Problem No. 1
(a) (i), (iii), (v) and (vi)
(b) (i) assign,
   (ii) loops,
   (iii) call/return,
   (iii) The answers are different because different HLL instructions represent different amounts of work. Even though “assign” HLL instructions are more frequent than “loops”, each “assign” instruction translates to significantly fewer machine instructions. Also, different machine instructions result in different no. of memory accesses, depending on how much data they need to read/write from/to memory, which explains the difference between answers to (ii) and (iii).
(c) (iv) Larger than both VAX and PDP-11

Problem No. 2
(a) Regularity, Orthogonality and Composability
(b) Complex solutions at the ISA level to directly implement high-level language constructs often turn out to be more trouble than they are worth. For example, consider the CALLS instruction in the VAX ISA. First, this instruction is tailored to a specific high-level language. Second, in order to be applicable to all the potential function call scenarios, this instruction is designed to be too general and often ends up doing a lot more work (for example, backing up all the registers) than is needed by a typical function call.
   When primitive operations are provided instead of complex instructions, the compiler writer can recompose the primitive operations to closely model the features needed by a particular language.
(c) Using both the register and memory addressing modes makes the compiler simpler and reduces the total code size. In contrast, MIPS ISA allows only the load/store instructions to access the memory, whereas the arithmetic instructions can operate only on the register operands. MIPS made this choice to simplify the processor design and to make pipelining easier.

Problem No. 3
(a)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Clock Cycles</th>
<th>Frequency</th>
<th>Effective CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>All ALU instructions</td>
<td>1.0</td>
<td>26.9% + 5.1% + 6.6% + 1.5% + 1.1% + 2.1% + 9.4% + 4.8% + 4.4% + 0.1% = 62%</td>
<td>1.0 * 0.62 = 0.62</td>
</tr>
<tr>
<td>Loads/Stores</td>
<td>2.5</td>
<td>20.1% + 5.1% = 25.2%</td>
<td>2.5 * 0.252 = 0.63</td>
</tr>
</tbody>
</table>
Conditional Branches

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Taken</td>
<td>11% * 75% = 8.25%</td>
</tr>
<tr>
<td></td>
<td>Not Taken</td>
<td>11% * 25% = 2.75%</td>
</tr>
<tr>
<td>Jumps</td>
<td>2.0</td>
<td>0.8% + 0.4% + 0.4% = 1.6%</td>
</tr>
</tbody>
</table>

Jumps = 0.8% + 0.4% + 0.4% = 1.6%

Total effective CPI = 0.62 + 0.63 + 0.2475 + 0.0275 + 0.032 = 1.557

Note: In the above calculations, “load imm” and “cond move” were counted as ALU instructions. Alternatively, if they were counted as “Loads/stores” and “Conditional Branches”, the solution will be considered correct due to the ambiguity regarding instruction category classification.

(b) Clock frequency = 3 GHz

Clock period = 1/Clock Frequency = 1/(3 * 10⁹) = 0.333*10⁻⁹ s = 0.333ns

Average CPI = 1.557

Since:

Execution time = Number of instructions * Clock period * Average CPI

Instruction executed per unit time = \[
\frac{\text{Number of instructions}}{\text{Execution time}} = \frac{1}{\text{Clock period} \times \text{Average CPI}}
\]

= 1 / (0.333 * 10⁻⁹ * 1.557)

= 1.927 * 10⁹ instructions per second

= 1.927 million instructions per millisecond

(c) Option 1: Reduce the Load/Store CPI from 2.5 cycles to 2 cycles

This will reduce the total CPI by (2.5 – 2) * 0.252 = 0.126 cycles

New CPI = 1.557 – 0.126 = 1.431

Instruction executed per second = 1 / (0.333 * 10⁻⁹ * 1.431) = 2.096 * 10⁹

Option 2: Increase the clock frequency from 3GHz to 3.3GHz

New clock period = 1/(3.3 * 10⁹) = 0.303ns

Instruction executed per second = 1 / (0.303 * 10⁻⁹ * 1.557) = 2.119 * 10⁹

Conclusion: Option 2 has the greater impact on performance

Problem No. 4

(a) This code represents a loop in which the loop counter is initialized to 0. In each iteration, the loop counter gets incremented and then compared with 27. The loop terminates, when the loop counter reaches a value of 27. Therefore the loop has 27 iterations.

Also, in each iteration, the loop counter is ANDed with “3”. This “AND” operation is equivalent to checking the 2 least significant bits in the loop counter. If both the bits are “0” (which happens in iterations 0, 4, 8, 12, 16, 20, 24), the result of the AND operation is “0”. In that case, the contents of R20 are incremented by 4. In all the other loop iterations (iterations 1, 2, 3, 5, 6, 7, 9, 10, 11, ……, 23, 25, 26), the contents of R20 are decremented by 6.

(b) The loop terminates when R18 reaches a value of 27. Therefore:

Final contents of R18 = 27
R20 starts with a value of 50. In 7 of the 27 iterations, the contents of R20 are incremented by 4; whereas in the remaining 20 iterations, the contents of R20 are decremented by 6. Therefore:

Final contents of R20 = 50 + (7*4) – (20*6) = -42

**Problem No. 5**

(a) Assembly code for the given C program is as follows:

```
ADDI R1, R0, #1               # R1 initialized with the initial value of loop counter (i = 0)
ADDI R8, R0, #200             # R8 keeps track of final value of loop counter
ADDI R2, R0, #3000            # R2 points to A[0]
ADDI R3, R0, #6000            # R3 points to B[0]
ADDI R4, R0, #9000            # R4 points to C[0]

LOOP:    LW  R5, #0(R3)         # R5 = B[i-1]
         LW  R6, #4(R4)          # R6 = C[i]
         SUB  R7, R5, R6        # R7 = B[i-1] - C[i]
         SW  R7, #4(R2)         # A[i] = contents of R7
         ADDI R2, R2, #4        # R2 points to A[i+1]
         ADDI R3, R3, #4        # R3 points to B[i]
         ADDI R4, R4, #4        # R4 points to C[i+1]
         ADDI R1, R1, #1        # i = i+1
         BNE  R1, R8, LOOP      # continue to loop if I < 200
```

(b) The assembly code has 5 instructions before the loop and 9 instructions within the loop. The loop is executed for 199 iterations. Therefore:

Total number of dynamic instructions = 5 + (199*9) = 1796 instructions

**Problem No. 6**

(a) The addressing modes for the three instructions are as follows:

i. Direct or Absolute

ii. Indexed

iii. Autodecrement

iv. Scaled

(b) Equivalent sequences of instructions using SIMPLE addressing modes are as follows:

i. LD  R6, 600(R31)
    ADD R3, R6

ii. ADD R4, R5
    LD  R7, #0(R4)
    ADD R6, R7

iii. SUBI R2, #d
    LD  R3, #0(R2)
    ADD R1, R3
iv. MULI R2, #d
    ADD R2, R3
    LD R4, #200(R2)
    ADD R1, R4

Note: The above instruction sequences assume that SIMPLE uses a 2-operand addressing format, which was not explicitly specified in the problem statement. A solution which uses 3-operand addressing format but achieves the same functionality would also be considered correct. Also, it was assumed that the source registers can be overwritten with new values as an after-effect of this instruction sequence (e.g., register R4 in ii). If these registers need to retain their original values, then extra instructions would need to be added to restore them.