CSc 21100 - Fundamentals of Computer Systems

Instructor: Prof. Zheng Peng

Project: 03

Name: Kristina Ilyovska

Section: CC1

Date: November 18, 2021

Task 1: 2-to-4 decoder

The function we implement is a selector that gives us the appropriate output (y3,y2,y1,y0) based on the binary digit representation of the 2-bit input (i1, i0) when the enable bit is enabled (en=1). If the enable bit is not enabled (en=0) then the chip is not activated and the output is always zero no matter what the input was.

Entity declaration

```
entity v2to4dec is
        Port ( en : in  STD_LOGIC;              --enable bit
        i0 : in  STD_LOGIC;                     --input 0
        i1 : in  STD_LOGIC;                     --input 1
        y0 : out  STD_LOGIC;                    --outputs
        y1 : out  STD_LOGIC;
        y2 : out  STD_LOGIC;
        y3 : out  STD_LOGIC);
end v2to4dec;
```

Architecture definition

```
architecture v2to4dec_arch of v2to4dec is
begin
y0 <= en and not i0 and not i1;                 --truth table representation
y1 <= en and not i0 and i1;
y2 <= en and i0 and not i1;
y3 <= en and i0 and i1;
end v2to4dec_arch;
```

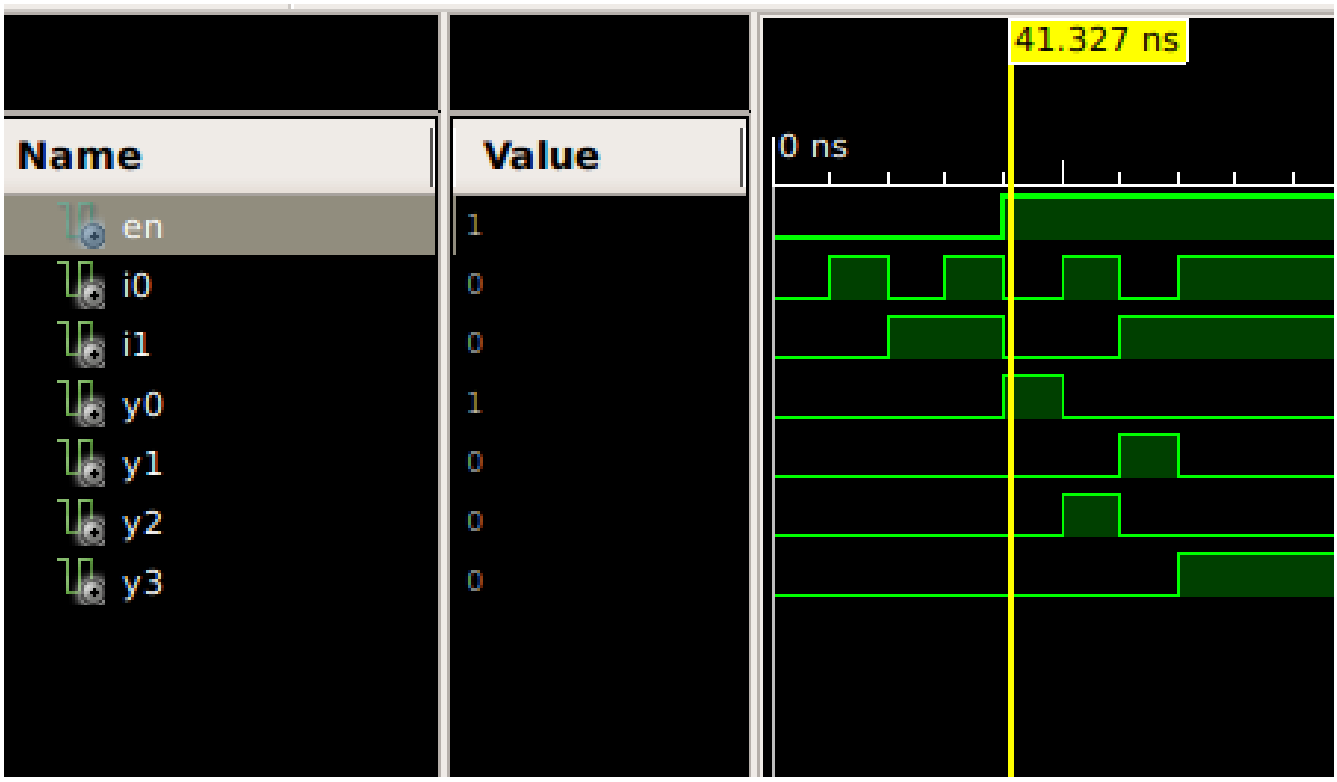
Testbench program

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY v2to4dec_tb IS
END v2to4dec_tb;

ARCHITECTURE structural OF v2to4dec_tb IS
        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT v2to4dec
        PORT(
        en : IN  std_logic;
        i0 : IN  std_logic;
```

```
        i1 : IN  std_logic;
        y0 : OUT  std_logic;
        y1 : OUT  std_logic;
        y2 : OUT  std_logic;
        y3 : OUT  std_logic
        );
        END COMPONENT;

  --Inputs
  signal en : std_logic := '0';
  signal i0 : std_logic := '0';
  signal i1 : std_logic := '0';

 --Outputs
  signal y0 : std_logic;
  signal y1 : std_logic;
  signal y2 : std_logic;
  signal y3 : std_logic;

BEGIN
   -- Instantiate the Unit Under Test (UUT)
  uut: v2to4dec PORT MAP (
        en => en,
        i0 => i0,
        i1 => i1,
        y0 => y0,
        y1 => y1,
        y2 => y2,
        y3 => y3
        );
  -- Stimulus process
  stim_proc: process

  begin
         en <= '0';
        i1 <= '0'; i0 <= '0'; wait for 10 ns;
        i1 <= '0'; i0 <= '1'; wait for 10 ns;
        i1 <= '1'; i0 <= '0'; wait for 10 ns;
        i1 <= '1'; i0 <= '1'; wait for 10 ns;

        en <= '1';
        i1 <= '0'; i0 <= '0'; wait for 10 ns;
        i1 <= '0'; i0 <= '1'; wait for 10 ns;
        i1 <= '1'; i0 <= '0'; wait for 10 ns;
```

```
        i1 <= '1'; i0 <= '1'; wait for 10 ns;
        wait;
  end process;
END;
```

Wave form



The yellow vertical line separates the points where the enable bit is set to 0 (to the left of the line) and where the enable bit is set to 1 (to the right of the line).
We can see that no matter what the values of the two inputs, i0 and i1, are when the enable bit is set to 0, the output is always 0.
If we focus on the right side of the line where the enable bit is set to 1 and the decoder is enabled, we can see the following four cases:

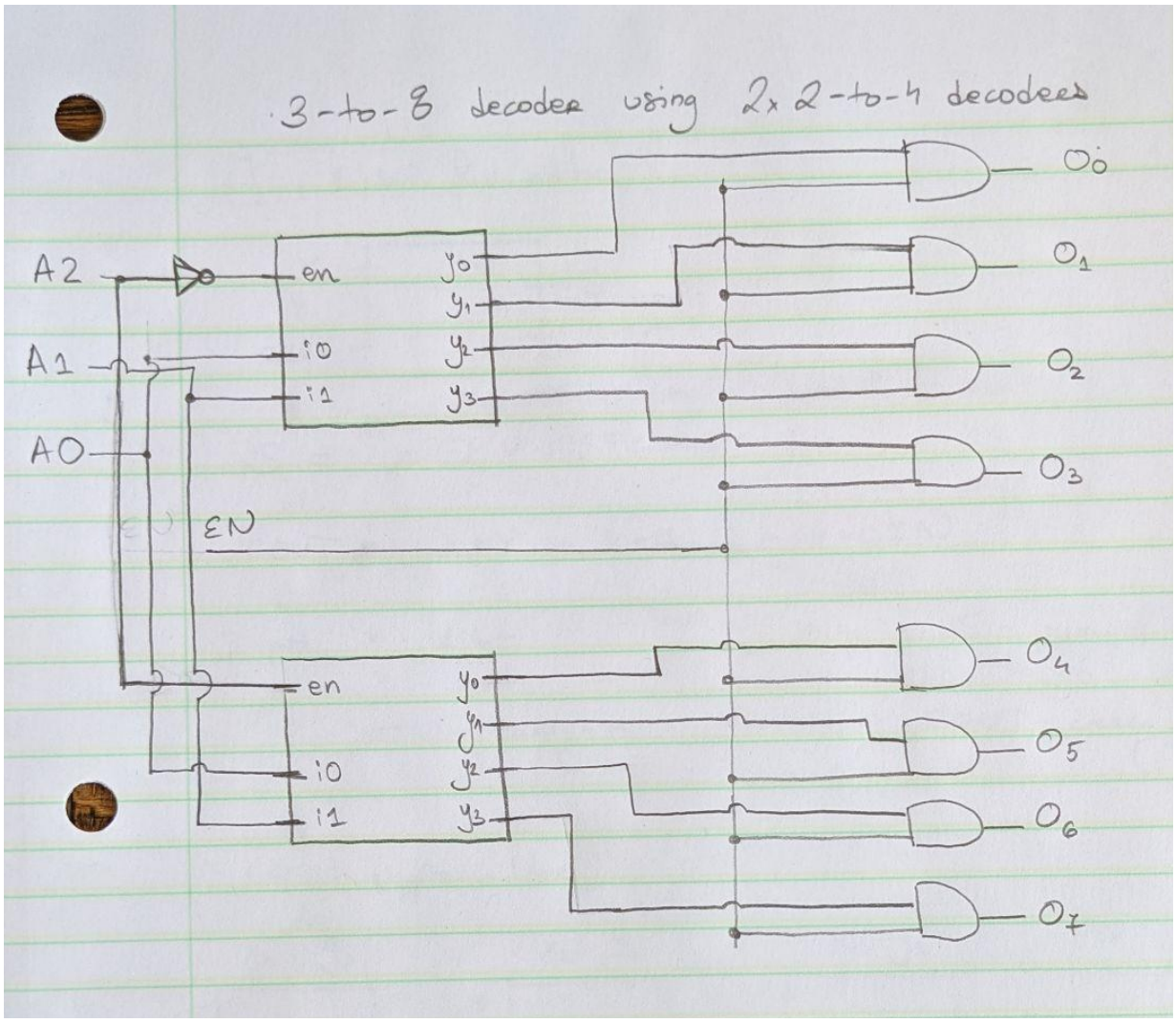
- i0 = 0, i1 = 0 → y0 = 1, y1 = 0, y2 = 0, y3 = 0
- i0 = 0, i1 = 1 → y0 = 0, y1 = 1, y2 = 0, y3 = 0
- i0 = 1, i1 = 0 → y0 = 0, y1 = 0, y2 = 1, y3 = 0
- i0 = 1, i1 = 1 → y0 = 0, y1 = 0, y2 = 0, y3 = 1

The behavior of the simulated program matches the truth table provided in the assignment.

Task 2: 3-to-8 decoder

The 3-to-8 decoder selects the correct output based on the 3-bit input provided. It uses two 2-to-4 decoders and 8 AND gates. The most significant bit of the input is used as the enabling bit to select which 2-to-4 decoder is used. The enabling bit is used to determine if the output given by the appropriate 2-to-4 decoder should be the final output (when the enabling bit is on) or no output should be provided (when the enabling bit is off).

Circuit diagram



Entity declaration

```
entity v3to8dec is
    port ( a: in  STD_LOGIC_VECTOR(2 downto 0);
```

```
        en: in  STD_LOGIC;
        o: out  STD_LOGIC_VECTOR(7 downto 0));
end v3to8dec;
```

Architecture definition

```
architecture v3to8dec_arch of v3to8dec is

signal A2_L: STD_LOGIC;
signal Y_TEMP: STD_LOGIC_VECTOR(7 downto 0);

component v2to4dec
   port (i0: in STD_LOGIC;
                  i1: in STD_LOGIC;
                  en: in STD_LOGIC;
                  y3: out STD_LOGIC;
                  y2: out STD_LOGIC;
                  y1: out STD_LOGIC;
                  y0: out STD_LOGIC);
end component;

component AND2 port(i0, i1: in STD_LOGIC; o: out STD_LOGIC); end component;
component INV port(i: in STD_LOGIC; o: out STD_LOGIC); end component;

begin
U1: INV port map(A(2), A2_L);          --invert most significant bit, used as selector
DEC1: v2to4dec                         --2-to-4 decoder
port map (en=>A2_L,
                  i1=>a(0),
                  i0=>a(1),
                  y0=>Y_TEMP(7),
                  y1=>Y_TEMP(6),
                  y2=>Y_TEMP(5),
                  y3=>Y_TEMP(4));
DEC2: v2to4dec                         --2-to-4 decoder
port map(en=>a(2),
        i1=>a(0),
                  i0=>a(1),
                  y0=>Y_TEMP(3),
                  y1=>Y_TEMP(2),
                  y2=>Y_TEMP(1),
                  y3=>Y_TEMP(0));
U2: AND2 port map(Y_TEMP(7), en, o(7));--select correct output based on decoder output and
enable bit
```

```
U3: AND2 port map(Y_TEMP(6), en, o(6));
U4: AND2 port map(Y_TEMP(5), en, o(5));
U5: AND2 port map(Y_TEMP(4), en, o(4));
U6: AND2 port map(Y_TEMP(3), en, o(3));
U7: AND2 port map(Y_TEMP(2), en, o(2));
U8: AND2 port map(Y_TEMP(1), en, o(1));
U9: AND2 port map(Y_TEMP(0), en, o(0));
end v3to8dec_arch;
```

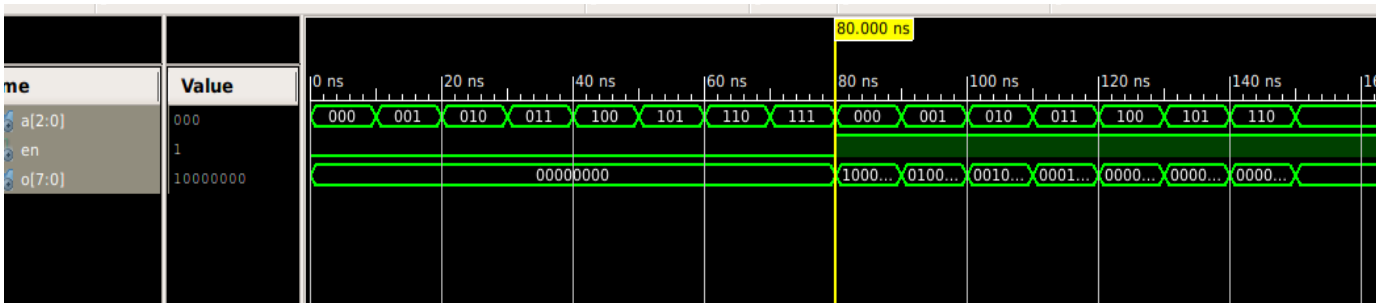
Testbench program

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY v3to8dec_tb IS
END v3to8dec_tb;

ARCHITECTURE structural OF v3to8dec_tb IS
        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT v3to8dec
        PORT(
        a : IN  std_logic_vector(2 downto 0);
        en : IN  std_logic;
        o : OUT  std_logic_vector(7 downto 0)
        );
        END COMPONENT;

  --Inputs
  signal a : std_logic_vector(2 downto 0) := (others => '0');
  signal en : std_logic := '0';
  --Outputs
  signal o : std_logic_vector(7 downto 0);

BEGIN
   -- Instantiate the Unit Under Test (UUT)
  uut: v3to8dec PORT MAP (
        a => a,
        en => en,
        o => o
        );
  -- Stimulus process
  stim_proc: process
  begin
        en<='0'; a<="000"; wait for 10 ns;
```

```
        en<='0'; a<="001"; wait for 10 ns;
        en<='0'; a<="010"; wait for 10 ns;
        en<='0'; a<="011"; wait for 10 ns;
        en<='0'; a<="100"; wait for 10 ns;
        en<='0'; a<="101"; wait for 10 ns;
        en<='0'; a<="110"; wait for 10 ns;
        en<='0'; a<="111"; wait for 10 ns;

        en<='1'; a<="000"; wait for 10 ns;
        en<='1'; a<="001"; wait for 10 ns;
        en<='1'; a<="010"; wait for 10 ns;
        en<='1'; a<="011"; wait for 10 ns;
        en<='1'; a<="100"; wait for 10 ns;
        en<='1'; a<="101"; wait for 10 ns;
        en<='1'; a<="110"; wait for 10 ns;
        en<='1'; a<="111"; wait for 10 ns;
       wait;
  end process;
END;
```

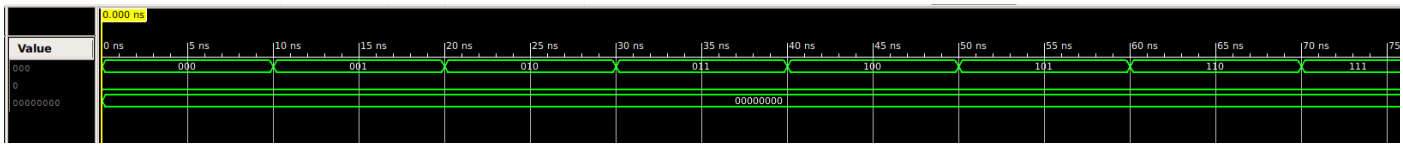
Waveform

Below Full wave form, the yellow vertical line separates the points where the enable bit is disabled from where the enable bit is enabled.

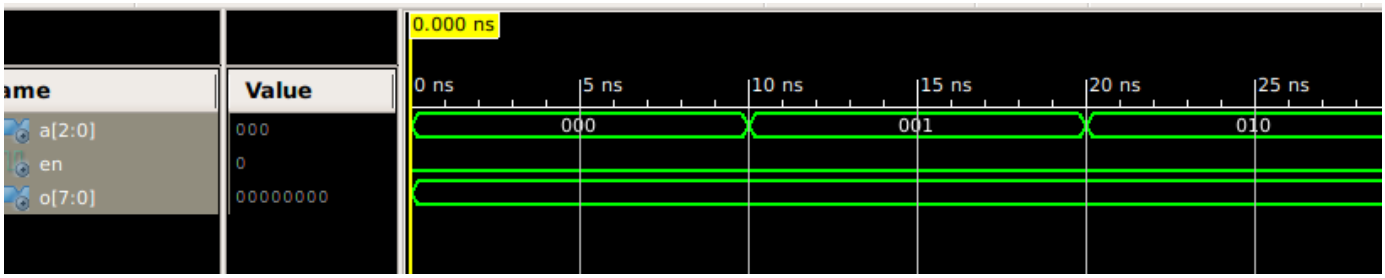


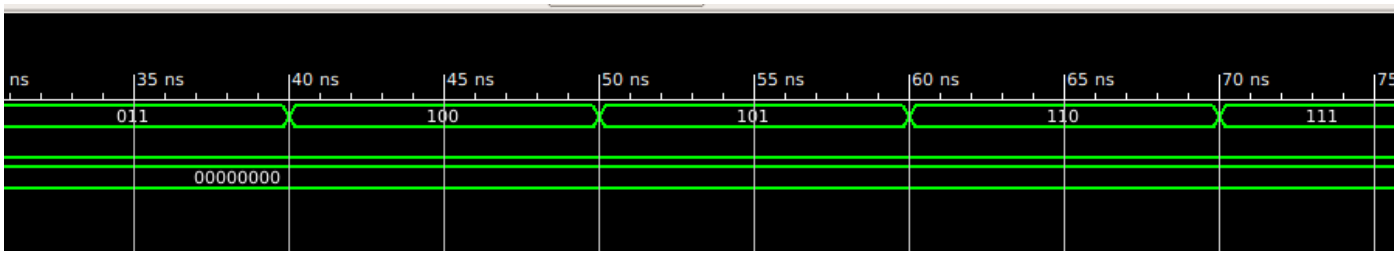
Below The enable bit is disabled

- We see that when the enable bit is disabled, the output that we receive is always “00000000”, or O_0



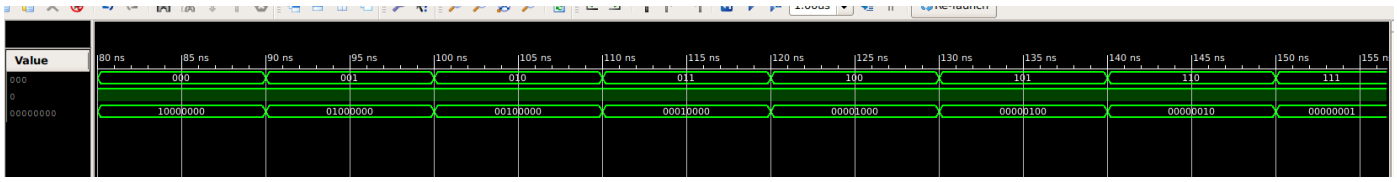
Below Some close ups when the enable bit is disabled:



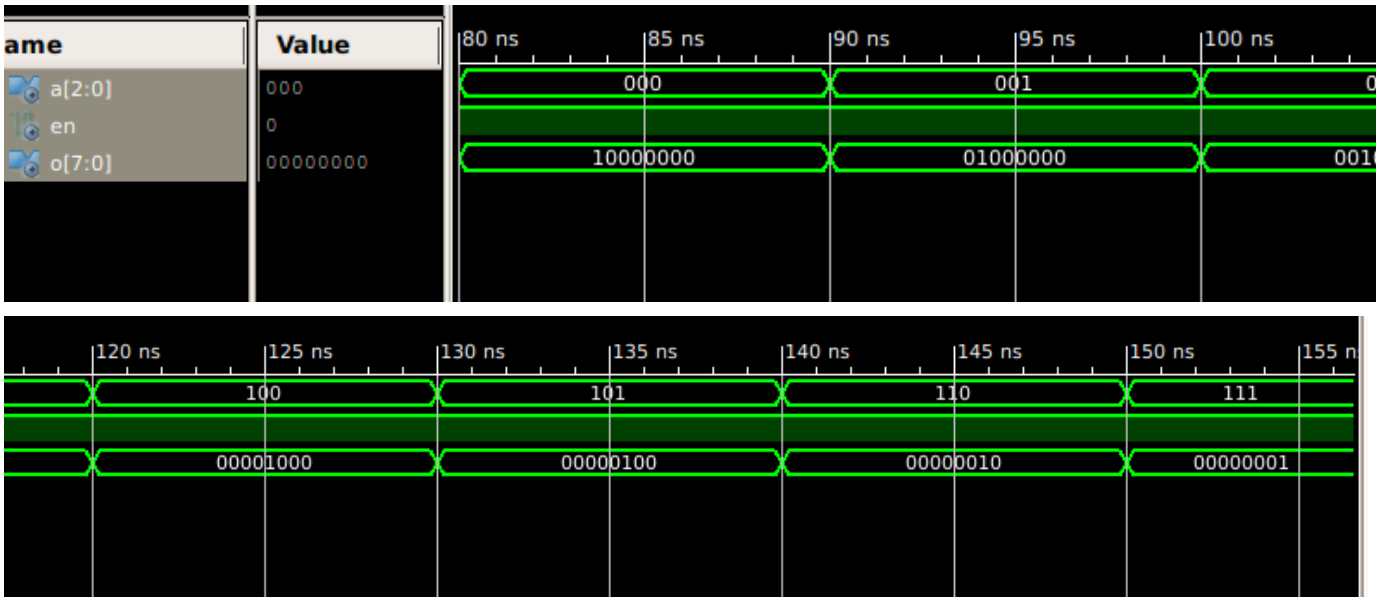


Below When the enable bit is enabled we see that the 3-to-8 decoder works as expected:

- When the input is 000, the output is 100000000, or O_0.
- When the input is 001, the output is 010000000, or O_1.
- When the input is 010, the output is 001000000, or O_2.
- …
- When the input is 110, the output is 00000010, or O_6.
- When the input is 111, the output is 00000001, or O_7.



Below Some close up screenshots of the waveform:

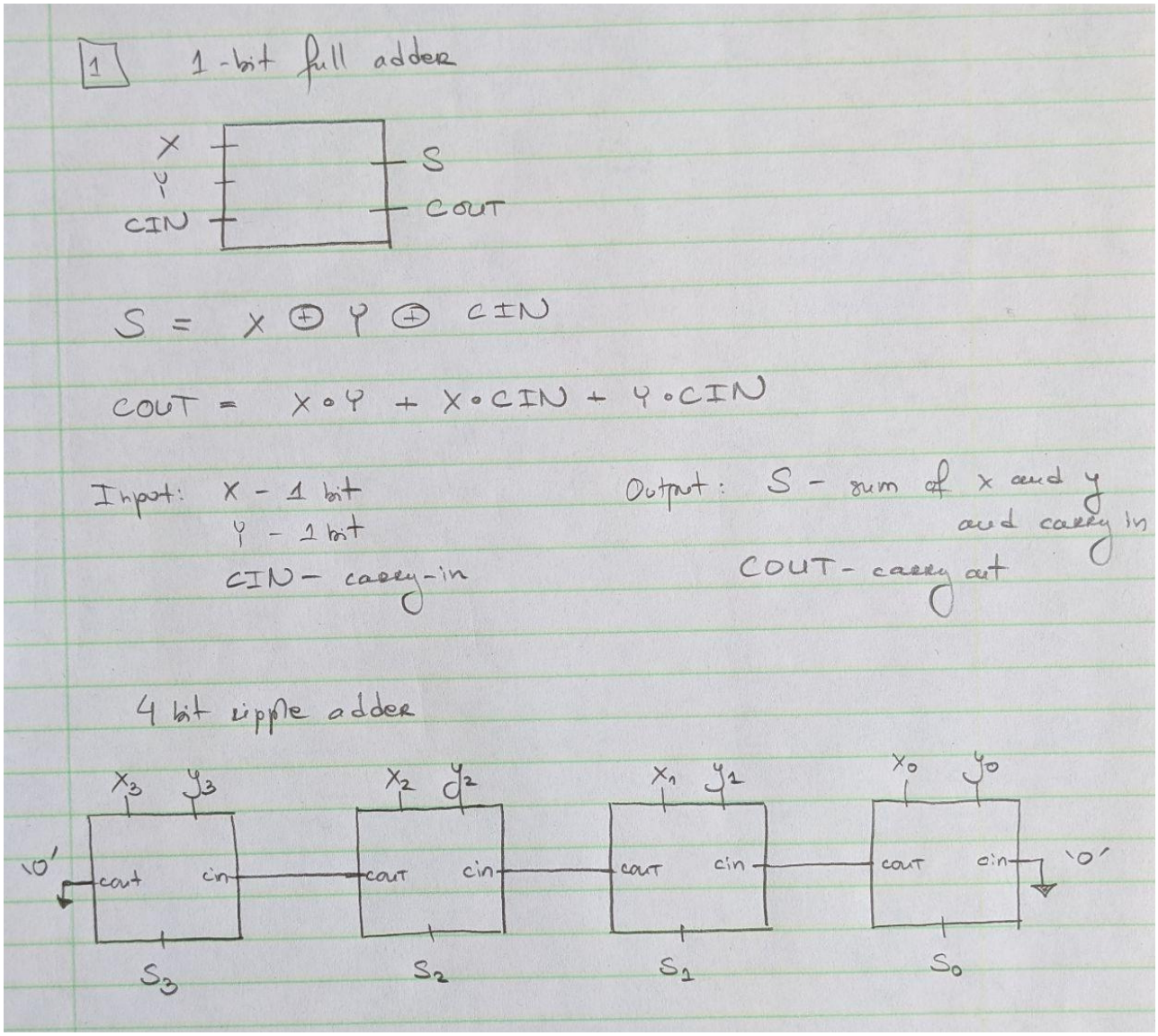


From the waveform we can see that the circuit is working as expected. When we input a particular 3-bit binary string, we are getting an 8-bit binary string with 1 at the correct position to signify what the binary number that was inputted was.

Task 3: 4-bit ripple adder

The 4-bit ripple adder uses 4 full 1-bit adders to add two 4-bit binary numbers correctly. It accounts for any carry in and carry out values up to 4-bit. The 4-bit ripple adder uses 4 1-bit full adders implemented separately. If there is a carry in in the least significant digit adder, the carry in is taken into account in the calculation. If there is a carry out in the most significant digit adder, the carry out is disregarded.

Circuit diagram



Entity declaration for 1-bit adder

```
entity adder1b is
        Port ( X : in  STD_LOGIC;
        Y : in  STD_LOGIC;
        CIN : in  STD_LOGIC;
        S : out  STD_LOGIC;
        COUT : out  STD_LOGIC);
end adder1b;
```

Architecture definition for 1-bit adder

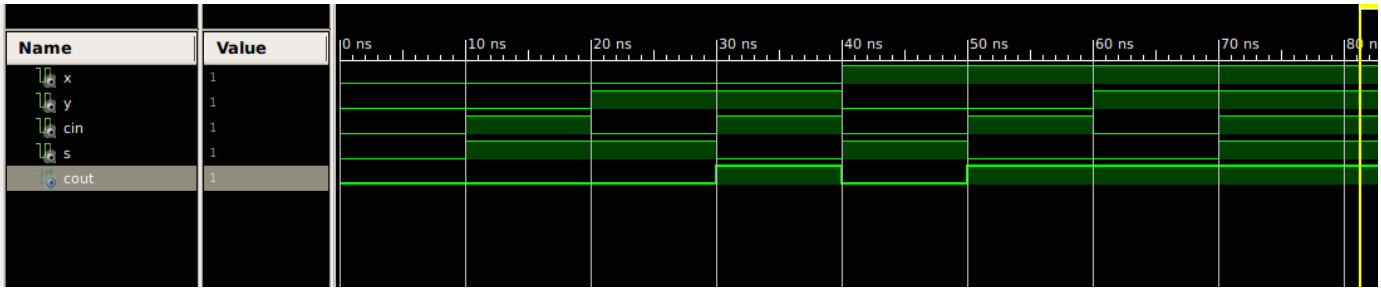
```
architecture adder1b_arch of adder1b is
begin
S <=X xor Y xor CIN;
COUT <=(X and Y) or (X and CIN) or (Y and CIN);
end adder1b_arch;
```

Testbench program for 1-bit adder

```
ENTITY adder1b_tb IS
END adder1b_tb;

ARCHITECTURE structural OF adder1b_tb IS
        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT adder1b
        PORT(
        X : IN  std_logic;
        Y : IN  std_logic;
        CIN : IN  std_logic;
        S : OUT  std_logic;
        COUT : OUT  std_logic
        );
        END COMPONENT;

  --Inputs
  signal X : std_logic := '0';
  signal Y : std_logic := '0';
  signal CIN : std_logic := '0';
  --Outputs
  signal S : std_logic;
  signal COUT : std_logic;

BEGIN
```

```
 -- Instantiate the Unit Under Test (UUT)
  uut: adder1b PORT MAP (
          X => X,
          Y => Y,
          CIN => CIN,
          S => S,
          COUT => COUT
          );

  -- Stimulus process
  stim_proc: process
  begin
          X<='0'; Y<='0'; CIN<='0'; wait for 10 ns;
           X<='0'; Y<='0'; CIN<='1'; wait for 10 ns;
           X<='0'; Y<='1'; CIN<='0'; wait for 10 ns;
           X<='0'; Y<='1'; CIN<='1'; wait for 10 ns;
           X<='1'; Y<='0'; CIN<='0'; wait for 10 ns;
           X<='1'; Y<='0'; CIN<='1'; wait for 10 ns;
           X<='1'; Y<='1'; CIN<='0'; wait for 10 ns;
           X<='1'; Y<='1'; CIN<='1'; wait for 10 ns;
          wait;
  end process;
END;
```

Waveform for 1-bit adder



Above We can see that the 1-bit full adder is adding two 1-bit binary digits correctly.

Entity declaration for 4-bit adder

```
entity adder4b is
          port ( X_4b : in  STD_LOGIC_VECTOR(3 downto 0);
          Y_4b : in  STD_LOGIC_VECTOR(3 downto 0);
          S_4b : out  STD_LOGIC_VECTOR(3 downto 0);
          CIN_4b : in  STD_LOGIC;
```

```
        COUT_4b : out  STD_LOGIC);
end adder4b;
```

Architecture definition for 4-bit adder

```
architecture adder4b_arch of adder4b is

signal CIN_1, CIN_2, CIN_3: STD_LOGIC;              --intermediary variables
component adder1b port(X, Y, CIN: in STD_LOGIC; COUT, S: out STD_LOGIC); end
component;

begin
ADDER0: adder1b
port map (X=> X_4b(0),
        Y=> Y_4b(0),
                CIN=> CIN_4b,
                S=> S_4b(0),
                COUT=> CIN_1);
ADDER1: adder1b
port map (X=> X_4b(1),
        Y=> Y_4b(1),
                CIN=> CIN_1,
                S=> S_4b(1),
                COUT=> CIN_2);
ADDER2: adder1b
port map (X=> X_4b(2),
        Y=> Y_4b(2),
                CIN=> CIN_2,
                S=> S_4b(2),
                COUT=> CIN_3);
ADDER3: adder1b
port map (X=> X_4b(3),
        Y=> Y_4b(3),
                CIN=> CIN_3,
                S=> S_4b(3),
                COUT=> COUT_4b);
end adder4b_arch;
```

Testbench program for 4-bit adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
```

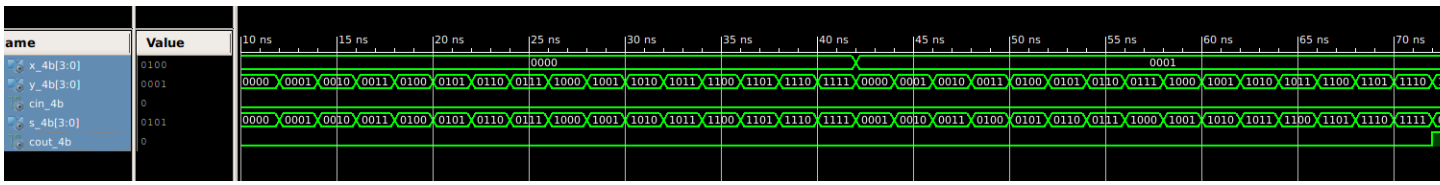
```
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY adder4b_tb IS
END adder4b_tb;

ARCHITECTURE behavior OF adder4b_tb IS
        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT adder4b
        PORT(
        X_4b : IN  std_logic_vector(3 downto 0);
        Y_4b : IN  std_logic_vector(3 downto 0);
        S_4b : OUT  std_logic_vector(3 downto 0);
        CIN_4b : IN  std_logic;
        COUT_4b : OUT  std_logic
        );
        END COMPONENT;

  --Inputs
  signal X_4b : std_logic_vector(3 downto 0) := (others => '0');
  signal Y_4b : std_logic_vector(3 downto 0) := (others => '0');
  signal CIN_4b : std_logic := '0';

  --Outputs
  signal S_4b : std_logic_vector(3 downto 0);
  signal COUT_4b : std_logic;

BEGIN
   -- Instantiate the Unit Under Test (UUT)
  uut: adder4b PORT MAP (
        X_4b => X_4b,
        Y_4b => Y_4b,
        S_4b => S_4b,
        CIN_4b => CIN_4b,
        COUT_4b => COUT_4b
        );

  -- Stimulus process
  stim_proc: process
  begin
         wait for 10 ns;
         CIN_4b<='0';
         for i in 0 to 15 loop
                X_4b<= std_logic_vector(to_unsigned(i, 4));
```

```
                for j in 0 to 15 loop
                        Y_4b<= std_logic_vector(to_unsigned(j, 4));
                        wait for 2 ns;
                end loop;
        end loop;

        CIN_4b <='1';
        for i in 0 to 15 loop
                X_4b <= std_logic_vector(to_unsigned(i, 4));
                for j in 0 to 15 loop
                        Y_4b<= std_logic_vector(to_unsigned(j, 4));
                        wait for 2 ns;
                end loop;
        end loop;
        wait;
  end process;

END;
```

Waveform for 4-bit adder

Since the waveform is quite large to be examined due to the exhaustive testbench program that tests all possible calculations in the presence and absence of a carry-in in a single screenshot, below are selected examples from the wave form.

Below This is part of the waveform that shows adding 0000 together with all possible combinations of 4-bit binary numbers when there is no carry in and part of the addition of 0001 and all possible 4-bit binary numbers when there is no carry in.



Below We can see below that we are adding 0101 to all possible combinations of 4-bit binary numbers when there is no carry in.

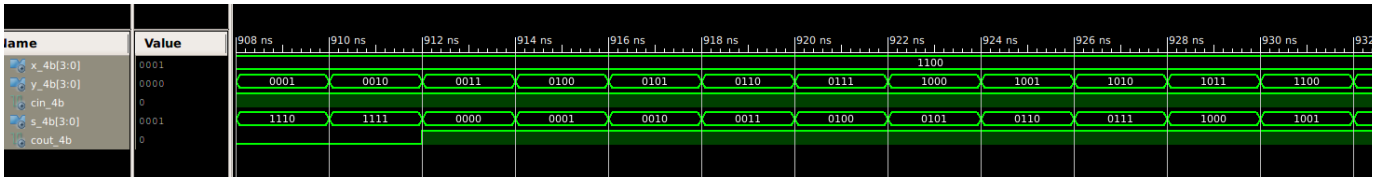
If we trace the behavior of the circuit we can be assured that the 4-bit ripple adder works correctly.

- 0101 + 0110 = 1011 when there is no inputted carry in and we do not get a carry out.
- 0101 + 1011 = 0000 when there is no inputted carry in and we do get a carry out.

Below We can see below that we are adding 1100 to all possible combinations of 4-bit binary numbers when there is a carry in.

If we trace the behavior of the circuit we can be assured that the 4-bit ripple adder works correctly.

- 1100 + 0010 = 1111 when there is inputted carry in and we do not get a carry out.
- 1100 + 1011 = 1000 when there is inputted carry in and we do get a carry out.



From the above examples, we can be assured that the circuit works as expected.