**1a.** Most of the time, the x97 uses 2's complement representation for integers. On an 8-bit version of x97, what is the range of numbers that can be represented in 2's complement form?

1000 0000 (---) 0 111 [11]
biggest negative biggest positive
-128
127

**1b.** The x97 designers decided that just using 2's complement all of the time was boring, and added a new processor mode which uses a different representation for integers which they call "sign and magnitude". In this form, the most significant bit is simply used to indicate whether the integer is positive or negative (the "sign"); the other bits are used for the value of the number. On an 8-bit machine, what is the range of numbers that can be represented with "sign and magnitude" representation?

-127  $\leftarrow$  7 127 11111111 0111 1111 assume 1 => negative in MSB 0 => positive

1c. "Sign and magnitude" form, much like many other aspects of x97, has some problems as compared to 2's complement. What are they? Are there any ways in which "sign and magnitude" is better than 2's complement?

Problems:

-> 2 representations of zero

10000000 and 0000 0000

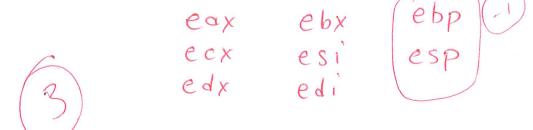
-> slightly smaller range of #s

-> harder to do addition, subtraction

Benefits:

-> easy to explain

2a. The x97 has a new instruction set, quite different than the x86. One example is found in the registers: instead of all the crazy names for general-purpose registers (that the Intel engineers never seemed to be able to remember), there are just a uniform set of registers named r1, r2, ..., r32. Actually, you can help out Intel here too; what are the names of the Intel x86 general-purpose registers?



2b. On x97, all instructions are register based, meaning that they only can have registers (like r1 through r32) as their operands; further, all operands are specified explicitly. Thus, something as simple as an add instruction looks like this: add register1, register2, register3. In this add instruction, the contents of register1 and register2 are added together; the result is put into register3. Given the following x86 add instruction, specifically add reg1, reg2, how would you rewrite it in an equivalent form on x97?

2c. Immediate values are generated a little differently on x97 too. On x86, a mov \$10, %eax would put the value 10 into register eax. On x97, you have a specific init instruction, which takes two operands: the first is the target register, and the second is an immediate value. Rewrite the mov \$10, %eax instruction in x97 assembly:

2d. On x97, there are a number of conditional jump instructions, which look like this: jXX reg1, reg2, target. For example, the jle will jump to the target address if reg1 is less than or equal to reg2. Other similar instructions exist for jump greater, greater-than-or-equal, jump-if-equal, etc. What is the x86 equivalent of the x97 jump instruction jle reg1, reg2, target?

jle target

jle target

first: does compare,
sets cond. codes (ccs)

second: does jump
based on CCs

**2e.** Moving values among registers is easy in x97; you just use the rmove instruction. The instruction takes two operands, e.g., rmove reg1, reg2 and moves the contents of reg1 into reg2. How is this similar to x86? How is it different?

similar x86 instruction: mov

e.g., mov % eax, %ebx

but x86 mov is move general

and can have src or dst

as a memory location too

2f. One last difference is found in how memory is accessed. On x97, there are two specific instructions to access memory: load and store. The load instruction has the following form: load register1, register2, which treats register1 as an address; it then loads the value at that address into register2. The store instruction is similar, but stores the contents of register1 into the memory location of register2. You now have to translate the following x86 instruction into x97 form: movl 20(reg1,reg2,1),reg3. What sequence of instructions could you use on x97 to perform the equivalent load from memory?

woul 20 (% eax, % ebx, 1), % ecx { 1) comple address:
eax + ebx + 20
2) fetch address,
put in ex

x97:
init V4, 20
add T1, T4
load T2, T4
load T4, T3

add T4, T3

add T4, T2

dst; V3

4a. Consider the following x86 code snippet:

```
foo:
         pushl %ebp
         movl %esp, %ebp
                                  y = ecx
0 = eqx (result)
x = edx
1F(x \le x) finish
         movl 12(%ebp), %ecx
         xorl %eax, %eax
         movl 8(%ebp), %edx
         cmpl %ecx, %edx
          jle .L3
.L5:
                                  edx = e4x + X 
x = x - 1
if(x > y)
         addl %edx, %eax
         decl %edx
          cmpl %ecx, %edx
          jg .L5
.L3:
          leave
         ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: only use symbolic variables x, y, i, and result, from the source code in your expressions below — do *not* use register names, as that wouldn't make any sense!)

**4b.** Now rewrite the x86 assembly from the previous problem (4a) into x97; if you need some new instructions, please feel free to define them, but keep consistent to the x97 philosophy!

y is in  $r_2$  //convention x is in  $r_1$  // convention foo! init  $r_3$ , 0 // for result init  $r_{31}$ , 0 // put 0 in ret. register jle  $r_1$ ,  $r_2$ , .L3 init  $r_4$ , 2 // ret 2 in  $r_4$ .L5: add  $r_1$ ,  $r_3$ ,  $r_3$ sub  $r_1$ ,  $r_4$ ,  $r_1$ Jg  $r_1$ ,  $r_2$ , .L5 .L3: