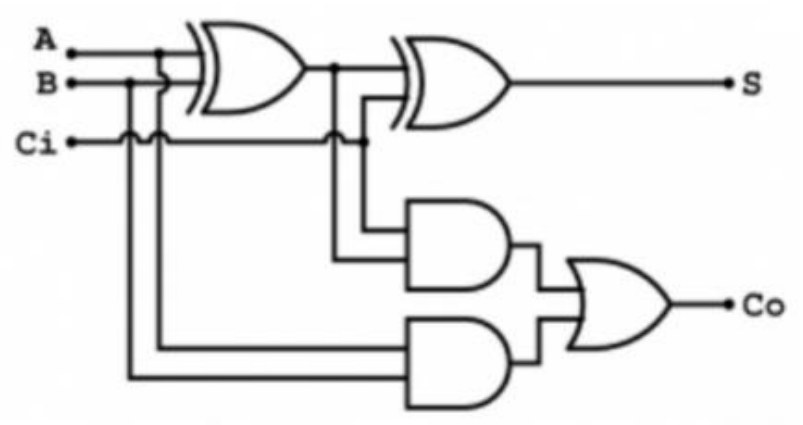
| EECS 2070 02 Digital Design Labs 2020<br>Lab 1 |
| --- |
| **學號：108062281　姓名：莊晴雯** |

1. 實作過程

- lab1_1 & lab1_1_t

  In lab1_1 we need to design a Verilog module that models a 1-bit full adder with some basic components stated (AND, OR, NOT, XOR), and the inputs and outputs provided (inputs: *a, b, c, sub*; outputs: *d, e*) using Gate Level Implementation. The first thing we need to do after typing in the template given by the TAs is to declare *a, b, c, d, e* and *sub* as *wire.* Based on the definitions of *wire* and *reg*, wires are driven from the outputs of instantiated modules and by continuous assignments, and are used for connecting different elements. As for *reg*, they are to be assigned to from behavioral statements, and they represent data storage elements and will retain their value until the next value is assigned to them, but not through assign statement. I designed this module based on the block diagram provided in the PDF (attached below).




  This time, the behavior of *b* in the full-adder depends on *sub*. Because we need to use a Gate Level Implementation, we need to think of a way that models the behavior of *b* which depends on *sub*. When *sub* is equal to 1 (*sub == 1'b1*), the value of *b* must be negated. According to the behavior of *b* that is dependent of *sub (if sub == 1'b1)*, *b* must be negated, I tried drawing the truth table for *b* and *sub*, and ended up with this result:

| *b* | *sub* | *b ^ sub* |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

As we can observe, when *sub* is equal to 1, *b* needs to be negated, which is shown on the last two rows, and when *sub* is equal to 0, *b* retains its value, as shown on the first two rows. Hence, if we look carefully at the truth table, it represents the behavior or XOR. Therefore, we can conclude it as *b^sub*, which we can use in implementing lab1_1 using Gate Level Modelling.

This time, I also need to declare more variables namely *xor_a_b*, *xor_sub_b*, *and_ab_c*, and *and_a_b* to store the outputs from the gates. I implemented this design using the block diagram provided in the PDF. The *a* in my design is the input *A* in the above picture, and so is the *b* in my design refers to the input *B* from the picture above. The *c* in my design refers to *Ci* in the picture above. The *d* is the *S* from the picture above, which is the sum, and my *e* is the *Co* from the picture above, which is the carry out generated. *xor_sub_b* was resulted from *sub^b* because of the above explanation (truth table), *xor_a_b* is the output resulted from *a^xor_sub_b*, which is the output from the first XOR gate from the above picture, and for the *S*, which is *d* in my design, is resulted from *xor_a_b^c*. To get the *Co*, we first must let *xor_a_b* and *c* go through an AND gate, which results in *and_ab_c* in my design, then I let *a* and *xor_sub_b* go through an AND gate in which will result in *and_a_b*. Lastly, to output the *Co*, which is *e* in my design, we need to do *and_ab_c* | *and_a_b*.

As for lab1_1_t (the testbench):

1. TO DO 1: instantiating lab1_1 with correct interconnection is to fill in the parentheses with the correct variables in module lab1_1.
2. TO DO 2: calling the task function test for it to execute at positive clock edges. Just type in *test();*.
3. TO DO 3: increasing the counter {*a, b, c, sub*} each by 1 at negative clock edges. We need to type in *{a, b, c, sub} = {a, b, c, sub} + 4'b1;*.
4. TO DO 4: executing the task function *printerror* if the behavior of the module lab1_1 is incorrect. One condition is already provided as an

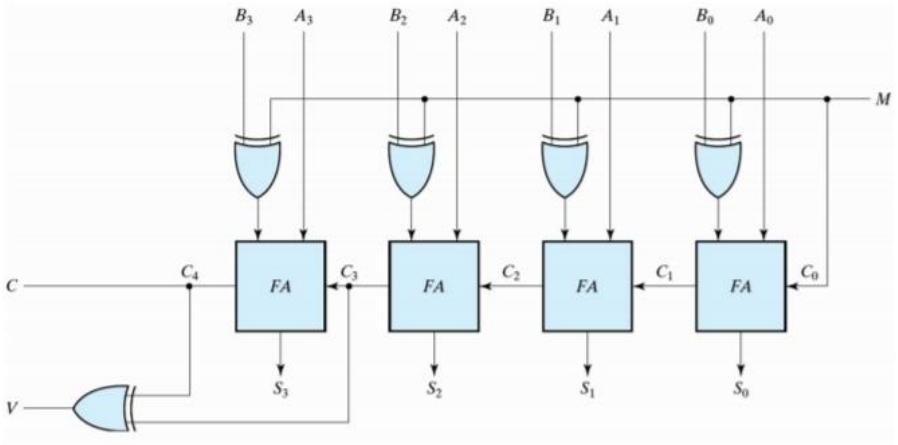
example, the other condition is *(sub == 0 && {e,d} !== a + b + c)* because if *sub* is equal to 0 but *{e, d}* is not equal to *a + b + c*, then it is wrong. Hence, we need to call *printerror* function.

5. TO DO 5: set the value of the signal "*pass*" to 0 and display the value of *a, b, c, d, e* and *sub* when error happens. We need to type in *pass = 1'b0;* and complete the variables as follows: *$display($time," Error:  a = %b, b = %b, c = %b, d = %b, e = %b, sub = %b", a, b, c, d, e, sub);*.

- lab1_2 & lab1_2_t

  In lab1_2 we need to design a Verilog module that models a 4-bit 2's complement adders by using four 1-bit full-adder designed in the previous module (lab1_1). Therefore, we need to instantiate module lab1_1 into module lab1_2. This time also, the behavior of *b* is determined by the value of *sub*. If *sub* is equal to 1, the output will be *a-b*. Otherwise, the output will be *a+b*. The behavior of *b* which is determined by *sub* is already written in module lab1_1. Therefore, we won't need to worry about it and we can directly instantiate the module lab1_1 to module lab1_2 like stated below. But firstly, do not forget to declare *a, b, sub, d* as *wire*, also add more variables (*c0, c1, c2, c3*) which are declared as *wire* and is used for the carries that are generated from each one full-adder and serves as input for another full-adder. After this, we only need to instantiate module lab1_1 to module lab1_2 like this:

  *lab1_1 full_adder(.a(a[0]), .b(b[0]), .c(sub), .sub(sub), .d(d[0]), .e(c0));*
  *lab1_1 full_adder_1(.a(a[1]), .b(b[1]), .c(c1), .sub(sub), .d(d[1]), .e(c1));*
  *lab1_1 full_adder_2(.a(a[2]), .b(b[2]), .c(c2), .sub(sub), .d(d[2]), .e(c2));*
  *lab1_1 full_adder_3(.a(a[3]), .b(b[3]), .c(c3), .sub(sub), .d(d[3]), .e(c3));*



I used the gate-level implementation above as reference. Here, *a[0], a[1], a[2], a[3]* each serves as *A0, A1, A2, A3* and *b[0], b[1], b[2], b[3]* each serves as *B0, B1, B2, B3*. *sub* serves as *M*. As for the outputs, *d[0], d[1], d[2], d[3]* serves as *S0, S1, S2, S3*, and for the carry, each *c0, c1, c2, c3* serves as *C0, C1, C2, C3*. All the inputs are passed to lab1_1 module, and we need to give each of it a different name, following the correct syntax of module instantiation. Here, I used *full_adder, full_adder_1, full_adder_2, and full_adder_3* as the names. In this lab01, I also used the named port mapping method in the module instantiation, where the order ports of the submodule won't need to be the same order as the module, which is more recommended. After this, the instantiated module lab1_1 will compute the inputs and generate the output as desired.

As for lab1_2_t (the testbench):

1. TO DO 1: instantiating lab1_2 with correct interconnection is to fill in the parentheses with the correct variables in module lab1_2.
2. TO DO 2: Set the correct iteration times. I set it to 2**9 because there are 9 bits. Therefore, there are 2^9 possible combinations to be tested.
3. TO DO 3: calling the task function test for it to execute at positive clock edges. Just type in *test();*.
4. TO DO 4: increasing the counter {*a, b, sub*} each by 1 at negative clock edges. We need to type in *{a, b, sub} = {a, b, sub} + 1'b1;*.
5. TO DO 5: executing the task function *printerror* if the behavior of the module lab1_1 is incorrect. The conditions are when *sub* is equal to 1 but *d* is not equal to *a-b*, and when *sub* is equal to 0 but *d* is not equal to *a+b*. Hence, the code is as follows:

   ```
   if ((sub == 1 && d !== a - b) || (sub == 0 && d !== a + b)) begin
           printerror;
   end
   ```

6. TO DO 6: set the value of the signal "*pass*" to 0 and display the value of *a, b, d,* and *sub* when error happens. We need to type in *pass = 1'b0;* and *$display($time," Error:  a = %b, b = %b, d = %b, sub = %b", a, b, d, sub);* to display the value of *a, b, d* and *sub* when error happens.

- lab1_3 & lab1_3_3

  In lab1_3 we are required to design a 4-bit ALU which performs the following operations, depending on the value of *aluctr*:

  ```
  if(aluctr == 2'b00)
      d = a + b;
  else if(aluctr == 2'b01)
      d = a + b;
  else if(aluctr == 2'b10)
      d = a & b;
  else
      d = a ^ b;
  ```

  Since we need to instantiate module lab1_2 into module lab1_3, and the module lab1_2, which implements a 4-bit 2's-complement adders, can only perform addition and subtraction, meanwhile our ALU needs to also implement the operators & and ^, so I designed this using behavioral modeling. This design consists of instantiation, *always* block, and the *if else* statements because we need to perform the & and ^ operation, which I wrote inside the always block, continued with the if else statement. The operation '+' and ' –' can be performed using instantiation, so I implemented it by:

  ```
  lab1_2 bit_adder(.a(a), .b(b), .sub(aluctr[0]), .d(d1));
  ```

  The line above takes in the inputs and generates our desired output which is *d1*. I will later pass the output generated from the instantiation above, which is *d1*, to the output of this module, which is *d*, inside the *if else* statement from the *always* block. I used *d1* for temporary storage because we cannot use *reg* inside the instantiation method. I have tried experimenting with it and got the error message: *[VRFC 10-3236] concurrent assignment to a non-net 'd1' is not permitted.* Therefore, I declared *d1* as *wire*.

  Then for the *always* block, I used *if else* statements inside it which serves as a multiplexer that covers all the cases from *aluctr == 2'd00* until *aluctr == 2'd11*. For *aluctr == 2'd00* and *aluctr == 2'd01*, we can directly pass the value *d1* to *d*, since the output we desired was already generated by the instantiation method and the value was already assigned to *d1*. As for *aluctr == 2'd10* and *aluctr == 2'd11*, I directly implement it using *d = a & b* and *d = a ^ b* because we cannot perform & and ^ using the module lab1_2.

As for lab1_3_t (the testbench):
The testbench was already completely provided.

2. 學到的東西與遇到的困難

I need to familiarize myself at programming in Verilog language again since I have not done programming in Verilog for quite a time. After reviewing the materials provided by the Professor and TAs, also doing some research at the internet, reading information about Verilog and its syntaxes, also discussing and exchanging ideas with my friend, I eventually got familiarized with the language again in no time. In this lab, I also learned a new thing which is to reuse the previous modules (lab1_3 reuses lab1_2 module, and lab1_2 reuses lab1_1 module). This kind of instantiation method made me quite confused at the beginning, but after a few tries and learning from previous mistakes, also by looking at the examples from the internet, I finally understood how it really works. I have also learned that if we use the instantiation method, we cannot use *reg* declaration even though it is the output in the module instantiation, as I will get the error message*: [VRFC 10-3236] concurrent assignment to a non-net 'd1' is not permitted.* So instead, I used the declaration *wire* to solve this problem. I have also learned about the two methods of port mapping that we can use, which are positional port mapping and named port mapping. However, named port mapping is more recommended to avoid mistakes and errors, which is why I used named port mapping in this lab01. I also learned one new thing which is $monitor, that can show all the information of our inputs and outputs during the simulation.

3. 想對老師或助教說的話

I have tried learning about Basys3 FPGA and Vivado the first time I received the teaching materials from iLMS, but to be honest I got quite confused while watching the video and reading the manual on my own. As for Vivado, I have gotten quite familiarized with it because of this lab01, but I am still struggling to understand more about Basys3 FPGA. In the future, I hope to be able to familiarize myself with the Basys3 FPGA done in the upcoming labs and maybe more supplementary materials about Basys3 FPGA.