Tufts University

School of Engineering
Department of Computer and Electrical Engineering

EE103 - Introduction to VLSI Design
Fall 2013 — Dr. Denis Daly

Lab #3, Part II: Schematic Design and Simulation of an 8-Bit Carry Look Ahead Adder

Due: Wednesday, November 13th, 2013

Names: Noah Kurinsky and Nicholas Davis

Email: Noah.Kurinsky@tufts.edu, Nicholas.Davis@tufts.edu

Date Submitted: November 12, 2013

Objective

In this lab, we will implement the basic blocks of an 8-bit 2s compliment adder with look-ahead carry. These parts consist of an 8-bit 1s complimenter, a 1-bit full adder, and a 4-bit carry lookahead unit. After creating these components in Cadence, we will analyze them through verilog simulations, utilizing the verilog primitives constructed for basic logic gates in the first part of Lab 3. Finally, we make some comments regarding future directions for our 8-bit adder implementation.

Introduction

For our final project, we have chosen to create an 8-bit adder/subtractor. For our first iteration we have implemented this with carry lookahead logic, which will keep the time delay for the adder down by computing the carry out bit of each stage in parallel. In order to test our initial implementation we have broken down the adder into modular components. The 1s complimenter component takes in a control signal, and 8 bits of input. When the control signal is high, the 8 inputs are inverted, and when the control signal is low, they are left alone.

For our design, we do not actually want to 1s compliment the second 8-bit number, but instead we want to have a 2s compliment system. The reason we broke out the 1s compliment logic into a separate module is because with our full adder implementation, as long as the control signal is tied to the carry-in for the first full adder and the carry logic, the second 8-bit number will be in effect 2s compliment (1s compliment + 1). As mentioned, the second module is a 1-bit full adder. This adder takes in 2 input bits, 1 carry-in bit and outputs a sum bit, a propagation bit and a generation bit. The last module in our design is the carry lookahead unit. This unit takes in 4 sets of propagation and generation bits (8 bits total) as well as the first carry-out bit in the system. The carry lookahead unit outputs 4 carry-out bits.

The carry look ahead logic (implemented as a Carry Look-ahead Unit, or CLU) entails removing the carry function from the standard full adder, and computing carry bits in parallel to decrease the number of gates from input to output in the higher bits. The logic for this lookahead carry is simple for the carry in terms of the previous carry, generation, and propagation terms:

$$c_{i+1} = G_i + P_i c_i \tag{1}$$
$$s_i = P_i \oplus c_i \tag{2}$$

where P_i and G_i are computed by the full adder instead of a carry bit according to the following equations:

$$G_i = a_i \cdot b_i \tag{3}$$
$$P_i = a_i \oplus b_i \tag{4}$$

where a_i and b_i represent input bits and c_i is the carry input to the full adder. Using these equations, an equation for each carry can be generated in terms of only propagation, generation and the first carry input; this means each carry is no longer dependent on the previous carry and they can be computed quickly in parallel. The equations for a 4–bit CLU are seen below:

$$c_1 = G_0 + P_0 \cdot c_0 \tag{5a}$$
$$c_2 = G_1 + P_1 G_0 + P_1 \cdot P_0 \cdot c_0 \tag{5b}$$
$$c_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0 \tag{5c}$$
$$c_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0 \tag{5d}$$

Here it is easy to see how larger bit implementations become excessively large.

The final architecture represents a combination of carry-look ahead and ripple carry logic, to balance the speed gained from the look-ahead logic with the complexity, area, and delay inherent to 8-bit CLUs. We construct a 4-bit CLU, and use two rippled CLUs to compute 8-bit carry look-ahead logic with one ripple step. This is not necessarily as fast as a single 8-bit unit, but is more efficient in terms of area and power. In the following section we detail the development process, and in the subsequent section we discuss our results. Finally, we briefly discuss the direction we will take in our final project.

Development and Simulation

To design the adder, we first constructed the basic functional modules out of the primitive gates designed in Part I of Lab 3. For each module, we constructed a cadence schematic out of primitive gates, verified it, and created a symbol view with all input and output pins specified. We then simulated the 1s compliment module, and the full adder module separately to make sure that they functioned correctly. Once all of the symbols had been completed and the schematics had been verified and simulated, we took the symbols and constructed a schematic for the full 8-bit 2s compliment adder, wiring the components together appropriately and creating i/o pins for both input numbers, an add/subtract control bit, and the addition output. Once the full schematic was complete, we verified it through simulation with Verilog-XL, which generated the verilog code for us using the gate primitive functional views. When simulating each schematic, we used the input signals shown in the stimulus files (see appendix A); for the final circuit, the four input combinations are those given in the lab.

1 1‘s Complement Module

Our 1s compliment implementation is very simple. Each input bit is simply XOR’d with the control bit. Therefore, when the control is 1, all the bits will flip, and when the control is 0 the will stay the same. This manifests itself simply as 8 XOR gates in parallel, with 9 inputs and 8 outputs, as seen in the schematic in figure 1 and illustrated in the symbol in figure 2. In the lab, it is specified that we create a separate module for 2s compliment, and not for 1s compliment, but in our design, the 2s compliment functionality is built into the overall design, and cannot be separated out. In our design, in order to complete the 2s compliment logic, the control signal for the 1s compliment module is tied to the carry-in of the least significant adder in the design. This completes the 2s compliment algorithm, which is essentially to flip all of the bits in the second number, and add 1.

Because we cannot separate the 2s compliment logic, we test our 1s compliment logic separately to show that it will work as expected. In order to test this module, we must simulate it using Verilog XL, and give the input signals a few different operating conditions (different input patterns). Because we know what the module is supposed to output given a set input, we can determine that the module has been implemented correctly by covering all possible edge cases, as well as a base case. In figure 3, you can see the output wave form of our simulation. We fed in 4 different input combinations to verify that when requested, all input bits are properly inverted, and if not the module behaves transparently. As you can see from the figure, the module behaves as expected, and therefore we can move on to simulating the next module.

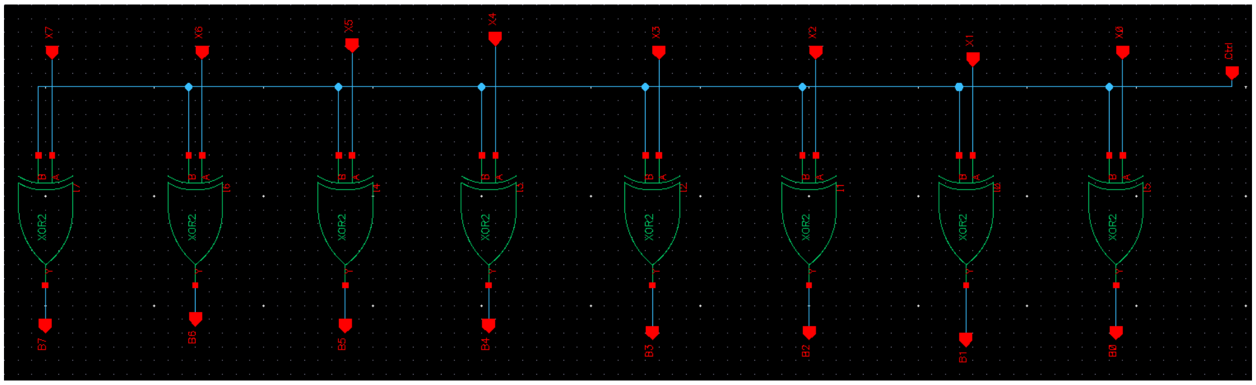

Figure 1: The schematic of the one's complement module. Along the top, all of the inputs are lined up labeled X0-X7, and each is connected to the first input of an XOR gate. A ninth input is wired to the other terminal of all the gates, and is called control; this specifies whether to flip the bits. The gate outputs are connected to pins labeled B0-B7, as they will be referred to in the full 8-bit adder.

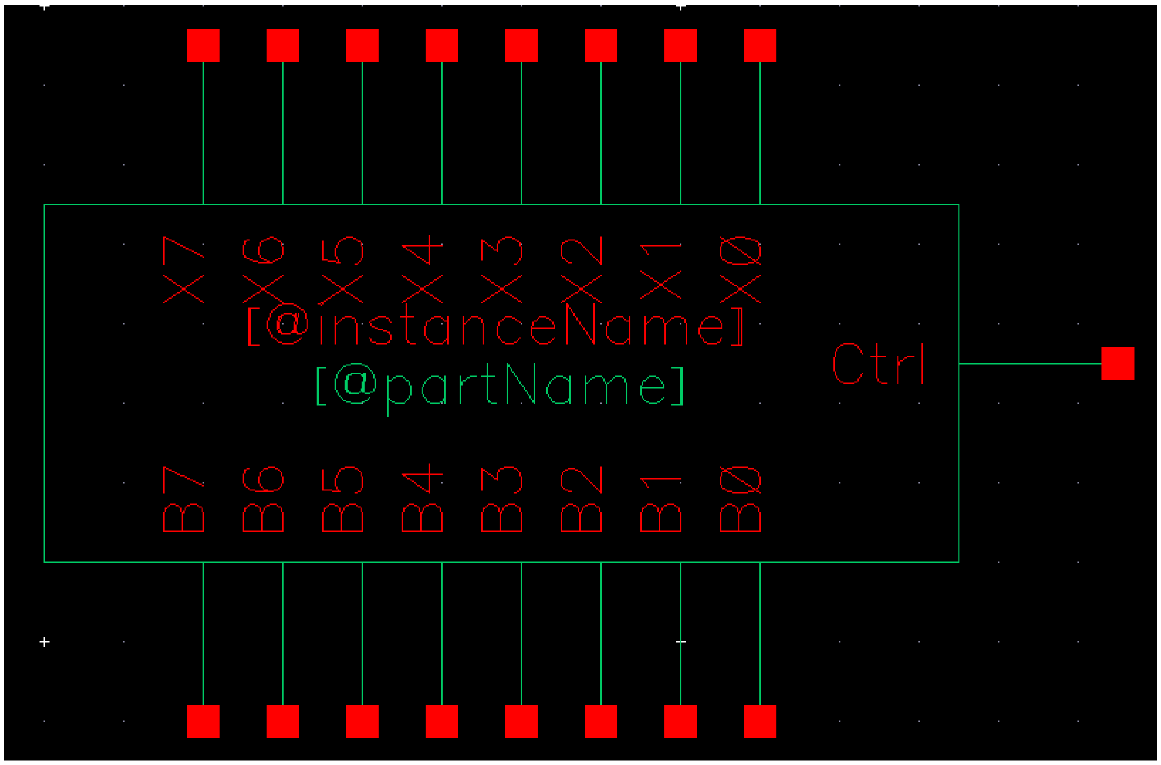

Figure 2: The symbol for the One's complement circuit with inputs and outputs matching those seen in figure 1.

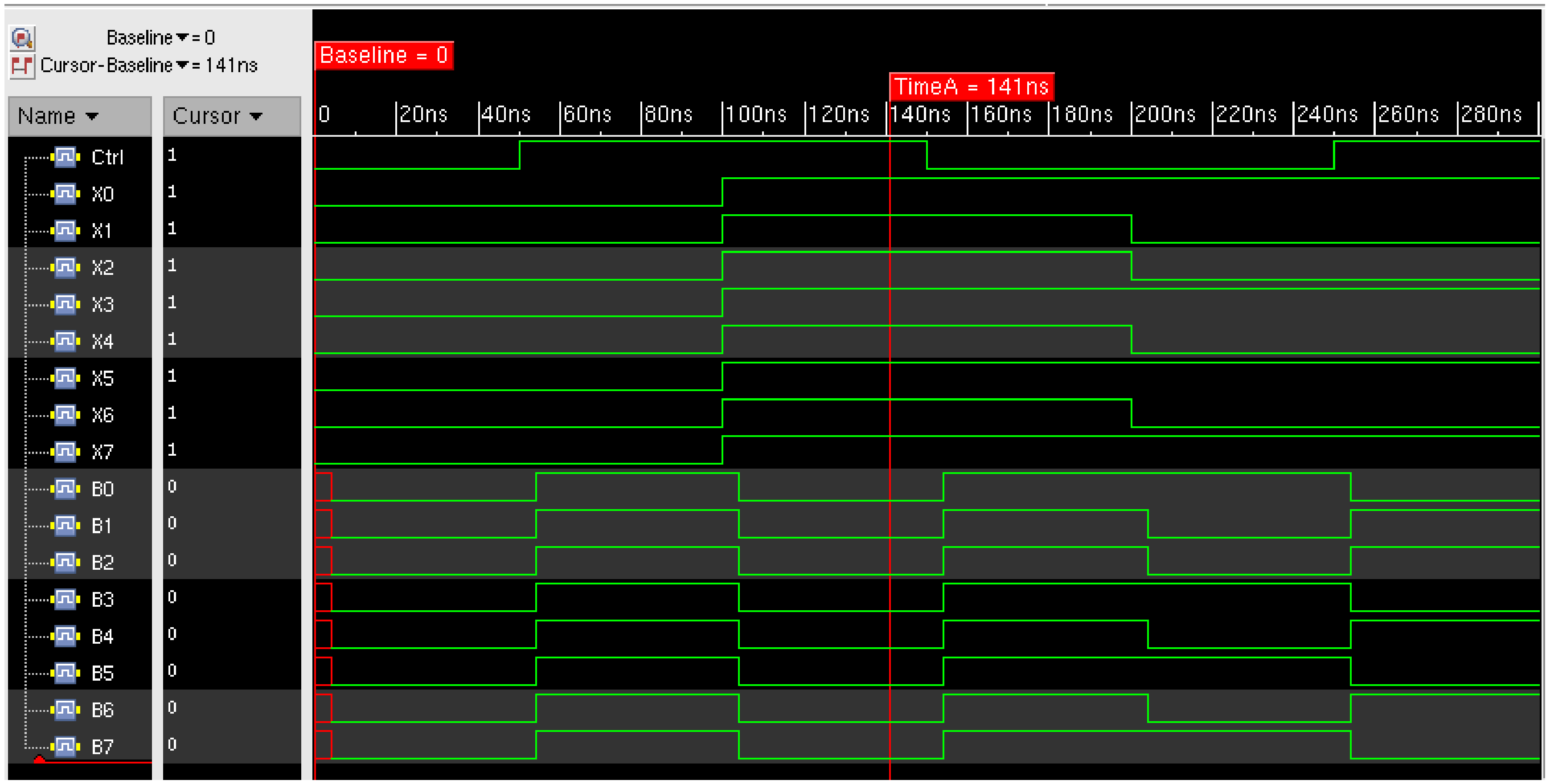

Figure 3: The one's complement simulation, with the control signal at the top of the plot, the 8 input bits after that, and the 8 output bits at the bottom; the labels corresponding to the symbol can be seen along the left column. The first sequence shows a small delay from the XOR gates followed by valid output. T

2 Full Adder Module

A traditional full adder takes in 3 bits (input A, input B, and carry-in) and outputs 2 bits (sum, and carry-out). Because we are implementing our adder using carry lookahead logic, we must change the output of our full adder appropriately. Our full adder takes in 3 bits (input A, input B, and carry-in), but it outputs 3 bits. These 3 bits are the sum, propagation bit, and generation bit. These two bits are determined by following boolean algebra expressions introduced in the introduction (see eqns (3) and (4)). The propagation and generation bits will be used later as inputs to the carry lookahead unit module.

This module must be tested in the same way as the 1s compliment module, however, because it is only a 1-bit module, the test cases are much more limited. We will be using 8 of these modules in total, but as long as we can verify the design of the single 1 bit adder, we know that all of them will work. The stimulus file in section A1 was used to create the waveform which drove one full adder unit through all possible input sequences. In figure 5, you can see the output waveform of the Verilog XL simulation. As you can see, the module behaves as expected, and therefore we can move on to the carry lookahead unit.

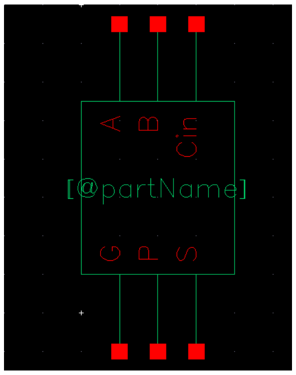

Figure 4: The symbol for the one-bit full adder, with input and outputs matching those found in the schematic in figure 6.

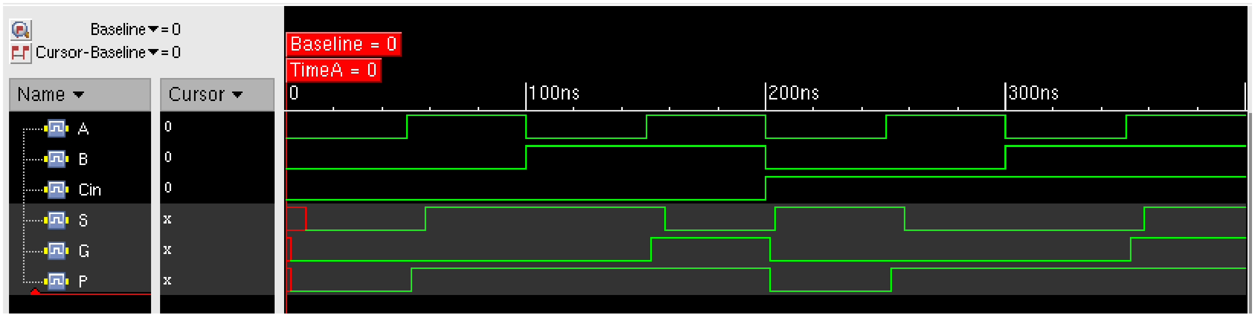

Figure 5: Verilog-XL simulation results for the full-adder schematic shown in figure 6. On the top are the three inputs, Cin, A, and B, and on the botth we have S, G, and P. Checking the outputs against equations (2), (3), and (4), we see that all outputs function correctly. For instance, at the end of the simulation, with all inputs high, we have both a sum of one and a command to generate a carry; when A and B are 1 and C is 0, we have 0 sum and both propagate and generate. Finally, propagation is not requested only when A and B are 0.

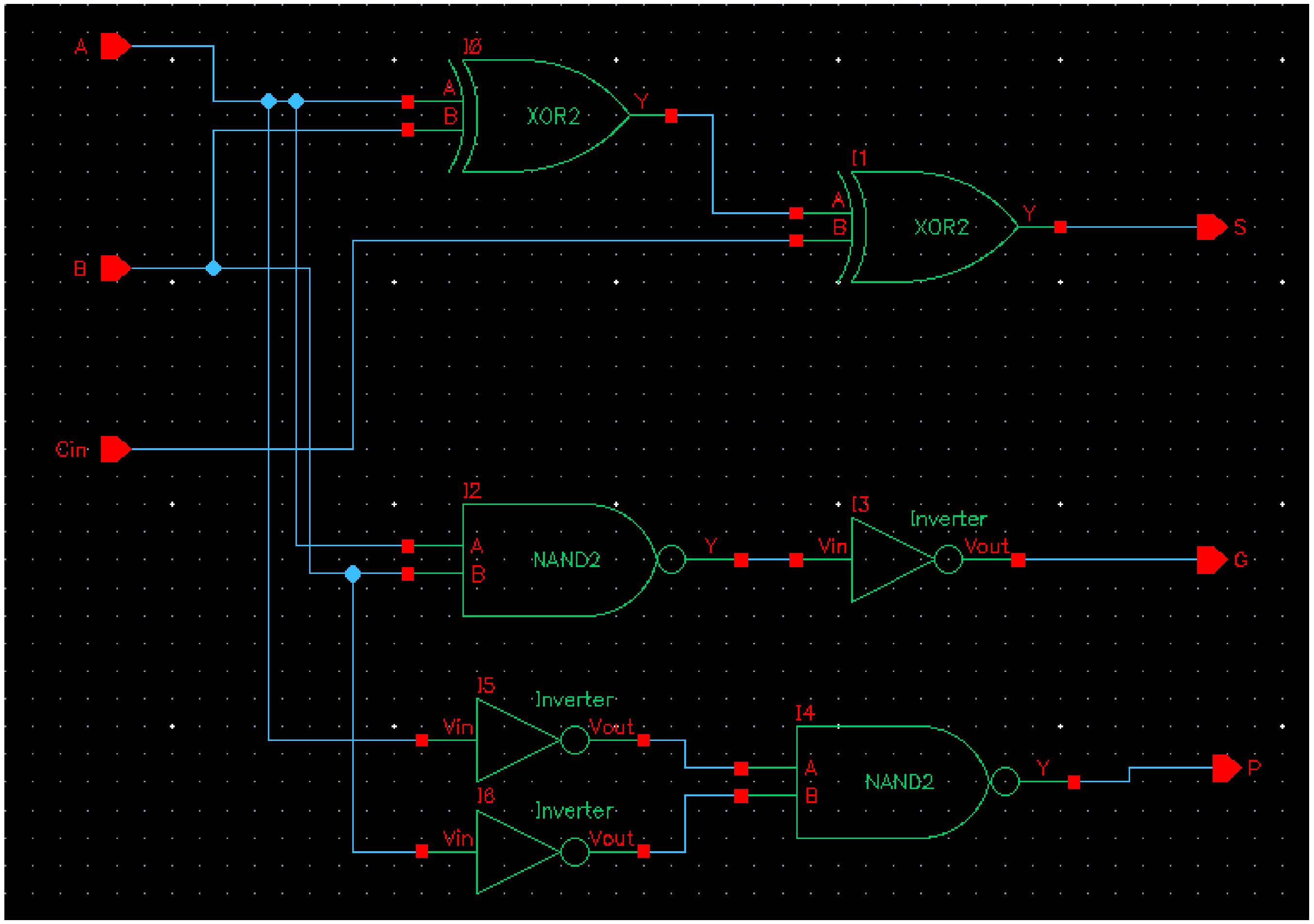

Figure 6: Full adder schematic, with the inputs on the left and outputs on the right. At the top, we see the sum generation, made of two stages of 2-input XOR gates. The middle row shows the generation logic matching that in equation (3), and the bottom shows what is essentially an OR gate, substituted for the XOR as, from equation (1), the condition where P would be low in an XOR but high in an OR is valid anyhow, as the G will normally make this equation evaluate to true.

3 Carry Look-Ahead Unit (CLU) Module

This module will compute 4 carry bits in parallel, which will significantly speed up our adder from the traditional ripple-carry implementation. These carry bits are generated in parallel by using the propagation and generation bits from each adder. Because of the algorithms for each carry bit, the hardware complexity is greater than that of a ripple-carry adder. This module is not simulated separately from the main adder since we have simulated the other 2 modules, and thus can expect errors to mainly result from this logic; it is also not integral to the computation, merely a means of accelerating it, whereas the other components are the basic components required to perform this calculation.

We implemented the carry lookahead unit using only NAND gates and inverters. This means that the hardware complexity of the unit will be larger than if we used AND/OR/inverter logic, but it will be easier to optimize on the transistor level, and can still possibly be smaller in size; in addition, this gate combination may be more efficient than using NOR and XOR gates due to the high delay inherent in these gates. The schematic for the CLU unit can be seen in figure 9, and the symbol can be seen below, in figure 7.

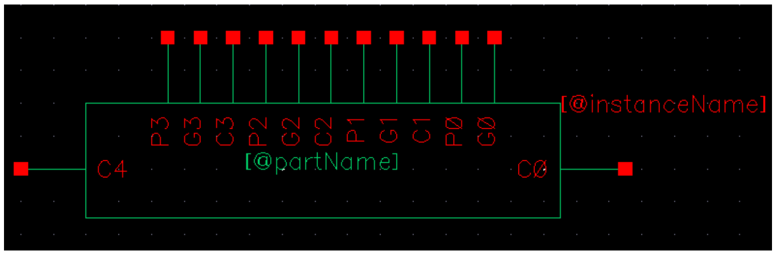

Figure 7: Symbol for the Carry Look-Ahead Unit associated with the schematic in figure 9.

4 Top Level Integration

The final product created by connecting all of our modules is the 8-bit 2s compliment carry look-ahead adder. This adder is made up of a single 1s compliment module, 8 full adders, and 2 4-bit carry lookahead units, as seen in the schematic in figure 10; the symbol we created for this circuit can be seen in figure 8. In the schematic, the digits of B, the second argument, are fed to the one's complement block, which also takes the control line as input. This line specifies whether addition or subtraction is to be done, a 0 meaning addition and a 1 meaning subtraction. Treating this input also as the carry-in, we achieve 2's complement addition, as we can treat the B argument as the one's complement plus one when subtraction is requested. The two CLU units, each four bits wide, are at the bottom, and the propagate and generate signals are fed to the blocks accordingly. The full adders receive the carries from this CLU unit, and the final carry of the first unit is used as the carry in to the second unit and second four bit group.

In order to verify that this adder works as expected, we implement 4 specific input cases given by the lab in order to verify that the module works, in addition to a simple initialization test case. These five test conditions are given as

$$00000000 + 00000000 = 0\ \ 00000000 \tag{A}$$
$$01111111 + 00000001 = 0\ \ 10000000 \tag{B}$$
$$10001111 + 01111111 = 1\ \ 00001110 \tag{C}$$
$$10000000 - 00000001 = 1\ \ 01111111 \tag{D}$$
$$01111111 - 10001111 = 0\ \ 11110000 \tag{E}$$

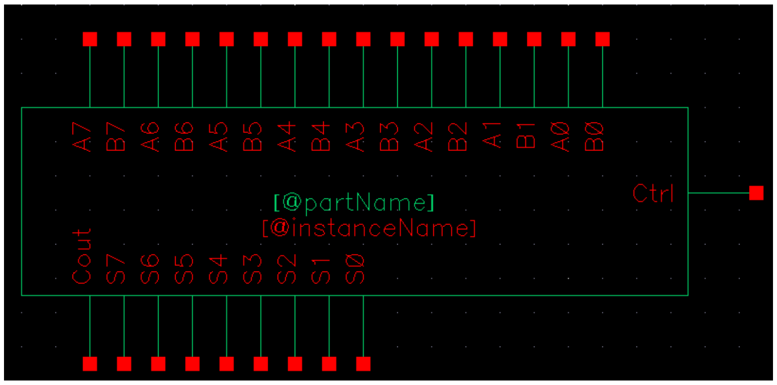

Figure 8: Symbol for the end product of our efforts, the 8-bit full adder with 2‘s complement subtraction and look-ahead carry logic. On the top are the 16 inputs corresponding to two 8-bit numbers, and on the right is the control bit which specifies whether to perform addition or subtraction. On the bottom are the 8 output bits and the carry out, which essentially indicates whether overflow or underflow has occurred. The associated schematic can be seen in figure 10.

We input these cases into our stimulus file (seen in section A3 for this simulation, and run it using Verilog-XL. As shown in figure 11 and table 1, the output of our adder was correct for every case. It successfully performed addition and subtraction, and even was able to return 2s complemented sums (negative numbers). We also analyzed our simulation results for hold time, setup time, and propagation delay. These three delays can also be seen below in table 1. For the table, the propagation delay is the time between input applied and all outputs settling to their final values, the hold time is essentially analogous to the contamination delay (time after transition that the previous output is still stored) and setup time is the difference between the two – essentially, the duration during which the inputs must be held constant for a valid output to ever be seen.

Carry	Y[7:0]	Hold Time [ns]	Setup Time [ns]	Prop. Delay [ns]
0	00000000	10	8	18
0	10000000	5	17	22
1	00001110	7	7	14
1	01111111	8	17	25
0	11110000	8	5	13

Table 1: Adder simulation results, as read from figure 11.

The cumulative metrics from the above table are then that our propagation delay is 25 ns, our hold time is 5 ns, and our setup time is about 17 ns.

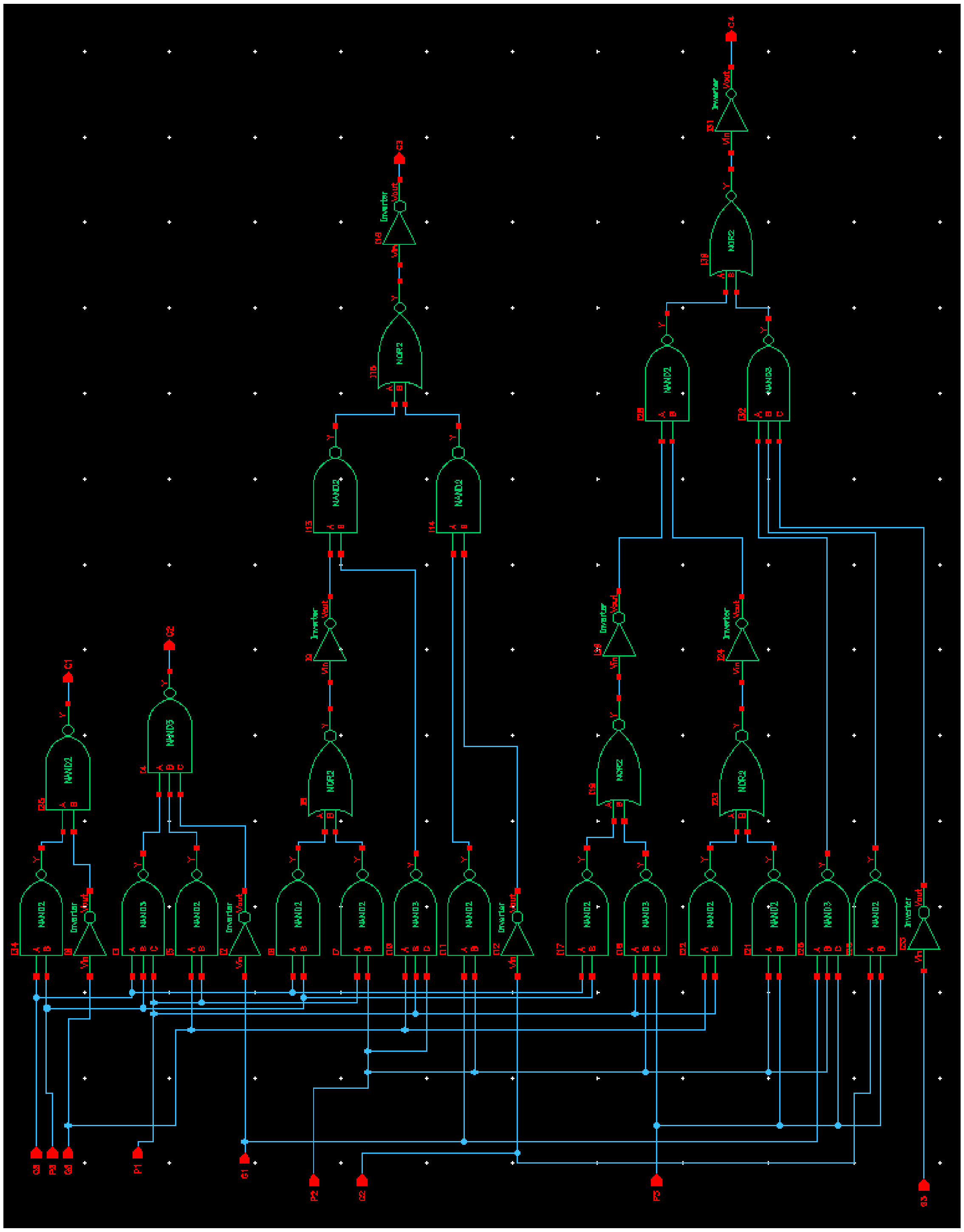

Figure 9: The CLU unit schematic, with inputs for four sets of propagation and generation signals and a carry in, and four carries as outputs. As expected from the equations in the introduction, the gate complexity increases with carry number; we opted to implement only NANDs where possible instead of constructing larger gates to minimize delay and logic complexity; this is the main bottleneck (especially for c4) and will be improved upon in the final design.

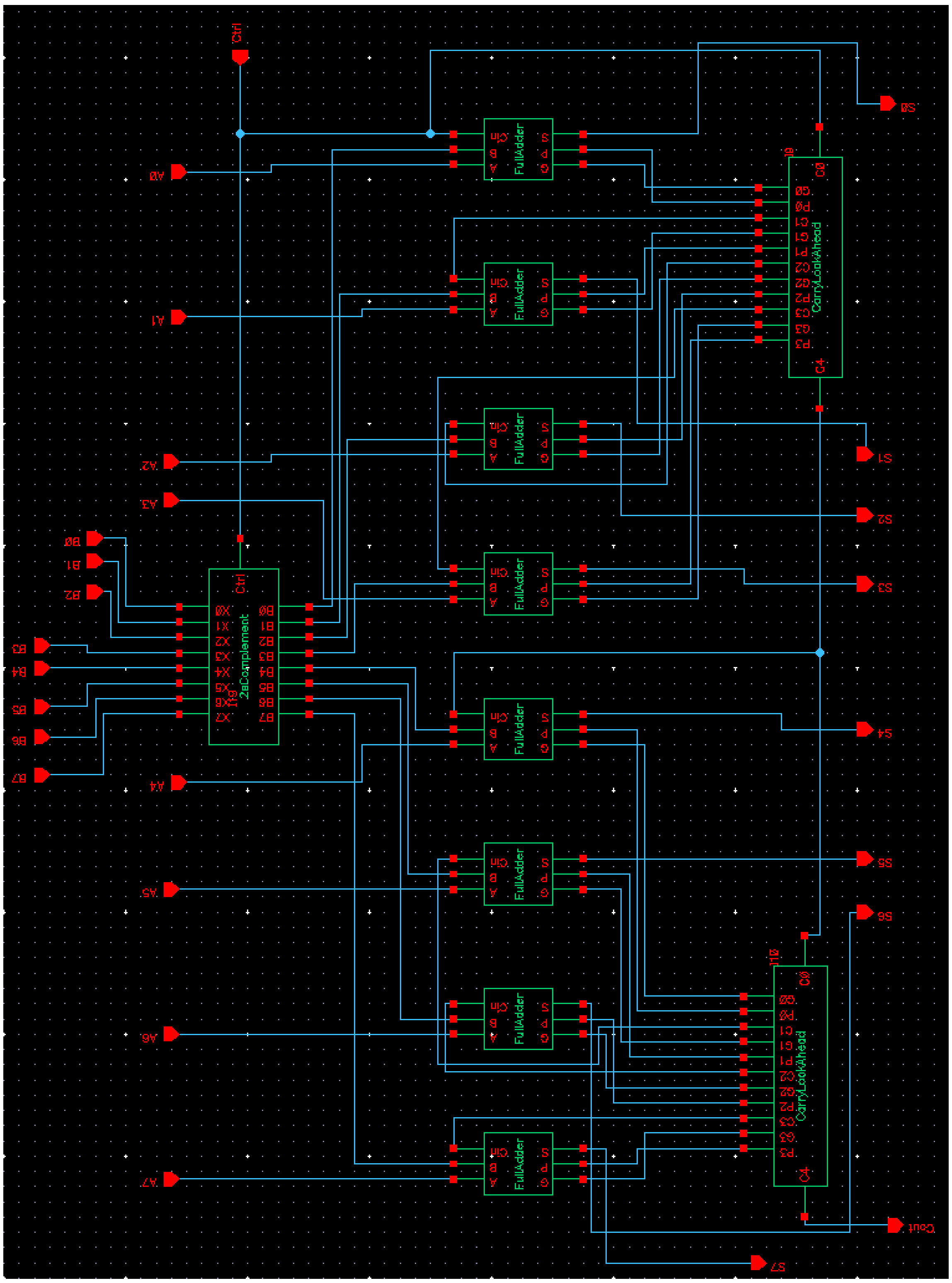

Figure 10: Schematic of the full 8-bit adder with subtraction and look-ahead carry. Note the control input is passed to an adder, the one's complement logic, and the CLU to perform 2's complement subtraction. We have one 1's complement block, 8 full adder blocks, and two CLUs with ripple between the 4-bit CLU stages. All inputs are on the top, and outputs are at the bottom. The wiring is self explanatory as the pins of our modules are marked with expected inputs and outputs.

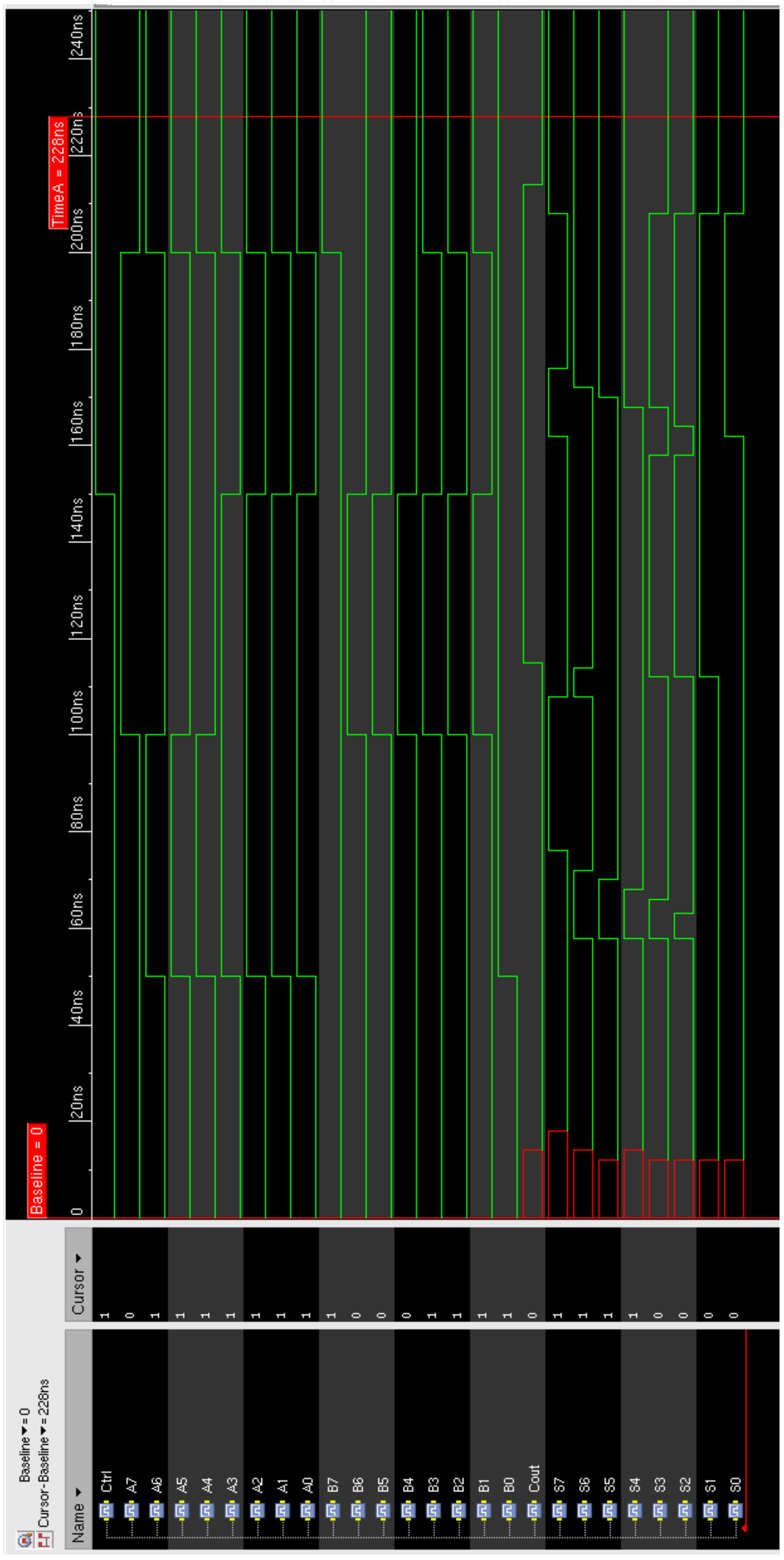

Figure 11: Full simulation of the 8-bit full adder, using the four requested computations from the lab as specified by the stimulus file in section A3. At the top is the control signal, specifying whether to add or subtract, then the first number followed by the second number. The last 9 rows are first the carry-out, and then the 8-bit output. There are actually 5 test cases here, the 5 specified and the all 0 input case. The initialization case shows that there is significant delay especially in the last sum bit, though in general all bits suffer large initialization delays. See the main text for more details about each test case. In summary, all cases were computed successfully with a maximum propagation delay of about 22ns.

Discussion

There are a few main points we took from the exercise of implementing our initial design. First, there is a large trade-off between speed and simplicity; we could have employed a ripple carry adder with many XOR gates, but it would have been smaller and simpler at the cost of speed. In our future design, we will have to take a look at the specific gates we will need for each part, and potentially design special gates such as the and-or-invert, which was seen frequently in a few of the schematics; in addition, some of the signals are always used in inverted form, and we may be able to use this to reduce gate numbers.

The largest bottleneck seems to have been our CLU unit, which has both the longest path length and largest effect on delay between different digits, although it is improved from how it would be for the ripple carry adder. We will have to consider whether to rethink how this logic is designed, and we may be able to improve our speed by using domino logic or other similar types of implementations; this of course will come at the cost of power, as we would have to introduce a clock. In addition, our design is not modular, that is, you could not string two together to create a 16-bit adder; whether we modify the design to make this possible is still under consideration.

Finally, we saw that despite the initial delay reduction logic, our design does suffer from much delay. We will take a careful look at component sizing, attempt to size certain paths to minimize delay, and implement some functions at the transistor level to simplify the design, minimizing area and therefore capacitance. This lab has shown us that our design has much room for improvement despite the start we got in implementing simple improvements.

In conclusion, in this lab we learned how to unit test our products using verilog simulations and cadence schematics. We created an end-to-end product, acceptance tested its functionality, and learned about the hardware development process. The combination of unit testing and acceptance testing is critical to understand in industry, and will help us have a faster and more accurate development cycle for the final project. As shown in our analysis, this lab was a success. Every module worked as expected, as did the final adder.

A Stimulus Files

Below are the stimulus files used to run the Verilog simulations of all components simulated in this lab, including the Full Adder, 1‘s Complement Circuit, and 8-bit Full Adder.

1 Full Adder

```
initial
begin

    A = 1'b0;
    B = 1'b0;
    Cin = 1'b0;

    #50 A = 1'b1;
    B = 1'b0;
    Cin = 1'b0;

    #50 A = 1'b0;
    B = 1'b1;
    Cin = 1'b0;

    #50 A = 1'b1;
    B = 1'b1;
    Cin = 1'b0;

    #50 A = 1'b0;
    B = 1'b0;
    Cin = 1'b1;

    #50 A = 1'b1;
    B = 1'b0;
    Cin = 1'b1;

    #50 A = 1'b0;
    B = 1'b1;
    Cin = 1'b1;

    #50 A = 1'b1;
    B = 1'b1;
    Cin = 1'b1;

    #50 $finish;

end
```

2 1's Complement

```
initial
begin

   Ctrl = 1'b0;
   X0 = 1'b0;
   X1 = 1'b0;
   X2 = 1'b0;
   X3 = 1'b0;
   X4 = 1'b0;
   X5 = 1'b0;
   X6 = 1'b0;
   X7 = 1'b0;

   #50 Ctrl = 1'b1;
   X0 = 1'b0;
   X1 = 1'b0;
   X2 = 1'b0;
   X3 = 1'b0;
   X4 = 1'b0;
   X5 = 1'b0;
   X6 = 1'b0;
   X7 = 1'b0;

   #50 Ctrl = 1'b1;
   X0 = 1'b1;
   X1 = 1'b1;
   X2 = 1'b1;
   X3 = 1'b1;
   X4 = 1'b1;
   X5 = 1'b1;
   X6 = 1'b1;
   X7 = 1'b1;

   #50 Ctrl = 1'b0;
   X0 = 1'b1;
   X1 = 1'b1;
   X2 = 1'b1;
   X3 = 1'b1;
   X4 = 1'b1;
   X5 = 1'b1;
   X6 = 1'b1;
   X7 = 1'b1;

   #50 Ctrl = 1'b0;
   X0 = 1'b1;
   X1 = 1'b0;
```

```
    X2 = 1'b0;
    X3 = 1'b1;
    X4 = 1'b0;
    X5 = 1'b1;
    X6 = 1'b0;
    X7 = 1'b1;

    #50 Ctrl = 1'b1;
    X0 = 1'b1;
    X1 = 1'b0;
    X2 = 1'b0;
    X3 = 1'b1;
    X4 = 1'b0;
    X5 = 1'b1;
    X6 = 1'b0;
    X7 = 1'b1;

    #50 $finish;

end
```

3 8-Bit Full Adder

```
initial
begin
    A7 = 1'b0;
    A6 = 1'b0;
    A5 = 1'b0;
    A4 = 1'b0;
    A3 = 1'b0;
    A2 = 1'b0;
    A1 = 1'b0;
    A0 = 1'b0;
    B7 = 1'b0;
    B6 = 1'b0;
    B5 = 1'b0;
    B4 = 1'b0;
    B3 = 1'b0;
    B2 = 1'b0;
    B1 = 1'b0;
    B0 = 1'b0;
    Ctrl = 1'b0;

    #50 A7 = 1'b0;
    A6 = 1'b1;
    A5 = 1'b1;
    A4 = 1'b1;
    A3 = 1'b1;
    A2 = 1'b1;
    A1 = 1'b1;
    A0 = 1'b1;
    B7 = 1'b0;
    B6 = 1'b0;
    B5 = 1'b0;
    B4 = 1'b0;
    B3 = 1'b0;
    B2 = 1'b0;
    B1 = 1'b0;
    B0 = 1'b1;
    Ctrl = 1'b0;

    #50 A7 = 1'b1;
    A6 = 1'b0;
    A5 = 1'b0;
    A4 = 1'b0;
    A3 = 1'b1;
    A2 = 1'b1;
    A1 = 1'b1;
    A0 = 1'b1;
    B7 = 1'b0;
    B6 = 1'b1;
    B5 = 1'b1;
    B4 = 1'b1;
    B3 = 1'b1;
    B2 = 1'b1;
    B1 = 1'b1;
    B0 = 1'b1;
    Ctrl = 1'b0;

    #50 A7 = 1'b1;
    A6 = 1'b0;
    A5 = 1'b0;
    A4 = 1'b0;
    A3 = 1'b0;
    A2 = 1'b0;
    A1 = 1'b0;
    A0 = 1'b0;
    B7 = 1'b0;
    B6 = 1'b0;
    B5 = 1'b0;
    B4 = 1'b0;
    B3 = 1'b0;
    B2 = 1'b0;
    B1 = 1'b0;
    B0 = 1'b1;
    Ctrl = 1'b1;

    #50 A7 = 1'b0;
    A6 = 1'b1;
    A5 = 1'b1;
    A4 = 1'b1;
    A3 = 1'b1;
    A2 = 1'b1;
    A1 = 1'b1;
    A0 = 1'b1;
    B7 = 1'b1;
    B6 = 1'b0;
    B5 = 1'b0;
    B4 = 1'b0;
    B3 = 1'b1;
    B2 = 1'b1;
    B1 = 1'b1;
    B0 = 1'b1;
    Ctrl = 1'b1;
    #50 $finish;
end
```