Assignment 4 Solutions

1.a \( a_0 = 8 * a_1 - 15 + a_2 * a_3 \)

    push 8
    push \( a_1 \)
    mult
    push 15
    sub
    push \( a_2 \)
    push \( a_3 \)
    mult
    add
    pop \( a_0 \)

Common mistakes:
- Forgetting to pop into \( a_0 \) at the end
- Unnecessary pops/pushes (the stack is arbitrarily large, so you can keep values there)

1.b Encoding scheme

There are 7 instructions so we need 7 different opcodes, which can be stored in 3 bits (\(2^3 = 8\)). Instructions can specify either one register or one immediate value. There are 4 registers (2 bits), but since immediates are 32 bits we need a 32-bit field to store them.

Example:

\[
\begin{array}{c|c}
\text{opcode} & 32\text{-bit immediate} \\
\hline
34 & 32 & 31 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{opcode} & \text{reg.} \\
\hline
34 & 32 & 31 & 2 & 1 & 0 \\
\end{array}
\]

Notice that this encoding wastes a lot of bits for non-immediate instructions, but changing this requires changing the ISA.

Each instruction is 35 bits long, so our 10-instruction program is 350 bits total.

Common mistakes:
- Not using a fixed instruction length (it specifies this in the problem statement, and this is RISC)
- Splitting the 32-bit value into two halves (this is not part of the ISA we gave you)
- Trying to encode the 32-bit value with a 5-bit field (5 bits means you can only specify \(2^5 = 32\) different values, not \(2^{32}\) values!)
- Including a separate "register" field; recall in MIPS that you can overlap fields and the processor will know what to do based on opcode
1.c Equivalent MIPS (not graded)

```
ori $a0, $0, 8
mul $a0, $a0, $a1
addi $a0, $a0, -15
mul $at, $a2, $a3
add $a0, $a0, $at
```

5 instructions x 32 bits = 160 bits total

1.d Adding a branch (not graded)

There are many different ways to add a branch instruction, but since this is a stack machine, it is natural to use the stack. For example, let's compare the top two values of the stack, and branch by a 32-bit offset if they are equal. We have space in our previous instruction encoding for one more opcode, so no change is necessary.

Example:
```
beq <label>
```

Semantics:
```
if (Stack[top] == Stack[top-1])
then PC = PC + offset
```

<table>
<thead>
<tr>
<th>34</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>32-bit signed offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. x86 to C and MIPS

C code:
```c
void foo(void) {
    int i;
    int sum = 0;

    for (i = 0; i < 100; i++) {
        sum = sum + i;
    }
}
```

MIPS code:
```mips
.text
.globl __start

__start:
    ori $t0, $0, 0       # t0 is sum (%ebx)
    ori $t1, $0, 0       # t1 is i   (%ecx)
    ori $t2, $0, 100     # t2 is constant 100
    loop:
        beq $t1, $t2, done # bge is a pseudo-instruction
        sll $0, 0, 0       # nop in branch delay slot
        add $t0, $t0, $t1   # sum = sum + i
        j loop
        addi $t1, $t1, 1    # i++ in branch delay slot
    done:
        # Exit
        addiu $v0, $0, 10   # Prepare to exit (system call 10)
        syscall
        # Exit
```

Common mistakes:
- 0x63 is a hex value! It is equivalent to 99 in decimal.
- Using <= 100 instead of < 100
- Using < 99 instead of <= 99
- Order of the comparison in `cmpl $0x63, %ecx`

3. x86 and gcc

Answers vary depending on machine/compiler.

The purpose of this exercise was to make you more familiar with the intricacies of x86. We also wanted you to see how changing optimization levels can greatly affect your output program. Compilers today are very good at reducing waste and improving performance. For example, if you compile the code from question 2 with -O3, gcc can figure out that the entire loop is statically deterministic, and replaces it with the correct value computed at compile time!

Because of these transformations, optimized assembly is difficult to read, so when you're debugging a program you usually want to use -O0. (But occasionally, there might be a bug in the optimizer itself, in which case you won't see it without optimizations...)