



GMU CS695 Advanced Computer Architecture

Spring 2023

Lecture 2

# Processor Instruction Set Architecture & Language

Reference:

CMPE 140, Hyeran Jeon

CMPE 140, Donald Hung

CMPE 140, Haonan Wang

Computer Architecture: A Quantitative Approach

Licensed for use under a Creative Commons Attribution-  
NonCommercial-ShareAlike 3.0 Unported License.

Lishan Yang



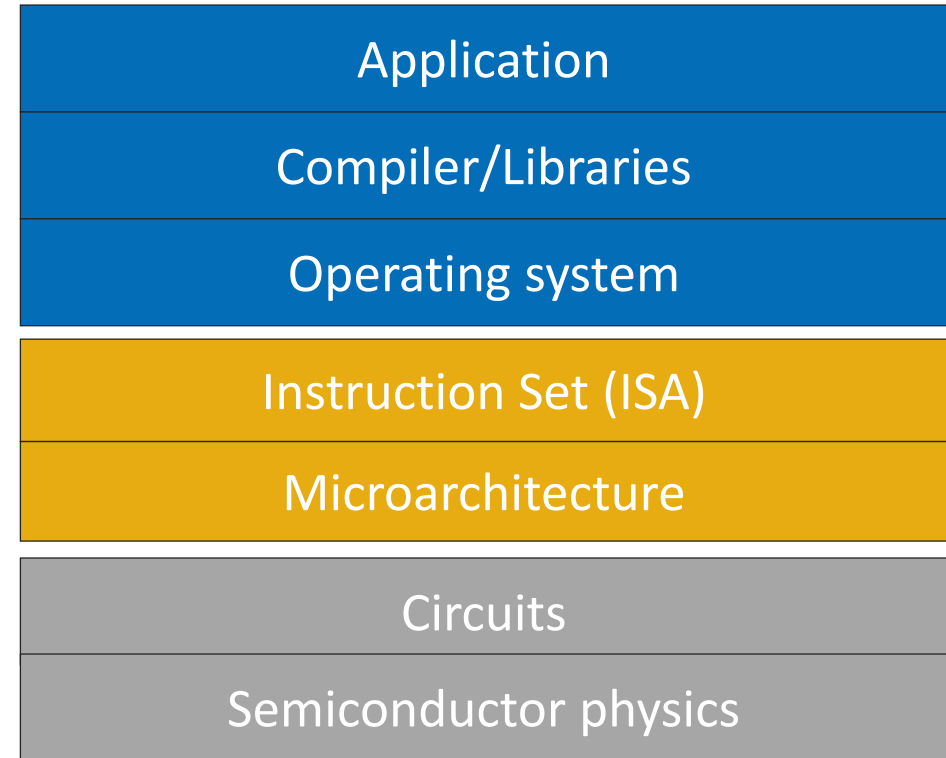
# Instruction Set Architecture & Microarchitecture

- **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



# 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)

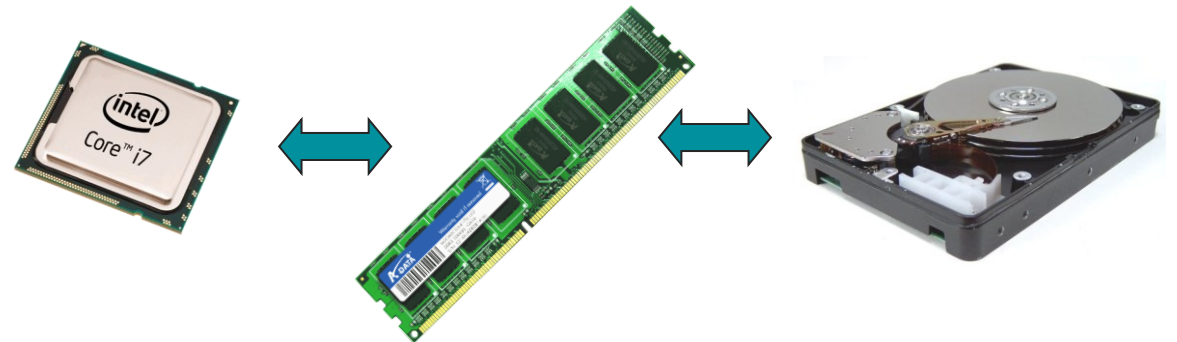
Provides  
operands to  
CPU

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

## Storage (HDD)

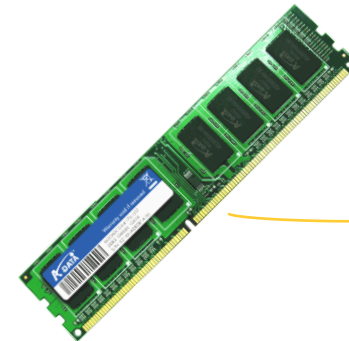
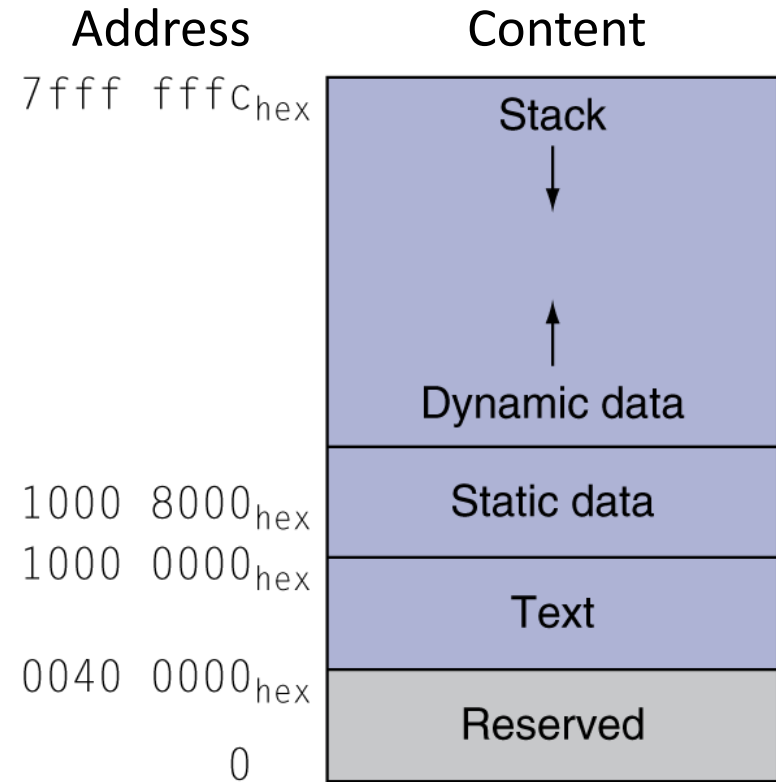
Provides file  
inputs and  
program code

(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



# Memory Layout

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

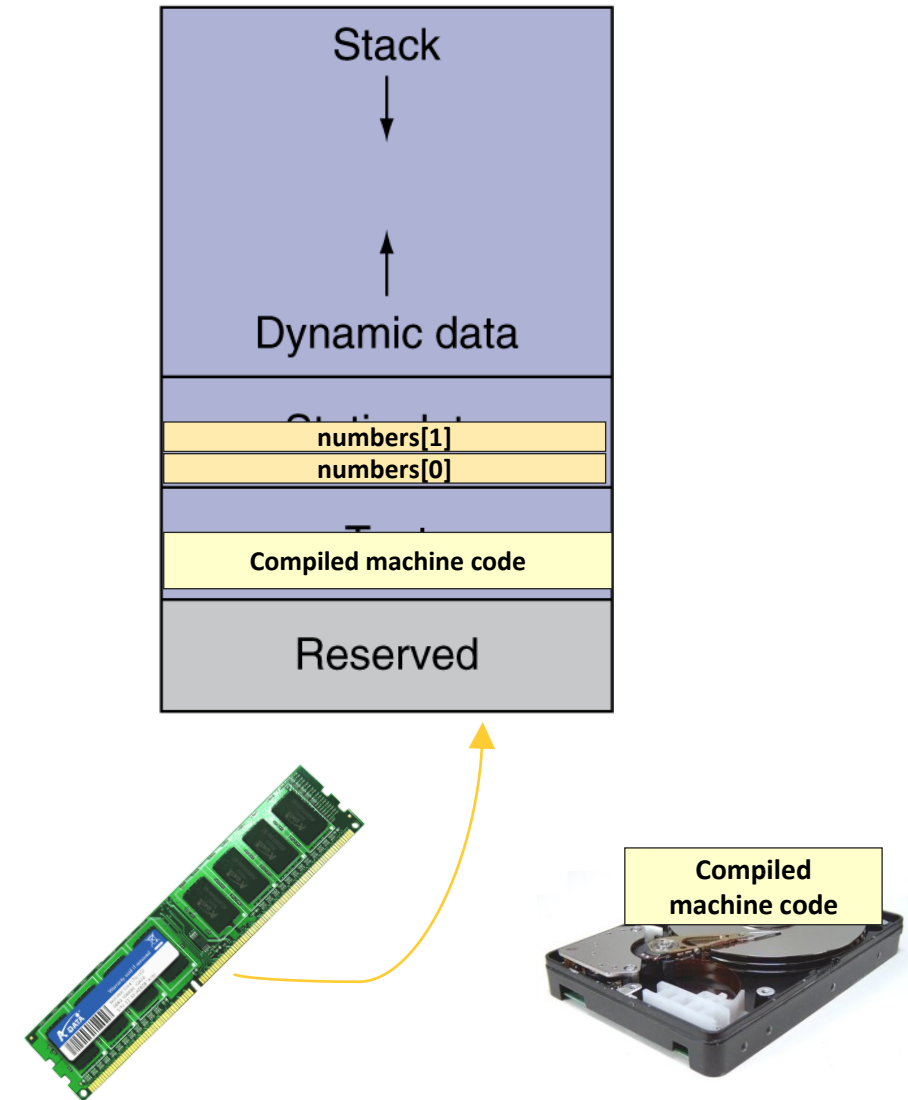
int numbers[2];

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];

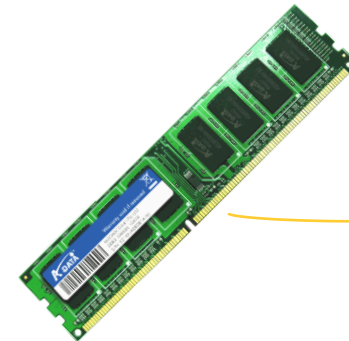
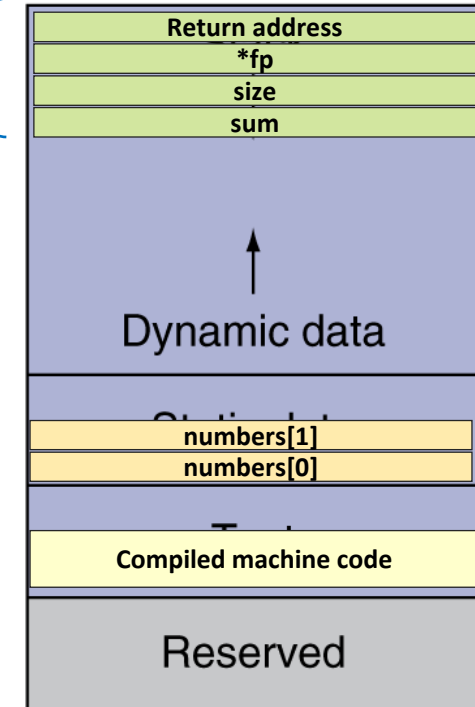
void myfunction(void)
{
    FILE *fp;
    int size = 2;
    int sum = 0;

    /* 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];

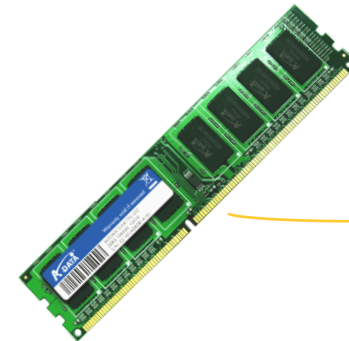
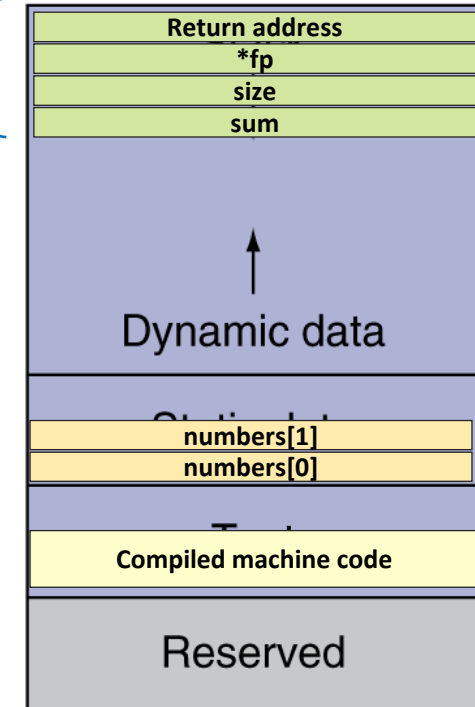
void myfunction(void)
{
    FILE *fp;
    int size = 2;
    int sum = 0;

    /* 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

Sizes

Speed

100s of Bytes

(CPU)

Register

10 ns

KB ~ MB

Cache

100 ns

MB ~ GB

Memory

1  $\mu$ s

GB ~ TB

Disk

1 ms

>TB

Tape

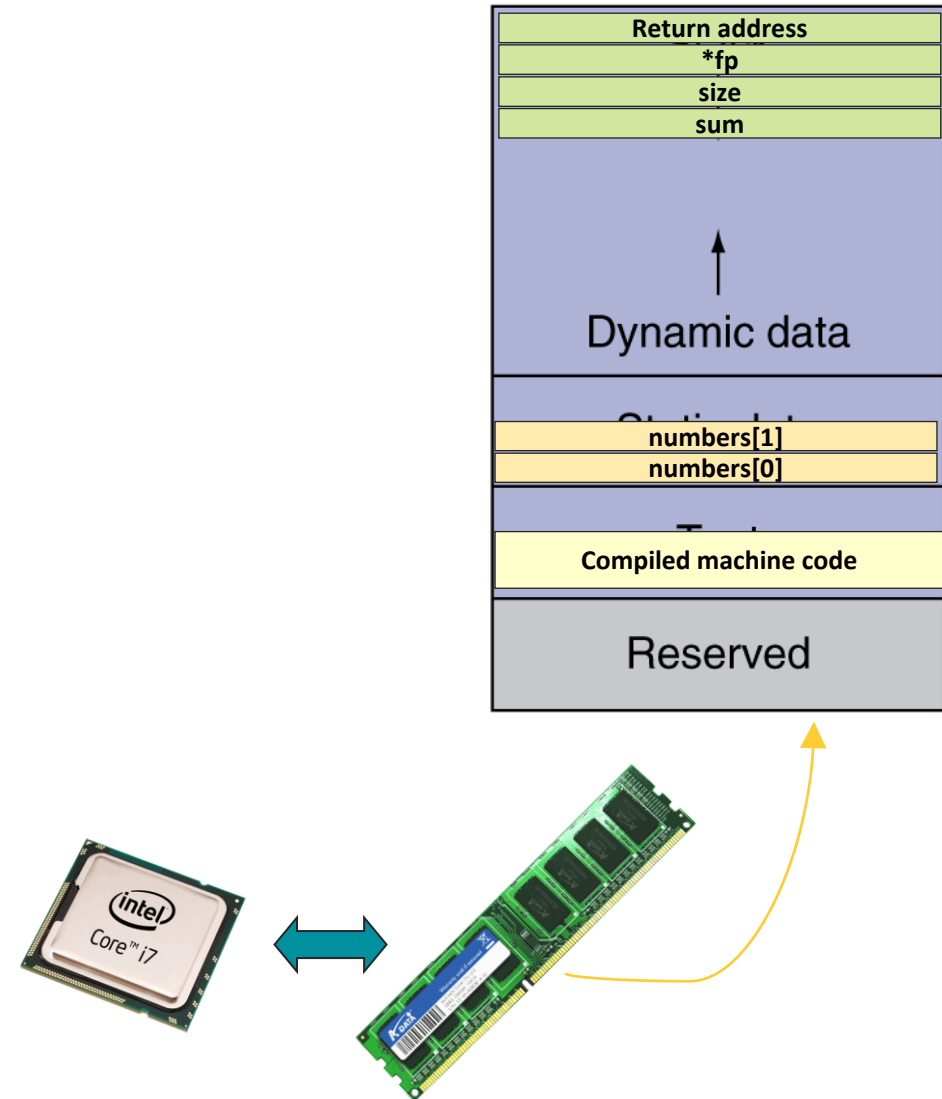
1 sec





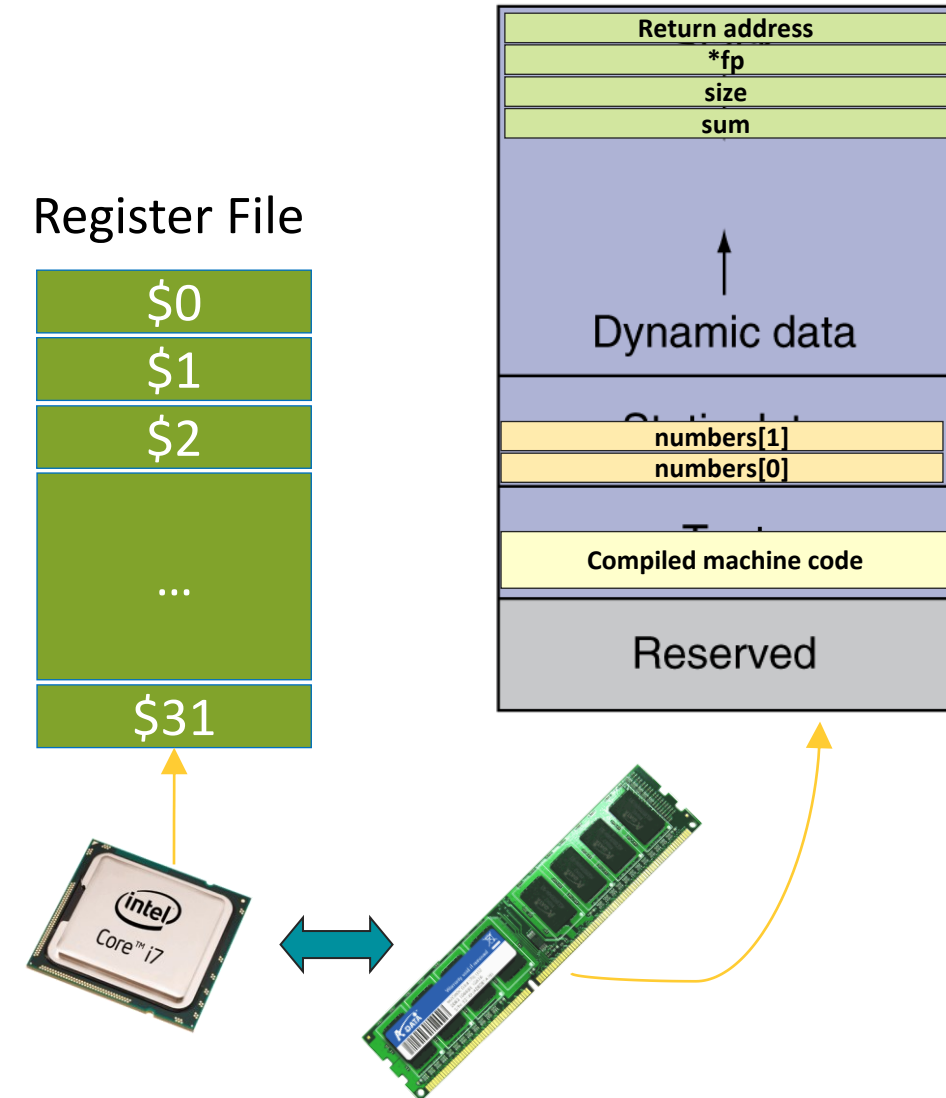
# Memory Access Is Slow

- Variables are all stored in Memory, which is **outside of CPU**
  - Slow..
- How can we execute the operations faster?
  - Use on-chip memory



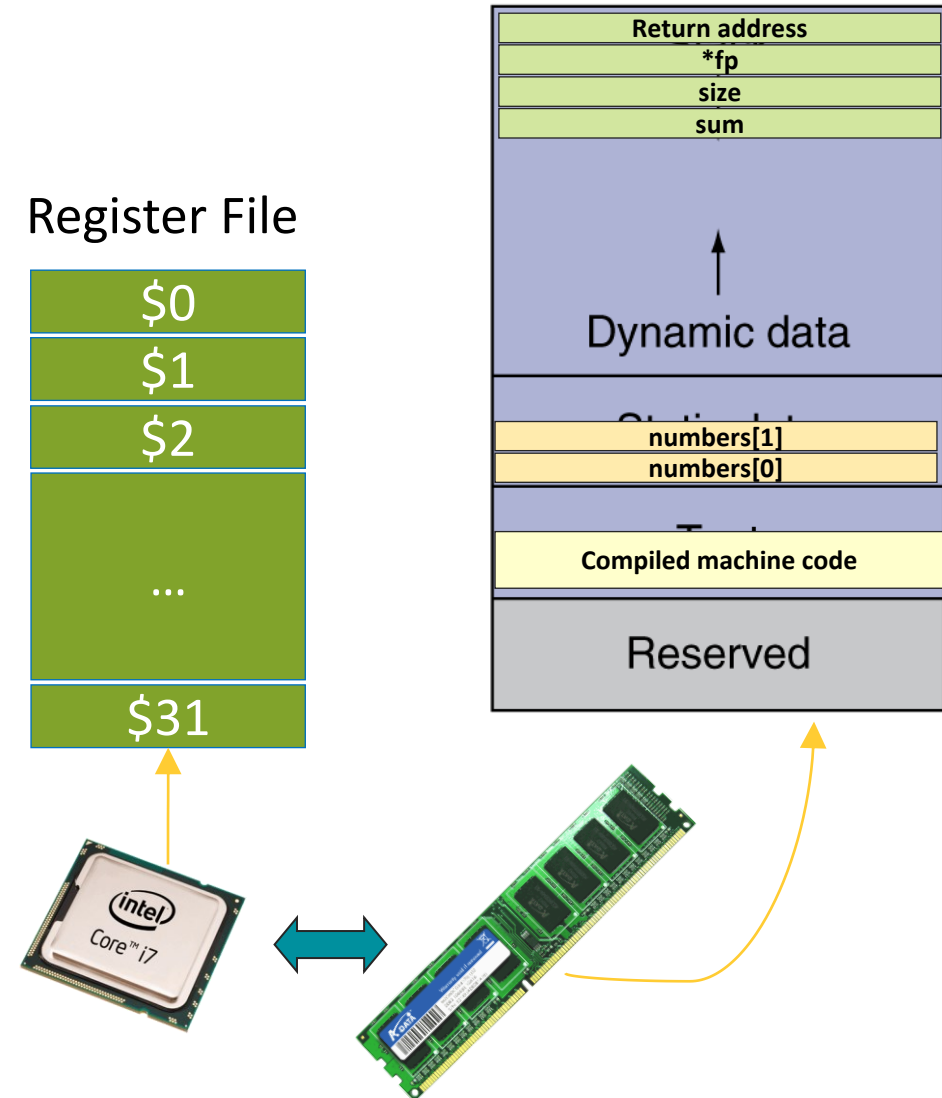
# Register File

- **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 : 0<sup>th</sup> register,  
\$1 : 1<sup>st</sup> register



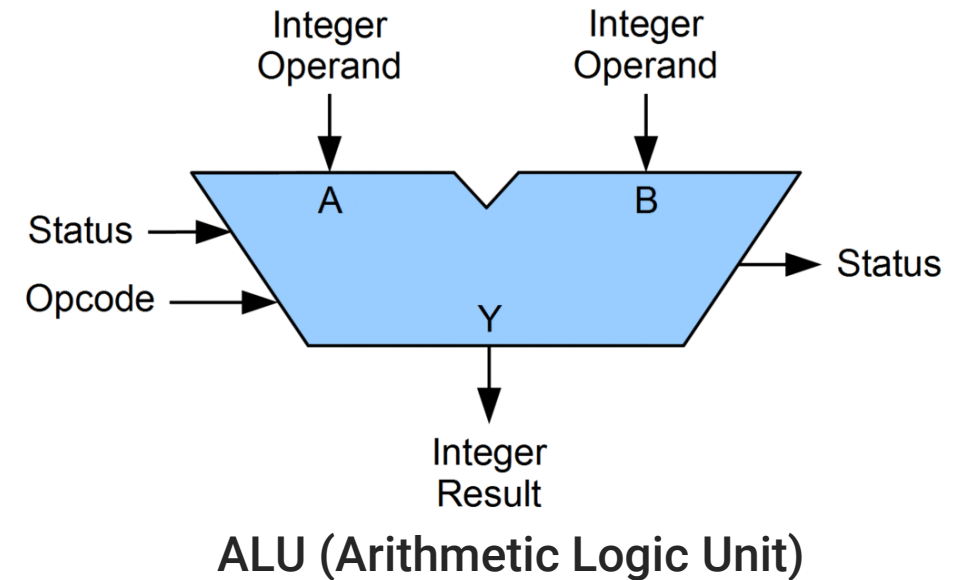
# Typical Execution Steps

1. Load operand values from memory to registers
2. Do computation on registers
3. Move the results from register to memory



# 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

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

Compile

MIPS Assembly code

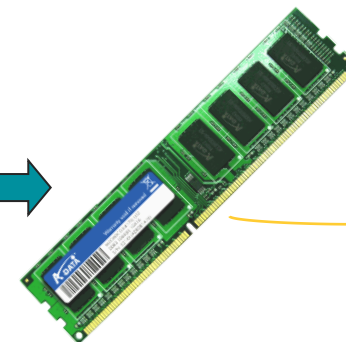
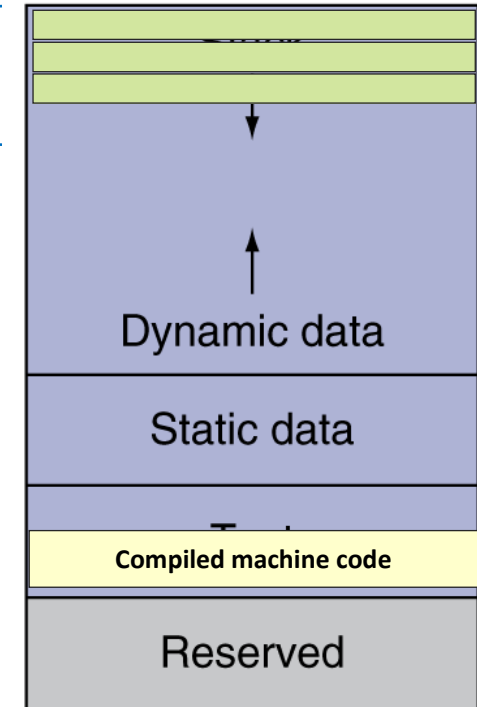
executing line

```
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
```

Stack region  
allocation for test()

Register File

\$0
\$1
\$2
\$3
...
\$31



# Code Example: Memory & Registers

C code

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

Compile

MIPS Assembly code

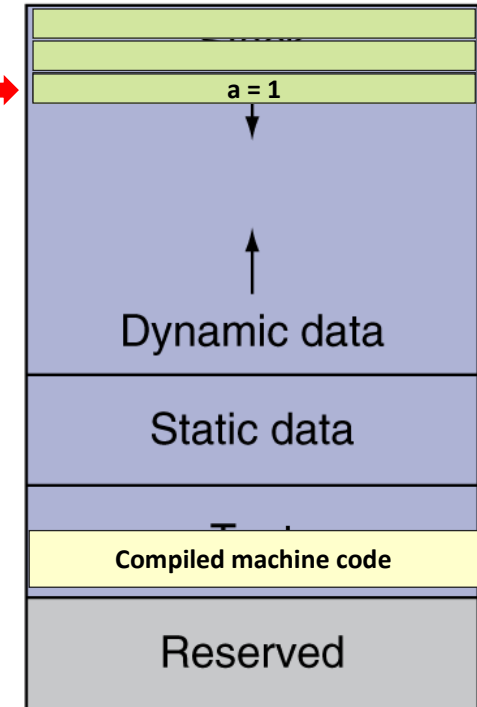
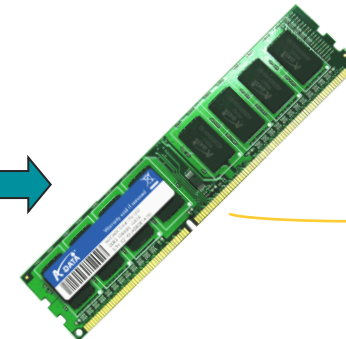
```
test():  
    addiu    $sp,$sp,-32  
    sw       $fp,28($sp)  
    move     $fp,$sp  
    li       $2,1                                # 0x1  
    sw       $2,8($fp)  
    li       $2,2                                # 0x2  
    sw       $2,12($fp)  
    lw       $3,8($fp)  
    lw       $2,12($fp)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

\$0
\$1
\$2
\$3
...
\$31





# Code Example: Memory & Registers

C code

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

Compile

MIPS Assembly code

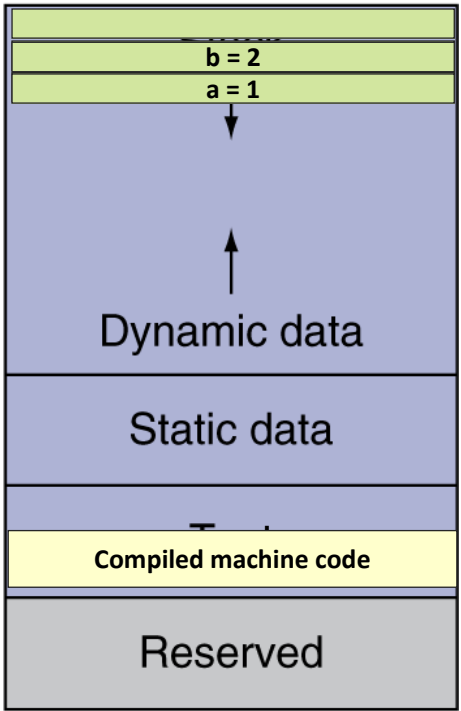
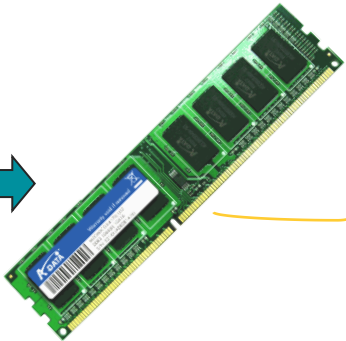
```
test():  
    addiu    $sp,$sp,-32  
    sw      $fp,28($sp)  
    move     $fp,$sp  
    li      $2,1                # 0x1  
    sw      $2,8($fp)  
    li      $2,2                # 0x2  
    sw      $2,12($fp)  
    lw      $3,8($fp)  
    lw      $2,12($fp)  
    nop  
    addu     $2,$3,$2 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

executing line



Register File

\$0
\$1
\$2
\$3
...
\$31



# Code Example: Memory & Registers

C code

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

Compile

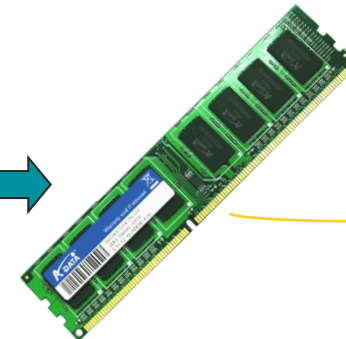
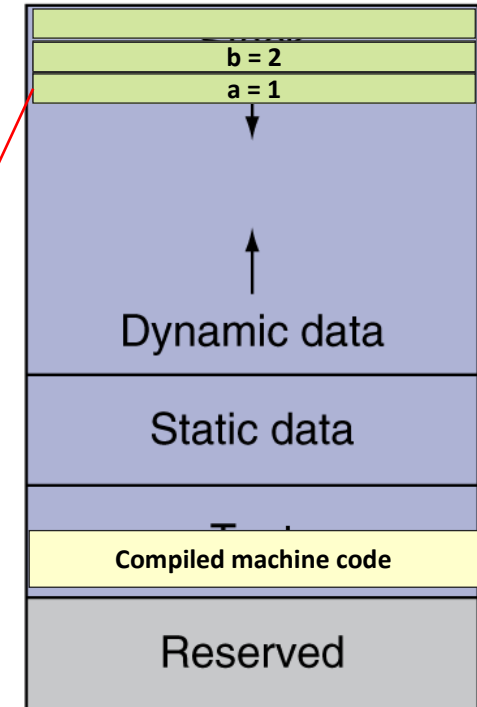
MIPS Assembly code

```
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









# Registers of MIPS CPUs

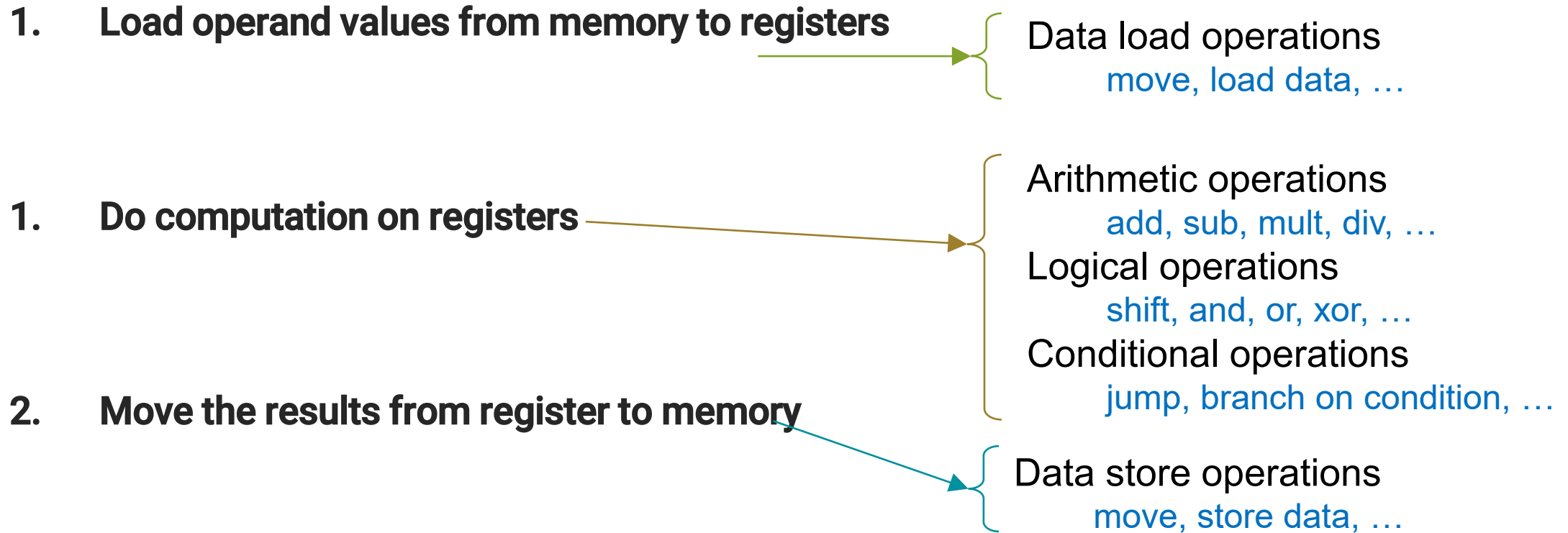
Assembler Name	Register Number	Description
\$zero	\$0	Constant 0 value
\$at	\$1	Assembler temporary
\$v0-\$v1	\$2-\$3	Function return values
\$a0-\$a3	\$4-\$7	Function Arguments
\$t0-\$t7	\$8-\$15	Temporaries
\$s0-\$s7	\$16-\$23	Saved Temporaries
\$t8-\$t9	\$24-\$25	Temporaries
\$k0-\$k1	\$26-\$27	Reserved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





# A Brief Review

- One line of HLL can involve:



# Operations of MIPS CPUs

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

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

# Exercise: Register Reuse

- **Example:**

- Assume g, h, i and j are loaded to registers, \$t0, \$t1, \$t2, \$t3

- HLL:

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

- Assembly (with add & sub operations)?

- add \$t0, \$t0, \$t1

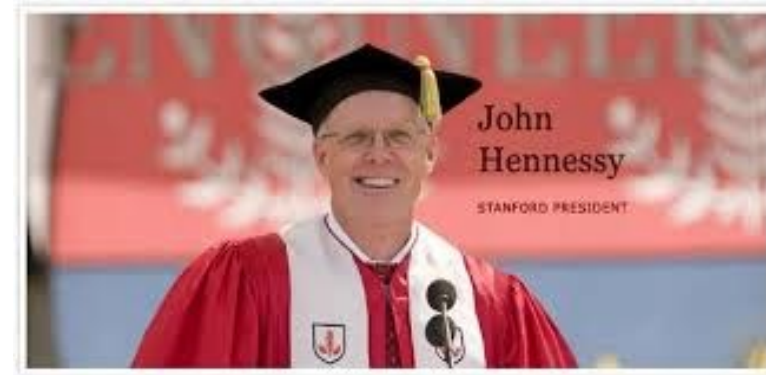
- add \$t2, \$t2, \$t3

- sub \$t2, \$t0, \$t2

- **Register values are maintained unless overwritten**

# 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...



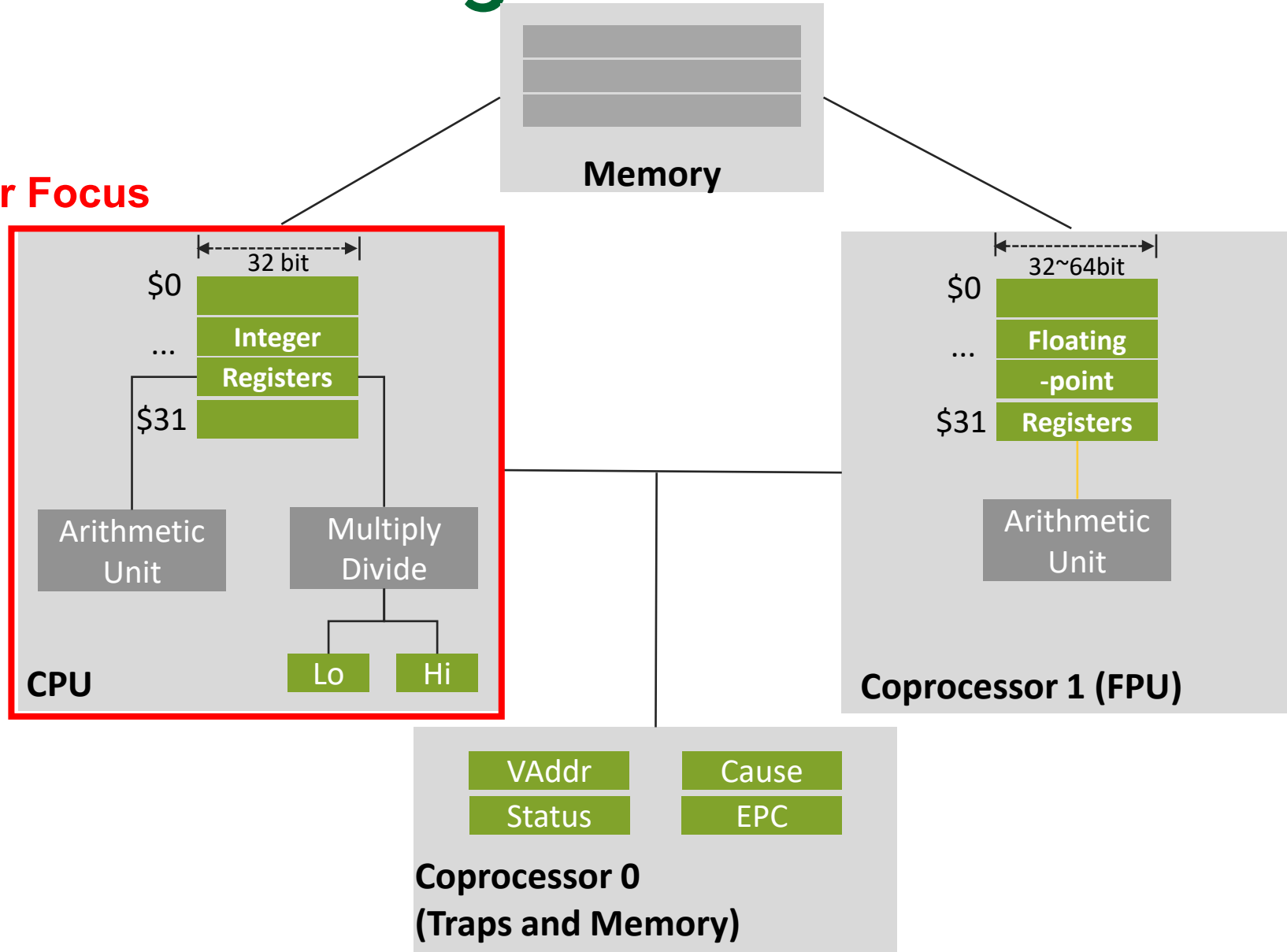
# Two types of Computers

Check RISC-V  
<https://riscv.org/>

- **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

**Our Focus**



# 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)

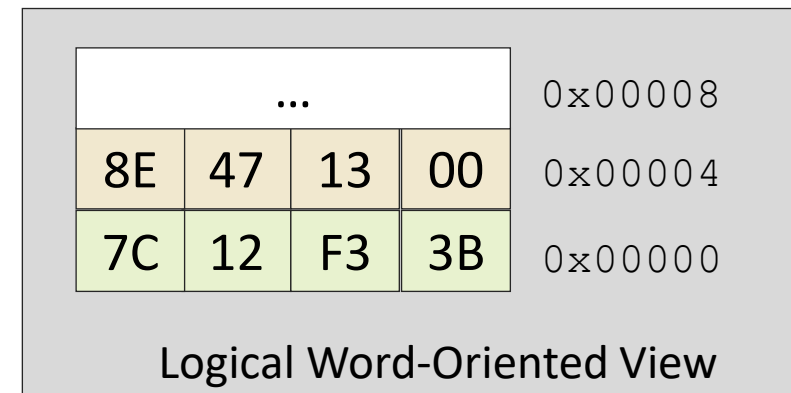
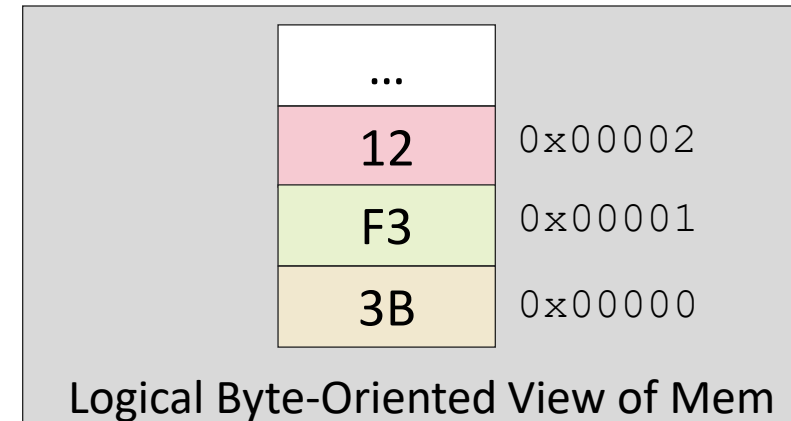
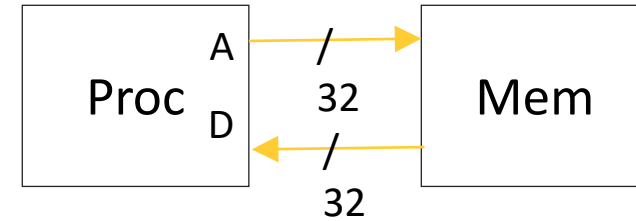


# MIPS Data Sizes

- **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

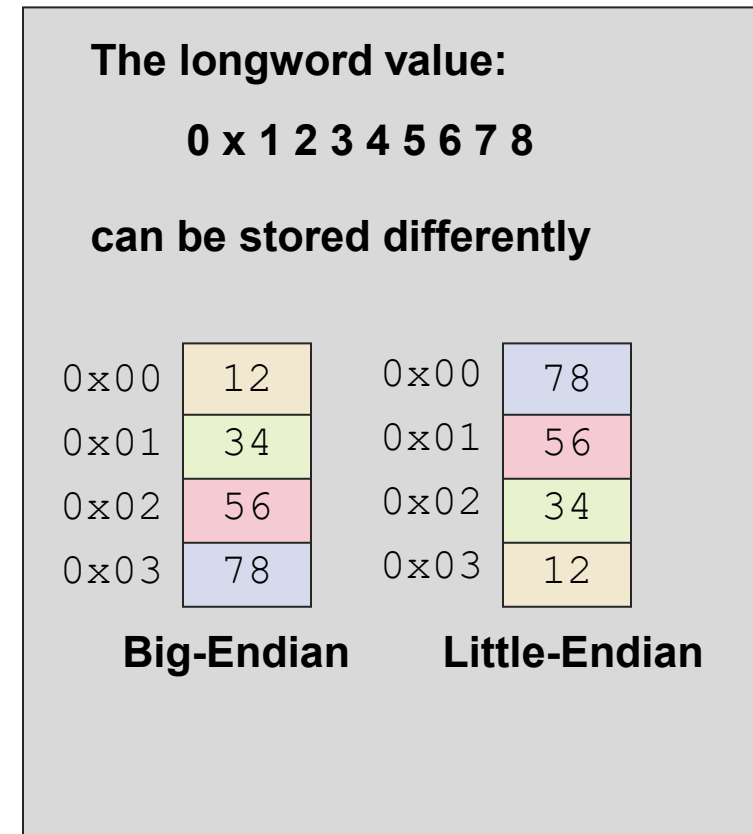
# Byte-oriented vs. Word-oriented Memory

- **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

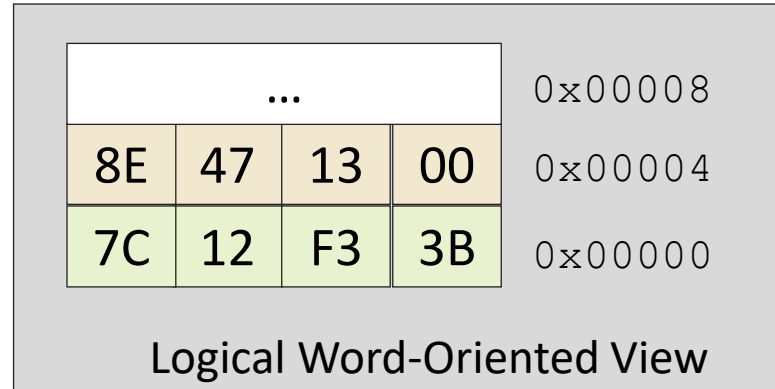


# Endian-ness

- **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



# 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

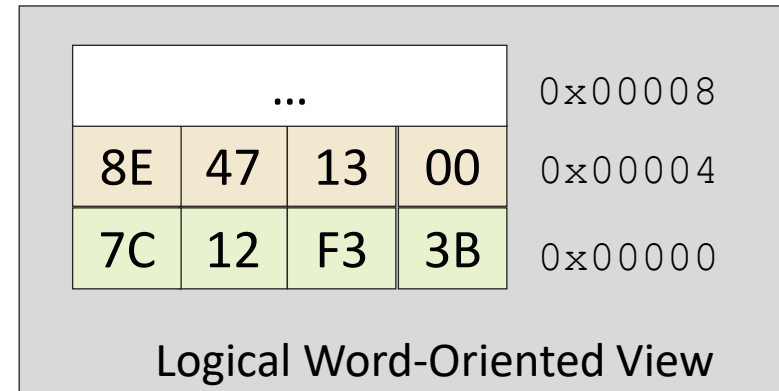
0x0000 is ( 0x3B)

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 )

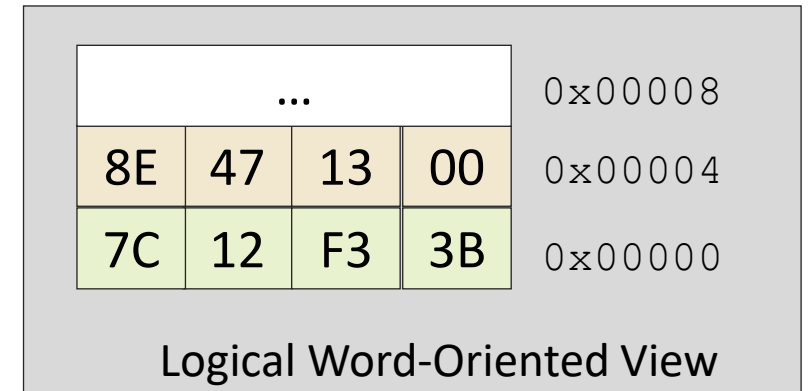
0x0001 is ( Incorrect ) addressing

0x0002 is ( 0x7C12 )

The word value in address

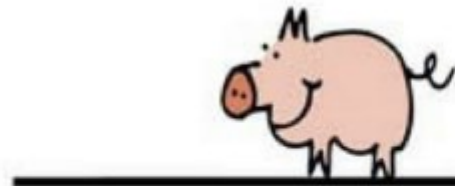
0x0002 is ( Incorrect ) addressing

0x0004 is ( 0x8E471300 )



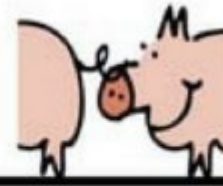
# Tips to Remember Endianness

SIMPLY EXPLAINED



BIG-ENDIAN

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



LITTLE-ENDIAN

oxCA...  
will be  
BE | BA

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

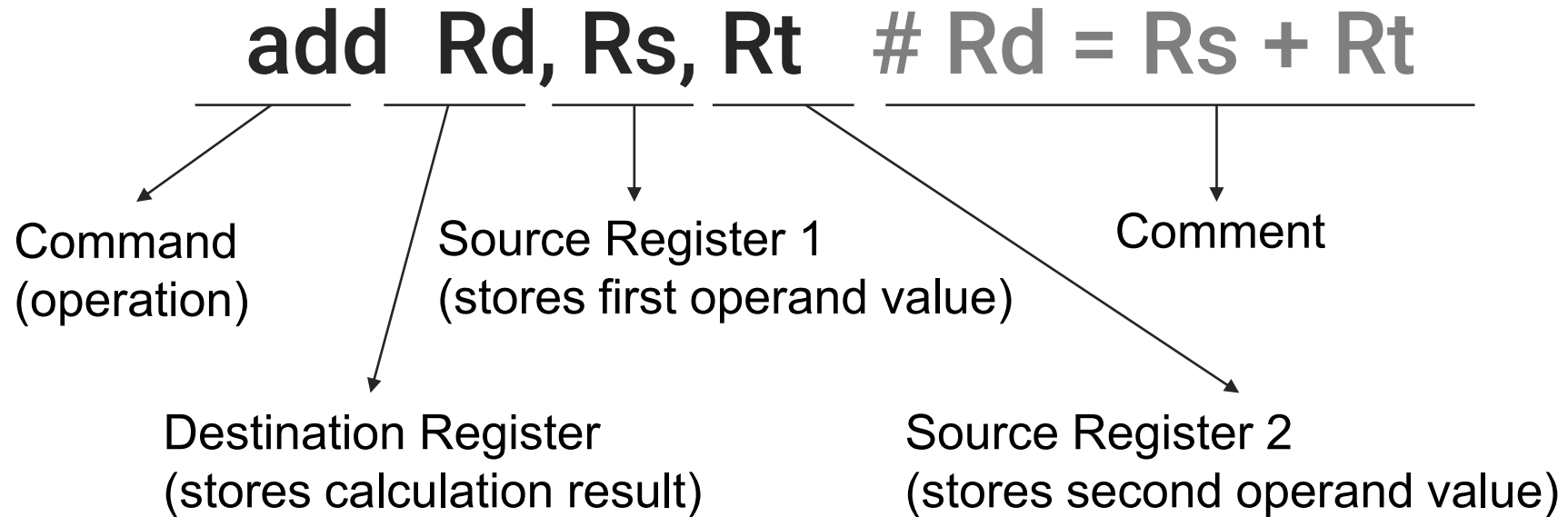
Looks normal (same as  
writing order) when address  
is from low to high

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



# Basic Instruction Formats

- **Example: Format with 3 registers**



# 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 (destination)**

- **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

add \$3, \$3, \$4

slt     \$2, \$3, \$4

#  $2 = 10 - 8$

#  $10 = 8 + 2$

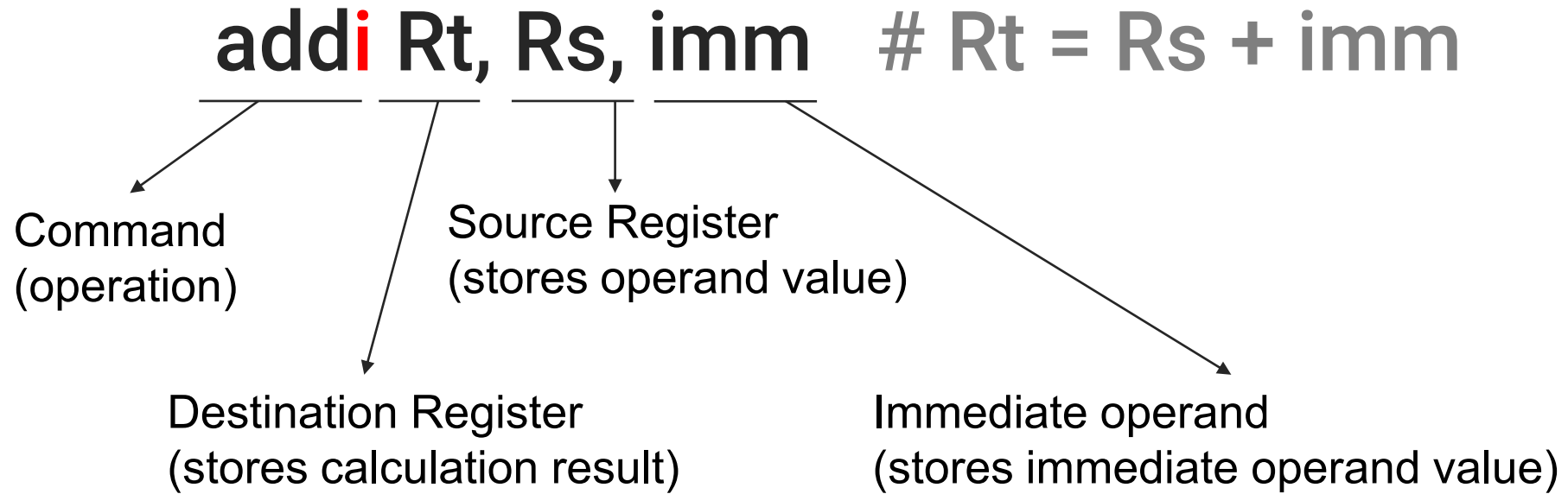
#  $(10 < 2)? \rightarrow \text{No}$   
→  $\$2 = 0$

Register File

\$0
\$1
\$2 (0)
\$3 (10)
\$4 (2)
...
\$31

# Instructions Using Immediate Value

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



# 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

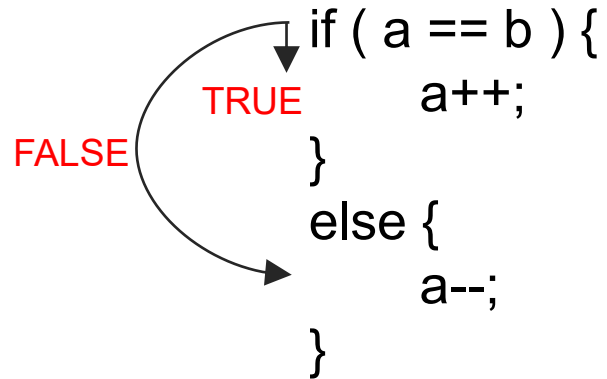
- We call the instructions that use **two registers** and **one immediate value** as **I-type Instructions**
  - 1 register and 1 immediate value as source, 1 register for result
- **Examples:**
  - **subi**      Rt, Rs, imm      # Rt = Rs – imm
  - **andi**      Rt, Rs, imm      # Rt = Rs & imm
  - **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:**



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

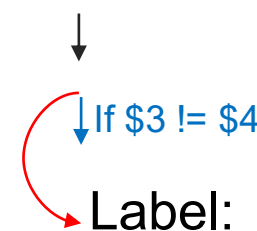
- **Example:**

↓  
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

If \$3 == \$4 (red text)      If \$3 != \$4 (blue text)



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

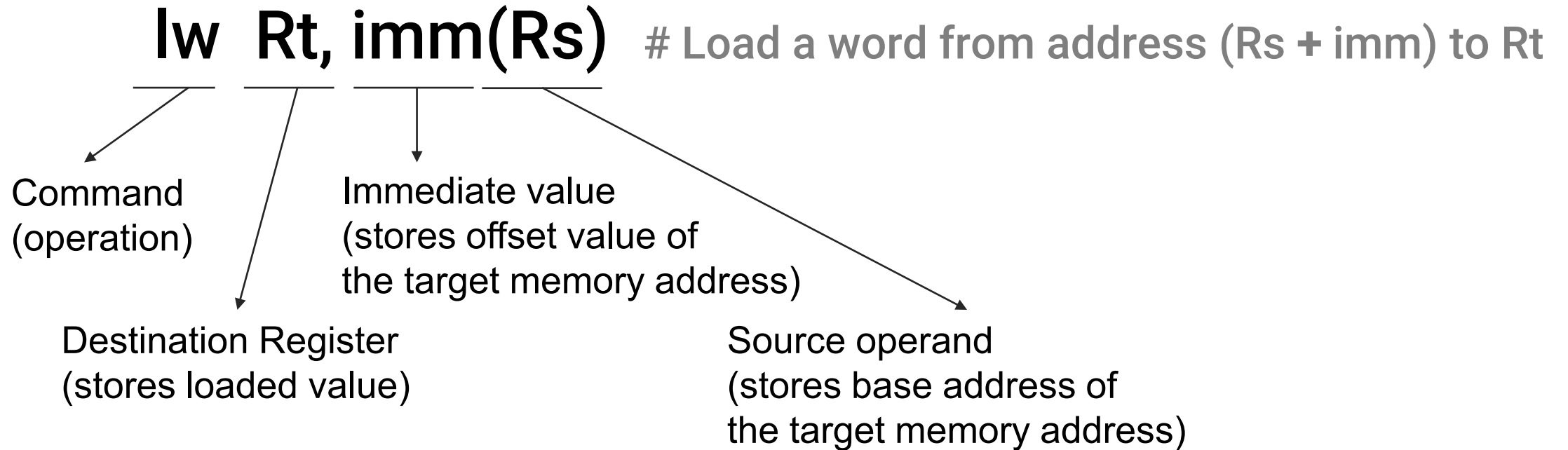
Comparing the same  
register values  
→ always equal

## Pseudo-instruction

Instruction “b imm” is not  
actually supported by the  
hardware.  
Assembler replaces “b imm”  
to “beq \$0, \$0, imm”

# I-Type: Memory Instructions

- Memory operations use special format to present memory address
- Example:

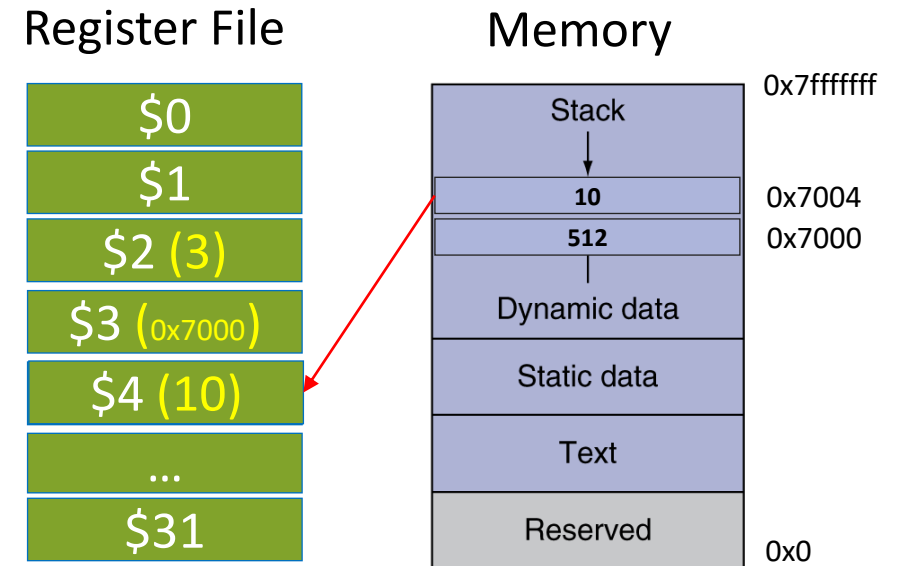


# I-Type: Memory Instructions

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

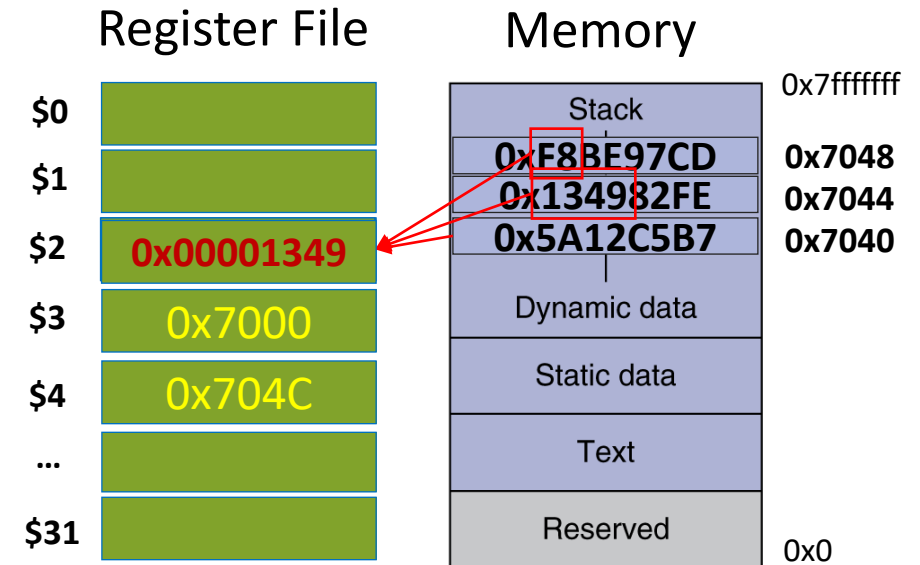
# I-Type: Memory Instructions

- **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.



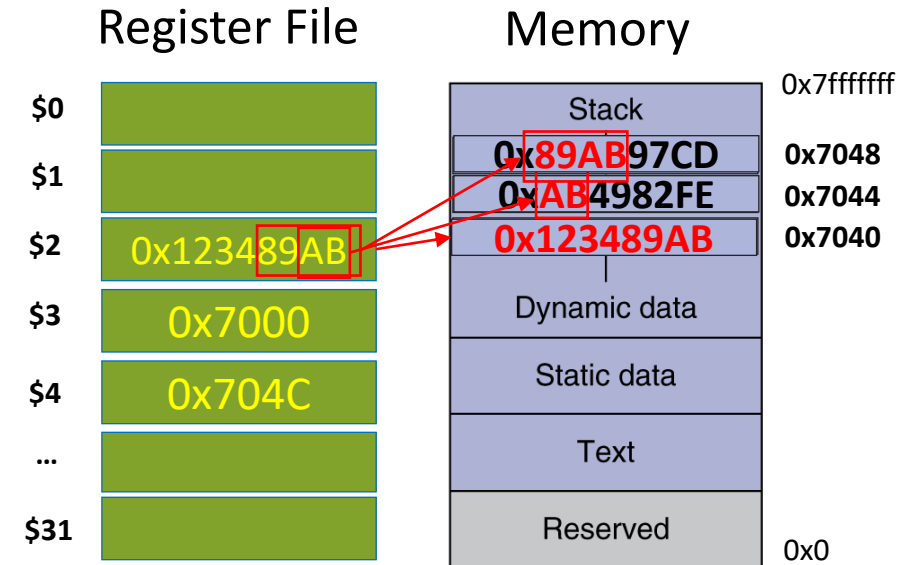
# I-Type: Memory Instructions

- **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
  - **lb<sub>u</sub> \$2, -1(\$4)**      # load byte unsigned
    - # If lb in the earlier example is lb<sub>u</sub>,
    - # \$2 will be updated with **0x000000F8**
  - **lh<sub>u</sub> \$2, -6(\$4)** # load half-word unsigned
    - # If lh in the earlier example is lh<sub>u</sub>,
    - # \$2 will be updated with **0x00001349**



# Signed vs. Unsigned Instructions

# Exercise

- 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

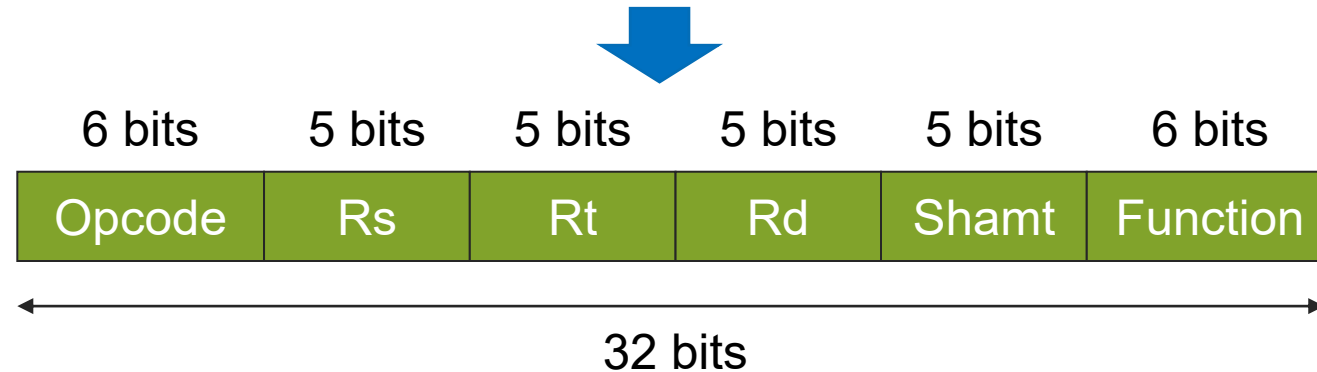
MIPS

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

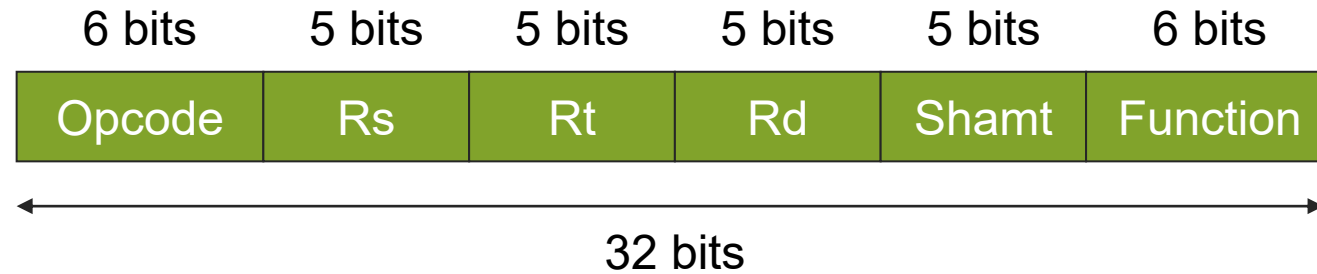
# Machine Code of R-Type Instructions

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

add Rd, Rs, Rt



# Machine Code of R-Type Instructions



- **Opcode**

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

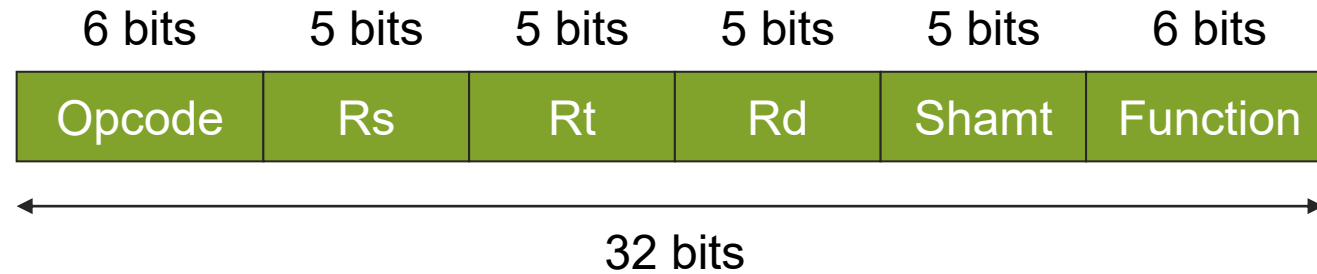
- **Example:**

- **add**
- **and**
- **or**
- **many more**

0	Rs	Rt	Rd	Shamt	0x20
0	Rs	Rt	Rd	Shamt	0x24
0	Rs	Rt	Rd	Shamt	0x25

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

# Machine Code of R-Type Instructions



- **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



# 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 following shift instructions?
- sll \$5, \$4, 3 # shift left logical the value of \$4 by 3 bits
- srl \$5, \$4, 2 # shift right logical the value of \$4 by 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 2N**

srl \$5, \$4, 2

Logical Right Shift by 2 bits:

0's shifted in...  
\$5 0 0 ... 0 0 1 1 = +3  
0x00000003

sll \$5, \$4, 3

Logical Left Shift by 3 bits:

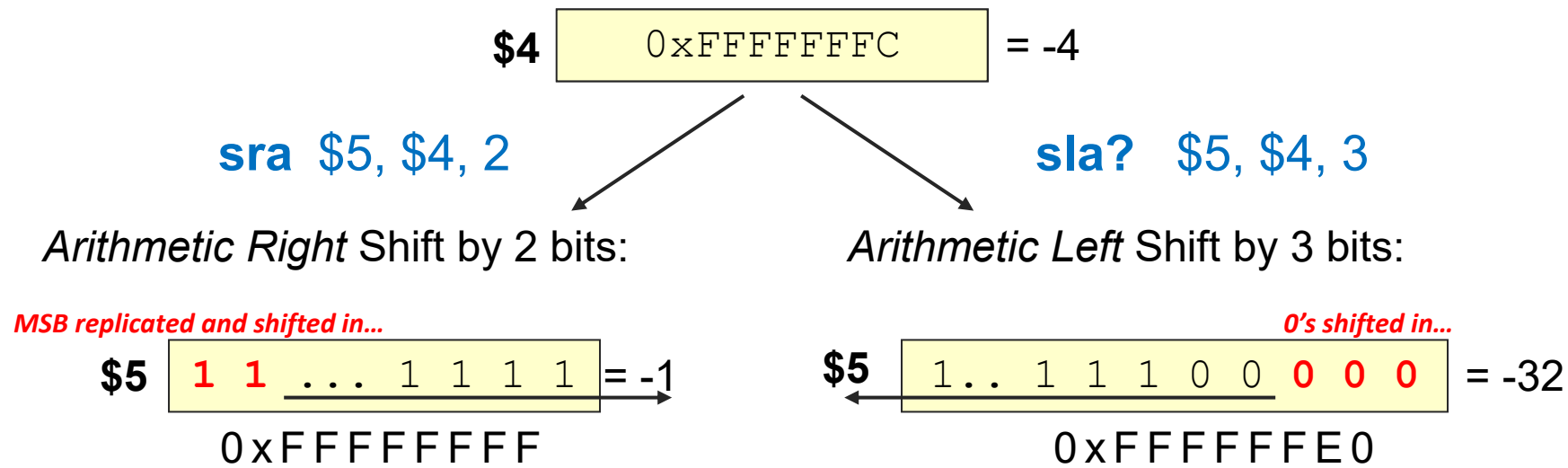
0's shifted in...  
\$5 ... 0 1 1 0 0 0 0 0 = +96  
0x00000060

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



# Example: Arithmetic Shift

- Shift right with replicating MSB
- Shift left with filling in 0s
- Example:
  - Assume that the original value of \$4 is 0xFFFFFFC (= -4)
  - What is the value of \$5 after executing the following shift instruction?
  - `sra $5, $4, 2` # shift right arithmetic the value of \$4 by 2 bits



# Example: Arithmetic Shift

- Shift right with replicating MSB
- Shift left with filling in 0s
- Example:

- Assume the initial value of \$4 is 0xFFFFF0C (= -4)
- After executing the following shift instruction
- # shift right arithmetic the value of \$4

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)

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

\$4 0xFFFFF0C = -4

sra \$5, \$4, 2

sla? \$5, \$4, 3

Arithmetic Right Shift by 2 bits:

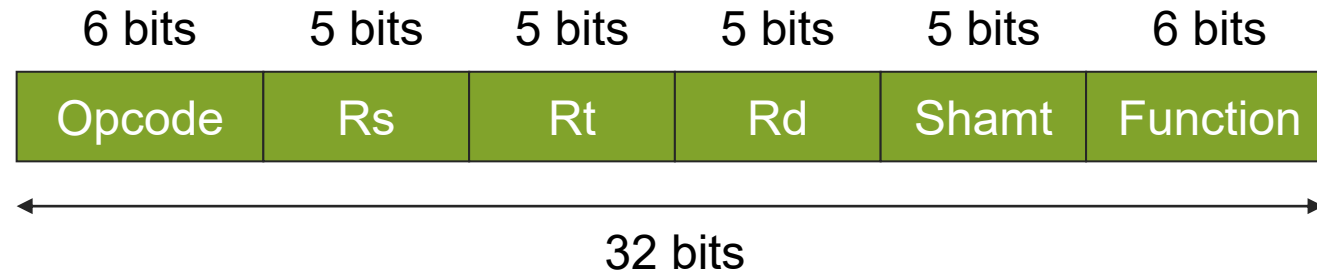
Arithmetic Left Shift by 3 bits:

MSB replicated and shifted in...

\$5 1 1 ... 1 1 1 1 = -1  
0xFFFFFFFF

0's shifted in...

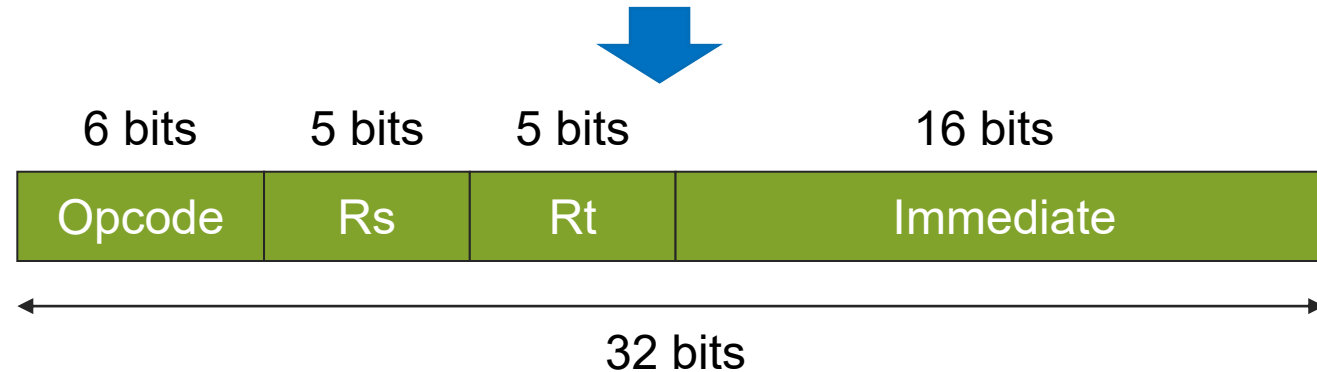
\$5 1 ... 1 1 1 0 0 0 0 0 = -32  
0xFFFFF0E0



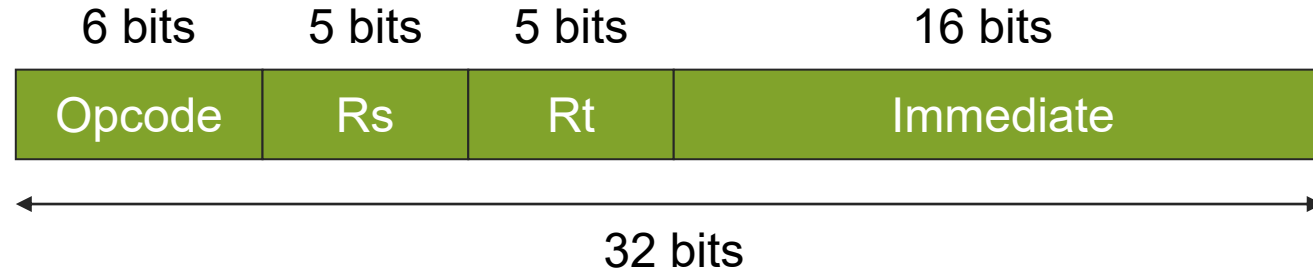
# Machine Code of I-Type Instructions

- **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**



# Machine Code of I-Type Instructions



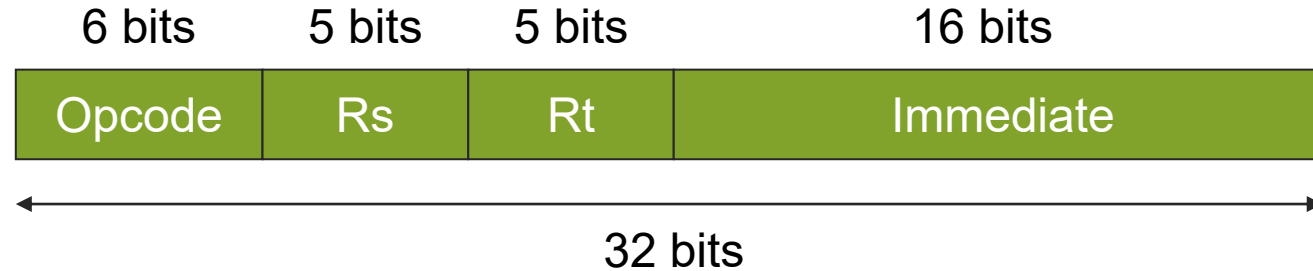
- **Opcode**

- Indicate Command/Operation of an instruction
- Example:

- **addi**
- **andi**
- **ori**
- **many more;**

001000	Rs	Rt	Immediate
001100	Rs	Rt	Immediate
001101	Rs	Rt	Immediate

# Machine Code of I-Type Instructions



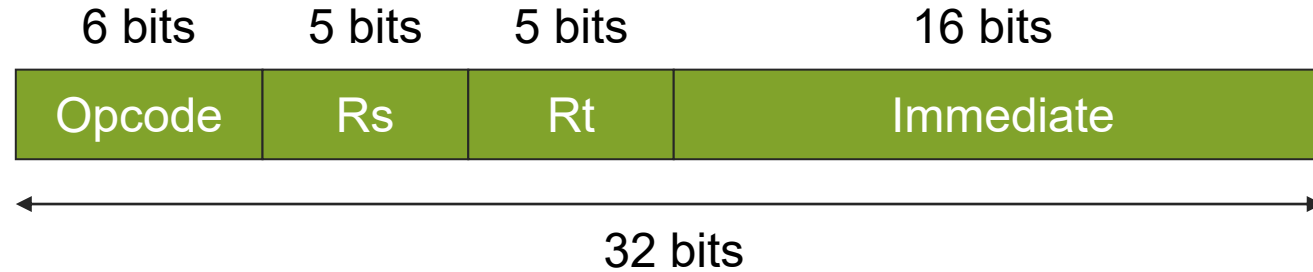
- **Rs, Rt**

- 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



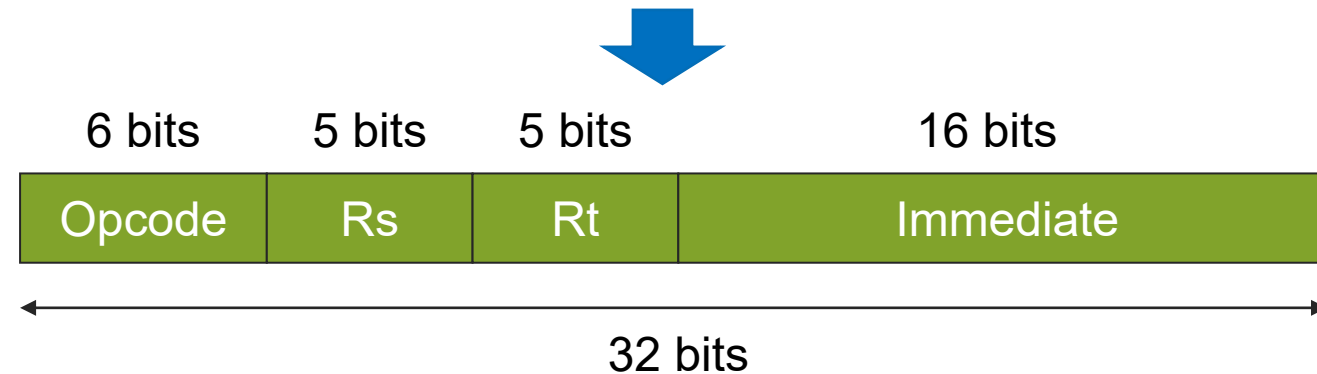
- **Immediate**

- Indicate the immediate value

- Example:

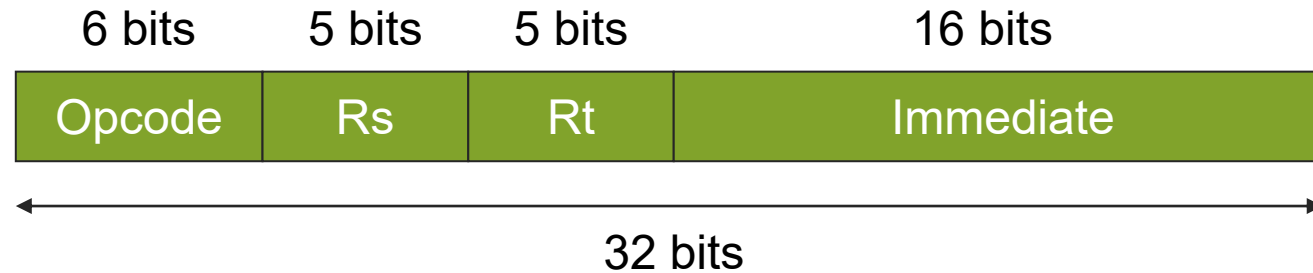
- **addi**    \$5, \$4, 1
- **andi**    \$13, \$12, 0xA
- **ori**     \$t3, \$t2, 12

001000	00100	00101	1
001100	01100	01101	0xA
001101	01010	01011	12





# Machine Code of I-Type Instructions



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

- Example:

- lb     \$5, 4(\$4)
    - lh     \$13, 0x10(\$12)
    - lw     \$t3, -16(\$t2)
    - sb     \$3, 0xffff0(\$4)
    - sh     \$2, 8(\$8)
    - sw     \$s1, 10(\$s0)

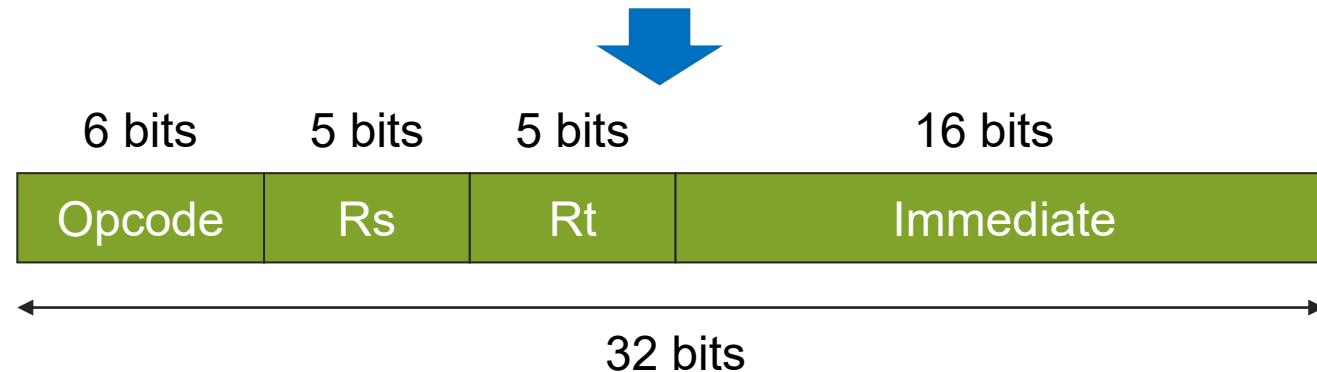
0x20	00100	00101	4
0x21	01100	01101	0x10
0x23	01010	01011	-16
0x28	00100	00011	0xffff0
0x29	01000	00010	8
0x2B	10000	10001	10

# 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



# 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 above or below”*; not branch to absolute address
- necessary because in many cases, we do not know (at compile time) where our code will be loaded in the memory space

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	<b>bne \$t0, \$s5, Exit</b>
80010	addi \$s3, \$s3, 1
80014	<b>b Loop</b>
80018	Exit: ...

Each instruction is 4-byte long  
→ Assembly code line address increases by 4 bytes

# 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 above or below”*; not branch to absolute address

- necessary because in many cases, we do not know (at compile time) where our code will be loaded in the memory space

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	<b>bne \$t0, \$s5, Exit</b>
80010	addi \$s3, \$s3, 1
80014	<b>b Loop</b>
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**

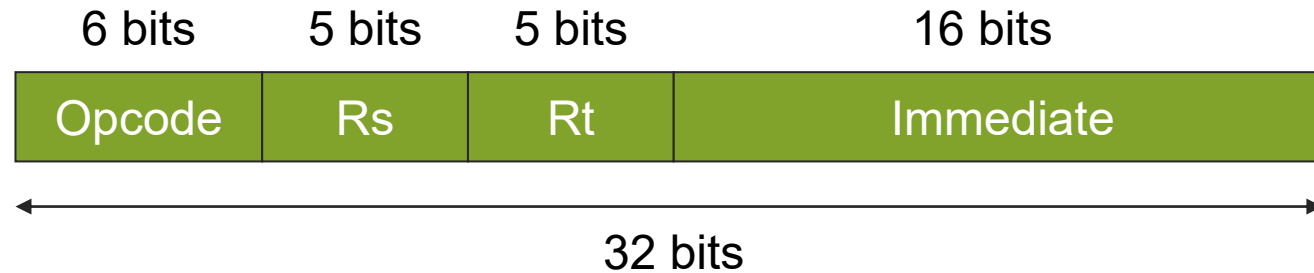
Address increases by 4 bytes

# 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	<b>bne \$t0, \$s5, Exit</b>
80010	addi \$s3, \$s3, 1
80014	<b>b Loop</b>
80018	Exit: ...

# Machine Code of I-Type Instructions



```
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
- **beq**    \$0, \$0, Loop

0x5	01000	10101	2
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`                       $\# R[2] = 0 | 1 = 1$

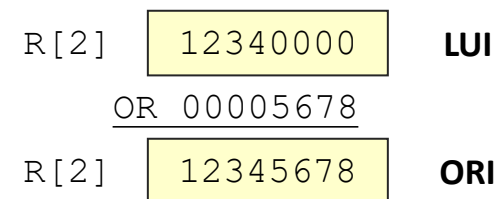
**LUI**: the immediate value is loaded to the MSB 16 bits of the target register

- If immediate is more than 16 bits

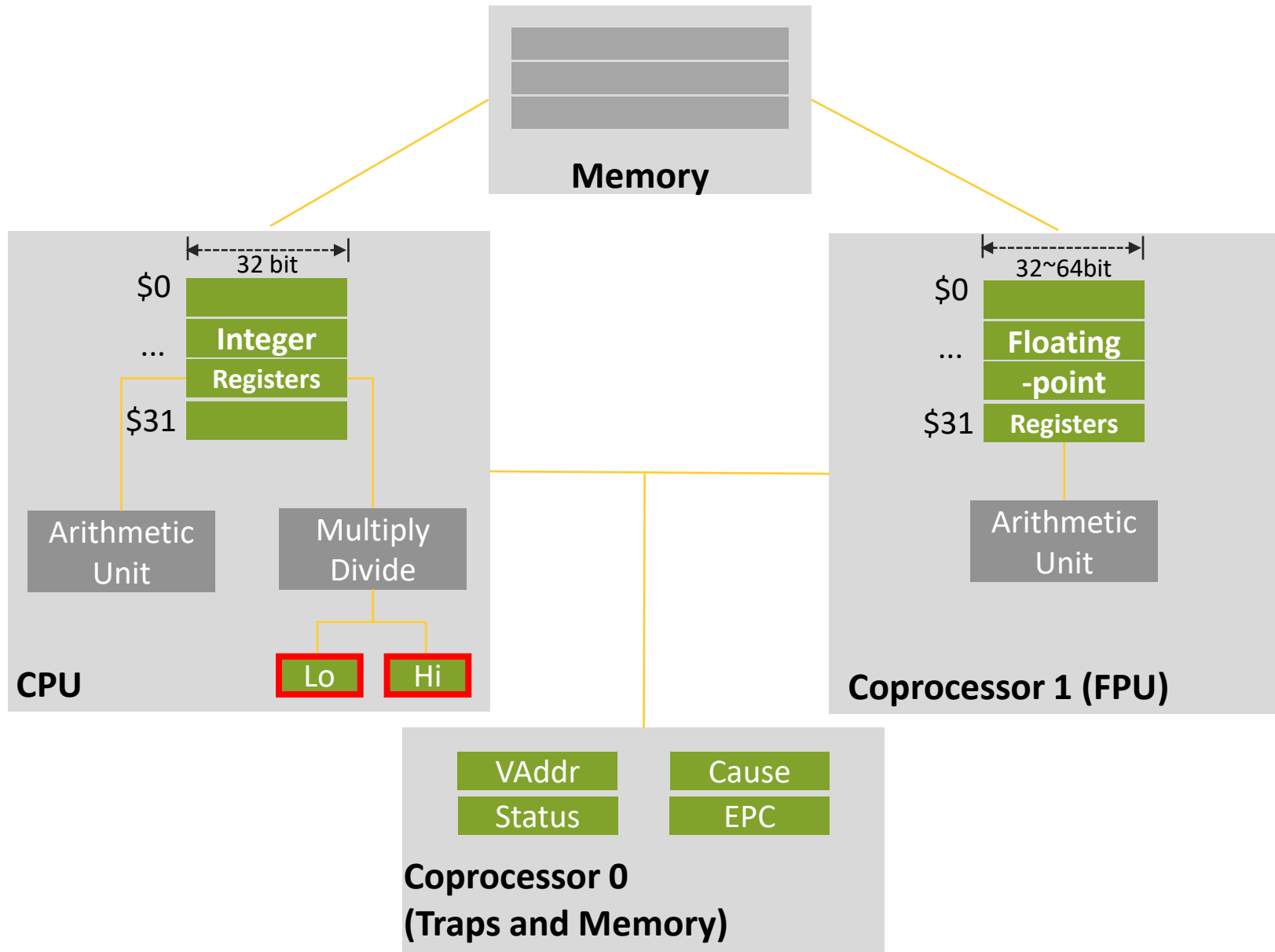
- immediates limited to 16 bits so we must load constant with a 2-instruction sequence using the special **LUI (Load Upper Immediate)** instruction

- To load \$2 with 0x12345678

- `lui $2, 0x1234`
- `ori $2, $2, 0x5678`



# Mult & Div Instructions





# 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

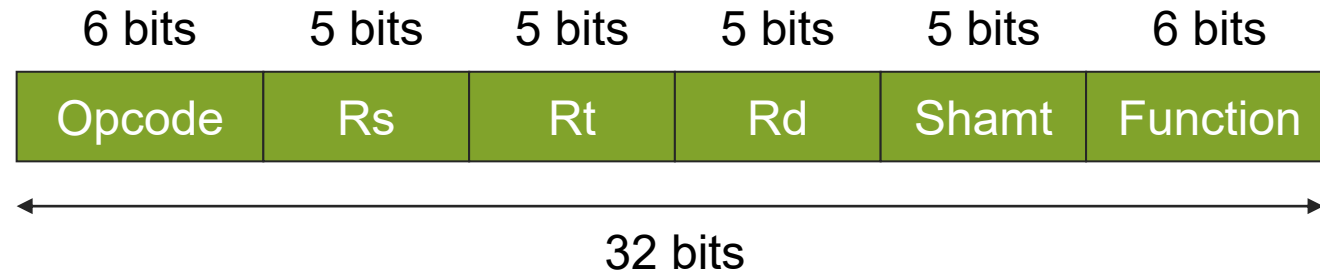
- **Division**

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

- **Moves the contents of lo/hi registers to GPR (General Purpose Register)**

- **mflo**      \$2      # \$2 = lo
  - **mfhi**      \$3      # \$3 = hi

# Machine Code of Mult/Div/Mflo/Mfhi



- **Mult/Div/Mflo/Mfhi**

- Example:

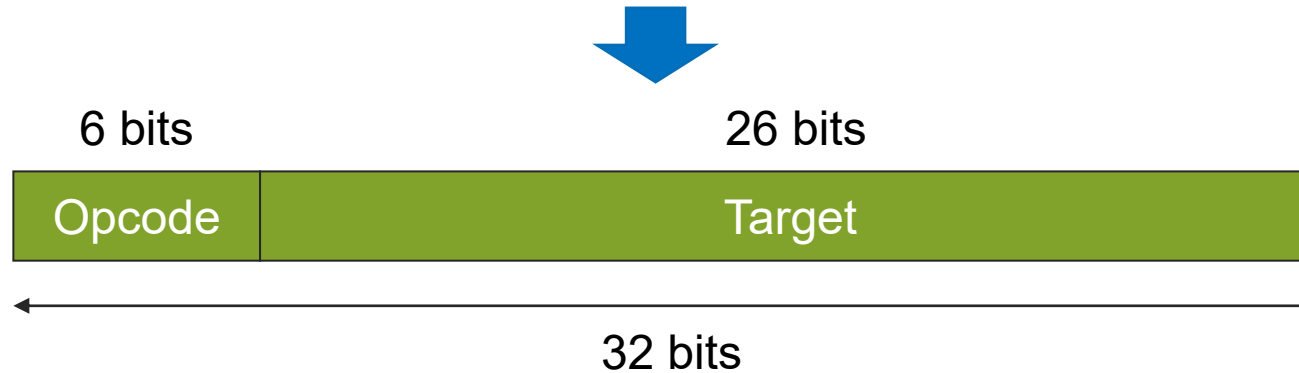
- mult    \$5, \$4
    - div     \$5, \$4
    - mflo    \$5
    - mfhi    \$4

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

# J-Type Instruction

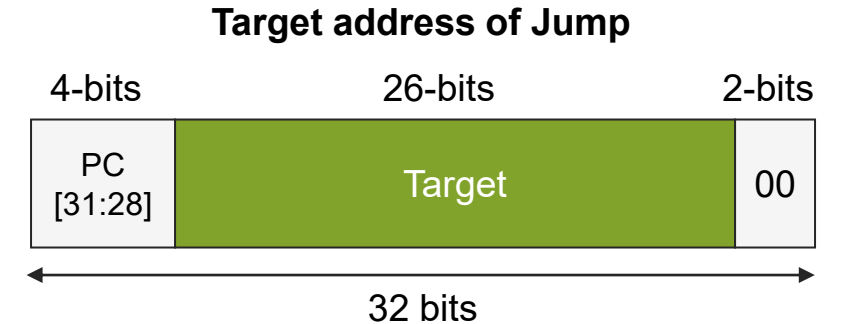
- There is another type, J, in MIPS for Jump instruction
  - Similar to unconditional branch

**j target # Jump to target**



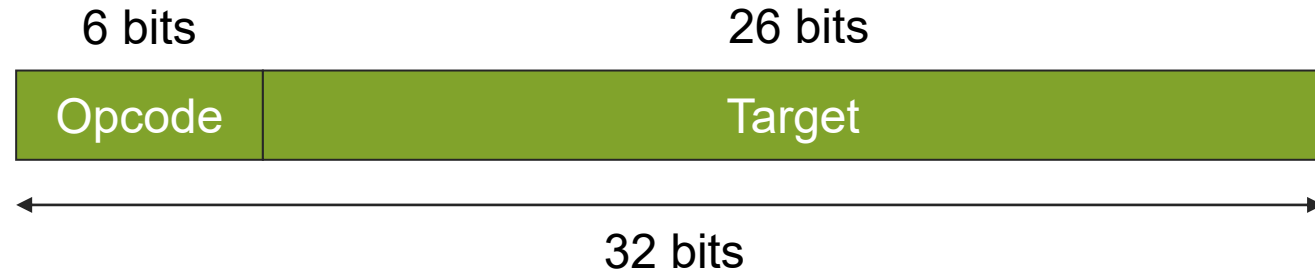
# Jump Target Addressing

- 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 field = 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	<b>j Loop</b>
00080018	Exit: ...

# Machine Code of J-Type Instructions



- **Jump in J-Type machine code format**

- Example:

- j      Loop



# 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

**slti** Rt, Rs, imm

# if (Rs < imm) Rt = 1; else Rt = 0;

- 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;
}
```

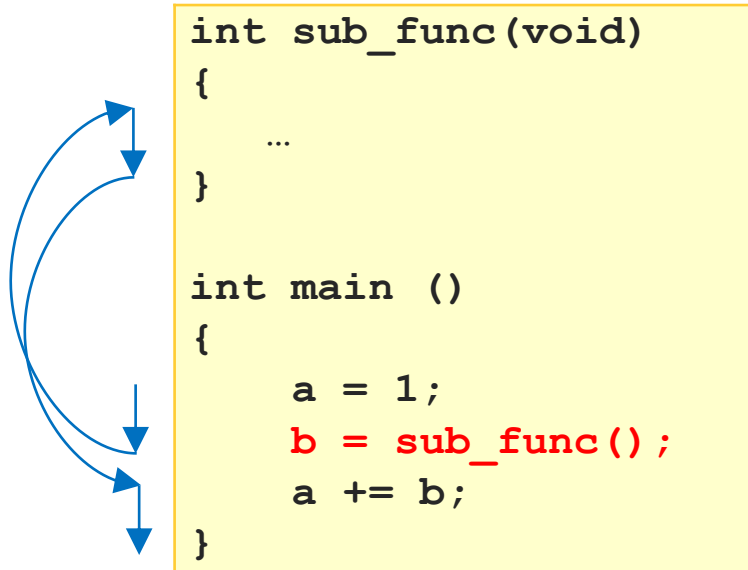
MIPS

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

```
for:    add    $s0, $0, $0
        add    $s1, $0, $0
        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?



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

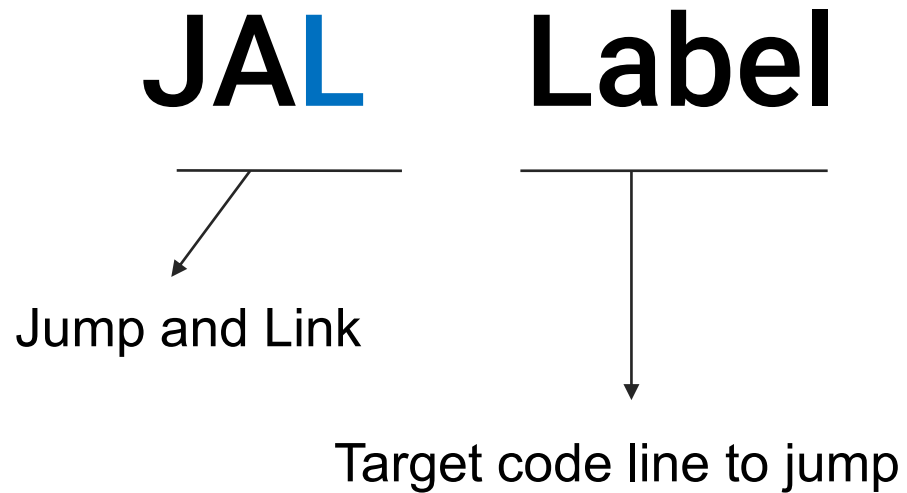
# Special Registers

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

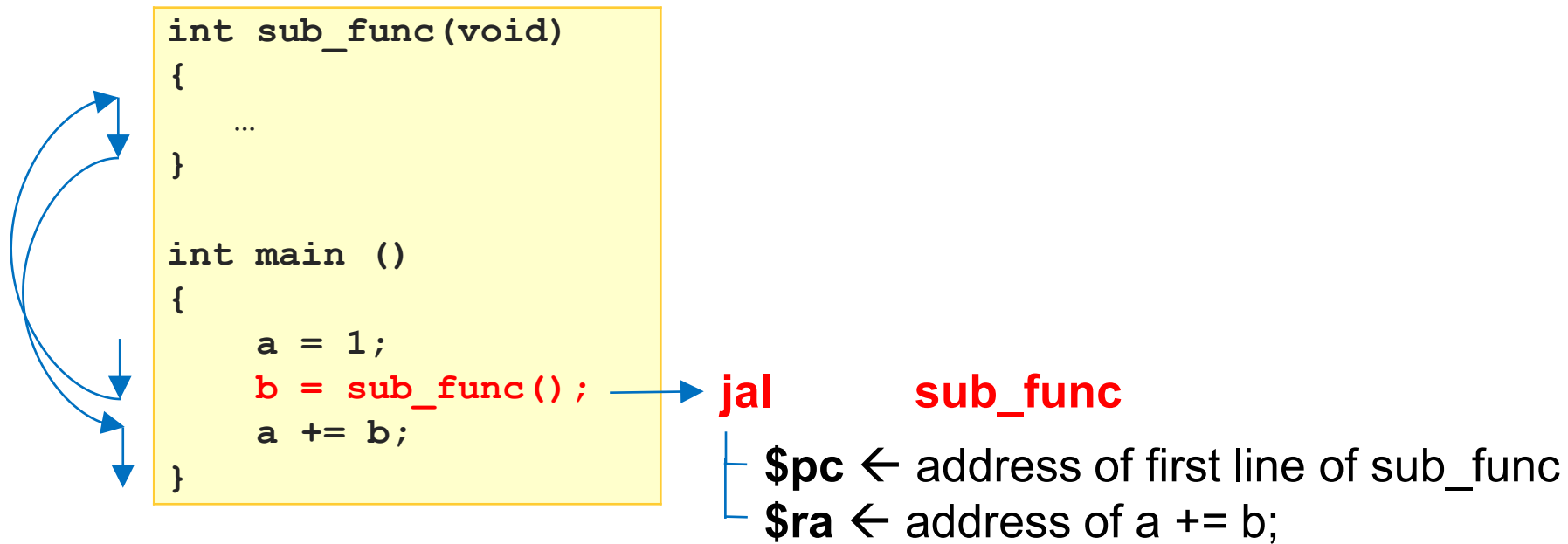
- Most of assembly languages provide a special jump (or branch) instruction that **Jump + Update Return Address Register**



1. Jump to Label and
2. Update \$ra with return address  
(JAL's next instruction address)

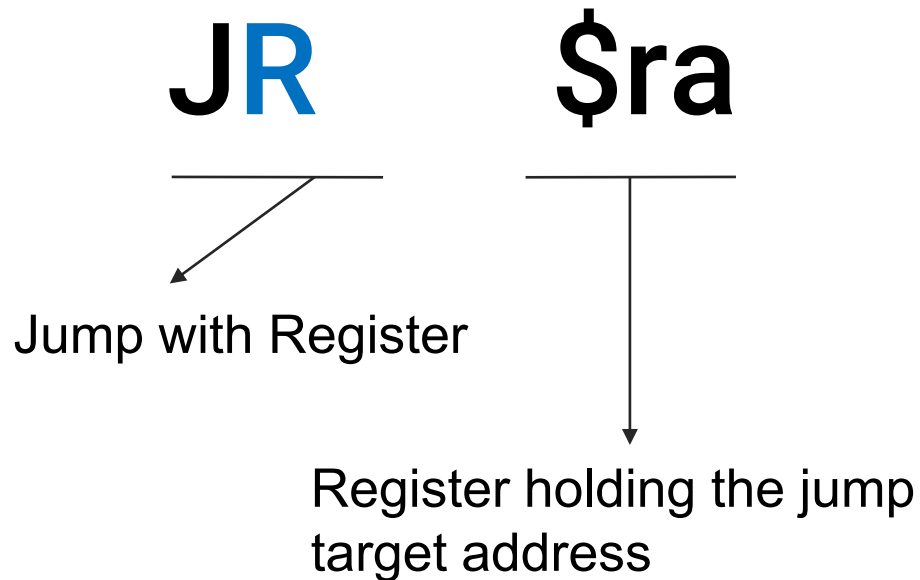
# Calling a Subroutine

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



# Jump with Register

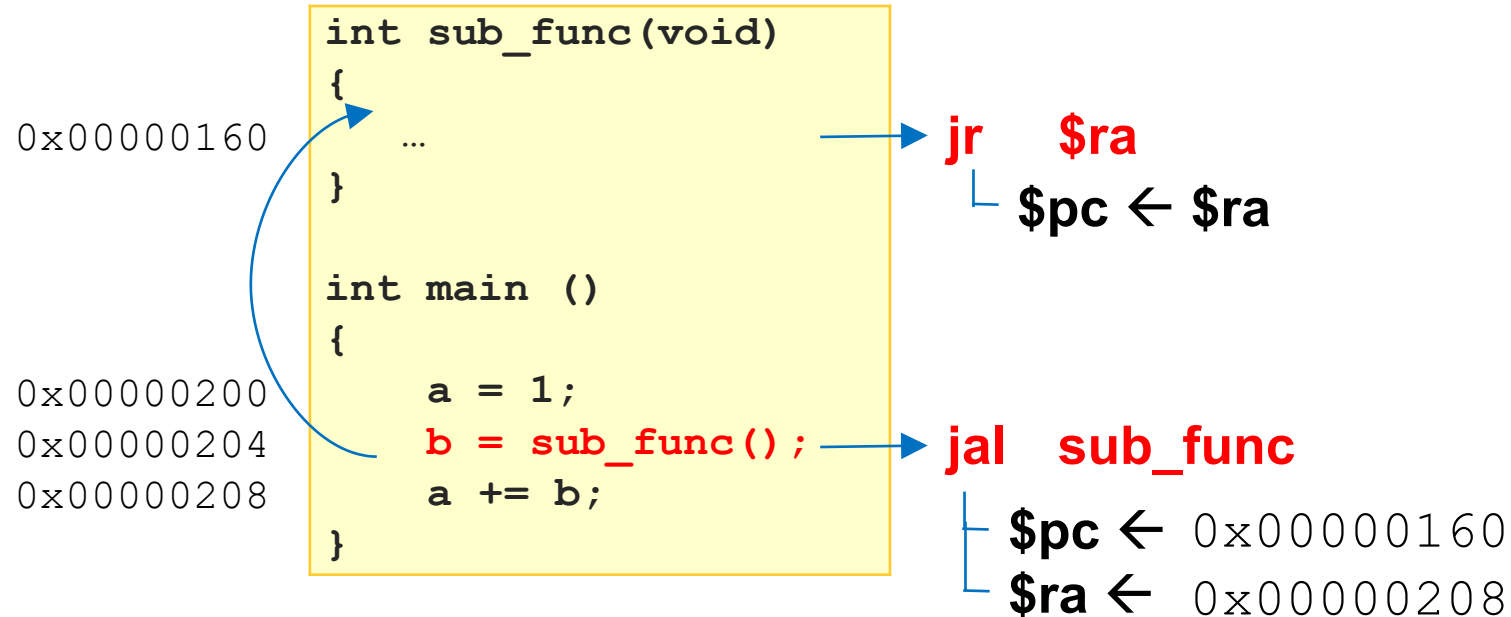
- We can return to the Caller by running Jump with Register instruction with \$ra as an operand



# Jump to address in \$ra  
# (\$pc = \$ra)

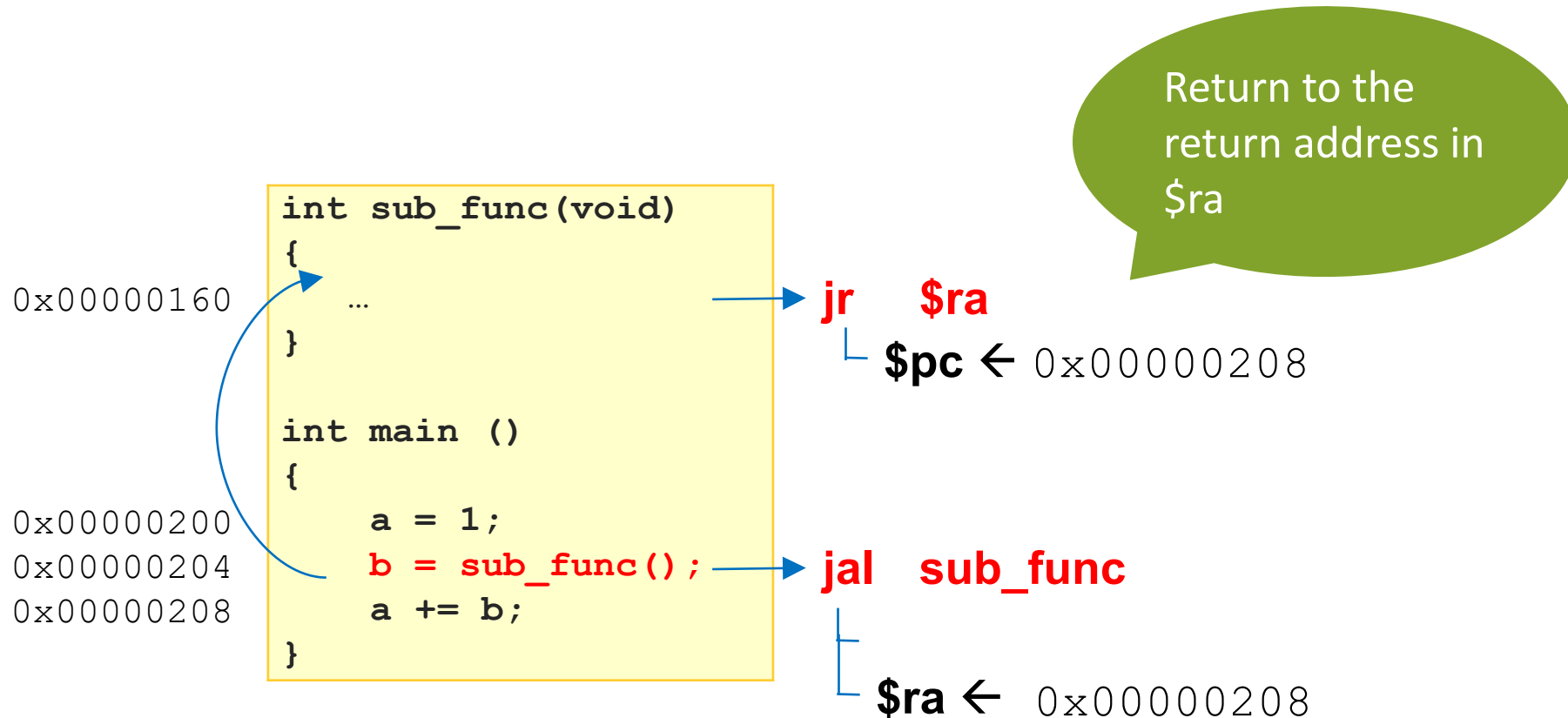
# Jump with Register

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



# Jump with Register

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





# Exercise

- Translate the given high-level language code to MIPS assembly.
  - Assume that the address of integer variable x and y are in \$4 and \$1 respectively.

High-level language

```
if (x > y)
    x = x + y;
else
    x = 1;
```

Tasks to do:

- 1) Load x value to a register
- 2) Load y value to another register
- 3) Check if y is less than x
- 4) If true, run add operation
- 5) Else, store 1 to one register
- 6) Store the result value to x in memory

# Exercise

- Translate the given high-level language code to MIPS assembly.
  - Assume that the address of integer variable x and y are in \$4 and \$1 respectively.

## High-level language

```
if (x > y)
    x = x + y;
else
    x = 1;
```

If false, you should not execute add  
→ Create a label for else case and branch

### Tasks to do:

- 1) Load x value to a register
- 2) Load y value to another register
- 3) Check if y is less than x
- 4) If true, run add operation
- 5) Else, store 1 to one register
- 6) Store the result value to x in memory

## MIPS

	lw	\$2,	0(\$4)	
	lw	\$3,	0(\$1)	
	slt	\$1,	\$3,	\$2
	beq	\$1,	\$0,	Else
	add	\$2,	\$2,	\$3
Else:	addi	\$2,	\$0,	1

# Exercise

- Translate the given high-level language code to MIPS assembly.
  - Assume that the address of integer variable x and y are in \$4 and \$1 respectively.

## High-level language

```
if (x > y)
    x = x + y;
else
    x = 1;
```

### Tasks to

- 1) Load x into register \$2
- 2) Load y into register \$3
- 3) Check if x > y
- 4) If true, calculate x + y
- 5) Else, store 1 into register \$2
- 6) Store the result value to x in memory

Wait, when  $x > y$ , the calculation result should be also stored to the memory without executing the else code  
→ Add unconditional branch and skip addi

## MIPS

	lw	\$2,	0(\$4)	
	lw	\$3,	0(\$1)	
	slt	\$1,	\$3,	\$2
	beq	\$1,	\$0,	Else
	add	\$2,	\$2,	\$3
	b		Next	
Else:	addi	\$2,	\$0,	1
Next:	sw	\$2,	0(\$4)	

# 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 5<sup>th</sup> 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

# Example

## High-level language

```
int t;

int main(void) {
    int y;
    ...
    y = diffofsums(2, 3, 4, 5);
    ...
}

int diffofsums (int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

## MIPS Assembly

```
main:
    ...
                                # arg 0 = 2
                                # arg 1 = 3
                                # arg 2 = 4
                                # arg 3 = 5
                                # call subroutine
                                # y = returned value
    ...

diffofsums:
                                # $t0 = f + g
                                # $t1 = h + i
    sub    $s0, $t0, $t1        # result = (f + g) - (h + i)
                                # put return value in $v0
                                # return to caller
```

# Example (Addressing)

- Question: What values the following registers will have in each condition?

- After jal

- \$pc
- \$ra

- After jr

- \$pc

Address

0x0000015C

0x00000160

0x00000168

.

.

.

0x0000016E

MIPS Assembly

main:

```
...
addi $a0, $0, 2    # arg 0 = 2
addi $a1, $0, 3    # arg 1 = 3
addi $a2, $0, 4    # arg 2 = 4
addi $a3, $0, 5    # arg 3 = 5
jal  diffofsums    # call subroutine
add  $s0, $v0, $0  # y = returned value
...
```

diffofsums:

```
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
jr   $ra              # return to caller
```

# Register Value Overwriting

- Register file is a shared resource
- Example:
  - In the previous code, \$t0, \$t1, and \$s0 are updated by Callee
  - Assume that main function wanted to compare the diffofsums return value with value 10 and \$t0 has value 10 before calling subroutine
  - After calling the subroutine, \$t0 is updated with intermediate compute result → incorrect comparison

main:

```
...  
addi $t0, $0, 10 # main wanted to use $t0  
addi $a0, $0, 2  # arg 0 = 2  
addi $a1, $0, 3  # arg 1 = 3  
addi $a2, $0, 4  # arg 2 = 4  
addi $a3, $0, 5  # arg 3 = 5  
jal  diffofsums  # call subroutine  
add  $s0, $v0, $0 # y = returned value  
slt  $t1, $s0, $t0 # to compare with comp result  
...
```

diffofsums:

```
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  
jr   $ra           # return to caller
```


After returning from diffofsums, main function will see the values of \$t0, \$t1, and \$s0 are updated unexpectedly



# Register Value Overwriting


- Use Stack memory to protect register values
  - Callee backs up the values of registers to stack memory that will be updated in its function body
- Callee restores the values from stack to original registers before returning to Caller

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

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 + l
sub    $s0, $t0, $t1    # result = (f + g) - (h + i)
add    $v0, $s0, $0     # put return value in $v0

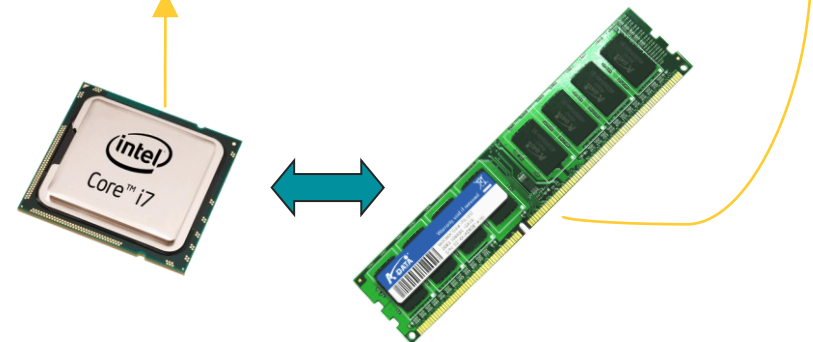
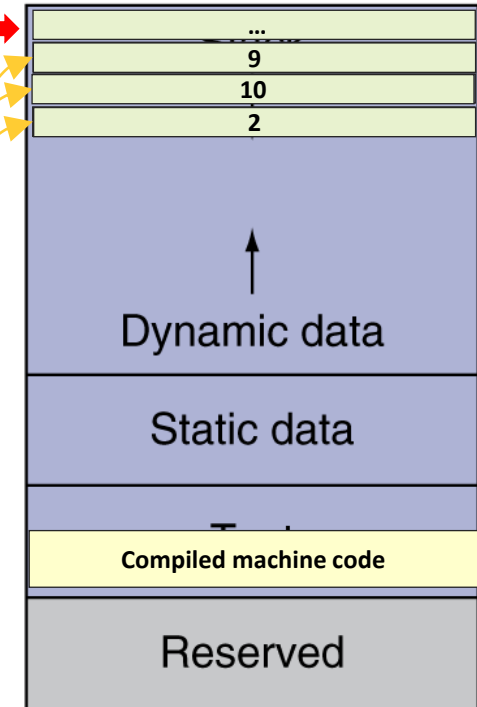
lw    $t1, 0($sp)      # restore $t1 from stack
lw    $t0, 4($sp)      # restore $t0 from stack
lw    $s0, 8($sp)      # restore $s0 from stack
addi   $sp, $sp, 12     # deallocate stack space

jr     $ra              # return to caller
```

Stack region  
allocation for  
diffofsums

\$sp →

Register File



# Register Value Overwriting

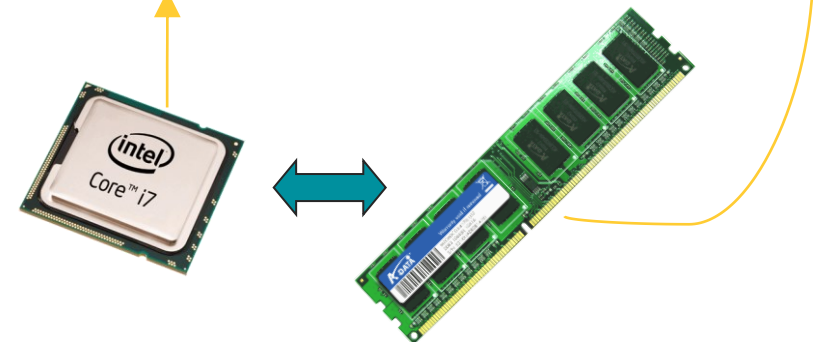
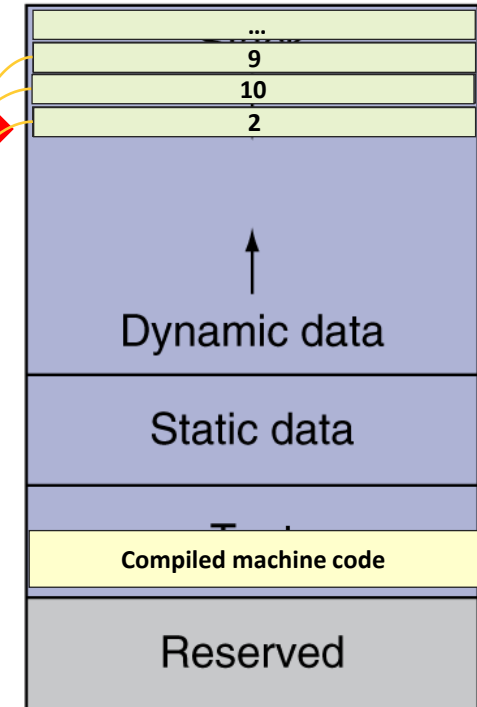
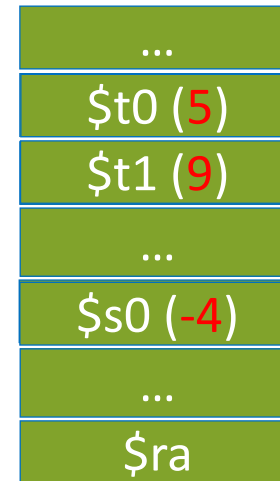
Before returning to Caller,  
Register values are revoked

diffofsums:

<b>addi</b>	<b>\$sp, \$sp, -12</b>	# make space on stack # to store 3 registers
<b>sw</b>	<b>\$s0, 8(\$sp)</b>	# save \$s0 on stack
<b>sw</b>	<b>\$t0, 4(\$sp)</b>	# save \$t0 on stack
<b>sw</b>	<b>\$t1, 0(\$sp)</b>	# save \$t1 on stack
<b>add</b>	<b>\$t0, \$a0, \$a1</b>	# \$t0 = f + g
<b>add</b>	<b>\$t1, \$a2, \$a3</b>	# \$t1 = h + i
<b>sub</b>	<b>\$s0, \$t0, \$t1</b>	# result = (f + g) - (h + i)
<b>add</b>	<b>\$v0, \$s0, \$0</b>	# put return value in \$v0
<b>lw</b>	<b>\$t1, 0(\$sp)</b>	# restore \$t1 from stack
<b>lw</b>	<b>\$t0, 4(\$sp)</b>	# restore \$t0 from stack
<b>lw</b>	<b>\$s0, 8(\$sp)</b>	# restore \$s0 from stack
<b>addi</b>	<b>\$sp, \$sp, 12</b>	# deallocate stack space
<b>jr</b>	<b>\$ra</b>	# return to caller

Stack region  
allocation for  
diffofsums

Register File



# 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

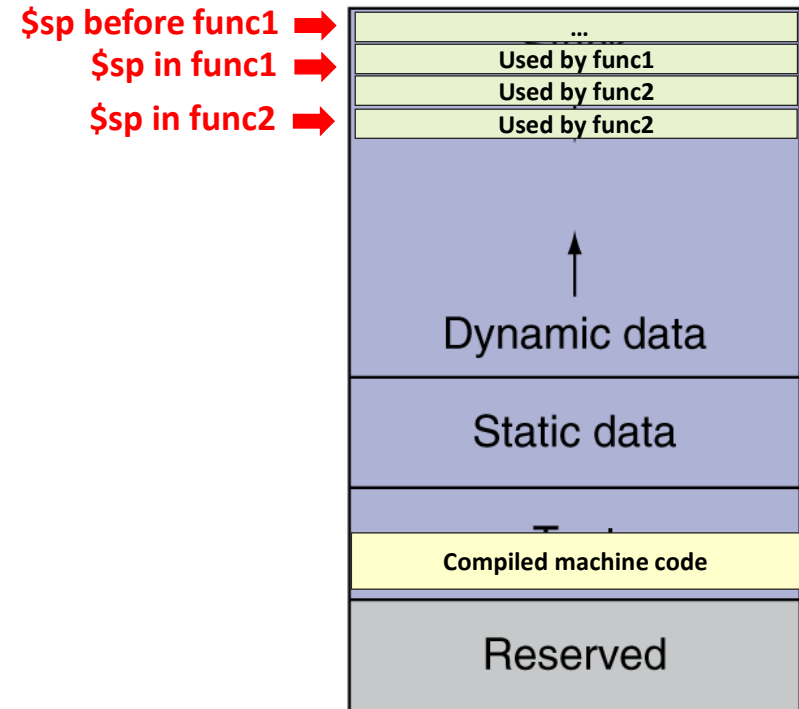
    jal     func2            # jump to func2
    ...

    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.

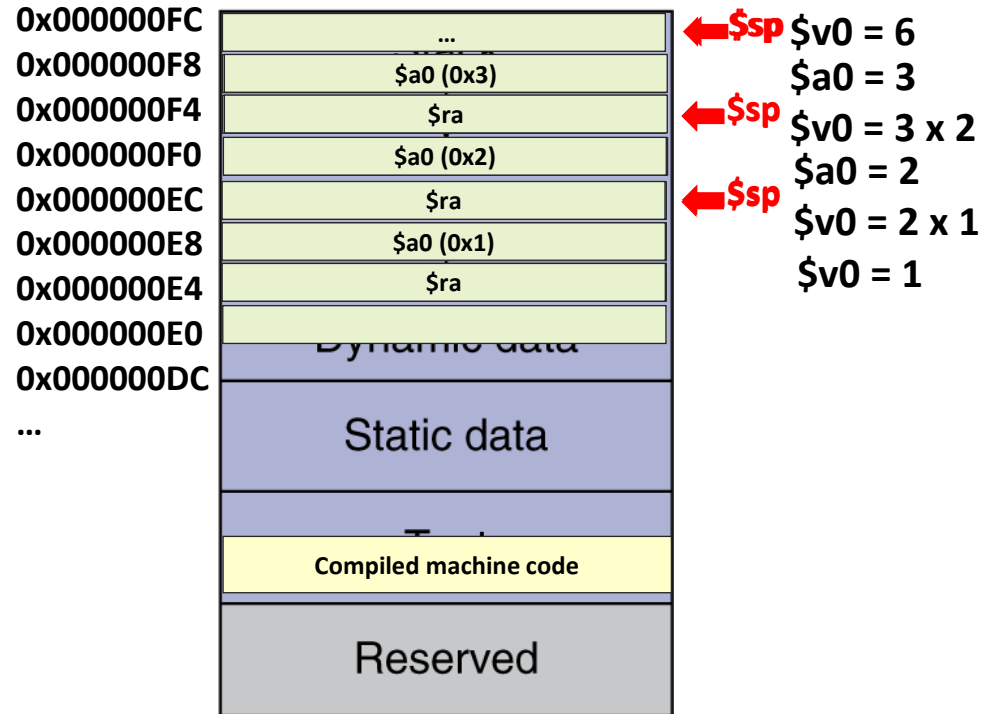
## High-level language

```
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
    
```

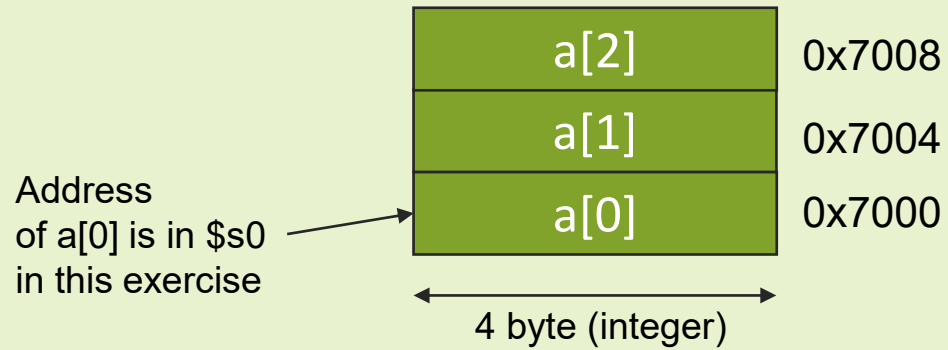
# Exercise 1

- Translate the given high-level language code to MIPS assembly
  - Assume that the base address of an *integer* array *a* is in register \$s0
    - base address of an array: the address of the first element of the array

High-level language     `a[1]++;`

## Memory allocation for Arrays

If you defined `int a[3];` and the system allocated a chunk of memory for this 3-element array from 0x7000, the address of elements would be like below



## MIPS

```
lw    $t3, 4($s0)
addi  $t3, $t3, 1
sw    $t3, 4($s0)
```

# Exercise 2

- Translate the given high-level language code to MIPS assembly
  - Assume that the base address of a *char* array *a* is in register \$s0
    - base address of an array: the address of the first element of the array
  - Assume that the value of *i* is in \$t0.

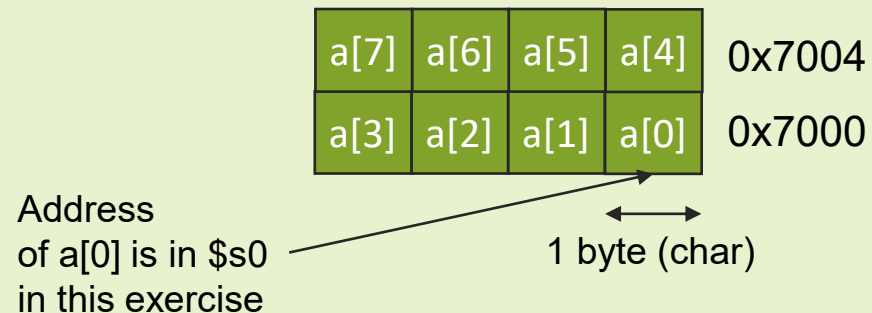
High-level language

```
a[i]--;
```

MIPS

## Memory allocation for Arrays

If you defined `char a[8]`; and the system allocated a chunk of memory for this 8-element array from 0x7000, the address of elements would be like below





# Exercise 2

- Translate the given high-level language code to MIPS assembly
  - Assume that the base address of a *char* array *a* is in register *\$s0*
    - base address of an array: the address of the first element of the array
  - Assume that the value of *i* is in *\$t0*.

High-level language

`a[i]--;`

MIPS

```
add $t2, $s0, $t0
lb   $t3, 0($t2)
subi $t3, $t3, 1
sb   $t3, 0($t2)
```

Now we know that the address of *a[i]* is  $\$s0 + 1\text{byte} * i = \$s0 + \$t0$

Is this a valid operation to get one byte from  $\$s0 + \$t0$ ?

`lb $t1, $t0($s0)`

No, because offset should be an immediate value, not a register id

→ We should change the base address value, not offset value  
i.e. new base:  $\$t2 = \$s0 + \$t0$   
and then load one byte from `0($t2)`