Ψηφιακά Συστήματα **VLSI**

8ο εξάμηνο, Ακαδημαϊκή περίοδος 2024-2025

3η Εργαστηριακή Άσκηση

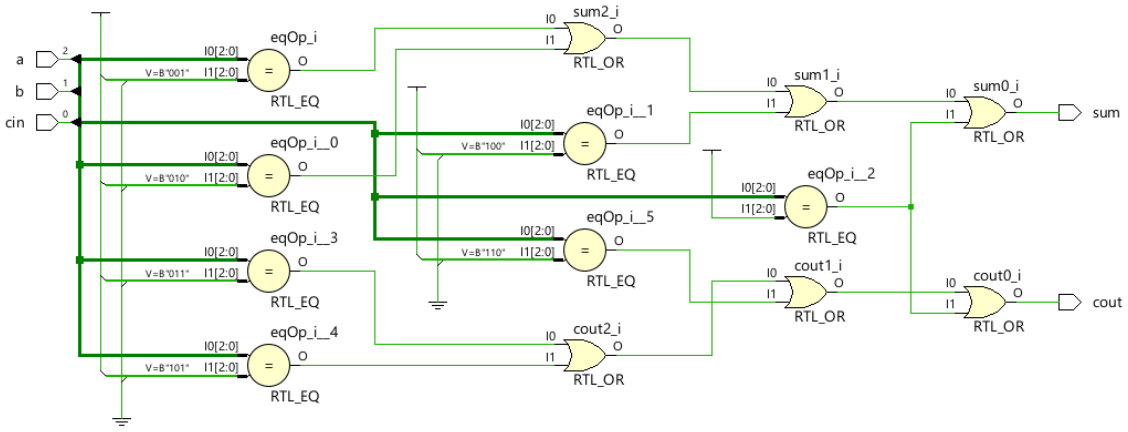
Δημήτρης Καμπανάκης: 03121012
Αγγελική Ζέρβα: 03121101

Θέμα 1: Σύγχρονος Πλήρης Αθροιστής (Full Adder - FA) με περιγραφή συμπεριφοράς (Behavioral).

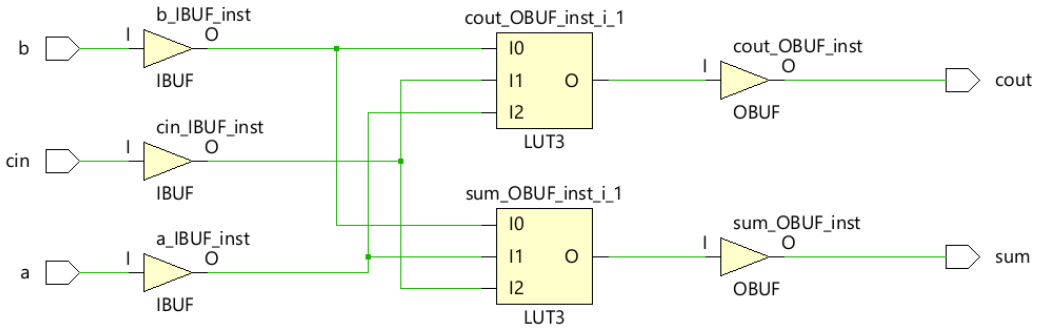
Ο κώδικας σε vhdl για την περιγραφή του Full Adder και της αρχιτεκτονικής του σε behavioral είναι ο εξής:

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

entity full_adder 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;

architecture behavioral of full_adder is

    signal input: std_logic_vector(2 downto 0);

    begin
        input <= a & b & cin;
        process (input)
        begin

        ---for SUM
        if (input = "001" or input = "010" or input = "100" or input = "111") then
        sum <= '1';

        else
        sum <= '0';
        end if;

        ---for CARRY
        if (input = "011" or input = "101" or input = "110" or input = "111") then
        cout <= '1';
        else
        cout <= '0';
        end if;

        end process;
end Behavioral;
```

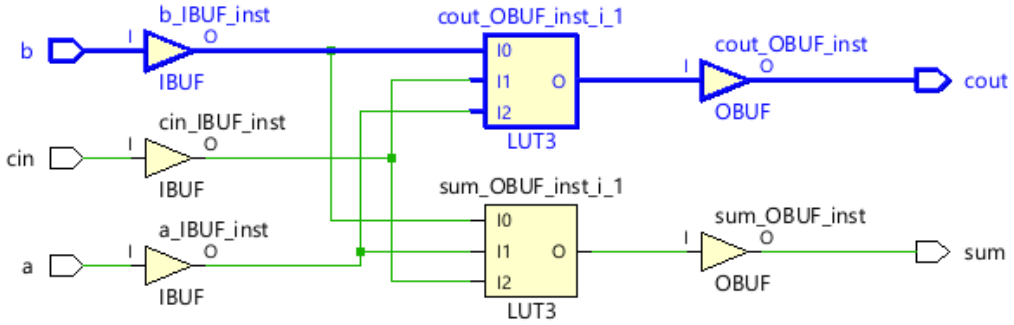
Με τη χρήση του elaborate design παίρνουμε το εξής σχηματικό:



Με τη χρήση του synthesis παίρνουμε το εξής σχηματικό:



Χρησιμοποιώντας το timing summary για το κύκλωμα μας μπορούμε να βρούμε το κρίσιμο μονοπάτι του κυκλώματος μας, το οποίο φαίνεται παρακάτω:



Όπως φαίνεται παρακάτω η χρονική καθυστέρηση του critical path του κυκλώματος είναι 7.012 ns.

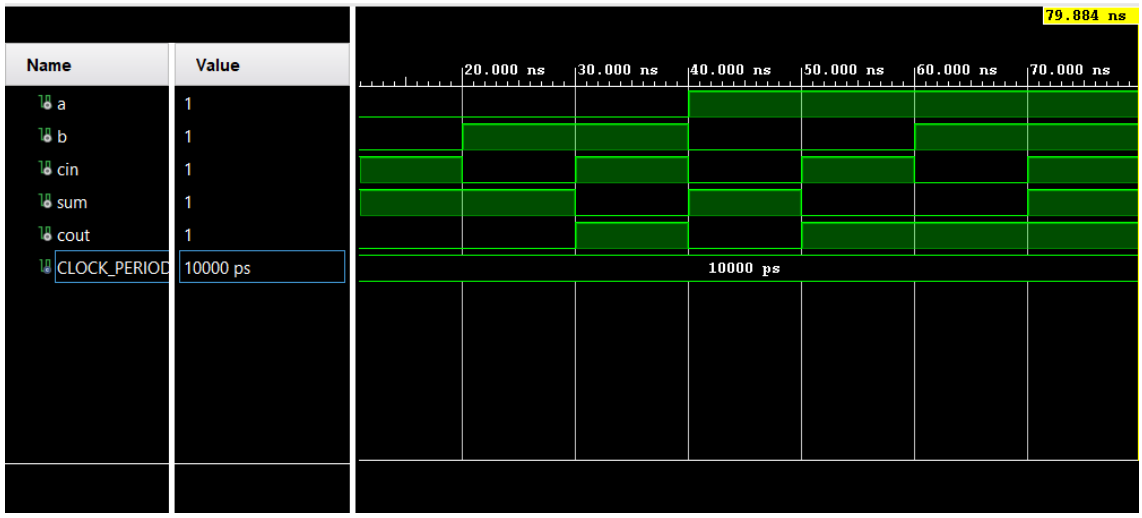
Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	3	2	2	b	cout	7.012	3.968	3.044	∞	input port clock
Path 2	∞	3	2	2	b	sum	6.947	3.771	3.176	∞	input port clock

Ο εντοπισμός του κρίσιμου μονοπατιού είναι σημαντικός, καθώς μπορούμε να παρατηρήσουμε ποιο μονοπάτι δεδομένων είναι το πιο χρονοβόρο στην σχεδίαση μας. Έτσι, αν χρειαστούμε βελτιώσεις στον χρόνο ξέρουμε ποιο μονοπάτι χρειάζεται να αλλάξει. Επιπλέον, η μέγιστη χρονική καθυστέρηση επηρεάζει άμεσα την επιλογή της ελάχιστης συχνότητας ρολογιού.

Για την προσομοίωση και τον έλεγχο της σωστής λειτουργίας του κυκλώματός μας χρησιμοποιούμε το παρακάτω testbench:

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

entity full_adder_tb is
end entity;

architecture tb of full_adder_tb is
    component full_adder 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 a, b, cin: std_logic;
    signal sum,cout: std_logic;

    constant CLOCK_PERIOD : time := 10 ns;


begin
    uut: full_adder port map(
        a => a,
        b => b,
        cin => cin,
        sum => sum,
        cout => cout
    );

    process
    begin
        a <= '0';
        b <= '0';
        cin <= '0';
        wait for CLOCK_PERIOD;

        a <= '0';
        b <= '0';
        cin <= '1';
        wait for CLOCK_PERIOD;

        a <= '0';
```

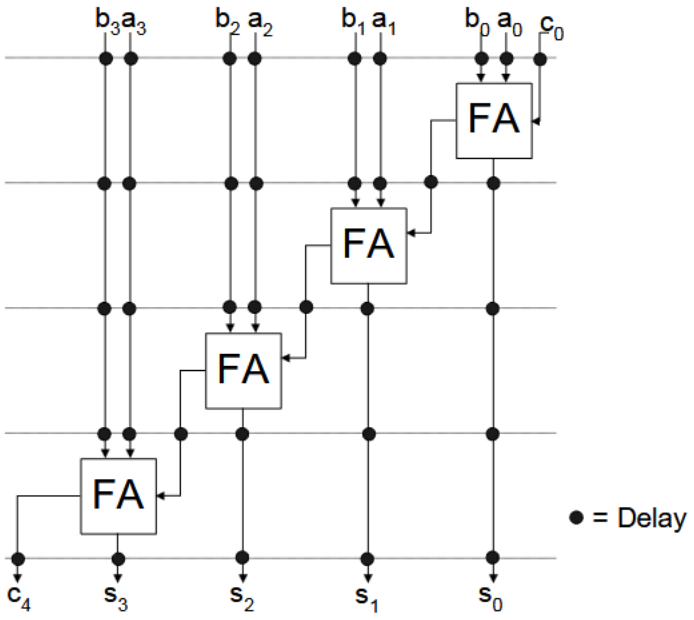
```
        b <= '1';
        cin <= '0';
        wait for CLOCK_PERIOD;

        a <= '0';
        b <= '1';
        cin <= '1';
        wait for CLOCK_PERIOD;

        a <= '1';
        b <= '0';
        cin <= '0';
        wait for CLOCK_PERIOD;

        a <= '1';
        b <= '0';
        cin <= '1';
        wait for CLOCK_PERIOD;

        a <= '1';
        b <= '1';
        cin <= '0';
        wait for CLOCK_PERIOD;

        a <= '1';
        b <= '1';
        cin <= '1';
        wait for CLOCK_PERIOD;

        wait;
    end process;

end tb;
```



Παρατηρούμε ότι έχουμε ορθά αποτελέσματα για όλους τους συνδυασμούς εισόδων.

Θέμα 2: Σύγχρονος Αθροιστής διάδοσης κρατουμένου των 4 bits με χρήση της τεχνικής Pipeline

Η σχεδίαση του σύγχρονου αθροιστή διάδοσης κρατουμένου των 4 bits (Synchronous Carry Propagation adder) με πλήρεις αθροιστές και χρήση της τεχνικής pipeline βασίζεται στο παρακάτω κύκλωμα:



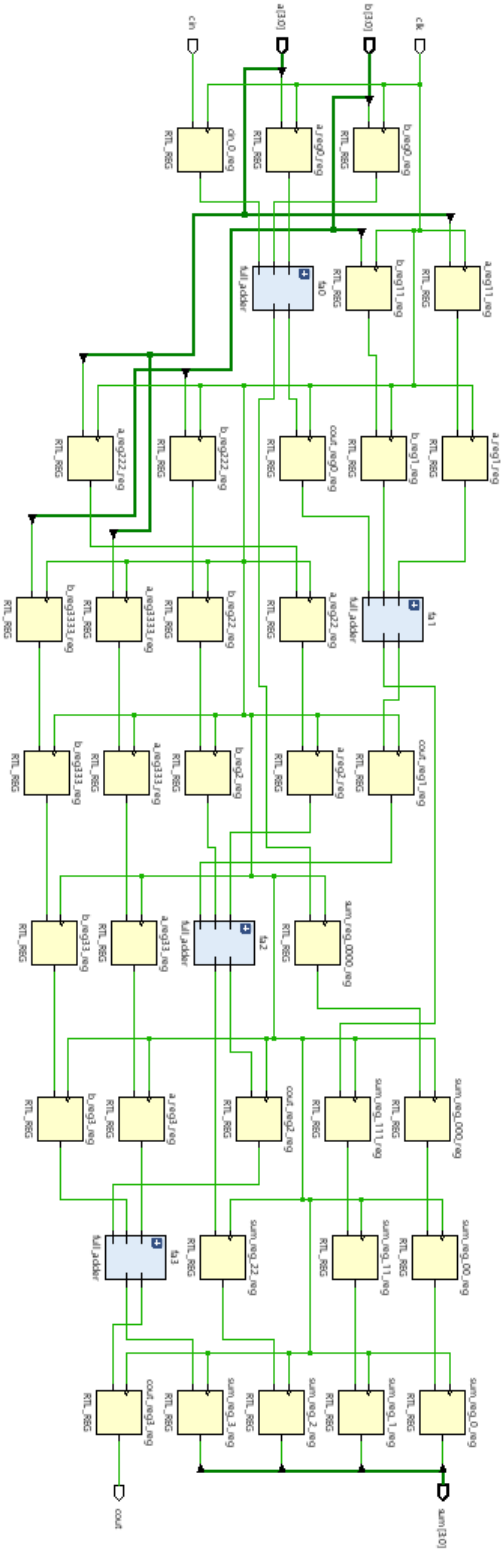
Ο κώδικας σε vhdl για την περιγραφή του κυκλώματος και της αρχιτεκτονικής του είναι ο εξής:

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

entity pipelined_adder is
    Port ( clk  : in STD_LOGIC;
           a    : in STD_LOGIC_VECTOR (3 downto 0);
           b    : in STD_LOGIC_VECTOR (3 downto 0);
           cin  : in STD_LOGIC;
           sum  : out STD_LOGIC_VECTOR (3 downto 0);
           cout : out STD_LOGIC);
end pipelined_adder;

architecture Behavioral of pipelined_adder is

    signal temp_sum: std_logic_vector(3 downto 0);
    signal carry : std_logic_vector(3 downto 0);

    -- Pipeline Registers
    signal a_reg0, a_reg11, a_reg1, a_reg222, a_reg22, a_reg2 : std_logic;
    signal a_reg3333, a_reg333, a_reg33, a_reg3 : std_logic;
    signal b_reg0, b_reg11, b_reg1, b_reg222, b_reg22, b_reg2 : std_logic;
    signal b_reg3333, b_reg333, b_reg33, b_reg3 : std_logic;
    signal sum_reg_0000, sum_reg_000, sum_reg_00, sum_reg_0  : std_logic;
    signal sum_reg_111, sum_reg_11, sum_reg_1, sum_reg_22, sum_reg_2, sum_reg_3 : std_logic;
```

```
    --Carry signals
    signal cout_reg0, cout_reg1, cout_reg2, cout_reg3 : std_logic;

    signal cin_0 : std_logic;

    -- Full Adder Component
    component full_adder is
        Port ( a    : in STD_LOGIC;
               b    : in STD_LOGIC;
               cin  : in STD_LOGIC;
               sum  : out STD_LOGIC;
               cout : out STD_LOGIC);
    end component;

begin
    -- **Pipeline Registers for input**
    process (clk)
    begin
        if rising_edge(clk) then
            cin_0 <= cin;
            a_reg0 <= a(0);
            b_reg0 <= b(0);
            a_reg11 <= a(1);
            b_reg11 <= b(1);
            a_reg222 <= a(2);
            b_reg222 <= b(2);
            a_reg3333 <= a(3);
            b_reg3333 <= b(3);

            a_reg1 <= a_reg11;
            b_reg1 <= b_reg11;
            a_reg22 <= a_reg222;
            b_reg22 <= b_reg222;
            a_reg333 <= a_reg3333;
            b_reg333 <= b_reg3333;
            sum_reg_0000 <= temp_sum(0);
            cout_reg0 <= carry(0);

            a_reg2 <= a_reg22;
            b_reg2 <= b_reg22;
            a_reg33 <= a_reg333;
            b_reg33 <= b_reg333;
            sum_reg_000 <= sum_reg_0000;
            sum_reg_111 <= temp_sum(1);
            cout_reg1 <= carry(1);


            a_reg3 <= a_reg33;
            b_reg3 <= b_reg33;
            sum_reg_00 <= sum_reg_000;
            sum_reg_11 <= sum_reg_111;
            sum_reg_22 <= temp_sum(2);
            cout_reg2 <= carry(2);
```

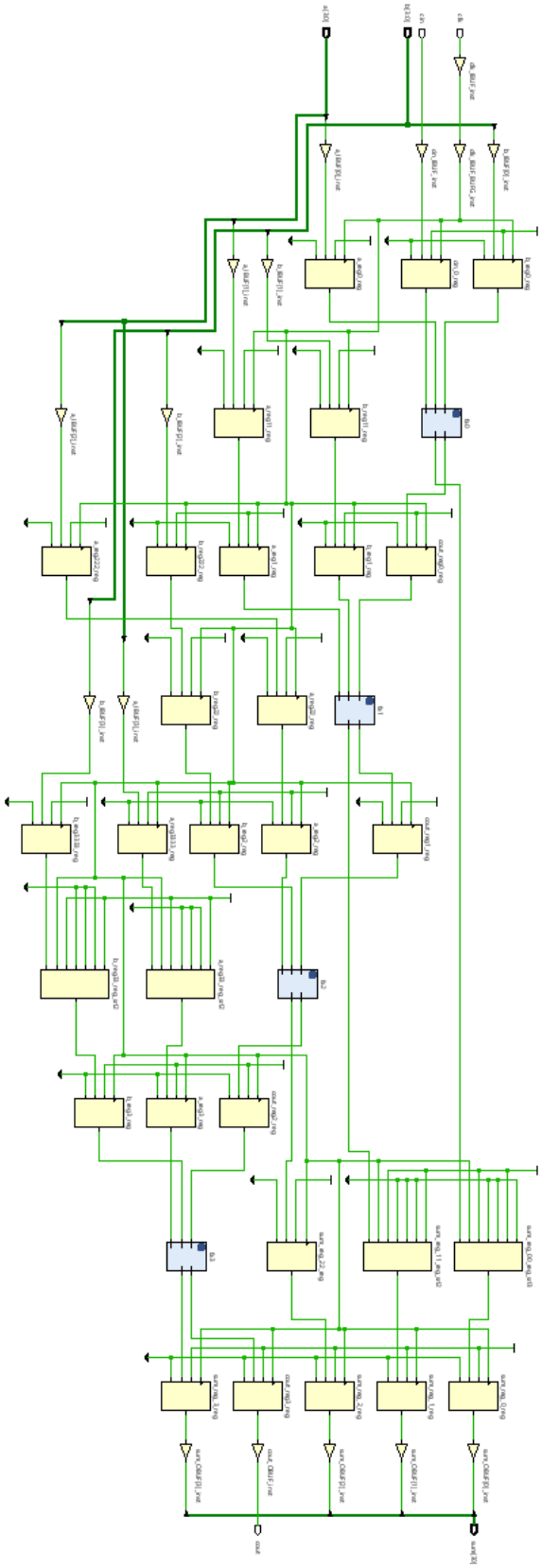
```
            sum_reg_0 <= sum_reg_00;
            sum_reg_1 <= sum_reg_11;
            sum_reg_2 <= sum_reg_22;
            sum_reg_3 <= temp_sum(3);
            cout_reg3 <= carry(3);
        end if;
    end process;

    -- FAs
    fa0: full_adder port map (
        a    => a_reg0,
        b    => b_reg0,
        cin  => cin_0,
        sum  => temp_sum(0),
        cout => carry(0)
    );

    fa1: full_adder port map (
        a    => a_reg1,
        b    => b_reg1,
        cin  => cout_reg0,
        sum  => temp_sum(1),
        cout => carry(1)
    );

    fa2: full_adder port map (
        a    => a_reg2,
        b    => b_reg2,
        cin  => cout_reg1,
        sum  => temp_sum(2),
        cout => carry(2)
    );

    fa3: full_adder port map (
        a    => a_reg3,
        b    => b_reg3,
        cin  => cout_reg2,
        sum  => temp_sum(3),
        cout => carry(3)
    );
    -- **Final Output**
    sum  <= sum_reg_3 & sum_reg_2 & sum_reg_1 & sum_reg_0;
    cout <= cout_reg3;

end Behavioral;
```

Με τη χρήση του elaborate design παίρνουμε το εξής σχηματικό:

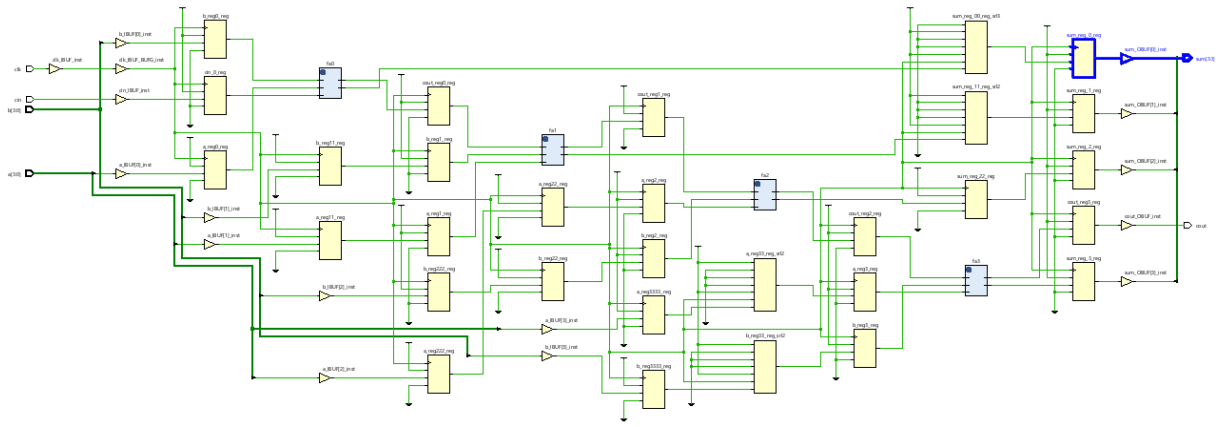


Στο παραπάνω σχηματικό μπορούμε να παρατηρήσουμε τους τέσσερεις full adders, καθώς και τους επιπλέον buffers που χρησιμοποιούνται για τις καθυστερήσεις.

Με τη χρήση του synthesis παίρνουμε το εξής σχηματικό:



Χρησιμοποιώντας το timing summary για το κύκλωμα μας μπορούμε να βρούμε το κρίσιμο μονοπάτι του κυκλώματος μας, το οποίο φαίνεται παρακάτω:



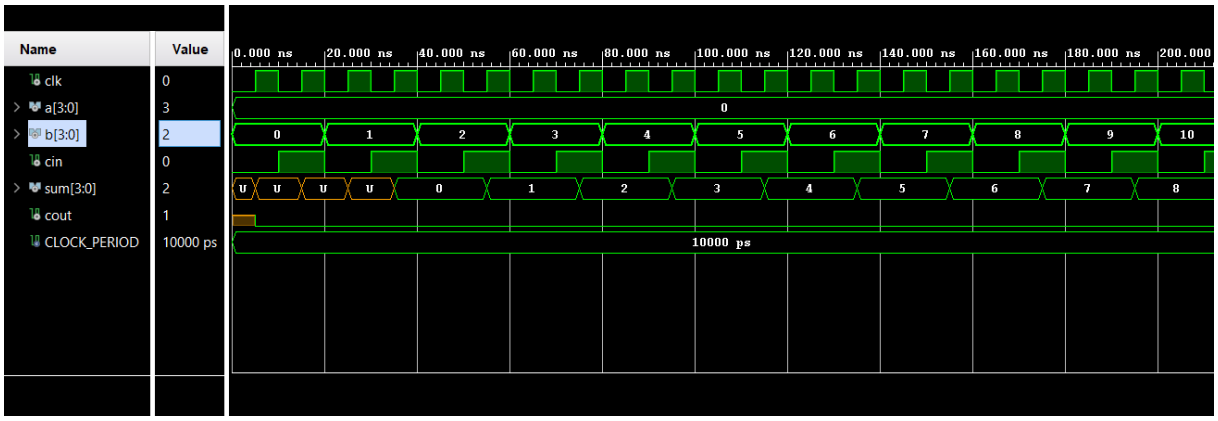
Όπως φαίνεται παρακάτω η χρονική καθυστέρηση του critical path του κυκλώματος είναι 5.169 ns. Αξίζει να σημειώσουμε ότι αφού μιλάμε για τεχνική pipelining η καθυστέρηση αφορά στάδιο του pipeline και όχι path από την είσοδο στην έξοδο .

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	2	1	1	sum_reg_0_reg/C	sum[0]	5.196	3.336	1.860	∞	
Path 2	∞	2	1	1	sum_reg_3_reg/C	sum[3]	5.094	3.219	1.875	∞	
Path 3	∞	2	1	1	cout_reg3_reg/C	cout	5.077	3.074	2.003	∞	
Path 4	∞	2	1	1	sum_reg_1_reg/C	sum[1]	4.992	3.266	1.726	∞	
Path 5	∞	2	1	1	sum_reg_2_reg/C	sum[2]	4.823	3.150	1.673	∞	
Path 6	∞	1	1	1	cin	cin_0_reg/D	2.079	1.024	1.054	∞	input port clock
Path 7	∞	1	1	1	b[3]	b_reg3333_reg/D	2.055	1.005	1.050	∞	input port clock
Path 8	∞	1	1	1	b[0]	b_reg0_reg/D	2.049	1.017	1.031	∞	input port clock
Path 9	∞	1	1	1	b[1]	b_reg11_reg/D	2.036	1.018	1.018	∞	input port clock
Path 10	∞	1	1	1	a[2]	a_reg222_reg/D	1.993	0.988	1.005	∞	input port clock

Για την προσομοίωση και τον έλεγχο της σωστής λειτουργίας του κυκλώματός μας χρησιμοποιούμε το παρακάτω testbench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tb_pipelined_adder is
end tb_pipelined_adder;

architecture tb of tb_pipelined_adder is
    component pipelined_adder
        Port ( clk  : in STD_LOGIC;
               a    : in STD_LOGIC_VECTOR (3 downto 0);
               b    : in STD_LOGIC_VECTOR (3 downto 0);
               cin  : in STD_LOGIC;
               sum  : out STD_LOGIC_VECTOR (3 downto 0);
               cout : out STD_LOGIC);
    end component;
```

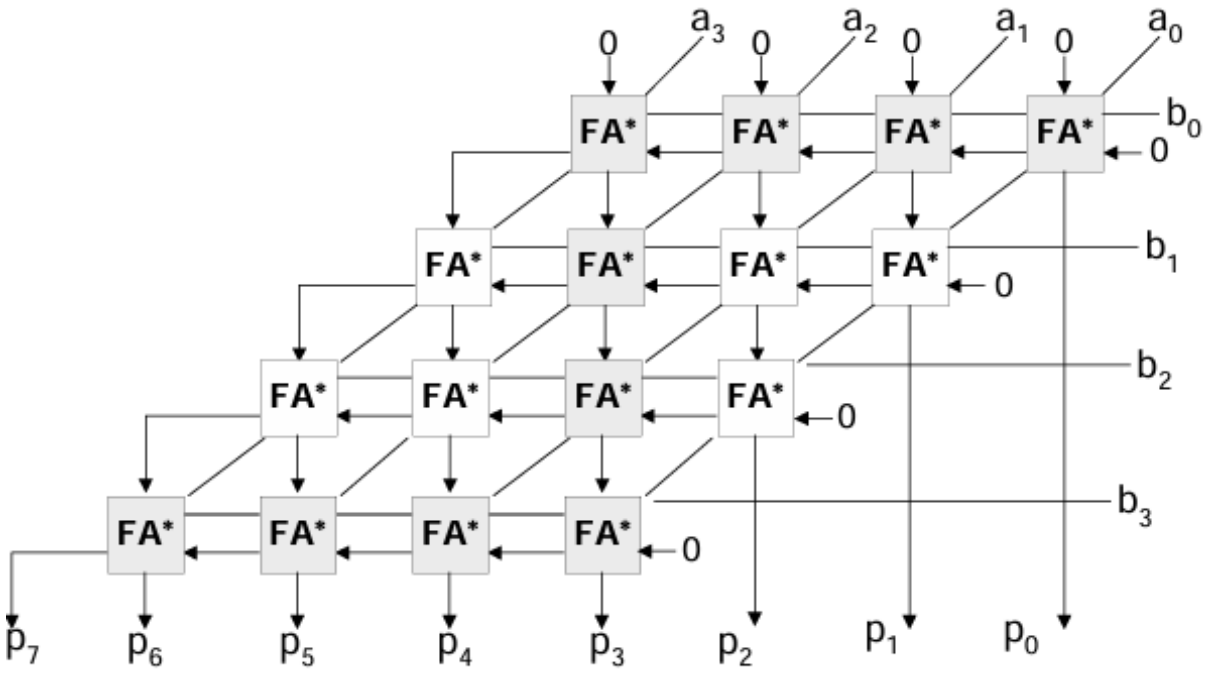
```
    -- Signals
    signal clk  : STD_LOGIC := '0';
    signal a    : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
    signal b    : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
    signal cin  : STD_LOGIC := '0';
    signal sum  : STD_LOGIC_VECTOR (3 downto 0);
    signal cout : STD_LOGIC;

    -- Clock period definition
    constant CLOCK_PERIOD : time := 10 ns;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: pipelined_adder port map (
        clk  => clk,
        a    => a,
        b    => b,
        cin  => cin,
        sum  => sum,
        cout => cout
    );

    -- Stimulus Process: Check all possible inputs
    STIMULUS: process
    begin
        -- Iterate through all possible values of a, b, cin (256 cases)
        for i in 0 to 15 loop
            for j in 0 to 15 loop
                for k in 0 to 1 loop
                    a   <= std_logic_vector(to_unsigned(i, 4));
                    b   <= std_logic_vector(to_unsigned(j, 4));
                    cin <= std_logic(to_unsigned(k, 1)(0));

                    wait for CLOCK_PERIOD; -- Wait for next clock cycle
                end loop;
            end loop;
        end loop;

        -- Stop simulation after all test cases
        wait;
    end process;

    GEN_CLK: process
    begin
        while true loop
            clk <= '0';
            wait for CLOCK_PERIOD / 2;
            clk <= '1';
            wait for CLOCK_PERIOD / 2;
        end loop;
    end process;

end tb;
```



Παρατηρούμε ότι το πρώτο αποτέλεσμα παράγεται στη 5η θετική ακμή του ρολογιού μετά από 5 κύκλους καθυστέρησης που έχουμε εισάγει για την επίτευξη του συγχρονισμού του pipeline. Μετά τον 5ο κύκλο αρχίζουν σε κάθε κύκλο του ρολογιού να παράγονται τα σωστά αποτελέσματα. Επομένως, μπορούμε να επαληθεύσουμε την ορθή λειτουργία του κυκλώματός μας.

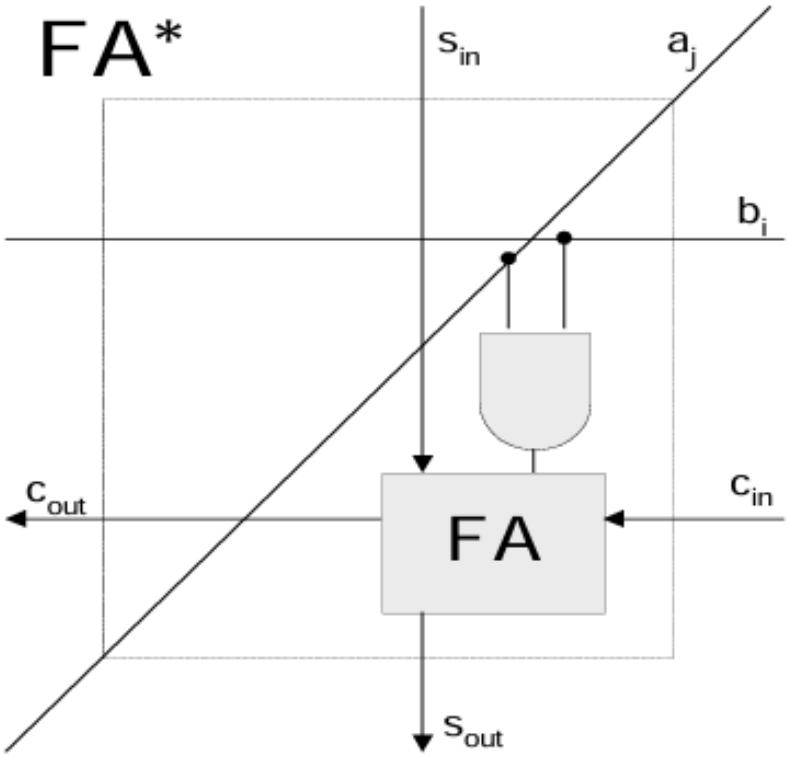
Θέμα 3: Συστολικός Πολλαπλασιαστής διάδοσης κρατουμένου των 4 bits

Η σχεδίαση του συστολικού πολλαπλασιαστή διάδοσης κρατουμένου των 4 bits βασίζεται στο παρακάτω κύκλωμα:

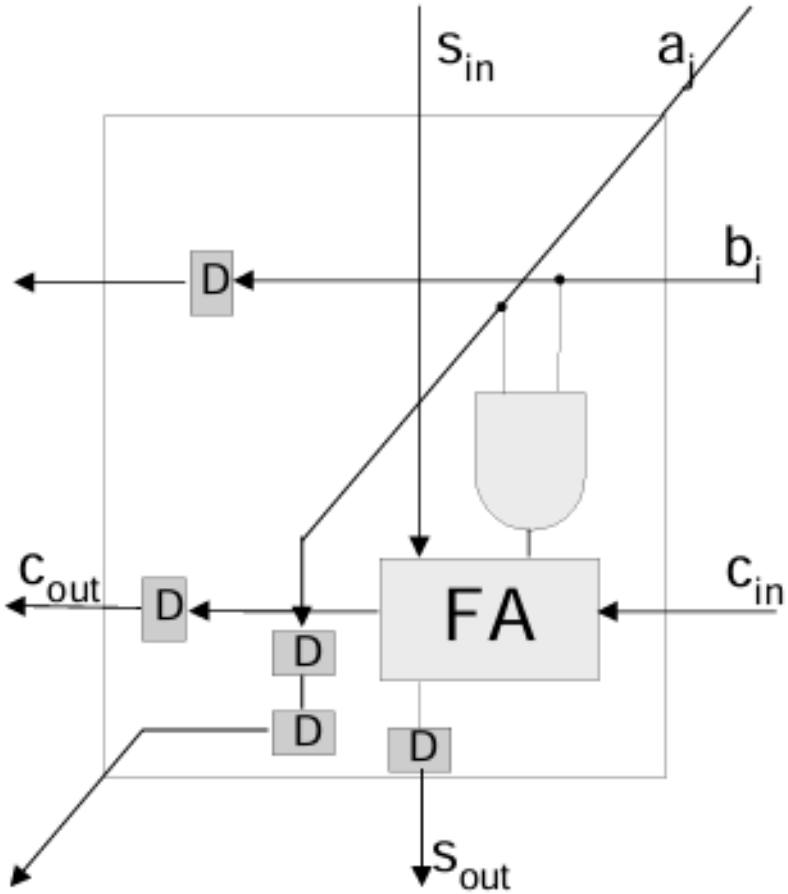


Σημειώνεται ότι για να είναι ένα κύκλωμα συστολικό θα πρέπει να πληρούνται οι εξής τρεις προϋποθέσεις:
• Τα κύτταρα που αποτελούν το κύκλωμα πρέπει να είναι πανομοιότυπα.
• Πρέπει να υποστηρίζεται η λογική της συνεχούς διοχέτευσης.
• Απαγορεύεται να υπάρχουν γραμμές σημάτων μεταξύ των κυττάρων που να μην διακόπτονται από στοιχεία καθυστέρησης (latches).

Αρχικά, πρέπει να σχεδιάσουμε την βασική δομική μονάδα του πολλαπλασιαστή, τον full adder *, η δομή του οποίου φαίνεται παρακάτω. Πρακτικά, το βασικό κύταρρο του πολλαπλασιαστή αποτελείται από έναν πλήρη αθροιστή και μία πύλη and και η λειτουργία του είναι να παράγει το ενδιάμεσο γινόμενο των 2 bit από την πύλη and και στην συνέχεια να το προσθέτει με το ενδιάμεσο γινόμενο και το κρατούμενο της προηγούμενης βαθμίδας.



Για να σχεδιάσουμε το ζητούμενο κύκλωμα, αρχικά πρέπει να σχεδιάσουμε το βασικό κύταρρο του πολλαπλασιαστή. Σεβόμενοι τις αρχές για να είναι το τελικό μας κύκλωμα συστολικό παρατηρούμε ότι πρέπει να προστεθούν ένας μανδαλωτής στις γραμμές s_{out}, c_{out} και b και 2 μανδαλωτές στην γραμμή a. Επιλέγουμε για ευκολία να βάλουμε τα στοιχεία καθυστέρησης στην έξοδο των γραμμών, μέσα στην βασική μονάδα του κυκλώματος. Προκύπτει, λοιπόν, το παρακάτω συστολικό κύταρρο.

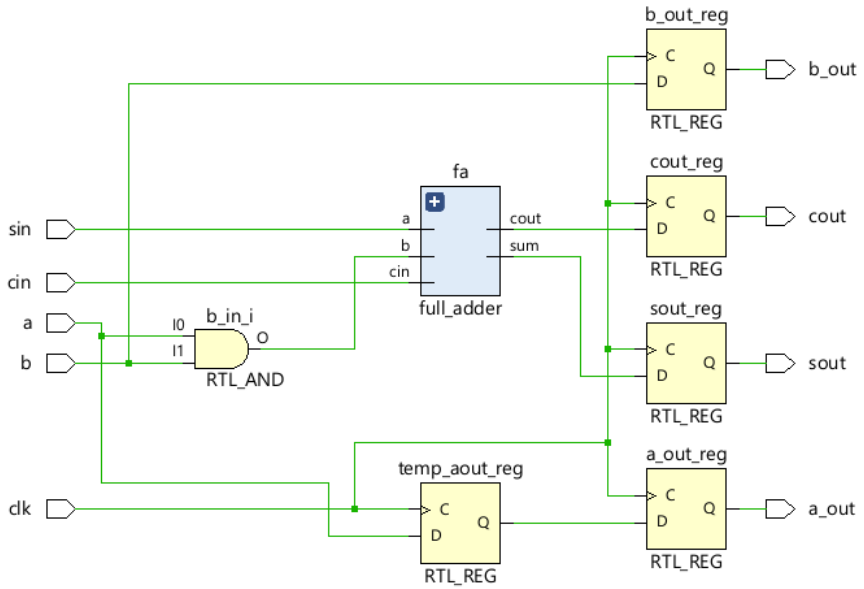


Ο κώδικας σε vhdl για την περιγραφή του Full Adder * και της αρχιτεκτονικής του σε structural είναι ο εξής:

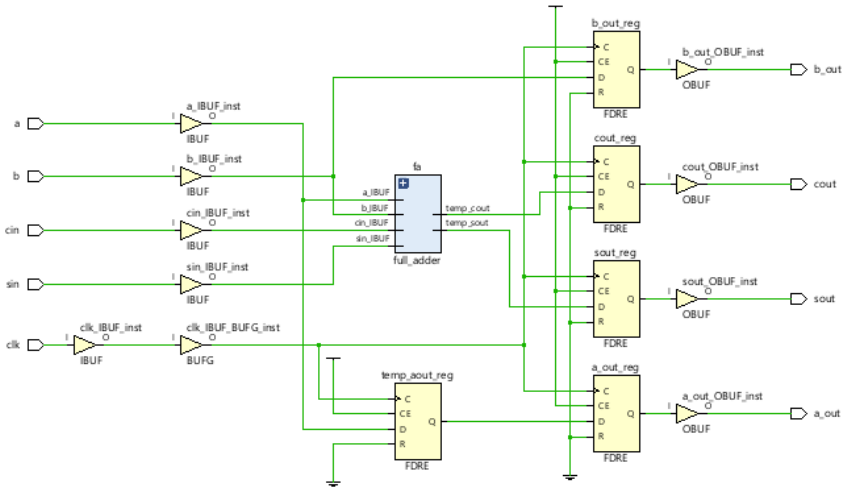
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity full_adder_star is
    Port (
            clk : in STD_LOGIC;
            a : in STD_LOGIC;
            b: in STD_LOGIC;
            cin : in STD_LOGIC;
            sin : in STD_LOGIC;
            sout : out STD_LOGIC;
            cout : out STD_LOGIC;
            a_out : out STD_LOGIC;
            b_out : out STD_LOGIC
        );

end full_adder_star;
```

```
architecture structural of full_adder_star is

    -- Full Adder Component
    component full_adder 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 b_in : std_logic;
        signal temp_sout, temp_cout, temp_aout : std_logic;


    begin
        b_in <= a and b;

        fa: full_adder port map (
        a    => sin,
        b    => b_in,
        cin  => cin,
        sum  => temp_sout,
        cout => temp_cout
        );
        process (clk)
            begin
                if rising_edge(clk)  then
                    sout <= temp_sout;
                    cout <= temp_cout;
                    b_out <= b;
                    a_out <= temp_aout;
                    temp_aout <= a;
                end if;

        end process;
end structural;
```

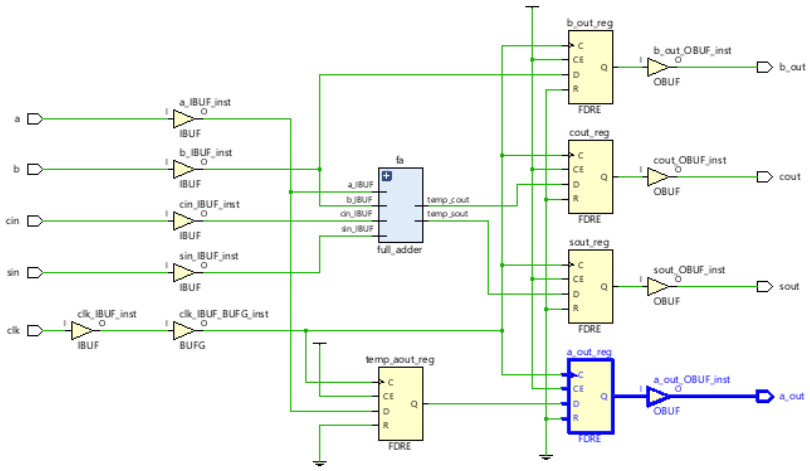
Με χρήση του elaborate design έχουμε το παρακάτω σχηματικό:



Με χρήση του synthesis έχουμε το παρακάτω σχηματικό:



Χρησιμοποιούμε το timing summary για να βρούμε το κρίσιμο μονοπάτι και την αντίστοιχη καθυστέρηση για το κύκλωμα:



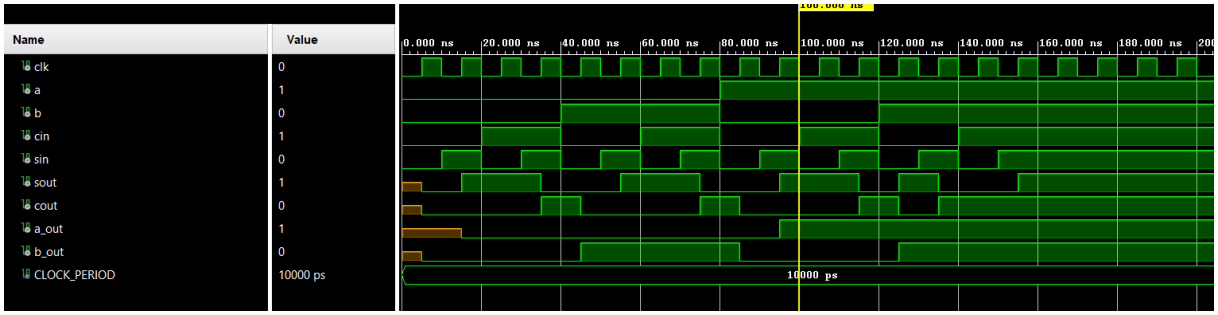
Name	Slack	Levels	Routes	High Fanout	From	To	Total ...	Logic Delay	Net Delay	Requirement	Source Clock	Destin
Path 1	∞	2	2	1	a_out_reg/C	a_out	4.076	3.276	0.800	∞		
Path 2	∞	2	2	1	b_out_reg/C	b_out	4.076	3.276	0.800	∞		
Path 3	∞	2	2	1	cout_reg/C	cout	4.076	3.276	0.800	∞		
Path 4	∞	2	2	1	sout_reg/C	sout	4.076	3.276	0.800	∞		
Path 5	∞	2	3	3	b	sout_reg/D	1.934	1.134	0.800	∞	input port clock	
Path 6	∞	2	3	2	sin	cout_reg/D	1.906	1.106	0.800	∞	input port clock	
Path 7	∞	1	2	3	b	b_out_reg/D	1.782	0.982	0.800	∞	input port clock	

Όπως έχουμε παρατηρήσει και στα προηγούμενα κυκλώματα, όταν αναφερόμαστε σε χρονική καθυστέρηση του critical path με τεχνική pipelining η καθυστέρηση αφορά καθυστέρηση από register σε register, δηλαδή καθυστέρηση σε στάδιο του pipeline. Η καθυστέρηση που υπολογίσαμε θα χρησιμοποιηθεί παρακάτω.
Τέλος, για να είμαστε σίγουροι ότι το κύκλωμα μας λειτουργεί σωστά πριν σχεδιάσουμε τον πολλαπλασιαστή δημιουργούμε το παρακάτω testbench:

```
    library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;


entity full_adder_star_tb is
end entity;

architecture tb of full_adder_star_tb is
    component full_adder_star is
    Port (
            clk : in STD_LOGIC;
            a : in STD_LOGIC;
            b: in STD_LOGIC;
            cin : in STD_LOGIC;
            sin : in STD_LOGIC;
            sout : out STD_LOGIC;
            cout : out STD_LOGIC;
            a_out : out STD_LOGIC;
            b_out : out STD_LOGIC
        );
    end component;
```

```
    signal clk, a, b, cin, sin : std_logic;
    signal sout, cout, a_out, b_out : std_logic;

    constant CLOCK_PERIOD : time := 10 ns;


begin
    uut: full_adder_star port map(
            clk => clk,
            a => a,
            b => b,
            cin => cin,
            sin => sin,
            sout => sout,
            cout => cout,
            a_out => a_out,
            b_out =>b_out
    );



    -- Stimulus Process: Check all possible inputs
    STIMULUS: process
    begin
        for i in 0 to 1 loop
            for j in 0 to 1 loop
                for k in 0 to 1 loop
                    for l in 0 to 1 loop
                        a   <= std_logic(to_unsigned(i, 1)(0));
                        b   <= std_logic(to_unsigned(j, 1)(0));
                        cin <= std_logic(to_unsigned(k, 1)(0));
                        sin <= std_logic(to_unsigned(l, 1)(0));

                    wait for CLOCK_PERIOD; -- Wait for next clock cycle
                    end loop;
                end loop;
            end loop;
        end loop;

        -- Stop simulation after all test cases
        wait;
    end process;

    GEN_CLK: process
    begin
        while true loop
            clk <= '0';
            wait for CLOCK_PERIOD / 2;
            clk <= '1';
            wait for CLOCK_PERIOD / 2;
        end loop;
    end process;
```

```
end tb;
```



Μπορούμε να παρατηρήσουμε, λοιπόν, ότι για όλες τις εισόδους έχουμε σωστή έξοδο και μπορεί το κύκλωμα μας να χρησιμοποιηθεί στον σχεδιασμό του πολλαπλασιαστή. Επιπλέον, μπορούμε να παρατηρήσουμε ότι τα αποτελέσματα παράγονται μετά από κάποια αρχική καθυστέρηση, η οποία θα σχολιαστεί παρακάτω.
Ο κώδικας σε vhdl για την περιγραφή του πολλαπλασιαστή και η structural αρχιτεκτονική του φαίνονται παρακάτω:

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

entity systolic_multiplier_4bits is
    Port (
            clk : in std_logic;
            a : in std_logic_vector(4-1 downto 0);
            b: in std_logic_vector(4-1 downto 0);
            p : out std_logic_vector(8-1 downto 0)
        );

end systolic_multiplier_4bits;

architecture structural of systolic_multiplier_4bits is

    -- Full Adder Star Component
    component full_adder_star is
    Port (
            clk : in STD_LOGIC;
            a : in STD_LOGIC;
            b: in STD_LOGIC;
            cin : in STD_LOGIC;
            sin : in STD_LOGIC;
            sout : out STD_LOGIC;
            cout : out STD_LOGIC;
            a_out : out STD_LOGIC;
            b_out : out STD_LOGIC
        );
    end component;

     signal p5, p4, p44, p3, p33, p333, p2, p22, p222, p2222, p22222 : std_logic;
    signal p1, p11, p111, p1111, p11111, p1_6, p1_7, p0, p00, p000, p0000, p00000, p0_6, p0_7, p0_8, p0
    signal a1,  a2, a22, a3, a33, a333 : std_logic;
```

```
signal b1, b11, b2, b22, b222, b2222, b3, b33, b333, b3333, b33333, b3_6 : std_logic;

-- signals for every level

signal a_temp_0, a_temp_1, a_temp_2, a_temp_3 : std_logic_vector(4-1 downto 0);
signal b_temp_0, b_temp_1, b_temp_2, b_temp_3 : std_logic_vector(4-1 downto 0);
signal s_temp_0, s_temp_1, s_temp_2, s_temp_3 : std_logic_vector(4-1 downto 0);
signal c_temp_0, c_temp_1, c_temp_2, c_temp_3 : std_logic_vector(4-1 downto 0);
signal carry0_3, carry1_3, carry2_3 : std_logic;

begin

fas1: full_adder_star port map(
    clk => clk,
    a => a(0),
    b => b(0),
    cin => '0',
    sin => '0',
    sout => s_temp_0(0),
    cout => c_temp_0(0),
    a_out => a_temp_0(0),
    b_out => b_temp_0(0)
    );

    fas2: full_adder_star port map(
    clk => clk,
    a => a1,
    b => b_temp_0(0),
    cin => c_temp_0(0),
    sin => '0',
    sout => s_temp_0(1),
    cout => c_temp_0(1),
    a_out => a_temp_0(1),
    b_out => b_temp_0(1)
    );

    fas3_1: full_adder_star port map(
    clk => clk,
    a => a2,
    b => b_temp_0(1),
    cin => c_temp_0(1),
    sin => '0',
    sout => s_temp_0(2),
    cout => c_temp_0(2),
    a_out => a_temp_0(2),
    b_out => b_temp_0(2)
    );

    fas3_2: full_adder_star port map(
    clk => clk,
    a => a_temp_0(0),
    b => b1,
    cin => '0',
    sin => s_temp_0(1),
```

```
sout => s_temp_1(0),
cout => c_temp_1(0),
a_out => a_temp_1(0),
b_out => b_temp_1(0)
);

fas4_1: full_adder_star port map(
clk => clk,
a => a3,
b => b_temp_0(2),
cin => c_temp_0(2),
sin => '0',
sout => s_temp_0(3),
cout => c_temp_0(3),
a_out => a_temp_0(3),
b_out => b_temp_0(3)
);


fas4_2: full_adder_star port map(
clk => clk,
a => a_temp_0(1),
b => b_temp_1(0),
cin => c_temp_1(0),
sin => s_temp_0(2),
sout => s_temp_1(1),
cout => c_temp_1(1),
a_out => a_temp_1(1),
b_out => b_temp_1(1)
);

fas5_1: full_adder_star port map(
clk => clk,
a => a_temp_0(2),
b => b_temp_1(1),
cin => c_temp_1(1),
sin => s_temp_0(3),
sout => s_temp_1(2),
cout => c_temp_1(2),
a_out => a_temp_1(2),
b_out => b_temp_1(2)
);

fas5_2: full_adder_star port map(
clk => clk,
a => a_temp_1(0),
b => b2,
cin => '0',
sin => s_temp_1(1),
sout => s_temp_2(0),
cout => c_temp_2(0),
a_out => a_temp_2(0),
b_out => b_temp_2(0)
);
```

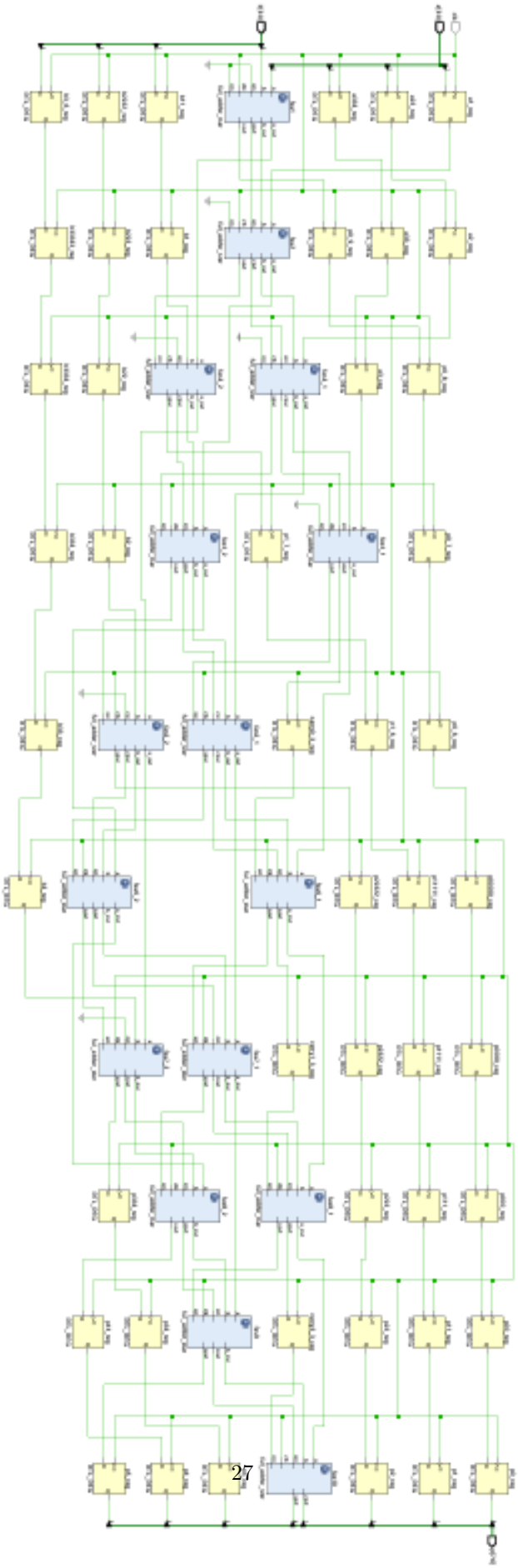
```
fas6_1: full_adder_star port map(
clk => clk,
a => a_temp_0(3),
b => b_temp_1(2),
cin => c_temp_1(2),
sin => carry0_3,
sout => s_temp_1(3),
cout => c_temp_1(3),
a_out => a_temp_1(3),
b_out => b_temp_1(3)
);



fas6_2: full_adder_star port map(
clk => clk,
a => a_temp_1(1),
b => b_temp_2(0),
cin => c_temp_2(0),
sin => s_temp_1(2),
sout => s_temp_2(1),
cout => c_temp_2(1),
a_out => a_temp_2(1),
b_out => b_temp_2(1)
);

fas7_1: full_adder_star port map(
clk => clk,
a => a_temp_1(2),
b => b_temp_2(1),
cin => c_temp_2(1),
sin => s_temp_1(3),
sout => s_temp_2(2),
cout => c_temp_2(2),
a_out => a_temp_2(2),
b_out => b_temp_2(2)
);

fas7_2: full_adder_star port map(
clk => clk,
a => a_temp_2(0),
b => b3,
cin => '0',
sin => s_temp_2(1),
sout => s_temp_3(0),
cout => c_temp_3(0),
a_out => a_temp_3(0),
b_out => b_temp_3(0)
);

fas8_1: full_adder_star port map(
clk => clk,
a => a_temp_1(3),
```

```
    b => b_temp_2(2),
    cin => c_temp_2(2),
    sin => carry1_3,
    sout => s_temp_2(3),
    cout => c_temp_2(3),
    a_out => a_temp_2(3),
    b_out => b_temp_2(3)
    );



    fas8_2: full_adder_star port map(
    clk => clk,
    a => a_temp_2(1),
    b => b_temp_3(0),
    cin => c_temp_3(0),
    sin => s_temp_2(2),
    sout => s_temp_3(1),
    cout => c_temp_3(1),
    a_out => a_temp_3(1),
    b_out => b_temp_3(1)
    );

    fas9: full_adder_star port map(
    clk => clk,
    a => a_temp_2(2),
    b => b_temp_3(1),
    cin => c_temp_3(1),
    sin => s_temp_2(3),
    sout => s_temp_3(2),
    cout => c_temp_3(2),
    a_out => a_temp_3(2),
    b_out => b_temp_3(2)
    );

    fas10: full_adder_star port map(
    clk => clk,
    a => a_temp_2(3),
    b => b_temp_3(2),
    cin => c_temp_3(2),
    sin => carry2_3,
    sout => s_temp_3(3),
    cout => c_temp_3(3),
    a_out => a_temp_3(3),
    b_out => b_temp_3(3)
    );

process (clk)
begin
    if rising_edge(clk) then

    p5 <= s_temp_3(2);
    p4 <= p44;
    p44 <= s_temp_3(1);
```

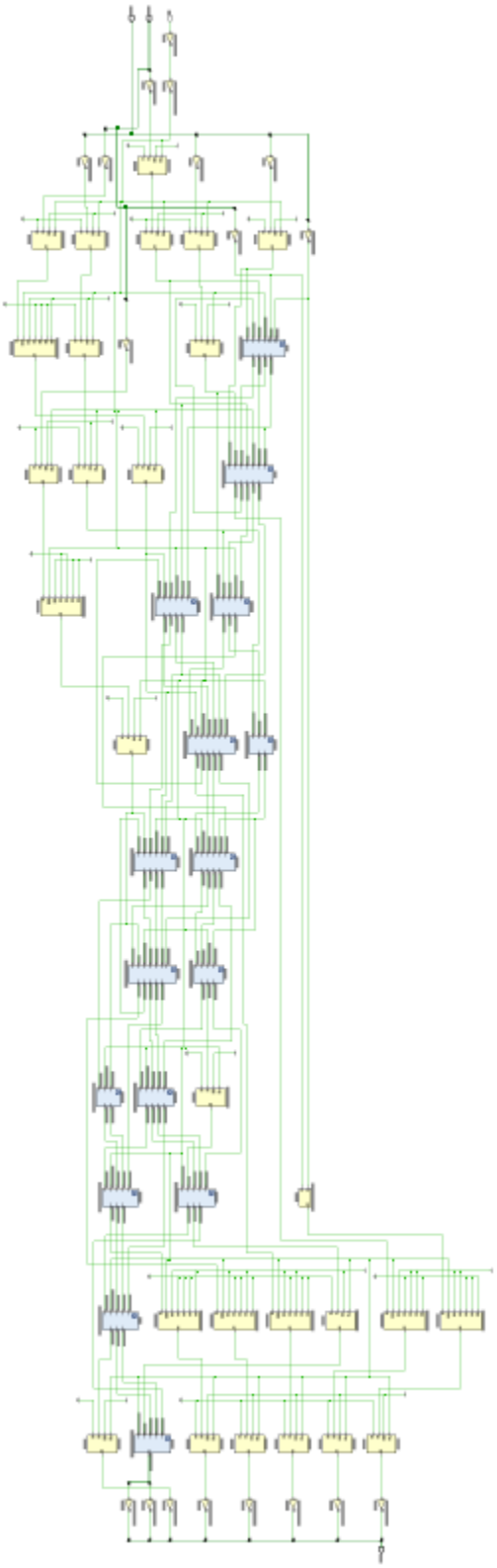
```
p3 <= p33;
p33 <= p333;
p333 <= s_temp_3(0);
p2 <= p22;
p22 <= p222;
p222 <= p2222;
p2222 <= p22222;
p22222 <= s_temp_2(0);
p1 <= p11;
p11 <= p111;
p111 <= p1111;
p1111 <= p11111;
p11111 <= p1_6;
p1_6 <= p1_7;
p1_7 <= s_temp_1(0);
p0 <= p00;
p00 <= p000;
p000 <= p0000;
p0000 <= p00000;
p00000 <= p0_6;
p0_6 <= p0_7;
p0_7 <= p0_8;
p0_8 <= p0_9;
p0_9 <= s_temp_0(0);

a3 <= a33;
a33 <= a333;
a333 <= a(3);
a2 <= a22;
a22 <= a(2);
a1 <= a(1);

b3 <= b33;
b33 <= b333;
b333 <= b3333;
b3333 <= b33333;
b33333 <= b3_6;
b3_6 <= b(3);
b2 <= b22;
b22 <= b222;
b222 <= b2222;
b2222 <= b(2);
b1 <= b11;
b11 <= b(1);

carry0_3 <= c_temp_0(3);
carry1_3 <= c_temp_1(3);
carry2_3 <= c_temp_2(3);


p <= c_temp_3(3) & s_temp_3(3) & p5 & p4 & p3 & p2 & p1 & p0;
```

```
        end if;
    end process;
end structural;
```

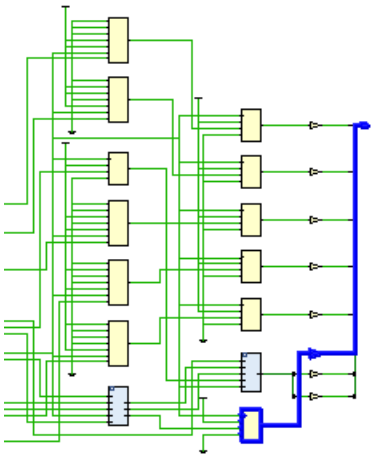
Ο παραπάνω πολλαπλασιαστής αποτελείται από 15 full adders * χωρισμένους σε 10 επίπεδα. Επιπλέον, έχουν σχεδιαστεί οι απαραίτητοι registers για τις καθυστερήσεις που χρειάζονται στην είσοδο και στην έξοδο του πολλαπλασιαστή. Όπως έχουμε αναφέρει και παραπάνω, οι ενδιάμεσες καθυστερήσεις ανάμεσα στους full adders * σχεδιάζονται μέσα στα βασικά κύτταρα full adders *.
Με χρήση του elaborate design έχουμε:



Με χρήση του synthesis έχουμε:



Για να βρούμε το critical path του συνολικού μας κυκλώματος και την αντίστοιχη χρονική καθυστέρηση με χρήση του timing summary έχουμε:

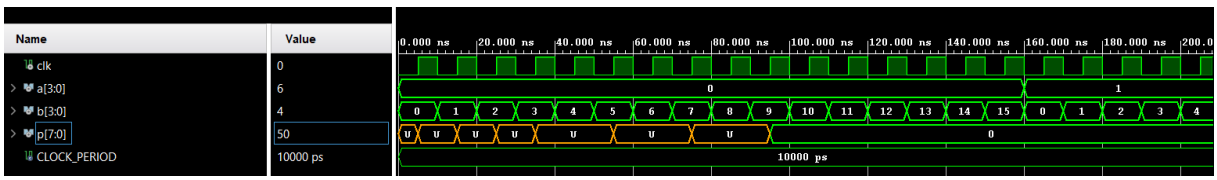


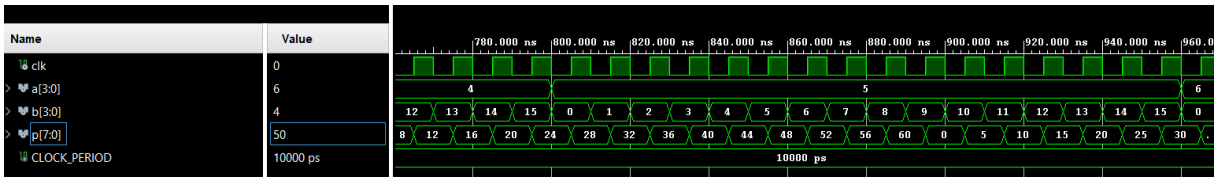
Name	Slack	Levels	Routes	High Fanout	From	To	Total ...	Logic Delay	Net Delay	Requirement	Source Clock	Desti
Path 1	∞	2	2	1	p5_reg/C	p[5]	4.076	3.276	0.800	∞		
Path 2	∞	2	2	1	fas10/sout_reg/C	p[6]	4.076	3.276	0.800	∞		
Path 3	∞	2	2	1	fas10/cout_reg/C	p[7]	4.076	3.276	0.800	∞		
Path 4	∞	2	2	1	p0_reg/C	p[0]	4.058	3.258	0.800	∞		
Path 5	∞	2	2	1	p1_reg/C	p[1]	4.058	3.258	0.800	∞		
Path 6	∞	2	2	1	p2_reg/C	p[2]	4.058	3.258	0.800	∞		
Path 7	∞	2	2	1	p3_reg/C	p[3]	4.058	3.258	0.800	∞		

Για να ελέγξουμε την ορθή λειτουργία του πολλαπλασιαστή χρησιμοποιούμε το παρακάτω testbench, το οποίο δοκιμάζει όλους τους πιθανούς συνδυασμούς δύο αριθμών 4 bits.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;


entity systolic_multiplier_4bits_tb is
end entity;

architecture tb of systolic_multiplier_4bits_tb is
    component systolic_multiplier_4bits is
    Port (
            clk : in std_logic;
            a : in std_logic_vector(4-1 downto 0);
            b: in std_logic_vector(4-1 downto 0);
            p : out std_logic_vector(8-1 downto 0)
        );
    end component;

    signal clk : std_logic;
```

```
    signal a, b : std_logic_vector(4-1 downto 0);
    signal p : std_logic_vector(8-1 downto 0);

    constant CLOCK_PERIOD : time := 10 ns;


begin
    uut: systolic_multiplier_4bits port map(
            clk => clk,
            a => a,
            b => b,
            p => p
    );



    -- Stimulus Process: Check all possible inputs
    STIMULUS: process
    begin
        for i in 0 to 15 loop
            for j in 0 to 15 loop
                a   <= std_logic_vector(to_unsigned(i, 4));
                b   <= std_logic_vector(to_unsigned(j, 4));

                    wait for CLOCK_PERIOD; -- Wait for next clock cycle
            end loop;
        end loop;

        -- Stop simulation after all test cases
        wait;
    end process;

    GEN_CLK: process
    begin
        while true loop
            clk <= '0';
            wait for CLOCK_PERIOD / 2;
            clk <= '1';
            wait for CLOCK_PERIOD / 2;
        end loop;
    end process;

end tb;
```





Στις παραπάνω φωτογραφίες δεν μπορούν να φανούν όλα τα αποτελέσματα, αλλά από την προσομοίωση όλων των συνδυασμών μπορούμε να παρατηρήσουμε την ορθή λειτουργία του κυκλώματος. Στην διαδικασία της προσομοίωσης μπορούμε να παρατηρήσουμε και την αρχική καθυστέρηση, μετά από την οποία αρχίζουμε να έχουμε αποτελέσματα για κάθε κύκλο. Σχολιάζεται παρακάτω η λειτουργικότητα της.

Σχόλια - Παρατηρήσεις

- Στο αρχικό κύκλωμα του πολλαπλασιαστή, αν δεν χρησιμοποιηθεί κάποια τεχνική pipeline η καθυστέρηση για την παραγωγή ενός αποτελέσματος είναι συνολικά 10 φορές η καθυστέρηση ενός full adder *. Με την συστολικότητα του κυκλώματος επιτυγχάνουμε, μετά από τους 10 πρώτους κύκλους, οι οποίοι είναι η καθυστέρηση που έχουμε μέχρι την παραγωγή του πρώτου αποτελέσματος, να έχουμε συνεχόμενα αποτελέσματα κάθε κύκλο. Με αυτόν τον τρόπο ουσιαστικά χάνουμε κάποιους κύκλους στην αρχή με σκοπό όμως να γλιτώσουμε σημαντικά σε συνολικό χρόνο.

- Αν ένα κύκλωμα που είναι συστολικό με κάποια στάδια θέλουμε να το κάνουμε συστολικό με πιο πολλά στάδια, αυτό που χάνουμε είναι επιπλέον κύκλοι καθυστέρησης στην αρχή. Επειδή, όμως, όπως έχουμε αναφέρει και παραπάνω το κρίσιμο μονοπάτι αποτελεί διαδρομή από register σε register, αν κάνουμε το κύκλωμα "πιο συστολικό" το κρίσιμο μοοπάτι έχει λιγότερη χρονική καθυστέρηση και έτσι θα μπορούμε να αυξήσουμε την συχνότητα του ρολογιού, αφού ως γνωστόν αυτή εξαρτάται άμεσα από την χρονική καθυστέρηση του critical path.

- Για τις καθυστερήσεις των παραπάνω κυκλωμάτων μπορούμε να παρατηρήσουμε ότι ο full adder star δίνει το πρώτο αποτέλεσμα στην 1η θετική ακμή του ρολογιού (και το πρώτο a δίνεται στην 2η) και ο systolic multiplier στην 10η θετική ακμή του ρολογιού.

- Ο συνολικός χρόνος για παραγωγή n αποτελεσμάτων ενός συστολικού κυκλώματος είναι:

$$T = T_{latency} + nT_{stage}$$

όπου $T_{latency}$ η αρχική καθυστέρηση μέχρι το 1ο αποτέλεσμα, n ο αριθμός των αποτελεσμάτων και T_{stage} η καθυστέρηση του σταδίου του συστολικού κυκλώματος. Θα μπορούσε να ονομαστεί και χρόνος ρολογιού, αφού ουσιαστικά η καθυστέρηση αυτή καθορίζει την συχνότητα ρολογιού.