CS-354 Final (Spring '17 @ Epic)
Fun With Just One

Please Read All Questions Carefully!

There are 15 total numbered pages.

Please put your FULL NAME on THIS page only.

Name: ______________________________

This exam is inspired by the number "one", you know, that first number after zero and before all the rest of them? It's a good, solid number, and we all know that being "number one" is pretty nice – it means you won!

To be "number one" on this exam, you'll have to answer some questions about computer systems as it relates to the number one. For example, the first set of questions are based on a fictional instruction set with only **one** instruction; pretty cool, huh? After that, you get a bunch of C **one liners** that implement some interesting function. And so on.

For each answer, be as clear and concise as you can; however, you are allowed to write **more than one** sentence, so please do so if need be. And if you remember only one thing from this exam, let it be this: the exam has fifteen pages and five questions (not just **one** and **one**, alas).

Score: ____

1. **Q1. The single-instruction instruction set.**

   When we learned about x86, we learned one new instruction after the other. Add, subtract, multiply, divide, compare, jump, call, return, move, etc. So many instructions!

   However, it turns out that you can build a fully functional computer system with many fewer instructions; in fact, you can do it with just one. In this question, we explore one such instruction: **subleq**.

   The subleq instruction (**su**btract and **b**ranch if **l**ess than or **eq**ual to zero) subtracts the contents at address **a** from the contents at address **b**, stores the result at address **b**, and then, if the result is not positive, transfers control to address **c**; if the result is positive, execution proceeds to the next instruction in sequence. In code form:

   ```
   subleq a, b, c; Mem[b]=Mem[b]-Mem[a]; if (Mem[b]≤0) goto c;
   ```

   To show you can use this single instruction as a replacement for many others, your task is to write the assembly sequence using only subleq instructions that implements various x86-like instructions.

   To make your life a little easier, you can use the term **zero** in place of a or b. This is a memory location that has been initialized to the value zero. However, if you do use zero as the second operand (i.e., as b), the result of the subtraction overwrites the zero.

   (a) **JMP c**. Write one or more subleq instructions to implement an unconditional jump to address c.

   (b) **ADD a, b**. Write one/more subleq's to implement the addition of a and b, with the result placed in the memory location b.

(c) **MOV a, b**. Write one/more subleq's to implement the copy of the contents at memory location a to memory location b.

(d) **CMP 0, b; JEQ c**. Write one/more subleq's to implement a conditional jump to target c if (and only if) b is equal to zero.

We now discuss the single-instruction instruction set more qualitatively.

(e) What do you think the biggest advantages of the single-instruction instruction set are? List as many positives as you can think of.

(f) What are the biggest disadvantages? Again, list as many as you can.

(g) Are there any instructions you think would be hard to implement with subleq? Explain.

2. **Q2. C one-liners.**

   In the following question, we examine C code that can be expressed as a single line. For each of the questions, please describe what the C code is doing. Be as precise as you can.

   Hint: Sometimes, to figure out some code, it is useful to work through some examples to see what the code is doing. Feel free to use the backs of pages of this exam to do so.

   (a) `void cp(char *s, char *d) {while (*d++=*s++);}`

   (b) `void doit(int status) {if (status=1) LaunchNukes();}`

   (c) `int conv(char c) {return c - '0';}`

   (d) `int cntdig(int n) {int c=0; while (n!=0) {n /= 10; ++c;} return c;}`

(e) `int bits(int n) {int b=0; while(n) { n=n&(n-1); b++; } return b;}`

(f) `float a(int*n,int L){int i=0;float s=0.0;for(;i<L;++i){s+=n[i];}return s/L;}`

(g) `void swp(int *a, int *b) {*b=*a-*b; *a=*a-*b; *b=*a+*b;}`

(h) `int gcd(int a, int b) { while(b>0) { int t = a%b; a=b; b=t; } return a;}`

(i) `int haszero(unsigned int x) {return ((x-0x01010101)&(~x)&0x80808080)!=0;}`

(j) `void cntd(int x) {while(x-->0){printf("%d\n",x);}}`

3. **Q3. x86 in a single byte or two.**

   As you know now (from Homework 5), x86 assembly instructions are transformed into binary "opcodes" that the CPU can directly understand and execute. In this question, you get to perform a reverse disassembly of various opcodes, writing the assembly form of them (much like objdump does when used as a disassembler).

   To help you with this task (or rather, make it possible), here is a table of instructions and their bit encodings:

Instruction	**What it does**	**Binary Opcode**
AAA	ASCII Adjust after Addition	0011 0111
ADC	ADD with Carry (register1 to register2)	0001 0001 : 11 reg1 reg2
CMC	Complement Carry Flag	1111 0101
DEC	Decrement by 1 (register)	1111 1111 : 11 001 reg
INC	Increment by 1 (register, alternate encoding)	0100 0 reg
JMP	Unconditional Jump (short)	1110 1011 : 8-bit displacement
MOV	Move Data (register1 to register2)	1000 1001 : 11 reg1 reg2
NOP	No Operation	1001 0000
OR	Logical Inclusive OR (immediate to EAX)	0000 1101 : immediate data (4 bytes)
POP	Pop a Word from the Stack (register)	0101 1 reg

   To assist further, here are the register encodings, in binary:

eax	000
ecx	001
edx	010
ebx	011
esp	100
ebp	101
esi	110
edi	111

   Now, for the following sequence of bytes, disassemble each, writing the x86 assembly of what is represented by the byte sequence. Each sequence is shown in **hexadecimal form**. Note: **if an opcode does not exist, write down "bad opcode"**.

   (Question starts on next page)

(a) 40414243

(b) ebffffc5

(c) 37f54559

(d) 0d03020100

(e) ffeeddcc

We now discuss binary opcodes and their encoding more qualitatively.

(g) What makes disassembling x86 challenging? Could you change the encoding to make disassembly easier?

(h) x86 encoding tries hard to use short sequences of bytes to represent popular instructions. Why do you think they do this?

(i) More modern systems use more registers than x86 (e.g., 32 instead of 8). If x86 adds more registers, what would have to change about its instruction encodings?

4. **Q4. Single line truths about OS virtualization.**

   In the last part of class, we learned a little bit about how operating systems work, particularly focusing on CPU and memory virtualization. In this question, we explore single statements about such virtualization; your job is to comment whether each single statement is **true** or **false**, and then **(IMPORTANT!) explain why.**

   (a) The operating system requires an interrupt timer to go off every once in a while in order to virtualize the CPU.

   (b) When virtualizing memory, the system translates every virtual address that a program generates, except for pointer dereferences.

   (c) Using a simple linear (array-based) page table allows for very fast translations from virtual to physical addresses.

(d) A translation-lookaside buffer (TLB) is not needed for correct and fast virtualization of memory.

(e) The primary benefit of a multi-level page table is that it enables efficient and correct translation from virtual to physical addresses.

5. **Q5. I/O devices and performance estimations.**

   Our final question in this singular exam focuses on I/O devices, namely hard-disk drives (HDDs) and solid-state storage devices (SSDs).

   For the question, you should estimate how long a single I/O request will take assuming various parameters (as stated in the question). Show your work and explain your answers. For all answers, *approximations* are OK, especially if the exact answer is very close to your approximation.

   (a) Assume you have an HDD with the following parameters: 10,000 RPM, maximum seek is 21 milliseconds, and transfer rate is 100 MB/s. How long will a 4KB read take assuming *worst case* rotation and seek?

   (b) Assume you have an HDD with the following parameters: 10,000 RPM, maximum seek is 21 milliseconds, and transfer rate is 100 MB/s. How long will a 4KB read take assuming *average case* rotation and seek? (write down any assumptions you make)

   (c) Assume you have an HDD with the following parameters: 10,000 RPM, maximum seek is 21 milliseconds, and transfer rate is 100 MB/s. How long will a **4MB** read take assuming *average case* rotation and seek? (write down any assumptions you make)

(d) Assume you have an SSD with the following parameters: a 4KB page size, and a 4 MB block size. Further assume the following operation latencies: 10 microsecond read, 100 microsecond program, and 1 millisecond (1000 microsecond) erase time. How long does it take to read 40KB of data?

(e) Assuming the same SSD parameters as above, how long does it take to write 40KB (assuming that there are erased blocks already available?)

(f) Assuming the same SSD parameters as above, how long does it take to write 40KB (assuming that there are **NO** erased blocks already available?)