**ECE 385**

Spring 2023

Experiment #3

**16-Bit Adders**

Noah DuVal, Jason Wright

JZ / Friday 3:00 pm

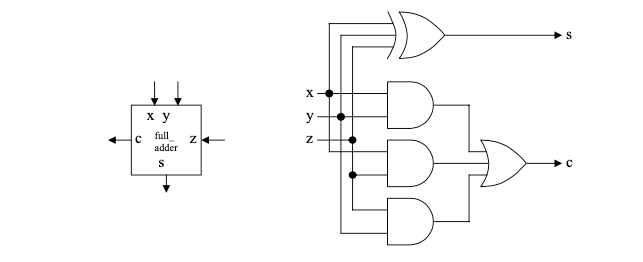
Jason Zhu

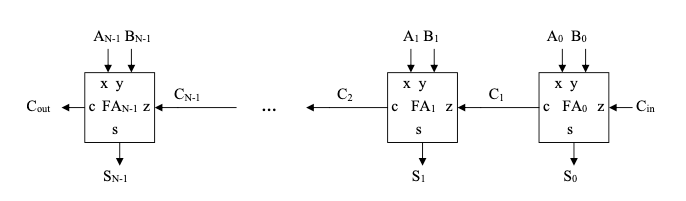
**Introduction**

When it comes to creating a computational adder in the language of computers there are multiple ways to implement it. Depending on the implementation of the adder there can be different advantages or disadvantages to the circuit. For example, a few of the metrics chosen to be looked at in this lab were the timing, memory, and power (dynamic, static, and total) of the circuit. The three adders to which were studied in this experiment were the Carry-Ripple, Carry-Select, and the Carry-lookahead. All of which are able to perform the addition of two 16-bit numbers by using a 4x4 hierarchical design.

**Ripple Carry Adder**

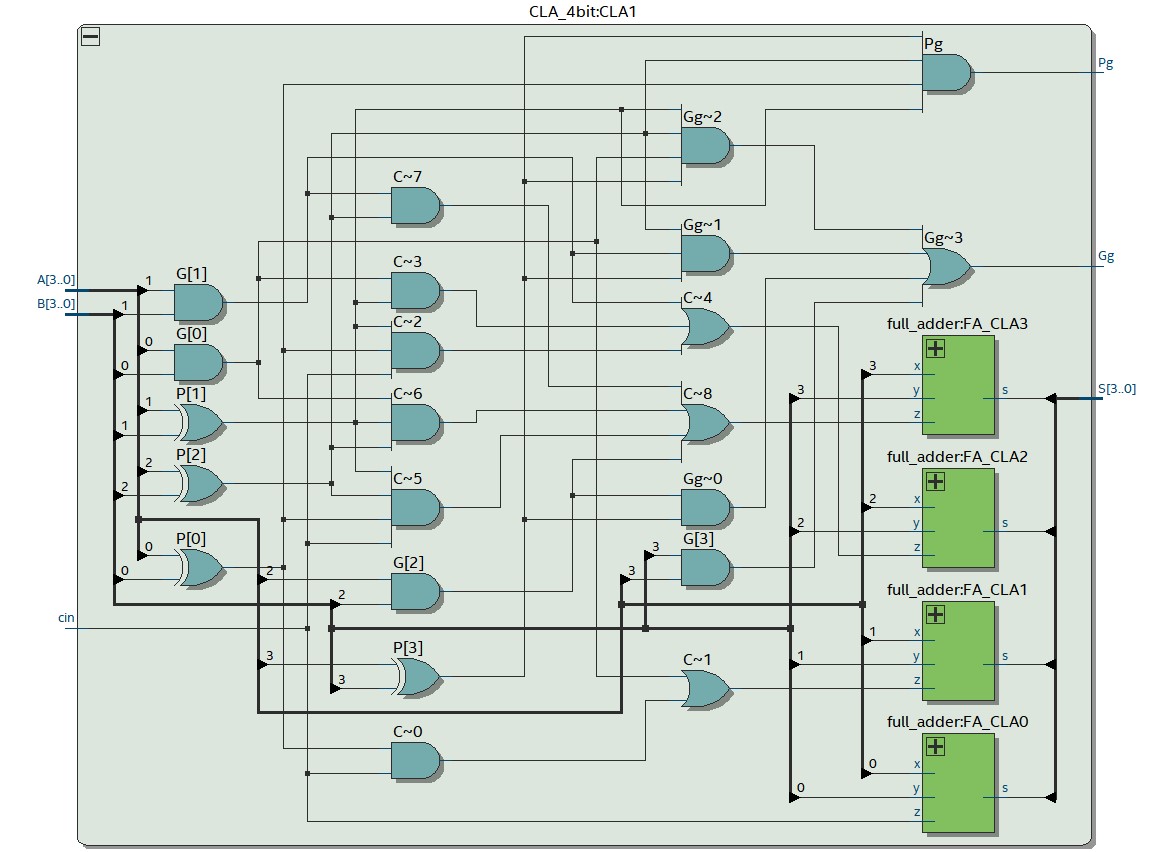
Being the simplest design of all of the adders the ripple carry adder merely consists of 16 consecutive full adders or N-bit adders depending on the desired number of bits to be added. Each of the adders are connected to one another through the carry bits to which creates the ‘ripple’ effect. Each full adder takes in two bits X and Y to be joined by a carry in bit Z as seen in Figure 1 below. The first adder receives Cin from a sequence of logic and will output a one bit sum labeled ‘S’ and the carry out bit C1 to which becomes the next carry in bit for the next adder.

**Figure 1: Full-Adder Block Diagram**

**Figure 2: N-Bit Carry-Ripple Adder Block Diagram**

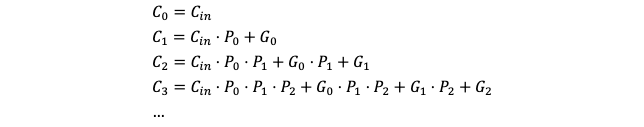
**Carry Lookahead Adder**

With processing time in mind the goal of the carry-lookahead adder is to minimize the ripple through parallelization of the system. In the block diagram below you can see the incorporation of the parallelization by using the new generating (G) and propagating (P) outputs. It is then through combinational logic that the full adders ahead of the first bit are able to predict the carry in bits and have the output instantly calculated instead of waiting for the ripple. The equation G(A,B) = A \* B is to determine if a carry in bit is needed as if and only if both A and B are 1 is a carry in bit needed. The propagation equation P(A,B) = A XOR B is used to determine whether or not the generated bit is to be carried to the next adder as it is only propagated if either A or B is 1.

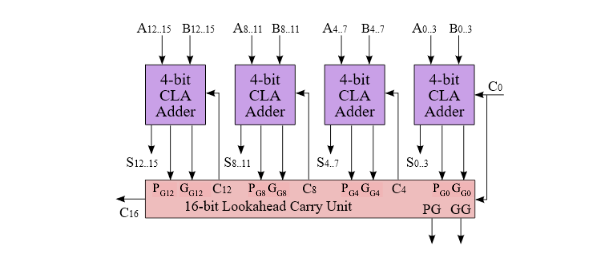


**Figure 3: 4-Bit CLA Block Diagram**

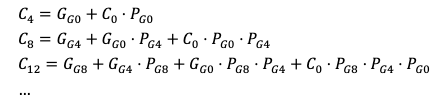
While the two equations of G(A,B) and P(A,B) are the deciding factors for the carry in bits the logic behind the carry in bits is a bit more complicated and gets exponentially more complex the larger the adder. As seen in figure 4 after only three carry in bits the equation gets quite expensive reducing the effectiveness of the design the larger it gets.

**Figure 4: Serial Carry-Lookahead Adder Ci Equations**

Thus, in order to account for the fact that the equation for the higher bits such as C15 would be absurdly large the design is turned into a 4x4 hierarchical design instead. In order to do that four 4-bit CLA’s are combined to create the entire 16-bit CLA as seen in figure 5. However, due to the hierarchical design there are only three carry bits needed. Thus, only every fourth carry-in bit is to be calculated (C4, C8, C12). Therefore, the previous outputs ‘G’ and ‘P’ are now grouped together creating the group generate (GG) and the group propagate (PG) with their equations seen in figure 6 and the carry bits in figure 7.

**Figure 5: 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram**

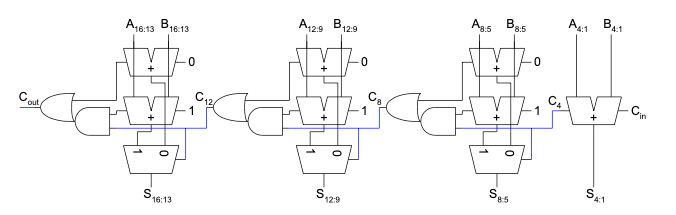
**Figure 6: Group Equations**

**Figure 7: Hierarchical Carry-Bit Equations**

While one way to extend this design is to continue adding more and more layers to the design, increasing the number bits it can calculate by a factor of four every time. For example, a 4x4x4 design would be a 64-bit adder with no additional time added to the ripple of the system. However, as with everything these advantages over the ripple adder will also come with matched disadvantages as this design requires more gates and components making it a costlier design than the ripple adder. For that reason we will next look at the carry select adder that holds different advantages and disadvantages over the ripple adder.

**Carry Select Adder**

Another approach to reduce the delay from the ripple effect is to instead of waiting for the carry-in to compute the sum you compute the sum twice. Once as if the carry in bit is a 1 and again as if the carry in bit is a 0. Then use muxes to decide which result to use once the actual carry-in bits arrive. However, while this design is quicker than the previous two it also is costly as it uses almost twice the full adders and additionally requires more power. Although it does have a smaller area than the CLA once again offering different specs based on what your system is in need of.

**Figure 8: 16-Bit Carry-Select Adder Block Diagram**

To implement this design as a 16-bit CSA as represented in figure 8 there are four parts, three of which are identical. The first one being the four least significant bits. This portion of the circuit consists of only one 4-bit carry ripple adder as the carry-in bits are already known from Cin. The following parts of the circuit each identically consist of two 4-bit CRA’s one calculating the input with a 0 as the carry in and the other with 1. Upon receiving the carry-in, the already computed results are selected using a 2:1 mux. Lastly, in order to string together the 4-bit sections there is “glue” logic produced by a single AND and OR gate to determine the carry-in for the next 4-bit section.

**Comparison and Performance**

All 3 adders utilized zero memory, and the frequency was not analyzed because we did not opt to complete the extra-credit timing analyzer required to generate it.

The power dissipated by the system was found to be 98.78 mW for all three adders, as the components that make up the adders (the only difference in the 3 different runs) account for almost none of the power dissipation. For this reason, the suggested graph was omitted.

**Module Descriptions**

Module: adder\_toplevel

Inputs: Clk, Reset\_Clear, Run\_Accumulate, [9:0] SW

Outputs: [9:0] LED; [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5; [16:0] Out

Description: This is the top level logic that calls the control, register, adder, and hex driver modules. Additionally, it assigns the input buttons and switches to their respective variables for use in other modules, and defines the system outputs.

Purpose: This module is the interface between all the other modules as the top level and is needed to define the connections between most of them.

Module: reg\_17

Inputs: Clk, Reset, Load; [16:0] D

Outputs: [16:0] Data\_Out

Description: This is a 17 bit register that has two options: reset and load. On reset, the data is set to zeroes, and on load (on a positive clock edge, because it is a synchronous module) it stores the value of D.

Purpose: This is needed to hold the data of the two adder input registers A and B as well as the output Out.

Module: mux2\_1\_7

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

Outputs: [16:0] Q\_Out

Description: This is a 17 bit multiplexer. It uses the selection bit S to send one of two inputs A or B to the output.

Purpose: This mux selects between either register B or the sum of registers A and B (stored in A) and sends it to the output register. This is needed to either hold Out’s value or store the a new value into it.

Module: HexDriver

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This is a selector that uses the 4 bit input to choose one of 16 different cases which correspond to various hex displays.

Purpose: This module is needed to display the various registers on the hex displays for readability.

Module: control

Inputs: Clk, Reset, Run

Outputs: Run\_0

Description: This is where the state machine for the system is defined. It uses the system inputs Reset and Run to navigate between a rest state, an add state, and a hold state. Run\_0 is set to high only in the add state.

Purpose: This module tells the rest of the system when to run the adders and load the values. It also prevents multiple run-throughs from occurring on a single button press.

Module: full\_adder

Inputs: x, y, z

Outputs: s, c

Description: This module is a single full adder, It produces a sum and carry bit in response to the 3 input bits.

Purpose: This is needed to implement all 3 of the adders designed by this lab. Every other adder module is just different combinations of full adders.

Module: ripple\_adder

Inputs: [15:0] A, B; cin

Outputs: [15:0] S; cout

Description: This module sequentially calls the full\_adder module once for all 16 bits, using the previous carry out bit as the carry in for the next adder. It stores the final result in the 16 bit sum variable.

Purpose: This module implements addition in the simplest way possible.

Module: ripple\_4

Inputs: [3:0] A, B; cin

Outputs: [3:0] S; cout

Description: This is simply a 4 bit version of the ripple\_adder function.

Purpose: This module is needed to implement the select\_adder module, which uses multiple smaller ripple adders working in parallel.

Module: mux2\_4

Inputs: S; [3:0] A, B

Outputs: Q\_out

Description: This module is a 2 bit multiplexer which uses the select bit S to decide which 4 bit input A or B makes it to the output Q\_out.

Purpose: This module is needed in the carry-select adder to choose between the two possible sums that are generated by the select\_adder module.

Module: select\_adder

Inputs: [15:0] A, B; cin

Outputs: [15:0] S; cout

Description: This module calls the ripple\_4 module 7 times. The first call is for the first 4 bits of the inputs. The remaining adders are called in pairs, with each set of 4 bits called twice, once with carry-in = 0 and once with carry-in = 1. Finally, 3 mux2\_4 modules are defined two choose which ripple\_4 outputs are carried into the sum variable.

Purpose: This module is made to accomplish the addition task faster by allowing 4 bits to be added sequentially 4 times in parallel rather than all 16 bits to be completed sequentially. A small amount of time is added at the end for the carry out bits to be sent to the muxes.

Module: CLA\_4bit

Inputs: [3:0] A, B; cin

Outputs: [3:0] S; Gg, Pg

Description: This module calls the full\_adder module 4 times using the A and B inputs. The carry-in bit for each module is calculated using combinational logic in order to prevent it from being slowed by the rippling of the ripple adder. Once complete, it also calculates the Gg and Pg bits which are needed for the full carry lookahead module to function.

Purpose: This implements addition without requiring any rippling delays.

Module: lookahead\_adder

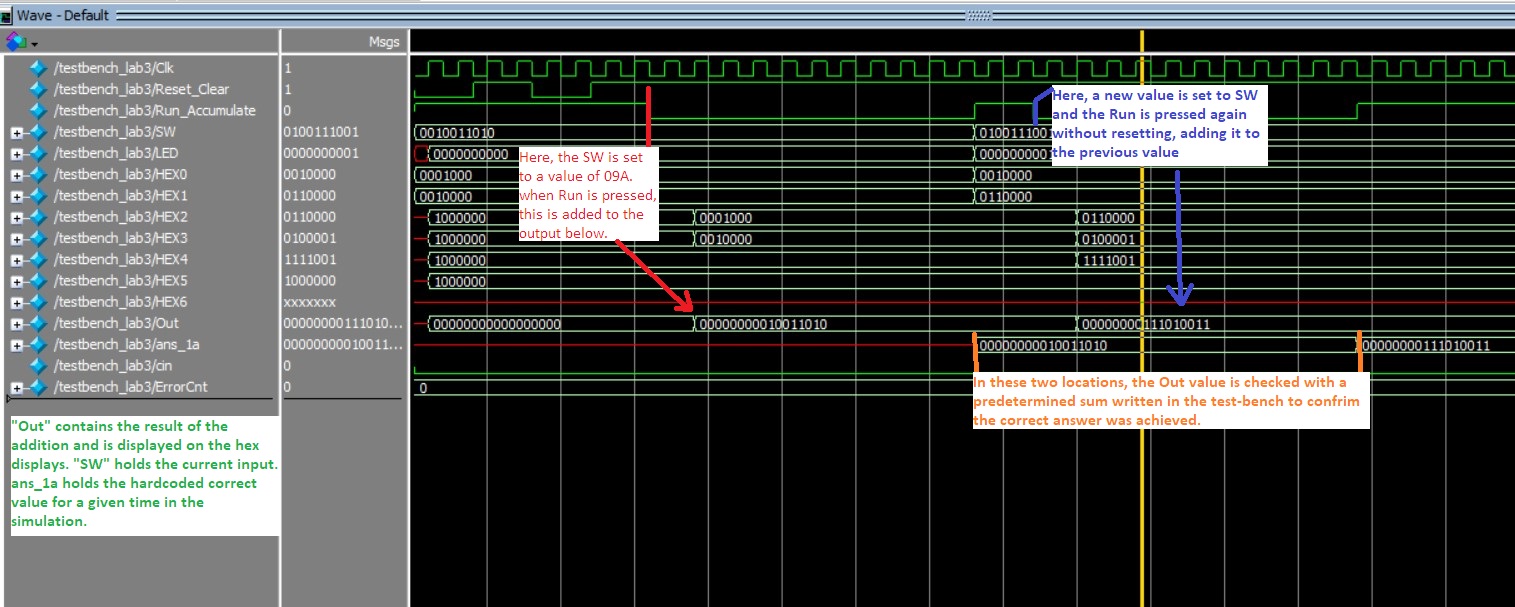
Inputs: [15:0] A, B; cin

Outputs: [15:0] S; cout

Description: This module is the full 16-bit carry lookahead adder. It defines 4 of the CLA\_4bit modules, with their carry-ins being computed combinationally using the outputs (Gg and Pg) of the previous 4 bit adders.

Purpose: This implements addition by allowing each 4 bit segment to be added in parallel. It does still require rippling between these segments, but to a much lesser extent than the ripple adder.

**Simulation Results**



**Figure 9: Annotated Simulation**

The above simulation (Figure 9) showcases the results of a simple testbench performed on the select adder. Two random numbers were chosen, those being (in decimal) 154 and 313. The simulation checks first that the first value was successfully added to the empty register. After the second input is added, it then checks the sum with a hardcoded correct answer to detect errors.

**Post Lab Questions**

4x4 is most likely not the ideal hierarchy. The ideal could be determined experimentally by using timing analysis. To complete this, one could try several different hierarchies to find which one completes fastest. This is because in all cases there will be some degree of waiting involved. For example, the select adder still requires the carry-in bit to move through the mux of each 4-bit adder once they are all done. In one case, the adders could complete early and be waiting on that bit, or the bit could reach the mux early and be waiting on the adders to finish. In the first case, you could afford to use 5-bit or 6-bit adders to reduce wait time, and in the second case you may want to use only 3-bit adders. Each adder type may have a different ideal hierarchy as well, so this experiment would need to be conducted for each one separately.

|  | Ripple | Lookahead | Carry Select |
| --- | --- | --- | --- |
| LUT | 78 | 91 | 83 |
| DSP | 0 | 0 | 0 |
| Memory | 0 | 0 | 0 |
| Flip-Flop | 20 | 20 | 20 |
| Static Power | 89.94 mW | 89.94 mW | 89.94 mW |
| Dynamic Power | 0 | 0 | 0 |
| Total Power | 98.78 mW | 98.78 mW | 98.78 mW |

Note: Frequency was omitted from this table because the extra-credit timing analysis was not completed to generate this.

The adders were not a significant enough source of power dissipation to cause changes in the analyzer, so the power values remained unchanged for all. The discrepancy between static power and total power is caused by the “I/O Power” which amounted to 8.84 mW for all cases.

The number of flip-flops stayed at 20 because the adders did not require the use of any extra registers. No DSP blocks or memory bits needed to be used.

The number of look-up tables is proportional to the number of logic operations needed to complete the goals. Ripple required the least which is intuitive, as the other 2 adders were essentially additions to a basic ripple adder (meaning they still required at least the same number of full adder elements). Lookahead required the most because of the large number of logic gates used to implement the P and G bits. Carry Select required extra LUTs for the muxes and extra adders, but not as many as Lookahead.

**Conclusion**

Over the course of this experiment while some bugs were larger than others, most of them mainly consisted of simple and small errors that had large effects. For example, there were a few instances where the bit sizing for certain inputs and outputs were off by one and would throw the entire system off. Resulting in a lot of don’t cares and high impedance outputs on the model sim testbench. Additionally, there was an error with our state machine when it came to running the computation back to back as we later realized that the first add state had to be skipped during consecutive calculations. Outside of that it went smooth for the most part. As far as the lab manual is concerned there remains no complaints as the graphics and walk-through provided allows you thoroughly visualize how the system is supposed to work allowing you to think through what it is you need to accomplish.