EE671: Assignment 5

Anubhav Bhatla, 200070008

November 2, 2022

1 Components

The same components as the previous assignment have been referenced for making the MAC circuit:

Function	Delay (in ps)
AND	48
A + B.C	66
XOR	76
A.B + C.(A+B)	76
Half-Adder	76
Full-Adder	152

The code 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

We start off with the product bits calculated using an 8×8 product array. These were then reduced and propagated to finally get no more than 2 bits in each column. Given below is the stage-wise breakdown of how the bits were reduced and propagated:

2.1 Stage-1

- The capacity for the next stage is 6 bits, therefore any column with less than equal to 6 bits is directly propagated to the next stage. Therefore all bits till the 4th column are directly propagated.
- Column 5 has 7 bits, therefore we use a Half-Adder to reduce the bits by 1 and propagate the rest.
- Column 6 has 8 bits + 1 carry bit coming from the previous column. Therefore we use 1 Full-Adder and 1 Half-Adder to reduce the bits by 3 and propagate the rest.
- Column 7 has 9 bits + 2 carry bits from the previous column. Therefore we use 2 Full-Adders and 1 Half-Adder to reduce the bits by 5 and propagate the rest.
- Column 8 has 8 bits + 3 carry bits from the previous column. Therefore we use 2 Full-Adders and 1 Half-Adder to reduce the bits by 5 and propagate the rest.
- Column 9 has 7 bits + 3 carry bits from the previous column. Therefore we use 2 Full-Adders to reduce the bits by 4 and propagate the rest.
- Column 10 has 6 bits and 2 carry bits from the previous column. Therefore we use 1 Full-Adder to reduce the bits by 2 and propagate the rest.
- Columns 11 to 15 have less than equal to 6 bits, therefore we propagate all the bits.

2.2 Stage-2

- The capacity for the next stage is 4 bits, therefore any column with less than equal to 4 bits is directly propagated to the next stage. Therefore all bits till the 2nd column are directly propagated.
- Column 3 has 5 bits. Therefore we use 1 Half-Adder to reduce the bits by 1 and propagate the rest.
- Column 4 has 6 bits + 1 carry bit from the previous column. Therefore we use 1 Full-Adder and 1 Half-Adder to reduce the bits by 3 and propagate the rest.
- Columns 5 to 11 have 6 bits + 2 carry bits from the previous column. Therefore we use 2 Full-Adders to reduce the bits by 4 and propagate the rest.
- Column 12 has 4 bits + 2 carry bits from the previous column. Therefore we use 1 Full-Adder to reduce the bits by 2 and propagate the rest.
- Columns 13 to 15 have less than equal to 4 bits, therefore we propagate all the bits.

2.3 Stage-3

- The capacity for the next stage is 3 bits, therefore any column with less than equal to 3 bits is directly propagated to the next stage. Therefore all bits till the 1st column are directly propagated.
- Column 2 has 4 bits. Therefore we use 1 Half-Adder to reduce the bits by 1 and propagate the rest.
- Columns 3 to 13 have 4 bits + 2 carry bits from the previous column. Therefore we use 1 Full-Adder to reduce the bits by 2 and propagate the rest.
- Columns 14 to 15 have less than equal to 3 bits, therefore we propagate all the bits.

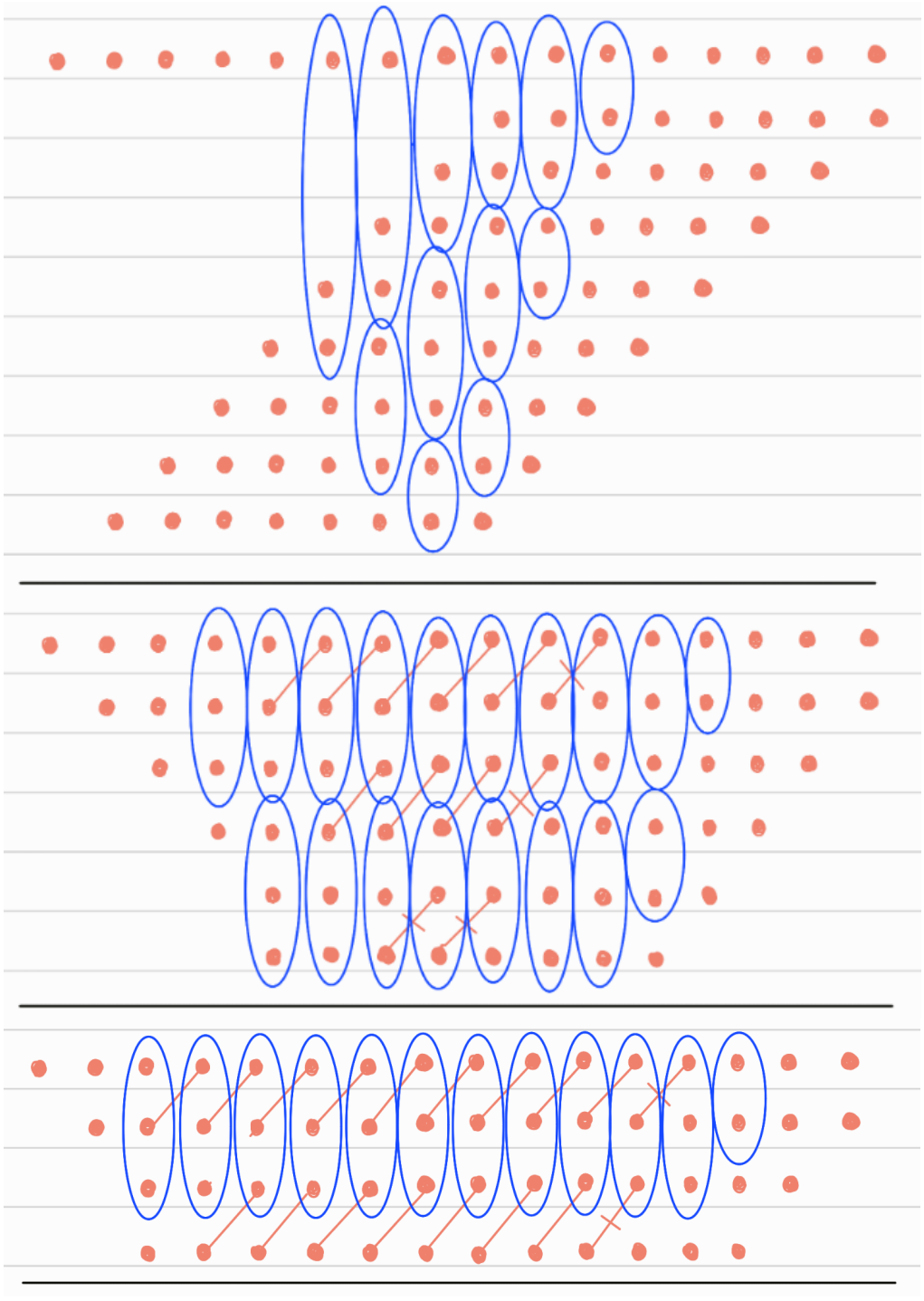
2.4 Stage-4

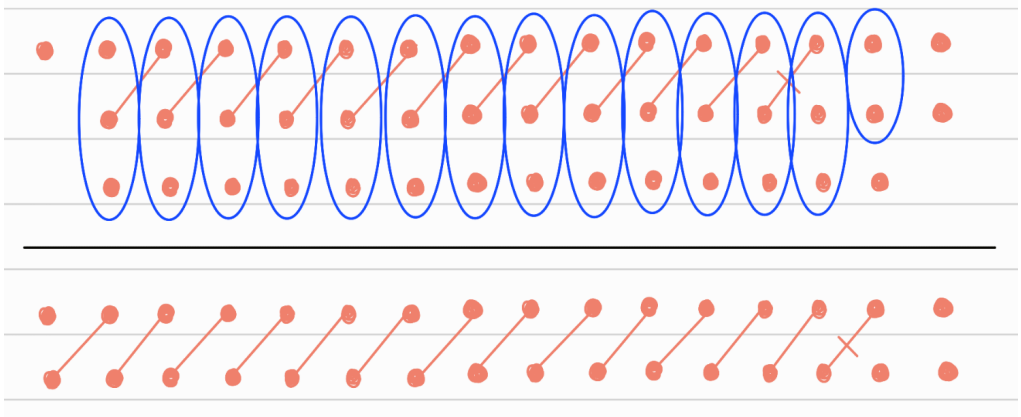
- The capacity for the next stage is 2 bits, therefore any column with less than equal to 2 bits is directly propagated to the next stage. Therefore all bits till the 0th column are directly propagated.
- Column 1 has 3 bits. Therefore we use 1 Half-Adder to reduce the bits by 1 and propagate the rest.
- Columns 2 to 14 have 3 bits + 1 carry bit from the previous column. Therefore we use 1 Full-Adder to reduce the bits by 2 and propagate the rest.
- Column 15 has only 1 bit, therefore we propagate it to the final stage.

2.5 Final Output

- We now have 2 rows of 16 bits each. These need to be added to get the final output.
- We use the 16-bit Brent-Kung Adder designed in the previous assignment to do this addition.

The dot diagram for the above reduction and propagation scheme is given below:





Given below is the VHDL code (**mac.vhd**) for implementing the Multiply-and-Accumulate circuit using the Design given above:

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

-- Half Adder
entity ha is
port (
A, B: in std_ulogic;
S, Cy: out std_ulogic
);
end entity ha;

architecture trivial of ha is
begin
sum: xorgate port map (A => A, B => B, uneq => S);
carry: andgate port map (A => A, B => B, prod => Cy);
end architecture trivial;

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

```
use work.gates.all;

-- Full Adder
entity fa is
port (
A, B, C: in std_ulogic;
S, Cy: out std_ulogic
);
end entity fa;

architecture trivial of fa is
signal t1,t2: std_logic;

begin
x1: xorgate port map (A => A, B => B, uneq => t1);
x2: xorgate port map (A => C, B => t1, uneq => S);
aterm: andgate port map (A => A, B => B, prod => t2);
carry: abcgate port map (A => t2, B => t1, C => C, abc => Cy);
end architecture trivial;

library IEEE;
use IEEE.std_logic_1164.all;
library work;
use work.gates.all;
library work;
use work.bkadder16.all;

entity mac is
port (
A, B: in std_logic_vector(7 downto 0);
C: in std_logic_vector(15 downto 0);
O: out std_logic_vector(16 downto 0)
);
end entity mac;

architecture arch of mac is

component ha is
```

```
port (
A, B: in std_ulogic;
S, Cy: out std_ulogic
);
end component ha;

component fa is
port (
A, B, C: in std_ulogic;
S, Cy: out std_ulogic
);
end component fa;

--Product bits
signal P0: std_logic_vector(7 downto 0);
signal P1: std_logic_vector(7 downto 0);
signal P2: std_logic_vector(7 downto 0);
signal P3: std_logic_vector(7 downto 0);
signal P4: std_logic_vector(7 downto 0);
signal P5: std_logic_vector(7 downto 0);
signal P6: std_logic_vector(7 downto 0);
signal P7: std_logic_vector(7 downto 0);

--Stage-1 bits
signal S1_0: std_logic_vector(15 downto 0);
signal S1_1: std_logic_vector(14 downto 0);
signal S1_2: std_logic_vector(13 downto 1);
signal S1_3: std_logic_vector(12 downto 2);
signal S1_4: std_logic_vector(11 downto 3);
signal S1_5: std_logic_vector(11 downto 4);

--Stage-2 bits
signal S2_0: std_logic_vector(15 downto 0);
signal S2_1: std_logic_vector(14 downto 0);
signal S2_2: std_logic_vector(13 downto 1);
signal S2_3: std_logic_vector(13 downto 2);

--Stage-3 bits
```

```
signal S3_0: std_logic_vector(15 downto 0);
signal S3_1: std_logic_vector(14 downto 0);
signal S3_2: std_logic_vector(14 downto 1);

--Stage-4 bits
signal S4_0: std_logic_vector(15 downto 0);
signal S4_1: std_logic_vector(15 downto 0);

begin

--Partial product bits
P0_0: andgate port map (A => A(0), B => B(0), prod => P0(0));
P0_1: andgate port map (A => A(0), B => B(1), prod => P0(1));
P0_2: andgate port map (A => A(0), B => B(2), prod => P0(2));
P0_3: andgate port map (A => A(0), B => B(3), prod => P0(3));
P0_4: andgate port map (A => A(0), B => B(4), prod => P0(4));
P0_5: andgate port map (A => A(0), B => B(5), prod => P0(5));
P0_6: andgate port map (A => A(0), B => B(6), prod => P0(6));
P0_7: andgate port map (A => A(0), B => B(7), prod => P0(7));

P1_0: andgate port map (A => A(1), B => B(0), prod => P1(0));
P1_1: andgate port map (A => A(1), B => B(1), prod => P1(1));
P1_2: andgate port map (A => A(1), B => B(2), prod => P1(2));
P1_3: andgate port map (A => A(1), B => B(3), prod => P1(3));
P1_4: andgate port map (A => A(1), B => B(4), prod => P1(4));
P1_5: andgate port map (A => A(1), B => B(5), prod => P1(5));
P1_6: andgate port map (A => A(1), B => B(6), prod => P1(6));
P1_7: andgate port map (A => A(1), B => B(7), prod => P1(7));

P2_0: andgate port map (A => A(2), B => B(0), prod => P2(0));
P2_1: andgate port map (A => A(2), B => B(1), prod => P2(1));
P2_2: andgate port map (A => A(2), B => B(2), prod => P2(2));
P2_3: andgate port map (A => A(2), B => B(3), prod => P2(3));
P2_4: andgate port map (A => A(2), B => B(4), prod => P2(4));
P2_5: andgate port map (A => A(2), B => B(5), prod => P2(5));
P2_6: andgate port map (A => A(2), B => B(6), prod => P2(6));
P2_7: andgate port map (A => A(2), B => B(7), prod => P2(7));
```

```
P3_0: andgate port map (A => A(3), B => B(0), prod => P3(0));
P3_1: andgate port map (A => A(3), B => B(1), prod => P3(1));
P3_2: andgate port map (A => A(3), B => B(2), prod => P3(2));
P3_3: andgate port map (A => A(3), B => B(3), prod => P3(3));
P3_4: andgate port map (A => A(3), B => B(4), prod => P3(4));
P3_5: andgate port map (A => A(3), B => B(5), prod => P3(5));
P3_6: andgate port map (A => A(3), B => B(6), prod => P3(6));
P3_7: andgate port map (A => A(3), B => B(7), prod => P3(7));

P4_0: andgate port map (A => A(4), B => B(0), prod => P4(0));
P4_1: andgate port map (A => A(4), B => B(1), prod => P4(1));
P4_2: andgate port map (A => A(4), B => B(2), prod => P4(2));
P4_3: andgate port map (A => A(4), B => B(3), prod => P4(3));
P4_4: andgate port map (A => A(4), B => B(4), prod => P4(4));
P4_5: andgate port map (A => A(4), B => B(5), prod => P4(5));
P4_6: andgate port map (A => A(4), B => B(6), prod => P4(6));
P4_7: andgate port map (A => A(4), B => B(7), prod => P4(7));

P5_0: andgate port map (A => A(5), B => B(0), prod => P5(0));
P5_1: andgate port map (A => A(5), B => B(1), prod => P5(1));
P5_2: andgate port map (A => A(5), B => B(2), prod => P5(2));
P5_3: andgate port map (A => A(5), B => B(3), prod => P5(3));
P5_4: andgate port map (A => A(5), B => B(4), prod => P5(4));
P5_5: andgate port map (A => A(5), B => B(5), prod => P5(5));
P5_6: andgate port map (A => A(5), B => B(6), prod => P5(6));
P5_7: andgate port map (A => A(5), B => B(7), prod => P5(7));

P6_0: andgate port map (A => A(6), B => B(0), prod => P6(0));
P6_1: andgate port map (A => A(6), B => B(1), prod => P6(1));
P6_2: andgate port map (A => A(6), B => B(2), prod => P6(2));
P6_3: andgate port map (A => A(6), B => B(3), prod => P6(3));
P6_4: andgate port map (A => A(6), B => B(4), prod => P6(4));
P6_5: andgate port map (A => A(6), B => B(5), prod => P6(5));
P6_6: andgate port map (A => A(6), B => B(6), prod => P6(6));
P6_7: andgate port map (A => A(6), B => B(7), prod => P6(7));

P7_0: andgate port map (A => A(7), B => B(0), prod => P7(0));
P7_1: andgate port map (A => A(7), B => B(1), prod => P7(1));
```

```
P7_2: andgate port map (A => A(7), B => B(2), prod => P7(2));
P7_3: andgate port map (A => A(7), B => B(3), prod => P7(3));
P7_4: andgate port map (A => A(7), B => B(4), prod => P7(4));
P7_5: andgate port map (A => A(7), B => B(5), prod => P7(5));
P7_6: andgate port map (A => A(7), B => B(6), prod => P7(6));
P7_7: andgate port map (A => A(7), B => B(7), prod => P7(7));

--Stage-1
S1_0(0) <= C(0);
S1_1(0) <= P0(0);

S1_0(1) <= C(1);
S1_1(1) <= P0(1);
S1_2(1) <= P1(0);

S1_0(2) <= C(2);
S1_1(2) <= P0(2);
S1_2(2) <= P1(1);
S1_3(2) <= P2(0);

S1_0(3) <= C(3);
S1_1(3) <= P0(3);
S1_2(3) <= P1(2);
S1_3(3) <= P2(1);
S1_4(3) <= P3(0);

S1_0(4) <= C(4);
S1_1(4) <= P0(4);
S1_2(4) <= P1(3);
S1_3(4) <= P2(2);
S1_4(4) <= P3(1);
S1_5(4) <= P4(0);

S1_0_5: ha port map (A => C(5), B => P0(5), S => S1_0(5),
Cy => S1_1(6));
S1_1(5) <= P1(4);
S1_2(5) <= P2(3);
S1_3(5) <= P3(2);
```

```
S1_4(5) <= P4(1);
S1_5(5) <= P5(0);

S1_0_6: fa port map (A => C(6), B => P0(6), C => P1(5), S => S1_0(6),
Cy => S1_1(7));
S1_2_6: ha port map (A => P2(4), B => P3(3), S => S1_2(6),
Cy => S1_3(7));
S1_3(6) <= P4(2);
S1_4(6) <= P5(1);
S1_5(6) <= P6(0);

S1_0_7: fa port map (A => C(7), B => P0(7), C => P1(6), S => S1_0(7),
Cy => S1_1(8));
S1_2_7: fa port map (A => P2(5), B => P3(4), C => P4(3), S => S1_2(7),
Cy => S1_3(8));
S1_4_7: ha port map (A => P5(2), B => P6(1), S => S1_4(7),
Cy => S1_5(8));
S1_5(7) <= P7(0);

S1_0_8: fa port map (A => C(8), B => P1(7), C => P2(6), S => S1_0(8),
Cy => S1_1(9));
S1_2_8: fa port map (A => P3(5), B => P4(4), C => P5(3), S => S1_2(8),
Cy => S1_3(9));
S1_4_8: ha port map (A => P6(2), B => P7(1), S => S1_4(8),
Cy => S1_5(9));

S1_0_9: fa port map (A => C(9), B => P2(7), C => P3(6), S => S1_0(9),
Cy => S1_1(10));
S1_2_9: fa port map (A => P4(5), B => P5(4), C => P6(3), S => S1_2(9),
Cy => S1_3(10));
S1_4(9) <= P7(2);

S1_0_10: fa port map (A => C(10), B => P3(7), C => P4(6), S => S1_0(10),
Cy => S1_1(11));
S1_2(10) <= P5(5);
S1_4(10) <= P6(4);
S1_5(10) <= P7(3);
```

```
S1_0(11) <= C(11);
S1_2(11) <= P4(7);
S1_3(11) <= P5(6);
S1_4(11) <= P6(5);
S1_5(11) <= P7(4);

S1_0(12) <= C(12);
S1_1(12) <= P5(7);
S1_2(12) <= P6(6);
S1_3(12) <= P7(5);

S1_0(13) <= C(13);
S1_1(13) <= P6(7);
S1_2(13) <= P7(6);

S1_0(14) <= C(14);
S1_1(14) <= P7(7);

S1_0(15) <= C(15);

--Stage-2
S2_0(0) <= S1_0(0);
S2_1(0) <= S1_1(0);

S2_0(1) <= S1_0(1);
S2_1(1) <= S1_1(1);
S2_2(1) <= S1_2(1);

S2_0(2) <= S1_0(2);
S2_1(2) <= S1_1(2);
S2_2(2) <= S1_2(2);
S2_3(2) <= S1_3(2);

S2_0_3: ha port map (A => S1_0(3), B => S1_1(3), S => S2_0(3),
Cy => S2_1(4));
S2_1(3) <= S1_2(3);
S2_2(3) <= S1_3(3);
S2_3(3) <= S1_4(3);
```

```
S2_0_4: fa port map (A => S1_0(4), B => S1_1(4), C => S1_2(4),
S => S2_0(4), Cy => S2_1(5));
S2_2_4: ha port map (A => S1_3(4), B => S1_4(4), S => S2_2(4),
Cy => S2_3(5));
S2_3(4) <= S1_5(4);

S2_0_5: fa port map (A => S1_0(5), B => S1_1(5), C => S1_2(5),
S => S2_0(5), Cy => S2_1(6));
S2_2_5: fa port map (A => S1_3(5), B => S1_4(5), C => S1_5(5),
S => S2_2(5), Cy => S2_3(6));

S2_0_6: fa port map (A => S1_0(6), B => S1_1(6), C => S1_2(6),
S => S2_0(6), Cy => S2_1(7));
S2_2_6: fa port map (A => S1_3(6), B => S1_4(6), C => S1_5(6),
S => S2_2(6), Cy => S2_3(7));

S2_0_7: fa port map (A => S1_0(7), B => S1_1(7), C => S1_2(7),
S => S2_0(7), Cy => S2_1(8));
S2_2_7: fa port map (A => S1_3(7), B => S1_4(7), C => S1_5(7),
S => S2_2(7), Cy => S2_3(8));

S2_0_8: fa port map (A => S1_0(8), B => S1_1(8), C => S1_2(8),
S => S2_0(8), Cy => S2_1(9));
S2_2_8: fa port map (A => S1_3(8), B => S1_4(8), C => S1_5(8),
S => S2_2(8), Cy => S2_3(9));

S2_0_9: fa port map (A => S1_0(9), B => S1_1(9), C => S1_2(9),
S => S2_0(9), Cy => S2_1(10));
S2_2_9: fa port map (A => S1_3(9), B => S1_4(9), C => S1_5(9),
S => S2_2(9), Cy => S2_3(10));

S2_0_10: fa port map (A => S1_0(10), B => S1_1(10), C => S1_2(10),
S => S2_0(10), Cy => S2_1(11));
S2_2_10: fa port map (A => S1_3(10), B => S1_4(10), C => S1_5(10),
S => S2_2(10), Cy => S2_3(11));

S2_0_11: fa port map (A => S1_0(11), B => S1_1(11), C => S1_2(11),
```

```
S => S2_0(11), Cy => S2_1(12));
S2_2_11: fa port map (A => S1_3(11), B => S1_4(11), C => S1_5(11),
S => S2_2(11), Cy => S2_3(12));

S2_0_12: fa port map (A => S1_0(12), B => S1_1(12), C => S1_2(12),
S => S2_0(12), Cy => S2_1(13));
S2_2(12) <= S1_3(12);

S2_0(13) <= S1_0(13);
S2_2(13) <= S1_1(13);
S2_3(13) <= S1_2(13);

S2_0(14) <= S1_0(14);
S2_1(14) <= S1_1(14);

S2_0(15) <= S1_0(15);

--Stage-3
S3_0(0) <= S2_0(0);
S3_1(0) <= S2_1(0);

S3_0(1) <= S2_0(1);
S3_1(1) <= S2_1(1);
S3_2(1) <= S2_2(1);

S3_0_2: ha port map (A => S2_0(2), B => S2_1(2), S => S3_0(2),
Cy => S3_1(3));
S3_1(2) <= S2_2(2);
S3_2(2) <= S2_3(2);

S3_0_3: fa port map (A => S2_0(3), B => S2_1(3), C => S2_2(3),
S => S3_0(3), Cy => S3_1(4));
S3_2(3) <= S2_3(3);

S3_0_4: fa port map (A => S2_0(4), B => S2_1(4), C => S2_2(4),
S => S3_0(4), Cy => S3_1(5));
S3_2(4) <= S2_3(4);
```

```
S3_0_5: fa port map (A => S2_0(5), B => S2_1(5), C => S2_2(5),
S => S3_0(5), Cy => S3_1(6));
S3_2(5) <= S2_3(5);

S3_0_6: fa port map (A => S2_0(6), B => S2_1(6), C => S2_2(6),
S => S3_0(6), Cy => S3_1(7));
S3_2(6) <= S2_3(6);

S3_0_7: fa port map (A => S2_0(7), B => S2_1(7), C => S2_2(7),
S => S3_0(7), Cy => S3_1(8));
S3_2(7) <= S2_3(7);

S3_0_8: fa port map (A => S2_0(8), B => S2_1(8), C => S2_2(8),
S => S3_0(8), Cy => S3_1(9));
S3_2(8) <= S2_3(8);

S3_0_9: fa port map (A => S2_0(9), B => S2_1(9), C => S2_2(9),
S => S3_0(9), Cy => S3_1(10));
S3_2(9) <= S2_3(9);

S3_0_10: fa port map (A => S2_0(10), B => S2_1(10), C => S2_2(10),
S => S3_0(10), Cy => S3_1(11));
S3_2(10) <= S2_3(10);

S3_0_11: fa port map (A => S2_0(11), B => S2_1(11), C => S2_2(11),
S => S3_0(11), Cy => S3_1(12));
S3_2(11) <= S2_3(11);

S3_0_12: fa port map (A => S2_0(12), B => S2_1(12), C => S2_2(12),
S => S3_0(12), Cy => S3_1(13));
S3_2(12) <= S2_3(12);

S3_0_13: fa port map (A => S2_0(13), B => S2_1(13), C => S2_2(13),
S => S3_0(13), Cy => S3_1(14));
S3_2(13) <= S2_3(13);

S3_0(14) <= S2_0(14);
S3_2(14) <= S2_1(14);
```

```
S3_0(15) <= S2_0(15);

--Stage-4
S4_0(0) <= S3_0(0);
S4_1(0) <= S3_1(0);

S4_0_1: ha port map (A => S3_0(1), B => S3_1(1), S => S4_0(1),
Cy => S4_1(2));
S4_1(1) <= S3_2(1);

S4_0_2: fa port map (A => S3_0(2), B => S3_1(2), C => S3_2(2),
S => S4_0(2), Cy => S4_1(3));

S4_0_3: fa port map (A => S3_0(3), B => S3_1(3), C => S3_2(3),
S => S4_0(3), Cy => S4_1(4));

S4_0_4: fa port map (A => S3_0(4), B => S3_1(4), C => S3_2(4),
S => S4_0(4), Cy => S4_1(5));

S4_0_5: fa port map (A => S3_0(5), B => S3_1(5), C => S3_2(5),
S => S4_0(5), Cy => S4_1(6));

S4_0_6: fa port map (A => S3_0(6), B => S3_1(6), C => S3_2(6),
S => S4_0(6), Cy => S4_1(7));

S4_0_7: fa port map (A => S3_0(7), B => S3_1(7), C => S3_2(7),
S => S4_0(7), Cy => S4_1(8));

S4_0_8: fa port map (A => S3_0(8), B => S3_1(8), C => S3_2(8),
S => S4_0(8), Cy => S4_1(9));

S4_0_9: fa port map (A => S3_0(9), B => S3_1(9), C => S3_2(9),
S => S4_0(9), Cy => S4_1(10));

S4_0_10: fa port map (A => S3_0(10), B => S3_1(10), C => S3_2(10),
S => S4_0(10), Cy => S4_1(11));
```

```
S4_0_11: fa port map (A => S3_0(11), B => S3_1(11), C => S3_2(11),
S => S4_0(11), Cy => S4_1(12));

S4_0_12: fa port map (A => S3_0(12), B => S3_1(12), C => S3_2(12),
S => S4_0(12), Cy => S4_1(13));

S4_0_13: fa port map (A => S3_0(13), B => S3_1(13), C => S3_2(13),
S => S4_0(13), Cy => S4_1(14));

S4_0_14: fa port map (A => S3_0(14), B => S3_1(14), C => S3_2(14),
S => S4_0(14), Cy => S4_1(15));

S4_0(15) <= S3_0(15);

--Final output
Output: bk_adder port map (A => S4_0, B => S4_1, Cin => '0',
S => O(15 downto 0), Cout => O(16));

end architecture arch;
```

3 Simulation and Testing

In order test my design I wrote a testbench (**Testbench.vhdl**), capable of reading inputs, outputs and mask bits from an external file called TRACE-FILE.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 := 32;  -- # 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 5ns, apply input,
  -- wait for 1 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;
```

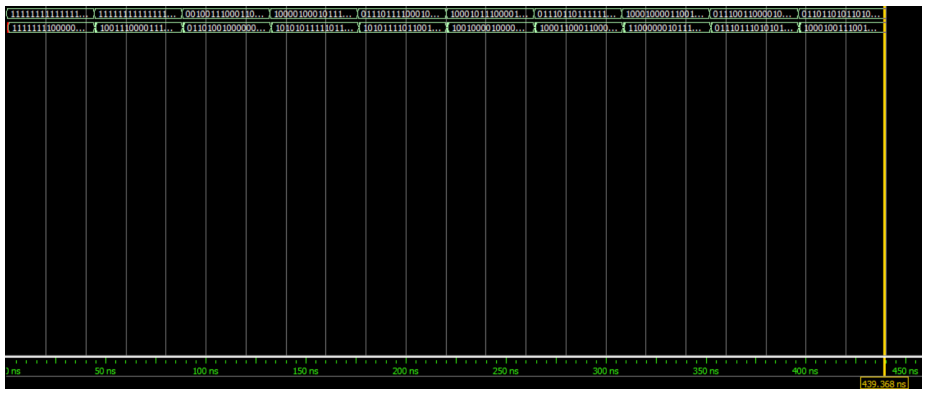
The file containing the inputs and outputs (**TRACEFILE.txt**) is given below:

```
11111111111111111111111111111111 11111111000000000 1111111111111111
11111111111111110101010110101010 10011100001110001 1111111111111111
00100111000110101111101010101111 01101001000000000 1111111111111111
10000100010111011101110111110101 10101011111011110 1111111111111111
01110111100010011110101011111101 10101111011001011 1111111111111111
10001011100001101011001011010111 10010000100000100 1111111111111111
01110110111111111100101011001101 10001100011000001 1111111111111111
10001000011001101111101111111110 11000000101110000 1111111111111111
01110011000010001010010011000001 01110111010101100 1111111111111111
01101101011010011011001011101111 10001001110010111 1111111111111111
```

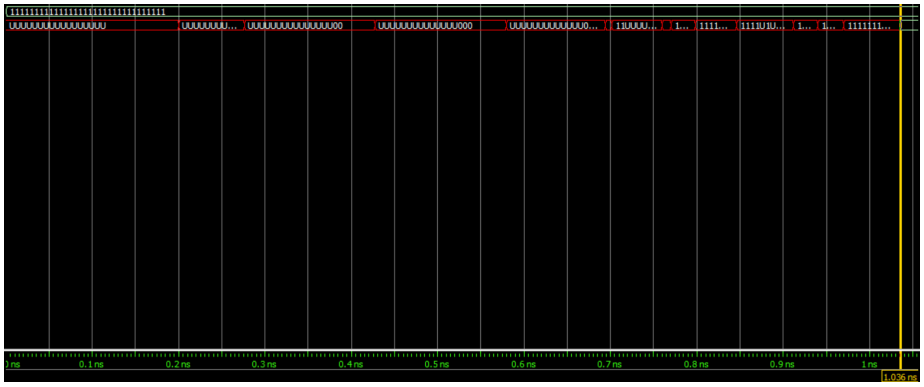
Given below is the screenshot of the RTL Simulation output showing that all the test cases passed successfully:

```
# run -all
# ** Note: SUCCESS, all tests passed.
#    Time: 440 ns  Iteration: 0  Instance: /testbench
```

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 MAC circuit can be calculated by considering the critical path. The critical path would include passing 4 Full-Adders and the final Brent-Kung Adder. A Full-Adder has a theoretical delay equal to that of 2 XOR gates i.e. $152ps$ and the Brenk-Kung Adder has a theoretical delay of $548ps$. This leads to a total theoretical delay of $4 \times 152ps + 548ps = 1156ps$. This theoretical delay approximately matches with the observed delay in the RTL Simulation output ($1036ps$). This slight mismatch in delays might be because these particular inputs do not lead to the critical path delay for the Brent Kung Adder.