1 ISA Evaluation

You are given the following minimalist stack-based ISA:

4 registers, $a0 - $a3.

Instructions (R = any register, X = 32-bit immediate value):
push R ; puts the value of R onto the top of the stack
pop R ; pops the value on top of the stack into R
push X ; pushes X onto the top of the stack
add ; pops 2 values from the stack, adds them, and pushes the result
mult ; pops 2 32-bit values from the stack, multiplies them, and pushes the result ; (only lower 32-bits)
sub ; pops 2 values from the stack, subtracts them, and pushes the result ; (note that "push $a1; push $a2; sub;" will leave ($a1-$a2) on the stack, ; not ($a2-$a1))
div ; pops 2 values from the stack, divides them, and pushes the result ; (note that "push $a1; push $a2; div;" will leave ($a1/$a2) on the stack, ; not ($a2/$a1))

1.1

Write a program in this ISA to compute $a0 = 8\times a1 - 15 + a2\times a3$ (Assume normal order of operations: Multiply comes before add/subtract)

push 8
push $a1
mult
push 15
sub
push $a2
push $a3
mult
add
pop $a0

Common mistakes:
- Forgetting to pop into $a0$ at the end
- Unnecessary pops/pushes (the stack is arbitrarily large, so you can keep values there)
1.2

If each instruction is of a fixed length, what is the minimum number of bits required to encode each instruc-
tion? Briefly explain your encoding scheme. Be sure to follow the principles of RISC ISA design. How many
bits long is your whole program?

There are 7 instructions which will require 3 bits. Instructions can specify either one
register or one immediate value. There are 4 registers which will require 2 bits in an
instruction. However, we will need 32 bits for 32-bit immediates.

Example:

<table>
<thead>
<tr>
<th>opcode</th>
<th>32-bit immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>32 31 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>0</th>
<th>reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>32 31 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Each of our instructions are 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
  25 = 32 different values, not 232 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

1.3

Write the same program in MIPS. How many bits long is that program?

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

5 instructions * 32 bits = 160 bits total

1.4

Briefly describe how you would add a branch instruction to this ISA. Give its encoding and its semantics.

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.
beq <label>

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

-----------------------------------

<table>
<thead>
<tr>
<th>opcode</th>
<th>32-bit signed offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>0</td>
</tr>
</tbody>
</table>

2 x86

The following is x86 assembly. Write equivalent code in C and MIPS assembly. (It will probably be easier for you to write the C code first, and then the MIPS assembly equivalent.) Your MIPS code must be fully executable assembly, and you must run it in Spim. Include a screenshot of your entire Spim window (not just the registers). In your write-up, include a mapping between the MIPS registers you use and the x86 registers.

.text
.globl main

main:
   movl $0x0, %ebx  
   movl $0x0, %ecx  
   jmp check
loop:
   mov %ecx, %eax  
   add %eax, %ebx  
   addl $0x1, %ecx
check:
   cmpl $0x63, %ecx  
   jle loop
   ret

2.1 C Code:

void main() {
   int counter;  // %ecx
   int base = 0;  // %ebx
   int acc;  // %eax
   for (counter = 0; counter < 100; i++) {
      acc = counter;
      base += acc;
   }
}

It is admittedly ambiguous whether %eax returns or not. If so, the code would be
   return acc;

Multiple answers were accepted.
2.2 MIPS Code:

```assembly
.text
.globl __start
__start:
    ori $t0, $0, 0 # t0 is base (%ebx)
    ori $t1, $0, 0 # t1 is counter (%ecx)
    ori $t2, $0, 0 # t2 is acc (%eax)
    ori $t3, $0, 100 # t3 is constant 100
loop:
    beq $t1, $t2, done # bge is a pseudoinstruction
    sll $0, $0, 0 # nop in branch delay slot
    add $t2, $0, $t1 # acc = counter
    add $t0, $t0, $t2 # base += acc;
    j loop
    addi $t1, $t1, 1 # acc++ in branch delay slot
done:
    # Exit
    addiu $v0, $0, 10 # Prepare to exit (system call 10)
    syscall
```

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...)