Lab 2- Adders and Complexity

Cody Joy

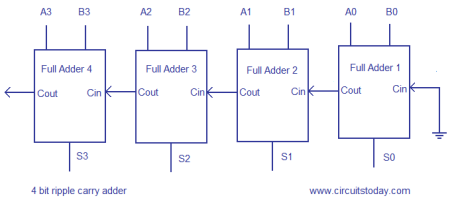
1 Introduction

The purpose of this lab is to create 8-bit ripple, conditional, and carry-lookahead adders to analyze the speed of the solution and the complexity of the circuit.

Normally when addition is needed in Verilog, the programmer can just add a plus sign. While this is easy to use, it is important to be able to understand how the program completes this addition. Three of the most common types of adders are the ripple, the conditional, and the carry-lookahead.

The ripple-adder is essentially full adders cascaded together, as shown in Figure 1. In the first full adder, the first bit of number A and B are added together with cin. The output is a Cout and a sum. This Cout is added with the second bit of number A and B in full adder two. This Cout is then input into the third full adder. This cascading occurs for each stage of full adder through the total number of bits in the two numbers to be added. This adder is the easiest of the three to construct.

Figure 1: 4-bit ripple adder



The second type of adder is the conditional adder. The conditional adder by solving for both the case where the original Cin is 1 and the case where the original Cin is 0. The first step is to calculate the carry and sum bit from the addition of the bit from number A, number B, and Cin. For an 8 bit number, 16 different 2 bit numbers are calculated, 8 with carry and 8 without carry. From here, bits 0 and 1, 2 and 3, 4 and 5, and 6 and 7 are combined using muxes to create 3 bit numbers, either the carry or without carry 2 bit odd number, based on the sum bit of the without carry number, is combined with the sum bit of the of the even number. This process is continued until only a with carry and without carry number are left. The correct number is based on if there was or wasn't an initial carry. An example is shown in Figure 4, with Figures 2 and 3

showing the Conditional Adder block diagram. While the conditional adder is more difficult to construct, it is easier for the computer to construct.

Figure 2: Conditional Adder

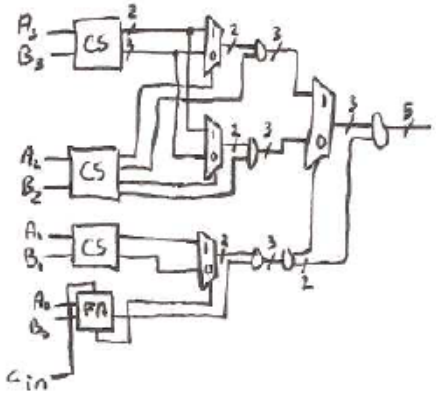
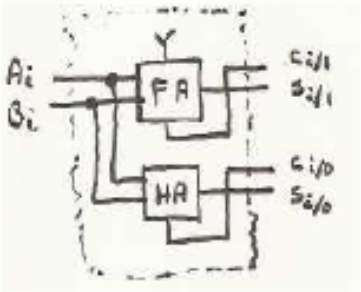


Figure 3: CS Sub Block



The third type of adder is the carry-lookahead adder. The carry-lookahead works by pre-generating 2-level carries. These 2-level logic carries are generate, propagate, and carry. The generate is created by the bit of A AND the bit of B. The propagate is the bit of A OR the bit of B. The carry is generate OR propagate AND the previous value of the carry. The carry-lookahead can then be used like a ripple adder by solving each of these carries one after the other. The final value of carry can also be calculated by substituting the previous values of carry. The allows the user to solve for the final value of carry based on the initial carry and all the previously generated values of generate and propagate, as shown in the following equation.

$$C_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + C_0P_0P_1P_2P_3$$

Figure 5 shows a diagram of the carry-lookahead. The sum is then simply just the EXOR of P and C.

Figure 4: Conditional Adder Example

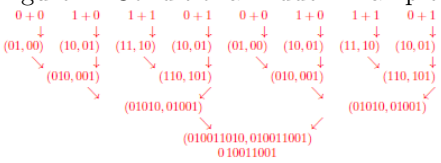
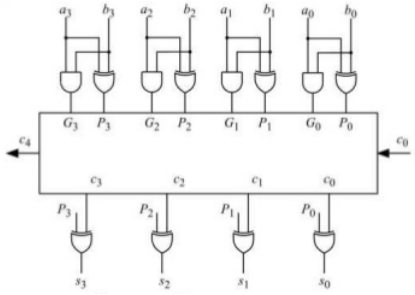


Figure 5: Carry-Lookahead



2 Interface

To implement the three types of adders, Verilog programs were created that were to be run on Nexys 4 DDR boards. The initial string of 8 bits numbers to be added together were created using the switches on the board. The first 8 for number A and the second 8 for number B. The output is displayed using the first 8 LEDs. The button BTNC is used to determine if addition or sutraction will be completed. The button will not be pushed for addition, but will be pressed for subtraction.

3 Design

The first type of adder is the ripple adder. As shown in figure 1, the ripple adder is symply full adders cascaded together. An initial Carry in is given. This carry in can either be 0 for addition or 1 for subtraction. Bit 0 of the two numbers to be added are also input into the full adder. A schematic of the full adder is shown in figure 6. The full adder has two output, the sum and the carry out. Both of these outputs are one bit numbers. The carryout is input as the carry in of the next full adder, which is then combined with the bit 1 of the numbers to be added. This full adder also creates a sum and a carry bit. Again, the carry out bit become the carry in bit for the next full adder. This process continues for the number of bits of the two numbers to be added together. The final sum is the concationation of all of the individual bit sums found from the full adder. While the ripple adder is simple to construct, the time to calculate the values is very slow. This can be seen in figure 9, the simulation of the ripple adder.

To calculate an 8 bit number, the signal will need to travel through 24 different logic gates. While this may not be extraordinarily slow, it is much slower than either the conditional adder or the carry-lookahead adder.

Figure 6: Full Adder Gate Level Logic

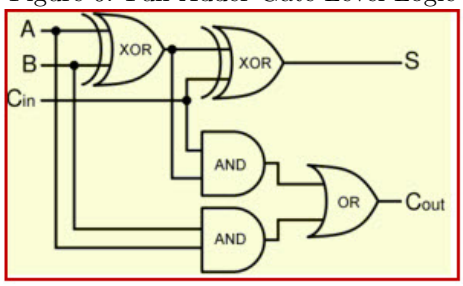
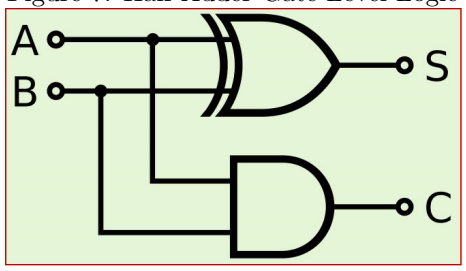


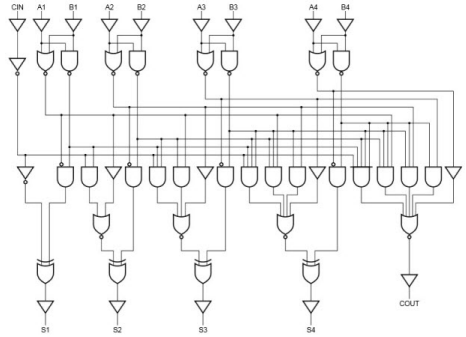
Figure 7: Half Adder Gate Level Logic



The second type of adder is the conditional sum adder. A diagram of this adder is shown in figure 2. The first step of the conditional adder is to input the different bits into the CS sub block, shown in figure 3. This CS sub block will produce two results, a sum and a carry from the full adder and a sum and carry from the half adder. Since the full adder is conected to power, carry in is always set to 1. This means that the sum and carry from the full adder will produce a number with a carry in of 1, and the half adder will produce a number with a carry in of 0. In this way, the initial carry in does not have to be known until the end. The CS sub block will produce two 2 bit numbers. In the next step, two input muxes will be used to combine multiple 2 bit numbers into three bit numbers. The results from the half adder and full adder of the even CS blocks are input into the two input mux. The odd CS blocks are each of the different even bits of the number to be added together. The carry bit of the even CS block is used to choose between the two inputs. The result chosen by the mux is then concatonated together with the sum bit of the odd CS block. This process occurs for the with carry and without carry case. After the end of the first stages of muxes, there is now currently half the number of input wire as originally. The next stage will follow the same procedure as the previous stage. The carry bit will choose between two 3 bit numbers, and this number will be

concatonated with the remaining 2 bits to create a 5 bit number. FInally, for an 8 bit number, there should be four 5 bit numbers remaining. Two from the without carry case, and two from the with carry case. The next step will be to combine the two without carry into one final 9 bit without carry number and combine the two with carry numbers into one final 9 bit with carry number. Bits 0 through 7 are the addition of the two 8 bit numbers and bit 8 is the carry out of the adder. Overall, this adder is more difficult to build, but it is easier and faster for the computer to read. For an 8 bit number, the signal will travel through 12 gates, half of the number of gates of the ripple carrier. Finally, the output is shown all at once, not cascaded like the ripple carrier. This is shown in figure 10, the simulation of the conditional adder.

Figure 8: Carry-Lookahead Gate Level Logic



The third type of adder is the carry-lookahead adder. This adder can be done multiple ways. One way is similar to the ripple adder, which we found to be a slower design. The other method is the method utilized for this project. A schematic for this method can be found in figure 8. For this method of the carry-lookahead, two values are calculated for each bit, a carry and a propagate. The generate for each bit is calculated as the bit of A AND the bit of B. The propagate is calculated as the bit of A OR the bit of B. These values of generate, propagate, and c in are used to calculate the next bits of c in. This equation for the fourth carry in is shown in the equation above. The sum is simply just A EXOR B EXOR cin(excluding the final value of cin). This process can be used to calculate an 8 bit number, but the equation for C in 8 would become very long and complicated. To combat this, the 8 bit number was calculated using two different stage of the carry-lookahead adder. Each of these adder would solve for 4 bits. C in 4 of the first four bits would be used as C in 0 of the last four bits. Overall, the adder is easier to build, as compared with the conditional adder, but is more complicated as compared to the ripple adder4. Like the conditional adder, the carry-lookahead adder is faster than the ripple adder. Like the conditional adder, the carry-lookahead adder signal will travel through 12 gates, half the number of gates as the ripple carrier. Like the conditional adder, the final solution is not rippled. All logic schematics included with report (see attached documents).

4 Implementation

The first adder is the ripple adder, shown in appendix B, The test bench for the ripple adder is shown in appendix A. This program was provided, so the only code created by me was the test bench. The ripple adder works by using the generate statement. The generate will from from a value of 0 to one less than the input number of bits. First, a new value of y is calculated (yt). The value is changed based on the mode, or if addition or subtraction will be completed. Next, the next value of the carry is calculated. This is calculated by x, yt, and the previous value of the carry. The final calculation is the sum. This is calculated using the value of x, yt, and the previous value of the carry. With the generate statement, any two values can be added together. The number of bits just need to be specificed. As can be seen with the code, the execution of the ripple adder is simple to complete, and using the generate statement allows for any two binary numbers to be added together.

The second adder is the conditional sum adder, shown in appendix D. THe test bench is shown in appendix C. The programing of the conditional sum adder is more challenging than the ripple adder. The first step was to calculate the value of y. If subtraction was desired, the value of y was changed (using the ones compliment). If addition was desired. the value of y would stay the same. The second step is to calculate all the values from the CS block. To achieve this, two additional modules were created, HA (half adder) and FA (full adder). The solution from the full adder was assigned to a wire named reswc (result with carry). The solution from the half adder was assigned to a wire named resnc (result no carry). The next step is to start combining using 2 input muxes. A new wave (either tempnc or tempwc) were used as the solution to each of the different muxes. The value of the temporary wave and the sum bits of either the resnc or reswc wire were concatonated together into the next stage of either resnc or reswc wires. This next stage of wire would be three bits. The wire before the mux were only two bits. The next stage is to combine these wire again using a two input mux. A second stage of temporary wave is used as the solution of the mux. As in the previous stage, the temporary wave is concatonated with the sum bits of either the resnc or reswc to create 5 bit wires. This whole process is repeated one more time to create 9 bit wires. One wire is the solution if addition was desired, the other solution is if subtraction was desired. To differentiate between the two, one more mux is used. The final answer is a nine bit number. Bit eight is the carry out, and the remaining bits are the sum. One potential was to make this solution either is with the use of the generate statement. The main reason the generate statement was not used was because in order to create some of these wires, I need to extract part of the information from the wire. Using an unpacked array, this was not allowed. A fully packed array was not allowed either. A solution using only a generate is possible using a different method, but would take more time to implement. With this program only eight bit numbers can be added together. A generate statement would allow for any power of two bit numbers to be added together.

The final adder is the carry-lookahead adder, shown in appendix I. The eight

bit module is shown in appendix H, and the test bench is shown in appendix G. The Carry Lookahead calculates part of the answer using a generate statement, and part of it without the generate statement. In the generate statement, the new value of y is determined (the same was as with the other adder methods). The various values of propogate and generate are calculated. After the generate, the next four values of carry are calculated (using the generate and propogate). The sum is then calculated using the x, y, and the first four values of the carry. The fifth carry value is the carry out. With more work, the entire code could be placed within the generate, but since the values to calulcate the carry changed enough, it was easier to calculate without the generate. In order to calculate an eight bit number, the code for the eight bit carry lookahead is used. This code seconds bits 3 to 0 of x and y to the four bit adder. The carry out is used along with bits 7 to 4 to calculate the rest of the solution. In all, the 4 bit adder is used twice. The final answer will be an eight bit value with a final carry out from the 4 bit adder. This process can be used with any length of bit (the number of times the 4 bit adder will be used will change, however).

5 Test Bench Design

Eight seperate testbenches were completed for the three different adders. Four testbenches were for addition, and four were for subtraction. The first test bench was the addition of 01111111 with 0000001. The purpose of this test bench was to ensure that the 1 is able to be passed through. The subtraction was completed of 10000000 minus 01111111 to ensure that the opposite of the previous test bench could occur. The next two testbenches were the addition and sutraction of 11111111 and 00000000. These two testbenches were used to ensure that the 0's would be able to be passed along. The next two test benches were for the addition and subtraction of 01110101 with 01010101. These test benches were used to determine if a repeating pattern will be passed through. The last two test benches were used to see if I could crash the program with a random string of two numbers. These numbers are 01110111 and 00111001.

6 Simulation

Figure 9 shows the simulation results from the ripple adder. As can be seen, the results are expected. One issue with the simulation of the ripple adder is the lag time associated witht he ripple adder. Because of the nature of how the ripple adder works, it takes some time for the signal to ripple through the adder. Due to this lag time, extra time was added for the first value. Also, if the simulation results are analyzed closely, there is a delay between the input values changing and the output values changing. This delay time does not occur with the other adder types.

Figure 10 shows the simulation results from the conditional adder. As can be seen, the results are expected.

Figure 9: Ripple Adder Simulation Results

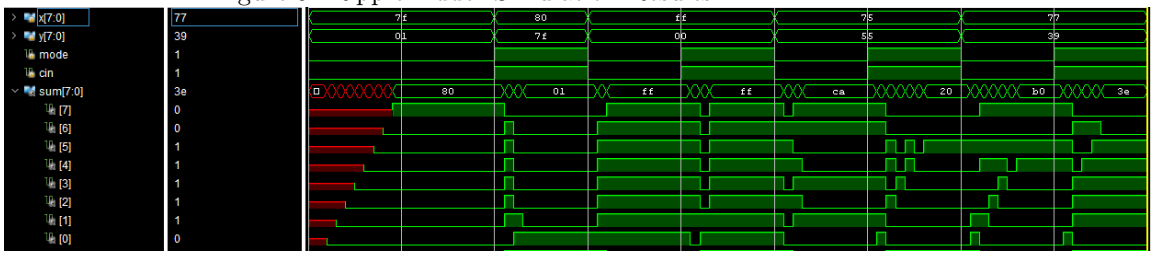


Figure 10: Conditional Adder Simulation Results

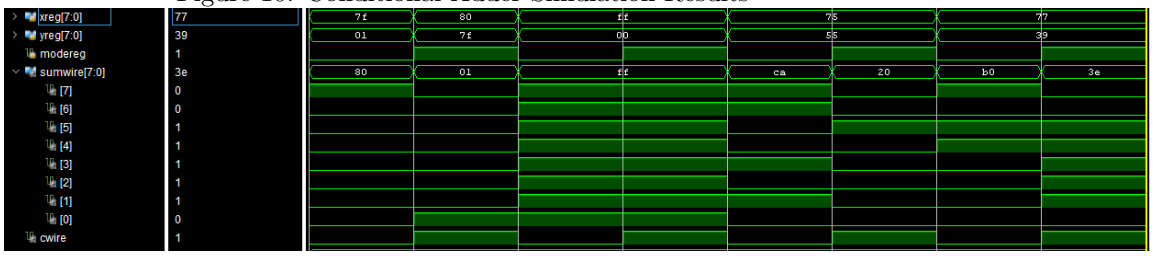
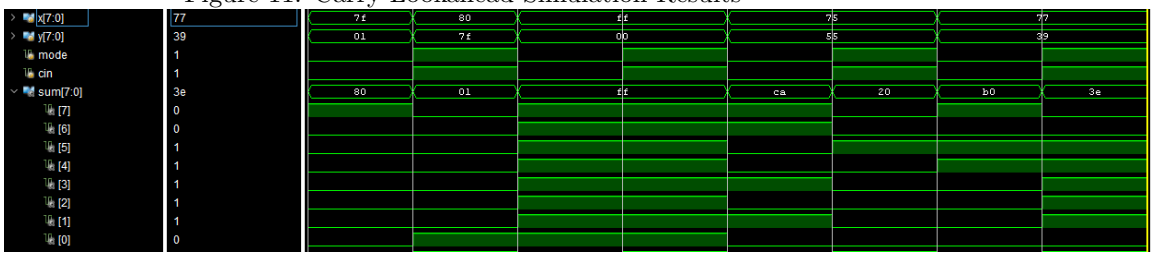


Figure 11 shows the simulation results from the conditional adder. As can be seen, the results are expected.

Figure 11: Carry Lookahead Simulation Results



7 FPGA Realization and Final Verification

To finally implement each of the three different programs, the code was modified (modified code not included in order to keep the report from becoming to long). The first addition was to change x and y from being an input to being a wire. The new input of the system were the sixteen switches. The number x becames switches 0-7 and the number y became the switches 8-15. The next changes was to change the output from sum to LEDs 0-7. The final implementation change was to remove the mode and replace it with the pressing of BTNC. The pressing of BTNC would differentiate between addition and subtraction. If

BTNC was not pressed, addition would be completed. If BTNC was pressed, subtraction would occur. In order to verify that the outputs were correct. The same test benches as simulation were used. Each of the different 8 test benches were implemented on the board and compared with the values obtained from simulation. The values shown on the 8 LEDs were the same as the binary values shown in simulation. In order to ensure that the correct values will be shown , each of the switches must be fully up or down, and the button must be pressed firmly for a few seconds. A debounce circuit was not used in order to ensure that the correct values were obtained from the switches and button.

8 Conclusions

In conclusion, three different adder designs were created and tested for speed and complexity. These three adders are: the ripple adder, the conditional adder, and the carry-lookahead adder. The ripple adder had the advantage of not being complex to design, but was slower in executing. The conditional and carry-lookahead adders were faster in executing, but was more complex to design. While the ripple adder was the easiest to program, the conditional adder and carry-lookahead adder were easier for the program to run. All three adder types were created using Verilog and tested on a Nexys 4 DDR board using 16 switches as the two unput numbers and 1 button for addition or subtraction. The reuslts were displayed using 8 LEDs. Four different test benches were used to verify that the code would produce the desired results. The results came out exactly as expected.

In order to better optimize the programs, the conditional adder would be constructed using a generate statement and the carry-lookahead adder would calculate everything using the generate, not just solving for part of the solution. Two of the most important takeaways from this lab that I learned were the difference between packed and unpacked arrays and the importance of the order of a multidimensional array. One final important takeaway was the balance between getting the perfect code i wanted and the time available. In order to complete every adder using only a generate statement would be possible, but would take much more time to complete.

A Ripple Testing

```
‘timescale 1ns / 1ps
module ripple_test;


    reg [7:0] x;
    reg [7:0] y;
    reg mode;
    reg cin;
    wire [7:0] sum;
    wire cout;

    ripple RA(

        .x(x),
        .y(y),
        .mode(mode),
        .cin(cin),
        .sum(sum),
        .cout(cout)
    );

    initial
    begin
       x <= 127;
       y <= 1;
       mode <= 0;
       cin <= 0;
       #200;
       x <= 128;
       y <= 127;
       mode <= 1;
       cin <= 1;
       #100;
       x <= 255;
       y <= 0;
       mode <= 0;
       cin <= 0;
       #100;
       mode <= 1;
       cin <= 1;
       #100;
       x <= 117;
       y <= 85;
```

```
        mode <= 0;
        cin <= 0;
        #100;
        mode <= 1;
        cin <= 1;
        #100;
        x <= 119;
        y <= 57;
        mode <= 0;
        cin <= 0;
        #100;
        mode <= 1;
        cin <= 1;
        #100;
        $finish;
    end


endmodule
```

B Ripple Code

```
‘timescale 1ns / 1ps
module ripple#(parameter bits=8)(
    input [bits-1:0] x,
    input [bits-1:0] y,
    input mode,
    input cin,
    output [bits-1:0] sum,
    output cout
    );
    wire [bits:0] carry;
    wire [bits-1:0] ty;

    assign carry[0]=cin;
    assign cout=carry[bits];

    generate
    genvar i;
    for (i=0;i<bits;i=i+1) begin:fa
        assign #10 ty[i]= y[i]^mode;
        assign #10 carry[i+1]= x[i]&ty[i] | x[i]&carry[i]
            | ty[i]&carry[i];
        assign #10 sum[i]= x[i]^ty[i]^carry[i];
    end
    endgenerate

endmodule
```

C Conditional Adder Testing

```
`timescale 1ns / 1ps

module Conditional_adder_test;

parameter bits = 8;
reg [bits-1:0] xreg;
reg [bits-1:0] yreg;
reg modereg;
wire [7:0] sumwire;
wire cwire;

Conditional_Adder2#() Conditional_Adder_1(
    .x(xreg),
    .y(yreg),
    .mode(modereg),
    .sum(sumwire),
    .cout(cwire)
    );

initial begin

xreg <= 127;
yreg <= 1;
modereg <= 0;
#100;
xreg <= 128;
yreg <= 127;
modereg <= 1;
#100;
xreg <= 255;
yreg <= 0;
modereg <= 0;
#100;
modereg <= 1;
#100;
xreg <= 117;
yreg <= 85;
modereg <= 0;
#100;
modereg <= 1;
#100;
xreg <= 119;
yreg <= 57;
```

```
modereg <= 0;
#100;
modereg <= 1;
#100;
$finish;

end

endmodule
```

D Conditional Adder Code

```
`timescale 1ns / 1ps

module Conditional_Adder2#(parameter bits=8, levels=4)(
    input [7:0] x,
    input [7:0] y,
    input mode,
    output [7:0] sum,
    output cout
    );

    wire[15:0] HAres;
    wire[15:0] FAres;
    wire [1:0] resnc00;
    wire [1:0] resnc01;
    wire [1:0] resnc02;
    wire [1:0] resnc03;
    wire [1:0] resnc04;
    wire [1:0] resnc05;
    wire [1:0] resnc06;
    wire [1:0] resnc07;
    wire [1:0] reswc00;
    wire [1:0] reswc01;
    wire [1:0] reswc02;
    wire [1:0] reswc03;
    wire [1:0] reswc04;
    wire [1:0] reswc05;
    wire [1:0] reswc06;
    wire [1:0] reswc07;
    wire [1:0] tempnc00;
    wire [1:0] tempnc01;
    wire [1:0] tempnc02;
    wire [1:0] tempnc03;
    wire [1:0] tempwc00;
    wire [1:0] tempwc01;
    wire [1:0] tempwc02;
    wire [1:0] tempwc03;
    wire [2:0] resnc10;
    wire [2:0] resnc11;
    wire [2:0] resnc12;
    wire [2:0] resnc13;
    wire [2:0] reswc10;
    wire [2:0] reswc11;
    wire [2:0] reswc12;
```

```
wire [2:0] reswc13;
wire [2:0] tempnc10;
wire [2:0] tempnc11;
wire [2:0] tempwc10;
wire [2:0] tempwc11;
wire [4:0] resnc20;
wire [4:0] resnc21;
wire [4:0] reswc20;
wire [4:0] reswc21;
wire [4:0] tempnc20;
wire [4:0] tempwc20;
wire [8:0] resnc30;
wire [8:0] reswc30;

wire [7:0] yt;
assign yt[0] = y[0]^mode;
assign yt[1] = y[1]^mode;
assign yt[2] = y[2]^mode;
assign yt[3] = y[3]^mode;
assign yt[4] = y[4]^mode;
assign yt[5] = y[5]^mode;
assign yt[6] = y[6]^mode;
assign yt[7] = y[7]^mode;

HA1 sumnocarry(.x(x), .y(yt), .r(HAres));
FA1 sumwithcarry(.x(x), .y(yt),.cin(1),.r(FAres));

assign resnc00 = HAres[1:0];
assign resnc01 = HAres[3:2];
assign resnc02 = HAres[5:4];
assign resnc03 = HAres[7:6];
assign resnc04 = HAres[9:8];
assign resnc05 = HAres[11:10];
assign resnc06 = HAres[13:12];
assign resnc07 = HAres[15:14];
assign reswc00 = FAres[1:0];
assign reswc01 = FAres[3:2];
assign reswc02 = FAres[5:4];
assign reswc03 = FAres[7:6];
assign reswc04 = FAres[9:8];
assign reswc05 = FAres[11:10];
assign reswc06 = FAres[13:12];
assign reswc07 = FAres[15:14];

assign tempnc00 = resnc00[1]?reswc01:resnc01;
assign resnc10 = {tempnc00,resnc00[0]};
```

```
assign tempwc00 = reswc00[1]?reswc01:resnc01;
assign reswc10 = {tempwc00,reswc00[0]};
assign tempnc01 = resnc02[1]?reswc03:resnc03;
assign resnc11 = {tempnc01,resnc02[0]};
assign tempwc01 = reswc02[1]?reswc03:resnc03;
assign reswc11 = {tempwc01,reswc02[0]};
assign tempnc02 = resnc04[1]?reswc05:resnc05;
assign resnc12 = {tempnc02,resnc04[0]};
assign tempwc02 = reswc04[1]?reswc05:resnc05;
assign reswc12 = {tempwc02,reswc04[0]};
assign tempnc03 = resnc06[1]?reswc07:resnc07;
assign resnc13 = {tempnc03,resnc06[0]};
assign tempwc03 = reswc06[1]?reswc07:resnc07;
assign reswc13 = {tempwc03,reswc06[0]};
assign tempnc10 = resnc10[2]?reswc11:resnc11;
assign resnc20 = {tempnc10,resnc10[1:0]};
assign tempwc10 = reswc10[2]?reswc11:resnc11;
assign reswc20 = {tempwc10,reswc10[1:0]};
assign tempnc11 = resnc12[2]?reswc13:resnc13;
assign resnc21 = {tempnc11,resnc12[1:0]};
assign tempwc11 = reswc12[2]?reswc13:resnc13;
assign reswc21 = {tempwc11,reswc12[1:0]};
assign tempnc20 = resnc20[4]?reswc21:resnc21;
assign resnc30 = {tempnc20,resnc20[3:0]};
assign tempwc20 = reswc20[4]?reswc21:resnc21;
assign reswc30 = {tempwc20,reswc20[3:0]};
assign sum[7:0] = mode?reswc30[7:0]:resnc30[7:0];
assign cout = mode?reswc30[8]:resnc30[8];


endmodule
```

E Full Adderr

```
`timescale 1ns / 1ps
module FA1(
    input [7:0]x,
    input [7:0]y,
    output [15:0]r,
    input cin
    );
    assign r[0]=  ((~cin) & x[0] &(~y[0])) | ((~cin) &(~x
        [0])&y[0] ) | (cin &(~x[0]) &(~y[0])) | (cin & x
        [0] & y[0])      ;
    assign r[1] = ((~cin) & x[0] &y[0]) | ((cin) &(~x[0])
       &y[0] ) | (cin &(x[0]) &(~y[0])) | (cin & x[0] & y
        [0])  ;
    assign r[2]=  ((~cin) & x[1] &(~y[1])) | ((~cin) &(~x
        [1])&y[1] ) | (cin &(~x[1]) &(~y[1])) | (cin & x
        [1] & y[1])      ;
    assign r[3] = ((~cin) & x[1] &y[1]) | ((cin) &(~x[1])
       &y[1] ) | (cin &(x[1]) &(~y[1])) | (cin & x[1] & y
        [1])  ;
    assign r[4]=  ((~cin) & x[2] &(~y[2])) | ((~cin) &(~x
        [2])&y[2] ) | (cin &(~x[2]) &(~y[2])) | (cin & x
        [2] & y[2])      ;
    assign r[5] = ((~cin) & x[2] &y[2]) | ((cin) &(~x[2])
       &y[2] ) | (cin &(x[2]) &(~y[2])) | (cin & x[2] & y
        [2])  ;
    assign r[6]=  ((~cin) & x[3] &(~y[3])) | ((~cin) &(~x
        [3])&y[3] ) | (cin &(~x[3]) &(~y[3])) | (cin & x
        [3] & y[3])      ;
    assign r[7] = ((~cin) & x[3] &y[3]) | ((cin) &(~x[3])
       &y[3] ) | (cin &(x[3]) &(~y[3])) | (cin & x[3] & y
        [3])  ;
    assign r[8]=  ((~cin) & x[4] &(~y[4])) | ((~cin) &(~x
        [4])&y[4] ) | (cin &(~x[4]) &(~y[4])) | (cin & x
        [4] & y[4])      ;
    assign r[9] = ((~cin) & x[4] &y[4]) | ((cin) &(~x[4])
       &y[4] ) | (cin &(x[4]) &(~y[4])) | (cin & x[4] & y
        [4])  ;
    assign r[10]=  ((~cin) & x[5] &(~y[5])) | ((~cin) &(~
        x[5])&y[5] ) | (cin &(~x[5]) &(~y[5])) | (cin & x
        [5] & y[5])      ;
    assign r[11] = ((~cin) & x[5] &y[5]) | ((cin) &(~x
        [5])&y[5] ) | (cin &(x[5]) &(~y[5])) | (cin & x[5]
         & y[5])  ;
```

```
assign r[12]=  ((~cin) & x[6] &(~y[6])) | ((~cin) &(~
    x[6])&y[6] ) | (cin &(~x[6]) &(~y[6])) | (cin & x
    [6] & y[6])      ;
assign r[13] = ((~cin) & x[6] &y[6]) | ((cin) &(~x
    [6])&y[6] ) | (cin &(x[6]) &(~y[6])) | (cin & x[6]
     & y[6])  ;
assign r[14]=  ((~cin) & x[7] &(~y[7])) | ((~cin) &(~
    x[7])&y[7] ) | (cin &(~x[7]) &(~y[7])) | (cin & x
    [7] & y[7])      ;
assign r[15] = ((~cin) & x[7] &y[7]) | ((cin) &(~x
    [7])&y[7] ) | (cin &(x[7]) &(~y[7])) | (cin & x[7]
     & y[7])  ;

endmodule
```

F Half Adder

```
`timescale 1ns / 1ps

module HA1(
    input [7:0]x,
    input [7:0]y,
    output [15:0]r
    );
    assign r[0] = x[0]^y[0];
    assign r[1] = x[0]&y[0];
    assign r[2] = x[1]^y[1];
    assign r[3] = x[1]&y[1];
    assign r[4] = x[2]^y[2];
    assign r[5] = x[2]&y[2];
    assign r[6] = x[3]^y[3];
    assign r[7] = x[3]&y[3];
    assign r[8] = x[4]^y[4];
    assign r[9] = x[4]&y[4];
    assign r[10] = x[5]^y[5];
    assign r[11] = x[5]&y[5];
    assign r[12] = x[6]^y[6];
    assign r[13] = x[6]&y[6];
    assign r[14] = x[7]^y[7];
    assign r[15] = x[7]&y[7];

endmodule
```

G Carry Lookahead Test

```
‘timescale 1ns / 1ps
module Carry_Lookahead_Test;

    reg [7:0] x;
    reg [7:0] y;
    reg mode;
    reg cin;
    wire [7:0] sum;
    wire cout;

    Carry_Lookahead_8bit CLA(

        .x(x),
        .y(y),
        .mode(mode),
        .cin(cin),
        .sum(sum),
        .cout(cout)
    );

    initial
    begin
    x <= 127;
    y <= 1;
    mode <= 0;
    cin <= 0;
    #100;
    x <= 128;
    y <= 127;
    mode <= 1;
    cin <= 1;
    #100;
    x <= 255;
    y <= 0;
    mode <= 0;
    cin <= 0;
    #100;
    mode <= 1;
    cin <= 1;
    #100;
    x <= 117;
    y <= 85;
    mode <= 0;
```

```
        cin <= 0;
        #100;
        mode <= 1;
        cin <= 1;
        #100;
        x <= 119;
        y <= 57;
        mode <= 0;
        cin <= 0;
        #100;
        mode <= 1;
        cin <= 1;
        #100;
        $finish;

        end

endmodule
```

H Carry Lookahead 8 bit

```
‘timescale 1ns / 1ps
module Carry_Lookahead_8bit(
    input [7:0] x,
    input [7:0] y,
    input mode,
    input cin,
    output [7:0] sum,
    output cout

    );

    wire cout1;

Carry_Lookahead_4bit_block#(4) CarLook1 (
    .x(x[3:0]),
    .y(y[3:0]),
    .mode(mode),
    .cin(cin),
    .sum(sum[3:0]),
    .cout(cout1)
    );
Carry_Lookahead_4bit_block#(4) CarLook2 (
    .x(x[7:4]),
    .y(y[7:4]),
    .mode(mode),
    .cin(cout1),
    .sum(sum[7:4]),
    .cout(cout)
    );

endmodule
```

I Carry Lookahead 4 bit

```
`timescale 1ns / 1ps
module Carry_Lookahead_4bit_block(

input [3:0] x,
input [3:0] y,
input mode,
input cin,
output [3:0] sum,
output cout
    );

wire [3:0] yt;
wire [3:0] gen;
wire [3:0] prop;
wire [4:0] car;
//wire [3:0] answer;

assign car[0] = cin;
assign cout = car[4];

generate
genvar i;
    for(i=0;i<4;i=i+1) begin
        assign yt[i] = y[i]^mode;
        assign prop[i] = x[i]|yt[i];
        assign gen[i] = x[i]&yt[i];
    end
endgenerate

assign car[1] = gen[0]|(prop[0]&car[0]);
assign car[2] = gen[1]|(prop[1]&gen[0])|(prop[1]&prop[0]&
    car[0]);
assign car[3] = gen[2]|(prop[2]&car[1])|(prop[2]&prop[1]&
    gen[0])|(prop[2]&prop[1]&prop[0]&car[0]);
assign car[4] = gen[3]|(prop[3]&gen[2])|(prop[3]&prop[2]&
    gen[1])|(prop[3]&prop[2]&prop[1]&gen[0])|(prop[3]&prop
    [2]&prop[1]&prop[0]&car[0]);
assign sum[3:0] = x[3:0]^yt[3:0]^car[3:0];

endmodule
```