Jay Noppone Pornpitaksuk

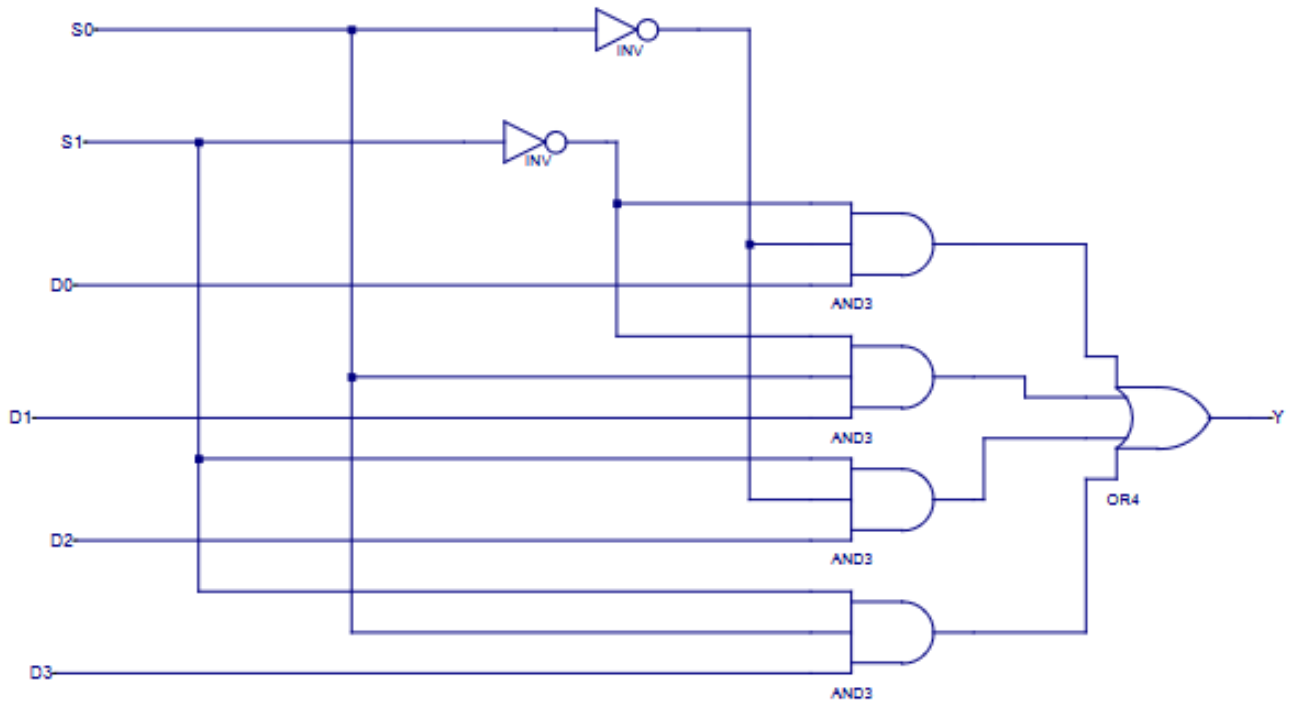
Lab 01

CSC34300

Professor Fazli

Task 1: 4-1 Multiplexer

1.1 Schematic/Circuit Diagram:



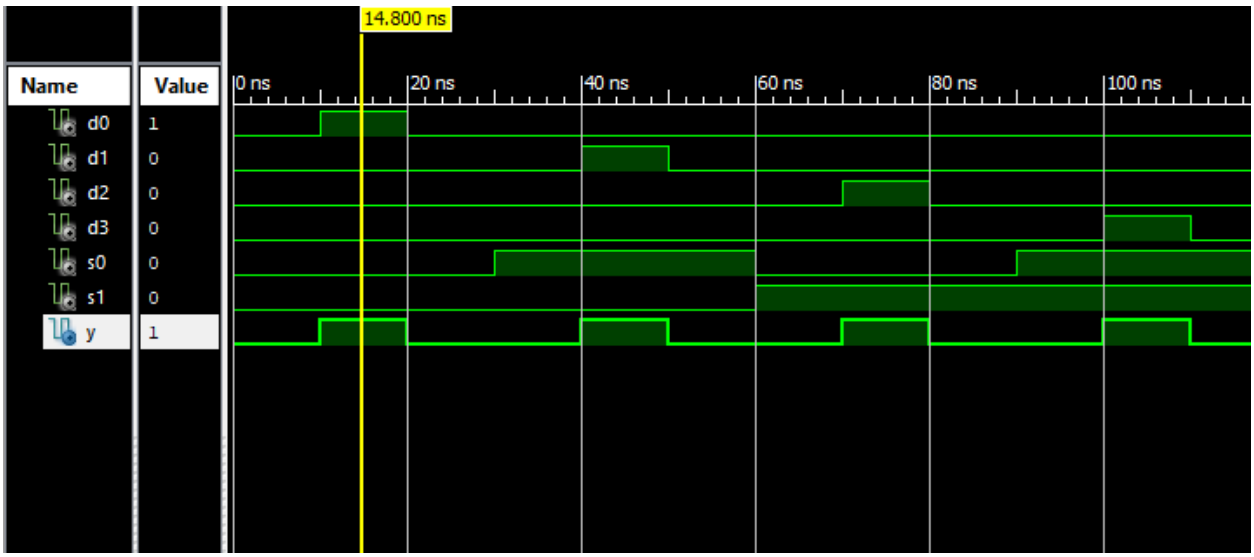
To build the multiplexer above, we require two inverse gates to invert the S0 and S1 selectors. Four AND3 gates are needed to determine which decision/choice is accepted between D0-3, which would then be output as Y. For example, to select the D0 output, we must put the S0 and S1 as 0, 0 to indicate choosing the 0^{th} choice. To have this output, they must both be 1s so they can be indicated as a 1 when reaching the OR4 gate. For the D1 selection, the goal is to be able to input selectors S1, S0 as 0,1 to represent the binary number 1. This means we must convert the S1 into its inverse to have both values at 1,1 just like the system set up for the D0. This then allows for D1 to be selected when the user inputs 0,1 for S1,S0.

1.2 Entity, Architecture, and Test Bench

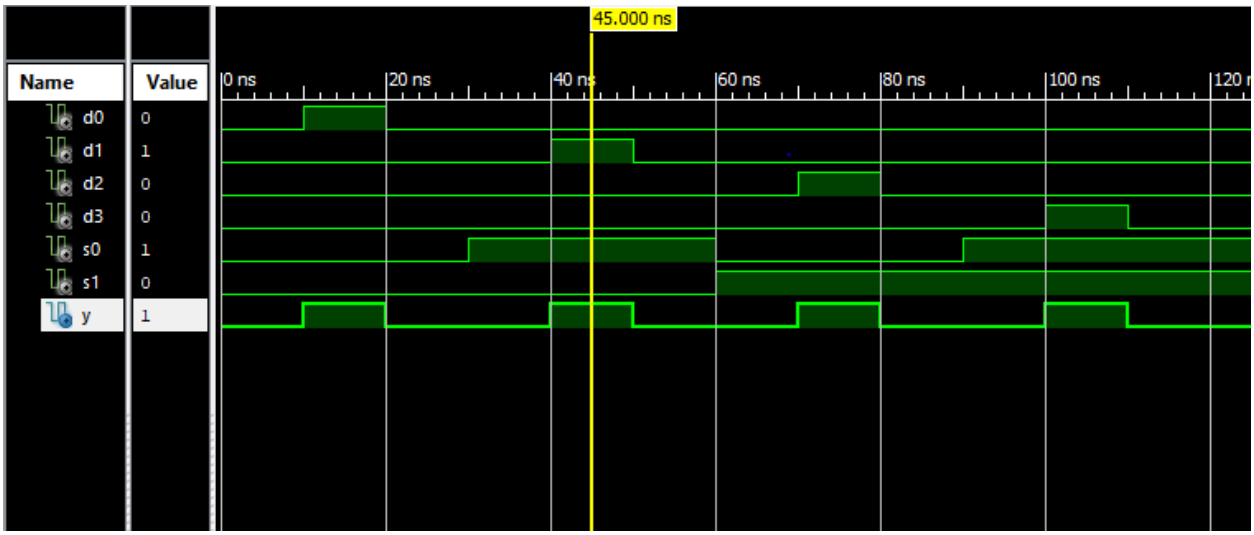
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library unisim;
use unisim.vcomponents.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mux is
port ( D0: in STD_LOGIC;
       D1: in STD_LOGIC;
       D2: in STD_LOGIC;
       D3: in STD_LOGIC;
       S0: in STD_LOGIC;
       S1: in STD_LOGIC;
       Y: out STD_LOGIC);
end mux;

architecture Behavioral of mux is
signal S0_L, S1_L: STD_LOGIC;
signal S1L_S0L_D0, S1L_S0_D1, S1_S0L_D2, S1_S0_D3: STD_LOGIC;
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
   U1: INV port map (S0, S0_L);
   U2: INV port map (S1, S1_L);
   U3: AND3 port map (S1_L, S0_L, D0, S1L_S0L_D0);
   U4: AND3 port map (S1_L, S0, D1, S1L_S0_D1);
   U5: AND3 port map (S1, S0_L, D2, S1_S0L_D2);
   U6: AND3 port map (S1, S0, D3, S1_S0_D3);
   U7: OR4 port map (S1L_S0L_D0, S1L_S0_D1, S1_S0L_D2, S1_S0_D3, Y);

end Behavioral;
```

```
-- Stimulus process
stim_proc: process
begin
   -- Initialization
   D0 <= '0'; D1 <= '0'; D2 <= '0'; D3 <= '0';
   -- Select D0
   S1 <= '0'; S0 <= '0'; wait for 10 ns;
   D0 <= '1'; wait for 10 ns;
   D0 <= '0'; wait for 10 ns;
   -- Select D1
   S1 <= '0'; S0 <= '1'; wait for 10 ns;
   D1 <= '1'; wait for 10 ns;
   D1 <= '0'; wait for 10 ns;
   -- Select D2
   S1 <= '1'; S0 <= '0'; wait for 10 ns;
   D2 <= '1'; wait for 10 ns;
   D2 <= '0'; wait for 10 ns;
   -- Select D3
   S1 <= '1'; S0 <= '1'; wait for 10 ns;
   D3 <= '1'; wait for 10 ns;
   D3 <= '0'; wait for 10 ns;
   WAIT; -- WAIT FOREVER
end process;
```

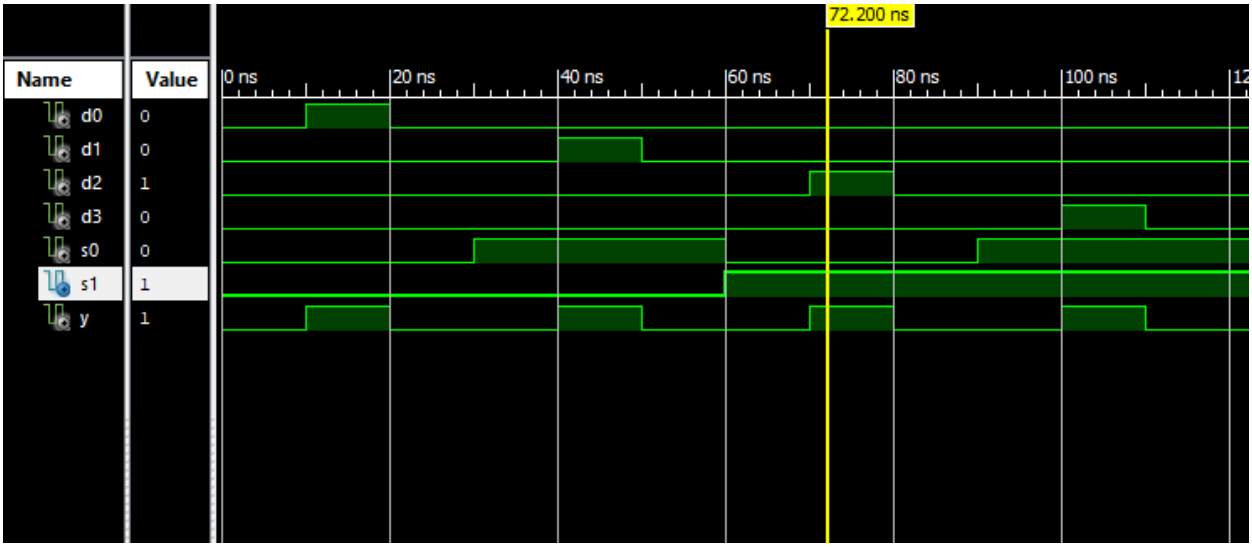
1.3 Results



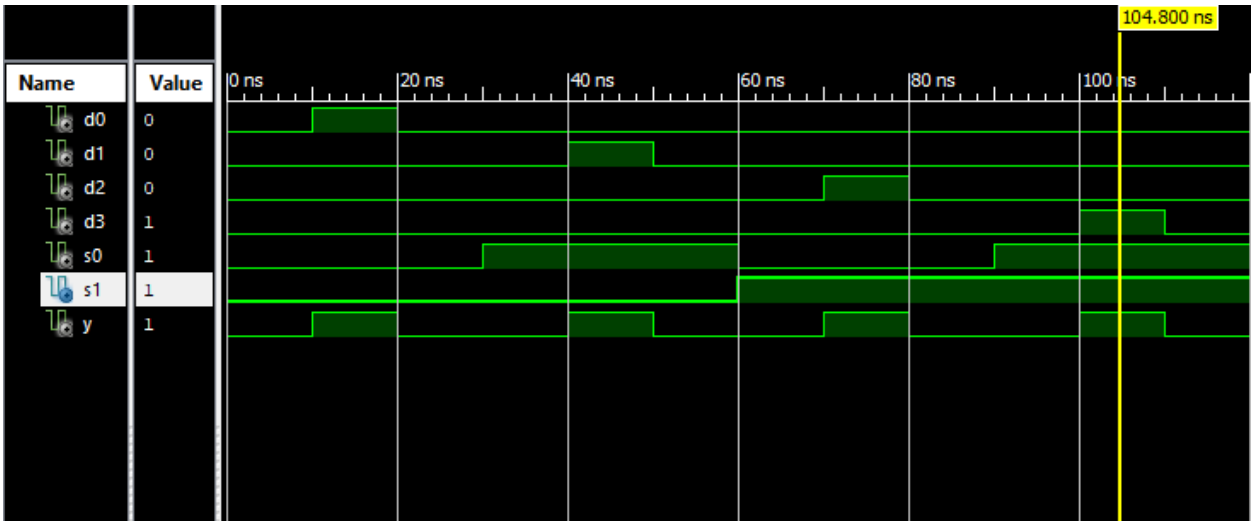
The above shows the multiplexer selecting neither S1 nor S0, resulting in the result Y being D0, which is shown above to have a value of 1, while the other selections are at value 0.



With the same reasoning as the previous image, the above screenshot shows that when S0 is 1, and S1 is 0, just like in the truth table given, the selected value in the multiplexer is D1



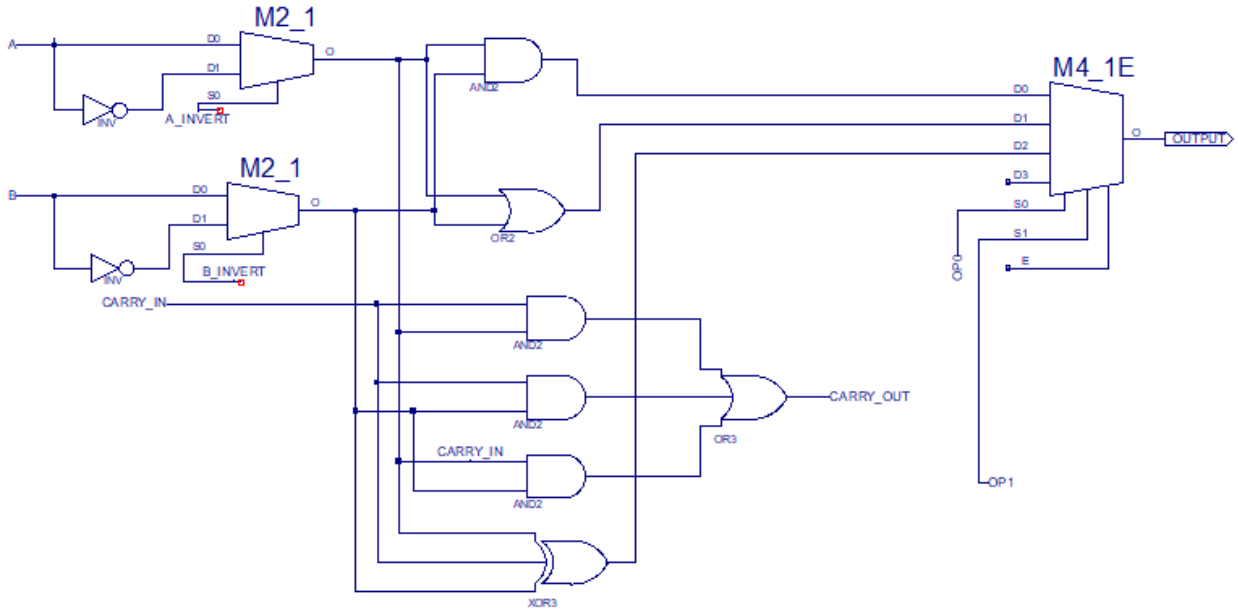
Above, our multiplexer program has S1 at the value 1, and S0 at 0. The program correctly chooses D2 as its current output value Y because the selectors 1 and 0 both compute as 10, which represents 2, hence picking the option D2,



Finally, for the final output value in the truth table, both the S1 and S0 are set to 1; therefore, correctly choosing the output D3 for Y, just like shown in the truth table.

Task 2: 1-bit ALU

2.1 Schematic/Circuit Diagram:



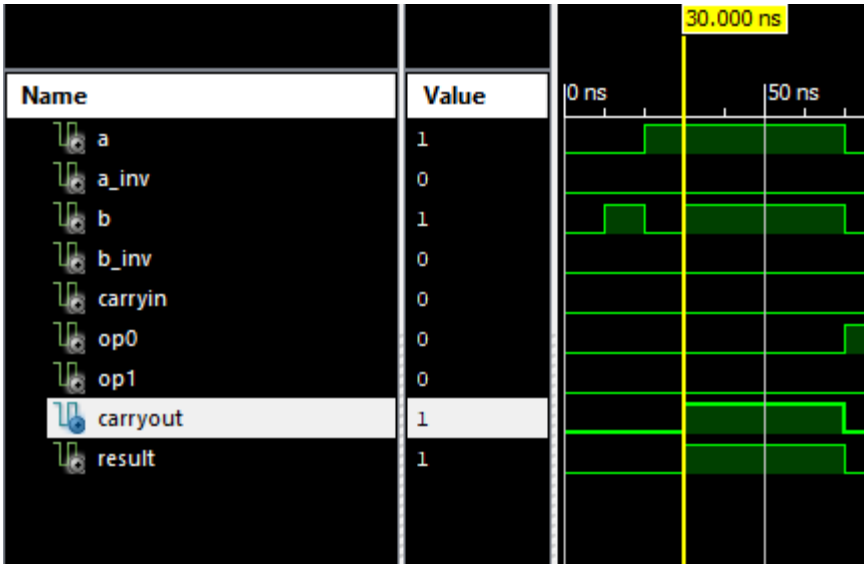
Shown above is the schematic for the 1-bit ALU. Inputs A and B are processed by port mapping A to A', B to B', which are then used for the 2x1 mux that are shown up twice above. D0 represents the regular inputs A and B, and D1 represents inverted values which are chosen by the A_invert and B_invert values. Outputs of these 2x1 muxes are then used for the AND, OR, and adder components of the diagram. Whether the output will be based on the AND, OR, or the adder is based on the selector given. A selector with the value of 00 will select the D0 option, which is the AND operator, 01 will select D1 for the OR operator, then 10 will select D2 for the addition/subtraction operator. The schematic is compatible with subtraction because we are able to insert a CarryIn value that is connected to the adder component of the circuit diagram, and that we are able to invert the value B.

2.2 Entity, Architecture, and Test Bench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library unisim;
use unisim.vcomponents.all;

entity one_bit_alu is
port ( A: in STD_LOGIC;
       A_inv: in STD_LOGIC;
       B: in STD_LOGIC;
       B_inv: in STD_LOGIC;
       CarryIn: in STD_LOGIC;
       OP0: in STD_LOGIC;
       OP1: in STD_LOGIC;
       CarryOut: out STD_LOGIC;
       Result: out STD_LOGIC);
end one_bit_alu;

architecture Behavioral of one_bit_alu is
signal A_L, B_L: STD_LOGIC;
signal AM, BM: STD_LOGIC;
signal AND_AM_BM, OR_AM_BM, CIN_AND_AM, CIN_AND_BM, AM_AND_BM, AM_CIN_XOR_BM: STD_LOGIC;
signal mux_d4: STD_LOGIC;
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
component mux21 port (D0, D1, S0: in STD_LOGIC; Y: out STD_LOGIC); end component;
component AND2 port (I0, I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR2 port (I0, I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component mux port (D0, D1, D2, D3, S0, S1: in STD_LOGIC; Y: out STD_LOGIC); end component;
component XOR3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
   U1: INV port map (A, A_L);
   U2: INV port map (B, B_L);
   U3: mux21 port map (A, A_L, A_inv, AM);
   U4: mux21 port map (B, B_L, B_inv, BM);
   U5: AND2 port map (AM, BM, AND_AM_BM);
   U6: OR2 port map (AM, BM, OR_AM_BM);
   U7: AND2 port map (AM, CarryIn, CIN_AND_AM);
   U8: AND2 port map (BM, CarryIn, CIN_AND_BM);
   U9: AND2 port map (AM, BM, AM_AND_BM);
   U10: XOR3 port map (AM, BM, CarryIn, AM_CIN_XOR_BM);
   U11: OR3 port map (CIN_AND_AM, CIN_AND_BM, AM_AND_BM, CarryOut);
   U12: mux port map (AND_AM_BM, OR_AM_BM, AM_CIN_XOR_BM, mux_d4, OP0, OP1, Result);

end Behavioral;
```

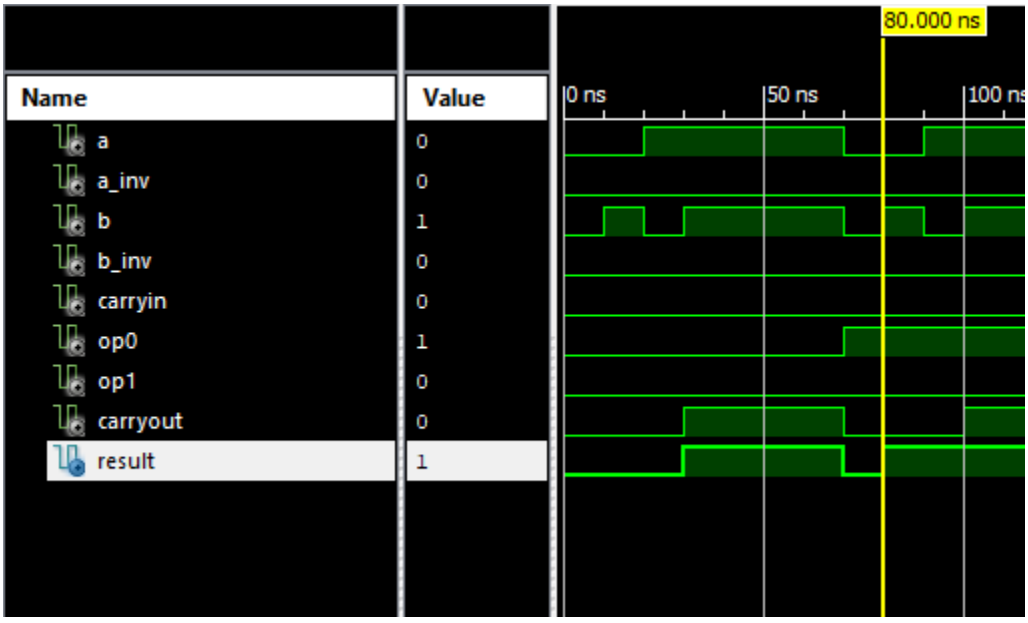
```
  -- Stimulus process
  stim_proc: process
  begin
  -- A AND B
  A_inv <= '0'; B_inv <= '0'; OP1 <= '0'; OP0 <= '0'; CarryIn <= '0';
  A <= '0'; B <= '0'; wait for 10 ns;
  A <= '0'; B <= '1'; wait for 10 ns;
  A <= '1'; B <= '0'; wait for 10 ns;
  A <= '1'; B <= '1'; wait for 40 ns;
  -- A OR B
  A_inv <= '0'; B_inv <= '0'; OP1 <= '0'; OP0 <= '1'; CarryIn <= '0';
  A <= '0'; B <= '0'; wait for 10 ns;
  A <= '0'; B <= '1'; wait for 10 ns;
  A <= '1'; B <= '0'; wait for 10 ns;
  A <= '1'; B <= '1'; wait for 40 ns;
  -- A + B
  -- CarryIn = '0'
  A_inv <= '0'; B_inv <= '0'; OP1 <= '1'; OP0 <= '0'; CarryIn <= '0';
  A <= '0'; B <= '0'; wait for 10 ns;
  A <= '0'; B <= '1'; wait for 10 ns;
  A <= '1'; B <= '0'; wait for 10 ns;
  A <= '1'; B <= '1'; wait for 40 ns;
  -- CarryIn = '1'
  A_inv <= '0'; B_inv <= '0'; OP1 <= '1'; OP0 <= '0'; CarryIn <= '1';
  A <= '0'; B <= '0'; wait for 10 ns;
  A <= '0'; B <= '1'; wait for 10 ns;
  A <= '1'; B <= '0'; wait for 10 ns;
  A <= '1'; B <= '1'; wait for 40 ns;
  -- A - B
  A_inv <= '0'; B_inv <= '1'; OP1 <= '1'; OP0 <= '0'; CarryIn <= '1';
  A <= '0'; B <= '0'; wait for 10 ns;
  A <= '0'; B <= '1'; wait for 10 ns;
  A <= '1'; B <= '0'; wait for 10 ns;
  A <= '1'; B <= '1'; wait for 40 ns;
  wait;
  end process;

END;
```

2.3 Results

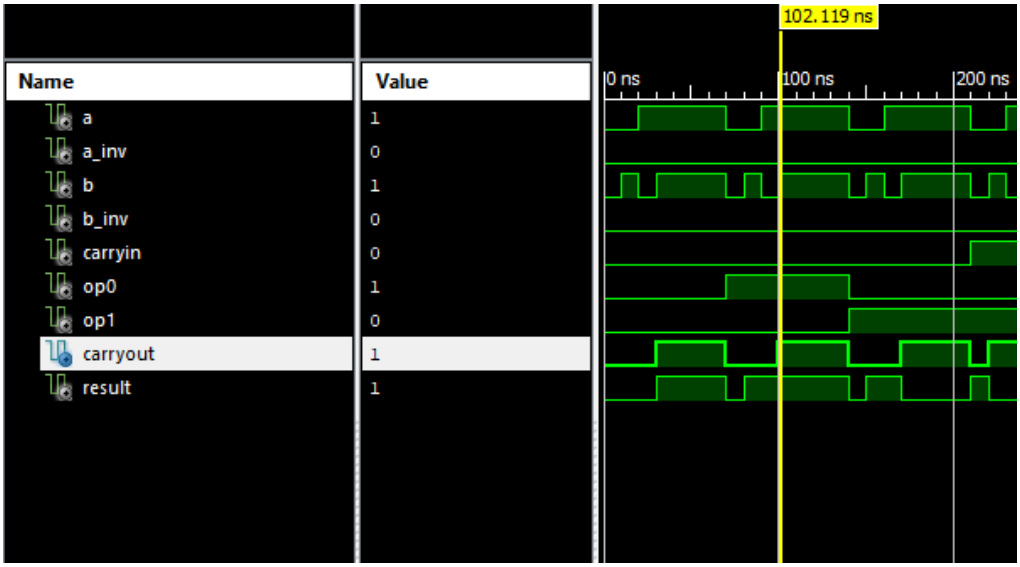


Above is the waveform for the AND operation (since op0=0, op1=0, thus selecting option 0 which is the AND operation). When we have inputs a and b as 1 and 1 respectively, this means 1

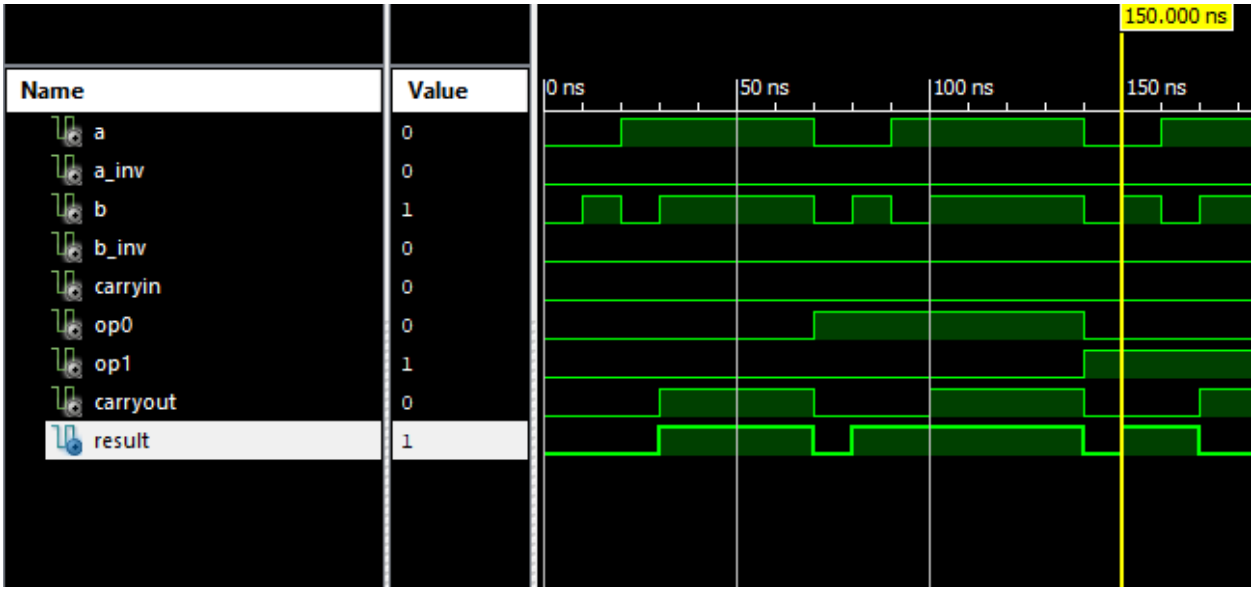
AND 1, which returns 1 because they are both 1. The reason that the carryout is also raised from 0 to 1 is because all the operations occur at the same time; however, we filter out which process we actually want as an output. So, the carryout is calculated, but is never actually used because we chose the AND operation's result.



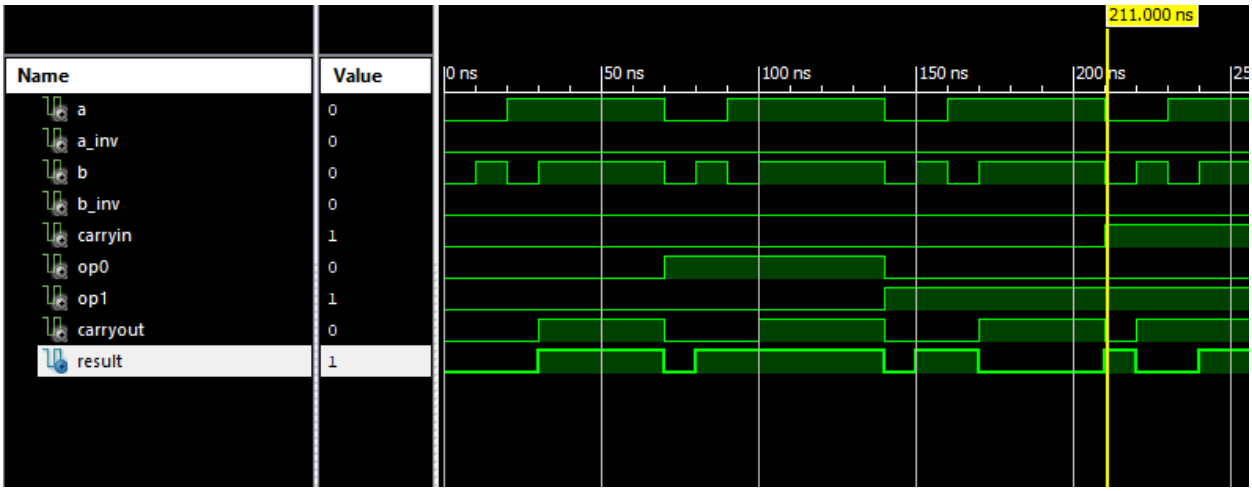
The above shows a new operation, 01, which stands for just 1 or the OR operation with our code and system. This compares inputs a and b which are 0 and 1 respectively; since one of them is a 1, the result will be a 1.



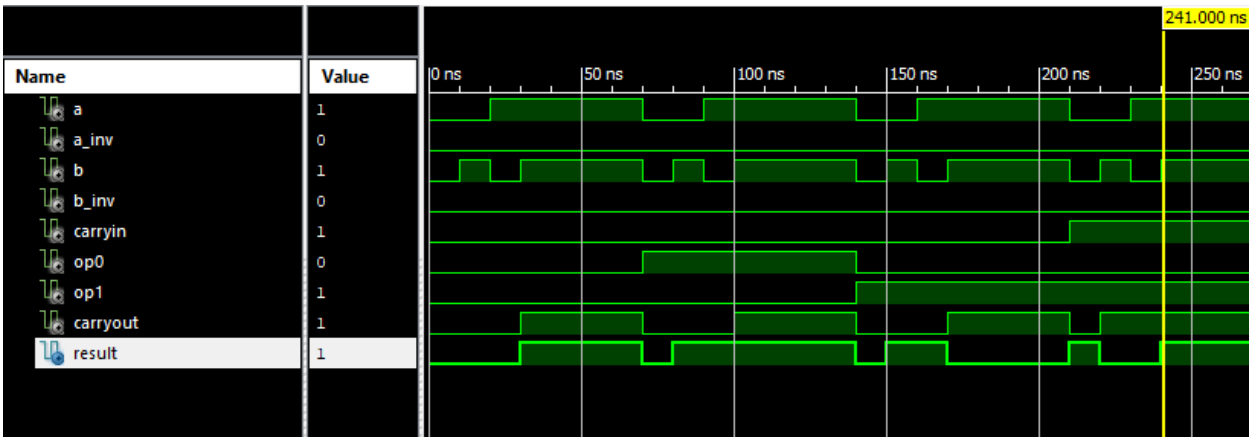
Above, we can see that we have different inputs for a and b, but with the same operation as the previous screenshot above. Since our a and b are 1 and 1 respectively, the program already calculates the addition operation, and since 1 + 1 = 10, the carryout will be 1, but the result will be based on the operation, which is 01, which is the OR operation, thus bringing a result of 1.



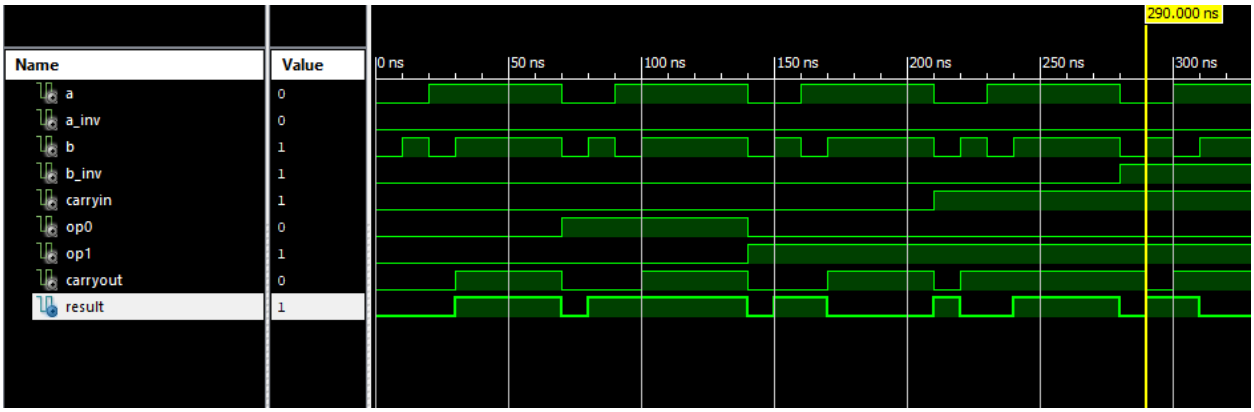
The operation in the above screenshot is now 10, standing for operation 2, which is the addition process, coming after the OR operation in the last screenshot. We have no CarryIn nor inversions for either A_inv nor B_inv, so we will be adding a and b, which are 0 and 1 respectively. The result of adding 0 to 1, is just 1, hence the result being 1.



In the above screenshot, we are also adding, but we have a CarryIn value of 1, therefore adding a + b + CarryIn. Since both a and b are both 0, we just add the 1 + 0, then finally outputting the result as 1.



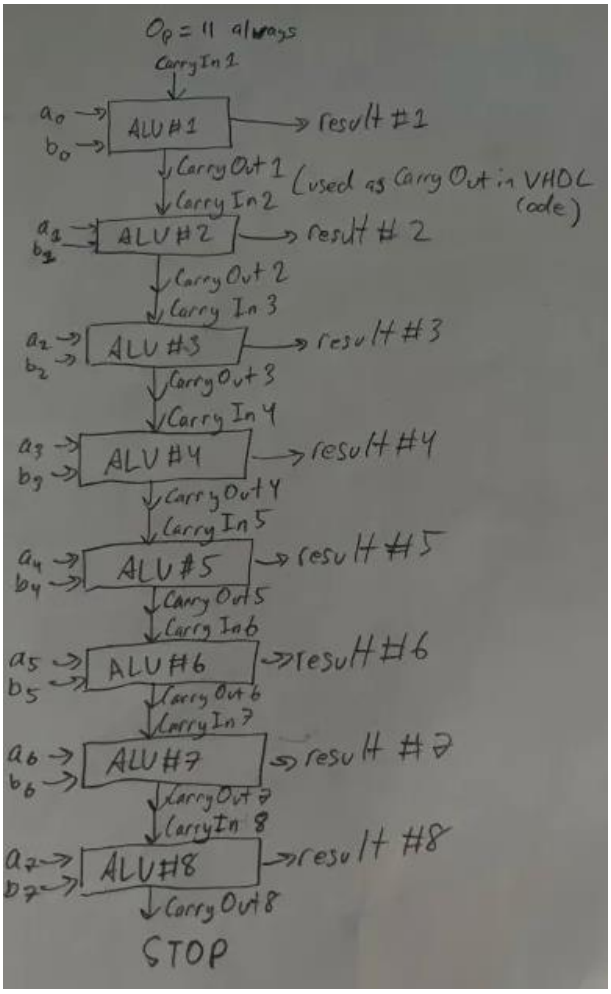
In this addition operation above, we have a CarryIn at 1 like before, but we have a and b as values 1 and 1, thus performing the operation 1 + 1 + 1, which gives us the value 11, which thus sends the left 1 value to the carryout, and the right 1 to the result output.



In the above, like stated in the test bank, we are performing the subtraction operation A – B, which is technically just A + B' + 1. We see that the b_inv is set to 1 to invert the B value, and CarryIn is set to 1 to add 1 to the B complement. Our a and b are 0 and 1 respectively, thus being 0 and 0 after inversion, then adding 1 to the 0 + 0. This results in just 1 as the result, resulting in no carryout.

Task 3: 8-bit ALU

3.1 Schematic/Circuit Diagram:



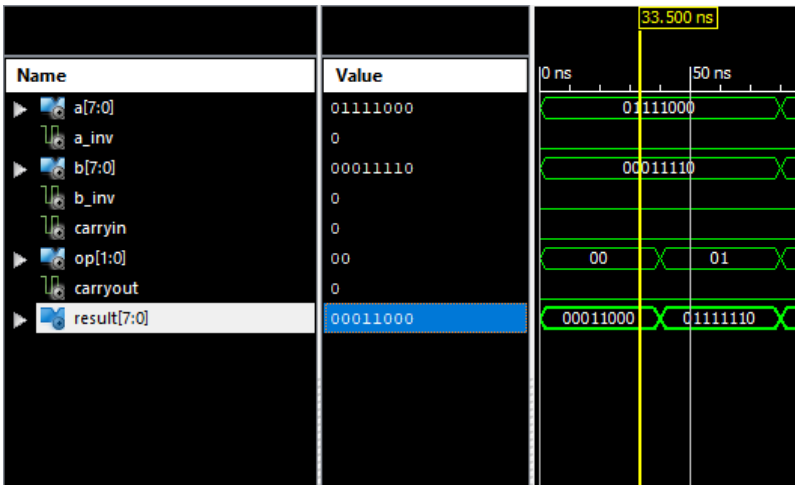
In the above schematic diagram, we reuse the 1-bit ALU diagram that we built in task 2. The first ALU will use the original CarryIn given, and then assign whatever CarryOut generated to the given CarryOut signal from the port. Then, we can declare individual CarryOut signals for the following ALU's to be used for the next ALU when being reused from the previous bit. Each result will be assigned to the vector generated from 0-7th place on the binary number, all collectively being the result at the end.

3.2 Entity, Architecture, and Test Bench

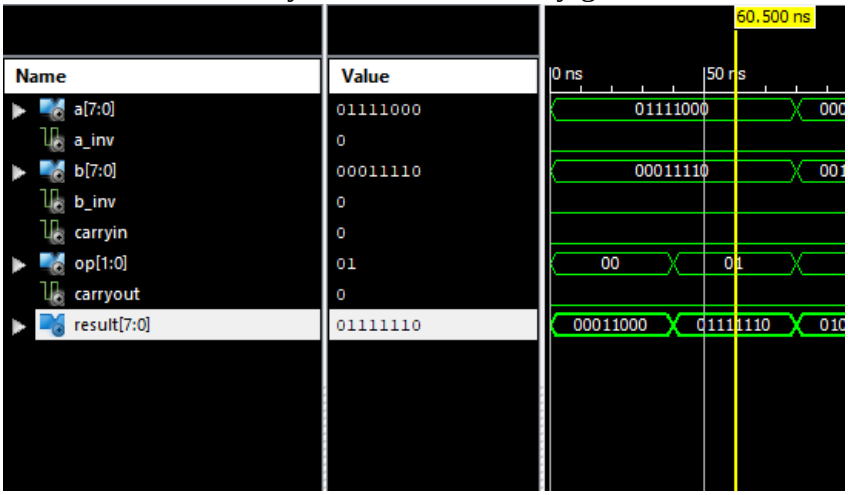
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;

entity eight_bit_adder is
port (A : in STD_LOGIC_VECTOR (7 downto 0);
      A_inv : in STD_LOGIC;
      B : in STD_LOGIC_VECTOR (7 downto 0);
      B_inv : in STD_LOGIC;
      CarryIn : in STD_LOGIC;
      OP : in STD_LOGIC_VECTOR (1 downto 0);
      CarryOut : inout STD_LOGIC;
      Result : out STD_LOGIC_VECTOR (7 downto 0));
end eight_bit_adder;

architecture Behavioral of eight_bit_adder is
signal CarryOut2, CarryOut3, CarryOut4, CarryOut5: STD_LOGIC;
signal CarryOut6, CarryOut7, CarryOut8: STD_LOGIC;
component one_bit_alu port (A, A_inv, B, B_inv, CarryIn, OP0, OP1: in STD_LOGIC;
                            CarryOut, Result: out STD_LOGIC); end component;
begin
   U1: one_bit_alu port map (A(0), A_inv, B(0), B_inv, CarryIn, OP(0), OP(1), CarryOut, Result(0));
   U2: one_bit_alu port map (A(1), A_inv, B(1), B_inv, CarryOut, OP(0), OP(1), CarryOut2, Result(1));
   U3: one_bit_alu port map (A(2), A_inv, B(2), B_inv, CarryOut2, OP(0), OP(1), CarryOut3, Result(2));
   U4: one_bit_alu port map (A(3), A_inv, B(3), B_inv, CarryOut3, OP(0), OP(1), CarryOut4, Result(3));
   U5: one_bit_alu port map (A(4), A_inv, B(4), B_inv, CarryOut4, OP(0), OP(1), CarryOut5, Result(4));
   U6: one_bit_alu port map (A(5), A_inv, B(5), B_inv, CarryOut5, OP(0), OP(1), CarryOut6, Result(5));
   U7: one_bit_alu port map (A(6), A_inv, B(6), B_inv, CarryOut6, OP(0), OP(1), CarryOut7, Result(6));
   U8: one_bit_alu port map (A(7), A_inv, B(7), B_inv, CarryOut7, OP(0), OP(1), CarryOut8, Result(7));

end Behavioral;
```

```
BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: eight_bit_adder PORT MAP (
          A => A,
          A_inv => A_inv,
          B => B,
          B_inv => B_inv,
          CarryIn => CarryIn,
          OP => OP,
          CarryOut => CarryOut,
          Result => Result
        );


   -- Stimulus process
   stim_proc: process
   begin
      -- A AND B
      OP <= "00"; CarryIn <= '0'; A_inv <= '0'; B_inv <= '0';
      A <= "01111000"; B <= "00011110"; wait for 40 ns;
      -- A OR B
      OP <= "01"; CarryIn <= '0'; A_inv <= '0'; B_inv <= '0';
      A <= "01111000"; B <= "00011110"; wait for 40 ns;
      -- A + B
      OP <= "10"; CarryIn <= '0'; A_inv <= '0'; B_inv <= '0';
      A <= "00001111"; B <= "00111000"; wait for 40 ns;
      -- A - B
      OP <= "10"; CarryIn <= '1'; A_inv <= '0'; B_inv <= '1';
      A <= "01010101"; B <= "00101010"; wait for 40 ns;
      wait;
   end process;

END;
```

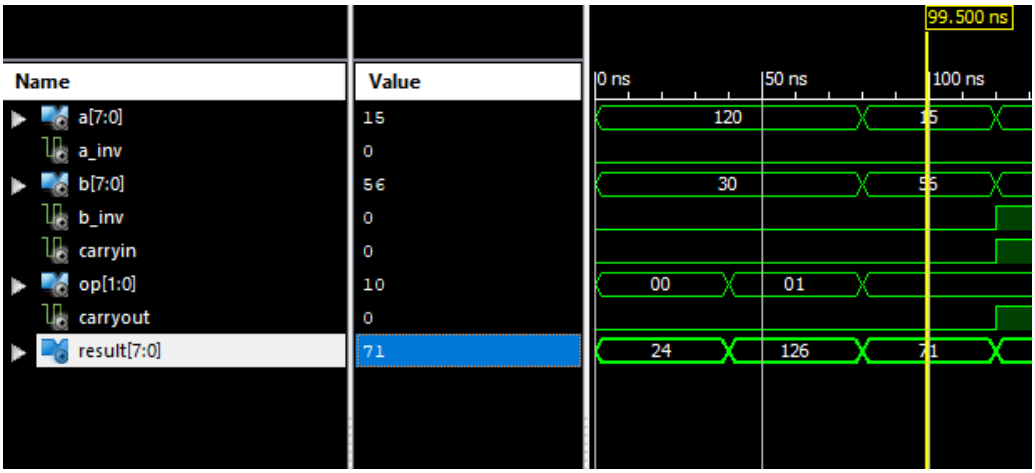
3.3 Results



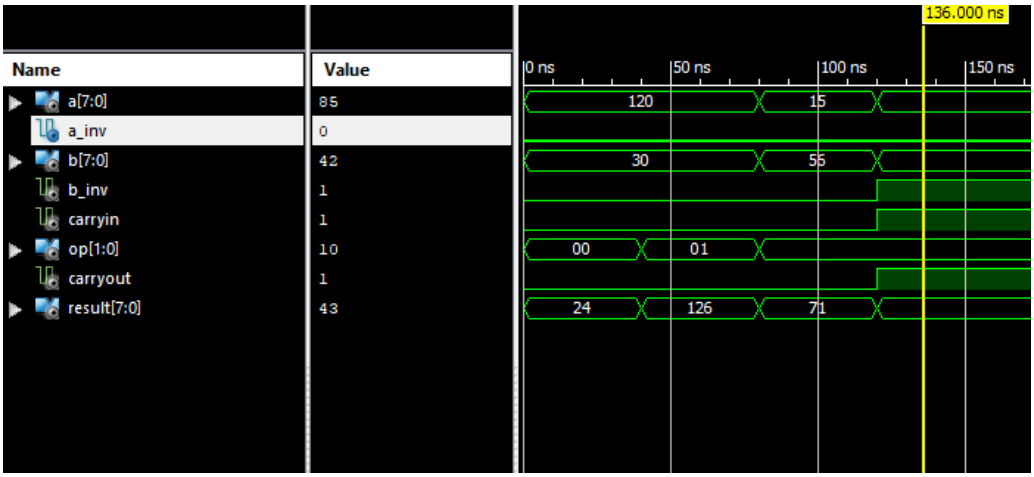
In the first test, we compute 01111000 AND 00011110, and the program written correctly outputs the AND operation on each bit. Whenever 1 AND 1 is computed, the result is 1; otherwise, it will always be 0. This correctly gives us the result of 00011000



Above is the second test of using the OR operation. We compute 01111000 OR 00011110, where each bit compares using the OR operation. Wherever there is a 1 in either of the bits, the result becomes 1; otherwise, if there are no 1s, then the output is 0. The code correctly gives us the result of 01111110.



In the third test, we use the addition, which is selected with the Op 10, which indicates the 2nd operation. Above, the addition is correctly computed where a and b are 15 and 56 respectively. The code that builds the 8-bit ripple adder correctly computes 15+56, which correctly outputs 71.



Finally, we test the subtraction function which instead of doing vanilla subtraction, the computation A-B would simply just be A+B'+1, hence the b_inv & carryin both being set to 1 to satisfy this equation. In the waveform above, we see that the output correctly computes 43, which is the result of 85-42.