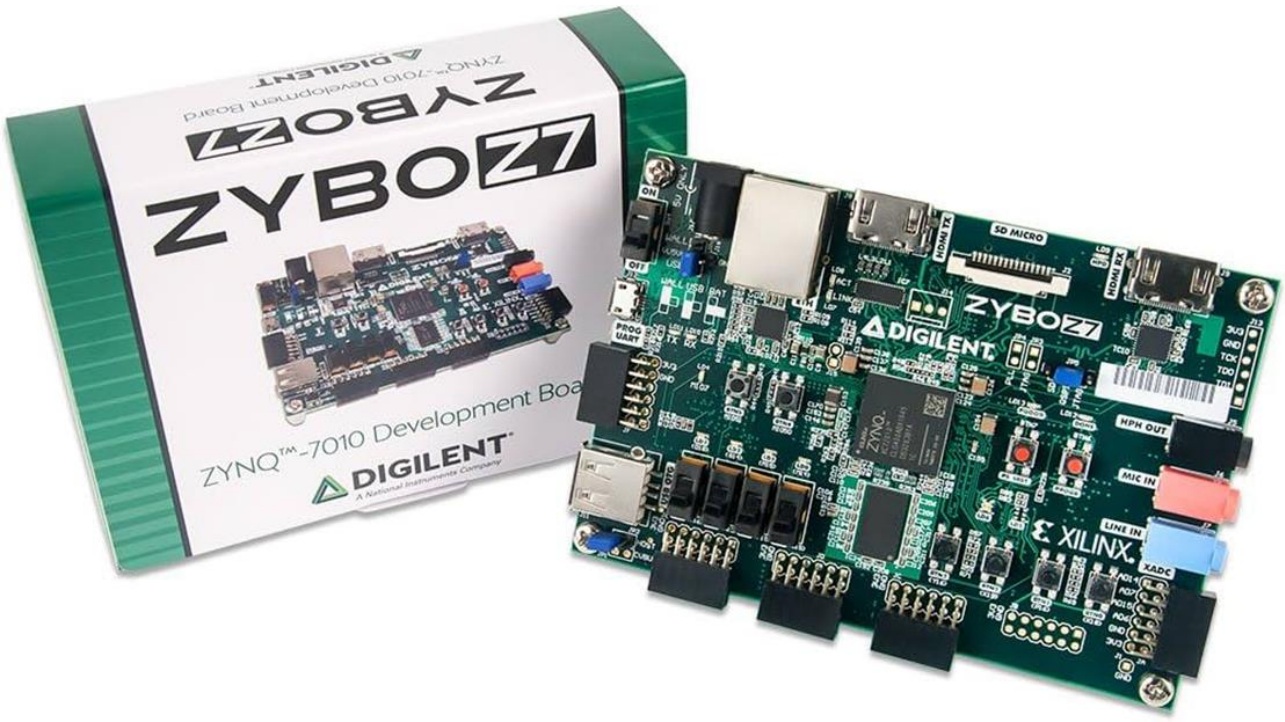
Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών



Μάθημα: Ψηφιακά Συστήματα VLSI

Θέμα: 3η Εργαστηριακή Άσκηση – Σχεδίαση Μονάδων Υλικού με την Τεχνική Pipelining

Εξάμηνο: 8ο

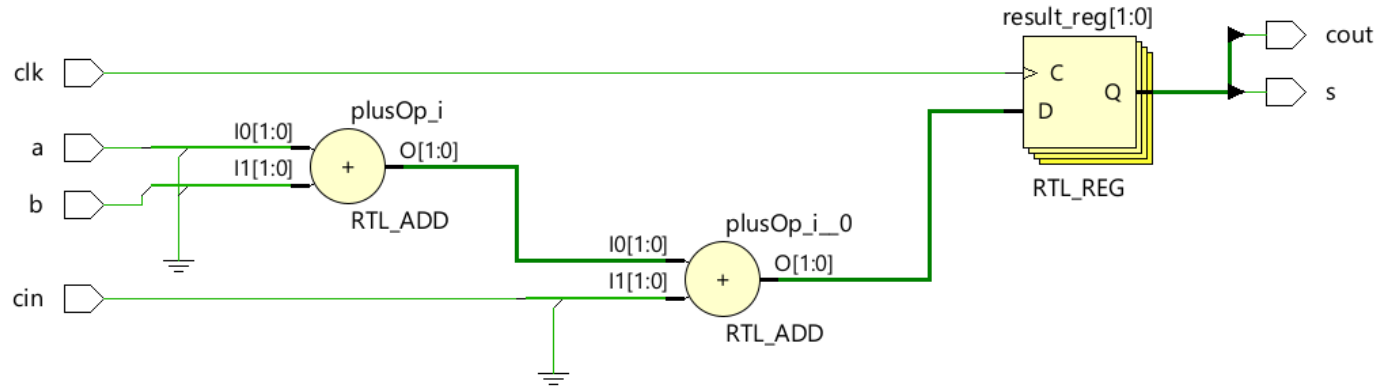
Ομάδα: 6

Συνεργάτες:
- Ακύλας Αντώνιος 03121152
- Κουμπιάς Ιωάννης-Χρυσοβαλάντης 03121053

Άσκηση 1

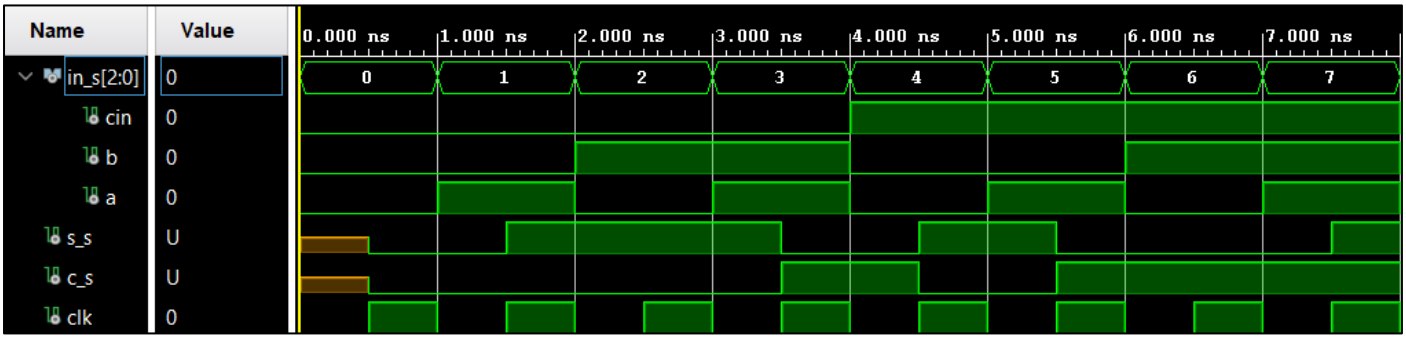
Στο πρώτο κομμάτι του εργαστηρίου σχεδιάσαμε έναν Σύγχρονο Πλήρη Αθροιστή (Full Adder - FA) σε περιγραφή συμπεριφοράς (Behavioral). Η κύρια διαφορά με τον πλήρη αθροιστή που σχεδιάσαμε στο προηγούμενο εργαστήριο είναι η εξαγωγή του αποτελέσματος με τη θετική ακμή του ρολογιού. Συνεπώς προστίθενται registers στην έξοδο για την αποθήκευση του αθροίσματος και του κρατουμένου. Παρουσιάζουμε τον κώδικα VHDL καθώς και το RTL Schematic του κυκλώματος που προκύπτει:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity fa_sync is
    port(
        a: in std_logic;
        b: in std_logic;
        cin: in std_logic;
        clk: in std_logic;
        s: out std_logic;
        cout: out std_logic
    );
end fa_sync;

architecture behavioral of fa_sync is
    signal result: std_logic_vector(1 downto 0);
begin
    sync_add: process(clk)
    begin
        if clk'event and clk='1' then
            result <= ('0' & a) + ('0' & b) + ('0' & cin);
        end if;
    end process;
    s <= result(0);
    cout <= result(1);
end behavioral;
```

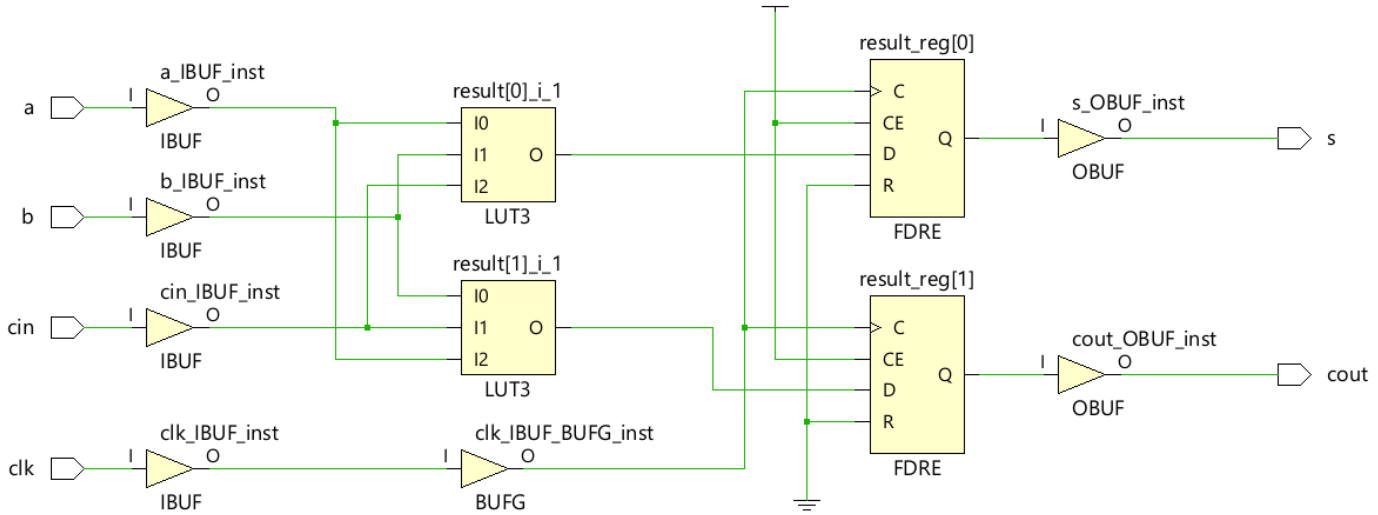


Στην συνέχεια δημιουργήσαμε κατάλληλο testbench που παράγει όλα τα διαφορετικά inputs για τον έλεγχο της ορθής λειτουργίας του κυκλώματος. Ακολουθούν ο κώδικας VHDL του testbench καθώς και τα αποτελέσματα του simulation:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity fa_sync_tb is
end fa_sync_tb;

architecture fa_sync_tb_arch of fa_sync_tb is
    component fa_sync
        port(
            a: in std_logic;
            b: in std_logic;
            cin: in std_logic;
            clk: in std_logic;
            s: out std_logic;
            cout: out std_logic
        );
    end component;

    constant CLKP : time := 1ns;

    signal in_s: std_logic_vector(2 downto 0);
    signal s_s,c_s,clk: std_logic;
begin
    UUT: fa_sync port map (in_s(0),in_s(1),in_s(2),clk,s_s,c_s);

    clk_proc: process
    begin
        clk <= '0';
        wait for CLKP/2;
        clk <= '1';
        wait for CLKP/2;
    end process;

    testSequence: process
    begin
        in_s <= "000";
        for i in 0 to 7 loop
            wait for CLKP;
            in_s <= in_s + 1;
        end loop;
        wait;
    end process;
end fa_sync_tb_arch;
```



Έχοντας βεβαιωθεί πως το κύκλωμα λειτουργεί όπως είναι αναμενόμενο, προχωράμε στη διαδικασία σύνθεσης με στόχο τον υπολογισμό του critical path και κατ' επέκταση της μέγιστης χρονικής καθυστέρησης με την βοήθεια του Vivado. Παραθέτουμε το schematic της σύνθεσης αξιοποιώντας πλέον πόρους του FPGA και τις μετρήσεις των χρονικών καθυστερήσεων ανά μονοπάτι:



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	∞	2	2	1	result_reg[1]/C	cout	4.076	3.276	0.800
Path 2	∞	2	2	1	result_reg[0]/C	s	4.076	3.276	0.800
Path 3	∞	2	3	2	cin	result_reg[1]/D	1.932	1.132	0.800
Path 4	∞	2	3	2	cin	result_reg[0]/D	1.906	1.106	0.800

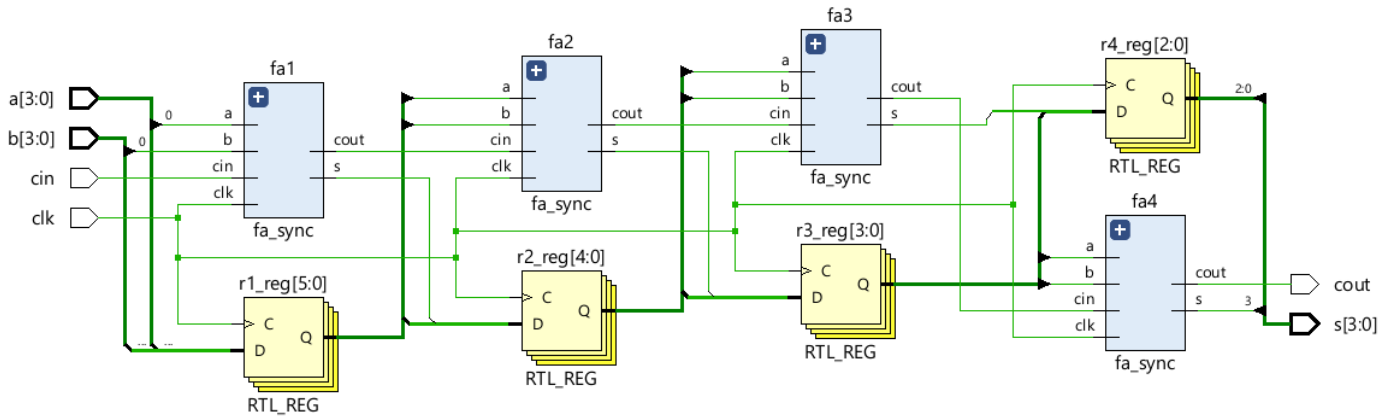
Το critical path της σύνθεσης φάινεται πρώτο στα Unconstrained Paths του timing report. Στις μετρήσεις συνυπολογίζονται τόσο η καθυστέρηση λόγω λογικής (Logic Delay) όσο και η καθυστέρηση λόγω του routing των συνδέσεων στο FPGA (Net Delay). Παρατηρούμε πως τα κρίσιμα μονοπάτια είναι από τον καταχωρητή result_reg[1] προς το output cout και από τον καταχωρητή result_reg[2] προς το output s. Ακολουθούν τα μονοπάτια cin – result_reg[1] και cin – result_reg[0] με μικρότερη καθυστέρηση που οφείλεται στη λογική.

Άσκηση 2

Στο δεύτερο κομμάτι του εργαστηρίου σχεδιάσαμε έναν Σύγχρονο Αθροιστή διάδοσης κρατουμένου των 4 bits με χρήση της τεχνικής pipeline. Η διαφορά με τον Αθροιστή της προηγούμενης εργαστηριακής άσκησης είναι πως οι αθροιστές δεν εξάγουν τα αποτελέσματά τους κατευθείαν στην έξοδο, αλλά σε καταχωρητές, οι οποίοι τα διαδίδουν στο pipeline έως ότου να πρέπει να γίνει η εξαγωγή στην έξοδο την κατάλληλη χρονική στιγμή. Έτσι όταν ένας πλήρης αθροιστής ως μονάδα τελείωσει με έναν υπολογισμό και διαδόσει το κρατούμενο στον επόμενο, μπορεί να στείλει το άθροισμα σε καταχωρητή και να ξεκινήσει στον αμέσως επόμενο κύκλο ρολογιού έναν νέο υπολογισμό. Παρουσιάζουμε τον κώδικα VHDL καθώς και το RTL Schematic του κυκλώματος που προκύπτει:

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

entity rca4_pipe is
    port(
        a: in std_logic_vector(3 downto 0);
        b: in std_logic_vector(3 downto 0);
        cin: in std_logic;
        clk: in std_logic;
        s: out std_logic_vector(3 downto 0);
        cout: out std_logic
    );
end rca4_pipe;
```

```
architecture rca4_arch of rca4_pipe is
    component fa_sync is
        port(
            a: in std_logic;
            b: in std_logic;
            cin: in std_logic;
            clk: in std_logic;
            s: out std_logic;
            cout: out std_logic
        );
    end component;
    signal r1: std_logic_vector(5 downto 0) := (others => '0');
    signal r2: std_logic_vector(4 downto 0) := (others => '0');
    signal r3: std_logic_vector(3 downto 0) := (others => '0');
    signal r4: std_logic_vector(2 downto 0) := (others => '0');
    signal s_t: std_logic_vector(2 downto 0);
    signal c_t: std_logic_vector(2 downto 0);
begin
    pipeline: process(clk)
    begin
        if clk'event and clk='1' then
            r4 <= s_t(2) & r3(1 downto 0);
            r3 <= r2(4 downto 3) & s_t(1) & r2(0);
            r2 <= r1(5 downto 2) & s_t(0);
            r1 <= b(3) & a(3) & b(2) & a(2) & b(1) & a(1);
        end if;
    end process;
    fa1: fa_sync port map(a(0),b(0),cin,clk,s_t(0),c_t(0));
    fa2: fa_sync port map(r1(0),r1(1),c_t(0),clk,s_t(1),c_t(1));
    fa3: fa_sync port map(r2(1),r2(2),c_t(1),clk,s_t(2),c_t(2));
    fa4: fa_sync port map(r3(2),r3(3),c_t(2),clk,s(3),cout);
    s(2 downto 0) <= r4(2 downto 0);
end rca4_arch;
```



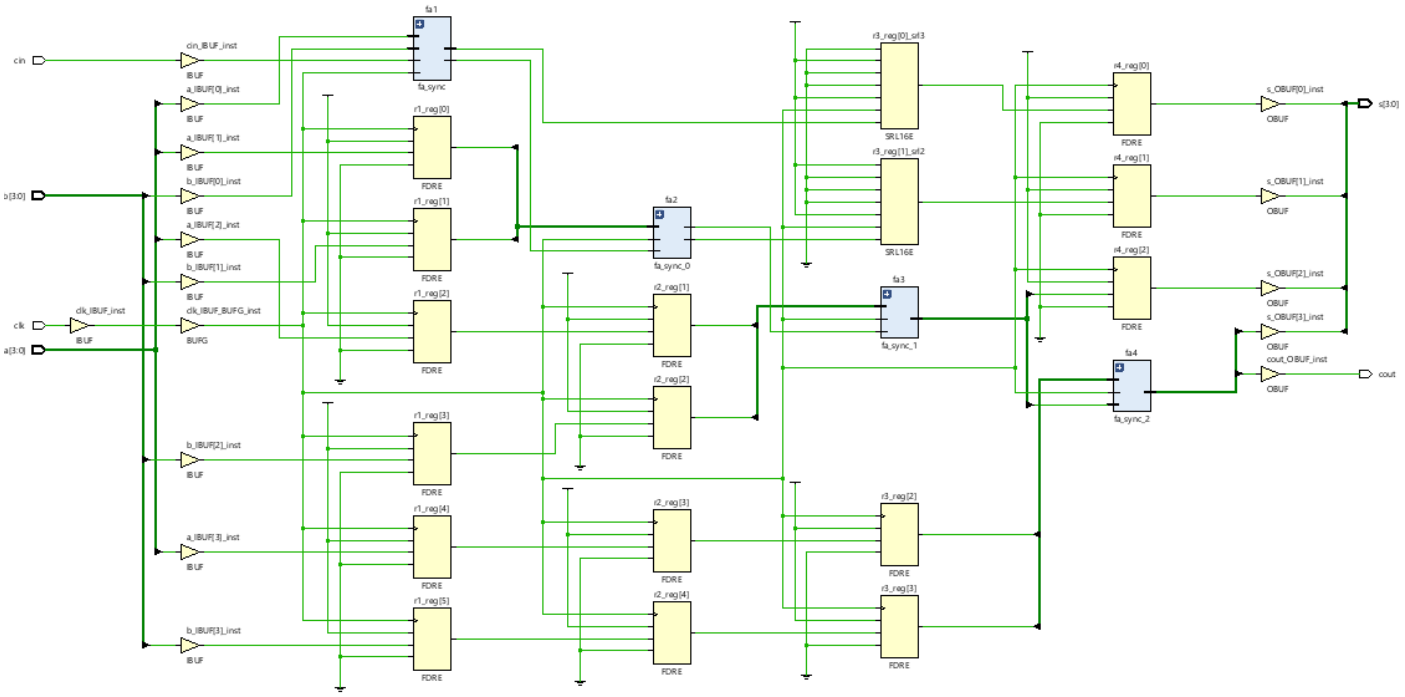
Στην συνέχεια δημιουργήσαμε κατάλληλο testbench που παράγει όλα τα διαφορετικά inputs για τον έλεγχο της ορθής λειτουργίας του κυκλώματος. Ακολουθούν ο κώδικας VHDL του testbench καθώς και κάποια αντιπροσωπευτικά αποτελέσματα του simulation:

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rca4_pipe_tb is
end rca4_pipe_tb;

architecture rca4_pipe_tb_arch of rca4_pipe_tb is
    component rca4_pipe is
        port(
            a: in std_logic_vector(3 downto 0);
            b: in std_logic_vector(3 downto 0);
            cin: in std_logic;
            clk: in std_logic;
            s: out std_logic_vector(3 downto 0);
            cout: out std_logic
        );
    end component;

    constant CLKP : time := 1ns;

    signal a_s: std_logic_vector(3 downto 0);
    signal b_s: std_logic_vector(3 downto 0);
    signal cin_s: std_logic;
    signal s_s: std_logic_vector(3 downto 0);
    signal cout_s: std_logic;
    signal clk: std_logic;
begin
    clk_proc: process
    begin
        clk <= '0';
        wait for CLKP/2;
        clk <= '1';
        wait for CLKP/2;
    end process;

    UUT: rca4_pipe port map (a_s,b_s,cin_s,clk,s_s,cout_s);
    testSequence: process
        begin
            a_s <= "0000";
            b_s <= "0000";

            cin_s <= '0';
            for i in 0 to 15 loop
                for j in 0 to 15 loop
                    wait for CLKP;
                    a_s <= a_s + 1;
                end loop;
                b_s <= b_s + 1;
            end loop;
```

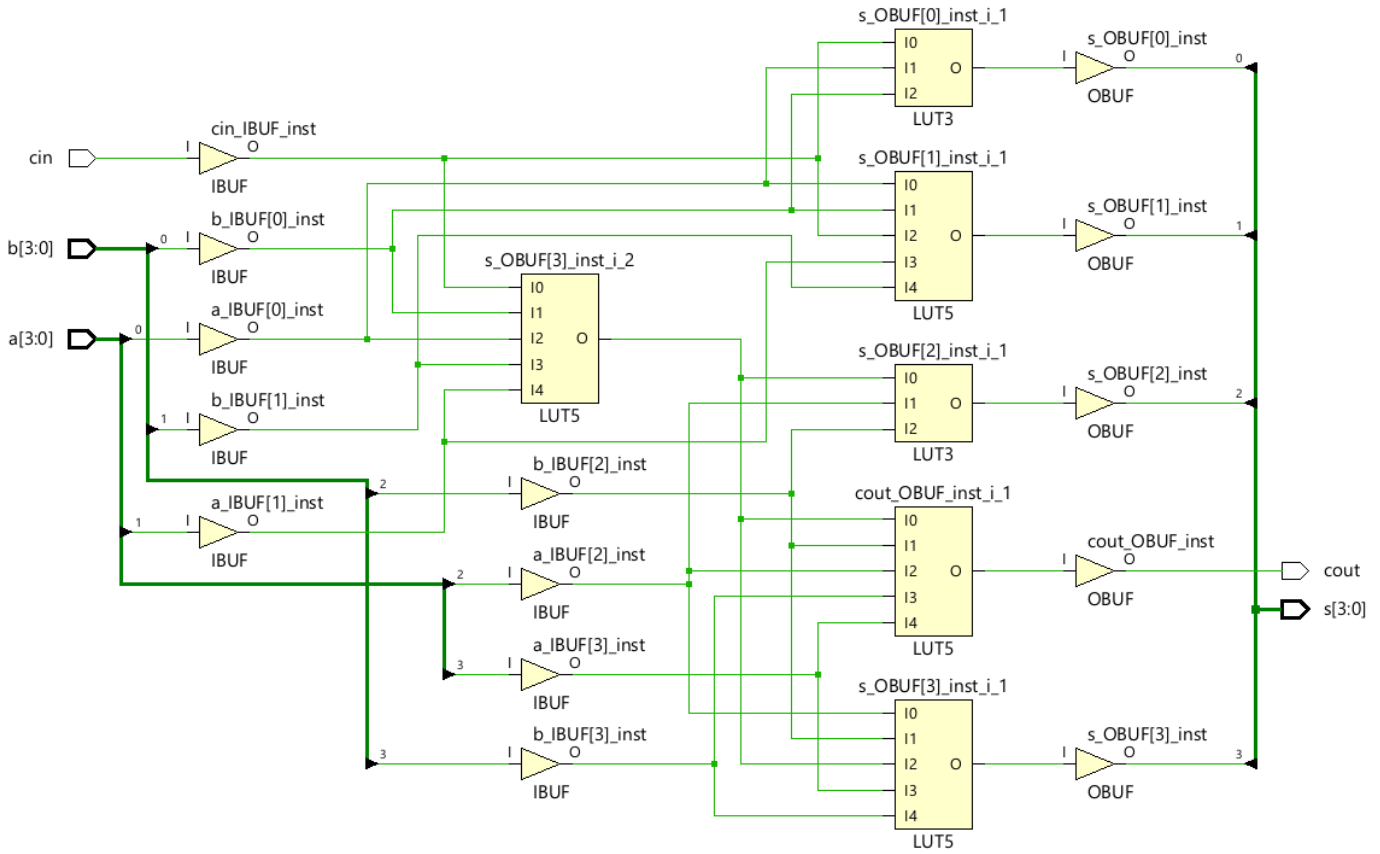
```
            cin_s <= '1';
            for i in 0 to 15 loop
                for j in 0 to 15 loop
                    wait for CLKP;
                    a_s <= a_s + 1;
                end loop;
                b_s <= b_s + 1;
            end loop;
            wait;
    end process;
end rca4_pipe_tb_arch;
```

Αξίζει να παρατηρήσουμε πως μεσολαβούν 3 κύκλοι ρολογιού μέχρι να ξεκινήσει να εξάγει έγκυρα αποτελέσματα ο αθροιστής. Αυτό οφείλεται στο βάθος του pipeline το οποίο παράγει έγκυρα αποτελέσματα σε κάθε κύκλο ρολογιού αφού έχει γίνει και ο υπολογισμός του 4^{ου} και τελευταίου bit του πρώτου αποτελέσματος.

Έχοντας βεβαιωθεί πως το κύκλωμα λειτουργεί όπως είναι αναμενόμενο, προχωράμε στη διαδικασία σύνθεσης με στόχο τον υπολογισμό του critical path και κατ' επέκταση της μέγιστης χρονικής καθυστέρησης με την βοήθεια του Vivado. Παραθέτουμε το schematic της σύνθεσης αξιοποιώντας πλέον πόρους του FPGA και τις μετρήσεις των χρονικών καθυστερήσεων ανά μονοπάτι:



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	∞	2	2	1	fa4/result_reg[1]/C	cout	4.076	3.276	0.800
Path 2	∞	2	2	1	r4_reg[2]/C	s[2]	4.076	3.276	0.800
Path 3	∞	2	2	1	fa4/result_reg[0]/C	s[3]	4.076	3.276	0.800
Path 4	∞	2	2	1	r4_reg[0]/C	s[0]	4.058	3.258	0.800
Path 5	∞	2	2	1	r4_reg[1]/C	s[1]	4.058	3.258	0.800
Path 6	∞	2	3	2	cin	r3_reg[0]_srl3/D	2.239	1.106	1.133
Path 7	∞	2	3	2	cin	fa1/result_reg[1]/D	1.932	1.132	0.800
Path 8	∞	2	2	2	r1_reg[0]/C	r3_reg[1]_srl2/D	1.836	0.751	1.085
Path 9	∞	1	2	1	a[1]	r1_reg[0]/D	1.782	0.982	0.800
Path 10	∞	1	2	1	b[1]	r1_reg[1]/D	1.782	0.982	0.800



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	∞	4	5	3	b[0]	cout	5.970	3.904	2.066
Path 2	∞	4	5	3	b[0]	s[2]	5.970	3.904	2.066
Path 3	∞	4	5	3	b[0]	s[3]	5.964	3.898	2.066
Path 4	∞	3	4	3	b[0]	s[0]	5.351	3.752	1.599
Path 5	∞	3	4	2	b[1]	s[1]	5.351	3.752	1.599

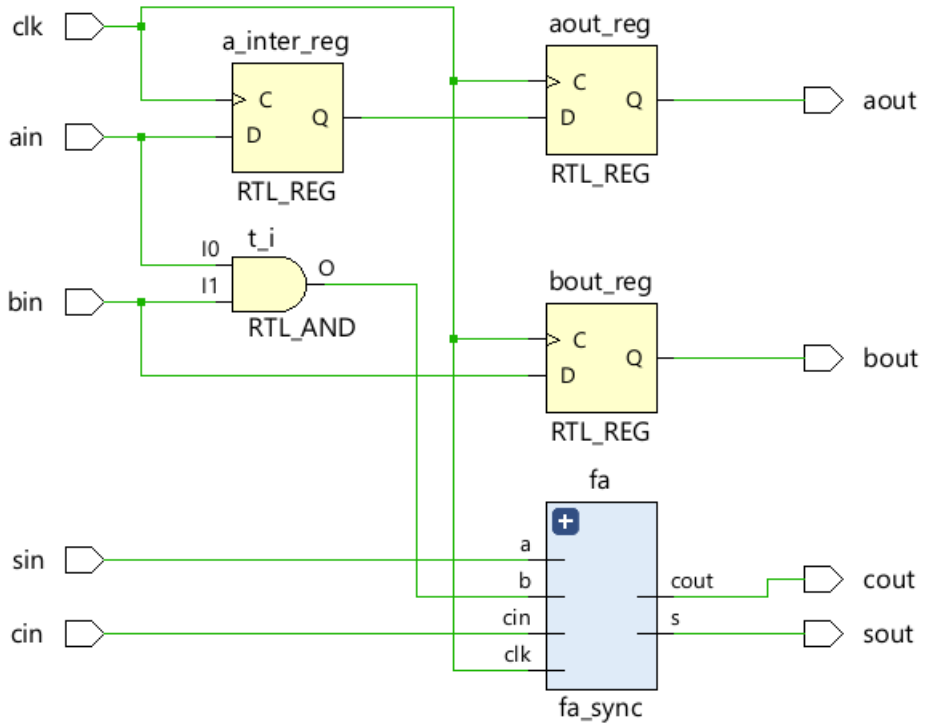
Παρατηρώ πως η χρήση της τεχνικής του pipeline για την κατασκευή του αθροιστή 4 bits αυξάνει σημαντικά την πολυπλοκότητα του κυκλώματος, οδηγώντας σε περεταίρω κατανάλωση πόρων κυρίως λόγω των registers που επιτυγχάνουν τον συγχρονισμό των δεδομένων. Ωστόσο όπως φαίνεται και από το timing summary με τα critical paths, το path στον pipeline αθροιστή είναι σημαντικά μικρότερο από αυτό στον απλό, αφού στην δεύτερη περίπτωση περιοριζόμαστε από το dataflow. Συνεπώς με την τεχνική του pipeline, δημιουργήσαμε ένα σαφώς μεγαλύτερο κύκλωμα με καθυστέρηση κατά την εκκίνησή του, που όμως πετυχαίνει σημαντικά καλύτερο throughput.

Άσκηση 3

Στο τρίτο και τελευταίο κομμάτι του εργαστηρίου σχεδιάσαμε έναν Συστολικό Πολλαπλασιαστή διάδοσης κρατουμένων των 4 bits κάνοντας χρήση των σύγχρονων Πλήρων Αθροιστών της πρώτης άσκησης. Για την δημιουργία ενός απλού συνεχούς pipeline θα αρκούσε η εισαγωγή καθυστερήσεων μεταξύ των κυττάρων ενός κανονικού πολλαπλασιαστή 4 bits. Ωστόσο αν και προκύπτει το ίδιο αποτέλεσμα στο συγκεκριμένο κύκλωμα, επιλέξαμε να παραμείνουμε τυπικοί στον ορισμό του συστολικού pipeline και πρώτα σχεδιάσαμε ένα συστολικό κύτταρο, με την επανάληψη του οποίου μπορούμε να κατασκευάσουμε το μεγαλύτερο μέρος της λειτουργικότητας του πολλαπλασιαστή. Παρουσιάζουμε τον κώδικα VHDL καθώς και το RTL Schematic του συστολικού κυττάρου:

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

entity mul_cell is
    port(
        ain: in std_logic;
        bin: in std_logic;
        cin: in std_logic;
        sin: in std_logic;
        clk: in std_logic;
        aout: out std_logic;
        bout: out std_logic;
        cout: out std_logic;
        sout: out std_logic
    );
end mul_cell;

architecture mul_cell_arch of mul_cell is
    component fa_sync is
        port(
        a: in std_logic;
        b: in std_logic;
        cin: in std_logic;
        clk: in std_logic;
        s: out std_logic;
        cout: out std_logic
    );
    end component;
    signal a_inter,t: std_logic;
begin
```

```
    sync_mul: process(clk)
    begin
        if clk'event and clk='1' then
            aout <= a_inter;
            a_inter <= ain;
            bout <= bin;
        end if;
    end process;
    t <= ain and bin;
    fa: fa_sync port map(
        a => sin,
        b => t,
        cin => cin,
        clk => clk,
        s => sout,
        cout => cout
    );
end mul_cell_arch;
```



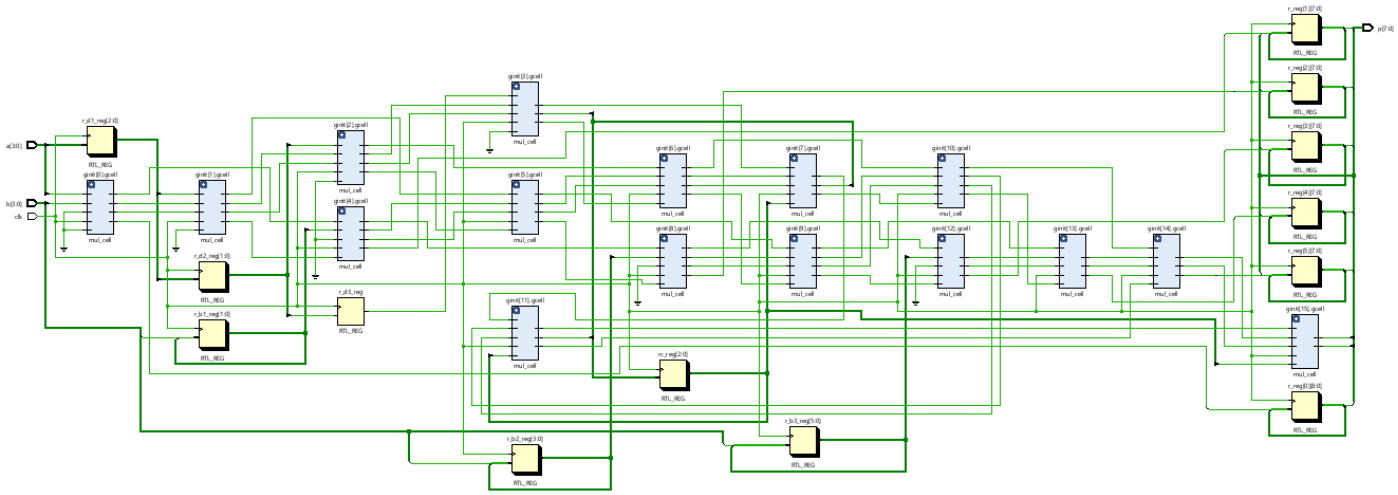
Στην συνέχεια, αξιοποιώντας 16 τέτοια κύτταρα καθώς και επιπλέον καταχωρητές για τον σωστό συγχρονισμό του pipeline ως προς την διάδοση των inputs a και b, των εξωτερικών κρατουμένων cout καθώς και των γινομένων σχεδιάσαμε ολόκληρο τον συστολικο πολλαπλασιαστή. Ακολουθεί ο κώδικας VHDL καθώς και το RTL Schematic του κυκλώματος:

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

entity mul_syst 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 mul_syst;
```

```
architecture mul_syst_arch of mul_syst is
    component mul_cell is
        port(
        ain: in std_logic;
        bin: in std_logic;
        cin: in std_logic;
        sin: in std_logic;
        clk: in std_logic;
        aout: out std_logic;
        bout: out std_logic;
        cout: out std_logic;
        sout: out std_logic
    );
    end component;

    -- signals
    signal ai, bi, ci, si, ao, bo, co, so: std_logic_vector(15 downto 0);

    -- horizontal registers (by levels of deep)
    signal r_b1: std_logic_vector(1 downto 0);
    signal r_b2: std_logic_vector(3 downto 0);
    signal r_b3: std_logic_vector(5 downto 0);

    -- diagonal registers (by levels of deep)
    signal r_d1: std_logic_vector(2 downto 0);
    signal r_d2: std_logic_vector(1 downto 0);
    signal r_d3: std_logic;

    -- outer carry register (gtsm)
    signal rc: std_logic_vector(2 downto 0);

    -- vertical register array (by bit position)
    type reg_array is array (0 to 5) of std_logic_vector(8 downto 0);
    signal r: reg_array;

begin
    --pipeline gtsm (c outer)
    pipeline_outer: process(clk)
    begin
        if clk'event and clk='1' then
            rc <= co(11) & co(7) & co(3);
        end if;
    end process;

    --pipeline diagonal (a)
    pipeline_diag: process(clk)
    begin
        if clk'event and clk='1' then
            r_d3 <= r_d2(1);
            r_d2 <= r_d1(2 downto 1);
            r_d1 <= a(3 downto 1);
        end if;
    end process;
```

```
    --pipeline horizontal (b)
    pipeline_hor: process(clk)
    begin
        if clk'event and clk='1' then
            for i in 5 downto 1 loop
                r_b3(i) <= r_b3(i-1);
            end loop;
            r_b3(0) <= b(3);

            for i in 3 downto 1 loop
                r_b2(i) <= r_b2(i-1);
            end loop;
            r_b2(0) <= b(2);

            r_b1(1) <= r_b1(0);
            r_b1(0) <= b(1);
        end if;
    end process;

    -- pipeline vertical
    pipeline_vert: process(clk)
    begin
        if clk'event and clk='1' then
            for ii in 0 to 5 loop
                for jj in 8 downto 1 loop
                    r(ii)(jj) <= r(ii)(jj-1);
                end loop;
                if ii < 3 then
                    r(ii)(0) <= so(4*ii);
                else
                    r(ii)(0) <= so(9+ii);
                end if;
            end loop;
        end if;
    end process;

    -- cells init
    ginit: for i in 0 to 15 generate
        gcell: mul_cell port map(ai(i), bi(i), ci(i), si(i), clk, ao(i), bo(i), co(i),
so(i));
    end generate;

    -- inter wires
    ga: for i in 4 to 15 generate
       ai(i) <= ao(i-4);
    end generate;

    ci(0) <= '0';
    ci(4) <= '0';
    ci(8) <= '0';
    ci(12) <= '0';
    gbci: for i in 0 to 3 generate
        gbcj: for j in 1 to 3 generate
```

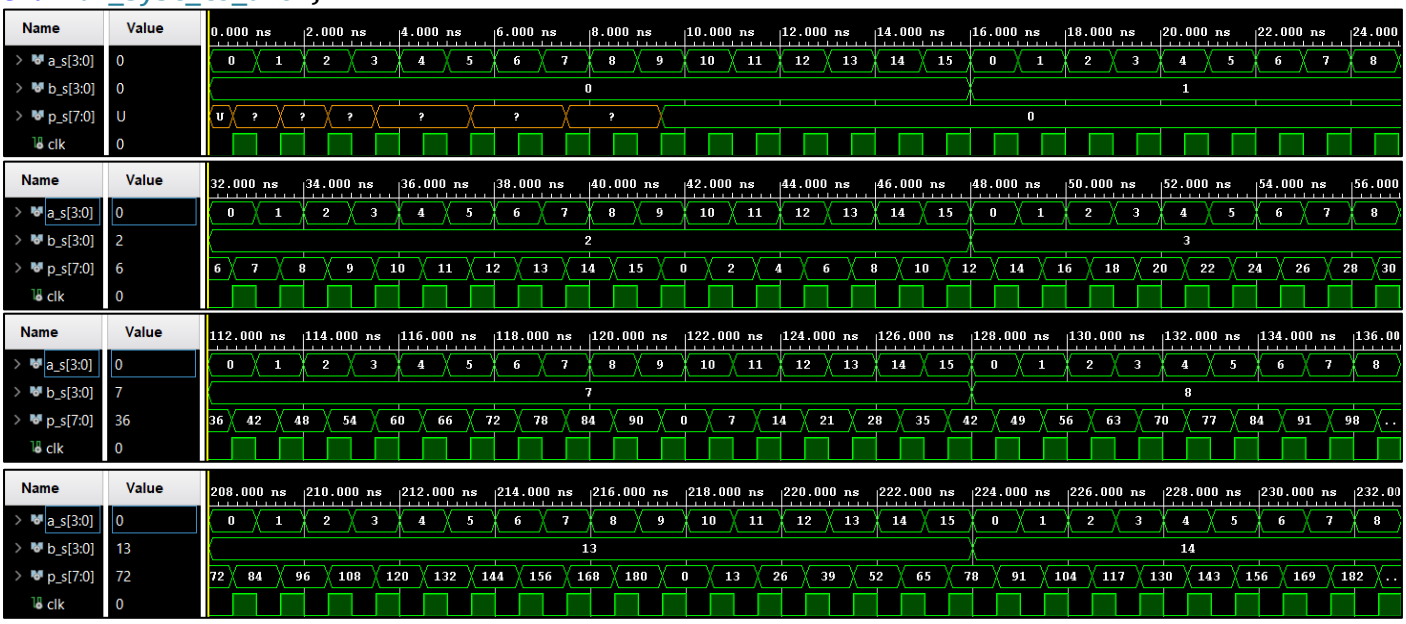
```
            bi(4*i+j) <= bo(4*i+j-1);
            ci(4*i+j) <= co(4*i+j-1);
        end generate;
    end generate;

    si(3 downto 0) <= "0000";
    gs: for i in 1 to 3 generate
        si(4*i) <= so(4*(i-1)+1);
        si(4*i+1) <= so(4*(i-1)+2);
        si(4*i+2) <= so(4*(i-1)+3);
        si(4*i+3) <= rc(i-1);
    end generate;

    -- input
    ai(0) <= a(0);
    ai(1) <= r_d1(0);
    ai(2) <= r_d2(0);
    ai(3) <= r_d3;
    bi(0) <= b(0);
    bi(4) <= r_b1(1);
    bi(8) <= r_b2(3);
    bi(12) <= r_b3(5);

    -- result
    p <= co(15) & so(15) & r(5)(0) & r(4)(1) & r(3)(2) & r(2)(4) & r(1)(6) & r(0)(8);
end mul_syst_arch;
```

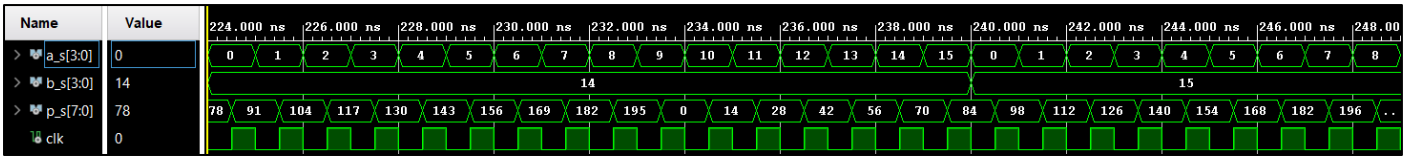


Στην συνέχεια δημιουργήσαμε κατάλληλο testbench που παράγει όλα τα διαφορετικά inputs για τον έλεγχο της ορθής λειτουργίας του κυκλώματος. Ακολουθούν ο κώδικας VHDL του testbench καθώς και κάποια αντιπροσωπευτικά αποτελέσματα του simulation:

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mul_syst_tb is
end mul_syst_tb;

architecture mul_syst_tb_arch of mul_syst_tb is
    component mul_syst is
        port(
```

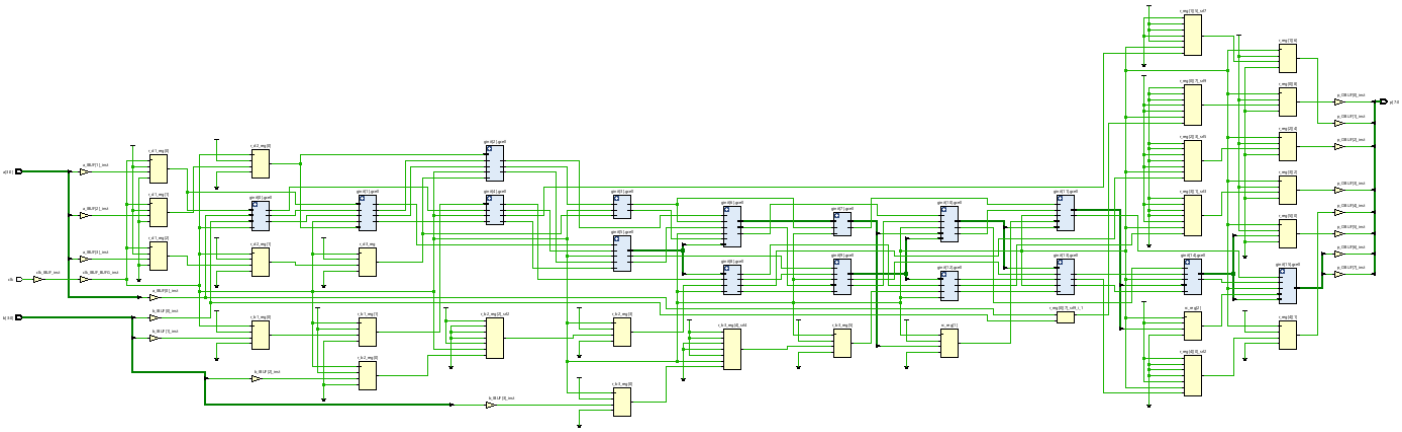
```
            a: in std_logic_vector(3 downto 0);
            b: in std_logic_vector(3 downto 0);
            clk: in std_logic;
            p: out std_logic_vector(7 downto 0)
        );
    end component;

    constant CLKP : time := 1ns;

    signal a_s: std_logic_vector(3 downto 0);
    signal b_s: std_logic_vector(3 downto 0);
    signal p_s: std_logic_vector(7 downto 0);
    signal clk: std_logic;
begin
    clk_proc: process
    begin
        clk <= '0';
        wait for CLKP/2;
        clk <= '1';
        wait for CLKP/2;
    end process;

    UUT: mul_syst port map (a_s,b_s,clk,p_s);
    testSequence: process
        begin
            a_s <= "0000";
            b_s <= "0000";
            for i in 0 to 15 loop
                for j in 0 to 15 loop
                    wait for CLKP;
                    a_s <= a_s + 1;
                end loop;
                b_s <= b_s + 1;
            end loop;
    end process;
end mul_syst_tb_arch;
```





Αξίζει να παρατηρήσουμε πως μεσολαβούν 9 κύκλοι ρολογιού μέχρι να ξεκινήσει να εξάγει έγκυρα αποτελέσματα ο πολλαπλασιαστής. Αυτό οφείλεται στο βάθος του pipeline το οποίο έχει μέγιστο βάθος 10 για να γίνει σωστός συγχρονισμός των δεδομένων μέχρι την παραγωγή του κάθε αποτελέσματος.

Έχοντας βεβαιωθεί πως το κύκλωμα λειτουργεί όπως είναι αναμενόμενο, προχωράμε στη διαδικασία σύνθεσης με στόχο τον υπολογισμό του critical path και κατ' επέκταση της μέγιστης χρονικής καθυστέρησης με την βοήθεια του Vivado. Παραθέτουμε το schematic της σύνθεσης αξιοποιώντας πλέον πόρους του FPGA και τις μετρήσεις των χρονικών καθυστερήσεων ανά μονοπάτι:



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	∞	2	2	1	r_reg[5][0]/C	p[5]	4.076	3.276	0.800
Path 2	∞	2	2	1	ginit[15].gcell/fa/result_reg[0]/C	p[6]	4.076	3.276	0.800
Path 3	∞	2	2	1	ginit[15].gcell/fa/result_reg[1]/C	p[7]	4.076	3.276	0.800
Path 4	∞	2	2	1	r_reg[0][8]/C	p[0]	4.058	3.258	0.800
Path 5	∞	2	2	1	r_reg[1][6]/C	p[1]	4.058	3.258	0.800
Path 6	∞	2	2	1	r_reg[2][4]/C	p[2]	4.058	3.258	0.800
Path 7	∞	2	2	1	r_reg[3][2]/C	p[3]	4.058	3.258	0.800
Path 8	∞	2	2	1	r_reg[4][1]/C	p[4]	4.058	3.258	0.800
Path 9	∞	2	3	2	b[0]	r_reg[0][7]_srl9/D	2.239	1.106	1.133
Path 10	∞	2	2	2	ginit[1].gcell/fa/result_reg[0]/C	r_reg[1][5]_srl7/D	1.836	0.751	1.085

Παρατηρώ στο timing summary πως η χρονική διάρκεια των critical paths είναι εντυπωσιακά μικρή χάρη στην παρεμβολή registers μεταξύ των δομικών μονάδων του κυκλώματος και του κατάλληλου συγχρονισμού. Στην περίπτωση που υλοποιούσαμε το κύκλωμα χωρίς την τεχνική του pipeline θα είχαμε μικρότερη κατανάλωση πόρων κυρίως σε registers και μικρότερη πολυπλοκότητα (και ίσως μικρότερο net delay), ωστόσο θα υπήρχε μεγάλη καθυστέρηση εξαιτίας του dataflow και της μειωμένης συχνότητας ρολογιού που θα απαιτούσε. Αντιθέτως στην προκειμένη περίπτωση η παρεμβολή registers και η χρήση συστολικών κυττάρων αυξάνει σημαντικά το τελικό throughput του κυκλώματος.