1. **Subroutines, Functions and the Stack.** Consider the following C/C++ program fragment:

```c
int sub1(int a, int b) {
    int c;
    int d;
    ...
    d = sub2(b, c);
    return d;
}

int sub2(int &x, int &y) {
    int e, f, g;
    ...
    x = y + g;
    return f;
}
```

a. Show the symbol table that the compiler will generate for sub1.

<table>
<thead>
<tr>
<th>Sym</th>
<th>Scope</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Local</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>Local</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>Local</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>Local</td>
<td>-1</td>
</tr>
</tbody>
</table>

b. Show the stack as it will appear while the machine is executing the region called “more code” in sub1.

To the right, in blue.

```
R6 →
g
R5 →
e
Dyn. Link
Ret. Addr.
Ret. Val.
```

To the right, in red.

```
R6 →
&b (x)
&c (y)
R6 →
d
R5 →
c
Dyn. Link.
Ret. Addr.
Ret. Val.
```

c. Show the stack as it will appear just before the jsr sub2 instruction in sub1.

To the right, in blue.

```
R6 →
g
R5 →
e
Dyn. Link
Ret. Addr.
Ret. Val.
```

To the right, in green.

```
R6 →
&b (x)
&c (y)
R6 →
d
R5 →
c
Dyn. Link.
Ret. Addr.
Ret. Val.
```

d. What assembly language code will be generated by the compiler for the line “return d;” in sub1.

```
LDR R0, R5, #1
STR R0, R5, #3
+stack clean up
code & RET
```

```
LDR R0, R5, #5
LDR R1, R0, #0
LDR R2, R5, #2
ADD R1, R1, R2
LDR R0, R5, #4
STR R1, R0, #0
```

e. Show the stack as it will appear while executing the region labeled “more code” in sub2.

To the right, in green.

```
R6 →
g
R5 →
e
Dyn. Link
Ret. Addr.
Ret. Val.
```

```
R6 →
&b (x)
&c (y)
R6 →
d
R5 →
c
Dyn. Link.
Ret. Addr.
Ret. Val.
```

```
R6 →
&b (x)
&c (y)
R6 →
d
R5 →
c
Dyn. Link.
Ret. Addr.
Ret. Val.
```

f. Show the assembly language code that will be generated for the line “x = y + g;” in sub2.
g. Show the assembly language code that would be generated just after the `jsr sub2` statement in `sub1` to clean up after the subroutine call. Assume that the assembler will temporarily store the result of the `sub2` call in register R0.

```
LDR  R0, R6, #0 ; Return val of SUB2 -> R0
ADD  R6, R6, #3 ; Remove (pop) ret. val. & 2 parameters
```
2. **The stack.** Suppose that you knew that a buffer overflow vulnerability existed in the program that prompts for the login password in Microsoft Windows Vista ®. Describe in detail how you could use this vulnerability to gain root access to a machine running this OS.

You would also need to know exactly how many local variables the login routine allocates, their order, and the size of each. You then know what the stack for the login routine looks like. Suppose it looks like this:

You can now craft a “password” that fills the buffer with executable code, along with just enough junk/filler to overwrite all the other local variables and place the address of the code into the return address.

When the RET instruction is executed to return from the subroutine, the address of the buffer containing your code will be placed into the PC, and on the next FETCH cycle, the CPU will begin fetching and executing your code.

Assuming that the program that prompts for the password is running in as the Administrative user (also called the superuser, or root) your program is now running with the privileges of the Administrator, and can modify any file on the system.

3. **The data path.** Using the reference sheet on the last page of these review problems, list the components of the LC-3 CPU that would be involved in executing all stages, except fetch, of the LC-3 instruction: `LDI R3, VAR1`

The components that would be involved in this operation are highlighted in the LC-3 data path diagram at the end of these solutions.

4. **The data path.** What is the purpose of the 6-bit line that carries IR[5:0] to the ADDR2MUX in the LC-3 architecture? Give an example of an instruction that uses this line.

This line allows the rightmost 6 bits of the instruction word to be added to other values, including values from the register file. Consider the instruction `LDR R0, R5, #3`. During the *Evaluate Address* phase, the CPU must add the value in R5 together with the six-bit offset specified in the LDR instruction to determine where in memory the data should be loaded from. It uses the adder and the ADDR2MUX to do this addition.
5. **Polled I/O.** Suppose we extend the LC-3 by adding a network card. To send a byte of data to the network, you need to place it into the low-order byte of the card’s Network Data Register (NDR), which is memory mapped at address xFFCC. The network card has a Network Status Register (NSR), mapped to address xFFCE. Bit #7 of the NSR is the ready-to-send bit (bit 0 is the least significant bit; bit 15 is the most significant). When bit #7 is 1, the card is ready to send data. When bit #7 of the NSR is 0, the device is not ready to send, and no data should be written into the NDR. Write a short LC-3 program that uses polling-I/O to send two characters, found in memory at addresses x3100 and x3101, to the network.

```
; R1 = pointer to the array of characters to send
LD   R1, TOSEND

; R2 = Number of characters remaining to send
AND  R2, R2, #0
ADD  R2, R2, #2

; POLLING LOOP:
; Get the contents of the status register
POLL LDI  R0, NSR

; Check if bit 7 is 0
LD   R3, B7MASK
AND  R3, R3, R0

; If so, keep polling
BRz  POLL

; If not, then we can send the next character
; Get the character into R0
LDR  R0, R1, #0

; Send the character to the Network Data Register
STI  R0, NDR

; Increment R1 to point to the next character
ADD  R1, R1, #1

; Decrease the number of characters left to send by one
ADD  R2, R2, #-1

; If there are still characters to send, do it all again
BRp  POLL

; Otherwise, we’re done
HALT

TOSEND .FILL x3100 ; The address of the first character
```
6. **Interrupts & the stack.** Describe the function of the registers Saved.USP and Saved.SSP in processing interrupts on the LC-3.

These two registers duplicate the contents of the stack pointer (R6) so that the system can maintain two stacks. The user stack (for subroutine stack frames and activation records) and the supervisor stack (for interrupt processing).

When an interrupt is triggered, if the system is in user mode (supervisor bit in the Processor Status Register = 0) the stack pointer is switched to the supervisor stack pointer by copying the contents of R6 into Saved.USP, and then copying the contents of Saved.SSP into R6. Now the system can save the Processor Status Register (PSR) and the PC onto the supervisor stack before processing the interrupt. When the interrupt service routine completes, it calls the RTI (return from interrupt) instruction, which restores the PC and the PSR to their original contents. If this results in a switch back to user mode, then the stack pointer is switched back to the user stack by copying the contents of R6 into Saved.SSP and copying the contents of Saved.USP into R6. Now the stack pointer (R6) is pointing at the user stack again and the user-level program that was interrupted can continue.

7. Suppose that you are asked to write a device driver for a new (interrupt-driven) wireless network card for the LC-3 computer. What are the parts of the device driver? How will your driver interact with the operating system?

The device driver will consist of two parts, an initialization part, which runs at boot time, and an interrupt service routine (ISR) that runs when the device is interrupted. When the system powers up, the status register on the device should have a 0 in its interrupt-enable bit, so that the device can not trigger any interrupts. After the operating system is started, it runs the initialization part of the device driver. This code asks the operating system for a vector number, and places that vector number into the vector registers on the device.

Next, the initialization code loads the ISR into memory (by asking the O/S for a free block of memory of the appropriate size). Finally, the init code places the address of the ISR into the vector table at the appropriate address for the vector number supplied by the operating system. After all of this is done, the device is ready to process interrupts, and so the init code sets the interrupt-enable bit on the device to 1, enabling it to process interrupts.

When the device triggers an interrupt, a long sequence of events is initiated (see the slides on interrupts for more details). During this process, the device will eventually supply its vector number to the CPU. The CPU will look up the address of the ISR in the vector table and jump to this code to process the interrupt.

8. What is DMA, and how can it cause problems when used in conjunction with a cache memory?
DMA is direct memory access. DMA allows I/O devices such as network cards and hard disk drives to access memory directly without the intervention of the CPU. This can cause a problem when cached copies of memory contents differ from the values stored in the RAM. We haven’t discussed this in much more detail than this, so this is all you need to know for now. You should also understand why DMA can make ordinary memory LD and ST instructions execute more slowly.

9. **Review.** What is the difference between these two assembler directives?

```assembly
VAR1 .FILL x0002
VAR1 .BLKW 2
```

I’m leaving this one for you to think about.

10. **Review.** Suppose I declare an array as follows:

```assembly
ARR .BLKW 10
```

Show three different ways I can access the 4th element of the array in assembly language.

Here is one:

```assembly
LEA R1, ARR
LDR R0, R1, #3
```

I’ll leave it up to you to find two more.
CSCI 250 – COMPUTER ORGANIZATION

Solutions to the Practice Problems for the Final Exam