0) Compare zero-, one-, two-, and three-address machines by writing programs to compute
\[ X = \frac{A + B}{D - E \times B} \]
for each of the four machines. The instructions available for use are as follows:

<table>
<thead>
<tr>
<th>3 Address</th>
<th>2 Address</th>
<th>1 Address (Accumulator machine)</th>
<th>0 Address (Stack machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE (X ← Y)</td>
<td>MOVE (X ← Y)</td>
<td>LOAD M</td>
<td>PUSH M</td>
</tr>
<tr>
<td>ADD (X ← Y + Z)</td>
<td>ADD (X ← X + Y)</td>
<td>ADD M</td>
<td>ADD</td>
</tr>
<tr>
<td>SUB (X ← Y - Z)</td>
<td>SUB (X ← X - Y)</td>
<td>SUB M</td>
<td>SUB</td>
</tr>
<tr>
<td>MUL (X ← Y * Z)</td>
<td>MUL (X ← X * Y)</td>
<td>MUL M</td>
<td>MUL</td>
</tr>
<tr>
<td>DIV (X ← Y / Z)</td>
<td>DIV (X ← X / Y)</td>
<td>DIV M</td>
<td>DIV</td>
</tr>
</tbody>
</table>
1) How relocatable (i.e., can it be moved in memory) is the code in memory if direct addressing is used?

2) How many bits are needed to represent a direct address on a 32-bit machine?

3) From your programming experience, what range of Integer values would cover 90% of the constant Integer values used in your programs?

4) How many binary bits would you need to represent this range of values?

5) What determines how many bits are needed to represent a register in a machine language instruction?

6) Which of the following programming language constructs would be good candidates for using PC-relative addressing?
   a) conditional branch used when implementing loops
   b) calling a subprogram
   c) accessing a global variable
   d) accessing a local variable
<table>
<thead>
<tr>
<th>Type of Instruction</th>
<th>MIPS Assembly Language</th>
<th>Register Transfer Language Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Access (Load and Store)</td>
<td>lw $4, Mem</td>
<td>$4← [Mem]</td>
</tr>
<tr>
<td></td>
<td>sw $4, Mem</td>
<td>Mem←$4</td>
</tr>
<tr>
<td></td>
<td>lw $4, 16($3)</td>
<td>$4← [Mem at address in $3 + 16]</td>
</tr>
<tr>
<td></td>
<td>sw $4, Mem</td>
<td>[Mem at address in $3 + 16]←$4</td>
</tr>
<tr>
<td>Move</td>
<td>move $4, $2</td>
<td>$4← $2</td>
</tr>
<tr>
<td></td>
<td>li $4, 100</td>
<td>$4← 100</td>
</tr>
<tr>
<td>Load Address</td>
<td>la $5, mem</td>
<td>$4← load address of mem</td>
</tr>
<tr>
<td>Arithmetic Instruction (reg. operands only)</td>
<td>add $4, $2, $3</td>
<td>$4← $2 + $3</td>
</tr>
<tr>
<td></td>
<td>mul $10, $12, $8</td>
<td>$10← $12 * $8 (32-bit product)</td>
</tr>
<tr>
<td></td>
<td>sub $4, $2, $3</td>
<td>$4← $2 - $3</td>
</tr>
<tr>
<td>Arithmetic with Immediates (last operand must be an integer)</td>
<td>addi $4, $2, 100</td>
<td>$4← $2 + 100</td>
</tr>
<tr>
<td></td>
<td>mul $4, $2, 100</td>
<td>$4← $2 * 100 (32-bit product)</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>bgt $4, $2, LABEL</td>
<td>Branch to LABEL if $4 &gt; $2</td>
</tr>
<tr>
<td></td>
<td>(bge, blt, ble, beq, bne)</td>
<td></td>
</tr>
<tr>
<td>Unconditional Branch</td>
<td>j LABEL</td>
<td>Always Branch to LABEL</td>
</tr>
</tbody>
</table>

**MIPS Logical Instructions**

- and $4, $5, $6 $4←$5 (bit-wise AND) $6
- andi $4, $5, 0x5f $4←$5 (bit-wise AND) 5f₁₆
- or $4, $5, $6 $4←$5 (bit-wise OR) $6
- ori $4, $5, 0x5f $4←$5 (bit-wise OR) 5f₁₆
- xor $4, $5, $6 $4←$5 (bit-wise Exclusive-OR) $6
- xori $4, $5, 0x5f $4←$5 (bit-wise Exclusive-OR) 5f₁₆
- nor $4, $5, $6 $4←$5 (bit-wise NOR) $6
- not $4, $5 $4← NOT $5 #inverts all the bits

**MIPS Shift and Rotate Instructions**

- sll $4, $5, 3 $4←shift left $5 by 3 positions. Shift in zeros (only least significant 5-bits of immediate value are used to shift)
- sllv $4, $5, $6 Similar to sll, but least significant 5-bits of $6 determine the amount to shift.
- srl $4, $5, 3 $4← shift right $5 by 3 positions. Shift in zeros
- sr lv $4, $5, $6 Similar to srl, but least significant 5-bits of $6 determine the amount to shift.
- sra $4, $5, 3 $4←shift right $5 by 3 positions. Sign-extend (shift in sign bit)
- sralv $4, $5, $6 Similar to sra, but least significant 5-bits of $6 determine the amount to shift.
- rol $4, $5, 3 $4←rotate left $5 by 3 positions
- rol $4, $5, $6 Similar to above, but least significant 5-bits of $6 determine the amount to rotate.
- ror $4, $5, 3 $4←rotate right $5 by 3 positions
- ror $4, $5, $6 Similar to above, but least significant 5-bits of $6 determine the amount to rotate.
Fibonacci Sequence: 0 1 1 2 3 5 8 13 21
Position in Sequence: 0 1 2 3 4 5 6 7 8

A high-level language program to calculate the $n$\textsuperscript{th} fibonacci number would be:

\begin{verbatim}
temp2 = 0
temp3 = 1
for i = 2 to n do
    temp4 = temp2 + temp3
    temp2 = temp3
    temp3 = temp4
end for
result = temp4
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\textbf{HLL variables} & \textbf{Trace of Program (time $\rightarrow$)} & \textbf{MIPS registers} \\
\hline
\textbf{temp2} & 0 & 1 & 1 & 2 & 3 & 5 & 8 & $2$
\hline
\textbf{temp3} & 1 & 1 & 2 & 3 & 5 & 8 & 13 & $3$
\hline
\textbf{temp4} & 1 & 2 & 3 & 5 & 8 & 13 & 21 & $4$
\hline
\textbf{i} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & $6$
\hline
\end{tabular}
\end{table}

A complete assembly language MIPS program to calculate the $n$\textsuperscript{th} fibonacci number.

\begin{verbatim}
.data
n: .word 8 # variable in memory
result: .word 0 # variable in memory

.text
.globl main
main:
    li $2, 0
    li $3, 1
    lw $5, n # load "n" into $5
    li $6,
for_loop:
    bgt $6, $5, end_for
    add $4, $2, $3
    move $2, $3
    move $3, $4
    addi $6, $6, 1
    j for_loop
end_for:
    sw $4, result
    li $v0, 10 # system code for exit
    syscall
\end{verbatim}