CDA 3103: Study Set 5

BASIC ASSEMBLY, TYPES OF INSTRUCTIONS, MEMORY FORMAT, NAVIGATING JUMPS AND BRANCHES, CODE CONVERSION

Review: Basic Assembly

When a higher level programming language is compiled it is converted first into an assembly language. It is then further processed into machine code which can then be run on a processor.

The “instructions” are the individual commands that a computer can understand. The “instruction set” is the collection of all of the commands. The “instruction set architecture” is the processor that is capable of running a particular instruction set.

MIPS is an instruction set architecture. It has a processor and an instruction set. Most arithmetic instructions are formed from an operation, a destination, and two source operands.

add $t0, $t1, $t2

For example, the above add operation will add the contents of registers t1 and t2 together and place the result into the t0 register.

Review: Basic Assembly

These are called “R-Type Instructions”. Most R-Type instructions have the same format:

<operation> <destination register>, <source register1>, <source register2>

Here are some examples:

add	add $t1,$t2,$t3	$t1 = $t2 + $t3
subtract	sub $t1,$t2,$t3	$t1 = $t2 – $t3
set less than	slt $t1,$t2,$t3	$t1 = ($t2 < $t3)
and	and $t1,$t2,$t3	$t1 = $t2 & $t3
or	or $t1,$t2,$t3	$t1 = $t2 | $t3

There are also unsigned versions of several of these operations.

Review: Basic Assembly

There are some exceptions to the usual R-Type format. These are some of the special R-Type instructions:

shift left logical	sll $t1,$t2,10	$t1 = $t2 << 10
shift right logical	srl $t1,$t2,10	$t1 = $t2 >> 10

In these cases we specify the number of bits we want a single operand to be shifted.

Review: Basic Assembly

Integer multiplication and division are also R-Type instructions with some unique notes. Recall that our result register in these cases is twice the size of our input registers. We create this double register by combining two regular sized registers called Hi and Lo.

Hi is the upper portion of the product or remainder register and Lo is the lower portion of the product or remainder register.

multiply	mult $t1, $t2	HiLo = $t1 * $t2	
divide	div $t1, $t2	Lo = $t1 / $t2	Hi = $t1 % $t2

If we want to store these results in our register file, we need to move them from Hi or Lo to a register in the register file:

Move from Hi	mfhi $t1	$t1 = Hi
Move from Lo	mflo $t1	$t1 = Lo

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

```
_______ $t0, $s2, $s3
sub     $t1, _______, $s4
div     $s3, _______
_______ $t2
sub     $t0, $t0, _______
add     $s1, _______, $t2
```

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

Partial Credit 1: The compiler has to break down each line of C into the most basic steps. Our arithmetic operations can only have two operands, so we will need to calculate the values in parentheses separately. We can store these temporary quantities in our temporary registers.

Solution 1: The first step is to add b and c together. We store this in $t0. So our first instruction is add.

```
_add_   $t0, $s2, $s3
sub     $t1, ______, $s4
div     $s3, ______
______  $t2
sub     $t0, $t0, ______
add     $s1, ______, $t2
```

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

Partial Credit 2: Another temporary quantity is b-d. In a subtraction operation, the order is important.

Solution 2: The second line is a subtraction, so we can use this to calculate b-d. We need to subtract d from b, so b should be the first operand and d should be the second. D's register and the destination are already filled in: we just need to add in B's register.

```
_add_   $t0, $s2, $s3
sub     $t1, __$s2__, $s4
div     $s3, ______
______  $t2
sub     $t0, $t0, ______
add     $s1, ______, $t2
```

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

Partial Credit 3: Our third and final temporary quantity is c/d. Remember that division instructions look a bit different from other arithmetic instructions because the destination is pre-determined.

Solution 3: The destination will be the Hi and Lo registers for division, so we only need to specify the input operands. Like subtraction, order is important. To divide c by d, we list c's register first, then d's.

```
_add_   $t0, $s2, $s3
sub     $t1, __$s2__, $s4
div     $s3, __$s4__
_____   $t2
sub     $t0, $t0, _____
add     $s1, _____, $t2
```

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

Partial Credit 4: In order to use the result of our division operation we need to move it to a more accessible location. We have two move operations. Which is needed here?

```
_add_   $t0, $s2, $s3
sub     $t1, __$s2__, $s4
div     $s3, __$s4__
_mflo_  $t2
sub     $t0, $t0, ______
add     $s1, ______, $t2
```

Solution 4: In the division algorithm, the upper portion of the result register holds the remainder and the lower portion of the result register holds the quotient. Therefore, we need to move from Lo to a different register.

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

Partial Credit 5: Now we can start constructing a. We have $t0 holding b+c, $t1 holding b-d, and $t2 holding c/d. In C these would be combined left to right with $t0-$t1 happening first.

Solution 5: To calculate $t0-$t1 we will do a subtraction operation with $t0 as the first operand and $t1 as the second. We can store the result back in $t0 as we no longer need its previous value.

```
_add_   $t0, $s2, $s3
sub     $t1, __$s2__, $s4
div     $s3, __$s4__
_mflo_  $t2
sub     $t0, $t0, __$t1__
add     $s1, ______, $t2
```

Example: Basic Assembly

Given: Consider the following line of C code:

a = (b+c) – (b-d) + (c/d)

where integers {a, b, c, d} reside in {$s1,$s2,$s3,$s4} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction or register:

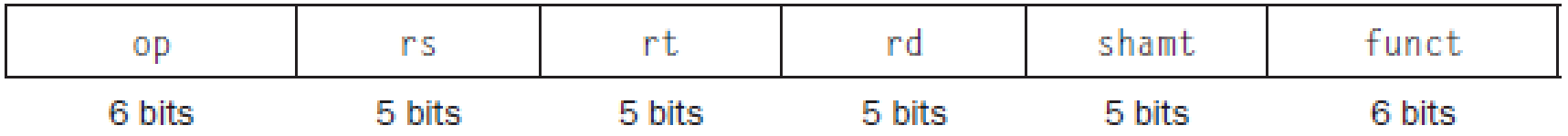
Partial Credit 6: Now we have $t0 = (b+c) – (b-d) and we just need to add in the quantity (c/d) which is stored in $t2.

Solution 6: This final operation is an add, where we want the result to end up in a's register. To add in the remaining temporary quantity, we add $t2 to $t0.

```
_add_   $t0, $s2, $s3
sub     $t1, __$s2__, $s4
div     $s3, __$s4__
_mflo_  $t2
sub     $t0, $t0, __$t1__
add     $s1, __$t0__, $t2
```

Review: Types of Instructions

Each instruction type has a unique format. All instructions take 32 bits to specify. RType instructions have 6 fields:



The op field is also known as the Opcode or the Operation Code. All R-Type instructions have an Opcode of 000000.

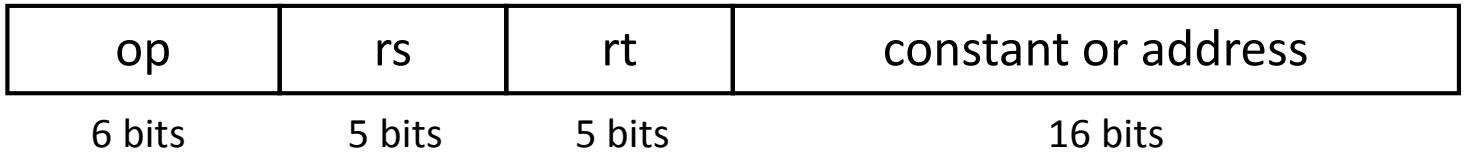
Rs and Rt are the two source registers. Rd is the destination register.

Shamt is the shift amount which is only used in shift operations. For other R-types this field would be set to 0.

The funct field is known as Function. This specifies which R-Type instruction we are performing.

Review: Types of Instructions

Another type of instruction in MIPS is the I-Type. It has the following format:



The op field is also known as the Opcode or the Operation Code. Each I-Type instruction has a unique Opcode.

Rs is a source register. Rt may be a source or a destination register depending on the instruction

The remaining 16 bits are used as a constant either for arithmetic or addressing. For arithmetic I-Types this constant represents a numerical value that will participate in our operation.

Review: Types of Instructions

Here are some of the more common arithmetic I-Type instructions:

add immediate	addi $1,$2,100	$1 = $2 + 100
and immediate	andi $1,$2,10	$1 = $2 & 10
or immediate	ori $1,$2,10	$1 = $2 | 10
set less than imm	slti $1,$2,100	$1 = ($2 < 100)

Example: Arithmetic I-Types

Given: Consider the following line of C code:

a = (b+c+d) – (e+5)

where integers {a, b, c, d, e} reside in {$s1,$s2,$s3,$s4, $s5} respectively. What is the smallest number of instructions we can use to complete the same task in MIPS?

Example: Arithmetic I-Types

Given: Consider the following line of C code:

a = (b+c+d) – (e+5)

where integers {a, b, c, d, e} reside in {$s1,$s2,$s3,$s4, $s5} respectively. What is the smallest number of instructions we can use to complete the same task in MIPS?

Partial Credit 1: To determine the required number of instructions we need to convert the C code to MIPS while trying to use as few assembly instructions as possible. There are two quantities in parentheses that we will need to calculate first. Let's start with (e+5).

Solution 1:

```
addi    $t0, $s5, 5     #to add 5 to e, we need to use addi, the add instruction that allows us
                        #to add a constant value to a register.  Since we do not want to change
                        #the variable e, we need to store the result in a temporary register.
```

Example: Arithmetic I-Types

Given: Consider the following line of C code:

a = (b+c+d) – (e+5)

where integers {a, b, c, d, e} reside in {$s1,$s2,$s3,$s4, $s5} respectively. What is the smallest number of instructions we can use to complete the same task in MIPS?

Partial Credit 2: Now we need to calculate the second quantity (b+c+d). With using r-type instructions we can add two variables at a time. The fewest instructions we can use to calculate this quantity is two.

Solution 2:

```
add     $t1, $s2, $s3     #first we add b and c together and store that in a temporary register
add     $t1, $t1, $s4     #then we add to the temporary register the value in d
```

Example: Arithmetic I-Types

Given: Consider the following line of C code:

a = (b+c+d) – (e+5)

where integers {a, b, c, d, e} reside in {$s1,$s2,$s3,$s4, $s5} respectively. What is the smallest number of instructions we can use to complete the same task is MIPS?

Partial Credit 3: Now that we have both quantities, we can perform the subtraction to calculate A.

Solution 3:

```
sub     $s1, $t1, $t0     #subtract the value in $t0 from the value in $t1 to get A
```

Example: Arithmetic I-Types

Given: Consider the following line of C code:

a = (b+c+d) – (e+5)

where integers {a, b, c, d, e} reside in {$s1,$s2,$s3,$s4, $s5} respectively. What is the smallest number of instructions we can use to complete the same task in MIPS?

Partial Credit 4: Now that we have converted the C code segment we can count the number of instructions used.

Solution 4:

```
addi    $t0, $s5, 5
add     $t1, $s2, $s3
add     $t1, $t1, $s4
sub     $s1, $t1, $t0
```

The smallest number of instructions we can use to complete the same task in MIPS is 4.

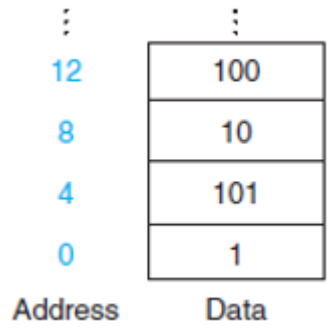
Review: Memory Format

Memory may be viewed as an array of bytes. Each index of the array refers to a single byte in memory, which is 8 bits. We say that memory is byte addressed.

In order to access a full word (4 bytes) we need our data in memory to be word aligned. This means the first byte of the word needs to be in an address that is evenly divisible by 4.

To place a value in memory or retrieve a value from memory, we need to specify the length of that value and the byte that it starts with. Lengths can be a single byte, a half word, or a word.

This diagram shows an array of words in memory. Each word begins at an address (index) that is evenly divisible by four. The four bytes that make up the content “101” are stored in memory locations 4, 5, 6, and 7.



Review: Memory Format

To take a value from memory and place it in the register file, we use one of the load operations:

load word	lw $1, 8($2)	$1=Mem[8+$2]
load halfword	lh $1, 6($3)	$1=Mem[6+$3]
load byte	lb $1, 5($3)	$1=Mem[5+$3]

The type of instruction specifies the length of the value we are loading. To specify the starting byte we construct an address from the source register and the 16-bit constant.

Review: Memory Format

To take a from the register file and place it in memory, we use one of the store operations:

store word	sw $3, 8($4)	Mem[$4+8]=$3
store halfword	sh $3, 6($2)	Mem[$2+6]=$3
store byte	sb $2, 7($3)	Mem[$3+7]=$3

The type of instruction specifies the length of the value we are storing. To specify the starting byte we construct an address from the source register and the 16-bit constant.

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[2] = b + 400

where A is an array of words and the base address of A is stored in $s0. Integer {b} resides in {$s1}. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

_____ $t0, _____, 400
_____ $t0, 8(_____)

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[2] = b + 400

where A is an array of words and the base address of A is stored in $s0. Integer {b} resides in {$s1}. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 1: There are two different tasks we need to accomplish. The first is calculating the quantity on the right side of the equal sign. Which instruction will let us add a constant?

Solution 1: Add immediate, written addi, allows us to add a constant to a value in a register.

```
_addi_    $t0, ______, 400
______    $t0, 8(______)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[2] = b + 400

where A is an array of words and the base address of A is stored in $s0. Integer {b} resides in {$s1}. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 2: There are two different tasks we need to accomplish. The first is calculating the quantity on the right side of the equal sign. To which register should we add 400?

Solution 2: On the right side of the equal sign, we are adding b and 400. Since b is stored in register $s1, that is what we need to add 400 to.

```
_addi_    $t0, __$s1__, 400
______    $t0, 8(______)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[2] = b + 400

where A is an array of words and the base address of A is stored in $s0. Integer {b} resides in {$s1}. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 3: There are two different tasks we need to accomplish. The second task is to place our calculated value in the array at the correct index. Which instruction places a value in memory?

Solution 3: To place an integer value in memory from the register file, we use "store word"

```
_addi_   $t0, __$s1__, 400
__sw__   $t0, 8(_____)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[2] = b + 400

where A is an array of words and the base address of A is stored in $s0. Integer {b} resides in {$s1}. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 4: There are two different tasks we need to accomplish. The second task is to place our calculated value in the array at the correct index. How do we calculate the correct address in memory for index 2?

```
_addi_    $t0, __$s1__, 400
__sw__    $t0, 8(__$s0__)
```

Solution 4: The array begins at the base address, which is stored in $s0. To get to index 2, we need to advance 2 words or 8 bytes. To calculate this address, we add 8 bytes to the base.

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

```
addi    _______, $s1, _______
sll     $t1, $s2, _______
_______ $t1, _______, $t1
sw      $t0, _______($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 1: There are two blanks in the first instruction. Since the instruction is addi, we should infer that the second blank needs to be a number instead of a register. The only number being added is 400 and we are adding this to B, which is stored in register $s1 and $s1 is the other source for this instruction.

```
addi    ______, $s1, _400_
sll     $t1, $s2, ______
_____   $t1, ______, $t1
sw      $t0, ______($t1)
```

Solution 1:

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 2: There are two blanks in the first instruction. The second blank is for the destination register. This value should be placed in memory, so we can determine which register should be used by looking at the later sw instruction.

```
addi    __$t0__, $s1, _400_
sll     $t1, $s2, ______
_____   $t1, ______, $t1
sw      $t0, ______($t1)
```

Solution 2:

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 3: The second instruction is sll. Recall that this is "Shift Left Logical". What would we need to shift? The register being shifted is $s2, which is i.

To determine a memory location based on A[i], we need to calculate the address that is "i" words beyond the base address of A. Since memory is byte addressed, this will need to be i*4 bytes beyond the base address of A. Instead of multiplying by 4, we should shift left by 2 because it will be faster.

Solution 3:

```
addi    __$t0__, $s1, _400_
sll     $t1, $s2, __2__
_____   $t1, _____, $t1
sw      $t0, _____($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 4: To determine a memory location based on A[i], we need to calculate the address that is "i" words beyond the base address of A.

Now that we have i*4, we can add that to the base address of A to calculate our memory location. Both i*4 and the base address of A are stored in registers, so we will need to use the R-Type instruction add.

Solution 4:

```
addi    __$t0__, $s1, _400_
sll     $t1, $s2, __2__
__add__ $t1, ______, $t1
sw      $t0, ______($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 5: To determine a memory location based on A[i], we need to calculate the address that is "i" words beyond the base address of A.

Now that we have i*4, we can add that to the base address of A to calculate our memory location. Both i*4 and the base address of A are stored in registers. The base address of A is stored in $s0.

Solution 5:

```
addi    __$t0__, $s1, _400_
sll     $t1, $s2, __2__
__add__ $t1, __$s0__, $t1
sw      $t0, ______($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B + 400

where A is an array of words and the base address of A is stored in $s0. Integers {b, i} reside in {$s1, $s2} respectively. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 6: Now that we have the full address of A[i] stored in register $t1, we can place the value of B+400 in memory at that location. We use the instruction store word to do this. What should the offset be?

Since we have calculated the explicit address, there is no additional offset.

Solution 6:

```
addi    __$t0__, $s1, _400_
sll     $t1, $s2, __2__
__add__ $t1, __$s0__, $t1
sw      $t0, __0__($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

```
sll     $t0, _____, 2
_____   $t1, $s1, $t0
_____   $t1, 0($t1)
addi    $t2, _____, 400
add     $t1, $s0, _____
_____   $t2, 0($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 1: In this problem, there are two different memory access instructions. We need to first load the value that is stored at B[i] into a register in order to add 400 to it. Then we can place the result into memory at A[i].

The first instruction is a shift left logical. What variable would we need to shift? To calculate memory addresses we will need to go "i" words or i*4 bytes beyond our base addresses. To calculate i*4, we shift i left twice.

Solution 1:

```
sll     $t0, __$s2__, 2
_____   $t1, $s1, $t0
_____   $t1, 0($t1)
addi    $t2, _____, 400
add     $t1, $s0, _____
_____   $t2, 0($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 2: To finish calculating the memory address for B[i], we add i*4 to the base address of B, using an add instruction.

Solution 2:

```
sll     $t0, __$s2__, 2
__add__ $t1, $s1, $t0
_____   $t1, 0($t1)
addi    $t2, _____, 400
add     $t1, $s0, _____
_____   $t2, 0($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 3: Now that we have the address for B[i], we can retrieve it's value and place into a temporary register. We use load word to take an integer out of memory and place it in a register.

The offset is 0 because we have already calculated the explicit address of B[i].

Solution 3:

```
sll     $t0, __$s2__, 2
__add__ $t1, $s1, $t0
__lw__  $t1, 0($t1)
addi    $t2, ______, 400
add     $t1, $s0, ______
______  $t2, 0($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 4: Now we can add 400 to the value at B[i]. Since we stored that value in register $t1, we want to add 400 to register $t1.

Solution 4:

```
sll     $t0, __$s2__, 2
__add__ $t1, $s1, $t0
__lw__  $t1, 0($t1)
addi    $t2, __$t1__, 400
add     $t1, $s0, ______
______  $t2, 0($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 5: To store the result of B[i] + 400, we need to calculate the memory location of A[i]. We have already calculated and preserved the value of i*4 in register $t0, so we can add this to the base address of A stored in $s0.

Solution 5:

```
sll     $t0, __$s2__, 2
__add__ $t1, $s1, $t0
__lw__  $t1, 0($t1)
addi    $t2, __$t1__, 400
add     $t1, $s0, __$t0__
_____   $t2, 0($t1)
```

Example: Arrays, Loading, and Storing

Given: Consider the following segment of C code:

A[i] = B[i] + 400

where both A and B are arrays of words whose base addresses are stored in $s0 and $s1 respectively. Integer i is stored in register $s2. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Partial Credit 6: Finally, we can stored B[i] + 400 into A[i]. The value of B[i] + 400 is stored in $t2. The address of A[i] is stored in $t1. We use sw to store the contents of $t2 into the memory location in $t1.

Solution 6:

```
sll     $t0, __$s2__, 2
__add__ $t1, $s1, $t0
__lw__  $t1, 0($t1)
addi    $t2, __$t1__, 400
add     $t1, $s0, __$t0__
__sw__  $t2, 0($t1)
```

Review: Jumps and Branches

Jumps and branches allow to leave our current location in instruction memory and go to a different location. A branch is a conditional jump:

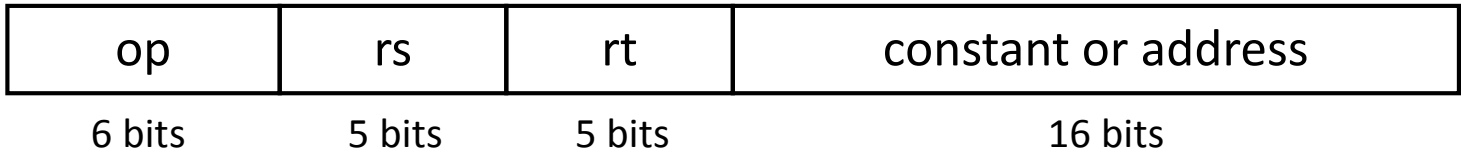
Branch if equal	beq $s1, $s2, L	if (s1 == s2), go to L
Branch if not equal	bne $s1, $s2, L	if (s1 != s2), go to L

A jump is unconditional:

jump	j 10000	PC = PC:40000

Review: Jumps and Branches

Branches are I-Type instructions, with the same I-Type format as previous I-Type instructions:



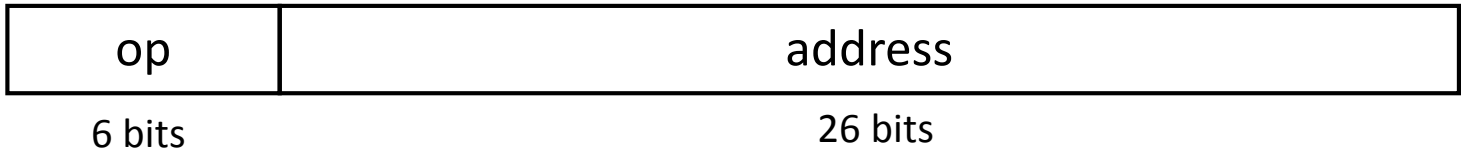
The op field is also known as the Opcode or the Operation Code. Each I-Type instruction has a unique Opcode.

Rs and RT are both source registers. We will compare the data in these two registers to determine if they are the same (branch if equal) or different (branch if not equal).

The remaining 16 bits are used to calculate a “branch target address”. If the condition is met this value will be added to the new program counter as the next instruction address.

Review: Jumps and Branches

Jumps are J-Type instructions. They have the following format:



The op field is also known as the Opcode or the Operation Code. Each J-Type instruction has a unique Opcode.

The remaining 26 bits are used to calculate a “pseudo-direct address”. This 28 bit address is concatenated with the first 4 bits of the current program counter to calculate the next instruction address.

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
        addi    $t1, $zero, 1
        addi    $t2, $zero, 2
        addi    $t3, $zero, 2
L1:     beq     $t2, $zero, L2
        sub     $t2, $t2, $t1,
        add     $t3, $t3, $t3
        j       L1
L2:
```

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                addi    $t1, $zero, 1
2                addi    $t2, $zero, 2
3                addi    $t3, $zero, 2
4        L1:     beq     $t2, $zero, L2
5                sub     $t2, $t2, $t1,
6                add     $t3, $t3, $t3
7                j       L1
8        L2:
```

Partial Credit 1: To determine the value store in the register, we need to track the values for all three temporary registers being used in the fragment of code. Registers $t1, $t2, and $t3 are given initial values in the first three lines.

Solution 1:

$t1 = 1 + 0 = 1
$t2 = 2 + 0 = 2
$t3 = 2 + 0 = 2

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

Partial Credit 2: In line 4 we have a beq instruction. This instruction is branch if equal. We need to check the two register parameters and see if they are equal to each other. If they are we will jump to L2.

```
1                addi    $t1, $zero, 1
2                addi    $t2, $zero, 2
3                addi    $t3, $zero, 2
4        L1:     beq     $t2, $zero, L2
5                sub     $t2, $t2, $t1,
6                add     $t3, $t3, $t3
7                j       L1
8        L2:
```

Solution 2:
$t1 = 1
$t2 = 2
$t3 = 2

Since $t2 is equal to 2 and not 0, we will not jump but instead continue to line 5

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Partial Credit 2: In line 5 we subtract $t1 from $t2 and place the result back into the $t2 register. This will change the value of $t2.

Solution 2:

$t1 = 1
$t2 = 2 – 1 = 1
$t3 = 2

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

Partial Credit 3: In line 6 we add $t3 to itself and place the result back into the $t3 register. This will change the value of $t3.

Solution 3:
$t1 = 1
$t2 = 1
$t3 = 2 + 2 = 4

```
1                addi    $t1, $zero, 1
2                addi    $t2, $zero, 2
3                addi    $t3, $zero, 2
4        L1:     beq     $t2, $zero, L2
5                sub     $t2, $t2, $t1,
6                add     $t3, $t3, $t3
7                j       L1
8        L2:
```

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

Partial Credit 4: Line 7 contains an unconditional jump back to L1 in line 4.

Solution 4:
$t1 = 1
$t2 = 1
$t3 = 4

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Partial Credit 4: In line 4 we have a beq instruction. This instruction is branch if equal. We need to check the two register parameters and see if they are equal to each other. If they are we will jump to L2.

Solution 4:
$t1 = 1
$t2 = 1
$t3 = 4

Since $t2 is equal to 1 and not 0, we will not jump but instead continue to line 5

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Partial Credit 4: In line 5 we subtract $t1 from $t2 and place the result back into the $t2 register. This will change the value of $t2.

Solution 4:

$t1 = 1
$t2 = 1 - 1 = 0
$t3 = 4

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Partial Credit 5: In line 6 we add $t3 to itself and place the result back into the $t3 register. This will change the value of $t3.

Solution 5:

$t1 = 1
$t2 = 0
$t3 = 4 + 4 = 8

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

Partial Credit 5: Line 7 contains an unconditional jump back to L1 in line 4.

Solution 5:
$t1 = 1
$t2 = 0
$t3 = 8

```
1                addi    $t1, $zero, 1
2                addi    $t2, $zero, 2
3                addi    $t3, $zero, 2
4        L1:     beq     $t2, $zero, L2
5                sub     $t2, $t2, $t1,
6                add     $t3, $t3, $t3
7                j       L1
8        L2:
```

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Partial Credit 6: In line 4 we have a beq instruction. This instruction is branch if equal. We need to check the two register parameters and see if they are equal to each other. If they are we will jump to L2.

Solution 6:
$t1 = 1
$t2 = 0
$t3 = 8

Now that $t2 is equal to zero, we will jump to L2.

Example: Jumps and Branches

Given: What value is contained in **$t3** upon completion of the MIPS code below: _____

```
1                   addi    $t1, $zero, 1
2                   addi    $t2, $zero, 2
3                   addi    $t3, $zero, 2
4           L1:     beq     $t2, $zero, L2
5                   sub     $t2, $t2, $t1,
6                   add     $t3, $t3, $t3
7                   j       L1
8           L2:
```

Partial Credit 6: In line 8 there is no instruction, so we will end the code segment.

Solution 6:

$t1 = 1
$t2 = 0
$t3 = 8

The value contained $t3 is 8.

Example: Jumps and Branches

Given: Consider the following fragment of C code:

```
i=0;
while (i <= 20) {
        A[i] = i;
        i++;
}
```

Assume that A is an array of words and that the base address of A is in $s0. Integer i resides in $s1. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

```
        addi    $s1, _____, _____
loop:   _____   $t1, $s1, 21
        _____   $t1, $zero, end
        sll     $t2, $s1, _____
        add     $t2, $t2, $s0
        _____   $s1, 0($t2)
        addi    $s1, $s1, 1
        j loop
end:    …
```

Example: Jumps and Branches

Given: Consider the following fragment of C code:

```
i=0;
while (i <= 20) {
        A[i] = i;
        i++;
}
```

Assume that A is an array of words and that the base address of A is in $s0. Integer i resides in $s1. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Solution 1:

```
        addi    $s1, __$zero__, __0__
loop:   _____   $t1, $s1, 21
        _____   $t1, $zero, end
        sll     $t2, $s1, _____
        add     $t2, $t2, $s0
        _____   $s1, 0($t2)
        addi    $s1, $s1, 1
        j loop
end:    …
```

Partial Credit 1: The first step in the C fragment is to initialize i to zero. We can do this with an add-immediate instruction by adding the constant 0 to the zero register.

Example: Jumps and Branches

Given: Consider the following fragment of C code:

```
i=0;
while (i <= 20) {
        A[i] = i;
        i++;
}
```

Assume that A is an array of words and that the base address of A is in $s0. Integer i resides in $s1. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Solution 2:

```
        addi    $s1, __$zero__, __0__
loop:   ___slti___  $t1, $s1, 21
        _________   $t1, $zero, end
        sll     $t2, $s1, _________
        add     $t2, $t2, $s0
        _________   $s1, 0($t2)
        addi    $s1, $s1, 1
        j loop
end:    …
```

Partial Credit 2: To begin the while loop we need to verify thati is less than or equal to 20. Because i is an integer, this is the same as check to see if i is less than 21. We use the instruction slti to set $t1 to 1 if i is less than 21. $t1 will be 0 otherwise.

Example: Jumps and Branches

Given: Consider the following fragment of C code:

```
i=0;
while (i <= 20) {
        A[i] = i;
        i++;
}
```

Assume that A is an array of words and that the base address of A is in $s0. Integer i resides in $s1. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Solution 3:

```
        addi    $s1, __$zero__, __0__
loop:   ___slti___  $t1, $s1, 21
        ___beq___   $t1, $zero, end
        sll     $t2, $s1, _____________
        add     $t2, $t2, $s0
        _____________   $s1, 0($t2)
        addi    $s1, $s1, 1
        j loop
end:    …
```

Partial Credit 3: If i is less than 21, then $t1 will be equal to 1. If i is 21 or greater, $t1 will be equal to 0. If $t1 is equal to 0, then the condition (i<=20) was not true and we should exit the loop. We will use branch if equal to jump to the end of the loop if $t1 is 0.

Example: Jumps and Branches

Given: Consider the following fragment of C code:

```
i=0;
while (i <= 20) {
        A[i] = i;
        i++;
}
```

Assume that A is an array of words and that the base address of A is in $s0. Integer i resides in $s1. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Solution 4:

```
        addi        $s1, __$zero__, __0__
loop:   ___slti___  $t1, $s1, 21
        ___beq___   $t1, $zero, end
        sll         $t2, $s1, __2__
        add         $t2, $t2, $s0
        ___________ $s1, 0($t2)
        addi        $s1, $s1, 1
        j loop
end:    …
```

Partial Credit 4: Within the loop, we need to set A[i] equal to i. To accomplish this we need to calculate the exact address in memory of the ith index of A. This location is "i" words or i*4 bytes beyond the base of A. To multiply i by 4, we shift it to the left by 2 bits. This is then added to the base address of A.

Example: Jumps and Branches

Given: Consider the following fragment of C code:

```
i=0;
while (i <= 20) {
        A[i] = i;
        i++;
}
```

Assume that A is an array of words and that the base address of A is in $s0. Integer i resides in $s1. Complete the corresponding assembly language fragment by writing in the correct instruction, register, or numerical value:

Solution 4:

```
        addi    $s1, __$zero__, __0__
loop:   ___slti___  $t1, $s1, 21
        ___beq___   $t1, $zero, end
        sll     $t2, $s1, __2__
        add     $t2, $t2, $s0
        ___sw___    $s1, 0($t2)
        addi    $s1, $s1, 1
        j loop
end:    …
```

Partial Credit 4: To place the value of i into A[i] we use the store word instruction. $t2 contains the calculate address of A + i * 4, so we do not include any additional offset.

Review: Procedures

Most programming languages allow you to separate out individual tasks and subtasks into their own functions or methods. The abstract term for this in assembly is “procedures”. The code for procedures is appended to the end of the main text and we access them by jumping to their starting location and jumping back when finished. There are two special jump instructions for this purpose:

jump and link	jal 10000	$31 = PC + 4;	PC = 40000
jump register	jr $31	PC = $31	

Register $31 is known as the return address register. Parameters are passed to the procedure through argument registers and results are returned through return value registers.

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Solution 1:

```
recSum:   slti        $t0, $a0, 2          #$t0 = 1 if n <= 1
```

Partial Credit 1: This function contains two major sections: the if and the else. We should assume that n will arrive in the first argument register: $a0. So our first step in the function is to see if $a0 is less than or equal to 1. Since this is an integer, we can check to see if $a0 is less than 2.

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Solution 2:

```
recSum:  slti   $t0, $a0, 2      #$t0 = 1 if n <= 1
         beq    $t0, $zero, else #jump to else if $t0=0
```

Partial Credit 2: If $a0 is not less than 2 then $t0 will be equal to 0 and we want to jump to the else statement.

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Solution 3:

```
recSum:  slti   $t0, $a0, 2        #$t0 = 1 if n <= 1
         beq    $t0, $zero, else   #jump to else if $t0=0
         add    $v0, $a0, $zero    #place $a0 in $v0
         jr     $ra                #return
```

Partial Credit 3: If $a0 is less than 2 then we should return n. We need to place n (stored in $a0) into the first return value register ($v0). Then we can use our return instruction to return to the calling location stored the return address register ($ra).

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Solution 4:

```
recSum:  slti   $t0, $a0, 2        #$t0 = 1 if n <= 1
         beq    $t0, $zero, else   #jump to else if $t0=0
         add    $v0, $a0, $zero    #place $a0 in $v0
         jr     $ra                #return
else:    addi   $sp, $sp, -8       #move the sp down 2
         sw     $ra, 4($sp)        #store $ra in memory
         sw     $a0, 0($sp)        #store $a0 in mem
```

Partial Credit 4: In the else portion of the function we have a function call. To call a function we need to place the parameter n-1 into $a0. But we also need to preserve the parent function's copy of $a0. Similarly, when we call the function, $ra is automatically replaced and we need to preserve a copy of the parent function's $ra. We store both of these on the stack.

First, move the stack pointer ($sp) to accommodate two new values. Then store $a0 and $ra on the stack.

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Solution 5:

```
recSum:  slti   $t0, $a0, 2        #$t0 = 1 if n <= 1
         beq    $t0, $zero, else   #jump to else if $t0=0
         add    $v0, $a0, $zero    #place $a0 in $v0
         jr     $ra                #return
else:    addi   $sp, $sp, -8       #move the sp down 2
         sw     $ra, 4($sp)        #store $ra in memory
         sw     $a0, 0($sp)        #store $a0 in mem
         addi   $a0, $a0, -1       #subtract 1 from $a0
```

Partial Credit 5: To perform the actual function call we need to place the parameter n-1 into $a0. Now that we have preserved the parent function's copy of $a0 on the stack, we can modify $a0.

Remember that MIPS does not contain a subi instruction. Instead we use addi with a negative number.

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Solution 6:

```
recSum:  slti   $t0, $a0, 2          #$t0 = 1 if n <= 1
         beq    $t0, $zero, else     #jump to else if $t0=0
         add    $v0, $a0, $zero      #place $a0 in $v0
         jr     $ra                  #return
else:    addi   $sp, $sp, -8         #move the sp down 2
         sw     $ra, 4($sp)          #store $ra in memory
         sw     $a0, 0($sp)          #store $a0 in mem
         addi   $a0, $a0, -1         #subtract 1 from $a0
         jal    recSum               #call recSum
```

Partial Credit 6: Everything is now set up for the function call, so we use the instruction jump and link to jump to the function and setup $ra for when the function returns.

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Partial Credit 7: When we return from our recursive call, we restore the stack to it's previous state.

Solution 7:

```
recSum:  slti   $t0, $a0, 2        #$t0 = 1 if n <= 1
         beq    $t0, $zero, else   #jump to else if $t0=0
         add    $v0, $a0, $zero    #place $a0 in $v0
         jr     $ra                #return
else:    addi   $sp, $sp, -8       #move the sp down 2
         sw     $ra, 4($sp)        #store $ra in memory
         sw     $a0, 0($sp)        #store $a0 in mem
         addi   $a0, $a0, -1       #subtract 1 from $a0
         jal    recSum             #call recSum
         lw     $a0, 0($sp)        #restore $a0
         lw     $ra, 4($sp)        #restore $ra
         addi   $sp, $sp, 8        #move the sp up 2
```

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Partial Credit 8: Now that we have returned from the function call, we should have the result of "recSum(n-1)" stored in $v0. We need to add n to this and place the result in $v0 to be our return value.

Solution 8:

```
recSum: slti    $t0, $a0, 2         #$t0 = 1 if n <= 1
        beq     $t0, $zero, else    #jump to else if $t0=0
        add     $v0, $a0, $zero     #place $a0 in $v0
        jr      $ra                 #return
else:   addi    $sp, $sp, -8        #move the sp down 2
        sw      $ra, 4($sp)         #store $ra in memory
        sw      $a0, 0($sp)         #store $a0 in mem
        addi    $a0, $a0, -1        #subtract 1 from $a0
        jal     recSum              #call recSum
        lw      $a0, 0($sp)         #restore $a0
        lw      $ra, 4($sp)         #restore $ra
        addi    $sp, $sp, 8         #move the sp up 2
        add     $v0, $v0, $a0       #v0 = n+recSum(n-1)
```

Example: Procedures

Given: Convert the following function written in the C programming language into a MIPS procedure.

```
int recSum(int n) {
        if (n <= 1)
                return n;
        else
                return n + recSum(n-1);
}
```

Partial Credit 9: To end the function we use the return instruction to return to the calling location stored the return address register ($ra).

Solution 9:

```
recSum: slti    $t0, $a0, 2         #$t0 = 1 if n <= 1
        beq     $t0, $zero, else    #jump to else if $t0=0
        add     $v0, $a0, $zero     #place $a0 in $v0
        jr      $ra                 #return
else:   addi    $sp, $sp, -8        #move the sp down 2
        sw      $ra, 4($sp)         #store $ra in memory
        sw      $a0, 0($sp)         #store $a0 in mem
        addi    $a0, $a0, -1        #subtract 1 from $a0
        jal     recSum              #call recSum
        lw      $a0, 0($sp)         #restore $a0
        lw      $ra, 4($sp)         #restore $ra
        addi    $sp, $sp, 8         #move the sp up 2
        add     $v0, $v0, $a0       #v0 = n+recSum(n-1)
        jr      $ra                 #return
```