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**.

All code is assumed to be run on our Linux PCs in CSIF.

1. In each case below, give an instruction that pops one word from the stack, placing the popped value in the specified place.

(a) (5) The popped value goes into EAX.

(b) (5) The popped value goes into the memory word pointed to by EAX.

(c) (5) The popped value is not written to any register or memory location.

2. Consider the machine code for our source file **Total.s** analyzed in Section 3 of our PLN unit on subroutines (and the supplement).

(a) (10) Suppose the JNZ instruction had accidentally been written **jns done** (note: two errors). What machine code would have been generated?

(b) (10) What value will be in ECX after the first execution of **addl $4, %ecx**?

3. (5) By inspecting p.7 of our PLN unit on subroutines, deduce the general machine code format for an immediate mode PUSH instruction. Use notation like that in our PLN unit on machine language.

4. (25) Suppose we have an I/O device that deposits data in a port at 0x8888. The actual data is a 4-bit unsigned integer, placed in bits 7-4 of the port. Bit 2 of the port serves as a Data Ready flag, with 1 meaning that data has arrived. The programmer is responsible for resetting that bit to 0 after fetching the data from the port.

Write a FULL assembly language program to handle this device. It will loop around, watching for data. Each time data arrives, the program will print it to the screen, by calling **printf()** with **%d** format. The code is assumed to run constantly, i.e. it is a "dedicated" program.

DO NOT USE MORE THAN 25 in^2 OF SPACE FOR THIS PROBLEM. MAKE SURE TO WORK OUT YOUR CODE ON SCRATCH PAPER FIRST, AND THEN COPY IT NEATLY AND LEGIBLY TO YOUR EXAM SHEET. (Remember, turn in only your exam sheet, with no supplementary sheets of paper.)

5. (10) When an interrupt occurs, some state information of the interrupted program is pushed onto the stack during a Step D. The information is later restored during a (i) Step A, (ii) Step B, (iii) Step C, (iv) Step D, (v) a combination of two or more of the above, or (vi) none of the above.

6. (10) Suppose someone looks at our code **TryAddOne.c** and **AddOne.s** on p.11 of our PLN unit on subroutines and claims to have a shorter way to do it: They remove lines 25, 30 and 35, and replace line 32 by **incl 4(%esp)**. Then this will (i) cause a seg fault, (ii) not cause a seg fault but will cause an incorrect value of **x** to be printed out, (iii) result in a link-time error, (iv) result in an assembly-time error, or (v) run correctly.

7. Consider the function **g()** in our supplement to the subroutines PLN unit. Suppose two of the lines were changed to

```
int *g(int u)
```

and

```
return &v;
```

(a) (10) Show code which would likely be generated by the compiler for the **return** statement.

(b) (5) Why would C code like this be dangerous?

Solutions:

1.

```
popl %eax
popl (%eax)
add $4, %esp
```

2.a 0x7900 (op code 79, jump distance 0)

2.b The program initializes ECX to the address of **x**, which is at the beginning of the **.text** segment and thus is at 0x08049094. So, after the first execution of the instruction, c(ECX) is 0x08049098.

3. The instruction **push $x+4** has its operand specified in immediate mode, so use this one. The value of the operand (at assembly time) is 0 +4 = 4, i.e. 0x00000004 is 32-bit form. Due to little-endianness, that shows up as 04000000 in the machine code listing, 6804000000. Thus the op code is 68, and the general format at 68IMM4.

4.

```
.globl _start
.text
_start
     movl $0, %ecx  # make sure high bits are 0s
lp:
     inb $0x8888, %cl  # read port
     movl %cl, %bl  # make copy, as will destroy original
     andb $0x04, %bl  # test Bit 2
     jz lp
     shrb $4, %cl  # move value to lower bits
     # print
     pushl %ecx
     pushl $fmt
     call printf
     add $8, %esp
     # reset Bit 2
     andb $0xfb, %bl
     outb %bl,  $0x8888
     jmp lp  # program runs contiuously
fmt:
     .string "%d\n"
```

5. The restoration is done by an IRET instruction. The action of any instruction occurs during Step C, so the answer is (iii).

6. The instruction will increment the value of **&x**, which had been placed on the stack prior to the call. So, the wrong place in memory will be incremented, and the answer is (ii).

7.a As noted in the PLN, the compiler likes to use the LEA instruction to calculate addresses, especially of stack items. The C convention for Intel machines is that return values are passed via EAX. So, for full credit:

```
leal -4(%ebp), %eax
leave
ret
```

Note the need for the LEAVE instruction.

The following would not be likely as the compiler's choice, but would work and gets part credit:

```
movl %ebp, %eax
sub $4, %eax
leave
ret
```

This gets a little less credit:

```
movl %esp, %eax
leave
ret
```

The problem here is that we are not guaranteed that ESP still points to **v**, because the call to **getchain** may change it.

7.b If the caller plans to use **&v** but does not do so before another subroutine call or other usage of the stack, **v** may get overwritten.

Some people answered “Then **v** would be out of scope” or something similar. This is wrong. Remember, there is no such thing as variable type or scope at the machine level. It’s only in our minds.