Recommended Design Techniques for ECE241 Project

Frank Franjo Plavec
Department of Electrical and Computer Engineering
University of Toronto

DISCLAIMER: The information contained in this document does NOT contain official grading policy. The information provided here is based on my personal experience with ECE241 course projects in the previous years. Its purpose is to warn you of some common mistakes and answer some common questions student in earlier years had. As grading policies and project requirements change from year to year, please consult course web site or your instructor for official policies.

THIS DOCUMENT MAY CONTAIN SOME MISTAKES. I will do my best to point those mistakes to you if I discover any, but I cannot make any guarantees.

All information in this document is MY PERSONAL PREFERRED WAY OF DOING VARIOUS TASKS RELATED TO HARDWARE DESIGN. It is by no means the only possible way to perform these tasks. Also, this document does not cover, nor does it attempt to cover all aspects of various problems discussed. Therefore, you should not make any implications on aspects of the problems not mentioned in this document. In other words, if the document states X, and you try to do Y, which is "very similar to X", do not assume that statements this document makes for X necessarily hold for Y. CHECK YOUR ASSUMPTIONS against your textbook, course notes, your instructor's and/or TA's advice, compilation and simulation results from Quartus, and finally, common sense.

Verilog and Quartus Issues

When using Verilog for the first time in a real project, users are often tempted to use fancy features of the language to make their lives easier. Unfortunately, if one succumbs to those temptations, they usually make their lives harder. The main reason for that is that Verilog, the way it is used in ECE241 labs and the way Quartus II interprets it, is not a programming language. Verilog is a hardware description language, meaning that various blocks of code directly map into hardware. Therefore, the designers must always have in mind the circuit they want to implement, not the program they want to write.

NOT ALL STATEMENTS THAT ARE LEGAL IN VERILOG WILL TRANSLATE INTO HARDWARE THAT PERFORMS THE FUNCTIONS THAT A HUMAN WOULD EXPECT WHEN READING THE CODE. Through following examples I will attempt to show some common mistakes and misconceptions. General rule is that it is NOT advisable to be creative when writing the Verilog code. It is much better to follow the practices known to work well. Too much creativity may result in Quartus misinterpreting the intentions of the designer and producing unexpected results.

Many useful tips on common mistakes in writing Verilog code can be found IN SECTION A15 OF THE TEXTBOOK (S. Brown, Z. Vranesic: *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill: New York, 2003).

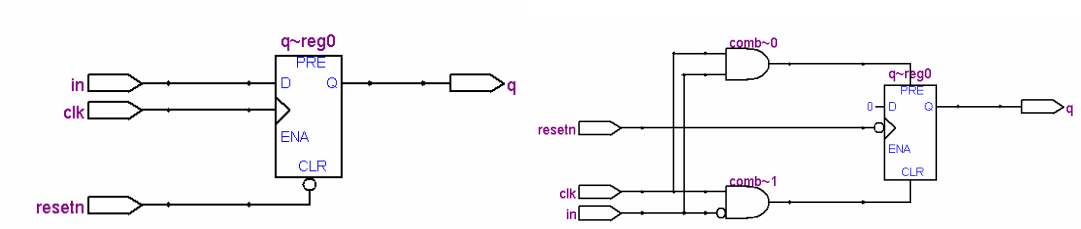
Asynchronous inputs to registers

One of the best examples of how even a minor change in the code can result in a major change in the resulting circuit is syntax for specifying asynchronous inputs to flip-flops. The following two pieces of code might seem to implement the same functionality:

```
always @ (posedge clk or
negedge resetn)
if (resetn == 0) q <= 0;
else q <= in;
```

```
always @ (posedge clk or
negedge resetn)
if (clk == 1) q <= in;
else q <= 0;
```

However, compiling these two pieces of code with Quartus II produces significantly different results



The underlying reason is in how Quartus "infers" which signal in the sensitivity list is a clock, and which is an asynchronous input. An edge sensitive always block with more than one signal in the sensitivity list has to include a chain of if-else statements; the last else in the chain is assumed to specify the clock signal and the clock event. Therefore in the first code sample, the last else specifies that on the active edge of the clock `clk`, `q` will become equal `in`. The clock itself is inferred from the fact that the only possible way to reach that else branch is the positive edge on the `clk` signal.

The situation is reversed in the second code sample: the last else specifies that on the active edge of the clock `q` will become equal 0. The clock itself is inferred from the fact that the only possible way to reach that else branch is the negative edge on the `resetn` signal.

The schematics for these examples were obtained using RTL Viewer tool. It is very useful to analyze how your code maps into hardware. You can access it through

Tools->Netlist Viewers->RTL Viewer. You have to compile your design first, or at least perform Analysis & Elaboration.

MORAL: WRITE THE CODE THE WAY IT IS RECOMMENDED IN THE TEXTBOOK!

Edge-Sensitive vs. Level-Sensitive always Blocks

There are three simple rules to follow:

a. If you want to implement a register, use edge-sensitive always block (posedge, negedge)

b. If you want to implement combinational logic (gates) use level-sensitive always block. Alternatively, use an `assign` statement, in which case you do not use an always block

   When you use level-sensitive always block, instead of writing `always @ (a or b or c)`, you can simply write

   ```
   always @(*)
   ```

   * automatically includes all the signals used inside the always block into the sensitivity list.

   It is impossible to specify an incomplete sensitivity list. The following code does not make sense:

   ```
   reg out;
   always @(in1)
   begin
    out = in1 & in2 & in3;
   end
   ```

   You should keep in mind that level-sensitive always block describes logic gates. Can you imagine a 3-input AND gate whose output only changes when only one of its inputs changes, but not the other two?

   This code is still legal. Quartus will issue a warning and assume that you forgot to include signals in2 and in3 into the sensitivity list, and produce AND gate as if those signals were in the sensitivity list. That's why it makes sense to simply write `always @(*)`.

   It is also important to note that `out` is not a register. The fact that variable is defined as `reg` only means that it is going to be assigned a value inside an always block. Whether that variable gets implemented as a register or not depends on the type of the always block (edge-sensitive → register,

   level-sensitive → combinational logic)

c. Always block cannot contain both edge-sensitive and level-sensitive signals. i.e `always @(a or posedge b)` is an illegal statement. One always block either describes a register or combinational logic, not both!

Blocking vs. Non-blocking Assignments

Verilog supports two types of assignments: blocking (=) and non-blocking (<=). There are three simple rules that should keep you out of trouble:

a. Always use ONLY blocking assignments inside level-sensitive always blocks (describing combinational logic)
b. Always use ONLY non-blocking assignments inside edge-sensitive always blocks
c. Therefore, never mix blocking and non-blocking assignments inside the same always block.

Any differences you might have learned between blocking and non-blocking assignments do not matter much if you stick to the three rules above, and keep in mind that an edge-sensitive always block always produces a register, and level-sensitive always block always produces combinational logic, as described in the previous section.

Module Instantiation

Module instantiation is equivalent to plugging a piece of hardware whose functionality is specified in a given module into the design. MODULE INSTANTIATION IS **NOT** A FUNCTION CALL. Therefore, modules can NOT be instantiated within always blocks (the meaning of that would be: “plug in a new piece of hardware every time some signals change”, which obviously does not make sense).

“Initializing” variables

Often times you may want to “initialize” some variable and attempt to do the following (which is WRONG!!!):

```
reg out = 0;
```

This just does NOT work, because you are designing hardware. In hardware, wires cannot have “initial values”. If you need some wire to have certain value when the circuit is powered up, you should design a register with a reset input. When the reset is asserted, the value of the register should become the “initial value”. If you do not understand the last sentence, check the textbook to see how registers with reset signals are implemented in Verilog. Note that Quartus will still compile the code above, but will ignore the “initialization”.

How do I write a function?

Avoid using Verilog functions or tasks, because they do NOT behave anything like C functions, and they are not particularly useful for small designs like yours.

Avoid 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 @(in or en)
begin
     if (en) out = in;
end
```

The rationale for doing this goes something like this: “I need a temporary variable which will only change if enable is high, otherwise it should stay the same”. The person reasoning like this is thinking programming instead of hardware design. First of all, THERE IS NO SUCH THING AS A TEMPORARY VARIABLE in Verilog. If you need some value to be retained, USE CLOCKED REGISTER!

Variant of this code is this:

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

These two pieces of code are identical and they both produce a latch, and they both represent a VERY BAD DESIGN PRACTICE. Latches can slow down the operational frequency of your circuitry and in some cases even introduce unpredictable behaviour, depending on your coding style. 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 a latch for signals b and c.

```
always @(in1 or in2 or in3)
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 (1'bx means 1 bit 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.

Simulation Issues

Sometimes you may have to simulate a complex circuit for the purpose of debugging. For efficient debugging, it is often useful to be able to monitor the values of the signals internal to the circuit. That is not always possible because Quartus performs many optimizations, and usually does not keep track of exact signal names in the process. Therefore, to monitor the internal signals one has to bring those signals out to the top level of the design and make them outputs of your circuit, so that they become visible in the simulation waveform.

How NOT to Implement Algorithms

The most challenging task when writing a Verilog code is describing various algorithms that your project requires. Since Verilog is so similar to C, one is tempted to implement an algorithm similarly to C code. In such a case, one is tempted to use `for` or `while` loops, functions and tasks. As previously mentioned, Verilog functions and tasks should be avoided, because they are not very useful when the circuits are synthesized. `for` and `while` loops should be avoided in MOST cases, however, they can be useful sometimes, if one is aware of their meaning and limitations. The following discussion will focus on `for` loops, but it is also applicable to `while` loops. `for` loop can be used to write some

code in a shorter way. Assume you have a 4-bit signal, and you would like to reverse bits in that signal. One way to do that is to write the following code:

```
wire [3:0] a;
wire [3:0] b;
assign b[3] = a[0];
assign b[2] = a[1];
assign b[1] = a[2];
assign b[0] = a[3];
```

A shorter way to write this is using the following code:

```
wire [3:0] a;
reg [3:0] b;
integer i;
always @(*)
begin
     for (i=0; i<=3; i=i+1)
          b[i] = a[3-i];
end
```

Using `for` loop in this way is legal and desirable. However, the main limitation of the `for` loop is that loop index limits (0 and 3 in the previous example) have to be constants or parameters. It is not possible to specify something like this:

```
wire [3:0] a;
always @(*)
begin
     for (i=0; i<=a; i=i+1)
          // do something
end
```

This code results in compile time error “Error: Verilog HDL For Statement error: must use only constant expressions in terminating conditions”

This limitation seriously reduces usefulness of the `for` loop, and it is obvious that it can hardly be used to describe an algorithm. Another FLAWED approach with using `for` loops is the following code:

```
module temp (a, b, clk);
input clk; input [3:0] a; output [3:0] b;
reg [3:0] b; integer i;
always @(posedge clk)
begin
     for (i=0; i<=3; i=i+1)
     if (a[i] == 1)
          b <= b + 4'b0001;
end
endmodule
```

The intention of the designer was to describe an algorithm that counts the number of bits of `a` that are equal to 1, and outputs that number on `b`. This code is flawed for several reasons. First of all, it is unclear when this "algorithm" has completed its work and produces a correct result. Second, the code doesn't produce circuitry even close to what was intended. The code produced is in effect equivalent to the code one would write if one manually wrote all the steps of the `for` loop. Therefore, this code is equivalent to:

```
module temp (a, b, clk);
input clk; input [3:0] a; output [3:0] b;
reg [3:0] b;
always @(posedge clk)
begin
     if (a[0] == 1) b <= b + 4'b0001;
     if (a[1] == 1) b <= b + 4'b0001;
     if (a[2] == 1) b <= b + 4'b0001;
     if (a[3] == 1) b <= b + 4'b0001;
end
endmodule
```

If you are not convinced that these two pieces of code produce the same result, type them into Quartus and synthesize, and use RTL Viewer to check the resulting circuitry.

You might think that the second code actually counts the bits in `a` that are set. However it does NOT. If there are multiple assignments to the same variable inside an always block, only the last assignment is taken into account. Therefore the above code will increment (increase by 1) a value in register `b` if input `a` has at least one bit set, regardless of the actual number of bits that are set. This will occur on every positive clock edge, thus producing a simple counter. This is far from the intended behaviour. In case `a` has all bits set to 0, the value of the register `b` will not change.

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. In the example above, counting bits in an input signal can 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. FSM first resets the counter and loads the shift register (in parallel) from the input. 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.

How to "slow down" the clock

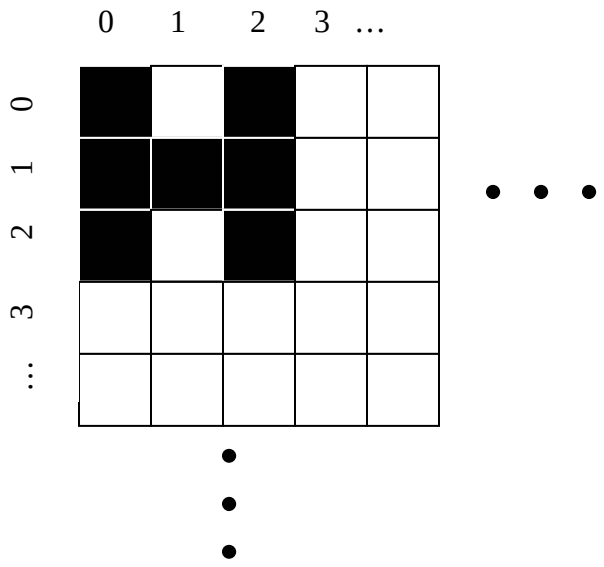
In many projects there is usually a need for some part of the circuitry to operate slower than the clock that is usually available on the development board. For instance, if a project includes animation on VGA display, the object will usually move too fast across the screen if the animation is implemented by an FSM clocked at 50 MHz. Therefore, clock needs to be "slowed down", or divided. Clock division refers to the fact that the clock frequency is divided by a certain factor.

The approach that works well in FPGAs (and in other technologies) is to use a synchronous enable signal to "slow down" the operation of one or more flip-flops. The idea is that, instead of slowing down the clock, we can simply disable the flip-flop for a number of clock cycles, and only enable it once in a while. For instance, if a flip-flop is only enabled every 10^{th} clock cycle, its operation is identical to the flip-flop driven with 10 times lower clock frequency. A simple way to obtain a signal that will enable the flip-flop only every x cycles is to have a counter that counts from 0 to (x-1). Then the enable signal should only go high when the counter is equal to (x-1). One enable signal will typically be used for many flip-flops, because we usually want to slow down the operation of a part of the circuit, not just a single flip-flop.

How to draw and move simple objects across the screen

In lab 7 you learned how to move a dot across the screen using 2 counters. Part 3 of the lab 7 deals with moving a small picture across the screen. A suggested way of doing this is to copy contents of a memory module that resides outside the VGA controller to a specific location on the screen. This is useful to draw small pictures.

Alternatively, if your object is simple enough, you can draw it by using a simple FSM. First, we have to have a pixel map of the object we want to draw. Assume, we want to draw letter **H** in the top left of the screen, as in the figure below.



One way to draw this object is to design an FSM that will output the row, column and colour of each pixel and write them to video memory. A sequence for letter H would be:

row	column	colour
0	0	1
0	1	0
0	2	1
1	0	1
1	1	1
1	2	1
2	0	1
2	1	0
2	2	1

The above table is derived assuming colour 1 means the pixel is on (black in the picture above), and 0 means the pixel is off (white in the picture above).

Such an FSM would then always output the letter **H** to the top left corner of the screen. The simplest way to move this object on the screen is to have two counters, much like in lab 7, which will define the coordinate of the top left corner of letter H on the screen. Let us call these counters offset_x, and offset_y. We now use an adder to add the column output of the FSM to the offset_y counter, and another adder to add the row output of the FSM to the offset_x counter. Connecting the outputs of these adders to the video memory

(instead of connecting signals row and column directly), we get a circuit that can draw letter **H** anywhere on the screen. If you are not clear on how this works, try writing a table, like the one above, of values that will be produced by the described circuit when the counters have some non-zero values (e.g offset_x = 10, offset_y = 20).

If the counters are controlled by buttons, the letter can be moved across the screen. Of course, the FSM has to be designed to erase the letter at the old position before updating the counters and drawing it at the new position. Erasing is simply done by setting all the pixels at the old position to 0. Moving a picture that you copy from memory is done in a similar way, as is described in part 3 of lab 7.