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.

Solution appears below. Note that immediate mode PC-addressing is used to specify the constant 65. In PC addressing the register field of the operand specifier is set to 15 (the VAX PC is register 15), this changes the meaning of some of the modes. For general addressing (the register field is not 15) mode 8 is autoincrement mode, for PC addressing mode 8 is immediate mode.

SOLUTION:

Instruction: locc #65, r2, (r3)
Syntax: locc char.rb, len.rw, addr.ab
Sections: opcode immediate_mode_op register_mode_op register_deferred_op

opcode -> 8 bits: 0x3a

immediate_mode_op -> operand_specifier immediate
operand_specifier -> mode(=immediate) reg(=PC) -> (4 bits) 0x8 (4 bits) 0xf
immediate -> (8 bits) 0x41

register_mode_op -> operand_specifier

register_deferred_op -> operand_specifier

-opcode- ---- 1st operand ------ 2nd op ---- 3rd op -
locc imm PC* 65 reg r2 reg-d r3
mode mode mode

0x3a 0x8 0xf 0x41 0x5 0x2 0x6 0x3 <- Encoded value.
7 0 7 4 3 0 7 4 3 0 <- Bit position.
(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.

The second operand now uses byte displacement deferred (shown as bdd below), and the third operand uses absolute addressing.

```assembly
.data
STR_ADDR: # Assume address is 0x1234
.asciiz "My string."
.text

locc #65, @B^8(r2), @#STR_ADDR
```

<table>
<thead>
<tr>
<th>opcode</th>
<th>1st operand</th>
<th>2nd op</th>
<th>3rd op</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>locc</td>
<td>imm 65</td>
<td>bdd r2</td>
<td>abs</td>
<td>32-bit</td>
</tr>
<tr>
<td>mode</td>
<td>mode</td>
<td>mode</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>0x3a</td>
<td>0x8</td>
<td>0xf</td>
<td>0x41</td>
<td></td>
</tr>
<tr>
<td>0xb</td>
<td>0x2</td>
<td>0x8</td>
<td>0x9</td>
<td></td>
</tr>
<tr>
<td>0xf</td>
<td>0x1234</td>
<td>0x2</td>
<td>0x31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
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:

```plaintext
# 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

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.

Solution appears below. An xor is shown following the calls to show how long the calls would take.

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

One cycle slower, because it has to check the mask to determine which registers to save.

```
# 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 MIPS MIPS MIPS MIPS MIPS MIPS MIPS

jal myroutine

SOLUTION:

# Cycle 0 1 2 3 4 5 6 7 8 9

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

jal myroutine

calls $0, myroutine

# IF ID ME EX ME ME ME

 RR RR EX EX EX WB
 RR RR

xor

# IF ID EX ...

Cycle 2: ME Load first word of myroutine, which specifies which regs to save.
    Word loaded into special 16-bit $m reg.
    Initial value of $m register is 0000 0000 0100 0110
    RR Retrieve fp register

Cycle 3: EX Set $r = clz($m) (Count number of leading zeros, $r will be 1)
    Compute: $addr = fp + ( $r << 2 ).
    RR Retrieve from register file: $rx = $r1
    At end of cycle set least-significant "1" bit of $m to 0.
    New value of $m: 0000 0000 0100 0000

Cycle 4: ME Store $rx ($r1) at address $addr
    EX, RR: Same operation as in cycle 3. (But $r will be 2)
    New value of $m: 0000 0000 0100 0100

Cycle 5: ME Store $rx ($r2) at address $addr
    EX: Same operation as in cycle 3. (But $r will be 6)
    RR: Read r6 and also $sp
    New value of $m: 0000 0000 0000 0000

Cycle 6: ME Store $rx ($r6) at address $addr
    EX Add $sum = $sp + 0

Cycle 7: WB Write $sum to register $fp
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.

The instructions appear below. Two cycles per character are needed because there is one comparison unit but two comparisons are needed: the character loaded and the character count. The worst-case time to find a character is \( 4 + 2n \) cycles.

SOLUTION

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>locc</td>
<td>#65, r2, (r3)</td>
<td>IF</td>
<td>ID</td>
<td>RR</td>
<td>CM</td>
<td>ME</td>
<td>CM</td>
<td>ME</td>
<td>CM</td>
<td>...</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EX</td>
<td>CM</td>
<td>EX</td>
<td>CM</td>
<td>EX</td>
<td>CM</td>
<td>EX</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EX</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 1: ID: $char = \text{lit. Assume literal addressing for character.}$
Cycle 2: RR: $\text{len} = \text{rx, assuming register addressing; \text{addr} = ry}$
Cycle 3: CM: Check if the $\text{len}$ is non-zero (if so proceed to X).
EX: Decrement $\text{len}$
Cycle 4: ME: Retrieve byte at $\text{addr}$.
CM: Check if the $\text{len}$ is non-zero (if so proceed to X).
EX: $\text{len} = \text{len} - 1$;
Cycle 5: CM: Check if byte equals $\text{char}$, if so proceed to X.
EX: $\text{addr} = \text{addr} + 1$
Cycle 6: ME: Retrieve byte at $\text{addr}$.
CM: Check if the $\text{len}$ is non-zero (if so proceed to X).
EX: $\text{len} = \text{len} - 1$;
Cycle X: WB: Write condition code with 1 if char found, 0 otherwise.
(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 \texttt{locc}?

\texttt{locc}:

\begin{verbatim}
# 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
\end{verbatim}

From the diagram below it can be seen that an iteration takes 8 cycles (cycle 6 to 14), and so the routine takes \( 4 + 8n \) cycles to find the character in the worst case (when the character is not in the string).

**SOLUTION:** Analyze the loop:

\begin{verbatim}
LOOP: beq $t0, $a0 FOUND    IF ID EX ME WB
       addi $a2, $a2, 1    IF ID EX ME WB
START: bne $a2, $t1, LOOP   IF ID ------> EX ME WB
       lb $t0, 0($a2)     IF ------> ID EX ME WB
# Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
LOOP: beq $t0, $a0 FOUND    IF ID ------> EX ME WB
       addi $a2, $a2, 1    IF ------> ID EX ME WB
START: bne $a2, $t1, LOOP   IF ID ------> EX ME WB
       lb $t0, 0($a2)     IF ------> ID EX ME WB
LOOP: beq $t0, $a0 FOUND    IF ...
# Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
\end{verbatim}
(c) Which implementation has the speed advantage? Explain.

Based on the given code fragments, the VAX. However the MIPS code can be re-written to reduce execution time on the given implementation. For example, the code below, an unrolled version of the code above, searches at the rate of \( \frac{10}{4} \) cycles per character.

```
lb $t0, 0($a2)
lb $t5, 1($a2)
lb $t6, 2($a2)

LOOP: beq $t0, $a0, FOUND  IF ID EX ME WB
lb $t7, 3($a2)  IF ID EX ME WB
beq $t5, $a0, FOUND  IF ID EX ME WB
addi $a2, $a2, 4  IF ID EX ME WB
beq $t6, $a0, FOUND  IF ID EX ME WB
lb $t0, 0($a2)  IF ID EX ME WB
beq $t7, $a0, FOUND  IF ID EX ME WB
lb $t5, 1($a2)  IF ID EX ME WB
bne $a2, $t1, LOOP  IF ID EX ME WB
lb $t6, 2($a2)  IF ID EX ME WB
```

(d) Can instructions be added to MIPS consistent with RISC principles that would substantially improve its performance? If not, explain what gives 10cc an inherent advantage on CISC.

Auto-increment addressing would save one instruction. This problem specifically disallows loading a word. If it were allowed an instruction could test each byte position for a match.