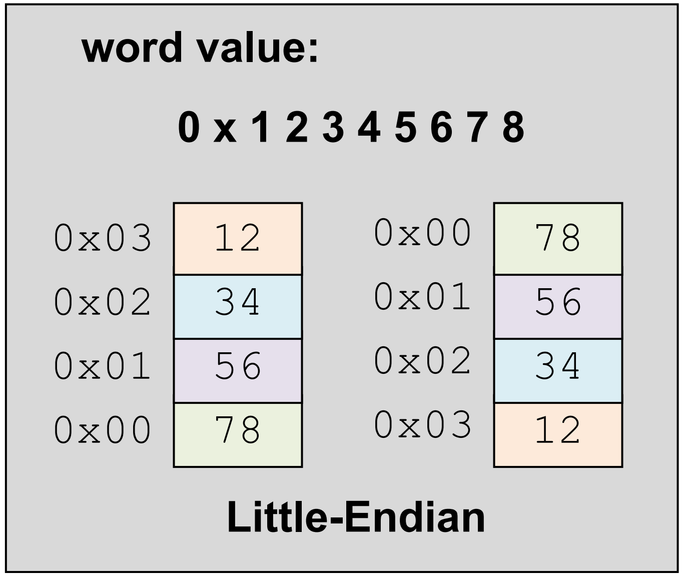
oxCAFEBABE
will be stored as
CA | FE | BA | BE

oxCAFEBABE
will be stored as
BE | BA | FE | CA

www.thebittheories.com

Looks normal (same as writing order) when address is <u>from low to high</u>

Looks strange here
But same as writing order when address is <u>from high to low</u>

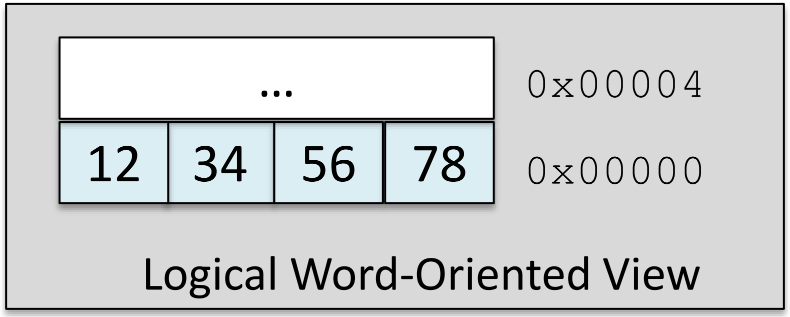


word value:

0 x 1 2 3 4 5 6 7 8

0x03	12	0x00	78
0x02	34	0x01	56
0x01	56	0x02	34
0x00	78	0x03	12

Little-Endian

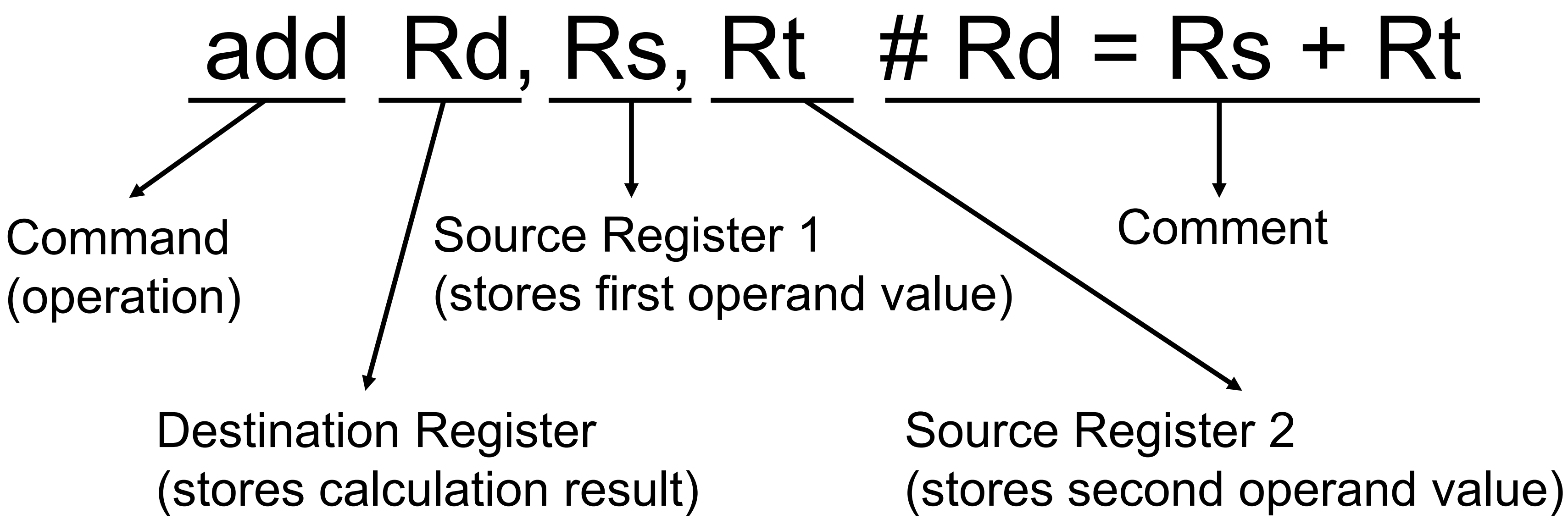


...				0x00004
12	34	56	78	0x00000

Logical Word-Oriented View

Basic Instruction Formats

- **Example: Format with 3 registers**



add Rd, Rs, Rt # Rd = Rs + Rt

Command (operation)

Destination Register (stores calculation result)

Source Register 1 (stores first operand value)

Source Register 2 (stores second operand value)

Comment

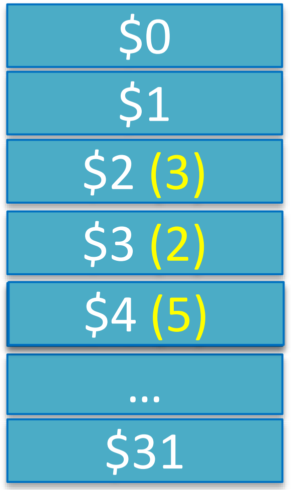
Basic Instruction Formats

- **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

Register File



$0

$1

$2 (3)

$3 (2)

$4 (5)

…

$31

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

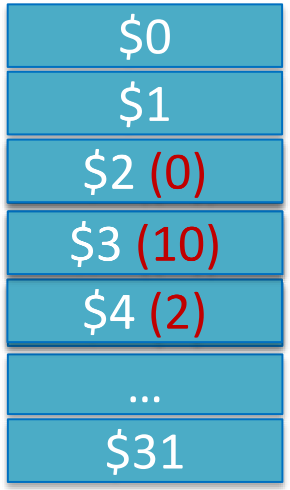
Replace Rd, Rs, Rt with actual registers

R-Type Instructions

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

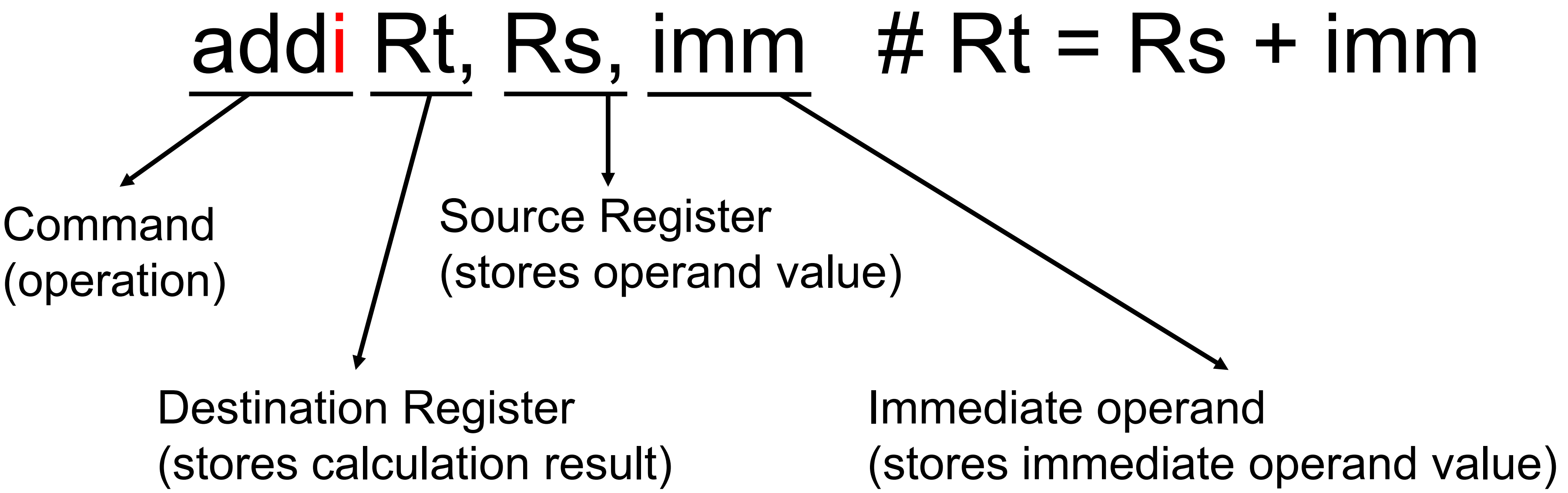
```
sub $4, $2, $3          # 2 = 10 - 8
add $3, $3, $4          # 10 = 8 + 2
slt   $2, $3, $4        # (10 < 2)? →No
                           → $2 = 0
```

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

Command (operation)

Destination Register (stores calculation result)

Source Register (stores operand value)

Immediate operand (stores immediate operand value)

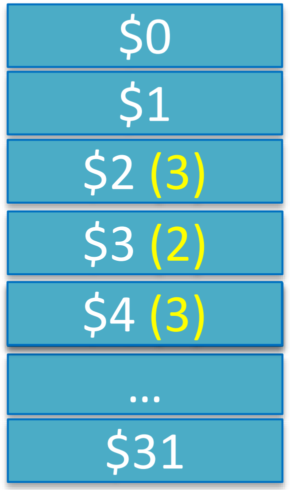
Instructions Using Immediate Value

- **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

Register File



$0
$1
$2 (3)
$3 (2)
$4 (3)
…
$31

I-Type Instructions

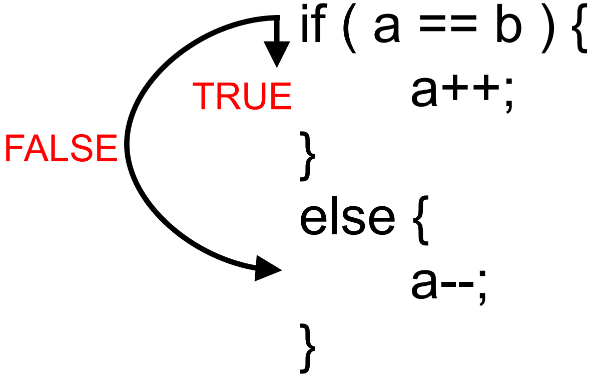
- We call the instructions that use two registers and one immediate value as **I-type** Instructions
  - 1 register and 1 immediate value as source, 1 register for result

- **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--

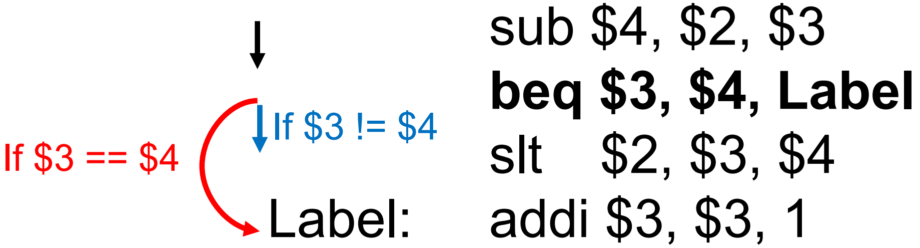
I-Type: Branch Instructions

- **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"

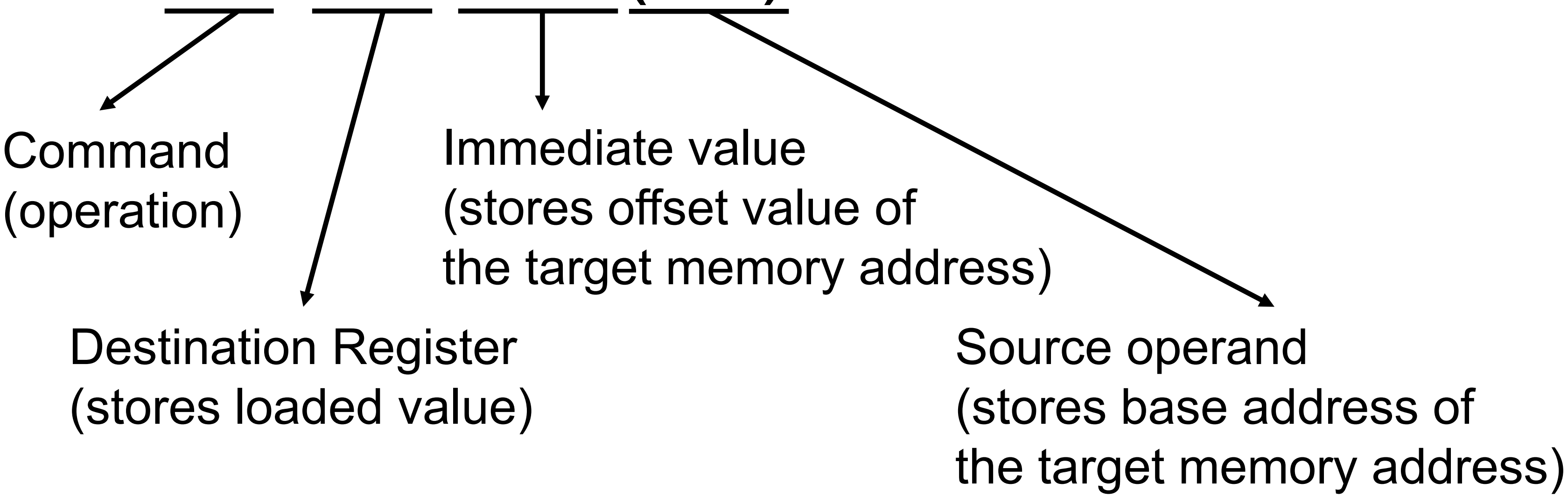
check more pseudo instructions in textbook

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

- **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

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

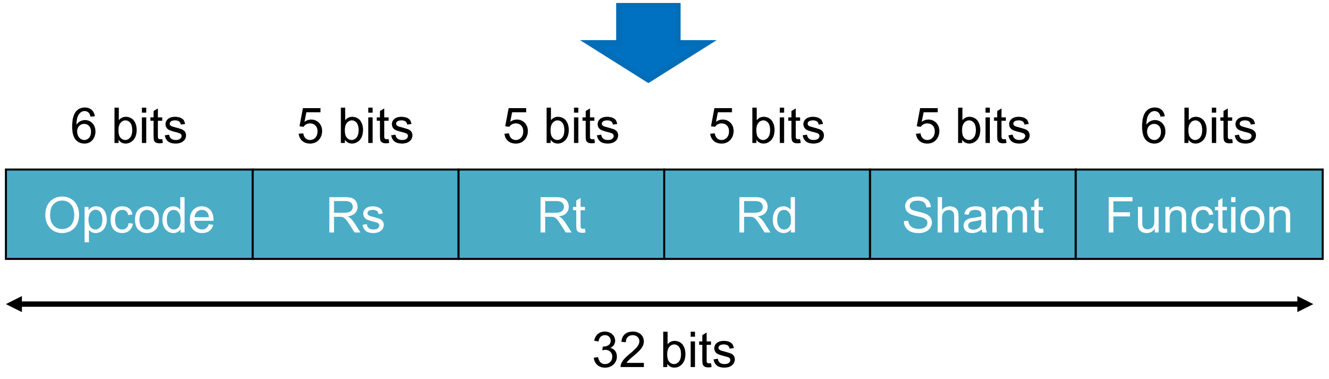
    # If lh in the earlier example is lhu,

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

Machine Code of R-Type Instructions

- **All instructions in MIPS are 32-bit width**
  - Within 32-bit data, command and three register ids should be presented

add Rd, Rs, Rt



6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Rs	Rt	Rd	Shamt	Function

32 bits

Machine Code of R-Type Instructions

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Rs	Rt	Rd	Shamt	Function

←→

32 bits

- **Opcode**
  - Indicate Command/Operation of an instruction with combination of Function field
  - Example:

add	0	Rs	Rt	Rd	Shamt	0x20
and	0	Rs	Rt	Rd	Shamt	0x24
or	0	Rs	Rt	Rd	Shamt	0x25

    - **many more**
  - R-type instructions always have all zeros in Opcode field and use Function field to distinguish the instructions

Machine Code of R-Type Instructions

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Rs	Rt	Rd	Shamt	Function

32 bits

- **Rs, Rt, Rd**
  - Indicate source and destination register ids used by an instruction
  - Example:
    - **add $5, $4, $3**
    - **and $13, $8, $2**
    - **or $t8, $s0, $s1**

000000	$4	$3	$5	Shamt	100000
000000	$8	$2	$13	Shamt	100100
000000	$s0 (16)	$s1 (17)	$t8 (24)	Shamt	100101

  - Each field has 5 bits because we have 32 (=2^5) registers in MIPS

Machine Code of R-Type Instructions

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Rs	Rt	Rd	Shamt	Function

32 bits

- **Shamt**
  - Indicate the shift amount that is used for shift instructions only; the other R-type instructions have all zeros in this field
  - Example:

add	**$5, $4, $3**	000000	00100	00011	00101	00000	100000
and	**$13, $8, $2**	000000	01000	00010	01101	00000	100100
or	**$t8, $s0, $s1**	000000	10000	10001	11000	00000	100101

Shift Instructions

- **Special format R-type Instruction**
  - The second operand is an immediate value (shamt) that indicates the amount of shift

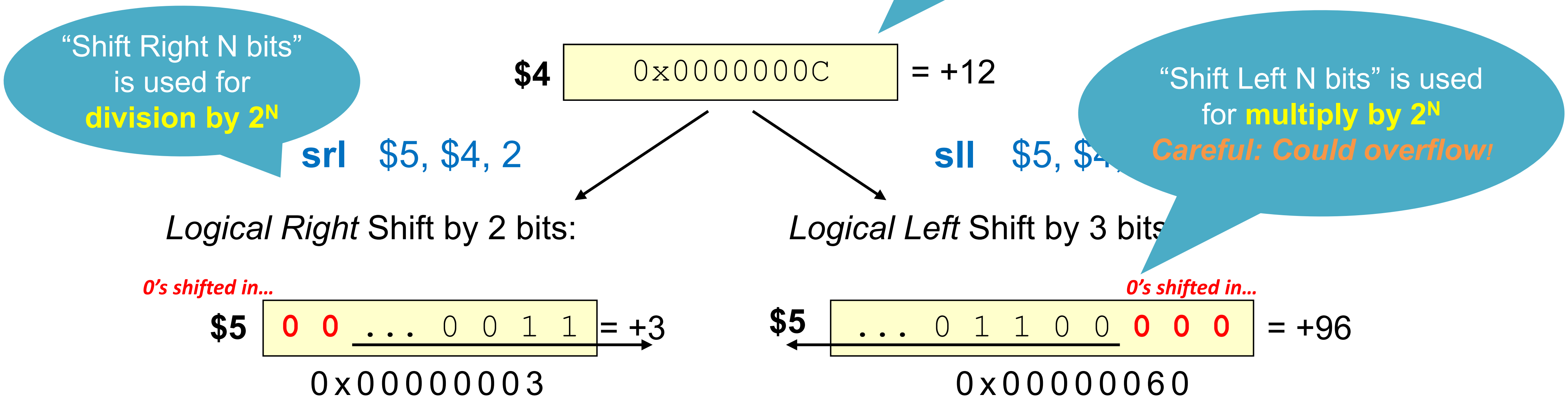
- **Logical Shifting**
  - Shift towards left/right with filling in 0s
  - Example:
    - **sll** Rd, Rt, shamt # Rd = Rt << shamt (Rt: unsigned data)
    - **srl** Rd, Rt, shamt # Rd = Rt >> shamt (Rt: unsigned data)

- **Arithmetic Shifting**
  - Shift towards left/right with maintaining the value's sign bit
  - Example:
    - **sra** Rd, Rt, shamt # Rd = Rt >> shamt (Rt: signed data)

Example: Logical Shift

- **Shift towards left/right with filling in 0s**
- **Example:**
  - Assume that the original value of $4 is 0x0000000C
  - What is the value of $5 after executing the followi
  - **sll** $5, $4, 3 # shift left logical the value o
  - **srl** $5, $4, 2 # shift right logical the value of $ 2 bits

Shift operation should be done in bit level → **translate given value to binary first**



“Shift Right N bits” is used for **division by 2^N**

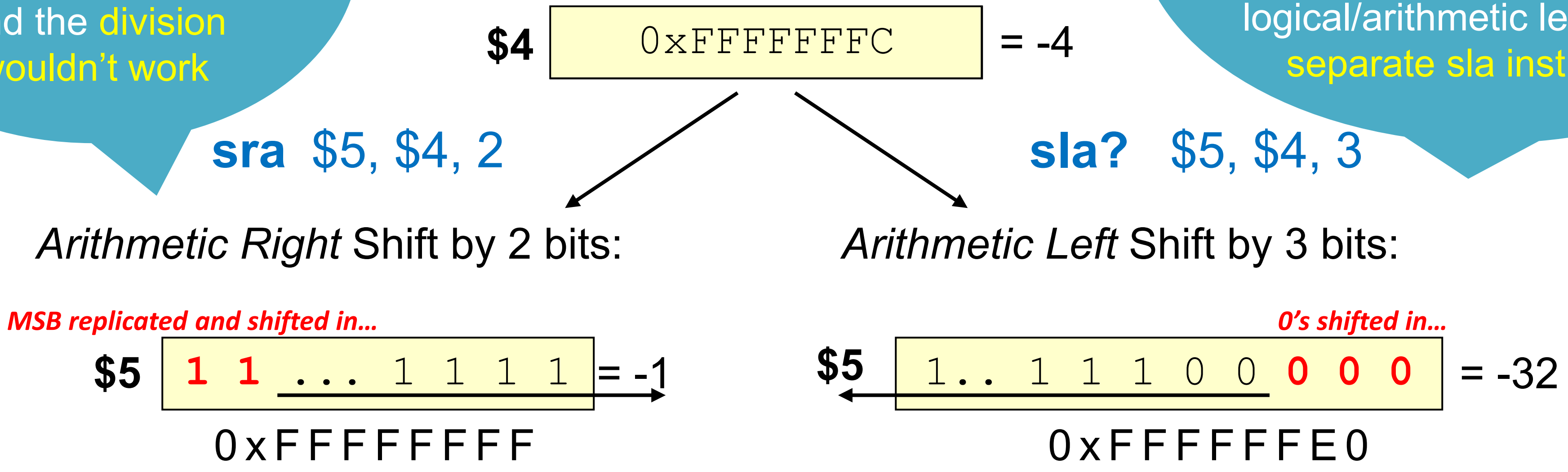
“Shift Left N bits” is used for **multiply by 2^N**
Careful: Could overflow!

Example: Arithmetic Shift

- **Shift right with replicating MSB**
- **Shift left with filling in 0s**
- **Example:**
  - – the original value of $4 is 0xFFFFFFFC (= -4)
  - of $5 after executing the following shift instru
  - # shift right arithmetic the value of $4 b

Notice if we shifted in 0s (like a logical right shift) our result would be a positive number and the division wouldn't work

Notice there is no difference between an arithmetic and logical left shift. We always shift in 0s.
MIPS uses sll for both logical/arithmetic left shift (no separate sla instruction)



$4 0xFFFFFFFC = -4

sra $5, $4, 2

sla? $5, $4, 3

Arithmetic Right Shift by 2 bits:

MSB replicated and shifted in…

$5 1 1 . . . 1 1 1 1 = -1

0xFFFFFFFF

Arithmetic Left Shift by 3 bits:

0's shifted in…

$5 1 . . 1 1 1 0 0 0 0 0 = -32

0xFFFFFFE0

Machine Code of R-Type Instructions

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Rs	Rt	Rd	Shamt	Function

32 bits

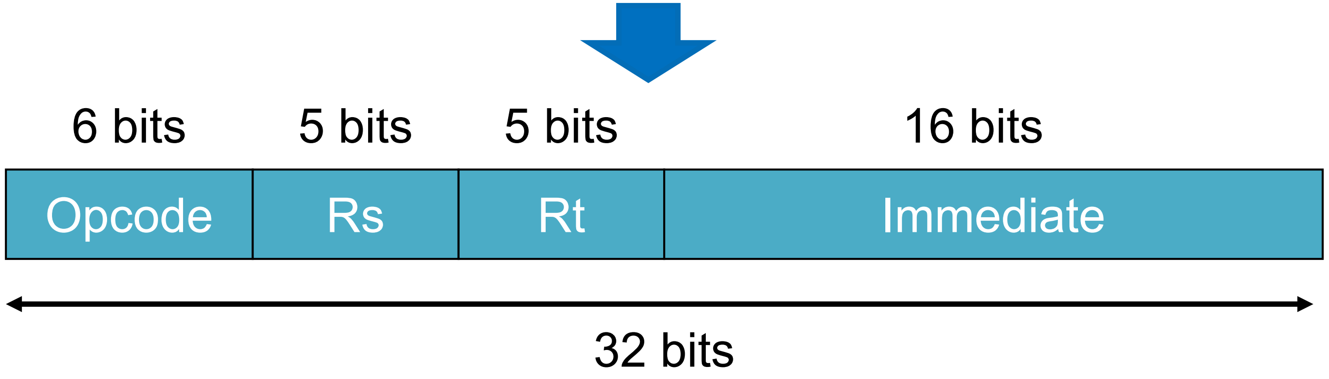
- **Shamt**
  - Indicate the shift amount that is used for shift instructions only; the other R-type instructions have all zeros in this field
  - Example:

• **sll**	**$5, $4, 3**	000000	00000	00100	00101	3	000000
• **srl**	**$5, $4, 2**	000000	00000	00100	00101	2	000010
• **sra**	**$5, $4, 2**	000000	00000	00100	00101	2	000011

Machine Code of I-Type Instructions

- **All instructions in MIPS are 32-bit wide**
  - Within 32-bit data, command, two register ids, and an immediate value should be presented

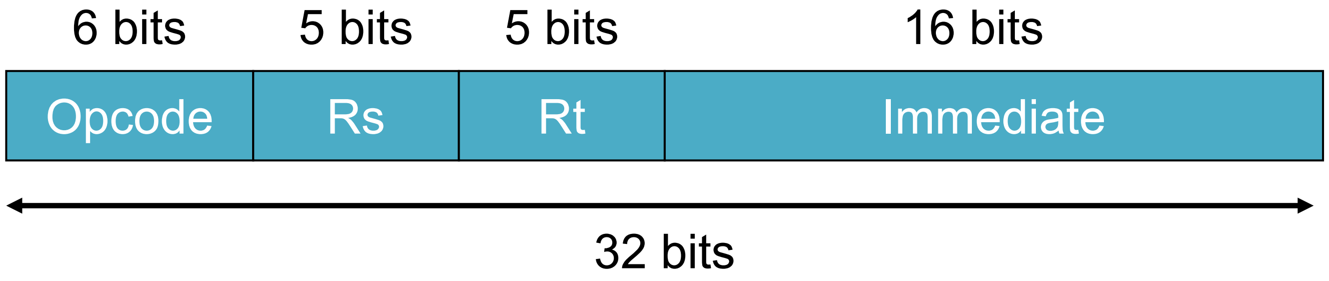
addi Rt, Rs, imm



6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

32 bits

Machine Code of I-Type Instructions



6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

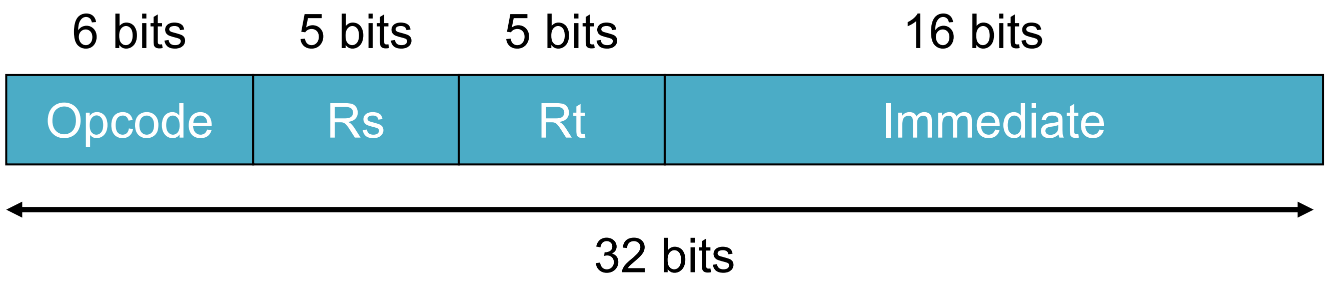
32 bits

- **Opcode**
  - Indicate Command/Operation of an instruction
  - Example:

addi	001000	Rs	Rt	Immediate
andi	001100	Rs	Rt	Immediate
ori	001101	Rs	Rt	Immediate

- **many more;**

Machine Code of I-Type Instructions



6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

32 bits

- **Rs, Rt**
  - Indicate source and destination register ids used by an instruction
  - Example:
    - **addi $5, $4, 1**
    - **andi $13, $12, 0xA**
    - **ori $t3, $t2, 12**

001000	$4	$5	Immediate
001100	$12	$13	Immediate
001101	$t2 (10)	$t3 (11)	Immediate

Machine Code of I-Type Instructions

6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

32 bits

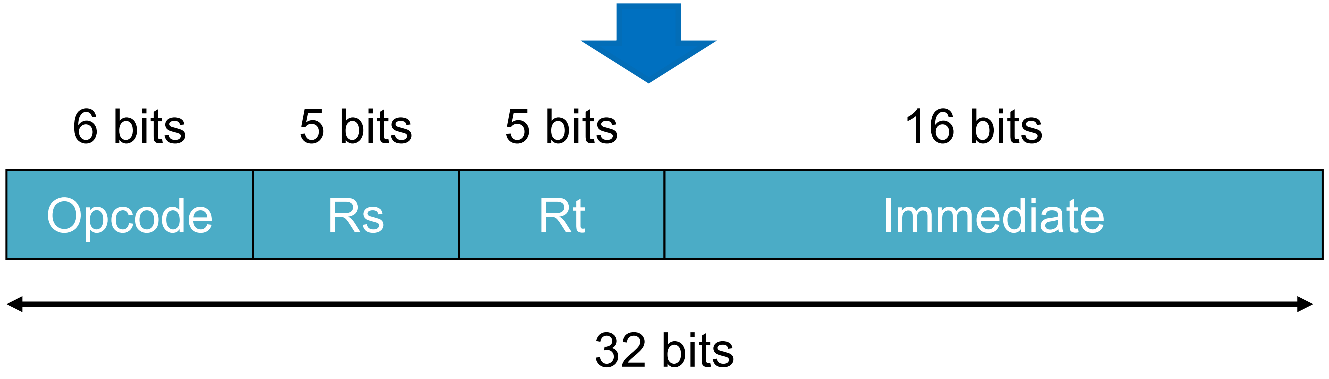
- **Immediate**
  - Indicate the immediate value
  - Example:

• **addi**	**$5, $4, 1**	001000	00100	00101	1
• **andi**	**$13, $12, 0xA**	001100	01100	01101	0xA
• **ori**	**$t3, $t2, 12**	001101	01010	01011	12

Machine Code of I-Type Instructions

- **Special I-Type instructions: Load/Store**
  - Within 32-bit data, command, two register ids, and an offset value should be presented

lw Rt, imm(Rs)



6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

32 bits

Machine Code of I-Type Instructions

6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

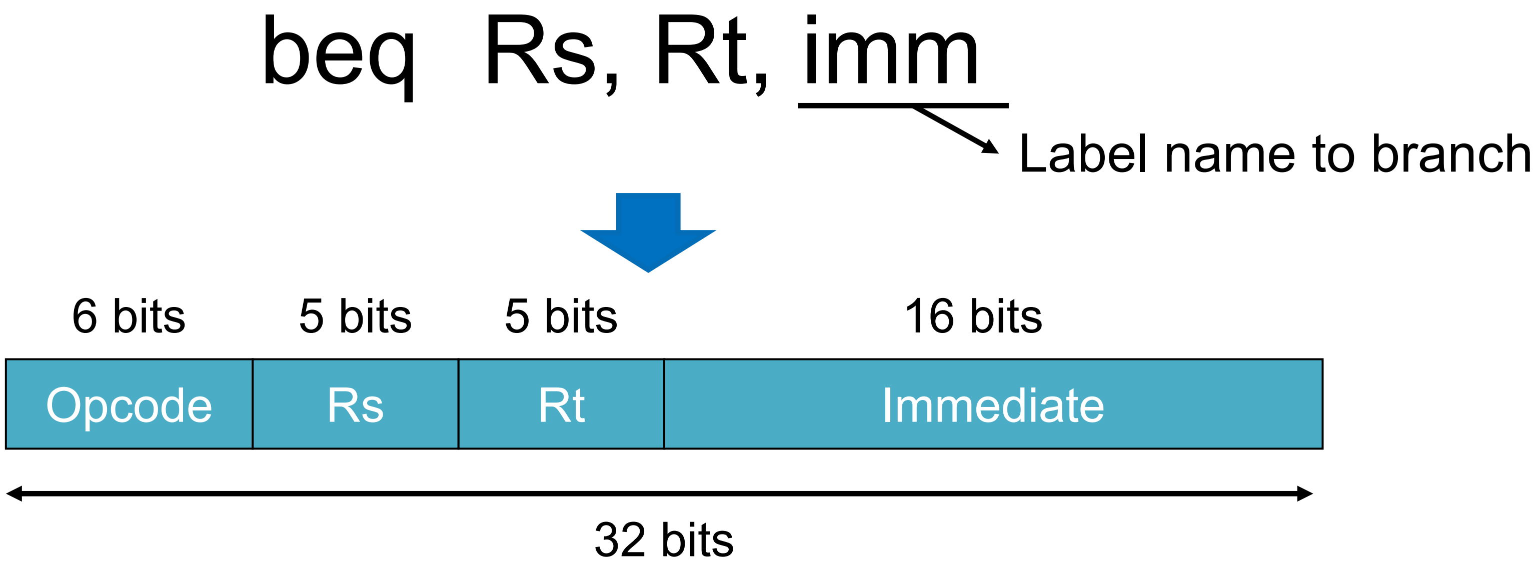
32 bits

- **Load/Store in I-Type machine code format**
  - Example:

lb	**$5, 4($4)**	0x20	00100	00101	4
lh	**$13, 0x10($12)**	0x21	01100	01101	0x10
lw	**$t3, -16($t2)**	0x23	01010	01011	-16
sb	**$3, 0xfff0($4)**	0x28	00100	00011	0xfff0
sh	**$2, 8($8)**	0x29	01000	00010	8
sw	**$s1, 10($s0)**	0x2B	10000	10001	10

Machine Code of I-Type Instructions

- **Special I-Type instructions: Branch**
  - uses the same format but needs a special treatment for immediate field value



beq Rs, Rt, imm

Label name to branch

6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

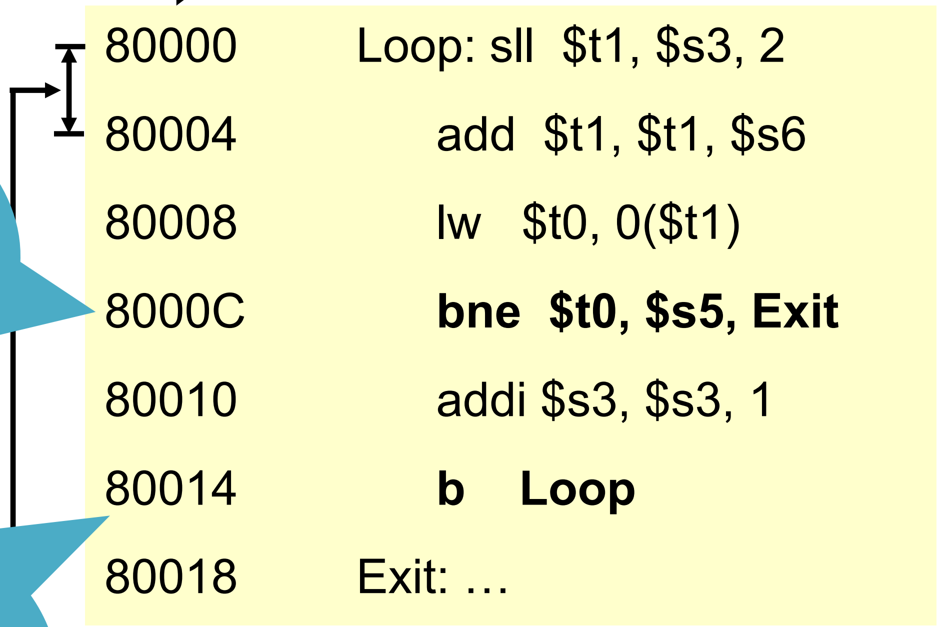
32 bits

Branch Target Addressing

- **PC-relative addressing**
  - MIPS calculates the branch target address based on the branch instruction's next instruction's address
  - *"Branch to N lines abo*
    branch to absolute a
  - necessary because in
    do not know (at compile time) our
    code will be loaded in the m

ng
dress increases by 4 bytes

Assume that the code is loaded to **0x00080000** in memory



```
80000        Loop: sll  $t1, $s3, 2
80004              add  $t1, $t1, $s6
80008              lw   $t0, 0($t1)
8000C              bne  $t0, $s5, Exit
80010              addi $s3, $s3, 1
80014              b    Loop
80018        Exit: …
```

Branch two lines below from bne's next instruction (addi)
→ Immediate field will be filled with 2

Branch 6 lines above from b's next line (Exit)
→ Immediate field will be filled with -6

Branch Target Addressing

- **Why branch's next instruction should be considered?**
  - Because MIPS increments program counter (PC) by 4 before executing an instruction
  - This already incremented PC value is used for branch target address calculation

- **Branch Target Address Calculation**
  - **Target address = branch's next instruction address + offset x 4**
  - **bne $t0, $s5, Exit**
    - Exit (0x00080018) = addi address (0x00080010) + distance (2) x 4 byte/inst
  - **b Loop**
    - Loop (0x00080000) = Exit address (0x00080018) + distance (-6) x 4 byte/inst

```
80000     Loop: sll  $t1, $s3, 2
80004           add  $t1, $t1, $s6
80008           lw   $t0, 0($t1)
8000C           bne  $t0, $s5, Exit
80010           addi $s3, $s3, 1
80014           b    Loop
80018     Exit: …
```

Machine Code of I-Type Instructions

6 bits	5 bits	5 bits	16 bits
Opcode	Rs	Rt	Immediate

32 bits

```
80000s      Loop: sll  $t1, $s3, 2
80004             add  $t1, $t1, $s6
80008             lw   $t0, 0($t1)
8000C             bne  $t0, $s5, Exit
80010             addi $s3, $s3, 1
80014             b    Loop
80018       Exit: …
```

- **Branch in I-Type machine code format**
  - Example:

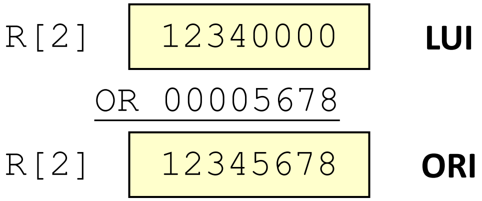
bne	**$t0, $s5, Exit**	0x5	01000	10101	2
beq	**$0, $0, Loop**	0x4	00000	00000	-6

Loading an Immediate

- **What if you want to load an immediate value to a register?**
- **If immediate (constant) is 16 bits or less**
  - Use **ori** or **addi** instruction with $0 register
  - Examples : You want to load value 1 to $2
    - addi $2, $0, 1 // R[2] = 0 + 1 = 1
    - ori $2, $0, 0x1 // R[2] = 0 | 1 = 1

LUI: the immediate value is loaded to the MSB 16 bits of the target register

- **If immediate is more than 16 bits**
  - Immediates limited to 16 bits so we must load constant with a 2-instruction sequence using the special LUI (Load Upper Immediate) instruction
  - To load $2 with 0x12345678
    - lui $2, 0x1234
    - ori $2, $2, 0x5678

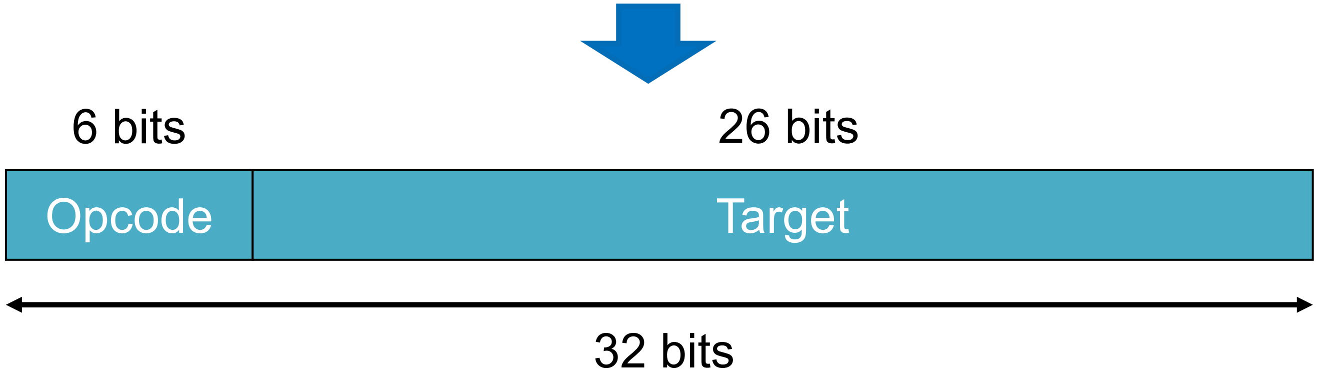


```
R[2]  12340000   LUI
   OR 00005678
R[2]  12345678   ORI
```

J-Type Instruction

- **There is another type, J, in MIPS for Jump instruction**
  - Similar to unconditional branch

j target # Jump to target



6 bits

26 bits

Opcode	Target

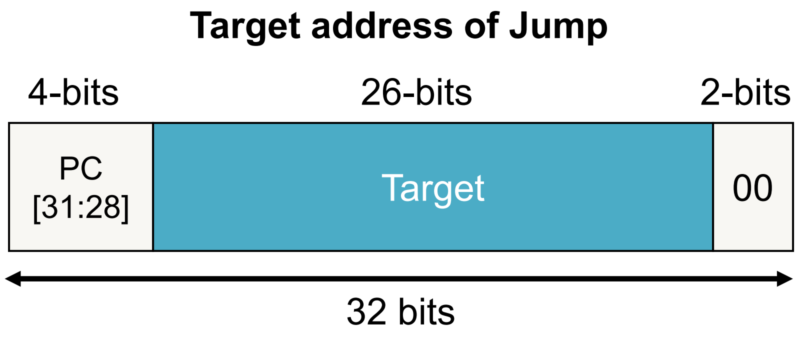
32 bits

Jump Target Addressing

- **Jump instruction provides larger scale jump than branch**

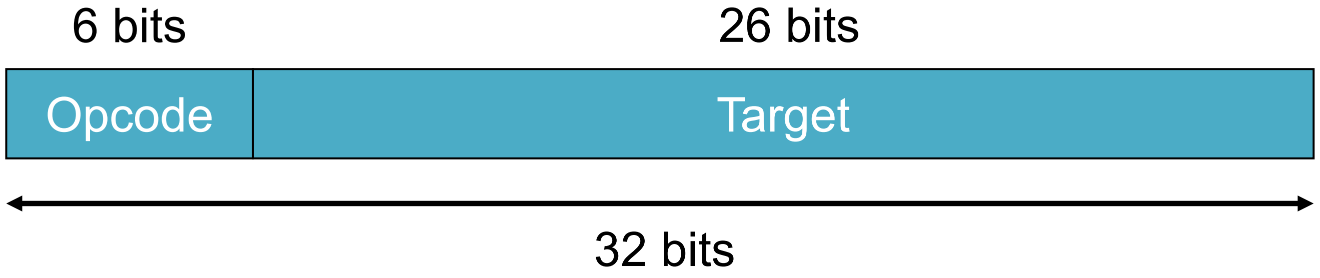
- **Target address = First 4 bits of jump's next instruction address : Last 28 bits of (target x 4)**

- **Example: j Loop (0x00080000)**
  - **The first 4 bits of Exit = 0x0**
  - **Last 28 bits of target x 4 = 0080000**
  - **target = 0x0020000**

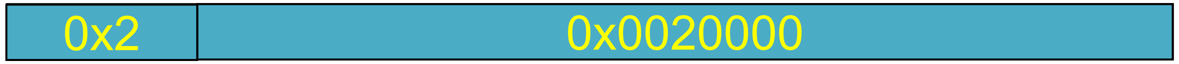


```
00080000    Loop: sll  $t1, $s3, 2
00080004          add  $t1, $t1, $s6
00080008          lw   $t0, 0($t1)
0008000C          bne  $t0, $s5, Exit
00080010          addi $s3, $s3, 1
00080014          j    Loop
00080018    Exit: …
```

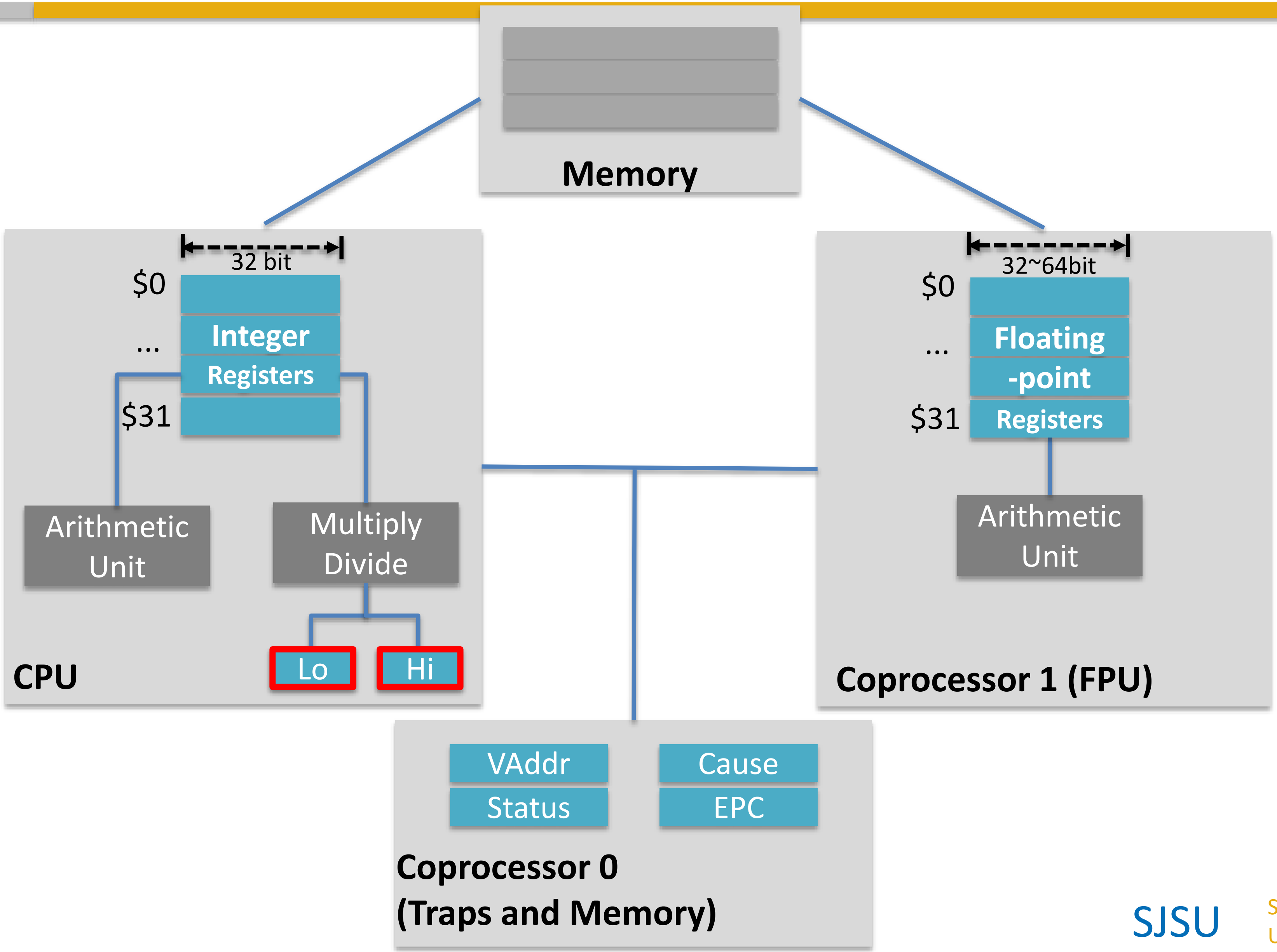
Machine Code of J-Type Instructions



- **Jump in J-Type machine code format**
  - Example:
    - **j Loop**



Mult & Div Instructions



SJSU SAN JOSÉ STATE UNIVERSITY

Mult & Div Instructions

- **Mult & Div use special registers, lo and hi**

- **Multiplication**
  - **mult** Rs, Rt # lo = lower 32-bit of Rs * Rt
    # hi = higher 32-bit of Rs * Rt

- **Division**
  - **div** Rs, Rt # lo = quotient of Rs/Rt
    # hi = remainder of Rs/Rt

- **Moves the contents of lo/hi registers to GPR**
  - **mflo** $2 # $2 = lo
  - **mfhi** $3 # $3 = hi

Machine Code of Mult/Div/Mflo/Mfhi

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Rs	Rt	Rd	Shamt	Function

32 bits

- **Mult/Div/Mflo/Mfhi**
  - Example:
    - **mult $5, $4**
    - **div $5, $4**
    - **mflo $5**
    - **mfhi $4**

000000	00101	00100	00000	00000	0x18
000000	00101	00100	00000	00000	0x1a
000000	00000	00000	00101	00000	0x12
000000	00000	00000	00100	00000	0x10

Loops in MIPS: While Loops

- **Loop code consists of**
  - Condition check code
    - To decide to continue the loop iteration or exit
  - Jump to the loop entry to run the next iteration

Example: Find *x* where $2^x = 128$

High-level language

```
int pow = 1;
int x = 0;

while (pow != 128)
{
        pow = pow * 2;
        x = x + 1;
}
```

MIPS

```
# $s0 = pow, $s1 = x

            addi      $s0, $0, 1
            add       $s1, $0, $0
            addi      $t0, $0, 128
            beq       $s0, $t0, done
            sll       $s0, $s0, 1
            addi      $s1, $s1, 1
            j         while
done:
```

Loops in MIPS: For Loop

- **Loop code consists of**
  - Condition check code
    - To decide to continue the loop iteration or exit
  - Jump to the loop entry to run the next iteration

Example: Add numbers from 0 to 9

High-level language

```
int sum = 0;
int i;

for (i = 0; i != 10; i++)
{
        sum = sum + i;
}
```

MIPS

```
# $s0 = i, $s1 = sum

            add     $s0, $0, $0
            add     $s1, $0, $0
            addi    $t0, $0, 10
            beq     $s0, $t0, done
            add     $s1, $s1, $s0
            addi    $s0, $s0, 1
            j       for
done:
```

Less Than Comparisons

- **The previous for loop can be rewritten by using “less than” operation like below**

Example: Add numbers from 0 to 9

High-level language

```
int sum = 0;
int i;

for (i = 0; i < 10; i++)
{
        sum = sum + i;
}
```

MIPS

```
# $s0 = i, $s1 = sum

            add     $s0, $0, $0
            add     $s1, $0, $0
            addi    $t0, $0, 10
for:        beq     $s0, $t0, done
            add     $s1, $s1, $s0
            addi    $s0, $s0, 1
            j       for
done:
```

Less Than Comparisons

- **To reduce the number of instructions, you can also use slti or sltui instead of slt**

Example: Add numbers from 0 to 9

High-level language

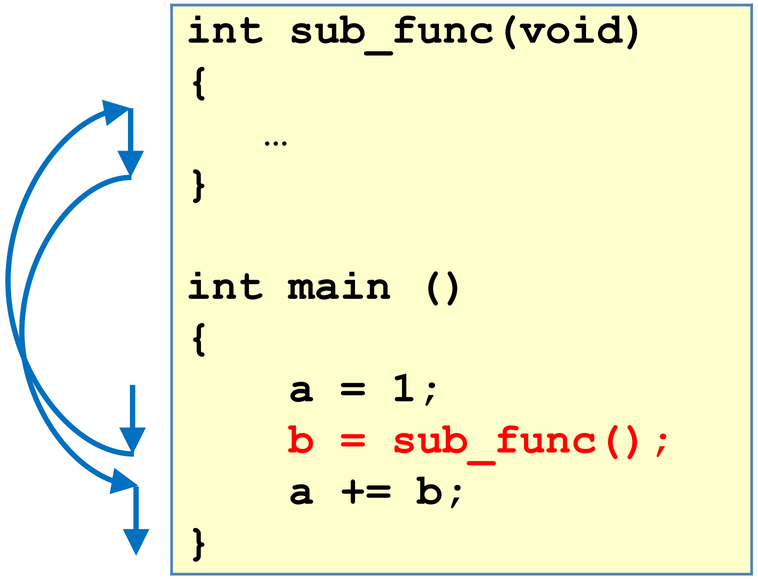
```
int sum = 0;
int i;

for (i = 0; i < 10; i++)
{
        sum = sum + i;
}
```

MIPS

```
# $s0 = i, $s1 = sum

          add     $s0, $0, $0
          add     $s1, $0, $0
for:      slti    $t1, $s0, 10
          beq     $t1, $0, done
          add     $s1, $s1, $s0
          addi    $s0, $s0, 1
          j       for
done:
```

Calling a Subroutine

- **How are subroutines executed?**



```
int sub_func(void)
{
    …
}

int main ()
{
    a = 1;
    b = sub_func();
    a += b;
}
```

How can we jump back to the Caller function, and resume the execution from the immediate following line of the subroutine calling line?

→ We should record the return address before jumping to the subroutine

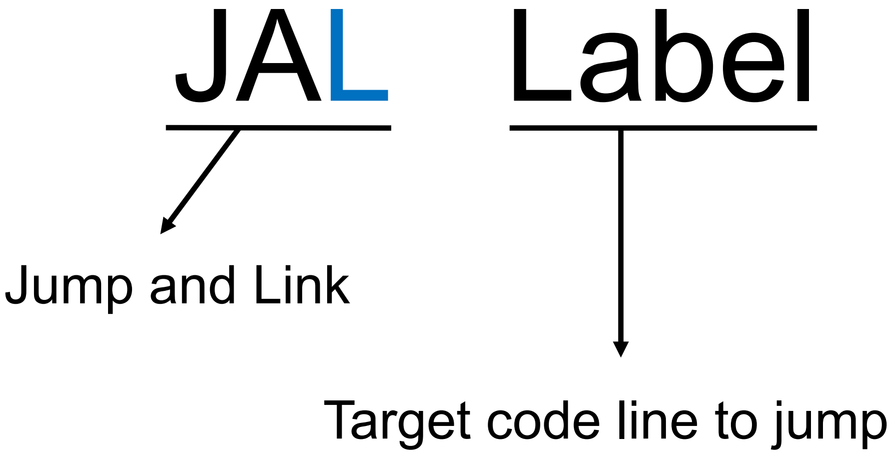
Special Registers

Assembler Name	Register Number	Description
$zero	$0	Constant 0 value
$at	$1	Assembler temporary
$v0-$v1	$2-$3	Function return values
$a0-$a3	$4-$7	Function Arguments
$t0-$t7	$8-$15	Temporaries
$s0-$s7	$16-$23	Saved Temporaries
$t8-$t9	$24-$25	Temporaries
$k0-$k1	$26-$27	Reserved for OS kernel
$gp	$28	Global Pointer (Global and static variables/data)
$sp	$29	Stack Pointer
$fp	$30	Frame Pointer
$ra	$31	Return Address

$ra register holds the return address

Jump and Link Instruction

- **Most of assembly languages provide a special jump (or branch) instruction that Jump + Update Return Address Register**



JAL Label

Jump and Link

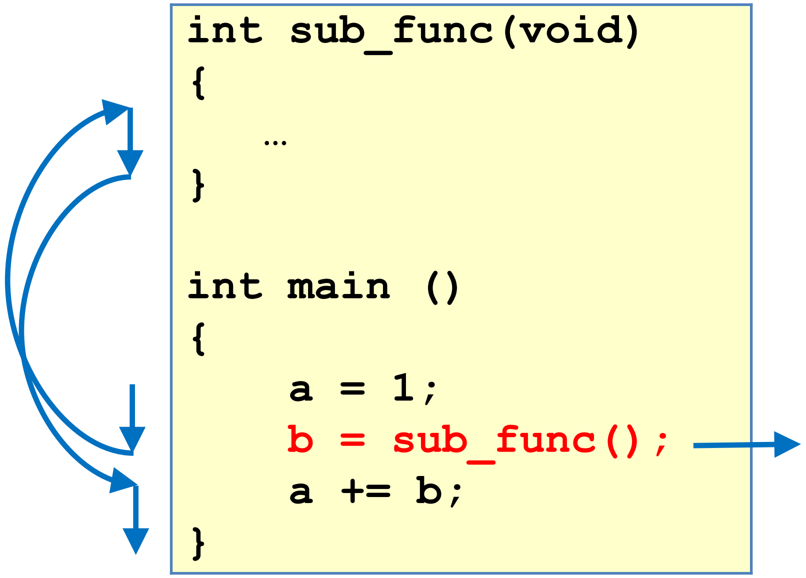
Target code line to jump

1. Jump to Label and

2. Update $ra with return address (JAL's next instruction address)

Calling a Subroutine

- **How can we jump back to address in $ra?**



```
int sub_func(void)
{
    …
}

int main ()
{
    a = 1;
    b = sub_func();
    a += b;
}
```

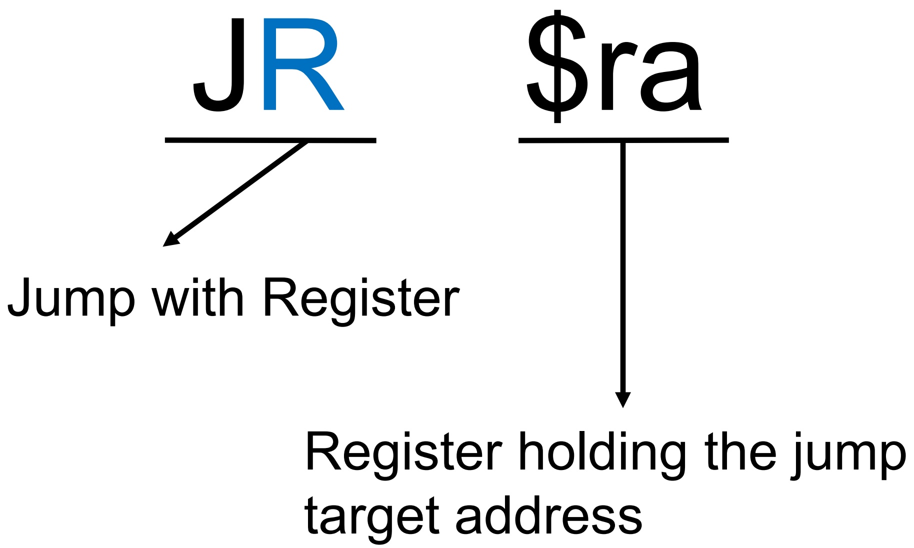
jal sub_func

$pc ← address of first line of sub_func

$ra ← address of a += b;

Jump with Register

- **We can return to the Caller by running Jump with Register instruction with $ra as an operand**



JR $ra

Jump with Register

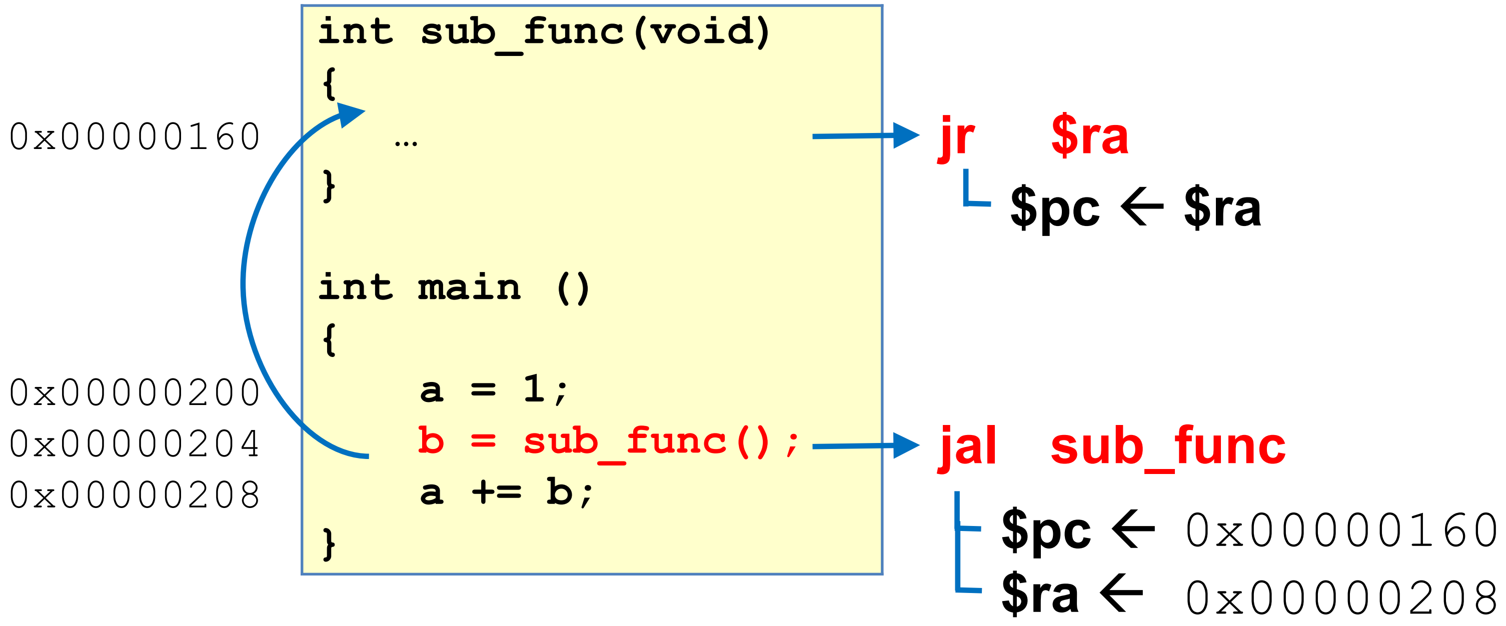
Register holding the jump target address

; Jump to address in $ra

; ($pc = $ra)

Jump with Register

- **Assume that the addresses in the previous example are like below**



```
           int sub_func(void)
           {
0x00000160     …
           }

           int main ()
           {
0x00000200     a = 1;
0x00000204     b = sub_func();
0x00000208     a += b;
           }
```

jr $ra

└ **$pc ← $ra**

jal sub_func

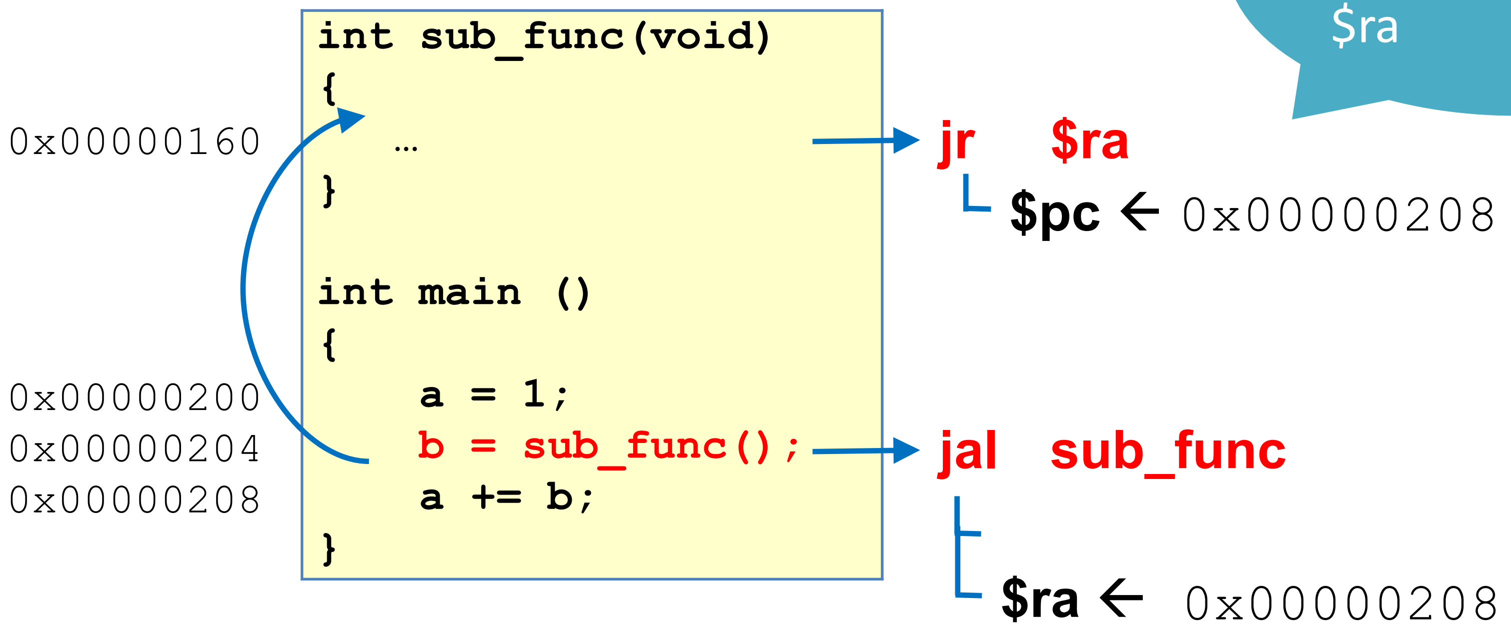
├ **$pc ←** 0x00000160

└ **$ra ←** 0x00000208

Jump with Register

- **Assume that the addresses in the previous example are like below**

Return to the return address in $ra



```
            int sub_func(void)
            {
0x00000160      …
            }

            int main ()
            {
0x00000200      a = 1;
0x00000204      b = sub_func();
0x00000208      a += b;
            }
```

jr $ra

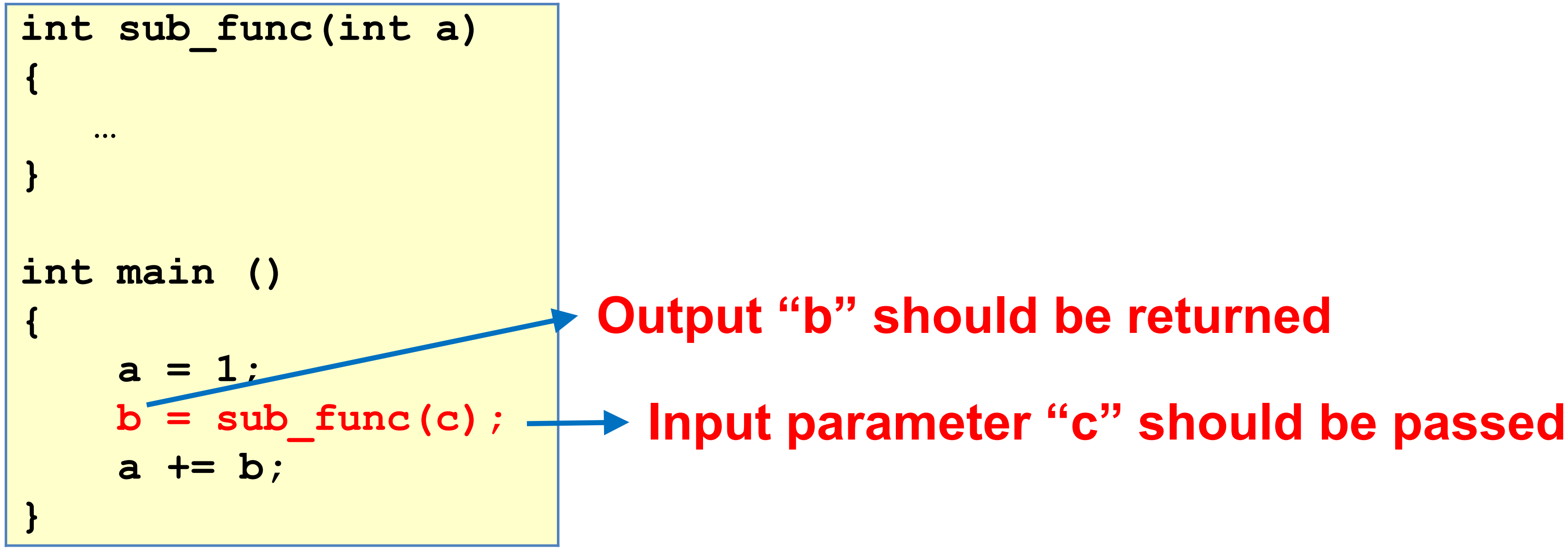
$pc ← 0x00000208

jal sub_func

$ra ← 0x00000208

Parameter Passing

- How can we pass the parameters to/from a subroutine?



```
int sub_func(int a)
{
   …
}

int main ()
{
    a = 1;
    b = sub_func(c);
    a += b;
}
```

Output “b” should be returned

Input parameter “c” should be passed

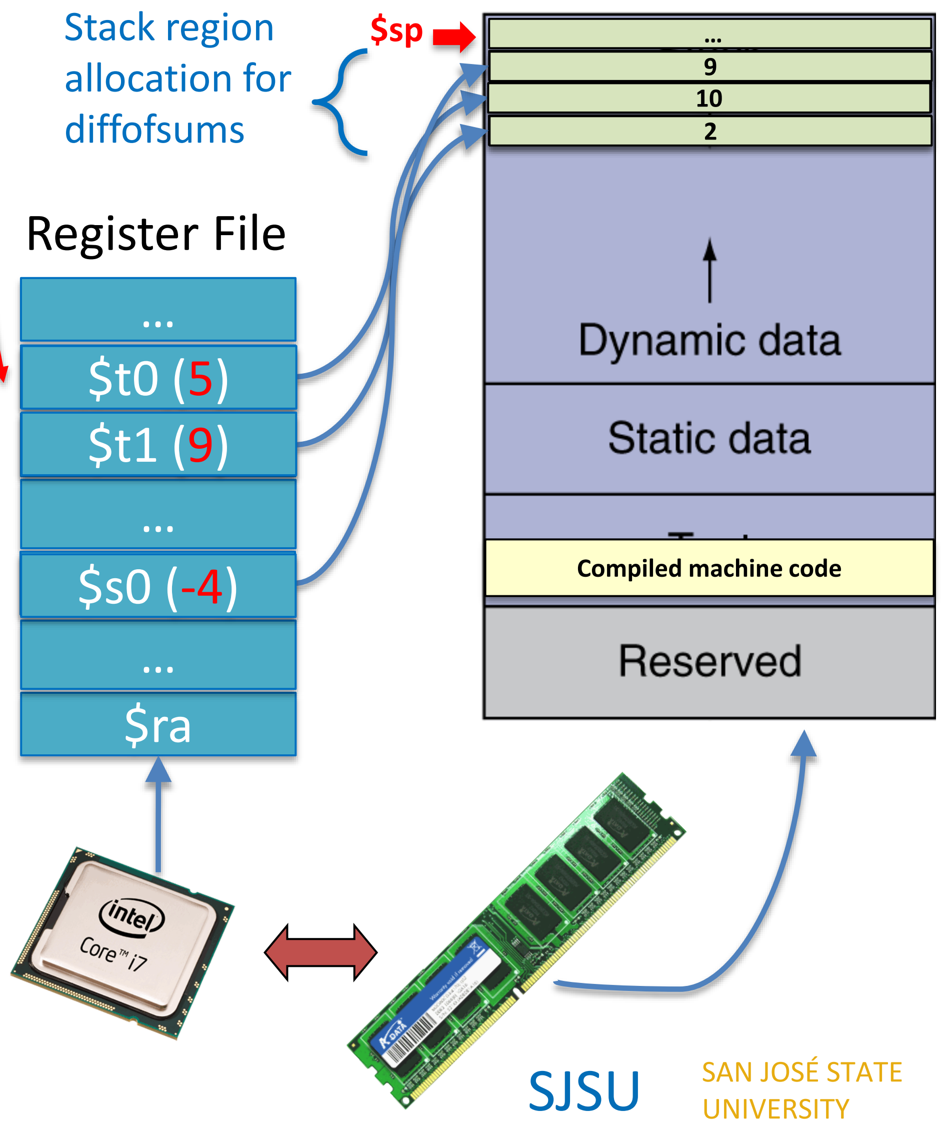
Registers For Parameter Passing

- **Input Parameters**
  - Up to 4 parameters in $a0 ~ $a3
  - If more than 4 parameters, use stack from 5th parameter
- **Return Value**
  - For 32-bit return value, **$v0** is used
  - $v1 also is used when the return value is 64 bits long
    - i.e. $v0 holds the bottom 32 bits and $v1 holds the top 32 bits

Register Value Overwriting

After computations in the function, registers are updated

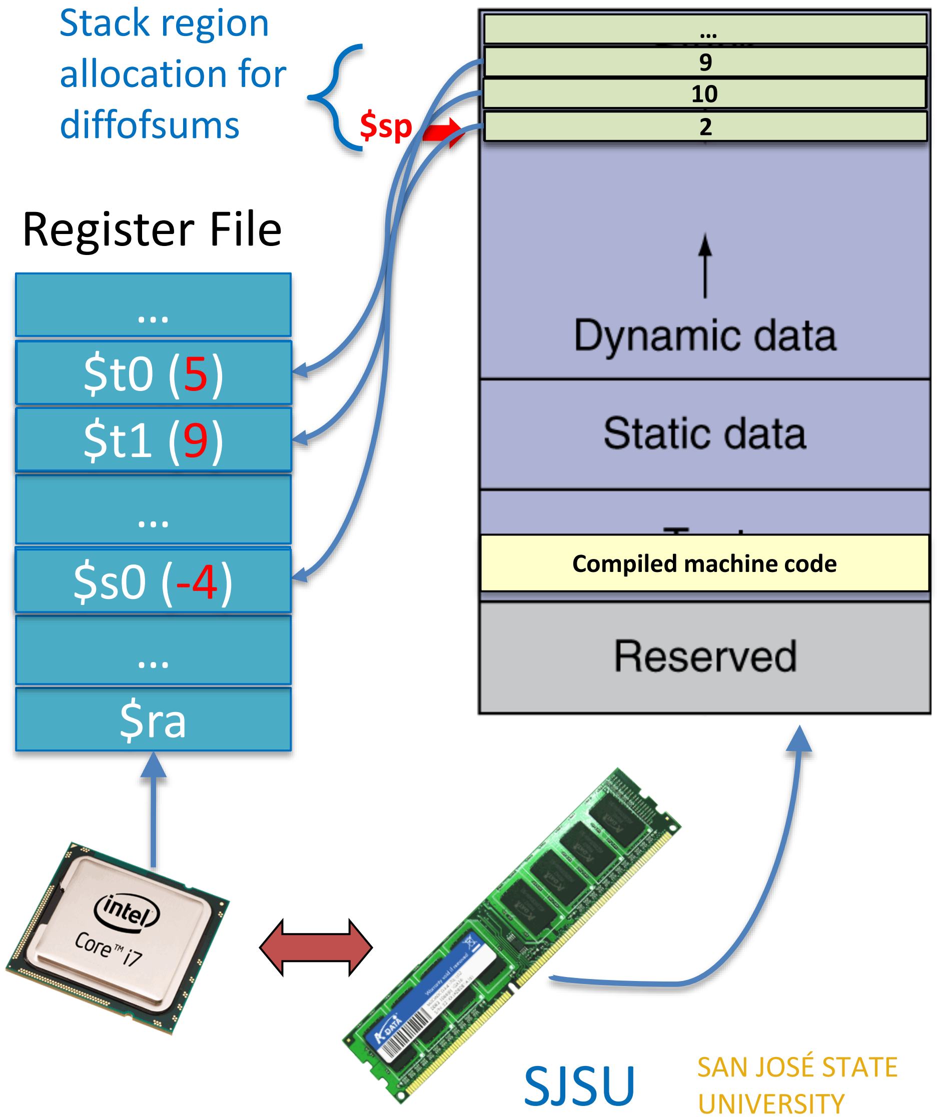
```
diffofsums:
        addi   $sp, $sp, -12        # make space on stack
                                    # to store 3 registers
        sw     $s0, 8($sp)          # save $s0 on stack
        sw     $t0, 4($sp)          # save $t0 on stack
        sw     $t1, 0($sp)          # save $t1 on stack

        add    $t0, $a0, $a1        # $t0 = f + g
        add    $t1, $a2, $a3        # $t1 = h + I
        sub    $s0, $t0, $t1        # result = (f + g) – (h + i)
        add    $v0, $s0, $0         # put return value in $v0

        lw     $t1, 0($sp)          # restore $t1 from stack
        lw     $t0, 4($sp)          # restore $t0 from stack
        lw     $s0, 8($sp)          # restore $s0 from stack
        addi   $sp, $sp, 12         # deallocate stack space

        jr     $ra                  # return to caller
```



Register Value Overwriting

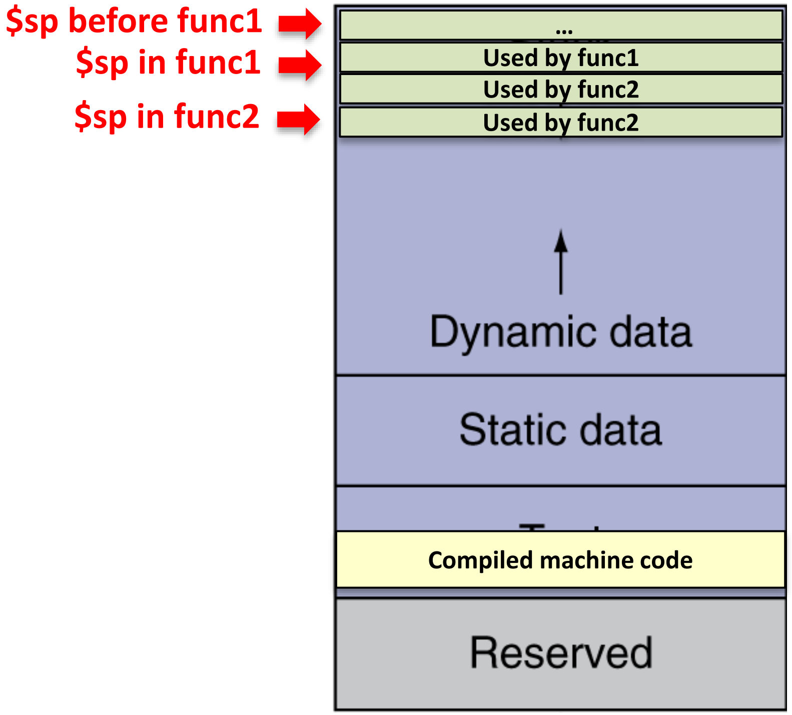
Before returning to Caller,
Register values are revoked

```
diffofsums:
        addi   $sp, $sp, -12        # make space on stack
                                    # to store 3 registers
        sw     $s0, 8($sp)          # save $s0 on stack
        sw     $t0, 4($sp)          # save $t0 on stack
        sw     $t1, 0($sp)          # save $t1 on stack

        add    $t0, $a0, $a1        # $t0 = f + g
        add    $t1, $a2, $a3        # $t1 = h + I
        sub    $s0, $t0, $t1        # result = (f + g) – (h + i)
        add    $v0, $s0, $0         # put return value in $v0

        lw     $t1, 0($sp)          # restore $t1 from stack
        lw     $t0, 4($sp)          # restore $t0 from stack
        lw     $s0, 8($sp)          # restore $s0 from stack
        addi   $sp, $sp, 12         # deallocate stack space

        jr     $ra                  # return to caller
```



Example: Nested Function Call

```
func1:
        addi   $sp, $sp, -4          # make space on stack
                                     # to store $ra register
        sw     $ra, 0($sp)           # save $ra on stack

        jal    func2                 # jump to func2
        …

        lw     $ra, 0($sp)           # restore $ra from stack
        addi   $sp, $sp, 4           # deallocate stack space

        jr     $ra                   # return to caller


func2:
        addi   $sp, $sp, -8          # make space on stack
                                     # to store 2 registers
        …
        addi   $sp, $sp, 8           # deallocate stack space

        jr     $ra                   # return to func1 (caller)
```



Why does func1 store $ra in stack before calling func2?

Example: Recursion

- **Recursive functions should keep its input parameters and return address to the stack because all the recursions will try to use the same registers for these.**

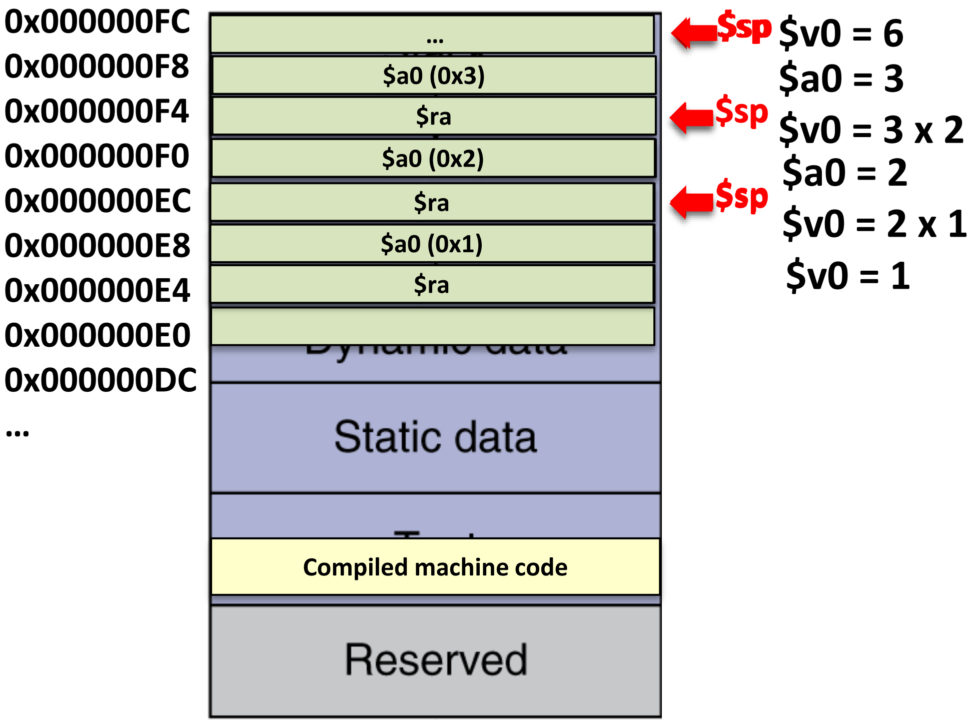
High-level language

```
Int factorial (int n)
{
        if (n <= 1)
                return 1;
        else
                return (n * factorial (n-1));
}
```

MIPS Assembly

```
factorial:    addi   $sp, $sp, -8  # make room
              sw     $a0, 4($sp)   # store input (n)
              sw     $ra, 0($sp)   # store return address
              addi   $t0, $0, 2    # $t0 = 2
              slt    $t0, $a0, $t0 # n <= 1?
              beq    $t0, $0, else # no: go to else (recursion)
              addi   $v0, $0, 1    # yes: return 1
              addi   $sp, $sp, 8   # restore $sp
              jr     $ra           # return
else:         addi   $a0, $a0, -1  # n = n -1
              jal    factorial     # recursive call
              lw     $ra, 0($sp)   # restore return address
              lw     $a0, 4($sp)   # restore input
              addi   $sp, $sp, 8   # restore $sp
              mul    $v0, $a0, $v0          # n * factorial(n-1)
              jr     $ra           # return
```

Example: Recursion for 3!



```
factorial:  addi  $sp, $sp, -8  # make room
            sw    $a0, 4($sp)   # store input (n)
            sw    $ra, 0($sp)   # store return address
            addi  $t0, $0, 2    # $t0 = 2
            slt   $t0, $a0, $t0 # n <= 1?
            beq   $t0, $0, else # no: go to else (recursion)
            addi  $v0, $0, 1    # yes: return 1
            addi  $sp, $sp, 8   # restore $sp
            jr    $ra           # return
else:       addi  $a0, $a0, -1  # n = n -1
            jal   factorial     # recursive call
            lw    $ra, 0($sp)   # restore return address
            lw    $a0, 4($sp)   # restore input
            addi  $sp, $sp, 8   # restore $sp
            mul   $v0, $a0, $v0         # n * factorial(n-1)
            jr    $ra           # return
```

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY