Simulation of Booth Multiplier with Verilog-XL

November 30, 2011

Robert D'Angelo & Scott Smith

Tufts University

Electrical and Computer Engineering

EE-103 Lab 3: Part II

Professor:

Dr. Valencia Joyner Koomson

Abstract

In this lab an 8x8 modified booth multiplier is designed and tested at the gate level using the AMS simulator in Cadence Design System. Using logic gate primitives constructed in Part I of this lab exercise, the following blocks were simulated to complete the booth multiplier architecture: half-adder, full-adder, 4-bit carry lookahead adder (CLA), 12-bit CLA, CLA logic, booth encoder, and booth decoder. Each of these blocks along with the complete architecture of the multiplier were simulated successfully. There is an average delay of ??? per multiply operation.

Contents

List of Figures

List of Tables

1 Booth Multiplier Overview

1.1 Booth's Algorithm

Traditional hardware multiplication is performed in the same way multiplication is done by hand: partial products are computed, shifted appropriately, and summed. This algorithm can be slow if there are many partial products (i.e. many bits) because the output must wait until each sum is performed. Booth's algorithm cuts the number of required partial products in half. This increases the speed by reducing the total number of partial product sums that must take place.

The algorithm exploits the fact that multiplication by a sequence of 1's can be computed simply with inversions and shifts, simpler operations than adding. This algorithm first encodes the start, middle, end, or absence of a sequence of 1's in the multiplier term from groupings of three bits, each with an overlapping bit from the previous grouping. These encodings are then used to compute the partial products from the multiplicand by either multiplying it by 1 (i.e. no change), multiplying it by 2 (shift left one bit), or multiplying it by -1 (2's complement). The encodings are shown in Table 1. These partial products are shifted by two bits for each partial product after the first. The product is equal to the sum of these terms.

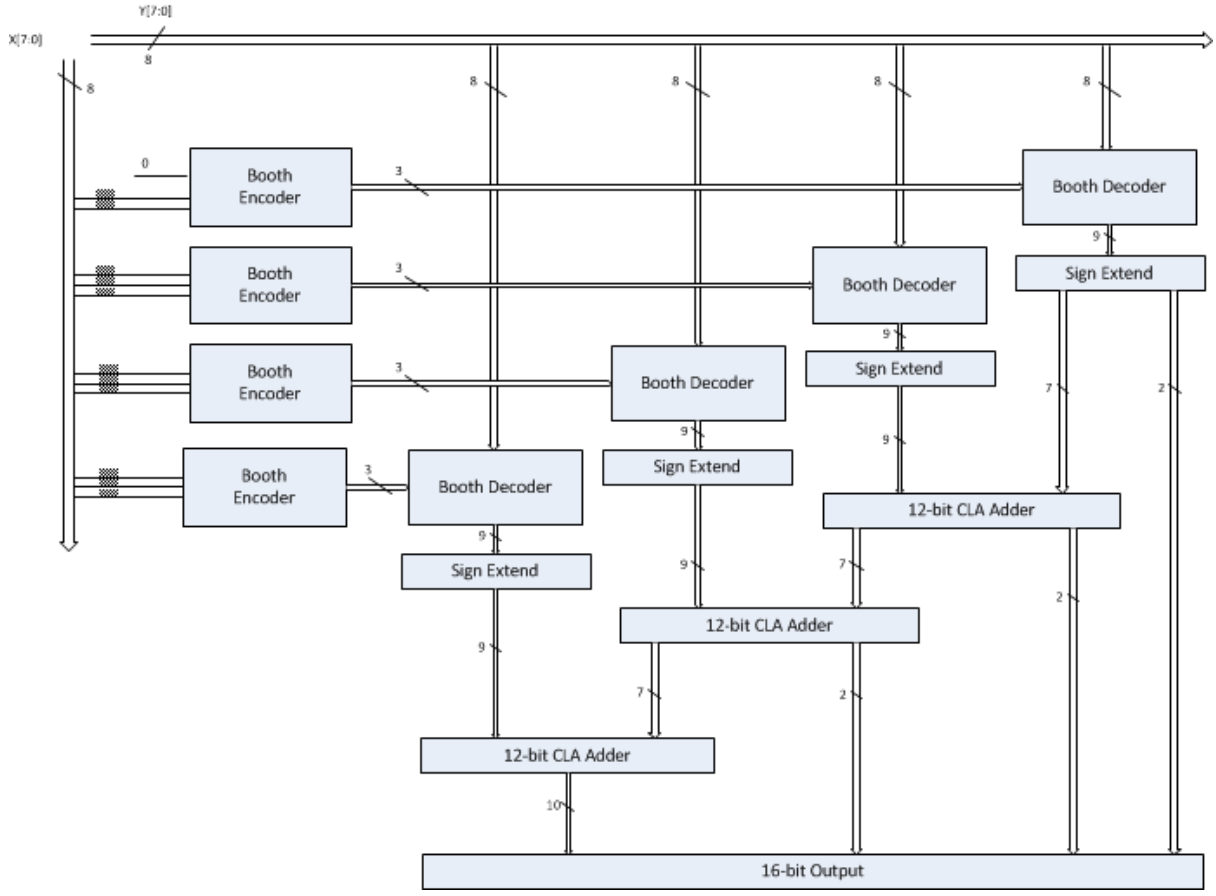
Table 1: Booth Encoding

Grouping	Partial Product
000	0*Multiplicand
001	1*Multiplicand
010	1*Multiplicand
011	2*Multiplicand
100	-2*Multiplicand
101	-1*Multiplicand
110	-1*Multiplicand
111	0*Multiplicand

1.2 Architecture

Figure 1 below shows a block diagram of the proposed Booth multiplier implementation. This circuit takes in two 8-bit binary numbers and outputs the 16-bit product. The multiplier, X[7:0], is divided into four groupings: 0, X0, X1; X1, X2, X3; X3, X4, X5; X5, X6, X7. Each of these groupings is passed into a Booth encoder, which outputs bits corresponding to the operations described in Table 1 (x0, x1, x2, x-1). Each group of these selection bits are sent to a Booth decoder block, which outputs the appropriate partial product term based on the selected operation. These partial products are then sign extended so that sign bits are taken into account during the summing. Finally, the canonical shift and add multiplication is implemented using 12-bit carry lookahead adders (CLA). The first two bits of each partial product are passed directly to the output to account for the shifting. A standard array multiplier would typically require 8 partial products, and thus 8 adders. However, this implementation reduces the number of partial products to only four, significantly improving speed. Furthermore, the CLA provides another speed boost to the system.

Figure 1: Modified Booth Multiplier Architecture



2 Design and Simulation

2.1 Booth Encoder

Single Booth Encoder

Table 2 shows the truth table for a Booth encoder. The encoder takes inputs x_{2i+1}, x_{2i}, and $x_{2i-1}$from the multiplier bus and produces a 1 or a 0 for each operation: single, double, negative. Figure 2 shows the schematic that implements Table 2. Figure 3 shows a symbol view of the encoder. This block was simulated using the Analog Mixed Signal (AMS) simulator. Figure 4 shows the simulation results. Based on the simulation, this block has a propogation delay of ???. The Verilog test bench code, which sweeps all test cases can be seen in Program Listing 1.

Table 2: Booth Encoder Truth Table

x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	Single	Double	Negate
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	2Y	0	1	0
1	0	0	-2Y	0	1	1
1	0	1	-Y	1	0	1
1	1	0	-Y	1	0	1
1	1	1	-0 (=0)	0	0	1

Figure 2: Booth Encoder Schematic

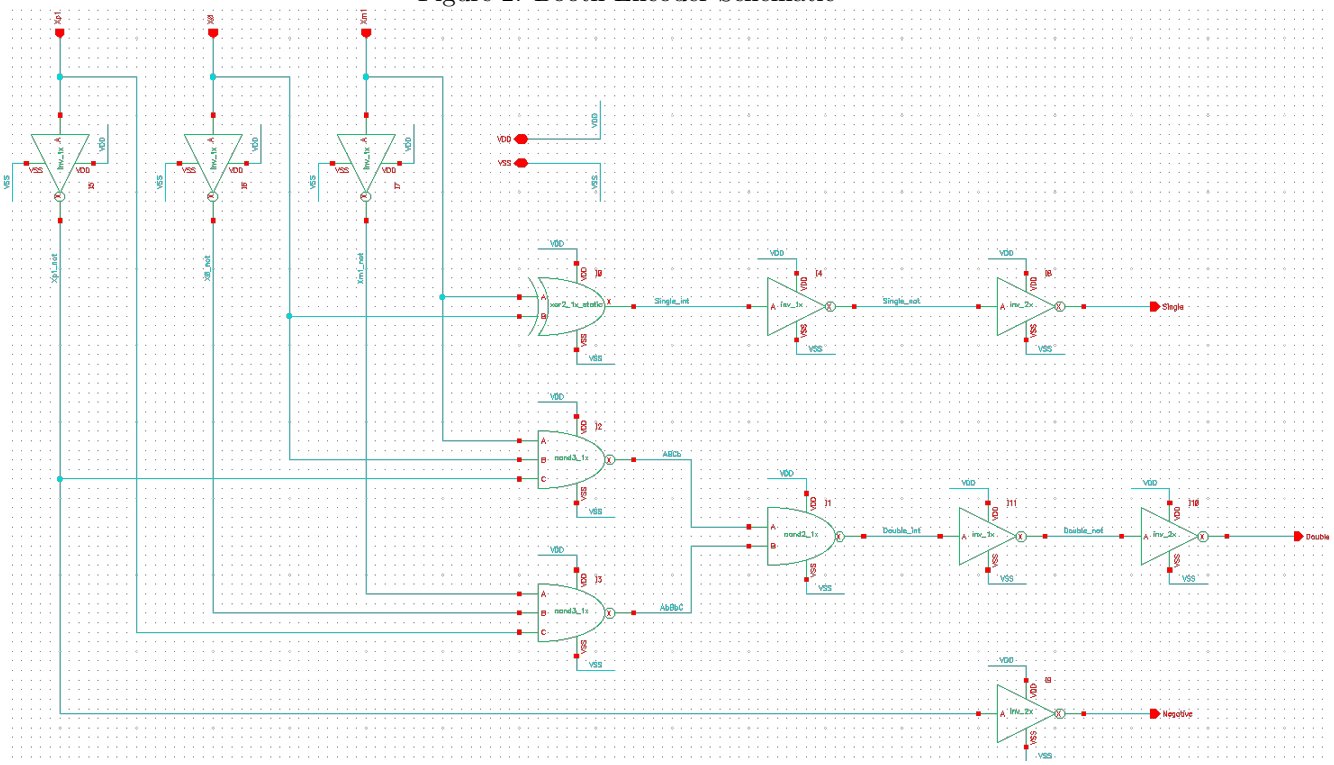


Figure 3: Booth Encoder Symbol

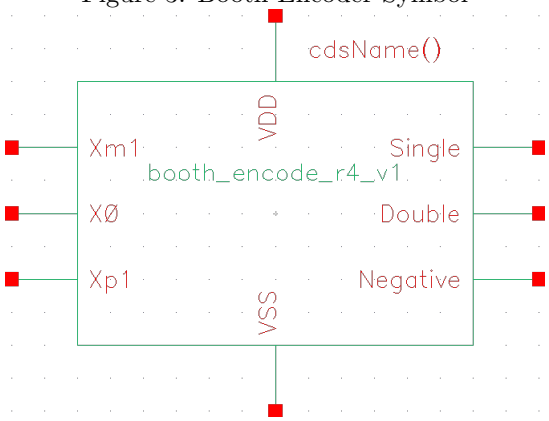


Figure 4: Booth Encoder Test Bench Schematic

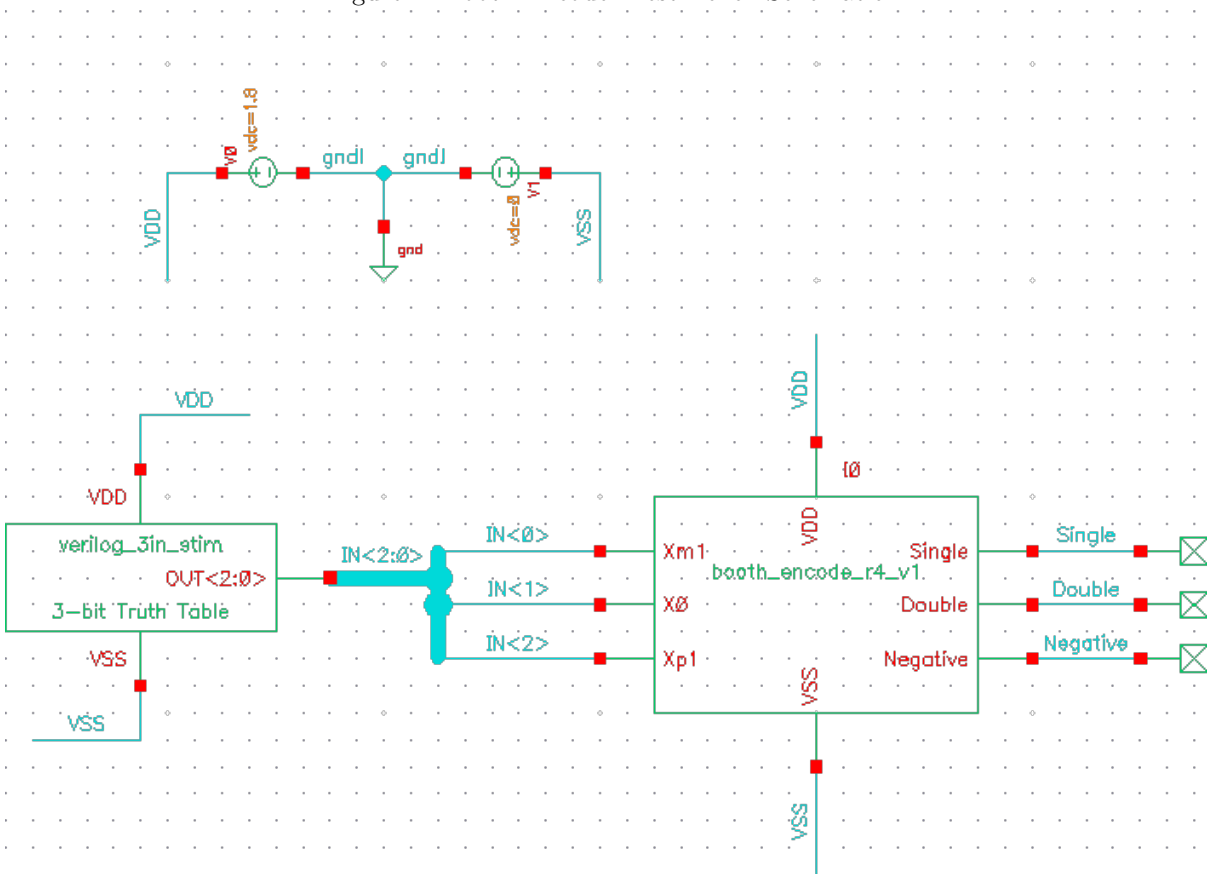
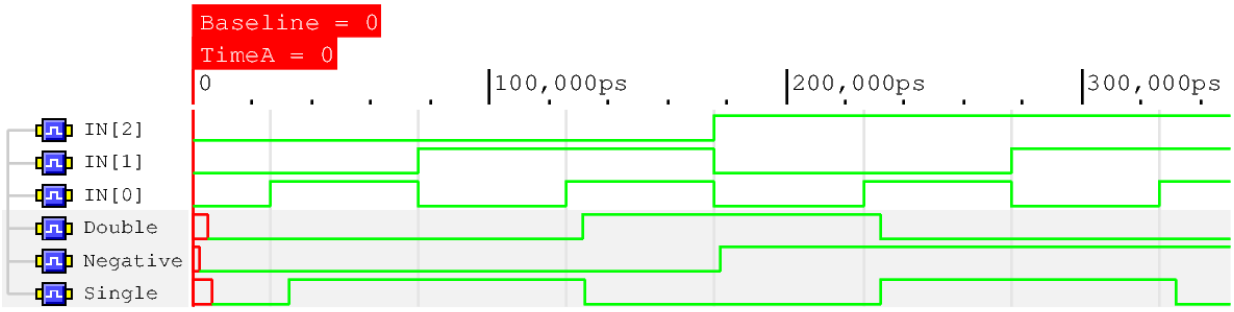


Figure 5: Booth Encoder AMS Simulation




```
//Verilog HDL for "lab3_sims", "verilog_3in_stim" "functional"

`timescale 1ns/1ps
module verilog_3in_stim ( inout wire VDD,
                          inout wire VSS,
                          output reg [2:0] OUT );
  assign VDD = 1'b1;
  assign VSS = 1'b0;
  reg clock;
  initial begin
    clock = 1'b0;
    OUT = 3'b000;
  end
  always begin
    #25.0 clock = ~clock;
  end
  always @ (posedge clock) begin
    OUT = #1.0 OUT + 3'b001;
  end
  always @ (negedge clock) begin
    if( OUT == 3'b111 )
      #50 $finish;
  end
endmodule
```

Listing 1: Booth Encoder Verilog Stimulus

Complete Booth Encoder

Four single bit encoders are combined to produce a complete Booth encoder as shown in Figure 6. This block was generated to simplify the final schematic. Figure 7 shows a symbol view of this block. This block was simulated using AMS. Figure 8 shows the test bench schematic. The Verilog test bench block with all test cases can be viewed in Program Listing 2. The simulation results are displayed in Figure 9.

Figure 6: 4-bit Booth Encoder Schematic

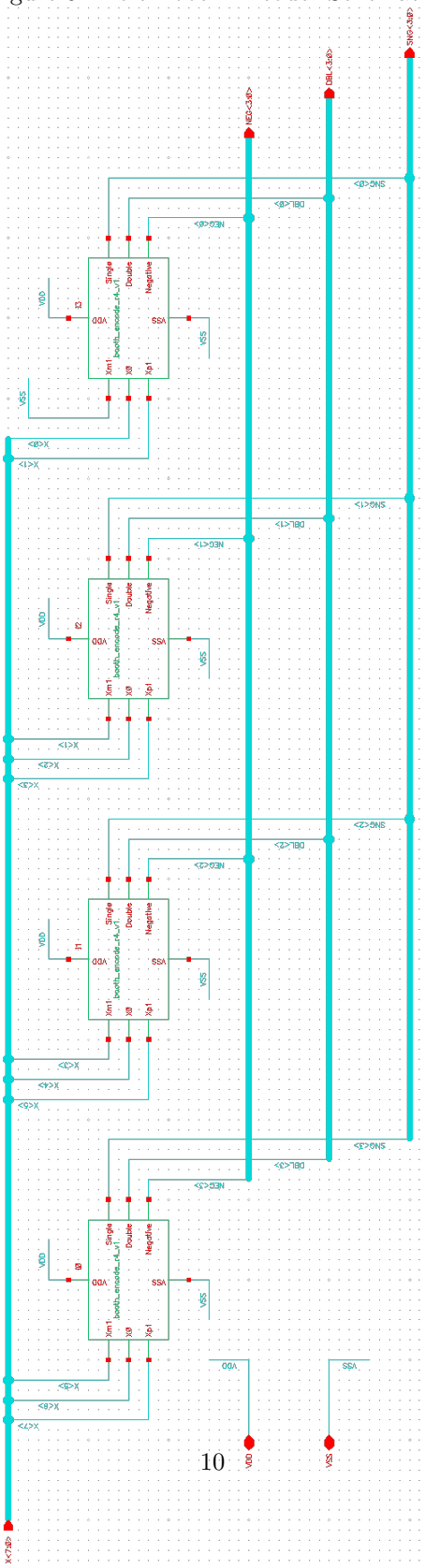


Figure 7: 4-bit Booth Encoder Symbol

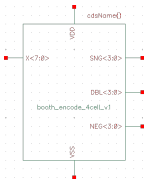
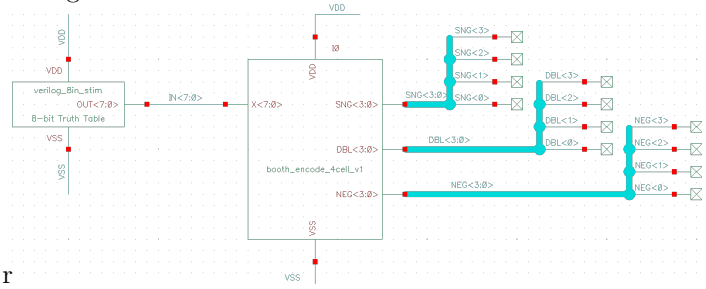


Figure 8: 4-bit Booth Encoder Test Bench Schematic



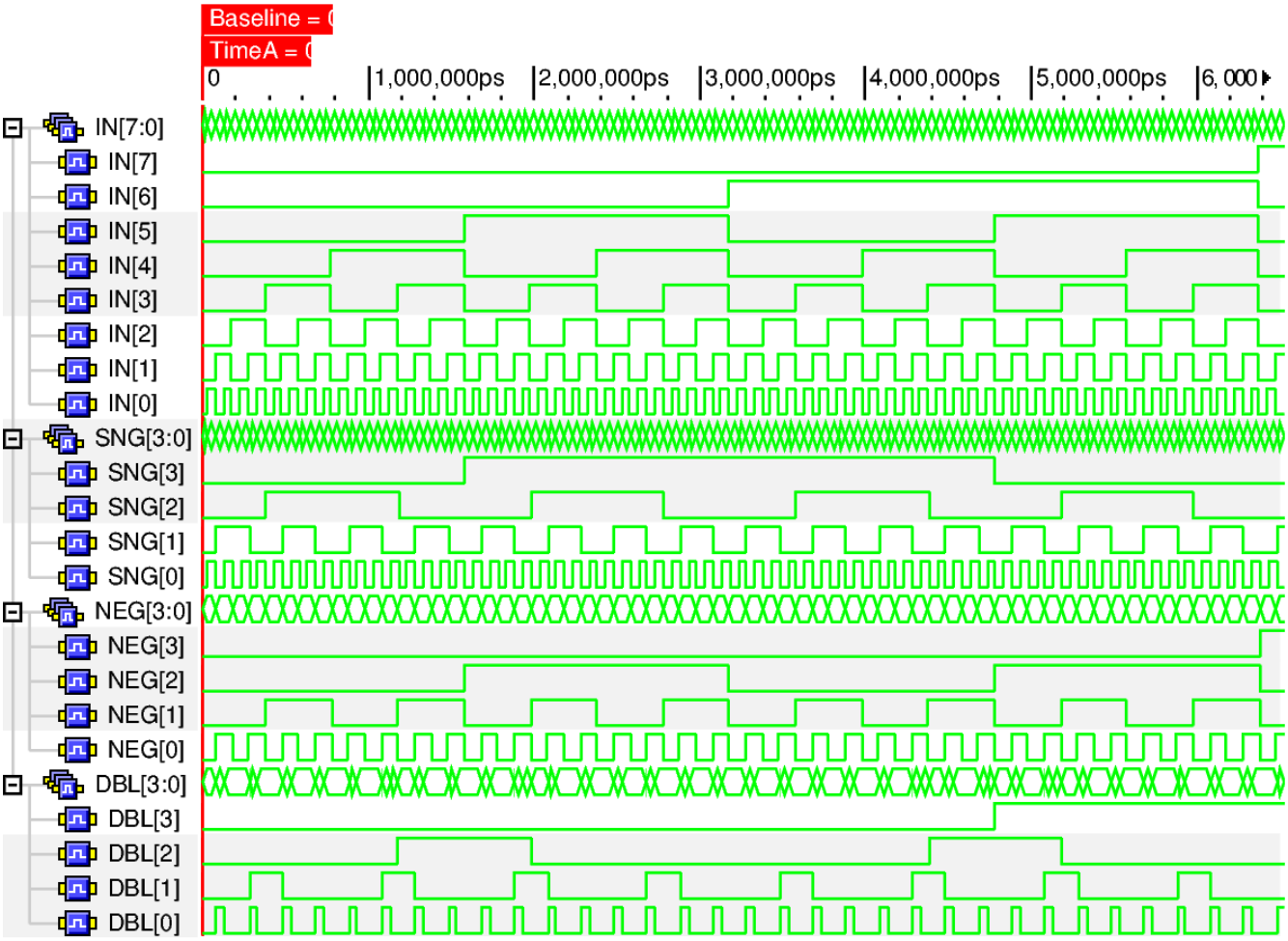
r

```
//Verilog HDL for "lab3_sims", "verilog_8in_stim" "functional"

`timescale 1ns/1ps
module verilog_8in_stim ( inout wire VDD,
                          inout wire VSS,
                          output reg [7:0] OUT );
  assign VDD = 1'b1;
  assign VSS = 1'b0;
  reg clock;
  initial begin
    clock = 1'b0;
    OUT = 8'h00;
  end
  always begin
    #25.0 clock = ~clock;
  end
  always @ (posedge clock) begin
    OUT = #1.0 OUT + 8'h01;
  end
  always @ (negedge clock) begin
    if( OUT == 8'hFF )
      $finish;
  end
endmodule
```

Listing 2: Booth Encoder Verilog Stimulus

Figure 9: 4-bit Booth Encoder AMS Simulation



2.2 Booth Decoder

Single Bit Booth Decoder

Table 3 shows a truth table of the decoder block for generating a single bit of a partial product with inputs from the encoder output bits and two of the multiplicand bits. This truth table does not show the partial product if it is negated. This function is accomplished by simply performing the XOR of the PPij in the table with the NEG output of the Booth encoder so that if NEG is asserted, the output is invertered; however, if NEG is low, the output is unchanged. From Table 3 it can be seen that if SNG is asserted Yj is passed to the output indicating multiply by unity. If DBL is asserted, Yj-1 is passed to the output, indicating a shift by one bit to the left. If SNG and DBL are both zero, the output is zero. SNG and DBL cannot both be asserted, so these cases are not shown. Figure 10 shows a gate level implementation of the single bit Booth decoder, and Figure 11 shows a symbol view. This block has a worst case delay of about 7ns.

Table 3: Booth Decoder Truth Table (NEG not included)

SNG	Yj	DBL	Yj-1	PPij (pre-xor)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	1	0	0	1
1	1	0	1	1

Figure 10: Booth Decoder Schematic

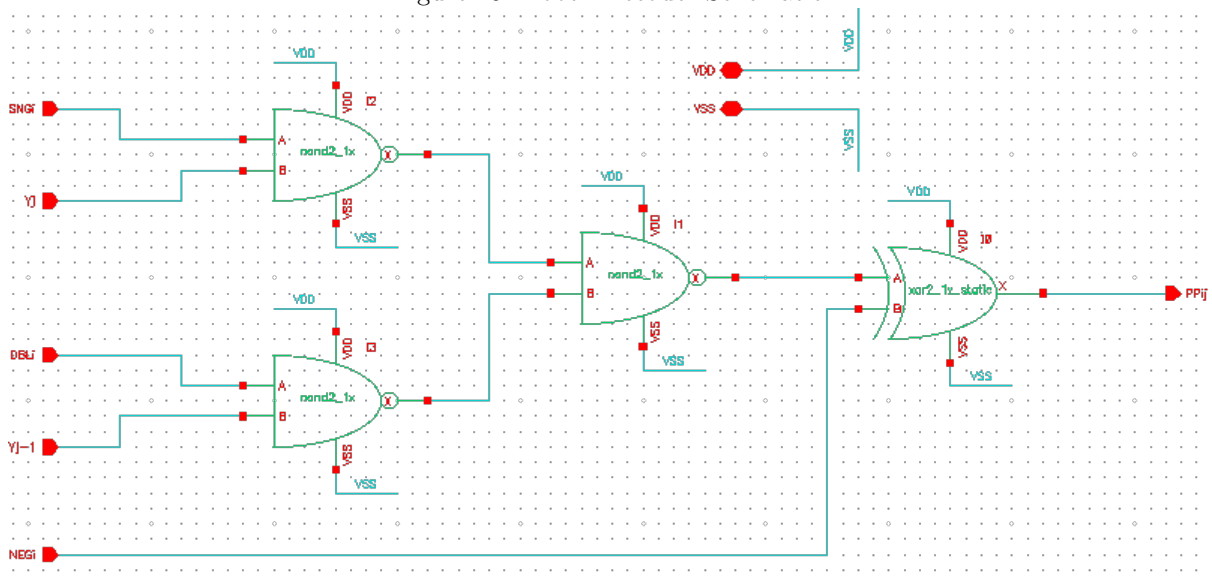
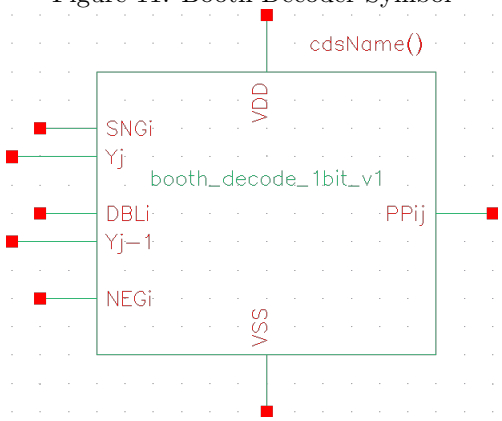


Figure 11: Booth Decoder Symbol



Half Adder

The single bit decoder blocks invert the partial product bits when NEG is asserted; however, one must be added to the least significant bit (LSB) so that 2's complement convention is followed. A cascade of half-adders at the output (shown in Figure XX) implements this sum. Figure 9 shows a gate level view of the half adder. This circuit produces a sum bit from the XOR of inputs A and B and a carry bit from the function A AND B. The NAND and Inverter primitives are used to create the AND function. Figure 10 shows the symbol view, Figure 11 shows a functional simulation, and Figure 12 shows the Verilog stimulus code.

Figure 12: Half Adder Schematic

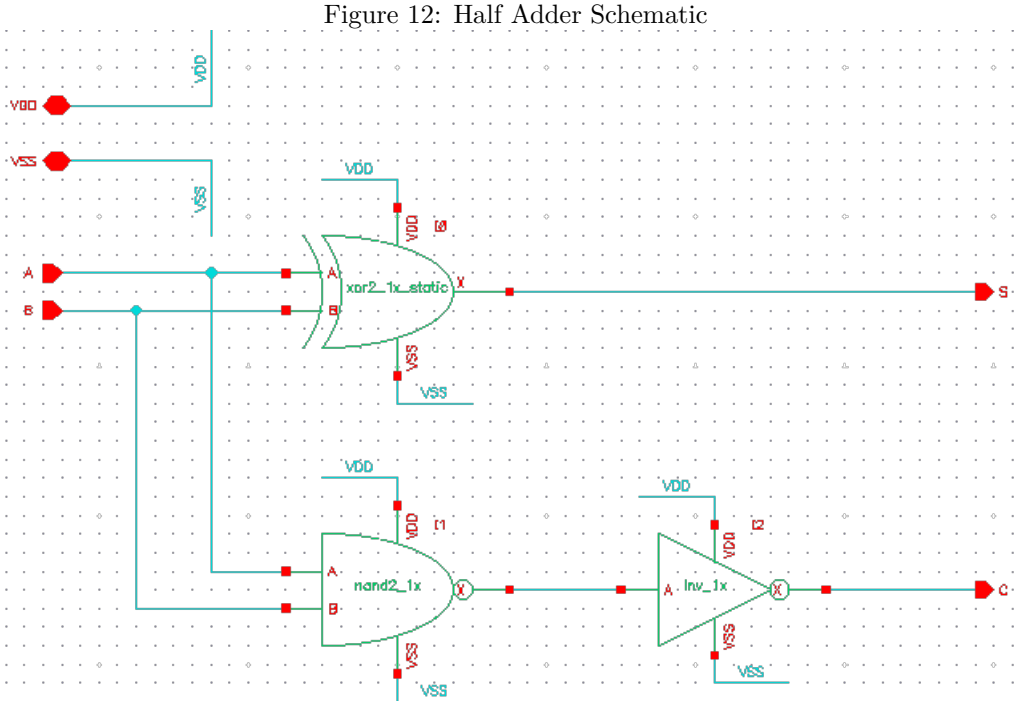


Figure 13: Half Adder Symbol

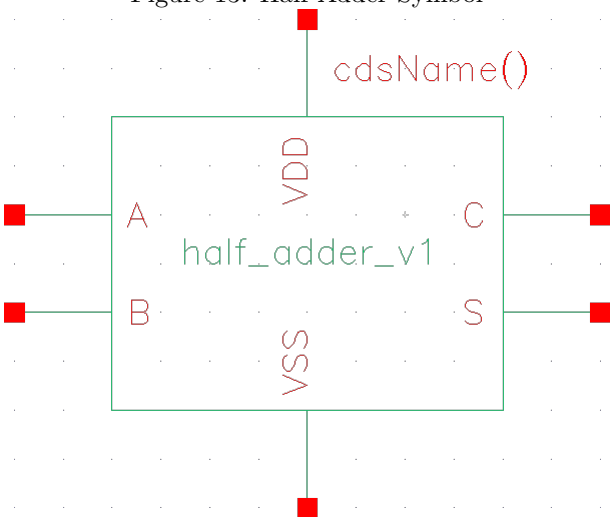


Figure 14: Half Adder Verilog X Simulation


```
// Verilog stimulus file.
// Please do not create a module in this file.

initial
begin

     A = 1'b0; B = 1'b0;
#10  A = 1'b0; B = 1'b1;
#10  A = 1'b1; B = 1'b0;
#10  A = 1'b1; B = 1'b1;
#10  $finish;
end
```

Listing 3: Booth Encoder Verilog Stimulus

Eight-bit Decoder

In order to produce a complete partial product, an 8-bit Booth decoder block was implemented. This decoder consists of nine single bit Booth decoders, each of which sends its partial product bit to a half adder. Each half adder sums the current partial product bit with the carry from the previous half adder. The first half adder sums the NEG bit with the first partial product bit, PP<0>. This step completes the 2's complement representation of the partial product by adding '1' to the LSB if NEG is asserted. In the first decoder, the Y_{j-1} bit is tied to VSS ('0'). This connection ensures that when DBL is asserted, a zero is shifted in.

Note that this schematic actually produces a 9-bit output. The 9th bit is produced by connecting Y_{j-1} of the 9th decoder to Y<7>, the MSB of the multiplicand. If a shift is occuring, the 9th bit, PP<8>, will be correctly replaced with the 8th bit, Y<7>. If no shift occurs, PP<8> will become a sign extension bit that is simply equal to the MSB, i.e. the sign bit, Y<7>.

Figure 15 shows a symbol view of the 8-bit decoder, and Figure 16 shows a gate level schematic. This circuit was simulated using AMS. Figure 17 shows the test bench schematic used in the simulation, and Figure 18 shows the simulation results. The verilog code of the test bench block is shown in Listing 4. All test cases simulated successfully. This block has a worst case delay of about 33ns.

Figure 15: 8-bit Booth Decoder Symbol

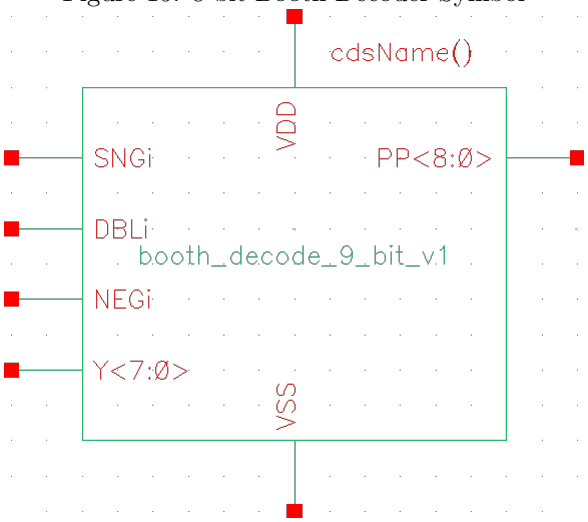


Figure 16: 8-bit Booth Decoder Schematic

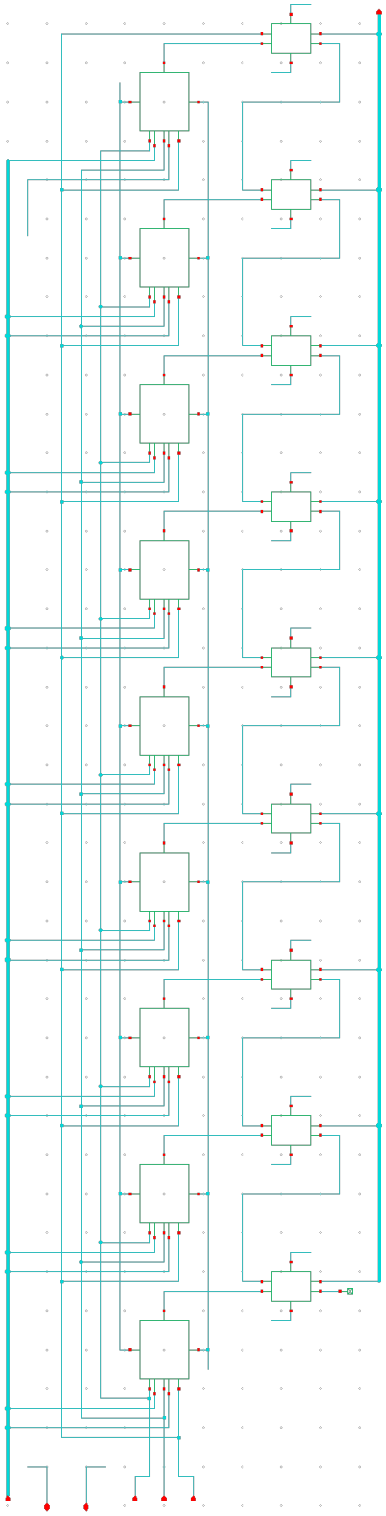


Figure 17: 8-bit Booth Decoder Test Bench Schematic

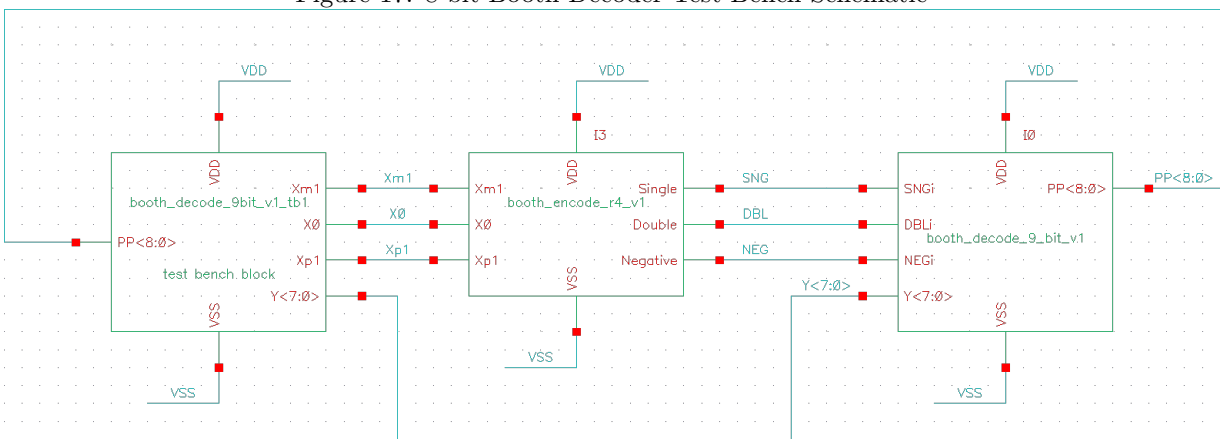
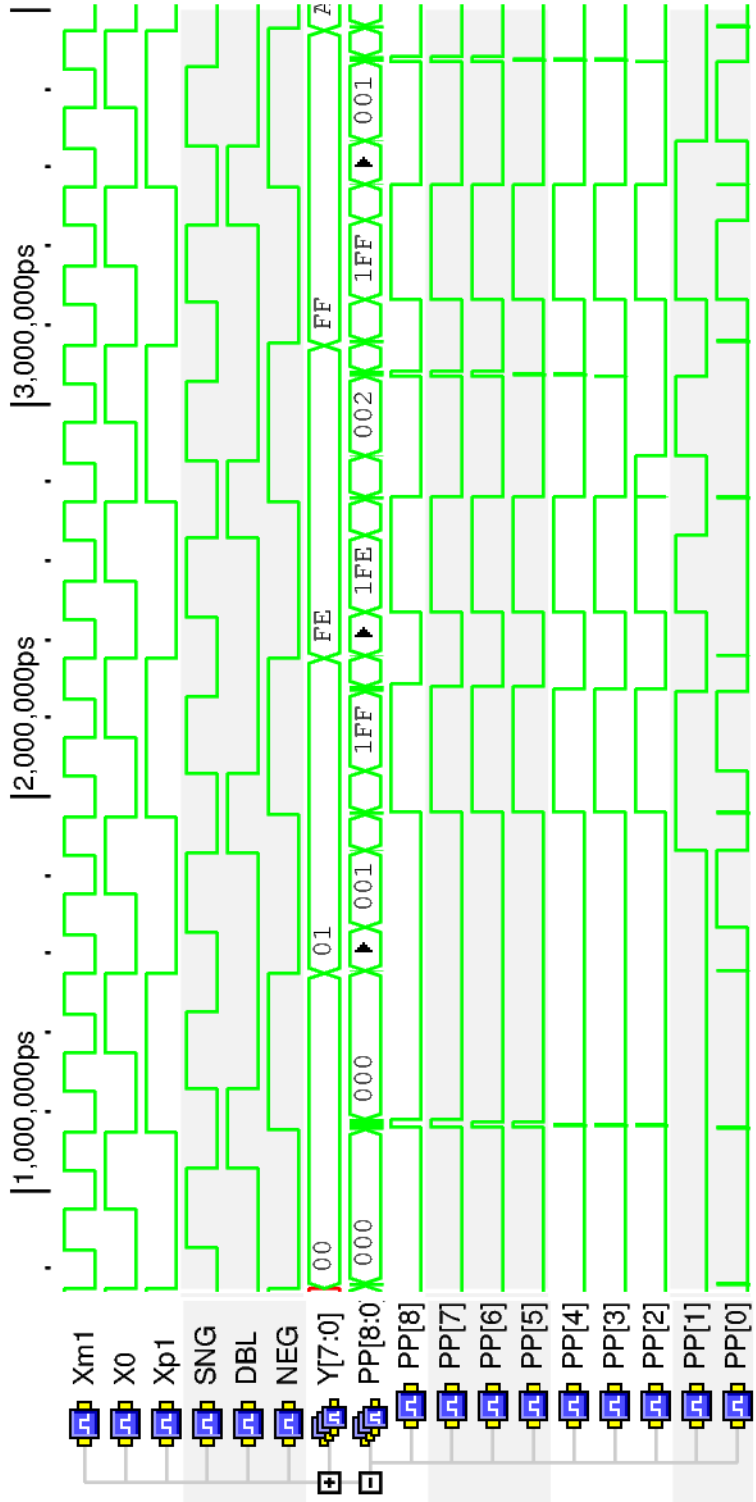


Figure 18: 8-bit Booth Decoder AMS Simulation



```
//Verilog HDL for "lab3_sims", "booth_decode_9bit_v1_tb1" "functional"
// Testbench for booth_decode_9bit_v1_tb1

module booth_decode_9bit_v1_tb1 ( output wire Xm1,
                                  output wire X0,
                                  output wire Xp1,
                                  output reg [7:0] Y,
                                  input wire [8:0] PP,
                                  inout wire VDD,
                                  inout wire VSS );

  parameter NUM0 = 8'b00000000;
  parameter NUM1 = 8'b00000001;
  parameter NUM2 = 8'b11111110;
  parameter NUM3 = 8'b11111111;
  parameter NUM4 = 8'b10101010;
  parameter NUM5 = 8'b01010101;
  parameter NUM6 = 8'b10000000;
  parameter NUM7 = 8'b11001100;

  reg clk;
  reg [2:0] X;
  reg [3:0] sel;

  assign VDD = 1'b1;
  assign VSS = 1'b0;

  assign Xm1 = X[0];
  assign X0  = X[1];
  assign Xp1 = X[2];

  // Initialize variables.
  initial begin
    clk = 1'b0;
    X = 3'b000;
    sel = 4'h0;
  end
  // Clock for advancing through the test cases.
  always begin
    #50.0 clk = ~clk;
  end
  // Counter for X to generate all possibilities.
  always @ (posedge clk) begin
    X = X + 3'b001;
  end
  // 'sel' variable to iterate through the case statement below.
  always @ (posedge clk) begin
```

```
    if( X == 3'b000 ) begin
      sel =  sel + 4'h1;
    end
  end
  // Different cases for Y.
  always @ (sel or Y) begin
    case(sel)
      4'h1: begin
        Y = NUM0;
        end
      4'h2: begin
        Y = NUM1;
        end
      4'h3:begin
        Y = NUM2;
        end
      4'h4: begin
        Y = NUM3;
        end
      4'h5: begin
        Y = NUM4;
        end
      4'h6: begin
        Y = NUM5;
        end
      4'h7: begin
        Y = NUM6;
        end
      4'h8: begin
        Y = NUM7;
        end
      default: begin
        Y = 8'h00;
      end
    endcase
  end
  // End simulation when we finish looking at all our cases.
  always @ (posedge clk) begin
    if( sel == 4'h8 ) begin
      if( X == 3'b111 ) begin
        #100;
        $finish;
      end
    end
  end
endmodule
```

Listing 4: Booth Encoder Verilog Stimulus

2.3 Twelve-bit Carry Lookahead Adder

Full Adder

The first block required for a multi-bit adder is the full adder. This circuit is identical to the half adder except that it has an additional input, Cin, so that a carry from a previous addition may be passed along. Furthermore, instead of a carry out, Cout, propogate (P) and generate (G) signals are produced. The logic for this schematic is as follows:

$$S = A \oplus B \oplus C_{in} \tag{2.1}$$

$$P = A \oplus B \tag{2.2}$$

$$G = A \times B \tag{2.3}$$

These two signals can be used to generate the carry out bit simultaneously for faster addition (see 4-bit Carry Lookahead Adder). Figure 19 shows a gate level implementation of the full adder schematic. Note that ~G is also generated. This bit can be used instead of G to simplify some of carry lookahead logic. Figure 20 shows a symbol view of the full adder. This block was verified using Verilog X. The simulation output can be seen in Figure 21, and the stimulus code can be seen in Listing 5. All possible inputs were swept, and all outputs were correct. There is a average of 4ns propagation delay from input to sum.

Figure 19: Full Adder Schematic

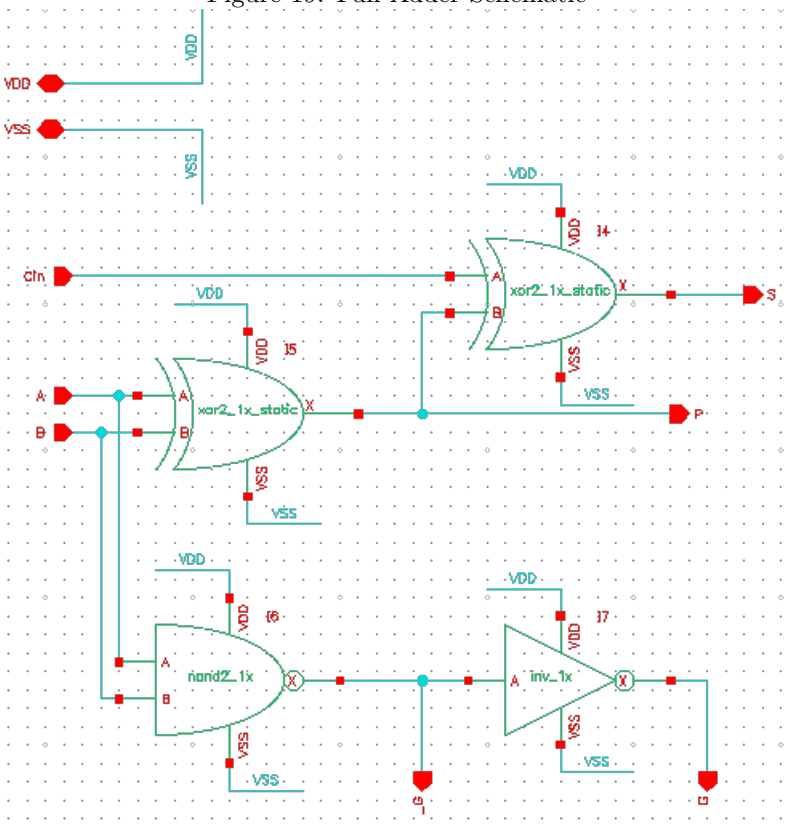


Figure 20: Full Adder Symbol

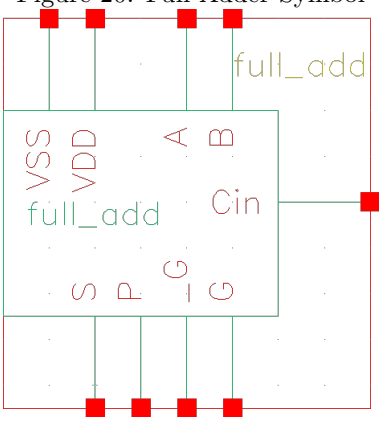
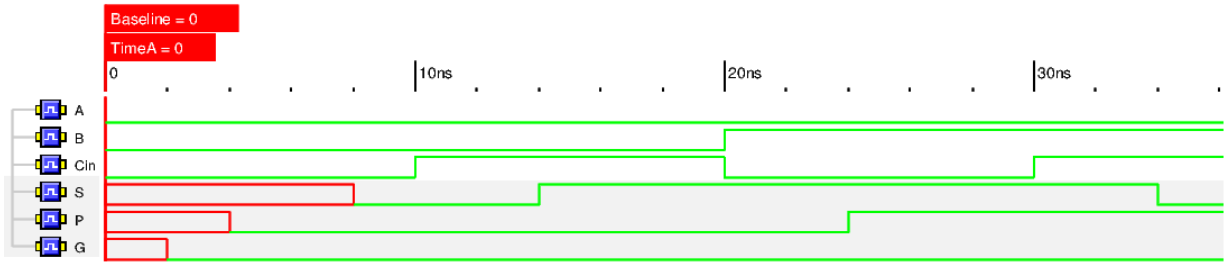


Figure 21: Full Adder Verilog X Simulation




```
// Verilog stimulus file.
// Please do not create a module in this file.

initial
begin
        A = 1'b0; B = 1'b0; Cin = 1'b0;
    #10 A = 1'b0; B = 1'b0; Cin = 1'b1;
    #10 A = 1'b0; B = 1'b1; Cin = 1'b0;
    #10 A = 1'b0; B = 1'b1; Cin = 1'b1;
    #10 A = 1'b1; B = 1'b0; Cin = 1'b0;
    #10 A = 1'b1; B = 1'b0; Cin = 1'b1;
    #10 A = 1'b1; B = 1'b1; Cin = 1'b0;
    #10 A = 1'b1; B = 1'b1; Cin = 1'b1;
    #10 $finish;
end
```

Listing 5: Booth Encoder Verilog Stimulus

Four-bit Carry Lookahead Adder

In order to improve the speed of the adder, carry lookahead logic was employed to compute the carry bits while simultaneously computing the sum bits. This method becomes less effective as the number of bits surpasses four; therefore, a 4-bit CLA was implemented. This block can be cascaded to produce higher bit adders.

Figure 22: 4-bit Adder Schematic

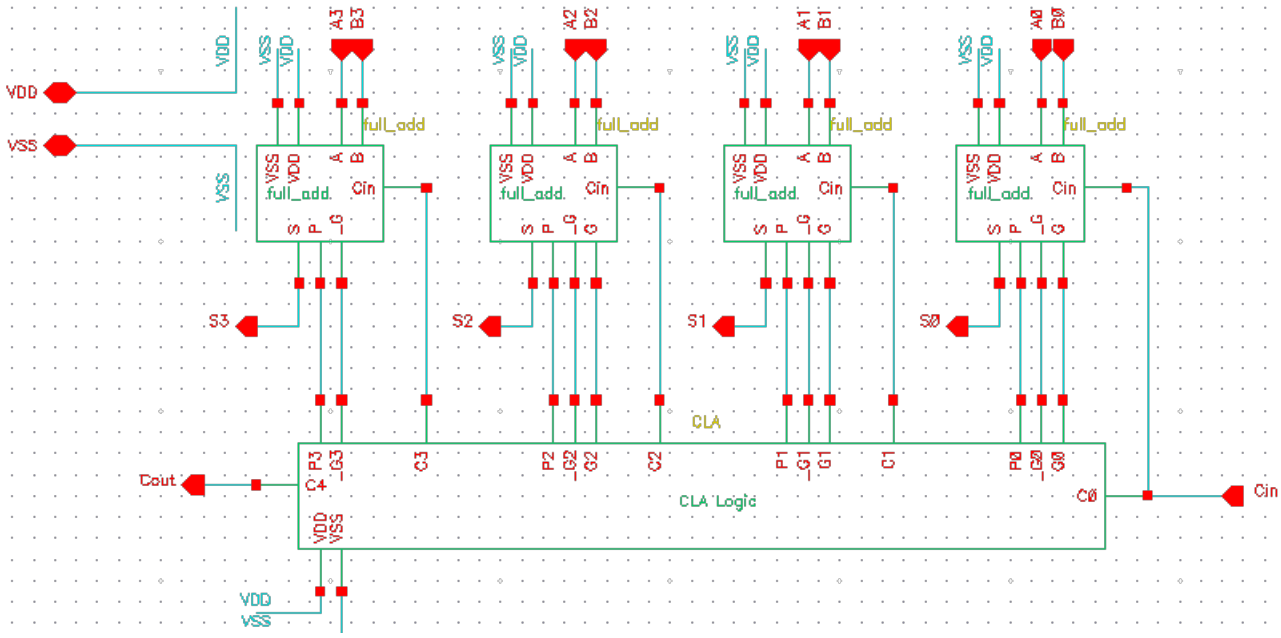


Figure 23: 4-bit Adder Symbol

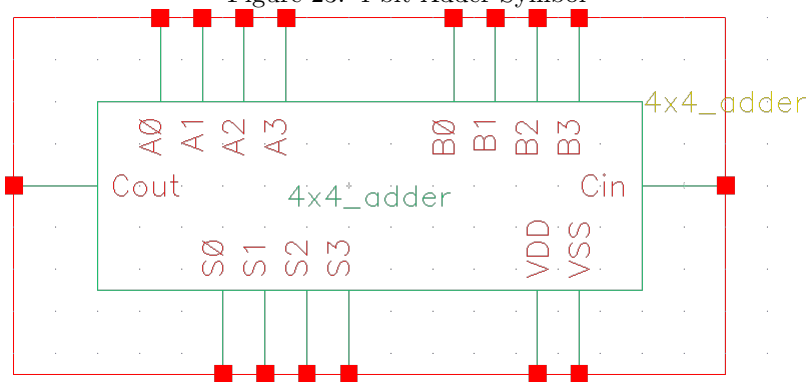


Figure 22 shows the schematic diagram of the 4-bit CLA, and Figure 23 shows a symbol view. This circuit consists of four full adders, which each pass propogate and generate signals to the CLA logic, which

computes the carry bits and feeds them into the carry-in input of the appropriate full adders. The carry bits are generated by the CLA based on the following logic:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \tag{2.4}$$

$$C_{i+1} = G_i + C_i P_i \tag{2.5}$$

Expanding Eq 2.5 for each carry bit of the four bit adder yields Eq 2.6-2.9, which contains only the first carry in bit and propogate and generate bits. Thus, the carry bits can be computed in parallel with the sum bits, which increases the speed of the adder compared to a ripple style adder.

$$C_1 = G_0 + C_0 P_0 \tag{2.6}$$

$$C_2 = G_1 + G_0 P_1 + C_0 P_0 P_1 \tag{2.7}$$

$$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2 \tag{2.8}$$

$$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3 \tag{2.9}$$

Observing these equations reveals that two more primitive gates are required to implement this logic: four and five input NAND gates. These gates were constructed using the logic primitives designed in Part I of this lab exercise. The schematics for the four and five input NAND gates are shown in Figure 24 and Figure 25, respectively. The CLA logic circuit that implements Eq 2.6-2.9 is shown in Figure 26. The code in Program Listing 6 was used to simulate the adder. Results are shown in Figure 27. All test cases were successful.

Figure 24: Four input NAND gate

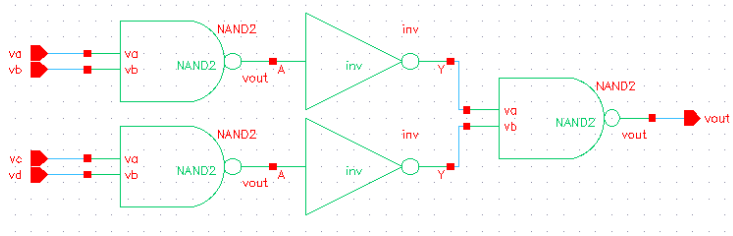


Figure 25: Five input NAND gate

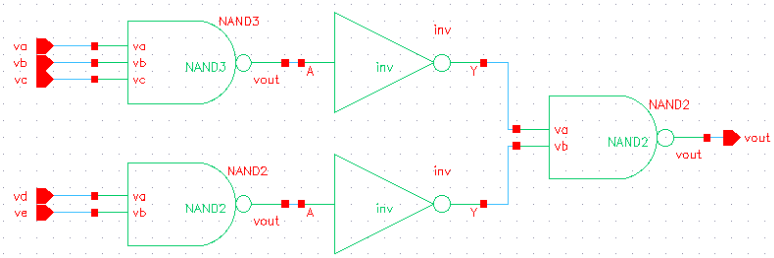


Figure 26: Carry Lookahead Logic Schematic

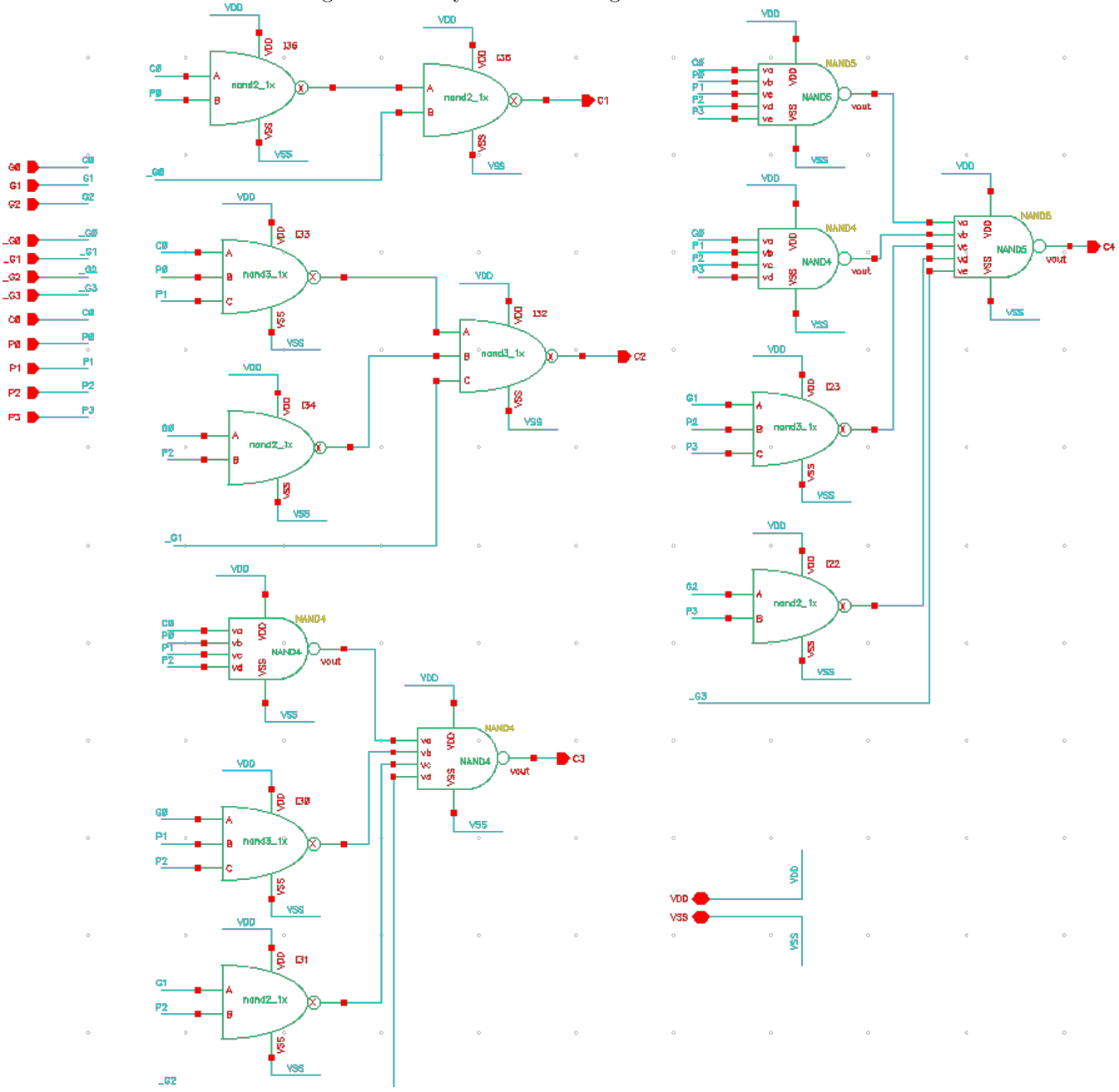
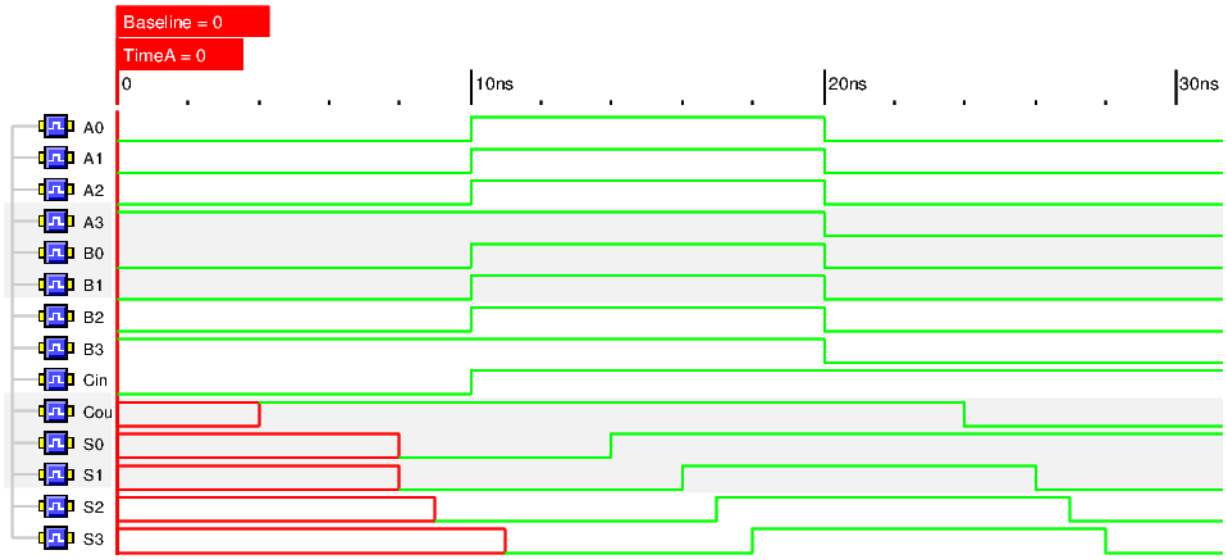


Figure 27: 4-bit Adder Simulation




```
// Verilog stimulus file.
// Please do not create a module in this file.

initial
begin
     A0 = 1'b0; A1 = 1'b0; A2 = 1'b0; A3 = 1'b1;
     B0 = 1'b0; B1 = 1'b0; B2 = 1'b0; B3 = 1'b1;
     Cin = 1'b0;
#10  A0 = 1'b1; A1 = 1'b1; A2 = 1'b1; A3 = 1'b1;
     B0 = 1'b1; B1 = 1'b1; B2 = 1'b1; B3 = 1'b1;
     Cin = 1'b1;
#10  A0 = 1'b0; A1 = 1'b0; A2 = 1'b0; A3 = 1'b0;
     B0 = 1'b0; B1 = 1'b0; B2 = 1'b0; B3 = 1'b0;
     Cin = 1'b1;
#20  $finish;
end
```

Listing 6: Booth Encoder Verilog Stimulus

12-bit Carry Lookahead Adder

In order to sum the 9-bit partial products from the decoders, a 12-bit CLA was implemented from the 4-bit CLA blocks. The block level schematic is shown in Figure 28. A symbol view of this circuit is shown in Figure 26. This circuit cascades four bit CLAs in the same way that the four bit CLAs cascade full adders. Within the original CLA logic schematic in Figure 24 the carry out bits of each four bit adder are also computed. This implementation provides another layer of carry lookahead logic for the 12-bit adder to further improve speed. This circuit was tested using the testbench schematic shown in Figure 27. The simulation results can be seen in Figure 29. The verilog code of the stimulus block is shown in Program Listing 7. All test cases were successful in this simulation. The average propagation delay was 8ns to compute one sum.

Figure 28: 12-bit Carry Lookahead Adder Symbol

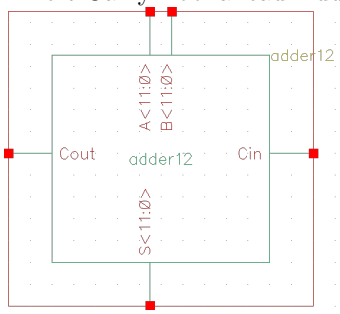


Figure 29: 12-bit Carry Lookahead Adder Testbench Schematic

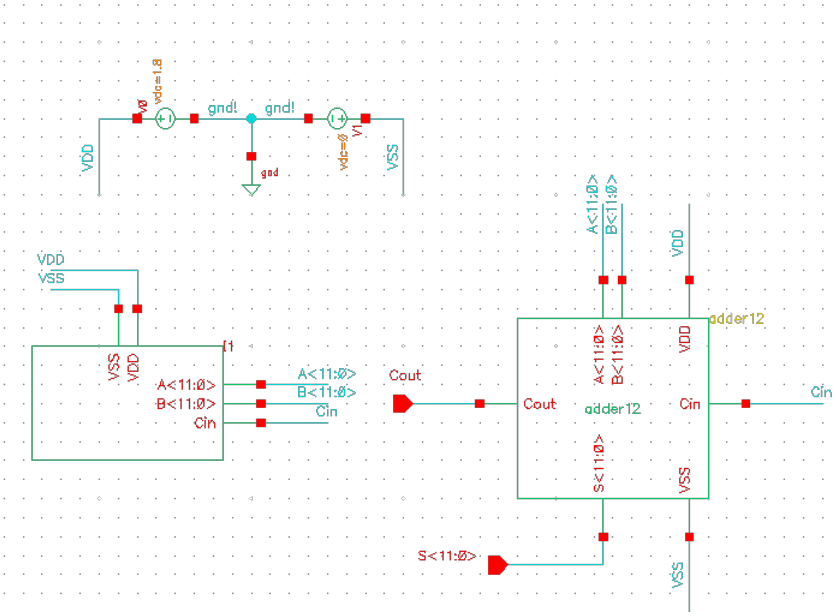


Figure 30: 12-bit Carry Lookahead Adder Schematic

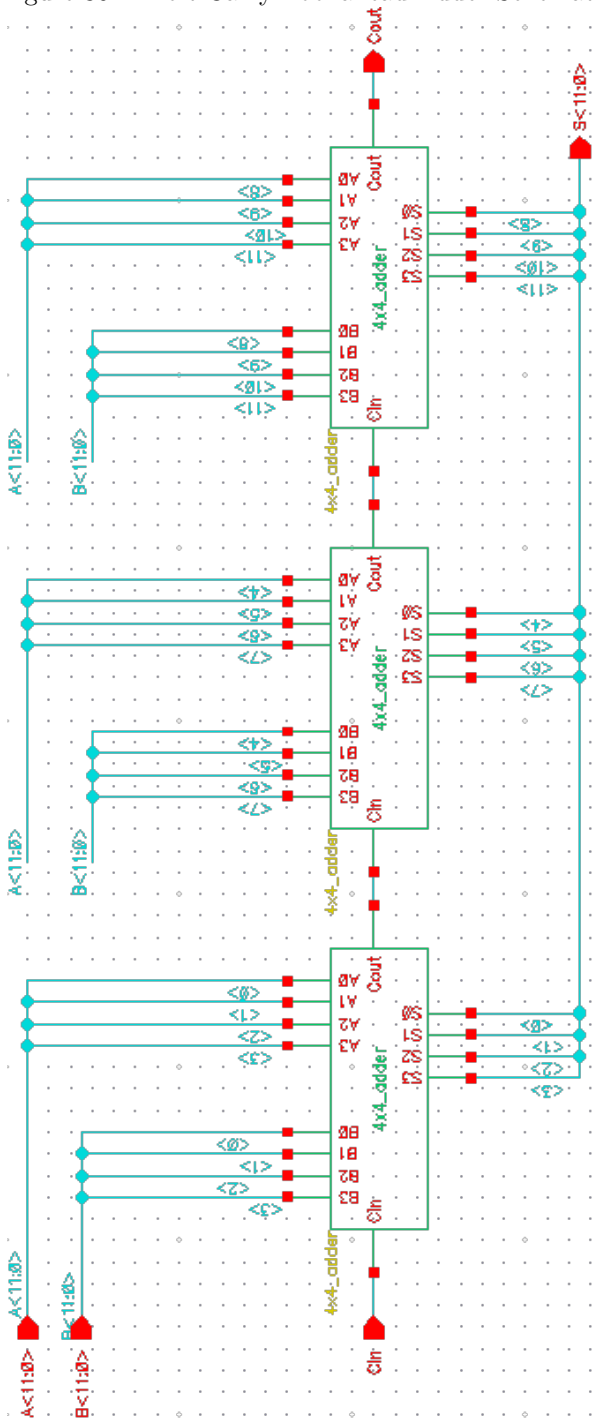
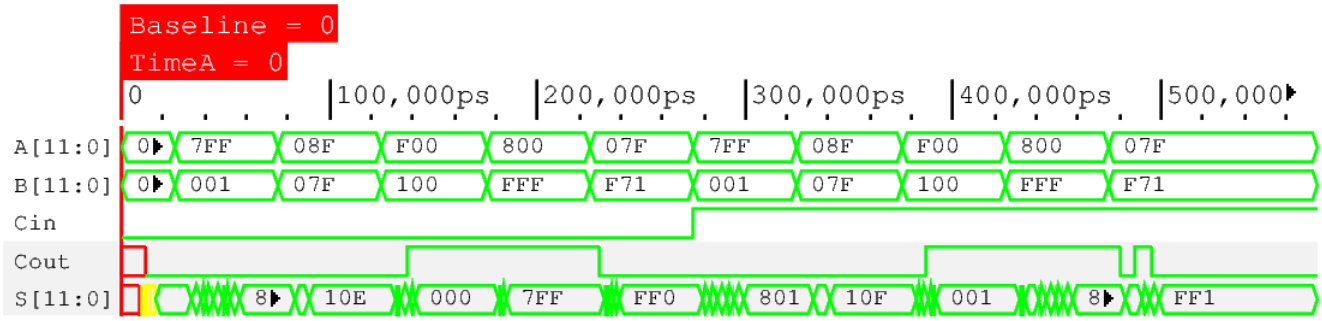


Figure 31: 12-bit Carry Lookahead Adder AMS Simulation




```
//Verilog HDL for "ee103", "adder_12bit_vstim" "functional"
module adder_12bit_vstim ( output reg [11:0] A,
                           output reg [11:0] B,
                           output reg Cin,
                           inout wire VDD,
                           inout wire VSS );

  reg clk;
  reg [3:0] sel;
  initial begin
    sel = 4'b0000;
    clk = 1'b0;
    A = 12'h000;
    B = 12'h000;
    Cin = 1'b0;
  end

  always begin
    #25.0 clk = ~clk;
  end
  always @ (posedge clk) begin
    sel = sel + 4'b0001;
  end
  always @ (sel) begin
  case (sel)
    4'b0001: begin
      A = 12'h7FF;
      B = 12'h001;
      Cin = 1'b0; // S = 800
      end
    4'b0010: begin
      A = 12'h08F;
      B = 12'h07F;
      Cin = 1'b0; // S = 10E
      end
    4'b0011: begin
```

```
      A = 12'hF00;
      B = 12'h100;
      Cin = 1'b0; // S = (1)000
      end
    4'b0100: begin
      A = 12'h800;
      B = 12'hFFF;
      Cin = 1'b0; // S = (1)7FF
      end
    4'b0101: begin
      A = 12'h07F;
      B = 12'hF71;
      Cin = 1'b0; // S = FF0
      end
    4'b0110: begin
      A = 12'h7FF;
      B = 12'h001;
      Cin = 1'b1; // S = 801
      end
    4'b0111: begin
      A = 12'h08F;
      B = 12'h07F;
      Cin = 1'b1; // S = 10F
      end
    4'b1000: begin
      A = 12'hF00;
      B = 12'h100;
      Cin = 1'b1; // S = (1)001
      end
    4'b1001: begin
      A = 12'h800;
      B = 12'hFFF;
      Cin = 1'b1; // S = (1)800
      end
    4'b1010: begin
      A = 12'h07F;
      B = 12'hF71;
      Cin = 1'b1; // S = FF1
      end
    default: begin
      #50 $finish;
      end
  endcase
  end
endmodule
```

Listing 7: Booth Encoder Verilog Stimulus

2.4 Sign Extension Trick

When summing the partial products to compute the final product, each partial product must be sign extended by different amounts (i.e. the first partial product must be extended by 6 bits, the second by 4, etc.) so that the sign bits of each partial product are reproduced correctly. The sign extension would require extra logic; however, the following technique preserves the sign bits without having to sign extend each partial product:

1. Invert the MSB of each partial product generated by the decoder.
2. Add '1' to the MSB of the first partial product.
3. Add '1' in front of each partial product.

This technique was implemented by first placing an inverter between the MSB of each 9-bit partial product and the corresponding input bit of the 12-bit adder. Then, the 10th input bit to each of the 12-bit adders is connected to VDD. This effectively adds '1' in front of each partial product. For the first partial product, however, this cannot be done. Adding '1' to the MSB could produce a carry bit, which must be taken into account. Furthermore, adding '1' in front of the MSB of the first partial product would produce a carry if the previous addition of '1' to the MSB produced a carry. This can be accomplished by cascading two half adders where each has one input tied to VDD, the first adder has a second input tied to the MSB of the first partial product, and the second adder has a second input connected to the carry out of the first adder. The two sums and final carry are then tied to input bits 6, 7 and 8 of the 12-bit adder respectively. However, this method will be quite slow. Table 4 shows the truth table for the addition. PP08 is the MSB of the first partial product, ~PP08 is the inverted PP08 from step 1 above. B6-B8 refer to the inputs to the 12-bit adder. Since ~PP08 can only be 1 or 0, there are only two possibilities for the addition. If ~PP08 = 1, a carry will be produced by the first addition, and another carry will be produced by the second addition resulting in B8B7B6 = 011. If ~PP8 = 0, no carry will be produced and B8B7B6 = 100. From the truth table it can be seen that B6 = B7 = ~PP08, and B8 = PP08. Thus, this circuit can be simplified to simply connecting these nodes as such. The two approaches are shown in Figure 32 and Figure 33.

Table 4: Sign Extension Truth Table

~PP08	PP08	B8	B7	B6
0	1	1	0	0
1	0	0	1	1

Figure 32: Sign Extension Trick with Half Adders

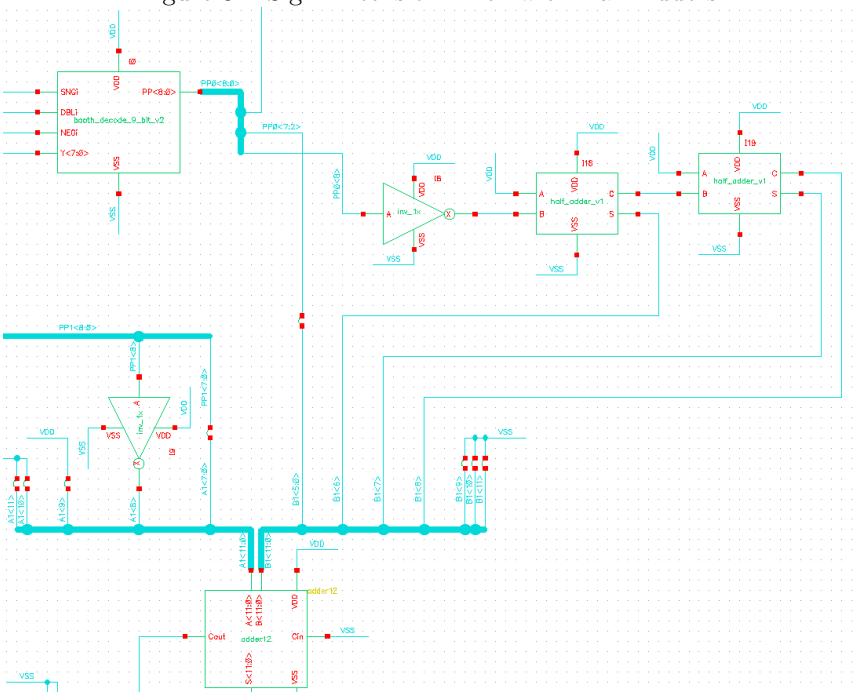
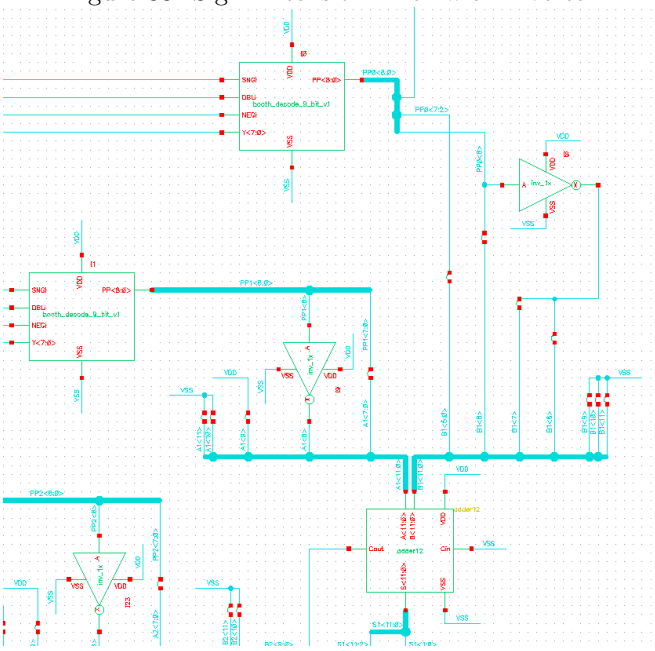


Figure 33: Sign Extension Trick with Inverter



2.5 Signed 8x8 Modified Booth Multiplier

From the blocks discussed above an 8x8 modified Booth multiplier was constructed following the architecture proposed in Section 1.2. The final schematic of the modified Booth multiplier is shown in Figure 36, and a symbol view is shown in Figure 34. This circuit takes in two 8-bit 2's complement binary numbers and produces a 16-bit output equal to the product of the two inputs. This circuit was simulated using AMS. The test bench schematic is shown in Figure 35. The simulation results are shown in Figure 37, and the stimulus code is shown in Program Listing 8. Unfortunately, this circuit does not operate as desired. Four of the output bits are incorrect. We believe this is due to a flaw in our implementation of the sign extension trick. The test case shows the output for every combination of 107, 105, -107 and -105. The delay is measured at 35ns per multiply, worst case.

Figure 34: 8x8 Booth Multiplier Symbol

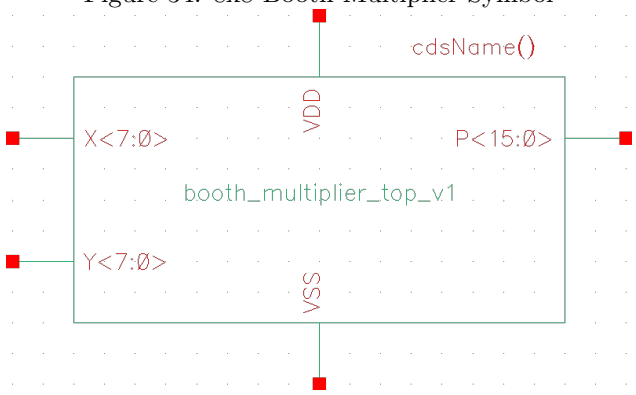


Figure 35: 8x8 Booth Multiplier Testbench Schematic

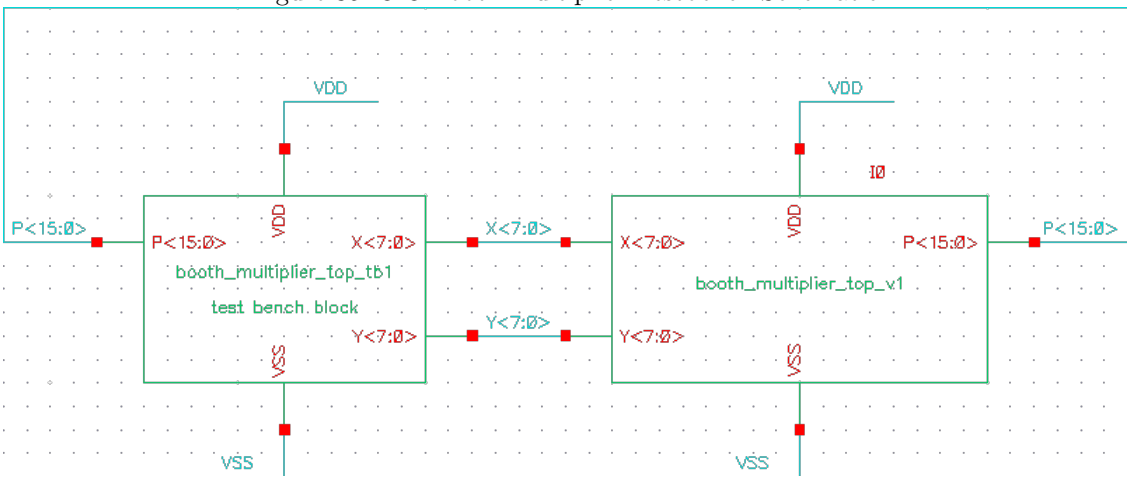


Figure 36: 8x8 Booth Multiplier Schematic

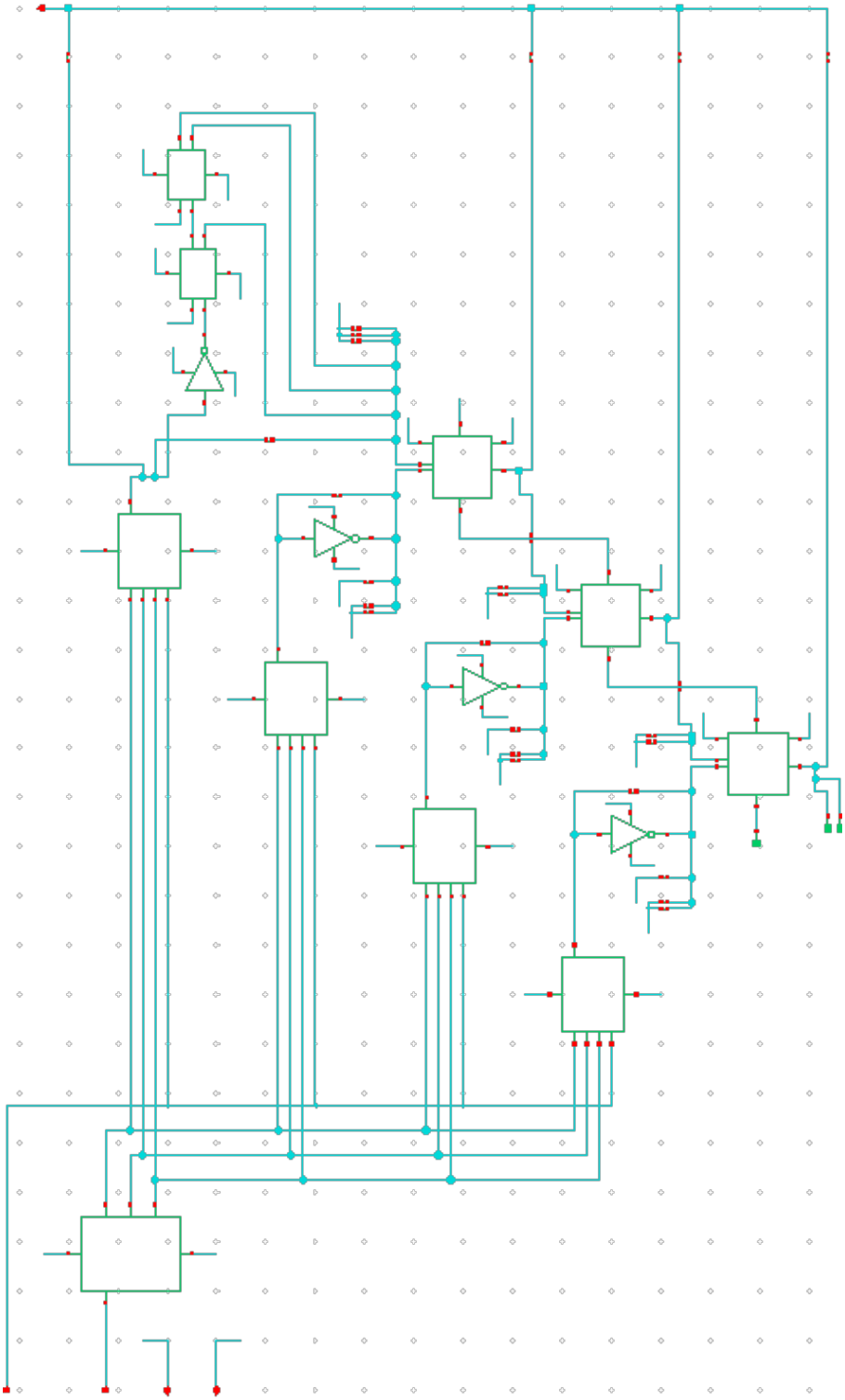
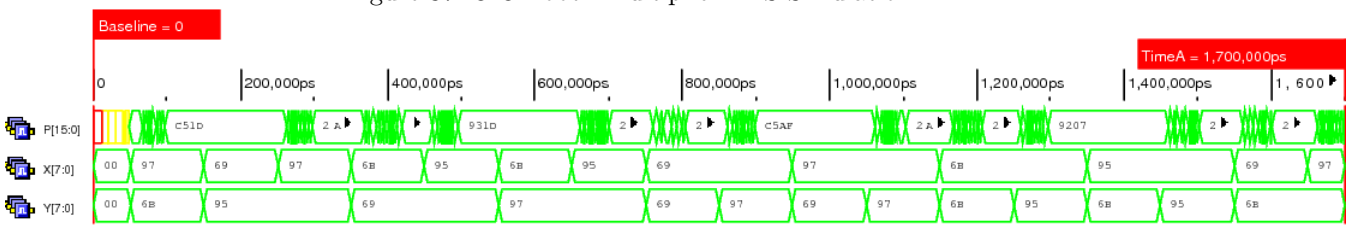


Figure 37: 8x8 Booth MultiplierAMS Simulation




```
//Verilog HDL for "lab3_sims", "booth_multiplier_top_tb1" "functional"

// Testbench for booth_multiplier_top_v1
// Scott Smith
// November 28, 2011
// STATUS - SEEMS TO WORK
module booth_multiplier_top_tb1 ( output reg [7:0] X,
                                  output reg [7:0] Y,
                                  input wire [15:0] P,
                                  inout wire VDD,
                                  inout wire VSS );

  // This test uses two numbers in various sign configurations
  // and orders (8 cases total).
  //parameter NUM1p = 8'b01101001; // +105
  //parameter NUM1n = 8'b10010111; // -105 (151 unsigned)
  //parameter NUM2p = 8'b01101011; // +107
  //parameter NUM2n = 8'b10010101; // -107 (149 unsigned)
parameter NUM1p = 8'b00000000; // 0
parameter NUM1n = 8'b00000000; // -0 (carry lost)
parameter NUM2p = 8'b00000001; // 1
parameter NUM2n = 8'b11111111; // -1


  reg clk;
  reg [3:0] sel;

  assign VDD = 1'b1;
  assign VSS = 1'b0;

  // Initialize variables.
  initial begin
    clk = 1'b0;
    sel = 4'h0;
    X = 8'h00;
    Y = 8'h00;
  end
  // Clock for advancing through the test cases.
```

```
always begin
  #50.0 clk = ~clk;
end
// 'sel' variable to iterate through the case statement below.
always @ (posedge clk) begin
  sel =  sel + 4'h1;
end
// Different test cases using the paramters above.
always @ (sel or X or Y) begin
  case(sel)
    4'h0: begin
      X = NUM1p;
      Y = NUM2p;
      end
    4'h1: begin
      X = NUM1n;
      Y = NUM2p;
      end
    4'h2:begin
      X = NUM1p;
      Y = NUM2n;
      end
    4'h3: begin
      X = NUM1n;
      Y = NUM2n;
      end
    4'h4: begin
      X = NUM2p;
      Y = NUM1p;
      end
    4'h5: begin
      X = NUM2n;
      Y = NUM1p;
      end
    4'h6: begin
      X = NUM2p;
      Y = NUM1n;
      end
    4'h7: begin
      X = NUM2n;
      Y = NUM1n;
      end
    //-----cases where a number is multiplied by itself-----//
    4'h8: begin
      X = NUM1p;
      Y = NUM1p;
      end
    4'h9: begin
```

```
          X = NUM1p;
          Y = NUM1n;
          end
        4'hA: begin
          X = NUM1n;
          Y = NUM1p;
          end
        4'hB: begin
          X = NUM1n;
          Y = NUM1n;
          end
        4'hC: begin
          X = NUM2p;
          Y = NUM2p;
          end
        4'hD: begin
          X = NUM2p;
          Y = NUM2n;
          end
        4'hE: begin
          X = NUM2n;
          Y = NUM2p;
          end
        4'hF: begin
          X = NUM2n;
          Y = NUM2n;
          end
        default: begin
          X = X;
          Y = Y;
        end
      endcase
    end
    // End simulation when we finish looking at all our cases.
    always @ (negedge clk) begin
      if( sel == 4'hF ) begin
        // Add some delay since this clk causes the inputs to update,
        // but does not tell us what is going on at the output.
        #200;
        $finish;
      end
    end
    // Add code to do error-checking automatically...
endmodule
```

Listing 8: Booth Encoder Verilog Stimulus

3 Conclusions

Table 5 below summarizes the delay seen through each block in this design. Several techniques can be used in the final project to reduce these delay times.

Table 5: Delay Summary

Block	Propagation Delay
Half Adder	4ns
Full Adder	4ns
4-bit Adder	8ns
12-bit Adder	8ns
Single Bit Decoder	7ns
Booth Multiplier	35ns

This lab demonstrates the successful gate level implementation of an 8-bit signed modified booth multiplier. However, the most significant four bits are incorrect. We believe this is due to our sign extension implementation. The first step of the final project will be to debug this technique. This design will be optimized for maximum speed using IBM's 180nm process, CMRF7SF. A layout will be constructed and post layout simulation will be performed so that this device can be manufactured.