Problem No. 1

(a) (i) and (iv)

(b) (i) assign and if,

(ii) call/return and loops,

(iii) The answers are different because different HLL instructions represent different amounts of work. Even though “assign” and “if” HLL instructions are far more frequent than “call/return” and “loops” in HLL, each “assign” and “if” HLL instruction translates to significantly fewer machine instructions.

(c) (i) Because VAX supports complex instructions and addressing modes, the instruction encoding in VAX is more efficient and compact as compared to RISC. As a result, the number of instructions in a RISC program is substantially higher than an equivalent VAX program.

(ii) In case of RISC, the arithmetic and logical instructions can only use register operands and no memory operands. In contrast, VAX allows any instruction to use any combination of register and memory operands. Consequently, a RISC compiler ends up allocating most of the program data in registers, where it is re-used by multiple arithmetic/logical instructions. Thus, RISC makes more efficient usage of general purpose registers, resulting in fewer data accessed from memory.

Problem No. 2

(a) Regularity, Orthogonality and Composability

(b) Consider the case of conditional branches. There are six different relations that can be used to specify the comparison between the operands of a conditional branch:

(i) equal to, (ii) not equal to, (iii) less than, (iv) less than or equal to, (v) greater than, (vi) greater than or equal to.

If the instruction set includes a direct implementation of all these six relations, then there is an obvious choice for the compiler to use, whenever it needs to generate the code for a conditional branch. However, if only a subset (say three or four) of the above conditions are supported in the ISA, then the job of the compiler becomes tougher. For example, the compiler must decide, which combination of the supported conditions would result in the cheapest implementation. This is an example of the violation of one vs. all principle.

(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</td>
<td>10.1% + 3.7% + 6.3% + 0.1% + 0.1% + 1.1% + 0.2% + 17.6% + 1.5% + 0.1% = 40.8%</td>
<td>1 * 0.408 = 0.408</td>
</tr>
<tr>
<td>Loads/Stores</td>
<td>3</td>
<td>30.3% + 4.3% = 34.6%</td>
<td>3 * 0.346 = 1.038</td>
</tr>
<tr>
<td>Conditional Branches</td>
<td></td>
<td>17.5%, 75% taken</td>
<td></td>
</tr>
<tr>
<td>Taken</td>
<td>3</td>
<td>17.5% * 75% = 13.1%</td>
<td>3 * 0.131 = 0.393</td>
</tr>
<tr>
<td>Not Taken</td>
<td>1</td>
<td>17.5% * 25% = 4.4%</td>
<td>1 * 0.044 = 0.044</td>
</tr>
<tr>
<td>Jumps</td>
<td>1.6</td>
<td>0.7% + 3.2% + 3.2% = 7.1%</td>
<td>1.6 * 0.071 = 0.114</td>
</tr>
</tbody>
</table>

Total effective CPI = 0.408 + 1.038 + 0.393 + 0.044 + 0.114 = 1.997

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 = 3.2 GHz

Clock period = 1/Clock Frequency = 1/(3.2 * 10^9) = 0.3125 * 10^-9 s = 0.3125 ns

Average CPI = 1.997

Since:

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

= \( \frac{1}{(0.3125 \times 10^{-9} \times 1.997)} \)

= 1.602 * 10^9 instructions per second

= 1.602 instructions per microsecond

(c) Option 1: Reduce the Load/Store CPI from 3 cycles to 2.4 cycles

This will reduce the total CPI by \((3 - 2.4) \times 0.346 = 0.2076\) cycles

New CPI = 1.997 – 0.2076 = 1.7894

Instruction executed per second = \( \frac{1}{(0.3125 \times 10^{-9} \times 1.7894)} \) = 1.788 * 10^9

Option 2: Increase the clock frequency from 3.2GHz to 3.5GHz

New clock period = \( \frac{1}{(3.5 \times 10^9)} \) = 0.2857 ns

Instruction executed per second = \( \frac{1}{(0.2857 \times 10^{-9} \times 1.997)} \) = 1.753 * 10^9

Conclusion: Option 1 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 by 3 and then compared with 24. The loop terminates, when the loop counter reaches a value of 24. Therefore the loop has $24/3 = 8$ iterations.

Also, in each iteration, the loop counter is ANDed with “2”. This “AND” operation is equivalent to checking the second least significant bit in the loop counter. If this bit is “0” (which happens in the 1st, 4th, 5th and 8th iterations), the result of the AND operation is “0”. In that case, the contents of R20 are decremented by 4. In the remaining iterations (2nd, 3rd, 6th and 7th), the contents of R20 are incremented by 6.

(b) The loop terminates when R18 reaches a value of 24. Therefore:

R20 starts with a value of 30. In 4 of the 8 iterations, the contents of R20 are decremented by 4; whereas, in the remaining 4 iterations, the contents of R20 are decremented by 6. Therefore:

Final contents of R20 = 30 + (4*6) – (4*4) = 38

**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] - 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. Autoincrement

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. `LD R3, #0(R2)`
    `ADD R1, R3`
    `ADDI R2, #d`

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 or register R2 in iv). If these registers need to retain their original values, then extra instructions would need to be added to restore them.