1. Pre-Check

This section is designed as a check to help you determine whether you understand the concepts covered in class. Please answer "true/false" to the following questions and include an explanation.

1.1. After calling a function in MIPS, when this function returns, registers **`$t0-$t9`** may have been modified during the execution of the function, but the values of registers **`$v0`** and **`$v1`** will be preserved.

1.2. If **`$a0`** is a pointer to the beginning of an array **`x`**, the MIPS instruction "**`lw $s0, 4($a0)`**" will always load the value of **`x[1]`** in **`$s0`**.

1.3. Assuming integers are 4 bytes wide, adding the ASCII character '**`d`**' to the address of an array of integers would point to the 26th element of that array (assuming the array is large enough).

1.4. The MIPS instructions "**`j label`**" and "**`jal label`**" do exactly the same thing.

2. Programming in MIPS

Consider the array (definition in C):

```
int arr[] = {1,2,3,4,5,6,0};
```

Suppose register **`$s0`** contains the address of the element at index 0 in `arr`. Also assume that the size of integers is 4 bytes. What do the following statements do in the MIPS code snippets below (note that some code has errors)

2.1.
```
lw $t0, 12($s0)
add $t1, $t0, $s0
sw $t0, 4($t1)
```

2.2.
```
addiu $s1, $s0, 27
lh $t0, -3($s1)
```

2.3.
```
addiu $s1, $s0, 24
lh $t0, -3($s1)
```

2.4.
```
addiu $t0, $0, 12
sw $t0, 6($s0)
```

2.5.
```
addiu $t0, $0, 8
sw $t0, -4($s0)
```

2.6.
```
addiu $s1, $0, 10
addiu $t0, $0, 6
sw $t0, 2($s1)
```

In the above code snippets, what other instructions could be used instead of each "read from/save to" memory to remove alignment errors? (Edit ONLY the problematic instruction).

3. C ↔ MIPS conversions

Convert instruction for instruction from (resp. to) C language to (resp. from) MIPS Assembler.

	C	MIPS

3.1.	`// $s0 -> a, $s1 -> b`
	`// $s2 -> c, $s3 -> z`
	`int a = 4, b = 5, c = 6, z;`
	`z = a + b + c + 10;`	

3.2.	`// $s0 -> int *p = intArr;`
	`// $s1 -> a;`
	`*p = 0;`
	`int a = 2;`
	`p[1] = p[a] = a;`	

3.3.

```
// $s0 -> a, $s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
}
else
{
    b = a - 1;
}
```

3.4.

```
    addi $s0, $0, 0
    addi $s1, $0, 1
    addi $t0, $0, 30
loop:
    beq $s0, $t0, exit
    add $s1, $s1, $s1
    addi $s0, $s0, 1
    j loop
exit:
```

3.5.

```
// $s0 -> n, $s1 -> sum
// assume n is initially > 0
for(int sum = 0; n > 0; n--) {
  sum += n;
}
```

4. Arrays and Lists in MIPS

Consider the array (definition in C):

```
int arr[] = {3,1,4,1,5,9};
```

Assume the first element in this array is at memory address **`0xBFFFFF00`**. Consider also the following definition (in C) of a linked list (suppose that the first node of the linked list is located at memory address **`0xABCD0000`**).

```
struct ll {
    int val;
    struct ll* next;
}
struct ll* lst;
```

Note that the `next` field of the last node in the list contains the value **`NULL`** (i.e., the value 0).

Now suppose that register **`$s0`** contains the address of **`arr`** (i.e. **`$s0 = 0xBFFFFF00`**); and that register **`$s1`** contains the address of the first node of the list **`lst`** (i.e. **`$s1 = 0xABCD0000`**). Finally, suppose that integers and pointers in our system are 4 bytes wide.

Indicate for each question below what the MIPS instruction block does (The code blocks in the questions and their results are independent of each other).

4.1.
```
        lw $t0, 0($s0)
        lw $t1, 8($s0)
        add $t2, $t0, $t1
        sw, $t2, 4($s0)
```

4.2.
```
loop:   beq $s1, $0, end
        lw $t0, 0($s1)
        addi $t0, $t0, 1
        sw $t0, 0($s1)
        lw $s1, 4($s1)
        j loop
end:
```

4.3.
```
        add $t0, $0, $0
loop:   slti $t1, $t0, 6
        beq $t1, $0, end
        sll $t2, $t0, 2
        add $t3, $s0, $t2
        lw $t4, 0($t3)
        sub $t4, $0, $t4
        sw $t4, 0($t3)
        addi $t0, $t0, 1
        j loop
end:
```

5. Calling convention in MIPS

5.1. How can we pass arguments to functions in MIPS?

5.2. How are values returned by functions in MIPS?

5.3. What is **`$sp`** and how should it be used in the context of MIPS functions?

5.4. If a function needs to preserve the values of some registers before calling another function, which registers should the caller function save before the call and restore after the callee function returns?

5.5. Which registers should the callee function preserve (i.e. restore to initial state) before returning to the caller function?

5.6. In a bug-free program, which registers are guaranteed to be the same after a function call? Which registers are not guaranteed to be the same?

6. MIPS functions

Implement in MIPS the `sumSquare` function that computes the sum below

$$n^2 + (n-1)^2 + (n-2)^2 + \cdots + 1^2$$

Where n is a parameter passed to the function. If n is not positive then the function must return zero.

6.1. Let's start by coding the body of the function (i.e. the summation of quadratures).

```
sumSquare:
```

6.2. Since `sumSquare` is a function, and since we must follow MIPS calling conventions, in which register should the value of n be passed to this function? Which registers should be used for computing quadratures and summation? And in which register(s) should we return the result? On the other hand, we also need to make sure that when `sumSquare` returns, it does not modify any registers that the calling function might use. Considering your answers to the questions above, correct your code in 6.1 if necessary and add a prologue and epilogue to

`sumSquare`: