Review and follow the general instructions from lab 1.

This lab consists of several parts. In the first part, you will be adding the following three instructions to the WashU-2 processor and then simulate the processor running a test program that demonstrates that they work correctly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ACC Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002 lShift</td>
<td>ACC = ACC &lt;&lt; 1</td>
</tr>
<tr>
<td>0003 rShift</td>
<td>ACC = ACC &gt;&gt; 1</td>
</tr>
<tr>
<td>Dxxx or</td>
<td>ACC = ACC</td>
</tr>
</tbody>
</table>

You should use the provided assembler to generate the actual machine code for your test program. The version of the assembler that is provided in your repository already supports the new instructions.

In the second part, you will be writing a division subprogram that computes the integer quotient and remainder, for a given pair of input values. You will also implement a test program that verifies that it works correctly for all arguments. Then you will run a simulation of the processor running the test program to verify that it works correctly.

In the third part, you will be implementing a simple interrupt subsystem for the WashU-2 processor. An interrupt is a signal that is typically generated by an input device in order to inform the processor when some new input is available. In our case, we will be extending the console so that it generates an interrupt whenever a new data value is written to memory by the user. When the processor receives an interrupt, it suspends the current program and branches to a special interrupt subprogram. When the interrupt subprogram completes its processing, control is returned to the program that was interrupted.

The final part of the lab is to write a simple Etch-a-sketch program that will be run as an interrupt subprogram, which is executed whenever the user turns the knob.

To implement the interrupt subsystem, you will add an interrupt input (of type std_logic) to the processor, plus two single bit registers and three 16 bit registers.

- `intEn` is a single bit register, used to enable the interrupt subsystem. When it is zero, the interrupt input is ignored.
- `intPend` is a single bit register, which is set to indicate that there is a pending interrupt that has not yet been handled. It is set when interrupt goes high, while the interrupt subsystem is enabled (`intEn=1`).
- `intVec` is a register that holds the address of the subprogram that should be executed in order to process an interrupt.
• *pcShadow* is a register used to hold the address that was in the PC just before control was transferred to the interrupt subprogram. It is used at the end of the interrupt subprogram to resume execution at the point where it was interrupted.

• *accShadow* is a register used to save the contents of the ACC from just before control was transferred to the interrupt subprogram. It is used at the end of the interrupt subprogram to restore the ACC value, so that the interrupted program can continue where it left off.

There are several new instructions that you will need to implement. These are summarized below.

\[
\begin{align*}
0\text{FF0} & \quad \text{enInt} \quad \text{intEn}=1; \text{intPend}=0 \\
0\text{FF1} & \quad \text{disInt} \quad \text{intEn}=0 \\
0\text{FF2} & \quad \text{setVec} \quad \text{intVec}=\text{ACC} \\
0\text{FF3} & \quad \text{retInt} \quad \text{PC}=\text{pcShadow}; \text{ACC}=\text{accShadow}; \text{intEn}=1; \text{intPend}=0;
\end{align*}
\]

In addition, you will need to add logic to respond when the interrupt input goes high. More precisely, if interrupt goes high when intEn=1, the processor should immediately set intPend=1. Then, during tick 0 of the next instruction fetch, the processor should abort the current fetch, set pcShadow=PC, accShadow=ACC, PC=intVec, intEn=0, intPend=0 and tick=0. This will suspend execution of the current program and start execution of the interrupt subprogram, after saving the PC and ACC values so that they can be restored later. Notice that it also disables interrupts, so that if the interrupt signal goes high again while the interrupt subprogram is running, the new interrupt will be ignored. Also, notice that the retInt instruction, which is executed at the end of the interrupt subprogram, restores the saved values of the PC and ACC and re-enables interrupts.

You will also need to modify the console so that it has an interrupt output, which it raises high whenever some new input is received. To allow more convenient use of the knob for data input, we will use two switches swt(3) and swt(2) to define an expanded snoopMode signal. When snoopMode="00", the console is in “output mode”; in this mode, the knob controls snoopAdr and the snoopData value is periodically updated to show M[snoopAdr]. When snoopMode="10", the console is in “input mode”; in this mode, the knob controls snoopData and when btn(1) is pressed, the value in snoopData is written to M[snoopAdr]. When snoopMode="11", the console is in “continuous input mode”; in this mode, the knob controls snoopData and changes to snoopData cause the value in snoopData to be written to M[snoopAdr]. This new mode allows us to control the value in a memory location just by turning the knob. This is more convenient for applications, like the Etch-a-Sketch application that we’ll be implementing.

To inform the processor that a new data value has been written to memory, the console should raise the interrupt signal whenever it writes a new value. Note, that the memory write may not happen exactly when the snoopData value changes, since the console “schedules” memory writes to occur only at periodic intervals. It’s important that the interrupt be triggered by the actual memory write, not the change to snoopData.

As mentioned above, the last part of the project is to implement a simple *Etch-a-Sketch* program, which runs as an interrupt subprogram. To use the *Etch-a-Sketch* program, a user first sets the snoopAdr register to location 03F0, then puts the console into continuous input mode, by putting both swt(3) and swt(2) in the up position. Now, whenever the user turns the knob, location 03F0 will change and the processor will be interrupted. The interrupt subprogram should read the value at location 03F0 and interpret it as an (x,y) location on the VGA display. Specifically, the top eight bits should be interpreted as the x-coordinate for some pixel and the bottom eight bits...
as the y-coordinate. Note that the WashU-2’s VGA display is 160x120 pixels, so eight bits is enough for each of the two coordinates. The program will read the display buffer location containing the specified pixel and will change the color of the pixel to white, before writing the word back to the display buffer. Note that since each word contains five pixels, we’ll need to identify the correct pixel within the word and modify just that pixel. It turns out that to determine the correct display buffer address, we will need to perform a division operation. The division subprogram from the first part of the lab, will be used for this.

Whenever the program is not handling input from the console, it should execute the test program from the second part of the lab. When the processor first begins execution, the program should first set the interrupt vector to point to the first instruction of the *Etch-a-Sketch* subprogram, and then enable interrupts. Next, it should begin executing the division-test program. Now, whenever the user changes a stored data value, the processor will be interrupted and the *Etch-a-Sketch* subprogram will be run to process the new input value. After the subprogram has updated the appropriate pixel, control will return to the division-test program.

There is one subtle issue that arises in this program. Both the division-test program and the *Etch-a-Sketch* program require the division subroutine. But this creates the possibility that both programs will attempt to execute the division subroutine at the same time. Unfortunately, the simple subroutine linkage mechanism that we are using will not work correctly when there are multiple overlapping calls to the division subroutine. We will get around this limitation by using two separate copies of the division subroutine, one used by the test program and the other by the *Etch-a-Sketch* program.

For this lab, you may work with a partner. You only need to turn in a single lab report, but it should include the names of both partners, with one name circled. Commit the repository of the partner whose name is circled on the report.