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.
FALSE! **`$v0`** and **`$v1`** registers are often used to store the values returned by a function.

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 **`$s0`**.
FALSE! This is only true for data types that are four bytes wide such as integers or floats. For data types like (char) that are only one byte wide, **`4($a0)`** will return the fifth element in the array (or data beyond the array if the actual size of the array is less than the requested index).

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).
TRUE! There is no fundamental difference between an integer, a character, and a memory address in Assembly (they are all bags of bits). So it is possible to manipulate data this way (this is not recommended, though).

1.4. The MIPS instructions "**`j label`**" and "**`jal label`**" do exactly the same thing.
FALSE! Although both instructions jump to the address specified by **`label`**, the "**`jal label`**" instruction additionally stores the address of the instruction that succeeds it in **`$ra`** register.

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)                                  # 2.6) lb, lh
add $t1, $t0, $s0
sw $t0, 4($t1)          # arr[2] <-- 4           # 2.6) sb, sh
```

```
2.2.    addiu $s1, $s0, 27
        lh $t0, -3($s1)         #   $t0 <-- 0           # 2.6) lw, lb

2.3.    addiu $s1, $s0, 24
        lh $t0, -3($s1)         #   misalignment error  # 2.6) lb

2.4.    addiu $t0, $0, 12
        sw $t0, 6($s0)          #   misalignment error  # 2.6) sh, sb

2.5.    addiu $t0, $0, 8
        sw $t0, -4($s0)         #   index out of bounds # 2.6) sh, sb

2.6.    addiu $s1, $0, 10
        addiu $t0, $0, 6        #
        sw $t0, 2($s1)          #   arr[3] <-- 6        # 2.6) sh, sb
```

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). Indeed, if the immediate constant in the read or save instruction modulo 4 is equal to 2 (i.e. `imm % 4 == 2`), then **`lh`** and **`sh`** can replace **`lw`** and **`sw`**. On the other hand, since all integer values in the array can be represented with less than 8 bits, the **`lb`** instruction (resp. **`sb`**) can always replace the **`lw`** and **`lh`** (resp. **`sw`** and **`sh`**) instructions.

See above for answers preceded by "2.6)" for whether **`lb`**, **`sb`**, **`lh`**, **`sh`**, **`lw`** or **`sw`** can work.

3. C ↔ MIPS conversions

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

3.1.

C	MIPS

`// $s0 -> a, $s1 -> b`
`// $s2 -> c, $s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`
	`addi $s0, $0, 4`
	`addi $s1, $0, 5`
	`addi $s2, $0, 6`
	`add  $s3, $s0, $s1`
	`add  $s3, $s3, $s2`
	`addi $s3, $s3, 10`

3.2.

`// $s0 -> int *p = intArr;`
`// $s1 -> a;`
`*p = 0;`
`int a = 2;`
`p[1] = p[a] = a;`
	`sw $0, 0($s0)`
	`addi $s1, $0, 2`
	`sw $s1, 4($s0)`
	`sll $t0, $s1, 2`
	`add $t0, $t0, $s0`
	`sw $s1, 0($t0)`

3.3.

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

```
    addi $s0, $0, 5
    addi $s1, $0, 10
    add $t0, $s0, $s0
    bne $t0, $s1, else
    xor $s0, $0, $0
    j exit
else:
    addi s1, s0, -1
exit:
```

3.4.

```
// $s0 -> s0, $s1 -> s1
// assuming integer values
// compute s1 = 2^30

s1 = 1;
for(s0 = 0; s0 != 30; s0++) {
    s1 *= 2;
}
```

```
    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;
}
```

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

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)
```
-> arr[1] = arr[0] + arr[2]

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:
```
-> increments the val field by 1 in each node of the linked list.

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:
```
-> flips the sign of all elements in the array arr.

5. Calling convention in MIPS

5.1. How can we pass arguments to functions in MIPS?
Use the four registers `$a0-$a3` to pass arguments to functions.

5.2. How are values returned by functions in MIPS?
Use registers `$v0` and `$v1` to return values from functions.

5.3. What is **`$sp`** and how should it be used in the context of MIPS functions?
`sp` stands for "stack pointer". `$sp` is the MIPS register used to represent the top of a hardware stack. The content of `$sp` is decremented to create more space (i.e. the memory address in `$sp` decreases in this case). To free up space in the stack, the `$sp` register is incremented (i.e. the memory address in `$sp` increases). The hardware stack is mainly used to save (and later restore) the contents of registers during function calls.

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?
Registers `$v0-$v1`, `$t0-$t9` and `$a0-$a3`.

5.5. Which registers should the callee function preserve (i.e. restore to initial state) before returning to the caller function?
Registers `$s0-$s7`, `$gp`, `$sp`, `$fp`, and `$ra`.

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?
Registers `$v0-$v1`, `$t0-$t9`, and `$a0-$a3` are not guaranteed to be the same after a function call (for this reason, they should be backed up, if necessary, by the caller function). Registers `$s0-$s7`, `$gp`, `$sp`, `$fp`, and `$ra` are guaranteed to be the same after a function call (i.e. the callee function **MUST** retain or restore the values of these registers before returning).

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:
       add $s0, $a0, $0     # assign to $s0 the value of n (expected in $a0)
       add $s1, $0, $0      # set the accumulator ($s1) to zero.
loop:  beq $s0, $0, end     # if n == 0 then goto end (we're done!).
       add $a0, $s0, $0     # Initialise $a0 with the value in $s0
                            # (prepares for function calling)
       jal square           # call the function "square"
       add $s1, $s1, $v0    # add the squared value to the accumulator
       addi $s0, $s0, -1    # decrement $s0 by 1 (i.e. n=n-1)
       j loop               # jump to loop!
end:   add $v0, $s1, $0     # assign to $v0 the accumulated sum
       jr $ra               # return to the caller function
```

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`:

```
                            # Prologue
          addi  $sp, $sp, -12  # Allocate space for 3 words in the stack
          sw $ra, 0($sp)       # Save the returning address on the stack
          sw $s0, 4($sp)       # Save the content of $s0 on the stack
          sw $s1, 8($sp)       # Save the content of $s1 on the stack


### put here the body of the answer to question 6.1 ###
```

```
                        # Epilogue
lw $ra, 0($sp)          # Restore initial value of $ra from the stack
lw $s0, 4($sp)          # Restore initial value of $s0 from the stack
lw $s1, 8($sp)          # Restore initial value of $s1 from the stack
addi  $sp, $sp, 12      # free allocated space on the stack (3 words)
jr $ra                  # return to the caller function
```

Note here that the **`$ra`** register is saved/restored in the prologue and epilogue of the "callee" `sumSquare` function even though it is a register that is normally saved by the "caller" function. This is due to the fact that `sumSquare` becomes a "caller" function with respect to the "callee" `square` function. In this case, the calling rules and conventions for the "caller" function must be applied.