U77561194 Zehao Hui
U28088400 Yunlu Deng

LAB 2

EC605
September 29, 2021

Task1: Gate Level Full Adder

Here is the schematic and simulation result of task1,

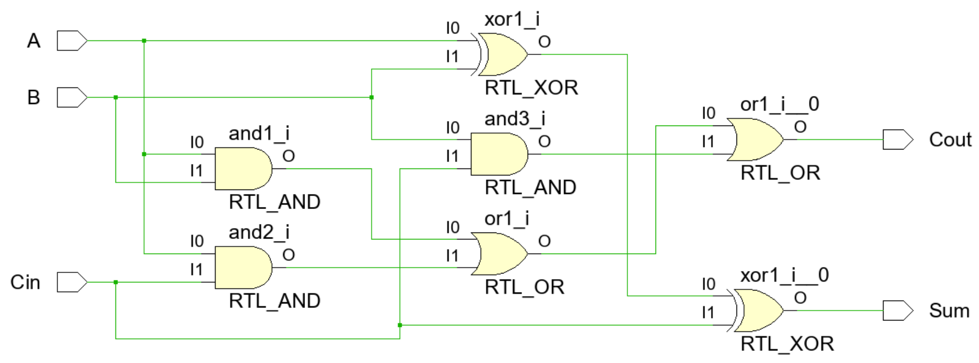


Figure 1: Schematic for task1

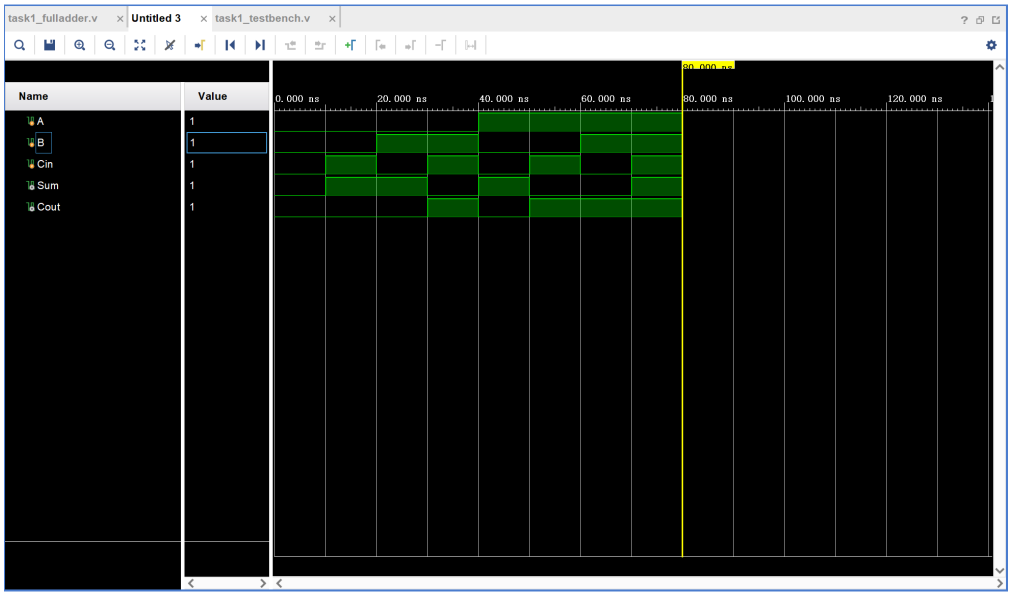


Figure 2: Simulation for task1

Then I will describe how full adder works. First we can get the truth table of it.

Cin	A	B	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

By drawing K-maps, we can get the result of Cout and Sum,

$$Cout = AB + CinB + CinA$$
$$Sum = A \oplus B \oplus C$$

So we can get this schematic and the simulation result is true.

Task2: Ripple Carry Adder

Here is the schematic and simulation result of task3,

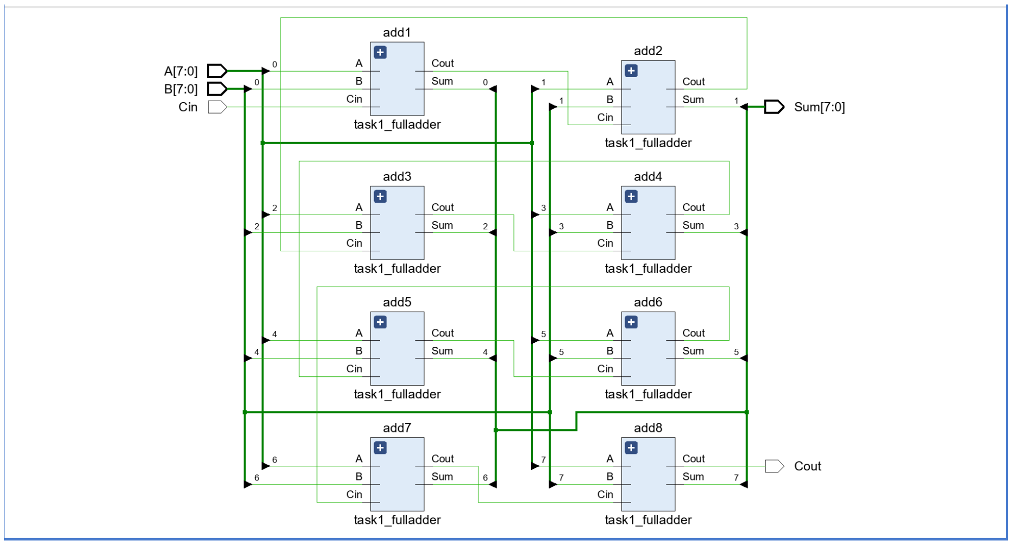
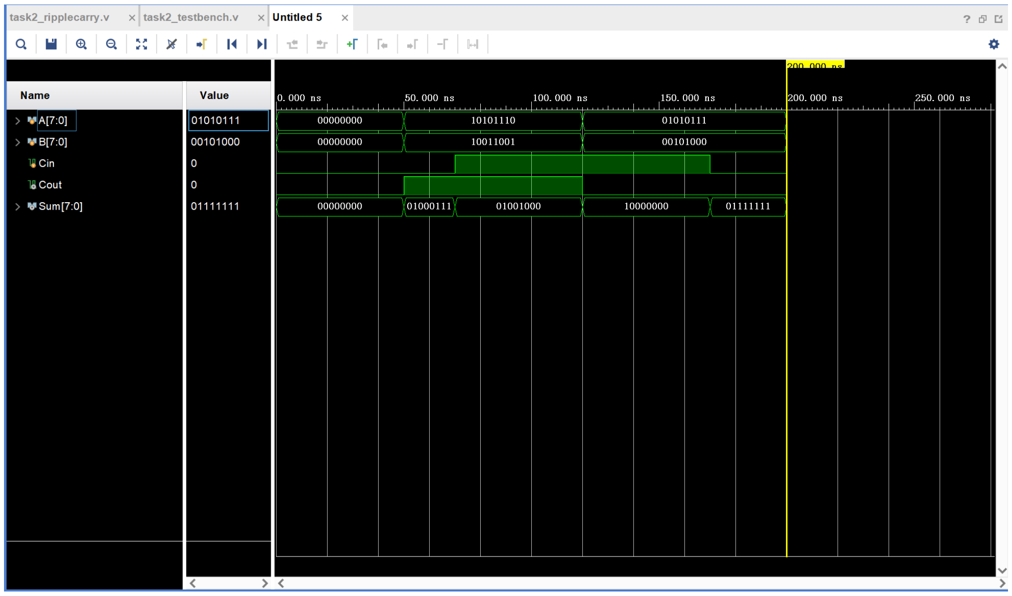


Figure 3: Schematic for task2



```
A[7:0]     01010111   00000000 | 10101110 | 01010111
B[7:0]     00101000   00000000 | 10011001 | 00101000
Cin        0
Cout       0
Sum[7:0]   01111111   00000000 | 01000111 | 01001000 | 10000000 | 01111111
```

Figure 4: Simulation for task2

Then I will describe how ripple adder works. The ripple carry adder just do what full adder did but it can calculate several bits numbers. It will calculate from low digit to high digit. The Cout of low digit will be the Cin for the next digit and then we can get the final result.

Task3: Carry Select Adder

Here is the schematic and simulation result of task3,

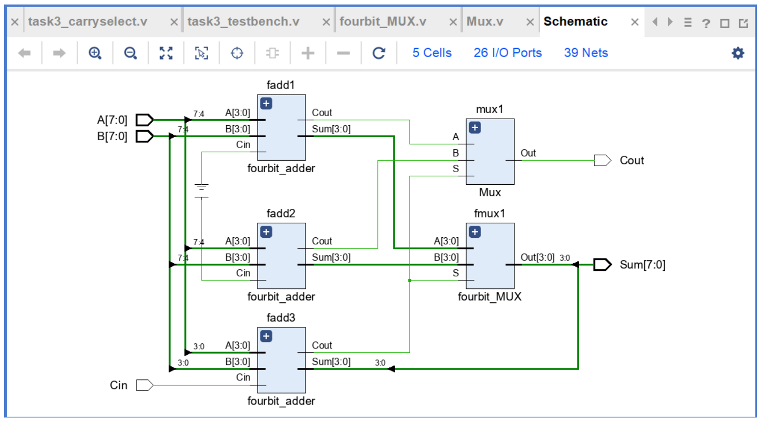
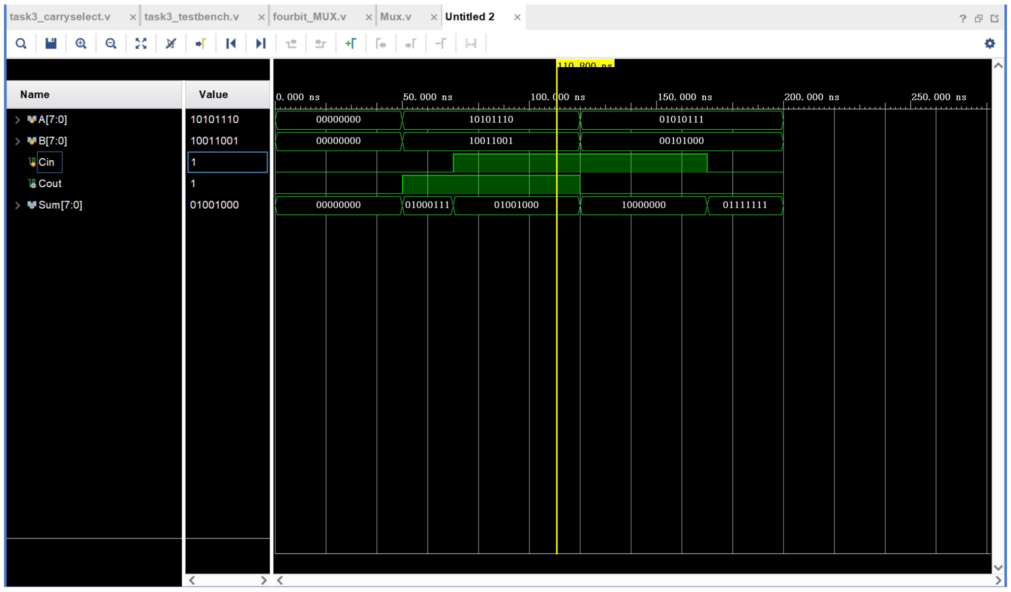


Figure 5: Schematic for task3



Name	Value			
A[7:0]	10101110	00000000	10101110	01010111
B[7:0]	10011001	00000000	10011001	00101000
Cin	1			
Cout	1			
Sum[7:0]	01001000	00000000	01000111, 01001000	10000000, 01111111

Figure 6: Simulation for task3

Then I will describe how carry select adder works. For example, if we calculate two 8 bits binary number. We can first assume Cin=0 or Cin=1 to add the upper four digits. At the same time, we can calculate the lower four digits and get a Cout from this calculation. Then we use a 4 bits 2:1 MUX to decide which is the real sum of the upper four digits. Besides, we need a 1 bit 2:1 MUX to decide which is the real Cout.

Task4: ALU

Here is the schematic and simulation result of task4,

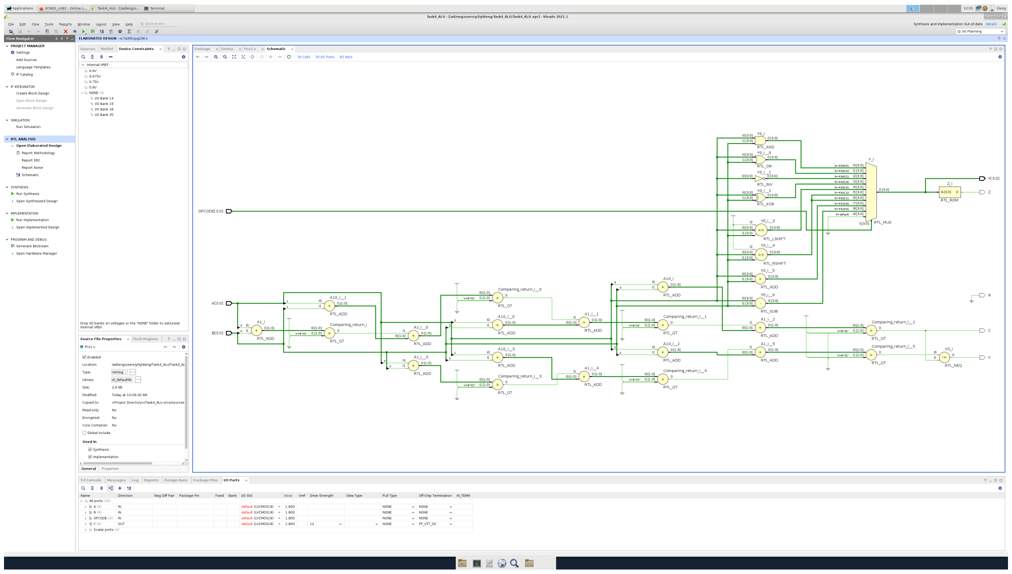


Figure 7: Schematic for task4

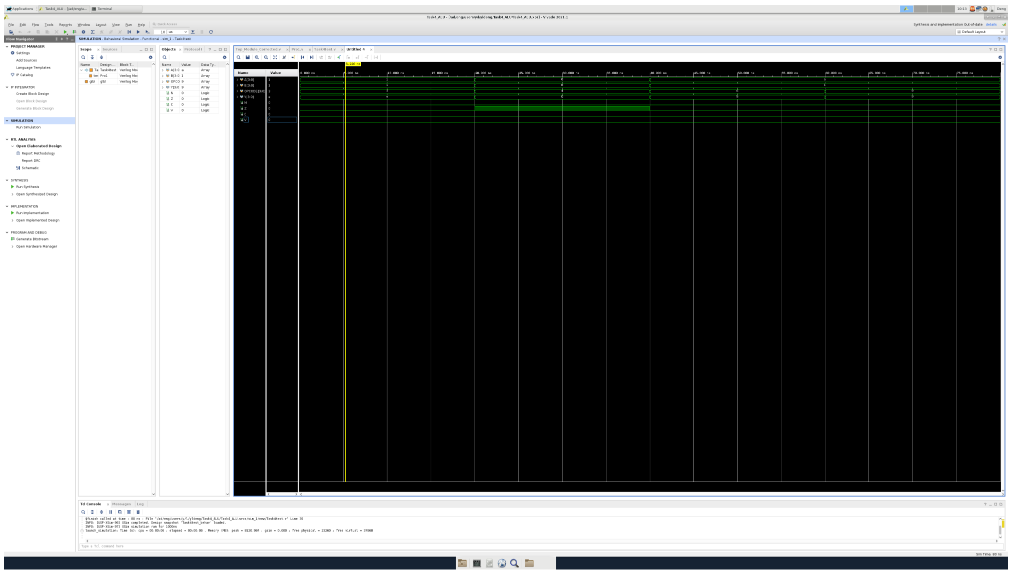


Figure 8: Simulation for task4

In the Pro1.v, I write four functions to simulate the bit adder process and catch the Carryin and Carryout. The first comparing function is used for compare. If input A larger than input B, it will return 1. Otherwise, return 0. The function BitAdder is used to calculate if there

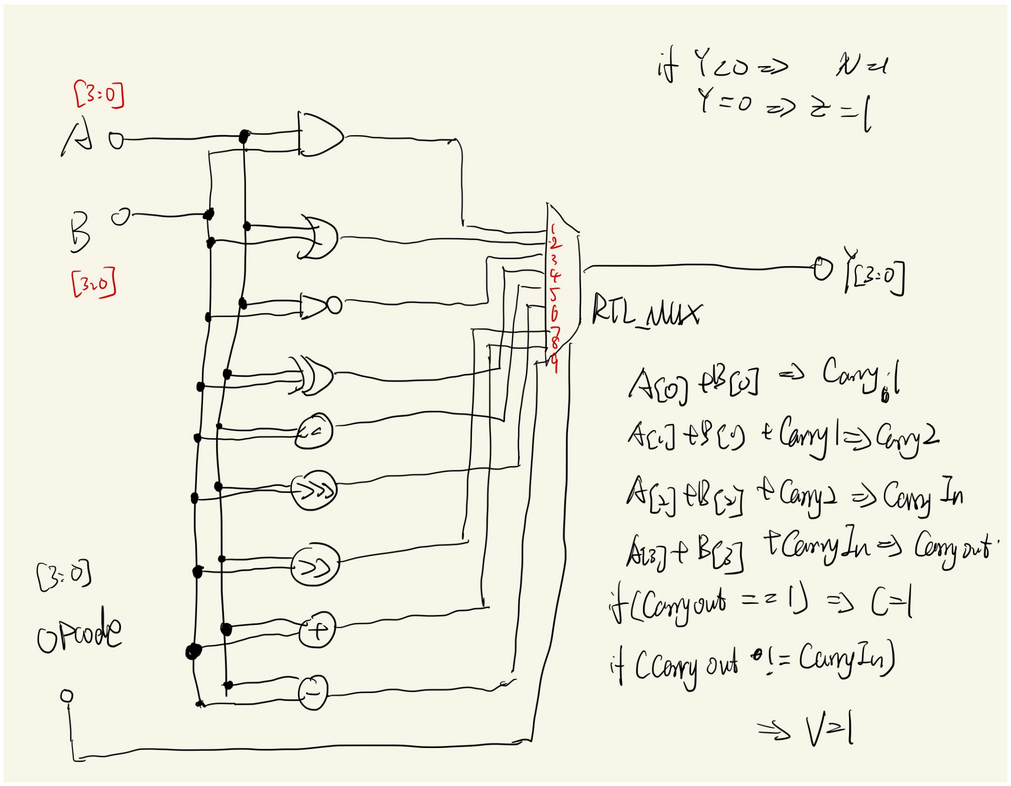

if Y<0 => N=1
Y=0 => Z=1
[3:0]
A
B
[3:0]
RTL_MUX
Y[3:0]
A[0]+B[0] => Carry 1
A[1]+B[1] + Carry1 => Carry2
A[2]+B[2] + Carry2 => Carry In
A[3]+B[3] + Carry In => Carry out
if (Carryout == 1) => C=1
if (Carry out != Carry In)
=> V=1
[3:0]
OPcode

Figure 9: block diagram for task4.2

is the Carryout after the 1-bit calculation. The main idea is that if the result of bitA plus bitB plus Carryin is larger than 1, it will produce a Carryout 1. Otherwise, it produces 0. The Carryin of the lowest bit adder is equal to 0. The third and forth function Calcout and Calcin aims to simulate the process of four-bit adder. During these two processes, we will catch the final Carryout and last-bit Carryin as the output.

After that, I write an always-case block to determine the way of processing inputA and inputB, based on the given input OPCODE. At the end, output the flag based on the different case. 1. If Y is smaller than 0, flag N = 1; 2. If Y = 0, flag Z = 1; 3. If Carryout = 1, flag C = 1; 4. If Carryout != Carryin, flag V = 1;

In the TopModuleCorrected.v, I combine the input with 12 switches and output with 4 leds and seven-segment display. Using three always block to show the output Y in hex on seven-segment display. I follow the instruction: https://www.fpga4student.com/2017/09/seven-segment-led-display-controller-basys3-fpga.html