There are two sections:
   I. True/False ........................ 30 points; (15 questions, 2 points each)
   II. Problems .......................... 70 points; (10 questions, 7 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)

   T  F  1. The MIPS simulator we’ve used in class is different from an industrial implementation, because an industrial implementation has additional instructions that let you read and write from memory.

   T  F  2. The datapath diagram shown in our textbook is incomplete because it does not show the control lines from the control unit.

   T  F  3. Every location in the text segment is accessible from a single branch statement.

   T  F  4. In a recursive program $ra doesn’t always need to be stored on the stack.

   T  F  5. There is at least one MIPS assembler instruction that can directly store a result of an ALU calculation into memory.

   T  F  6. In addition to the 32 MIPS register file registers having mnemonic names such as $t3, the registers can be referred to by their number, starting from $1 up to and including $32.

   T  F  7. Usage conventions for register use (what registers are used for what purposes) are strictly enforced by the assembler.

   T  F  8. A carry-out at the most significant bit after an addition of two signed numbers always indicates overflow.

   T  F  9. Overflow will never occur if two signed numbers of different signs are added.

   T  F  10. If we only have one or two parameters to send to a non-recursive function, then we can use registers and don’t need to use the stack.

   T  F  11. A Load/Store architecture indicates the existence of memory in addition to having CPU registers.

   T  F  12. The MIPS pipeline has a separate instruction cache and data cache because this allows greater concurrency.

   T  F  13. The trap handler discussed in class could be made to be reentrant if it had a separate kernel stack.

   T  F  14. In the pipelined architecture discussed in class each instruction must perform an action at each stage of the pipeline in order to achieve maximum speedup.

   T  F  15. Code that works in non-pipelined mode is not compatible with pipelined code. The opposite is also true.
II. Problems: (7 points each)

Where there are multiple steps, please circle your final answer. You must show your work for credit.

1. Consider the following instruction:
   \texttt{addi \$a2, \$t3, -2}
   Write this instruction in hexadecimal.
   \begin{align*}
   2 &= \text{0010} \\
   -2 &= \ldots 1 \ 1 \ 1 \ 0 \\
   \text{Rt} &= \text{\$a2 is register \pm 6} \\
   \text{Rs} &= \text{\$t3 \ y = \pm 11} \\
   \text{Imm} &= \text{0x2166FFFE (4 pts.)}
   \end{align*}

2. When we take a high-level language while loop (e.g. while (x > 0) ) and implement it in MIPS, we reverse the order of the condition check. Why is this?
   
   It eliminates one of the branch statements.

3. Why is the load address (la) instruction implemented as a macro?
   
   Immediate mode for an instruction only has 16 bits for the immediate value. Addresses has 32 bits, so we must load on address in 2 parts, 16 bits at a time.

4. Suppose we want to print out bit 5 from register \$t0, where the right-most least-significant bit is bit 0. Write the code that most efficiently does this.
   
   \begin{verbatim}
   _8_7_6_5_4_3_2_1_0
   \end{verbatim}
   5 shifts

   \begin{verbatim}
   srl \$t0, \$t0, 5    #shift right by 5 places
   andi \$a0, \$t0, 1 #mask all bits except right-most one
   #then syscall
   Or if you interpreted this to mean print in place:
   andi \$a0, \$t0, 16 #16_{10} = 0x20
   #then syscall
   \end{verbatim}

CS 366 - Computer Architecture II, Midterm Exam #1, page 2 of 5
5. Why are the $k0 and $k1 registers reserved for a particular use?

They are used in the Trap Handler code to handle exceptions. The trap
handler can't use registers like any other code because those values must
be maintained in the caller. Similarly, it can't use the stack.
$k0$ and $k1$ are reserved for the Trap Handler's use for this reason.

6. The exception handler code has two conditions for returning. One of them adds 4 to the EPC, and the
other doesn't. Why are there two distinct cases instead of just one?

The code handles both Exceptions (caused internally to the program, e.g.
divide by 0) and Interrupts (caused externally to the program, e.g. timer
interrupt).

An interrupt necessitates an instruction restart, while an exception
needs to skip the offending statement, and so 4 is added to
the EPC.

7. After running a program using PCSpim, the logfile is saved and examined. You notice that the instruction

```
bltz $v0 0x5c
```

is translated into hex as:

```
0x04400017
```

Where did the rightmost two hex digits (17) come from?

- $0x5c$ is the PC offset, which in binary is:

```
011100
```

Since we know instructions always start on a 4-word boundary, we don't
need to store the 2 least significant bits. The system will
adjust when the value is used. This gives us an effective offset
for branching of $2^{18}$ rather than $2^{16}$.

```
0101 1100
```

right shifted by 2 is:

```
0001 0111
```

which in hex is:

```
0x 7
```
8. Rewrite the following MIPS code with its high-level language equivalent, making it as intuitive as possible from the point of view of the high-level language.

```mips
.data
label1: .word top, label2, label3, label4
prompt: .asciiz "Enter a number:"
result: .asciiz "Answer is:"

.text
main:
    li $s0, 253

.top:
    li $v0, 4       # display prompt
    la $a0, prompt
    syscall
    li $v0, 5       # Read int
    syscall
    li $t3, 3       # error check
    bgt $v0, $t3, top
    la $a1, label1  # jump-table address
    sll $t0, $v0, 2 # compensate for word addressing
    add $t1, $a1, $t0 # offset into jump table
    lw $t2, 0($t1)  # load value from jump table
    jr $t2          # goto that case

label2:
    b done

label3:
    srl $s0, $s0, 1
    b done

label4:
    srl $s0, $s0, 2

done:
    andi $s0, $s0, 1  # mask to leave only rightmost bit
    li $v0, 4         # string print
    la $a0, result
    syscall
    li $v0, 1         # print int result
    move $a0, $s0
    syscall

.exit:
    li $v0, 10        # exit
    syscall

print("Enter a number");
read n;
switch (n) {
    case 1: break;
    case 2: // shift right n by 1 place;
        n = n / 2;
        break;
    case 3: // shift right n by 2 places;
        n = n / 4;
        break;
    ?
}

print("Answer is: \n");
```

\( n = n \div 2; \)

\( \text{// mask all bits except rightmost} \)

\( \text{print("Answer is: \n" + n);} \)
9. The code shown at left below is a function that finds how many values in an array are evenly divisible by four. The address of the array is passed to the function using $a0, and the number of words in the array is passed to the function using $a1. Register $v0 is used to return the answer.

```
Div4:
    li $v0, 0  # clear answer
    li $t3, 3  # set bit mask
    b skip
loop:
    lw $t2, 0($a0)  # load all 1's of array
    addi $a0, $a0, 4  # set for word boundary for next time
    and $t0, $t2, $t3  # mask bits 0 & 1
    bnez $t0, skip
    addi $v0, $v0, 1  # increment counter
skip:
    addi $al, $a1, -1  # decrement loop counter
    bgez $al, loop
    jr $ra
```

a) Circle the lines in the code above at left that could cause problems in a pipelined architecture that has branch delays and load delays.

b) Now rewrite the code in the space at right above, changing the order of lines to allow the code to run correctly in a pipelined architecture. Draw an arrow from each old location of a line above at left to your new location of that line above at right. *Only use nop if there are no other alternatives.*

10. Show the pseudo-code for a recursive function that takes a string in an array in memory and copies to another array in reverse order.

```
array original
array copy
  $a0 = address of original
  $a1 = address of copy
check: if ([$a0]) != null { // check contents at $a0
    $a0 ++; // advance to next character
    push $ra;
    jal check; // recursive call
} else { // do not print the null character
    pop $ra; // return
    jr $ra
}

Copy ($a0) to ($a1) // copy character
  $a1 ++;
  pop $ra
  jr $ra
```

```
caller must store null at $a1
```