



ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ  
Εθνικό και Καποδιστριακό  
Πανεπιστήμιο Αθηνών

**Μάθημα:**

Σχεδίαση Ψηφιακών Συστημάτων

**Διδάσκοντες:**

Νεκτάριος Κρανίτης

Διονύσης Βασιλόπουλος

**Τίτλος εργασίας:**

«Project 1 2023-24»

**Ονοματεπώνυμο Φοιτήτριας:**

Χριστοφιλοπούλου Βασιλική (1115202000216)

Σεπτέμβριος 2024

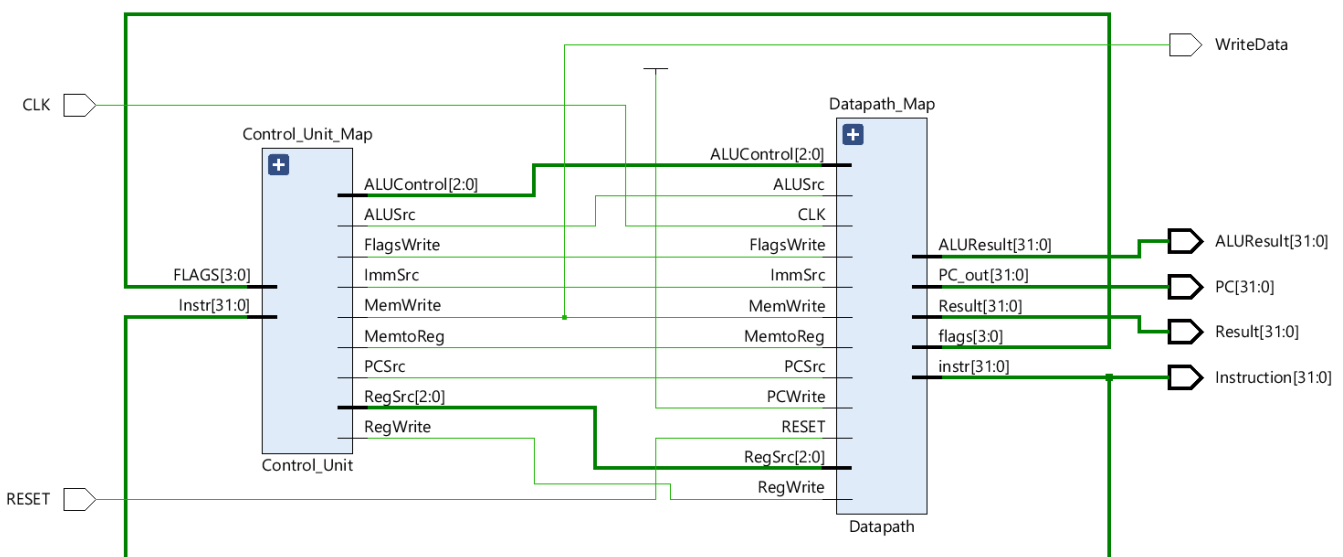
# Contents

Chapter 1: Εισαγωγή .....	3
Chapter 2: Σύνολο εντολών .....	4
Chapter 3: Διαδρομή Δεδομένων (Data Path) .....	6
3. 1 Datapath Componets .....	6
3. 2 Περιγραφή datapath.....	7
3.2.1 RTL διάγραμμα datapath .....	13
3. 3 Περιγραφή Components.....	14
3.3.1 Step 1 : Instruction_Memory, PC, PCPlus4.....	14
3.3.2 Step 2: Extend, Register_File, Mux2To1, PCPlus8 .....	15
3.3.3 Step 3: ALU, Status_Register, Mux2To1 .....	17
3.3.4 Step 4: Data_Memory .....	19
3.3.5 Step 5: MuxForStep5.....	20
Chapter 4: Μονάδα ελέγχου (Control Unit).....	21
4. 1 Control Unit Componets .....	21
4. 2 Περιγραφή control unit .....	21
4.2.1 RTL διάγραμμα control unit.....	24
4. 3 Περιγραφή Components.....	24
4.3.1 Step 1: InstrDec.....	24
4.3.2 Step 2: Logic (WELogic & PCLogic) .....	26
4.3.3 Step 3: CONDLlogic .....	28
Chapter 5: Επεξεργαστής – Processor .....	31
5. 1 Δομή επεξεργαστή.....	31
5. 2 Μέγιστη συχνότητα - Χειρότερη κρίσιμη διαδρομή - Χειρότερη σύντομη διαδρομή. ....	34
Chapter 6: Προσομοίωση .....	37
6. 1 Δοκιμαστικός κώδικας .....	37
6. 2 Κώδικας προσομοίωσης .....	38
6. 3 Behavioral Simulation .....	39
6. 4 Post-Synthesis Simulation .....	40
6. 5 Post-Implementation Simulation .....	40
6. 6 Synthesis και Implementation .....	41

# Chapter 1: Εισαγωγή

Στόχος της παρούσας εργασίας είναι η δημιουργία της μικροαρχιτεκτονικής ενός απλοποιημένου επεξεργαστή ενός κύκλου αρχιτεκτονικής ARM σε τεχνολογία FPGA με τη χρήση του εργαλείου Vivado IDE (Integrated Development Environment) της Xilinx.

Ο επεξεργαστής processor αποτελείται από 2 τμήματα, την διαδρομή δεδομένων (datapath), η οποία επεξεργάζεται τα δεδομένα στις κατάλληλες αριθμητικές και λογικές μονάδες και την μονάδα ελέγχου (control unit), η οποία αποκωδικοποιεί την εντολή και παράγει τα κατάλληλα σήματα ελέγχου.



RTL -Schematic 1

Συνεπώς, ο επεξεργαστής αποτελείται από δυο components (Control και Data\_Path), όπου το καθένα αποτελείται από τα δικά του δομικά στοιχεία, και τα σήματα εξόδου τη μονάδας ελέγχου αποτελούν την είσοδο της διαδρομής δεδομένων, όπως παρουσιάζονται στην παραπάνω εικόνα προκειμένου να εκτελούνται σωστά οι εντολές.

## Chapter 2: Σύνολο εντολών

Στην συγκεκριμένη εργασία υλοποιήθηκαν όλες οι ζητούμενες εντολές που παρουσιάζονται στην ενότητα 1-3 της εκφώνησης. Πιο αναλυτικά έχουμε τις ακόλουθες:

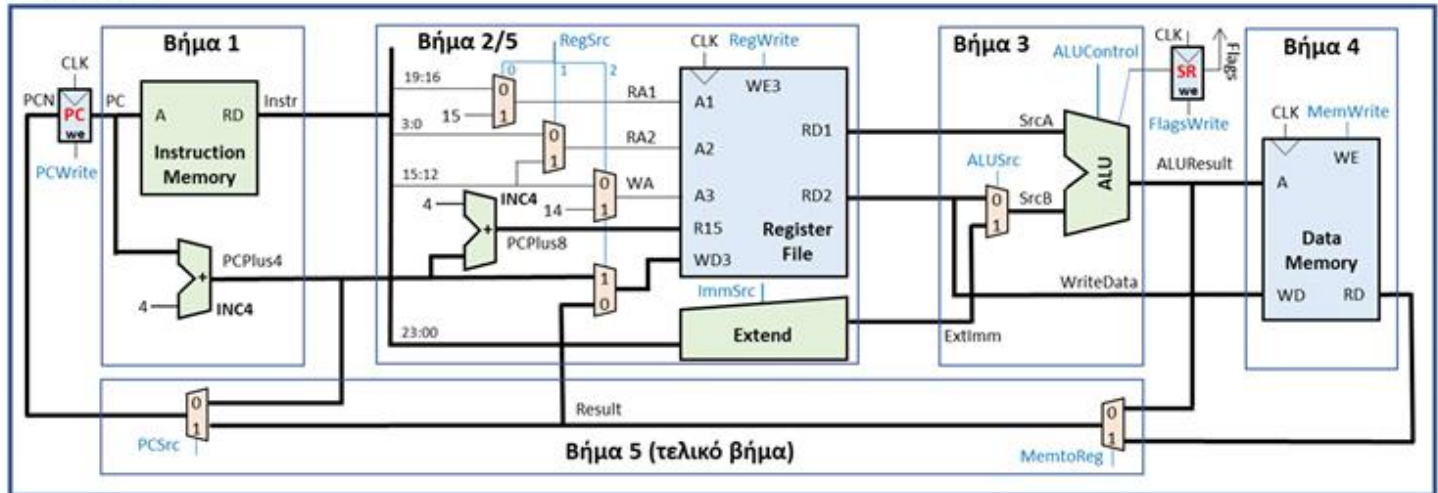
- ❖ **LDR Rd, [Rn,#imm12]:** Τοποθετεί τα περιεχόμενα της θέσης μνήμης στη διεύθυνση  $Rn + \#imm12$  στον καταχωρητή Rd.
- ❖ **LDR Rd, [Rn,#-imm12]:** Τοποθετεί τα περιεχόμενα της θέσης μνήμης στη διεύθυνση  $Rn - \#imm12$  στον καταχωρητή Rd
- ❖ **STR Rd, [Rn,#imm12]:** Τοποθετεί τα περιεχόμενα του καταχωρητή Rd στη θέση μνήμης στη διεύθυνση  $Rn + \#imm12$ .
- ❖ **STR Rd, [Rn,#-imm12]:** Τοποθετεί τα περιεχόμενα του καταχωρητή Rd στη θέση μνήμης στη διεύθυνση  $Rn - \#imm12$ .
- ❖ **ADD Rd, Rn, #imm8:** Εκτελεί την πράξη πρόσθεσης μεταξύ των περιεχομένων του καταχωρητή Rn και του #imm8, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **ADD Rd, Rn, Rm:** Εκτελεί την πράξη πρόσθεσης μεταξύ των περιεχομένων του καταχωρητή Rn και των περιεχομένων του καταχωρητή Rm, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **SUB Rd, Rn, #imm8:** Εκτελεί την πράξη αφαίρεσης μεταξύ των περιεχομένων του καταχωρητή Rn και του #imm8, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **SUB Rd, Rn, Rm:** Εκτελεί την πράξη αφαίρεσης μεταξύ των περιεχομένων του καταχωρητή Rn και των περιεχομένων του καταχωρητή Rm, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **AND Rd, Rn, #imm8:** Εκτελεί την πράξη AND (λογική σύζευξη) μεταξύ των περιεχομένων του καταχωρητή Rn και του #imm8, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **AND Rd, Rn, Rm:** Εκτελεί την πράξη AND (λογική σύζευξη) μεταξύ των περιεχομένων του καταχωρητή Rn και των περιεχομένων του καταχωρητή Rm, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **ORR Rd, Rn, #imm8:** Εκτελεί την πράξη OR (λογική διάζευξη) μεταξύ των περιεχομένων του καταχωρητή Rn και του #imm8, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **ORR Rd, Rn, Rm:** Εκτελεί την πράξη OR (λογική διάζευξη) μεταξύ των περιεχομένων του καταχωρητή Rn και των περιεχομένων του καταχωρητή Rm, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **EOR Rd, Rn, #imm8:** Εκτελεί την πράξη XOR (αποκλειστική λογική διάζευξη) μεταξύ των περιεχομένων του καταχωρητή Rn και του #imm8, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.

- ❖ **EOR Rd, Rn, Rm:** Εκτελεί την πράξη XOR (αποκλειστική λογική διάζευξη) μεταξύ των περιεχομένων του καταχωρητή Rn και των περιεχομένων του καταχωρητή Rm, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **CMP Rn, #imm8:** Ενημερώνει τις σημαίες για την πράξη αφαίρεσης μεταξύ των περιεχομένων του καταχωρητή Rn και του #imm8.
- ❖ **CMP Rn, Rm:** Ενημερώνει τις σημαίες για την πράξη αφαίρεσης μεταξύ των περιεχομένων του καταχωρητή Rn και των περιεχομένων του καταχωρητή Rm.
- ❖ **MOV Rd, #imm8:** Τοποθετεί το άμεσο δεδομένο #imm8 στον καταχωρητή Rd.
- ❖ **MOV Rd, Rm:** Τοποθετεί τα περιεχόμενα του καταχωρητή Rm στον καταχωρητή Rd.
- ❖ **NOP:** Ουσιαστικά ισοδυναμεί με την εντολή MOV R0, R0. Σημειώνεται ότι στον FASMARM, η εντολή NOP δεν μεταφράζεται έτσι, οπότε μπορεί να χρειαστεί να γραφεί ως MOV R0, R0 όπως φαίνεται και στο παράδειγμα αργότερα.
- ❖ **MVN Rd, #imm8:** Τοποθετεί στον καταχωρητή Rd το αντίθετο του άμεσου δεδομένου #imm8.
- ❖ **MVN Rd, Rm:** Τοποθετεί στον καταχωρητή Rd το αντίθετο των περιεχομένων του καταχωρητή Rm.
- ❖ **LSL Rd, Rm, #shamt5:** Κάνει αριστερή ολίσθηση των περιεχομένων του καταχωρητή Rm κατά #shamt5 θέσεις και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **LSR Rd, Rm, #shamt5:** Κάνει δεξιά λογική ολίσθηση των περιεχομένων του καταχωρητή Rm κατά #shamt5 θέσεις και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.
- ❖ **B Label:** Μετακινεί τον καταχωρητή προγράμματος (PC) στη διεύθυνση που αντιστοιχεί στην ετικέτα Label.
- ❖ **BL Label:** Μετακινεί τον καταχωρητή προγράμματος (PC) στη διεύθυνση που αντιστοιχεί στην ετικέτα Label και αποθηκεύει την τρέχουσα διεύθυνση στον καταχωρητή R14.

## Chapter 3: Διαδρομή Δεδομένων (Data Path)

Για την δημιουργία του datapath ακολουθείται η λογική πορεία που περιγράφεται από την εκφώνηση και την παρακάτω εικόνα.

Datapath



### 3. 1 Datapath Componets

Πιο αναλυτικά, αποτελείται από τα ακόλουθα δομικά στοιχεία:

1. Instruction\_Memory --step 1
2. PC
3. PCPlus4
4. Extend --step 2
5. Register\_File
6. Mux2To1
7. ALU --step 3
8. Status\_Register
9. Data\_Memory --step 4
10. MuxForStep5 --step 5

## 3. 2 Περιγραφή datapath

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;    -- For counting and addition
use IEEE.NUMERIC_STD.ALL;           -- For type conversions

entity Datapath is
  generic (
    M : positive := 32              -- data word length);
  Port (
    -- STEP 1:
    CLK      : in STD_LOGIC;
    RESET    : in STD_LOGIC;
    PCWrite  : in STD_LOGIC;
    PC_out   : out STD_LOGIC_VECTOR (M-1 downto 0);
    instr    : out STD_LOGIC_VECTOR (M-1 downto 0);

    -- STEP 2:
    RegSrc   : in STD_LOGIC_VECTOR (2 downto 0);
    RegWrite : in STD_LOGIC;
    ImmSrc   : in STD_LOGIC;

    -- Step 3:
    ALUSrc   : in STD_LOGIC;
    ALUControl : in STD_LOGIC_VECTOR (2 downto 0);
    FlagsWrite : in STD_LOGIC;
    ALUResult : out STD_LOGIC_VECTOR (M-1 downto 0);
    flags     : out STD_LOGIC_VECTOR (3 downto 0);

    -- STEP 4:
    MemWrite : in STD_LOGIC;

    -- STEP 5:
    MemtoReg : in STD_LOGIC;
    PCSrc     : in STD_LOGIC;
    Result    : out STD_LOGIC_VECTOR (M-1 downto 0));
end Datapath;

architecture Structural of Datapath is

  -- STEP 1: Instruction Memory
  component Instruction_Memory is
    generic (
      M : positive := 32; -- data word length
      N : positive := 6 -- address length);
    Port (
      ADDR      : in STD_LOGIC_VECTOR(N-1 downto 0);
      DATA_OUT : out STD_LOGIC_VECTOR(M-1 downto 0));
```

```

end component;

component PC is
generic(
    M : positive := 32 -- data word length);
Port (
    CLK      : in std_logic;
    RESET    : in std_logic;
    WE       : in std_logic;
    D        : in STD_LOGIC_VECTOR(M-1 downto 0);
    Q        : out STD_LOGIC_VECTOR(M-1 downto 0));
end component;

component PCPlus4 is
generic(
    M : positive := 32 -- data word length);
Port (
    PC       : in  STD_LOGIC_VECTOR (M-1 downto 0);
    PCPlus4 : out  STD_LOGIC_VECTOR (M-1 downto 0));
end component;

-- STEP 2: Register File
component Extend is
Port (
    ImmSrc  : in STD_LOGIC;
    X       : in STD_LOGIC_VECTOR (23 downto 0);
    Y       : out STD_LOGIC_VECTOR (31 downto 0));
end component;

component Register_File is
generic (
    N : positive := 4; -- address length
    M : positive := 32 -- data word length);
port (
    CLK : in STD_LOGIC;
    WE  : in STD_LOGIC;
    A1  : in STD_LOGIC_VECTOR (N-1 downto 0);
    A2  : in STD_LOGIC_VECTOR (N-1 downto 0);
    A3  : in STD_LOGIC_VECTOR (N-1 downto 0);
    R15 : in STD_LOGIC_VECTOR (M-1 downto 0);    -- R15 = PC
    WD  : 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 Mux2To1 is
generic (
    M : positive := 4 -- data word length);

```



```

Port (
  A : in STD_LOGIC_VECTOR (M-1 downto 0);
  B : in STD_LOGIC_VECTOR (M-1 downto 0);
  S : in STD_LOGIC;
  Y : out STD_LOGIC_VECTOR (M-1 downto 0));
end component;

--STEP 3: Alu
component ALU is
generic (
  M : positive := 32; -- data word length
  N : positive := 3 -- command word length);
Port (
  SRC_A      : in STD_LOGIC_VECTOR (M-1 downto 0);
  SRC_B      : in STD_LOGIC_VECTOR (M-1 downto 0);
  ALU_CONTROL : in STD_LOGIC_VECTOR (N-1 downto 0);
  SA5        : in STD_LOGIC_VECTOR (4 downto 0);
  ALU_RESULT  : out STD_LOGIC_VECTOR (M-1 downto 0);
  ALU_Flags   : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component Status_Register is
generic(
  N : positive := 4 -- data word length);
Port (
  CLK      : in std_logic;
  RESET    : in std_logic;
  WE       : in std_logic;
  D        : in STD_LOGIC_VECTOR (N-1 downto 0);
  Q        : out STD_LOGIC_VECTOR (N-1 downto 0));
end component;

--STEP 4: Data Memory
Component 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 component;

component MuxForStep5 is
generic (
  M : positive := 32 -- data word length);

```

```

Port (
    A : in STD_LOGIC_VECTOR (M-1 downto 0);
    B : in STD_LOGIC_VECTOR (M-1 downto 0);
    S : in STD_LOGIC;
    Y : out STD_LOGIC_VECTOR (M-1 downto 0));
end component;

-----
-- STEP 1: Instruction Memory
signal Q_PC      : STD_LOGIC_VECTOR (M-1 downto 0);
signal DATA_OUT_IM : STD_LOGIC_VECTOR (M-1 downto 0);
signal PCPlus4_P4 : STD_LOGIC_VECTOR (M-1 downto 0);

-- STEP 2: Register File
signal RD1_RF      : STD_LOGIC_VECTOR (M-1 downto 0);
signal RD2_RF      : STD_LOGIC_VECTOR (M-1 downto 0);
signal Y_M0         : STD_LOGIC_VECTOR (3 downto 0);
signal Y_M1         : STD_LOGIC_VECTOR (3 downto 0);
signal Y_M2         : STD_LOGIC_VECTOR (3 downto 0);
signal Y_M3         : STD_LOGIC_VECTOR (M-1 downto 0);
signal Y_EX         : STD_LOGIC_VECTOR (M-1 downto 0);
signal PCPlus4_P8   : STD_LOGIC_VECTOR (M-1 downto 0);

--STEP 3: Alu
signal ALU_RESULT_AL : STD_LOGIC_VECTOR (M-1 downto 0);
signal ALU_Flags_AL  : STD_LOGIC_VECTOR (3 downto 0);
signal Q_SR          : STD_LOGIC_VECTOR (3 downto 0);
signal Y_M4          : STD_LOGIC_VECTOR (M-1 downto 0);

-- STEP 4: Data Memory
signal DATA_OUT_DM : STD_LOGIC_VECTOR (M-1 downto 0);

-- STEP 5 : Results
signal Y_M5 : STD_LOGIC_VECTOR (M-1 downto 0);
signal Y_M6 : STD_LOGIC_VECTOR (M-1 downto 0);
-----
begin

-- STEP 1: Instruction Memory
Instruction_Memory_Map: Instruction_Memory
port map(
    ADDR      => Q_PC(7 downto 2),
    DATA_OUT  => DATA_OUT_IM);

PC_Map: PC
port map(
    CLK      => CLK,
    RESET    => RESET,

```

```

WE      => PCWrite,
D       => Y_M6,
Q       => Q_PC);

PCPlus4_Map: PCPlus4
port map(
    PC      => Q_PC,
    PCPlus4 => PCPlus4_P4);

-- STEP 2: Register File
Extend_Map: Extend
port map(
    ImmSrc  => ImmSrc,
    X       => DATA_OUT_IM(23 downto 0),
    Y       => Y_EX);

Register_File_Map: Register_File
port map(
    CLK => CLK,
    WE  => RegWrite,
    A1  => Y_M0,
    A2  => Y_M1,
    A3  => Y_M2,
    R15 => PCPlus4_P8,
    WD  => Y_M3,
    RD1 => RD1_RF,
    RD2 => RD2_RF);

Mux2To1_Map0: Mux2To1  -- RegSrc(0)
port map(
    A => DATA_OUT_IM(19 downto 16),
    B => "1111",
    S => RegSrc(0),
    Y => Y_M0);

Mux2To1_Map1: Mux2To1  -- RegSrc(1)
port map(
    A => DATA_OUT_IM(3 downto 0),
    B => DATA_OUT_IM(15 downto 12),
    S => RegSrc(1),
    Y => Y_M1);

Mux2To1_Map2: Mux2To1  -- RegSrc(2)
port map(
    A => DATA_OUT_IM(15 downto 12),
    B => "1110",
    S => RegSrc(2),
    Y => Y_M2);

```

```

PCPlus8_Map : PCPlus4
port map(
    PC      => PCPlus4_P4,
    PCPlus4 => PCPlus4_P8);

--STEP 3: Alu
ALU_Map: ALU
port map(
    SRC_A      => RD1_RF,
    SRC_B      => Y_M4,
    ALU_CONTROL => ALUCONTROL,
    SA5        => Y_EX(11 downto 7),
    ALU_RESULT  => ALU_RESULT_AL,
    ALU_Flags   => ALU_Flags_AL);

Status_Register_Map: Status_Register
port map(
    CLK      => CLK,
    RESET    => RESET,
    WE       => FlagsWrite,
    D        => ALU_Flags_AL,
    Q        => Q_SR);

MuxForStep5_Map_SrcB: MuxForStep5
port map(
    A => RD2_RF,
    B => Y_EX,
    S => ALUSrc,
    Y => Y_M4);

-- STEP 4: Data Memory
Data_Memory_Map: Data_Memory
port map(
    CLK      => CLK,
    WE       => MemWrite,
    ADDR     => ALU_RESULT_AL(4 downto 0),
    DATA_IN  => RD2_RF,
    DATA_OUT => DATA_OUT_DM);

-- STEP 5 : Results
MuxForStep5_Map_WD: MuxForStep5      --For step 2, output is WD
port map(
    A => Y_M5,
    B => PCPlus4_P4,
    S => RegSrc(2),
    Y => Y_M3);

MuxForStep5_Map_MREG: MuxForStep5      --MemtoReg

```

```

port map(
    A => ALU_RESULT_AL,
    B => DATA_OUT_DM,
    S => MemtoReg,
    Y => Y_M5);

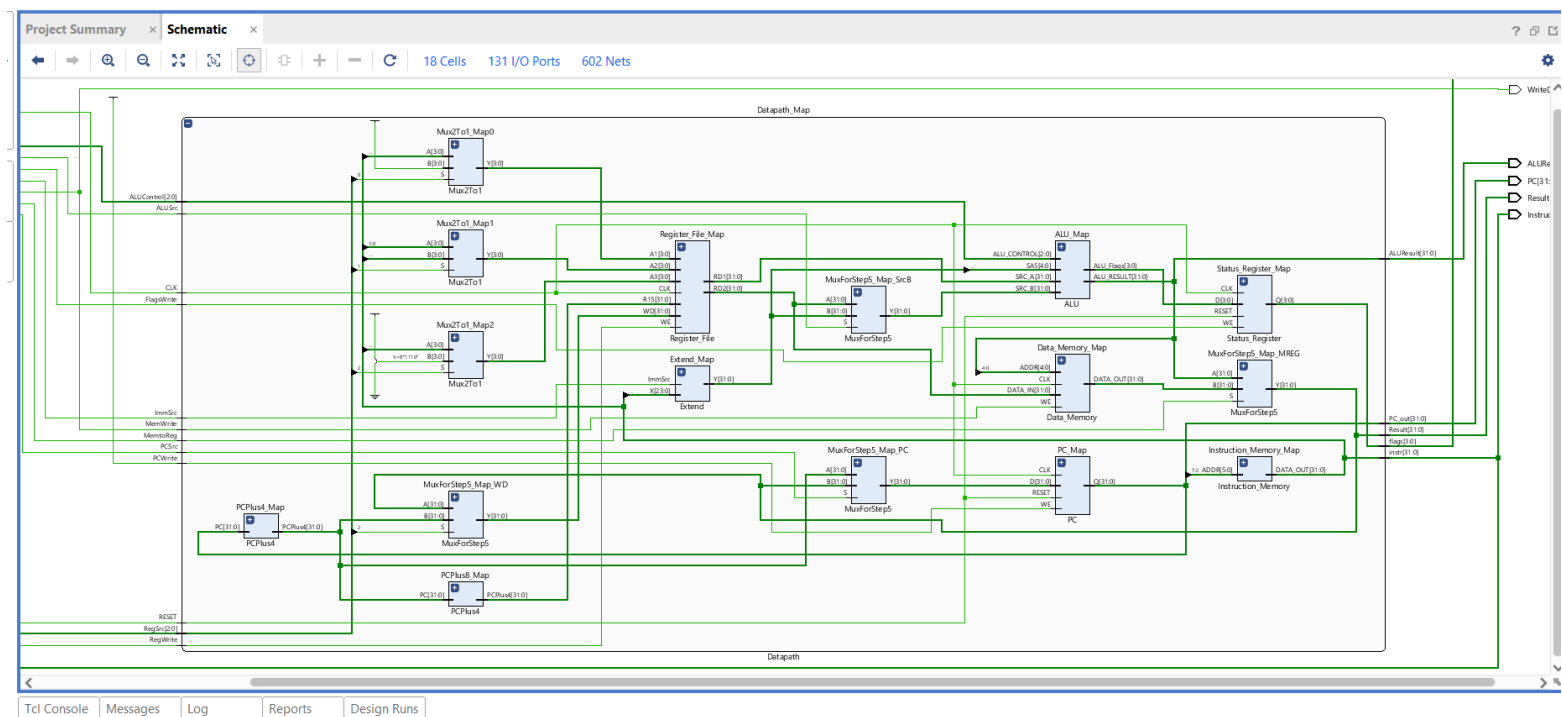
MuxForStep5_Map_PC: MuxForStep5
port map(
    A => PCPlus4_P4,
    B => Y_M5,
    S => PCSrc,
    Y => Y_M6);

PC_out      <= Q_PC;
instr       <= DATA_OUT_IM;
ALUResult   <= ALU_RESULT_AL;
Result      <= Y_M5;
flags       <= Q_SR;

end Structural;

```

### 3.2.1 RTL διάγραμμα datapath



RTL -Schematic 2 : Datapath

Για να μπορέσουμε να κατανοήσουμε το παραπάνω γράφημα, χρειάζεται να περιγράψουμε και να αναλύσουμε την χρήση κάθε οντότητας.

## 3. 3 Περιγραφή Components

### 3.3.1 Step 1 : Instruction\_Memory, PC, PCPlus4

Σε αυτό το component αποθηκεύουμε τις εντολές. Ειδικότερα, όταν εκτελείται μια εντολή προσκομίζεται στο instruction memory όπου διαβάζει το σήμα ADDR από τον πίνακα ROM\_Array και το δίνει ως έξοδο στο DATA\_OUT.

```
entity Instruction_Memory is
  generic (
    M : positive := 32;    -- data word length
    N : positive := 6      -- address length
  );
  Port (
    ADDR      : in STD_LOGIC_VECTOR(N-1 downto 0);
    DATA_OUT  : out STD_LOGIC_VECTOR(M-1 downto 0)
  );
end Instruction_Memory;
```

Entity 1 : Instruction Memory

```
entity PC is
  generic(
    M : positive := 32 -- data word length
  );
  Port (
    CLK      : in std_logic;
    RESET    : in std_logic;
    WE       : in std_logic;
    D        : in STD_LOGIC_VECTOR(M-1 downto 0);
    Q        : out STD_LOGIC_VECTOR(M-1 downto 0)
  );
end PC;
```

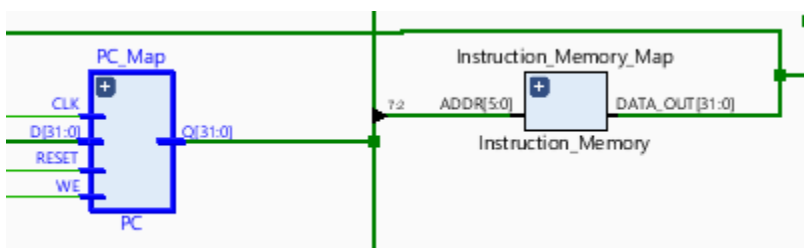
Entity 2 : PC

Έπειτα, έχουμε έναν καταχωρητή ο οποίος κρατάει την διεύθυνση της τρέχουσας εντολής. Κατά την ενεχόμενη ακμή του ρολογιού και εφόσον ισχύει WE = '1', δίνουμε το σήμα ως είσοδο στον PC και την επόμενη ανερχόμενη ακμή τρέχουμε την εντολή(Q <= D;).

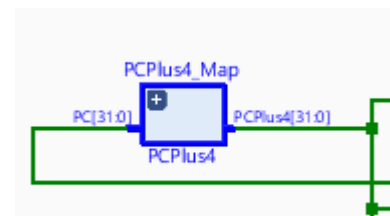
Τέλος, θα χρειαστούμε έναν αθροιστή για να αυξάνουμε τον καταχωρητή PC ώστε να δείχνει στην διεύθυνση της επόμενης εντολής.

```
architecture Behavioral of PCPlus4 is    -- STEP 2-1.3.
begin
  PCPlus4 <= std_logic_vector(unsigned(PC) + 4); -- PC = PC + 4
end Behavioral;
```

Architecture 1 : PCPlus4



RTL-Schematic 3 : Instruction Memory & PC



RTL-Schematic 4 : PCPlus4

### 3.3.2 Step 2: Extend, Register\_File, Mux2To1, PCPlus8

Επιπλέον, χρειαζόμαστε μια μονάδα που να εφαρμόζει επέκταση προσήμου (Extend) σε μια προσημασμένη τιμή 32 bit. Όταν το ImmSrc είναι 0 τότε κάνουμε επέκταση μηδενός από τα 12 bit στα 32 bit. Όταν το ImmSrc είναι 1 τότε κάνουμε επέκταση προσήμου από τα 24 bit στα 32 bit.

```
begin
  process(X, ImmSrc)
  begin
    if (ImmSrc = '0') then
      Y <= std_logic_vector(resize(unsigned(X(11 downto 0)),32));
    else
      Y <= std_logic_vector(resize(signed(X&"00"),32));
    end if;
  end process;
end Behavioral;
```

Architecture 2 : Extend

```
architecture Behavioral of Register_File is -- STEP 2-2.1

  type RF_array is array (0 to 2*N-1)
  of STD_LOGIC_VECTOR (M-1 downto 0);
  signal RF : RF_array;

begin
  process (CLK, A3, WE, WD)
  begin
    if (CLK = '1' and CLK'event) then
      if (WE = '1') then
        RF(to_integer(unsigned(A3))) <= WD;
      end if;
    end if;
  end process;

  with A1 select
    RD1 <= R15 when "1111", --LDR
          RF(to_integer(unsigned(A1))) when others;

  RD2 <= RF(to_integer(unsigned(A2))); --STR(RegSrc =1) or ALU(S)-R(RegSrc =0)

end Behavioral;
```

Architecture 3 : Register File

περιεχόμενο του register που βρίσκεται στη διεύθυνση A2

Προκειμένου να παραχθεί καθένα από τα A1,A2,A3 επιλέγονται τα αντίστοιχα bit από το σήμα εισόδου και δίνονται ως είσοδο σε έναν πολυπλέκτη Mux2To1 με σήμα ελέγχου το RegSrc(0), RegSrc(1), RegSrc(2) αντίστοιχα.

Το αρχείο καταχωρητών (Register File) αποτελείται από 6 εισόδους εκ των οποίων οι τρεις για αριθμούς καταχωρητών A1,A2,A3 και η R15 όπου είναι το PC+8. Υπάρχει και ένα σήμα ελέγχου Reg\_Write (WD). Όταν το WE = '1' το περιεχόμενο του register που βρίσκεται στη διεύθυνση A3 ενημερώνεται με την τιμή του WD.

Με την εντολή with A1 select, ορίζεται ότι αν η διεύθυνση A1 είναι 1111, τότε το RD1 παίρνει την τιμή του R15. Σε κάθε άλλη περίπτωση, διαβάζεται το register στη διεύθυνση A1. Το RD2 διαβάζει το

```
Mux2To1_Map0: Mux2To1 -- RegSrc(0)
port map(
  A => DATA_OUT_IM(19 downto 16),
  B => "1111",
  S => RegSrc(0),
  Y => Y_M0
);

Mux2To1_Map1: Mux2To1 -- RegSrc(1)
port map(
  A => DATA_OUT_IM(3 downto 0),
  B => DATA_OUT_IM(15 downto 12),
  S => RegSrc(1),
  Y => Y_M1
);

Mux2To1_Map2: Mux2To1 -- RegSrc(2)
port map(
  A => DATA_OUT_IM(15 downto 12),
  B => "1110",
  S => RegSrc(2),
  Y => Y_M2
);
```

Port Map 1 : Mux2To1

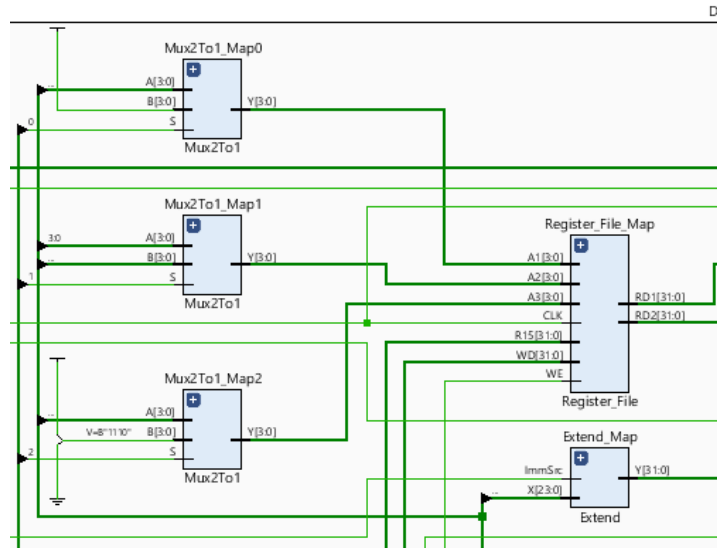
```

PCPlus8_Map : PCPlus4
port map(
    PC      => PCPlus4_P4,
    PCPlus4 => PCPlus4_P8
);

```

Port Map 2 : PCPlus8

Προσθέτουμε ακόμα 4 στο αποτέλεσμα του  $PC + 4$  και το βάζουμε ανεξαρτήτως της τιμής RegWrite στον καταχωρητή 15.



RTL-Schematic 5 : Extend, Register File, Mux2To1



### 3.3.3 Step 3: ALU, Status\_Register, Mux2To1

Η ALU δέχεται δύο αριθμούς και ανάλογα με την τιμή του ALUControl κάνει μία από τις ακόλουθες πράξεις:

1. Πρόσθεση
2. Αφαίρεση
3. Λογικό και
4. Λογικό xor
5. Μεταφορά της δεύτερης εισόδου στο αποτέλεσμα
6. Μεταφορά της δεύτερης εισόδου στο αποτέλεσμα αντεστραμμένης
7. Λογική ολίσθηση στα αριστερά της πρώτης εισόδου κατά όσο υποδεικνύει η δεύτερη είσοδος
8. Λογική ολίσθηση στα δεξιά της πρώτης εισόδου κατά όσο υποδεικνύει η δεύτερη είσοδος

[illegible]

### Architecture 4 : ALU control

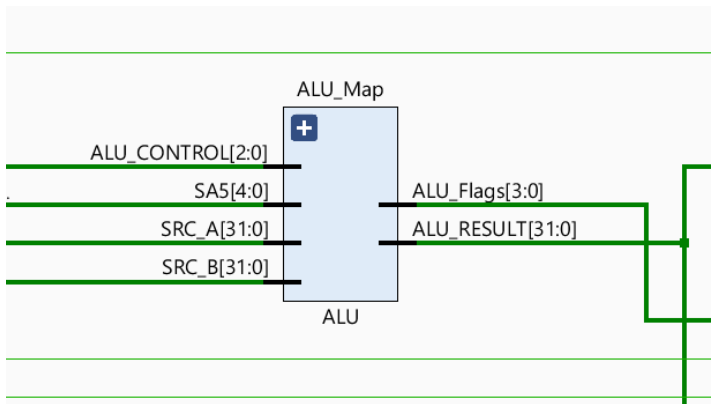
Επιπλέον, έχουμε έναν καταχωρητή καταστάσεων (status register) ο οποίος αποθηκεύει τις τιμές των σημαιών neg, z, c, v καθώς το αποτέλεσμα μπορεί να είναι αρνητικό, μηδέν, να έχει κρατούμενο ή να υπερχειλίζει, για κάθε πράξη που έχει s=1, δηλαδή: αφαίρεση, LSL, LSR, ASR, ROR.

```
ZER <= '0' when ((RES NOR RES) /= "11111111111111111111111111111111") else
      '1' when ((RES NOR RES) = "11111111111111111111111111111111") else
      'Z';

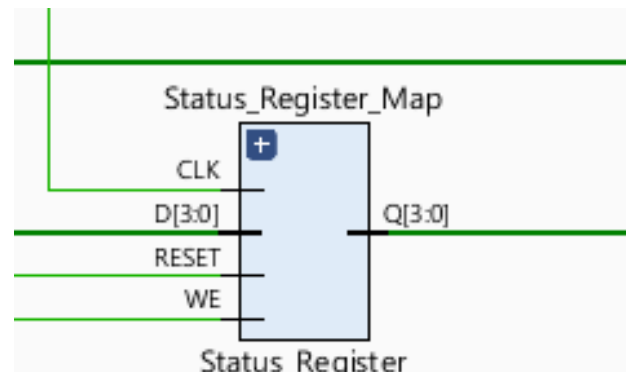
NEG <= RES(31);

ALU_Flags <= NEG & ZER & AS_C & AS_O when (ALU_CONTROL = "000" or ALU_CONTROL = "100") else
      NEG & ZER & '0' & '0' when (ALU_CONTROL = "010" or ALU_CONTROL = "110") else
      "0000";
```

### Architecture 5 : Flags

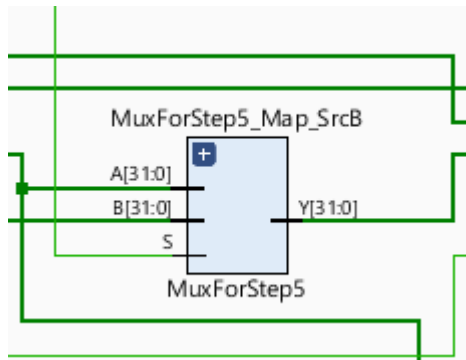


RTL -Schematic 7 : ALU



RTL -Schematic 6 : Status Register

Υπάρχει και ένας πολυπλέκτης με σήμα επιλογής το Alusrc για τον προσδιορισμό του SrcB.



RTL -Schematic 8 : Mux for SrcB

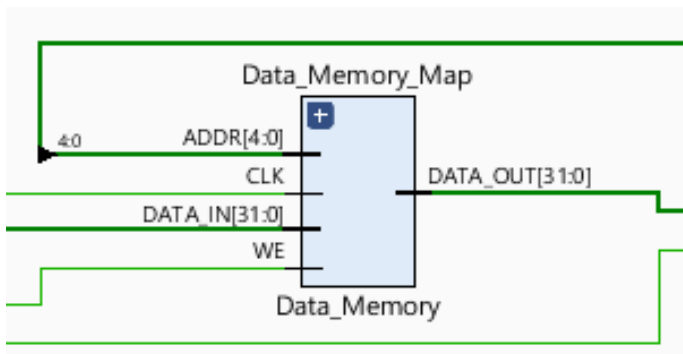
### 3.3.4 Step 4: Data\_Memory

```
architecture Behavioral of Data_Memory is      -- STEP 2-4.1.

    type Data_Memory is array (0 to 2**N-1)
    of STD_LOGIC_VECTOR (M-1 downto 0);
    signal RAM : Data_Memory;

begin
    Block_RAM: process (CLK, WE, ADDR, DATA_IN)
    begin
        if (CLK = '1' and CLK'event) 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;
```

Architecture 6 : Data memory



RTL-Schematic 9 : Data memory

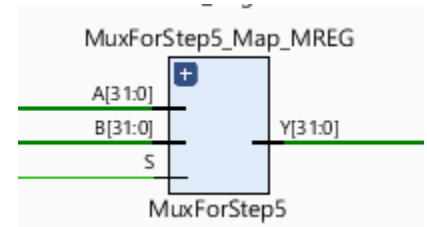
Εφόσον, το σήμα εγγραφής MemWrite είναι ενεργοποιημένο τότε γράφεται στην μνήμη η είσοδος WD κατά την ανερχόμενη ακμή του ρολογιού.

### 3.3.5 Step 5: MuxForStep5

Το πέμπτο στάδιο αποτελείται από τρεις πολυπλέκτης 2 σε 1.

➤ Ένας πολυπλέκτης με εισόδους το αποτέλεσμα προς εγγραφή RD και το ALUResult και με σήμα ελέγχου το MemtoReg:

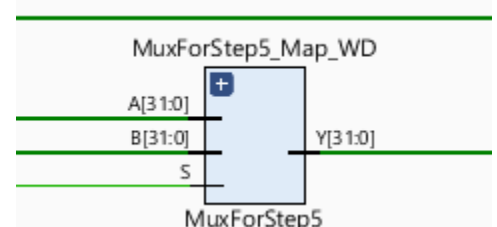
- Εάν είναι απενεργοποιημένο: η τιμή που προέρχεται στην είσοδο δεδομένων εγγραφής των καταχωρητών προέρχεται από την ALU.
- Εάν είναι ενεργοποιημένο: η τιμή που προέρχεται στην είσοδο δεδομένων εγγραφής των καταχωρητών προέρχεται από την μνήμη δεδομένων.



RTL -Schematic 10:  
MuxForStep5\_Map\_MREG

➤ Ένας πολυπλέκτης με εισόδους το αποτέλεσμα του προηγούμενου πολυπλέκτη και της τιμής PC + 4 με σήμα ελέγχου το RegSrc(2):

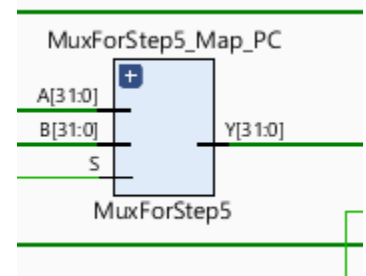
- Εάν είναι απενεργοποιημένο: επιλέγει το αποτέλεσμα του προηγούμενου πολυπλέκτη.
- Εάν είναι ενεργοποιημένο: επιλέγει την τιμή PC +4.



RTL -Schematic 11 : MuxForStep5\_Map\_WD

➤ Ένας πολυπλέκτης με εισόδους το αποτέλεσμα του πρώτου πολυπλέκτη και της τιμής PC + 4 με σήμα ελέγχου το PCSrc:

- Εάν είναι απενεργοποιημένο: επιλέγει την τιμή PC +4.
- Εάν είναι ενεργοποιημένο: επιλέγει το αποτέλεσμα του πρώτου πολυπλέκτη



RTL -Schematic 12 :  
MuxForStep5\_Map\_PC

## Chapter 4: Μονάδα ελέγχου (Control Unit)

### 4. 1 Control Unit Componets

Πιο αναλυτικά, αποτελείται από τα ακόλουθα δομικά στοιχεία:

1. Instruction Decoder --step 1
2. Write Enable Logic --step 2
3. PC Logic
4. Conditional Logic --step 3

### 4. 2 Περιγραφή control unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;    -- For counting and addition
use IEEE.NUMERIC_STD.ALL;           -- For type conversions

entity Control_Unit is              -- STEP 4-5
    generic(
        M : positive := 32);
    Port (
        Instr      : in STD_LOGIC_VECTOR(M-1 downto 0);
        FLAGS      : 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(2 downto 0);
        ImmSrc     : out STD_LOGIC;
        MemWrite    : out STD_LOGIC;
        FlagsWrite  : out STD_LOGIC;
        RegWrite    : out STD_LOGIC;
        PCSrc      : out STD_LOGIC);
end Control_Unit;

architecture Structural of Control_Unit is
    component 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);
            ALUControl : out STD_LOGIC_VECTOR(2 downto 0);
```

```

        ALUSrc      : out STD_LOGIC;
        MemtoReg    : out STD_LOGIC;
        ImmSrc      : out STD_LOGIC;
        NoWrite_in  : out STD_LOGIC);
end component;

component WELogic is
Port (
    op              : in STD_LOGIC_VECTOR (1 downto 0);
    S_L             : in STD_LOGIC;
    NoWrite_in      : in STD_LOGIC;
    MemWrite_in     : out STD_LOGIC;
    FlagsWrite_in   : out STD_LOGIC;
    RegWrite_in     : out STD_LOGIC);
end component;

component PCLogic is
Port (
    RegWrite_in : in STD_LOGIC;
    Rd          : in STD_LOGIC_VECTOR(3 downto 0);
    op          : in STD_LOGIC;
    PCSrc_in    : out STD_LOGIC);
end component;

component 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 component;

-- InstrDec signals
signal RegSrc_OUT      : STD_LOGIC_VECTOR (2 downto 0);
signal ALUSrc_OUT      : STD_LOGIC;
signal MemtoReg_OUT    : STD_LOGIC;
signal ALUControl_OUT  : STD_LOGIC_VECTOR (2 downto 0);
signal ImmSrc_OUT      : STD_LOGIC;
signal NoWrite_in_OUT  : STD_LOGIC;

-- WELogic signals
signal MemWrite_in_OUT : STD_LOGIC;
signal FlagsWrite_in_OUT : STD_LOGIC;
signal RegWrite_in_OUT : STD_LOGIC;

-- PCLogic signal
signal PCSrc_in_OUT : STD_LOGIC;

-- CONDLogic signal

```

```

signal CondEX_in_OUT : STD_LOGIC;

begin
  InstrDec_Map : InstrDec
  port map(
    op          => Instr(27 downto 26),
    funct       => Instr(25 downto 20),
    sh          => Instr(6 downto 5),
    RegSrc      => RegSrc_OUT,
    ALUSrc      => ALUSrc_OUT,
    MemtoReg    => MemtoReg_OUT,
    ALUControl  => ALUControl_OUT,
    ImmSrc      => ImmSrc_OUT,
    NoWrite_in  => NoWrite_in_OUT);

  WELogic_Map : WELogic
  port map(
    op          => Instr(27 downto 26),
    S_L         => Instr(20),
    NoWrite_in  => NoWrite_in_OUT,
    MemWrite_in => MemWrite_in_OUT,
    FlagsWrite_in => FlagsWrite_in_OUT,
    RegWrite_in => RegWrite_in_OUT);

  PCLogic_Map : PCLogic
  port map(
    RegWrite_in => RegWrite_in_OUT,
    Rd          => Instr(15 downto 12),
    op          => Instr(27),
    PCSrc_in    => PCSrc_in_OUT);

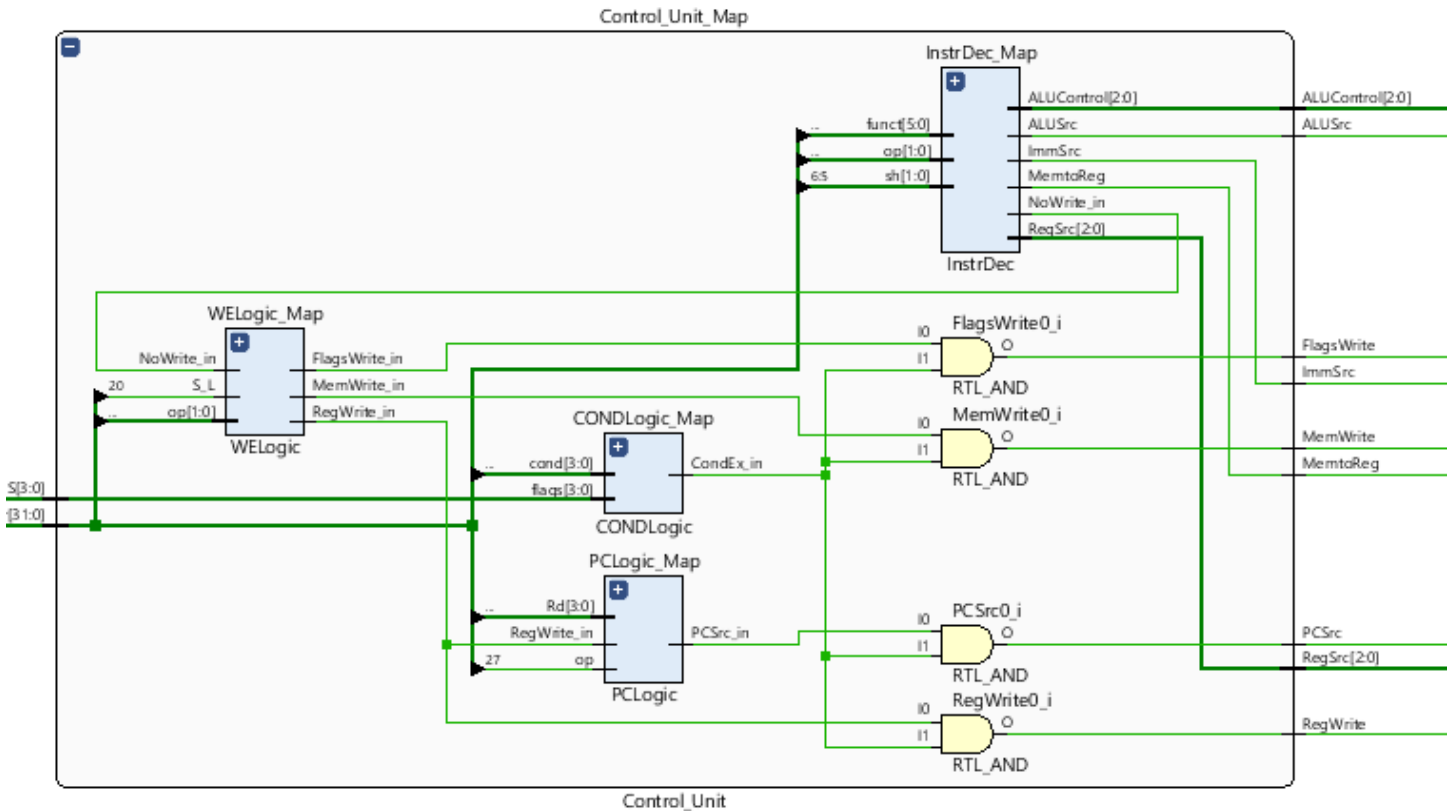
  CONDLogic_Map : CONDLogic
  port map(
    cond        => Instr(31 downto 28),
    flags       => FLAGS,
    CondEx_in   => CondEX_in_OUT);

  RegSrc      <= RegSrc_OUT;
  ALUSrc      <= ALUSrc_OUT;
  MemtoReg    <= MemtoReg_OUT;
  ALUControl  <= ALUControl_OUT;
  ImmSrc      <= ImmSrc_OUT;

  MemWrite    <= '1' when (MemWrite_in_OUT and CondEX_in_OUT) = '1' else '0';
  FlagsWrite  <= '1' when (FlagsWrite_in_OUT and CondEX_in_OUT) = '1' else '0';
  RegWrite    <= '1' when (RegWrite_in_OUT and CondEX_in_OUT) = '1' else '0';
  PCSrc       <= '1' when (PCSrc_in_OUT and CondEX_in_OUT) = '1' else '0';
end Structural;

```

## 4.2.1 RTL διάγραμμα control unit



Για να κατανοήσουμε τον παραπάνω κώδικα, είναι σημαντικό να περιγράψουμε και να αναλύσουμε τη λειτουργία κάθε τμήματος και στοιχείου που περιέχεται σε αυτό.

## 4. 3 Περιγραφή Components

### 4.3.1 Step 1: InstrDec

Η οντότητα InstrDec είναι υπεύθυνη για την αποκωδικοποίηση εντολών σε ένα ψηφιακό κύκλωμα. Βάσει του κωδικού λειτουργίας και των κωδικών συνάρτησης, παράγει τα κατάλληλα σήματα ελέγχου που καθοδηγούν τη λειτουργία διαφόρων στοιχείων, όπως η ALU και το αρχείο καταχωρητών.

```
entity InstrDec is -- STEP 4-1.3
    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      : in STD_LOGIC_VECTOR(2 downto 0);
        ALUControl  : out STD_LOGIC_VECTOR(2 downto 0);
        ALUSrc      : out STD_LOGIC;
        MemtoReg    : out STD_LOGIC;
        ImmSrc      : out STD_LOGIC;
        NoWrite_in  : out STD_LOGIC
    );
end InstrDec;
```

Entity 3 : InstrDec



Τα σήματα ελέγχου:

1. **RegSrc:** Πρόκειται για τον αριθμό καταχωρητή προορισμού για τον καταχωρητή εγγραφής όπου για τις ALU(S)-I, CMP\_I, MOV\_I και MVN\_I έχει τιμή διαφορετική του "000".
2. **AluSrc:** Αυτή η έξοδος καθορίζει αν το δεύτερο λειτουργικό που θα χρησιμοποιήσει η ALU θα προέρχεται από έναν καταχωρητή ή θα είναι μια άμεση τιμή.
3. **MemtoReg:** Αποφασίζει αν θα στείλει το αποτέλεσμα της ALU ή την τιμή της μνήμης στο αρχείο καταχωρητών.
4. **AluControl:** Με βάση το είδος της εντολής επιλέγεται η σωστή τιμή της πράξης που θα εκτελεστεί όπως αυτό το πεδίο ορίστηκε στην οντότητα της ALU.
5. **ImmSrc:** Αυτό το σήμα υποδεικνύει αν η άμεση τιμή θα προέρχεται από έναν καταχωρητή ή από μια σταθερή τιμή βάσει του κωδικού λειτουργίας.
6. **NoWrite in:** Αυτό το σήμα ελέγχει αν η εγγραφή στο αρχείο καταχωρητών είναι επιτρεπτή με βάση τη λειτουργία που εκτελείται.

OP	FUNCT	SH	RegSrc	ALUSrc0	ALUSrc	ImmSrc	ALU Control	Mem To Reg	NoWrite
01	010001	X	0X0	X	1	0	0001	1	0
01	011001	X	0X0	X	1	0	0000	1	0
01	010000	X	010	X	1	0	0001	0	0
01	011000	X	010	X	1	0	0000	0	0
00	10100X	X	0X0	X	1	0	0000	0	0
00	10010X	X	0X0	X	1	0	0001	0	0
00	10000X	X	0X0	X	1	0	0010	0	0
00	11100X	X	0X0	X	1	0	0011	0	0
00	10001X	X	0X0	X	1	0	0100	0	0
00	00100X	X	000	X	0	X	0000	0	0
00	00010X	X	000	X	0	X	0001	0	0
00	00000X	X	000	X	0	X	0010	0	0
00	01100X	X	000	X	0	X	0011	0	0
00	00001X	X	000	X	0	X	0100	0	0
00	110101	X	0X0	X	1	0	0001	0	1
00	010101	X	000	X	0	X	0001	0	1
00	111010	00	0X0	X	1	0	0101	0	0
00	111110	X	0X0	X	1	0	0110	0	0
00	011010	00	000	1	1	0	0111	0	0
00	011110	X	000	X	0	X	0110	0	0
00	011010	00	0X0	1	1	0	0111	0	0
00	011010	01	0X0	1	1	0	1000	0	0
00	011010	10	0X0	1	1	0	1001	0	0
00	011010	11	0X0	1	1	0	1010	0	0
10	10XXXX	X	1X1	X	1	1	0000	0	0
10	11XXXX	X	1X1	X	1	1	0000	0	0

### 4.3.2 Step 2: Logic (WELogic & PCLogic)

#### ➤ Write Enable Logic

Η οντότητα WELogic είναι υπεύθυνη για την παραγωγή σημάτων ελέγχου που καθοδηγούν τη διαδικασία εγγραφής στη μνήμη, την ενημέρωση των σημάτων και την εγγραφή στους καταχωρητές, με βάση τις τρέχουσες εντολές.

OP	S/L	NoWrite	RegWrite	MemWrite	FlagsWrite
01	1	0	1	0	0
01	0	0	0	1	1
00	0	0	1	0	0
00	1	1	0	0	1
00	1	0	1	0	1
10	X	X	0	0	0

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;    -- For counting and addition
use IEEE.NUMERIC_STD.ALL;           -- For type conversions

entity WELogic is                    -- STEP 4-2
    Port (
        op          : in STD_LOGIC_VECTOR (1 downto 0);
        S_L         : in STD_LOGIC;
        NoWrite_in   : in STD_LOGIC;
        MemWrite_in  : out STD_LOGIC;
        FlagsWrite_in : out STD_LOGIC;
        RegWrite_in  : out STD_LOGIC
    );
end WELogic;

architecture Behavioral of WELogic is

begin

    MemWrite_in <= '0' when (op = "00") else
        '1' when (op = "01" and S_L = '0' and NoWrite_in = '0') else
        '0' when (op = "01" and S_L = '1' and NoWrite_in = '0') else
        '0' when (op = "10") else
        'Z';

    FlagsWrite_in <= '0' when (op = "00" and S_L = '0' and NoWrite_in = '0') else
        '1' when (op = "00" and S_L = '1') else
        '0' when (op = "01") else
```

```

        '0' when (op = "10") else
        'Z';

RegWrite_in <= '1' when (op = "00" and S_L = '0' and NoWrite_in = '0') else
        '1' when (op = "00" and S_L = '1' and NoWrite_in = '0') else
        '0' when (op = "00" and S_L = '1' and NoWrite_in = '1') else
        '1' when (op = "01" and S_L = '1' and NoWrite_in = '0') else
        '0' when (op = "01" and S_L = '0' and NoWrite_in = '0') else
        '0' when (op = "10" and NoWrite_in = '0') else
        '1' when (op = "10" and NoWrite_in = '1') else
        'Z';

end Behavioral;

```

### ➤ PCLogic

Η οντότητα PCLogic είναι υπεύθυνη για τη λογική που καθορίζει αν η διεύθυνση του μετρητή προγράμματος (PC) θα αλλάξει ή θα παραμείνει στην επόμενη εντολή με βάση τις συνθήκες που παρέχονται. Αυτό είναι κρίσιμο για τη ροή του προγράμματος στην CPU.

OP	Rd	RegisterWrite	PCSource
0	0-14	1	0
0	15	1	1
0	0-14	1	0
0	15	1	1
0	X	0	0
0	X	0	0
1	X	0	1

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;    -- For counting and addition
use IEEE.NUMERIC_STD.ALL;           -- For type conversions

entity PCLogic is                    -- STEP 4-3
    Port (
        RegWrite_in : in STD_LOGIC;
        Rd           : in STD_LOGIC_VECTOR(3 downto 0);
        op           : in STD_LOGIC;
        PCSrc_in     : out STD_LOGIC
    );
end PCLogic;

architecture Behavioral of PCLogic is

begin

```

```

PCSrc_in <= '0' when (op = '0' and RegWrite_in = '0') else
    '1' when (op = '0' and RegWrite_in = '1' and Rd = "1111") else
    '0' when (op = '0' and RegWrite_in = '1' and not(Rd = "1111")) else
    '1' when (op = '1') else
    'Z';

end Behavioral;

```

### 4.3.3 Step 3: CONDLLogic

Παίρνει ως είσοδο τα 4 πρώτα bit που αντιστοιχούν στο πεδίο cond ως εξής:

flag(0) = N

flag(1) = Z

flag(2) = C

flag(3) = V

και για κάθε σήμα του πίνακα που ακολουθεί, παράγει τα αντίστοιχα μνημονικά συνθήκης με τις εξισώσεις Boole των σημαιών που τις ικανοποιούν.

cond <sub>3:0</sub>	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 <sub>3:0</sub>	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	$\bar{N}\oplus\bar{V}$
1011	LT	Signed less	$N\oplus V$
1100	GT	Signed greater	$\bar{Z}N\oplus\bar{V}$
1101	LE	Signed less or equal	$Z+(N\oplus V)$
1110	AL (ή none)	Always / unconditional	1
1111	none	For unconditional instructions	1

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;    -- For counting and addition
use IEEE.NUMERIC_STD.ALL;           -- For type conversions

entity CONDLogic is    -- STEP 4-4.3
    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

begin

-- TABLE 1-1.4. / -- STEP 4-4.2.
--flag(0) = N
--      1  Z
--      2  C
--      3  V

    CondEx_in <= '0' when (cond = "0000" and flags(2) = '0') else
                  '1' when (cond = "0000" and flags(2) = '1') else
                  '0' when (cond = "0001" and flags(2) = '1') else
                  '1' when (cond = "0001" and flags(2) = '0') else
                  '0' when (cond = "0010" and flags(1) = '0') else
                  '1' when (cond = "0010" and flags(1) = '1') else
                  '0' when (cond = "0011" and flags(1) = '1') else
                  '1' when (cond = "0011" and flags(1) = '0') else
                  '0' when (cond = "0100" and flags(3) = '0') else
                  '1' when (cond = "0100" and flags(3) = '1') else
                  '0' when (cond = "0101" and flags(3) = '1') else
                  '1' when (cond = "0101" and flags(3) = '0') else
                  '0' when (cond = "0110" and flags(0) = '0') else
                  '1' when (cond = "0110" and flags(0) = '1') else
                  '0' when (cond = "0111" and flags(0) = '1') else
                  '1' when (cond = "0111" and flags(0) = '0') else
                  '0' when (cond = "1000" and not(flags(1) = '1' and flags(2) = '0')) else
                  '1' when (cond = "1000" and flags(1) = '1' and flags(2) = '0') else
                  '0' when (cond = "1001" and not(flags(1) = '0' and flags(2) = '1')) else
                  '1' when (cond = "1001" and flags(1) = '0' and flags(2) = '1') else
                  '0' when (cond = "1010" and not(flags(3) = flags(0))) else
                  '1' when (cond = "1010" and flags(3) = flags(0)) else
                  '0' when (cond = "1011" and not(flags(3) /= flags(0))) else
                  '1' when (cond = "1011" and flags(3) /= flags(0)) else
                  '0' when (cond = "1100" and not(flags(2) = '0' and flags(3) = '0')) else

```

```
        '1' when (cond = "1100" and flags(2) = '0' and flags(3) = '0') else
        '0' when (cond = "1101" and not(flags(2) = '1' and flags(3) /= flags(0)))
else
        '1' when (cond = "1101" and flags(2) = '1' and flags(3) /= flags(0)) else
        '1' when (cond = "1110" or cond = "1111") else
        'Z';
end Behavioral;
```

# Chapter 5: Επεξεργαστής – Processor

## 5. 1 Δομή επεξεργαστή

Συνεπώς, ο επεξεργαστής αποτελείται από την σύνδεση του data path και control unit όπου τα σήματα εξόδου της μονάδας ελέγχου αποτελούν είσοδο για τη διαδρομή δεδομένων έτσι ώστε να είναι ενεργοποιημένες οι κατάλληλες σημαίες για την σωστή εκτέλεση των εντολών.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;    -- For counting and addition
use IEEE.NUMERIC_STD.ALL;           -- For type conversions

entity Processor is                  -- STEP 5
    generic(
        M : positive := 32);
    Port (
        CLK          : in STD_LOGIC;
        RESET        : in STD_LOGIC;
        PC           : out STD_LOGIC_VECTOR (M-1 downto 0);
        ALUResult    : out STD_LOGIC_VECTOR (M-1 downto 0);
        WriteData    : out STD_LOGIC;
        Instruction  : out STD_LOGIC_VECTOR (M-1 downto 0);
        Result       : out STD_LOGIC_VECTOR (M-1 downto 0)
    );
end Processor;

architecture Structural of Processor is

    component Datapath is
        generic (
            M : positive := 32          -- data word length
        );
        Port (
            -- STEP 1:
            CLK          : in STD_LOGIC;
            RESET        : in STD_LOGIC;
            PCWrite      : in STD_LOGIC;
            PC_out       : out STD_LOGIC_VECTOR (M-1 downto 0);
            instr        : out STD_LOGIC_VECTOR (M-1 downto 0);

            -- STEP 2:
            RegSrc       : in STD_LOGIC_VECTOR (2 downto 0);
            RegWrite     : in STD_LOGIC;
            ImmSrc       : in STD_LOGIC;
        );
    end component;

    -- Datapath instantiation
    Datapath_U0 : Datapath
        generic map (
            M => M
        )
        port map (
            CLK => CLK,
            RESET => RESET,
            PCWrite => PCWrite,
            PC_out => PC,
            instr => Instruction,
            RegSrc => RegSrc,
            RegWrite => RegWrite,
            ImmSrc => ImmSrc
        );

    -- ALU instantiation
    ALU_U0 : ALU
        generic map (
            M => M
        )
        port map (
            A => ALU_A,
            B => ALU_B,
            ALUResult => ALUResult
        );

    -- Register File instantiation
    RegFile_U0 : RegisterFile
        generic map (
            M => M
        )
        port map (
            WriteData => WriteData,
            Instruction => Instruction,
            Result => Result
        );
end Structural;
```

```

    -- Step 3:
    ALUSrc      : in STD_LOGIC;
    ALUControl  : in STD_LOGIC_VECTOR (2 downto 0);
    FlagsWrite  : in STD_LOGIC;
    ALUResult   : out STD_LOGIC_VECTOR (M-1 downto 0);
    flags       : out STD_LOGIC_VECTOR (3 downto 0);

    -- STEP 4:
    MemWrite    : in STD_LOGIC;

    -- STEP 5:
    MemtoReg    : in STD_LOGIC;
    PCSrc       : in STD_LOGIC;
    Result      : out STD_LOGIC_VECTOR (M-1 downto 0)

);
end component;

component Control_Unit is
generic(
    M : positive := 32);
Port (
    Instr      : in STD_LOGIC_VECTOR(M-1 downto 0);
    FLAGS      : in STD_LOGIC_VECTOR(3 downto 0);

    RegSrc     : out STD_LOGIC_VECTOR(2 downto 0);    --out signals from files
InstrDec, Logic
    ALUSrc     : out STD_LOGIC;
    MemtoReg    : out STD_LOGIC;
    ALUControl  : out STD_LOGIC_VECTOR(2 downto 0);
    ImmSrc     : out STD_LOGIC;
    MemWrite    : out STD_LOGIC;
    FlagsWrite  : out STD_LOGIC;
    RegWrite    : out STD_LOGIC;
    PCSrc       : out STD_LOGIC
);
end component;

--signal for the match-up

-- DataPath
signal INSTR   : STD_LOGIC_VECTOR (M-1 downto 0);
signal FLAG    : STD_LOGIC_VECTOR (3 downto 0);
signal PCReg   : STD_LOGIC_VECTOR (M-1 downto 0);
signal ALURes  : STD_LOGIC_VECTOR (M-1 downto 0);

```



```

signal Res      : STD_LOGIC_VECTOR (M-1 downto 0);

-- Control Unit
signal RegSrc_0    : STD_LOGIC_VECTOR (2 downto 0);
signal ALUSrc_0    : STD_LOGIC;
signal MemtoReg_0  : STD_LOGIC;
signal ALUControl_0 : STD_LOGIC_VECTOR (2 downto 0);
signal ImmSrc_0    : STD_LOGIC;
signal RegWrite_0  : STD_LOGIC;
signal MemWrite_0  : STD_LOGIC;
signal FlagsWrite_0 : STD_LOGIC;
signal PCSrc_0     : STD_LOGIC;

```

```
begin
```

```
  Datapath_Map : DataPath
```

```
  port map(
```

```

    CLK      => CLK,
    RESET    => RESET,
    PCWrite   => '1',
    RegSrc    => RegSrc_0,
    ALUSrc    => ALUSrc_0,
    MemtoReg  => MemtoReg_0,
    ALUControl => ALUControl_0,
    ImmSrc    => ImmSrc_0,
    RegWrite  => RegWrite_0,
    MemWrite  => MemWrite_0,
    FlagsWrite => FlagsWrite_0,
    INSTR     => Instr,
    flags     => FLAG,
    PCSrc     => PCSrc_0,
    PC_out    => PCReg,
    ALUResult => ALURes,
    Result    => Res

```

```
);
```

```
  Control_Unit_Map : Control_Unit
```

```
  port map(
```

```

    Instr      => INSTR,
    FLAGS      => FLAG,
    RegSrc     => RegSrc_0,
    ALUSrc     => ALUSrc_0,
    MemtoReg   => MemtoReg_0,
    ALUControl => ALUControl_0,
    ImmSrc     => ImmSrc_0,
    RegWrite   => RegWrite_0,
    MemWrite   => MemWrite_0,
    FlagsWrite => FlagsWrite_0,

```

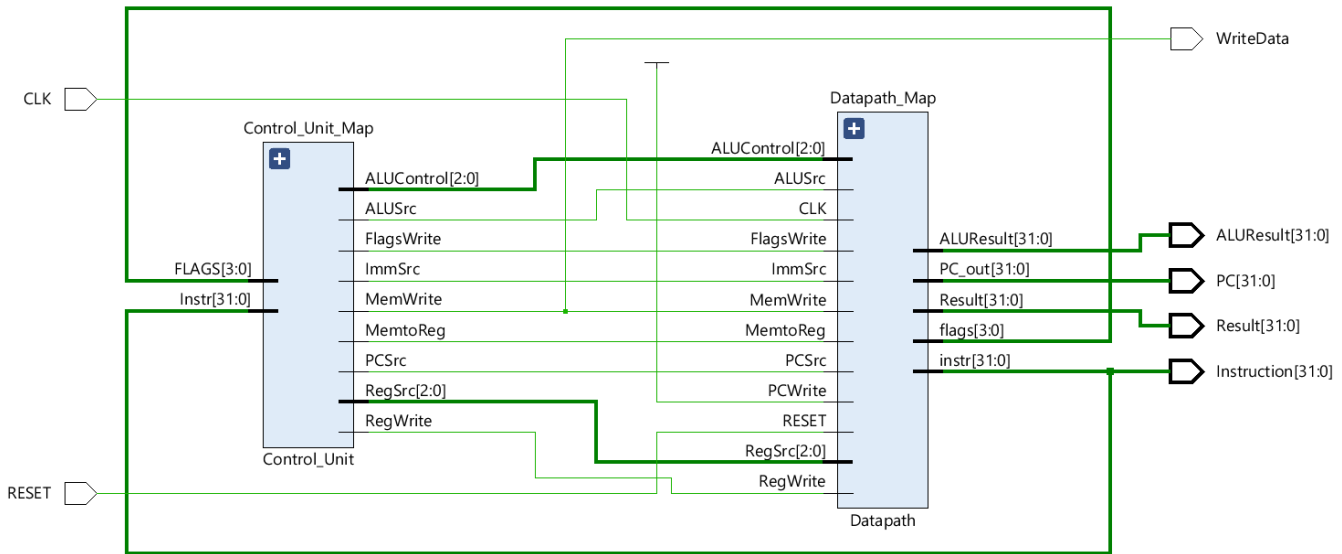
```

PCSrc      => PCSrc_0);

Instruction <= INSTR;
PC          <= PCReg;
ALUResult   <= ALURes;
WriteData   <= MemWrite_0;
Result      <= Res;

end Structural;

```



RTL -Schematic 13 : Processor

## 5. 2 Μέγιστη συχνότητα - Χειρότερη κρίσιμη διαδρομή - Χειρότερη σύντομη διαδρομή.

Από το Report Timing Summary προκύπτει ότι η μέγιστη συχνότητα λειτουργίας είναι τα 74.455 MHz ή 13.431 ns περίοδος ρολογιού.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 0.297 ns	Worst Hold Slack (WHS): 0.279 ns	Worst Pulse Width Slack (WPWS): 5.465 ns	
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns	
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 808	Total Number of Endpoints: 808	Total Number of Endpoints: 165	
All user specified timing constraints are met.			

Timing 1

Παρατηρούμε ότι το worst negative slack (WNS) είναι θετικό συνεπώς η κρίσιμη διαδρομή ικανοποιεί την περίοδο του ρολογιού και το Worst Hold Slack (WHS) είναι θετικό άρα δεν παραβιάζεται ο χρόνος διατήρησης ο χρόνος διατήρησης (hold time).

Η κρίσιμη διαδρομή έχει καθυστέρηση διάδοσης 13.129ns εκ των οποίων τα 4.142ns αφορούν στην λογική ενώ τα 8.987ns αφορούν στην δικτύωση. Η αβεβαιότητα του ρολογιού εκτιμάται στα 0,035ns. Τα επίπεδα λογικής (logic level) είναι 21.

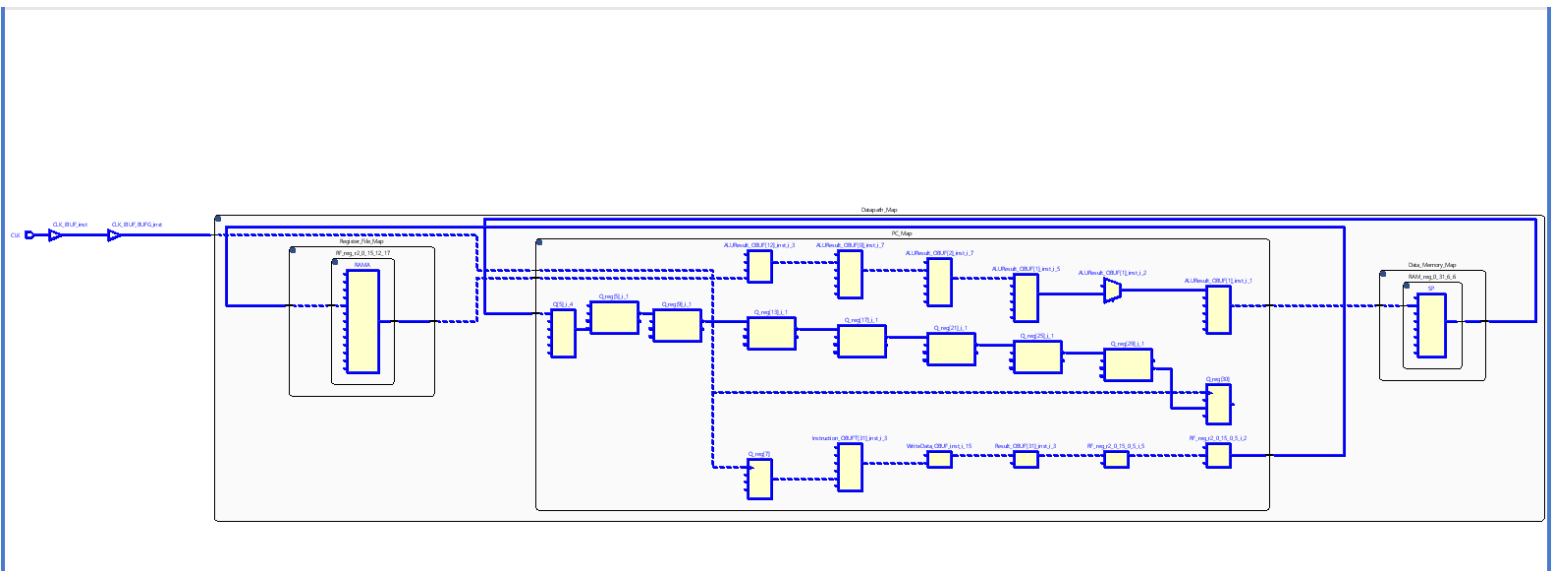
Summary

Name	Path 1
Slack	0.297ns
Source	Datapath_Map/PC_Map/Q_reg[7]/C (rising edge-triggered cell FDRE clocked by CLK {rise@0.000ns fall@6.716ns period=13.431ns})
Destination	Datapath_Map/PC_Map/Q_reg[30]/D (rising edge-triggered cell FDRE clocked by CLK {rise@0.000ns fall@6.716ns period=13.431ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	13.431ns (CLK rise@13.431ns - CLK rise@0.000ns)
Data Path Delay	13.129ns (logic 4.142ns (31.547%) route 8.987ns (68.453%))
Logic Levels	21 (CARRY4=7 LUT2=3 LUT3=1 LUT4=1 LUT6=6 MUXF7=1 RAMD32=1 RAMS32=1)
Clock Path Skew	-0.031ns
Clock U...tainty	0.035ns

Source Clock Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)	
(clock CLK rise edge)	(r) 0.000	0.000			

Timing 2: Κρίσιμη διαδρομή



RTL-Schematic 14 : Μονοπάτι κρίσιμης διαδρομής

Η σύντομη διαδρομή έχει καθυστέρηση μόλυνσης 0,399 ns εκ των οποίων τα 0,209 ns αφορούν στην λογική ενώ τα 0,190 ns αφορούν στην δικτύωση. Η αβεβαιότητα του ρολογιού εκτιμάται στα 0,000 ns. Τα επίπεδα λογικής (logic level) είναι 1.

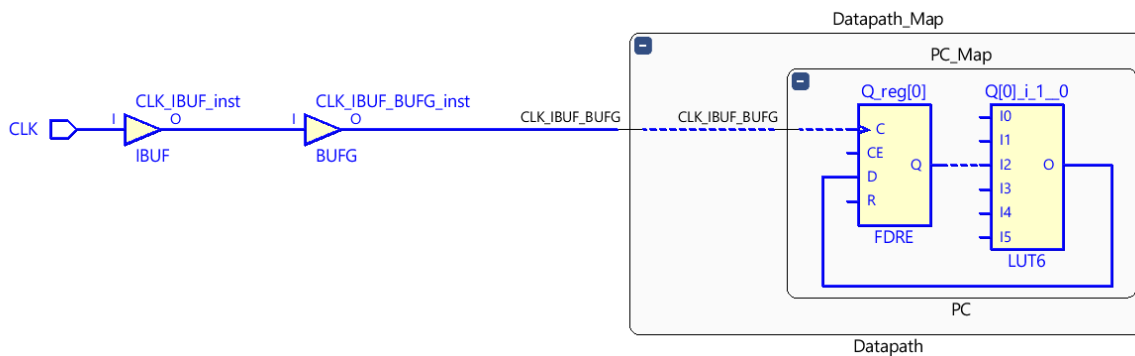
Summary

Name	Path 11
Slack (Hold)	0.279ns
Source	Datapath_Map/PC_Map/Q_reg[0]/C (rising edge-triggered cell FDRE clocked by CLK (rise@0.000ns fall@6.716ns period=13.431ns))
Destination	Datapath_Map/PC_Map/Q_reg[0]/D (rising edge-triggered cell FDRE clocked by CLK (rise@0.000ns fall@6.716ns period=13.431ns))
Path Group	CLK
Path Type	Hold (Min at Fast Process Corner)
Requirement	0.000ns (CLK rise@0.000ns - CLK rise@0.000ns)
Data P...Delay	0.399ns (logic 0.209ns (52.433%) route 0.190ns (47.567%))
Logic Levels	1 (LUT6=1)
Clock ... Skew	0.000ns

Source Clock Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)	
(clock CLK rise edge)	(r) 0.000	0.000			
	(r) 0.000	0.000	Site: Y9	CLK	

Timing 3 : Σύντομη διαδρομή



Timing 4 : Μονοπάτι σύντομης διαδρομής

# Chapter 6: Προσομοίωση

## 6. 1 Δοκιμαστικός κώδικας

Η εργασία έχει δοκιμαστεί και εκτελείται για τις ακόλουθες εντολές:

```
X"E3E00000" -- MVN r0, #0
X"E3A01002" -- MOV r1, #2
X"E0812000" -- ADD r2, r1, r0
X"E0413000" -- SUB r3, r1, r0
X"E0234002" -- EOR r4, r3, r2
X"E0035002" -- AND r5, r3, r2
X"E1A06080" -- MOV r6, r0, LSL #1
X"E1A070C1" -- MOV r7, r1, ROR #2
X"E5802002" -- STR r2, [r0, #2]
X"E5023001" -- LDR r3, [r0, #1]
X"E5900002" -- LDR r0, [r0, #2]
X"E5121001" -- LDR r1, [r2, #1]
X"E1500001" -- CMP r0, r1
X"02800001" -- BEQ mylabel
X"EBFFFFFFC" -- B mylabel
```

Αυτό το πρόγραμμα επαληθεύει τη σχεδίαση και τη λειτουργία του επεξεργαστή με τους εξής τρόπους:

- **Δοκιμή όλων των βασικών εντολών:** Κάθε βασική εντολή ARM (μετακίνηση, πρόσθεση, αφαίρεση, λογικές πράξεις) έχει δοκιμαστεί στο πρόγραμμα.
- **Ροές δεδομένων:** Το πρόγραμμα περιλαμβάνει εντολές αποθήκευσης και φόρτωσης που ελέγχουν τη σωστή λειτουργία της διαδρομής δεδομένων.
- **Συγκρίσεις και βρόχοι:** Οι εντολές CMP και οι βρόχοι ελέγχουν την ορθή εκτέλεση των εντολών ελέγχου ροής (branching).

Αυτό το πρόγραμμα παρέχει μια σφαιρική δοκιμή της λειτουργικότητας του επεξεργαστή και της διαδρομής δεδομένων του, διασφαλίζοντας ότι όλες οι εντολές λειτουργούν όπως αναμένονται.

## 6. 2 Κώδικας προσομοίωσης

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

entity Processor_tb is
    generic(
        M : positive := 32);
end Processor_tb;

architecture Behavioral of Processor_tb is

    component Processor is
        generic(
            M : positive := 32);
    Port (
        CLK          : in STD_LOGIC;
        RESET        : in STD_LOGIC;
        PC           : out STD_LOGIC_VECTOR (M-1 downto 0);
        ALUResult    : out STD_LOGIC_VECTOR (M-1 downto 0);
        WriteData    : out STD_LOGIC;
        Instruction  : out STD_LOGIC_VECTOR (M-1 downto 0);
        Result       : out STD_LOGIC_VECTOR (M-1 downto 0));
    end component Processor;

    signal CLK          : STD_LOGIC;
    signal RESET        : STD_LOGIC;
    signal WE           : STD_LOGIC;
    signal WriteData    : STD_LOGIC;
    signal PC           : STD_LOGIC_VECTOR (M-1 downto 0);
    signal Instr        : STD_LOGIC_VECTOR (M-1 downto 0);
    signal ALU_RESULT   : STD_LOGIC_VECTOR (M-1 downto 0);
    signal RESULT       : STD_LOGIC_VECTOR (M-1 downto 0);

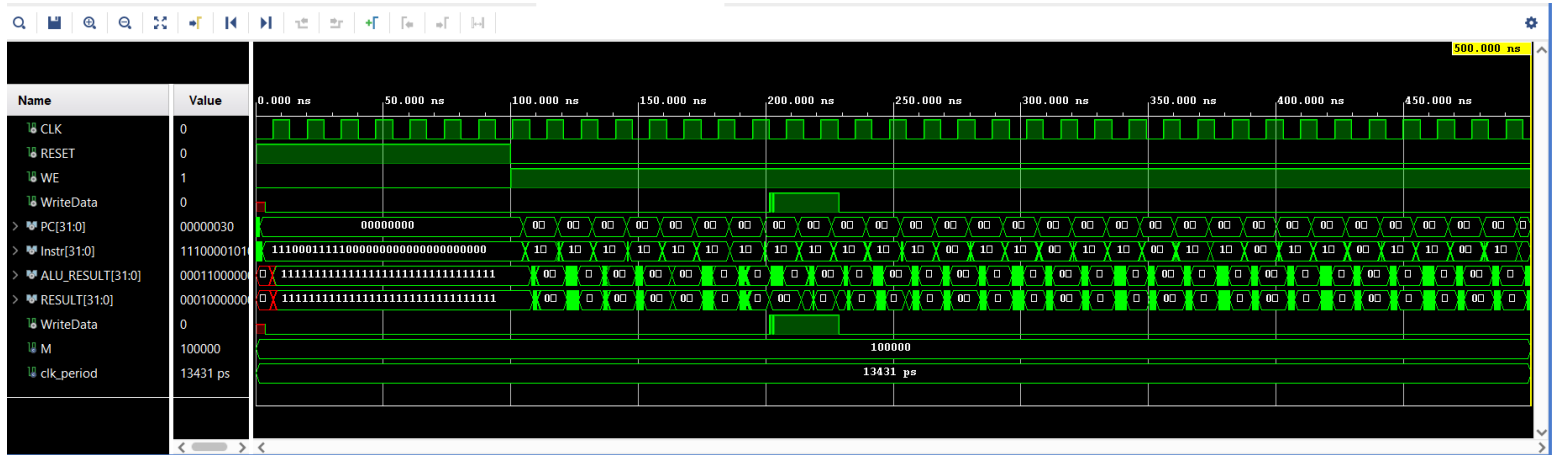
    constant clk_period: time := 13.431 ns;

begin

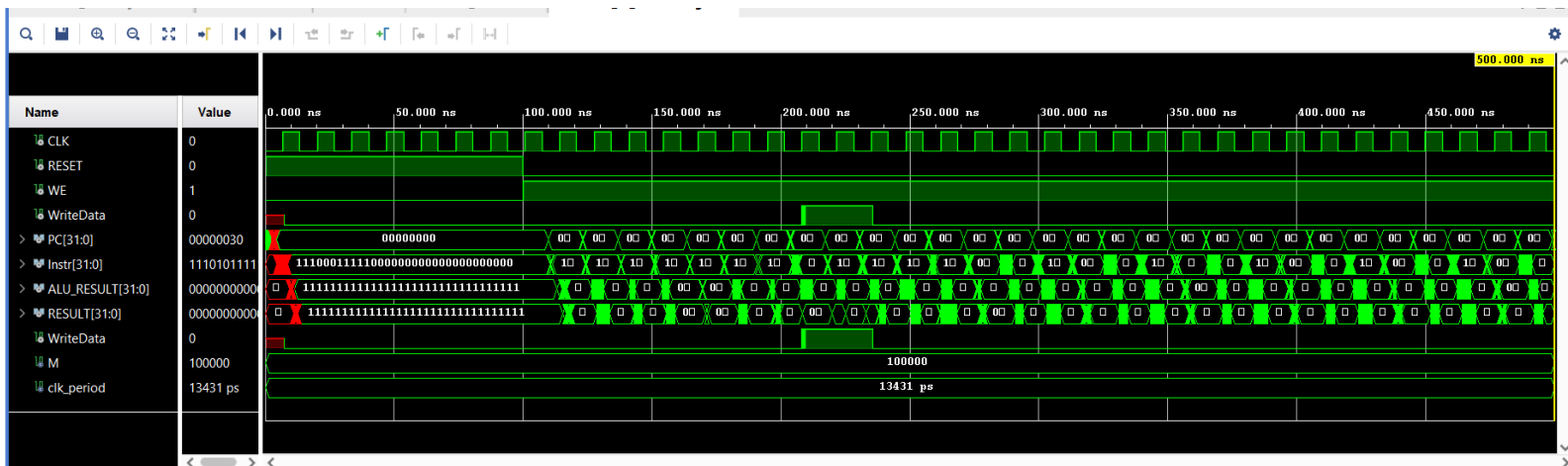
    Processor_Map: Processor
    port map(
        CLK          => CLK,
        RESET        => RESET,
        Instruction  => Instr,
        PC           => PC,
        ALUResult    => ALU_RESULT,
        WriteData    => WriteData,
        Result       => RESULT);
```



## 6. 4 Post-Synthesis Simulation



## 6. 5 Post-Implementation Simulation



Συμπεράσματα:

- Παρατηρούμε ότι τα διαγράμματα χρονισμού είναι ίδια.
- Και στα δύο στάδια προσομοίωσης, το κύκλωμα φαίνεται να ακολουθεί τη σωστή λογική λειτουργία. Τα σήματα όπως το WE, RESET, και το CLK παρουσιάζουν την αναμενόμενη συμπεριφορά. Τα δεδομένα στα πεδία PC, Instr, ALU\_RESULT, και RESULT αλλάζουν όπως αναμένεται με την εξέλιξη του χρόνου, υποδηλώνοντας ότι το κύκλωμα εκτελεί σωστά τις εντολές του.




- Στο post-implementation διάγραμμα, παρατηρούνται κάποιες χρονικές καθυστερήσεις που είναι φυσιολογικές και αναμενόμενες σε αυτό το στάδιο. Παρόλα αυτά, οι καθυστερήσεις δεν φαίνεται να προκαλούν παραβιάσεις στον χρονισμό του κυκλώματος.

## 6. 6 Synthesis και Implementation

Utilization							
Hierarchy							
Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Bonded IOB (200)	BUFGCTRL (32)	
Processor	638	36	26	1	131	1	
Datapath_Map (Datapath)	638	36	26	1	0	0	

### Synthesis

**Status:**  Complete

**Messages:** No errors or warnings

**Part:** xc7z020clg484-1

**Strategy:** [Vivado Synthesis Defaults](#)

**Report Strategy:** [Vivado Synthesis Default Reports](#)

**Incremental synthesis:** None

- **Εκμετάλλευση Πόρων:** Η συνολική εκμετάλλευση των πόρων είναι χαμηλή (π.χ. LUTs, Registers), υποδεικνύοντας ότι ο σχεδιασμός είναι αποδοτικός και μπορεί να επεκταθεί χωρίς να απαιτεί περισσότερους πόρους.
- **Διαθεσιμότητα Πόρων:** Υπάρχουν αρκετοί διαθέσιμοι πόροι για πιθανές τροποποιήσεις ή βελτιώσεις, κάτι που είναι θετικό για μελλοντική ανάπτυξη.
- **IOB Χρήση:** Η υψηλή εκμετάλλευση των IOBs υποδεικνύει ότι η συνδεσιμότητα με εξωτερικά κυκλώματα ή συσκευές είναι σημαντική και θα πρέπει να παρακολουθείται για μελλοντικές επεκτάσεις.