ECE 385

Fall 2019

Experiment #4

Introduction to SystemVerilog, FPGA, EDA, and 16-bit Adders

Pouya Akbarzadeh and Patrick Stach

Section: ABC

Lab TA: Vikram Anjur, Yuhong Li

4.1 Introduction

4.1.1 The Goal of the Lab

In this lab we built four different circuits. First part of the lab we extended a 4-bit processor to be an 8-bit processor. In the second part we built three different adders. These adders are the carry ripple adder, carry lookahead adder, and finally the carry select adder. The 8-bit processor was similar to the one built in lab 3 (TTL lab). We just had to expand it. More details of this process can be seen in section 4.2.2.

4.1.2 The Three Adders

§ **Carry Select Adder**

Carry select adder is where the logic computes the (n+1) bit sum of two numbers (with n-bits).

§ **Carry Look-ahead adder**

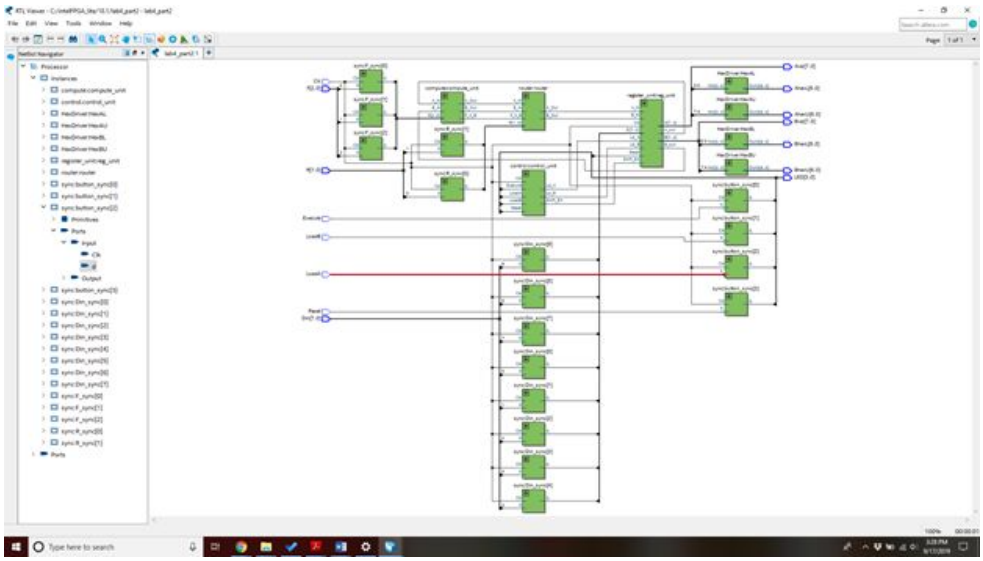
Carry look-ahead adder is an adder that improves the speed of the calculation by taking into account both cases possible (with or without carry in).

§ **Ripple Adder**

A ripple carry adder uses the carry-out of each full adder is the carry in of the succeeding next most significant full adder.

4.2 Part 1 – Serial Logic Processor

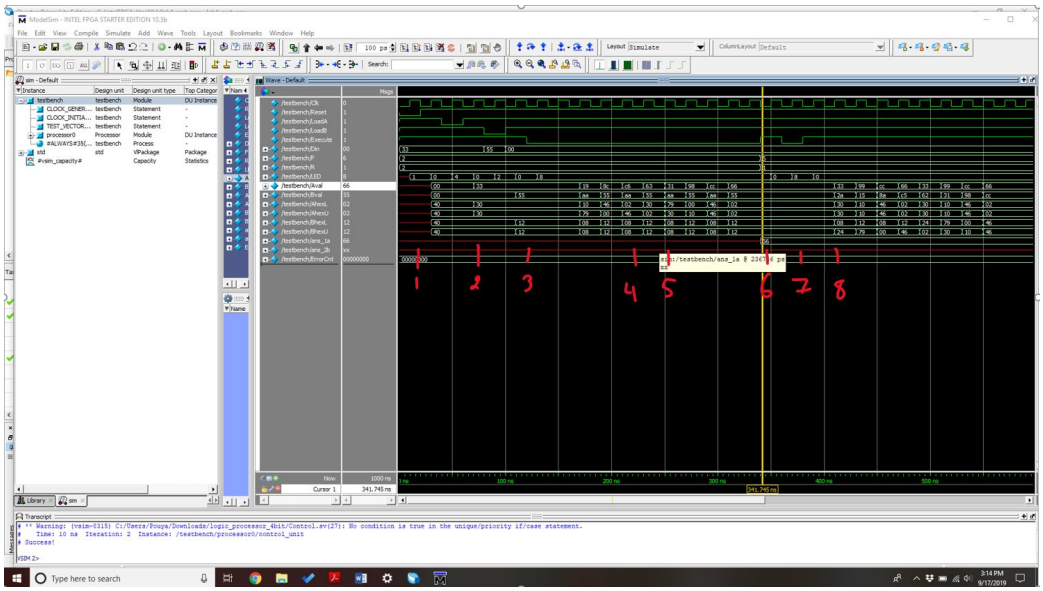
4.2.1 Block Diagram



4.2.2 Explanation

Since most of the files were given to us, there were a few changes needed to be made to go from an 4-bit to 8-bit processor. The given FSM for this control logic had five states. However, for an 8-bit processor we needed 10 states. A 4-bit enum logic took care of that. Since these new states were added we also needed to add states to the “always_comb” cases. The rest of the changes were rather simple. Anything that was [3:0] (4-bits) we had to replace with [7:0] (8-bits) since there was more data. These changes were made in the processor.sv file. Instead of using the Reg_4.sv we had to use Reg_8.sv instead. We also changed a few more [3:0] instances to [7:0]. Finally the register_unit.sv was modified just like the others with [3:0] to [7:0].

4.2.3 Simulation



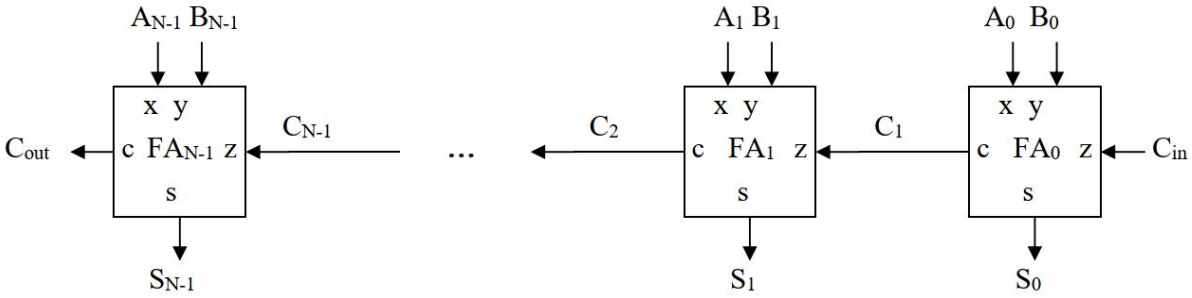
The diagram shows that LoadA and LoadB become active then A and B go through the OR operation. Then followed by A and B are go through XOR operation. Then it stores it into register A. Then A and B are XNORed. Output should be routed toward B. Then register values are swapped.

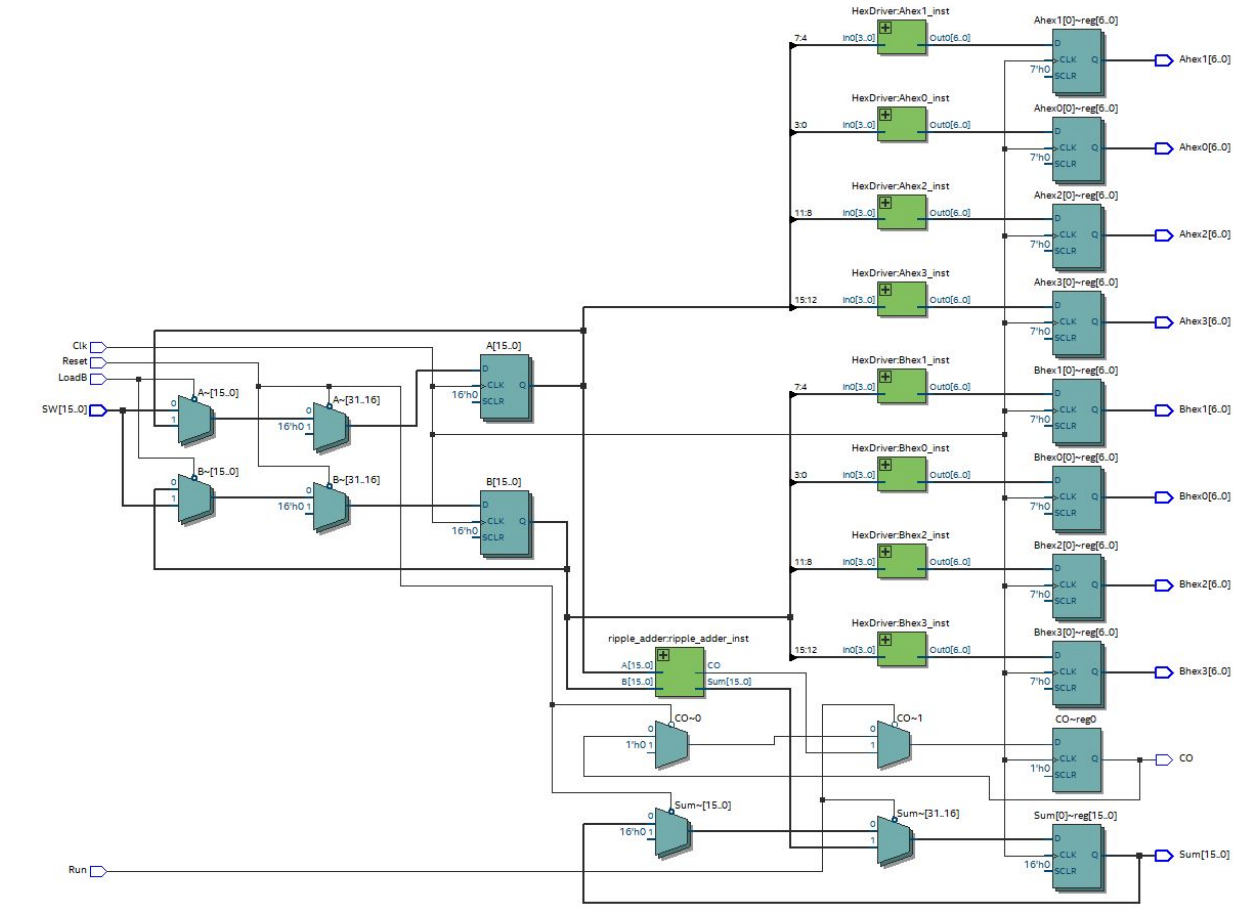
4.3 Part 2 – Adders

4.3.1 Ripple Adder

Ripple adder is a circuit in which we can calculate the sum of two binary numbers. It can be made with full adders. The full adders are connected in a cascade format with the carry out from each full adder to the carry-in of the next full-adder in the chain. While the design is rather simple it is slow. It is slow because the full adder has to wait for the carry-bit to be calculated.

Diagram:




Clk
Reset
LoadB
SW[15..0]
Run
A~[15..0]
A~[31..16]
B~[15..0]
B~[31..16]
A[15..0]
B[15..0]
ripple_adder:ripple_adder_inst
A[15..0]
B[15..0]
CO
Sum[15..0]
CO~0
CO~1
Sum~[15..0]
Sum~[31..16]
HexDriver:Ahex1_inst
HexDriver:Ahex0_inst
HexDriver:Ahex2_inst
HexDriver:Ahex3_inst
HexDriver:Bhex1_inst
HexDriver:Bhex0_inst
HexDriver:Bhex2_inst
HexDriver:Bhex3_inst
Ahex1[0]~reg[6..0]
Ahex0[0]~reg[6..0]
Ahex2[0]~reg[6..0]
Ahex3[0]~reg[6..0]
Bhex1[0]~reg[6..0]
Bhex0[0]~reg[6..0]
Bhex2[0]~reg[6..0]
Bhex3[0]~reg[6..0]
CO~reg0
Sum[0]~reg[15..0]
Ahex1[6..0]
Ahex0[6..0]
Ahex2[6..0]
Ahex3[6..0]
Bhex1[6..0]
Bhex0[6..0]
Bhex2[6..0]
Bhex3[6..0]
CO
Sum[15..0]

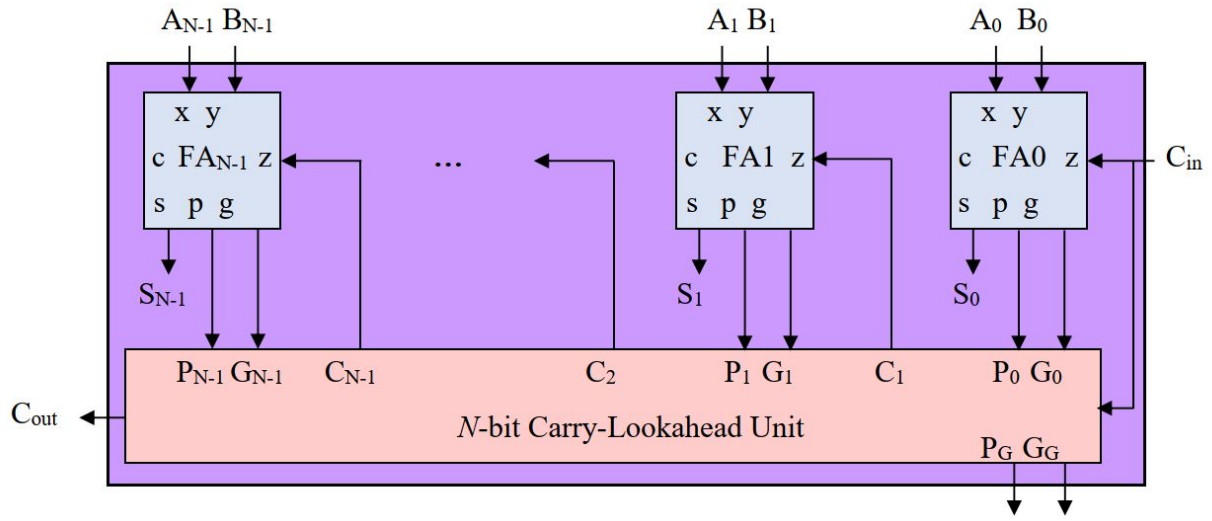
```
module ripple_adder
(
    input   logic[15:0]     A,
    input   logic[15:0]     B,
    output  logic[15:0]     Sum,
    output  logic           CO
);

//4x4 design

logic c0, c1, c2 ;
four_bit_ra FRA0(.x(A[3:0]),     .y(B[3:0]),    .cin(0),    .s(Sum[3:0]),     .cout(c0));
four_bit_ra FRA1(.x(A[7:4]),     .y(B[7:4]),    .cin(c0),   .s(Sum[7:4]),     .cout(c1));
four_bit_ra FRA2(.x(A[11:8]),    .y(B[11:8]),   .cin(c1),   .s(Sum[11:8]),    .cout(c2));
four_bit_ra FRA3(.x(A[15:12]),   .y(B[15:12]),  .cin(c2),   .s(Sum[15:12]),   .cout(CO));
endmodule


module four_bit_ra(
                    input [3:0] x,
                    input [3:0] y,
                    input cin,
                    output logic [3:0] s,
                    output logic cout

                    );

//making 4bit FA
//we take 4 1-bit adders and combine them
   logic c0, c1, c2;

   full_adder fa0(.x(x[0]), .y(y[0]), .cin(cin), .s(s[0]), .cout(c0));
   full_adder fa1(.x(x[1]), .y(y[1]), .cin(c0), .s(s[1]), .cout(c1));
   full_adder fa2(.x(x[2]), .y(y[2]), .cin(c1), .s(s[2]), .cout(c2));
   full_adder fa3(.x(x[3]), .y(y[3]), .cin(c2), .s(s[3]), .cout(cout));
   endmodule

//this is a full adder

module full_adder(
                    input x,
                    input y,
                    input cin,
                    output logic s,
                    output logic cout
                    );
//logic for sum and carry out
 assign s = x^y^cin;
 assign cout = (x&y) | (y&cin) | (cin&x);
endmodule
```

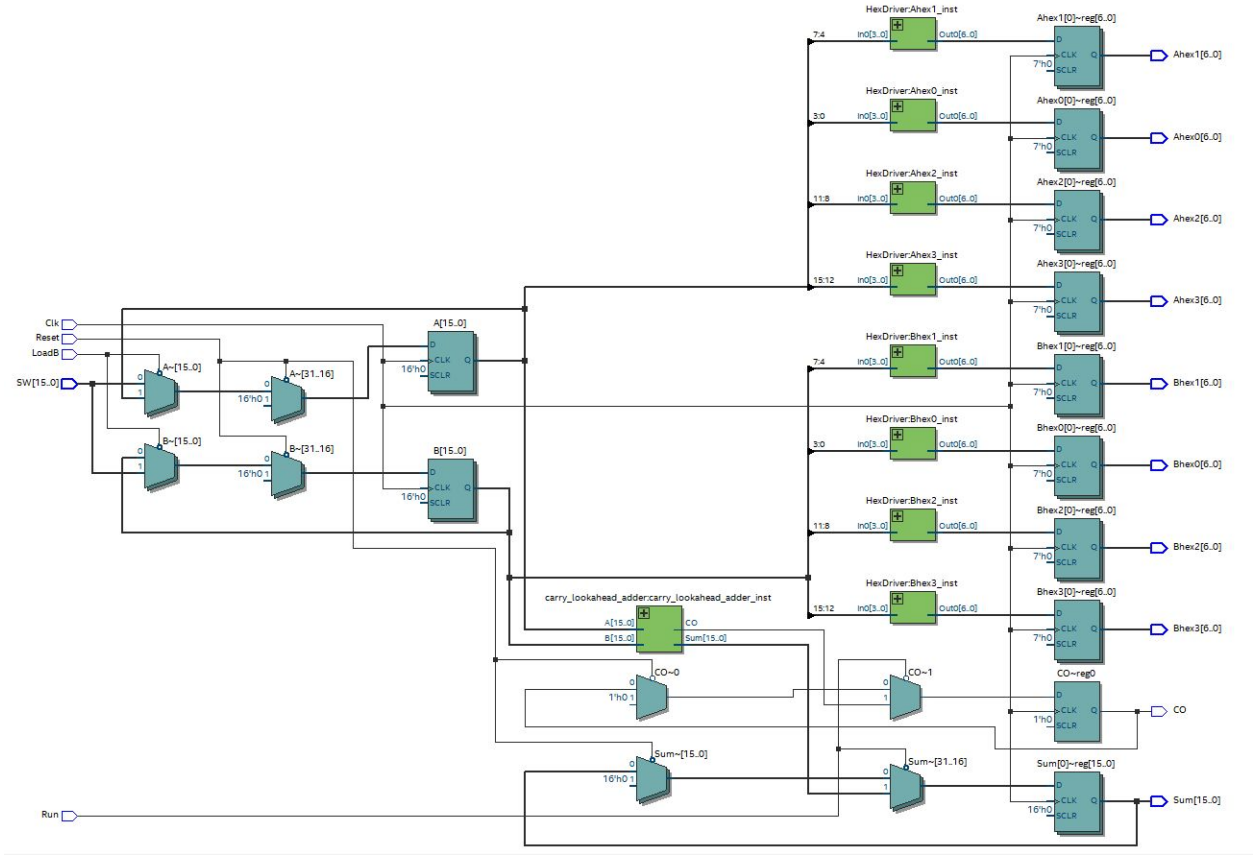
4.3.2 Carry-Lookahead Adder

Carry lookahead adder is a faster circuit to calculate the sum of two binary numbers. This adder reduced the propagation delay by adding more logic. The Carry-lookahead adder works by doing some complex logic. The carry generate and carry propagate are used instead of waiting for actual carry bits. The carry-lookahead adder makes predictions using immediate available inputs and predicts the carryout based on any value the carry-in can be.

```
module carry_lookahead_adder
(
    input   logic[15:0]     A,
    input   logic[15:0]     B,
    output  logic[15:0]     Sum,
    output  logic           CO
);

//4x4 design
//4 4-bit adders to make 16 bit adder
logic c0,c1,c2;
four_bit_lookahead_carry CLA0(.A(A[3:0]),       .B(B[3:0]),       .cin(0),       .Sum(Sum[3:0]),         .cout(c0));
four_bit_lookahead_carry CLA1(.A(A[7:4]),       .B(B[7:4]),       .cin(c0),      .Sum(Sum[7:4]),         .cout(c1));
four_bit_lookahead_carry CLA2(.A(A[11:8]),      .B(B[11:8]),      .cin(c1),      .Sum(Sum[11:8]),        .cout(c2));
four_bit_lookahead_carry CLA3(.A(A[15:12]),     .B(B[15:12]),     .cin(c2),      .Sum(Sum[15:12]),          .cout(CO));
    /* TODO
     *
     * Insert code here to implement a CLA adder.
     * Your code should be completly combinational (don't use always_ff or always_latch).
     * Feel free to create sub-modules or other files. */

endmodule

module four_bit_lookahead_carry(
                    input logic[3:0] A,
                    input logic [3:0] B,
                    input logic cin,
                    output logic [3:0] Sum,
                    output logic cout
                    );

                    logic c0, c1, c2, c3;
                    logic G0, G1, G2, G3;
                    logic P0, P1, P2, P3;
                    //logic that was given to us
                    always_comb
                    begin
                    G0=A[0]&B[0];
                    G1=A[1]&B[1];
                    G2=A[2]&B[2];
                    G3=A[3]&B[3];

                    P0=A[0]^B[0];
                    P1=A[1]^B[1];
                    P2=A[2]^B[2];
                     P3=A[3]^B[3];
                    //followed logic that was given to us
                    c0=cin;
                    c1=cin&P0 | G0;
                    c2=cin&P0&P1 | G0&P1 | G1;
                    c3=cin&P0&P1&P2 | G0&P1&P2 | G1&P2 | G2;
                    cout=c3;
                    //assign cout=CO;
                    //followed logic that was given to us
                    Sum[0] = A[0]^B[0]^c0;
                    Sum[1] = A[1]^B[1]^c1;
                    Sum[2] = A[2]^B[2]^c2;
                    Sum[3] = A[3]^B[3]^c3;
                    end


endmodule
```

Diagram:





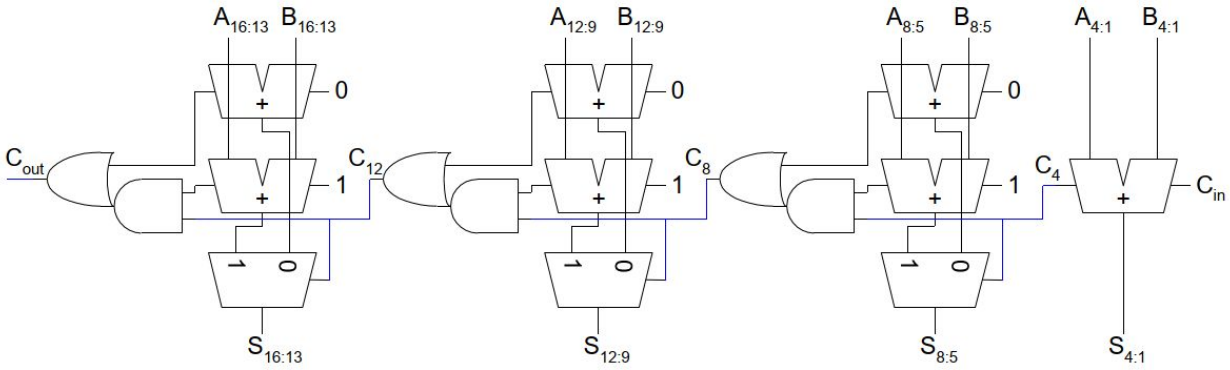
4.3.3 Carry-Select Adder

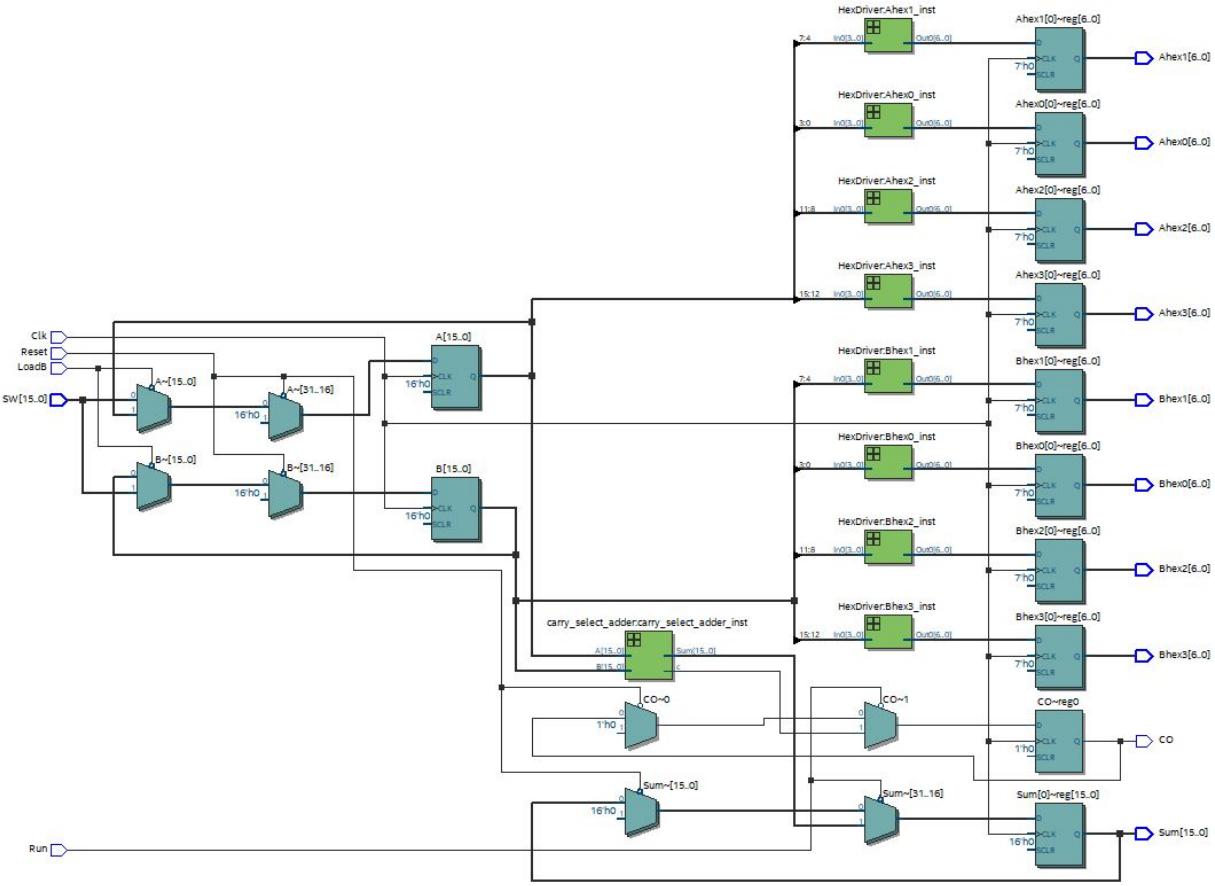
The carry-select adder is made of two ripple carry adders and a multiplexer. The process carry-select adder adds two n-bit numbers twice. One calculation makes the assumption that the carry-in bit is one and the other calculation assumes its zero. Both of these calculations are done and the correct sum and the carry-out is then selected by the multiplexer based on carry-in is

known. This design is better than the ripple adder because you don't have to wait for the calculation of the carry-in every-time. You just pick one of the two options. This does increase complexity, however.

```
module carry_select_adder
(
    input   logic[15:0]     A,
    input   logic[15:0]     B,
    output  logic[15:0]     Sum,
    output  logic           C
);
    // cout0 is always assuming cin=0
    logic cout1_0, cout2_0, cout3_0; //carryout bits assuming cin =0
    logic cout1_1, cout2_1, cout3_1; //carryout bits assuming cin =1
    logic cout0, cout1, cout2; //actual carryout bits

    logic [15:0] Sum_0; //Sum assuming cin=0
    logic [15:0] Sum_1; //Sum assuming cin=1

    // only need one FRA for[3:0] because we know cin=0
    four_bit_ra FRA0(.x(A[3:0]), .y(B[3:0]), .cin(0), .s(Sum[3:0]), .cout(cout0));

    // two FRA for [7:4], one for cin=0, other for cin=1
    four_bit_ra FRA1_0(.x(A[7:4]), .y(B[7:4]),   .cin(0), .s(Sum_0[7:4]), .cout(cout1_0));
    four_bit_ra FRA1_1(.x(A[7:4]), .y(B[7:4]),   .cin(1), .s(Sum_1[7:4]), .cout(cout1_1));

    // two FRA for [11:8], one for cin=0, other for cin=1
    four_bit_ra FRA2_0(.x(A[11:8]), .y(B[11:8]), .cin(0), .s(Sum_0[11:8]), .cout(cout2_0));
    four_bit_ra FRA2_1(.x(A[11:8]), .y(B[11:8]), .cin(1), .s(Sum_1[11:8]), .cout(cout2_1));

    // two FRA for [15:12], one for cin=0, other for cin=1
    four_bit_ra FRA3_0(.x(A[15:12]), .y(B[15:12]), .cin(0), .s(Sum_0[15:12]), .cout(cout3_0));
    four_bit_ra FRA3_1(.x(A[15:12]), .y(B[15:12]), .cin(1), .s(Sum_1[15:12]), .cout(cout3_1));

    mux M1 (Sum_0[7:4], Sum_1[7:4],
          cout1_0, cout1_1,
          cout0,
          Sum[7:4],
          cout1);
    mux M2 (Sum_0[11:8], Sum_1[11:8],
          cout2_0, cout2_1,
          cout1,
          Sum[11:8],
          cout2);
    mux M3 (Sum_0[15:12], Sum_1[15:12],
          cout3_0, cout3_1,
          cout2,
          Sum[15:12],
          C);

endmodule
//mux chooses which case is correct
//thus we dont have to wait for calculation based on carry in
module mux
(     input logic [3:0] A0, A1,
    input logic C0, C1,
    input logic sel,
    output logic [3:0] out,
    output logic Cout);

    always_comb
    begin
       if(sel==0)
       begin
          out <= A0;
          Cout <= C0;
       end
       else
       begin
          out <= A1;
          Cout <= C1;
       end
    end
endmodule
```

Diagram:





4.4 Answers to post-lab questions

4.4.1 Pre Lab Questions

A. Complete the bit-serial logic processor exercise from the Introduction to SystemVerilog and Tutorial (IQT. 1-40). Include a copy of the generated diagram from Quartus of the 8-bit logic processor and the simulation waveform (with annotations) in your Lab 4 lab report.

DONE

B. Design, document, and implement a 16-bit carry-ripple adder, a 16-bit carry-lookahead adder, and a 16-bit carry-select adder in SystemVerilog. Use the provided code (from the website) as a testing framework.

DONE

C. Document design analysis for the three adders in the table below. Plot out the data from the table for comparison studies. Normalize the data across the three adders with the carry-ripple adder. When normalizing, choose data from one the carry-ripple adder as the baseline, and then divide the other two with the baseline number. Say, you got 20 from carry-ripple, 21 from carry-select, and 23 from carry-lookahead, the numbers after normalization becomes 20/20=1.0, 21/20=1.05, 23/20=1.15, respectively. The resulting plot should resemble the one below (the plot below does not use real data).

4.4.2 Post Lab Questions

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?

To answer this, we will first make the distinction between “Memory” and “Flip-Flop”. Since Flip-Flops can be used not only for memory but for other purposes, we will count something as a Flip-Flop if it’s not used for storing data. For example, we will consider a counter to be composed of Flip-Flops, but a shift register will be considered “Memory.” The flip-flops we used to store the next state bits would be considered “Memory”

The design with IQT is better because IQT automatically optimizes the design and uses the most optimised design. For example, our combinational logic may not have been the most efficient and used more gates or chips then necessary, where when IQT would have automatically realized that

2) In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal? If not, how would you go about designing the ideal hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?)

To determine whether a 4x4 hierarchy is ideal we have to consider that there are compromises based on the type of hierarchy used. We have to consider the tradeoffs- if we increase the operational frequency, or speed, we inherently add more complexity which translates to more surface area used on a board, more power consumption, possibly more expensive, etc. Let's consider a 8x2 hierarchy. If we made the individual modules 2 bits, we will only need to wait for the propagation delay of a 2 bit ripple adder compared to a 4-bit adder, and based on those already calculated computations we will select the correct output through a mux. While this would make it faster, it would require a lot more processing power since it is calculating many more possibilities compared to the 4x4 hierarchy and be more complex since we would need more multiplexers to select the correct output for all 8 of those modules. We would have to run simulations on a few possible configurations

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 SystemVerilog circuit.

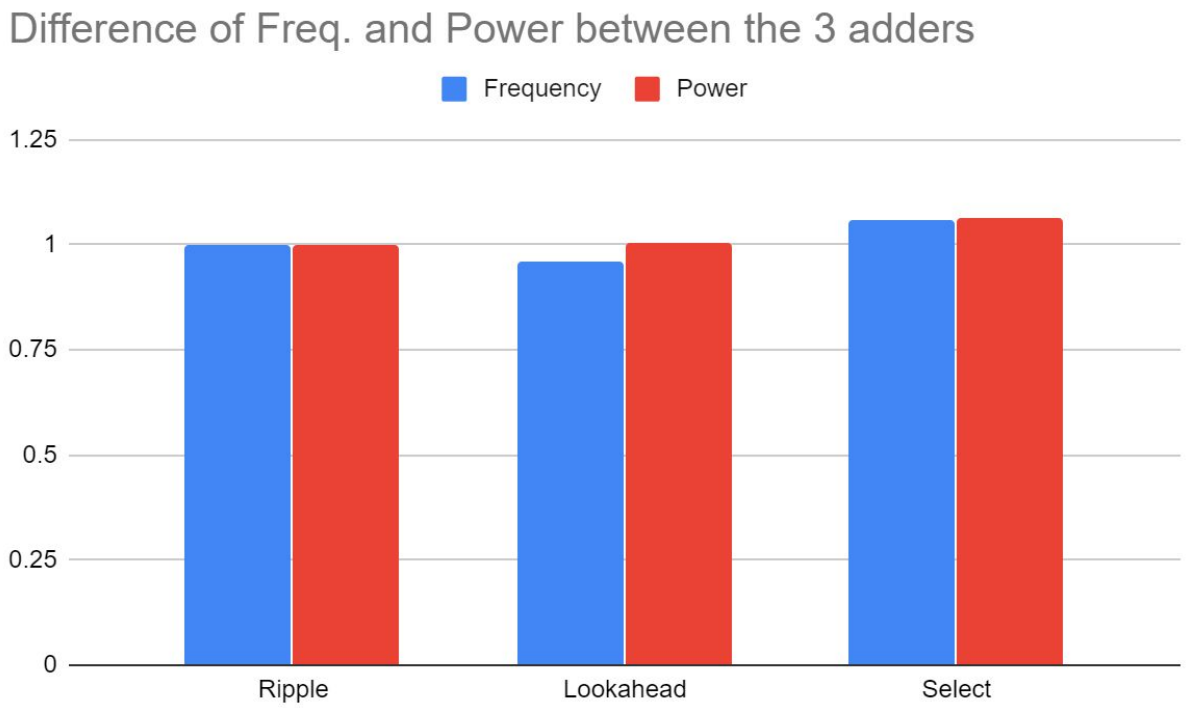
	Ripple	Lookahead	Select
LUT	122	116	133
DSP	None	None	None
Memory (BRAM)	0	0	0
Flip-Flop	105	105	105
Frequency	106.45 Mhz	103.2Mhz	112.76 Mhz
Static Power	42.87mW	42.86mW	42.87mW
Dynamic Power	1.67mW	1.58mW	2.00mW
Total Power*	86.89mW	87.18mW	92.65mW

*Total Power= Total Thermal Power Dissipation tab (including I/O Thermal Power Dissipation)

Normalized table:

	Ripple	Lookahead	Select
Memory*	N/A	N/A	N/A
Frequency	1	0.96	1.059
Power	1	1.003	1.0662

*Cannot divide by zero to normalize memory (BRAM)



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 4.12 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?

The select adder and ripple adder make sense in terms of each other- while the select adder is faster than the ripple adder (112.76 Mhz compared to 106.45 Mhz), it has higher complexity than the ripple adder (133 LUTs instead of 122) which makes sense to why it uses more power (92.65mW vs 86.69mW). We would similarly expect the lookahead adder to be faster than the ripple adder, with a tradeoff of more complexity and higher power dissipation. However, we saw

something unusual- a lower frequency than the ripple adder (103.2 Mhz) and essentially not much more power consumption (87.18mW compared to 86.89). It was clear after seeing these numbers that even though our carry-lookahead adder was producing the correct answers, we were running into the same limitation of the carry ripple adder in the fact that we were connecting the Couts from one module to the Cin to the other module. It makes sense that the frequency was even lower than the ripple because not only did we have to wait for the carry bits to propagate, we also needed to wait for the logic for the propagation and generate bits. Had we implemented this correctly with the Cins generated from the group propagates and group generates, we would have seen higher operating frequency since we wouldn't have to wait for the carry-outs to propagate to the carry-ins, being able to perform more operations per second directly leading to more power dissipation.

4.5 Conclusion

In conclusion, this lab allowed us to understand basic syntax and constructs of System-Verilog. We also looked at 3 different adders. How they are designed, how we can design them, key differences and behaviors of these adders. Overall this lab was a great introduction to the FPGA board and System-Verilog.