Name: ____________________________

Directions: Work only on this sheet (on both sides, if needed); do not turn in any supplementary sheets of paper. There is actually plenty of room for your answers, as long as you organize yourself BEFORE starting writing. In order to get full credit, SHOW YOUR WORK.

1. (10) Suppose we are storing signed integers using "8s complement arithmetic." In other words, to store $\pm n$, we would "wind forward or backward" from 000 n times, on a 3-digit "odometer" having digits 0-7. Find the representation of -12.

2. (10) Execution of a **call** instruction will change the contents of two Intel registers: _____ and _____.

3. (15) The following code will put 0s in alternate elements of a word array, starting at the element pointed to by EAX. The number of elements to be zeroed is in EBX. (EAX and EBX are initialized in previous code, not shown here.) The code does change EAX and EBX as it executes. Fill in the blanks.

```
top:  movl ____, ____
      addl ____, ____
      decl ____
      jnz top
```

4. (10) Consider the C **struct**

```
struct zs {
   int i;
   char a,b;
}
```

Suppose we wish to have the equivalent of an array of 50 elements of type **zs** in the **.data** segment of an assembly language program, starting at the label **zs50**. We do not care what the initial values in the array are, and we will NOT align on word boundaries. Show TWO ways to do allocate the needed space.

5. (10) DDD will display an entire array **x** in a C program if we move the mouse pointer to any instance of **x** in the source code window. But if we try the same thing on an array **y** in an assembly language program, DDD will only display the first element of the array. State clearly and precisely the cause of this disparity.

6. (10) Consider the instruction

```
movl $w, %ecx
```

Suppose the offset within the **.data** segment of **w** is 4096. Give the exact machine code generated by the assembler, in hex, for the above instruction. (Note: $4096 = 16^3$.)

7. (15) Suppose we have a word in the **.data** segment with the label **g**, and have code which places some value in EAX. We want to place 1 in ECX if the product of the two quantities is greater than or equal to 0x20000, or put 0 in ECX if not. Fill in the blanks:

```
     imull \_\_\_\_\_
     cmpl \_\_\_\_\_, \_\_\_\_\_
     jge grrrrr1
     movl \_\_\_\_\_, %ecx
     jmp grrrrr2
grrrrr1:
     movl \_\_\_\_\_, %ecx
grrrrr2: ...
```

8. Here is the complete assembly and machine code for a certain program:

```
1                    .text
2                    .globl _start
3 0000 89E0          _start: movl %esp, %eax
4 0002 C7000800         top: movl $8, (%eax)
5      0000
6 0008 83C004                addl $4, %eax
7 000b EBF5                  jmp top
```

Suppose the **.text** segment begins at location 28000, the **.data** segment begins at location 32800, and that c(ESP) = 80000. (All addresses in this problem are decimal unless otherwise indicated. No guarantee that all this information is needed.)

(a) (10) What will c(EIP) be near the end of the execution of the instruction in line 7?

(b) (10) Suppose this program is run with no virtual memory access protections, i.e. we can do any kind of access to any spot in memory, and suppose we have 4G of memory, i.e. the full address space. What will be the temporally-last value which we can be sure of for c(EAX)?

Solutions:

1. Wind backwards 12 times from 000, getting 777 for -1, 776 for -2, etc., to get 764 for -12. **OR:** First find +12, which is 015. Take the "7s complement," yielding 763, and add 1, yielding 764.

2. EIP, ESP

3.

```
top:
   movl $0, (%eax)
   addl $8, %eax
   decl %ebx
   jnz top
```

4.

One way:

```
.space 300
```

Another way:

```
.rept 50
.long 0
.byte 0
.byte 0
.endr
```

5. There are no data types, such as arrays, at the hardware level, and thus not at the assembler level. The assembler has no way of knowing that you intend something to be an array, so even with `--gstabs` the assembler cannot put such information in the symbol table. There is no problem in C, though, since the programmer specifies an array.

6. Hex b900100000.

7.

```
   imull g
   cmpl $0x20000, %eax
   jge grrrrr1
   movl $0, %ecx
   jmp grrrrr2
grrrrr1:
   movl $1, %ecx
```

8.a The instruction is a jump to 28000+0002 = 28002, so the circuitry for the instruction will put this value in EIP.

8.b The value in EAX will keep increasing by 4 until it gets to $2^{32} - 4$, after which the next increase will wrap around to 0. Eventually c(EAX) reaches 28000, the start of the **.text** segment. On the next iteration, c(EAX) = 28004, pointed to bytes within the instruction on line 4. The MOV instruction on that line will thus overwrite itself, and thus that instruction will never be executed again. The value in EAX will reach 28008.