

## CS 110 Practice Final Examination 2 Solutions

---

**1**

a) Rewrite the Pascal subroutine using LINK, UNLINK and RTD

```

PSUB    LINK        A6, #0

        ; <<Subroutine body goes in here>>

        UNLK        A6
        RTD         #8
    
```

b) Rewrite the preceding code (caller and routine) using C calling conventions and making use of LINK, UNLK, and RTS:

```

CSUB    LINK        A6, #0

        ; <<Subroutine body goes in here>>

        UNLK        A6
        RTS

; Main Program
PEA     RVAR(A5)           ; Push parameters in
MOVE.L  VAR(A5), -(A7)     ; opposite order
JSR     CSUB
ADD.L   #8, A7 ; Caller cleans up parameters
    
```

**2**

The trap handler for Trap #2 is currently executing. The (SSP) = \$0000 8800, (PC) = \$0000 4000, (SR) = \$2404. TRAP #3 is the next instruction to be executed. Show the addresses and contents of the stack following the execution of the TRAP #3 instruction.

Address	Contents (word per line)
0000 87F8	2404 (SR)
0000 87FA	0000 (PC)
0000 87FC	4002
0000 87FE	0000 (format/offset word, assuming 68010+, in the Palm, this isn't present)
0000 8800	
0000 8802	

3

The state of memory is as follows with (SSP) = 0000 2004.

Address	Contents
0000 2000	1278
0000 2002	0660
0000 2004	8800
0000 2006	1212
0000 2008	CCCC
0000 200A	0000

What is the address of the next instruction to execute and the corresponding stack pointer in the following cases?

	Instruction Address	Stack Pointer
RTD #2	<i>8800 1212</i>	<i>0000 200A</i>
RTR	<i>1212 CCCC</i>	<i>0000 200A</i>
RTS	<i>8800 1212</i>	<i>0000 2008</i>
RTE	<i>1212 CCCC</i>	<i>0000 200C</i>

4.

The following procedure is part of the Mac's ToolBox:

```
procedure SetHandleSize (theHandle:Handle; newSize: Size);
```

All pointers (including handles) are 4 bytes long in the Mac. Size is a long integer representing the size of a heap block in bytes. Long integers are 4 bytes long. The tool box routines conform to Pascal-style calling conventions.

Address	Contents
0000 2000	4E70
0000 2002	3456
0000 2004	22EA
0000 2006	0004
0000 2008	2000
0000 200A	4356
0000 200C	A352
0000 200E	1344
0000 2010	1554

The preceding table lists the contents of a portion of memory during the execution of a call to SetHandleSize. Assume the stack pointer was at \$2008 *just prior* to the instruction JSR SetHandleSize which invoked the procedure. Fill in the following table with the values for the quantities assuming the above conditions.

theHandle	<i>A352 1344</i>
newSize	<i>2000 4356</i>
Old Frame Pointer	<i>4E70 3456</i>
Return Address	<i>22EA 0004</i>

## 5

Here is some code:

```
Start      LEA          MyCode(PC), A0
           MOVE.W     (a0), $0
           MOVEA.L   #0, a0
           JMP       $0
MyCode    MOVE.W     (a0)+, (a0)
```

A) What is the preceding piece of code supposed to do (in general, what is its “goal in life”)?

It writes itself over all of memory, starting at location 0.

B) Under what conditions does this code cease execution? Why?

Our code would stop execution when it tries to write off the end of memory at which point a bus error would occur.

C) If we were running on a 1 megabyte Mac Plus (with a 68000 cpu) and the above code had been running for a while (about 1000 lines of code) but then a level 5 interrupt occurred, what would be the address of the next instruction to be executed?

Since we’ve already executed our move instruction about 1000 times, we’ve already written over the level 5 interrupt trap vector. With a little hand assembly, this means that the 32-bit trap vector is now 3098 3098. If you’re really on the ball, you will note that the 68000 has 24-bit addressing, so we ignore the top 8 bits giving a PC value of 98 3098. This is the desired answer. Of course, on a 1 meg. 68000 there is no memory at 98 3098, but that’s another problem...

Note (Spring ’99): Wow, Tim asks some crazy questions, huh?

D) What assembly programming no-no does this code do?

It is an example, in a sense, of self-modifying code. It also tramples memory not allocated for it, which is also bad practice.

6.

Consider the following piece of code

```
MOVE.W    #$FF00, D0
MOVE.W    #$9, D1
CMP.W     D0, D1
LEA       $12(A3, D3.L), A1
JMP       (A1)
```

The program is running in user mode at interrupt level 0 and is loaded at address \$8800. An autovector 2 interrupt occurs while the program is executing the LEA instruction.

What is the address of the next instruction to execute?

PC = **the address stored in \$68, autovector #2**

What is the value of the SR when the next instruction executes?

SR = **00 10 0 010 000 x0001**  
= **2201 or 2211**

```
MOVE.W    #$FF00, D0
MOVE.W    #$9, D1
CMP.W     D0, D1
d1-d0 = $9-$FF00 = $9 + $100 = $109
nzvc
```

7

a) Even if you didn't use JSR and related routines (which would stomp on values that are on your stack), you would be in trouble as soon as interrupt happened (because the interrupt would stomp on your stack). Sure, you could turn off interrupts (well, except for level 7 interrupts, which can't be turned off), but it would be just too much trouble.

b) Lots of answers are possible. The main answers mentioned in class were that a real system would offer preemptive multi-tasking, library routines for doing common tasks, and protection of user programs from each other.

c) Each recursive call needs to push things onto the stack, and accessing memory is slow. The only thing really required to be on the stack is the return address (depending on whether your function really uses recursion). If you are using C or Pascal calling conventions on the Mac, you will get an even bigger slowdown because you will have to push every parameter and local variable onto the stack regardless of whether it is needed.

d) A lock is a shared variable that is set if a task/process is currently in the critical section that the lock protects. Locks enforce mutual exclusion because every task must wait until the lock is free and then grab the lock before entering the critical section. TAS is key for implementing locks so that the test and the set can be atomic.

8. No. LINK also saves to old FP on the stack.

9.

TASKTBL	DS.L	16
SWITCH	LSL	#2,D0
	MOVE	A7,TASKTBL(A5,D0)
	RSL	#2,D0
	ADD	#1,D0
	CMP	#4,D0
	BLS	NEXT
	MOVE	#0,D0
NEXT	LSL	#2,D0
	MOVE.L	TASKTBL(A5,D0),A7
	RSL	#2,D0
	RTE	