The problems below ask about VAX instructions, which were not yet covered in class. For information on these instructions see the VAX Macro and Instruction Set manual linked to the EE 4720 references page.

Problem 1:   The VAX `locc` instruction finds the first occurrence of a character in a string (see example below). The first operand specifies the character to find (A in the example), the second operand specifies the length of the string (in register `r2`), and the third operand specifies the address of the first character of the string (register `r3` below).

```
# Find first occurrence of 65 (ASCII A) in memory starting at
# address r3 and continuing for the next r2 characters.
locc #65, r2, (r3)
```

(*a*) Show how the sample instruction above is encoded. Include the name of each field and its value *for the example above, not for the general case. In the original assignment the third argument was shown as* `r3`, *not* (`r3`) *which is correct.*

(*b*) Provide an example of `locc` in which the encoded second and third operands each require more space than the example above. At least one of these operands should use a memory addressing mode that is not available in MIPS. Show the instruction in assembler and show its encoding.

For the problems below consider a MIPS implementation similar to the one illustrated below and a *DF-equivalent* VAX implementation. Like the MIPS implementation, the DF-equivalent VAX implementation can read two registers per cycle, write one register per cycle, perform one ALU operation per cycle, and one memory operation per cycle (not including fetch). The DF-equivalent VAX implementation may or may not be pipelined and regardless does not suffer any kind of penalty for the complexity and size of its control logic. Assume that the DF-equivalent VAX takes one cycle to fetch an instruction and one cycle to decode an instruction, regardless of the instruction's size or complexity.

Unlike MIPS the DF-equivalent VAX may be able to simultaneously use its ALU and memory port for the same instruction (in the illustrated MIPS implementation they would be for two different instructions). The 2-read, 1-write register restriction only applies to registers defined by the ISA. As with MIPS pipeline latches, the DF-equivalent VAX can read or write as many temporary registers per cycle that it needs.

When showing the execution of an instruction on the DF-equivalent VAX use something like a pipeline diagram and explain what's going on when things aren't clear. For example, here is how an `add` instruction might execute:

```
# Note: Destination is rightmost register (r3)
Cycle                   0  1  2  3  4  5  6
add 123(r1), (r2)+, r3  IF ID EX ME ME EX WB
                                    EX WB
sub                        IF ID          EX
Cycle 2: EX: 123 + r1
Cycle 3: ME  load (123+r1)
Cycle 4: ME: load (r2)
Cycle 4: EX: r2 + 4
Cycle 5: EX: add (123+r1) + (r2)
Cycle 5: WB: wb r2+4 to r2
Cycle 6: WB: WB sum to r3.
```

In the example above the `add` instruction can be said to have taken four cycles since that's how long the `sub` might have had to wait to execute (to avoid overlap).

Use the following MIPS implementation for comparison:
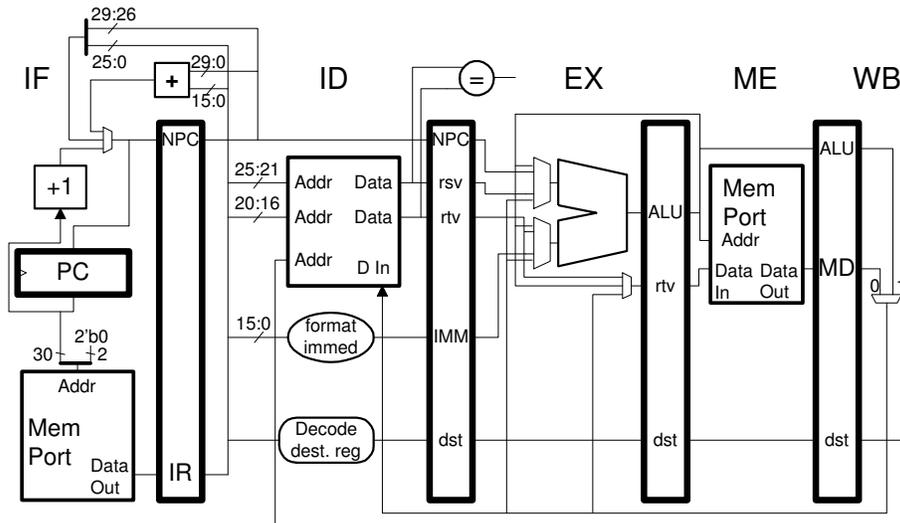
Problem 2: The MIPS `jal` instruction supports a procedure call by saving a return address in `r31`, other activities normally done on a procedure call, such as saving registers to the stack, must be performed using additional MIPS instructions. In contrast the VAX `calls` instruction not only saves a return address but also saves registers in the stack and performs other common activities.

MIPS and VAX examples are shown below in which the VAX code uses a `calls` instruction and the MIPS code performs a roughly equivalent operation. In particular, in both code samples three registers must be saved on the stack. (The `calls` instruction performs additional actions, but for this problem assume it does only what the MIPS code shows.)

(*a*) Show how the `calls` instruction would execute in the DF-equivalent VAX implementation. Note that the `calls` instruction reads the word at the beginning of the called routine to determine which registers to save.

(*b*) Is the DF-equivalent VAX implementation substantially faster on this instruction, about the same, or slower?

```
# VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX
        calls $0, myroutine

myroutine:
        .data
        .word 0x046
        xor ...

# MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS


        jal myroutine

myroutine:
        sw $r6, 0x18($fp)
        sw $r2, 0x8($fp)
        sw $r1, 0x4($fp)
        addi $fp, $sp, 0
        xor ...

myroutine:
# Cycle              0  1  2  3  4  5  6  7  8  9
  sw $r6, 0x18($fp)  IF ID EX ME WB
  sw $r2, 0x8($fp)      IF ID EX ME WB
  sw $r1, 0x4($fp)         IF ID EX ME WB
  addi $fp, $sp, 0            IF ID EX ME WB
  xor ...                       IF ID EX ME WB
```

Problem 3: The VAX `locc` instruction is another example of an instruction that would not be included in a RISC ISA because it could not be pipelined in any reasonable way. For this problem assume that implementations of character location can only read one byte at a time. (A fast implementation might read a word and check each position for the sought byte, but not in this problem.)

(*a*) What is the minimum amount of time that the DF-equivalent VAX implementation might take to execute `locc` with a length parameter equal to n? Show how the instruction would execute.

(*b*) The MIPS routine below performs the same operation (except for the r0 and r1 return values). In terms of n how long does it take to compute `locc`?

```
locc:
        # Call Values:
        #   a0: char: Character to find.
        #   a1: len:  Length of string.
        #   a2: addr: Address of first character of string.
        # Return Value:
        #   v0: 0 if character not found, 1 if found.
        #   Note: Other locc return values not computed.

        j START
        add $t1, $a1, $a2     # $t1: Stop address ( last char + 1 )
LOOP:
        beq $t0, $a0 FOUND
        addi $a2, $a2, 1
START:
        bne $a2, $t1, LOOP
        lb $t0, 0($a2)

        jr $ra
        addi $v0, $0, 0

FOUND:
        jr $ra
        addi $v0, $0, 1
```

(*c*) Which implementation has the speed advantage? Explain.

(*d*) Can instructions be added to MIPS consistent with RISC principles that would substantially improve its performance? If not, explain what gives `locc` an inherent advantage on CISC.