Lab 5: Datapaths and Adders

EE 316: Digital Logic Design

1 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 two types of adders.
- 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.

2 Background

In this lab, you will design a ripple-carry adder and a carry-lookahead adder. Ripple-carry and carry-lookahead adders are two fundamental types of adders. In this section, we will discuss how they are implemented.

2.1 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).

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.

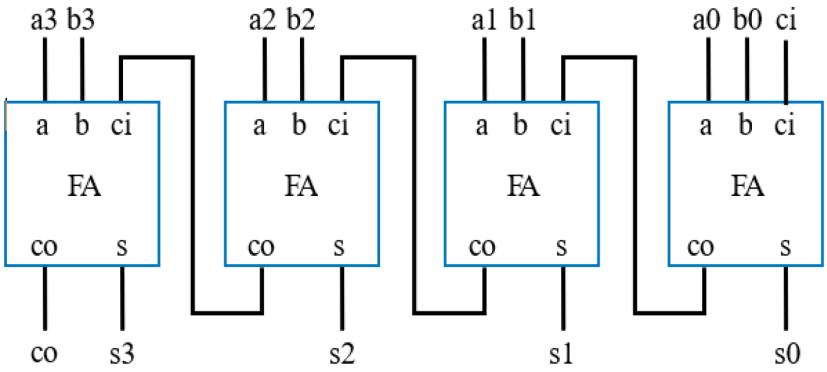

Figure 1: 4-bit Ripple-Carry Adder (RCA)

2.2 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 and the "generate" bit:

$$P_i = a_i \oplus b_i,$$

$$G_i = a_i b_i.$$

From these bits, we can generate our sum and carry-out bits:

$$S_i = P_i \oplus C_i,$$

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

Furthermore, you can expand S_i, C_i and C_{i+1} into a function of P_i, G_i, and C_0. For example, if you wanted to calculate the second sum bit S_2 in a 4-bit CLA, you could write

$$S_2 = P_2 \oplus C_2.$$

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

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

This feature of the CLA design significantly cuts down on the calculation time, as the critical path of the CLA is the critical path of a little over a full adder. However, the CLA uses significantly more gates than the RCA, which is a disadvantage of using a CLA.

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 a constraint 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.

3 Procedure

3.1 Part A: RCA Adder

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. Use the board clock for the `clk` signal.

```
module loadreg(
    input clk, load,
    input D,
    output reg Q
);
initial Q = 0;
always @(posedge clk) begin
    if(load) Q<=D;
    end
```

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 this:

```
module adder(
    input a, b, Cin,
    output S, Cout
);
```

3. Design a top module named `rca.v` that implements a 5-bit RCA using your module from the previous step. Note that because you want your RCA output `Q` to update ONLY on button press, you will need to pass your sum through a load register before passing it to `Q`. Your module declaration should look like this:

```
module rca(
    input clk, load,
    input [3:0] a, b,
    input Cin,
    output [4:0] Q
);
```

a[3:0]	b[3:0]	Cin	S	Cout
0001	0101	0		
0111	0111	0		
1000	0111	1		
1100	0100	0		
1000	1000	1		
1001	1010	1		
1111	1111	0		

Table 1: Adder Testcases

4. Complete Table 1, write a testbench with the testcases given in Table 1 and simulate your circuit, then program and test on-board. Be sure to screenshot your waveforms and save your bitstream to your desktop for easy access.

3.2 Part B: CLA Adder

In this section, you will build a 4-bit carry-lookahead adder using **dataflow modeling**. Use the same input and output parameters as the RCA.

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 in Section 2.2.2. (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**. The module declaration and first few lines of the module should look like this:

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

3. Modify the testbench from part A to simulate your circuit, then program and test on-board. Be sure to screenshot your waveforms and save your bitstream to your desktop for easy access.

3.3 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 `SW[9]` to switch between your two adders. The functionality is described as follows:

- `SW[9] = 0` - `LED[4:0]` should display the output of the RCA, and `LED[9:5]` are off.
- `SW[9] = 1` - `LED[9:5]` should display the output of the CLA, and `LED[4:0]` are off.

As before, use the center button `btnC` to update your LEDs on button press. Therefore, if the user switches between modes, the LEDs should maintain the previous display until the user pushes the button.

1. Draw a high-level block diagram of your system. You may use blocks for your adders and load register. It should contain four blocks.
2. Design a module named `datapath.v` that serves as your top module. 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 modeling you wish. (If you want to get fancy, you can look into conditional a.k.a. ternary operators.)

   Note: 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 may also need to adjust the width of your load register.
3. Simulate and run on-board.
4. In the left sidebar, under "RTL Analysis", click "Open Elaborated Design". Take a high-level screenshot of the entire circuit and a screenshot of the circuit for each instantiated adder module. (You may need to expand the adder modules to see the full circuits.)
5. Using your screenshots, calculate the critical path delay and total area of each adder module, and fill in the table below. Assume that an XOR gate has 3ns delay and 6 area units, and an AND gate has 3ns delay and 2 area units, an OR gate has 2ns delay and 4 area units.

	Area (area units)	Delay (ns)
RCA		
CLA		

4 Sample Checkout Questions

1. What is the tradeoff between an RCA and CLA? Which one is generally preferred?
2. Why is dataflow modeling often used to design datapath components?
3. Explain how each adder works.
4. Explain how the load register works.
5. Why is a clock signal required for your load register? (Hint: Think synchronous vs asynchronous.)
6. How would you implement a reset for your load register?
7. How did you do the multiplexing for part C?

8. Why is multiplexing useful in datapath components?
9. For a 64-bit adder, is the RCA or CLA preferred? Why?
10. If you were to rewrite your adder code in behavioral modeling, how would you do that?
11. How would you design a subtractor using your adders? Incrementor? Decrementor? Multiplier? Bit-shifter?