ECE 385

Fall 2020

Experiment #4

Introduction to System Verilog, FPGA, CAD, and 16-bit adder

Name: Xiao Shuhong & Lu Yicheng

Lab Section: D225

1. Introduction

1.1 Introduction of adder experiment

the circuit we build in lab 4 is a 2*8-bits logic processor consists of a register unit, a computation unit, a routing unit and a control unit.

the register unit is made up of two 8-bit shift registers used to hold the value.

the computation unit can realize 8 kinds of bitwise operation: AND, NAND, OR, NOR, XOR, XNOR, 1111 and 0000.

the routing unit choose where the result goes to, 4 kinds of mode can be chosen: remain unchanged, exchange A and B, keep A and result goes to B, keep B and result goes to A.

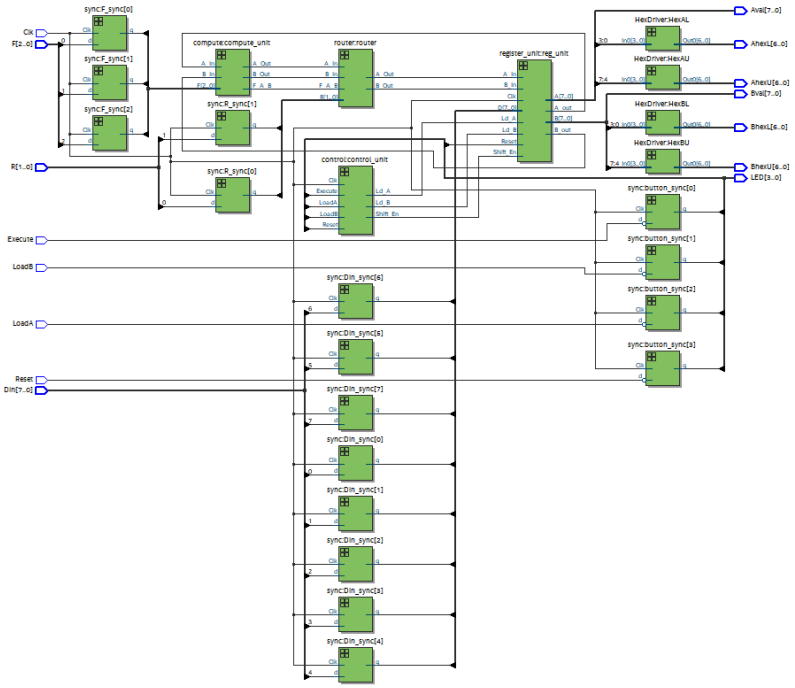
the control unit, based on a Mealy state machine, accept LoadA, LoadB, Excute and Clock as input, to decided whether any operation begin or keep unchanged.

1.2 Introduction of processor experiment

Three adders have same high-level function. They all receive two 16bits input A and B and return the output sum and carry out. One can first use switches and LOADB button to load the expected value to B. And then use switches to represent A. Finally, after RUN button is pressed, the correct answer will be displayed by LEDs in Binary form. Also, when the RESET button is pressed, all things will be cleared.

2 Part 1 - Serial Logic Processor

2.1 block diagram



2.2 description- what we done to extend processor from 4 bits to 8

For module reg_4:

1, extend input logic D, output logic Data_Out to 8 bits.

2, when reset is high, we load 8'h0 to Data_Out istead of 4.

3, When shift_en is high, we load { Shift_In, Data_Out[7:1] } instead of { Shift_In, Data_Out[3:1] }

For register_unit:

1, extend input logic D, output logic A and B to 8 bits.

For processor:

1,extend input logic Din , output logic Aval, logic A, B, Din_S to 8 bits.

2, add two extra HexDriver to represent A[7:4] and B[7:4].

3, extend Din_sync[3:0] to Din_sync[7:0].

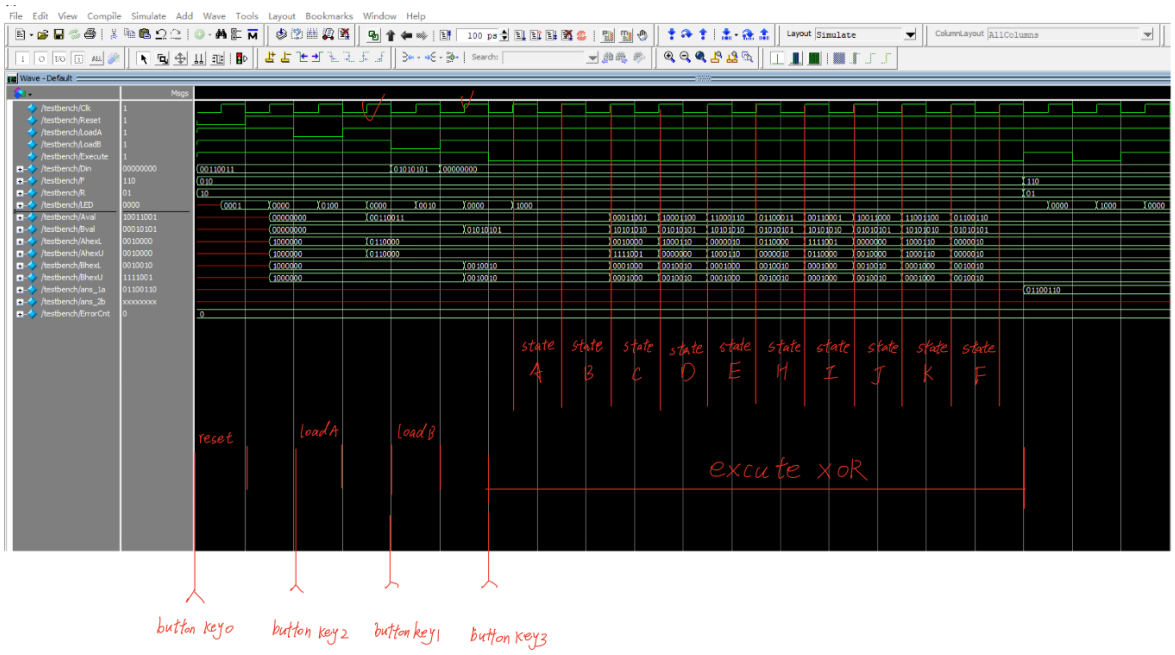
For control:

1, We extend 6 states to 10 states. This is because we have to do bitswise operation 8 times instead of 4. The extra states are used to let registers shift. In this case, we only need to declare that's the next states when stay in these extra states. What should be done during these extra states are already defined in the default case.

For "router" and "compute", nothing needs to be done since "compute" and only "router" only focus on bitwise operation. In other words, they do not care whether the processor is 2*4 bits or 2*8 bits.

2.3 Include a simulation of the processor that has notes that give information such as what operation is being performed, where the result was stored, etc.

2.3.1



State A is the “wait state”. State F is the “have already finished the operation, and if execute==0, go to state A” state.

Function：

XOR

Routing：

The result will be loaded to register A while register B stay the same.

Input：

A: 0011 0011

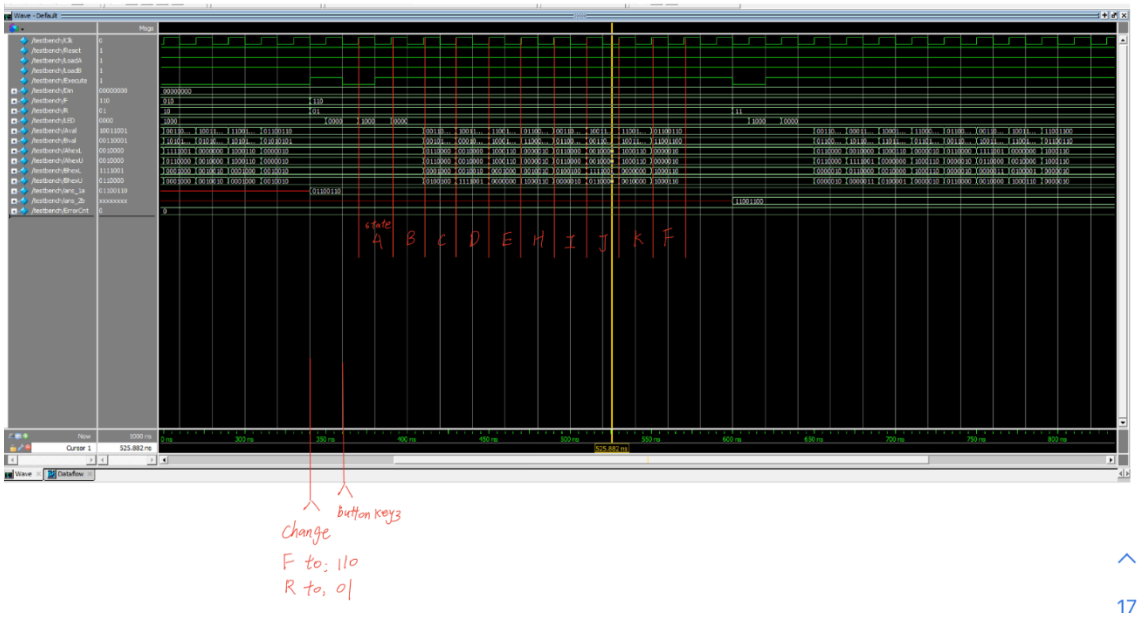
B: 0101 0101

Output：

A: 0110 0110

B: 0101 0101

2.3.2



17

Function：

XNOR

Routing：

The result will be loaded to register B while register A stay the same.

Input：

A: 0110 0110

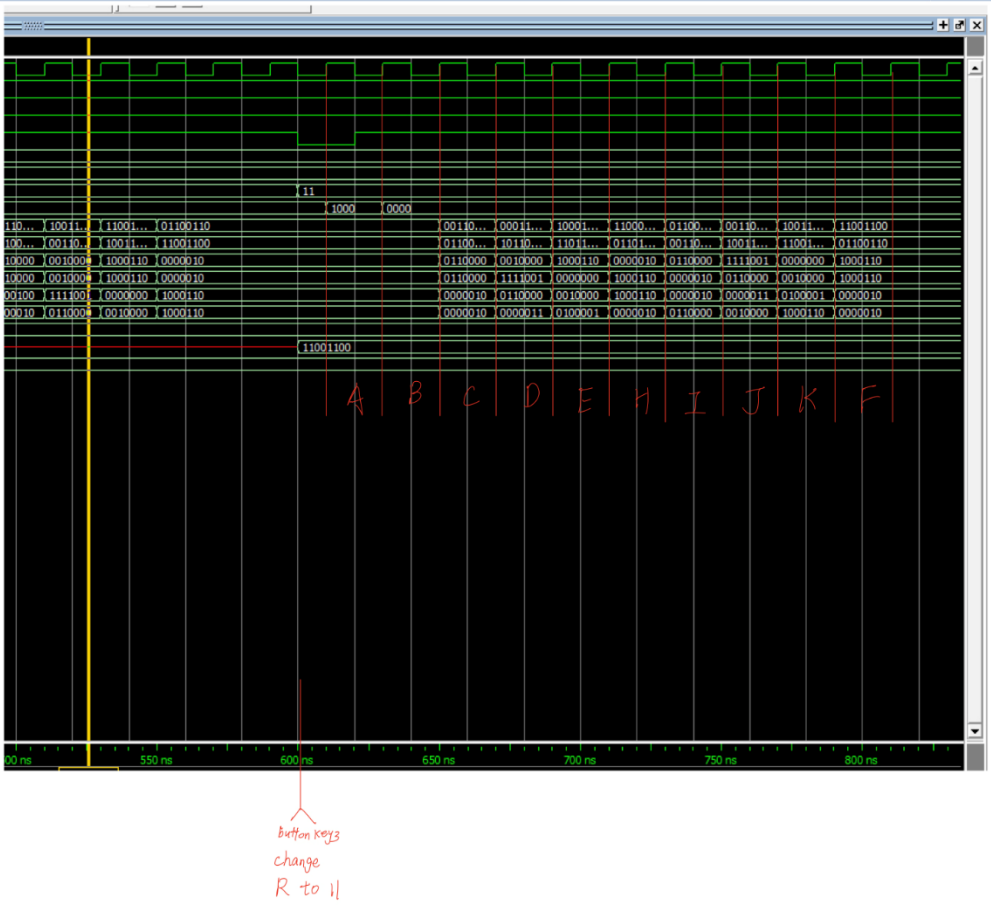
B: 0101 0101

Output：

A: 0110 0110

B: 1100 1100

2.3.3



Function：

XNOR

Routing：

A and B will swap their value

Input：

A: 0110 0110

B: 1100 1100

Output：

A: 1100 1100

B: 0110 0110

3,Part 2 – Adders

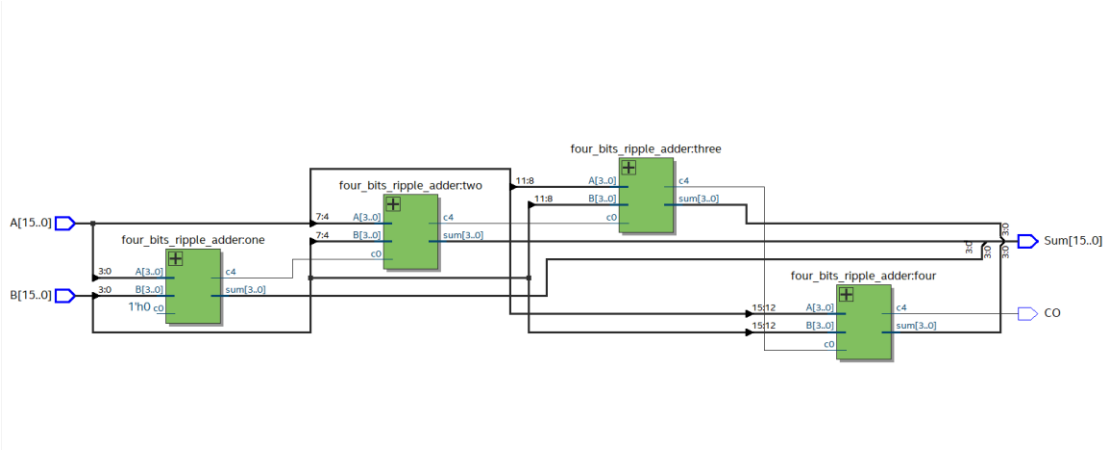
a. carry ripple adder

i. written description of the architecture of the adder

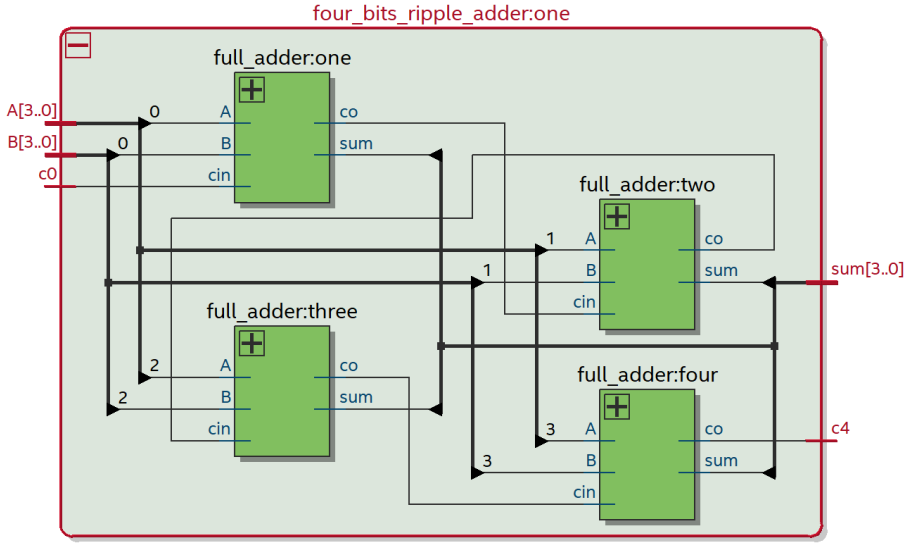
Our 16-bit carry ripple adder is just a serial connection of 4 4-bit carry ripple adders, while each 4-bit carry ripple adder is a serial connection of single carry ripple adder with expression Sum=A^B^Cin, Cout=(A&B)|(B&Cin)|(A&Cin).

The unit structure here in carry ripple adder is single-bit carry adder. Signal Cout of i^{th} adder is the Cin of $i+1^{th}$ adder.

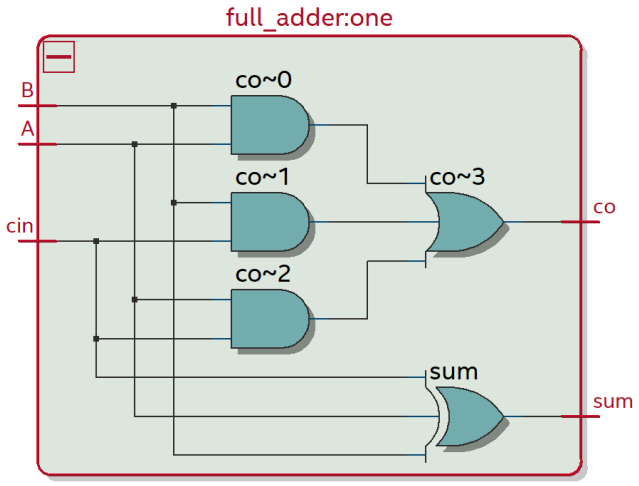
ii. block diagram



16-bit carry ripple adder



4-bit carry ripple adder



single bit carry ripple adder

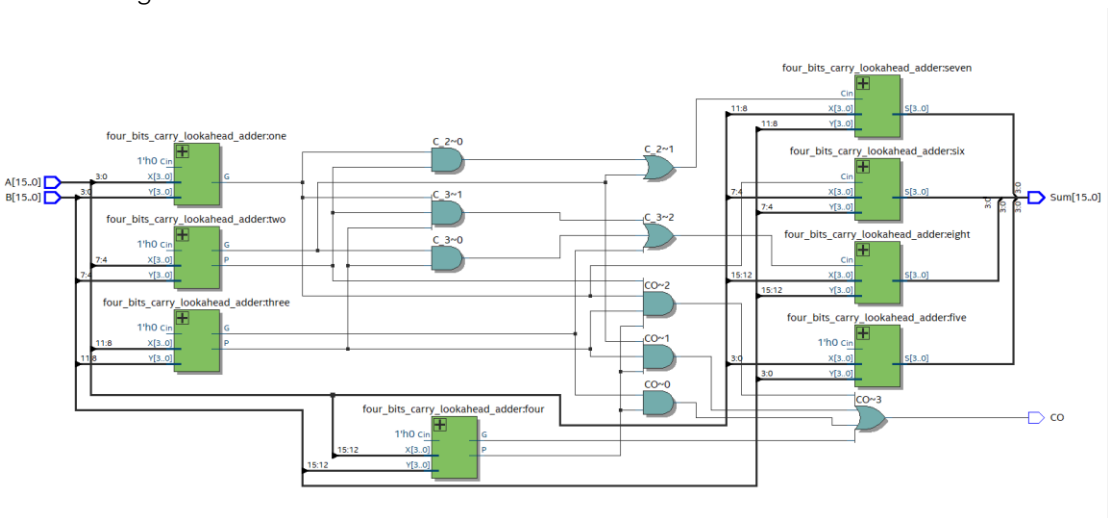
b. carry lookahead adder

i. written description of the architecture of the adder

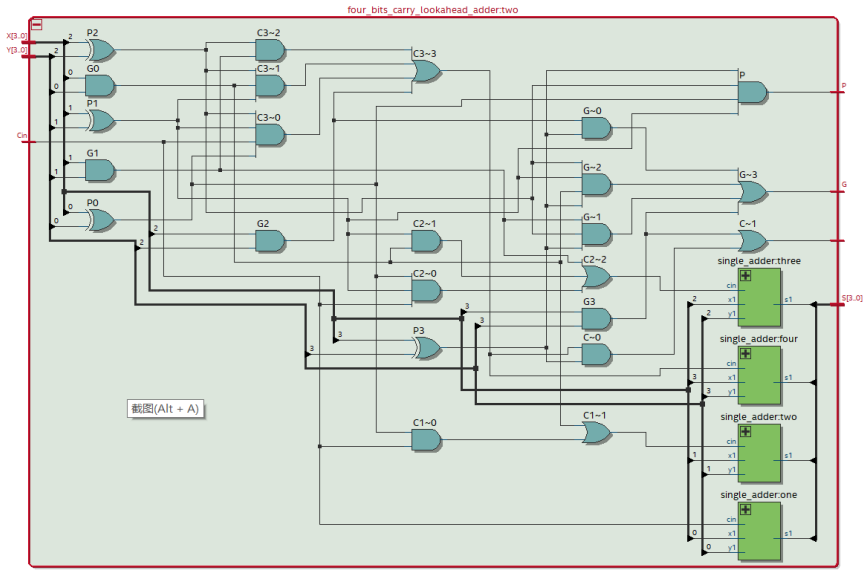
The unit structure here in carry lookahead adder is still single bit carry adder, while instead of form 4-bit carry ripple adder, we construct 4-bit lookahead adder. We use concept of generating(G) and propagating(P) signal to help us predict what its carry-out for any value of its carry-in for each single carry adder. G will be 1 if and only if both A and B be 1(G=A&B), P will be one if and only if either A or B is 1(P=A^B).

That is, Carry-out of i^{th} adder C_{i+1} can be represented as P_i, G_i and Carry-in C_i as $C_{i+1} = G_i|(P_i \& C_i)$. Then, our 4-bit lookahead adder can be connected serial to build 16-bit lookahead adder.

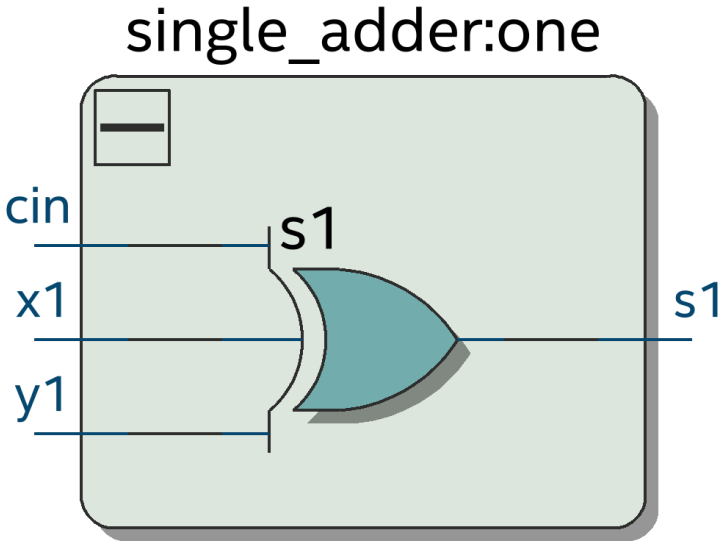
ii. block diagram



16-bit lookahead adder



4-bit lookahead adder



single bit carry ripple adder

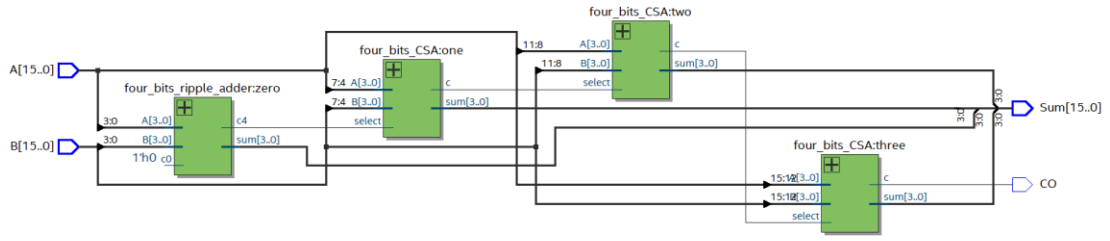
c. carry select adder

i. written description of the architecture of the adder

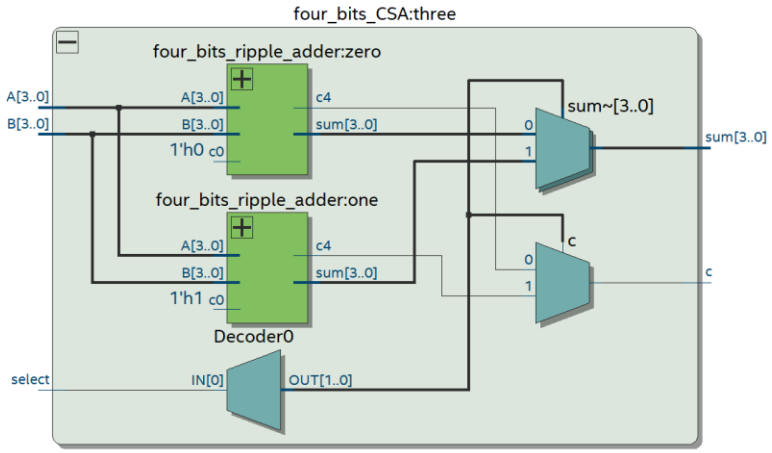
For our carry select adder, the unit structure here is still single-bit carry ripple adder. First, we construct 4-bit carry select adder which use 8 single-bit carry ripple adder, for each bit, we have two single bit ripple adder, one accept carry-in 0 while another one accept 1, that is, we have all kind of possible path of each carry-in and carry-out. For 16-bit carry select adder, we just serially connect 4 4-bit carry select adder.

One more thing to modify from what describe above is for the least significant bit of 16-bit carry select adder, as carry-in must be 0, we can just use 1 single-bit carry ripple adder for this bit. So, totally, we need 31 single-bit adder.

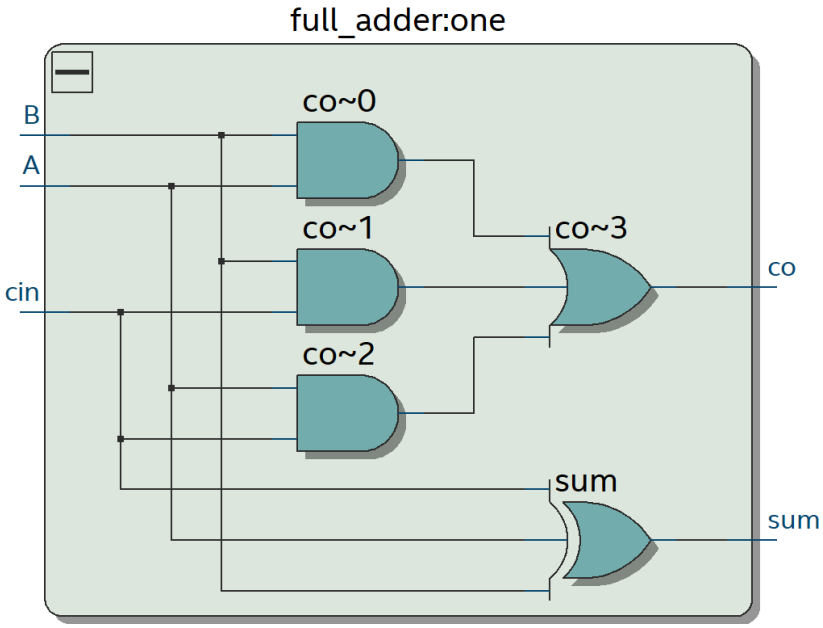
ii. block diagram



16-bit carry select adder



4-bit carry select adder



single-bit carry ripple adder

d. Describe at a high level the area, complexity, and performance tradeoffs.

for carry-ripple adder, the idea is very simple, that is, just serial connect 16 single adders. Totally, around 48 gates are needed to build this module. The system Verilog code is also simple to describe it, we first generate 4-bit adder by using four single bit adder module, assign carry-out of right bit to left bit as carry-in. Then for 16-bit module, 4 4-bit modules are used, assign carry-out of right bit to left bit as carry-in. However, it is not so satisfied on performance, every single adder should wait until its right adder finish operation and spread carry-out, that is, we will have 16 delays from the beginning of operation to the end.

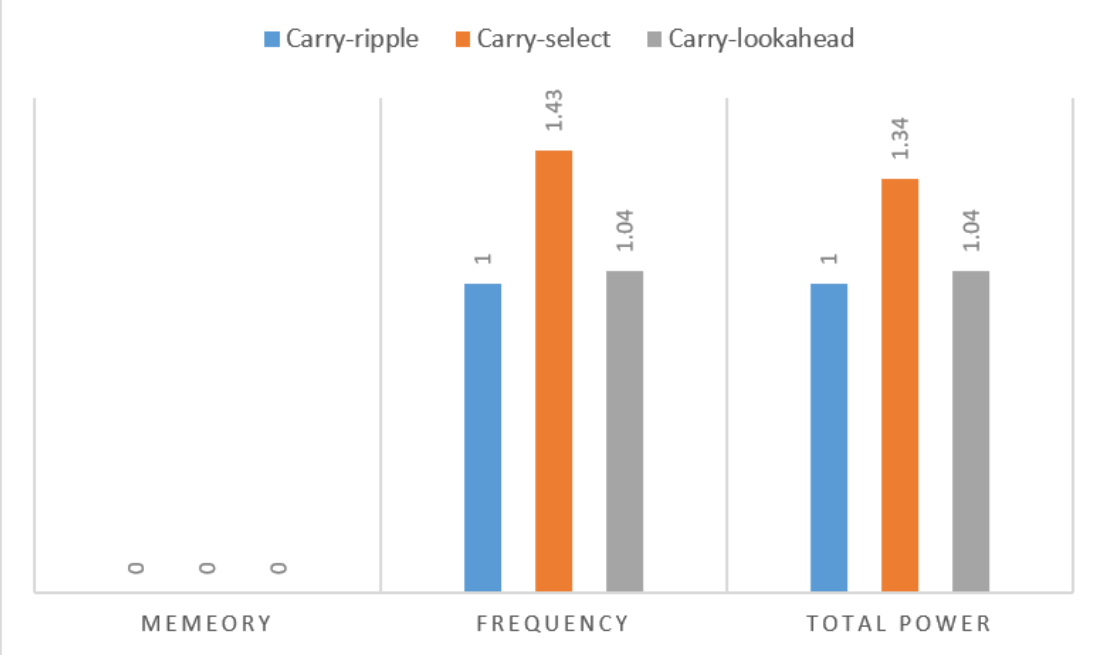
for carry-lookahead adder, around 140 gates are need which is much larger in area than carry ripple adder. and, the code is also much more complexity, for at least we need so many inter-value P and G, as well as the connection. For performance, it is different from carry ripple adder that we don't wait carry-out from last bit to help us calculate, all the signal P and G can be determined at the same time as only A and B affect them. That is, we will have much less delay than carry ripple adder. From the addition begin, after 2 delay, we get P0 and G0, as well as all P and G. Then for 3 more delays, we arrived at the final 4-bit adder, from C12 coming, 3 more delays are needed and we can get final answer, that is, 8 bits in total, compare with 16 bits for carry ripple adder.

for carry select adder, it also needs more area and a much complexity code than carry ripple adder. The idea of carry select adder is using more circuit to achieve less operation time. Around 80 gates are needed to achieve it, and in code, we need to achieve mux by if-else or case compare with the simplest serial connection in carry ripple adder. For performance, it is greater than carry ripple adder, for every bit we calculate two possible situations: with carry-in 0 and 1. When the carry-out of the first 4-bit adder arrived, we can determine the carry-out of our second adder verry quickly through mux, without waiting to calculate A and B, as well as for

third and fourth adder. That is, for the first adder, 4 bits are used as it is same as a carry ripple adder, and then 1 delay for the rest of 3. that is, after 7 delays, we will get the result.

e. Document the performance of each adder

	Carry-ripple	Carry-select	Carry-lookahead
Memory (BRAM)	0	0	0
Frequency	1(62.1MHZ)	1.43(88.9MHZ)	1.34(83.28MHZ)
Total Power	1(155.80nW)	1.04(162.62mW)	1.04(162.31mW)



4. Answers to the 2 Post lab Questions

4.1 Compare the usage of LUT, Memory, and Flip-Flop of your bit-serial logic processor exercise in the IQT with your TTL design in Lab 3. Make an educated guess of the usage of these resources for TTL assuming the processor is extended to 8-bit. Which design is better, and why?

IQT:

LUT	72
Memory (BRAM)	0
Flip-Flop	63

4 bits TTL:

LUT	22
Memory (BRAM)	0
Flip-Flop	11

Guess: 8 bits TTL:

LUT	25
Memory (BRAM)	0
Flip-Flop	24

Sine we need only a little extra combinational logic to extend(just need more states), so we guess LUT is 25.
Sine we need have to double the flip-flops to extend 4 bits to 8 bits, so we guess the Flip-Flop id 24.

TTL is much better than the IQT. The reason is that TTL design is a low-level design while SQT design is a relatively high design. As a result, we can simply every small component in the circuit under TTL design which is impossible in the SQT design.

4.2 Is CSA in this lab ideal? If not, how to design an ideal one on FPGA?

No. Thinking we have N adders to form the CSA. Then to have the min delay, the first adder is X bits, the second adder is X bits, the third adder is X+1 bits, the fourth adder is X+2 bits and so on.

If N=4:

X+X+(X+1)+(X+2)=16 have no solution.

If N=5:

X+X+(X+1)+(X+2)+)+(X+3)=16 =>X=2

So, we have delay 2+4=6. While the CSA have delay 4+3=7.

4.3 For the adders, refer to the Design Resources and Statistics in IQT.16-18 and complete the following design statistics table for each adder. This is more comprehensive than the above design analysis and is required for every System Verilog circuit.

ripple_adder:

LUT	114
DSP	0
Memory (BRAM)	0
Flip-Flop	105
Frequency	62.1 MHz
Static Power	98.55 mW
Dynamic Power	2.92 mW
Total Power	155.80 mW

Carry lookahead adder

LUT	131
DSP	0
Memory (BRAM)	0
Flip-Flop	105
Frequency	83.28 MHz
Static Power	98.57 mW
Dynamic Power	6.65 mW
Total Power	162.31 mW

Carry select adder

LUT	125

DSP	0
Memory (BRAM)	0
Flip-Flop	105
Frequency	88.9 MHz
Static Power	98.57 mW
Dynamic Power	6.89 mW
Total Power	162.62 mW

Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot makes sense? Are they complying with the theoretical design expectations, e.g., the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder? Which design consumes more power than the other as you expected, why?

Yes, they make sense. Some comply with the theoretical design expectations.

For LUT:

Carry lookahead adder have more LUTs then other two adders. This makes sense since CLA need lots of combinational logic to compute P and G.

For Flip-Flop:

We only use Flip-Flop in the top-level. As a result, all three adders should have same numbers of Flip-Flop.

For Frequency:

the maximum operating frequency: Carry select adder > carry-lookahead adder > carry-ripple adder. This makes sense because in our estimation, carry-lookahead adder Carry select adder have only 7 delays and Carry select adder have 8 delays.

For Power:

The CLA consume the maximum power. CLA and CSA consume more power then ripple adder. Because these two adders need more combinational logic

5,Conclusion

5.1 Describe any bugs and countermeasures taken during this lab.

1, we were very confused about the use of synchronizer at first. Because we think we can still get correct result without it. However, the key point of using synchronizer is that we can make our design synchronized. The absence of synchronizers may fail the entire design thought not in this lab.

2, When design the carry lookahead adder, we first design in this way:

```
module four_bits_carry_lookahead_adder(
  input  logic[3:0] X,
  input  logic[3:0] Y,
  input  logic   C0,
  input    logic       G0,
  input    logic       P0,
  output logic[3:0] S,
  output  logic   G,
  output   logic       P,
  output   logic       C

);
  logic G1,P1,C1,G2, P2,C2,G3,P3,C3;

  single_adder one(.x1(X[0]), .y1(Y[0]), .c0(C0), .p0(P0), .g0(G0),  .s1(S[0]), .g1(G1), .p1(P1), .c1(C1) );
  single_adder two(.x1(X[1]), .y1(Y[1]), .c0(C1), .p0(P1), .g0(G1),  .s1(S[1]), .g1(G2), .p1(P2), .c1(C2) );
  single_adder three(.x1(X[2]), .y1(Y[2]), .c0(C2), .p0(P2), .g0(G2),  .s1(S[2]), .g1(G3), .p1(P3), .c1(C3) );
  single_adder four(.x1(X[3]), .y1(Y[3]), .c0(C3), .p0(P3), .g0(G3),  .s1(S[3]), .g1(G), .p1(P), .c1(C) );


endmodule
```

So, we can not get correct C1,C2,C3 at the same time. Because C2 depend on C1, C3 depend on C2. We have to write in this way:

```
assign C0=Cin;
assign C1=Cin&P0|G0;
assign C2=(Cin&P0&P1)|(G0&P1)|G1;
assign C3=(Cin&P0&P1&P2)|(G0&P1&P2)|(G1&P2)|G2;
```

So that we can get C1,C2,C3 at the same time.

5.2 Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

The lab manual is quite clear and no need to be improved. We get the key idea about the three different adders through the short description in manual.

5.3 Any additional summary you want to include

In this lab, we learn how to write a state machine and learn what can be write in always_comb and what can not. Generally speaking, we think should strictly follow the following rule:

1, It is better to write all combinational logic in always_comb. In other word, it is better to not include the if, case, = in always_ff.

2, if we use if or case, we must define all possible condition.

3, when we construct state machine, we can not set A=1 in state a but say nothing about A in other state(unless you set A defiantly).