



Σχεδίαση Ψηφιακών Συστημάτων

Ρόμπερτ Πολοβίνα – 23390338

Εργασία Εξαμήνου – Σχεδίαση MIPS

2024-2025

Μέρος Πρώτο: Εισαγωγή

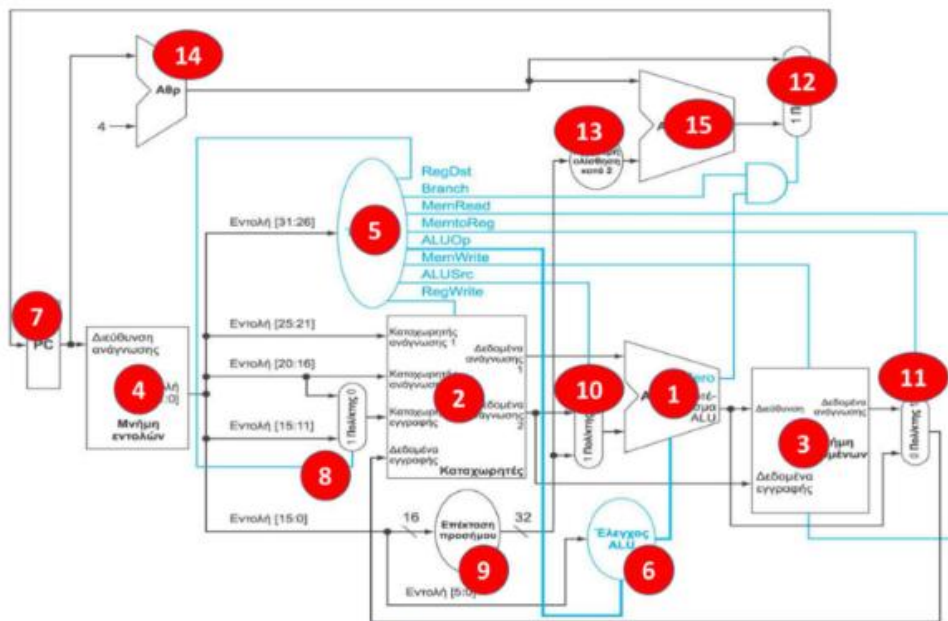
Σκοπός της εργασίας είναι η σχεδίαση και υλοποίηση του επεξεργαστή MIPS απλού κύκλου.

Ο επεξεργαστής έχει ως είσοδο τα σήματα reset και clock, ενώ δεν έχει έξοδο. Το σήμα reset οδηγεί τη μονάδα PC στην τιμή 0. Επίσης οδηγεί το Register file και μηδενίζει όλους τους καταχωρητές. Ο επεξεργαστής θα εκτελεί τις εντολές: add, sub, addi, lw, sw, bne.

Ο επεξεργαστής περιλαμβάνει τις ακόλουθες μονάδες:

1	ALU
2	Register file
3	Μνήμη δεδομένων
4	Μνήμη εντολών (ROM)
5	Μονάδα ελέγχου
6	Μονάδα ελέγχου ALU

7	PC
8	5-πλό πολυπλέκτη 2-σε-1
9	Μονάδα επέκτασης προσήμου 16-σε-32
10, 11, 12	32-πλό πολυπλέκτη 2-σε-1
13	Μονάδα ολίσθησης αριστερά κατά 2 (32-bit)
14, 15	Αθροιστή 32 bits



Εικόνα 1: Υλοποίηση Επεξεργαστή

Η παρούσα εργασία αποτελείται από τρία μέρη, με το πρώτο να είναι αυτή η εισαγωγή. Στο δεύτερο μέρος θα δούμε τους κώδικες και τους ελέγχους των διαφόρων components τα οποία αποτελούν τον MIPS, και στο τρίτο παρουσιάζεται η διασύνδεση όλων αυτών των εξαρτημάτων ώστε να δημιουργηθεί και να λειτουργήσει ο επεξεργαστής μας.

Πριν προχωρήσουμε θα ήθελα να επισημάνω τρία πράγματα. Πρώτον, οι εικόνες που συνοδεύουν το κάθε εξάρτημα έχουν επαρκή ανάλυση. Λόγω όμως της εγγενούς δυσκολίας της απεικόνισης τους σε σελίδες με διάταξη πορτραίτου, η χρήση της λειτουργίας της μεγέθυνσης κρίνεται απαραίτητη.

Δεύτερον, στα Testbenches μου δεν υλοποιώ το ρολόι μέσα σε process, όπως συνηθίζεται, αλλά σε μία μόνο γραμμή κώδικα:

```
clk <= not clk after clk_period/2;
```

Τον τρόπο αυτόν υλοποίησης τον διδάχθηκα στο εργαστήριο του μαθήματος ως μία πιο γρήγορη και απλή εναλλακτική.

Τέλος, επειδή παρατηρείται σε πάνω από ένα Testbench θα ήθελα να εξηγήσω εδώ την ύπαρξη της επόμενης γραμμής κώδικα:

```
std.env.stop;
```

Σε οποιαδήποτε Testbenches περιλαμβάνεται η παραπάνω εντολή, η προσομοίωση έτρεχε απ' αόριστον μετά την επιλογή του "Run". Η συμπεριφορά αυτή παρατηρείται και σε testbenches στα οποία χρησιμοποιούμε ρολόι αλλά και σε άλλα στα οποία δεν γίνεται χρήση ρολογιού. Επίσης, οι κώδικες είναι σχετικά απλοί και δεν διαφέρουν θεμελιωδώς μεταξύ τους. Συνεπώς δεν μπόρεσα να καταλάβω γιατί συμβαίνει αυτό το πρόβλημα σε μερικά testbenches και όχι σε άλλα.

Εν τέλει η εντολή αυτή ήταν η γρηγορότερη και ευκολότερη λύση χωρίς να επηρεάζει τα αποτελέσματα των δοκιμών και γενικότερα για τους σκοπούς της εργασίας ήταν απλά αρκετή για να σταματάω την εκτέλεση του testbench εκεί που πρέπει.

Μέρος Δεύτερο: Components

1.1.1. ALU

Κώδικας ALU:

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.numeric_std.ALL;

entity ALU is port(
    ALUin1: in std_logic_vector(31 downto 0);
    ALUin2: in std_logic_vector(31 downto 0);
    ALUctrl: in std_logic_vector(3 downto 0);
    ALUresult: out std_logic_vector(31 downto 0);
    zero: out std_logic);
end ALU;

architecture behavioral of ALU is
    signal result: std_logic_vector(31 downto 0);
begin
    process(ALUin1, ALUin2, ALUctrl)
    begin
        case(ALUctrl) is
            when "0000" => result <= ALUin1 AND ALUin2;
            when "0001" => result <= ALUin1 OR ALUin2;
            when "0010" => result <= std_logic_vector(signed(ALUin1) +
signed(ALUin2));

            when "0110" => result <= std_logic_vector(signed(ALUin1) -
signed(ALUin2));

            when others => result <= (others => '0');
        end case;
    end process;
    zero <= '1' when result = x"00000000" else '0';
    ALUresult <= result;
end behavioral;
```

Η ALU είναι υπεύθυνη να εκτελεί πράξεις αναλόγως το σήμα ελέγχου που δέχεται από την μονάδα ελέγχου της. Στο testbench ελέγχουμε την ALU κάνοντας τις πράξεις που ορίζει η εκφώνηση.

Κώδικας ALU tb:

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

entity ALU_tb is
end ALU_tb;

architecture behavior of ALU_tb is
    signal ALUin1: std_logic_vector(31 downto 0);
    signal ALUin2: std_logic_vector(31 downto 0);
    signal ALUctrl: std_logic_vector(3 downto 0);
    signal ALUresult: std_logic_vector(31 downto 0);

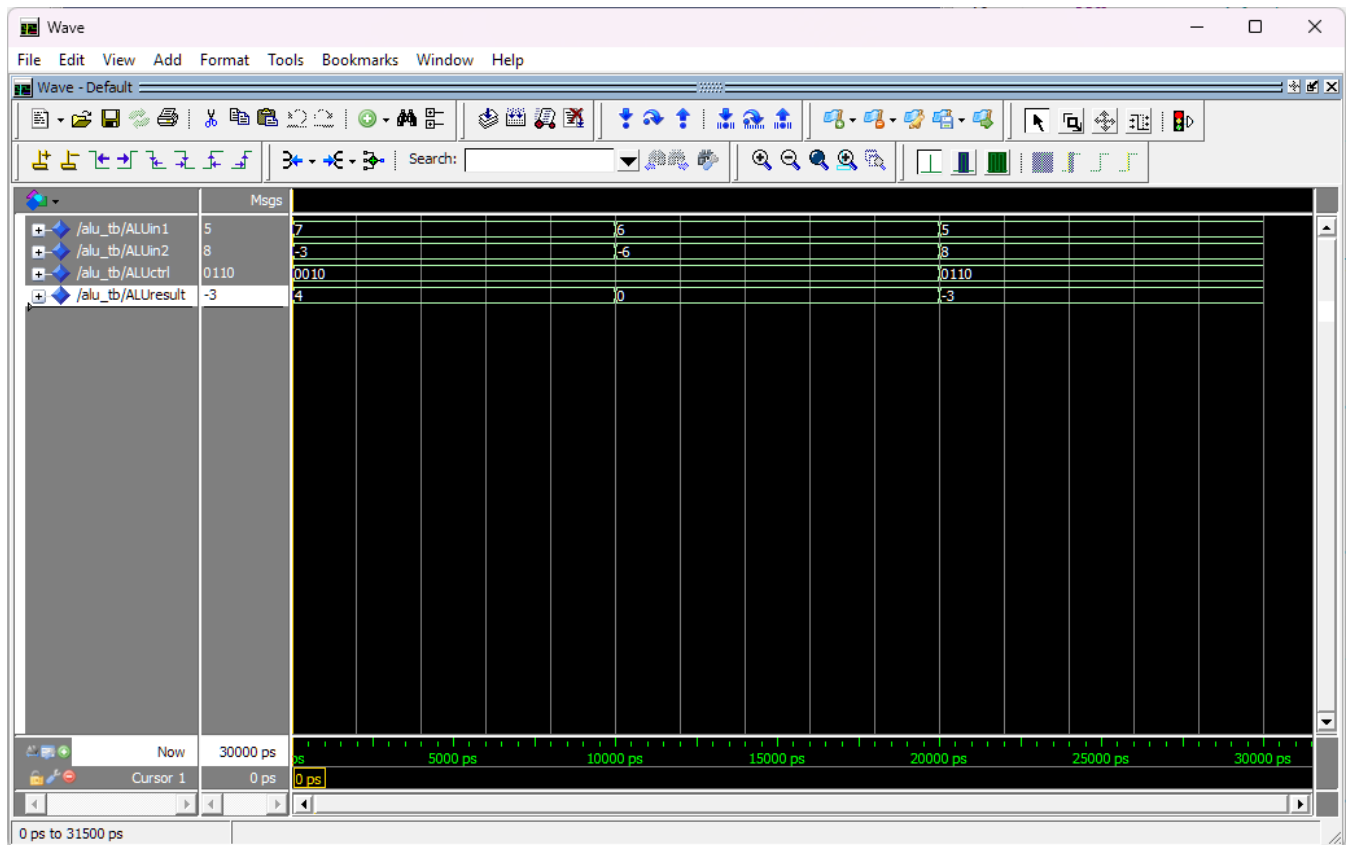
begin
    uut: entity work.ALU
    port map(
        ALUin1 => ALUin1,
        ALUin2 => ALUin2,
        ALUctrl => ALUctrl,
        ALUresult => ALUresult);

    process
    begin
        -- 7+(-3)
        ALUin1 <= std_logic_vector(to_signed(7,32));
        ALUin2 <= std_logic_vector(to_signed(-3,32));
        ALUctrl <= "0010";
        wait for 10 ns;

        -- 6+(-6)
        ALUin1 <= std_logic_vector(to_signed(6,32));
        ALUin2 <= std_logic_vector(to_signed(-6,32));
        ALUctrl <= "0010";
        wait for 10 ns;

        -- 5-8
        ALUin1 <= std_logic_vector(to_signed(5,32));
        ALUin2 <= std_logic_vector(to_signed(8,32));
        ALUctrl <= "0110";
        wait for 10 ns;

        wait;
    end process;
end behavior;
```



1.1.2. Register File

Κώδικας Register File:

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

entity Registerfile is port(
clk: in std_logic;
reset: in std_logic;
read_reg1: in std_logic_vector(4 downto 0);
read_reg2: in std_logic_vector(4 downto 0);
write_reg: in std_logic_vector(4 downto 0);
write_data: in std_logic_vector(31 downto 0);
write_enable: in std_logic;
read_data1: out std_logic_vector(31 downto 0);
read_data2: out std_logic_vector(31 downto 0));
end Registerfile;

architecture behavioral of Registerfile is
    type reg_array is array(15 downto 0) of std_logic_vector(31 downto 0);
    signal registers: reg_array := (others => (others => '0'));
begin

    read_data1 <= registers(to_integer(unsigned(read_reg1)));
    read_data2 <= registers(to_integer(unsigned(read_reg2)));

    process(clk, reset)
    begin
        if reset = '1' then
            registers <= (others => (others => '0'));
        elsif rising_edge(clk) then
            if write_enable = '1' then
                registers(to_integer(unsigned(write_reg))) <= write_data;
            end if;
        end if;
    end process;
end behavioral;
```

Το αρχείο καταχωρητών έχει υλοποιηθεί ως 16 θέσεις των 32 bits. Η ανάγνωση δεδομένων συμβαίνει εκτός του process. Πριν προχωρήσουμε στην εγγραφή των δεδομένων, ελέγχουμε για το σήμα reset. Αν υπάρχει, οι καταχωρητές μηδενίζονται.

Έπειτα ελέγχουμε το σήμα του ρολογιού και αν έχουμε σήμα εγγραφής. Αν ναι, προχωράμε στην εγγραφή των δεδομένων. Στο testbench ελέγχουμε την λειτουργία reset καθώς και τις παρακάτω:

- Εγγραφή του 6 στον καταχωρητή \$4
- Εγγραφή του 9 στον καταχωρητή \$5
- Εγγραφή του 3 στον καταχωρητή \$6
- Ανάγνωση των καταχωρητών \$4 και \$5

Κώδικας File Register tb:

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

entity Registerfile_tb is
end Registerfile_tb;

architecture behavior of Registerfile_tb is
    signal clk: std_logic := '0';
    signal reset: std_logic := '0';
    signal read_reg1: std_logic_vector(4 downto 0);
    signal read_reg2: std_logic_vector(4 downto 0);
    signal write_reg: std_logic_vector(4 downto 0);
    signal write_data: std_logic_vector(31 downto 0);
    signal write_enable: std_logic;
    signal read_data1: std_logic_vector(31 downto 0);
    signal read_data2: std_logic_vector(31 downto 0);

    constant clk_period: time := 10 ns;

begin
    uut: entity work.Registerfile port map(
        clk => clk,
        reset => reset,
        read_reg1 => read_reg1,
        read_reg2 => read_reg2,
        write_reg => write_reg,
        write_data => write_data,
        write_enable => write_enable,
        read_data1 => read_data1,
        read_data2 => read_data2);

    clk <= not clk after clk_period/2;

    process
    begin
        --Reset
        reset <= '1';
        wait for clk_period;
        reset <= '0';

        -- 6 -> $4
        write_enable <= '1';
        write_reg <= "00100";
        write_data <= std_logic_vector(to_signed(6,32));
        wait for clk_period;

        -- 9 -> $5
        write_reg <= "00101";
        write_data <= std_logic_vector(to_signed(9,32));
        wait for clk_period;

        -- 3 -> $6
        write_reg <= "00110";
```



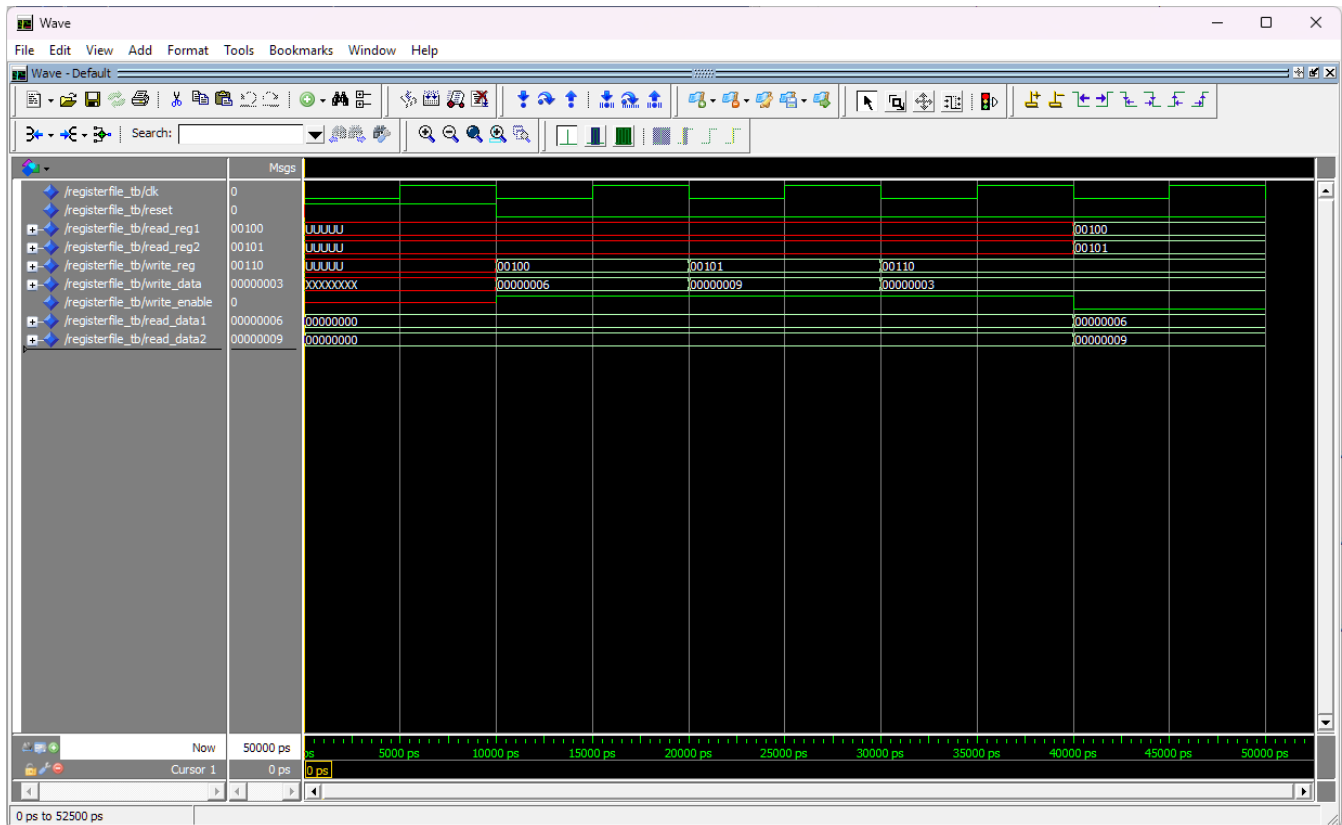
```

write_data <= std_logic_vector(to_signed(3,32));
wait for clk_period;

write_enable <= '0';
read_reg1 <= "00100"; --$4
read_reg2 <= "00101"; --$5
wait for clk_period;
std.env.stop;

wait;
end process;
end behavior;

```



1.1.3. Data Memory

Κώδικας Datamem:

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

entity Datamem is port(
  clk: in std_logic;
  mem_write: in std_logic;
  mem_read: in std_logic;
  address: in std_logic_vector(31 downto 0);
  write_data: in std_logic_vector(31 downto 0);
  read_data: out std_logic_vector(31 downto 0));
end Datamem;

architecture behavioral of Datamem is
  type memory_array is array(15 downto 0) of std_logic_vector(31 downto 0);
  signal memory: memory_array := (others => (others => '0'));

begin
  process(clk)
  begin
    if rising_edge(clk) then
      if mem_write = '1' then
        memory(to_integer(unsigned(address))) <= write_data;
      end if;
    end if;
  end process;

  process(mem_read, address, memory)
  begin
    if mem_read = '1' then
      read_data <= memory(to_integer(unsigned(address)));
    end if;
  end process;
end behavioral;
```

Η μνήμη δεδομένων έχει επίσης υλοποιηθεί ως 16 θέσεις των 32 bits. Εδώ η ανάγνωση και η εγγραφή γίνονται σε 2 διαφορετικά processes. Για την εγγραφή ελέγχουμε το σήμα εγγραφής δεδομένων και αν είναι 1, προχωράμε στην εγγραφή στην επιθυμητή θέση μνήμης.

Για την ανάγνωση ελέγχουμε εάν το σήμα ανάγνωσης δεδομένων είναι 1 και αν ναι, πραγματοποιείται η ανάγνωση των δεδομένων της θέσης μνήμης που θέλουμε.

Στο testbench ελέγχουμε τις παρακάτω λειτουργίες:

- Εγγραφή του 9 στη θέση μνήμης 2.
- Εγγραφή του 4 στη θέση μνήμης 3.
- Ανάγνωση της θέσης μνήμης 2.

Κώδικας Datamem tb:

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

entity Datamem_tb is
end Datamem_tb;

architecture behavior of Datamem_tb is
    signal clk: std_logic := '0';
    signal mem_write: std_logic;
    signal mem_read: std_logic;
    signal address: std_logic_vector(31 downto 0);
    signal write_data: std_logic_vector(31 downto 0);
    signal read_data: std_logic_vector(31 downto 0);

    constant clk_period: time := 10 ns;

begin
    uut: entity work.Datamem port map(
        clk => clk,
        mem_write => mem_write,
        mem_read => mem_read,
        address => address,
        write_data => write_data,
        read_data => read_data);

    clk <= not clk after clk_period/2;

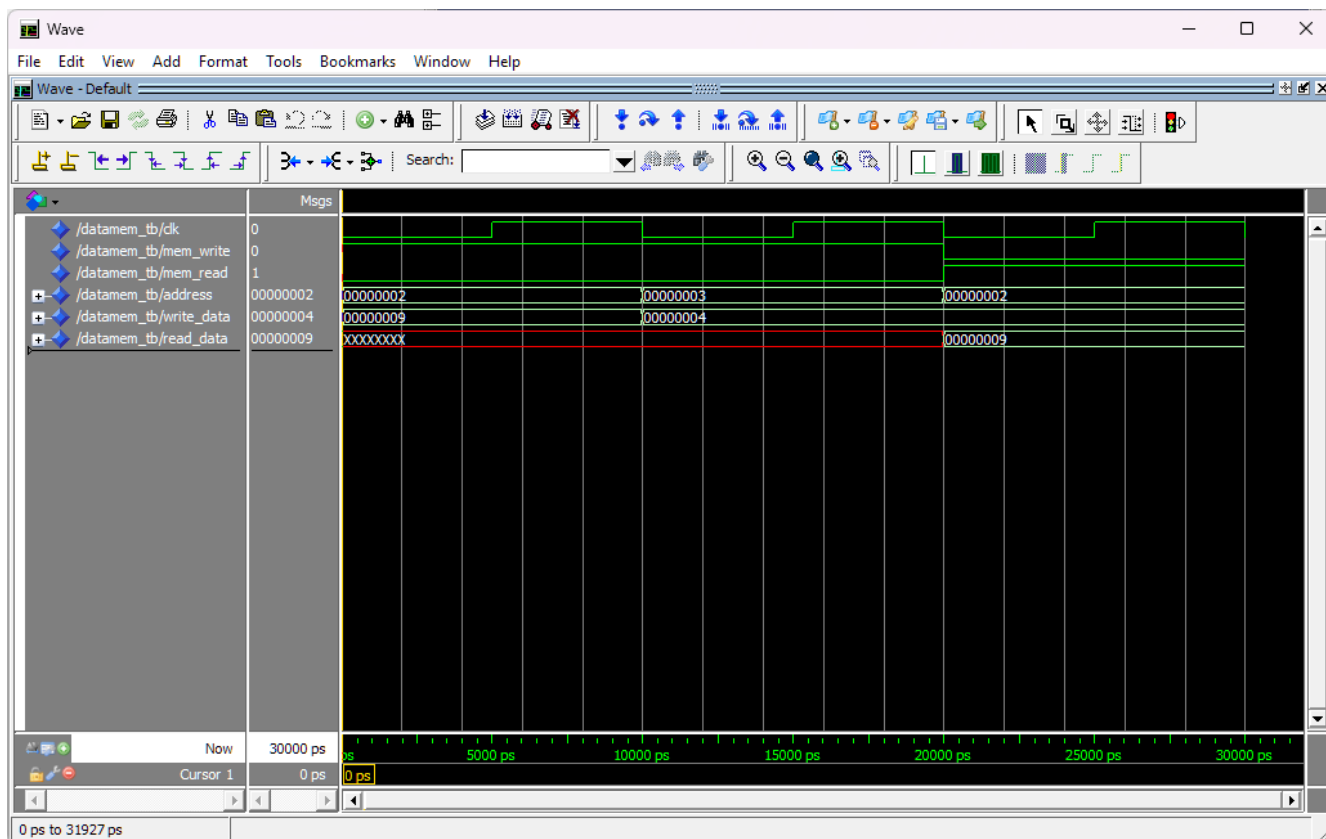
    process
    begin
        mem_write <= '1';
        mem_read <= '0';

        -- write 9 -> 2
        address <= x"00000002";
        write_data <= std_logic_vector(to_signed(9,32));
        wait for clk_period;

        -- write 4 -> 3
        address <= x"00000003";
        write_data <= std_logic_vector(to_signed(4,32));
        wait for clk_period;

        -- read <- 2
        mem_write <= '0';
        mem_read <= '1';
        address <= x"00000002";
        wait for clk_period;
        std.env.stop;

        wait;
    end process;
end behavior;
```



1.1.4. Instruction Memory

Κώδικας Imem:

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

entity Imem is port(
address: in std_logic_vector(31 downto 0);
instruction: out std_logic_vector(31 downto 0));
end Imem;

architecture behavioral of Imem is
    type mem_array is array(15 downto 0) of std_logic_vector(31 downto 0);

    signal memory: mem_array := (
        0 => x"20000000", --addi $0, $0, 0
        1 => x"20040000", --addi $4, $0, 0
        2 => x"20030001", --addi $3, $0, 1
        3 => x"20050003", --addi $5, $0, 3
        4 => x"AC830000", --sw $3, 0($4)
        5 => x"20630001", --addi $3, $3, 1
        6 => x"20840001", --addi $4, $4, 1
        7 => x"20A5FFFF", --addi $5, $5, -1
        8 => x"14A0FFFB", --bne $5, $0, L1
        others => (others => '0'));

    begin
        instruction <= memory(to_integer(unsigned(address(3 downto 0))));
    end behavioral;
```

Όπως και οι προηγούμενες μνήμες, έτσι και η μνήμη εντολών έχει υλοποιηθεί ως 16 θέσεις των 32 bits. Προς διευκόλυνση, οι ζητούμενες εντολές είναι hardcoded στην μνήμη.

Στο τέλος χρησιμοποιούμε μόνο τα τελευταία 4 bits των διευθύνσεων διότι αλλιώς το Modelsim εμφάνιζε το επόμενο σφάλμα κατά την εκκίνηση της προσομοίωσης:

“Fatal error in Architecture behavioral at C:/altera/13.1/modelsim_ase/win32aloem/3.2.1.4 Imem.vhd line 27”.

Αυτό συνέβαινε διότι μία εντολή αποτελείται από 32 bits ενώ ο πίνακας μας έχει 16 θέσεις, συνεπώς υπήρχε πιθανότητα να βγούμε εκτός ορίων.

Η πιο συνηθισμένη λύση σε απλουστευμένες προσομοιώσεις όπως αυτή, είναι να κρατάμε μόνο τα τελευταία 4 bits διότι μόνο τόσα χρειάζονται για να προσπελάσουμε θέση σε μνήμη έως 16 θέσεων ($2^4 = 16$).

Στο testbench πραγματοποιούμε ανάγνωση της εντολής που περιέχεται στην θέση μνήμης 4.

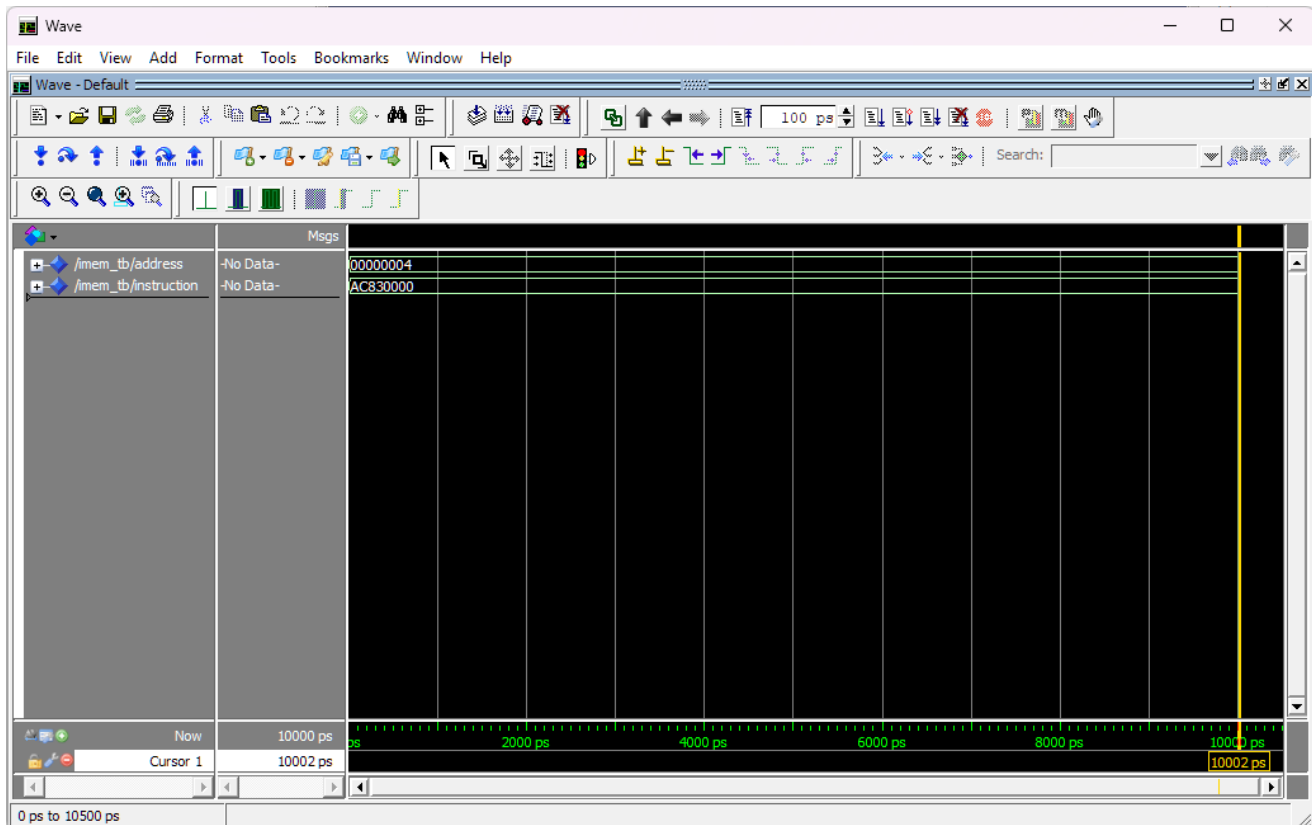
Κώδικας Imem tb:

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

entity Imem_tb is
end Imem_tb;

architecture behavior of Imem_tb is
    signal address: std_logic_vector(31 downto 0);
    signal instruction: std_logic_vector(31 downto 0);

begin
    uut: entity work.Imem
    port map(
        address => address,
        instruction => instruction);
    process
    begin
        address <= x"00000004";
        wait for 10 ns;
        std.env.stop;
    end process;
end behavior;
```



1.1.5. Control Unit

Κώδικας Control:

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

entity Control is port(
opcode: in std_logic_vector(5 downto 0);
regdst: out std_logic;
ALUsrc: out std_logic;
memtoreg: out std_logic;
regwrite: out std_logic;
memread: out std_logic;
memwrite: out std_logic;
branch: out std_logic;
ALUop: out std_logic_vector(1 downto 0));
end Control;

architecture behavioral of Control is
begin
    process(opcode)
    begin
        --Initialization
        regdst <= '0';
        ALUsrc <= '0';
        memtoreg <= '0';
        regwrite <= '0';
        memread <= '0';
        memwrite <= '0';
        branch <= '0';
        ALUop <= "00";

        case opcode is
            --Type R instr.
            when "000000" =>
                regdst <= '1';
                ALUsrc <= '0';
                memtoreg <= '0';
                regwrite <= '1';
                ALUop <= "10";

            --addi
            when "001000" =>
                regdst <= '0';
                ALUsrc <= '1';
                memtoreg <= '0';
                regwrite <= '1';
                ALUop <= "00";

            --lw
            when "100011" =>
                regdst <= '0';
                ALUsrc <= '1';
                memtoreg <= '1';
                regwrite <= '1';
```

```

    memread <= '1';
    ALUop <= "00";

    --sw
    when "101011" =>
        ALUsrc <= '1';
        memwrite <= '1';
        ALUop <= "00";

    --bne
    when "000101" =>
        branch <= '1';
        ALUop <= "01";

    when others =>
        null;

    end case;
end process;
end behavioral;

```

Η μονάδα ελέγχου είναι υπεύθυνη να διερμηνεύει τις εντολές, αναλύει τον *opcode* τους και να καθορίζει ποια σήματα θα ενεργοποιηθούν στον επεξεργαστή για την εκτέλεση της κάθε εντολής. Εδώ ξεκινάμε με μία αρχικοποίηση και συνεχίζουμε με τον ρητό καθορισμό των σημάτων που θα ενεργοποιηθούν για τις εντολές **add**, **sub**, **addi**, **lw**, **sw** και **bne**.

Στο testbench εισάγουμε τα opcodes τριών εντολών και θέλουμε να δούμε εάν η μονάδα ελέγχου θα θέσει στα σήματα τις σωστές τιμές.

Κώδικας Control tb:

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

entity Control_tb is
end Control_tb;

architecture behavior of Control_tb is
    signal opcode: std_logic_vector(5 downto 0);
    signal regdst: std_logic;
    signal ALUsrc: std_logic;
    signal memtoreg: std_logic;
    signal regwrite: std_logic;
    signal memread: std_logic;
    signal memwrite: std_logic;
    signal branch: std_logic;
    signal ALUop: std_logic_vector(1 downto 0);

begin
    uut: entity work.Control
    port map(
        opcode => opcode,
        regdst => regdst,
        ALUsrc => ALUsrc,
        memtoreg => memtoreg,
        regwrite => regwrite,
        memread => memread,
        memwrite => memwrite,
        branch => branch,
        ALUop => ALUop);

    process
    begin
        --addi $1, $0, 4
        opcode <= "001000";
        wait for 10 ns;

        --sw $6, $4
        opcode <= "101011";
        wait for 10 ns;

        --bne $5, $0, L1
        opcode <= "000101";
        wait for 10 ns;

        wait;
    end process;
end behavior;
```

- Βλέπουμε ότι για την εντολή `addi $1, $0, 4` (opcode 00100), έχουμε τα επιθυμητά σήματα που θέσαμε στον κώδικα.

+ /control_tb/opcode	001000
/control_tb/regdst	0
/control_tb/ALUsrc	1
/control_tb/memtoereg	0
/control_tb/regwrite	1
/control_tb/memread	0
/control_tb/memwrite	0
/control_tb/branch	0
+ /control_tb/ALUop	00

- Το ίδιο ισχύει και για την εντολή `sw $6, $4` (opcode 101011).

+ /control_tb/opcode	101011
/control_tb/regdst	0
/control_tb/ALUsrc	1
/control_tb/memtoereg	0
/control_tb/regwrite	0
/control_tb/memread	0
/control_tb/memwrite	1
/control_tb/branch	0
+ /control_tb/ALUop	00

- Παρομοίως και για την `bne $5, $0, L1` (opcode 000101).

+ /control_tb/opcode	000101
/control_tb/regdst	0
/control_tb/ALUsrc	0
/control_tb/memtoereg	0
/control_tb/regwrite	0
/control_tb/memread	0
/control_tb/memwrite	0
/control_tb/branch	1
+ /control_tb/ALUop	01

1.1.6. ALU Control

Κώδικας ALUcontrol:

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

entity ALUcontrol is port(
    funct: in std_logic_vector(5 downto 0);
    ALUop: in std_logic_vector(1 downto 0);
    ALUcontrol: out std_logic_vector(3 downto 0));
end ALUcontrol;

architecture behavioral of ALUcontrol is
begin
    process(funct, ALUop)
    begin
        case ALUop is
            --lw, sw, addi
            when "00" => ALUcontrol <= "0010";

            --bne
            when "01" => ALUcontrol <= "0110";

            --R type
            when "10" =>
                case funct is
                    when "100000" => ALUcontrol <= "0010";
                    when "100010" => ALUcontrol <= "0110";
                    when others => ALUcontrol <= "1111";
                end case;

            when others => ALUcontrol <= "1111";
        end case;
    end process;
end behavioral;
```

Η μονάδα ελέγχου της ALU λαμβάνει το σήμα ALUOp από την κεντρική μονάδα ελέγχου και με βάση το σήμα αυτό, «λέει» στην ALU τι πράξη πρέπει να εκτελέσει.

Ο παραπάνω κώδικας υλοποιεί ακριβώς αυτό. Για τις εντολές R-Type όμως (**add**, **sub**), χρειάζεται να ληφθεί υπόψιν και άλλη μία τιμή, η *funct*, η οποία θα καθορίσει το εξερχόμενο σήμα της μονάδας ALU control.

Στον έλεγχο του testbench δοκιμάζουμε την έξοδο για διάφορες τιμές του *funct*, με βάση την εκφώνηση.

Κώδικας ALUcontrol tb:

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

entity ALUcontrol_tb is
end ALUcontrol_tb;

architecture behavior of ALUcontrol_tb is
    signal funct: std_logic_vector(5 downto 0);
    signal ALUop: std_logic_vector(1 downto 0);
    signal ALUcontrol: std_logic_vector(3 downto 0);

begin
    uut: entity work.ALUcontrol port map(
        funct => funct,
        ALUop => ALUop,
        ALUcontrol => ALUcontrol);

    process
    begin

        funct <= "100000";
        ALUop <= "10";
        wait for 10 ns;

        funct <= "100010";
        ALUop <= "10";
        wait for 10 ns;

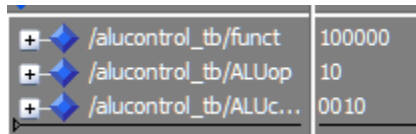
        funct <= "111111";
        ALUop <= "00";
        wait for 10 ns;

        funct <= "111111";
        ALUop <= "01";
        wait for 10 ns;

        wait;
    end process;
end behavior;
```

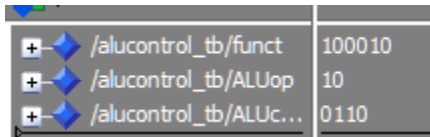
Βλέπουμε ότι:

- για $funct=100000$ & $ALUop=10$, έχουμε $ALUcontrol=0010$,



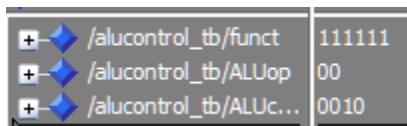
+ /alucontrol_tb/funct	100000
+ /alucontrol_tb/ALUop	10
+ /alucontrol_tb/ALUc...	0010

- για $funct=100010$ & $ALUop=10$, έχουμε $ALUcontrol=0110$,



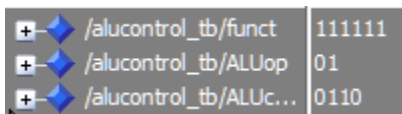
+ /alucontrol_tb/funct	100010
+ /alucontrol_tb/ALUop	10
+ /alucontrol_tb/ALUc...	0110

- για $funct=111111$ & $ALUop=00$, έχουμε $ALUcontrol=0010$ και



+ /alucontrol_tb/funct	111111
+ /alucontrol_tb/ALUop	00
+ /alucontrol_tb/ALUc...	0010

- και για $funct=111111$ & $ALUop=01$, έχουμε $ALUcontrol=0110$.



+ /alucontrol_tb/funct	111111
+ /alucontrol_tb/ALUop	01
+ /alucontrol_tb/ALUc...	0110

Τα αποτελέσματα συνάδουν πλήρως με τον κώδικα.

1.1.7. Program Counter

Κώδικας PC:

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

entity PC is port(
clk: in std_logic;
reset: in std_logic;
pc_in: in std_logic_vector(31 downto 0);
pc_out: out std_logic_vector(31 downto 0));
end PC;

architecture behavioral of PC is
    signal pc_reg: std_logic_vector(31 downto 0);

    begin
        process(clk, reset)
            begin
                if reset = '1' then
                    pc_reg <= (others => '0');
                elsif rising_edge(clk) then
                    pc_reg <= pc_in;
                end if;
            end process;

            pc_out <= pc_reg;
        end behavioral;
```

Ο Program Counter είναι υπεύθυνος να δείχνει στην επόμενη εντολή του προγράμματος που η CPU πρέπει να εκτελέσει. Δέχεται ως είσοδο μια διεύθυνση μνήμης η οποία είναι και η έξοδος του. Στο testbench δοκιμάζουμε ως είσοδο δύο τιμές και βλέπουμε αν η έξοδος του PC θα είναι σωστή.

Κώδικας PC tb:

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

entity PC_tb is
end PC_tb;

architecture behavior of PC_tb is
    signal clk: std_logic := '0';
    signal reset: std_logic := '0';
    signal pc_in: std_logic_vector(31 downto 0);
    signal pc_out: std_logic_vector(31 downto 0);

    constant clk_period: time := 10 ns;

begin
    uut: entity work.PC
        port map(
            clk => clk,
            reset => reset,
            pc_in => pc_in,
            pc_out => pc_out);

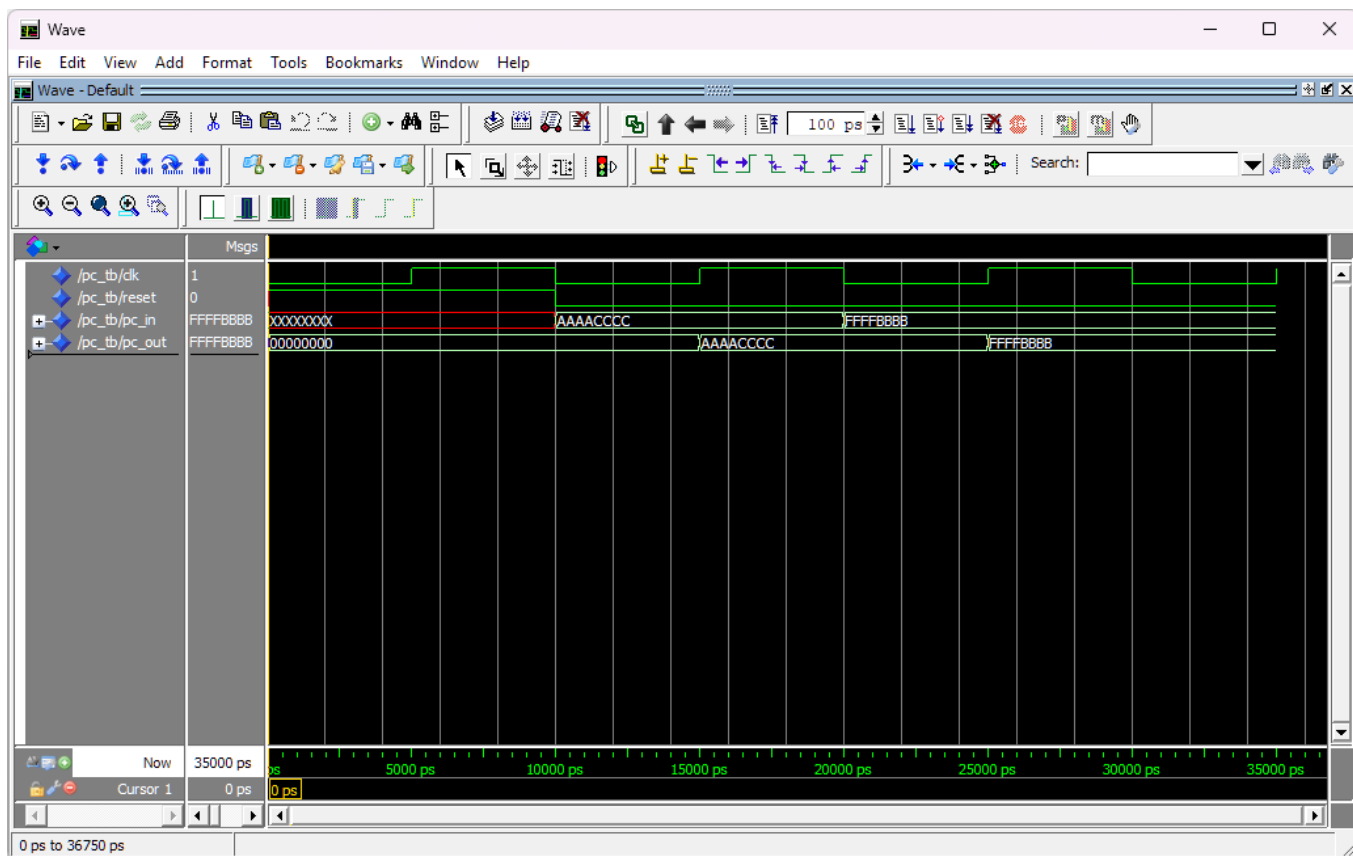
    clk <= not clk after clk_period/2;

    process
    begin
        reset <= '1';
        wait for clk_period;
        reset <= '0';

        pc_in <= x"AAAACCCC";
        wait for clk_period;

        pc_in <= x"FFFFBBBB";
        wait for clk_period;

        wait;
    end process;
end behavior;
```



1.1.8. 5MUX2to1

Κώδικας 5Mux2to1:

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

entity Mux5_2to1 is port(
in0: in std_logic_vector(4 downto 0);
in1: in std_logic_vector(4 downto 0);
set: in std_logic;
output: out std_logic_vector(4 downto 0));
end Mux5_2to1;

architecture behavioral of Mux5_2to1 is
begin
    process(in0, in1, set)
    begin
        if set = '0' then
            output <= in0;
        else output <= in1;
        end if;
    end process;
end behavioral;
```

Ο 5-bit Πολυπλέκτης 2 σε 1 χρησιμοποιείται σε πολλαπλά σημεία του MIPS για να επιτυγχάνεται η σωστή διαδρομή για κάθε εντολή. Ως εισόδους έχει 2 αριθμούς των 5 bit και καλείται να διαλέξει έναν απ' τους δύο. Η επιλογή γίνεται με βάση κάποιο από τα σήματα της κεντρικής μονάδας ελέγχου *Control*.

Στο testbench δοκιμάζουμε 2 ζευγάρια αριθμών με διαφορετικές τιμές σήματος επιλογής για να επιβεβαιώσουμε ότι θα κάνει την σωστή επιλογή.

Κώδικας 5Mux2to1 tb:

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

entity Mux5_2to1_tb is
end Mux5_2to1_tb;

architecture behavior of Mux5_2to1_tb is
    signal in0: std_logic_vector(4 downto 0);
    signal in1: std_logic_vector(4 downto 0);
    signal set: std_logic;
    signal output: std_logic_vector(4 downto 0);

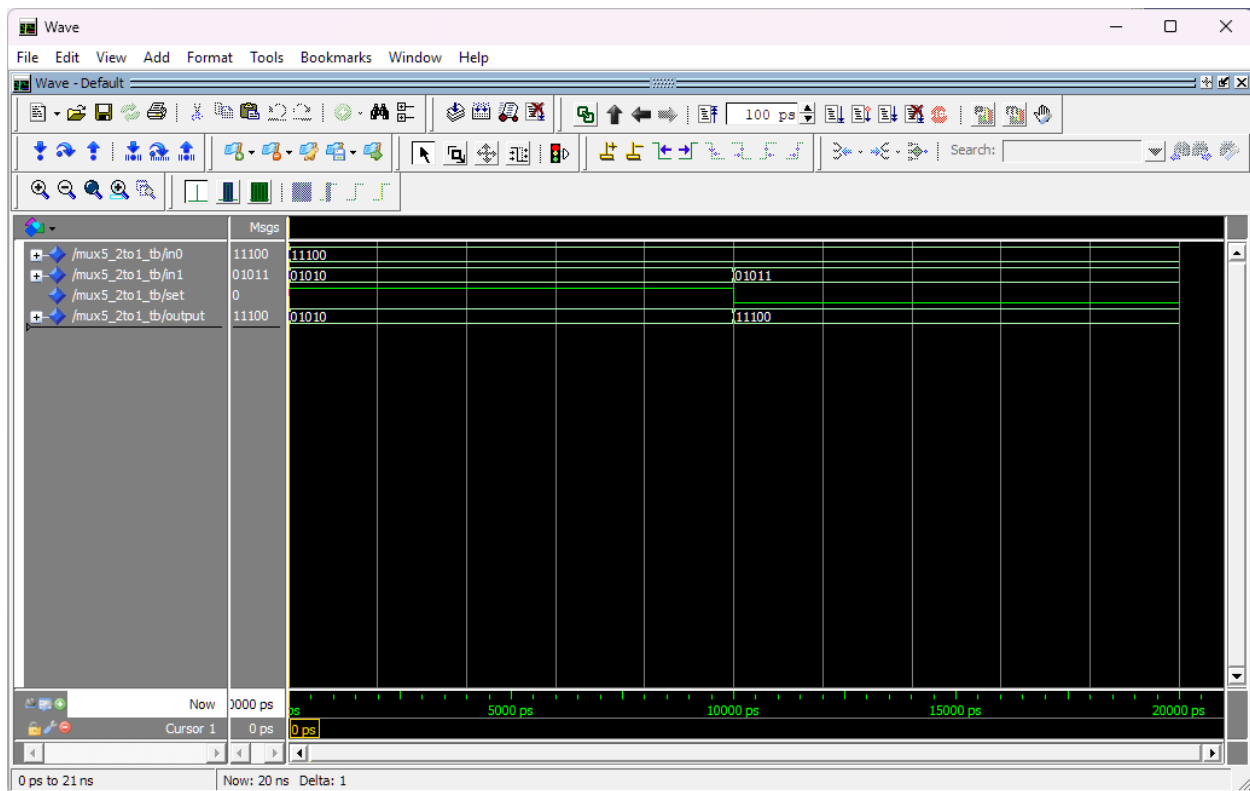
begin
    uut: entity work.Mux5_2to1 port map(
        in0 => in0,
        in1 => in1,
        set => set,
        output => output);

    process
    begin

        in0 <= "11100"; --0x1C
        in1 <= "01010"; --0x0A
        set <= '1';
        wait for 10 ns;

        in0 <= "11100";
        in1 <= "01011";
        set <= '0';
        wait for 10 ns;

        wait;
    end process;
end behavior;
```



1.1.9. Sign Extension Unit

Κώδικας Signextension:

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

entity Signextension is port(
in16: in std_logic_vector(15 downto 0);
out32: out std_logic_vector(31 downto 0));
end Signextension;

architecture behavioral of Signextension is
begin
    process(in16)
        variable tmp: signed(31 downto 0);
    begin
        tmp:= resize(signed(in16),32);
        out32 <= std_logic_vector(tmp);
    end process;
end behavioral;
```

Κώδικας Signextension tb:

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

entity Signextension_tb is
end Signextension_tb;

architecture behavior of Signextension_tb is
    signal in16: std_logic_vector(15 downto 0);
    signal out32: std_logic_vector(31 downto 0);

begin
    uut: entity work.Signextension port map(
        in16 => in16,
        out32 => out32);

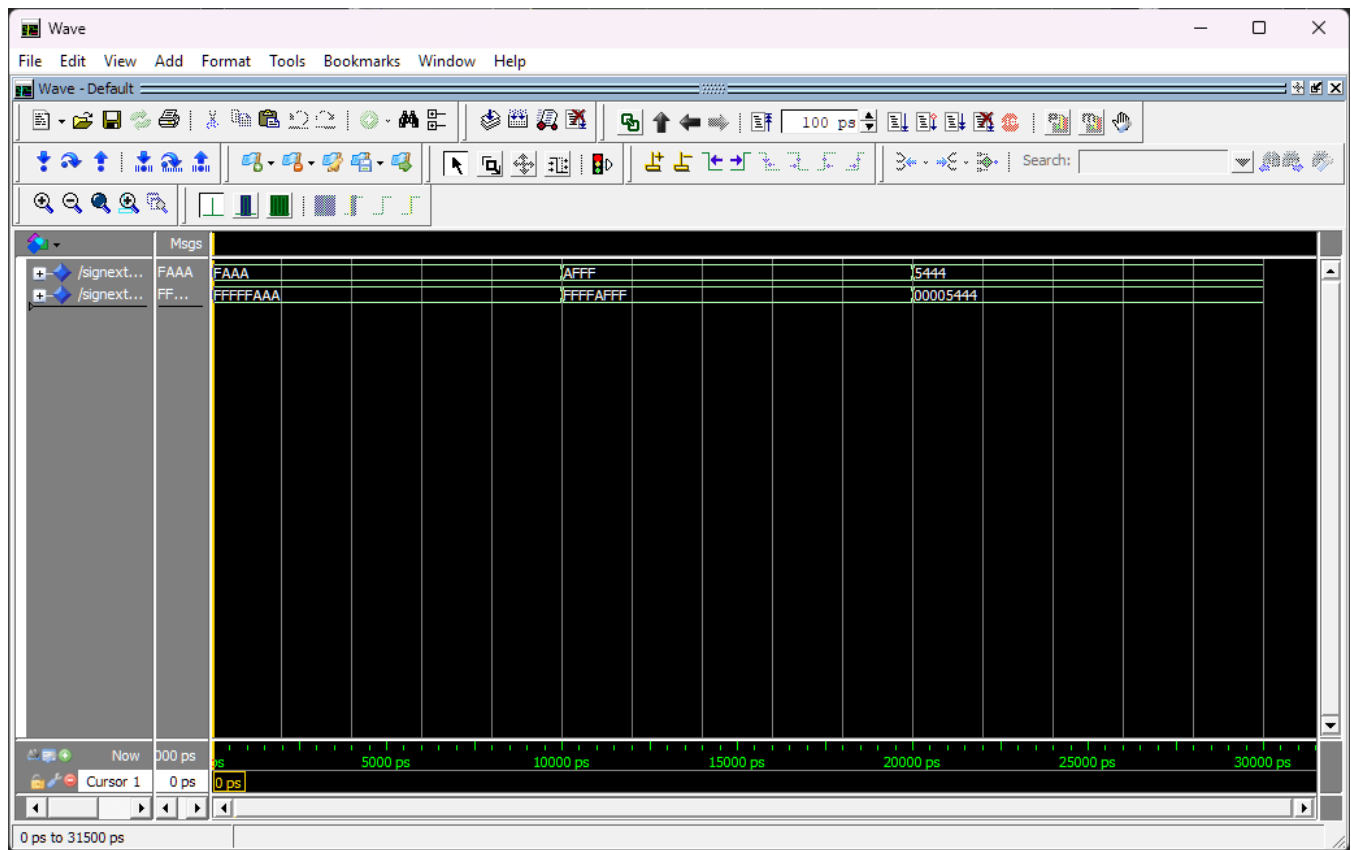
    process
    begin
        in16 <= x"FAAA";
        wait for 10 ns;

        in16 <= x"FFFF";
        wait for 10 ns;

        in16<= x"5444";
        wait for 10 ns;

        wait;
    end process;
end behavior;
```

Η λειτουργία της μονάδας επέκτασης προσήμου είναι αρκετά απλή. Έχει ως είσοδο μια τιμή 16 bits και καλείται να την επεκτείνει στα 32 bits.



1.1.10. 32MUX2to1

Κώδικας 32Mux2to1:

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

entity Mux32_2to1 is port(
in0: in std_logic_vector(31 downto 0);
in1: in std_logic_vector(31 downto 0);
set: in std_logic;
output: out std_logic_vector(31 downto 0));
end Mux32_2to1;

architecture behavioral of Mux32_2to1 is
begin
    process(in0, in1, set)
    begin
        if set = '0' then
            output <= in0;
        else output <= in1;
        end if;
    end process;
end behavioral;
```

Ο 32bit Πολυπλέκτης 2 σε 1 έχει ακριβώς την ίδια λειτουργία με τον Πολυπλέκτη 5 bit παραπάνω, με την μόνη διαφορά ότι ως εισόδους έχει δύο αριθμούς των 32 bits αντί των 5 bits.

Κώδικας 32Mux2to1 tb:

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

entity Mux32_2to1_tb is
end Mux32_2to1_tb;

architecture behavior of Mux32_2to1_tb is
    signal in0: std_logic_vector(31 downto 0);
    signal in1: std_logic_vector(31 downto 0);
    signal set: std_logic;
    signal output: std_logic_vector(31 downto 0);

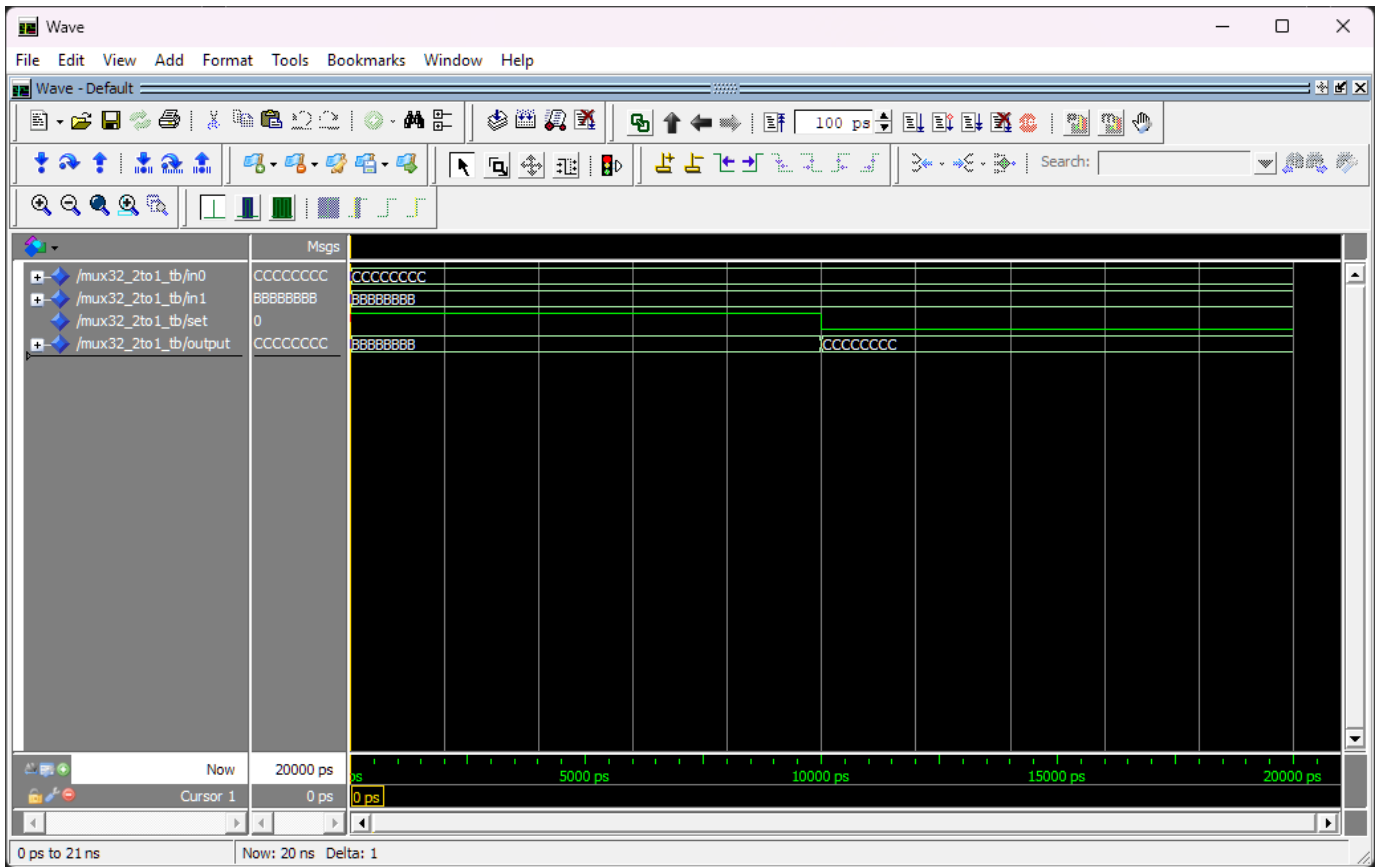
begin
    uut: entity work.Mux32_2to1 port map(
        in0 => in0,
        in1 => in1,
        set => set,
        output => output);

    process
    begin

        in0 <= x"CCCCCCCC";
        in1 <= x"BBBBBBBB";
        set <= '1';
        wait for 10 ns;

        in0 <= x"CCCCCCCC";
        in1 <= x"BBBBBBBB";
        set <= '0';
        wait for 10 ns;

        wait;
    end process;
end behavior;
```



1.1.11. Shift Left 2

Κώδικας Leftshift:

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

entity Leftshift is port(
input: in std_logic_vector(31 downto 0);
output: out std_logic_vector(31 downto 0));
end Leftshift;

architecture behavioral of Leftshift is
begin
    output <= input(29 downto 0) & "00";
end behavioral;
```

Κώδικας Leftshift tb:

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

entity Leftshift_tb is
end Leftshift_tb;

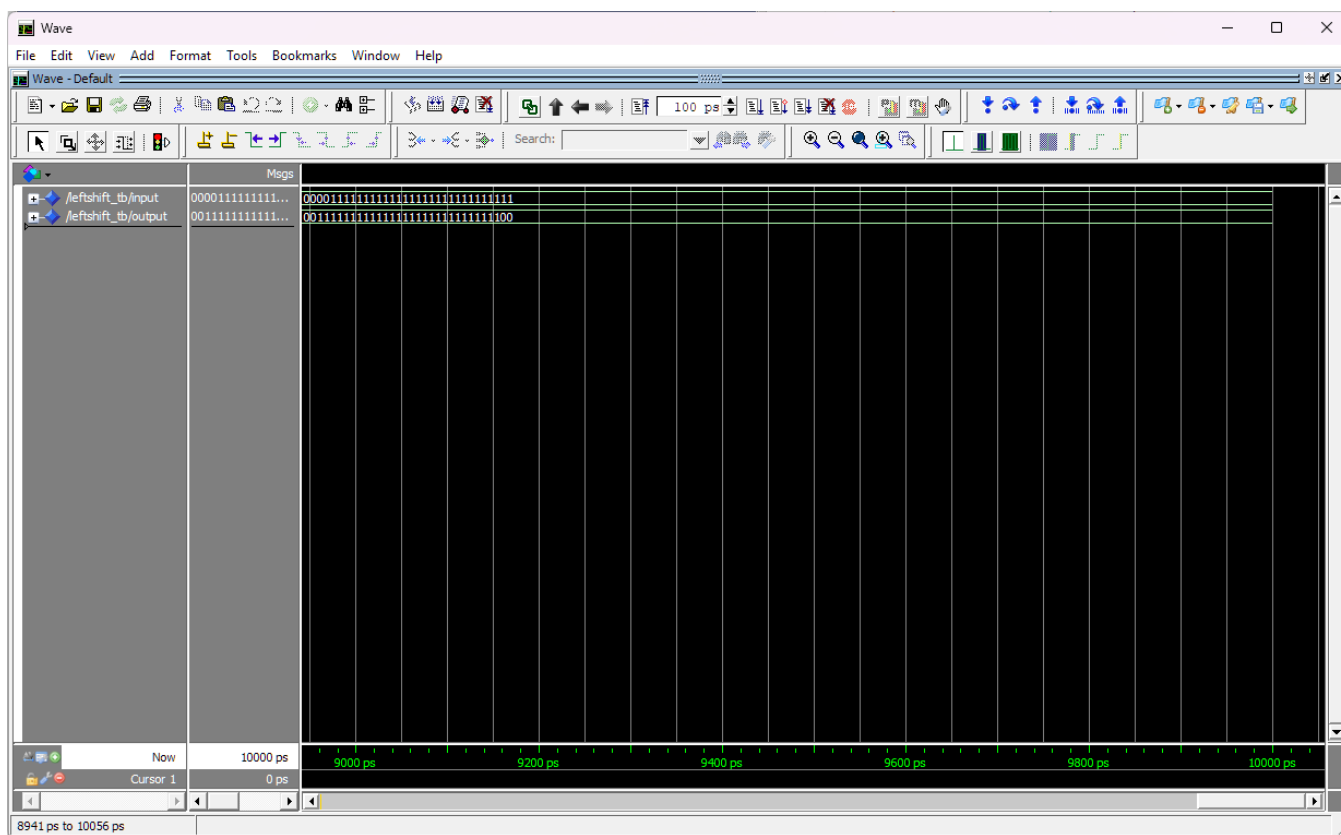
architecture behavior of Leftshift_tb is
    signal input: std_logic_vector(31 downto 0);
    signal output: std_logic_vector(31 downto 0);

begin
    uut: entity work.Leftshift port map(
        input => input,
        output => output);

    process
    begin
        input <= x"0FFFFFFF";
        wait for 10 ns;

        wait;
    end process;
end behavior;
```

Η μονάδα αριστερής ολίσθησης κατά 2 είναι επίσης μία αρκετά απλή μονάδα. Είναι υπεύθυνη, όπως δηλώνει το όνομά της, για την ολίσθηση προς τα αριστερά ενός αριθμού 32 bit. Στο testbench δοκιμάζουμε αυτή τη λειτουργία με μία τιμή.



1.1.12. 32-bit Adder

Κώδικας Adder32:

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

entity Adder32 is port(
in0: in std_logic_vector(31 downto 0);
in1: in std_logic_vector(31 downto 0);
result: out std_logic_vector(31 downto 0));
end Adder32;

architecture behavioral of Adder32 is
begin
    result <= std_logic_vector(unsigned(in0)+unsigned(in1));
end behavioral;
```

Κώδικας Adder32 tb:

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

entity Adder32_tb is
end Adder32_tb;

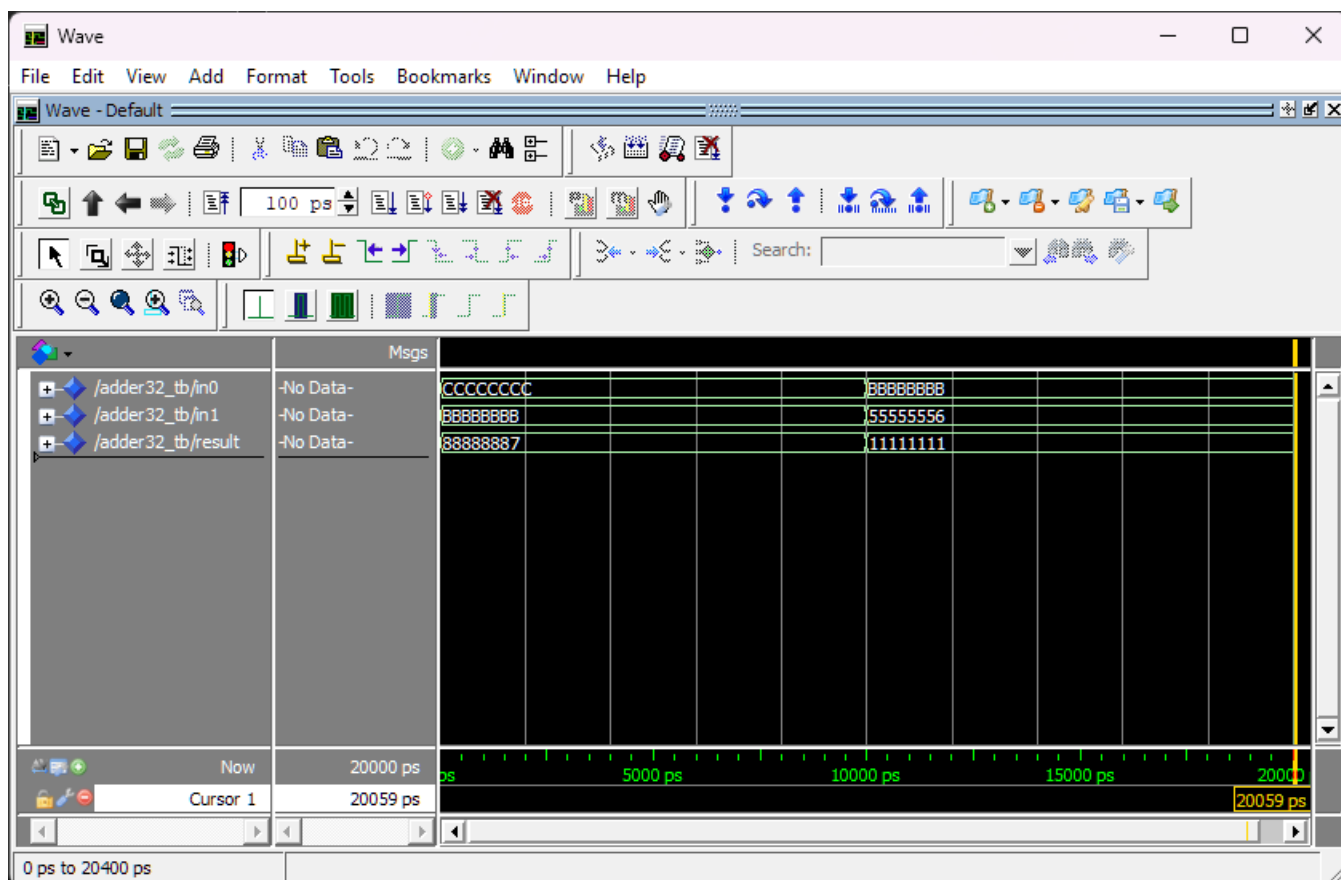
architecture behavior of Adder32_tb is
    signal in0, in1, result: std_logic_vector(31 downto 0);
begin
    uut: entity work.Adder32 port map(
        in0 => in0,
        in1 => in0,
        result => result);

    process
    begin
        in0 <= x"CCCCCCCC";
        in1 <= x"BBBBBBBB";
        wait for 10 ns;

        in0 <= x"BBBBBBBB";
        in1 <= x"55555556";
        wait for 10 ns;

        wait;
    end process;
end behavior;
```

Ο αθροιστής αναλαμβάνει την πρόσθεση δύο αριθμών, που αποτελούν και τις εισόδους του, και το άθροισμα τους αποτελεί την μοναδική του έξοδο. Για τους σκοπούς της εργασίας δεν λαμβάνεται υπόψιν η υπερχείλιση.



Μέρος Τρίτο: MIPS

Κώδικας MIPS:

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

entity MIPS is port(
    clock: in std_logic;
    reset: in std_logic);
end MIPS;

architecture behavioral of MIPS is
    component ALU port(
        ALUin1: in std_logic_vector(31 downto 0);
        ALUin2: in std_logic_vector(31 downto 0);
        ALUctrl: in std_logic_vector(3 downto 0);
        ALUresult: out std_logic_vector(31 downto 0);
        zero: out std_logic);
    end component;

    component Registerfile port(
        clk: in std_logic;
        reset: in std_logic;
        read_reg1: in std_logic_vector(4 downto 0);
        read_reg2: in std_logic_vector(4 downto 0);
        write_reg: in std_logic_vector(4 downto 0);
        write_data: in std_logic_vector(31 downto 0);
        write_enable: in std_logic;
        read_data1: out std_logic_vector(31 downto 0);
        read_data2: out std_logic_vector(31 downto 0));
    end component;

    component Datamem port(
        clk: in std_logic;
        mem_write: in std_logic;
        mem_read: in std_logic;
        address: in std_logic_vector(31 downto 0);
        write_data: in std_logic_vector(31 downto 0);
        read_data: out std_logic_vector(31 downto 0));
    end component;

    component Imem port(
        address: in std_logic_vector(31 downto 0);
        instruction: out std_logic_vector(31 downto 0));
    end component;

    component Control port(
        opcode: in std_logic_vector(5 downto 0);
        regdst: out std_logic;
        ALUsrc: out std_logic;
        memtoreg: out std_logic;
        regwrite: out std_logic;
        memread: out std_logic;
        memwrite: out std_logic;
```

```

    branch: out std_logic;
    ALUOp: out std_logic_vector(1 downto 0));
end component;

component ALUcontrol port(
    funct: in std_logic_vector(5 downto 0);
    ALUOp: in std_logic_vector(1 downto 0);
    ALUcontrol: out std_logic_vector(3 downto 0));
end component;

component PC port(
    clk: in std_logic;
    reset: in std_logic;
    pc_in: in std_logic_vector(31 downto 0);
    pc_out: out std_logic_vector(31 downto 0));
end component;

component Mux5_2to1 port(
    in0: in std_logic_vector(4 downto 0);
    in1: in std_logic_vector(4 downto 0);
    set: in std_logic;
    output: out std_logic_vector(4 downto 0));
end component;

component Signextension port(
    in16: in std_logic_vector(15 downto 0);
    out32: out std_logic_vector(31 downto 0));
end component;

component Mux32_2to1 port(
    in0: in std_logic_vector(31 downto 0);
    in1: in std_logic_vector(31 downto 0);
    set: in std_logic;
    output: out std_logic_vector(31 downto 0));
end component;

--H μονάδα Leftshift παραλείπεται

--H μονάδα Adder32 δεν προστίθεται ως component

--Control
signal RegDst: std_logic;
signal Branch: std_logic;
signal MemRead: std_logic;
signal MemToReg: std_logic;
signal ALUOp: std_logic_vector(1 downto 0);
signal MemWrite: std_logic;
signal ALUSrc: std_logic;
signal RegWrite: std_logic;
--PC
signal PCOut: std_logic_vector(31 downto 0);
--ALU
signal Zero: std_logic;
signal ALUout: std_logic_vector(31 downto 0);
--Register File
signal RegOut1: std_logic_vector(31 downto 0);
signal RegOut2: std_logic_vector(31 downto 0);

```

```

--MUX
signal MUXtoReg: std_logic_vector(4 downto 0);
signal MUXtoALU: std_logic_vector(31 downto 0);
signal MUXtoPC: std_logic_vector(31 downto 0);
signal MUXtoRegWrite: std_logic_vector(31 downto 0);
signal Branch_isTrue: std_logic;
--ALU Control
signal ALUctrl_out: std_logic_vector(3 downto 0);
--Sign Extend
signal SignExOut: std_logic_vector(31 downto 0);
--Data Mem
signal DataMemOut: std_logic_vector(31 downto 0);
--Instruction Mem
signal IMOut: std_logic_vector(31 downto 0);
--Adder32
signal ONE: std_logic_vector(31 downto 0);
signal AddressNoBranch: std_logic_vector(31 downto 0);
signal AddressBranch: std_logic_vector(31 downto 0);

begin
    ONE <= std_logic_vector(to_unsigned(1,32));
    FA_PC: entity work.Adder32 port map(
        in0 => PCOut,
        in1 => ONE,
        result => AddressNoBranch);

    FA_Branch: entity work.Adder32 port map(
        in0 => AddressNoBranch,
        in1 => SignExOut,
        result => AddressBranch);

    Prog_Counter: PC port map(
        pc_in => MUXtoPC,
        pc_out => PCOut,
        clk => clock,
        reset => reset);

    Instr_Mem: Imem port map(
        address => PCOut,
        instruction => IMOut);

    RegFile: Registerfile port map(
        read_reg1 => IMOut(25 downto 21),
        read_reg2 => IMOut(20 downto 16),
        write_reg => MUXtoReg,
        write_data => MUXtoRegWrite,
        write_enable => RegWrite,
        read_data1 => RegOut1,
        read_data2 => RegOut2,
        clk => clock,
        reset => reset);

    Ctrl: Control port map(
        opcode => IMOut(31 downto 26),
        regdst => RegDst,
        ALUsrc => ALUSrc,
        memtoreg => MemToReg,

```

```

    regwrite => RegWrite,
    memread => MemRead,
    memwrite => MemWrite,
    branch => Branch,
    ALUop => ALUOp);

ALUctrl: ALUcontrol port map(
    funct => IMOut(5 downto 0),
    ALUop => ALUOp,
    ALUcontrol => ALUctrl_out);

ALU1: ALU port map(
    ALUin1 => RegOut1,
    ALUin2 => MUXtoALU,
    ALUctrl => ALUctrl_out,
    ALUresult => ALUout,
    zero => Zero);

Data_Mem: Datamem port map(
    mem_write => MemWrite,
    mem_read => MemRead,
    address => ALUout,
    write_data => RegOut2,
    read_data => DataMemOut,
    clk => clock);

MUXreg: Mux5_2to1 port map(
    in0 => IMOut(20 downto 16),
    in1 => IMOut(15 downto 11),
    set => RegDst,
    output => MUXtoReg);

MUXalu_in: Mux32_2to1 port map(
    in0 => RegOut2,
    in1 => SignExOut,
    set => ALUSrc,
    output => MUXtoALU);

MUXdatamem: Mux32_2to1 port map(
    in1 => DataMemOut,
    in0 => ALUout,
    set => MemToReg,
    output => MUXtoRegWrite);

Branch_isTrue <= Branch AND (not Zero);

MUXaddress: Mux32_2to1 port map(
    in0 => AddressNoBranch,
    in1 => AddressBranch,
    set => Branch_isTrue,
    output => MUXtoPC);

Sign_Ext: Signextension port map(
    in16 => IMOut(15 downto 0),
    out32 => SignExOut);

end behavioral;

```


Κώδικας MIPS tb:

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

entity MIPS_tb is
end MIPS_tb;

architecture behavior of MIPS_tb is
    signal clock: std_logic := '0';
    signal reset: std_logic := '0';

    --200MHz => 5 ns
    constant clk_period: time := 5 ns;

begin
    uut: entity work.MIPS port map(
        clock => clock,
        reset => reset);

    clock <= not clock after clk_period/2;

    process
    begin
        reset <= '1';
        wait for 3 ns;

        reset <= '0';
        wait for 100 ns;
        std.env.stop;
        wait;
    end process;
end behavior;
```

Ο κώδικας του MIPS αποτελείται από τα 2 σήματα που περιλαμβάνει (clock & reset) και την διασύνδεση με όλα τα εξαρτήματα που τον αποτελούν. Περιλαμβάνονται και πολλά signals τα οποία δρουν ως τα καλώδια που συνδέουν τις διάφορες εξόδους των εξαρτημάτων με τις εισόδους άλλων εξαρτημάτων.

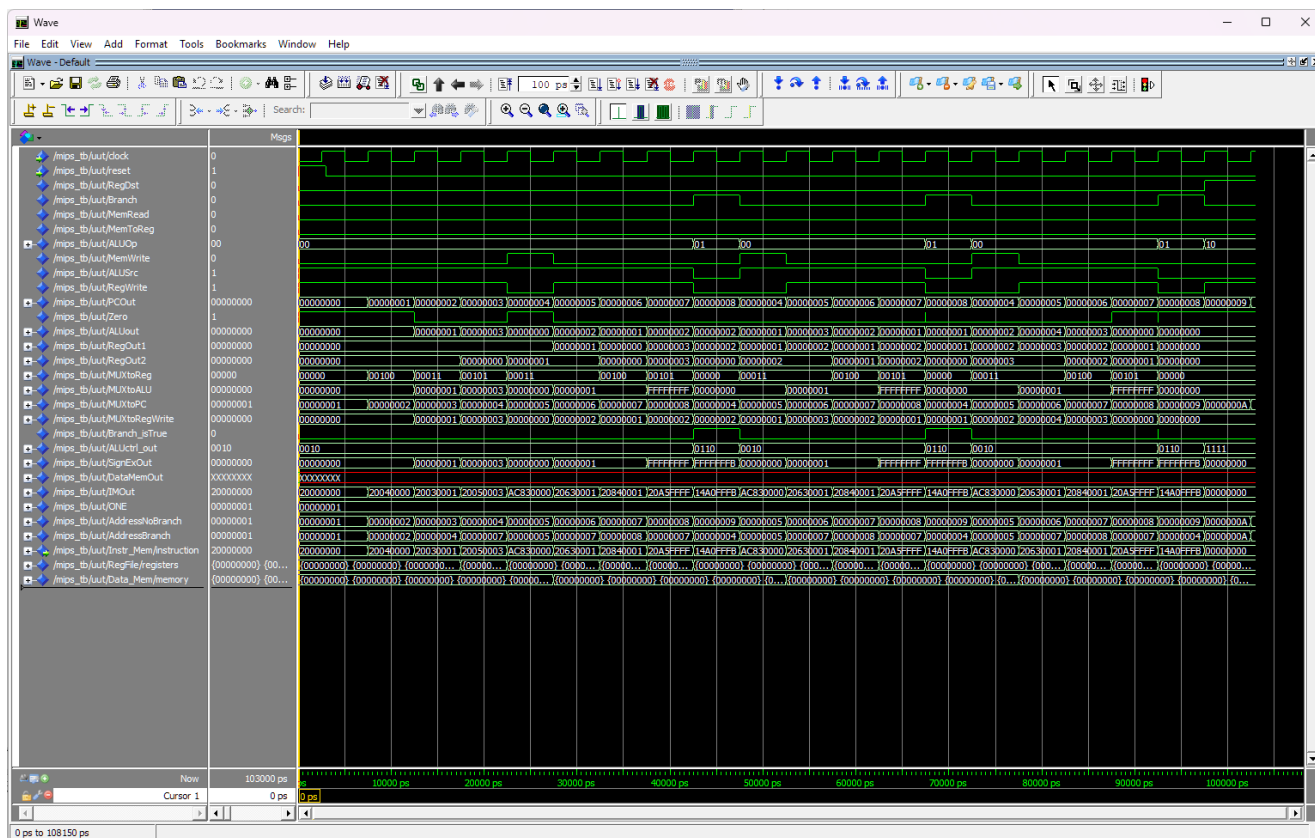
Η μονάδα αριστερής ολίσθησης έχει παραλειφθεί για ευκολία, σύμφωνα με τις οδηγίες της άσκησης. Οι αθροιστές δεν έχουν δηλωθεί στα components αλλά εισάγονται απευθείας ως entity γιατί αλλιώς είχαμε “No default binding for component instance” error.

Έχοντας τις εντολές hardcoded στην μνήμη εντολών, το testbench είναι αρκετά απλό και μικρό. Το ρολόι του συστήματος υπολογίστηκε στα 5 nanoseconds, με δεδομένο από την άσκηση 200 MHz συχνότητα:

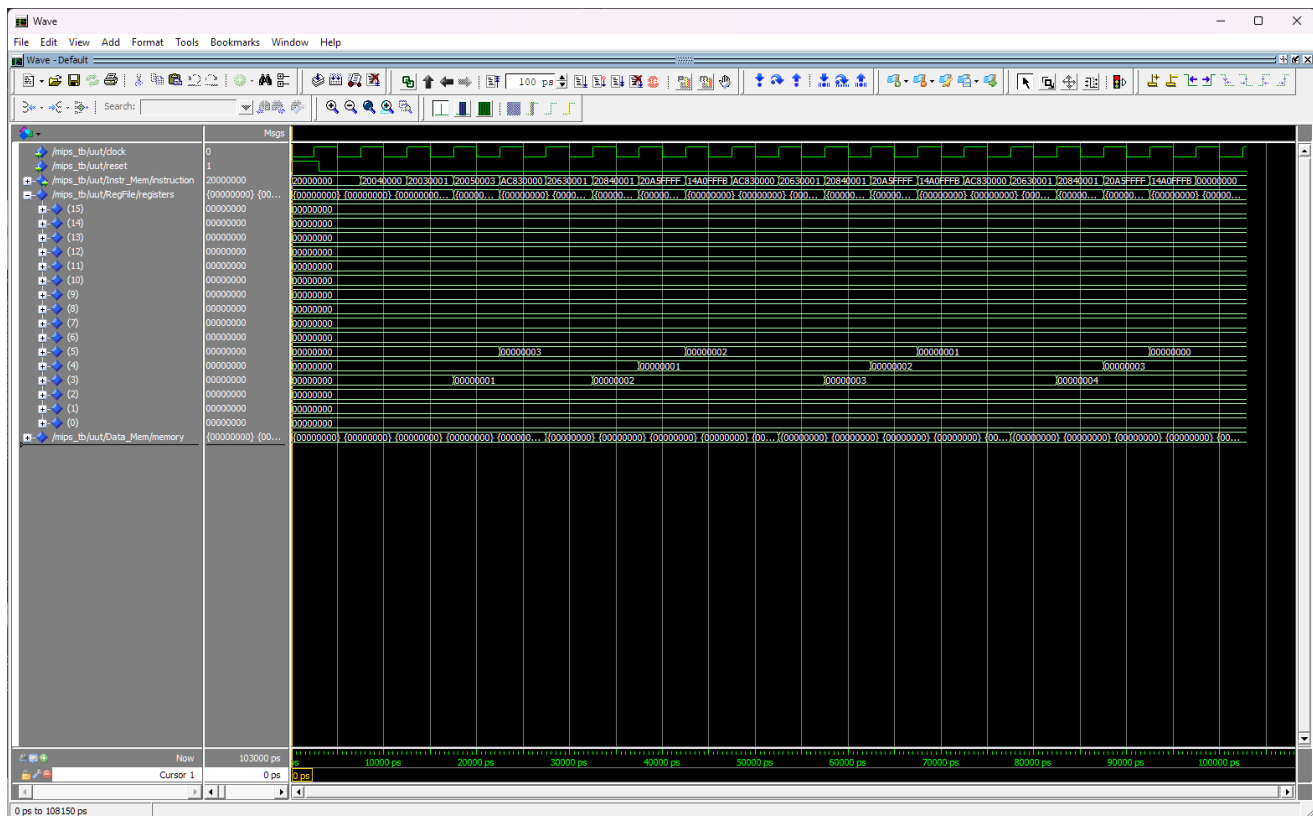
$$\text{Περίοδος} = 1 / 200 * 10^6 = 5 * 10^{(-9)} = 5 \text{ ns}$$

Αξίζει να σχολιαστεί ότι το σήμα επιλογής `Branch_isTrue` του MUX που επιλέγει ανάμεσα στην διεύθυνση της αμέσως επόμενης εντολής ή στην διεύθυνση της εντολής για jump, είναι αποτέλεσμα πράξης **AND** με το **αντεστραμμένο** σήμα `Zero` της ALU διότι εκτελούμε μόνο εντολή `bne`. Αυτό σημαίνει πως αν δεν αντιστρέψαμε το σήμα, το jump δεν θα συνέβαινε ποτέ. Αυτό γιατί η ALU δεν θα άναβε το σήμα `Zero` διότι όταν θα ελέγχονταν τα περιεχόμενα των καταχωρητών, δεν θα ήταν ίσα και άρα η διαφορά τους δεν θα ήταν μηδέν.

Τα αποτελέσματα παρουσιάζονται στις εικόνες που ακολουθούν. Όπως βλέπουμε από τον παλμό του ρολογιού, η προσομοίωση έτρεξε για 20 κύκλους.



Εικόνα 3.1. Τα βασικά σήματα του MIPS



Εικόνα 3.2. Τα περιεχόμενα των καταχωρητών. Από τις τιμές του καταχωρητή \$5, ο οποίος αποτελεί τον μετρητή της επανάληψης, καταλαβαίνουμε ότι ορθά μειώνεται από το 3 στο 0 και συνεπώς η επανάληψη τρέχει 3 φορές.

Μπορούμε επίσης να επιβεβαιώσουμε την ορθότητα της εκτέλεσης των εντολών και από τις δεκαεξαδικές τιμές που βλέπουμε στην γραμμή της Μνήμης Εντολών.

```

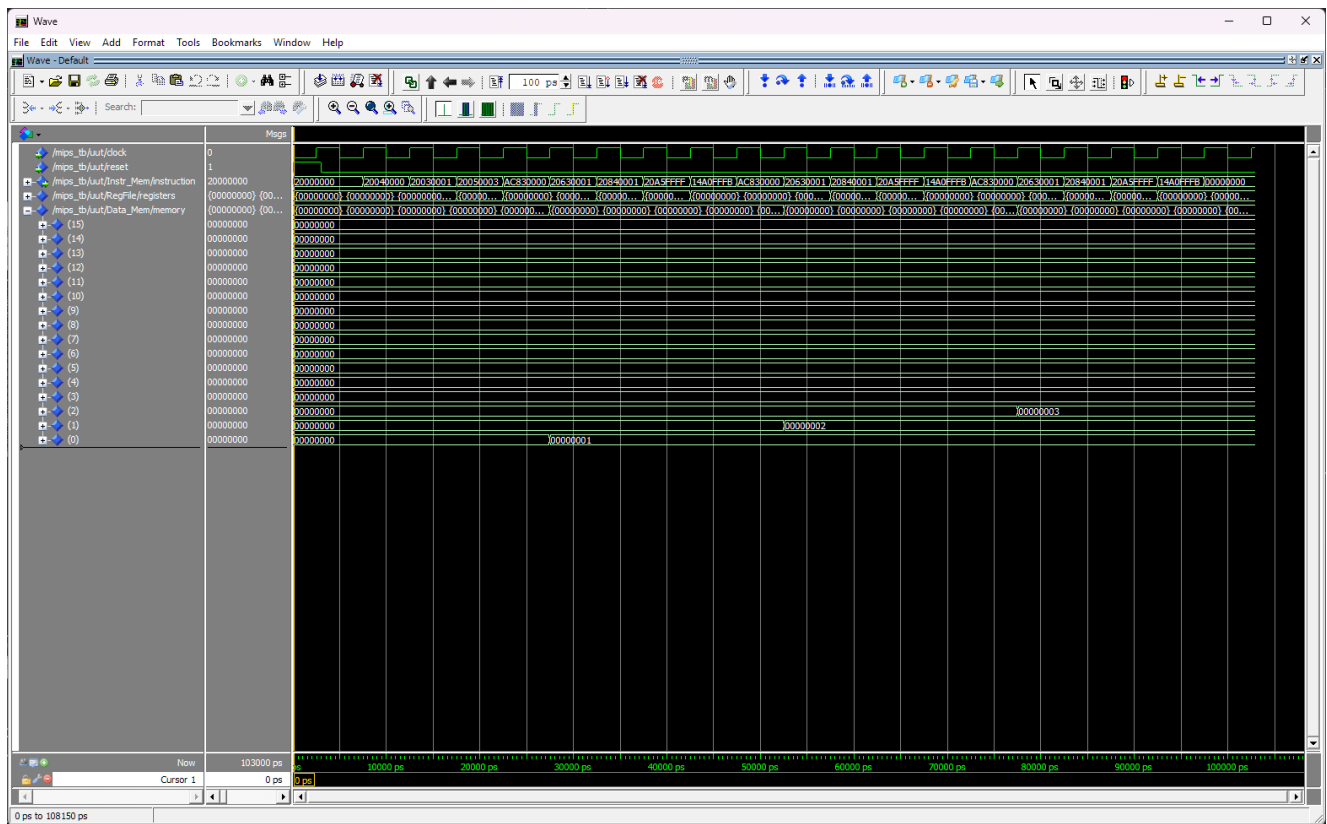
0 => x"20000000", --addi $0, $0, 0
1 => x"20040000", --addi $4, $0, 0
2 => x"20030001", --addi $3, $0, 1
3 => x"20050003", --addi $5, $0, 3
4 => x"AC830000", --sw $3, 0($4)
5 => x"20630001", --addi $3, $3, 1
6 => x"20840001", --addi $4, $4, 1
7 => x"20A5FFFF", --addi $5, $5, -1
8 => x"14A0FFFB", --bne $5, $0, L1

```

```

addi $0, $0, 0 # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
addi $4, $0, 0 # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
addi $3, $0, 1
addi $5, $0, 3
L1:
    sw $3, 0($4)
    addi $3, $3, 1
    addi $4, $4, 1
    addi $5, $5, -1
bne $5,$0,L1

```



Εικόνα 3.3. Τα περιεχόμενα της μνήμης δεδομένων. Εδώ βλέπουμε πως η εντολή

`sw $3, 0($4)` έχει εκτελεστεί με επιτυχία για τις τιμές των `$3` και `$4` που αλλάζουν σε κάθε επανάληψη.