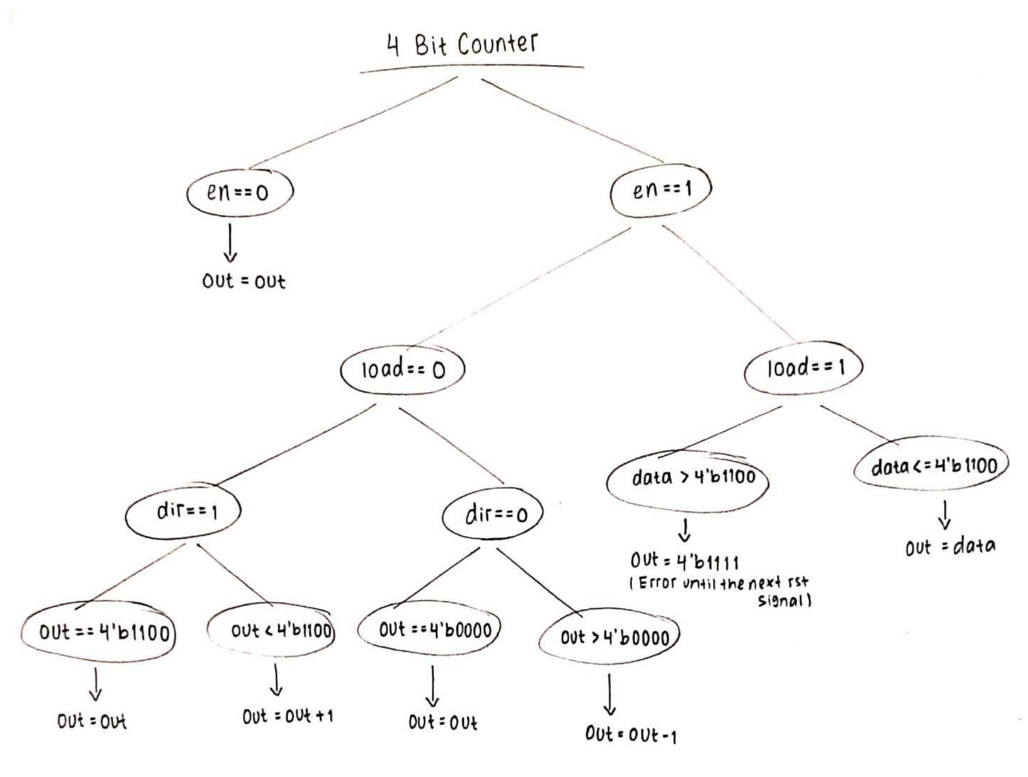
| EECS 2070 02 Digital Design Labs 2020<br>Lab 2 |
|---|
| **學號：108062281　　姓名：莊晴雯** |

1. 實作過程

- lab2_1 & lab2_1_t

  ***Source Code***

  In lab2_1 we are required to design a 4-bit counter between 0 and 12 with error detection which is triggered by negative clock edges, and we are provided with the inputs clk, rst, en, dir, load, and data, and we have the output out. Based on the conditions for the counter provided, I first mapped it as follows to make it easier to understand.



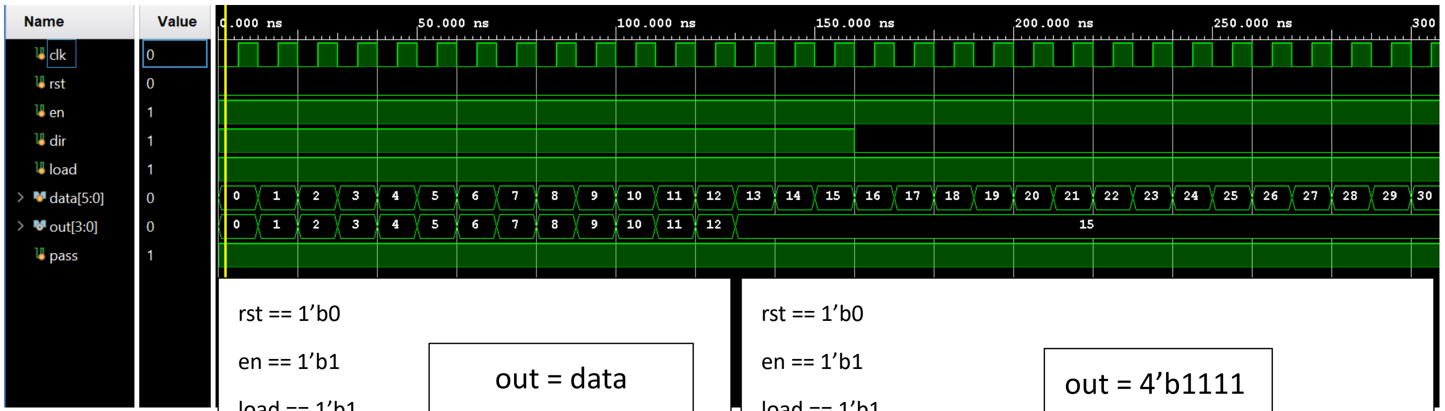
  After fully understanding how the counter actually works, I started typing in the code, based on the conditions stated. Firstly, declaring the reg and wire, then I started with designing the counter based on the conditions. For the counter, I designed it inside the always@(negedge clk) block because the counter itself is triggered by negative clock edges. I typed all the codes with behavioral modelling, that is, the if else statement. The outmost if statement is for the en and rst as its precedence is the biggest among all, followed by load, then dir when load is equal to 1’b0 and data when load is equal to 1’b1. The counter will count up, down, or load the input value when en == 1’b1, so I typed all the load, dir, and data conditions inside the else if(en == 1'b1 && rst !== 1'b1) block. If load is not enabled, that is, when load == 1’b0, dir will have two possibilities, which is dir == 1’b0 and dir == 1’b1. If dir == 1’b1, we need to check the previous output value first. If the previous output value is equal to 4’b1100, the output value will remain as 4’b1100. If the previous output value is smaller than 4’b1100 and greater than or equal to 4’b0000, out will results in the previous output value plus 4’b0001, because I realized that if the previous output is smaller than 4’b0000, the counter should not count up. Then when dir == 1’b0, we also need to check the previous output value. If the previous output value is equal to 4’b0000, the output value will remain as 4’b0000. If the previous output value is greater than 4’b0000 and smaller than or equal to 4’b1100, out will results in the previous output value minus 4’b0001 because if the output value is greater than 4’b1100, it means error (note that when out == 4’b1111 it results in an error, this is the error detection method). Then it remains for when load == 1’b1, we could just directly load the data as the output if it is smaller than or equal to 6’b001100. Otherwise, we will set 4’b1111 as the output if the input data is greater than 6’b001100, which indicates an error until the next reset signal occurred. After finishing with all the conditions, I then typed in the reset condition inside the always@(posedge rst) block because as stated in one of the counter conditions, rst is the positive-edge-triggered reset, and it will reset the counter value, which is out, to 4’b0000.

***<u>Testbench (lab2_1_t)</u>***

After finishing the source code, I started typing in the testbench. I used the template provided and the testbench from previous lab as my reference. In this testbench, I had the idea to make a new array op to store the values of en, load, and dir at the 3 most significant bits, and let the rest 4 bits indicate as out, for it to test every combination possibility of out with en, load, and dir, but I did not store the value of out inside op. So first inside the warmUp() task, I first initialize rst as 1’b0, op as 7’b1111111, en = op[6], load = op[5], dir = op[4]. op here will decrease its value by 1 inside the @(negedge clk) block. I also initialize data = 6’b000000, and a new variable check_out, which I first initialized as 4’b0000, to check the correctness of the output and indicate errors. Then I have pass to indicate errors, which is first initialized as 1’b1.

I set the rst to negate as big as every 720 ns to see all the possible combinations of all possible conditions. I set the number 720 by observing the waveform, also by the trial and error method. Inside the initial block, I first called the warmUp() task to initialize the all the starting values, then $display and $monitor to see indicate that the simulation is starting and to show the values, which can be uncommented to print the outputs. Inside the repeat block, 2**10 because there are 10 bits in total from data and out, there are the negedge and posedge clk. The @(negedge clk) block is to get the check_out value that will be compared later with out to indicate errors, while the @(posedge clk) block is to call the task test(). Inside the @(negedge clk) block, I increase the value of data with 1, decrease the value of op with 1, and set en as op[6], load as op[5], and dir as op[4]. Then I continue with the conditions for the counter to get the value of check_out which I will use for comparing it with the output out to check the correctness of my souce code. The testbench’s code to get the value of check_out is really similar to the one in the source code, but this time I replace the output with check_out. Inside the @(posedge clk) block which calls the task test(), it will check each conditions and correct outputs. If the output does not satisfy based on the conditions, then it will execute the printerror() task which will display the error message along with the values. After finishing with executing the repeat block, I displayed a message indicating that the simulation is terminating and display PASS if pass still remains as 1’b1, and ERROR is pass changes to 1’b0. Then, the process is done.

Here are all the possible outputs from each conditions looking from the waveform:



rst == 1’b0

en == 1’b1

load == 1’b1

data <= 4’b1100

out = data

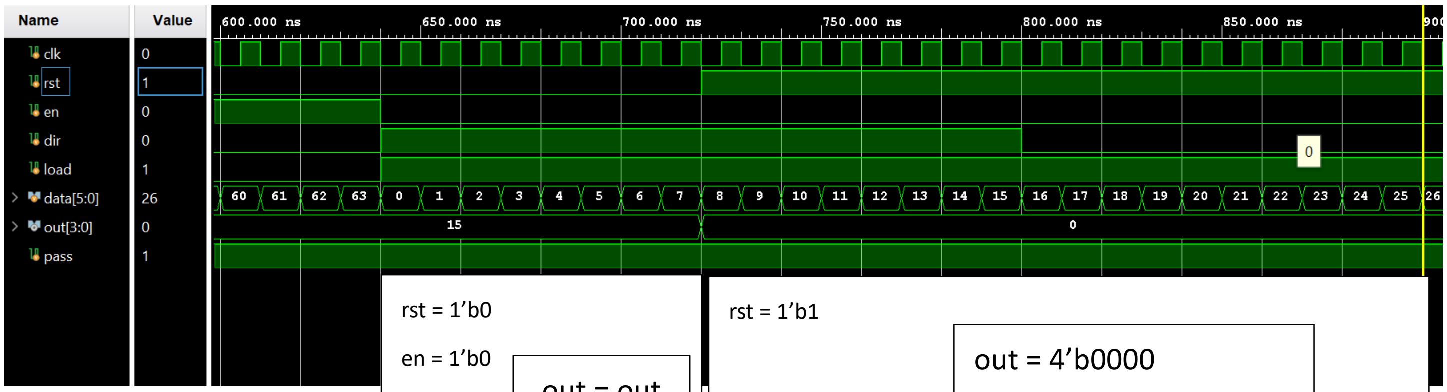
rst == 1’b0

en == 1’b1

load == 1’b1

data > 4’b1100

out = 4’b1111



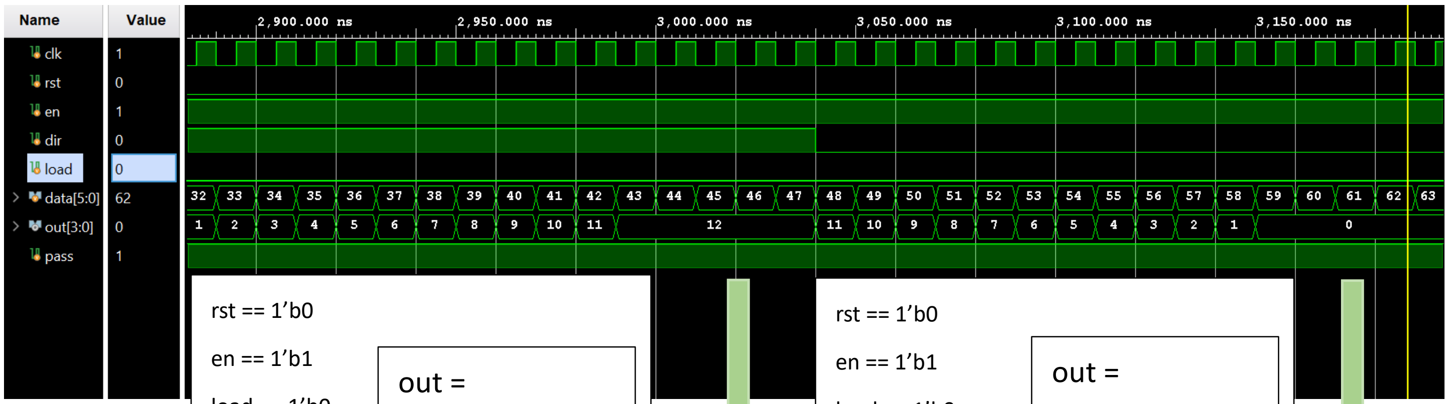
rst = 1’b0

en = 1’b0

out = out

rst = 1’b1

out = 4’b0000



rst == 1’b0

en == 1’b1

load == 1’b0

dir = 1’b1

out < 4’b1100

out =

out + 4’b0001

rst == 1’b0

en == 1’b1

load == 1’b0

dir = 1’b0

out > 4’b0000

out =

out - 4’b0001

rst == 1’b0

en == 1’b1

load == 1’b0

dir = 1’b1

out == 4’b1100

out = out

rst == 1’b0

en == 1’b1

load == 1’b0

dir = 1’b0

out == 4’b0000

out = out

- lab2_2 & lab2_2_t

  ***Source Code***

  In lab 2_2 we need to design an 8-bit Fibonacci counter which is triggered by positive clock edges, which counts up from 1 to 233 and then counts down from 233 to 1. We are provided with the inputs clk and rst, and the 8-bit output fn, which is the Fibonacci sequence. The first thing we always need to do is to declare the wire and reg. Here, I added f1 to help fn with the counting and flag to indicate the fn when to count up and when to count down. Because the counter and rst is triggered by positive edges, I designed both the counter and rst inside the always@(posedge clk) block using behavioral modelling. If rst is not equal to 1, then fn and f1 will start counting. The counter starts counting from 0 because of the rst signal in the testbench. It is first set as 1’b0 since both fn and f1 does not have any output yet. Then after 10 ns I negate is so it becomes 1’b1, but that is the reset condition that results in fn becoming 8’d1 and f1 becomes 8’d0. This situation should not take a long time so I set it to hold for 10 ns too for it to negate back to 1’b0 again so the Fibonacci Counter can start counting up. The flag stays at 0 because fn == 8'd1 && f1 == 8'd0, then it executes the formula for counting:

  ```
  fn <= fn + f1;
  f1 <= fn;
  ```

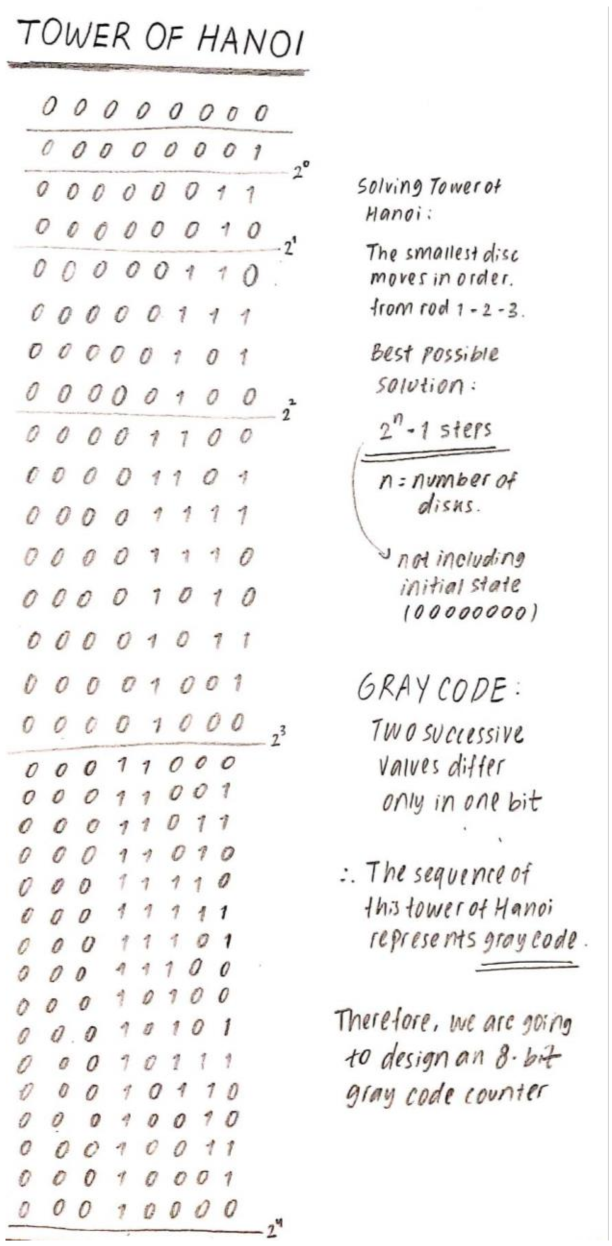
  <= means that they will assign the value obtained from the expression on the right hand side to the variable on the left hand side simultaneously and not sequentially, which is also known as non-blocking assignment, so the value of fn in the assignment statement f1 <= fn; will not change due to the previous assignment statement (fn <= fn + f1;) After fn reaches 233 (8’d233) and f1 reaches 144 (8’d144), flag will change its value to 1’b1 and the assignment statement inside if(flag == 1'b1) will get executed, which is to count down to 1. There is another condition for f1 to be greater than 8’b00000000 so that it meets the condition of the Fibonacci Counter as described in this lab2_2 for it to count like: …5321111235… After counting back down, if fn reaches 8'b00000001 and f1 reaches 8'b00000000 again, flag will be set to 0 and both fn and f1 will be set to its initial value again, and start counting all over again from the very beginning, without having to start counting to 0 to satisfy the condition of the Fibonacci counter provided. Then for the rst condition, if rst is equal to 1’b1, then fn will be reset to its initial value which is 8’d1 and f1 will be reset to 8’d0 too.

***Testbench (lab2_2_t)***

In the testbench, I first initialize the starting value of clk as 1’b1, flag as 1’b0, f1_test as 8’d0, fn_test as 8’d1, pass as 1’b1 and rst ad 1’b0. The counter starts counting from 0 because of the rst signal in the testbench. It is first set as 1’b0 since both fn and f1 does not have any output yet. Then after 10 ns I negate is so it becomes 1’b1, but that is the reset condition that results in fn becoming 8’d1 and f1 becomes 8’d0. This situation should not take a long time so I set it to hold for 10 ns too for it to negate back to 1’b0 again so the Fibonacci Counter can start counting up. Then as usual I used display and monitor to indicate that the simulation is starting and to display the value of fn which can be uncommented. Next, I declared a 2**8 repeat block because there are 8 bits of Fibonacci numbers, and inside the repeat block I called the test task inside the @(negedge clk) block. Test is used to check the output fn with fn_test. If they have the same value, then the output from the source code, which is fn, yields the correct value. If not, then it will call the task printerror which will print the errors occurred. Moreover, there is another if statement which I used to test the boundaries of the Fibonacci sequence, since it should only count from 1 to 233, not more or not less. Then inside the @(posedge clk) block, it is really similar with the code inside the source code. These code inside the @(posedge clk) block is used to assign the correct value to fn_test for it to check with the value of fn and determine its correctness. Last thing after finishing the repeat block is to print a message indicating that the simulation is terminating and print either a PASS message or ERROR message, depending on the value of pass.

- lab2_3 & lab2_3_t

  ***Main Idea***



TOWER OF HANOI

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
                    2^0
0 0 0 0 0 0 1 1
0 0 0 0 0 0 1 0
                    2^1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 0 1 0 1
0 0 0 0 0 1 0 0
                    2^2
0 0 0 0 1 1 0 0
0 0 0 0 1 1 0 1
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 0
0 0 0 0 1 0 1 0
0 0 0 0 1 0 1 1
0 0 0 0 1 0 0 1
0 0 0 0 1 0 0 0
                    2^3
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 1
0 0 0 1 1 0 1 1
0 0 0 1 1 0 1 0
0 0 0 1 1 1 1 0
0 0 0 1 1 1 1 1
0 0 0 1 1 1 0 1
0 0 0 1 1 1 0 0
0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 1
0 0 0 1 0 1 1 1
0 0 0 1 0 1 1 0
0 0 0 1 0 0 1 0
0 0 0 1 0 0 1 1
0 0 0 1 0 0 0 1
0 0 0 1 0 0 0 0
                    2^4
```

Solving Tower of Hanoi:

The smallest disc moves in order. from rod 1 - 2 - 3.

Best possible solution:

$2^n - 1$ steps

n = number of disks.

not including initial state (00000000)

GRAY CODE:

Two successive values differ only in one bit

∴ The sequence of this tower of Hanoi represents gray code.

Therefore, we are going to design an 8-bit gray code counter

When I saw this bonus problem for the first time, I looked up to YouTube on the best way to solve the Tower of Hanoi puzzle. I got the idea that the smallest disk should move in the order (from rod 1, to rod 2, to rod 3, back again to rod 1), ans that the best possible solution should be solved in (2^n)-1 steps, where n is the number of disks, and it does not include the intiial state when the disks have not been moved just yet, which is (0,0,0,0,0,0,0,0). I tried playing the puzzle myself on the internet and list the moves in order, only to realize that I was actually writing a gray code. The sequence of the best possible moves to solve the Tower of Hanoi puzzle is actually a gray code that increases by one everytime a disk moves. Therefore, we are actually designing a gray code up counter.

***Source Code***

I used the method of converting binary numbers to gray code. The method is to XOR the nth bit with the (n+1)th bit from the least significant bit, up to but not including the most significant bit, because the most significant bit of the gray code holds the same value as the most significant bit of the binary number, and then store the value from XOR inside the nth bit of the gray code. Which is why, I used this computation method in my design.

```
out[7] <= bin_ctr[7];
out[6] <= bin_ctr[7] ^ bin_ctr[6];
out[5] <= bin_ctr[6] ^ bin_ctr[5];
out[4] <= bin_ctr[5] ^ bin_ctr[4];
out[3] <= bin_ctr[4] ^ bin_ctr[3];
out[2] <= bin_ctr[3] ^ bin_ctr[2];
out[1] <= bin_ctr[2] ^ bin_ctr[1];
out[0] <= bin_ctr[1] ^ bin_ctr[0];
```

I first declared out and bin_ctr as reg of 8 bits and initialized their starting values both as 8’b00000000. Then because the counter is positive edge triggered, so I used the always @(posedge clk), and if rst is not equal to 1, the counter will start counting using the method above, and bin_ctr also needs to increase its value by 8’b0000001 every time since our main idea is to convert from binary numbers to gray code, and bin_ctr here is the binary digit counter. As for the reset condition, if reset is equal to 1’b1, then out and bin_ctr will hold its initial value which is 8’b00000000.

***Testbench (lab2_3_t)***

I declared pass and bin_ctr in this testbench, where pass will be used to indicate errors and bin_ctr will represent the binary numbers which will increase by one everytime. I set rst to negate every 2560 ns to see all the sequence from 00000000 to 10000000 because for every bit it increases, the wave will take 10ns, and we have 256 waves to display, including the initial move, which is 00000000, so it will be 2560, and moreover, it will also display the wave a little further for the case when rst == 1’b1. Inside the initial block I first called the warmUp() task to initialize all the starting values. Then as usual I used display and monitor that can be uncommented to indicate that the simulation is starting and to display the values of bin_ctr and out, which is the binary sequence and gray code sequence. Since there are 8 bits, I used 2**8 for the repeat block, which consists of @(posedge clk) block to increase the value of bin_ctr and @(negedge clk) block for calling the task test that will check the outputs with the correct answer. In the test task, it will call the printerror() task which will generate an error message if an error occurs and set the pass' value as 1'b0, which is when each output bits does not comply with the desired output (XOR the nth bit with the (n+1)th bit from the least significant bit, up to but not including the most significant bit, and put the value inside the nth bit of the gray code), and also if rst is equal to 1'b1 but out is not equal to 8'b00000000. I wrote the code as follows:

```
if(out[7] !== bin_ctr[7] ||
    out[6] !== bin_ctr[7] ^ bin_ctr[6] ||
    out[5] !== bin_ctr[6] ^ bin_ctr[5] ||
    out[4] !== bin_ctr[5] ^ bin_ctr[4] ||
    out[3] !== bin_ctr[4] ^ bin_ctr[3] ||
    out[2] !== bin_ctr[3] ^ bin_ctr[2] ||
    out[1] !== bin_ctr[2] ^ bin_ctr[1] ||
    out[0] !== bin_ctr[1] ^ bin_ctr[0])
      printerror();

else if(rst == 1'b1 && out !== 8'b00000000)
      printerror();
```

Last thing after finishing the repeat block is to print a message indicating that the simulation is terminating and print either a PASS message or ERROR message, depending on the value of pass.

2. 學到的東西與遇到的困難

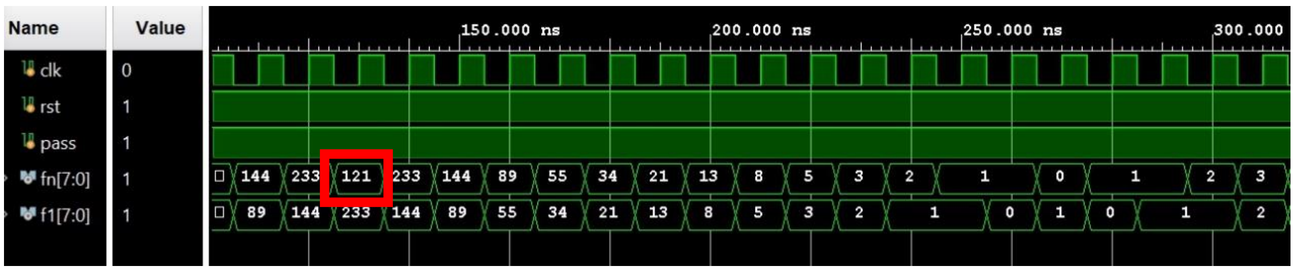
- lab2_1
  Having the thought of already completing both source code and test bench, I tried running the program. As expected, I run into some errors that I struggled for quite a long time until I realized my mistakes. I tried fixing the errors and got the PASS message at first, but then after looking more deeply into my code I realized I lacked a few things in both my source code and my testbench. I started to realize that I need more conditions to make the code runs smoothly. One of it is the reset condition, which was not there at first, which is why I added the condition if rst !== 1'b1 inside the always@(negedge clk). To fulfill the condition of a correct counter, the reset should not be equal to 1 for the counter to work. If the reset is equal to 1,

then the output will hold its initial value until the next reset signal occurred. I also added the condition out >= 4'b000 to when en == 1'b1, load == 1'b0, dir == 1'b1, and out < 4'b1100 (for the counter to add 4'b0001 for the next output value) because I realized that if out is smaller than 4'b0000, the counter should not count up. The same thing goes for when en == 1'b1, load == 1'b0, dir == 1'b0, and out > 4'b0000 (for the counter to subtract 4'b0001 for the next output value), I added the condition out <= 4'b1100 because if the output value is greater than 4'b1100, it means error (note that when out == 4'b1111 it results in an error). Then it remains for when load == 1'b1, we could just directly load the data as the output if it is smaller than or equal to 4'b1100. Otherwise, we will set 4'b1111 as the output if the input data is greater than 4'b1100, which indicates an error until the next reset signal occurred.

I also struggled in making the testbench at first. I have not had the thought of using check_out as an indicator to check the output, and so I experimented with many things until I ended up with the idea of using check_out. Also, I was confused on how to set rst, en, dir, and load for it to test with all the possible cases. I tried using always # but I couldn't find the right number of ns and I keep getting errors. Which is why, I decided to use op with 8 bits, and use the first three most significant bits as en, dir, and load, and let the rest four bits indicate as out.

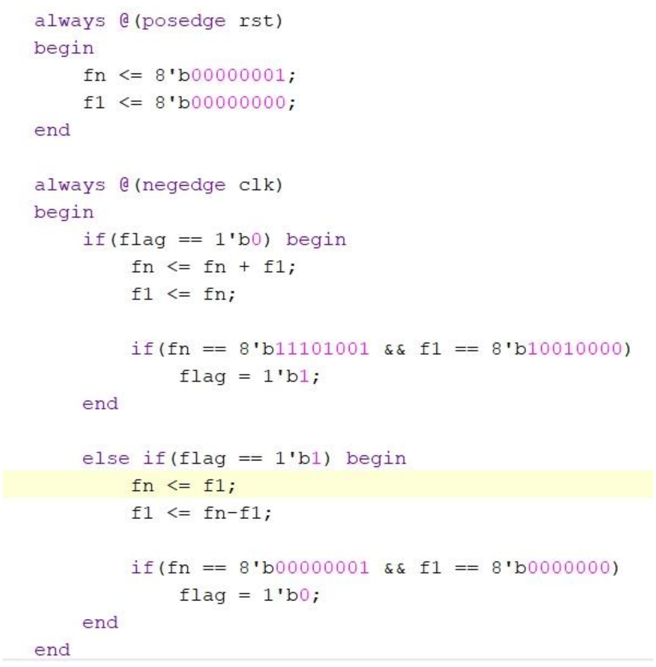
- lab2_2

  I tried using the for loop at first in counting the Fibonacci sequence, but then I encountered a problem when running the simulation, that is, the simulation won't stop running until a few minutes. I tried changing the simulation runtime and the number inside the repeat block thinking that it was the problem but it actually is not. I ended up consulting with the TA and the teacher about this problem and they pointed out the for loop and said that this might be the case why my code runs nonstop for a few minutes. Therefore, I changed my coding style and used the method that I used now, which is to add fn and f1 and then store it inside fn, also store the old fn



value to f1 simultaneously if rst is not equal to 1. I also did a little thinking before I ended up using the flag thing. But when using flag, I encountered a problem too, a number 121 occurred between the transition of counter up 233 to counter down, as shown on the picture below.

My previous lab2_2_t code (wrong)



```
always @(posedge rst)
begin
    fn <= 8'b00000001;
    f1 <= 8'b00000000;
end

always @(negedge clk)
begin
    if(flag == 1'b0) begin
        fn <= fn + f1;
        f1 <= fn;

        if(fn == 8'b11101001 && f1 == 8'b10010000)
            flag = 1'b1;
    end

    else if(flag == 1'b1) begin
        fn <= f1;
        f1 <= fn-f1;

        if(fn == 8'b00000001 && f1 == 8'b0000000)
            flag = 1'b0;
    end
end
```

I consulted with a few people regarding this problem and someone said that looking at my previous code (picture beside), what's probably happening is, once fn is 233 and f1 is 144, the entire if (flag == 1'b0) block executes again, all at once- fn gets 144 + 233 (which is 121 when truncated to 8 bits), f1 gets 233 (the value of fn), and the if (fn == 8'b11101001 && f1 == 8'b10010000) sees these values and updates flag (at which point it's too late – I have overflowed). I tried changing my code by re-assigning fn and f1 with their initial value in the else if(fn == 8'd1 && f1 == 8'd0) block and eventually resulted in the correct answer. As something that might confuse the TA when doing my demo, is that if I reassign the value of fn and f1 with their initial value again and I might get an infinite loop, this case won't happen because I have set flag for both boundaries, counting up and counting down. I have also learnt that it is recommended to not mix the usage of both blocking and non-blocking assignment. If you use non-blocking assignment once in your code, everything should better be also using non-blocking assignment. Unfortunately, I learnt this from my friend after the submission time for lab02 has passed, so I didn't get the chance to fix my code. In my code here, from this statement:

```
if(fn == 8'd233 && f1 == 8'd144)
    flag = 1'b1;
```

it can be changed to non-blocking assignment without resulting in an error by changing the conditions inside the if statement:

```
if(fn == 8'd144 && f1 == 8'd89)
    flag <= 1'b1;
```

In the upcoming labs, I will remind myself to be consistent and use one type of assignment and not mixing them up. Then to make my code more readable and easier regarding the values, I changed 8'b11101001 to 8’d233.

- lab2_3
  One of my bad habits is to keep forgetting on initializing a starting value. At first I got no values for the out and bin_ctr just because I forgot to give them a starting value and I keep trying to find the errors in my code and after a while I just realized that I need to set a starting value for both bin_ctr and out, or either resetting it first.

3. 想對老師或助教說的話

Everything was still okay overall, I enjoyed doing the lab though sometimes it might be a little bit stressful when being encountered with an error which took quite a long time to find a way or solution to solve it.