CSC258 Final Project Advice

This document contains advice that you may find useful when working on your final project. The document is divided into the following sections:

1. VGA
2. Generating Random Numbers
3. General Circuit Design and Verilog
4. ModelSim
5. General Debugging

1. VGA

1.1.   BMP2MIF file converter

A BMP2MIF file converter that you could use to convert a `.bmp` file to a `.mif` file can be found at the locations below. Note that this site is an external site and like with all external sites, we cannot guarantee any support for this program:

- Background information:
  http://www.eecg.utoronto.ca/~jayar/ece241_08F/vga/vga-bmp2mif.html
- Download page: http://www.eecg.utoronto.ca/~jayar/ece241_08F/vga/vga-download.html
- Direct download link: BMP2MIF Converter with source (zip):
  http://www.eecg.utoronto.ca/~jayar/ece241_08F/vga/bmp2mif.zip

The source code in that linked zip file has some comments that guide you through the process of how you can edit/modify an image in Paint. Make sure that your image, whether used as a background image or a sprite, has the appropriate dimensions.

Also, a piece of advice: Please try not to spend too much time on the aesthetics of your game early on. It's OK if your game has initially a black background and your moving objects are nothing more than moving squares. As long as the game is working properly which is the important part, you can spend more time polishing the graphics near the end of your project. If your project is challenging enough even without fancy graphics, you can get full marks on the project even if you don't get the fancy graphics working. If you received feedback that your project was too simple, and animation or background images were added to make its scope more appropriate, the situation is different, and you should pay more attention to graphics in that case, because your mark would depend on getting graphics working. However, even then it is advisable that you get the simple version of the project working first.

Note that if you are using Paint after you properly resize the image you need to save it as type 24-bit bitmap (`*.bmp`).

The BMP2MIF converter program should display the appropriate pixel dimensions of your image in the printed message ("Input file is ..x.. ..-bit depth"). Make sure these match the dimensions of your image and no extremely large values are displayed.

If you are running the converter on a Mac, you can run this first on your terminal if you have a non 24-bit bmp image:
`convert orig_image.bmp -colorspace rgb new_image.bmp`
You may also need to comment out the whole chunk of code enclosed in
`#if !defined (WIN32) ... #endif.`

1.2. Background and moving images (sprites)

If you have multiple different background images that you wish to use, you can instantiate a separate RAM module for each of your different background images and initialize them with those .mif files as needed. Then based on a given signal (e.g., the signal that specifies game termination) your FSM should start reading the contents of the appropriate RAM and send that information to the VGA adapter over multiple clock cycles. Of course, if the VGA needs to accept coordinates and colours from multiple sources, one or more multiplexers are needed on its inputs.

You will need some way to map (x, y) coordinates to memory addresses. The `vga_adapter.v` file, part of the starter kit provided to you for lab 7, shows one way you can achieve this. You should also be able to figure this out on your own, because this is similar to any other case when a 2-D data structure (e.g., array) needs to be represented in 1-D.

Images that move on the screen (animations) are often referred to as sprites. A sprite (https://en.wikipedia.org/wiki/Sprite_(computer_graphics)) is a small 2-D image that you want to move over a fixed background. All you need is to have some unique way of identifying the location of that sprite in your 2D grid of pixels. So maybe you can use counters to contain (x, y) coordinates of the upper left pixel of your moving image. Alternatively, you could use counters to contain coordinates of the bottom left pixel if that better suits your purpose. What you keep track of does not matter, as long as it uniquely identifies where you sprite currently is.

For a sprite that is not just a simple square of one colour, then the colour of each pixel the sprite consists of needs to be stored somewhere. You could store your sprite(s) in a separate RAM module that you include in your design. You could use a `.mif` file to initialize that RAM. You can either produce `mif` files using the BMP2MIF converter provided above, or manually edit the `.mif` file in Quartus.

As long as you know the starting address of your sprite in that memory module (if the memory module contains multiple sprites) and how many pixels your sprite consists of, you can read all the pixels of your sprite out of that memory and write them to the appropriate locations in the VGA adapter. Alternatively, you can use a separate memory module for each sprite. The advantage of the latter is that you don't have to keep track of the starting address of each sprite, but now you need additional logic (e.g., a multiplexer) to choose one of several memory modules.

2. Generating Random Numbers

If your project requires generation of random numbers (e.g., to randomize a sprite's starting position in the game), there are two simple ways this can be achieved.

If the user is pressing keys every time a random number needs to be generated, you can simply use a counter whose clock is running at 50MHz and whose enable signal is always enabled, so that it is always counting. When the user presses the relevant key, you can simply capture the value of this counter into a register and use this as a random number. Since the counter is counting at such a high rate (50 million times per second), no user can control what value will be in the counter when they press the key, making the output effectively random. You need of course to ensure that the counter's counting range is appropriate for your purpose.

If you need to generate random numbers at times when there is no user input, or you need to generate multiple random numbers, you will have to implement a pseudo-random number generator. A simple circuit that achieves this is a Linear Feedback Shift Register (LFSR). The description of a four-bit LFSR circuit is provided below. You can do some research to find how to implement LFSR circuits with more bits, if your project requires this. Note that you could use the counter approach described above as the seed to initialize the LFSR circuit.

A linear feedback shift register (LFSR) produces a sequence of pseudorandom numbers – at each new clock cycle a new pseudorandom number is generated using the previous state of the circuit.

The initial value of the LFSR is called the seed; as the LFSR is deterministic, the sequence of values that the circuit generates will be based on its seed. There are a number of different types of LFSRs; one is the 4-bit Fibonacci LFSR[1]:

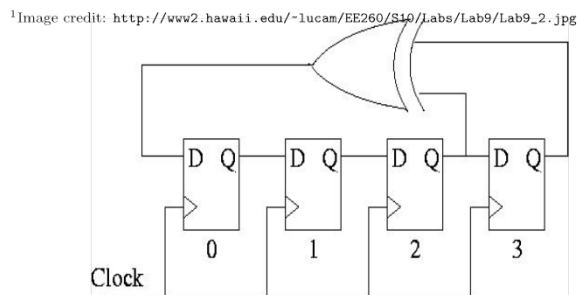[1]Image credit: http://www2.hawaii.edu/~lucam/EE260/S10/Labs/Lab9/Lab9_2.jpg

Figure 2: A four-bit Fibonacci LFSR. Note that if you use a seed of 0000, the LFSR will never change state! An animated example of this circuit in action is available at http://en.wikipedia.org/wiki/File:LFSR-F4.GIF

3. General Circuit Design and Verilog

3.1.   How *to Implement Algorithms*

The best way to implement an algorithm in hardware is to come up with a way to represent the algorithm as a finite state machine (FSM) controlling a datapath. For example, assume you want to count bits that are set in an input bus. This could be done by using a datapath consisting of a shift register and a counter. The shift register is used to store the input number, while the counter is used to count the bits. The FSM first resets the counter and loads the shift register from the parallel bus. Then the FSM enables the shift register to shift the number to the right by one position in each clock cycle. In each clock cycle the counter is incremented if the shift-register bit in position 0 is set. Once all the bits have been shifted, the counter contains the number of bits that are set, and FSM asserts signal done, meaning that the algorithm has finished, and the result can be read from the counter.

One important thing we tried to teach you in the labs was to always start with a schematic. This ensures that you understand what the circuit that you are trying to build is. Only after you have the schematic should you attempt to write Verilog code. This is because Verilog is a hardware description language, so you have to know what hardware you are trying to describe. If you start by immediately writing Verilog code, you may end up with hardware that you did not intend to build and whose behaviour you do not understand. Don't forget that RTL Viewer is your friend.

3.2. Avoiding latches

Latches and/or gated latches get inferred from your code if you do not specify all possible cases in if-else or case statements. For instance the following code will produce a latch:

```
always @(*)
begin
     if (en) out = in;
end
```

Variant of this code that will also produce a latch is this:

```
always @(*)
begin
     if (en) out = in;
     else out = out;
end
```

These two pieces of code are identical and they both produce a latch. As discussed in class, latches can introduce uncontrolled oscillations in your circuits and should not be used in synchronous circuit design (i.e., they should not be used in your lab projects). Similar problem occurs if you do not specify all possible cases in your case statements. That's why it is a good practice to always include a default statement, and that

DEFAULT STATEMENT CANNOT SPECIFY THAT A SIGNAL SHOULD REMAIN WHAT IT ALREADY IS (e.g., out=out;), because that is the same as not specifying the default statement at all.

Also, the default statement has to specify ALL THE SIGNALS assigned elsewhere in the case statement. E.g. the following code will result in latches for signals b and c.

```
always @(*)
begin
    case ({in1,in2})
    2'b00: begin
     a = 1'b0; b=1'b1; c=in3;
    end
    2'b01: begin
     a=1'b1; b=in2; c=in1;
    end
    default: begin
     a=1'bx;
    end
    endcase
end
```

The above code correctly specifies that in default case the value of a is don't care. However, it does not specify what the values of b and c should be in the default case. Therefore, signals b and c will be implemented as latches, which should be avoided.

4. ModelSim

Please review the various tips that were provided to you in lab handouts to use ModelSim effectively. For example, if you have a bus consisting of multiple wires, make sure you are not using force commands to set them one at a time. Instead use syntax such as 2#1010, described in lab 3 handout.

4.1.   Observing signals at lower levels of hierarchy

When you use `add wave {/*}` command to add signals to your waveform, by default you will only see signals from your top-level module. If you wish to see signals from your lower level-module, this is possible, but you have to specify the full path to the module that you wish to observe.

Let us assume that your project consists of top-level module called `game`, and lower level module called `datapath`, which includes two instances of module called `mycounter`, as follows:

```
module game (in1, in2, out, out2);
…
    datapath d0 (
        .a(in1),
        …
    );
…
endmodule

module datapath (a, b, c);
…
    mycounter c0 (
        .clk(a),
        …
    );
    mycounter c1 (
        .clk(a),
        …
    );
…
endmodule
```

To observe signals within the two counter modules in the waveform window, you would use commands such as the following in your `.do` file:
```
add wave {/game/d0/c0/*}
add wave {/game/d0/c1/*}
```

Alternatively, you can do `add wave -r {/*}` to add all signals from all the modules instantiated in your design hierarchy, after you have logged them all with `log -r /*`

5. General Debugging

When working on a complex project, you will almost certainly run into a situation where something in your circuit is not behaving as expected. In those situations, it is easy to fall into the trap of trying to figure out what is wrong by simply reading the code over and over again, and trying to tweak a little bit here and there. This is fine to do if it's an obvious mistake, but if you find that you have been doing this for a while and/or you are just guessing and changing things hoping it might work, it's time to break the vicious cycle and do some real debugging. You can use one of the following strategies:

1. Use ModelSim to simulate key parts of your circuit to see where things go wrong. While there is a small overhead of setting up ModelSim simulations, it usually pays off in the end because it saves you hours of modifying the code and not achieving anything. Testing each key module of your design separately also helps as this way you break dependencies that might exist between different parts of your circuit. For example, you might be getting an incorrect output not because module A does not work properly, but because module B that drives some of the inputs of module A is buggy. Separately testing modules A and B would help you quickly isolate the module where the bug is.

2. Use the RTL Viewer to ensure that the circuit produced by Quartus is the circuit you intended to build.

3. Check the Quartus Messages window for warnings and errors. Pay particular attention to Critical Warnings, such as ones warning you that some of your inputs are unused, some of your outputs are stuck at GND or VCC, or that your circuit contains latches.

4. If you are in the lab, connect your clock to one of the keys, so that you can observe how your circuit behaves on a cycle by cycle basis. You can connect some of your counters or state registers to HEX displays so that you can observe them in real time. Keep in mind that Quartus changes the values that appear in your state register for a given state. Use the Quartus State Machine Viewer to see what binary values Quartus assigned to which state.