**Verilog Lab 5 (ECS1005)**

**Objectives:**

The objectives of this lab are as follows

* How to model edge triggered Flip Flops.
* Understand blocking and non-blocking statements.
* Behavioural modelling of shift registers and counters.

**Reminder: Fixing the waveforms!**

When you click the Run button, it will compile. If it opens up the waveform, then you can simply skip the next step. If the wave form does not open, then click the shell screen, right next to the console on the right side of the screen as shown below.

A screen shot of a computer

Description automatically generated

On the shell, copy and paste the following command and enter:

chmod +x run.sh && ./run.sh

This command will re-compile the project and the waveform will launch as well.

**Task 1: Modelling a rising/falling edge triggered D-FF with an asynchronous/synchronous reset**

In the last lab we saw two procedural blocks (**initial** block and **always** block) in the highest abstraction level of Verilog modelling, i.e., behavioural modelling. *We saw that the* ***initial*** *block executes exactly* ***once*** *while* ***always*** *block is* ***always*** *active as long as the simulation is active*. The initial block is used only in the testbenches to initialize values of various variables. The always block can be used in both test benches and the design (or the DUTs). In Verilog lab04, we saw several examples of how the always block can be used to model combination logic elements, like multiplexers, decoders etc. Today, we are going to shift gears and see how the always block can be used to model the edge triggered sequential blocks or the D-FlipFlops (D-FF).

Let’s start modelling a D-FF <https://replit.com/@AyeshaKhalid5/Verilog-Lab4-Task1#D_ff.v>

To model a rising edge triggered D-FF, the sensitivity list of the always block MUST MENTION its triggering edge as shown below. (Referring to the section 9.1.2 of the book, we have)

Text, letter

Description automatically generated

Since we are using the elaborate description convention for inputs and outputs, our listing will slightly differ as follows. Since Qn is simply an inversion of Q, we also generally don’t need that.

Graphical user interface, text, application, chat or text message

Description automatically generated

**(For now, ignore why we are using <= instead of** **=)**. Remember we need to redefine the output Q as a reg as it is assigned inside a procedural always block. The always block above will model a rising edge triggered D-FF. The body of always block will be executed every time the triggering edge of the clock comes as that is added in the sensitivity list of the always block. For a negative edge triggered D-FF we simply replace the triggering event sensitively list by a negedge instead of a posedge.

|  |  |
| --- | --- |
| D Flip-flop symbol, edge |  |
|  |  |

To model a propagation delay of 1 time unit in the Flipflop, we can use Q<= #1 D;

Let’s now focus on the testbench. To model a clock, we first define a localparam (a local constant) using the keyword **localparam**. The clock period is here fixed to 10 time units. The clock is initialized to 0 using an initial block and toggled after every half clock time period using an always block.

A picture containing text

Description automatically generated

For observing the response of D-FF, the input D should respect the setup time(tsu) and the hold time(th) of the D-FF. To be careful, we change the input signal D with a delay of #1 time unit after the triggering edge of the clock so that it is taken at the next triggering or **rising** edge of the D-FF.

A screenshot of a computer code

Description automatically generated

Let’s now try to comprehend the response. The input D in x or undefined (red in color) till we have the first 2 posedges of the clock. After that it is made 0, 1 time unit after the second posedge. At the next triggering edge of the clk the Q output of the flip flop reflects 0 at the output too. Similarly, when D changes after 2 posedges, the Q output updates at the next triggering edge of the clock.



Let’s now add an asynchronous reset to this D-FF, we change the procedural block as follows.

|  |  |
| --- | --- |
| D-type flip flops |  |

Next we add a synchronous enable signal (called En) to this D-FF. Notice that the sensitivity list of the D-FF does not change as the enable signal is checked only if there is a posedge clock or triggering event.

|  |  |
| --- | --- |
|  |  |

Finally, convert the single bit D-FF to an n-bit register, we can convert the input/outputs and the regs in the module (let’s say a 4 bit D-FF).

Text

Description automatically generated with medium confidence

Let’s try making appropriate changes in the testbench to mimic the following response.

* As the simulation starts make

En=#1 0; asyncReset=#1 1; D=#1 4'd0;

* After three posedges have passed, deassert asyncReset, i.e., asyncReset=#1 0;
* After 2 more posedges have passed, make En=#1 1;D=#1 4'd10; After the next posedge simply make En=#1 0;
* Finish the simulation after 4 more posedges.

Match your response to the waveform below

A green and blue lines on a black background

Description automatically generated

**Blocking vs. Nonblocking in Verilog**

The concept of Blocking vs. Nonblocking signal assignments is a unique to hardware description. **The main reason to use either Blocking or Nonblocking assignments is to generate either combinational or sequential logic.** In software, all assignments work one at a time, so for example in the C code below:

Graphical user interface, text

Description automatically generated

The second line is only allowed to be executed once the first line is complete, this is an example of a blocking assignment. **One assignment blocks the next from executing until it is done**. In Verilog there is logic that can execute concurrently or at the same time as opposed to one-line-at-a-time and there needs to be a way to tell which logic is which.

A screenshot of a computer

Description automatically generated with medium confidence

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated |

The always block in the Verilog code on the left uses the Nonblocking Assignment (<=), which means that all the **three assignments will be simulated to execute in parallel**. Hence it will take 3 clock cycles for the value 1 to propagate from *r\_Test\_1* to *r\_Test\_3*. In the always block on the right, the Blocking Assignment is used. Here, all the three lines of code are executed one after the other, hence the value 1 will immediately propagate to r\_Test\_3 when the triggering edge of the clock comes for the first time*.*

*The Blocking assignment immedia*tely takes the expression value on the right-hand-side and assigns it to the left-hand side. *The non-Blocking assignment* stores the right-hand side expression values into temporary location and assigns them to the LHS variables as the triggering edge comes. Hence it does not matter which order they are executed as the temporary values saved are used to assign values to the LHS variables.

Read section 8.1.2 from the reference book.

Text, letter

Description automatically generated

Text

Description automatically generated with medium confidence

Here's a good rule of thumb for Verilog:

* If you want to create sequential logic use a clocked always block with Nonblocking assignments (<=).
* If you want to create combinational logic, use an always block with Blocking assignments (=).
* **DO NOT MIX** the blocking and non-blocking assignments in the same always procedural block.

**Task 2: Modelling Shift register based counters**

Flip-flop is a 1 bit memory cell which can be used for storing the digital data. To increase the storage capacity in terms of number of bits, we have to use **a group of flip-flop**. Such a group of flip-flop is known as a **Register**. The n-bit register will consist of n number of flip-flop and it is capable of storing an n-bit word.

Consider a 4 bit shift register that takes a single bit serial data in for FFA and every triggering edge of the clock shifts the register contents to the right. The basic data movement is shown below.

|  |  |
| --- | --- |
|  |  |

The 4 bit parallel data output is taken as a **state** of the register. D-FFs and shifting operation can also be used to construct simple counters. A [**Counter**](https://en.wikipedia.org/wiki/Counter_(digital)) is a circuit which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. Counters are used in digital electronics for counting purpose, they can count specific event happening in the circuit.

A Construct the following two shift register based counters.

* **A six-bit ring counter with D Flip-flops with the following diagram**Diagram

  Description automatically generated

A ring counter is a circular shift register that is initiated such that only one of its flip-flops is the state one while others are in their zero states.

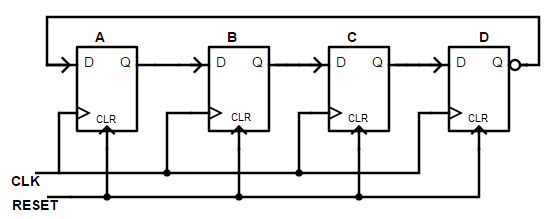
A ring counter is a [shift register](https://en.wikipedia.org/wiki/Shift_register) (a cascade connection of [flip-flops](https://en.wikipedia.org/wiki/Flip-flop_(electronics))) with the **output of the last one connected to the input of the first**, that is, in a ring. Typically, a pattern consisting of a single bit is circulated, so the state repeats every n clock cycles if n flip-flops are used.

Fork up the project used in task 1 for this task 2. That can be your starting point. Construct a 6 bit D-FF register with an asynchronous reset and a synchronous enable (En) signal as done in the Task 1, the input data D is not needed in this case. Since the counter starts with a 6 bit value 6'b100000, make that the reset value of the register. When the En signal is asserted (or high) the counter should simply keep shifting its values to right with its older LSB replaced to become the new MSB of the register, as shown below. Simulate in the test bench, first generate a reset signal and then En should be make 1 for as long as the counter needs to be kept working. Insert the waveform below.

A screenshot of a computer code

Description automatically generated

* **A 4-bit Johnson counter with D Flip-flops with the following diagram**



A [Johnson counter](https://en.wikipedia.org/wiki/Johnson_counter) is a modified ring counter, where the **output from the last stage is inverted and fed back as input to the first stage**. The register cycles through a sequence of bit-patterns, whose length is equal to twice the length of the shift register, continuing indefinitely.

**Hints:** Notice that the reset value of a Johnson counter is all zeros (4’b0000). We add the inverted output of the LSB of the counter to be become the input of the MSB.

A computer code with text

Description automatically generated with medium confidence

Verify the following table of states for Johnson counter. Since it’s a 4-bit Johnson counter we should have a total of 2x4=8 states. Insert the waveform below.

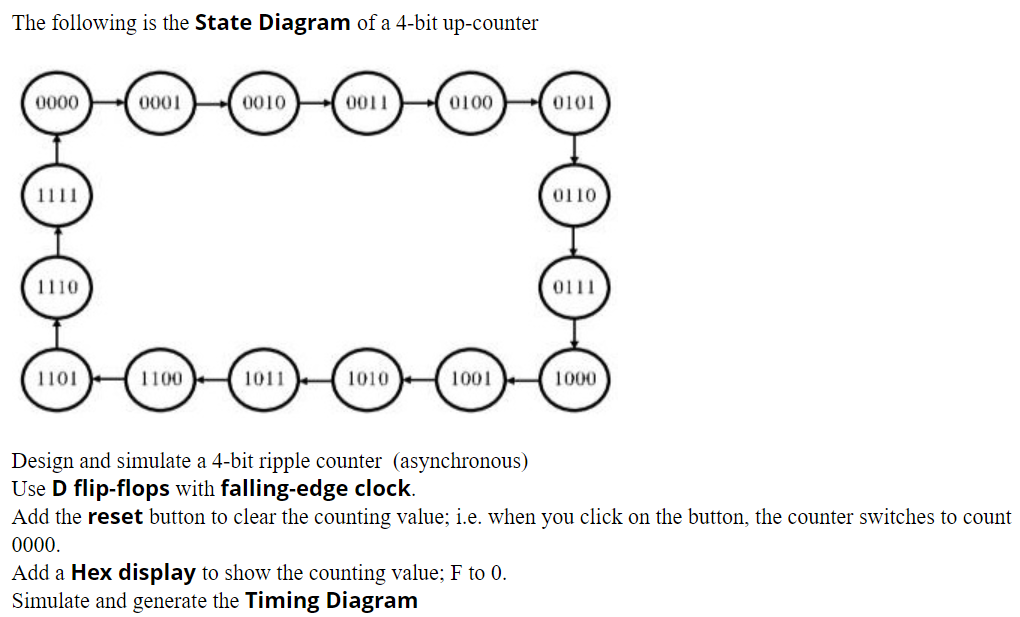
Table

Description automatically generated

**Task 3: Modelling Counters**

Model the following three counters.

1. **A 4-bit up counter**: Fork up the shift register work to make up a module called the up\_counter, that starts counting from 0 (after the asynchronous high asserted reset is given) and counts up in increments of 1 from 0, 1, 2, … 15, every rising edge of the clock. Its state diagram is given below. The reset value is 0.

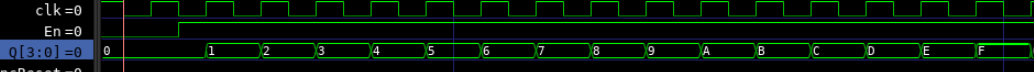


Hints: The interface of the counter should be as given below.

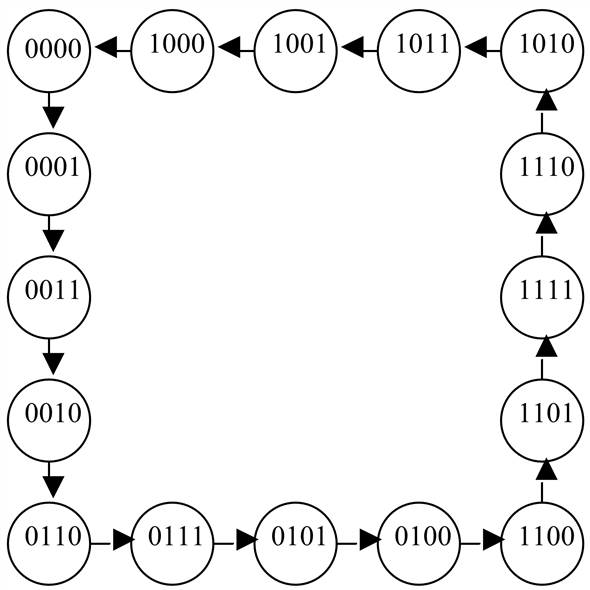
Text

Description automatically generated

Use the arithmetic operator in the body of the always block (Q <= Q+1;) with its reset value being 0. Let’s see if you can mimic the following waveform.



1. **A 4-bit down counter**: How to convert this up counter to a down counter? HINT: Use the arithmetic operator in the body of the always block (Q <= #1 Q+1) with a reset value of 4’b1111;. Can you also mimic a counter that counts up in increments of 2?
2. **4 bit Gray code**: How to convert this counter into a customized counter like a 4 bit Gray code counter. The state diagram of the 4-bit Gray code counter is as shown in the Figure below.



HINT: Use a case statement inside an always block to switch the register from one state to the next based on the diagram given.

Text

Description automatically generated with low confidence

Where the Q is assigned the Q\_in, in the always block for the D-FF (Q <= #1 Q\_in). Q\_in must be declared as a reg earlier. Drop the wave form below.

**Task DIY: A 4-bit shift right register with a parallel data load**

Diagram

Description automatically generated

We want to make a 4-bit shift right register with a parallel data load. There is going to be the following three inputs

* A single bit input called load\_Shiftbar that controls the operation of the 4-bit shift register.
* A 4 bit parallel input data (D) that is loaded into the shift register if load\_Shiftbar=1.
* A single bit input called serial input (s\_in) that is loaded in the MSB of the register when a shift load\_Shiftbar=0.

There is no reset or clear signal needed. We can model this shift register as follows.

A computer code with many letters

Description automatically generated with medium confidence

For the shifting operation, we use the concatenation operator ({}). The MSB is the serial input (s\_in) and the three bits are the shifted version of the Shift register (Q[3:1]). Can you tweak the testbench to generate the following testbench?

A screenshot of a computer

Description automatically generated with medium confidence