**ECE 385**

Fall 2021

Experiment #3

**16-bit adders**

Steven Dimov & Owen Shin

Section: ABE

TA: Abigail Wezelis

1. Introduction a. Summarize the high-level function performed by the three adders.

The main function of this lab was to create a 16-bit adder capable of storing a 16-bit number and extra carry out bit, that could be updated with the sum of that number and another 16-bit number inputted by the user. When our machine was prompted to perform an addition, it would update the register with the sum of the original 16 bit number and the inputted number, which was calculated through different types of combinational logic.

2. Adders

Full adder:

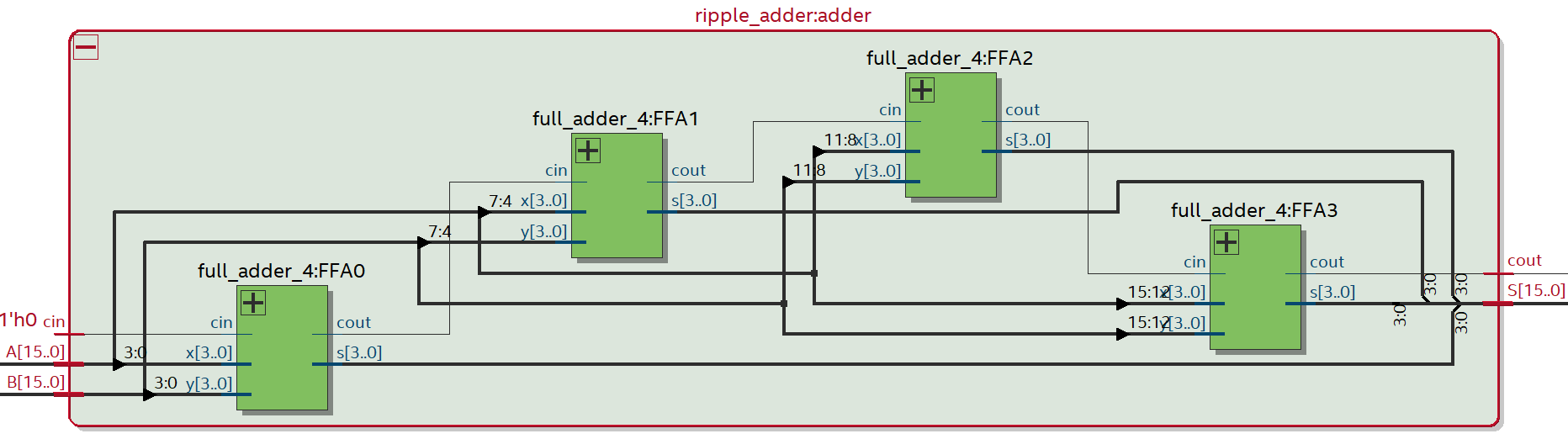
A full adder was necessary to create each kind of adder included in this lab. It served as a unit that could add two bits together with a carry-in input and produce a carry-out output.

a. Ripple Carry Adder

i. Written description of the architecture of the adder

The ripple carry adder is the simplest of these three designs, but also the slowest. The way it is arranged is by including several full adders in series with each other, so that the carry-in input of one full adder is wired with the carry-out output of the full adder of the next most significant bit (to the left of it). The reason why this design is slow is because each individual full adder within the ripple adder is dependent on the previous full adder’s carry-out bit. Since they are in series, each one has to wait on the one before it, not computing its own correct sum output and carry-out output until its carry input is finished being produced by the previous full adder. This arrangement’s delay will add up for each full adder included.

ii. Block diagram.



b. Carry Lookahead Adder

i. Written description of the architecture of the adder

The carry lookahead adder serves to improve upon the shortcomings of the ripple adder’s speed by having a separate module to oversee the carry-outputs of each full adder, so that they could be computed independently of each full adder using the two new outputs, P and G which are dependent only on the two initial inputs being added. Having this allows each full adder to be utilized at the same time, as opposed to waiting on a sequential line of carry-out outputs.

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

The P and G logic refer to the “propagate” and “generate” descriptions of what value the carry-out bit will be. When A and B are fed into the entire adder, the carry look ahead generator can check which places there will be a 1 carry out bit for certain, which is where bitwise A & B = 1. In this instance, a 1/high carry-out bit is generated regardless of its adjacent full-adders’ carry-out bit. In other cases, the carry-out bit may not necessarily take on a 1/high value. This design is advantageous as after only one gate delay the P and G values are produced and used for computing the carry-in values for the corresponding full-adder. In the carry look ahead generator, the carry-in bits for each adder are computed using the following expressions:

c0 = cin;

c1 = cin & p0|g0

c2 = (cin & p0|g0) & p1 | g1

c3 = ((cin & p0|g0) & p1|g1) & p2|g2

PG (propagate group) and GG (generate group) are additional outputs given to the carry look ahead adder. PG calculates whether or not the initial input Cin will propagate through the entire 4-bit sequence of full adders, whereas GG calculates whether or not a high carry-out will be generated somewhere along the sequence that will set Cout. Their expressions are as follows:

cout = (((cin & p0|g0) & p1|g1) & p2|g2) & p3|g3

PG = p0 & p1 & p2 & p3

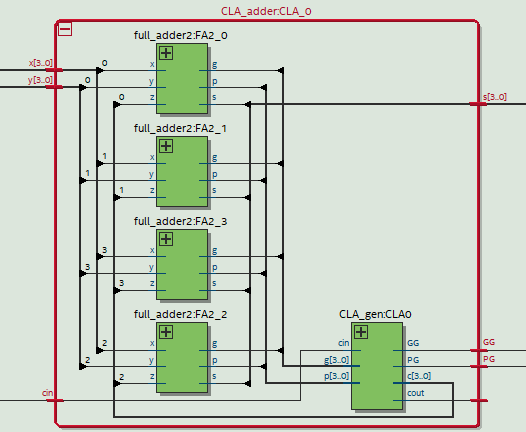
GG = g3|g2 & p3|g1 & p2 & p3|g0 & p1 & p2 & p3

iii. Describe how you created the hierarchical 4x4 adder.

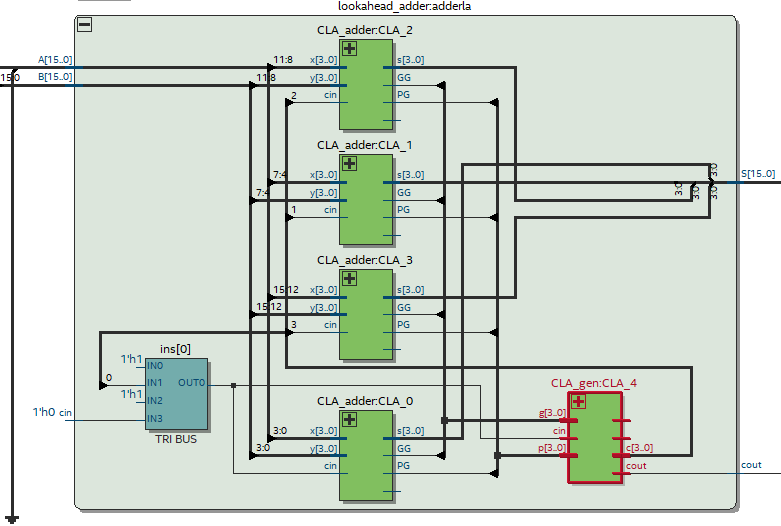
The 4x4 hierarchical adder is constructed by implementing five of the CLA (carry look ahead) generators. Four of them are controlling/managing the carry-ins and carry-outs of each 4-bit segment of the 16-bit inputs as described above. The fifth one is then used to manage/control each of those four CLA generators just mentioned. As opposed to implementing their carry-outs to the next CLA’s carry-in, this fifth module intakes these inputs and outputs from the four CLA generators and provides the appropriate carry-ins and carry-outs for all of them.

iv. Block diagram

1. Block diagram inside a single CLA (4-bits)



2. Block diagram of how each CLA was chained together.



c. Carry Select Adder 4.12

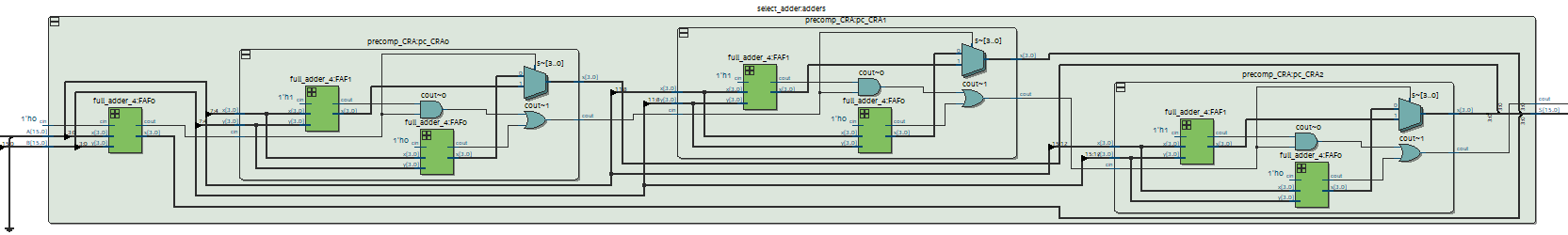
i. Written description of the architecture of the adder

The carry-select adder is another way to improve upon the ripple-adder’s slowness. It implements the idea of precomputing outputs and then selecting between them once the correct carry-in input is given. This requires an additional set of 16 full adders to compute both possible arrangements of the “unknown” Cin value and a series of multiplexers to select each correct sum value.

ii. Describe at a high level how the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later. Make sure you understand this!

The CSA (carry select adder) computes these sums in parallel by taking in the initial bitwise inputs A and B and then computing one possibility with Cin set high and another one with Cin set low. After that, using the actual (correct) Cin value the corresponding computation is selected using a mux. This idea is then repeated for each bitwise operation; in this case, 4 bits are grouped together in one CSA module. Taking four of these 4-bit CSAs, they can be rippled through each other, using muxes to select the carry-out of each CSA to construct a 16-bit adder.

iii. Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic.



Module: router.sv

Inputs: R, [15:0] A\_In, [16:0] B\_In

Outputs: [16:0] Q\_Out

**Post Lab Questions:**

The 4x4 hierarchy described in the designs of these adders are not ideal since they group together 4-bits instead of dealing with the bits at an individual level, excluding the case of the ripple adder. The carry-select adder then has to ripple in carry-outs, generating some gate delay, waiting on the previous adder group, where it could have rather simply just dealt with a smaller scale group, or at the lowest level.

**Conclusion:**

This week’s lab was quite successful as not many bugs were present during the process of coding our designs. The only issues we faced were regarding syntax and connecting the Cout with the LEDs of the FPGA for detecting overflow.