What is your name?:  

There are two sections:

I. True/False:  .40 points; (20 questions, 2 points each)
II. Problems:  .60 points; (10 questions, 6 points each)

100 points total

This test is worth 15% of your final grade. This test is open book and open notes. You have 60 minutes.

I. True/False: (2 pts. each)

**1.** Certain sections of high-level language code could be rewritten in assembler in order to speed up the entire program.

**2.** Programs must always be compiled, assembled, then linked before the executable image can be created.

**3.** Recent architectures that have been developed can be either MIPS or CISC, depending on how they are going to be used.

**4.** Using $t0 instead of $a0 to pass an argument to a subprogram will generate an error.

**5.** Loading the immediate value 255 into a word only affects the low-order byte.

**6.** Registers can be referred to either by their mnemonic name or their number.

**7.** A MIPS program will still execute correctly if the label main: is omitted.

**8.** An immediate value for a character in an instruction can be specified either as the ASCII value or as the character in single quotes.

**9.** Indexed addressing is especially useful for manipulating arrays.

**10.** The multiply and divide instructions take 16 times as long to execute as the add and subtract instructions.

**11.** It is a bad idea to use $a0 to store the results of some computation and store it for an extended section of code.

**12.** The jr command can be used with many different registers.

**13.** The sbrk system call is a system break, stopping the execution of the program.

**14.** Stack frames in a recursive program must all be the same size.

**15.** Arithmetic operations using numbers of opposite signs never results in overflow.

**16.** The stack grows towards address 0.

**17.** In the example we saw in class, ASCII values are stored little-endian in MIPS.

**18.** As a program is running, it can change one of its instructions to be something else by adding or subtracting from the opcode for that instruction.

**19.** In a recursive program, every subprogram must store $ra on the stack.
20. Consider a Fortran program that stores a 2-dimensional array in *column-major* order. Two for-loops, one nested inside the other, are used to iterate through the array elements. The most efficient way to write this code is to have the *row* variable change in the outer loop, with the *column* variable changing in the inner loop.

**II. Problems:** (6 points each)
Where there are multiple steps, please circle your final answer. You must show your work for credit.

1. What exactly does the *jai* instruction do?
   
   1. Store the address of the next instruction in $yrq$ (2 pts. ea.)
   2. Load the jump destination address into the PC
   3. Transfer control to the instruction in the PC, starting that instruction.

2. Multiplication in SPIM is implemented by the *mult Rs, Rt* instruction, followed by *mflo Rd* to retrieve the result. It turns out that we can do the same thing with a single instruction: *mul Rd, Rs, Rt*. How is this possible?
   
   `mul` is a *macro instruction* that itself does:
   
   1. `mult Rs, Rt`
   2. `mflo Rd`

3. If you didn’t know the difference between a *branch* and a *jump* instruction, how could you find the answer by executing a program?

   Use PCSPIM to generate a log file, then look at the logfile to see if the branch and jump instructions are implemented using a relative offset or transfer to an absolute address.

   (branch)                                     (jump)

4. Consider the following instruction:
   
   Write this instruction in hexadecimal.
   
   `Rd` `Rs` `Rt` `Reg.#`
   
   and $t2,sa3,ss1`

   
   `-10 7 17`

   
   $
   \begin{array}{cccc}
   0 & 0 & 0 & 0 \\
   0 & 0 & 0 & 0 \\
   0 & 0 & 0 & 0 \\
   1 & 0 & 0 & 1 \\
   0 & 1 & 0 & 1 \\
   \end{array}$

   
   `OxF15024`
5. Rewrite the SPIM code shown below in Java or C. Use the high-level language syntax and idioms you would expect for the equivalent code.

```assembly
li $t0, 0
li $a0, 3
labell:
    addi $t2, $t2, 2
    addi $t0, $t0, 1
    blt $t0, $a0, labell
label2:
```

```java
for (int i = 0; i < 3; i++)
{
    t2 = t2 + 2; // or t2 += 2
}
```

6. What is the mechanism that allows us to create a jump-table using a list of program labels?

   The mechanism is the 2-pass assembler:
   1. Pass 1 computes the address for each instruction and substitutes an address for each label
   2. Pass 2 converts each instruction into its binary equivalent.

7. After running a program using PCSpim, the logfile is saved and examined. You notice that the instruction `bne $1, $0, -40` is translated into hex as:

   0x1420FF[6]

   Where did the rightmost two hex digits (f6) come from?

   The offset of -40 in binary is:

   +40 = 00000000000000000000000000000000

   -40 = 11111111111111111111111111111111

   The rightmost 2 bits aren't stored, since we know addresses must occur on an address that is a multiple of 4. This gives:

   1111111111111100

   = 0x1F6

   which is the valued stored.
8. Consider the code given below, which is part of the Trap Handler. What is the purpose of this code?

```
mfc0 $a0, $14
andi $a0, $a0, 0x3
     This extracts the rightmost 2 bits of the EPC, which will
     subsequently be checked to ensure these 2 bits are both 0,
     thus guaranteeing the address is on a word-boundary.
```

9. Consider the code given below. What does this code do?
(Don't give an explanation for each line, but rather give the one-line summary that would be appropriate to use as the one line of documentation describing these two lines in a program.)

```
xor $t1, $t2, $t3
bgez $t1, label3
     branch to label3 if $t2 and $t3 have the same sign.
     (neg or pos)
```

10. Complete the code given below to implement a subprogram that copies a string from oldstring into the memory location starting at newstring. You don't have to print anything or exit the program, simply write the code for the subprogram, call the subprogram that does the copy, and return from the subprogram.

```
.data
oldstring:
    .asciiz "This is the old string."
newstring:
    .space 80
    .text
main:
    # write the subprogram to copy from oldstring to newstring, and call it
    jal subprog;
    ...
    Subprog:
    la $t0, oldstring
    la $t1, newstring
    loop: lb $t2, 0($t0)
         sb $t2, 0($t1)
         addi $t0, $t0, 1
         addi $t1, $t1, 1
         bnez $t2, loop
    jr $ra
```