CS-354 Midterm (Fall 2012)

The x97 Processor

Please Read All Questions Carefully!

There are twelve (12) total numbered pages.

Please put your FULL NAME (mandatory) on THIS page only.

Name: _____

Grading Page

	Points	Total Possible
Q1		20
Q2		20
Q3		20
Q4		20
Q5		20
Total		100

In this exam, you'll be helping with the construction of a new machine known as the x97. After years of sticking with the x86 product line, and after a few too many drinks, the engineers at Intel decided to build a completely different processor; thus x97 was born.

Unfortunately, with this new processor (and its new instruction set), no existing programs seem to run anymore! Thus, Intel has hired you to help them with this dire situation. I guess you must be pretty good for Intel to bet the entire future of their company on your success! Alternately, you are related to the CEO. Either way, their future is in your hands.

Please remember to read all questions carefully! And good luck.

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?

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?

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?

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`?

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?

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?

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?

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?)

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
```

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!)

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

  for (i=________; _____________________________; _________) {

    _______________;

  }

  return result;
}
```

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!

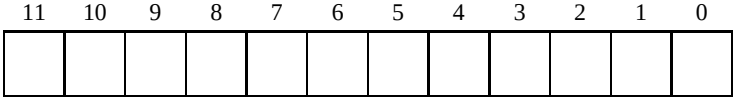
5a. In the following question, we'll do a cache lookup on a x97 machine. As it turns out, the Intel engineers left all the caching stuff alone; thus it works just the same as before. Assume the following is true:

- 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.

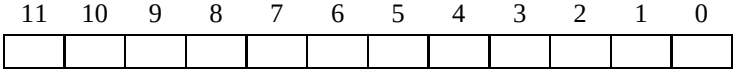
In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

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).



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



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
Cache Index (I)	0x
Cache Tag (T)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

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	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				