2024

Αναφορά 3ης εργαστηριακής Άσκησης

Εργαστήριο VLSI

ΠΑΠΑΔΟΠΟΥΛΟΣ ΣΠΥΡΙΔΩΝ (ΑΜ):03120033
ΕΜΜΑΝΟΥΗΛ ΞΕΝΟΣ (ΑΜ):03120850

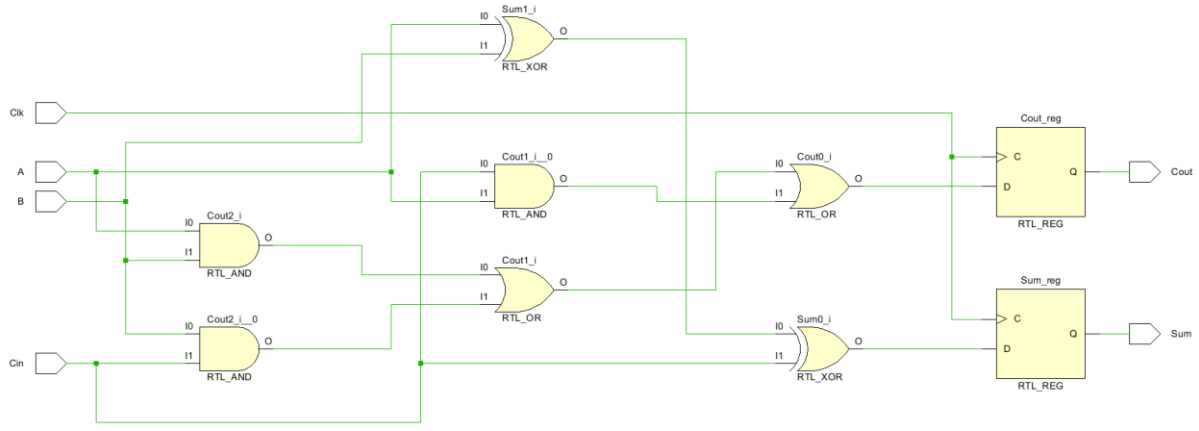
Ζήτημα Πρώτο: Σύγχρονος Πλήρης Αθροιστής

Σε αυτό το ζήτημα θα κατασκευαστεί ένας Πλήρης Αθροιστής που θα αποτελέσει την βασική δομή που θα χρησιμοποιηθεί για τον σύγχρονο Αθροιστή διάδοσης κρατουμένου των 4bits κάνοντας χρήση της τεχνικής Pipeline και για τον συστολικό Πολλαπλασιαστή διάδοσης κρατουμένων των 4 bits. Επίσης θα αναλυθεί και η υλοποίηση του σύγχρονου πλήρη αθροιστή.

Ξεκινώντας από τον σύγχρονο πλήρη αθροιστή, ο κώδικας VHDL είναι:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sync_full_adder is
  Port (
    A, B, Cin, Clk: in std_logic;
    Sum, Cout: out std_logic
    );
end sync_full_adder;

architecture Behavioral of sync_full_adder is
    signal input: std_logic_vector(2 downto 0);
begin
    input<=A & B & Cin;
    process(clk)
    begin
        if rising_edge(clk) then
            Sum <= A xor B xor Cin;
            Cout<=(A and B) or (B and Cin) or (Cin and A);
        end if;
    end process;
end Behavioral;
```

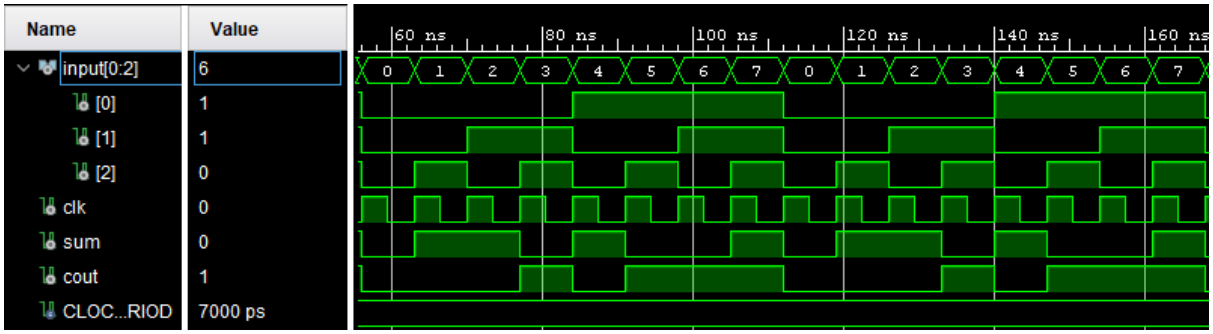
Το RTL σχήμα είναι:



Όπως αναμενόταν λοιπόν ο σύγχρονος πλήρης αθροιστής είναι ίδιος με έναν ασύγχρονο αθροιστή αλλά με δύο D Flip Flop στην έξοδο.

Ακολουθεί ο κώδικας του testbench που χρησιμοποιήθηκε ώστε να επαληθευτεί η σωστή λειτουργία του σύγχρονου Αθροιστή καθώς και το αποτέλεσμα της προσομοίωσης χρησιμοποιώντας αυτό το testbench.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity sync_full_adder_testbench is
end sync_full_adder_testbench;

architecture Behavioral of sync_full_adder_testbench is
    component sync_full_adder is
      Port (
        A, B, Cin, Clk: in std_logic;
        Sum, Cout: out std_logic
        );
    end component;
    signal input: std_logic_vector(0 to 2);
    signal clk, sum, cout: std_logic;
    constant CLOCK_PERIOD: time := 7 ns;
begin
    uut: sync_full_adder port map(input(2), input(1), input(0), clk, sum,
cout);

    clk_process: process
    begin
        clk <= '1';
        wait for CLOCK_PERIOD / 2;
        clk <= '0';
        wait for CLOCK_PERIOD / 2;
    end process;

    stimulus_process: process
    begin
        for i in 0 to 7 loop
            input <= std_logic_vector(to_unsigned(i, input'length));
            wait for CLOCK_PERIOD;
        end loop;
    end process;

end Behavioral;
```



Βλέπουμε λοιπόν ότι πράγματι παράγεται το σωστό αποτέλεσμα.

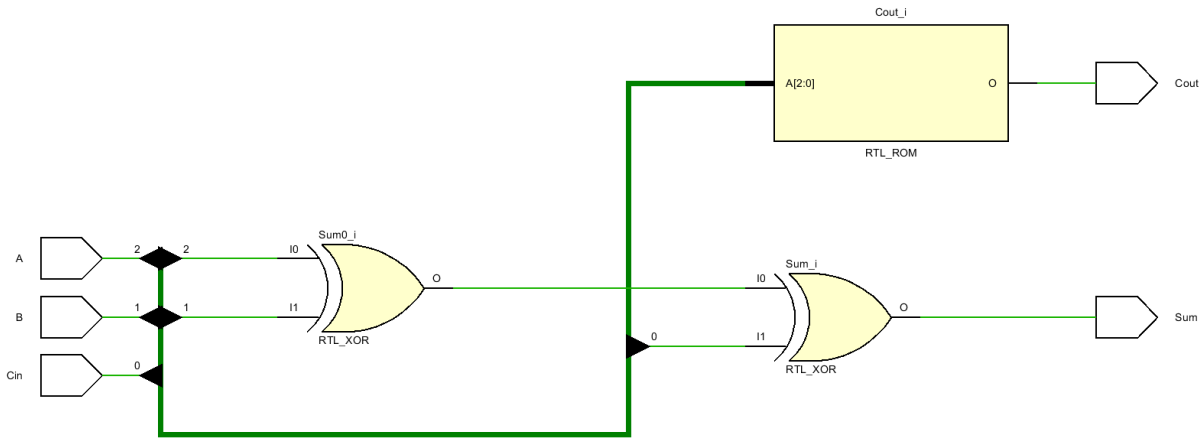
Τρέχοντας synthesis για να βρούμε το Critical Path παίρνουμε το ακόλουθο αποτέλεσμα

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	∞	2	2	1	Cout_reg/C	Cout	4.076	3.276	0.800	∞			
Path 2	∞	2	2	1	Sum_reg/C	Sum	4.076	3.276	0.800	∞			
Path 3	∞	2	3	2	Cin	Cout_reg/D	1.932	1.132	0.800	∞	input port clock		
Path 4	∞	2	3	2	Cin	Sum_reg/D	1.906	1.106	0.800	∞	input port clock		

Ο VHDL κώδικας του ασύγχρονου Πλήρη Αθροιστή είναι:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity full_adder is
  Port (
  A, B, Cin: in std_logic;
  Sum, Cout: out std_logic
  );
end full_adder;

architecture Behavioral of full_adder is
    signal input : std_logic_vector(2 downto 0);
begin
    input<= A & B & Cin;
    Sum<= A xor B xor Cin;
    with input select
    Cout<= '1' when "011"|"101"|"110"|"111",
           '0' when "000"|"001"|"010"|"100",
           'X' when others;

end Behavioral;
```

Το RTL σχηματικό είναι:

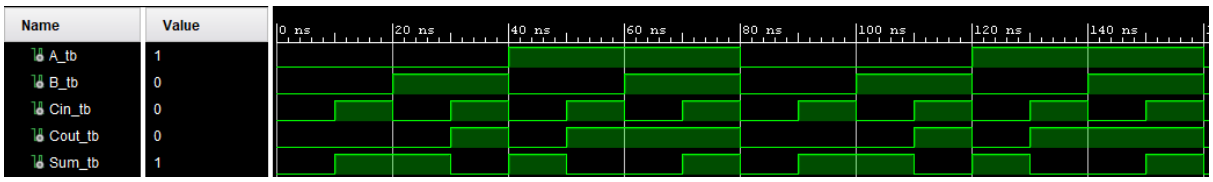


Ακολουθεί ο κώδικας του testbench που χρησιμοποιήθηκε ώστε να επαληθευτεί η σωστή λειτουργία του Αθροιστή καθώς και το αποτέλεσμα της προσομοίωσης χρησιμοποιώντας αυτό το testbench.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_testbench is
end full_adder_testbench;

architecture Behavioral of full_adder_testbench is
    component full_adder is
      Port (
        A, B, Cin: in std_logic;
        Sum, Cout: out std_logic);
    end component;

    signal A_tb, B_tb, Cin_tb, Cout_tb, Sum_tb: std_logic;
begin

    uut: full_adder port map(
        A=>A_tb,
        B=>B_tb,
        Cin=>Cin_tb,
        Sum=>Sum_tb,
        Cout=>Cout_tb
        );
    stimulus_process: process
        begin
            A_tb <= '0';
            B_tb <= '0';
            Cin_tb <= '0';
            wait for 10ns;

            A_tb <= '0';
            B_tb <= '0';
            Cin_tb <= '1';
```

```
            wait for 10ns;

            A_tb <= '0';
            B_tb <= '1';
            Cin_tb <= '0';
            wait for 10ns;

            A_tb <= '0';
            B_tb <= '1';
            Cin_tb <= '1';
            wait for 10ns;

            A_tb <= '1';
            B_tb <= '0';
            Cin_tb <= '0';
            wait for 10ns;

            A_tb <= '1';
            B_tb <= '0';
            Cin_tb <= '1';
            wait for 10ns;

            A_tb <= '1';
            B_tb <= '1';
            Cin_tb <= '0';
            wait for 10ns;

            A_tb <= '1';
            B_tb <= '1';
            Cin_tb <= '1';
            wait for 10ns;
      end process;
end Behavioral;
```

Το αποτέλεσμα της προσομοίωσης είναι:



Βλέπουμε λοιπόν ότι το αποτέλεσμα του Full Adder είναι σωστό βάσει της εισόδου του.

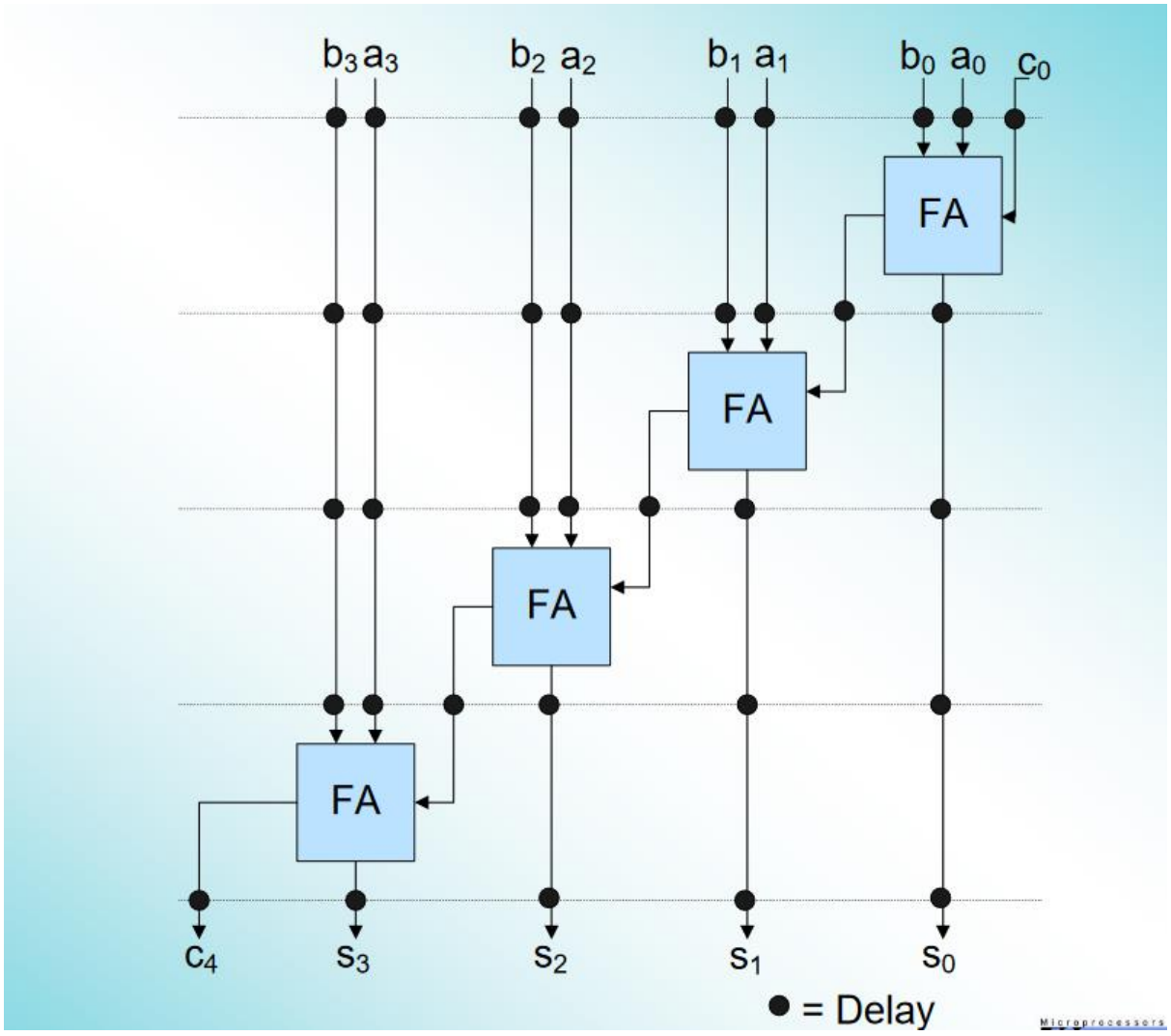
Τρέχοντας synthesis για να βρούμε το critical path αυτού του αθροιστή παίρνουμε το αποτέλεσμα:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	∞	3	4	2	B	Cout	5.377	3.778	1.599	∞	input port clock		
Path 2	∞	3	4	2	Cin	Sum	5.351	3.752	1.599	∞	input port clock		

Βλέπουμε λοιπόν ότι το critical path είναι λίγο μεγαλύτερο από το critical path του σύγχρονου Πλήρη Αθροιστή.

Ζήτημα Δεύτερο: Σύγχρονος Αθροιστής Διάδοσης Κρατουμένου με χρήση Pipeline

Σε αυτό το ζήτημα θα κατασκευαστεί ένας pipelined αθροιστής διάδοσης κρατουμένου Η δομή του pipelined αθροιστή θα είναι οι ακόλουθη:



Κάθε μαύρη κουκκίδα σημαίνει ότι χρειαζόμαστε να επιβάλουμε καθυστέρηση ενός κύκλου. Αυτή η καθυστέρηση μπορεί πολύ εύκολα να επιβληθεί χρησιμοποιώντας D Flip Flop(η εικόνα αυτή είναι από τις διαφάνειες του μαθήματος. Για την δική μας υλοποίηση και ορθή λειτουργία του Pipeline σύμφωνα με τις δοθείσες προδιαγραφές κρίναμε ότι η πρώτη σειρά από flip flop στην είσοδο δεν είναι απαραίτητη. Για αυτό δεν υπάρχουν τα αντίστοιχα flip flop στην υλοποίηση μας). Η λογική του αθροιστή αυτού είναι παρόμοια με αυτή του 4 bit αθροιστή χωρίς pipeline, δηλαδή ότι το sum του κάθε Full Adder αποτελεί ένα ψηφίο του τελικού αθροίσματος και το Cout του κάθε αθροιστή αποτελεί το Cin του επόμενου. Το τελευταίο κρατούμενο εξόδου είναι το MSB της εξόδου ενώ το πρώτο Cin τίθεται 0. Επιβάλλοντας καθυστερήσεις όπως φαίνεται στην παρακάτω εικόνα παρατηρούμε ότι την στιγμή που φτάνει το κρατούμενο εισόδου σε κάθε αθροιστή φτάνουν και τα bit εισόδου a[i] και b[i] οπότε υπολογίζεται με τα σωστά bit τα bit του αθροίσματος s[i]. Επίσης

παρατηρούμε ότι τα bit εξόδου φτάνουν ταυτόχρονα στην έξοδο οπότε θα δούμε σωστά την έξοδο μετά από 4 κύκλους.

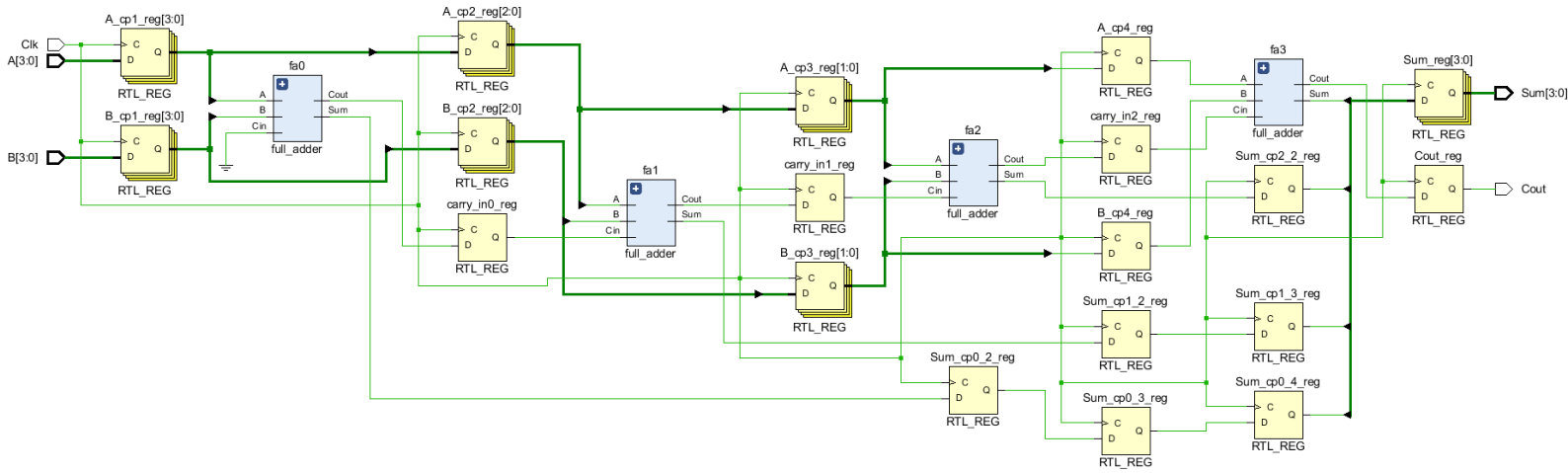
Ο κώδικας VHDL για τον παραπάνω 4 bit αθροιστή είναι:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity pipelined_4_bit_adder is
  Port (
  A, B: in std_logic_vector(3 downto 0);
  Sum: out std_logic_vector(3 downto 0);
  Cout: out std_logic;
  Clk: in std_logic
  );
end pipelined_4_bit_adder;

architecture Behavioral of pipelined_4_bit_adder is
    component full_adder is
      Port (
        A, B, Cin: in std_logic;
        Sum, Cout: out std_logic);
    end component;

    signal A_cp1, B_cp1: std_logic_vector(3 downto 0);
    signal A_cp2, B_cp2: std_logic_vector(2 downto 0);
    signal A_cp3, B_cp3: std_logic_vector(1 downto 0);
    signal A_cp4, B_cp4: std_logic;
    signal carry0, carry_in0, carry1, carry_in1, carry2, carry_in2, carry3:
std_logic;
    signal Sum_cp0_1, Sum_cp0_2, Sum_cp0_3, Sum_cp0_4, Sum_cp1_1, Sum_cp1_2,
Sum_cp1_3, Sum_cp2_1, Sum_cp2_2, Sum_cp3_1: std_logic;

begin
        fa0: full_adder port map(
                A => A_cp1(0),
                B => B_cp1(0),
                Cin => '0',
                Sum => Sum_cp0_1,
                Cout => carry0
              );

          fa1: full_adder port map(
            A => A_cp2(0),
            B => B_cp2(0),
            Cin => carry_in0,
            Sum => Sum_cp1_1,
            Cout => carry1
          );
```

```
    fa2: full_adder port map(
        A => A_cp3(0),
        B => B_cp3(0),
        Cin => carry_in1,
        Sum => Sum_cp2_1,
        Cout => carry2
     );

     fa3: full_adder port map(
       A => A_cp4,
       B => B_cp4,
       Cin => carry_in2,
       Sum => Sum_cp3_1,
       Cout => carry3
    );

  process (Clk)
    begin
      if rising_edge(Clk) then

        A_cp1 <= A(3 downto 0);
        B_cp1 <= B(3 downto 0);

        A_cp2 <= A_cp1(3 downto 1);
        B_cp2 <= B_cp1(3 downto 1);

        A_cp3 <= A_cp2(2 downto 1);
        B_cp3 <= B_cp2(2 downto 1);

        A_cp4 <= A_cp3(1);
        B_cp4 <= B_cp3(1);

        carry_in0<=carry0;
        carry_in1<=carry1;
        carry_in2<=carry2;

        Sum_cp0_2<=Sum_cp0_1;
        Sum_cp0_3<=Sum_cp0_2;
        Sum_cp0_4<=Sum_cp0_3;
        Sum(0)<=Sum_cp0_4;

        Sum_cp1_2<=Sum_cp1_1;
        Sum_cp1_3<=Sum_cp1_2;
        Sum(1)<=Sum_cp1_3;

        Sum_cp2_2<=Sum_cp2_1;
        Sum(2)<=Sum_cp2_2;
```

```
            Sum(3)<=Sum_cp3_1;

            Cout<=carry3;
          end if;
        end process;

end Behavioral;
```

Το RTL σχηματικό είναι:



Όπου βλέπουμε ότι κατασκευάστηκε πράγματι το κύκλωμα που φαίνεται στην παραπάνω εικόνα.

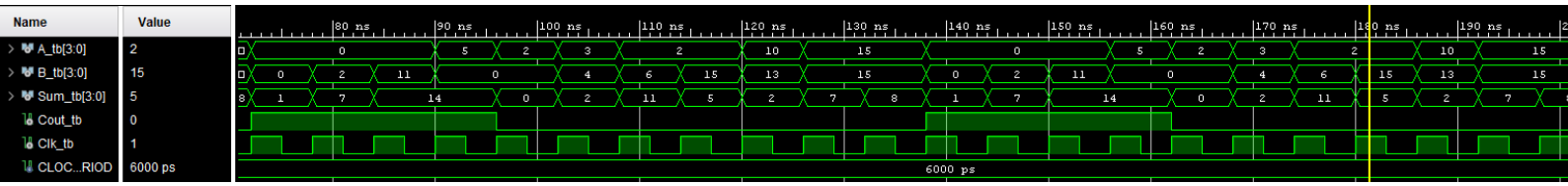
Κατασκευάζουμε το ακόλουθο testbench για να επαληθεύσουμε ο pipelined 4 bit adder εκτελεί σωστά την πρόσθεση.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity pipelined_4_bit_adder_testbench is
end pipelined_4_bit_adder_testbench;

architecture Behavioral of pipelined_4_bit_adder_testbench is
    component pipelined_4_bit_adder is
      Port (
      A, B: in std_logic_vector(3 downto 0);
      Sum: out std_logic_vector(3 downto 0);
      Cout: out std_logic;
      Clk: in std_logic
      );
    end component;

    signal A_tb, B_tb, Sum_tb : std_logic_vector(3 downto 0);
    signal Cout_tb, Clk_tb: std_logic;
    constant CLOCK_PERIOD: time := 6 ns;
```

```
begin

    uut: pipelined_4_bit_adder port map (
        A=>A_tb,
        B=>B_tb,
        Sum=>Sum_tb,
        Cout=>Cout_tb,
        Clk=>Clk_tb
        );

     clk_process: process
        begin
            clk_tb <= '1';
            wait for CLOCK_PERIOD / 2;
            clk_tb <= '0';
            wait for CLOCK_PERIOD / 2;
        end process;

      stimulus_process: process
      begin
        wait for CLOCK_PERIOD;
        --Testing if 0+0=0
          A_tb <= "0000";
          B_tb <= "0000";
          wait for CLOCK_PERIOD;

          --Testing if 0+x=x
          A_tb <= "0000";
          B_tb <= "0010";
          wait for CLOCK_PERIOD;

          A_tb <= "0000";
          B_tb <= "1011";
          wait for CLOCK_PERIOD;

          --Testing if x+0=x
          A_tb <= "0101";
          B_tb <= "0000";
          wait for CLOCK_PERIOD;

          A_tb <= "0010";
          B_tb <= "0000";
          wait for CLOCK_PERIOD;

          --Testing if 3+4=7
            A_tb <= "0011";
            B_tb <= "0100";
            wait for CLOCK_PERIOD;
```

```
            -- 6+2=8
             A_tb <= "0010";
             B_tb <= "0110";
             wait for CLOCK_PERIOD;

          --Testing if cout works
          -- 2+15=17
          A_tb <= "0010";
          B_tb <= "1111";
          wait for CLOCK_PERIOD;

          -- 10+13=23
          A_tb <= "1010";
          B_tb <= "1101";
          wait for CLOCK_PERIOD;

          -- 15+15=30
          A_tb <= "1111";
          B_tb <= "1111";
          wait for CLOCK_PERIOD;
      end process;

end Behavioral;
```

Το αποτέλεσμα της προσομοίωσης είναι το ακόλουθο:



Βλέπουμε λοιπόν ότι ο αθροιστής παράγει σωστά αποτελέσματα μετά από 4 κύκλους από την ώρα που βάλαμε την είσοδο.

Κάνοντας synthesis για να βρούμε το critical path παίρνουμε τα ακόλουθα αποτελέσματα:

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	∞	2	2	1	Cout_reg/C	Cout	4.076	3.276	0.800	∞			
Path 2	∞	2	2	1	Sum_reg[2]/C	Sum[2]	4.076	3.276	0.800	∞			
Path 3	∞	2	2	1	Sum_reg[3]/C	Sum[3]	4.076	3.276	0.800	∞			
Path 4	∞	2	2	1	Sum_reg[0]/C	Sum[0]	4.058	3.258	0.800	∞			
Path 5	∞	2	2	1	Sum_reg[1]/C	Sum[1]	4.058	3.258	0.800	∞			
Path 6	∞	2	2	2	B_cp2_reg[0]/C	Sum_cp1_3_reg_srl2/D	1.836	0.751	1.085	∞			

Βλέπουμε λοιπόν ότι το critical path είναι το μονοπάτι από τον καταχωρητή που αποθηκεύει το Cout στην έξοδο, και το μονοπάτι από τους καταχωρητές που αποθηκεύουν τα 2 MSB στην έξοδο.

Τα critical paths του non pipelined 4 bit αθροιστή είναι:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	∞	4	5	3	Cin	Cout	5.942	3.876	2.066	∞	input port clock		
Path 2	∞	4	5	3	Cin	Sum[2]	5.942	3.876	2.066	∞	input port clock		
Path 3	∞	4	5	3	Cin	Sum[3]	5.936	3.870	2.066	∞	input port clock		
Path 4	∞	3	4	2	B[1]	Sum[1]	5.379	3.780	1.599	∞	input port clock		
Path 5	∞	3	4	3	Cin	Sum[0]	5.351	3.752	1.599	∞	input port clock		

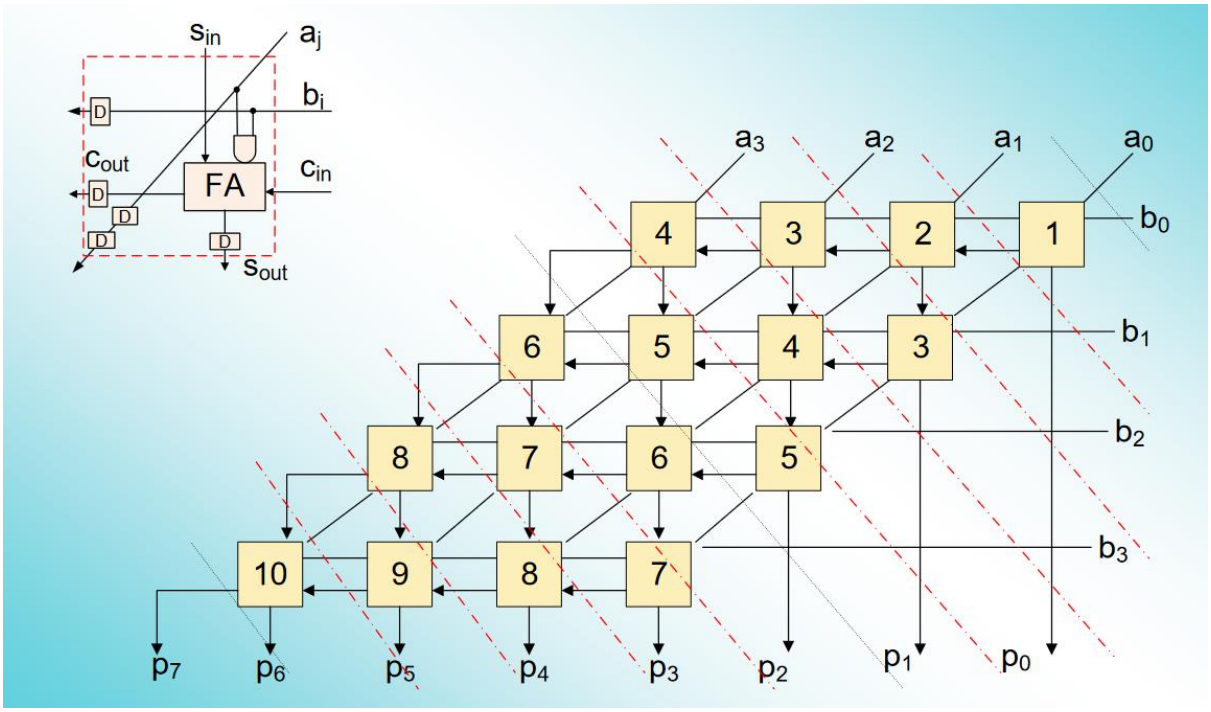
Βλέπουμε λοιπόν ότι το μεγαλύτερο critical path του non pipelined 4 bit adder είναι περίπου 50% από το critical path του pipelined 4 bit adder.

Θεωρώντας το non pipelined κύκλωμα ως ένα σύγχρονο κύκλωμα με περίοδο ίση με το critical path με την μεγαλύτερη καθυστέρηση παρατηρούμε ότι με το non pipelined κύκλωμα θα πάρουμε το αποτέλεσμα που χρειαζόμαστε άμεσα πιο γρήγορα, καθώς σε μία περίοδό του θα λάβουμε το αποτέλεσμα (δηλαδή περίπου 6ns), ενώ στο pipelined κύκλωμα χρειαζόμαστε 4 περιόδους (δηλαδή περίπου 16ns) ωστόσο το pipelined κύκλωμα έχει πολύ μεγαλύτερο throughput. Αν θέλουμε για παράδειγμα να εκτελέσουμε 4 προσθέσεις τότε θα χρειαστούμε με το non pipelined κύκλωμα 4 κύκλους, δηλαδή περίπου 24ns, ενώ με το pipelined θα χρειαστούμε 4 κύκλους για να υλοποιηθεί η πρώτη πρόσθεση και 3 για να υλοποιηθούν οι 3 εναπομείναντες προσθέσεις, αφού σε κάθε κύκλο μπορούμε να προσθέσουμε μία πρόσθεση που θα τρέχει παράλληλα με την επόμενη πρόσθεση. Επομένως στον 2ο κύκλο θα γίνει η πρόσθεση για το πρώτο bit της δεύτερης πρόσθεσης ενώ παράλληλα θα γίνει η πρόσθεση για το 2ο bit της πρώτης πρόσθεσης. Έτσι στον τέταρτο κύκλο θα τελειώσει η πρώτη πρόσθεση, μετά από έναν κύκλο η δεύτερη κτλ. Έτσι χρειαζόμαστε συνολικά 7 κύκλους δηλαδή περίπου 21ns.

Από τα παραπάνω παρατηρούμε ότι η pipelined εκδοχή του 4 bit adder χρησιμοποιεί κάθε πλήρη αθροιστή συνεχώς για όσο υπάρχουν δεδομένα, αφού με το που τελειώσει την πρόσθεση που εκτελεί ένας πλήρης αθροιστής έρχονται τα δεδομένα της επόμενης πρόσθεσης. Αντίθετα στον non pipelined αθροιστή όταν ένας πλήρης αθροιστής τελειώσει την πρόσθεση που επιτελεί θα παραμείνει για έναν κύκλο ανενεργός. Για παράδειγμα όταν ο πρώτος αθροιστής τελειώσει την πράξη για τα πρώτα bit θα περιμένει μέχρι να τελειώσουν την πρόσθεση και όλοι οι υπόλοιποι πλήρεις αθροιστές για να δεχθεί τα στοιχεία πάνω στα οποία θα επιτελέσει την δεύτερή του πρόσθεση. Είναι λοιπόν προφανές ότι η pipelined εκδοχή κάνει αποτελεσματικότερη διαχείριση πόρων.

Ζήτημα Τρίτο: Συστολικός Πολλαπλασιαστής Διάδοσης Κρατουμένου

Στο μέρος αυτό θα υλοποιήσουμε σε VHDL έναν συστολικό πολλαπλασιαστή χρησιμοποιώντας τους FA(χωρίς ρολόι) που δείξαμε στο πρώτο ερώτημα. Για την λογική σχεδίαση του συστολικού πολλαπλασιαστή χρησιμοποιήσαμε το σχεδιάγραμμα που δίνεται στις διαφάνειες του μαθήματος και παρατίθεται ακολούθως:



Κάθε τετράγωνο στο διάγραμμα αντιστοιχεί στο κύτταρο που φαίνεται πάνω αριστερά στην εικόνα. Σε αυτό το κύτταρο είναι ενσωματωμένα και τα flip flop που πρέπει να χρησιμοποιηθούν, ώστε σε κάθε κύκλο ρολογιού κάθε κύτταρο να παράγει ένα αποτέλεσμα(και συνεπώς σε κάθε κύκλο ρολογιού να έχουμε ένα αποτέλεσμα του πολλαπλασιασμού). Κάθε κύτταρο όπως βλέπουμε είναι αριθμημένο. Ο αριθμός κάθε κυττάρου αντιστοιχεί στον κύκλο τον οποίο το κύτταρο παράγει το αποτέλεσμα του για την πράξη ενός πολλαπλασιασμού. Δηλαδή θεωρώντας ότι στο πρώτο κύκλο παίρνουμε την είσοδο βλέπουμε ότι η έξοδος παράγεται στον δέκατο κύκλο(αυτό μπορούμε να το δούμε και ακολουθώντας ένα μονοπάτι από την είσοδο στην έξοδο και μετρώντας το πλήθος των flip flop που θα βρούμε στο μονοπάτι αυτό). Με βάση το σχεδιάγραμμα παρατηρούμε τα εξής(από αυτές τις παρατηρήσεις μάλλον καταλήξαμε σε αυτή την τοποθέτηση των flip flop):

- Το b_i σχεδόν πάντα κατευθύνεται στο επόμενο στην σειρά κύτταρο, το οποίο παράγει αποτέλεσμα στον αμέσως επόμενο κύκλο => Απαιτείται μόνο ένα flip flop να τα χωρίζει.
- Το Cout σχεδόν πάντα κατευθύνεται στο επόμενο στην σειρά κύτταρο, το οποίο παράγει αποτέλεσμα στον αμέσως επόμενο κύκλο => Απαιτείται μόνο ένα flip flop να τα χωρίζει.
- Το a τροφοδοτεί το επόμενο κύτταρο διαγώνια του τρέχοντος το οποίο δίνει το αποτέλεσμα τους πάντα δύο κύκλους μετά από το τρέχων κύτταρο και για αυτό εισάγουμε δύο flip flop .
- Τέλος το Sout πάντα τροφοδοτεί το αμέσως από κάτω κύτταρο στην εικόνα από το τρέχων το οποίο δίνει αποτέλεσμα έναν κύκλο μετά από το τρέχων. Για αυτό το λόγο βάλαμε εδώ ένα flip flop.

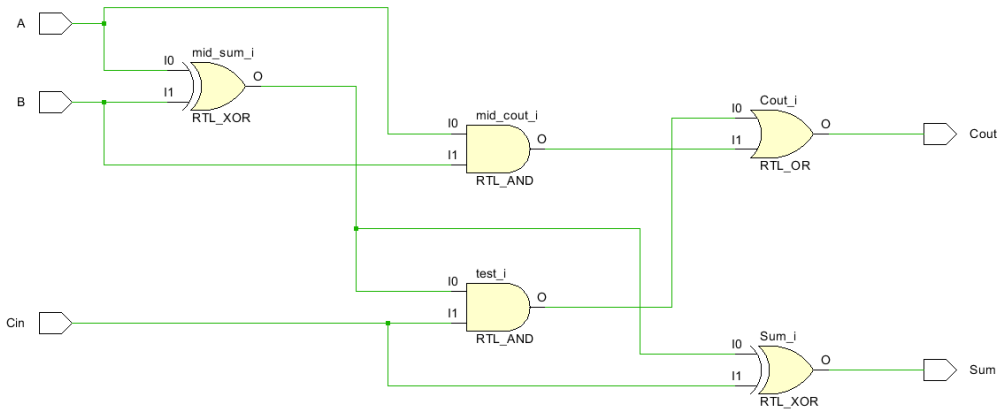
Αυτά ισχύουν σχεδόν σε όλες τις περιπτώσεις εκτός από τις εξής μεταβάσεις 4->6, 6->8 και 8->10. Εδώ βλέπουμε ότι το Cout πρέπει να τροφοδοτήσει ένα κύτταρο που δίνει το αποτέλεσμα του σε δύο κύκλους μακριά για αυτό θα εισάγουμε στο αντίστοιχο σημείο ένα ακόμη flip flop (αυτό θα είναι εκτός του βασικού κυττάρου)ώστε μαζί με το ήδη τοποθετημένο flip flop να δίνουν καθυστέρηση δύο κύκλων. Τέλος να σημειώσουμε ότι πρέπει για κάθε είσοδο και κάθε έξοδο να βάλουμε στην σειρά έναν αριθμό από flip flop απαραίτητο για τον συγχρονισμό , καθώς π.χ το b_1 θέλουμε να τροφοδοτεί το cell μας στον τρίτο κύκλο ή η έξοδος p_1 θέλουμε να καθυστερήσει 7 κύκλους, ώστε να συμβαδίζει με το αποτέλεσμα που δίνει το κύτταρο με την ένδειξη 10. Με τις

αντίστοιχες προσθήκες σε flip flop που περιγράψαμε πέρα από το βασικό κύτταρο πλέον μπορούμε εύκολα να περιγράψουμε το κύκλωμα μας εύκολα σε μία γλώσσα περιγραφής υλικού.

Εφόσον εξηγήσαμε την βασική δομή του κώδικα μας στην συνέχεια θα περάσουμε στην αναλυτική παρουσίαση του κώδικα μας σε VHDL. Αρχικά να αναφέρουμε ότι χρησιμοποιήσαμε από το πρώτο ερώτημα έναν σύγχρονο Full Adder, χωρίς όμως τα flip flop και απλώς για πληρότητα τον παραθέτουμε ακολούθως:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Full_Adder_no_pipe is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Cout : out STD_LOGIC);
end Full_Adder_no_pipe;

architecture Behavioral of Full_Adder_no_pipe is
signal mid_sum,mid_cout,test : std_logic;
begin
        mid_sum <= A xor B;
        Sum <= mid_sum xor Cin;
        mid_cout <= A and B;
        test <= mid_sum and Cin;
        Cout <= test or mid_cout;
end Behavioral;
```

Το RTL του Full Adder αυτού είναι:



Η τοπολογία είναι προφανής και δεν απαιτεί περαιτέρω ανάλυση. Στην συνέχεια ένα ακόμα βασικό component για την υλοποίηση μας είναι το απλό D flip-flop το οποίο παραθέτουμε ακολούθως:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_flip_flop_1_bit is
    Port ( Input : in std_logic;
           Clk : in std_logic ;
           Output : out std_logic);
end D_flip_flop_1_bit;

architecture Behavioral of D_flip_flop_1_bit is
    signal store_input : std_logic;
begin
        flip_flop_1bit : process (Clk)
            begin
                if Clk'event and Clk='1' then
                   store_input <= Input;
                end if;
            end process flip_flop_1bit;
    Output <=store_input;
end Behavioral;
```

Η λειτουργία του είναι αρκετά απλή. Αποτελείται από ένα process στο sensitivity list του οποίου έχουμε μόνο το ρολόι. Μέσα στο process ελέγχουμε με ένα if αν έχουμε μία θετική ακμή του ρολογιού και αν αυτό ισχύει αναθέτουμε σε ένα εσωτερικό σήμα την είσοδο. Στην έξοδο αναθέτουμε την τιμή του εσωτερικού αυτού σήματος και έτσι επιτυγχάνεται η λειτουργία του flip flop καθώς η έξοδος αλλάζει μόνο όταν αλλάζει το εσωτερικό σήμα. Συνεχίζουμε στο πλαίσιο των flip flop και στην συνέχεια φτιάχνουμε ένα component που θα μας βοηθήσει ώστε να είναι πιο ευανάγνωστη η υλοποίηση μας. Το component αυτό ουσιαστικά υλοποιεί N αριθμό flip flop στην σειρά, ώστε να επιτύχει καθυστέρηση N κύκλων σε ένα σήμα, χωρίς να χρειαστεί εμείς να γράψουμε ρητά τα N flip flop. Για να το κάνουμε αυτό χρησιμοποιούμε τον τύπο generic της VHDL και την δυνατότητα που παρέχει το for generate για αυτόματη παραγωγή component με βάση κάποιο loop. Παραθέτουμε ακολούθως τον κώδικα:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_flip_flop_N_seq is
    generic (
        N : positive
    );
    Port (
        Input : in std_logic ;
        Clk : in std_logic;
        Output : out std_logic
    );
end D_flip_flop_N_seq;
```

```
architecture struct of D_flip_flop_N_seq is
    signal store_input : std_logic;
    component D_flip_flop_1_bit is
       Port ( Input : in std_logic;
              Clk : in std_logic ;
              Output : out std_logic);
    end component;
    signal c : std_logic_vector (0 to N);
begin
  c(0) <= Input;
  flip_flops : for k in 0 to N-1 generate
                 Flip_flop : D_flip_flop_1_bit port map(c(k),Clk,c(k+1));
  end generate flip_flops;
  Output <= c(N);
end struct;
```

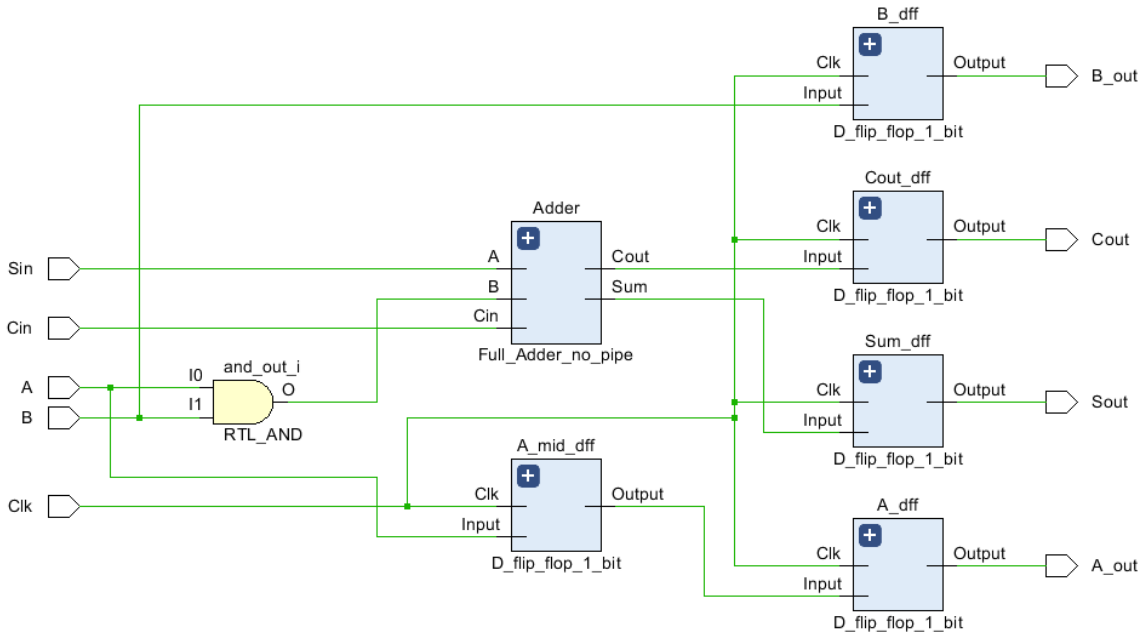
Παραπάνω βλέπουμε τον κώδικα για την υλοποίηση N flip flop στην σειρά. Όπως αναφέραμε χρησιμοποιείται ο τύπος generic για να καθοριστεί το πλήθος των flip flop που θα έχουμε στην σειρά, δηλαδή το πλήθος των κύκλων της καθυστέρηση που θα εισάγουμε στην είσοδο. Προφανώς χρησιμοποιούμε το component του ενός flip flop για να φτιάξουμε τα N εν σειρά flip flop. Ο κύριος κώδικας είναι αρκετά απλός, έχουμε ένα for loop στο οποίο σε κάθε iteration βάζουμε την έξοδο του προηγούμενου flip flop στην είσοδο του τρέχοντος. Η έξοδος του τελευταίου flip flop ανατίθεται στο output.

Τέλος, παραθέτουμε το τελευταίο component που θα χρειαστεί για την υλοποίηση μας , το οποίο θα είναι το κύτταρο του πολλαπλασιαστή. Για αυτό θα χρησιμοποιήσουμε προφανώς το D flip flop και τον Full adder που φτιάξαμε. Παραθέτουμε τον κώδικα:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Sys_mul_base_cell is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           Clk : in STD_LOGIC;
           Sin : in STD_LOGIC;
           A_out : out STD_LOGIC;
           B_out : out STD_LOGIC;
           Sout : out STD_LOGIC;
           Cout : out STD_LOGIC);
end Sys_mul_base_cell;

architecture Behavioral of Sys_mul_base_cell is
     component D_flip_flop_1_bit is
        Port ( Input : in std_logic;
               Clk : in std_logic ;
               Output : out std_logic);
```

```
    end component;
     component Full_Adder_no_pipe is
        Port ( A : in STD_LOGIC;
               B : in STD_LOGIC;
               Cin : in STD_LOGIC;
               Sum : out STD_LOGIC;
               Cout : out STD_LOGIC);
      end component;
      signal and_out,dff_B,dff_A,dff_Cout,dff_Sout : std_logic;
begin
and_out <= A and B;
Adder : Full_Adder_no_pipe port map (Sin,and_out,Cin,dff_Sout,dff_Cout);
B_dff : D_flip_flop_1_bit port map (B,Clk,B_out);
A_mid_dff : D_flip_flop_1_bit port map (A,Clk,dff_A);
A_dff : D_flip_flop_1_bit port map (dff_A,Clk,A_out);
Cout_dff : D_flip_flop_1_bit port map (dff_Cout,Clk,Cout);
Sum_dff : D_flip_flop_1_bit port map (dff_Sout,Clk,Sout);
end Behavioral;
```

Η υλοποίηση παραπάνω βασίζεται απολύτως στο σχηματικό που παραθέσαμε για το κύτταρο του πολλαπλασιαστή. Ο κώδικας είναι πολύ απλός, όπως φαίνονται στο σχήμα τα διάφορα component ακριβώς έτσι τα γράφουμε. Παραθέτουμε και το RTL του κυττάρου για να φανεί η ορθότητά του:



Παραπάνω στο RTL βλέπουμε ότι η υλοποίηση ακολουθεί ακριβώς το σχηματικό που δώσαμε για το κύτταρο του πολλαπλασιαστή γεγονός που επιβεβαιώνει την ορθότητα του. Εφόσον αναλύσαμε όλα τα βασικά component του πολλαπλασιαστή πλέον μπορούμε να παρουσιάσουμε την περιγραφή ολόκληρου του πολλαπλασιαστή σε VHDL:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Systolic_Mult is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Clk : in STD_LOGIC;
           P : out STD_LOGIC_VECTOR (7 downto 0));
end Systolic_Mult;

architecture Behavioral of Systolic_Mult is
    component D_flip_flop_N_seq is -- this component creates a delay of N cycles for a signal
                                   -- by creating N flip-flops connected in sequential order
        generic (
            N : positive
        );
        Port (
            Input : in std_logic ;
            Clk : in std_logic;
            Output : out std_logic
        );
    end component;
    component Sys_mul_base_cell is
        Port ( A : in STD_LOGIC;
               B : in STD_LOGIC;
               Cin : in STD_LOGIC;
               Clk : in STD_LOGIC;
               Sin : in STD_LOGIC;
               A_out : out STD_LOGIC;
               B_out : out STD_LOGIC;
               Sout : out STD_LOGIC;
               Cout : out STD_LOGIC);
    end component;
    component D_flip_flop_1_bit is
       Port ( Input : in std_logic;
              Clk : in std_logic ;
              Output : out std_logic);
    end component;
    type std_logic_matrix is array (4 downto 1, 4 downto 1) of std_logic;
    signal B_out, A_out, S_out, C_out : std_logic_matrix;
    signal A1_fix,A2_fix,A3_fix,B1_fix,B2_fix,B3_fix : std_logic;
    signal buf_4to6,buf_6to8,buf_8to10 : std_logic;

begin
    -- input delay
    A1_delay : D_flip_flop_1_bit   port map (A(1),Clk,A1_fix);
    A2_delay : D_flip_flop_N_seq generic map (N => 2)  port map (A(2),Clk,A2_fix);
    A3_delay : D_flip_flop_N_seq generic map (N => 3)  port map (A(3),Clk,A3_fix);
```

```
    B1_delay : D_flip_flop_N_seq generic map (N => 2)  port map (B(1),Clk,B1_fix);
    B2_delay : D_flip_flop_N_seq generic map (N => 4)  port map (B(2),Clk,B2_fix);
    B3_delay : D_flip_flop_N_seq generic map (N => 6)   port map (B(3),Clk,B3_fix);
    --1
    First_mul : Sys_mul_base_cell port map
(A(0),B(0),'0',Clk,'0',A_out(1,1),B_out(1,1),S_out(1,1),C_out(1,1));
    --2
    Second_mul : Sys_mul_base_cell port map
(A1_fix,B_out(1,1),C_out(1,1),Clk,'0',A_out(1,2),B_out(1,2),S_out(1,2),C_out(1,2));
    --3
    Third_up_mul : Sys_mul_base_cell port map
(A2_fix,B_out(1,2),C_out(1,2),Clk,'0',A_out(1,3),B_out(1,3),S_out(1,3),C_out(1,3));
    Third_down_mul : Sys_mul_base_cell port map
(A_out(1,1),B1_fix,'0',Clk,S_out(1,2),A_out(2,1),B_out(2,1),S_out(2,1),C_out(2,1));
    --4
    Fourth_up_mul : Sys_mul_base_cell port map
(A3_fix,B_out(1,3),C_out(1,3),Clk,'0',A_out(1,4),B_out(1,4),S_out(1,4),C_out(1,4));
    Fourth_down_mul : Sys_mul_base_cell port map
(A_out(1,2),B_out(2,1),C_out(2,1),Clk,S_out(1,3),A_out(2,2),B_out(2,2),S_out(2,2),C_out(2,2));
    --5
    Fifth_up_mul : Sys_mul_base_cell port map
(A_out(1,3),B_out(2,2),C_out(2,2),Clk,S_out(1,4),A_out(2,3),B_out(2,3),S_out(2,3),C_out(2,3));
    Fifth_down_mul : Sys_mul_base_cell port map
(A_out(2,1),B2_fix,'0',Clk,S_out(2,2),A_out(3,1),B_out(3,1),S_out(3,1),C_out(3,1));
    --6
    buffer_4to6 : D_flip_flop_1_bit   port map (C_out(1,4),Clk,buf_4to6);
    Sixth_up_mul : Sys_mul_base_cell port map
(A_out(1,4),B_out(2,3),C_out(2,3),Clk,buf_4to6,A_out(2,4),B_out(2,4),S_out(2,4),C_out(2,4));
    Sixth_down_mul : Sys_mul_base_cell port map
(A_out(2,2),B_out(3,1),C_out(3,1),Clk,S_out(2,3),A_out(3,2),B_out(3,2),S_out(3,2),C_out(3,2));
    --7
    Seventh_up_mul : Sys_mul_base_cell port map
(A_out(2,3),B_out(3,2),C_out(3,2),Clk,S_out(2,4),A_out(3,3),B_out(3,3),S_out(3,3),C_out(3,3));
    Seventh_down_mul : Sys_mul_base_cell port map
(A_out(3,1),B3_fix,'0',Clk,S_out(3,2),A_out(4,1),B_out(4,1),S_out(4,1),C_out(4,1));
    --8
    buffer_6to8 : D_flip_flop_1_bit   port map (C_out(2,4),Clk,buf_6to8);
    eighth_up_mul : Sys_mul_base_cell port map
(A_out(2,4),B_out(3,3),C_out(3,3),Clk,buf_6to8,A_out(3,4),B_out(3,4),S_out(3,4),C_out(3,4));
    eighth_down_mul : Sys_mul_base_cell port map
(A_out(3,2),B_out(4,1),C_out(4,1),Clk,S_out(3,3),A_out(4,2),B_out(4,2),S_out(4,2),C_out(4,2));
    --9
    ninth_mul : Sys_mul_base_cell port map
(A_out(3,3),B_out(4,2),C_out(4,2),Clk,S_out(3,4),A_out(4,3),B_out(4,3),S_out(4,3),C_out(4,3));
    --10
    buffer_8to10 : D_flip_flop_1_bit   port map (C_out(3,4),Clk,buf_8to10);
    tenth_mul : Sys_mul_base_cell port map
(A_out(3,4),B_out(4,3),C_out(4,3),Clk,buf_8to10,A_out(4,4),B_out(4,4),P(6),P(7));
```

```
    -- Fix final outputs with modular delay
    P0_fix : D_flip_flop_N_seq generic map (N => 9)  port map (S_out(1,1),Clk,P(0));
    P1_fix : D_flip_flop_N_seq generic map (N => 7)  port map (S_out(2,1),Clk,P(1));
    P2_fix : D_flip_flop_N_seq generic map (N => 5)  port map (S_out(3,1),Clk,P(2));
    P3_fix : D_flip_flop_N_seq generic map (N => 3)  port map (S_out(4,1),Clk,P(3));
    P4_fix : D_flip_flop_N_seq generic map (N => 2)  port map (S_out(4,2),Clk,P(4));
    P5_fix : D_flip_flop_1_bit   port map (S_out(4,3),Clk,P(5));

end Behavioral;
```

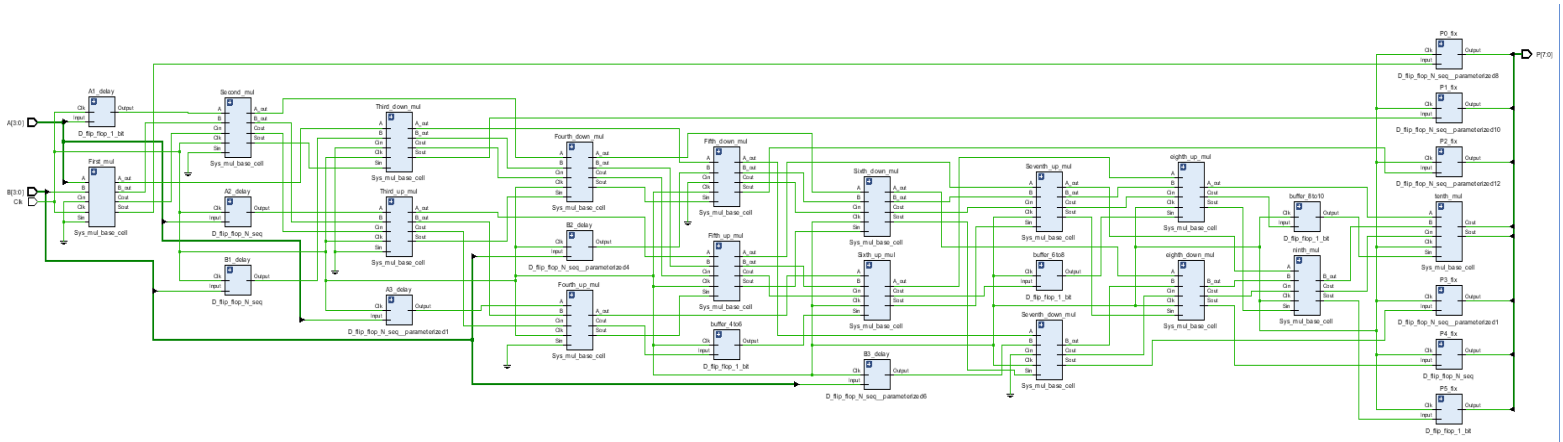
Ακολούθως θα αναλύσουμε τον κώδικα. Αρχικά προφανώς ορίζουμε το entity και τις εισόδους και τις εξόδους (A και B οι 4bit είσοδοι και P η 8 bit έξοδος). Στην συνέχεια εισάγουμε τα απαραίτητα components που είναι το κύτταρο του πολλαπλασιαστή και τα N flip flop στην σειρά (αναλύσαμε την λειτουργία τους παραπάνω). Ακολούθως ορίζουμε όποια βοηθητικά σήματα χρειαζόμαστε για την υλοποίηση μας:

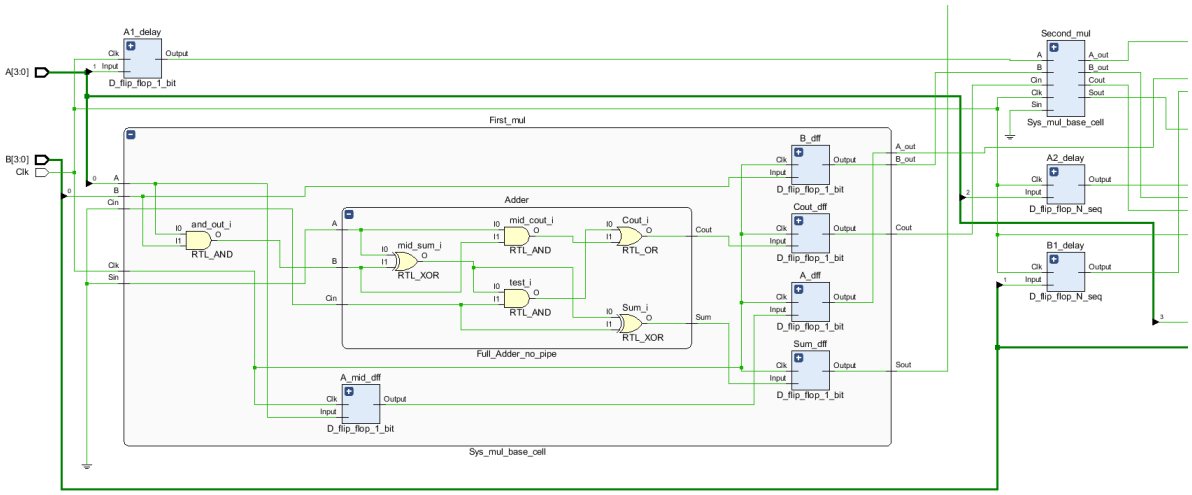
- Ορίζουμε έναν δισδιάστατο πίνακα 4*4, ώστε κάθε είσοδος και έξοδος ενός multiplier cell να χαρακτηρίζεται από την θέση της στον 4*4 πίνακα από multiplier cells όπως τον βλέπουμε στην εικόνα που παραθέσαμε στην αρχή. Ουσιαστικά κάθε κύτταρο του πολλαπλασιαστή έχει σαν εξόδους τα a , b , c_out και s_out . Η αρίθμηση ξεκινάει ανάποδα από ότι σε έναν συνηθισμένο πίνακα, με βάση το πως βλέπουμε την εικόνα, ώστε να συμβαδίζει περισσότερο με την ροή της πληροφορίας. Το κελί (1,1) αντιστοιχεί στο πάνω δεξιά κύτταρο του πολλαπλασιαστή, το οποίο παράγει πρώτο το αποτέλεσμα του, ενώ το κάτω αριστερά κύτταρο αντιστοιχεί στην θέση (4,4) . Με βάση αυτή την λογική έχουν δημιουργηθεί αυτοί οι 4*4 πίνακες, ώστε να είναι ξεκάθαρο κάθε σήμα από ποιο κύτταρο είναι έξοδος και ποια έξοδος είναι.
- Ορίζουμε κάποια σήματα που τα ονομάζουμε Ai_fix και Bi_fix (με i=1,2,3) τα οποία θα είναι τα κατάλληλα καθυστερημένα σήματα εισόδου που θα δοθούν στον πολλαπλασιαστή. Πιο συγκεκριμένα όπως εξηγήσαμε και παραπάνω ο πολλαπλασιαστής δεν χρησιμοποιεί όλες τις εισόδους στον πρώτο κύκλο και συνεπώς ανάλογα με τον κύκλο που κάθε είσοδος προσφέρεται στον πολλαπλασιαστή πρέπει να εφαρμόσουμε σε αυτή την είσοδο καθυστέρηση για ανάλογο αριθμό κύκλων. Συνεπώς αυτά τα σήματα θα είναι χρήσιμα για τον συγχρονισμό του κυκλώματος
- Ορίζουμε 3 σήματα buffers που ουσιαστικά συμβάλλουν ώστε να επιτευχθεί η καθυστέρηση ενός κύκλου στα σήματα C_out που μεταφέρονται από τα κελιά (4,i) στο (4,i+1) για λόγγους που εξηγήσαμε παραπάνω.

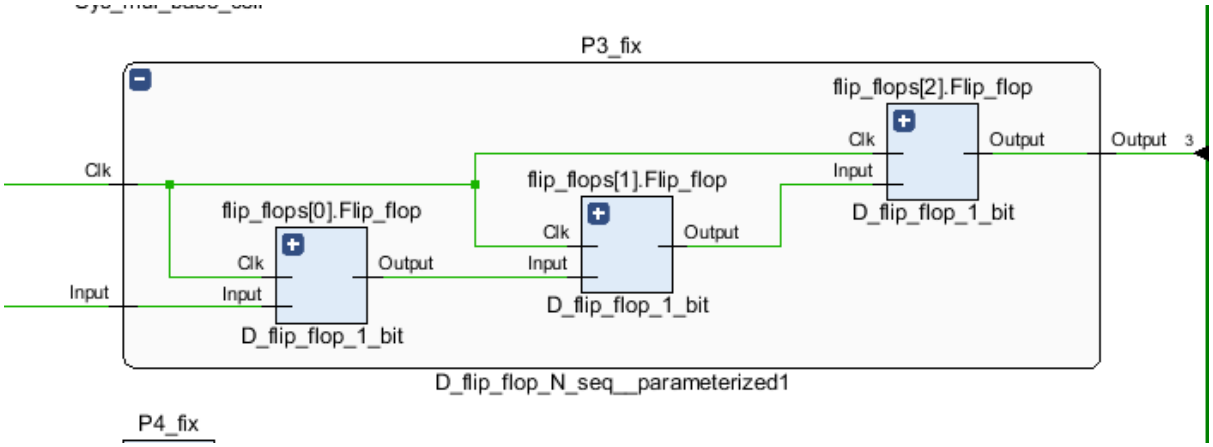
Εφόσον πλέον εξηγήσαμε την χρησιμότητα όλων των επιπλέον σημάτων προχωράμε στην περιγραφή της αρχιτεκτονικής του πολλαπλασιαστή. Αρχικά, χρησιμοποιώντας το component των εν σειρά flip flop εφαρμόζουμε την κατάλληλη καθυστέρηση σε κάθε είσοδο, όπως εξηγήσαμε και στον ορισμό των σημάτων Ai_fix και Bi_fix. Στην συνέχεια ορίζουμε ένα ένα τα κύτταρα του πολλαπλασιαστή με βάση την αρίθμησή τους στην εικόνα. Η αρίθμηση στην εικόνα βασίζεται στην εξής λογική: κάθε cell έχει τον αριθμό του κύκλου στον οποίο παράγει τις εξόδους του. Άρα το cell με τον αριθμό 1 παράγει όλες τις εξόδους του στον πρώτο κύκλο, αυτό με τον αριθμό 4 παράγει όλες τις εξόδους στον τέταρτο κύκλο κ.ο.κ (για αυτό και κάποια κύτταρα έχουν τον ίδιο αριθμό). Συνεπώς στην συνέχεια θέτουμε τις εισόδους και τις εξόδους για κάθε cell όπως φαίνεται στο σχήμα. Για κάθε cell προφανώς χρησιμοποιούμε το

component Που φτιάξαμε προηγουμένως. Σε κάποια σημεία προσθέτουμε και ένα flip flop σαν buffer χρησιμοποιώντας τα ανάλογα σήματα που αναλύσαμε παραπάνω. Τέλος, εφαρμόζουμε μέσω των N flip flop εν σειρά την κατάλληλη καθυστέρηση σε κάθε έξοδο ώστε το σήμα να είναι πλήρως συγχρονισμένο.

Με βάση τον παραπάνω κώδικα παράγουμε και το RTL που φαίνεται ακολούθως:





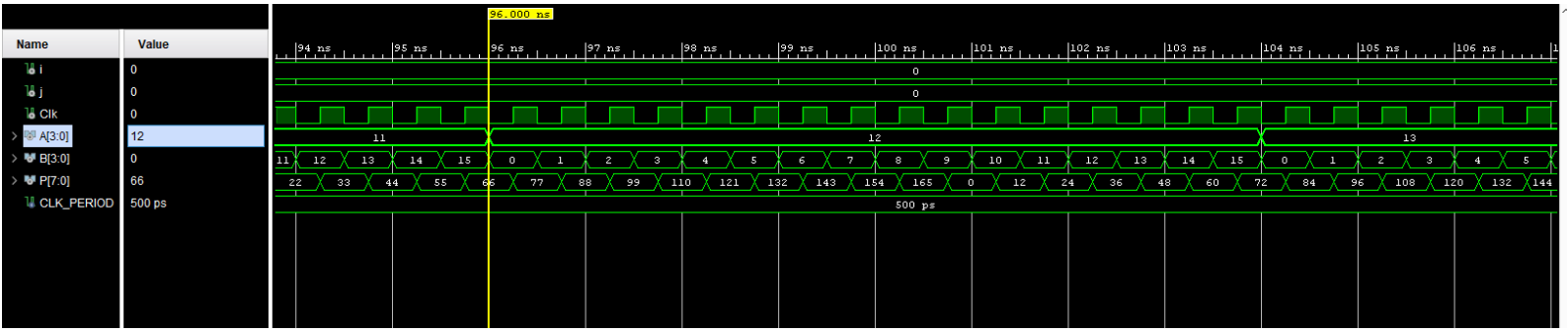


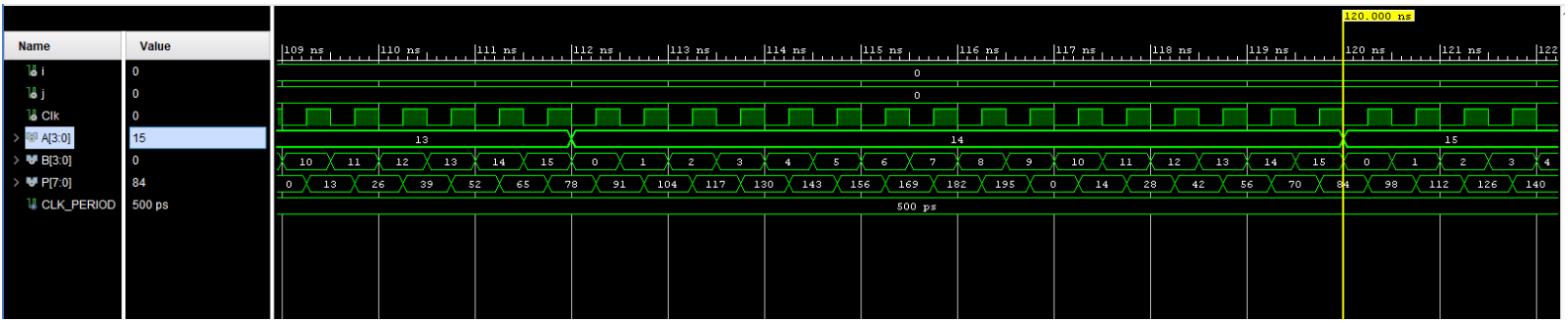
Παραπάνω βλέπουμε το RTL που παράγεται από το Vivado για τον κώδικα που παραθέσαμε. Η πρώτη εικόνα αποτελεί το ολοκληρωμένο κύκλωμα του πολλαπλασιαστεί και είναι ίδια ουσιαστικά με το σχεδιάγραμμα που βασίστηκε η λύση μας, μόνο που έχουν προστεθεί κάποια flip flop σε ορισμένα σημεία επιπλέον για να επιτευχθεί συγχρονισμός των εισόδων, ώστε κάθε είσοδος να λαμβάνεται όποτε χρειάζεται, και των εξόδων, ώστε όλες οι έξοδοι να παρέχονται στο τέλος ταυτόχρονα (επιπλέον έχουμε και κάποια flip flop που υλοποιούν τους buffers που αναφέραμε αναλυτικά παραπάνω). Από την εποπτική αυτή εξέταση συμπεραίνουμε ότι το κύκλωμα μας είναι ορθό. Στην συνέχεια παρέχουμε δύο εικόνες όπου έχουμε «ανοίξει» δύο component του πολλαπλασιαστή. Στην πρώτη εικόνα βλέπουμε το κύτταρο ενός πολλαπλασιαστή και στην δεύτερη βλέπουμε τα N flip flop για N=3 (ενδεικτικά επιλέχθηκε αυτό θα μπορούσε να είναι οποιοδήποτε N). Και αυτά τα υποκυκλώματα φαίνονται ορθά υλοποιημένα με βάση το RTL. Για να ελέγξουμε όμως και την ορθή λειτουργία τους θα γράψουμε ένα testbench για την παραπάνω περιγραφή υλικού. Στην συνέχεια παραθέτουμε το testbench και τα αποτελέσματα του:

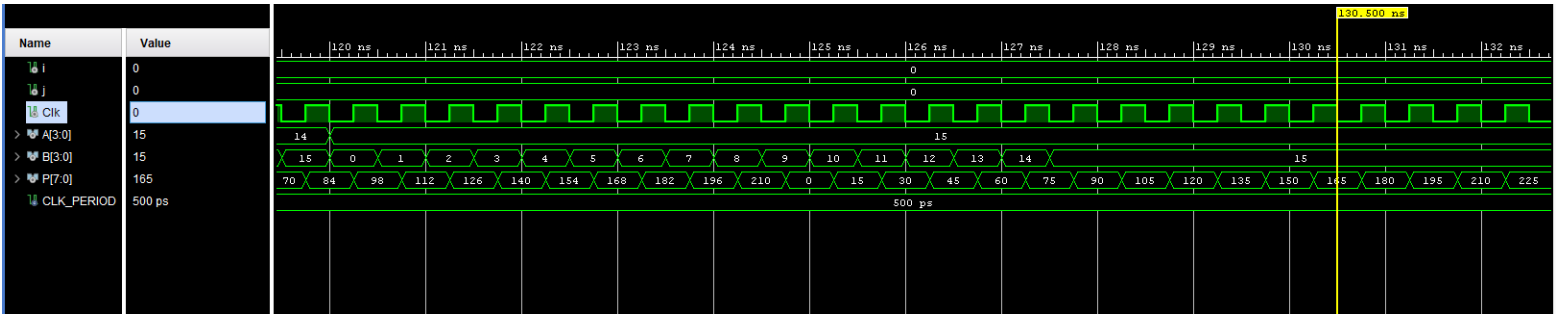
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity tb_mult is
end tb_mult;

architecture testbench of tb_mult is
    constant CLK_PERIOD  : time := 0.5 ns; -- Clock period
    component Systolic_Mult is
        Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
               B : in STD_LOGIC_VECTOR (3 downto 0);
               Clk : in STD_LOGIC;
               P : out STD_LOGIC_VECTOR (7 downto 0));
    end component;
    signal i,j : integer :=0;
    signal Clk : std_logic;
    signal A,B : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
begin
clock : process
        begin
            while true loop
                Clk <= '0';
                wait for CLK_PERIOD /2;
                Clk <= '1';
                wait for CLK_PERIOD /2;
            end loop;
        end process;
   uut: Systolic_Mult port map(A,B,Clk,P);
   stim_process: process
    begin
      for i in 0 to 15 loop
        for j in 0 to 15 loop
            A <= std_logic_vector(to_unsigned(i, A'length));
```

```
            B <= std_logic_vector(to_unsigned(j, B'length));
            wait for CLK_PERIOD ;
          end loop;
        end loop;
        wait;
    end process;
end testbench;
```

Στο testbench μας αρχικά ορίζουμε ένα κενό entity που θα είναι το testbench και στην αρχιτεκτονική του εισάγουμε σαν component τον πολλαπλασιαστή. Ορίζουμε στο πλαίσιο του testbench τις εισόδους και την έξοδο (A,B,Clk και P) και κάνουμε ένα map των σημάτων αυτών του testbench με τα αντίστοιχα σήματα του component. Στην συνέχεια, ορίζουμε ένα process για το ρολόι , ώστε αυτό να λειτουργεί συνέχεια και ανεξάρτητα από οποιαδήποτε άλλη διαδικασία. Τέλος, ορίζουμε μία διαδικασία όπου σε κάθε κύκλο ρολογιού δίνουμε μία διαφορετική είσοδο A και B . Αυτό το επιτυγχάνουμε με την χρήση δύο for loop ώστε να ελέγξουμε όλους τους πιθανούς συνδυασμούς A και B. Το αποτέλεσμα από την εκτέλεση του παραπάνω testbench είναι το ακόλουθο(για τα σήματα έχει επιλεχθεί να απεικονιστούν σε δεκαδικό σύστημα, ώστε να είναι πιο ευανάγνωστα).







Βλέπουμε και από το testbench ότι η υλοποίηση μας επιτυγχάνει το επιθυμητό αποτέλεσμα, δηλαδή σε κάθε κύκλο ρολογιού παράγει ένα αποτέλεσμα πολλαπλασιασμού ορθά (προφανώς βέβαια το αποτέλεσμα παράγεται δέκα κύκλους πιο αργά από όταν θα εφαρμοστεί η είσοδος). Το σημαντικότερο που επιτύχαμε με την παραπάνω υλοποίηση είναι ότι σε κάθε κύκλο ρολογιού παράγεται ένα αποτέλεσμα πολλαπλασιασμού! Αυτή την δύναμη μας δίνει η σωλήνωση επιτυγχάνοντας σημαντικά μεγαλύτερο throuput, όπως αναφέραμε και στο τέλος του 2ου ζητήματος.

Τέλος, αναφέρουμε το critical path , δηλαδή το μονοπάτι που εισάγει την μεγαλύτερη καθυστέρηση στο κύκλωμα που φτιάχνει ο synthesizer του Vivado. Αυτό μπορούμε να το βρούμε πραγματοποιώντας μία σύνθεση και βλέποντας το timing report:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	So
Path 1	∞	2	2	1	P5_fix/store_input_reg/C	P[5]	4.076	3.276	0.800	∞	
Path 2	∞	2	2	1	tenth_mul/Sum_...e_input_reg/C	P[6]	4.076	3.276	0.800	∞	
Path 3	∞	2	2	1	tenth_mul/Cout...re_input_reg/C	P[7]	4.076	3.276	0.800	∞	
Path 4	∞	2	2	1	P0_fix/flip_flop...tore_input_reg/C	P[0]	4.058	3.258	0.800	∞	
Path 5	∞	2	2	1	P1_fix/flip_flop...tore_input_reg/C	P[1]	4.058	3.258	0.800	∞	
Path 6	∞	2	2	1	P2_fix/flip_flop...tore_input_reg/C	P[2]	4.058	3.258	0.800	∞	

Παραπάνω βλέπουμε ότι έχουμε 2 critical path ουσιαστικά. Το ένα είναι ο χρόνος που κάνει ο δέκατος multiplier να παράγει τις εξόδους του ,ενώ το δεύτερο είναι ο χρόνος που κάνει η έξοδος του ένατου πολλαπλασιαστή να περάσει από 1 flip flop και να πάει στην έξοδο. Η συνολική καθυστέρηση καθενός μονοπατιού φαίνεται στο πεδίο total delay.