2024

Αναφορά 2ης εργαστηριακής Άσκησης

Εργαστήριο VLSI

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

Ζήτημα 1^{ο} : Hald Adder

Σε αυτό το ερώτημα ζητείται να υλοποιήσουμε έναν Half Adder σε dataflow αρχιτεκτονική. Γνωρίζουμε ότι ο Half Adder έχει ως είσοδο δύο bit (έστω A και B εδώ) και ως έξοδο το άθροισμα Sum και το κρατούμενο εξόδου Cout. Έτσι κατασκευάζουμε αρχικά τον πίνακα αληθείας για τον Half Adder:

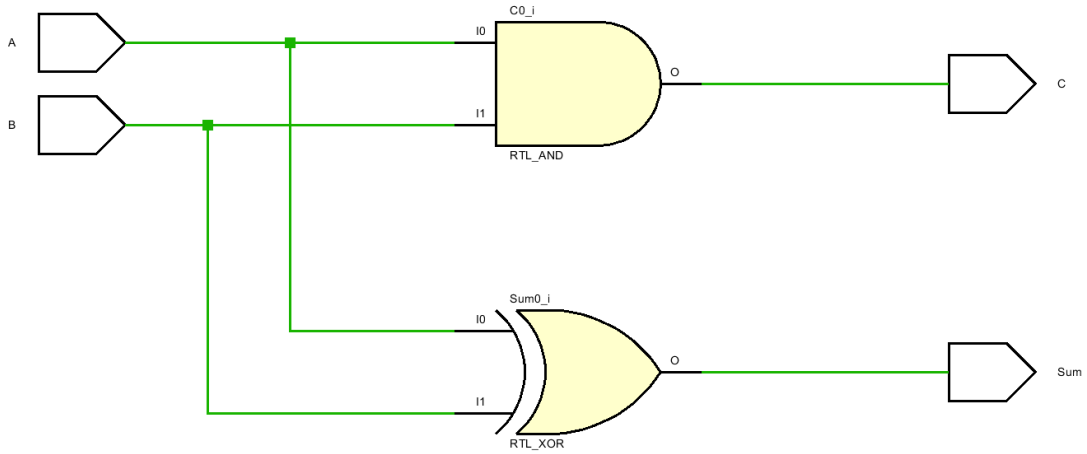
Είσοδος		Έξοδος	
A	**B**	**Cout**	**Sum**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Από τον παραπάνω πίνακα αληθείας παρατηρούμε ότι το Cout είναι ουσιαστικά η έξοδος μίας πύλης AND με εισόδους το A και το B ενώ το Sum μία XOR μεταξύ του A και του B. Με βάση αυτά κατασκευάζουμε τον Half Adder:

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

entity half_adder is
    Port (
        A, B: in std_logic;
        Sum, C: out std_logic);
end half_adder;

architecture Dataflow of half_adder is
begin
    hf: process(A, B)
    begin
        Sum <= A xor B;
        C <= A and B;
    end process;
end Dataflow;
```

Στον παραπάνω VDHL κώδικα ορίζουμε το entity του Half Adder με εισόδους και εξόδους όπως αναφέρθηκαν παραπάνω, ενώ τα Sum και C (το οποίο είναι το Cout) προκύπτουν χρησιμοποιώντας απλώς τις πύλες που αναφέρθηκαν παραπάνω.

Το RTL σχηματικό του Half Adder είναι το ακόλουθο:



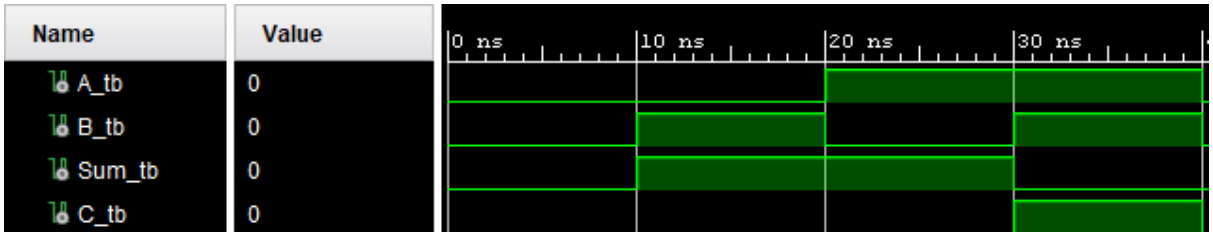
Στο σχηματικό βλέπουμε και πάλι την απλή αρχιτεκτονική του Half Adder.

Για να ελέγξουμε ότι λειτουργεί ο Half Adder που κατασκευάσαμε όπως αναμενόταν κατασκευάζουμε το ακόλουθο testbench:

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

entity half_adder_testbench is
end half_adder_testbench;

architecture Behavioral of half_adder_testbench is
    component half_adder
        port(
            A, B: in std_logic;
            Sum, C: out std_logic
            );
    end component;
    signal A_tb, B_tb, Sum_tb, C_tb: std_logic;
begin

    uut: half_adder port map(
        A => A_tb,
        B=> B_tb,
        Sum=> Sum_tb,
        C=> C_tb
        );
    stimulus_process: process
    begin
        A_tb <= '0';
        B_tb <= '0';
        wait for 10ns;
```

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

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

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

end Behavioral;
```

Σε αυτό το testbench θα εξετάσουμε ότι ο Half Adder έχει την αναμενόμενη έξοδο για κάθε πιθανή είσοδο. Το αποτέλεσμα της προσομοίωσης είναι:



Βλέπουμε λοιπόν ότι για κάθε τιμή του εισόδου ο Half Adder έχει την αναμενόμενη έξοδο.

Στην συνέχεια για να βρούμε το Critical Path τρέχουμε Synthesis και το αποτέλεσμα είναι το ακόλουθο:

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	3	4	2	A	C	5.377	3.778	1.599	∞	input port clock
Path 2	∞	3	4	2	A	Sum	5.351	3.752	1.599	∞	input port clock

Βλέπουμε λοιπόν ότι το Critical Path είναι το μονοπάτι από το A στο C.

Ζήτημα 2ο : Full Adder

Σε αυτό το ζήτημα ζητείται να φτιάξουμε έναν Full Adder. Ο Full Adder αποτελεί ουσιαστικά μία επέκταση του Half Adder αφού εκτός από τα A και B ως εισόδους παίρνει και ένα κρατούμενο εισόδου Cin ως είσοδο. Η χρησιμότητα αυτής της επιπλέον εισόδου θα φανεί στα επόμενα ερωτήματα.

Ο πίνακας αληθείας του Full Adder είναι ο ακόλουθος:

Είσοδος			**Έξοδος**	
A	**B**	**Cin**	**Cout**	**Sum**
0	0	0	0	0
0	0	1	0	1

0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Αρχικά είναι προφανές ότι θα χρησιμοποιήσουμε 2 Half Adders έτσι ώστε να αθροίσουμε τα 3 bits εισόδου. Από τον πίνακα αληθείας βλέπουμε λοιπόν ότι το Cout του Half Adder να είναι 1 όταν έχουμε παραπάνω από 1 άσσους στην είσοδο. Το ίδιο ακριβώς κάνει και το Cout σε έναν Half Adder. Έτσι μπορούμε χρησιμοποιώντας τους 2 Half Adder μπορούμε να το εντοπίσουμε αυτό και το Cout του Full Adder να είναι απλώς μία πύλη OR μεταξύ των Cout των 2 Half Adders. Αντίθετα με το Cout το Sum είναι 1 όταν έχουμε έναν ή τρεις άσσους στην είσοδο. Όπως είδαμε και στο προηγούμενο ζήτημα το Sum είναι 1 στον Half Adder όταν έχουμε 1 μόνο άσσο.

Από τα παραπάνω καταλαβαίνουμε ότι μπορούμε να κατασκευάσουμε έναν Full Adder με δύο Half Adder. Ο πρώτος Half Adder θα έχει ως είσοδο τα A και B (προφανώς θα μπορούσε να είναι οποιοσδήποτε από τους 3 συνδυασμούς μεταξύ των A, B και Cin). Για ευκολία θα ονομάσουμε τις εξόδους αυτού του Half Adder ως C1 για το Cout και S1 για το Sum. Στην συνέχεια ο δεύτερος Half Adder θα έχει ως είσοδο το Cin και το S1. Με αυτές τις εισόδους αν στον πρώτο Half Adder είχαμε άσσο αυτός θα «μεταφερόταν» στον δεύτερο Half Adder. Για ευκολία κάνουμε και έναν πίνακα αληθείας για τον δεύτερο Half Adder:

Είσοδος		**Έξοδος**	
S1	**Cin**	**C2**	**S2**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

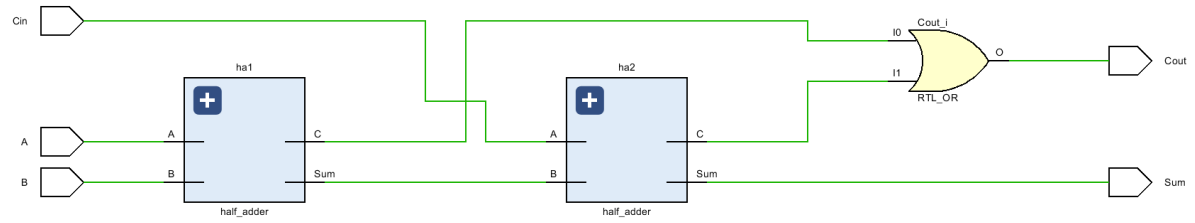
Παρατηρούμε λοιπόν ότι αν το S1 είναι 0, δηλαδή τα A και B είναι ή και τα δύο άσσοι ή 0 το S2 είναι ίσο με το Cin. Αντίθετα αν το S1 είναι 1, δηλαδή είναι μόνο ένα εκ των A και B 1 τότε το S2 είναι το αντίθετο από το Cin. Τα παραπάνω αποτελέσματα μας κάνουν να καταλάβουμε ότι το S2 είναι ουσιαστικά η έξοδος Sum του Full Adder αν παρατηρήσουμε προσεκτικά των πίνακα αληθείας του Full Adder. Επίσης βλέπουμε ότι το Cout είναι πράγματι το C1 OR C2.

Έτσι η structural αρχιτεκτονική του Full Adder είναι:

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

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

architecture Structural of full_adder is
component half_adder
```

```
        port(
            A, B: in std_logic;
            Sum, C: out std_logic
            );
    end component;
signal S1, C1, C2: std_logic;
begin
    ha1: half_adder port map(
        A=>A,
        B=>B,
        Sum=>S1,
        C=>C1
    );
    ha2: half_adder port map(
            A=>Cin,
            B=>S1,
            Sum=>Sum,
            C=>C2
        );
        Cout<=C1 or C2;

end Structural;
```

Στην συνέχεια παίρνουμε το RTL σχηματικό:



Βλέπουμε λοιπόν ότι το σχηματικό αποτελείται από ό,τι έχουμε περιγράψει προηγουμένως.

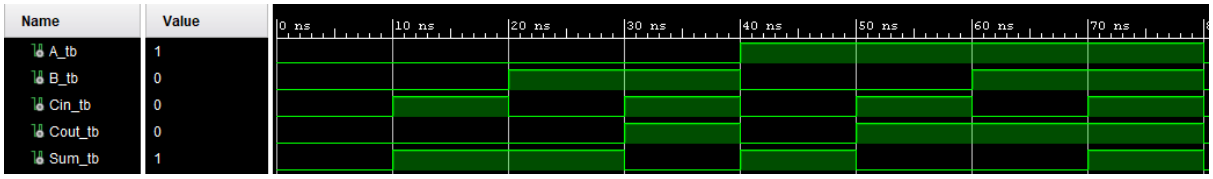
Για να ελέγξουμε ότι ο κώδικας που περιγράψαμε λειτουργεί σωστά κατασκευάσουμε το ακόλουθο testbench:

```
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;
```

Σε αυτό το testbench ελέγχουμε ότι έχουμε την κατάλληλη έξοδο στον Full Adder μας για κάθε πιθανή είσοδο. Το αποτέλεσμα του Simulation είναι:



Βλέπουμε λοιπόν ότι πράγματι έχουμε την αναμενόμενη έξοδο για κάθε είσοδο.

Για να βρούμε το critical path τρέχουμε synthesis και το αποτέλεσμα είναι:

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	A	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 αυτό από το A στο Cout με καθυστέρηση ίδια με αυτή στον Half Adder.

Ζήτημα 3ο: 4-bit Parallel Adder

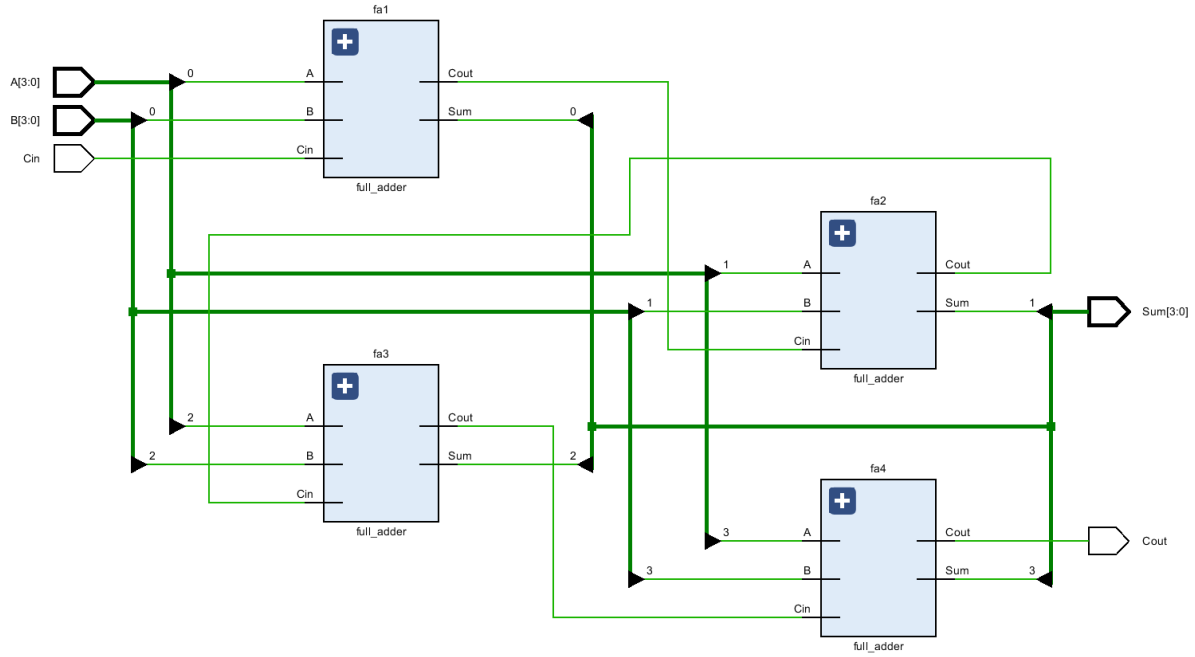
Αφού λοιπόν κατασκευάσαμε τον Full Adder είναι πολύ εύκολο να κατασκευάσουμε τον παράλληλο αθροιστή 4 bit χρησιμοποιώντας 4 Full Adder όπου το Cin του κάθενός θα είναι το Cout του προηγούμενου. Για τον πρώτο Full Adder το Cin του για να έχουμε απλή πρόσθεση 4 bit πρέπει να είναι συνεχώς 0. Ωστόσο θα μας χρησιμεύσει στον BCD Full Adder 4 ψηφίων να μπορούμε να θέσουμε το Cin του πρώτου Full Adder, επομένως θα το θέσουμε ως είσοδο και στον παράλληλο αθροιστή.

Ο κώδικας για τον παράλληλο αθροιστή είναι ο ακόλουθος:

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

entity parallel_adder_4_bit is
    Port (
    A, 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 parallel_adder_4_bit;
```

```
architecture Structural of parallel_adder_4_bit is
    component full_adder is
      Port (
        A, B, Cin: in std_logic;
        Sum, Cout: out std_logic);
    end component;
    signal carry: std_logic_vector(2 downto 0);
begin
    fa1: full_adder port map(
        A => A(0),
        B => B(0),
        Cin=>Cin,
        Sum => Sum(0),
        Cout=>carry(0)
   );

   fa2: full_adder port map(
          A => A(1),
          B => B(1),
          Cin=>carry(0),
          Sum => Sum(1),
          Cout=>carry(1)
     );

   fa3: full_adder port map(
                A => A(2),
                B => B(2),
                Cin=>carry(1),
                Sum => Sum(2),
                Cout=>carry(2)
           );

   fa4: full_adder port map(
                      A => A(3),
                      B => B(3),
                      Cin=>carry(2),
                      Sum => Sum(3),
                      Cout=>Cout
                 );

end Structural;
```

Το RTL σχηματικό του παραπάνω κώδικα είναι:

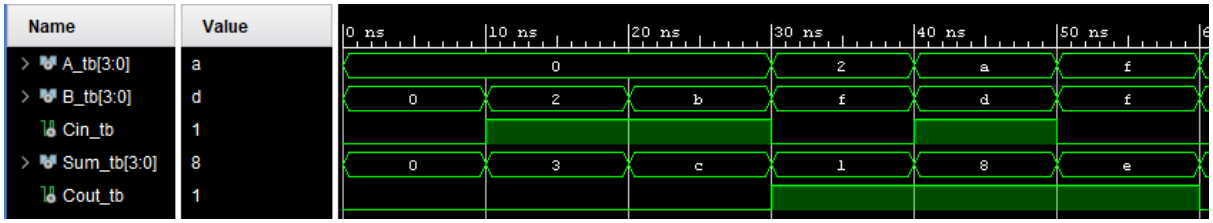


Στην συνέχεια θα φτιάξουμε ένα testbench για να ελέγξουμε ότι λειτουργεί σωστά στο παραπάνω κύκλωμα. Σε αυτό το testbench δεν θα εξετάσουμε κάθε πιθανή είσοδο αλλά κάποια επιλεγμένα testcases. Η είσοδος και το αναμενόμενο αποτέλεσμα φαίνεται σε σχόλια στον κώδικα:

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

entity parallel_adder_4_bit_testbench is
end parallel_adder_4_bit_testbench;

architecture Behavioral of parallel_adder_4_bit_testbench is
    component parallel_adder_4_bit is
        Port (
        A, 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;
    signal A_tb, B_tb: std_logic_vector (3 downto 0);
    signal Cin_tb : std_logic;
    signal Sum_tb: std_logic_vector (3 downto 0);
    signal Cout_tb: std_logic;
begin
    uut: parallel_adder_4_bit port map(
        A=>A_tb,
        B=>B_tb,
        Cin => Cin_tb,
        Sum=>Sum_tb,
        Cout=>Cout_tb
   );
```

```
   stimulus_process: process
   begin
   --Testing if 0+0=0
    Cin_tb<='0';
    A_tb <= "0000";
    B_tb <= "0000";
    wait for 10ns;

    --Testing if 0+x+1=x+1
    Cin_tb<='1';
    A_tb <= "0000";
    B_tb <= "0010";
    wait for 10ns;

    A_tb <= "0000";
    B_tb <= "1011";
    wait for 10ns;

    --Testing if cout works
    -- 2+15=17
    Cin_tb<='0';
    A_tb <= "0010";
    B_tb <= "1111";
    wait for 10ns;

    -- 10+13+1=24
    Cin_tb<='1';
    A_tb <= "1010";
    B_tb <= "1101";
    wait for 10ns;

    -- 15+15=30
    Cin_tb<='0';
    A_tb <= "1111";
    B_tb <= "1111";
    wait for 10ns;

    end process;


end Behavioral;
```

Ακολουθεί το αποτέλεσμα της προσομοίωσης:



Βλέπουμε λοιπόν ότι παίρνουμε πράγματι τα αναμενόμενα αποτελέσματα με βάση τα testcases.

Στην συνέχεια κάνουμε το synthesis για να βρούμε τα critical paths.

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 critical path είναι από το Cin στο Cout και από το Cin στο 2^{ο} σημαντικότερο bit.

Ζήτημα τέταρτο BCD Full Adder

Στο ερώτημα αυτό μας ζητήθηκε να υλοποιήσουμε έναν BCD πλήρη αθροιστή χρησιμοποιώντας σαν βασικό component τον Full Adder που περιγράψαμε παραπάνω. Αρχικά θα αναλύσουμε λίγο τον BCD και θα εξηγήσουμε την λογική σχεδίαση πάνω από την περιγραφή που δώσαμε για το κύκλωμα αυτό. Η BCD αναπαράσταση των αριθμών ουσιαστικά χρησιμοποιεί από 4 δυαδικά ψηφία για την αναπαράσταση κάθε ψηφίου του αντίστοιχου αριθμού στο δεκαδικό σύστημα. Συνεπώς, κάθε ψηφίο ενός δεκαδικού αριθμού μπορεί να αναπαρασταθεί από μία τετράδα από bits(π.χ για 9->1001) και συνεπώς κάθε δεκαδικός αριθμός μπορεί να αναπαρασταθεί από πολλές τετράδες από bits. Ένας BCD αθροιστής πρέπει να πραγματοποιεί την άθροιση δύο BCD αριθμών και να δίνει το αποτέλεσμα σε BCD μορφή. Το αποτέλεσμα θα είναι ένας αριθμός σε BCD μορφή με ίδιο πλήθος ψηφίων και ένα κρατούμενο εξόδου που θα αναπαριστά την τιμή του αμέσως επόμενου σημαντικού ψηφίου ,στο δεκαδικό σύστημα, από το αντίστοιχο MSB των αριθμών που χρησιμοποιήθηκαν για την άθροιση. Εφόσον αναλύσαμε πλήρως τι αναμένουμε από την λειτουργικότητα ενός BCD αθροιστή θα παραθέσουμε στην συνέχεια έναν πίνακα που αντιστοιχίζει κάθε δεκαδικό με τον αντίστοιχο δυαδικό αριθμό και την BCD αναπαράσταση(στην BCD αναπαράσταση οι αριθμοί κυμαίνονται από το 0 μέχρι το 9 συνεπώς ο μεγαλύτερος δεκαδικός αριθμός που μπορεί να συναντήσουμε συνυπολογίζοντας το κρατούμενο εισόδου είναι το 9+9+1=19. Για αυτό ο πίνακας μας θα φτάνει μέχρι αυτό τον αριθμό):

Δεκαδικός	**Άθροισμα σε δυαδικό**					**Άθροισμα σε BCD**				
	C’	S3’	S2’	S1’	S0’	C	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0

5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

Από τον παραπάνω πίνακα μπορούμε να βγάλουμε σημαντικά συμπεράσματα για το τελικό μας αποτέλεσμα. Αρχικά, προφανώς μέχρι τον αριθμό 9 η BCD αναπαράσταση είναι ίδια με την δυαδική. Συνεπώς αν το άθροισμα είναι μέχρι 9 θέλουμε ο αθροιστής μας να εκτελεί μία απλή άθροιση δύο 4-bit δυαδικών αριθμών. Αν ο αριθμός είναι μεγαλύτερος από 10 πρέπει να παρατηρήσουμε δύο πράγματα. Στην BCD αναπαράσταση έχουμε πάντα κρατούμενο C=1 και ότι η BCD αναπαράσταση με την δυαδική διαφέρουν κατά το δυαδικό 6(0110) (αν στην δυαδική αναπαράσταση προσθέσω το 6 θα πάρω την BCD για αριθμούς μεγαλύτερους του 9). Άρα καταλήγουμε στην ακόλουθη αρχιτεκτονική: Πραγματοποιούμε μία κανονική άθροιση των δύο δυαδικών αριθμών. Ανιχνεύουμε αν ο αριθμός είναι μεγαλύτερος από το 10, καθώς τότε η BCD αναπαράσταση θα είχε κρατούμενο εξόδου. Για να ελέγξουμε αν ο αριθμός είναι μεταξύ 10-19 πρέπει να ελέγξουμε ότι ισχύει ένα εκ των ακόλουθων:

- Αν το C'=1 (αν έχουμε κρατούμενο εξόδου τότε SUM = 16-19)
- Αν S3' S2' =1 (τότε το άθροισμα είναι SUM=12-15)
- Αν S3' S1'=1 (τότε το άθροισμα είναι SUM=10-11)

Αν ικανοποιείται ένα εκ των παραπάνω τότε η αντίστοιχη BCD άθροιση θα έχει κρατούμενο εξόδου. Για αυτό θα τοποθετήσουμε δύο AND πύλες για την δεύτερη και την Τρίτη λογική παράσταση και τέλος θα τοποθετήσουμε μία or για να πάρουμε το κρατούμενο εξόδου. Πλέον, με αυτό το κύκλωμα γνωρίζουμε αν θα έχουμε κρατούμενο εξόδου στην BCD άθροιση. Αν έχουμε κρατούμενο εξόδου, τότε πρέπει στο άθροισμα μας(binary αναπαράσταση) πρέπει να προσθέσουμε το 0110->6 για να πάρουμε την BCD αναπαράσταση . Αν δεν έχουμε κρατούμενο εξόδου ,C=0, τότε το αποτέλεσμα μας είναι ήδη σε BCD αναπαράσταση και δεν απαιτείται κάποια τροποποίηση. Άρα αν C=1 πρέπει να προσθέσουμε στο αποτέλεσμα της άθροισης το 0110, ενώ αν C=0 δεν πρέπει να κάνουμε τίποτα. Σε

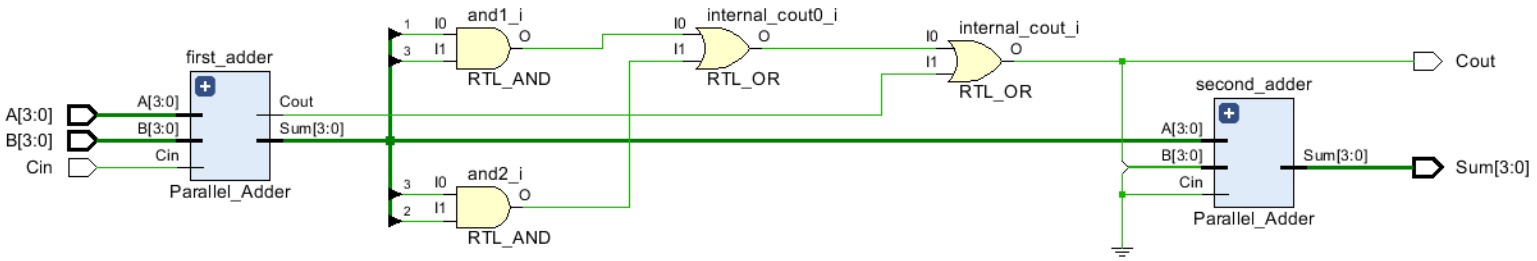
κάθε περίπτωση λοιπόν μπορούμε να προσθέσουμε το 0CC0 στο αποτέλεσμα της αρχικής άθροισης και να λάβουμε το αποτέλεσμα της άθροισης σε BCD μορφή. Με την παραπάνω ανάλυση περιγράψαμε την λογική σχεδίαση του BCD αθροιστή που θα φτιάξουμε και με βάση αυτή θα γράψουμε την περιγραφή υλικού σε VHDL. Αναμένουμε ότι το RTL σχηματικό που θα μας δώσει το Vivado θα έχει την δομή που μόλις περιγράψαμε, καθώς και ο κώδικας μας θα έχει την αντίστοιχη δομή. Αρχικά, παραθέτουμε τον κώδικα σε VHDL:

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

entity BCD_full_adder is
    Port ( 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 BCD_full_adder;

architecture Structural of BCD_full_adder is
    component Parallel_Adder is
        Port ( 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;
    signal and1,and2,mid_cout,internal_cout,dummy : std_logic;
    signal mid_sum,second_in : std_logic_vector ( 3 downto 0);
begin
   first_adder: Parallel_Adder port map (A,B,Cin,mid_sum,mid_cout);
   and1 <= mid_sum(1) and mid_sum(3);
   and2 <= mid_sum(3) and mid_sum(2);
   internal_cout <= and1 or and2 or mid_cout;
   Cout <= internal_cout;
   second_in <= '0' & internal_cout & internal_cout & '0';
   second_adder: Parallel_Adder port map (mid_sum,second_in,'0',Sum,dummy);

end Structural;
```

Στον κώδικα αυτό αρχικά ορίζουμε το entity του BCD Full Adder, παίρνει σαν είσοδο δύο binary αριθμούς από το 0-9 και ένα κρατούμενο εισόδου και δίνει το αποτέλεσμα σε BCD μορφή (ένα κρατούμενο εξόδου και ένας 4 bit BCD αριθμός). Αφού ορίσουμε το entity ορίζουμε το component του παράλληλου αθροιστή που υλοποιήσαμε στο ερώτημα 3 και στην συνέχεια ορίζουμε όποια σήματα είναι απαραίτητα για την υλοποίηση. Αναφορικά με την υλοποίηση, ορίζουμε τον πρώτο αθροιστή, όπου θα αθροίζονται οι 2 αριθμοί είσοδοι και το κρατούμενο εισόδου. Έπειτα, ορίζουμε δύο AND και μία or , η οποία όπως ορίσαμε θα μας δώσει το κρατούμενο εξόδου όπως εξηγήσαμε προηγουμένως. Τέλος ορίζουμε τον δεύτερο αθροιστή του οποίου το αποτέλεσμα θα είναι η BCD

αναπαράσταση του τελικού αποτελέσματος. Σαν είσοδο σε αυτόν βάζουμε το αποτέλεσμα του προηγούμενου αθροιστή και το 0CC0 (όπου C το κρατούμενο εξόδου της BCD άθροισης). Προφανώς, το κρατούμενο εισόδου το θέτουμε ίσο με 0 για τον αθροιστή αυτόν. Το κρατούμενο εξόδου δεν μας ενδιαφέρει οπότε το αναθέτουμε σε ένα dummy σήμα.

Με βάση την παραπάνω περιγραφή υλικού προκύπτει ένα RTL διάγραμμα από το Vivado και είναι το ακόλουθο:



Το παραπάνω διάγραμμα είναι ακριβώς ίδιο με αυτό που περιγράψαμε στην αρχή (για το πως θα υλοποιήσουμε τον BCD αθροιστή) γεγονός αναμενόμενο καθώς αν γίνει ορθή περιγραφή, το Vivado θα πρέπει να παίρνει την περιγραφή αυτή και να μπορεί να αποτυπώσει αυτό που θέλαμε εξ αρχής. Η μόνη διαφορά με την αρχική μας περιγραφή είναι ότι εμείς αναφέραμε μία πύλη or τριών εισόδων που θα καθορίζει το κρατούμενο εξόδου, ενώ εδώ χρησιμοποιούνται 2 or για να πραγματοποιήσουν κάτι απολύτως ισοδύναμο. Επιπλέον, μέσω του synthesized design μπορούμε και πάλι να δούμε το critical path του παραπάνω κυκλώματος, το οποίο παραθέτουμε ακολούθως:

Path 1	∞	4	5	4	B[1]	Sum[1]	5.976	3.904	2.072	∞	input port clock
Path 2	∞	4	5	4	B[1]	Sum[3]	5.976	3.904	2.072	∞	input port clock
Path 3	∞	4	5	4	Cin	Cout	5.948	3.876	2.072	∞	input port clock
Path 4	∞	4	5	4	Cin	Sum[2]	5.948	3.876	2.072	∞	input port clock
Path 5	∞	3	4	3	Cin	Sum[0]	5.351	3.752	1.599	∞	input port clock

Βλέπουμε ότι το μεγαλύτερο Setup time το έχει το MSB του Sum γεγονός αναμενόμενο, καθώς όπως δείξαμε παραπάνω σε έναν παράλληλο αθροιστή το MSB απαιτεί τον περισσότερο χρόνο για να υπολογιστεί. Σε αυτό το επίπεδο στην καθυστέρηση βέβαια είναι και το τρίτο πιο σημαντικό bit του BCD αριθμού στην έξοδο.

Τέλος, υλοποιήσαμε ένα testbench για να ελέγξουμε την ορθή λειτουργία του αθροιστή μας. Παραθέτουμε ακολούθως τον κώδικα για το testbench αυτό:

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

entity TB_BCD_FA is
end TB_BCD_FA;

architecture Testbench of TB_BCD_FA is
    component BCD_full_adder is
        Port ( 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;
        signal i,j : integer :=0;
        signal Cout,Cin : std_logic;
        signal A,B,Sum : std_logic_vector (3 downto 0);
begin
uut: BCD_full_adder port map(A,B,Cin,Sum,Cout);
   stim_process: process
    begin
      for i in 0 to 9 loop
        for j in 0 to 9 loop
            A <= std_logic_vector(to_unsigned(i, A'length));
            B <= std_logic_vector(to_unsigned(j, B'length));
            Cin <= '0';
            wait for 0.1 ns;
            Cin <= '1';
            wait for 0.1 ns;
          end loop;
        end loop;
        wait;
    end process;


end Testbench;
```

Αρχικά, όπως πάντα ορίζουμε ένα κενό entity, το οποίο θα αποτελέσει τον ορισμό του testbench. Στην συνέχεια στην αρχιτεκτονική ορίζουμε το entity που θέλουμε να ελέγξουμε(εν προκειμένω τον BCD Full Adder), καθώς και όλα τα ενδιάμεσα σήματα που θα μας είναι απαραίτητα. Σε αυτό το σημείο ορίζουμε και δύο ακεραίους που θα μας είναι απαραίτητοι για την προσομοίωση. Τέλος, στο κομμάτι της προσομοίωσης ορίζουμε δύο Loop(φωλιασμένα) ώστε να ελέγξουμε κάθε πιθανό συνδυασμό των δύο εισόδων. Σε κάθε for loop κάνουμε κάποιες αναθέσεις. Στις εισόδους A και B δίνω τιμή ίση με αυτή του αριθμού που γίνεται iterate κατά την διάρκεια της εκτέλεσης, ενώ για το Cin απλώς δίνω τόσο την τιμή 1 όσο και την 0 σε κάθε ανάθεση των A και B. Στο testbench λοιπόν μπορούμε να ελέγξουμε κάθε συνδυασμό εισόδων. Ακολουθούν τα αποτελέσματα του testbench:


Name
Value
Data Type
i
0
Integer
j
0
Integer
Cout
1
Logic
Cin
1
Logic
A[3...
9
Array
B[3...
9
Array
Su...
9
Array
Name
Value
i
0
j
0
Cout
0
Cin
0
A[3:0]
0
B[3:0]
0
Sum[3:0]
0
0.000 ns
0 ns
1 ns
2 ns
3 ns
4 ns
5 ns
6 ns


Name
Value
i
0
j
0
Cout
1
Cin
0
A[3:0]
7
B[3:0]
4
Sum[3:0]
1
14.820 ns
10 ns
11 ns
12 ns
13 ns
14 ns
15 ns
16

Από τα παραπάνω αποτελέσματα μπορούμε εύκολα να διαπιστώσουμε ότι η περιγραφή υλικού μας λειτουργεί ορθά ως BCD Full Adder.(ειδικά το δεκαεξαδικό εντός των κουτιών του Vivado, όπου φαίνονται οι τιμές των διάφορων σημάτων επιταχύνει σημαντικά την διαδικασία επαλήθευσης).

Ζήτημα πέμπτο BCD Parallel Adder

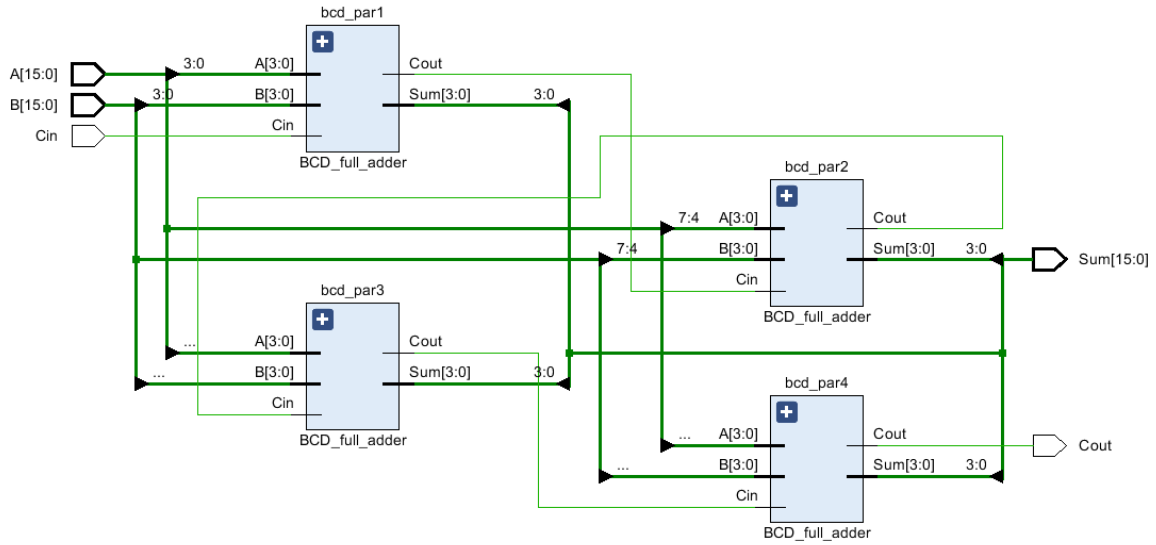
Στο ζήτημα αυτό μας ζητείται να χρησιμοποιήσουμε το component του BCD Full Adder που ορίσαμε και περιγράψαμε παραπάνω για να φτιάξουμε έναν BCD αθροιστή 4 bit. Για να υλοποιήσουμε αυτό τον αθροιστή θα δουλέψουμε με την ίδια λογική που φτιάξαμε και τον παράλληλο αθροιστή. Θα τοποθετήσουμε 4 BCD αθροιστές, κάθε ένας από τους οποίους θα αναλάβει και από ένα από τα 4 ψηφία του αριθμού που θα προκύψει σαν τελικό άθροισμα και το κρατούμενο εξόδου κάθε BCD θα πηγαίνει σαν κρατούμενο εισόδου στον BCD αθροιστή που υπολογίζει το αμέσως επόμενο σημαντικότερο ψηφίο της εισόδου(σε δεκαδικό σύστημα). Με βάση αυτή την λογική πίσω από την υλοποίηση του BCD παράλληλου αθροιστή παραθέτουμε τον ακόλουθο κώδικα:

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

entity BCD_parallel is
        Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
               B : in STD_LOGIC_VECTOR (15 downto 0);
               Cin: in std_logic ;
               Sum : out STD_LOGIC_VECTOR (15 downto 0);
               Cout : out STD_LOGIC);
end BCD_parallel;

architecture Structural of BCD_parallel is
```

```
    component BCD_full_adder is
        Port ( 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;
    signal Cout_inside : std_logic_vector (2 downto 0) ;
begin
    bcd_par1 : BCD_full_adder port map(A(3 downto 0),B(3 downto 0),Cin,Sum(3
downto 0),Cout_inside(0));
    bcd_par2 : BCD_full_adder port map(A(7 downto 4),B(7 downto
4),Cout_inside(0),Sum(7 downto 4),Cout_inside(1));
    bcd_par3 : BCD_full_adder port map(A(11 downto 8),B(11 downto
8),Cout_inside(1),Sum(11 downto 8),Cout_inside(2));
    bcd_par4 : BCD_full_adder port map(A(15 downto 12),B(15 downto
12),Cout_inside(2),Sum(15 downto 12),Cout);
end Structural;
```

Όπως βλέπουμε η δομική περιγραφή μας επιτρέπει να γράψουμε έναν αρκετά απλό κώδικα για την παραγωγή ενός σύνθετου κυκλώματος. Αρχικά στον κώδικα μας προφανώς ορίζουμε το entity, το οποίο σαν είσοδο θα έχει δύο 16-bit BCD αριθμούς (ουσιαστικά 2 αριθμούς τεσσάρων ψηφίων στο δεκαδικό σύστημα) και ένα κρατούμενο εισόδου και σαν έξοδο θα έχει αντίστοιχα ένα άθροισμα 16-bit και ένα κρατούμενο εξόδου. Στην συνέχεια παραθέτουμε τον ορισμό του component του BCD Full Adder του οποίου την υλοποίηση περιγράψαμε στο προηγούμενο ερώτημα. Τέλος, ορίζουμε ένα βοηθητικό σήμα το οποίο θα μας διευκολύνει στην διάδοση των ενδιάμεσων κρατουμένων εξόδου στην είσοδο του αμέσως επόμενου BCD(δηλαδή αυτού που υπολογίζει το αμέσως επόμενο σημαντικότερο ψηφίο, από το τρέχων, στο δεκαδικό σύστημα). Τέλος, στην αρχιτεκτονική απλώς ορίζουμε το mapping μεταξύ κάθε επι μέρους BCD Full Adder και των σημάτων που έχουμε, ώστε να πάρουμε το αποτέλεσμα που περιγράψαμε παραπάνω.

Στην συνέχεια παρουσιάζουμε το RTL σχηματικό που μας δίνει το Vivado με βάση την παραπάνω περιγραφή υλικού που παραθέσαμε:



Βλέπουμε ότι το σχηματικό είναι το αναμενόμενο, δηλαδή απλώς τέσσερις BCD Full-Adders. Όπως εξηγήσαμε πιο πάνω αναλυτικά ο κάθε ένας δίνει στο SUM την τετράδα bit, που αντιστοιχεί σε ένα ψηφίο του δεκαδικού αριθμού. Είναι πολύ σημαντικό , που το αποτέλεσμα του RTL σχηματικού είναι το αναμενόμενο, καθώς αυτό δείχνει πως ο VHDL κώδικας που γράψαμε αντιστοιχίζεται πλήρως σε αυτό που είχαμε σαν ιδέα για το υλικό. Επιπλέον, θα παραθέσουμε και το κρίσιμο μονοπάτι εκμεταλλευόμενοι την δυνατότητα που δίνει το Vivado για εύρεση της συνολικής καθυστέρησης κάθε μονοπατιού από την είσοδο στην έξοδο. Παραθέτουμε το αποτέλεσμα για αυτή την υλοποίηση:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	∞	10	11	7	Cin	Cout	9.517	4.620	4.897	∞	input port clock		
Path 2	∞	10	11	7	Cin	Sum[14]	9.517	4.620	4.897	∞	input port clock		
Path 3	∞	10	11	7	Cin	Sum[13]	9.511	4.614	4.897	∞	input port clock		
Path 4	∞	10	11	7	Cin	Sum[15]	9.511	4.614	4.897	∞	input port clock		
Path 5	∞	9	10	7	Cin	Sum[10]	8.920	4.496	4.424	∞	input port clock		
Path 6	∞	9	10	7	Cin	Sum[12]	8.920	4.496	4.424	∞	input port clock		

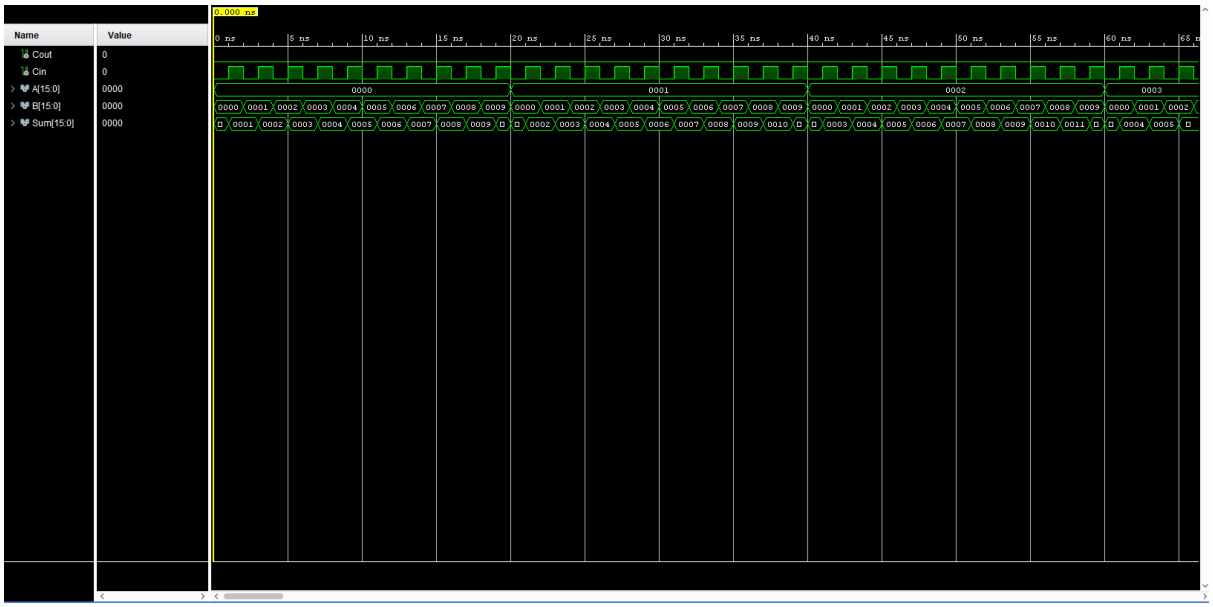
Από το παραπάνω διάγραμμα παρατηρούμε το αναμενόμενο, ότι δηλαδή το κρατούμενο εξόδου και τα 4 τελευταία Bits έχουν την μέγιστη συνολική καθυστέρηση, γεγονός αναμενόμενο, αφού για τον υπολογισμό των τεσσάρων σημαντικότερων bit πρέπει να πάρει το τρέχων στάδιο να πάρει το κρατούμενο εξόδου από το αμέσως προηγούμενο Άρα για τον υπολογισμό του τελευταίο σταδίου αναδρομικά βλέπουμε ότι απαιτείται να έχουν υπολογιστεί τα αποτελέσματα όλων των προηγούμενων . Για αυτό τον λόγο το μονοπάτι του Cout και του SUM(15) αποτελούν δύο critical paths.

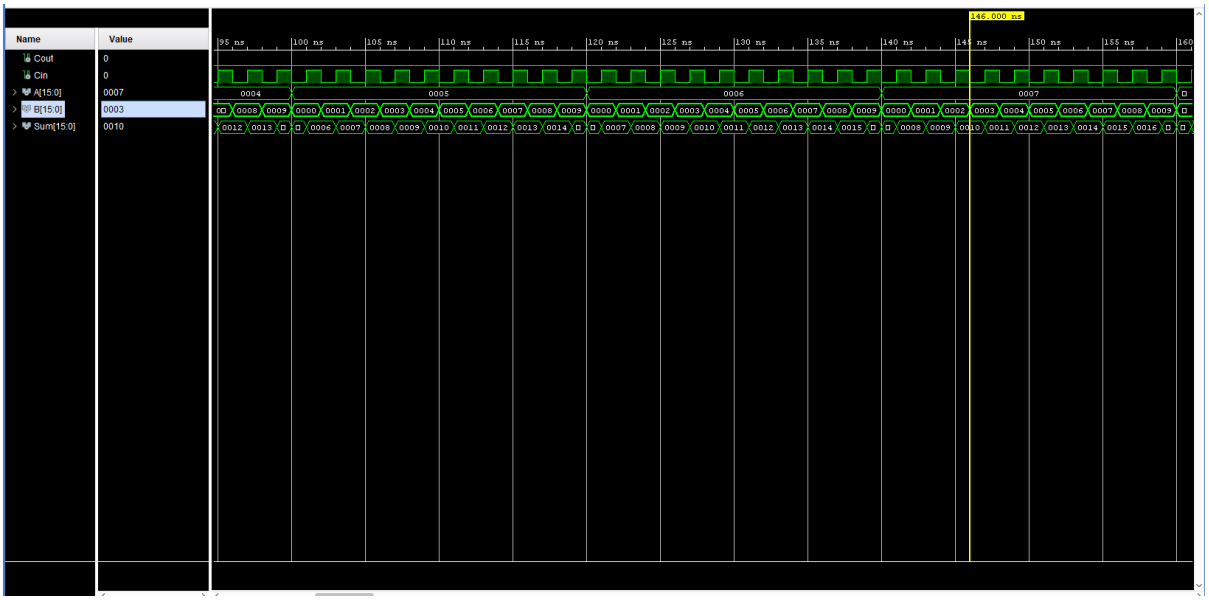
Τέλος για να ελέγξουμε την λειτουργία του κυκλώματος μας γράψαμε ένα testbench, που θα ελέγχει κάθε συνδυασμό εισόδων. Παραθέτουμε ακολούθως τον κώδικα για το testbench:

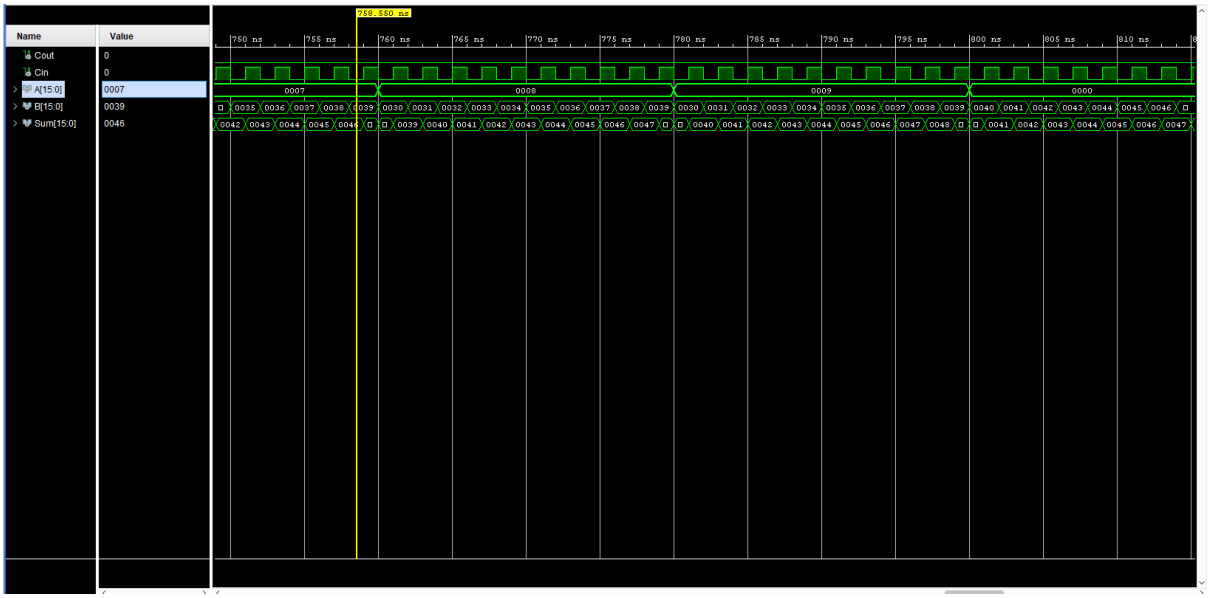
```
library IEEE;

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

entity tb_bcd_parallel is
end tb_bcd_parallel;

architecture Testbench of tb_bcd_parallel is
    component BCD_parallel is
```

```
        Port (
            A : in STD_LOGIC_VECTOR (15 downto 0);
            B : in STD_LOGIC_VECTOR (15 downto 0);
            Cin: in std_logic;
            Sum : out STD_LOGIC_VECTOR (15 downto 0);
            Cout : out STD_LOGIC
        );
    end component;

    signal Cout, Cin : std_logic;
    signal A, B, Sum : std_logic_vector (15 downto 0);
begin
    uut: BCD_parallel port map(A, B, Cin, Sum, Cout);

    stim_process: process
    begin
        for thousands_A in 0 to 9 loop
            for thousands_B in 0 to 9 loop
                for hundreds_A in 0 to 9 loop
                    for hundreds_B in 0 to 9 loop
                        for decades_A in 0 to 9 loop
                            for decades_B in 0 to 9 loop
                                for units_A in 0 to 9 loop
                                    for units_B in 0 to 9 loop
                                        A <= std_logic_vector(to_unsigned(thousands_A, 4)) &
                                             std_logic_vector(to_unsigned(hundreds_A, 4)) &
                                             std_logic_vector(to_unsigned(decades_A, 4)) &
                                             std_logic_vector(to_unsigned(units_A, 4));
                                        B <= std_logic_vector(to_unsigned(thousands_B, 4)) &
                                             std_logic_vector(to_unsigned(hundreds_B, 4)) &
                                             std_logic_vector(to_unsigned(decades_B, 4)) &
                                             std_logic_vector(to_unsigned(units_B, 4));
                                        Cin <= '0';
                                        wait for 1 ns;
                                        Cin <= '1';
                                        wait for 1 ns;
                                    end loop;
                                end loop;
                            end loop;
                        end loop;
                    end loop;
                end loop;
            end loop;
        end loop;
        wait;
    end process;

end Testbench;
```

Η λογική είναι ίδια με το προηγούμενο ερώτημα. Αρχικά φτιάχνουμε ένα entity για το testbench. Στην συνέχεια εισάγουμε το component που θέλουμε να ελέγξουμε (εν προκειμένω τον BCD Parallel Adder). Αμέσως μετά ορίζουμε τα A και B, μέσω ενός for loop όπου αναθέτουμε για κάθε ένα ψηφίο του αντίστοιχου τετραψήφιου δεκαδικού αροθμού κάθε πιθανό ψηφίο από το 0-9. Τέλος, παρουσιάζουμε τα αποτελέσματα του testbench(θα παραθέσουμε ενδεικτικά, τυχαία επιλεγμένες , κάποιες τιμές για τις εισόδους και τις αντίστοιχες εξόδους που προκύπτουν εφαρμόζοντας την υλοποίηση μας , καθώς η παρουσίαση όλων των αποτελεσμάτων κρίνεται εξαντλητική για έναν αναγνώστη). Συνεπώς, παρουσιάζουμε τα αποτελέσματα που περιγράψαμε ακολούθως:







Από τα παραπάνω διαγράμματα μπορούμε σε κάθε περίπτωση εύκολα να διαπιστώσουμε και εποπτικά την ορθή λειτουργία του BCD Parallel Adder που φτιάξαμε μέσω της περιγραφής υλικού (που παρουσιάσαμε παραπάνω).