1. Solution will be provided on Sunday (To allow people to develop their own solution to the ROT-13 problem in lab).

2. The HC11 has six addressing modes. A useful mnemonic is RED III (Red Eyes) for Relative, Extended, Direct, Immediate, Indexed, and Inherent.

**Relative** – Similar to PC-Relative in the LC-3, we compute a destination based on the location of the current PC. In the HC11, this is only implemented in branch instructions. However, the offset is computed from PC+2 instead of PC+1, and the range is from -128 to 127 bytes from the current PC+2 (Keep in mind that the HC11 is byte addressable, meaning each address corresponds to one byte, or 8 bits).

Examples: **BRA, BEQ, BLT**

**Extended** – Used to load or store any memory location within the entire memory map of the HC11. Extended addressing will access some 16-bit memory address. Keep in mind that this makes the minimum length of an instruction with extended addressing 3 bytes: One byte for the opcode, and two bytes for the memory address.

Example:

```
.sect .data  //Assume data memory starts at address 0x1000
var:  .word 0xBBBB
.sect .text
SUBD var     //Subtract MEM[0x1000] from D
SUBD 0x59AF   //Subtract MEM[0x59AF] from D
```

Note that in this example the memory we’re operating on is at some full 16-bit address.

**Direct** – Similar to Extended, but now the destination address is in the first 256 locations of memory, so we only need to define one byte for the destination / source address. The advantage here is that we have a smaller instruction size. A destination address will have the form 0x00 to 0xFF

Example:

```
.sect .data  //Assume data memory starts at address 0x0040
var:  .word 0x1000
.sect .text
SUBD var     //Subtract MEM[0x0040] from D
ADDA 0x11     //Add MEM[0x0011] to A
```

**Immediate** – Just like in the LC-3, we now want to include some constant number in our calculation. The number must be explicitly declared. We denote an immediate with a preceding # symbol. An immediate can be expressed in different bases and can also be the address of memory.
Example:

```asm
.sect .data
greetstring: .asciz "Hello Y'all"
.sect .text
LDAA #245       //8-bit decimal
LDAA #0x61       //8-bit hexadecimal
LDAA #041        //8-bit octal
LDAA #0b11000011 //8-bit binary
LDAA #'a'        //8-bit ascii
LDX #0x4142      //16-bit hexadecimal
LDX #greetstring //16-bit hexadecimal address loaded into X
                //Remember, this puts the ADDRESS into X, not
                //the memory at that address. This is
                //equivalent to LEA
```

**Indexed** – Similar to base+offset addressing, we use an index register as a base address and add an offset to it. The offset now occupies a full byte in memory and is unsigned, we can now access any location 0-255 entries away from the index register.

Example:

```asm
.sect .data
MyArray   .space 4    //An array of 4 bytes
.sect .text
LDX #MyArray //Put the address of MyArray in X
ADDD 0, X   //Do D = D + MEM[X]
ADDD 2, X   //Do D = D + MEM[X+2] (the next word)
ADDD 4, X   //Do D = D + MEM[X+4] (the word after that)
```

**Inherent** – Those instructions which do not have to declare an operand in the instruction are called inherent addressing. This is because the functionality of the instruction does not allow for use of any operand other than those the instruction will explicitly work with.

Examples:

```
INCB       // increment accumulator B
ASLA       // arithmetic shift left accumulator A
PSHY       // push index register Y onto stack
```
3. Things to consider are that we are going to multiply some 8-bit number by 10. Generally, we need to appreciate the fact that multiplying an n-bit number by an m-bit number will result in an n+m bit number. So this problem will have us multiply the 8-bit number by 10.

```assembly
.sect .data
NUM .byte //Assume some number is here
DEST .word //This is where we will store our number
.sect .text
main: //Let's assume there's code here that wants to multiply something by 10.
    LDAA #10 //A contains the multiplier when you reach the multiply function.
    JMP mult
mult: PSHA
    CLR A
    LDAB NUM //Load 8-bit number at NUM to accumulator B
    ADDD DEST //Add 8-bit number as 16-bit number to DEST
    PULA
    DECA
    BEQ done //Branch if Z bit is set

done:
```

Let's review what happens here. We initialize accumulator A to 10, then go to our multiply function. The multiply function preserves the value of A, clears it, then loads B with the number. Since B is the low-order byte of accumulator D, we can now add accumulator D to the memory at DEST. Then we pull A off the stack, decrement it, and when it reaches zero, we stop looping.

4. $B_1 = 00101010$ $B_2 = 11101010$

<table>
<thead>
<tr>
<th>Logical Left Shift by 2:</th>
<th>$B_1 = 10101000$</th>
<th>$B_2 = 10101000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Left Shift by 3:</td>
<td>$B_1 = 01010000$</td>
<td>$B_2 = 01010000$</td>
</tr>
<tr>
<td>Logical Right Shift by 2:</td>
<td>$B_1 = 00001010$</td>
<td>$B_2 = 00111010$</td>
</tr>
<tr>
<td>Logical Right Shift by 3:</td>
<td>$B_1 = 00000101$</td>
<td>$B_2 = 00011101$</td>
</tr>
<tr>
<td>Arithmetic Right Shift by 2:</td>
<td>$B_1 = 00001010$</td>
<td>$B_2 = 11111010$</td>
</tr>
<tr>
<td>Arithmetic Right Shift by 3:</td>
<td>$B_1 = 00000101$</td>
<td>$B_2 = 11111101$</td>
</tr>
<tr>
<td>Rotate Left by 2:</td>
<td>$B_1 = 10101000$</td>
<td>$B_2 = 10111010$</td>
</tr>
<tr>
<td>Rotate Left by 3:</td>
<td>$B_1 = 01010001$</td>
<td>$B_2 = 01010111$</td>
</tr>
<tr>
<td>Rotate Right by 2:</td>
<td>$B_1 = 10101000$</td>
<td>$B_2 = 10111010$</td>
</tr>
<tr>
<td>Rotate Right by 3:</td>
<td>$B_1 = 01000101$</td>
<td>$B_2 = 01011101$</td>
</tr>
</tbody>
</table>

Logical Shifting is the equivalent of adding zeros to the right or the left of the number, depending on the number you are shifting by.

Arithmetic Shifting is different; if we shift to the left, we shift in zeros, but if we shift to the right, the new bits we are shifting in must preserve the sign of the number, the top bit remains the same.

Rotating is equivalent to circular shifting; if we rotate by 1 to the right, then the right most bit becomes the top bit of the new number. If we rotate to the left by 1, then the highest (left most) bit becomes the least significant bit.