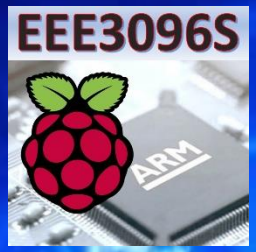


Get Verilog ready!

Hardware Description Languages

a taste in 5-lectures Part3

Embedded Systems II

L39

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Outline of Lecture

- Thinking activity review
- Writing the counter module
- Testing the counter module (test #0)
- Verilog simulation and some replication commands
- Counter Testbench and running tests

Where we were at…

Towards a Testbench Example…

Verilog 4-bit counter with testbench

The test bench we will only get to later once we have something to test!

Homework / Thinking Point

REVIEW OF HOMEWORK

- Next lecture we will focus on developing a 4-bit counter.
- Read over the description on the next slide and thinking about how you would develop a Verilog module to implement this
- Next lecture we will finish off this activity in class

A 4-bit counter!

Now let's get to viewing the sample solution for the design…

Counter Design Thoughts

You should have contemplated these aspects… that you should do before you jump into the actual coding.

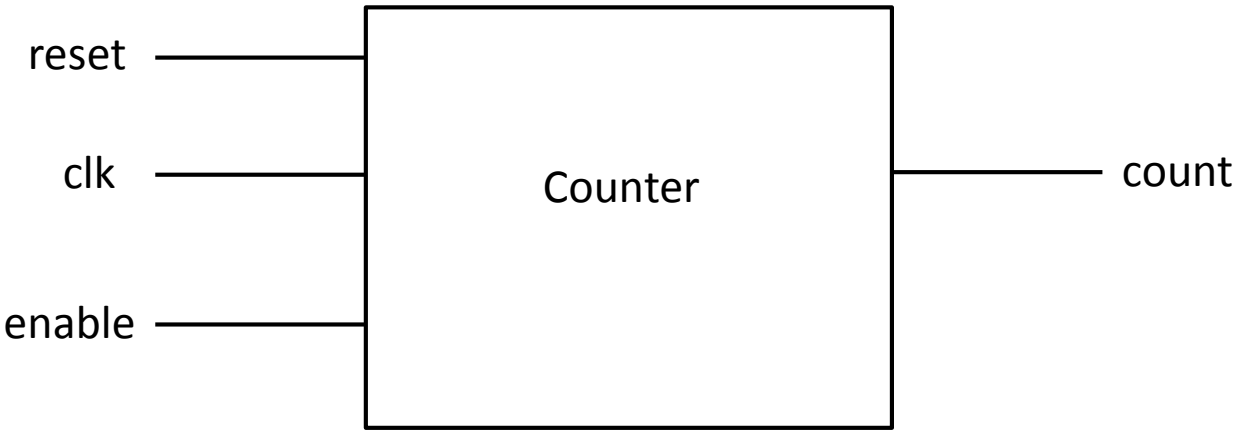
Let's think about what a 4-bit counter needs and how to implement it…

(1) A module

(2) interfaces: some inputs… reset and clock

(3) interfaces: an output… count value

(4) Maybe further embellishments … like enable line



Step 1: Write up the module interface

(Hint: start with keyword **module** and add ports)

Have a look at your own code or write a quick solution

Sample solution …

Step 1: Write up the module interface

```
//----------------------------------------------------
// Design Name : counter
// File Name   : counter.v
// Function    : 4 bit up counter
//----------------------------------------------------
module counter (clk, reset, enable, count);
 // Define port types and directions
 input clk, reset, enable;
 output [3:0] count;
 reg [3:0] count;
 more stuff is needed
endmodule
```

On towards Step 2…

Step 2: Implement the insides by instantiating built in modules in this new module.

```
//----------------------------------------------------
// Design Name : counter
// File Name   : counter.v
// Function    : 4 bit up counter
//----------------------------------------------------
module counter (clk, reset, enable, count);
 // Define port types and directions
 input clk, reset, enable;
 output [3:0] count;
 reg [3:0] count;

 your master piece here!!!   …
```

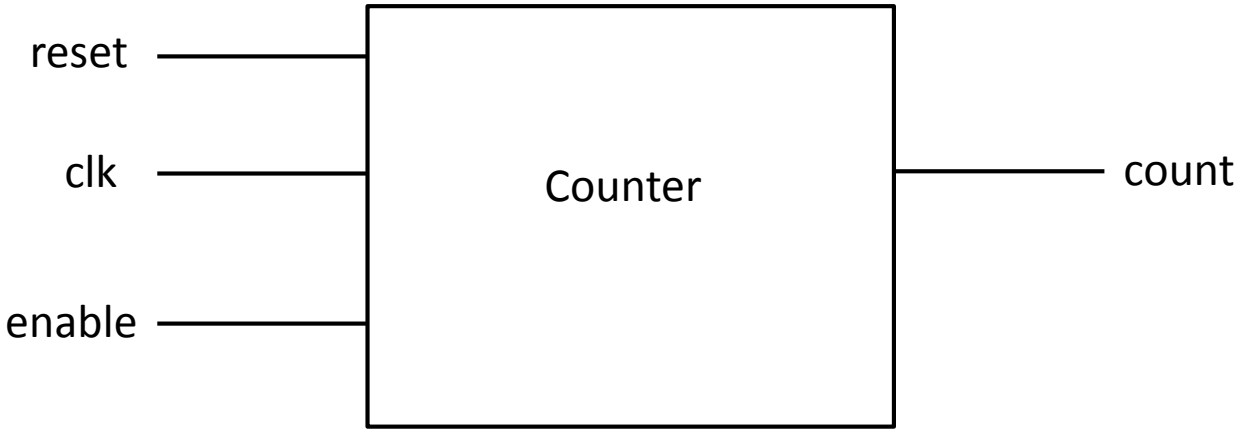
```
endmodule
```

Have a look at your own code or write a quick solution

Step 2: Implement the insides by instantiating built in modules in this new module.

```
//----------------------------------------------------
// Design Name : counter
// File Name   : counter.v
// Function    : 4 bit up counter
//----------------------------------------------------
module counter (clk, reset, enable, count);
 // Define port types and directions
 input clk, reset, enable;
 output [3:0] count;
 reg [3:0] count;
 always @ (posedge clk)
 if (reset == 1'b1) begin
   count <= 0;
 end else if ( enable == 1'b1) begin
   count <= count + 1;
 end
endmodule
```

Count Module Defined



Let's check if it compiles in iVerilog

Icarus Verilog

- With iVerilog you basically need a good text editor
- Should install gnuplot too, there are ways to graph waveforms

Verilog compiles the .v code into an executable. To do so:

iverilog -ooutputfile inputfile.v
Generates an executable file called *outputfile*

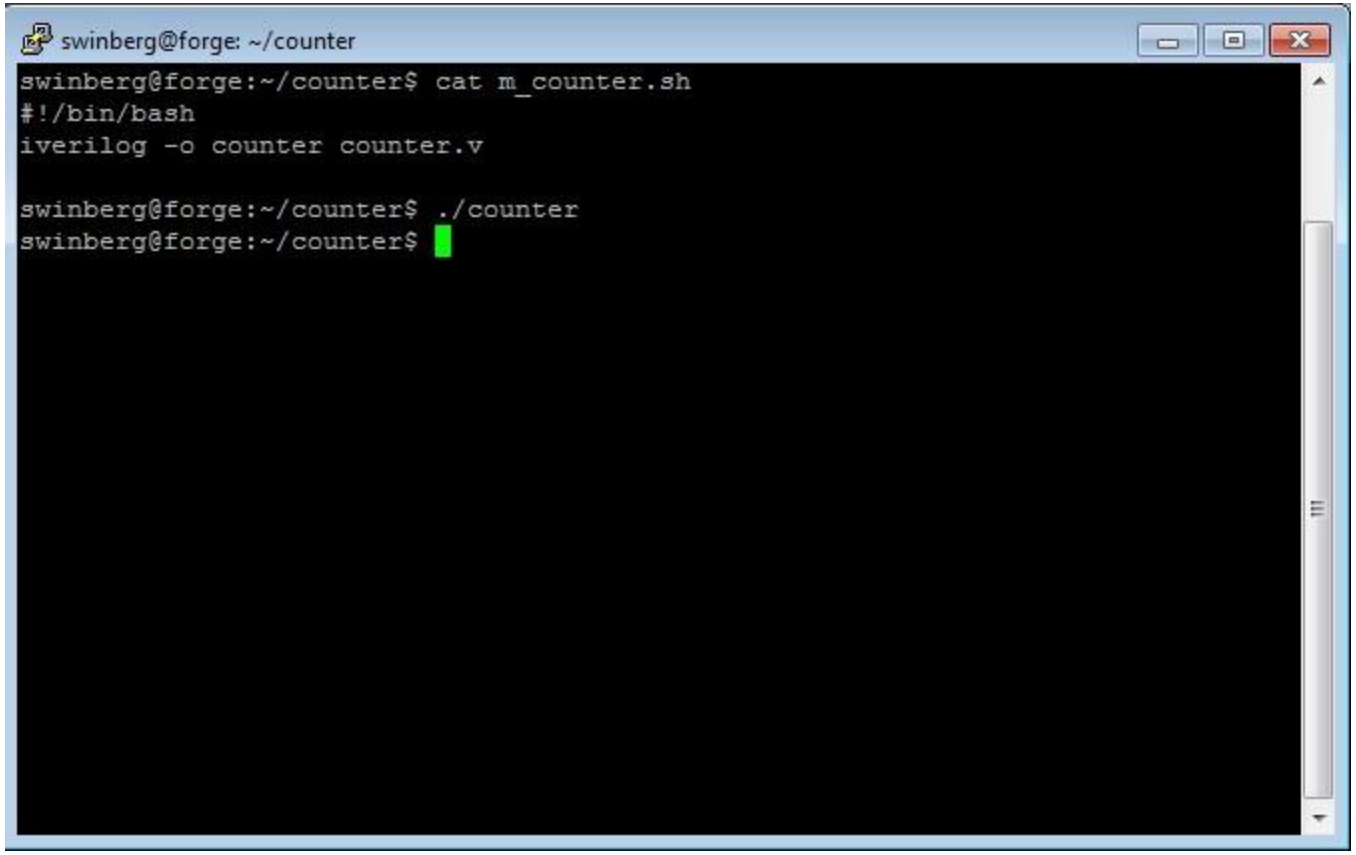
So lets do: *iverilog -ocount count.v* Or run m_counter.sh in the example zip file

And amazingly we see a counter file generated… what…

Nothing!
Because we don't have a testbench

Ooooh how fun! What exciting stuff will happen if we run it?!!!

Counter



Here is the rest of running counter on it’s own… Not too surprising.

Simulation Command

- Running the module that is just doing counting will of course not show anything interesting
- So we what we need are some simulation command, to write values to ports, to print out information, and do things like delays

Simulator (and Replication) Commands

- The following constructs are provided to help with doing simulations
  - **timescale** to specify timescale to use
  - ***#n*** delay for *n* timescale periods
  - **initial** block for a module (e.g. to initialize register values)
  - **$display** works like printf
  - **$monitor** command to monitoring changes to lines and displaying pin values whenever they change
  - **for** loops
  - **repeat** just repeats a block a number of times
  - **$finish** to exit the simulation

Some of these command (e.g. initial and for) can be synthesized but it depends on your tools and Verilog version. For example the initial block is often not synthesized for Verilog95.

Timescale

- Syntax:

  `timescale time_unit / time_precision

  e.g. timescale 1ns / 1ps
- The timescale can be put into a file to indicate to the simulator's precision
- Finer (smaller) timescale → slower but more accurate simulation
- In the example above:
  - time_unit defines each unit of time in the simulation. In the example each unit of time is 1ns (one nanosecond). The # delays are in time_unit quantities.
  - time_precision defines the precision (smallest unit of time) at which the simulator will run (i.e. the working of gates).
- Time_unit should be substantially greater than time_precision (100 to 1000 times greater is recommended). Remainers of time_unit/time_precision will be rounded off to the level of precision closest to the time_unit.

Delay

- The # operator is used to indicate a delay
- For example

  #10

  will delay for 10 time units while other things in other instantiated modules might be happening
- This can be used within the implementation of a module.
- Put it on the
  - left of a statement: to delay before the statement is performed, or put it
  - right to statement: to delay after the statement is performed.

Initial block

- The initial block is like a constructor for a Verilog module in simulation.
- It activates when the module first starts.
- Used in simulation to set up conditions and to implement test benches.

```
module testbench; // top level module
 wire  a, b, x; // set up some signals
 add  myadd(a,b,x); // module to test
 myAnd_tb1  tb(a, b, x); // use this test
endmodule;

module myAnd_tb1(a,b,x);
  input a,b;
  output x;
  reg         a, b; // registered inputs
  initial begin
     // log these signals as follows:
     $monitor ($time,
          "a=%b, b=%b, x=%b", a, b, x);
     // exercise the signals
    a = 0; b = 0;
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
    #10 $finish;  // tell simulator to quit
  end // end initial
endmodule
```

$display

- Use the $display command to print the current simulation time ($time) and/or the values of signal(s) at that point in time.

Example:

```
initial begin
 clock = 0;
 enable = 1;
 reset = 1;
 #10 enable= 0;
 #8  $display($time, ", Clock is %b",
              clock);
 $finish
end
always begin
   #5 clock = ~clock;
end
```

Displays to console:

12, Clock is 1

Why this is 1 and not 0?

Because this will be running, initial and the always block will run together

Formatting for $display:

%d : deciman %b : binary

%h : hex %f IEEE float if supported

$monitor

- Monitor has exactly the same structure as the $display command
- NOTE: Most simulators allow only one $monitor command to be active at any time in a simulation.

Example:

```
module monitorex ();
 reg clock, enable, reset;
initial begin
 $monitor("%g\t, Clock is %b",
          $time, clock);
 clock = 0; enable = 1;
 reset = 1;
 #10 enable= 0;
 #5  $finish;
end
always begin
   #5 clock = ~clock;
end
endmodule
```

Displays to console:

```
0     , Clock is 0
5     , Clock is 1
10    , Clock is 0
15    , Clock is 1
```

For loops

- These need to be used with care when you are placing them in non-simulation parts of the code – in such a case they operate as logic replication
- For example, this for loop is used as logic replication:

```
integer i;
for (i=0; i<=99; i=i+1) a = a*i;
```

would generate:

```
assign a = a * 0;
assign a = a * 1;
assign a = a * 2;
...
assign a = a * 99;
```

And will give you an error to say there are multiple concurrent drivers assigned to register a (because these lines will run concurrently)

For loops can be used (more safely) in simulation too, and you can make them behave more sequentially by using the delay operation, e.g.

```
integer i;
for (i=0; i<=99; i=i+1)
    a = a*i; #10
```

The above should run perfectly happily as it can at least be simulated and shouldn't have multiple drivers assigned to a.

repeat (*n*)

- The repeat command simply repeats a line a number of times. It can also be used for replicating logic.
- It is synthesizable.
- You can't tell what the index is, unless you define your own compile-time counter.

Example:

```
repeat (10)
begin
  #5 clk = !clk;   // toggle the clock 10 times
end
```

$finish

- The $finish command merely stops the simulation
- You can put $finish in various parts of your code, e.g. to stop operation if an exception is detected

Now you're equipped

With the basic Verilog commands that you need
to test the counter example…

TODO

Complete the design of the counter testbench.
Think of the test vectors to write to the ports and what outputs you should get.

(it's only about 5-10 minutes of pen-on-paper effort)

Shortly we will explore simulation commands in Verilog and show you the testbench solution.

Counter Testbench (v1)

Here is the basic starting point to the 4-bit counter testbench …

```
/* 4-bit counter Test bench version 1
   This just hooks up the test bench  */

module counter_tb;          // this will become the TLM
  reg clk, reset, enable; // define some regs, like
                          // global vars for module
  wire [3:0] count; // just need a wire to hook
                    // up count as it is a registered
                    // output of the counter module
  // instantiate the counter (U0 = unit under test)
  counter U0 (
    .clk    (clk),
    .reset  (reset),
    .enable (enable),
    .count  (count)
  );
endmodule
```

Any ideas what will happen when we run this?! …

Right, nothing will happen!
Now you can add quickly some simulation code to make it show and do something

Counter Testbench (v2)

Here is the basic starting point to the 4-bit counter testbench …

```
// 4-bit Upcounter testbench
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] count;
  // instantiate the module:
  counter U0 (
  .clk    (clk)    , .reset  (reset),
  .enable (enable),.count  (count)  );
  initial
   begin
    // Set up a monitor routine to printing out pins we are interested in...
    // But first do a display so that you know what columns are used
    $display("\t\ttime,\tclk,\treset,\tenable,\tcount");
    $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,count);
    // Now excercise the pins!!!
    clk = 0;
    reset = 0;
    enable = 0;
    #5 clk = !clk;    // The # says pause for x simulation steps
                      // The command just toggles the clock
    reset = 1;
    #5 clk = !clk;    // Let's just toggle it again for good measure
   end
endmodule
```

Any ideas what will happen when we run this?! …

Output of counter_tb2

Output:

```
time,    clk,     reset,   enable,  count
   0,    0,       0,       0,        x
   5,    1,       1,       0,        0
  10,    0,       1,       0,        0
```

So the results are looking more promising, but we don't see count counting! That's because all we have done is a reset, we haven't put reset low and continued to toggle the clock. So let's add that…

Counter Testbench (v3)

Here is the basic starting point to the 4-bit counter testbench …

```
// 4-bit Upcounter testbench
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] count;

  counter U0 (     // instantiate the module
  .clk    (clk),.reset  (reset),.enable (enable),.count  (count));

  initial
  begin
    // monitor routine to keep printing out the pins we are interested in...
    $display("\t\ttime,\tclk,\treset,\tenable,\tcount");
    $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,count);
    // Now exercise the pins!!!
    clk = 0; reset = 0; enable = 0;
    #5 clk = !clk;      // The # says pause for x simulation steps
                        // The command just toggles the clock
    reset = 1;
    #5 clk = !clk;      // Let's just toggle it again for good measure
    reset  = 0;         // Lower the reset line
    enable = 1;         // now start counting!!
    repeat (10) begin
       #5 clk = !clk;   // Let's just toggle it a few more times
    end
  end
endmodule
```

Any ideas what will happen when we run this?! …

Neat! Nice comments ☺

Output of counter_tb3

Output:

```
time,    clk,     reset,   enable,  count
   0,    0,       0,       0,        x
   5,    1,       1,       0,        0
  10,    0,       0,       1,        0
  15,    1,       0,       1,        1
  20,    0,       0,       1,        1
  25,    1,       0,       1,        2
  30,    0,       0,       1,        2
  35,    1,       0,       1,        3
  40,    0,       0,       1,        3
  45,    1,       0,       1,        4
  50,    0,       0,       1,        4
  55,    1,       0,       1,        5
  60,    0,       0,       1,        5
```

So that pretty conclusively proves that the counter is working. Of course to be totally sure you would need to run it past the count of 15 to see that it wraps properly etc.

Sample Code Solutions

- Here are the sample codes for the 4-bit counter project which you can also get from the Vula site

The Next Episode…

Lecture L40

- Bit shifting and reorganizing
- The case and other useful Verilog constructs