**Lab 5: Adders**

EE 316: Digital Logic Design

## Overview

This lab is intended for you to design various datapath components in Verilog. By the end of this lab, you should be able to:

- describe and implement a multiply-accumulate circuit.
- design datapath components in Verilog and use them in other datapath components.
- design a load register that stores variable input in Verilog.

This lab will require previous knowledge of adders and load registers. Be sure to review them before starting the lab.

*NB:* Traffic on the Linux server may slow your connection and Vivado speed down. Please make sure to account for this in managing your time to complete the lab.

## Background

Ripple-carry and carry-lookahead adders are two fundamental types of adders. In this section, we will discuss how they are implemented.

### *Ripple-Carry Adders*

Ripple-carry adders (RCAs) are essentially a bunch of one-bit full adders in series. A one-bit full adder has three inputs: $a, b,$ and $c_i$. $a$ and $b$ are the two operands, and $c_i$ is the carry-in value. The outputs of the adder are the sum $S = a \oplus b \oplus c$ and the carry-out $c_o = ab + bc + ac$. If you cascade many full adders together as shown in Figure 1, you can get an $n$-bit adder that has $n + 1$ outputs (the $n$-bit sum and the final carry-out).

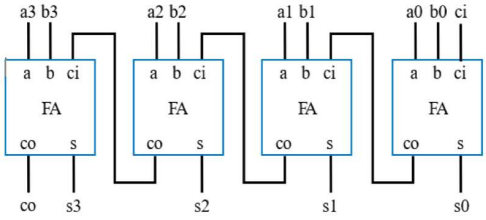


Figure 1: 4-bit Ripple-Carry Adder (RCA)

While ripple-carry adders are easier to design and use less gates, the time it takes to compute a sum using a ripple-carry adder is strictly a function of long it takes for the final carry-out to be calculated. If you want to calculate a 16-bit sum, you would have to wait until all 16 full adders in the 16-bit ripple-carry adder update before getting the correct carry-out, which may take an extremely long time.

The **critical path** of a circuit is the path in the circuit that takes the longest time to calculate. For the $n$-bit RCA, the critical path is the carry-out, as it depends on both inputs $a$ and $b$ and the carry-in - which depends on the previous full adder's carry-out, which depends on the previous

full adder's carry-in, and so on and so forth. The final carry-out thus depends on the initial carry-in, which must ripple (a.k.a. propagate) through the full adders as each one computes. Calculating the carry-out is the primary reason RCAs are not desired: it takes a long time to do.

***Carry-Lookahead Adders***

To solve the delay issues of RCAs, carry-lookahead adders (CLAs) "pre-calculate" all the carry-ins in advance based on the bits in the two operands. Two new values are used to calculate the sum and carry-out in the CLA: the "propagate" bit $P_i$ and the "generate" bit $G_i$:

$$P_i = a_i \oplus b_i$$

$$G_i = a_i b_i.$$

These values are calculated BEFORE the computation of the sum bits is completed. Once the propagate and generate bits are calculated, the sum and carry-out bits can be computed:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i.$$

Furthermore, we can expand $S_i, C_i$, and $C_{i+1}$ in terms of $P_i, G_i$, and $C_0$ by recursively substituting the equations for $C_i$. For example, for $S_2$ and $C_2$,

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$S_2 = P_2 \oplus (G_1 + P_1 G_0 + P_1 P_0 C_0)$$

*NB:* In this lab, we expect you to expand all sum and carry bits to the sum-of-products forms shown above. If you leave the equations in nested form (i.e. $G_1 + P_1(G_0 + P_0 C_0)$), you will get incorrect answers for checkout questions.

The block diagram in Figure 2 shows the dataflow of a CLA.

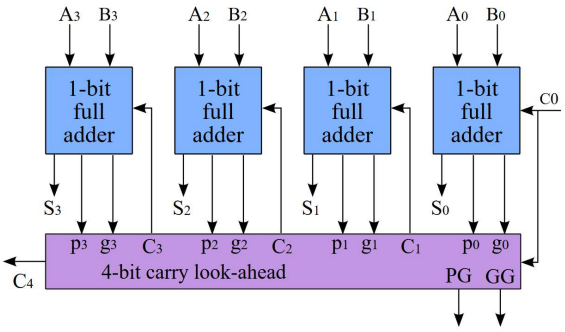


Figure 2. Carry-Lookahead Adder

As a result of the substitution property, even though the equations may become extremely long, it is theoretically possible to write an $n$-bit CLA using no more than four stages of gates, thus reducing the critical path of the adder. However, the trade-off is that the equations become long, which in hardware, represents more gates.

As computer systems become increasingly complex and require increasingly fast processing speeds, the trade-off between area and critical path delay becomes increasingly favorable to area. Why? Because current microprocessors have billions upon billions of transistors that can be used implement any desirable circuit - there is no lack of area for circuits to be placed, but there is an upper bound on how fast a circuit can calculate a value. A faster computer will want to shorten critical path delays as much as possible, and without the concern for area, a carry-lookahead adder is almost always the better option between the two adder types.

## Procedure

### *Part A: Ripple-Carry Adder (RCA)*

In this section, you will build a 4-bit ripple-carry adder **using dataflow modeling**. Your two operands will come on SW[7:4] and SW[3:0], and your carry-in bit will come on SW[8]. Display the sum and carry-out as a 5-bit pattern on the LEDs. Your sum should update only when the center button btnC is pressed. For example, if your switch input is SW[7:4]=4'b0111 and SW[3:0]=4'b0101, then when btnC is pressed, your output should be LED[4:0]=5'b01100 (LEDs 2 and 3 should be on). If SW[3:0] changes to 4'b0100, the LEDs should still contain LED[4:0]=5'b01100 until btnC is pressed, at which point, they will change to LED[4:0]=5'b01011. **For this part, you are only allowed to use bitwise operators in Verilog: & (AND), ~ (NOT), | (OR), ^ (XOR).**

1. Design a module named `loadreg.v` that captures the functionality of a load register. Your module should look something like the code below. Be sure to adjust the input and output bit vectors to the appropriate values. Note that this is a behavioral implementation of a load register; a structural/dataflow model would include declaring the gates for the SR latch, D latch, and D flipflop.
```
module loadreg(
    input clk, load,
    input D,
    output reg Q);
initial  Q = 0;
always @(posedge clk) begin
    if(load) Q <= D;
end
endmodule
```
2. Design a module that implements a one-bit full adder with carry-in and carry-out using dataflow modeling. Your module declaration should look like:
```
module adder(
    input a, b, Cin,
    output S, Cout);
```
Do not edit the module declaration in your code.
3. Design a top module named `rca.v` that implements a 5-bit RCA using your module from the previous step. Your module declaration should look like this:
```
module rca (
    input clk, load,
    input [3:0] a, b,
    input Cin,
```

```
    output [4:0] total);
```

Do not edit the module declaration in your code.

4. Fill out the table below that showcases various input combinations. Use the combinations below in your testbench to verify your adders' functionality. Name your testbench `tb_adders.v`

| a | b | Cin | sum | Cout |
|---|---|---|---|---|
| 0000 | 0000 | 0 | | |
| 0000 | 0001 | 1 | | |
| 0001 | 0101 | 0 | | |
| 0111 | 0111 | 0 | | |
| 1000 | 0111 | 1 | | |
| 1100 | 0100 | 0 | | |
| 1000 | 1000 | 1 | | |
| 1001 | 1010 | 1 | | |
| 1111 | 1111 | 0 | | |

To assist in waveform readability, **change the radix/base system that the waveform displays values in.** To do so, select the signal in the waveform window, right-click to see the dropdown menu, hover over the word "Radix", and click "Unsigned Decimal". The waveform should now display the signal values in unsigned decimal. Do this for all bit vectors in your waveform. Below is a snippet of the waveform that we expect (minus the load register functionality; you will need to show that in your waveforms as well):


5. Once you get your waveforms and testbench working, add a constraints file (`adder_constrs.xdc`) and generate the bitstream to make sure that the bitstream will be successful. Note that you will reuse the testbench and constraints file for part B.
6. Once you are almost certain that your code is working, use the verification script provided on the ECE Linux servers to test your adder.

***Part B: Carry-Lookahead Adder (CLA)***

In this part, you will build a 4-bit carry-lookahead adder using **dataflow modeling**. Use the same input and output parameters as the RCA. Remember to pass the output of the CLA into the load register.

1. Derive the sum and carry-out bits $S_0$ through $S_3$ and $C_0$ through $C_4$ for the 4-bit CLA in terms of propagate bits $P_i$, generate bits $G_i$, and the carry-in bit $C_0$. **Make sure that your expressions are in sum-of-products form and NOT in nested form**; you need to

expand all the equations out, as done for $S_2$ and $C_2$ in the Background section on the CLA. (It's not as tedious as you think; there is a pattern.)

2. Design a top module named `cla.v` that implements a 5-bit CLA using dataflow modeling. Your module declaration and first line of code should look like this:

```
module cla (
    input clk, load,
    input [3:0] a, b,
    input Cin,
    output [4:0] total);
wire [3:0] G, P, C;
```

   You may modify the wire declarations to fit your module needs, but you may not modify the module declaration.
3. Add to the testbench in part A to simulate and test your CLA; do not create a separate simulation source. Test both the RCA and the CLA at the same time: instantiate both in the testbench and declare additional wires to capture the extra outputs. Your final simulation screenshot should show the functionality of both adders on the same waveform.
4. Use the constraints file from part A to generate your bitstream. Be sure to test your code on the ECE Linux servers.

***Part C: Multiplexer***

For this part, you will be putting your adders from parts A and B together into a datapath. The datapath should output into a load register, as in the previous parts; however, this time, you will be using a select line to switch between your two adders. Your inputs are as follows:

- SW[15:12] and SW[11:8] – Inputs to your CLA
- SW[7:4] and SW[3:0] – Inputs to your RCA
- btnD – Carry-in bit to both adders
- btnC – Load button for load register
- btnU – Asynchronous select line that chooses between your RCA and CLA. If 0, LED[7:0] displays the output of your RCA. If 1, LED[15:8] displays the output of your CLA. (You won't use all 8 output bits, but the positions are for readability.)

Use (and do not modify) the following module declaration:

```
module datapath (
    input clk, load, sel, Cin
    input [15:0] sw
    output [15:0] led);
```

You will need to remove the load register in the RCA and CLA modules for this part; if you don't, it may take two button presses for your datapath to update to the correct value. You MUST instantiate your RCA, CLA, and load register modules; you cannot copy the code in those files into your datapath module. You may use whatever type of modeling you wish. (If you want to get fancy, you can look into conditional a.k.a. ternary operators.)

Once you finish designing your datapath, write a testbench `tb_datapath.v` and test it using the same testcases as in parts A and B. Don't forget to change the radix of the input and output values to reflect decimal values. Add a constraints file, `datapath_constrs.xdc`, and generate the bitstream. Use the Linux verification tool to also verify your functionality.

## Submission

When you submit labs, you will need to submit two things: your zipped source code and a PDF answering questions given in the lab document. Below are the instructions for creating your zip file. Your PDF should be submitted separately, NOT included inside your zip file.

1. Create the zip file with your design files, testbenches, constraints files, bitstreams, and verification output. You should have the following files:
    a. Sources: `rca.v, cla.v, loadreg.v, datapath.v`
        i. Use the modules you used for part C
    b. Testbenches: `tb_adders.v, tb_datapath.v`
    c. Constraints: `adder_constrs.xdc, datapath_constrs.xdc`
    d. Bitstreams: `rca.bit, cla.bit, datapath.bit`
    e. Verification tool output (from LRC servers): `lab5test-c.cert.out`
2. Create your PDF. Include the following, **in order**:
    a. A cover page with your name, EID, section unique number, and professor.
    b. Table of testcase sums.
    c. One single simulation waveform screenshot showing the functionality of both adders and the load register.
    d. Schematic of the RCA.
    e. Schematic of the CLA.
    f. One to three screenshots of the simulation waveform for your datapath showing the functionality of the select line and everything else.
    g. Schematic of the multiplexer/load register.
    h. One to two screenshots verifying that you ran the test scripts on the Linux servers successfully.
    i. Answers to the checkout questions below.
3. Submit your zip and PDF as one submission on Canvas.

## Checkout Questions

Answer the following questions in your PDF:

1. Calculate the critical path of the RCA and CLA given that an XOR gate uses 6 $um^2$ of area and has 3 ns delay, an AND gate uses 4 $um^2$ of area and has 3 ns delay, and an OR gate uses 4 $um^2$ of area and has 2 ns delay. Use the schematics generated from Vivado.
2. Why does the load register used in the datapath require a clock?
3. In 10 words or less, how would you change your load register module so that it loaded when btnC was both pressed and released?
4. In 20 words or less, why is multiplexing useful in datapath components?

5. Without using any logic gates or other components except one single CLA with 16-bit inputs a and b, draw the datapath circuit that would calculate the average of two 16-bit values x and y, truncated to the nearest integer.
6. Suppose I instantiated a subtractor, multiplier, and divider into your datapath such that it outputs into the same load register as your RCA and CLA. How many more buttons would you have to use to accommodate my additions?

**Rubric**

Below is the grading rubric for this lab. Please review it before submitting your lab to ensure you meet all the requirements.

| | Missing | Attempted | Half-correct | Almost correct | Fully correct |
|---|---|---|---|---|---|
| ***Assessed from zip file submitted on Canvas*** | | | | | |
| RCA | 0 | 2 | 3 | 4 | 5 |
| CLA | 0 | 2 | 3 | 4 | 5 |
| Datapath | 0 | 2 | 3 | 4 | 5 |
| Adder testbench | 0 | 2 | 3 | 4 | 5 |
| Datapath testbench | 0 | 2 | 3 | 4 | 5 |
| ***Assessed from deliverables PDF*** | | | | | |
| RCA + CLA schematics | 0 | 2 | 3 | 4 | 5 |
| Adder waveform | 0 | 2 | 3 | 4 | 5 |
| Datapath schematic | 0 | 2 | 3 | 4 | 5 |
| Datapath waveform | 0 | 2 | 3 | 4 | 5 |
| Checkout questions | 0 | 12 | 13 | 14 | 15 |
| On-time submission | 0 | 7 | 8 | 9 | 10 |
| ***Assessed from autograder screenshots*** | | | | | |
| RCA functionality | 0 | 7 | 8 | 9 | 10 |
| CLA functionality | 0 | 7 | 8 | 9 | 10 |
| Datapath functionality | 0 | 7 | 8 | 9 | 10 |

(Easter egg meme collection :))