EE671: Assignment 4

Anubhav Bhatla, 200070008

October 21, 2022

1 Components

The adder needs logic functions AND, XOR, A + B.C and A.B + C.(A+B) to compute different orders of G, P and final sum and carry outputs. The delays that I used for each of these functions is given below:

Function	Delay (in ps)
AND	48
A + B.C	66
XOR	76
A.B + C.(A+B)	76

The code that I wrote for these functions (**components.vhd**) is given below:

```
library IEEE;
use IEEE.std_logic_1164.all;
-- simple gates with trivial architectures
package gates is
-- A.B
component andgate is
port (
A, B: in std_ulogic;
prod: out std_ulogic
);
end component andgate;
-- A XOR B
component xorgate is
```

```
port (
A, B: in std_ulogic;
uneq: out std_ulogic
);
end component xorgate;
-- A + B.C
component abcgate is
port (
A, B, C: in std_ulogic;
abc: out std_ulogic
);
end component abcgate;
-- A.B + C.(A + B)
component Cin_map_G is
port(
A, B, Cin: in std_ulogic;
Bit0_G: out std_ulogic
);
end component Cin_map_G;
end package gates;

library IEEE;
use IEEE.std_logic_1164.all;
-- A.B
entity andgate is
port (
A, B: in std_ulogic;
prod: out std_ulogic
);
end entity andgate;
architecture trivial of andgate is
begin
prod <= A AND B AFTER 48 ps;
end architecture trivial;

library IEEE;
use IEEE.std_logic_1164.all;
-- A XOR B
```

```
entity xorgate is
port (
A, B: in std_ulogic;
uneq: out std_ulogic
);
end entity xorgate;
architecture trivial of xorgate is
begin
uneq <= A XOR B AFTER 76 ps;
end architecture trivial;

library IEEE;
use IEEE.std_logic_1164.all;
-- A + B.C
entity abcgate is
port (
A, B, C: in std_ulogic;
abc: out std_ulogic
);
end entity abcgate;
architecture trivial of abcgate is
begin
abc <= A OR (B AND C) AFTER 66 ps;
end architecture trivial;

library IEEE;
use IEEE.std_logic_1164.all;
-- A.B + C.(A + B)
entity Cin_map_G is
port(
A, B, Cin: in std_ulogic;
Bit0_G: out std_ulogic
);
end entity Cin_map_G;
architecture trivial of Cin_map_G is
begin
Bit0_G <= (A AND B) OR (Cin AND (A OR B)) AFTER 76 ps;
end architecture trivial;
```

2 Design & Working

Given below is the slot-wise breakdown of how different order G and P values are calculated along with the Carry and Sum bits:

2.1 Slot-1

- We directly insert C_0 as C_{in} as this is the input carry to the adder.
- We next calculate 0-th order G and P values using $G_i^0 = a_i \cdot b_i$ and $P_i^0 = a_i \oplus b_i$
- But for G_0^0 (right-most block) we calculate it using the available value of C_0 to compute the output carry directly using $G_0^0 = a_i \cdot b_i + C_0 \cdot (a_i + b_i)$
- Now G_0^0 represents C_1

2.2 Slot-2

- In order to compute first order G and P values, we use the following relations: $G_{i:i-1}^1 = G_i^0 + P_i^0 \cdot G_{i-1}^0$ and $P_{i:i-1}^1 = P_i^0 \cdot P_{i-1}^0$
- $G_{1:0}^1$ now represents C_2
- Since we already generated C_0 and C_1 in the previous slot along with P_1^0 and P_0^0, we can now generate the sum using $S_i = P_i^0 \oplus C_i$ for $i = 0, 1$

2.3 Slot-3

- Next we compute the second order G and P values using the following relations: $G_{i:i-3}^2 = G_{i:i-1}^1 + P_{i:i-1}^1 \cdot G_{i-2:i-3}^1$ and $P_{i:i-3}^2 = P_{i:i-1}^1 \cdot P_{i-2:i-3}^1$
- $G_{3:0}^2$ now represents C_4
- We can calculate C_3 using $C_3 = G_2^0 + P_2^0 \cdot C_2$

- Since we had C_2 available from the previous slot, we calculate S_2 using $S_2 = P_2^0 \oplus C_2$

2.4 Slot-4

- We compute the third order G and P values using the following relations: $G_{i:i-7}^3 = G_{i:i-3}^2 + P_{i:i-3}^2 \cdot G_{i-4:i-7}^2$ and $P_{i:i-7}^3 = P_{i:i-3}^2 \cdot P_{i-4:i-7}^2$
- $G_{7:0}^3$ now represents C_8
- Since we obtained C_4 in the previous slot, we can compute C_5 and C_6 using $C_5 = G_4^0 + P_4^0 \cdot C_4$ and $C_6 = G_{5:4}^1 + P_{5:4}^1 \cdot C_4$
- We can also compute the sums as $S_i = P_i^0 \oplus C_i$ for $i = 3, 4$

2.5 Slot-5

- We compute the fourth order G and P values using the following relations: $G_{15:0}^4 = G_{15:8}^3 + P_{15:8}^3 \cdot G_{7:0}^3$ and $P_{15:0}^4 = P_{15:8}^3 \cdot P_{7:0}^3$
- $G_{15:0}^4$ now represents C_{16}
- We can also obtain the following carry bits: $C_7 = G_6^0 + P_6^0 \cdot C_6$, $C_9 = G_8^0 + P_8^0 \cdot C_8$, $C_{10} = G_{9:8}^1 + P_{9:8}^1 \cdot C_8$ and $C_{12} = G_{11:8}^2 + P_{11:8}^2 \cdot C_8$
- We can also compute the sums as $S_i = P_i^0 \oplus C_i$ for $i = 5, 6, 8$

2.6 Slot-6

- We can now obtain the following carry bits: $C_{11} = G_{10}^0 + P_{10}^0 \cdot C_{10}$, $C_{13} = G_{12}^0 + P_{12}^0 \cdot C_{12}$ and $C_{14} = G_{13:12}^1 + P_{13:12}^1 \cdot C_{12}$
- We can also compute the sums as $S_i = P_i^0 \oplus C_i$ for $i = 7, 9, 10, 12$

2.7 Slot-7

- We can now obtain the final carry bit $C_{15} = G_{14}^0 + P_{14}^0 \cdot C_{14}$
- We can also compute the sums as $S_i = P_i^0 \oplus C_i$ for $i = 11, 13, 14$

2.8 Slot-8

- We now compute the final sum as $S_{15} = P_{15}^0 \oplus C_{15}$

Given below is the VHDL code (**bkadder.vhd**) for implementing the Brent Kung using the Design given above:

```
library IEEE;
use IEEE.std_logic_1164.all;
library work;
use work.gates.all;

entity bk_adder is
port (
A, B: in std_logic_vector(15 downto 0);
Cin: in std_logic;
S: out std_logic_vector(15 downto 0);
Cout: out std_logic
);
end entity bk_adder;

architecture arch of bk_adder is
signal G0,P0: std_logic_vector(15 downto 0);
signal G1,P1: std_logic_vector(7 downto 0);
signal G2,P2: std_logic_vector(3 downto 0);
signal G3,P3: std_logic_vector(1 downto 0);
signal G4,P4: std_logic;
signal C: std_logic_vector(15 downto 0);
```

```
begin
C(0) <= Cin;

--Slot 1 --0th order Gs and Ps

G_0: Cin_map_G port map (A => A(0), B => B(0), Cin => C(0),
Bit0_G => G0(0)); --this represents C1
P_0: xorgate port map (A => A(0), B => B(0), uneq => P0(0));
G_1: andgate port map (A => A(1), B => B(1), prod => G0(1));
P_1: xorgate port map (A => A(1), B => B(1), uneq => P0(1));
G_2: andgate port map (A => A(2), B => B(2), prod => G0(2));
P_2: xorgate port map (A => A(2), B => B(2), uneq => P0(2));
G_3: andgate port map (A => A(3), B => B(3), prod => G0(3));
P_3: xorgate port map (A => A(3), B => B(3), uneq => P0(3));
G_4: andgate port map (A => A(4), B => B(4), prod => G0(4));
P_4: xorgate port map (A => A(4), B => B(4), uneq => P0(4));
G_5: andgate port map (A => A(5), B => B(5), prod => G0(5));
P_5: xorgate port map (A => A(5), B => B(5), uneq => P0(5));
G_6: andgate port map (A => A(6), B => B(6), prod => G0(6));
P_6: xorgate port map (A => A(6), B => B(6), uneq => P0(6));
G_7: andgate port map (A => A(7), B => B(7), prod => G0(7));
P_7: xorgate port map (A => A(7), B => B(7), uneq => P0(7));
G_8: andgate port map (A => A(8), B => B(8), prod => G0(8));
P_8: xorgate port map (A => A(8), B => B(8), uneq => P0(8));
G_9: andgate port map (A => A(9), B => B(9), prod => G0(9));
P_9: xorgate port map (A => A(9), B => B(9), uneq => P0(9));
G_10: andgate port map (A => A(10), B => B(10), prod => G0(10));
P_10: xorgate port map (A => A(10), B => B(10), uneq => P0(10));
G_11: andgate port map (A => A(11), B => B(11), prod => G0(11));
P_11: xorgate port map (A => A(11), B => B(11), uneq => P0(11));
G_12: andgate port map (A => A(12), B => B(12), prod => G0(12));
P_12: xorgate port map (A => A(12), B => B(12), uneq => P0(12));
G_13: andgate port map (A => A(13), B => B(13), prod => G0(13));
P_13: xorgate port map (A => A(13), B => B(13), uneq => P0(13));
G_14: andgate port map (A => A(14), B => B(14), prod => G0(14));
P_14: xorgate port map (A => A(14), B => B(14), uneq => P0(14));
G_15: andgate port map (A => A(15), B => B(15), prod => G0(15));
P_15: xorgate port map (A => A(15), B => B(15), uneq => P0(15));
```

```
--C1
C(1) <= G0(0);

--Slot 2 --1st order Gs and Ps

G_1_0: abcgate port map (A => G0(1), B => P0(1), C => G0(0),
abc => G1(0)); --this represents C2
P_1_0: andgate port map (A => P0(1), B => P0(0), prod => P1(0));
G_3_2: abcgate port map (A => G0(3), B => P0(3), C => G0(2),
abc => G1(1));
P_3_2: andgate port map (A => P0(3), B => P0(2), prod => P1(1));
G_5_4: abcgate port map (A => G0(5), B => P0(5), C => G0(4),
abc => G1(2));
P_5_4: andgate port map (A => P0(5), B => P0(4), prod => P1(2));
G_7_6: abcgate port map (A => G0(7), B => P0(7), C => G0(6),
abc => G1(3));
P_7_6: andgate port map (A => P0(7), B => P0(6), prod => P1(3));
G_9_8: abcgate port map (A => G0(9), B => P0(9), C => G0(8),
abc => G1(4));
P_9_8: andgate port map (A => P0(9), B => P0(8), prod => P1(4));
G_11_10: abcgate port map (A => G0(11), B => P0(11), C => G0(10),
abc => G1(5));
P_11_10: andgate port map (A => P0(11), B => P0(10), prod => P1(5));
G_13_12: abcgate port map (A => G0(13), B => P0(13), C => G0(12),
abc => G1(6));
P_13_12: andgate port map (A => P0(13), B => P0(12), prod => P1(6));
G_15_14: abcgate port map (A => G0(15), B => P0(15), C => G0(14),
abc => G1(7));
P_15_14: andgate port map (A => P0(15), B => P0(14), prod => P1(7));
--C2
C(2) <= G1(0);
--S0
S_0: xorgate port map (A => P0(0), B => C(0), uneq => S(0));
--S1
S_1: xorgate port map (A => P0(1), B => C(1), uneq => S(1));

--Slot 3 --2nd order Gs and Ps
```

```
G_3_0: abcgate port map (A => G1(1), B => P1(1), C => G1(0),
abc => G2(0)); --this represents C4
P_3_0: andgate port map (A => P1(1), B => P1(0), prod => P2(0));
G_7_4: abcgate port map (A => G1(3), B => P1(3), C => G1(2),
abc => G2(1));
P_7_4: andgate port map (A => P1(3), B => P1(2), prod => P2(1));
G_11_8: abcgate port map (A => G1(5), B => P1(5), C => G1(4),
abc => G2(2));
P_11_8: andgate port map (A => P1(5), B => P1(4), prod => P2(2));
G_15_12: abcgate port map (A => G1(7), B => P1(7), C => G1(6),
abc => G2(3));
P_15_12: andgate port map (A => P1(7), B => P1(6), prod => P2(3));
--C3
C_3: abcgate port map (A => G0(2), B => P0(2), C => C(2),
abc => C(3));
--C4
C(4) <= G2(0);
--S2
S_2: xorgate port map (A => P0(2), B => C(2), uneq => S(2));

--Slot 4 --3rd order Gs and Ps

G_7_0: abcgate port map (A => G2(1), B => P2(1), C => G2(0),
abc => G3(0)); --this represents C8
P_7_0: andgate port map (A => P2(1), B => P2(0), prod => P3(0));
G_15_8: abcgate port map (A => G2(3), B => P2(3), C => G2(2),
abc => G3(1));
P_15_8: andgate port map (A => P2(3), B => P2(2), prod => P3(1));
--C5
C_5: abcgate port map (A => G0(4), B => P0(4), C => C(4),
abc => C(5));
--C6
C_6: abcgate port map (A => G1(2), B => P1(2), C => C(4),
abc => C(6));
--C8
C(8) <= G3(0);
--S3
S_3: xorgate port map (A => P0(3), B => C(3), uneq => S(3));
```

```
--S4
S_4: xorgate port map (A => P0(4), B => C(4), uneq => S(4));

--Slot 5 --4th order G and P

G_15_0: abcgate port map (A => G3(1), B => P3(1), C => G3(0),
abc => G4); --this represents C16
P_15_0: andgate port map (A => P3(1), B => P3(0), prod => P4);
--C7
C_7: abcgate port map (A => G0(6), B => P0(6), C => C(6),
abc => C(7));
--C9
C_9: abcgate port map (A => G0(8), B => P0(8), C => C(8),
abc => C(9));
--C10
C_10: abcgate port map (A => G1(4), B => P1(4), C => C(8),
abc => C(10));
--C12
C_12: abcgate port map (A => G2(2), B => P2(2), C => C(8),
abc => C(12));
--C16
Cout <= G4;
--S5
S_5: xorgate port map (A => P0(5), B => C(5), uneq => S(5));
--S6
S_6: xorgate port map (A => P0(6), B => C(6), uneq => S(6));
--S8
S_8: xorgate port map (A => P0(8), B => C(8), uneq => S(8));

--Slot 6

--C11
C_11: abcgate port map (A => G0(10), B => P0(10), C => C(10),
abc => C(11));
--C13
C_13: abcgate port map (A => G0(12), B => P0(12), C => C(12),
abc => C(13));
--C14
```

```
C_14: abcgate port map (A => G1(6), B => P1(6), C => C(12),
abc => C(14));
--S7
S_7: xorgate port map (A => P0(7), B => C(7), uneq => S(7));
--S9
S_9: xorgate port map (A => P0(9), B => C(9), uneq => S(9));
--S10
S_10: xorgate port map (A => P0(10), B => C(10), uneq => S(10));
--S12
S_12: xorgate port map (A => P0(12), B => C(12), uneq => S(12));

--Slot 7

--C15
C_15: abcgate port map (A => G0(14), B => P0(14), C => C(14),
abc => C(15));
--S11
S_11: xorgate port map (A => P0(11), B => C(11), uneq => S(11));
--S13
S_13: xorgate port map (A => P0(13), B => C(13), uneq => S(13));
--S14
S_14: xorgate port map (A => P0(14), B => C(14), uneq => S(14));

--Slot 8

--S15
S_15: xorgate port map (A => P0(15), B => C(15), uneq => S(15));

end architecture arch;
```

3 Simulation and Testing

In order to test my design, I wrote the following two Testbenches: the first one (**my_testbench.vhdl**) is not capable of reading inputs and outputs from a text file and they need to be provided in the file itself.

```
library std;
```

```
use std.textio.all;
library ieee;
use ieee.std_logic_1164.all;

entity Testbench is
end entity;

architecture Behave of Testbench is
  component DUT is
   port(input_vector: in std_logic_vector(32 downto 0);
        output_vector: out std_logic_vector(16 downto 0));
  end component;
  signal input_vector  : std_logic_vector(32 downto 0);
  signal output_vector : std_logic_vector(16 downto 0);

begin
    dut_instance: DUT
      port map(input_vector => input_vector, output_vector =>
      output_vector);
    process
        variable err_flag : boolean := false;
    begin
        --TEST CASE - 1
        input_vector <= '1' & "0101010101010101" & "1010101010101010";
        wait for 10 ns;
        if (output_vector /= ('1' & "0000000000000000")) then
            err_flag := true;
        end if;
        assert (not err_flag) report "FAILURE, test case - 1 failed."
        severity error;
        wait for 4 ns;

        --TEST CASE - 2
        input_vector <= '1' & "0000000000000000" & "0000000000000001";
        wait for 10 ns;
        err_flag := false;
        if (output_vector /= ('0' & "0000000000000010")) then
            err_flag := true;
```

```
end if;
assert (not err_flag) report "FAILURE, test case - 2 failed."
severity error;
wait for 4 ns;

--TEST CASE - 3
input_vector <= '1' & "1111111111111111" & "1111111111111111";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('1' & "1111111111111111")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 3 failed."
severity error;
wait for 4 ns;

--TEST CASE - 4
input_vector <= '0' & "0100100011000010" & "1000101100111011";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('0' & "1101001111111101")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 4 failed."
severity error;
wait for 4 ns;

--TEST CASE - 5
input_vector <= '1' & "1000000000010101" & "1011100111101011";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('1' & "0011101000000001")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 5 failed."
severity error;
wait for 4 ns;
```

```
--TEST CASE - 6
input_vector <= '0' & "1000010110101011" & "1111100110101000";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('1' & "0111111101010011")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 6 failed."
severity error;
wait for 4 ns;

--TEST CASE - 7
input_vector <= '1' & "1010110110011000" & "1100111001010001";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('1' & "0111101111101010")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 7 failed."
severity error;
wait for 4 ns;

--TEST CASE - 8
input_vector <= '0' & "1001011101110000" & "0010110010110001";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('0' & "1100010000100001")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 8 failed."
severity error;
wait for 4 ns;

--TEST CASE - 9
input_vector <= '1' & "0001001000001111" & "0001101011010011";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('0' & "0010110011100011")) then
```

```
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 9 failed."
severity error;
wait for 4 ns;

--TEST CASE - 10
input_vector <= '0' & "0100101101101111" & "0100101101101111";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('0' & "1001011011011110")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 10 failed."
severity error;
wait for 4 ns;

--TEST CASE - 11
input_vector <= '1' & "1111100100110011" & "0000111110011101";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('1' & "0000100011010001")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 11 failed."
severity error;
wait for 4 ns;

--TEST CASE - 12
input_vector <= '0' & "0001011000001000" & "1010100011101110";
wait for 10 ns;
err_flag := false;
if (output_vector /= ('0' & "1011111011110110")) then
    err_flag := true;
end if;
assert (not err_flag) report "FAILURE, test case - 12 failed."
severity error;
wait for 4 ns;
```

```
        --TEST CASE - 13
        input_vector <= '1' & "0011010100111111" & "0100000011111110";
        wait for 10 ns;
        err_flag := false;
        if (output_vector /= ('0' & "0111011000111110")) then
            err_flag := true;
        end if;
        assert (not err_flag) report "FAILURE, test case - 13 failed."
        severity error;
        wait for 4 ns;

        --TEST CASE - 14
        input_vector <= '0' & "1111101010101111" & "0010011100011010";
        wait for 10 ns;
        err_flag := false;
        if (output_vector /= ('1' & "0010000111001001")) then
            err_flag := true;
        end if;
        assert (not err_flag) report "FAILURE, test case - 14 failed."
        severity error;

        report "Testbench finished successfully." severity note;

        wait;
    end process;

end Behave;
```

The second testbench (**Testbench.vhdl**) is capable of reading inputs, outputs and mask bits from an external file called TRACEFILE.txt:

```
library std;
use std.textio.all;
library ieee;
use ieee.std_logic_1164.all;

entity Testbench is
```

```
end entity;

architecture Behave of Testbench is
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  constant number_of_inputs  : integer := 33;  -- # input bits
  constant number_of_outputs : integer := 17;  -- # output bits
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  component DUT is
   port(input_vector: in std_logic_vector(number_of_inputs-1 downto 0);
        output_vector: out std_logic_vector(number_of_outputs-1
        downto 0));
  end component;
  signal input_vector  : std_logic_vector(number_of_inputs-1 downto 0);
  signal output_vector : std_logic_vector(number_of_outputs-1 downto 0);

  -- create a constrained string
  function to_string(x: string) return string is
      variable ret_val: string(1 to x'length);
      alias lx : string (1 to x'length) is x;
  begin
      ret_val := lx;
      return(ret_val);
  end to_string;

  -- bit-vector to std-logic-vector and vice-versa
  function to_std_logic_vector(x: bit_vector) return std_logic_vector is
     alias lx: bit_vector(1 to x'length) is x;
     variable ret_val: std_logic_vector(1 to x'length);
  begin
     for I in 1 to x'length loop
        if(lx(I) = '1') then
          ret_val(I) := '1';
        else
          ret_val(I) := '0';
        end if;
     end loop;
```

```
    return ret_val;
  end to_std_logic_vector;

  function to_bit_vector(x: std_logic_vector) return bit_vector is
    alias lx: std_logic_vector(1 to x'length) is x;
    variable ret_val: bit_vector(1 to x'length);
  begin
    for I in 1 to x'length loop
      if(lx(I) = '1') then
        ret_val(I) := '1';
      else
        ret_val(I) := '0';
      end if;
    end loop;
    return ret_val;
  end to_bit_vector;

begin
  process
    variable err_flag : boolean := false;
    File INFILE: text open read_mode is "TRACEFILE.txt";
    FILE OUTFILE: text  open write_mode is "outputs.txt";

    -- bit-vectors are read from the file.
    variable input_vector_var: bit_vector (number_of_inputs-1 downto 0);
    variable output_vector_var: bit_vector (number_of_outputs-1 downto 0);
    variable output_mask_var: bit_vector (number_of_outputs-1 downto 0);

    -- for comparison of output with expected-output
    variable output_comp_var: std_logic_vector (number_of_outputs-1
    downto 0);
    constant ZZZZ : std_logic_vector(number_of_outputs-1 downto 0) :=
    (others => '0');

    -- for read/write.
    variable INPUT_LINE: Line;
    variable OUTPUT_LINE: Line;
    variable LINE_COUNT: integer := 0;
```

```
  begin
    while not endfile(INFILE) loop
  -- will read a new line every 44ns, apply input,
  -- wait for 40 ns for circuit to settle.
  -- read output.
          LINE_COUNT := LINE_COUNT + 1;
  -- read input at current time.
  readLine (INFILE, INPUT_LINE);
          read (INPUT_LINE, input_vector_var);
          read (INPUT_LINE, output_vector_var);
          read (INPUT_LINE, output_mask_var);
  -- apply input.
          input_vector <= to_std_logic_vector(input_vector_var);
  -- wait for the circuit to settle
  wait for 40 ns;
  -- check output.
          output_comp_var := (to_std_logic_vector(output_mask_var) and
(output_vector xor to_std_logic_vector(output_vector_var)));
  if (output_comp_var  /= ZZZZ) then
             write(OUTPUT_LINE,to_string("ERROR: line "));
             write(OUTPUT_LINE, LINE_COUNT);
             writeline(OUTFILE, OUTPUT_LINE);
             err_flag := true;
          end if;
          write(OUTPUT_LINE, to_bit_vector(input_vector));
          write(OUTPUT_LINE, to_string(" "));
          write(OUTPUT_LINE, to_bit_vector(output_vector));
          writeline(OUTFILE, OUTPUT_LINE);
  -- advance time by 4 ns.
  wait for 4 ns;
    end loop;

    assert (err_flag) report "SUCCESS, all tests passed." severity note;
    assert (not err_flag) report "FAILURE, some tests failed."
    severity error;

    wait;
```

```
  end process;

  dut_instance: DUT
      port map(input_vector => input_vector, output_vector =>
      output_vector);

end Behave;
```

The file containing the inputs and outputs (**TRACEFILE.txt**) is given below:

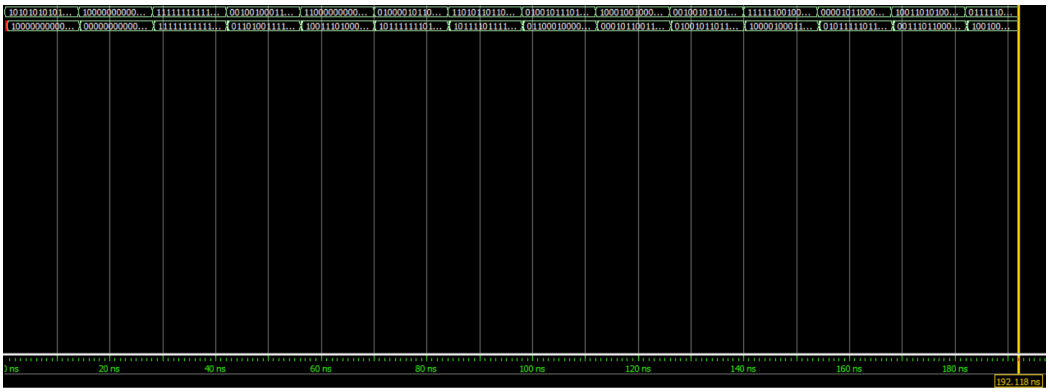
```
11111111111111111111111111111111 11111111111111111 11111111111111111
10000000000000000000000000000001 00000000000000010 11111111111111111
10101010101010101101010101010101 10000000000000000 11111111111111111
00100100011000010100010110011101 10110100111111101 11111111111111111
11000000000010101101110011101011 10011101000000001 11111111111111111
01000010110101011111110011010100 10111111101010011 11111111111111111
11010110110011000110011100101001 10111101111101010 11111111111111111
01001011101110000001011001011001 10100010000100001 11111111111111111
10001001000001111000110101101001 10010110011100011 11111111111111111
00100101101101111010010110110111 10010110110111110 11111111111111111
11111100100110011000011111001110 1 0000100011010001 11111111111111111
00001011000001000101010001110111 0 10111111011110110 11111111111111111
10011010100111111010000011111110 0 0111011000111110 11111111111111111
01111101010101111001001110001101 0 10010000111001001 11111111111111111
```

In order to test my design, I will be using the first testbench as we have a limited number of inputs to test.
Given below is the screenshot of the RTL Simulation output showing that none of the testcases failed:

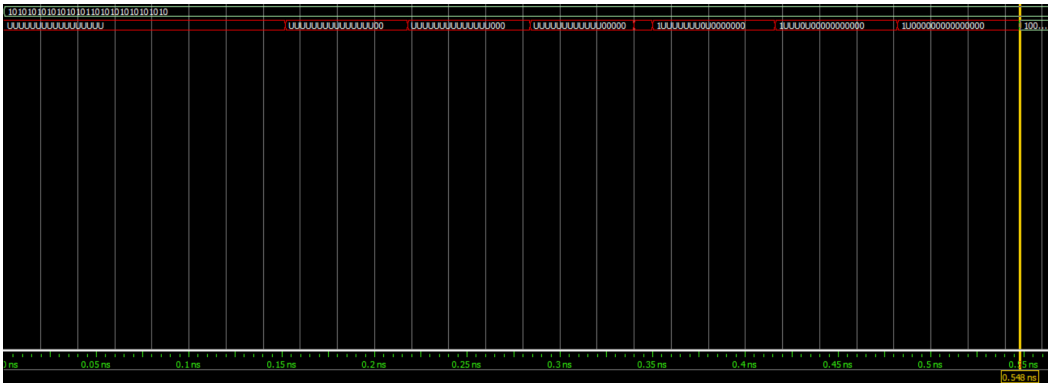
```
# run -all
# ** Note: Testbench finished successfully.
#    Time: 192 ns  Iteration: 0  Instance: /testbench

VSIM 2>
```

Given below is the screenshot of the RTL Simulation showing the input and output waveforms:



Given below is the zoomed in view of the delay between the time when we apply the first input and when we get the output:



The theoretical delay for the Brent Kung can be calculated by considering the critical path which would be the generation of S_{15}. In order to obtain S_{15}, we need to pass through 1 XOR gate, 6 A+B.C gates and 1 A.B+C.(A+B) gate which leads to a total theoretical delay of $76 + 66 * 6 + 76ps = 548ps$. This theoretical delay matches with the observed delay in the RTL Simulation output.