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 1111

biggest negative — biggest positive

−128 — 127

6

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 ⟷ 127

1 111 1111 — 0 111 1111

assume 1 ⇒ negative in MSB

0 ⇒ positive

6

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

1000 0000 and 0000 0000

→ slightly smaller range of #s

→ harder to do addition, subtraction

8

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?

eax ebx ebp -1
ecx esi esp
edx edi

3

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?

on x86:
add reg_1, reg_2 is $reg_2 = reg_2 + reg_1$
e.g., add %eax, %ebx

3

on x97:
(src) (src) (dst)
add reg_1, reg_2, reg_2 is the equivalent
e.g., add r_1, r_2, r_2

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:

x86:
mov $10, %eax

3

x97:
init r_1, 10

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`?

4

```
cmp  reg2, reg1
jle  target
```

two instruction sequence

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?

3

similar x86 instruction: mov

e.g., mov %eax, %ebx

but x86 mov is more 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?

4

x86:

```
movl  20(%eax, %ebx, 1), %ecx
      r1   r2        r3
```

1) compute address: eax + ebx + 20
2) fetch info@ address, put in ecx

x97:

```
init  r4, 20
add   r1, r4
add   r2, r4
load  r4, r3
```

uses r_4 for address calculation

src: r_1, r_2

dst: r_3

eax, ecx, edx — caller save

ebx, esi, edi — callee save

3a. One of the most complicated aspects of understanding x86 code is understanding how a procedure is called, and in particular how the stack is managed. Describe (in detail) what happens on x86 before, during, and after a procedure call on x86. What are the key steps? Which steps are optional? What is the state of the stack along the way? How are arguments accessed during the call?

(12)

before call:

1) save any caller-save regs (i.e., eax, ecx, edx) (if needed)
   e.g. push %eax

2) push arguments onto stack, from argN ... arg1 (if there are args)

3) call routine [needed]
   e.g. call 0x8000
   [implicit: pushes return addr onto stack]

4) save old base ptr, establish new bp [needed]
   { push %ebp
   { mov %esp, %ebp }

5) save caller-save registers (if needed)
   (i.e., ebx, esi, edi)
   e.g. push %ebx

6) make room on stack for locals (if needed)
   sub 0x10, %esp

7) run code of routine
   arg1: 0x8(%ebp) } how to access
   arg2: 0xC(%ebp) } args
   etc.

8) undo 6: add 0x10, %esp
9) undo 5: e.g., pop %ebx
10) undo 4: mov %ebp, %esp
    pop %ebp

11) return (pops ret. addr.)
12) undo 1: e.g., pop %eax

3b. On x97, the steps are a little different. One change is found in the way arguments are passed; specifically, arguments are always passed in registers, starting with argument1 in r1, argument2 in r2, etc., with up to 16 arguments. In addition, r31 is used to pass back a return value, and r32 stores the value of the return address. The rest of the registers are "callee save". How do these changes affect the call/return protocol? Is it more or less efficient, or about the same? Finally, is there still need for a stack? (why or why not?)

8

use r_{29} as sp,
r_{30} as bp

1) put args into registers
e.g. rmove r_{17}, r_1
init ~~20~~ r_2, 20

if more than 16 args, use stack

2) call routine (ret addr in r_{32})

3) use $r_x, r_{x+1}, \ldots, r_{16}$ as free regs (if not used for args)

use stack for overflow:
when too many args
when need to use callee save regs
etc.
(need some regs as bp, sp in that case, e.g. r_{29}, r_{30})

also:
have to be careful w/ r32
call w/in call will lose ret. addr!

4a. Consider the following x86 code snippet:

```
foo:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%ecx
        xorl %eax,%eax
        movl 8(%ebp),%edx
        cmpl %ecx,%edx
        jle .L3
.L5:
        addl %edx,%eax
        decl %edx
        cmpl %ecx,%edx
        jg .L5
.L3:
        leave
        ret
```

y ⇒ ecx
0 ⇒ eax (result)
x ⇒ edx
if (x ≤ y) finish

edx = eax + x
x = x - 1
if (x > y)

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!)

8(ebp) c(ebp)

```
int foo(int x, int y)
{
  int i, result=0;

  for (i=__0__; ____x > y____; __x--__) {

    result += x ;

  }

  return result;
}
```

10

or

```
for(i = x ; i > y ; i--) {
    result += i ;
}
```

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!

(10)

```
                y is in r2      // convention
                x is in r1      // convention

foo:    init    r3, 0           // for result
        init    r31, 0          // put 0 in ret. register

        jle     r1, r2, .L3
        init    r4, 1           // put 1 in r4
.L5:
        add     r1, r3, r3      // result += x

        sub     r1, r4, r1      // x --

        jg      r1, r2, .L5

.L3:
        ret
```

5a. In the following question, we'll do a cache lookup on a x97 machine. As it turns out, the Intel engineers left all the caching stuff alone; thus it works just the same as before. Assume the following is true:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 12 bits wide.
- The cache is 4-way set associative, with a 2-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

4-way Set Associative Cache																
Index	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1
0	29	0	34	29	87	0	39	AE	7D	1	68	F2	8B	1	64	38
1	F3	1	0D	8F	3D	1	0C	3A	4A	1	A4	DB	D9	1	A5	3C
2	A7	1	E2	04	AB	1	D2	04	E3	0	3C	A4	01	0	EE	05
3	3B	0	AC	1F	E0	0	B5	70	3B	1	66	95	37	1	49	F3
4	80	1	60	35	2B	0	19	57	49	1	8D	0E	00	0	70	AB
5	EA	1	B4	17	CC	1	67	DB	8A	0	DE	AA	18	1	2C	D3
6	1C	0	3F	A4	01	0	3A	C1	F0	0	20	13	7F	1	DF	05
7	0F	0	00	FF	AF	1	B1	5F	99	0	AC	96	3A	1	22	79

5a1. The box below shows the format of a physical address. Indicate (in the diagram) the fields that are used to determine the following: *O* (the block offset within the cache line), *I* (the cache index), and *T* (the cache tag).

11	10	9	8	7	6	5	4	3	2	1	0
T	T	T	T	T	T	T	T	I	I	I	O

5a2. Now, for the given physical address **0x3B6**, first write it in binary form:

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	0	1	1	0

5a3. Finally, fill in the following table with the correct values for the offset, index, tag, whether a cache hit or miss occurred, and if a hit, what value was returned, when address **0x3B6** was accessed.

Parameter	Value
Cache Offset (O)	0x 0
Cache Index (I)	0x 3
Cache Tag (T)	0x 3B
Cache Hit? (Y/N)	Y
Cache Byte returned	0x 66

5b. One last thing to do on this exam is to show your bosses that you really understand caches. The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S (# of sets)	t (tag bits)	s (index bits)	b (block offset bits)
1.	32	1024	4	4	1024/16 → 64	24	6	2
2.	32	1024	4	256	1	30	0	2
3.	32	1024	8	1	1024/8 → 128	22	7	3
4.	32	1024	8	128	1	29	0	3
5.	32	1024	32	1	1024/32 → 32	22	5	5
6.	32	1024	32	4	8	24	3	5

10