Verilog Coding for Successful Synthesis

By

P.Radhakrishnan,
Hardware Engineer,
Cisco Systems, Inc.,
170, Tasman Drive,
San Jose, CA 95134 (USA)

Jan 2003 (Issue-4)

Contents

Introduction

It has been nearly three years I had the least time to write anything of this sort, despite my sincere efforts to rob some time from my daily schedule. With a rugrat at home and with extended targets at work, it became almost impossible to have some peaceful time to think about such topics. However I realized that life will be like this always, and I need to fetch some time for these kinds of activities. Thus, re-born this effort. I hope to continue this for some more years as long as it is helpful to the students. My sincere thanks to Mrs.S.Thamaraiselvi, HOD, CS Department of our college for giving enough boost that made me jump into this activity again.

In this paper I have tried to address some of the important aspects of Verilog coding that would let us build successful logic that functions in an intended manner. Verilog by itself is a vast language that helps designers to build logic, libraries, test suites, complex algorithms and many other functional blocks. Those who are ASIC designers should follow some guidelines during their coding, to make their design error- free and reliable.

Prerequisite

Since this paper deals with the Verilog coding style that would help designers to achieve the intended function in the chips that they build, I assume that the reader has a moderate familiarity with this Hardware Description Language (HDL). Verilog has become very popular in the ASIC design industry these days, and there are many design wins, may be about 95%, in Verilog domain. VHDL is also a famous HDL, but I have serious doubts in the wide usage of this language in the ASIC design in today's industry. This does not mean that there is no one designing using VHDL. This paper itself is about the Verilog coding style and hence I take the liberty of using only Verilog in the illustrattions I have used in this paper.

I want to give an emphasis that this paper does not attempt to teach you the Verilog language, but trying to give the idea behind achieving **synthesizable logic** from the Verilog codes that you will be writing in the future.

Hardware Description Language

With every day's increasing design complexities and advancing technologies, the olden days' techniques of custom design by layout has become impossible to cope up with the changes required in the industry. Every time the next generation of devices are done, the performance demand increases by a factor of at least two and the design complexities by a factor of four or more. How do we build a sequential state machine in the digital lab? We try to deduce the state diagram that would meet the functional requirement and then determine the current state and the next state of that state machine, with the inputs which take the state machine from one state to the other. Once we did this, we try to find that combinational logic in the sum of product (SOP) form using the good old K-map. This will be the logic that feeds the D input of a flip-flop which will switch according to the output generated by the combinational equation. Imagine if we were to design a whole chip like this by deducing the equations with our traditional K-map technique, it would take decades to design current generation complex silicon chips!!

It was the US Department of Defense (DOD) lab that tried to do the initial work on the idea of HDL with the help of some universities. Just like a C compiler generating the machine code for the higher level source code that we write, they made VHDL as the higher level abstraction language that would describe the logical function in the form of boolean equations or behavioral code. This source code was cranked by a compiler to yield the logical representation of that code using logic gates and flip-flops, just as the C compiler yielding the machine code. This VHDL development happened in the 1980's. In the same time period another set of people were thinking in the same line in the commercial industry and their efforts led to Verilog. We are not going to debate Verilog *vs* VHDL in this paper. The above section gives just a pointer to how the VHDL or Verilog was thought of. What was once a documenting practice for remembering and reusing the logic, later bloomed into a new methodology called the hardware description language.

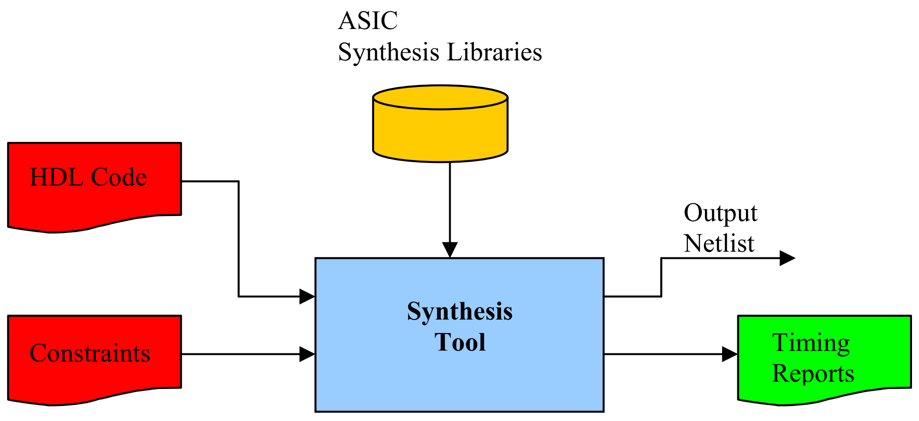
We may try to define the HDL in the following manner.

> *The Hardware Description Language is a software program that describes the functional behavior of a hardware piece that one wants to build. It gives a higher level of abstraction to the function so that it is easier for implementation and interpretation, whose lower level details are realized using automated tools.*

A higher level of abstraction can be achieved by using any constructs of the language that we have chosen. All these constructs need not necessarily be leading to a realizable implementation by the tools, though they depict the functionality of the intended hardware. Such pieces of HDLs are said to be functional models of the hardware piece that we were trying to implement. In this paper, we are going to give more emphasis to the Verilog HDL that would let us build actual realizable hardware logic using any synthesis tools. Wherever possible, I have tried to give some examples to illustrate the problem and the advantage of such coding styles. The HDL coding is also known as RTL (Register Transfer Level) coding, the reason of which will be explained later.

Synthesis

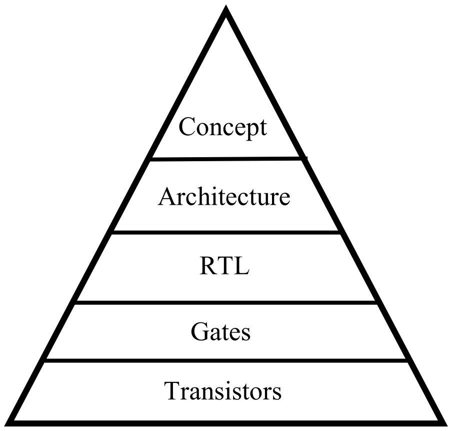
There was a section spent on synthesis in my paper named "*ASIC Design Flow*" the *issue-3* which you should read first. The synthesis step in the ASIC design flow converts the HDL behavioral code one wrote, into a netlist. The netlist is the database that has the logical components that builds the entire hardware/chip and the connectivity information. It will also contain the IO cells and the signals connected to them in the required manner. There are synthesis tool vendors who build tools that take the HDL code, input timing constraints and target to a particular library of cells and generate the netlist, having the same functionality. This process is not just a "***push button***" process. We need to understand the capabilities of the synthesis tool and use them accordingly to achieve the results that we wish to get. Synthesis tools will also give the timing reports for various paths in the design, which we have to review and see if it meets the timing requirement that our design demands. If the timings are not met, then we have to use different techniques to achieve the timing.



The above diagram shows very briefly about the inputs and the outputs during the synthesis phase. These are just the minimal inputs during synthesis, but there are other additional inputs that refine the synthesis process to get better results. We are not going to elaborate on those points. Example synthesis tools are Design Compiler from Synopsys, Inc., Build Gates from Cadence Inc., Leonardo Spectrum from Mentor Graphics, etc.

Think in Hardware

When we write a piece of a C code (say that takes in five quadratic equations and finds the solution for them), we write the portion of the code that will have the equation to find the solution and make it loop through the five inputs that we will be providing. What is done underneath at the machine level is pretty much not seen or even thought of while coding this program. This kind of approach will not work in the HDL coding. The first step that one needs to think is to imagine a hardware that will do this function. Once we get an idea of what logic would do the function, we have to build the logic using the HDL. HDL is also not so dumb to expect every part of the logic. There are smart operations (arithmetic, logic, etc.) that the HDLs support, that are recognized by synthesis tools to arrive at a suitable logic. But it is important to think in terms of the hardware elements that will make the required function in reality.



Any design concept that we want to use in an application needs to be implemented in the right way in the right place to reach the end user. A concept can evolve from a research or form a totally new thinking of a requirement and so on. Once we think that an idea is worth implementing for an application and, if we think that we should do an ASIC for that, there starts the ASIC design process. The concept has to be thoroughly analyzed for its correctness and get proven by some method. To start an ASIC to build those functions, the initial stepping stones are laid by looking at the feasibility of implementing the idea in the ASIC. A detailed ASIC architecture is done with the system designers and architects to refine the top level requirements into various building blocks. The problem has to be broken into many logical blocks and the feasibility of the individual blocks together providing the complete solution has to be worked out. After this phase, the specification of the chip will be frozen stating all the requirements that this chip has to do and then to make it further specific, a micro-architecture will also be done. Then these conceptual ideas will be transformed into behavioral coding (HDL) that will precisely represent the hardware behavior. Here is where we need to think all the time in terms of the hardware that we need to realize to achieve the functional goal. Funtional testing will be done on the completed code. This process is shown in the pyramidal structure shown above, where the next step is to convert the HDL behavioral descriptions into the equivalent gates. This is done by synthesis tool. The bottom most layer in the diagram is indicated as "Transistors" which represents the fusing of the transistors that builds the gates and interconnects in the silicon die. This whole layer in the pyramid is a complex one by itself, and some of the details are discussed in the "backend" phase of the technical paper "*ASIC design flow*", the *issue-3*.

Synthesizable Logic

You may wonder why the question of "Is this code synthesizable?" arises. This is because of the vast capabilities of Verilog which help describe any behavioral function of a design. Only a subset of the various constructs available in Verilog is synthesizable. Most of the people who start writing code for ASICs or FPGAs will realize the problem of suitability of the code for synthesis after completing the coding and verification if they are not careful enough to consider these factors. Before we take a look into the realizable logic, let us take look at an example of a code that can not be successfully synthesized.

Example:1 (Verilog Code)

```
module delay (in, out);

input in;

output out;

initial

  begin

    wait (in)

    out = 1'b1;

  end

endmodule
```

This code shown in example-1 is a very simple, legal construct in Verilog. All this code does is, after starting the simulation, waits for the signal ***in*** to go ***high***, and once this ***in=1***, the value of the output ***out*** changes to a ***1*** and stays ***1*** until the end of simulation. If you have a software perspective of this module, the constructs ***initial*** and ***wait*** have meaning with respect to the execution of the line

```
out = 1'b1;
```

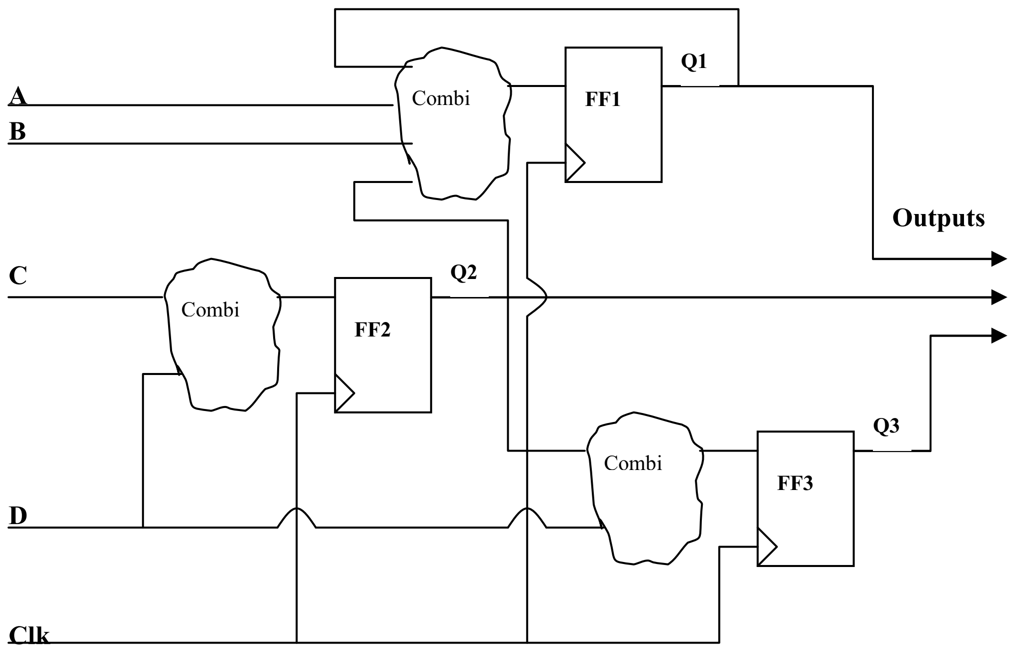
i.e. the simulator waits for the input signal ***in*** to go ***high,*** once the simulation time starts. After executing the above statement, the module does nothing until the end of simulation. This is what it is intended to do.

Now let us take this code through a synthesis tool to convert it into an equivalent logic with gates. The synthesis tool will give out an error right away indicating that the ***initial*** and ***wait*** constructs are not supported by the tool. The problem is that there is nothing in the real world in hardware that would have the effect on an ***initial*** loop. Also the ***wait*** construct can not be thought of "***as- is***", in the hardware. There are ways to implement this logic by thinking in terms of a ***zero*** to ***one*** transition detecting logic that would sense the singal ***in*** becoming active and then use the output of this edge detecting logic to trigger the output "***out***" to get the equivalent function. (This can be implemented by a level sensing logic also). This is what we mean by thinking in hardware.

If we start discussing the unsupported synthesis constructs and indicate them one after another with some examples, the list would be too big. Instead let us concentrate on the constructs that are commonly used by ASIC designers to realize logic circuits. The above example was just for counter argument and just to show that such constructs are not possible to synthesize.

Digital Logic Building Blocks

The building units (like the bricks) of any complex or simple digital circuit are the various gates which in turn are designed using switching transistors. There are two types of circuits, the combinational logic and sequential logic. There are elaborate explanations about these two different types of logic in the *issue-1*of my paper. (The reader is advised to read the previous three issues to get continuity and to get the best out of these papers). A combinational circuit changes its outputs based on the changes in its inputs, right away. These are built using logic gates. On the contrary, the sequential logic needs a basic clock (this is true for the synchronous sequential circuits) and the output will change at the clock transition event. The output of such circuits depends on the inputs and also the current outputs. These are built using flip-flops (memory cell to store the current state of the circuit) and logic gates. A simple picturing of a sequential logic is to imagine some combinational clusters feeding at the input of the flops. The picture shown below is a simplistic representaion of a sequential logic circuit.



During the design of an ASIC, we are actually building millions of these kinds of clusters that would collectively generate the functionality that we are aiming for. Any ASIC would consist of such combinational, sequential logics, memories, statemachines (which are again nothing but the sequential logics), PLLs, clock distribution networks, IO pads, test logic (DFT) and many other necessary logic.

Defining Combinational Logic

In this section we will look at how to define a combinational logic in verilog that can be synthesized into the intended gate to give the expected functionality. A combinational cluster generating an output can be easily represented by a Boolean equation. Verilog supports a series of logical operators that can be used to write these equations. Readers are advised to refer to a text book in Verilog or the IEEE Verilog manual that gives the complete set of operators and their precedence in an equation.

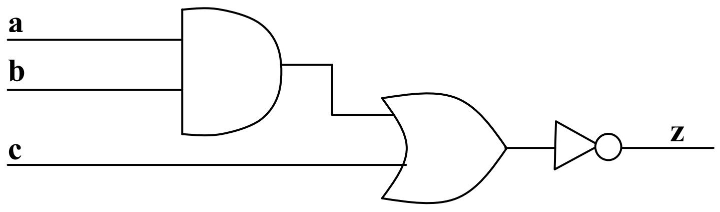
assign statement

Let us take the following example.

Example:2

```
module aoi (a, b, c, z);   // This is an AND OR INVertor logic
input a,b,c;
output z;
wire z;
assign  z = ~((a && b) || c);
endmodule
```

The above example uses too many parenthesis just to show the grouping clearly and not to confuse the reader with precedence issues. The usage of ***assign*** statement is one way of realizing a combinational logic. The above module written in Verilog will yield the logic shown below, after synthesis. The logic shwon below is not the output of any synthesis tool. I am giving this just for referentce and to idicate the logical equivalence. The synthesis tool will do optimization also to realize the logic with minmal functional gates so that the delay and area of implementaion is optimal. Any time ***a*** or ***b*** or ***c*** changes, the effect will be seen on the output ***z***.



always statement

The same logic that is given above can be realized using the code given in example-3 also. The coding style is a little different and the motivation for coding like this also is slightly different. It may be hard to appreciate why one would want to code the logic like this, but some prior experience in the ASIC design would let people realize the advantages of thinking in these lines.

Example:3

```
module aoiModified (a, b, c, z);
input a,b,c;
output z;
reg z;
always (a or b or c)
  begin
    z = ~((a && b) || c);
  end
endmodule
```

The code given in the example-3 yields exactly the same logic shown in the above diagram, after synthesis. Let us spend a little time in reviewing this code. The main difference you see between e.g-3 and e.g-2 is that the output ***z*** is declared as a register and there is, this ***always*** construct that is significantly different. Even though the node ***z*** is declared as a register, it is not synthesized as a flip-flop. (Registers are realized using flip-flops so that they can store the information sampled by them). It is declared as a register only to obey the syntax requirements of Verilog. The node ***z*** is still a combinational output just as shown in the diagram above. The variables shown inside the parenthesis next to the ***always*** statement are called the sensitivity list. The output ***z*** is sensitive to the three variables ***a, b*** and ***c***. If there are any changes in one of the three variables, that will get reflected in the output. Hence they are called sensitivity list. The key point here is that if we fail to specify any one of those variables in the sensitivity list, the effect of that missing variable will not be seen at the output node. This will lead to functional mismatches when we try to verify the logic using a test suite.

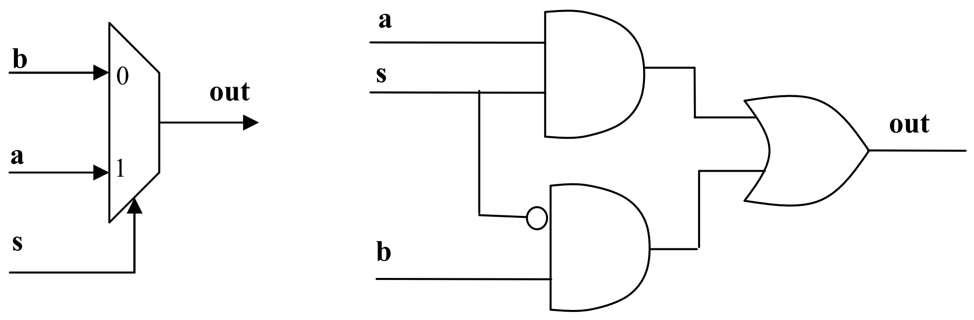
Multiplexer construct

Multiplexers, mux in short, are very common logic that will be seen in the logic design. Based on a selected signal mux will select one of the two inputs and provide it in its output. This is the function of a simple 2 to 1 mux. The code shown in example-4 does this mux function.

Example:4

```
module mux (a, b, s, out);

input a, b, s;   // inputs a,b and the select signal s

output out;

wire out;

assign out = s ? a : b;

endmodule
```

The code shown in e.g-4 is a commonly used mux implementation. When the select signal ***s*** has a value of ***1***, input ***a*** is selected and sent to the output. When the select signal is a ***0*** the input ***b*** is multiplexed to the output. A symbolic diagram and a logical diagram of the mux are shown below.



The same logic can be implemented using the code in e.g-5 also. Here I have used the ***always*** statement and sensitivity list to do the job. The synthesized results of both codes will be the same as shown in the figure above.

Example:5

```
module mux_modified (a, b, s, out);

input a, b, s;   // inputs a,b and the select signal s

output out;
```

```
reg out;
always (a or b or s)
  begin
    out = s ? a : b;
  end
endmodule
```

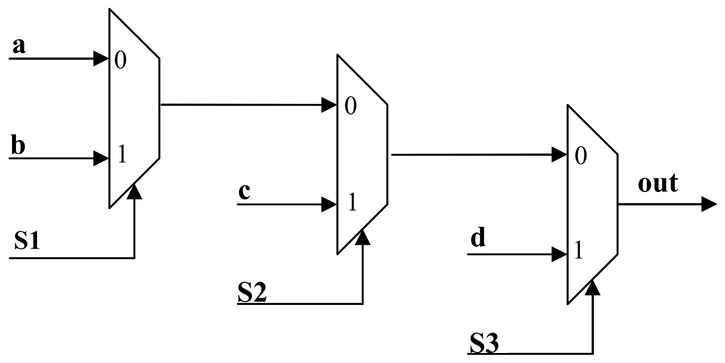
if-else statement

The ***if-else*** construct is one of the most common ways of implementing logic with the behavioral descriptions. The mux implementation that we discussed in the previous section can be done using this construct also as seen in the e.g-6.

Example:6

```
module mux (a, b, s, out);
input a, b, s;   // inputs a,b and the select signal s
output out;
reg out;
always (a or b or s)
  begin
    if (s) out = a;
    else   out = b;
  end
endmodule
```

This construct is used for designing priority encoders that look at one set of input condition first and then if that condition is not satisfied, looks at another condition and so on. The usage of ***if-else*** statements results in priority encoders. The priority encoders can be viewed as a series of muxes that are connected one after the other to create an ordered priority. A priority encoder can be visualized as in the figure below.



The ***always*** construct for this priority encoder using the ***if-else*** is given below in the e.g-7 in the next page.

```
Example:7
always (a or b or c or d or s1 or s2 or s3)
  begin
    if (s3)       out = d;
    else if (s2)  out = c;
    else if (s1)  out = b;
    else          out = a;
  end
```

The same logic can be implemented using the assign statement's mux construct also and this will still be synthesized as a priority encoder. Take some time to understand the idea of priority being implemented in an order. The top most priority is given to the select signal ***s3***. If this signal is set to ***1***, none of the other signals have any effect on the output. Only when ***s3*** is a ***0***, the other input signals get a chance to play a role in changing the output of the logic.

```
assign  out = s3 ? d :(s2 ? c :(s1 ? b : a));
```

case statement

The ***case*** statement is another way to build multiplexers or in general any combinational logic. The ***case*** statement by default generates priority encoder, traversing from the first case item to the bottom case items. The following two examples show the usage of ***case*** statements for implementing the same priority encoded mux we saw in e.g-7.

```
Example:8
always (a or b or c or d or s1 or s2 or s3)
  begin
    case (1'b1)
    s3 : out = d;
    s2 : out = c;
    s1 : out = b;
    default : out = a;
    endcase
  end
```

or

```
Example:9
always (a or b or c or d or s1 or s2 or s3)
  begin
    case ({s3, s2,s1})
    3'b100 : out = d;
    3'b010 : out = c;
    3'b001 : out = b;
    default : out = a;
    endcase
```

```
end
```

Generating priority encoder will end up in huge logic that will have input to the output propagation delays large. This is because the muxes are connected in a serial manner, and each of the combinational logic constituting the individual mux will take definite amount of time to generate its output. As a result of this, a combinational logic that has huge priority encoder structure will take more time to generate its output. If we try to use this kind of long paths in the sequential designs, we will have limitations in the frequency at which the logic can be operated. To realize fast operating logics, we need to have smaller propagation delays. There are some techniques the synthesis tools provide to take care of these kinds of problems. We will learn some of them in the later part of this paper.

Function statement

The ***function*** statement is used to synthesize combinational blocks that will return a value to either a vector or a scalar variable. It will take some inputs and generate an output which can be used in the other portions of the code by passing the output of the function. An example of the function is given below. This function generates the result of the addition of two 4-bit values and the carry input. This is a full adder implementation.

Example:10

```
function [4:0] fullAdder;
  input [3:0] a;      // inputs a and b are four bit values.
  input [3:0] b;
  input       cin;    // this is the input carry to the FA.
  begin
    fullAdder = a + b + cin;
  end
endfunction
```

Defining Sequential Logic

A sequential logic generally is constructed using a combinational logic that generates a set of outputs and a set of registers that remember the outputs of the combinational logic. As we discussed before we should imagine a sequential circuit as a combinational cluster feeding flip-flops. Sequential logics need a basic clock to sample the inputs at the D input of the flops. In Verilog sequential circuits are realized using the ***always*** statement with a special triggering condition defined based on the clock that it uses. Any sequential logic should have this switching condition in the construct as shown below.

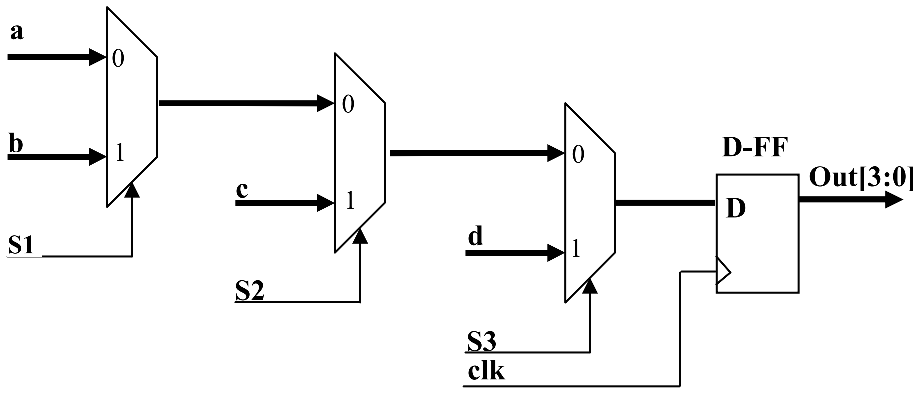
```
always @(posedge clk)  or  always @(negedge clk)
```

The key words ***posedge*** or ***negedge*** indicates that the sequential circuit uses the rising or the falling edge of the signal named as ***clk*** for it to make the transition form one state to the other. Since we agree that the input of the flop should be driven by a combinational logic, if we place any one of the code that we developed in the previous sections under this ***always*** loop, we can simply generate a sequential circuit that will sample the output of that combinational circuit every time the triggering condition arises. The triggering condition is either the positive transition or the negative transition of the clock. All the constructs that we used for the combinational logic can be used when realizing the sequential logic also. Another requirement for most of the sequential logic is that it should start in a known state when the logic starts functioning. Otherwise the outputs form it may not be predictable due to the very fact that the state may be unknown in the beginning. We will have a ***reset*** signal to the sequential circuit that will bring the flops in the design to known state (usually the reset will make all the flops to ***zero*** at reset) before any inputs are driven to it. Let us take a simple example below to see how to code a sequential logic in Verilog.

Example:11

```
module priorityEncodedMux (a, b, c, d, clk,
                             s1, s2, s3,
                             out);
input  [3:0]  a, b, c, d;
input         s1, s2, s3;
output [3:0]  out;
reg    [3:0]  out;
always @(posedge clk)
  begin
    case ({s3, s2,s1})
    3'b100 :  out <= d;
    3'b010 :  out <= c;
    3'b001 :  out <= b;
    default : out <= a;
    endcase
  end
endmodule
```

This e.g-10 is almost same as the e.g-7 and 8, except that the inputs and output are a 4 bit bus instead of a single bit. The output of the priority encoder is captured in four flops to make the logic a complete. Note that there are no reset signals in this code since the reset signal will not have any impact on the output. No matter what state the flops ***out*** are, the only controlling factors that will affect the output are the inputs and the select signals. Since ***out*** is not going as an input to this logic, as opposed to the definition of a sequential logic, there no impact from the initial state if the flops named ***out***. The following picture shows the representation of the function that will be inferred by a synthesis tool from the above code.



Counter

Counters are very common logics that everyone will be building in their ASICs. They keep tract of counts and let the other logics to look at it and take appropriate decisions. Counters are true sequential logics that have the feedback from the output to the input. The current state of the counter should be known to move the counter to the next state, which is one larger than the previous state. Take a look at the counter implementation in Verilog given in e.g.-11. In this e.g. there is an enable signal that enables the counter to move from its current state. When the enable signal is not present, the counter will not make any state transitions. Also the counter will roll over once it reaches its maximum count. And then start counting from the beginning as long as the enable signal is present. We have provided a reset signal also so that the counter will start from a known state instead of starting from any random number. This is a 8-bit counter. Please pay special attention to the usage of a non-blocking assignment in this example. The non-blocking assignment is specified by the operator "<=" in the equation. We discuss briefly about the blocking and non-blocking assignment in one of the following topics. Since this kind of Verilog coding describes the behavior of hardware at a register level, this is known as Register Transfer Level (RTL) coding method.

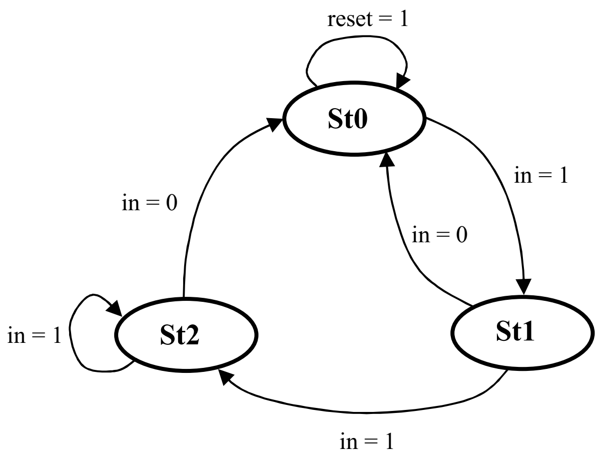
Example:12

```
module counter8Bit (clk, reset, enable,
                    count);
input         clk, reset, enable;
output [7:0]  count;
reg    [7:0]  count;
always @(posedge clk)
  begin
    if (reset)
      count <= 8'b0;            // non-blocking assignment is used
    else if (enable)
      count <= count + 1'b1;    // non-blocking assignment is used
  end
endmodule
```

State Machines

When building huge ASICs the state machine techniques of implementing the logic will be used extensively. State machines are nothing but sequencing engines that take a set of inputs, a particular clock and move their states from one to other in a deterministic manner. As the state machine (SM) moves through its states, the designer can generate the intended outputs using various states of the SM. By partitioning the design into smaller blocks, one could achieve better control over the logic and implement the intended function in a clean manner. SMs are commonly used practice in logic design. To show how we can build a state machine in Verilog, that can be synthesized with no problems, we shall consider a small state machine problem and deduce the Verilog code for it.

The figure shown below is called, the bubble diagram representation of the SM that we need to implement. The SM initially stays in the St0 when the reset signal is active and moves from St0 to other states based on the changes in the input signals. The function that we want to build here is an edge detecting logic that works synchronously with the clock that is provided to it. Whenever the input signal ***in*** makes a transition from ***zero*** to ***one***, the logic should generate a pulse with a duration of one clock. This is the requirement. This state machine is shown in the figure given below. This problem is not a complicated one that would need a SM to implement. But to demonstrate how to write a SM, I have used this example.

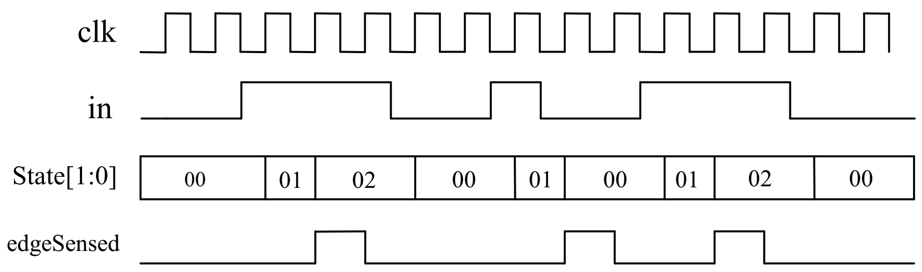


The above SM is coded in the example-13. There is an additional register viz. ***edgeSensed*** which is not actually needed. One of the state, bits (the bit[0]) of the SM itself will act as the one pulse signal indicating the rising edge of the input signal. I chose to register this bit into another flop and bring it as the output. This just shows how to code that output and also indicates that there can be multiple always @(posedge clk) sections in the code. Each of such sections indicates that there are that many sequential sections or groups of flops in the design.

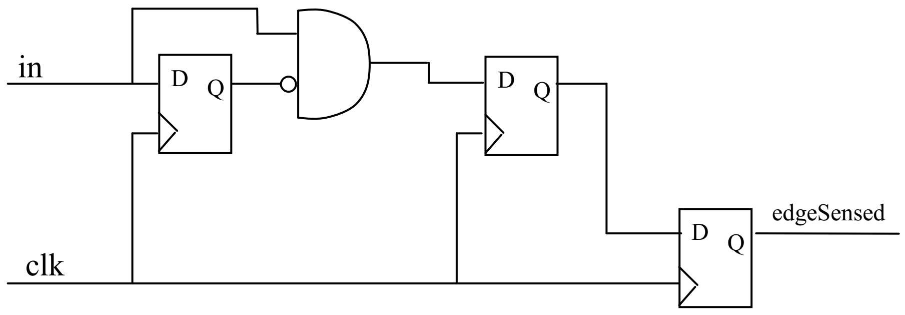
Example:13
```
module edgeDetect(
                 clk,
                 reset,
                 in,
                 edgeSensed);

input     clk, reset, in;
output    edgeSensed;

reg       edgeSensed;
reg [1:0] state;

parameter St0 = 2'b00,
          St1 = 2'b01,
          St2 = 2'b10;

always @(posedge clk)
  begin
    if (reset)
      state <= s0;
    else
      case (state)
      2'b00: if (in)
               state <= St1;
      2'b01: if (~in)
               state <= St0;
             else
               state <= St2;
      2'b10: if (~in)
               state <= St0;
      endcase
  end
```

```
always @(posedge clk)
  edgeSensed <= (state == St1);
endmodule
```

The code uses three registers, two for the SM and one for registering the SM bit[0] and sending it as the output. The output waveforms from this design are shown below. Every time there is a rising edge in the input signal ***in***, there is one pulse generated from the logic, thus functioning as an edge detecting circuit. Sine this logic is assumed to be synchronous, the input signal will be active at least for one cycle of the ***clk***.



As I mentioned before this is not a complicated problem to be done in a SM and let a synthesis tool generate the logic. We can deduce the logic by ourselves. The logic shown in figure below does the same function. The synthesis of the above code in e.g-13 will also give the following logic, without the effect of the reset signal.



Some suggestions

In this section, we will discuss some of the common problem that people would end up when they start coding in Verilog for synthesis. Though these are all valid Verilog constructs that would give outputs, they may not be helpful when one synthesizes the code.

No # for delay

Verilog can generate delays during simulation using the #delay construct. Depending on the scale that we define in the `timescale, we could get N units of delay before the right hand side is assigned to the left hand side. But in the synthesis world there is no notion like this delay.

The synthesis tools just disregard this construct. Look at the following Verilog assign statement.

```
assign #3 z = a && b;
```

In the above blocking assignment, the any change in the logical value of ***a*** or ***b*** will affect the ***z*** as seen by the equation. But the #3 will make this effect delayed by 3 time units. This can be thought analogous to the propagation delay of the AND gate. But in the gates or netlist world, there no such delay. The synthesis tool will connect the output of the AND gate directly to the output node named ***z***.

In the early ages of the HDL compilers, people used to use the #delay in the sequential logic assignments also as sown below.

```
always @(posedge clk)

  edgeSensed <= #1 (state == St1);
```

Here the one unit delay is used in a non-blocking assignment. This delay also achieves the same effect as seen before. The right hand side is evaluated and assigned to the left hand side variable one unit delay after the rising edge of the clock, which is the triggering signal. This delay can be seen as the ***clock-to-Q*** delay of a flop.

I recommend strongly not to use #delay in any portion of the synthesizable code. The improper accumulation of #delays in the assign statements will lead to wrong simulation results if not handled properly. In the sequential logic, though this #delay is not causing any adverse effects, it is truly unnecessary. The current days' compilers have advance schedulers that take care of any execution conflicts. So do not use #delays in your code, if you are designing a block that is to be synthesized. There may be places in test benches where this #delay constructs may found useful. I would even suggest that if we are designing synchronous logic and we are developing test bench for testing such designs, there are only very least requirements for using the #delays even in the test code.

One module – one file

When thinking about a function to implemented in an ASIC or an FPGA, try to imagine lots of smaller functions that would collectively build the top level requirement. With all these smaller modules, build the upper level blocks in a hierarchy. Hierarchical designs are very helpful to design, debug and implement in an ASIC. My advice is that use one module in one file, so that each file will have a relevant name indicating the module and concise and complete within itself. Whenever needed you may instantiate other children modules into the parent modules. Too many levels of hierarchy is also not good as it makes tracing across hierarchy difficult.

Try to split the control path design and the data path design into at least two different modules, that will help at a later stage when doing floorplanning of the design. By doing this, data path intense logic can be placed and routed with more care to meet the timing requirements.

Latch inference

If the recommendations for good style of coding are not followed we may end up in unintentional logic being inferred by the synthesis tools. Combinational latches are one such very dangerous logic that may get inferred by many synthesis tools. This mostly ends up because of improper clause definitions in the ***if-else, case*** or some other constructs. It is always a good practice to define a default clause in such situations, so that the tool can define a known state for the flops, thus avoiding any latch inference. Usually all the synthesis tools give out warning messages indicating such latch inferences.

Blocking, non-blocking statements

Readers are encouraged to look at the Verilog manual or other text books to understand the differences between a blocking and a non-blocking statement. My recommendation on this is to use blocking statements in the places when we are targeting a combinational logic, no matter it is done in an ***assign*** statement or in ***always*** loop. In places we want to infer a sequential logic using ***always @(posedge/negedge clk)***, use a non-blocking statement always. A non-blocking statement is the one that has the symbol <=.

parallel_case and full_case notion

This notion was initially started by Synopsys and now it is being followed in all the synthesis tools. As we have already mentioned, the ***case*** statement in the Verilog is treated as a priority encoder. Most of the time the ***case*** statement will have a ***default*** clause where there will be some assignments to the variables. This value is given to the respective variables if none of the case clauses are true. The order of priority in the case statement starts from the first case item in the ***case*** section of the code. To prevent priority encoders being inferred, the synthesis tools use a compiler directive called ***parallel_case*** that will let the tool build parallel logic instead of priority encoders. This will avoid the logic serial muxes and try to implement the combinational logic without the priority encoded structure. Though there may be more logic cells in this way of implementing, the propagation delay of the combinational logic will be smaller that the priority encoder implementation. The same effect can be achieved by using the ***if*** statement without the ***else*** clause. One thing that we have to remember in such implementation is that we are instructing the synthesis tool that all the input combinations are parallel. This means that the input signals that appear in the ***case*** statement exist in such a manner that no two case clauses are true at the same time. If we know that this is true, we can instruct the tool to treat the ***case*** as a parallel case. Otherwise there will be functional problems in the logic. If we take the e.g.-11 and change it to be a parallel case, we need to be sure that the select signals ***s1, s2 and s3***are mutually exclusive. Any comment starting with the key word ***synopsys*** will be take as a compiler directive. Here in this case we place the comment ***//synopsys parallel_case*** in the case statement so that the compiler will understand that this case statement has to be treated as a parallel case. *(The result from this synthesis is left to analysis by the reader. Synthesis this code and the code from e.g-11 and study the differences in the resulting netlist)*

Example:14

```
module parallelCaseMux (a, b, c, d, clk,
                    s1, s2, s3,
                    out);
input  [3:0]  a, b, c, d;
input         s1, s2, s3;
output [3:0]  out;
reg    [3:0]  out;
always @(posedge clk)
  begin
    case ({s3, s2,s1})  // synopsys parallel_case
    3’b100 :  out <= d;
    3’b010 :  out <= c;
    3'b001 :  out <= b;
    default : out <= a;
    endcase
  end
endmodule
```

Now we will take the ***full_case***. This a little bit advanced and tells the compiler that the combinations given by the designers are the only possible combinations in the design. If we take the previous e.g., the ***case*** statement has a three bit variable in it. It can have eight possible values ranging from 3'b000 to 3'b111. We are defining the behavior for the combinations 3'b100, 3'b010 and 3'b001 and for the other five possible values, the output is defined by the default clause. Let us assume that our circuit is in such a way that either one of the three select signal is always a ***1***. In such a case, the default clause will never get exercised as any one of the select signals will be active, not allowing the output to go to the value of ***a***. In such a case, we can instruct the synthesis tool that there are only three possible combinations and other combinations of the ***case*** variable do not exist in the design. By knowing this, the tool decides to optimize the logic only for the valid combinations and does not worry about other possible combinations. We need to be absolutely sure that the undefined combinations do not arise in the design. Otherwise it will lead to mismatch in the functional behavior. In this case, the default clause has no meaning. Take a look at the e.g.-15 to see the ***full_case***.

Example:15

```
module fullCaseMux (a, b, c, d, clk,
                    s1, s2, s3,
                    out);
input  [3:0]  a, b, c, d;
input         s1, s2, s3;
output [3:0]  out;
reg    [3:0]  out;
always @(posedge clk)
  begin
    case ({s3, s2,s1})  // synopsys full_case
    3'b100 :  out <= d;
    3'b010 :  out <= c;
    3'b001 :  out <= b;
                 // no default definition. input a is never used
    endcase
  end
endmodule
```

Both parallel and full case can be used at the same time like

// synopsys parallel_case full_case

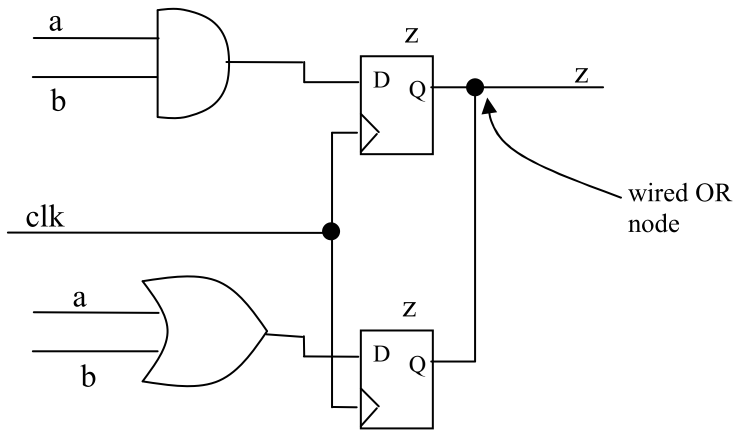
But the user has to be extremely careful in using this compiler directive. The above pair is said to be "evil twins". Only if the designer is sure of what he needs or what he knows about the design, he should use these compiler directives together. Think a little bit more about these compiler directives and plan to do some experiments with the usage of parallel and full case to see how the tool differentiates them. It will be interesting to see what the synthesis tool infers for such usages.

Wired OR inference

This is another dangerous inference that will lead to chip failures. When we define sequential logic, each of the flop assignments should be within one and only ***always*** loop. If we define the flops in two different ***always*** loops, that will lead to inference of two flops with the same name, thus shorting the outputs of the two flops. This will function like a wired OR flops that will switch for both the conditions defined in the two always loops. This will lead to faulty behavior of the logic. This is one of the common mistakes many would do. The following e.g. code and the figure explains this.

```
Example:16

reg z;

always @(posedge clk)

      z <= a && b;

always @(posedge clk)

      z <= a || b;
```

In the above section of the code, the register ***z*** is defined in two ***always*** loops. Hence there will be two inferences of the register ***z*** during synthesis. The inferred logic will look like this. Obviously, the output ***z*** will not behave properly because two out of the four combinations of the inputs will have conflicting outputs at the ***Q*** pin of the flops.



Comments

Place plenty of comments in the code that will make anyone easily understand the idea behind your approach. You yourself will be a stranger to your own code after few months of break form that design. Your comments will help yourself at that time understand the flow. Verilog has one line and blocked comments.

```
// This is a comment

/* This is a block of comment that

   can span through many lines */
```

Some more points

- While using ***casex*** or ***casez*** statement be cautious to see if they are really parallel or full

- The logical equivalence check syntax in Verilog (===) is not possible to get synthesized as there are no way in the hardware to check the don't care value of the bits, indicated by 1'bx. However the equality check (==) is synthesizable.

- Avoid the usage of ***`include*** in the Verilog files that describe the design. By doing this, each of the modules can be synthesized by itself. In case of any instantiated modules, you need to provide those modules to the synthesis tool separately.

- Use a separate file where all the ***`define*** macros that are common to all the design files are defined. For the local macros, plan to use the ***parameter*** construct.

- It is advised to have flip flops at the outputs from large modules (here, by saying "large modules", we mean large floorplannable blocks in the design). When you have flops at the outputs, the modules that receive these outputs can use them in their combinational logic and still have the entire cycle of the clock to resolve them. This will help during synthesis to get better timing results. Otherwise, if you send a combinational output to another module, which again passes that signal into its portion of another combinational logic, you will have to do time budgeting to get the timing work cleaner.

- As much as possible design the top level with two or three hierarchies. Current generation synthesis tools handle larger designs very easily and hence we can have larger partitions of the design.

Disclaimer & Conclusion

In this paper, I have not considered all the possible constructs of Verilog to explain various issues and also I have not covered all the synthesis issues that one would face during the chip design. The examples given here should be taken as guidelines and a lead, for those who are in the initial stages of learning Verilog to design ASICs or FPGAs. Some of the synthesis related comments may be difficult to digest in the beginning. But once if you get some familiarity with the synthesis flow and the tools, those points will make sense. You are welcome to get in touch with me to get more clarifications.

Share what you have learnt through this, to others by conducting a group discussion or a seminar so that others also can benefit through this effort. Send your feedback and questions to my email ID, which you can get from the HOD, CS Dept. Your feedback is very important to continue and improve this effort.

— 00 —

Trademarks note:

Synopsys and "Desgin Compiler" are registered trademarks of Synopsys Inc., Mountain View, CA, USA
Build Gates is a registered trademarks of Cadence Inc.
Leonardo Spectrum is a registered trademarks of Mentor Graphics.