Electrical & Computer
Engineering Department
UNIVERSITY OF
PELOPONNESE

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΧΕΔΙΑΣΜΟΣ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ ΣΕ FPGA

ΑΝΑΦΟΡΑ

ΕΡΓΑΣΤΗΡΙΑΚΟΥ PROJECT 3

ΑΚΑΔΗΜΑΙΚΟ ΈΤΟΣ 2025-2025

ΤΡΙΑΝΤΑΦΥΛΛΟΥ ΒΑΣΙΛΕΙΟΣ 20102

ΜΕΪΧΑΝΕΤΖΟΓΛΟΥ ΝΙΚΗΦΟΡΟΣ 21207

1. Περιγραφή υλοποίησης

1.1 Γενική Επισκόπηση και Στόχοι

Στο πλαίσιο της τρίτης εργαστηριακής άσκησης, κληθήκαμε να επεκτείνουμε την αρχιτεκτονική μιας βασικής CPU τύπου "Accumulator" (συσσωρευτή). Ο κύριος στόχος ήταν διπλός: αφενός η προσθήκη υλικού (hardware) για την υποστήριξη ενός δεύτερου καταχωρητή γενικού σκοπού (Register B) και αφετέρου η ριζική αλλαγή της Μονάδας Ελέγχου από Μικροπρογραμματιζόμενη (Microprogrammed) σε **Καλωδιωμένη (Hardwired)** (χρησιμοποιήθηκαν ως base τα αρχεία της Εργαστηριακής Άσκησης 5, τα οποία υλοποιήθηκαν σε Microprogrammed λογική).

Η μετάβαση σε Hardwired λογική κρίθηκε απαραίτητη για την εξοικείωση με τη σχεδίαση Μηχανών Πεπερασμένων Καταστάσεων (FSM), οι οποίες προσφέρουν ταχύτητα στην εκτέλεση, καθώς οι εντολές υλοποιούνται απευθείας από συνδυαστικά κυκλώματα και flip-flops, χωρίς την καθυστέρηση ανάγνωσης μικροεντολών από μνήμη ROM.

1.2 Σχεδίαση της Καλωδιωμένης Μονάδας Ελέγχου (FSM)

Η "καρδιά" του επεξεργαστή υλοποιήθηκε ως ένα σύγχρονο ακολουθιακό κύκλωμα (Finite State Machine). Σε αντίθεση με την προηγούμενη εργασία όπου υπήρχε ένας απλός μετρητής (sequencer), εδώ η FSM πρέπει να "γνωρίζει" σε ποιο στάδιο της εκτέλεσης βρίσκεται και να παίρνει αποφάσεις βάσει του τρέχοντος Opcode.

Ο κύκλος μηχανής χωρίστηκε στις εξής καταστάσεις:

1. **S_FETCH1 & S_FETCH2 (Ανάκληση):** Ο επεξεργαστής φέρνει τη διεύθυνση από τον PC και διαβάζει την εντολή από τη μνήμη. Εδώ είναι κρίσιμο να αυξηθεί ο PC (PC Increment) ώστε να δείχνει στην επόμενη θέση μνήμης.

2. **S_FETCH3 (Αποθήκευση στον IR):** Η εντολή μεταφέρεται στον Instruction Register. Σε αυτό το σημείο, η FSM έχει διαθέσιμο ολόκληρο το 8-bit Opcode (π.χ. 1D ή 1E) και μπορεί να αποφασίσει την επόμενη κίνηση.

3. **S_EXECUTE (Αποκωδικοποίηση & Εκτέλεση):** Εδώ γίνεται η ουσιαστική διαφορά. Αντί να τρέξουμε μια ρουτίνα μικροεντολών, η FSM ελέγχει τα bits του IR. Αν αναγνωρίσει τον κωδικό της ORB (1D), ενεργοποιεί ταυτόχρονα όλα τα απαραίτητα σήματα (BBUS, OROP, ACLOAD) ώστε η πράξη να γίνει σε έναν μόνο κύκλο ρολογιού.

4. **S_SKIP_OPERAND (Διαχείριση Διευθύνσεων):** Ένα σημαντικό πρόβλημα που επιλύθηκε ήταν η διαχείριση εντολών διαφορετικού μήκους. Οι παλιές εντολές (π.χ. LDAC) είναι δίμπαιτες (εντολή + διεύθυνση), ενώ οι νέες (ORB, XORB) είναι μονόμπαιτες. Για να μην "μπερδευτεί" ο επεξεργαστής και εκτελέσει τη διεύθυνση ως εντολή, προσθέσαμε αυτή την κατάσταση που αναγκάζει τον PC να προσπεράσει (skip) το byte του ορίσματος όταν συναντά δίμπαιτες εντολές.

1.3 Επέκταση Data Path και Χρήση Tri-State Buffers

Η προσθήκη του Καταχωρητή B (Register B) δημιούργησε την ανάγκη τροποποίησης του κοινού διαύλου (Common Data Bus).

Ο Register B σχεδιάστηκε ως ένας καταχωρητής 8-bit που συνδέεται παράλληλα με τους υπόλοιπους (R, AC, DR). Το κρισιμότερο σημείο της σχεδίασης ήταν η σύνδεσή του στον δίαυλο.

Επειδή ο δίαυλος είναι ένας κοινόχρηστος πόρος (shared resource), δεν μπορούν δύο καταχωρητές να γράφουν δεδομένα ταυτόχρονα σε αυτόν, καθώς θα προκαλούταν βραχυκύκλωμα (bus contention).

Για τον λόγο αυτό, χρησιμοποιήσαμε έναν **απομονωτή τριών καταστάσεων (Tri-state buffer)** στην έξοδο του B.

- Όταν το σήμα ελέγχου **BBUS** είναι ανενεργό ('0'), η έξοδος του B είναι σε κατάσταση υψηλής σύνθετης αντίστασης (High-Z), δηλαδή είναι ηλεκτρικά απομονωμένη από τον δίαυλο.
- Μόνο όταν η FSM δώσει εντολή (BBUS = '1'), ο buffer "ανοίγει" και τα δεδομένα του B περνούν στον δίαυλο για να φτάσουν στην ALU.

1.4 Υλοποίηση των Νέων Εντολών (ORB & XORB)

Οι νέες εντολές εκμεταλλεύονται την παραπάνω αρχιτεκτονική για να εκτελέσουν λογικές πράξεις. Η ροή δεδομένων που σχεδιάσαμε είναι η εξής:

- **ORB (1D):** Η FSM ενεργοποιεί το BBUS. Τα περιεχόμενα του B βγαίνουν στον δίαυλο. Η ALU, λαμβάνοντας το σήμα OROP, διαβάζει τον δίαυλο (είσοδος B της ALU) και τον συσσωρευτή AC (είσοδος A της ALU). Το αποτέλεσμα της πράξης OR γράφεται πίσω στον AC με την ακμή του ρολογιού.
- **XORB (1E):** Η διαδικασία είναι πανομοιότυπη, με τη διαφορά ότι η ALU λαμβάνει το σήμα XOROP.

1.5 Χρονισμός Μνήμης και Αντιμετώπιση Προβλημάτων

Κατά τη διάρκεια της εξομοίωσης, παρατηρήθηκε πρόβλημα στον συγχρονισμό μεταξύ της μνήμης RAM και του καταχωρητή DR. Επειδή χρησιμοποιήσαμε σύγχρονη μνήμη (IP Catalog RAM), τα δεδομένα εμφανίζονταν στην έξοδο της μνήμης με καθυστέρηση ενός κύκλου, με αποτέλεσμα ο DR να διαβάζει λανθασμένες τιμές στον κύκλο Fetch.

Για να λυθεί αυτό το πρόβλημα χωρίς να μειωθεί η συχνότητα λειτουργίας, εφαρμόσαμε τεχνική **Inverted Clock** (ανεστραμμένου ρολογιού) στη μνήμη ή χρήση **Ασύγχρονης Ανάγνωσης** (Asynchronous Read). Έτσι, εξασφαλίσαμε ότι τα δεδομένα είναι σταθερά στον δίαυλο πριν την ενεργή ακμή του ρολογιού που δειγματοληπτεί ο DR, επιτυγχάνοντας σωστή ανάκληση των εντολών.

2. Υλοποίηση σε VHDL και περιγραφή κώδικα

Σε αυτή την ενότητα αναλύονται τα επιμέρους τμήματα κώδικα VHDL που τροποποιήθηκαν ή δημιουργήθηκαν εκ νέου για την υλοποίηση της ζητούμενης αρχιτεκτονικής. Η έμφαση δίνεται στη μετάβαση σε Hardwired Control, στην ενσωμάτωση του Καταχωρητή B και στον συγχρονισμό της μνήμης.

2.1 Καλωδιωμένη Μονάδα Ελέγχου (Hardwired Control Unit)

Το αρχείο hardwired.vhd αντικατέστησε τον παλιό mseq.vhd. Αντί να χρησιμοποιήσουμε μετρητές και decoders, ορίσαμε μια **FSM (Finite State Machine)**.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hardwired is
    PORT (
        ir    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0); -- Είσοδος: Ο κωδικός εντολής (Opcode) από τον IR
        clock : IN  STD_LOGIC;
        reset : IN  STD_LOGIC;                    -- Ασύγχρονο Reset
        z     : IN  STD_LOGIC;                    -- Flag μηδενικού αποτελέσματος (Zero Flag)
        mOPs  : OUT STD_LOGIC_VECTOR(28 DOWNTO 0) -- Έξοδος: Διάνυσμα ελέγχου (Control Signals) για όλο το Data Path
    );
END hardwired;

ARCHITECTURE bhv OF hardwired IS
    -- Ορισμός των καταστάσεων της Μηχανής Πεπερασμένων Καταστάσεων (FSM).
    -- S_FETCH1-3: Κύκλοι ανάκλησης εντολής.
    -- S_EXECUTE: Κύκλος εκτέλεσης.
    -- S_SKIP_OPERAND: Νέα κατάσταση για τη διαχείριση δίμπαιτων εντολών.
    TYPE state_type IS (S_RESET, S_FETCH1, S_FETCH2, S_FETCH3, S_EXECUTE, S_SKIP_OPERAND);
    SIGNAL current_state, next_state : state_type;
BEGIN

    -- Sequential Logic
    -- Υπεύθυνη για τη μετάβαση στην επόμενη κατάσταση με βάση το ρολόι.
    PROCESS (clock, reset)
    BEGIN
        IF reset = '1' THEN
            current_state <= S_RESET; -- Σε περίπτωση Reset, πάμε στην αρχική κατάσταση
        ELSIF rising_edge(clock) THEN
            current_state <= next_state; -- Στην ακμή του ρολογιού, ενημερώνεται η κατάσταση
        END IF;
    END PROCESS;

   -- Combinational Logic
   -- Εδώ καθορίζονται τα σήματα εξόδου (mOPs) και η επόμενη κατάσταση.
   PROCESS (current_state, ir, z)
   BEGIN
       -- Default τιμές για αποφυγή δημιουργίας Latches.
       -- Μηδενίζουμε όλα τα σήματα ελέγχου στην αρχή κάθε κύκλου.
       mOPs <= (others => '0');
       next_state <= S_FETCH1; -- Default επόμενη κατάσταση είναι η Fetch

       CASE current_state IS
           -- Κατάσταση Reset: Απλή μετάβαση στην αρχή
           WHEN S_RESET =>
               next_state <= S_FETCH1;

           -- === FETCH CYCLE (Κύκλος Ανάκλησης) ===

           -- T0: Μεταφορά διεύθυνσης από PC -> AR
           WHEN S_FETCH1 =>
               mOPs(12) <= '1'; -- PCBUS: Ο PC βγάζει τη διεύθυνση στον δίαυλο
               mOPs(0)  <= '1'; -- ARLOAD: Ο AR φορτώνει τη διεύθυνση
               next_state <= S_FETCH2;

           -- T1: Ανάγνωση Μνήμης -> DR και αύξηση PC
           WHEN S_FETCH2 =>
               mOPs(17) <= '1'; -- MEMBUS: Η μνήμη βγάζει δεδομένα στον δίαυλο
               mOPs(2)  <= '1'; -- DRLOAD: Ο DR αποθηκεύει την εντολή
               mOPs(9)  <= '1'; -- PCINC: Αυξάνουμε τον PC για να δείχνει στην επόμενη θέση
               next_state <= S_FETCH3;

           -- T2: Μεταφορά Εντολής DR -> IR
           -- Σημαντικό: Εδώ έχουμε πλέον τον Opcode στον IR για να αποφασίσουμε μετά.
           WHEN S_FETCH3 =>
               mOPs(13) <= '1'; -- DRBUS: Ο DR βγάζει την εντολή στον δίαυλο
               mOPs(3)  <= '1'; -- IRLOAD: Ο IR αποθηκεύει την εντολή
               next_state <= S_EXECUTE; -- Πάμε για εκτέλεση
```

```
-- === EXECUTE CYCLE (Κύκλος Εκτέλεσης - T3) ===
WHEN S_EXECUTE =>
    next_state <= S_FETCH1; -- Μετά την εκτέλεση, επιστρέφουμε στο Fetch (εκτός αν αλλάξει παρακάτω)

    -- Αποκωδικοποίηση της εντολής (Opcode Decoding)
    CASE ir IS
        -- Διαχείριση Δίμπαιτων Εντολών (LDAC/STAC)
        -- Επειδή ακολουθεί αριθμός (operand), πρέπει να τον προσπεράσουμε
        WHEN x"10" | x"20" =>
             next_state <= S_SKIP_OPERAND;

        -- Λειτουργία: AC = AC OR B(opcode 1D)
        WHEN x"1D" =>
            mOPs(27) <= '1'; -- BBUS: Ενεργοποιούμε τον Tri-state buffer του B (δημιουργήθηκε νέο σήμα)
            mOPs(20) <= '1'; -- OROP: Εντολή στην ALU για λογική πράξη OR
            mOPs(6)  <= '1'; -- ACLOAD: Αποθήκευση του αποτελέσματος στον AC
            mOPs(7)  <= '1'; -- ZLOAD: Ενημέρωση του Zero Flag

        -- Λειτουργία: AC = AC XOR B(opcode 1E)
        WHEN x"1E" =>
            mOPs(27) <= '1'; -- BBUS: Ο B βγαίνει στον δίαυλο (RBUS=0)
            mOPs(21) <= '1'; -- XOROP: Εντολή στην ALU για λογική πράξη XOR
            mOPs(6)  <= '1'; -- ACLOAD: Αποθήκευση στον AC
            mOPs(7)  <= '1'; -- ZLOAD: Ενημέρωση Z Flag

        -- INAC (Opcode A0) - Χρησιμοποιείται για έλεγχο (AC++)
        WHEN x"A0" =>
            mOPs(23) <= '1'; -- ACINC: Αύξηση του AC κατά 1
            mOPs(6)  <= '1'; -- ACLOAD
            mOPs(7)  <= '1'; -- ZLOAD

                WHEN OTHERS => NULL; -- Για άλλες εντολές δεν κάνουμε τίποτα (NOP)
            END CASE;


            -- Κατάσταση SKIP: Αυξάνει τον PC κατά 1 ακόμα φορά.
            -- Αυτό χρειάζεται στις εντολές που πιάνουν 2 θέσεις μνήμης (Εντολή + Δεδομένο),
            -- ώστε ο PC να δείχνει στην επόμενη πραγματική εντολή και όχι στο δεδομένο.
            WHEN S_SKIP_OPERAND =>
                mOPs(9) <= '1'; -- PCINC
                next_state <= S_FETCH1;

        END CASE;
    END PROCESS;

END behavioral;
```

TYPE state_type IS (S_RESET, S_FETCH1, S_FETCH2, S_FETCH3, S_EXECUTE, S_SKIP_OPERAND);

SIGNAL current_state, next_state : state_type;

...

CASE current_state IS

-- Κύκλοι FETCH (T0, T1, T2)

WHEN S_FETCH1 => mOPs(12)<='1'; mOPs(0)<='1'; ... -- AR <- PC

WHEN S_FETCH2 => mOPs(17)<='1'; mOPs(2)<='1'; mOPs(9)<='1'; ... -- DR <- MEM, PC++

WHEN S_FETCH3 => mOPs(13)<='1'; mOPs(3)<='1'; ... -- IR <- DR

Ανάλυση:

- Ορίσαμε την κατάσταση **S_FETCH3** έτσι ώστε να μεταφέρει τα δεδομένα από τον DR στον IR (mOPs(13) = DRBUS, mOPs(3) = IRLOAD). Αυτό είναι κρίσιμο γιατί σε αυτό το σημείο ο IR αποκτά τον 8-bit κωδικό της εντολής (π.χ. 1D ή 1E).
- Η κατάσταση **S_EXECUTE** περιέχει ένα CASE ir IS statement, το οποίο λειτουργεί ως αποκωδικοποιητής εντολών.

<u>Υλοποίηση Νέων Εντολών (ORB / XORB):</u>

WHEN x"1D" => -- ORB

mOPs(27) <= '1'; -- BBUS: Ενεργοποίηση Tri-state του B

mOPs(20) <= '1'; -- OROP: Εντολή OR στην ALU

mOPs(6) <= '1'; -- ACLOAD: Αποθήκευση αποτελέσματος

mOPs(7) <= '1'; -- ZLOAD: Ενημέρωση Flag Z

Εδώ φαίνεται καθαρά η Hardwired λογική: Μόλις το IR γίνει 1D (Hex), το κύκλωμα ενεργοποιεί ταυτόχρονα το bit 27 (για να βγει ο B στον δίαυλο) και το bit 20 (για να κάνει OR η ALU).

2.2 Σύστημα Διαύλου (Data Bus)

Στο αρχείο data_bus.vhd, η κύρια αλλαγή ήταν η προσθήκη της εισόδου του Καταχωρητή B στον πολυπλέκτη που οδηγεί τον δίαυλο.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY data_bus IS
PORT (

    pc_out : IN  STD_LOGIC_VECTOR(15 DOWNTO 0); -- PC (16-bit)
    dr_out, tr_out, r_out, ac_out, mem_out : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8-bit καταχωρητές
    b_out : IN STD_LOGIC_VECTOR(7 DOWNTO 0);    -- Η έξοδος του Καταχωρητή B (8-bit)

    -- Κάθε σήμα (π.χ. drbus) λειτουργεί ως Enable για τον Tri-state buffer του αντίστοιχου καταχωρητή
    pcbus, drbus, trbus, rbus, acbus, membus, busmem : IN STD_LOGIC;

    bbus : IN STD_LOGIC; -- Enable για να βγει ο B στον δίαυλο

    -- Έξοδοι συστήματος
    dbus        : OUT STD_LOGIC_VECTOR(15 DOWNTO 0); -- Ο κεντρικός δίαυλος 16-bit
    mem_data_in : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)   -- Δεδομένα προς εγγραφή στη Μνήμη
);
END data_bus;

ARCHITECTURE bhv OF data_bus IS

    -- Εσωτερικό σήμα για την προσομοίωση της συμπεριφοράς του διαύλου
    SIGNAL internal_bus : STD_LOGIC_VECTOR(15 DOWNTO 0);

BEGIN

    --Combinational Logic
    -- όλα τα σήματα εισόδου στην process, καθώς η έξοδος αλλάζει άμεσα.
    PROCESS (pc_out, dr_out, tr_out, r_out, ac_out, mem_out, b_out,
             pcbus, drbus, trbus, rbus, acbus, membus, bbus)
    BEGIN
        -- Default State: High Impedance (Υψηλή Σύνθετη Αντίσταση - 'Z')
        -- Αυτό είναι κρίσιμο: Αν κανένα σήμα ελέγχου δεν είναι '1', ο δίαυλος είναι floating.
        -- Έτσι προσομοιώνουμε σωστά τα ηλεκτρονικά Tri-state buffers και αποφεύγουμε συγκρούσεις.
        internal_bus <= (others => 'Z');

        -- Priority Encoder Logic
        -- Ελέγχουμε ποιο σήμα ελέγχου είναι ενεργό και οδηγούμε τα αντίστοιχα δεδομένα στον δίαυλο.

        IF (pcbus = '1') THEN
            internal_bus <= pc_out; -- Ο PC είναι 16-bit, περνάει ως έχει.

         ELSIF (drbus = '1') THEN
             -- Padding: Ο DR είναι 8-bit. Γεμίζουμε τα 8 MSB με μηδενικά (x"00").
             internal_bus <= x"00" & dr_out;

         ELSIF (trbus = '1') THEN
             internal_bus <= x"00" & tr_out;

         ELSIF (rbus = '1') THEN
             internal_bus <= x"00" & r_out;

         ELSIF (acbus = '1') THEN
             internal_bus <= x"00" & ac_out;

         ELSIF (membus = '1') THEN
             internal_bus <= x"00" & mem_out;

         -- Όταν η FSM ενεργοποιήσει το σήμα 'bbus' (π.χ. στην εντολή ORB),
         -- τα περιεχόμενα του B οδηγούνται στον δίαυλο.
         ELSIF (bbus = '1') THEN
             internal_bus <= x"00" & b_out;

         END IF;
     END PROCESS;

    dbus <= internal_bus;

    -- Στέλνουμε δεδομένα στη RAM μόνο όταν το σήμα εγγραφής (busmem) είναι ενεργό.
    -- Αλλιώς στέλνουμε '0' για ασφάλεια.
    mem_data_in <= internal_bus(7 DOWNTO 0) WHEN busmem = '1' ELSE (others => '0');

END bhv;
```

PORT (... b_out : IN STD_LOGIC_VECTOR(7 DOWNTO 0); bbus : IN STD_LOGIC ...);

...

```
PROCESS (...)

BEGIN

   internal_bus <= (others => 'Z'); -- Default High-Impedance

   IF    (pcbus = '1') THEN internal_bus <= pc_out;

   ELSIF (drbus = '1') THEN internal_bus <= x"00" & dr_out;

   ...

   ELSIF (bbus = '1') THEN  -- ΝΕΑ ΠΡΟΣΘΗΚΗ

      internal_bus <= x"00" & b_out; -- Ο Β οδηγεί τον δίαυλο

   END IF;

END PROCESS;
```

Ανάλυση: Το σήμα bbus λειτουργεί ως σήμα επιλογής (Enable) για τον **Tri-state buffer** του καταχωρητή Β. Όταν είναι '1', τα περιεχόμενα του b_out περνούν στον internal_bus. Αν δεν είχαμε αυτή τη γραμμή κώδικα (ELSIF bbus...), ο καταχωρητής Β θα ήταν "αόρατος" για το υπόλοιπο σύστημα και η ALU δεν θα μπορούσε να διαβάσει τα δεδομένα του.

2.3 Ελεγκτής ALU (ALU Controller)

Στο αρχείο alus.vhd, έπρεπε να διορθώσουμε τη λογική επιλογής πράξης. Στην αρχική σχεδίαση, η ALU εκτελούσε λογικές πράξεις μόνο αν ήταν ενεργό το σήμα RBUS.

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

-- Λαμβάνει τα σήματα ελέγχου υψηλού επιπέδου από την FSM (π.χ. "κάνε πρόσθεση", "φόρτωσε AC")
-- και παράγει τον κωδικό επιλογής 7-bit (alus) που ελέγχει τους πολυπλέκτες μέσα στην ALU.
entity alus IS
port(
    --Σήματα ελέγχου κατευθείαν από mOPs
    rbus, acload, zload, andop, orop, notop, xorop, aczero, acinc, plus, minus, drbus : in std_logic;

    -- Έξοδος: Ο κωδικός λειτουργίας που πάει κατευθείαν στην ALU (alu.vhd)
    alus : out std_logic_vector(6 downto 0)
);
end alus ;

architecture arc of alus is
    -- Εσωτερικό σήμα για τη συνένωση όλων των εισόδων ελέγχου
    signal control : std_logic_vector(11 downto 0);
begin
    -- Concatenation των σημάτων ελέγχου σε ένα διάνυσμα 12-bit.
    -- Αυτό μας επιτρέπει να ελέγξουμε όλες τις συνθήκες ταυτόχρονα με ένα CASE statement.
    -- Η σειρά των bits είναι:
    -- 11:rbus, 10:drbus, 9:acload, 8:zload, 7:andop, 6:orop, 5:notop, 4:xorop, ...
    control <= rbus & drbus & acload & zload & andop & orop & notop & xorop & aczero & acinc & plus & minus ;

    process(control)
    begin
        case control is
```

```
            -- AND: rbus=1, acload=1, zload=1, andop=1 -> Output: Select AND Mux
            WHEN "101110000000" => alus <= "1000000" ;

            -- OR: rbus=1, acload=1, zload=1, orop=1 -> Output: Select OR Mux
            WHEN "101101000000" => alus <= "1100000" ;

            WHEN "001100100000" => alus <= "1110000" ; -- NOT (Μονόμπαιτη, δεν θέλει R)
            WHEN "101100010000" => alus <= "1010000" ; -- XOR (με R)
            WHEN "001100001000" => alus <= "0000000" ; -- CLAC (Clear AC)
            WHEN "001100000100" => alus <= "0001001" ; -- INAC (Increment AC)
            WHEN "101100000010" => alus <= "0000101" ; -- ADD (με R)
            WHEN "101100000001" => alus <= "0001011" ; -- SUB (με R)
            WHEN "101100000000" => alus <= "0000100" ; -- MOVR
            WHEN "011100000000" => alus <= "0000100" ; -- LDAC5

            -- Στις εντολές ORB/XORB, το 'rbus' είναι '0' (Bit 11 = 0), διότι
            -- χρησιμοποιούμε τον 'bbus' (που δεν ελέγχεται εδώ αλλά στο data_bus).
            -- Ωστόσο, πρέπει να δώσουμε στην ALU τον ίδιο κωδικό πράξης.

            -- ORB: rbus=0, orop=1 (Bit 6).
            -- Η έξοδος "1100000" είναι η ίδια με το απλό OR, γιατί η ALU
            -- δεν ενδιαφέρεται από πού ήρθαν τα δεδομένα, αλλά τι πράξη να κάνει.
            WHEN "001101000000" => alus <= "1100000" ;

            -- XORB: rbus=0, xorop=1 (Bit 4).
            -- Ομοίως, στέλνουμε τον κωδικό XOR στην ALU.
            WHEN "001100010000" => alus <= "1010000" ;

            -- Default περίπτωση: NOP (No Operation)
            WHEN others => alus <= "1111111" ;
        end case;
    end process;
end arc ;
```

-- Παλιά λογική (για πράξεις με R)

WHEN "101101000000" => alus <= "1100000"; -- OR (RBUS=1)

-- Νέα λογική (για πράξεις με B, όπου RBUS=0)

WHEN "001101000000" => alus <= "1100000"; -- ORB (RBUS=0, OROP=1)

WHEN "001100010000" => alus <= "1010000"; -- XORB (RBUS=0, XOROP=1)

Ανάλυση: Προσθέσαμε δύο νέες περιπτώσεις στο CASE statement. Παρατηρούμε ότι το πρώτο bit (που αντιστοιχεί στο RBUS) είναι **'0'**. Αυτό επιτρέπει στην ALU να εκτελεί την πράξη OR/XOR ακόμα και όταν τα δεδομένα έρχονται από τον B (μέσω του BBUS) και όχι από τον R. Χωρίς αυτή την αλλαγή, η ALU θα παρέμενε αδρανής κατά την εκτέλεση των ORB/XORB.

2.4 Διασύνδεση CPU και Μνήμης (Top Level Entity)

Στο αρχείο rs_cpu.vhd (Top Level), έγιναν οι συνδέσεις των νέων components. Το πιο κρίσιμο σημείο ήταν η διαχείριση του χρονισμού της μνήμης RAM.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE work.cpulib.all;

ENTITY rs_cpu IS
PORT(

    ARdata, PCdata : buffer std_logic_vector(15 downto 0);
    DRdata, ACdata : buffer std_logic_vector(7 downto 0);
    IRdata, TRdata : buffer std_logic_vector(7 downto 0);
    RRdata         : buffer std_logic_vector(7 downto 0);
    Bdata          : buffer std_logic_vector(7 downto 0);
    ZRdata         : buffer std_logic;
    clock, reset   : in std_logic;
    mOP            : buffer std_logic_vector(28 downto 0);
    addressBus     : buffer std_logic_vector(15 downto 0);
    dataBus        : buffer std_logic_vector(7 downto 0)
);
END rs_cpu;

ARCHITECTURE arc OF rs_cpu IS

    signal internal_bus    : std_logic_vector(15 downto 0); -- Ο Κεντρικός Δίαυλος 16-bit
    signal ram_q_out       : std_logic_vector(7 downto 0);  -- Έξοδος δεδομένων από RAM
    signal mem_data_to_ram : std_logic_vector(7 downto 0);  -- Είσοδος δεδομένων προς RAM
    signal alu_result      : std_logic_vector(7 downto 0);  -- Αποτέλεσμα πράξης ALU
    signal alu_z_flag      : std_logic;                     -- Zero Flag από την ALU
    signal alus_ctrl       : std_logic_vector(6 downto 0);  -- Κωδικός ελέγχου ALU

BEGIN

    -- Αντικατάσταση του παλιού Sequencer με την Hardwired FSM.
    -- Συνδέουμε το IR για να διαβάζει τα Opcodes και παράγει τα 29 bits ελέγχου (mOPs).
    CONTROL_UNIT: hardwired
PORT MAP (
            ir => IRdata,
            clock => clock,
            reset => reset,
            z => ZRdata,
            mOPs => mOP
          );

    -- Μεταφράζει τα σήματα της FSM (π.χ. mOP(20)=OROP) σε εντολές για την ALU.
    -- Συνδέουμε και το mOP(15) (RBUS) και το mOP(13) (DRBUS) για σωστή επιλογή εισόδων.
    ALU_CONTROLLER: alus
PORT MAP (
           andop => mOP(19),
           orop => mOP(20),
           xorop => mOP(21),
           notop => mOP(22),
           acinc => mOP(23),
           aczero => mOP(24),
           plus => mOP(25),
           minus => mOP(26),
           rbus => mOP(15),
           acload => mOP(6),
           zload => mOP(7),
           drbus => mOP(13),
           alus => alus_ctrl
         );

   -- Ο Πολυπλέκτης που αποφασίζει ποιος γράφει στον δίαυλο.
   -- Προστέθηκε η σύνδεση του B_out και του σήματος ελέγχου BBUS (mOP(27)).
   BUS_SYSTEM: data_bus
PORT MAP (
           pc_out => PCdata,
           dr_out => DRdata,
           tr_out => TRdata,
           r_out => RRdata,
           ac_out => ACdata,
           mem_out => ram_q_out,
           b_out => Bdata,  -- Σύνδεση του νέου καταχωρητή στο Bus System

           -- Mapping των σημάτων ελέγχου (Enable signals)
           pcbus => mOP(12),
           drbus => mOP(13),
           trbus => mOP(14),
           rbus => mOP(15),
           acbus => mOP(16),
           membus => mOP(17),
           busmem => mOP(18),
           bbus => mOP(27), -- Το νέο σήμα ελέγχου για τον B
           dbus => internal_bus,
           mem_data_in => mem_data_to_ram
         );
```

```
-- Instantiation των καταχωρητών και σύνδεση με τα σήματα φόρτωσης (Load) και αύξησης (Inc).
REG_PC: regnbit GENERIC MAP (n => 16) PORT MAP (internal_bus, clock, reset, mOP(1), mOP(9), PCdata);
REG_AR: regnbit GENERIC MAP (n => 16) PORT MAP (internal_bus, clock, reset, mOP(0), mOP(8), ARdata);
REG_DR: regnbit GENERIC MAP (n => 8)  PORT MAP (internal_bus(7 downto 0), clock, reset, mOP(2), '0', DRdata);
REG_IR: regnbit GENERIC MAP (n => 8)  PORT MAP (internal_bus(7 downto 0), clock, reset, mOP(3), '0', IRdata);
REG_TR: regnbit GENERIC MAP (n => 8)  PORT MAP (internal_bus(7 downto 0), clock, reset, mOP(4), '0', TRdata);
REG_R:  regnbit GENERIC MAP (n => 8)  PORT MAP (internal_bus(7 downto 0), clock, reset, mOP(5), '0', RRdata);
REG_AC: regnbit GENERIC MAP (n => 8)  PORT MAP (alu_result, clock, reset, mOP(6), '0', ACdata);

-- Register Z (Zero Flag)
-- Επειδή το regnbit είναι φτιαγμένο για vectors, και το Z είναι 1-bit signal,
-- κάνουμε map το bit(0) του din/dout για να αποφύγουμε Type Mismatch Error.
REG_Z:  regnbit GENERIC MAP (n => 1)
        PORT MAP (
            clk => clock,
            rst => reset,
            ld => mOP(7),
            inc => '0',
            din(0) => alu_z_flag,  -- Είσοδος: Το αποτέλεσμα από την ALU
            dout(0) => ZRdata      -- Εξοδος: Προς την FSM
        );

-- Υλοποίηση του καταχωρητή B (8-bit).
-- Συνδέουμε το σήμα φόρτωσης στο bit 28 (mOP(28)) για μελλοντική χρήση (Load B).
REG_B:  regnbit GENERIC MAP (n => 8)
         PORT MAP (internal_bus(7 downto 0), clock, reset, mOP(28), '0', Bdata);


-- Χρησιμοποιούμε "NOT clock" (Ανεστραμμένο Ρολόι).
-- Η IP Catalog RAM είναι σύγχρονη και διαβάζει στην ακμή. Αν χρησιμοποιούσαμε το ίδιο ρολόι
-- με τον CPU, ο DR θα προσπαθούσε να διαβάσει δεδομένα πριν αυτά είναι έτοιμα (Race Condition).
-- Με το NOT clock, η μνήμη βγάζει δεδομένα στη μέση του κύκλου, ώστε να είναι σταθερά
-- όταν ο DR κάνει latch στην επόμενη άνοδο.
MEMORY_UNIT: RAM PORT MAP (
        clock   => NOT clock,
        address => ARdata(7 downto 0),
        data    => mem_data_to_ram,
        wren    => mOP(11),
        q       => ram_q_out
    );

--Δέχεται ως είσοδο τον AC και τα 8-bit του Internal Bus.
--Αν είναι ενεργό το BBUS, το Internal Bus έχει την τιμή του B.
ALU_UNIT: alu GENERIC MAP (n => 8) PORT MAP (ACdata, internal_bus(7 downto 0), alus_ctrl, alu_result, alu_z_flag);
```

```
    --έξοδοι για παρατήρηση sim
    addressBus <= ARdata;
    dataBus    <= internal_bus(7 downto 0);
end arc;
```

MEMORY_UNIT: RAM PORT MAP (

clock => NOT clock, -- Inverted Clock για διόρθωση χρονισμού

address => ARdata(7 downto 0),

data => mem_data_to_ram,

wren => mOP(11),

q => ram_q_out

);

Ανάλυση:

Χρησιμοποιήσαμε την τεχνική του **ανεστραμμένου ρολογιού (NOT clock)** για τη μνήμη.

- **Το πρόβλημα:** Η IP Catalog RAM είναι σύγχρονη. Αν χρησιμοποιούσαμε το κανονικό ρολόι, όταν ο επεξεργαστής ζητούσε δεδομένα στον κύκλο FETCH2, η μνήμη θα τα έδινε στον *επόμενο* κύκλο, προκαλώντας λάθος ανάγνωση στον DR.
- **Η λύση:** Με το NOT clock, η μνήμη δειγματοληπτεί και βγάζει τα δεδομένα στο "μέσο" του κύκλου (στην κάθοδο του κυρίου ρολογιού). Έτσι, όταν έρχεται η επόμενη άνοδος (τέλος του FETCH2), τα δεδομένα είναι ήδη σταθερά στον δίαυλο και ο DR τα διαβάζει σωστά.

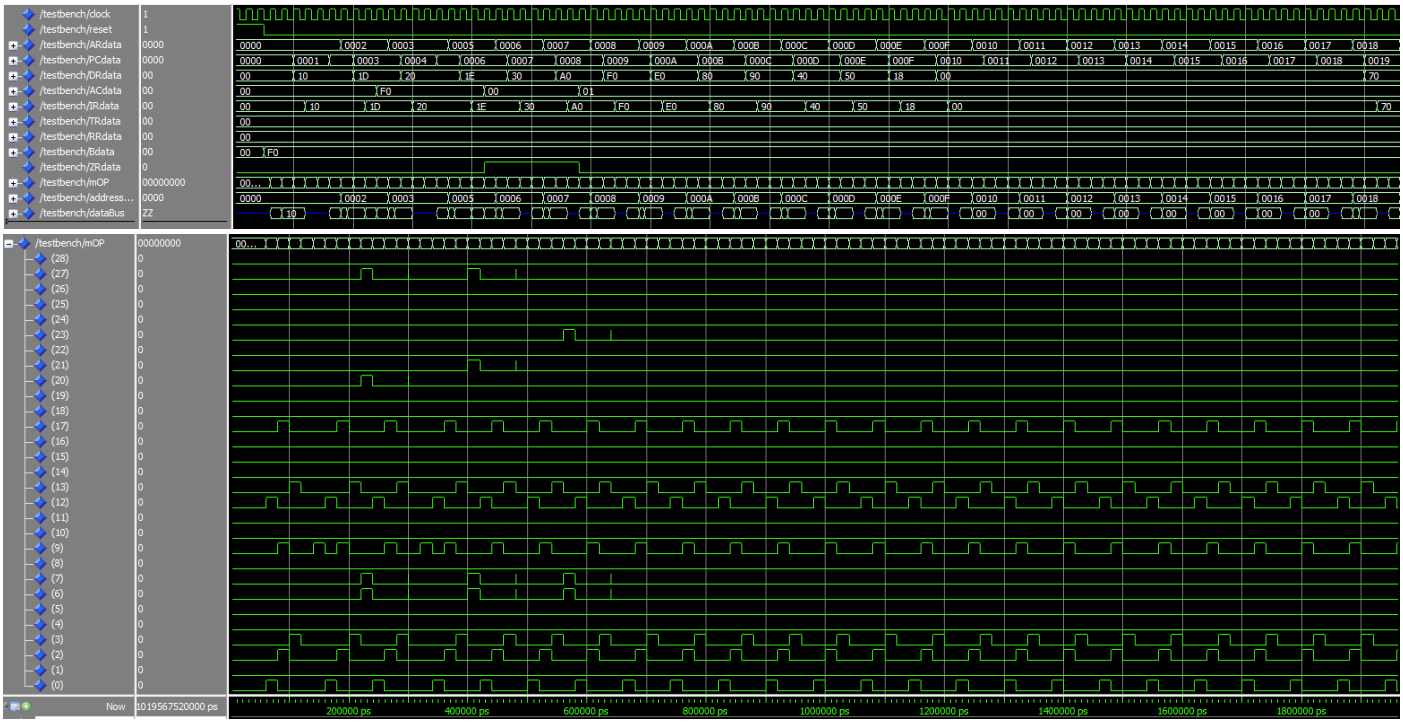
Επίσης, στο ίδιο αρχείο συνδέθηκε ο νέος καταχωρητής B:

REG_B: regnbit GENERIC MAP (n => 8)

PORT MAP (internal_bus(7 downto 0), clock, reset, mOP(28), '0', Bdata);

Συνδέσαμε το σήμα φόρτωσης του B στο bit 28 του διανύσματος ελέγχου mOP, ώστε να υπάρχει μελλοντική πρόβλεψη για εντολή φόρτωσης (Load B).

3. Αποτελέσματα Προσομοιώσεων και Περιγραφή testbench



3.1 Ροή Εκτέλεσης Προγράμματος (AR & PC)

Παρατηρώντας τα σήματα διευθυνσιοδότησης, επιβεβαιώνεται η σωστή διαχείριση των εντολών μεταβλητού μήκους (μονόμπαιτες και δίμπαιτες).

- PC (Program Counter) & AR (Address Register):

Οι τιμές του AR ακολουθούν την ακολουθία 0000 → 0002 → 0003 → 0005.

- Στη διεύθυνση **0000** εντοπίζεται η εντολή LDAC (Opcode 10), η οποία είναι δίμπαιτη (λαμβάνει όρισμα). Η FSM αναγνωρίζει τον κωδικό και μεταβαίνει στην κατάσταση S_SKIP_OPERAND. Αυτό έχει ως αποτέλεσμα ο PC να αυξηθεί δύο φορές (μια για την εντολή και μια για το όρισμα), οδηγώντας τον AR απευθείας στο **0002**, προσπερνώντας σωστά τη διεύθυνση 0001.
- Αντίστοιχα, στη διεύθυνση **0003** υπάρχει η εντολή STAC (Opcode 20), η οποία είναι επίσης δίμπαιτη, προκαλώντας άλμα του AR στο **0005**.
- Αντιθέτως, στις εντολές ORB (στο 0002) και XORB (στο 0005), η αύξηση είναι κατά 1 (π.χ. 0002 → 0003), καθώς πρόκειται για μονόμπαιτες εντολές που δεν απαιτούν όρισμα μνήμης.

3.2 Ανάλυση Δεδομένων και ALU (AC, Bdata, IR)

Η λειτουργία της ALU και η ροή δεδομένων επιβεβαιώνεται από τις τιμές των καταχωρητών AC, B και IR.

1. **Αρχικοποίηση:** Ο καταχωρητής **Bdata** έχει λάβει την τιμή **F0** (11110000) μέσω εντολής force, ενώ ο AC ξεκινά από το 00.

2. **Εκτέλεση ORB (IR = 1D):**

    - Στο χρονικό σημείο που ο IR λαμβάνει την τιμή 1D, ο καταχωρητής AC μεταβαίνει από 00 σε **F0**.(00_{16} ν $F0_{16} = F0_{16}$)

3. **Εκτέλεση XORB (IR = 1E):**

    - Όταν ο IR λαμβάνει την τιμή 1E, ο AC μεταβαίνει από F0 σε **00**.($F0_{16} \oplus F0_{16} = 00_{16}$

4. **Flag Z (ZR):**

    - Παρόλο που ο AC μηδενίστηκε, το σήμα ZR (Zero Flag) ενημερώνεται στο τέλος του κύκλου εκτέλεσης. Η σωστή λειτουργία του επιβεβαιώνεται από το γεγονός ότι ενεργοποιείται (1) όταν το αποτέλεσμα της ALU είναι μηδέν.

3.3 Σύστημα Διαύλου (DataBus)

Το σήμα DataBus εμφανίζει την τιμή ZZ (High Impedance) στα διαστήματα αδράνειας, γεγονός που αποδεικνύει ότι οι Tri-state buffers λειτουργούν σωστά και δεν υπάρχει σύγκρουση στο δίαυλο (Bus Contention).

Στις ενεργές περιόδους, ο δίαυλος μεταφέρει σωστά τους κωδικούς εντολών (10, 1D, 20, 1E) από τη μνήμη προς τον IR, καθώς και τα δεδομένα του καταχωρητή B (F0) προς την ALU κατά την εκτέλεση των νέων εντολών.

3.4 Σήματα Ελέγχου (mOPs)

Η ανάλυση του διανύσματος ελέγχου mOP επιβεβαιώνει ότι η FSM ενεργοποιεί τα σωστά bits για κάθε λειτουργία:

- **Bits 0, 2, 3 (Load Registers):** Ενεργοποιούνται διαδοχικά στους κύκλους Fetch για τη φόρτωση των AR, DR και IR.
- **Bits 12, 13, 17 (Bus Drivers):** Ελέγχουν ποιος οδηγεί τον δίαυλο (PC, DR ή Memory).
- **Bits 6, 7 (AC & Z Load):** Ενεργοποιούνται στο τέλος κάθε εντολής ALU για την αποθήκευση του αποτελέσματος.
- **Bits 20, 21 (ALU Opcodes):** Το bit **20** ενεργοποιήθηκε κατά την ORB (εντολή OR) και το bit **21** κατά την XORB (εντολή XOR).
- **Bit 27 (BBUS Enable - Κρίσιμο):** Το bit αυτό εμφανίστηκε ενεργό (1) αποκλειστικά κατά τη διάρκεια των εντολών ORB και XORB. Αυτό επιβεβαιώνει ότι η FSM άνοιξε τον tri-state buffer του καταχωρητή B μόνο όταν χρειαζόταν, επιτρέποντας στην ALU να διαβάσει τα περιεχόμενά του.

3.5 Περιγραφή του Κώδικα Testbench

```
    -- Outputs
    signal ARdata : std_logic_vector(15 downto 0);
    signal PCdata : std_logic_vector(15 downto 0);
    signal DRdata : std_logic_vector(7 downto 0);
    signal ACdata : std_logic_vector(7 downto 0);
    signal IRdata : std_logic_vector(7 downto 0);
    signal TRdata : std_logic_vector(7 downto 0);
    signal RRdata : std_logic_vector(7 downto 0);
    signal Bdata  : std_logic_vector(7 downto 0);
    signal ZRdata : std_logic;
    signal mOP    : std_logic_vector(28 downto 0);
    signal addressBus : std_logic_vector(15 downto 0);
    signal dataBus    : std_logic_vector(7 downto 0);

    -- Clock period definitions
    constant clock_period : time := 20 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: rs_cpu PORT MAP (
          ARdata => ARdata,
          PCdata => PCdata,
          DRdata => DRdata,
          ACdata => ACdata,
          IRdata => IRdata,
          TRdata => TRdata,
          RRdata => RRdata,
          Bdata  => Bdata,
          ZRdata => ZRdata,
          clock  => clock,
```

```
        reset  => reset,
        mOP    => mOP,
        addressBus => addressBus,
        dataBus    => dataBus
      );

  -- Clock process
  clock_process :process
  begin
      clock <= '0';
      wait for clock_period/2;
      clock <= '1';
      wait for clock_period/2;
  end process;


  -- Stimulus process
  stim_proc: process
  begin
      -- Reset για 2 κύκλους
      reset <= '1';
      wait for 40 ns;

      -- Απενεργοποίηση Reset
      reset <= '0';


      wait;
  end process;

END bhv;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY testbench IS
END testbench;

ARCHITECTURE bhv OF testbench IS

    COMPONENT rs_cpu
    PORT(
         ARdata : BUFFER  std_logic_vector(15 downto 0);
         PCdata : BUFFER  std_logic_vector(15 downto 0);
         DRdata : BUFFER  std_logic_vector(7 downto 0);
         ACdata : BUFFER  std_logic_vector(7 downto 0);
         IRdata : BUFFER  std_logic_vector(7 downto 0);
         TRdata : BUFFER  std_logic_vector(7 downto 0);
         RRdata : BUFFER  std_logic_vector(7 downto 0);
         Bdata  : BUFFER  std_logic_vector(7 downto 0);
         ZRdata : BUFFER  std_logic;
         clock  : IN  std_logic;
         reset  : IN  std_logic;
         mOP    : BUFFER  std_logic_vector(28 downto 0);
         addressBus : BUFFER  std_logic_vector(15 downto 0);
         dataBus    : BUFFER  std_logic_vector(7 downto 0)
        );
    END COMPONENT;

    -- Inputs
    signal clock : std_logic := '0';
    signal reset : std_logic := '0';
```

Για την επαλήθευση της σχεδίασης χρησιμοποιήθηκε το αρχείο testbench.vhd. Πρόκειται για μια οντότητα χωρίς εισόδους και εξόδους (κενό entity), η οποία λειτουργεί ως το "κέλυφος" της εξομοίωσης. Οι βασικές λειτουργίες που επιτελεί είναι οι εξής:

1. **Δημιουργία Στιγμιοτύπου (Instantiation):** Ενσωματώνει τον επεξεργαστή (rs_cpu) ως Unit Under Test (UUT), συνδέοντας τις θύρες του με εσωτερικά σήματα για την παρακολούθηση των τιμών (PC, AC, IR, Bus, κ.λπ.).

2. **Παραγωγή Ρολογιού (Clock Generation):** Η διεργασία clock_process παράγει ένα τετραγωνικό σήμα ρολογιού με περίοδο **20 ns** (συχνότητα 50 MHz), το οποίο τροφοδοτεί συνεχώς τον επεξεργαστή.

3. **Διαδικασία Reset (Stimulus Process):** Η διεργασία stim_proc ενεργοποιεί το σήμα reset ('1') στην αρχή της εξομοίωσης για **40 ns** (2 κύκλους ρολογιού) και στη συνέχεια το απενεργοποιεί ('0'), επιτρέποντας στον επεξεργαστή να ξεκινήσει την εκτέλεση του προγράμματος από τη διεύθυνση 0.

3.6 Εκτέλεση στο ModelSim (Transcript Commands)

Επειδή το Testbench ελέγχει αυτόματα το Ρολόι και το Reset, δεν χρειάζεται να τα ορίσουμε χειροκίνητα. Ωστόσο, καθώς στην παρούσα αρχιτεκτονική δεν έχει υλοποιηθεί εντολή φόρτωσης του καταχωρητή B από τη μνήμη, η αρχικοποίηση του B πρέπει να γίνει μέσω του περιβάλλοντος εξομοίωσης.

Για να ληφθούν τα τελικά αποτελέσματα, εκτελούνται οι εξής εντολές στην κονσόλα (Transcript) του ModelSim:

restart -f

force -freeze sim:/testbench/uut/Bdata 11110000 0 --force την τιμή F0 για τον B

run 2000 ns –καλός χρόνος για να φανούν όλα τα data, για μεγαλύτερους χρόνους παρουσιάζονται bugs στο modelsim

wave zoom full --καλύτερο wave view

4. Συμπεράσματα

Από την υλοποίηση και την εξομοίωση προέκυψαν τα εξής:

1. **Αναγκαιότητα FSM έναντι Αποκωδικοποιητών (Decoders):** Αρχικά εξετάστηκε η υλοποίηση της μονάδας ελέγχου με χρήση αποκωδικοποιητών (Decoders). Ωστόσο, η προσέγγιση αυτή απορρίφθηκε διότι οι αποκωδικοποιητές είναι αμιγώς συνδυαστικά κυκλώματα και αδυνατούν να συγκρατήσουν το history της εκτέλεσης (δηλαδή σε ποιο βήμα βρισκόμαστε: Fetch, Decode ή Execute). Η χρήση FSM κρίθηκε απαραίτητη για τον συγχρονισμό των σταδίων και κυρίως για τη διαχείριση των δίμπαιτων εντολών (LDAC, STAC), όπου απαιτείται η εισαγωγή εμβόλιμων καταστάσεων (S_SKIP_OPERAND) για τη σωστή αύξηση του PC, κάτι που με απλούς αποκωδικοποιητές θα αύξανε δραματικά την πολυπλοκότητα του κυκλώματος.(πολλά bugs και λάθος αποτελέσματα από μεριάς Quartus)

2. **FSM vs Microcode:** Η μετάβαση σε Hardwired FSM έκανε την εκτέλεση εντολών πιο άμεση, καθώς καταργήθηκε η μνήμη μικροεντολών. Ωστόσο, η σχεδίαση των καταστάσεων (States) έγινε πιο σύνθετη, ειδικά για τη σωστή διαχείριση των δίμπαιτων εντολών (Skip Operand).

3. **Διαχείριση Διαύλου:** Η προσθήκη του Register B ανέδειξε τη σημασία των **Tri-state buffers**. Καταλάβαμε πρακτικά πώς να απομονώνουμε τον καταχωρητή από τον κοινό δίαυλο (High-Z) και να τον ενεργοποιούμε μόνο όταν χρειάζεται (σήμα BBUS), αποφεύγοντας συγκρούσεις δεδομένων.

4. **Χρονισμός Μνήμης:** Ένα σημαντικό πρόβλημα που επιλύθηκε ήταν ο συγχρονισμός CPU-RAM. Η χρήση **ανεστραμμένου ρολογιού** στη μνήμη αποδείχθηκε απαραίτητη ώστε τα δεδομένα να είναι σταθερά πριν τα διαβάσει ο επεξεργαστής.

Συμπερασματικά, η εξομοίωση επιβεβαίωσε ότι οι νέες εντολές ORB και XORB εκτελούνται σωστά. Ο επεξεργαστής λειτουργεί με έναν επιπλέον καταχωρητή γενικού σκοπού, προσφέροντας μεγαλύτερη ευελιξία στον προγραμματισμό του.

5. Βιβλιογραφία

1. **Mano, M. M., & Kime, C. R.** (2015). *Λογική και Σχεδίαση Υπολογιστών*. 4η Έκδοση, Εκδόσεις Τζιόλα.
    - **Σελίδες 385-402:** "Σχεδίαση Μονάδας Ελέγχου: Καλωδιωμένος Έλεγχος (Hardwired Control)".
    - **Σελίδες 320-335:** "Μεταφορά Δεδομένων και Καταχωρητές (Datapath & Registers)".
2. **Brown, S., & Vranesic, Z.** (2011). *Σχεδίαση Ψηφιακών Συστημάτων με VHDL*. 3η Έκδοση, Εκδόσεις Τζιόλα.
    - **Σελίδες 450-468:** "Μηχανές Πεπερασμένων Καταστάσεων (FSM) στη VHDL".
    - **Σελίδες 195-200:** "Δομική περιγραφή και διασύνδεση στοιχείων (Structural VHDL & Port Maps)".
3. **Pedroni, V. A.** (2010). *Circuit Design with VHDL*. 2nd Edition, MIT Press.
    - **Σελίδες 95-97:** "Mapping Rules and Port Maps".
    - **Σελίδες 245-255:** "Simulation and Testbenches".