ECE 385

SP 2022

EXPERIMENT #3

Adders

Tianyu (Joe) Yu, Yulun(Ben) Wu

2/21/2022

Curtis Yu

I. Introduction

A. All three adders have the same inputs (10 bits A and B from switches and extended to 16 bits inside the register) and sum output (16 bits S), which expresses the sum of A and B.

II. Adders

A. Ripple Carry Adder

1. A 16 bit Ripple carry adder consist of 16 1-adders connecting in series, with the least significant bit's carry-out as the input for next bit's carry-in

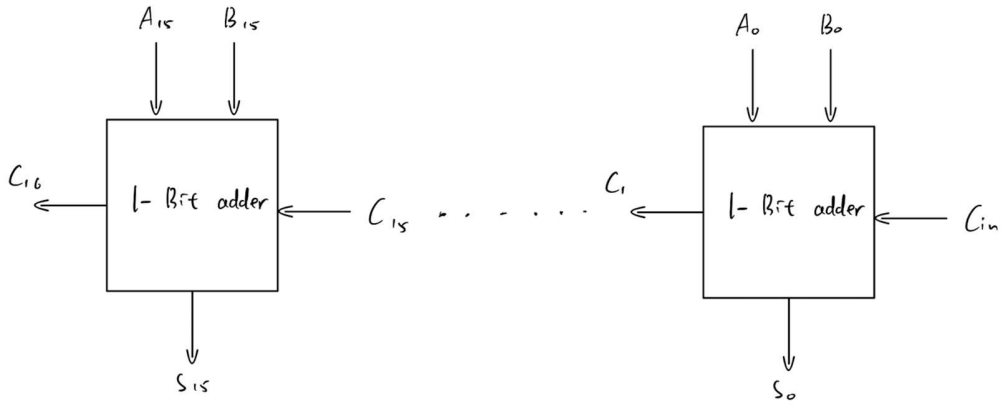

Figure 1. Block diagram for Ripple Carry Adder

B. Carry Lookahead Adder

1. Written description of the architecture of the adder

   The 16-bit Carry Lookahead Adder consists of 16 1-bit adders that get their carry-in bit from a Carry-lookahead module, which can calculate the carry-in bit faster.

2. Describe how the P and G logic are used.

   G stands for Generating, and it's calculated by A AND B, which indicates the carry out bit must be 1 once A and B are both 1. P, standing for propagating, is the result of A XOR B, and P stands for a possibility of being propagated that the carry out bit must be 1 if P is 1 and incoming carry in bit is 1. Thus we can have a logic expression for the first carry out bit, and make it as the carry in of the next digit's unit, we will deduct the next level's formula for the carry out bit.

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$
$$\ldots$$

Figure 2. The Algorithm for Calculating Each Carry Out Bit with P and G

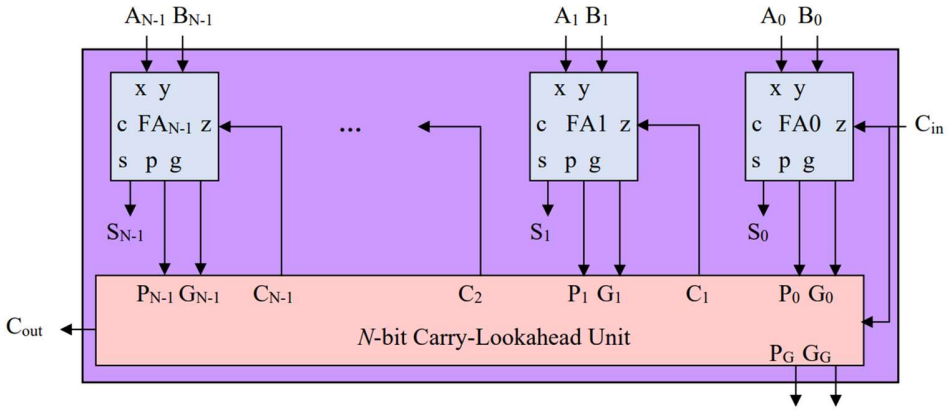

Figure 3. Block Diagram for Carry-Lookahead Adder

It's noticeable that neither P nor G depends on the carry in value, and we can build a series of combinational gates in the Carry-Lookahead Unit (shown in Figure 3), such that with known input A and B and the initial carry in bit C_{in}, we can calculate the carry out bit for each digit in parallel, and get the output S for each digit at once.

3. Describe how you created the hierarchical 4x4 adder. Block Diagram for a single CLA (4-bits), and Block Diagram for 4x4-bit hierarchical carry-lookahead adder

A 4x4 adder is to treat each 4-bit unit as a single digit, and connect them similarly as the way a basic 4-bit carry-lookahead adder does. First we need to define the new P_G and G_G to express the P and G correctly in the unit of 4 digits.

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

Figure 4. The Algorithm for Calculating P_G and G_G for Each 4-bits Unit

This can be interpreted as the net G, G_G, can only be 1, meaning there must be a carry out bit, once the G_0 and all P's are 1, or the second G_1 is 1 and the P's after it are 1, etc. P_G can only be 1, meaning there's a carry out bit once the carry in is 1, when all P's are 1. Now we have defined the P_G and G_G for each 4-bit unit, then we will have a general formula for the carry out bit of each unit:

$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$
...

Figure 5. The Algorithm for Calculating Each Carry Out Bit with P_G and G_G

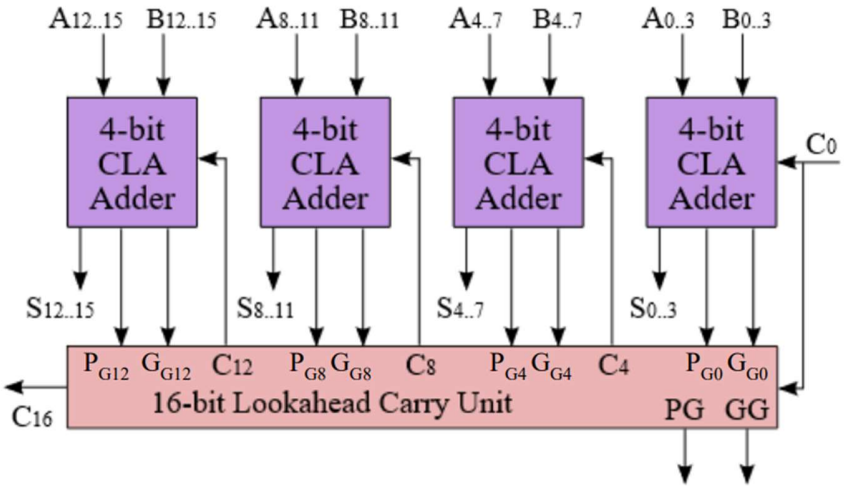

Figure 6. Block Diagram for 4x4-Bit Hierarchical Carry-Lookahead Adder

The output of each 4-bit lookahead unit is still the original output S, and we apply the formula for P_G and G_G above, and build a combinational logic circuit within the 16-bit lookahead carry unit, to calculate the P_G and G_G with each 4-bits group of input, and once we have the initial carry in bit, we can calculate each carry in bit and generate the output at the same time.

C. Carry Select Adder

1. Written description of the architecture of the adder

   The 16-bit carry select adder consists of 32 1-bit adders that are connected in multiple groups on two parallel lines, and each parallel line is a ripple carry adder. Multiple MUXs are also used to select the output and the carry-out bits.

2. High level description and block diagram.

   The carry select adder is fast because if we divide the 16-bit into 4 groups of 4 bits, we can parallel compute the result for both 0 and 1 carry-in for each 4 bit adder. When the previous 4-bit adder has complete its calculation, instead of going through the 4-bit ripple carry adder, it only needs to go through 5 MUXs (parallely, which means that it takes the same time as going through 1 MUX) to select the correct output depending on the carry-in bit. A MUX's gate delay is smaller than a 4-bit adder, therefore, the whole system will be faster.

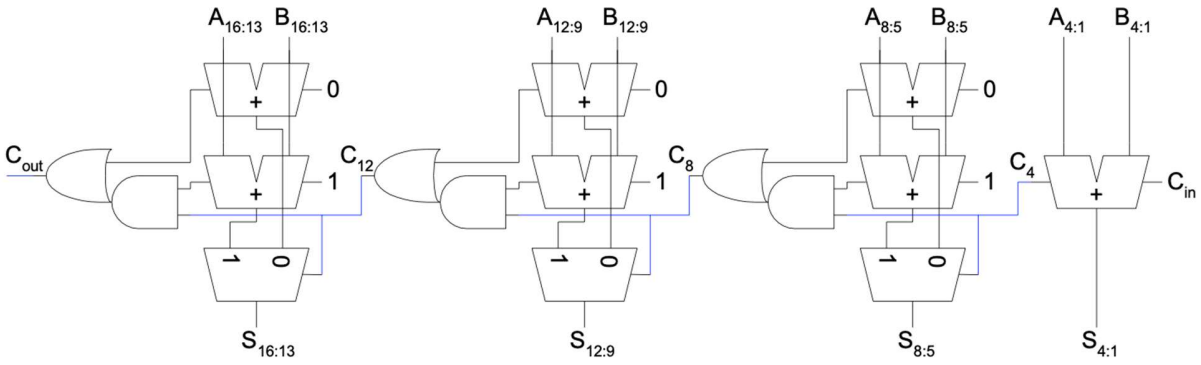

Figure 7. Block diagram for Carry select Adder

D. Written description of all .SV modules, see the Appendix I of the previous lab for the specific formatting

1. Module: ripple_adder

a) Inputs: [15:0] A, [15:0] B, cin,

b) Outputs: [15:0] S, cout,

c) Description: This module does not depend on clock and S will produce the sum of A and B via ripple adder method

d) Purpose: This module is used to perform 16-bit add calculation.

2. **Module: lookahead_adder**
    a) Inputs: [15:0] A, [15:0]B, cin,
    b) Outputs: [15:0] S, cout,
    c) Description: This module does not depend on clock and S will produce the sum of A and B via look ahead method.
    d) Purpose: This module is used to perform 16-bit add calculation.
3. **Module: select_adder**
    a) Inputs: [15:0] A, [15:0]B, cin,
    b) Outputs: [15:0] S, cout,
    c) Description: This module does not depend on clock and S will produce the sum of A and B via carry-select method.
    d) Purpose: This module is used to perform 16-bit add calculation.
4. **Module: router**
    a) Inputs: R, [15:0] A_In, [16:0]B_In,
    b) Outputs: [16:0] Q_Out;
    c) Description: This module does not depend on clock and Q_Out will produce the result in A or B depending on the value of R.
    d) Purpose: This module is a 17 bit parallel multiplexer.
5. **Module: reg_17**
    a) Inputs: [16:0] Din, Clk, Load, Reset
    b) Outputs: [16:0] Data_Out
    c) Description: This is a positive-edge triggered 17-bit register with asynchronous reset and synchronous load. When Load is high, data is loaded from Din into the register on the positive edge of Clk.
    d) Purpose: This module is used to create the registers that store operands A and B in the adder circuit.
6. **Module: HexDriver**
    a) Inputs: [3:0] In0
    b) Outputs: [6:0] Out0
    c) Description: It takes four bits input from In0 and converts to 7 bits for Hex-display.

d) Purpose: This module converts 4bit binary numbers from registers into hexadecimal so we can display using LEDs.

7. **Module: control**
   a) Inputs: Clk, Reset, Run,
   b) Outputs: Run_O
   c) Description: This is a state machine with asynchronous reset. Only in State B (when input Run is High), the output Run_O will be High.
   d) Purpose: This module serves the purpose of a state machine that gives an execution signal when we press a button and prevents it from doing multiple executions if we don't release the button.
8. **Module: ADDER4**
   a) Inputs: [3:0] A_4, [3:0] B_4, c_in_4
   b) Outputs: [3:0] S_4, c_out_4
   c) Description: This module does not depend on clock and S will produce the sum of A and B.
   d) Purpose: This module is used to perform 4-bit add calculation.
9. **Module: CLA_4**
   a) Inputs: [3:0] A_4, [3:0] B_4,, c_in_4
   b) Outputs: [3:0] s_4, G_out, P_out
   c) Description: This module does not depend on clock and S will produce the sum of A and B via look ahead method.
   d) Purpose: This module is a 4-bit CLA that is then used to build larger CLAs.
10. **Module: CS**
    a) Inputs: [3:0] A_4, [3:0] B_4,, c_in_4
    b) Outputs: [3:0] s_4,
    c) Description: This module does not depend on clock and S will produce the sum of A and B via carry select method.
    d) Purpose: This module is a 4-bit CS adder that is then used to build larger CS adders.
11. **Module: adder2**
    a) Inputs: Clk, Reset_Clear, Run_Accumulate, input [9:0] SW,

b) Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5,
c) Description: This is the top level module and will call on control, router, reg_17, adder, and HexDriver modules
d) Purpose: This is the top level module that takes in inputs from switches and buttons and displays the outputs on LEDs.

E. Describe at a high level the area, complexity, and performance tradeoffs between the adders.

1. The ripple carry-adder

   It takes up the least amount of space, as a trade off, its computational capacity grows linearly with time (Time here refers to the time given to complete an add computation). And obviously, it is the simplest among the three.

2. The carry cook-ahead adder

   It takes more space then ripple carry-adder, and is more complex, since more logic is involved to calculate the carry-in bit. However, the computational capacity grows exponentially with time, which greatly increases the computation power when we get to large bits operations.

3. The carry select adder

   It takes up more than double the space of a ripple carry adder, and with the involvement of MUXs, it is a little bit more complex. However, its computational capacity grows linearly, but with a larger slope, with time. (This is with the assumption that we stick with the AxN design, where A is constant, and N is the number we pick depending on the size of the adder. We can modify the adder so it can have even better computational capacity and takes up a little less space. This will be discussed in the Post-Lab section).

F. Document the performance of each adder by creating a graph as specified in Prelab part C (page 4.6 in the manual)

	Carry-ripple	Carry-select	Carry lookahead
Memory (BRAM)	0	0	0
Frequency	237.25	241.49	250
Total Power	98.86	98.86	98.86

Figure 5, table of performance of three adders

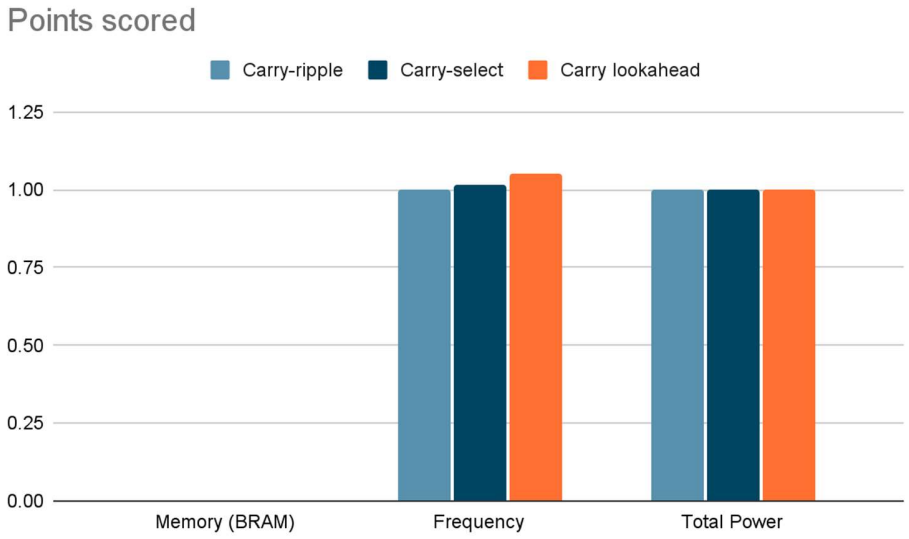

Figure 8. Graph of performance of three adders

G. Annotated simulation trace.

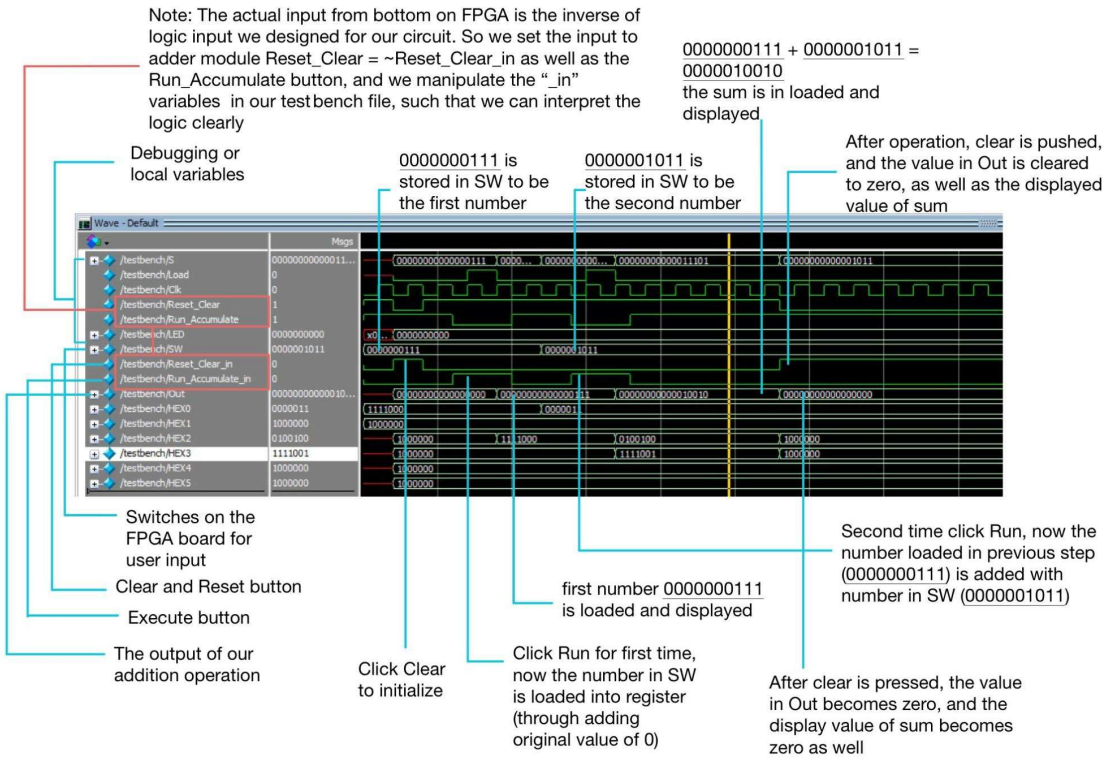

Figure 9. Annotated simulation trace.

III. Answers to the post-lab questions

A. 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?)

1. Is 4x4 hierarchy CSA ideal? If not, how would you go about designing the ideal hierarchy on the FPGA。

No, we can build it in such a way that when the "choose" signal has arrived at the MUX, the two possible results have been just added. For example, if we are designing a 64-bit adder, our original design puts adders into groups of 4, 16 x 4 adder, which has a total delay of 4-ripple adder plus 15 MUXs' gate delays. Now, if we assume the gate delay of an MUX is the same as one 1-bit adder, we can divide 64 bits into

groups of 4,4,5,6,7,8,9,10,11. In this way, the delay is reduced to 4-ripple adder plus 8 MUXs' gate delay, and we also save 7 MUXs' space. This feature will become more obvious as we increase the number of bits.

B. 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?

	Ripple Adder	Look-ahead Adder	Select Adder
LUT	78	90	82
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	20	20	20
Frequency (Mhz)	237.25	241.49	250.00
Static Power (mW)	0.00	0.00	0.00
Dynamic Power (mW)	89.84	89.84	89.84
I/O Power (mW)	8.92	8.92	8.92
Total Power (mW)	98.86	98.86	98.86

Figure 10. Design Statistics Table

Yes, the breakdown comparison makes sense. Actually the generated data in the table above reflects the expectation of our design, that the frequency of look-ahead adder is higher than the frequency of basic ripple adder, and the select adder, due to the parallel design, is even higher in frequency. All three designs share the same level of power consumption, which is not expected. We expect the look-ahead adder to have highest power consumption due to its greater number of combinational logic gates insides, which also reflects on LUT value. It's possible that the logic gates are not that power consuming, and our project is still on a relatively smaller scale, which can't make great contrast in power consumption.

IV. Conclusion

A. Bugs and Countermeasures

1. At the beginning of implementing the look-ahead adder, we mistakenly treated the P_G and G_G as the last P and G of the 4-bit look-ahead unit without extra calculations. So the result was not correct for a while until we noticed this point. Besides, in the MUX part of carry select adder, we used the “if” condition at the beginning, but the compilation error appeared a lot, and we decided to use the “unique case” method to solve the problem.

B. The block diagram in the pre-lab section was very helpful for our understanding on the carry-select and look-ahead adder.