Project 2: Μεθοδολογία Σχεδίασης Επεξεργαστή Πολλών Κύκλων

Άγγελος Ψημίτης – Χριστοδουλόπουλος ΑΜ : 2019-513

Σχεδίαση Ψηφιακών Συστημάτων Μάρτιος 2021

ΔΠΜΣ Ηλεκτρονικός Αυτοματισμός

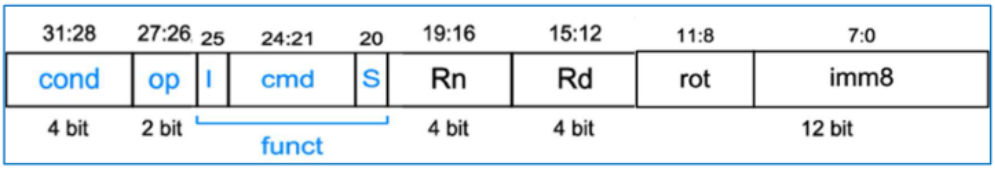
1. Περιγραφή των στοιχείων και της δομής του επεξεργαστή

1.1 Εντολές της αρχιτεκτονικής ARM που έχουν υλοποιηθεί στην παρούσα εργασία

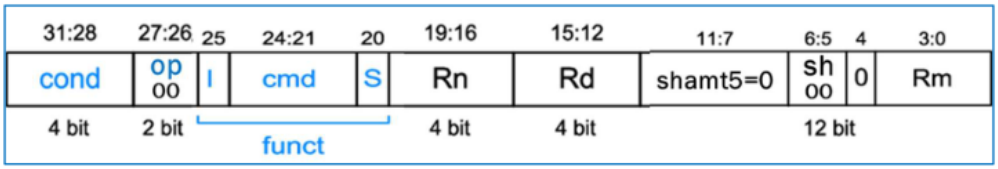
Εντολές επεξεργασίας δεδομένων (DP instructions) :

ADD(S), SUB(S), CMP, AND(S), XOR(S), MOV, MOVN –S =0,
LSR, ASR –S =0.

Οι DP εντολές κωδικοποιούνται σε λέξεις των 32 bit σύμφωνα με τις επόμενες δύο εικόνες.



Εντολές επεξεργασίας δεδομένων με άμεση διευθυνσιοδότηση ALU(S) –I



Εντολές επεξεργασίας δεδομένων με διευθυνσιοδότηση καταχωρητή ALU(S) –R

Εντολές Μνήμης (Memory Instructions) :

LDR, STR

Οι εντολές μνήμης κωδικοποιούνται σε λέξεις των 32 bit όπως φαίνεται στην επόμενη εικόνα.

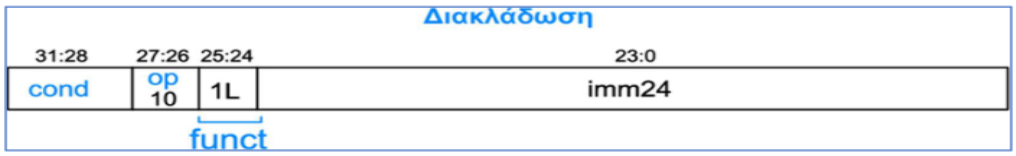
Εντολές Μνήμης , ο μη προσημασμένος άμεσος τελεστέος των 12 bit (imm12) χρησιμοποιείται ως offset με επέκταση μηδενός στα 32 bit.

LDR Rd, [Rn, #imm12]; Rd = DM[Rn + #imm12] (U=1)
LDR Rd, [Rn, #-imm12]; Rd = DM[Rn - #imm12] (U=0)

STR Rd, [Rn, #imm12]; DM[Rn + #imm12] = Rd (U=1)
STR Rd, [Rn, #-imm12]; DM[Rn - #imm12] = Rd (U=0)

Εντολές Διακλάδωσης (Branch Instrcutions):

B (Branch), BL (Branch Link)



Κωδικοποίηση εντολής διακλάδωσης στα 32 bit.

B label; PC = BTA = PC + 8 + imm24 x 4 (L = 0)
BL label; PC = BTA = PC + 8 + imm24 x 4; R14 = LR = PC+4 (L = 1)

1.2 Περιγραφή των κύριων ψηφιακών δομικών στοιχείων (components) της διαδρομής δεδομένων του επεξεργαστή.

Αρχείο Καταχωρητών (Register File)

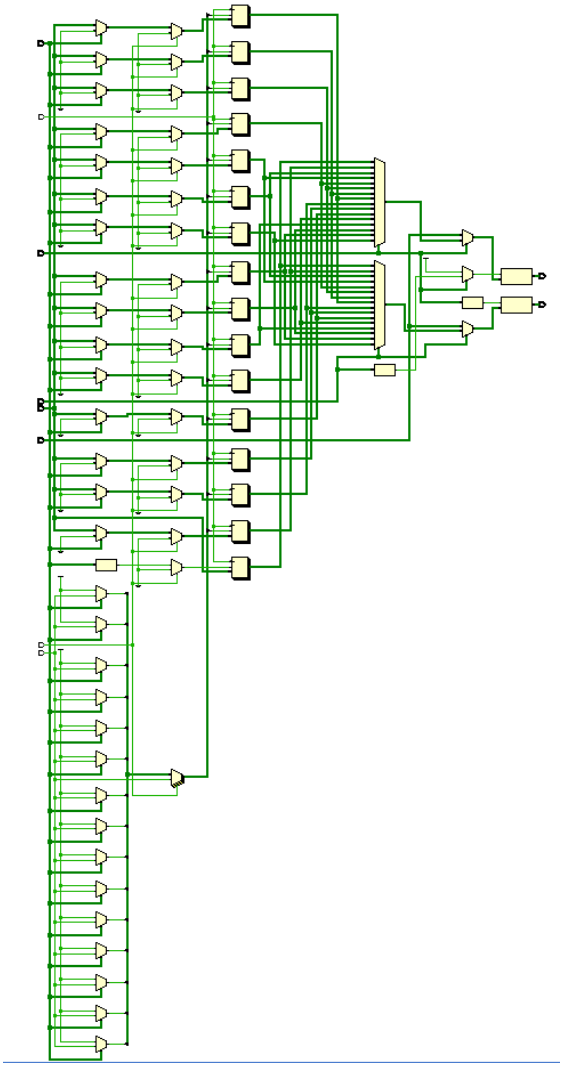
Το αρχείο καταχωρητών είναι μια διάταξη μνήμης RAM με διευθύνσεις των 4 bit για 16 καταχωρητές και δεδομένα (data) των 32 bit.
Διαθέτει σύγχρονη εγγραφή με έγκριση (όταν το σήμα WE = 1), ασύγχρονο διάβασμα, ενώ παρέχει τη δυνατότητα διαβάσματος δύο καταχωρητών ταυτόχρονα. Το RegFile module έχει σχεδιαστεί ως ακολουθιακό κύκλωμα το οποίο συγχρονίζεται με ένα εξωτερικό CLK σήμα καθώς κι ένα σύγχρονο RESET.

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

entity REG_FILE is
Generic( N : positive := 4; -- address length
         M : positive := 32); -- data word length
Port (
    CLK     :   in  std_logic;
    RESET   :   in  std_logic;
    WE3     :   in  std_logic;
    ADDR_A1 :   in  std_logic_vector(N-1 downto 0);
    ADDR_A2 :   in  std_logic_vector(N-1 downto 0);
    ADDR_A3 :   in  std_logic_vector(N-1 downto 0);
    WD3     :   in  std_logic_vector(M-1 downto 0);
    R15     :   in  std_logic_vector(M-1 downto 0);
    RD1     :   out std_logic_vector(M-1 downto 0);
    RD2     :   out std_logic_vector(M-1 downto 0)
    );
end REG_FILE;
```

Register File Entity

```
architecture Behavioral of REG_FILE is
    type RF_array is array (2**N-1 downto 0)
        of std_logic_vector(M-1 downto 0);
    signal RF : RF_array;
begin
    REG_FILE: process(clk)
    begin
        if(falling_edge(CLK))then
            if(RESET = '1')then
                for I in 0 to 2**N-2 loop
                    RF(I) <= (others => '0');
                end loop;
            end if;
            if(WE3 = '1')then
                RF(to_integer(unsigned(ADDR_A3))) <= WD3;
            end if;
        end if;
     end process;
     OUTPUT_PROC: process(ADDR_A1,ADDR_A2, R15)
     begin
        if(ADDR_A1 = "1111")then
            RD1 <= R15;
        elsif(ADDR_A2 = "1111")then
            RD2 <= R15;
        else
            RD1 <= RF(to_integer(unsigned(ADDR_A1)));
            RD2 <= RF(to_integer(unsigned(ADDR_A2)));
        end if;
     end process;
end Behavioral;
```

Register File Architecture



Register File (RTL – Elaborated Design)

Αριθμητική και Λογική Μονάδα (Arithmetic Logic Unit – ALU)

Πρόκειται για τη συνδυαστική λογική μονάδα που εκτελεί πράξεις σύμφωνα με την τιμή του σήματος ALUControl.

ALUControl	Operation	function	Unit
0000	ALUResult <= A +B	Add A and B	Arithmetic
0001	ALUResult <= A -B	Subtract A and B	Arithmetic
0010	ALUResult <= shiftleft(B)	Logical Shift Left B	Arithmetic
0011	ALUResult <= shiftright(B)	Logical Shift Right B	Arithmetic
0100	ALUResult <= shiftright(B)	Arithmetic Shift Right B	Arithmetic
0101	ALUResult <= ror(B)	Rotate B	Arithmetic
0110	ALUResult <= A	Transfer A	Arithmetic
0111	ALUResult <= B	Transfer B	Arithmetic
1000	ALUResult <= not(A)	Complement A	Logic
1001	ALUResult <= not(B)	Complement B	Logic
1010	ALUResult <= A and B	AND	Logic
1011	ALUResult <= A or B	OR	Logic
1100	ALUResult <= A nand B	NAND	Logic
1101	ALUResult <= A nor B	NOR	Logic
1110	ALUResult <= A xor B	XOR	Logic
1111	ALUResult <= not(A xor B)	XNOR	Logic

Πίνακας 1: Πίνακας αλήθειας για τη λειτουργία της ALU

Η υλοποίηση της ALU είναι structural και περιλαμβάνει 3 submodules:

Arithmetic Unit, Logic Unit κι έναν 2x1 Multiplexer για την επιλογή στην έξοδο της εξόδου της μιας ή της άλλης μονάδας ανάλογα με την τιμή του ALUControl(3) bit.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity ALU is
6  Generic( Width : positive := 32);
7  Port (
8      CLK        :  in  std_logic;
9      SrcA       :  in  std_logic_vector(Width-1 downto 0);
10     SrcB       :  in  std_logic_vector(Width-1 downto 0);
11     ALUControl :  in  std_logic_vector(3 downto 0);
12     ALUResult  :  out std_logic_vector(Width-1 downto 0);
13     Flags      :  out std_logic_vector(3 downto 0)  -- N,Z,C,V
14     );
15 end ALU;
```

ALU Entity

```
16
17 architecture Structural of ALU is
18
19 -- submodules declaration
20
21     component Logic_Unit
22     Generic (Width : positive := 32);
23     Port (
24     A   :  in  std_logic_vector(Width-1 downto 0);
25     B   :  in  std_logic_vector(Width-1 downto 0);
26     Sel :  in  std_logic_vector(2 downto 0);
27     S   :  out std_logic_vector(Width-1 downto 0)
28     );
29     end component;
30
31     component Arithmetic_Unit
32     Generic (Width : positive := 32);
33     Port (
34     A       :   in  std_logic_vector(Width-1 downto 0);
35     B       :   in  std_logic_vector(Width-1 downto 0);
36     Sel     :   in  std_logic_vector(2 downto 0);
37     shamt   :   in  std_logic_vector(4 downto 0);
38     S       :   out std_logic_vector(Width-1 downto 0);
39     Cout    :   out std_logic;
40     OV      :   out std_logic;
41     N       :   out std_logic;
42     Z       :   out std_logic
43     );
44     end component;
45
46     component MUX2x1
47     Generic (Width : positive := 32);
48     Port (
49     A0     :  in  std_logic_vector(width-1 downto 0);
50     A1     :  in  std_logic_vector(width-1 downto 0);
51     SEL    :  in  std_logic;
52     Y      :  out std_logic_vector(width-1 downto 0)
53     );
54     end component;
55
```

```
55
56     signal logic_out_sig      : std_logic_vector(Width-1 downto 0) := (others => '0');
57     signal arithmetic_out_sig : std_logic_vector(Width-1 downto 0) := (others => '0');
58     signal mux_out            : std_logic_vector(Width-1 downto 0) := (others => '0');
59
60 begin
61
62 --logic unit instantiation
63     logic_inst: Logic_Unit
64     port map(
65         A   => SrcA,
66         B   => SrcB,
67         Sel => ALUControl(2 downto 0),
68         S   => logic_out_sig
69      );
70
71 --arithmetic unit instantiation
72     arithmetic_inst: Arithmetic_Unit
73     port map(
74         A     => SrcA,
75         B     => SrcB,
76         Sel   => ALUControl(2 downto 0),
77         shamt => SrcB(11 downto 7),
78         S     => arithmetic_out_sig,
79         Cout  => Flags(2),
80         OV    => Flags(3),
81         N     => Flags(0),
82         Z     => Flags(1)
83         );
84
85 --multiplexer instantiation
86     mux_inst: MUX2x1
87     port map(
88         A0  => arithmetic_out_sig,
89         A1  => logic_out_sig,
90         SEL => ALUControl(3),
91         Y   => mux_out
92      );
93
94     ALUResult <= mux_out;
95 end Structural;
```

ALU Architecture

Arithmetic Unit

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

entity Arithmetic_Unit is
Generic (Width : positive := 32);
Port (
    A       :   in  std_logic_vector(Width-1 downto 0);
    B       :   in  std_logic_vector(Width-1 downto 0);
    Sel     :   in  std_logic_vector(2 downto 0);
    shamt   :   in  std_logic_vector(4 downto 0);
    S       :   out std_logic_vector(Width-1 downto 0);
    Cout    :   out std_logic;
    OV      :   out std_logic;
    N       :   out std_logic;
    Z       :   out std_logic
    );
end Arithmetic_Unit;

architecture Behavioral of Arithmetic_Unit is
signal S_sig      : std_logic_vector(Width-1 downto 0):= (others => '0');
signal S_sig_temp : std_logic_vector(Width-1 downto 0) := (others => '0');
begin

    Arithm_Proc: process(A,B,Sel)
    variable A_s, B_s, S_s : signed (Width+1 downto 0);
    variable shamt_n : natural range 0 to 31;
    variable X_u : unsigned (Width-1 downto 0);
    Variable X_s : signed (Width-1 downto 0);
    begin
        A_s := signed('0'&A(Width-1)&A);
        B_s := signed('0'&B(Width-1)&B);
        shamt_n := to_integer(unsigned(shamt));
        X_u := unsigned(B);
        X_s := signed(B);
        case Sel is
            when "000" =>
                    S_s := A_s + B_s;
                    S_sig      <= std_logic_vector(S_s(Width-1 downto 0));
                    Ov         <= S_s(Width) xor S_s(Width-1);
                    Cout       <= S_s(Width+1);
            when "001" =>
                    S_s := A_s - B_s;
                    S_sig      <= std_logic_vector(S_s(Width-1 downto 0));
                    OV         <= S_s(Width) xor S_s(Width-1);
                    Cout       <= S_s(Width+1);
            when "010"  => S_sig <= std_logic_vector(SHIFT_LEFT (X_u, shamt_n));
            when "011"  => S_sig <= std_logic_vector(SHIFT_RIGHT (X_u, shamt_n));
            when "100"  => S_sig <= std_logic_vector(SHIFT_RIGHT (X_s, shamt_n));
            when "101"  => S_sig <= std_logic_vector(ROTATE_RIGHT (X_s, shamt_n));
            when "110"  => S_sig <= A;
            when "111"  => S_sig <= B;
            when others => S_sig <= (others => '0');
        end case;
        N <= S_sig(Width-1) or '0';
        S_sig_temp <= S_sig nor x"00000000";
        if(S_sig_temp = x"11111111")then
            Z <= '1';
        else
            Z <= '0';
        end if;

    end process;

     S <= S_sig;

end Behavioral;
```

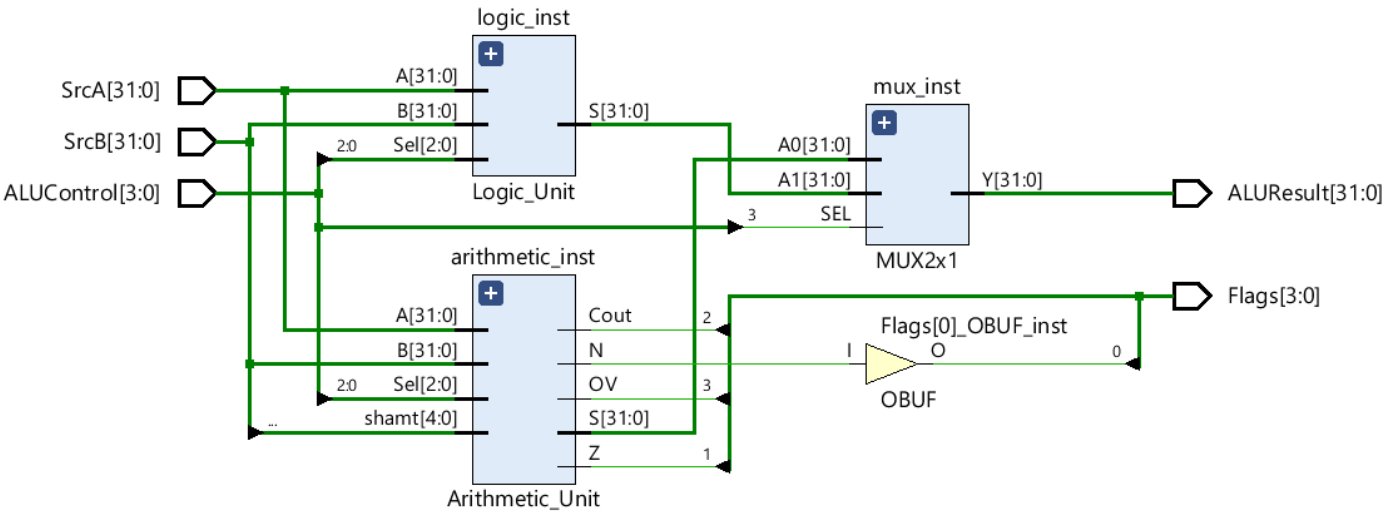
Arithmetic Unit entity and architecture

Για την υλοποίηση της πρόσθεσης/αφαίρεσης έχει χρησιμοποιηθεί προσημασμένος αθροιστής/αφαιρέτης (των 32 bit) με δημιουργία κρατουμένου (Cout) και σήματος υπερχείλισης (Ov). Όλα τα σήματα των σημαιών (N,Z,C,V) υπολογίζονται στην αριθμητική μονάδα προκειμένου να μπορεί η μονάδα ελέγχου να εκτελεί τις εντολές υπό συνθήκη.

Logic Unit

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Logic_Unit is
5  Generic (Width : positive := 32);
6  Port (
7      A   :  in  std_logic_vector(Width-1 downto 0);
8      B   :  in  std_logic_vector(Width-1 downto 0);
9      Sel :  in  std_logic_vector(2 downto 0);
10     S   :  out std_logic_vector(Width-1 downto 0)
11     );
12 end Logic_Unit;
13
14 architecture Behavioral of Logic_Unit is
15 signal S_sig : std_logic_vector(Width-1 downto 0) := (others => '0');
16 begin
17    Logic_Unit : process(A,B,Sel)
18    begin
19        case Sel is
20         when "000"  => S_sig <= not(A);
21         when "001"  => S_sig <= not(B);
22         when "010"  => S_sig <= A and B;
23         when "011"  => S_sig <= A or B;
24         when "100"  => S_sig <= A nand B;
25         when "101"  => S_sig <= A nor B;
26         when "110"  => S_sig <= A xor B;
27         when others => S_sig <= not(A xor B);
28       end case;
29    end process;
30 S <= S_sig;
31 end Behavioral;
```

Logic Unit entity and architecture



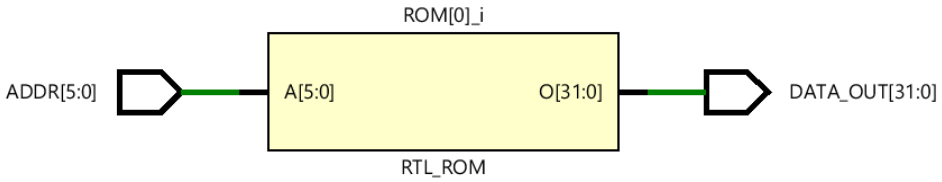
ALU RTL Elaborated Design

Μνήμη Εντολών (Instruction Memory)

Η μνήμη εντολών είναι μια διάταξη μνήμης ROM με διευθύνσεις (address) των 6 bit και δεδομένα (data) των 32 bit, επομένως αποθηκεύει 2^6 λέξεις μεγέθους 32 bit. Σε αυτήν αποθηκεύονται τα instructions του προγράμματος που είναι να εκτελεστεί.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity ROM is
6  Generic( N : positive := 6; -- address length
7           M : positive := 32); -- data word length
8  Port (
9      ADDR     :   in  std_logic_vector(N-1 downto 0);
10     DATA_OUT :   out std_logic_vector(M-1 downto 0));
11 end ROM;
12
13 architecture Behavioral of ROM is
14     type ROM_array is array (0 to 2**N-1)
15       of std_logic_vector(M-1 downto 0);
16     constant ROM : ROM_array := (
17       X"E3A00000", X"E3E01000", X"E0812000", X"E24230FF",
18       X"E1A00000", X"EAFFFFF9", X"00000000", X"00000000",
19       X"00000000", X"00000000", X"00000000", X"00000000",
20       X"00000000", X"00000000", X"00000000", X"00000000",
21       X"00000000", X"00000000", X"00000000", X"00000000",
22       X"00000000", X"00000000", X"00000000", X"00000000",
23       X"00000000", X"00000000", X"00000000", X"00000000",
24       X"00000000", X"00000000", X"00000000", X"00000000",
25       X"00000000", X"00000000", X"00000000", X"00000000",
26       X"00000000", X"00000000", X"00000000", X"00000000",
27       X"00000000", X"00000000", X"00000000", X"00000000",
28       X"00000000", X"00000000", X"00000000", X"00000000",
29       X"00000000", X"00000000", X"00000000", X"00000000",
30       X"00000000", X"00000000", X"00000000", X"00000000",
31       X"00000000", X"00000000", X"00000000", X"00000000",
32       X"00000000", X"00000000", X"00000000", X"00000000");
33 begin
34     DATA_OUT <= ROM(to_integer(unsigned(ADDR)));
35 end Behavioral;
```

Instruction Memory (ROM) με αποθηκευμένες τιμές τις εντολές assembly του παραδείγματος που μας δόθηκε.

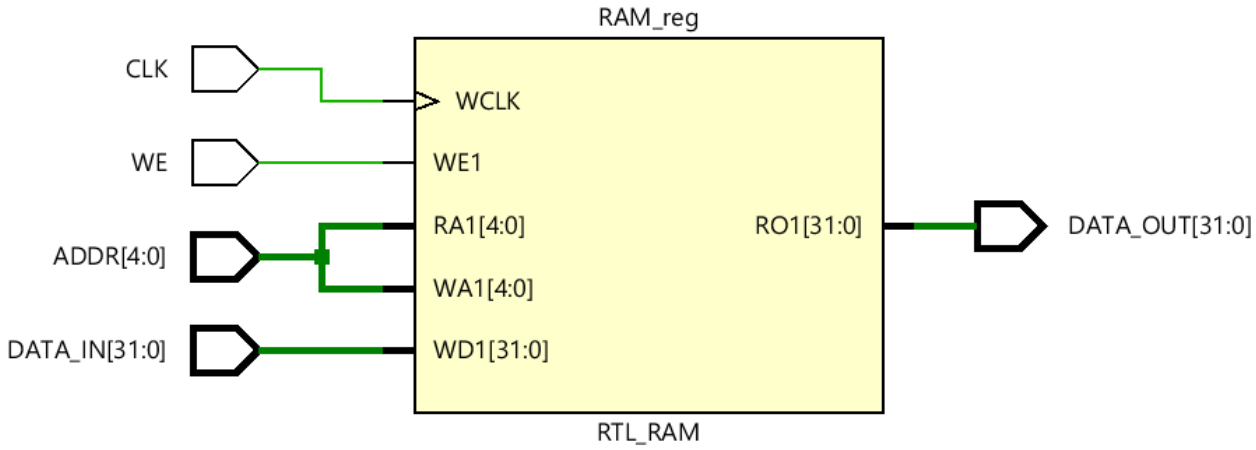


Instruction Memory RTL

Μνήμη Δεδομένων (Data Memory)

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

entity Data_Memory is
Generic ( N : positive := 5;  -- Address length
          M : positive := 32); -- Data word length
Port (
    CLK      :   in  std_logic;
    WE       :   in  std_logic;
    ADDR     :   in  std_logic_vector(N-1 downto 0);
    DATA_IN  :   in  std_logic_vector(M-1 downto 0);
    DATA_OUT :   out std_logic_vector(M-1 downto 0)
  );
end Data_Memory;

architecture Behavioral of Data_Memory is
    type RAM_array is array (2**N-1 downto 0)
        of std_logic_vector(M-1 downto 0);
    signal RAM : RAM_array;
begin
    Block_RAM: process(CLK)
    begin
        if(falling_edge(CLK))then
            if(WE = '1') then
                RAM(to_integer(unsigned(ADDR))) <= DATA_IN;
            end if;
        end if;
    end process;
    DATA_OUT <= RAM(to_integer(unsigned(ADDR)));
end Behavioral;
```

Η υλοποίηση έγινε με διάταξη μνήμης RAM χωρίς καταχωρητή εξόδου (Distributed RAM). Η διάταξη περιέχει σύγχρονη εγγραφή και ασύγχρονο διάβασμα.



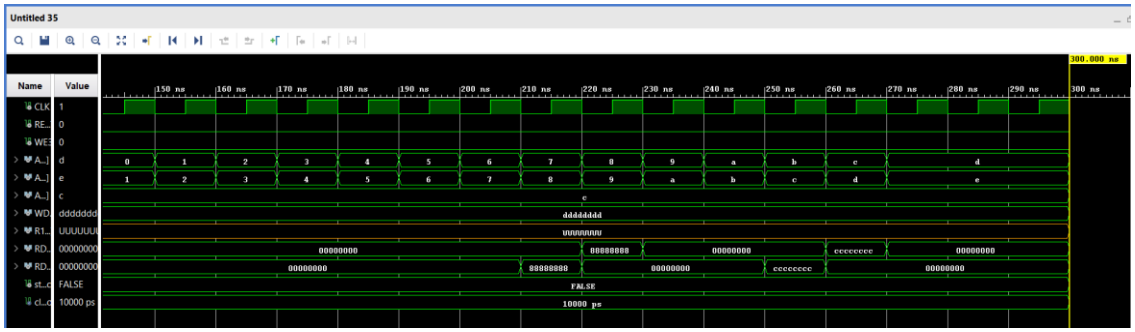
Data Memory RTL

1.3 Προγράμματα Δοκιμής – Προσομοιώσεις

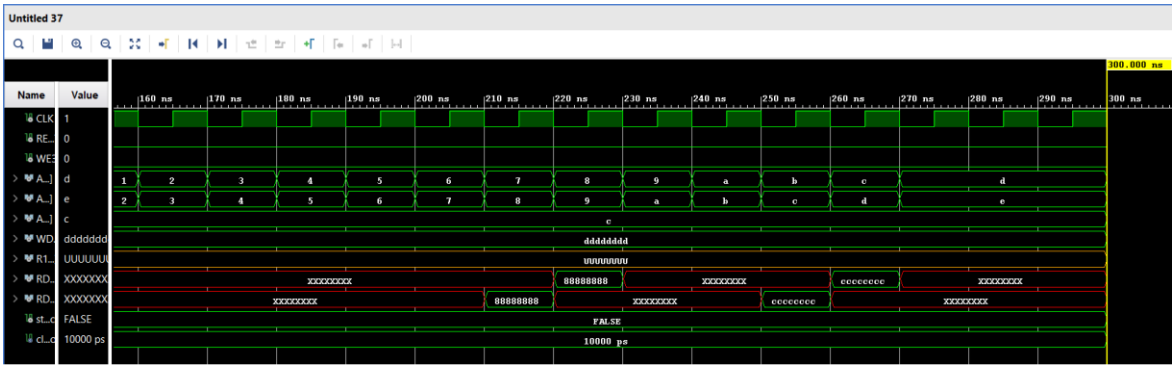
Register File Testbench

```
57      stimulus: process
58      begin
59       RESET <= '1';
60       wait for 100 ns;
61       wait until (CLK = '0' and CLK'event);
62       RESET <= '0';
63       WE3 <= '1';
64       wait for 1*clock_period;
65       ADDR_A3 <= "1000";    -- write register 8
66       WD3 <= x"88888888";
67       wait for 1*clock_period;
68       ADDR_A3 <= "1100";    -- write register 12
69       WD3 <= x"CCCCCCCC";
70       wait for 1*clock_period;
71       WE3 <= '0';
72       ADDR_A3 <= "1100";
73       WD3 <= x"DDDDDDDD"; -- should not write register 12 because WE3 = 0
74       wait for 1*clock_period;
75      -- Test Reading of all registers except R15
76       for I in 0 to 13 loop
77           ADDR_A1 <= std_logic_vector(to_unsigned(I,4));
78           ADDR_A2 <= std_logic_vector(to_unsigned(I+1,4));
79           wait for 1*clock_period;
80       end loop;
81       wait for 2*clock_period;
82       -- Message and simulation end
83       report "TESTS COMPLETED";
84       stop(2);
85       end process;
```

Στο stimulus process του TB γράφουμε τον R8, στη συνέχεια τον R12, ενώ προσπαθούμε να ξαναγράψουμε τον R12 με το σήμα WE κλειστό. Τελικά διαβάζουμε τις τιμές όλων των registers εκτός του R15 στον οποίο δεν έχουμε δώσει τιμή.

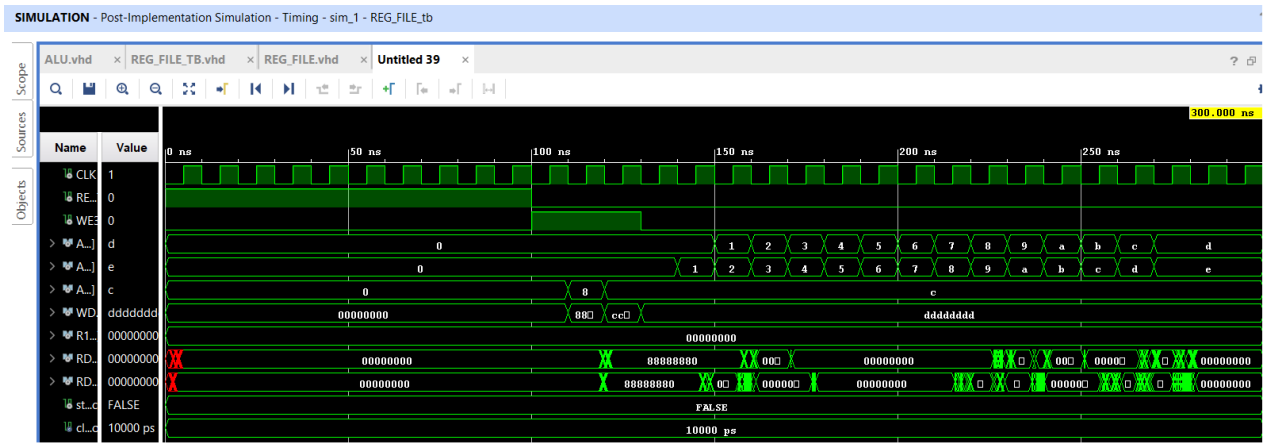


Behavioral Simulation του Register File.



Post Synthesis Simulation του Register File.

Παρατηρούμε ότι το Behavioral Simulation δίνει τα αποτελέσματα που περιμέναμε σύμφωνα με το testbench που δημιουργήσαμε. Όταν διαβάζουμε τους Registers, ο R8 έχει την τιμή x"88888888" ενώ ο R12 την τιμή x"CCCCCCCC", επιβεβαιώνοντας ότι οι καταχωρητές δεν γράφονται όταν το σήμα WE = '0'. Το post synthesis functional simulation ταυτίζεται με το behavioral model επιβεβαιώνοντας την σχεδίαση.



Post Implementation Timing Simulation for RegFile

ALU Testbench

```
54      stimulus: process
55      begin
56        wait for 20 ns;
57        SrcA  <= x"A0000004";
58        SrcB  <= x"20000208";
59        wait for 1*CLK_period;
60        ALUControl <= "0000";
61        wait for 1*CLK_period;
62        ALUControl <= "0001";
63        wait for 1*CLK_period;
64        ALUControl <= "0010";
65        wait for 1*CLK_period;
66        ALUControl <= "0011";
67        wait for 1*CLK_period;
68        ALUControl <= "0100";
69        wait for 1*CLK_period;
70        ALUControl <= "0101";
71        wait for 1*CLK_period;
72        ALUControl <= "0110";
73        wait for 1*CLK_period;
74        ALUControl <= "0111";
75        wait for 1*CLK_period;
76        ALUControl <= "1000";
77        wait for 1*CLK_period;
78        ALUControl <= "1001";
79        wait for 1*CLK_period;
80        ALUControl <= "1010";
81        wait for 1*CLK_period;
82        ALUControl <= "1011";
83        wait for 1*CLK_period;
84        ALUControl <= "1100";
85        wait for 1*CLK_period;
86        ALUControl <= "1101";
87        wait for 1*CLK_period;
88        ALUControl <= "1110";
89        wait for 1*CLK_period;
90        ALUControl <= "1110";
91        wait for 1*CLK_period;
92        ALUControl <= "1111";
93        wait for 2*CLK_period;
```

Αριστερά φαίνεται το stimulus process που υλοποιήθηκε στο testbench της ALU. Εκτελούμε όλες τις αριθμητικές και λογικές πράξεις του *πίνακα 1* μεταξύ των αριθμών x"A0000004" και x"20000208" δίνοντας τιμές στο σήμα ALUControl από το "0000" έως το "1111" αυξάνοντας κάθε φορά κατά 1. Όπως φαίνεται και από το behavioral simulation της ALU, οι τιμές στην έξοδο είναι οι αναμενόμενες ενώ οι σημαίες ανταποκρίνονται στην λογική με την οποία έχει σχεδιαστεί η μονάδα.

Συγκεκριμένα το σήμα Flags[3:0] κωδικοποιείται ως :

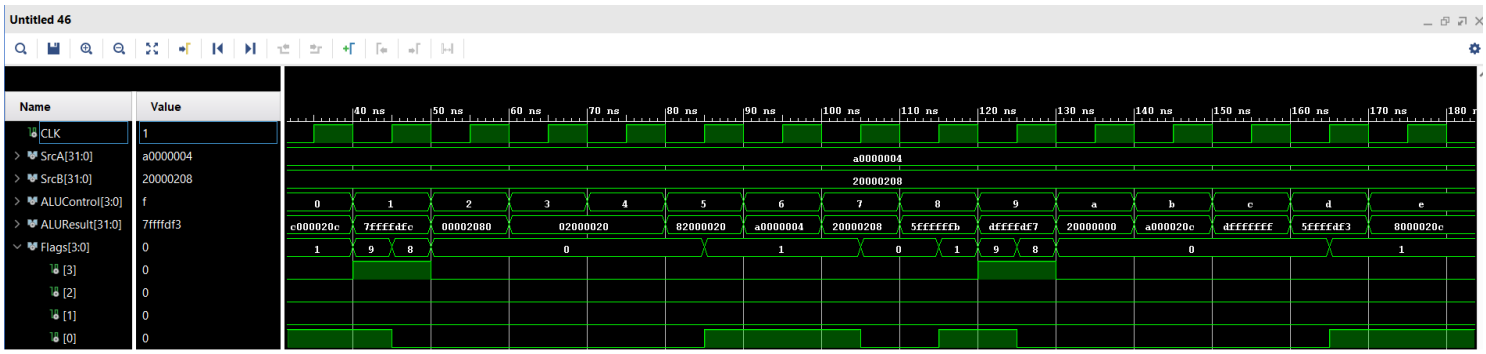
Flags(0) = N (Negative)

Flags(1) = Z (Zero)

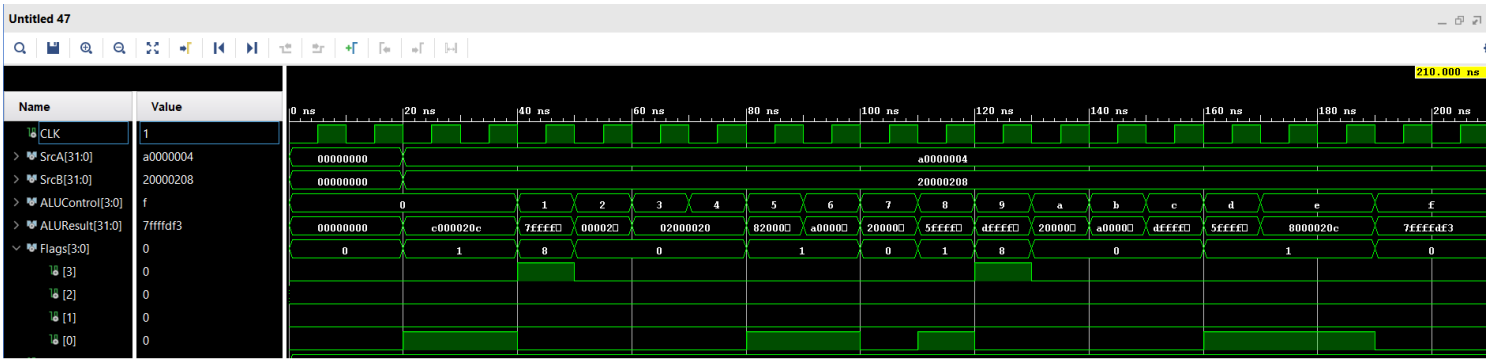
Flags(2) = C (Cout)

Flags(3) = V (Overflow)

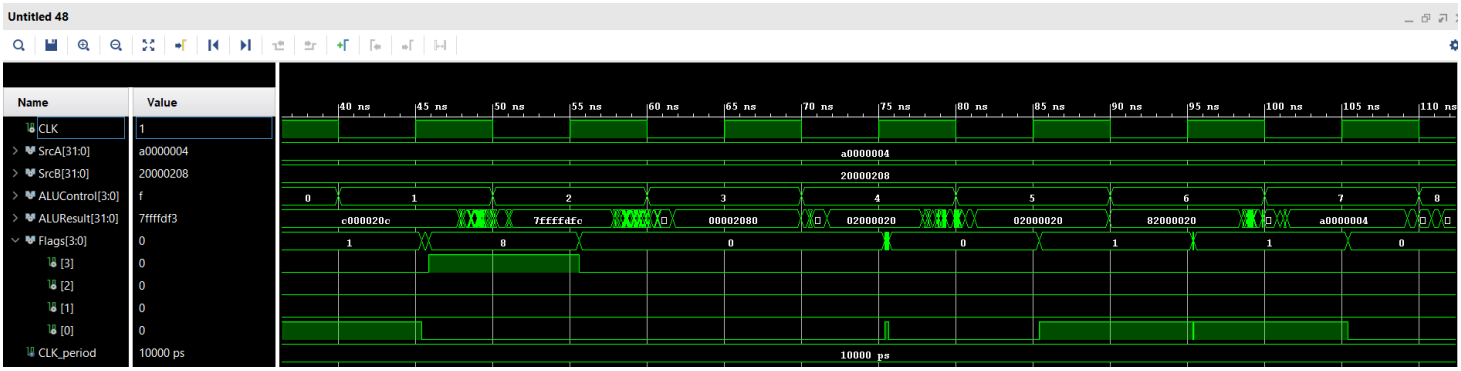
Παρατηρούμε για παράδειγμα ότι όταν προστίθενται οι 2 αριθμοί (ALUControl = x'0') το N παίρνει την τιμή 1 καθώς το αποτέλεσμα είναι αρνητικός αριθμός, αντίστοιχα όταν οι αριθμοί αφαιρούνται (ALUControl = x'1'), το V παίρνει την τιμή 1 ορθώς καθώς ο αριθμός που προκύπτει έχει μέτρο που ξεπερνάει την κωδικοποίηση των 32 bit. Όλα τα υπόλοιπα σήματα εξόδου φαίνεται να λειτουργούν σωστά.



ALU Behavioral Simulation



ALU Post Synthesis Functional Simulation



ALU Post Implementation Timing Simulation

1.4 Περιγραφή της διαδρομής δεδομένων του επεξεργαστή (Data_Path)

Στην ενότητα αυτή παρουσιάζεται η δομή του DataPath, μέσω της οποίας πραγματοποιούνται όλα τα βήματα εκτέλεσης της εντολής. Η δομή έχει υλοποιηθεί σε ανώτερο ιεραρχικό επίπεδο (structural). Το DataPath αποτελείται από τα εξής components :

Instruction Memory (ROM)
Register File (RAM)
ALU
Data Memory (RAM)
Incrementer By 4
Write Enable Register (με σύγχρονο RESET)
2x1Multiplexer (Sel[0:0])
3x1Multiplexer (Sel[1:0])
Extend

Χρησιμοποιώντας τα παραπάνω εννέα components κάνουμε instantiate όλες τις απαραίτητες υπομονάδες για την ορθή λειτουργία της διαδρομής δεδομένων.

Συγκεκριμένα, ο WERegister χρησιμοποιείται για την υλοποίηση των αρχιτεκτονικών καταχωρητών (Program Counter, Status Register), καθώς και για την υλοποίηση των μη αρχιτεκτονικών καταχωρητών (Instruction Register, PC Register, A/B/I/S Registers, Memory Address Register, Write Data Memory Register και Read Data Memory Register) που τοποθετούνται ανάμεσα στα διακριτά βήματα εκτέλεσης της εντολής.

Ο 2x1Multiplexer υλοποιεί έναν πολυπλέκτη για την ανάγνωση του αρχείου των καταχωρητών (A1/RD1, A2/RD2), υλοποιεί έναν πολυπλέκτη για την επιλογή σήματος Result μεταξύ των σημάτων του Read Data Memory Register και της ALU, υλοποιεί έναν πολυπλέκτη για την επιλογή του σήματος εγγραφής του αρχείου καταχωρητών (μεταξύ του PC+4 και του Result), υλοποιεί έναν πολυπλέκτη για την επιλογή του καταχωρητή στον οποίο θα γίνει η εγγραφή (μεταξύ του destination register Rd της εντολής - Instr[15:12] και του R14). Ο πολυπλέκτης αυτός είναι απαραίτητος προκειμένου να μπορεί να υλοποιηθεί η εντολή BL. Η οντότητα 2x1Mux υλοποιεί επίσης έναν πολυπλέκτη για την επιλογή του σήματος SrcB της δεύτερης εισόδου της ALU (Άμεση διευθυνσιοδότηση ή διευθυνσιοδότητση καταχωρητή) και η οντότητα 3x1Mux υλοποιεί τον πολυπλέκτη που επιλέγει την επόμενη εντολή που θα εκτελεστεί.

Η μονάδα INC4 (Incrementer by 4) χρησιμοποιείται για την αύξηση του PC σήματος (PC+4, διεύθυνση της επόμενης εντολής), καθώς και για την αύξηση του PC σήματος (PC+8) για την σωστή ανάγνωση του καταχωρητή R15.

Τέλος η μονάδα επέκτασης προσήμου/μηδενός (Extend) χρησιμοποιείται για την επέκταση του σήματος imm12 σε σήμα των 32 bit προκειμένου να χρησιμοποιηθεί απ' την ALU σε περίπτωση που η εντολή υποστηρίζει άμεση διευθυνσιοδότηση.

Ακολουθεί ο κώδικας VHDL της δομής του Data Path.

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

entity Data_Path is
Port (
    CLK         :   in  std_logic;
    RESET       :   in  std_logic;
    PCWrite     :   in  std_logic;
    IRWrite     :   in  std_logic;
    PCSrc       :   in  std_logic_vector(1 downto 0);
    RegSrc      :   in  std_logic_vector(2 downto 0);
    RegWrite    :   in  std_logic;
    MAWrite     :   in  std_logic;
    ImmSrc      :   in  std_logic;
    ALUSrc      :   in  std_logic;
    ALUControl  :   in  std_logic_vector(3 downto 0);
    FlagsWrite  :   in  std_logic;
    MemWrite    :   in  std_logic;
    MemtoReg    :   in  std_logic;
    Flags       :   out std_logic_vector(3 downto 0);
    PC          :   out std_logic_vector(31 downto 0);
    Instr       :   out std_logic_vector(31 downto 0);
    ALUResult   :   out std_logic_vector(31 downto 0);
    WriteData   :   out std_logic_vector(31 downto 0);
    Result      :   out std_logic_vector(31 downto 0)
    );
end Data_Path;
```

```vhdl
architecture Structural of Data_Path is

--Submodules Declaration
    component ROM
    Generic(N : positive := 6; -- address length
            M : positive := 32); -- data word length
    Port(
        ADDR     :   in  std_logic_vector(N-1 downto 0);
        DATA_OUT :   out std_logic_vector(M-1 downto 0)
        );
    end component;

    component REG_FILE
    Generic(N : positive := 4; -- address length
            M : positive := 32); -- data word length
    Port(
        CLK     :   in  std_logic;
        RESET   :   in  std_logic;
        WE3     :   in  std_logic;
        ADDR_A1 :   in  std_logic_vector(N-1 downto 0);
        ADDR_A2 :   in  std_logic_vector(N-1 downto 0);
        ADDR_A3 :   in  std_logic_vector(N-1 downto 0);
        WD3     :   in  std_logic_vector(M-1 downto 0);
        R15     :   in  std_logic_vector(M-1 downto 0);
        RD1     :   out std_logic_vector(M-1 downto 0);
        RD2     :   out std_logic_vector(M-1 downto 0)
        );
     end component;

    component ALU
    Generic(Width : positive := 32);
    Port(
        CLK         :  in  std_logic;
        SrcA        :  in  std_logic_vector(Width-1 downto 0);
        SrcB        :  in  std_logic_vector(Width-1 downto 0);
        ALUControl  :  in  std_logic_vector(3 downto 0);
        ALUResult   :  out std_logic_vector(Width-1 downto 0);
        Flags       :  out std_logic_vector(3 downto 0) -- N,Z,C,V
        );
    end component;

    component Data_Memory
    Generic(N : positive := 5;  -- Address length
            M : positive := 32); -- Data word length
    Port(
        CLK      :   in  std_logic;
        WE       :   in  std_logic;
        ADDR     :   in  std_logic_vector(N-1 downto 0);
        DATA_IN  :   in  std_logic_vector(M-1 downto 0);
        DATA_OUT :   out std_logic_vector(M-1 downto 0)
        );
    end component;

    component Extend
    Generic(Width_In  : positive := 24;
            Width_Out : positive := 32);
    Port(
        ImmSrc  :  in  std_logic;
        SZ_in   :  in  std_logic_vector(Width_In-1 downto 0);
        SZ_out  :  out  std_logic_vector(Width_Out-1 downto 0)
        );
    end component;
```

```vhdl
    component MUX3x1
    Generic(width : positive := 32);
    Port (
       A0     :  in  std_logic_vector(width-1 downto 0);
       A1     :  in  std_logic_vector(width-1 downto 0);
       A2     :  in  std_logic_vector(width-1 downto 0);
       SEL    :  in  std_logic_vector(1 downto 0);
       Y      :  out std_logic_vector(width-1 downto 0)
       );
    end component;

    component INC4
    Generic(width : positive := 32);
    Port (
        INC_IN  :  in  std_logic_vector(width-1 downto 0);
        INC_OUT :  out std_logic_vector(width-1 downto 0)
        );
    end component;

    component WE_REGN
    Generic (Width : positive := 32);
    Port (
        CLK    :   in  std_logic;
        Reset  :   in  std_logic;
        WE     :   in  std_logic;
        D      :   in  std_logic_vector(Width-1 downto 0);
        Q      :   out std_logic_vector(Width-1 downto 0)
    );
    end component;
```

```vhdl
    signal PC_IN_sig       : std_logic_vector(31 downto 0):= (others => '0');
    signal PC_OUT_sig      : std_logic_vector(31 downto 0):= (others => '0');
    signal IR_IN_sig       : std_logic_vector(31 downto 0):= (others => '0');
    signal IR_OUT_sig      : std_logic_vector(31 downto 0):= (others => '0');
    signal PCPlus4_sig     : std_logic_vector(31 downto 0):= (others => '0');
    signal PCP4REG_sig     : std_logic_vector(31 downto 0):= (others => '0');
    signal PCPlus8_sig     : std_logic_vector(31 downto 0):= (others => '0');
    signal RA1_sig         : std_logic_vector(3  downto 0):= (others => '0');
    signal RA2_sig         : std_logic_vector(3  downto 0):= (others => '0');
    signal WA_sig          : std_logic_vector(3  downto 0):= (others => '0');
    signal Result_sig      : std_logic_vector(31 downto 0):= (others => '0');
    signal RD1_sig         : std_logic_vector(31 downto 0):= (others => '0');
    signal RD2_sig         : std_logic_vector(31 downto 0):= (others => '0');
    signal WriteData_sig   : std_logic_vector(31 downto 0):= (others => '0');
    signal IREG_IN_sig     : std_logic_vector(31 downto 0):= (others => '0');
    signal ExtImm_sig      : std_logic_vector(31 downto 0):= (others => '0');
    signal SrcB_sig        : std_logic_vector(31 downto 0):= (others => '0');
    signal SrcA_sig        : std_logic_vector(31 downto 0):= (others => '0');
    signal Flags_sig       : std_logic_vector(3  downto 0):= (others => '0');
    signal ALUResult_sig   : std_logic_vector(31 downto 0):= (others => '0');
    signal MA_sig          : std_logic_vector(31 downto 0):= (others => '0');
    signal WD_sig          : std_logic_vector(31 downto 0):= (others => '0');
    signal Data_Memory_sig : std_logic_vector(31 downto 0):= (others => '0');
    signal RD_sig          : std_logic_vector(31 downto 0):= (others => '0');
    signal SREG_sig        : std_logic_vector(31 downto 0):= (others => '0');
    signal WD3_sig         : std_logic_vector(31 downto 0):= (others => '0');
begin
    --Program Counter instantiation
    PC_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => PCWrite,
    D     => PC_IN_sig,
    Q     => PC_OUT_sig
    );

    --Instruction Memory instantiation
    INSTR_MEM_inst: ROM
    Generic map(N => 6, M => 32)
    Port map(
    ADDR     => PC_OUT_sig(7 downto 2),
    DATA_OUT => IR_IN_sig
    );

    --Incrementer by 4 instantiation (pc+4)
    Inc4_inst: INC4
    Generic map(Width => 32)
    Port map(
    INC_IN  => PC_OUT_sig,
    INC_OUT => PCPlus4_sig
    );

    --Instruction Register instantiation
    IR_REG_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => IRWrite,
    D     => IR_IN_sig,
    Q     => IR_OUT_sig
    );
```

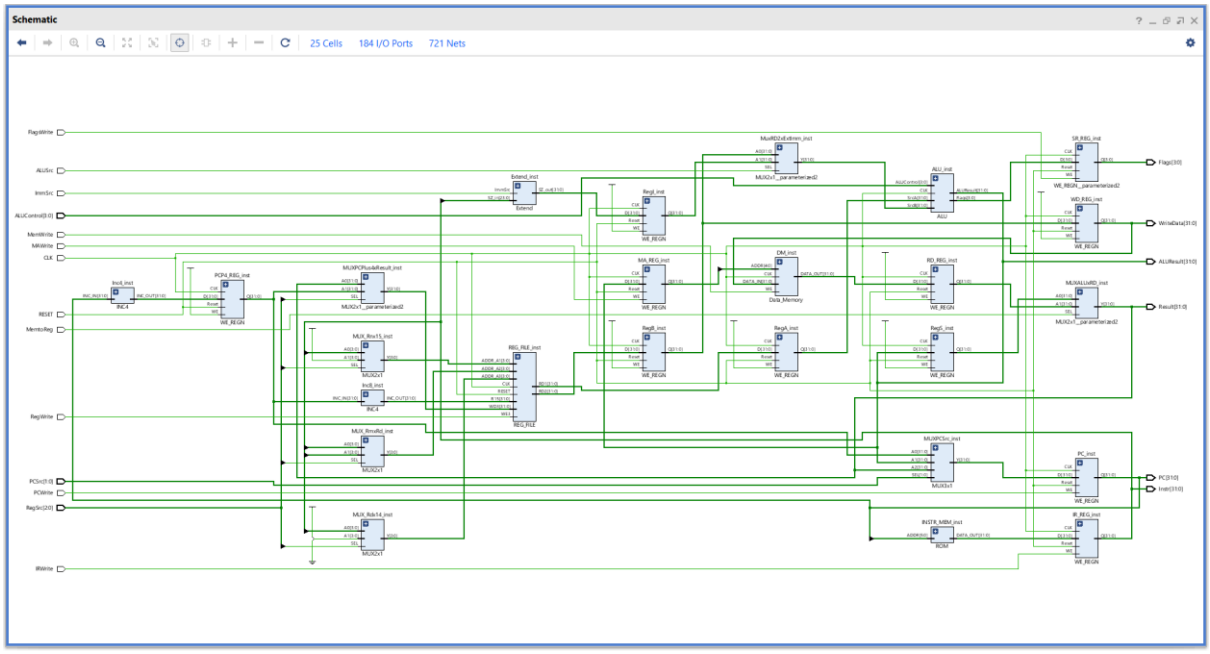
```
    --PCPlus4 Register instantiation
    PCP4_REG_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => PCPlus4_sig,
    Q     => PCP4REG_sig
    );

    --Incrementer by 8 instantiation (pc+8)
    Inc8_inst: INC4
    Generic map(Width => 32)
    Port map(
    INC_IN  => PCP4REG_sig,
    INC_OUT => PCPlus8_sig
    );

    --MuxRnx15 instantiation
    MUX_Rnx15_inst: MUX2x1
    Generic map(Width => 4)
    Port map(
    A0  => IR_OUT_sig(19 downto 16),
    A1  => "1111",
    SEL => RegSrc(0),
    Y   => RA1_sig
    );

    --MuxRmxRd instantiation
    MUX_RmxRd_inst: MUX2x1
    Generic map(Width => 4)
    Port map(
    A0  => IR_OUT_sig(3  downto 0),
    A1  => IR_OUT_sig(15 downto 12),
    SEL => RegSrc(1),
    Y   => RA2_sig
    );

    --MuxRdx14 instantiation
    MUX_Rdx14_inst: MUX2x1
    Generic map(Width => 4)
    Port map(
    A0  => IR_OUT_sig(15 downto 12),
    A1  => "1110",
    SEL => RegSrc(2),
    Y   => WA_sig
    );

    --Register File instantiation
    REG_FILE_inst: REG_FILE
    Generic map(N => 4, M => 32)
    Port map(
    CLK     => CLK,
    RESET   => RESET,
    WE3     => RegWrite,
    ADDR_A1 => RA1_sig,
    ADDR_A2 => RA2_sig,
    ADDR_A3 => WA_sig,
    WD3     => WD3_sig,
    R15     => PCPlus8_sig,
    RD1     => RD1_sig,
    RD2     => RD2_sig
    );

    --Extend module instantiation
    Extend_inst: Extend
    Generic map(Width_In  => 24, Width_Out => 32)
    Port map(
    ImmSrc => ImmSrc,
    SZ_in  => IR_OUT_sig(23 downto 0),
    SZ_out => IREG_IN_sig
    );
```

```
    --Register A instantiation
    RegA_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => RD1_sig,
    Q     => SrcA_sig
    );

    --Register B instantiation
    RegB_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => RD2_sig,
    Q     => WriteData_sig
    );

    --Register I instantiation
    RegI_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => IREG_IN_sig,
    Q     => ExtImm_sig
    );

    --MuxRD2xExtImm instantiation
    MuxRD2xExtImm_inst: MUX2x1
    Generic map(Width => 32)
    Port map(
    A0  => WriteData_sig,
    A1  => ExtImm_sig,
    SEL => ALUSrc,
    Y   => SrcB_sig
    );

    --Arithmetic Logic Unit instantiation
    ALU_inst: ALU
    Generic map(Width => 32)
    Port map(
    CLK         => CLK,
    SrcA        => SrcA_sig,
    SrcB        => SrcB_sig,
    ALUControl  => ALUControl,
    ALUResult   => ALUResult_sig,
    Flags       => Flags_sig
    );

    --Status Register instantiation
    SR_REG_inst: WE_REGN
    Generic map(Width => 4)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => FlagsWrite,
    D     => Flags_sig,
    Q     => Flags
    );
```

```
    --Memory Address Register (MA) instantiation
    MA_REG_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => MAWrite,
    D     => ALUResult_sig,
    Q     => MA_sig
    );

    --Memory Write Data Register (WD) instantiation
    WD_REG_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => WriteData_sig,
    Q     => WD_sig
    );

    --Data Memory (DM) instantiation (Distributed RAM)
    DM_inst: Data_Memory
    Generic map(N => 5, M => 32)
    Port map(
    CLK      => CLK,
    WE       => MemWrite,
    ADDR     => MA_sig(6 downto 2),
    DATA_IN  => WD_sig,
    DATA_OUT => Data_Memory_sig
    );

    --Memory Read Data Register (RD) instantiation
    RD_REG_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => Data_Memory_sig,
    Q     => RD_sig
    );

    --Register S instantiation
    RegS_inst: WE_REGN
    Generic map(Width => 32)
    Port map(
    CLK   => CLK,
    Reset => RESET,
    WE    => '1',
    D     => ALUResult_sig,
    Q     => SREG_sig
    );

    --MUXAluResultxReadDataMem instantiation
    MUXALUxRD_inst: MUX2x1
    Generic map(Width => 32)
    Port map(
    A0  => SREG_sig,
    A1  => RD_sig,
    SEL => MemtoReg,
    Y   => Result_sig
    );

    --MUX(PC+4)xResult to write WD3 Register File
    MUXPCPlus4xResult_inst: MUX2x1
    Generic map(Width => 32)
    Port map(
    A0  => Result_sig,
    A1  => PCP4REG_sig,
    SEL => RegSrc(2),
    Y   => WD3_sig
    );
```

```
    --MUXAluResultxReadDataMem instantiation
    MUXALUxRD_inst: MUX2x1
    Generic map(Width => 32)
    Port map(
    A0  => SREG_sig,
    A1  => RD_sig,
    SEL => MemtoReg,
    Y   => Result_sig
    );

    --MUX(PC+4)xResult to write WD3 Register File
    MUXPCPlus4xResult_inst: MUX2x1
    Generic map(Width => 32)
    Port map(
    A0  => Result_sig,
    A1  => PCP4REG_sig,
    SEL => RegSrc(2),
    Y   => WD3_sig
    );

    --Mux To choose next instruction's address
    MUXPCSrc_inst: MUX3x1
    Generic map(Width => 32)
    Port map(
    A0  => PCP4REG_sig,
    A1  => ALUResult_sig,
    A2  => RESULT_sig,
    SEL => PCSrc,
    Y   => PC_IN_sig
    );

    --DataPath Output Signals assertion
    PC        <=  PC_OUT_sig;
    Instr     <=  IR_OUT_sig;
    ALUResult <=  ALUResult_sig;
    WriteData <=  WD_sig;
    Result    <=  Result_sig;

end Structural;
```

VHDL υλοποίηση της Διαδρομής Δεδομένων (DataPath)



Data Path RTL

Στο RTL διάγραμμα διακρίνονται καθαρά όλες οι υπομονάδες της δομής της διαδρομής δεδομένων του επεξεργαστή, τα σήματα εισόδου/εξόδου καθώς και η επικοινωνία των μονάδων με τη βοήθεια των εσωτερικών σημάτων που ορίστηκαν στην υλοποίηση.

1.5 Περιγραφή των υπομονάδων της συνδυαστικής λογικής της μονάδας ελέγχου

Η συνδυαστική λογική της μονάδας ελέγχου αποτελείται από :

- Τον αποκωδικοποιητή εντολών (Instruction Decoder) , ο οποίος παράγει τα κατάλληλα σήματα με βάση τα οποία η διαδρομή δεδομένων υλοποιεί την ορθή εκτέλεση της εντολής.
- Τη λογική μονάδα ελέγχου συνθήκης (CONDLogic), η οποία ελέγχει εαν ικανοποιείται η συνθήκη που ορίζεται στο πεδίο cond της εντολής με βάση τις τρέχουσες τιμές των σημαιών N, Z, C, V (flags). Το σήμα εξόδου CondEx_in εγκρίνει την εκτέλεση της εντολής (υπό συνθήκη), όταν παίρνει την τιμή 1.

Instruction Decoder

Εντολή	$Instr_{27:26}$ op	$Instr_{25:20}$ funct	Τύπος	RegSrc	ALUSrc	ImmSrc	ALUControl	MemtoReg	NoWrite_in
ADD	00	10100X	DP Imm	0X0	1	0	0000	0	0
ADD	00	00100X	DP Reg	000	0	X	0000	0	0
SUB	00	10010X	DP Imm	0X0	1	0	0001	0	0
SUB	00	00010X	DP Reg	00	0	X	0001	0	0
CMP	00	110101	DP Imm	0X0	1	0	0001	X	1
CMP	00	010101	DP Reg	000	0	X	0001	X	1
AND	00	10000X	DP Imm	0X0	1	0	1010	0	0
AND	00	00000X	DP Reg	000	0	X	1010	0	0
ORR	00	11100X	DP Imm	0X0	1	0	1011	0	0
ORR	00	01100X	DP Reg	000	0	X	1011	0	0
XOR	00	10001X	DP Imm	0X0	1	0	1110	0	0
XOR	00	00001X	DP Reg	000	0	X	1110	0	0
MOV	00	111010	DP Imm	0X0	1	0	0111	0	0
MOV	00	011010	DP Reg	000	0	X	0111	0	0
MOVN	00	111110	DP Imm	0X0	1	0	1001	0	0
MOVN	00	011110	DP Reg	000	0	X	1001	0	0
LSL	00	011010	DP Reg	0X0	0	0	0010	0	0
ASR	00	011010	DP Reg	0X0	0	0	0100	0	0
LDR	01	011001	M imm+	0X0	1	0	0000	1	0
LDR	01	010001	M imm-	0X0	1	0	0001	1	0
STR	01	011000	M imm+	010	1	0	0000	X	0
STR	01	010000	M imm-	010	1	1	0001	X	0
B	10	10XXXX	B imm+	0X1	1	1	0000	0	0
BL	10	11XXXX	B imm+	1X1	1	1	0000	0	0

Πίνακας2:Πίνακας αλήθειας του Instruction Decoder

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

entity InstrDec is
Port (
     op         :   in  std_logic_vector(1 downto 0);
     funct      :   in  std_logic_vector(5 downto 0);
     sh         :   in  std_logic_vector(1 downto 0);
     RegSrc     :   out std_logic_vector(2 downto 0);
     ALUSrc     :   out std_logic;
     ImmSrc     :   out std_logic;
     AluControl :   out std_logic_vector(3 downto 0);
     MemtoReg   :   out std_logic;
     NoWrite_in :   out std_logic
     );
end InstrDec;

architecture Behavioral of InstrDec is
begin
    process (op, funct,sh)
    begin
        if(op = "00") then    -- Data Processing Instructions
            if(funct(5) = '1') then
                ALUSrc <= '1';
            else
                ALUSrc <= '0';
            end if;
            if(funct(0) = '1') then
                NoWrite_in <= '1';
            else
                NoWrite_in <= '0';
            end if;
            case funct(4 downto 1) is
              when "0100" =>
                    ALUControl <= "0000"; -- ADD
              when "0010" =>
                    ALUControl <= "0001"; -- SUB
              when "1010" =>
                    ALUControl <= "0001"; -- CMP

             when "0000" =>
                   ALUControl <= "1010"; -- AND
             when "1100" =>
                   ALUControl <= "1011"; -- ORR
             when "0001" =>
                   ALUControl <= "1110"; -- XOR
             when "1101" =>
                    if(sh = "00") then
                       ALUControl <= "0111"; -- MOV
                    elsif(sh = "01") then
                       ALUControl <= "0011"; -- LSR
                    elsif(sh = "10") then
                       ALUControl <= "0100"; -- ASR
                    else
                       ALUControl <= "0000";
                    end if;
             when "1111" =>
                   ALUControl <= "1001"; -- MONV
              when others =>
                   ALUControl <= "0000";
           end case;
           RegSrc   <= "000";
           ImmSrc   <= '0';
           MemtoReg <= '0';
        elsif(op = "01")then  --Memory Instructions
            if(funct(3) = '1')then
                ALUControl <= "0000"; --LDR/STR(+)
            else
                ALUControl <= "0001"; --LDR/STR(-)
            end if;
            RegSrc     <= "010";
            ALUSrc     <= '1';
            ImmSrc     <= '0';
            MemtoReg   <= '1';
            NoWrite_in <= '0';
        elsif(op = "10") then    -- Branch Instructions
            if(funct(4) = '1')then
                RegSrc <= "101";
            else
                RegSrc <= "001";
            end if;
            ALUSrc     <= '1';
            ImmSrc     <= '1';
            ALUControl <= "0000";
            MemtoReg   <= '0';
            NoWrite_in <= '0';
        else
            ALUControl <= "0000";
            ImmSrc     <= '0';
            RegSrc     <= "000";
            NoWrite_in <= '0';
            MemtoReg   <= '0';
            ALUSrc     <= '0';
        end if;
end process;

end Behavioral;
```

VHDL υλοποίηση του Instruction Decoder.

CONDLogic

Στον παρακάτω πίνακα φαίνονται τα μνημονικά συνθήκης με τις εξισώσεις Boole των σημαιών που τις ικανοποιούν. Η σχεδίαση της λογικής ελέγχου συνθήκης σε VHDL πραγματοποιείται βάσει των μνημονικών συνθήκης.

$cond_{3:0}$	Μνημονικό	Όνομα	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\bar{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\bar{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\bar{V}$

$cond_{3:0}$	Μνημονικό	Όνομα	CondEx
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z+\bar{C}$
1010	GE	Signed greater or equal	$\overline{N \oplus V}$
1011	LT	Signed less	$N \oplus V$
1100	GT	Signed greater	$\bar{Z}\,\overline{N \oplus V}$
1101	LE	Signed less or equal	$Z+(N \oplus V)$
1110	AL (ή none)	Always / unconditional	1
1111	none	For unconditional instructions	1

Πίνακας 3: Μνημονικά συνθήκης με τις εξισώσεις boole των σημαιών που τις ικανοποιούν.

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

entity CONDLogic is
Port (
    cond      :   in  std_logic_vector(3 downto 0);
    flags     :   in  std_logic_vector(3 downto 0);
    CondEx_in :   out std_logic
    );
end CONDLogic;

architecture Behavioral of CONDLogic is
-- N,Z,C,V
begin
    CondExec_proc: process(cond, flags)
    begin
        case cond is
            when "0000" => CondEx_in <= flags(1); --(Z)
            when "0001" => CondEx_in <= not(flags(1)); --(notZ)
            when "0010" => CondEx_in <= flags(2);  --(C)
            when "0011" => CondEx_in <= not(flags(2)); --(notC)
            when "0100" => CondEx_in <= flags(0); --(N)
            when "0101" => CondEx_in <= not(flags(0)); --(notN)
            when "0110" => CondEx_in <= flags(3); --(V)
            when "0111" => CondEx_in <= not(flags(3)); --(notV)
            when "1000" => CondEx_in <= not(flags(1)) and flags(2); --(notZ)AND(C)
            when "1001" => CondEx_in <= flags(1) or not(flags(2)); -- (Z)OR(notC)
            when "1010" => CondEx_in <= not(flags(0) xor flags(3)); --(not(N)xor(V))
            when "1011" => CondEx_in <= flags(0) xor flags(3); --(N)xor(V)
            when "1100" => CondEx_in <= not(flags(1)) and not(flags(0) xor flags(3)); -- (not(Z))AND(not((N)XOR(V)))
            when "1101" => CondEx_in <= flags(1) or (flags(0) xor flags(3)); -- (Z)OR((N)XOR(V))
            when "1110" => CondEx_in <= '1';
            when "1111" => CondEx_in <= '1';
            when others => CondEx_in <= '1';
         end case;
     end process;
end Behavioral;
```

1.6 Περιγραφή της μηχανής πεπερασμένων καταστάσεων (FSM) της μονάδας ελέγχου.

Η μηχανή πεπερασμένων καταστάσεων της μονάδας ελέγχου αποτελεί τον «ηλεκτρονικό εγκέφαλο» του επεξεργαστή. Συσχετίζει τα βήματα εκτέλεσης των εντολών με τους απαιτούμενους κύκλους του ρολογιού ενεργοποιώντας κατάλληλα τα σήματα έγκρισης εγγραφής των διαφόρων υπομονάδων της διαδρομής δεδομένων, καθώς και το σήμα επιλογής διεύθυνσης επόμενης εντολής PCSrc[1:0]. Σε περίπτωση που το σήμα CondEx_in = 0 η εκτέλεση της εντολής τερματίζεται πρόωρα με την εκτέλεση του τελευταίου βήματος (βήμα 5) όπου ενεργοποιείται το PCWrite έτσι ώστε στον επόμενο κύκλο να εκτελεσθεί το βήμα 1 της αμέσως επόμενης εντολής (διεύθυνση PC+4).

Το FSM αποτελεί ένα σύγχρονο ακολουθιακό κύκλωμα με σύγχρονο RESET και η μετάβαση στην αμέσως επόμενη κατάσταση πραγματοποιείται σε κάθε κατερχόμενη ακμή του ρολογιού. Η συνδυαστική λογική της επιλογής της επόμενης κατάστασης υλοποιήθηκε σύμφωνα με τον οδηγό της εργασίας. Παρακάτω παρουσιάζεται ο πίνακας αλήθειας της FSM δηλαδή οι τιμές των σημάτων εξόδου που ανταποκρίνονται σε κάθε δυνατό state.

Current state	IRWrite	RegWrite	MAWrite	MemWrite	FlagsWrite	PCSrc	PCWrite
S0	1	0	0	0	0	00	0
S1	0	0	0	0	0	00	0
S2a	0	0	1	0	0	00	0
S2b	0	0	0	0	0	00	0
S3	0	0	0	0	0	00	0
S4a	0	1	0	0	0	00	1
S4b	0	0	0	0	0	01	1
S4c	0	0	0	0	0	00	1
S4d	0	0	0	1	0	00	1
S4e	0	1	0	0	1	00	1
S4f	0	0	0	0	1	01	1
S4g	0	0	0	0	1	00	1
S4h	0	0	0	0	0	11	1
S4i	0	1	0	0	0	11	1

Πίνακας 4: Πίνακας αλήθειας του FSM. Το state S4i προστέθηκε ώστε να υλοποιείται η εντολή BL

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity FSM is
5  Port (
6      CLK          :   in   std_logic;
7      RESET        :   in   std_logic;
8      op           :   in   std_logic_vector(1 downto 0);
9      S            :   in   std_logic;
10     L            :   in   std_logic;
11     Rd           :   in   std_logic_vector(3 downto 0);
12     NoWrite_in   :   in   std_logic;
13     CondEx_in    :   in   std_logic;
14     PCWrite      :   out  std_logic;
15     IRWrite      :   out  std_logic;
16     RegWrite     :   out  std_logic;
17     FlagsWrite   :   out  std_logic;
18     MAWrite      :   out  std_logic;
19     MemWrite     :   out  std_logic;
20     PCSrc        :   out  std_logic_vector(1 downto 0)
21     );
22 end FSM;
23
24 architecture Behavioral of FSM is
25     type FSM_states is (S0,S1,S2a,S2b,S3,S4a,S4b,S4c,S4d,S4e,
26                         S4f,S4g,S4h,S4i);
27     signal current_state, next_state : FSM_states;
28     signal op_sig : std_logic_vector(1 downto 0);
29     signal Rd_sig : std_logic_vector(3 downto 0);
30     signal S_sig  : std_logic;
31     signal L_sig  : std_logic;
32     signal NoW    : std_logic;
33     signal cEx    : std_logic;
34 begin
```

```
begin

    op_sig  <= op;
    Rd_sig  <= Rd;
    S_sig   <= S;
    L_sig   <= L;
    NoW     <= NoWrite_in;
    cEx     <= CondEx_in;

    SYNC: process(CLK)
    begin
        if(falling_edge(CLK))then
            if(RESET = '1') then current_state <= S0;
            else current_state <= next_state;
            end if;
        end if;
    end process;

    ASYNC: process(current_state, op_sig, Rd_sig, S_sig, L_sig, NoW, cEx)
           begin
             next_state <= S0;
             case current_state is
                when S0  =>
                    IRWrite    <= '1';
                    RegWrite   <= '0';
                    MAWrite    <= '0';
                    MemWrite   <= '0';
                    FlagsWrite <= '0';
                    PCSrc      <= "00";
                    PCWrite    <= '0';
                    next_state <= S1;

                 when S1  =>
                     IRWrite    <= '0';
                     RegWrite   <= '0';
                     MAWrite    <= '0';
                     MemWrite   <= '0';
                     FlagsWrite <= '0';
                     PCSrc      <= "00";
                     PCWrite    <= '0';
                     if(cEx = '1')then
                         if(op_sig = "00")then
                             if(NoW = '1')then
                                 next_state <= S4g;
                             else
                                 next_state <= S2b;
                             end if;
                         elsif(op_sig = "10")then
                             if(L_sig = '0')then
                                 next_state <= S4h;
                             else
                                 next_state <=S4i;
                             end if;
                         elsif(op_sig = "01")then
                             next_state <= S2a;
                         else
                             next_state <= S0;
                         end if;
                     else
                         next_state <= S4c;
                     end if;
```

```
                when S2a  =>
                    IRWrite    <= '0';
                    RegWrite   <= '0';
                    MAWrite    <= '1';
                    MemWrite   <= '0';
                    FlagsWrite <= '0';
                    PCSrc      <= "00";
                    PCWrite    <= '0';
                    if(L_sig = '1')then
                        next_state <= S3;
                    else
                        next_state <= S4d;
                    end if;
                when S2b  =>
                    IRWrite    <= '0';
                    RegWrite   <= '0';
                    MAWrite    <= '0';
                    MemWrite   <= '0';
                    FlagsWrite <= '0';
                    PCSrc      <= "00";
                    PCWrite    <= '0';
                    if(Rd = "1111")then
                        if(S_sig = '0')then
                            next_state <= S4b;
                        else
                            next_state <= S4f;
                        end if;
                    else
                        if(S_sig = '0')then
                            next_state <= S4a;
                        else
                            next_state <= S4e;
                        end if;
                    end if;

                when S3  =>
                    IRWrite    <= '0';
                    RegWrite   <= '0';
                    MAWrite    <= '0';
                    MemWrite   <= '0';
                    FlagsWrite <= '0';
                    PCSrc      <= "00";
                    PCWrite    <= '0';
                    if(Rd_sig = "1111")then
                        next_state <= S4b;
                    elsif(Rd_sig = "0000")then
                        next_state <= S4a;
                    else
                        next_state <= S0;
                    end if;
                when S4a  =>
                    IRWrite    <= '0';
                    RegWrite   <= '1';
                    MAWrite    <= '0';
                    MemWrite   <= '0';
                    FlagsWrite <= '0';
                    PCSrc      <= "00";
                    PCWrite    <= '1';
                    next_state <= S0;
                when S4b  =>
                    IRWrite    <= '0';
                    RegWrite   <= '0';
                    MAWrite    <= '0';
                    MemWrite   <= '0';
                    FlagsWrite <= '0';
                    PCSrc      <= "10";
                    PCWrite    <= '1';
                    next_state <= S0;
```

```
            when S4c  =>
                IRWrite    <= '0';
                RegWrite   <= '0';
                MAWrite    <= '0';
                MemWrite   <= '0';
                FlagsWrite <= '0';
                PCSrc      <= "00";
                PCWrite    <= '1';
                next_state <= S0;
            when S4d  =>
                IRWrite    <= '0';
                RegWrite   <= '0';
                MAWrite    <= '0';
                MemWrite   <= '1';
                FlagsWrite <= '0';
                PCSrc      <= "00";
                PCWrite    <= '1';
                next_state <= S0;
            when S4e   =>
                IRWrite    <= '0';
                RegWrite   <= '1';
                MAWrite    <= '0';
                MemWrite   <= '0';
                FlagsWrite <= '1';
                PCSrc      <= "00";
                PCWrite    <= '1';
                next_state <= S0;
            when S4f   =>
                IRWrite    <= '0';
                RegWrite   <= '0';
                MAWrite    <= '0';
                MemWrite   <= '0';
                FlagsWrite <= '1';
                PCSrc      <= "10";
                PCWrite    <= '1';
                next_state <= S0;
```

```
            when S4g   =>
                IRWrite    <= '0';
                RegWrite   <= '0';
                MAWrite    <= '0';
                MemWrite   <= '0';
                FlagsWrite <= '1';
                PCSrc      <= "00";
                PCWrite    <= '1';
                next_state <= S0;
            when S4h   =>
                IRWrite    <= '0';
                RegWrite   <= '0';
                MAWrite    <= '0';
                MemWrite   <= '0';
                FlagsWrite <= '0';
                PCSrc      <= "11";
                PCWrite    <= '1';
                next_state <= S0;
            when S4i   =>
                IRWrite    <= '0';
                RegWrite   <= '1';
                MAWrite    <= '0';
                MemWrite   <= '0';
                FlagsWrite <= '0';
                PCSrc      <= "11";
                PCWrite    <= '1';
                next_state <= S0;
            when others =>
                next_state <= S0;
            end case;
        end process;
end Behavioral;
```

VHDL υλοποίηση του FSM του Control Unit

1.7 Περιγραφή της δομής της μονάδας ελέγχου (Control) του επεξεργαστή.

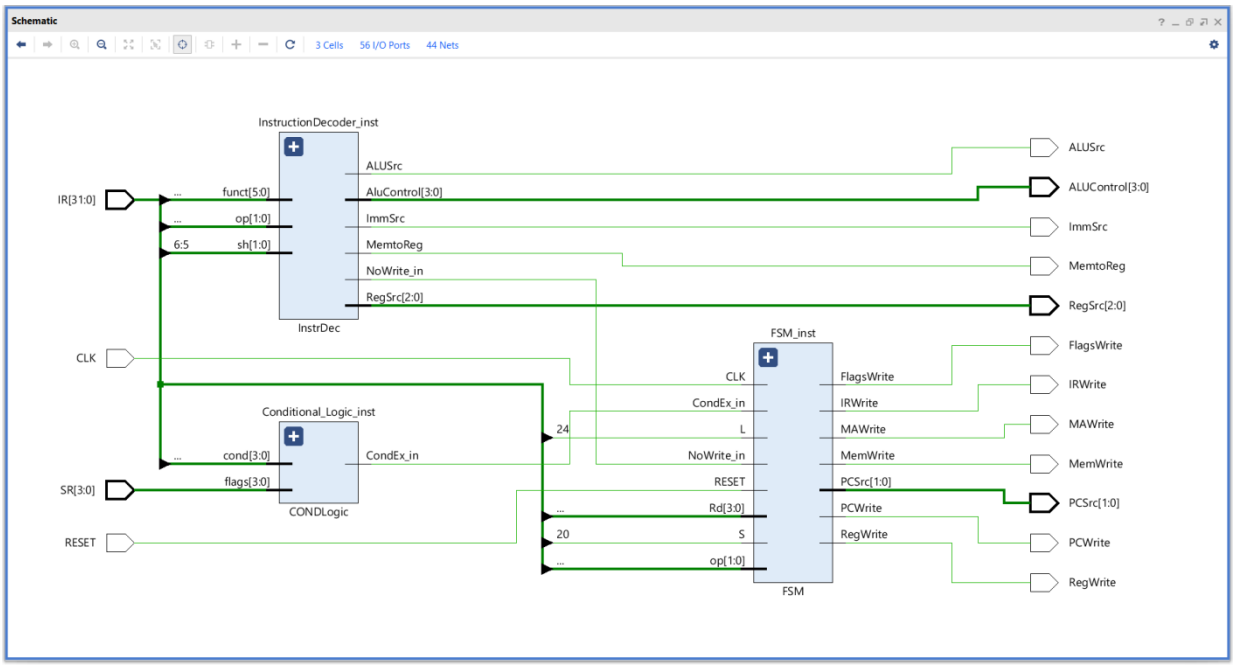
Στην ενότητα αυτή παρουσιάζεται η VHDL υλοποίηση της μονάδας ελέγχου δηλαδή του ενός από τα δύο βασικά modules του επεξεργαστή. Η σχεδίαση έχει γίνει σε ανώτερο ιεραρχικό επίπεδο (structural) και περιλαμβάνει τα submodules :

- Instruction Decoder
- Conditional Logic
- FSM

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

entity Control is
Port (
    CLK         :   in  std_logic;
    RESET       :   in  std_logic;
    IR          :   in  std_logic_vector(31 downto 0);
    SR          :   in  std_logic_vector(3  downto 0);
    RegSrc      :   out std_logic_vector(2 downto 0);
    ALUSrc      :   out std_logic;
    MemtoReg    :   out std_logic;
    ALUControl  :   out std_logic_vector(3 downto 0);
    ImmSrc      :   out std_logic;
    IRWrite     :   out std_logic;
    RegWrite    :   out std_logic;
    MAWrite     :   out std_logic;
    MemWrite    :   out std_logic;
    FlagsWrite  :   out std_logic;
    PCSrc       :   out std_logic_vector(1 downto 0);
    PCWrite     :   out std_logic
    );
end Control;
```

```vhdl
architecture Structural of Control is

--Submodules declaration
  component InstrDec
  Port(
    op          :   in  std_logic_vector(1 downto 0);
    funct       :   in  std_logic_vector(5 downto 0);
    sh          :   in  std_logic_vector(1 downto 0);
    RegSrc      :   out std_logic_vector(2 downto 0);
    ALUSrc      :   out std_logic;
    ImmSrc      :   out std_logic;
    AluControl  :   out std_logic_vector(3 downto 0);
    MemtoReg    :   out std_logic;
    NoWrite_in  :   out std_logic
    );
  end component;

  component CONDLogic
  Port(
    cond       :   in  std_logic_vector(3 downto 0);
    flags      :   in  std_logic_vector(3 downto 0);
    CondEx_in  :   out std_logic
    );
  end component;

  component FSM
  Port(
    CLK         :   in   std_logic;
    RESET       :   in   std_logic;
    op          :   in   std_logic_vector(1 downto 0);
    S           :   in   std_logic;
    L           :   in   std_logic;
    Rd          :   in   std_logic_vector(3 downto 0);
    NoWrite_in  :   in   std_logic;
    CondEx_in   :   in   std_logic;
    PCWrite     :   out  std_logic;
    IRWrite     :   out  std_logic;
    RegWrite    :   out  std_logic;
    FlagsWrite  :   out  std_logic;
    MAWrite     :   out  std_logic;
    MemWrite    :   out  std_logic;
    PCSrc       :   out  std_logic_vector(1 downto 0)
    );
  end component;

  signal NoWrite_in_sig : std_logic := '0';
  signal CondEx_in_sig  : std_logic := '0';
begin

    InstructionDecoder_inst: InstrDec
    port map(
    op         => IR(27 downto 26),
    funct      => IR(25 downto 20),
    sh         => IR(6  downto  5),
    RegSrc     => RegSrc,
    ALUSrc     => ALUSrc,
    MemtoReg   => MemtoReg,
    ALUControl => ALUControl,
    ImmSrc     => ImmSrc,
    NoWrite_in => NoWrite_in_sig
     );

    FSM_inst: FSM
    port map(
    CLK         => CLK,
    RESET       => RESET,
    op          => IR(27 downto 26),
    S           => IR(20),
    L           => IR(24),
    Rd          => IR(15 downto 12),
    NoWrite_in => NoWrite_in_sig,
    CondEx_in   => CondEx_in_sig,
    IRWrite     => IRWrite,
    RegWrite    => RegWrite,
    MAWrite     => MAWrite,
    MemWrite    => MemWrite,
    FlagsWrite => FlagsWrite,
    PCSrc       => PCSrc,
    PCWrite     => PCWrite
    );

    Conditional_Logic_inst: CONDLogic
    port map(
    cond        => IR(31 downto 28),
    flags       => SR,
    CondEx_in   => CondEx_in_sig
    );

end Structural;
```

VHDL υλοποίηση του Control Unit



Control Unit RTL

```
88  -- Stimulus process definition
89  Stimulus_process: process
90      begin
91  -- Syncronous RESET is deasserted on CLK falling edge
92  -- after GSR signal disable (it remains enabled for 100 ns)
93      RESET <= '1';
94      wait for 100 ns;
95      wait until (CLK = '0' and CLK'event);
96      RESET <= '0'; -- current stateS0, nxState S1
97      wait for 1*CLK_period;
98      IR(27 downto 26) <= "01";--current state S1, next state S2a
99      IR(31 downto 28) <= "1111";
100     wait for 1*CLK_period;
101     IR(24) <= '1';  -- current state S2a , nxState S3
102     wait for 1*CLK_period;
103     IR(15 downto 12) <= "0000";--curState S3, nxState S4a
104     wait for 1*CLK_period;
105     --curState S4a, nxState S0
106     wait for 1*CLK_period;
107     --currState S0, nxState S1
108     wait for 1*CLK_period;
109     IR(27 downto 26) <= "01";--current state S1, next state S2a
110     IR(31 downto 28) <= "1111";
111     wait for 1*CLK_period;
112     IR(24) <= '1';  -- current state S2a , nxState S3
113     wait for 1*CLK_period;
114     IR(15 downto 12) <= "1111";--curState S3, nxState S4b
115     wait for 1*CLK_period;
116     --curState S4a, nxState S0
117     wait for 1*CLK_period;
118     --currState S0, nxState S1
119     wait for 1*CLK_period;
120     IR(27 downto 26) <= "00"; --currStateS1, nxState S2b
121     IR(31 downto 28) <= "1111";
122     IR(20) <= '0';
123     wait for 1*CLK_period;
124     IR(20) <= '0';
125     IR(15 downto 12) <= "0000"; --curState S2b, nxState S4a
126     wait for 1*CLK_period;
```

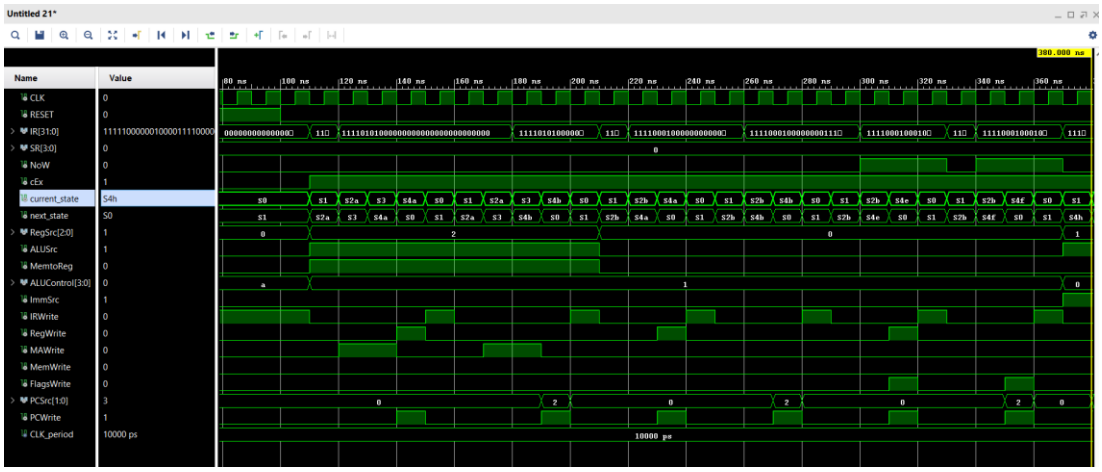
```
126     wait for 1*CLK_period;
127     --curState S4a, nxState S0
128     wait for 1*CLK_period;
129     --curState S0, nxstate S1
130     wait for 1*CLK_period;
131     IR(27 downto 26) <= "00"; --currStateS1, nxState S2b
132     IR(31 downto 28) <= "1111";
133     IR(20) <= '0';
134     wait for 1*CLK_period;
135     IR(20) <= '0';
136     IR(15 downto 12) <= "1111"; --curState S2b, nxState S4b
137     wait for 1*CLK_period;
138     --curState S4b, nxState S0
139     wait for 1*CLK_period;
140     --curState S0, nxstate S1
141     wait for 1*CLK_period;
142     IR(27 downto 26) <= "00"; --currStateS1, nxState S2b
143     IR(31 downto 28) <= "1111";
144     IR(20) <= '0';
145     wait for 1*CLK_period;
146     IR(20) <= '1';
147     IR(15 downto 12) <= "0000"; --curState S2b, nxState S4e
148     wait for 1*CLK_period;
149     --curState S4e, nxState S0
150     wait for 1*CLK_period;
151     --curState S0, nxstate S1
152     wait for 1*CLK_period;
153     IR(27 downto 26) <= "00"; --currStateS1, nxState S2b
154     IR(31 downto 28) <= "1111";
155     IR(20) <= '0';
156     wait for 1*CLK_period;
157     IR(20) <= '1';
158     IR(15 downto 12) <= "1111"; --curState S2b, nxState S4f
159     wait for 1*CLK_period;
160     --curState S4f, nxState S0
161     wait for 1*CLK_period;
162     --curState S0, nxstate S1
163     wait for 1*CLK_period;
164     IR(27 downto 26) <= "10"; --curState S1, nxState S4h
165     IR(31 downto 28) <= "1111";
```

Control Unit Testbench (Stimulus Process).

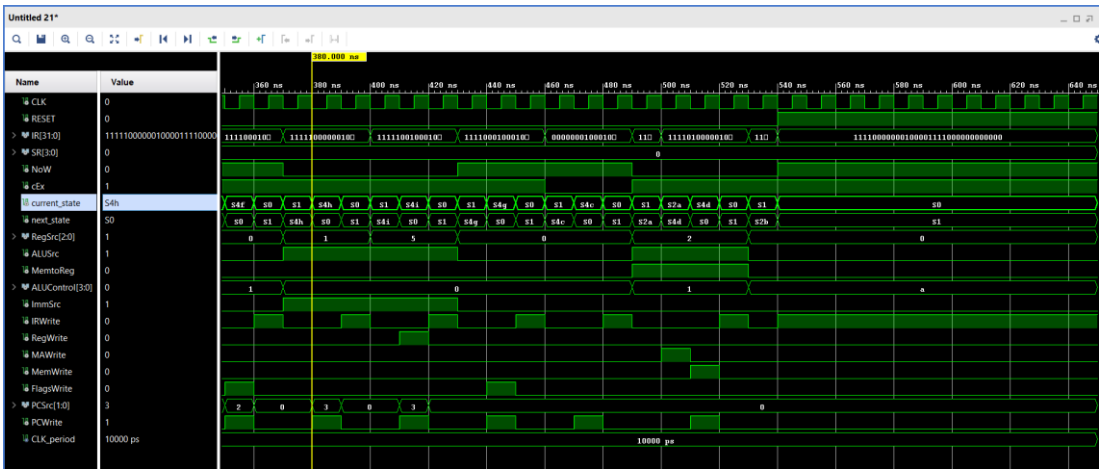
Εφαρμόζουμε κατάλληλα σήματα εισόδου ώστε το FSM module να περάσει από όλες τις δυνατές καταστάσεις του. Στο τέλος επαναφέρουμε το σήμα RESET στο 1 και επιβεβαιώνουμε ότι το επόμενο state είναι το S0 ανεξάρτητα απ'τα σήματα του sensitivity list στο συνδυαστικό process επιλογής επόμενης κατάστασης.

```
    IR(31 downto 28) <= "1111";
    IR(24) <= '0';
    wait for 1*CLK_period;
    --curState S4h, nxState S0
    wait for 1*CLK_period;
    --curState S0, nxstate S1
    wait for 1*CLK_period;
    IR(27 downto 26) <= "10"; --curState S1, nxState S4i
    IR(31 downto 28) <= "1111";
    IR(24) <= '1';
    wait for 1*CLK_period;
    --curState S4i, nxState S0
    wait for 1*CLK_period;
    --curState S0, nxState S1
    wait for 1*CLK_period;
    IR(27 downto 26) <= "00"; --curState S1, nxState S4g
    IR(31 downto 28) <= "1111";
    IR(20) <= '1';
    wait for 1*CLK_period;
    --curState S4g, nxState S0
    wait for 1*CLK_period;
    --curState S0, nxstate S1
    wait for 1*CLK_period;
    IR(31 downto 28) <= "0000"; --curState S1, nxState S4c
    wait for 1*CLK_period;
    --curState S4c, nxState S0
    wait for 1*CLK_period;
    --curState S0, nxstate S1
    wait for 1*CLK_period;
    IR(27 downto 26) <= "01"; --curState S1, nxState S2a
    IR(31 downto 28) <= "1111";
    wait for 1*CLK_period;
    IR(24) <= '0'; --curState S2a, nxState S4d
    wait for 1*CLK_period;
```

```
    --curState S4d, nxState S0
    wait for 1*CLK_period;
    --curState S0, nxState S1
    wait for 1*CLK_period;
    IR(27 downto 26) <= "00"; --curState S1, nxState S2b
    IR(31 downto 28)<= "1111";
    IR(20) <= '0';
    wait for 1*CLK_period; --curState S2b, nxState S0
    IR(20) <= '1';
    IR(15 downto 12) <= "1111";
    RESET <= '1';
    wait for 1*CLK_period;
    wait for 100 ns;

-- Message and simulation end
    report "TESTS COMPLETED";
    stop(2);
    end process;

end Behavioral;
```

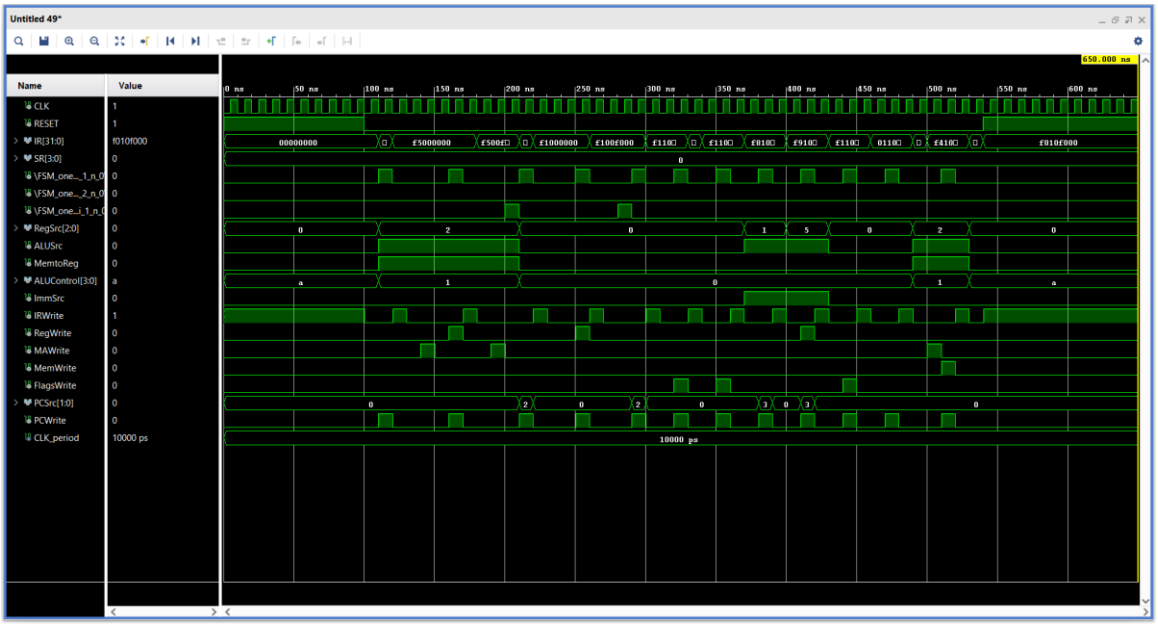


Control Unit Behavioral Simulation (FSM –based) (Part 1)

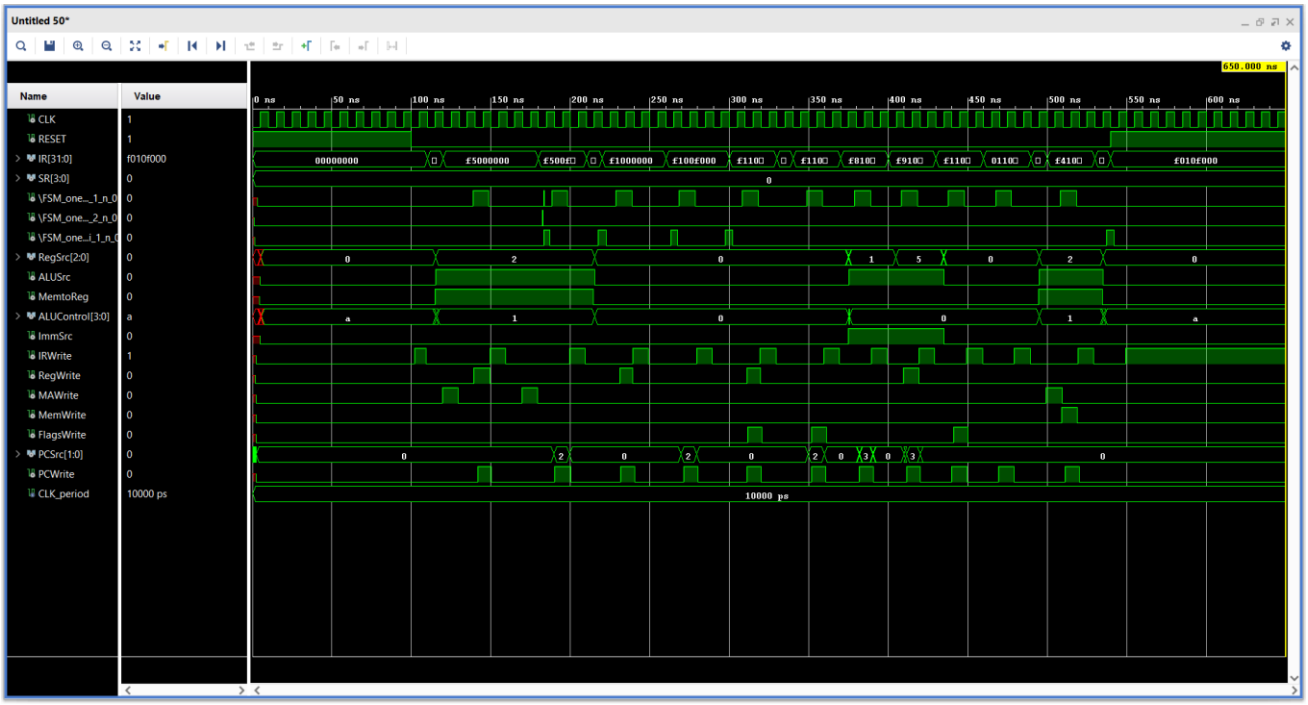


Control Unit Behavioral Simulation (FSM –based) (Part 2)

Το behavioural simulation ανταποκρίνεται ακριβώς στο testbench ενώ τα σήματα εξόδου είναι τα προβλεπόμενα, επομένως επιβεβαιώνεται ότι η λογική μονάδα λειτουργεί σύμφωνα με τις οδηγίες.



Control Unit Post Synthesis Functional Simulation



Control Unit Post Implementation Timing Simulation

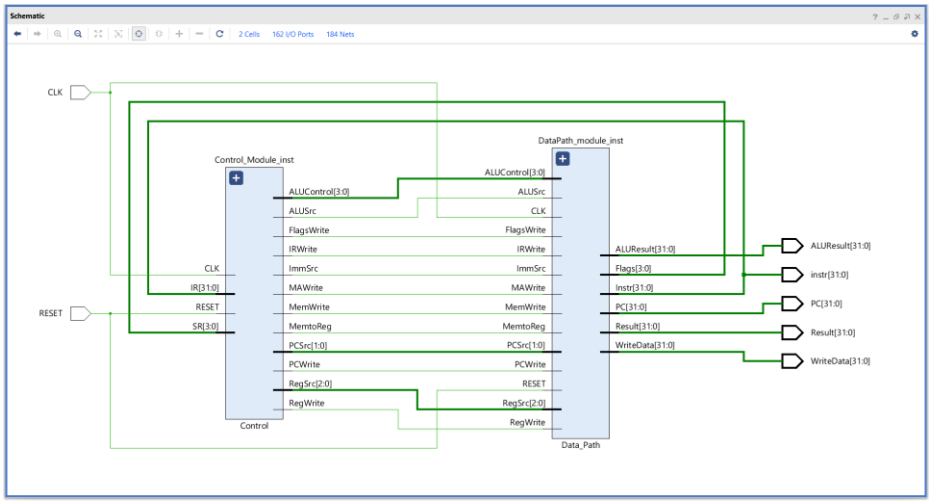
1.8 Περιγραφή της δομής του επεξεργαστή (Processor).

Η δομή του επεξεργαστή σχεδιάστηκε στο ανώτερο ιεραρχικά επίπεδο και περιλαμβάνει ως στοιχεία τη διαδρομή δεδομένων (Data Path) και τη μονάδα ελέγχου (Control Unit). Οι μόνες είσοδοι του επεξεργαστή είναι το σήμα του ρολογιού (CLK) για τον χρονισμό ολόκληρης της μονάδας καθώς και το σήμα RESET. Οι έξοδοι είναι κάποια από τα σήματα που παράγει η μονάδα διαδρομής δεδομένων (Instruction, PC, ALUResult, Result, WriteData).

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

entity Processor is
Port (
    CLK        :  in  std_logic;
    RESET      :  in  std_logic;
    PC         :  out std_logic_vector(31 downto 0);
    instr      :  out std_logic_vector(31 downto 0);
    ALUResult  :  out std_logic_vector(31 downto 0);
    WriteData  :  out std_logic_vector(31 downto 0);
    Result     :  out std_logic_vector(31 downto 0)
    );
end Processor;

architecture Structural of Processor is

 component Data_Path
 Port (
    CLK         :   in  std_logic;
    RESET       :   in  std_logic;
    PCWrite     :   in  std_logic;
    IRWrite     :   in  std_logic;
    PCSrc       :   in  std_logic_vector(1 downto 0);
    RegSrc      :   in  std_logic_vector(2 downto 0);
    RegWrite    :   in  std_logic;
    MAWrite     :   in  std_logic;
    ImmSrc      :   in  std_logic;
    ALUSrc      :   in  std_logic;
    ALUControl  :   in  std_logic_vector(3 downto 0);
    FlagsWrite  :   in  std_logic;
    MemWrite    :   in  std_logic;
    MemtoReg    :   in  std_logic;
    Flags       :   out std_logic_vector(3 downto 0);
    PC          :   out std_logic_vector(31 downto 0);
    Instr       :   out std_logic_vector(31 downto 0);
    ALUResult   :   out std_logic_vector(31 downto 0);
    WriteData   :   out std_logic_vector(31 downto 0);
    Result      :   out std_logic_vector(31 downto 0)
    );
end component;

component Control
Port (
    CLK         :   in  std_logic;
    RESET       :   in  std_logic;
    IR          :   in  std_logic_vector(31 downto 0);
    SR          :   in  std_logic_vector(3  downto 0);
    RegSrc      :   out std_logic_vector(2 downto 0);
    ALUSrc      :   out std_logic;
    MemtoReg    :   out std_logic;
    ALUControl  :   out std_logic_vector(3 downto 0);
    ImmSrc      :   out std_logic;
    IRWrite     :   out std_logic;
    RegWrite    :   out std_logic;
    MAWrite     :   out std_logic;
    MemWrite    :   out std_logic;
    FlagsWrite  :   out std_logic;
    PCSrc       :   out std_logic_vector(1 downto 0);
    PCWrite     :   out std_logic
    );
end component;
```

VHDL υλοποίηση του Processor

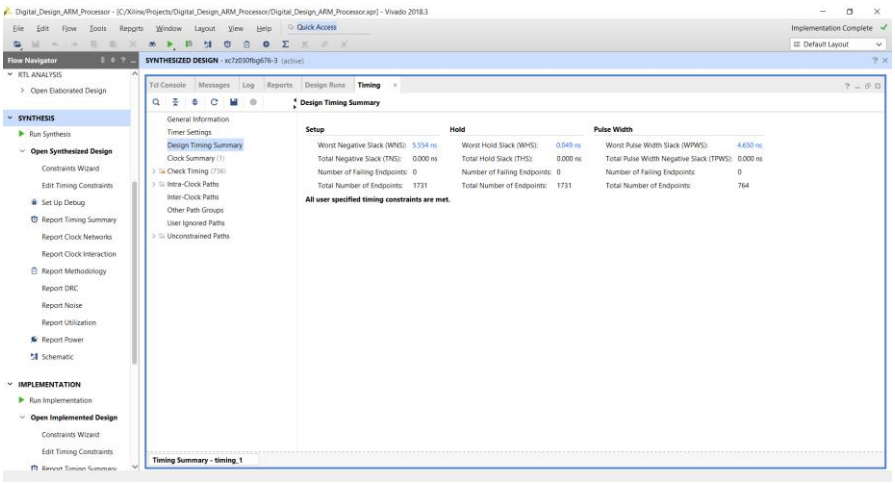
```
    signal PCWrite_sig    : std_logic := '0';
    signal IRWrite_sig    : std_logic := '0';
    signal PCSrc_sig      : std_logic_vector(1 downto 0) := (others => '0');
    signal RegSrc_sig     : std_logic_vector(2 downto 0) := (others => '0');
    signal RegWrite_sig   : std_logic := '0';
    signal MAWrite_sig    : std_logic := '0';
    signal ImmSrc_sig     : std_logic := '0';
    signal ALUSrc_sig     : std_logic := '0';
    signal ALUControl_sig : std_logic_vector(3 downto 0) := (others => '0');
    signal FlagsWrite_sig : std_logic := '0';
    signal MemWrite_sig   : std_logic := '0';
    signal MemtoReg_sig   : std_logic := '0';
    signal Flags_sig      : std_logic_vector(3  downto 0) := (others => '0');
    signal PC_sig         : std_logic_vector(31 downto 0) := (others => '0');
    signal Instr_sig      : std_logic_vector(31 downto 0) := (others => '0');
    signal ALUResult_sig  : std_logic_vector(31 downto 0) := (others => '0');
    signal WriteData_sig  : std_logic_vector(31 downto 0) := (others => '0');
    signal Result_sig     : std_logic_vector(31 downto 0) := (others => '0');

begin
    DataPath_module_inst: Data_Path
    port map(
    CLK         => CLK,
    RESET       => RESET,
    PCWrite     => PCWrite_sig,
    IRWrite     => IRWrite_sig,
    PCSrc       => PCSrc_sig,
    RegSrc      => RegSrc_sig,
    RegWrite    => RegWrite_sig,
    MAWrite     => MAWrite_sig,
    ImmSrc      => ImmSrc_sig,
    ALUSrc      => ALUSrc_sig,
    ALUControl  => ALUControl_sig,
    FlagsWrite  => FlagsWrite_sig,
    MemWrite    => MemWrite_sig,
    MemtoReg    => MemtoReg_sig,
    Flags       => Flags_sig,
    PC          => PC_sig,
    Instr       => Instr_sig,
    ALUResult   => ALUResult_sig,
    WriteData   => WriteData_sig,
    Result      => Result_sig
    );


    Control_Module_inst: Control
    port map(
    CLK         => CLK,
    RESET       => RESET,
    IR          => Instr_sig,
    SR          => Flags_sig,
    RegSrc      => RegSrc_sig,
    ALUSrc      => ALUSrc_sig,
    MemtoReg    => MemtoReg_sig,
    ALUControl  => ALUControl_sig,
    ImmSrc      => ImmSrc_sig,
    IRWrite     => IRWrite_sig,
    RegWrite    => RegWrite_sig,
    MAWrite     => MAWrite_sig,
    MemWrite    => MemWrite_sig,
    FlagsWrite  => FlagsWrite_sig,
    PCSrc       => PCSrc_sig,
    PCWrite     => PCWrite_sig
    );

    --Processor Output Signals Assertion
    PC         <= PC_sig;
    instr      <= Instr_sig;
    ALUResult  <= ALUResult_sig;
    WriteData  <= WriteData_sig;
    Result     <= Result_sig;

end Structural;
```



Processor RTL, διακρίνονται καθαρά οι βασικές μονάδες της δομής, τα σήματα εισόδου/εξόδου καθώς και τα σήματα επικοινωνίας μεταξύ των μονάδων

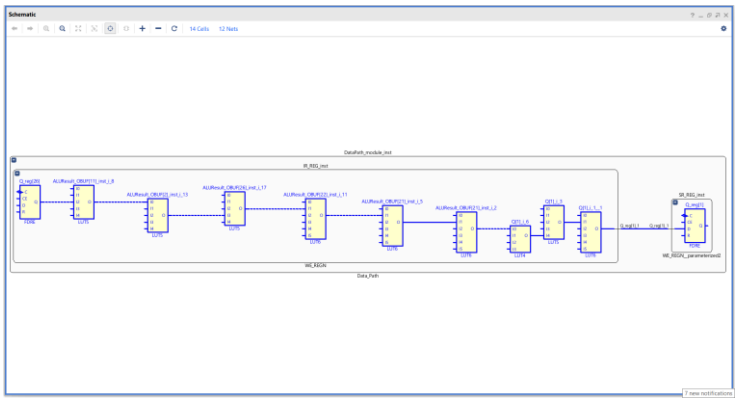
1.9 Εύρεση της μέγιστης συχνότητας λειτουργίας του επεξεργαστή.

Η πιο ακριβής ανάλυση χρονισμού που έχουμε διαθέσιμη είναι εκείνη που πραγματοποιεί το Vivado IDE στο implemented design model, επομένως πατάμε run implementation και στη συνέχεια ανοίγουμε το timing report.



Implementation Timing Report

Από το timing summary βλέπουμε ότι το critical path του κυκλώματος παίρνει χρόνο 5,54 ns. Δεδομένου ότι το critical path ορίζει και τη μεγαλύτερη δυνατή καθυστέρηση καταλαβαίνουμε ότι η μέγιστη συχνότητα λειτουργίας δε μπορεί να ξεπερνάει τα 1/5,54ns = 180 MHz.



Critical Path (5,54 ns)

2. Επαλήθευση της ορθής σχεδίασης και λειτουργίας του επεξεργαστή (processor).

2.1 Στο σημείο αυτό παρουσιάζεται το πρόγραμμα σε συμβολική γλώσσα αρχιτεκτονικής ARM που χρησιμοποιήθηκε για την επαλήθευση της ορθής σχεδίασης και λειτουργίας του επεξεργαστή.

```
MAIN_PROGRAM:
MOV R0, #5; R0 = 5
MOV R1, #8; R1 = 8
ADD R2, R1, R0; R2 = 5+8 = 13
SUB R3, R1, R0; R3 =8-5 = 3
MOV R3, R2; R3 = 13
STR R3, [R0]; DataMemory[5] = 13
LDR R4, [R0]; R4 = DataMemory[5] , R4 = 13
MOV R5, #15; R5 = 15
LSR R0, R5, #2;
ASR R0, R5, #2;
MOV R5, #0; R5 = 0
MOV R9, #0x000000FF; R9 = 255
MOV R0, R0
B MAIN_PROGRAM
```

Το πρόγραμμα αυτό ανέδειξε κάποια προβλήματα που υπήρχαν στη σχεδίαση. Αρχικά δε δούλευαν οι εντολές μνήμης STR και LDR. Μου πήρε αρκετή ώρα να αντιληφθώ ότι το πρόβλημα ήταν ότι κατά την εκτέλεση της STR η FSM ύστερα απ' την κατάσταση S2a δεν προχωρούσε στην S4d (η οποία ενεργοποιεί και το σήμα έγκρισης εγγραφής της μνήμης δεδομένων) αλλά περνούσε εσφαλμένα στην S3. Αυτό συνέβαινε διότι το σήμα L που καθορίζει και τη σωστή μετάβαση έπαιρνε την τιμή IR[24] αντί για την IR[20]. Το πρόβλημα διορθώθηκε πρόχειρα δεδομένης της έλλειψης χρόνου. Στη συνέχεια όμως αναδείχθηκαν κι άλλα προβλήματα σχετικά με τις εντολές LSR και ASR οι οποίες δε δίνουν τα αναμενόμενα αποτελέσματα, δυστυχώς δεν προλαβαίνω να το διορθώσω.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity ROM is
6  Generic( N : positive := 6; -- address length
7           M : positive := 32); -- data word length
8  Port (
9      ADDR     :   in  std_logic_vector(N-1 downto 0);
10     DATA_OUT :   out std_logic_vector(M-1 downto 0));
11 end ROM;
12
13 architecture Behavioral of ROM is
14     type ROM_array is array (0 to 2**N-1)
15       of std_logic_vector(M-1 downto 0);
16     constant ROM : ROM_array := (
17       X"E3A00005", X"E3A01008", X"E0812000", X"E0413000",
18       X"E1A03002", X"E5803000", X"E5904000", X"E3A0500F",
19       X"E1A00125", X"E1A00145", X"E3A05000", X"E3A090FF",
20       X"E1A00000", X"EAFFFFF1", X"00000000", X"00000000",
21       X"00000000", X"00000000", X"00000000", X"00000000",
22       X"00000000", X"00000000", X"00000000", X"00000000",
23       X"00000000", X"00000000", X"00000000", X"00000000",
24       X"00000000", X"00000000", X"00000000", X"00000000",
25       X"00000000", X"00000000", X"00000000", X"00000000",
26       X"00000000", X"00000000", X"00000000", X"00000000",
27       X"00000000", X"00000000", X"00000000", X"00000000",
28       X"00000000", X"00000000", X"00000000", X"00000000",
29       X"00000000", X"00000000", X"00000000", X"00000000",
30       X"00000000", X"00000000", X"00000000", X"00000000",
31       X"00000000", X"00000000", X"00000000", X"00000000",
32       X"00000000", X"00000000", X"00000000", X"00000000");
33 begin
34     DATA_OUT <= ROM(to_integer(unsigned(ADDR)));
35 end Behavioral;
```

Αριστερά φαίνεται η μνήμη ROM (instruction memory) που υλοποιεί το παραπάνω πρόγραμμα assembly.

Για το compile του προγράμματος χρησιμοποιήθηκε ο flat assembler for ARM version 1.43 και για την κωδικοποίηση σε δεκαεξαδικό σύστημα ο HxD editor.

Για την επαλήθευση της εκτέλεσης των παραπάνω εντολών θα παρατηρήσουμε τις τιμές που αποθηκεύονται στο εσωτερικό σήμα του Register File (RF array) , δηλαδή τις τιμές που αποθηκεύουν οι καταχωρητές του αρχείου καταχωρητών, καθώς και το εσωτερικό σήμα current state της FSM του Control.

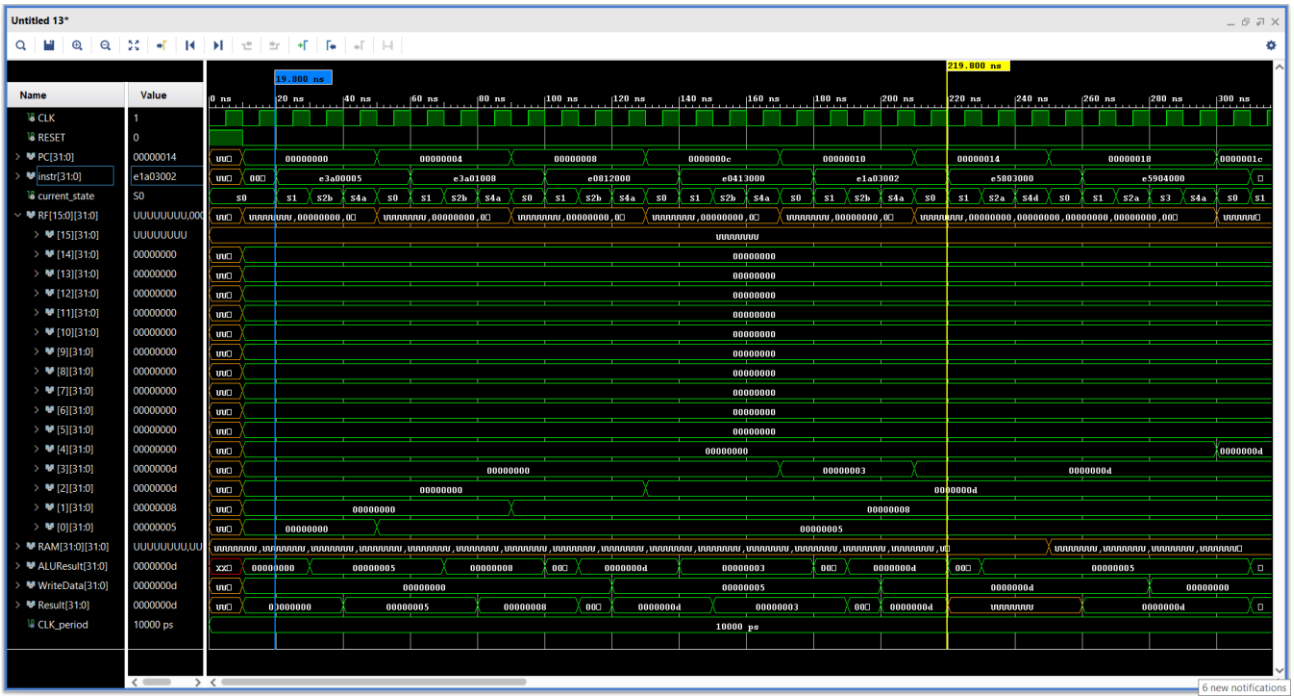
MOV R0, #5 x"E3A00005"

MOV R1, #8 x"E3A01008"

ADD R2, R1, R0 x"E0812000"

SUB R3, R1, R0 x"E0413000"

MOV R3, R2 x"E1A03002"

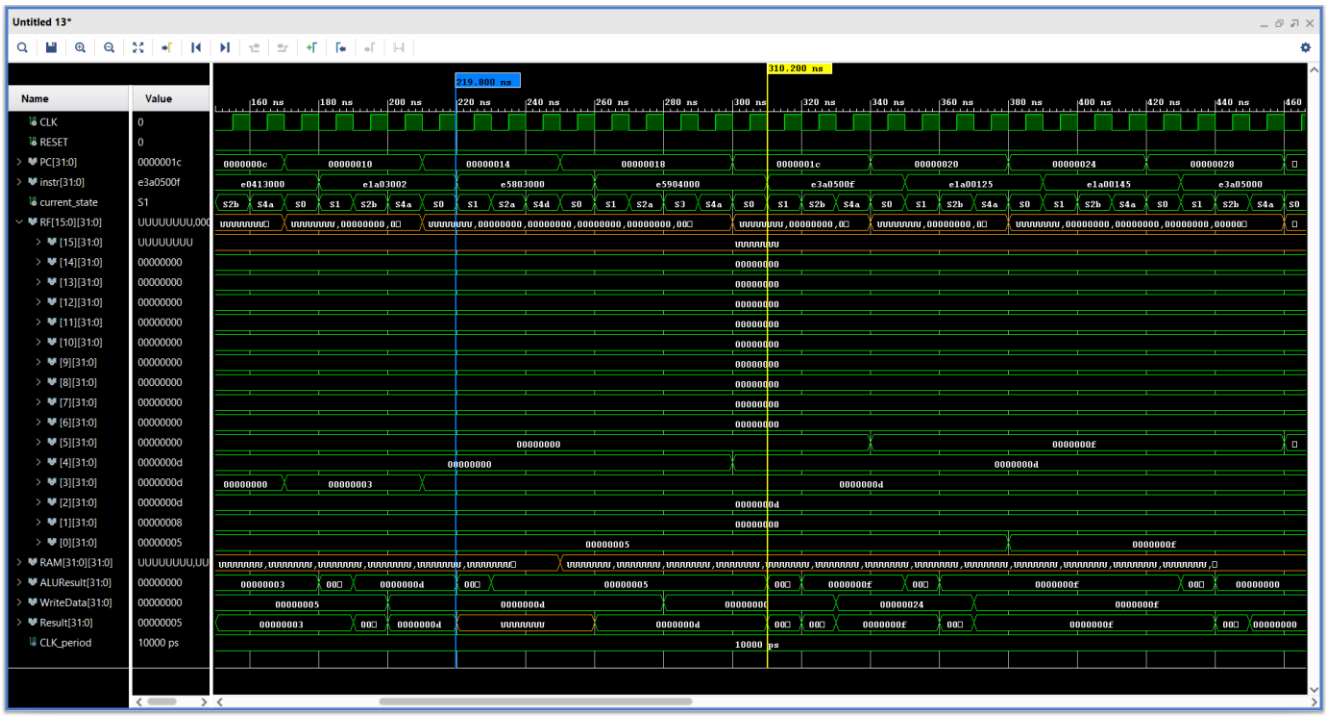


Τα όρια έχουν μπει για να ξεχωρίσουμε τις 5 παραπάνω εντολές επεξεργασίας δεδομένων. Φαίνεται ότι το current state σήμα του FSM παίρνει με τη σειρά τιμές οι οποίες είναι αποδεκτές σύμφωνα με τον πίνακα της ενότητας 4.3.3 του φυλλαδίου. Συγκεκριμένα το state γίνεται με τη σειρά S0 -> S1 -> S2b -> S4a -> S0 σε όλες τις παραπάνω εντολές.

ΤΟ RF array επίσης παίρνει τις σωστές τιμές. Δηλαδή ο register R0 παίρνει την τιμή '5' μόλις ολοκληρώνεται το 3ο clock cycle απ' τη στιγμή που διαβάζεται η εντολή απ' τον Instruction Register του datapath. Η πράξεις ADD / SUBB υλοποιούνται επίσης σωστά καθώς οι registers R2, R3 αποκτούν τις τιμές 13 και 3 αντίστοιχα.

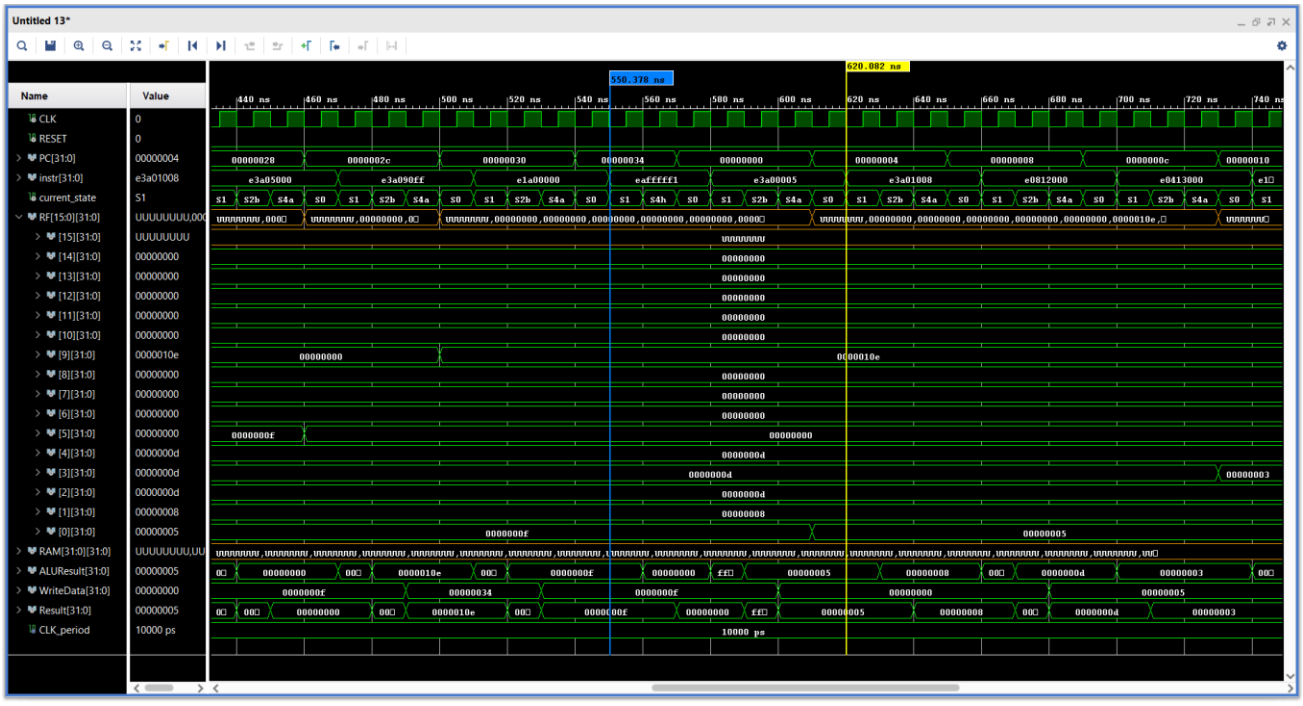
STR R3, [R0] x"E5803000"

LDR R4, [R0] x"E5904000"



Το διάγραμμα επικεντρώνει στις 2 εντολές μνήμης. Βλέπουμε ότι όταν εκτελείται η εντολή LDR R4, R[0] ο καταχωρητής R4 παίρνει σωστά την τιμή 13, πράγμα που σημαίνει ότι η μνήμη δεδομένων έχει εγγραφεί σωστά στην προηγούμενη εντολή.

Επίσης το current state signal πέρνει διαδοχικά τις τιμές S0->S1->S4d->S0 για την STR και S0 -> S1 -> S2a -> S3 -> S4a -> S0 για την LDR.



Τέλος, παρατηρούμε ότι κατά την εκτέλεση της τελευταίας εντολής (B MAIN_PROGRAM) επιστρέφουμε ακριβώς στην αρχή του προγράμματος κι οι εντολές ξεκινούν να εκτελούνται πάλι με τη σειρά. Αυτό επιβεβαιώνει ότι η εντολή διακλάδωσης λειτουργεί σωστά κάτι που φαίνεται κι απ' τον μετρητή προγράμματος ο οποίος μηδενίζεται και ξεκινά απ' την αρχή.