CMPE 200

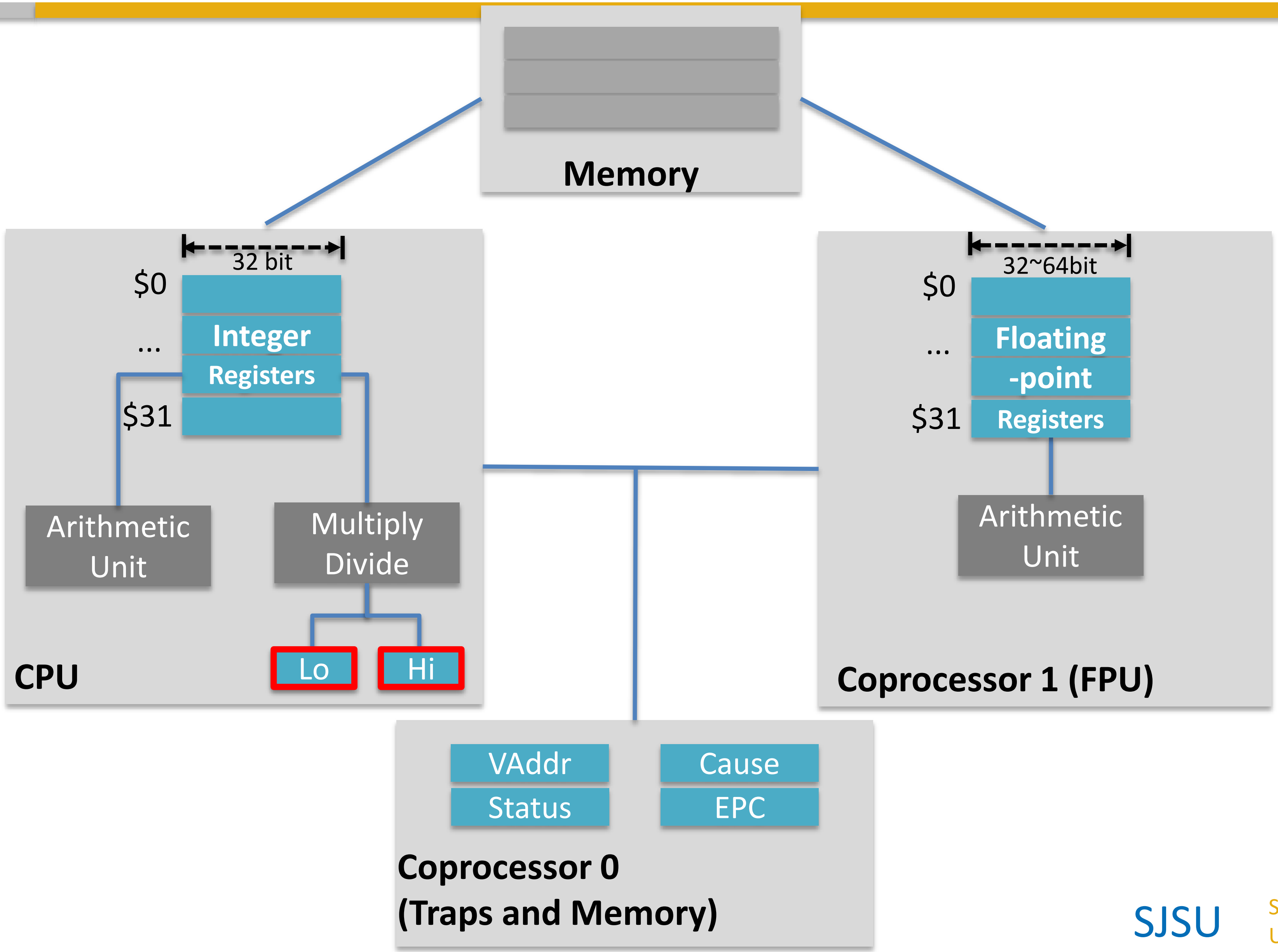
Computer Architecture & Design

Lecture 2.
Processor Instruction Set Architecture & Language (4)

Haonan Wang

SJSU SAN JOSÉ STATE UNIVERSITY

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**
  - **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

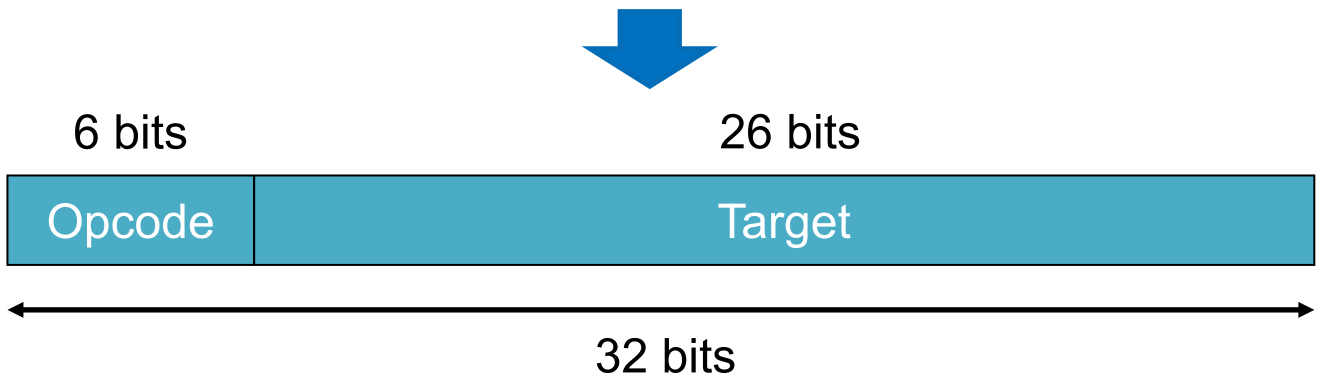
- **Mult/Div/Mflo/Mfhi**
  - Example:
    - **mult $5, $4**
    - **div $5, $4**
    - **mflo $5**
    - **mfhi $4**

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

J-Type Instruction

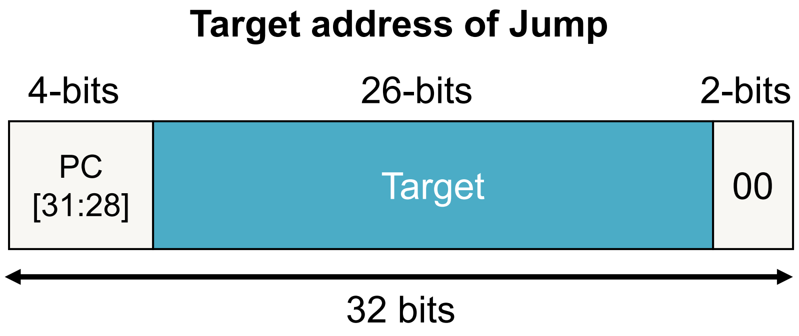
- **There is another type, J, in MIPS for Jump instruction**
  - Similar to unconditional branch

j   target   # Jump to target



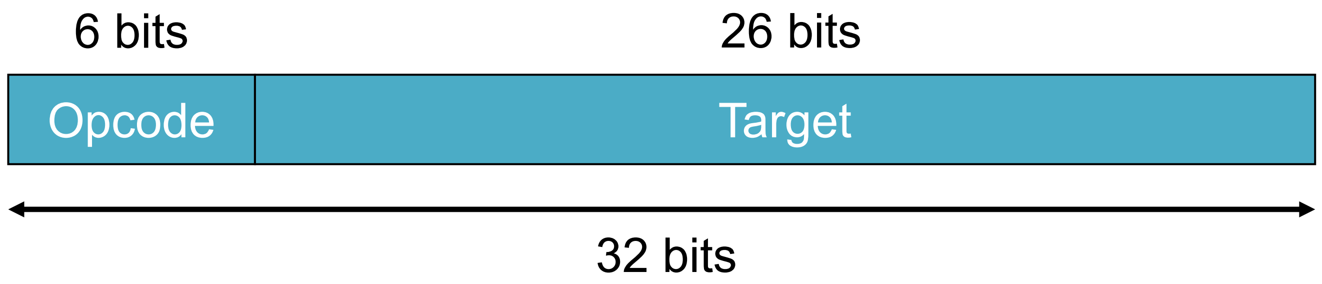
Jump Target Addressing

- **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          j    Loop
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
while:  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
for:    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 to address in $ra

; ($pc = $ra)

Jump with Register

Register holding the jump target address

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        …                      → jr  $ra
              }                               └ $pc ← 0x00000208

              int main ()
              {
0x00000200        a = 1;
0x00000204        b = sub_func();        → jal sub_func
0x00000208        a += b;                     └ $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 5^{th} 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 1

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 1 (Addressing)

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

- **After jal**
  - $pc :
  - $ra :

- **After jr**
  - $pc :

MIPS Assembly

Address				
	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
0x0000015C		**jal**	**diffofsums**	# call subroutine
0x00000160		add	$s0, **$v0**, $0	# y = returned value
		…		
	diffofsums:			
0x00000168		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 $v0
0x0000016E		**jr**	**$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

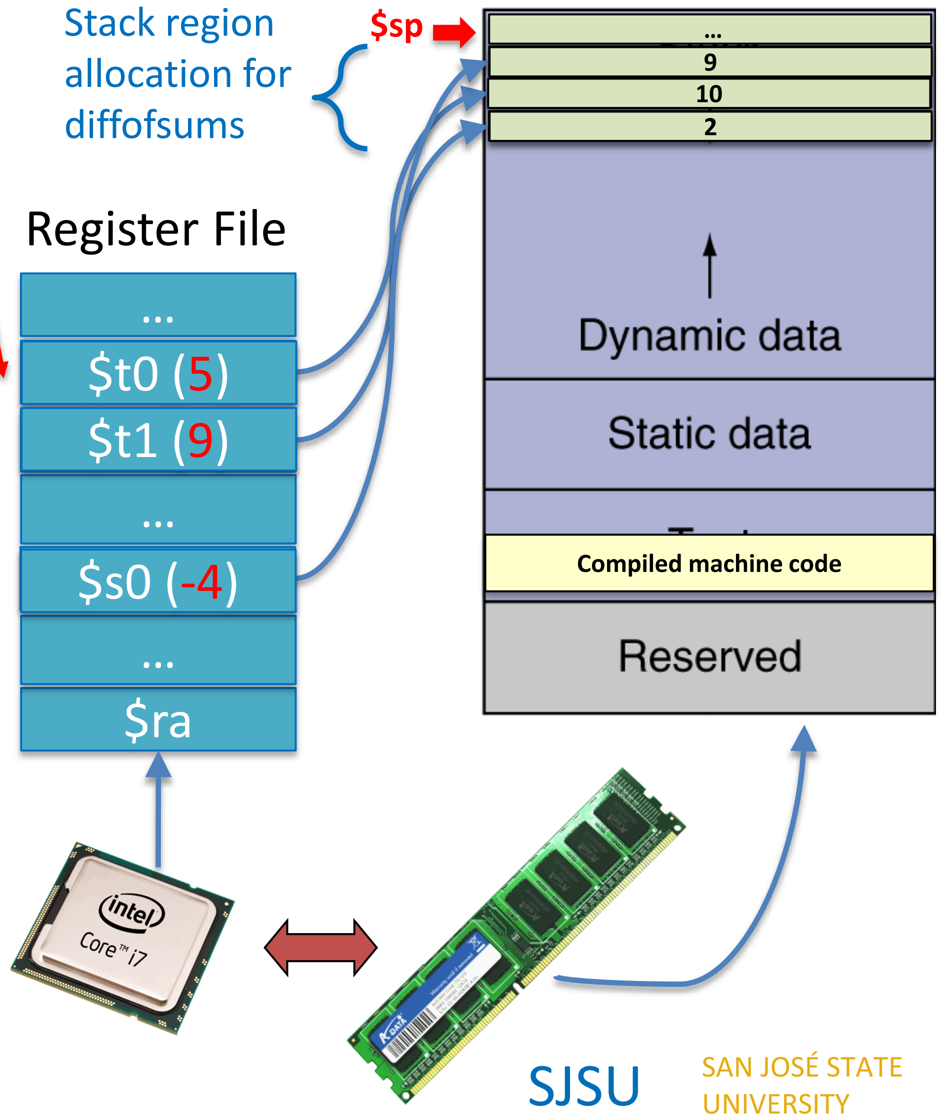
- **Use Stack memory to protect register values**
  - Callee backs up the values of registers to stack memory that will be updated in its function body
  - Callee restores the values from stack to original registers before returning to Caller

```
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

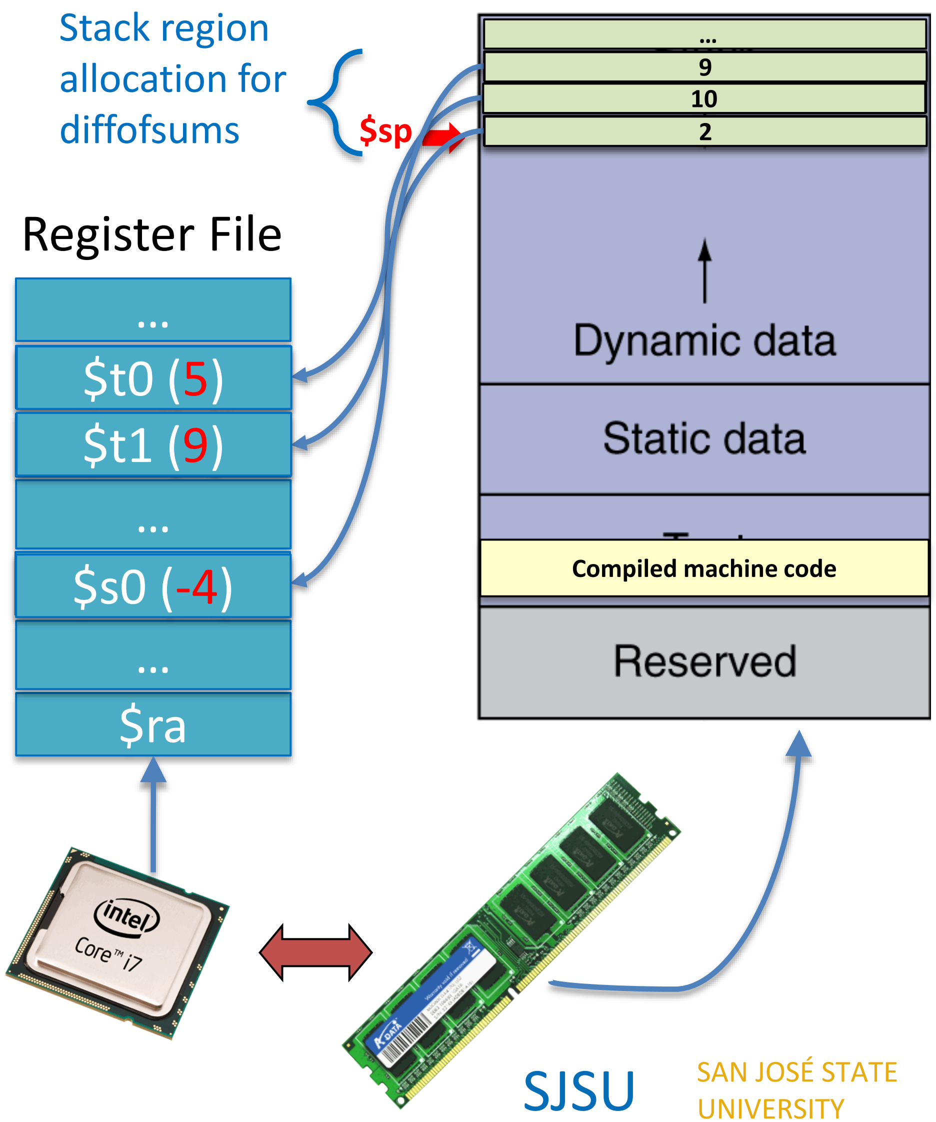
```
diffofsums:
        addi  $sp, $sp, -12      # make space on stack
                                 # to store 3 registers
        sw    $s0, 8($sp)        # save $s0 on stack
        sw    $t0, 4($sp)        # save $t0 on stack
        sw    $t1, 0($sp)        # save $t1 on stack

        add   $t0, $a0, $a1      # $t0 = f + g
        add   $t1, $a2, $a3      # $t1 = h + 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

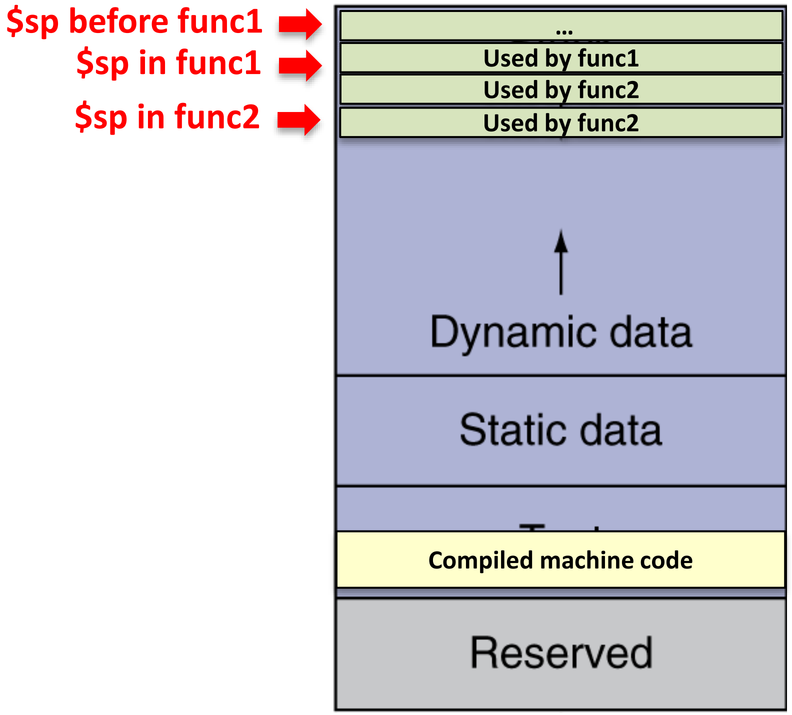
```
diffofsums:
        addi   $sp, $sp, -12         # make space on stack
                                     # to store 3 registers
        sw     $s0, 8($sp)           # save $s0 on stack
        sw     $t0, 4($sp)           # save $t0 on stack
        sw     $t1, 0($sp)           # save $t1 on stack

        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
```



SJSU SAN JOSÉ STATE UNIVERSITY

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.**

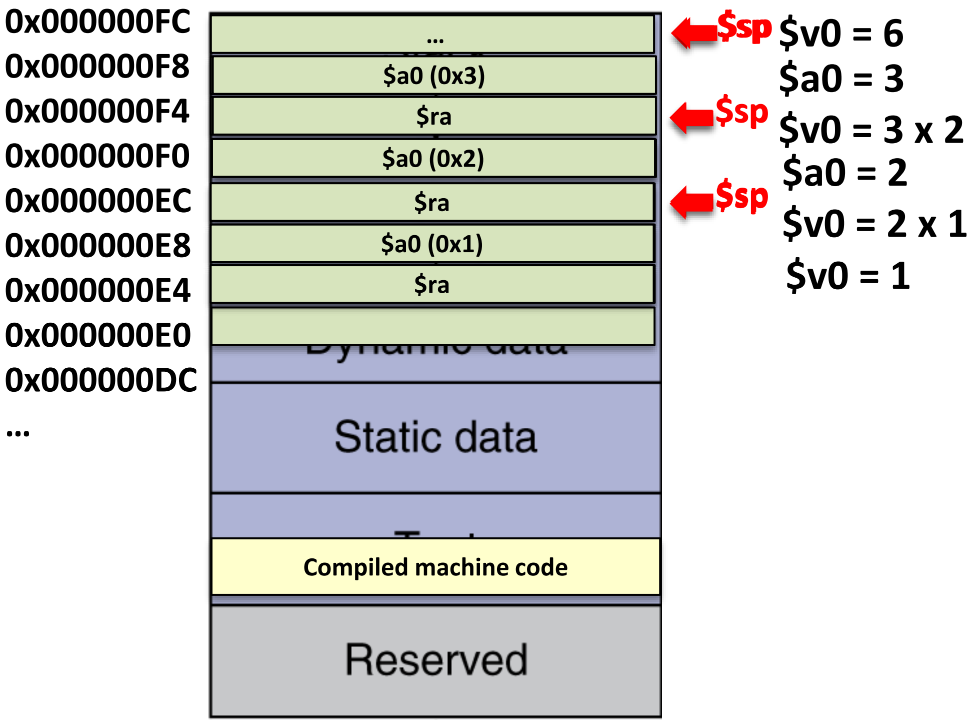
High-level language

```
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

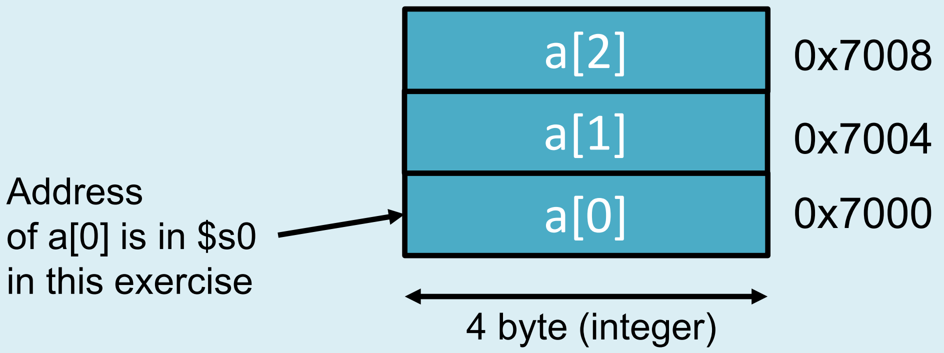
```
a[1]++;
```

MIPS

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

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



Exercise 2

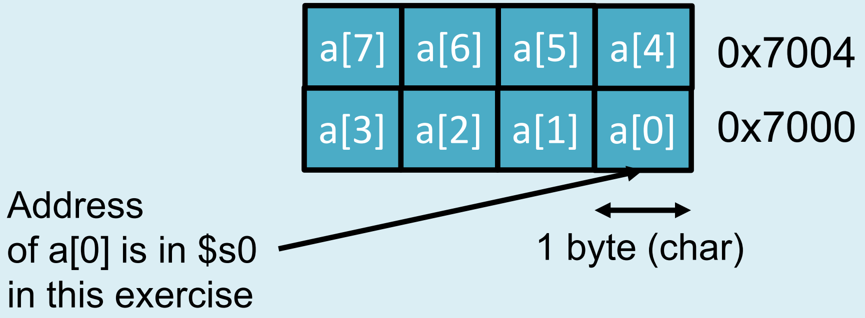
- **Translate the given high-level language code to MIPS assembly**
  - Assume that the base address of a ***char*** array a is in register $s0
    - base address of an array: the address of the first element of the array
  - Assume that the value of i is in $t0.

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 + 1byte * i = $s0 + $t0

Is this a valid operation to get o
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)**

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY