Πανεπιστήμιο Δυτικής Αττικής

Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών

Ακαδημαϊκό έτος 2023-2024

Σχεδίαση Ψηφιακών Συστημάτων

Εξαμηνιαία Εργασία Μαθήματος

ΖΟΥΜΑΣ ΗΛΙΑΣ

20390068

Σκοπός της εργασίας

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.

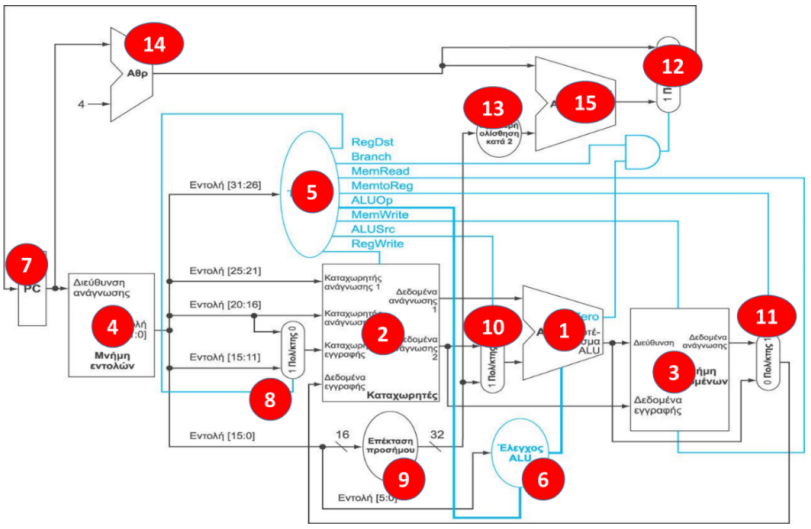
Ο επεξεργαστής θα εκτελεί τις εντολές: `add`, `sub`, `addi`, `lw`, `sw`, `bne`.

Για ευκολία στην υλοποίηση θεωρείται ότι:

- Η μνήμη εντολών έχει 16 θέσεις των 32 bits
- Η μνήμη δεδομένων έχει 16 θέσεις των 32 bits
- Το αρχείο καταχωρητών έχει 16 θέσεις των 32 bits
- Μετά την εκτέλεση κάθε εντολής το περιεχόμενο του PC αυξάνει κατά 1

Ο επεξεργαστής περιλαμβάνει τις ακόλουθες μονάδες:

A/A	Μονάδα
1	ALU ⇒ Αριθμητική και Λογική Μονάδα
2	Register File ⇒ Αρχείο Καταχωρητών
3	Data Memory (RAM) ⇒ Μνήμη δεδομένων
4	Instruction Memory (ROM) ⇒ Μνήμη εντολών
5	Control Unit ⇒ Μονάδα ελέγχου
6	ALU Control Unit ⇒ Μονάδα ελέγχου ALU
7	Program Counter (PC) ⇒ Μετρητής Προγράμματος
8	5-bit 2:1 MUX ⇒ 5-πλό πολυπλέκτη 2-σε-1
9	Sign Extension Unit ⇒ Μονάδα επέκτασης προσήμου 16-σε-32
10, 11, 12	32-bit 2:1 MUX ⇒ 32-πλό πολυπλέκτη 2-σε-1
13	32-bit Shifter ⇒ Μονάδα ολίσθησης αριστερά κατά 2 (32-bit)
14, 15	32-bit Adder ⇒ Αθροιστή 32 bits



Εικόνα 1: Υλοποίηση του Επεξεραστή - Απαρίθμηση Μονάδων

Υλοποίηση των Μονάδων - Components

Από την Εικόνα 1 παρατηρούμε πως ο επεξεργαστής MIPS απαρτίζεται από μονάδες (components) που συνδέονται μεταξύ τους με αλληλουχία για την εκτέλεση εντολών. Για να προχωρήσουμε στην υλοποιήση του επεξεργαστή θα αναπτύξουμε μεμονωμένες οντότητες (entities) οι οποίες θα περιγράφουν ποιοτικά κάθε μονάδα. Αυτή η προσέγγιση μας επιτρέπει να ελέγξουμε την ορθή λειτουργία των μονάδων ώστε να είναι σωστές και απαναχρησιμοποιήσιμες. Ο έλεγχος των μονάδων θα γίνει με την χρήση κώδικα Test Bench και των σημάτων κυματοφορφής (waves) από προσομοιώσεις (simulations).

Για την υλοποίηση χρησιμοποιήθηκαν τα εργαλεία:

- GHDL
- GTKWave
- ModelSim

Η σειρά ανάπτυξης των μονάδων θα έχει αύξουσα σειρά δυσκολίας.
Θα ξεκινήσουμε από τα απλότερα κυκλώματα όπως ο πολυπλέκτης και ο αθροιστής για να παρουσιάσουμε λεπτομέρως την διαδικασία ανάπτυξης και έλεγχου και στην συνέχεια θα συνεχίσουμε με τα πιο σύνθετα κυκλώματα όπως οι μονάδες μνήμης δεδομένων και εντολών, κλπ.

Πολυπλέκτης - Multiplexer (MUX)

Περιγραφή και Χρήση του Κυκλώματος

Το κύκλωμα του Πολυπλέκτη (Multiplexer - MUX) επιτρέπει την επιλογή ενός σήματος εισόδου και την προώθηση του στην έξοδο. Χρησιμοποιείται συχνά ως δρομολογητής (Data Routing), για το διαμοιρασμό δεδομένων (Data Sharing) και για αλλαγή σημάτων ως μοχλός (Signal Switching).

Ακολουθεί η υλοποίηση ενός απλού 1-bit 2:1 MUX, όπου όταν το bit επιλογής (selection bit -sel) έχει τιμή '0', έξοδος είναι η τιμή της πρώτης εισόδου (i0). Αλλιώς όταν έχει τιμή '1' έξοδος είναι η τιμή της δεύτερης εξόδου (i1). Το συγκεκριμένο κύκλωμα δεν θα χρησιμοποιήθει στον επεξεργαστή αλλά η μόνη διαφορά που έχει με τα κυκλώματα του 5-πλού και 32-πλού πολυπλέκτη είναι στο πλήθος των bits που χρησιμοποιούνται.

Η υλοποίηση του 5-πλού και 32-πλού πολυπλέκτη θα χρησιμοποιεί `std_logic_vector` των 5 και 32 bits αντίστοιχα.

Το συγκεκριμένο κύκλωμα θα μας βοηθήσει στην παρουσίαση του έλεγχου ενός component με την προσέγγιση των *Table Driven Tests*, όπου ορίζεται σε μία εγγραφή (record) οι είσοδοι και η εκτιμώμενη έξοδος ενός κυκλώματος.
Έπειτα μέσω πίνακα (array) μπορούμε να ελέγξουμε μια σειρά από περιπτώσεις, επιτρέποντας μας τον εξαντλητικό έλεγχο της συμπεριφοράς ενός κυκλώματος και την εύκολη προσθήκη περισσοτέρων περιπτώσεων ελέγχου.

1-bit 2:1 MUX

Entity & Architecture

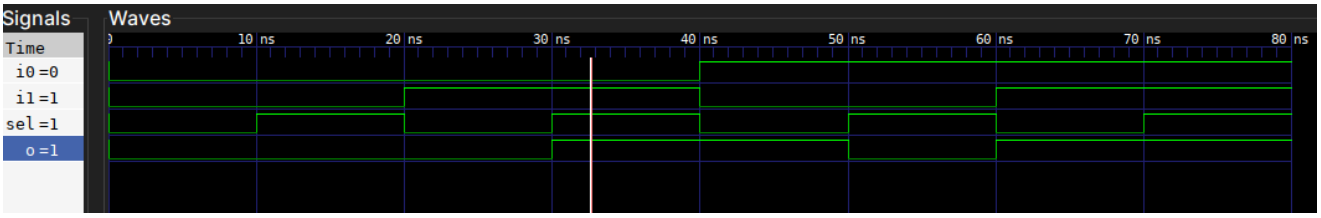
```
-- Mux2to1.vhd
-- Implementation of 1-bit 2:1 MUX

library ieee;
use ieee.std_logic_1164.all;

-- Define the 1-bit 2:1 MUX entity
entity mux2_1 is
-- i0, i1, sel are inputs while o is output all of type std_logic.
port (
  i0, i1: in std_logic;
  o: out std_logic;
  sel: in std_logic
);
end entity mux2_1;

-- A behavioral architecture describes how a component should behave
-- without concern about time and implementation restrictions.
architecture behavioral of mux2_1 is
begin
  o <= i0 when sel = '0' else i1;
end architecture behavioral;
```

Test Bench

```
-- Mux2to1_tb.vhd
-- Test Bench of 1-bit 2:1 MUX

library ieee;
use ieee.std_logic_1164.all;

-- A test bench entity is usually empty
entity mux2_1_tb is end entity mux2_1_tb;

architecture behavioral of mux2_1_tb is
  -- Component declaration for the 1-bit 2:1 MUX
  component mux2_1
  port (
    i0, i1: in std_logic;
    o: out std_logic;
    sel: in std_logic
  );
  end component;

  -- Signal declarations to connect to the MUX
  signal i0, i1, sel, o: std_logic;

begin
  -- Instantiate the MUX commonly labeled as uut - Unit Under Test
  uut: mux2_1 port map (
    i0 => i0,
    i1 => i1,
    o => o,
    sel => sel
  );

  -- Stimulus process: Apply values to inputs and observe behavior
  stim_proc: process
    -- Record that states the input values and the expected output
    type pattern_type is record
      -- The inputs of the MUX
      i0, i1, sel: std_logic;
      -- The expected output of the MUX
      o: std_logic;
    end record;

    -- The pattern array that allows us to easily add more cases, or test
exhaustively.
    type pattern_array is array (natural range <>) of pattern_type;

    -- The list of test cases - the patterns to apply
    constant patterns: pattern_array := (
      ('0', '0', '0', '0'),
      ('0', '0', '1', '0'),
      ('0', '1', '0', '0'),
      ('0', '1', '1', '1'),
```

```
        ('1', '0', '0', '1'),
        ('1', '0', '1', '0'),
        ('1', '1', '0', '1'),
        ('1', '1', '1', '1')
      );
    begin
      -- Check each pattern
      for i in patterns'range loop

        -- Set the inputs
        i0 <= patterns(i).i0;
        i1 <= patterns(i).i1;
        sel <= patterns(i).sel;

        -- Wait for the results
        wait for 10 ns;

        -- Check the results
        assert o = patterns(i).o
        report "test failed for pattern " & integer'image(i)
        severity error;

      end loop;

      -- finish the simulation
      report "end of simulation"
      severity note;
      wait;

    end process;

  end architecture behavioral;
```

Wave



Εικόνα 2: Κυματομορφή του 1-bit MUX 2:1

5-bit MUX 2:1

Entity & Architecture

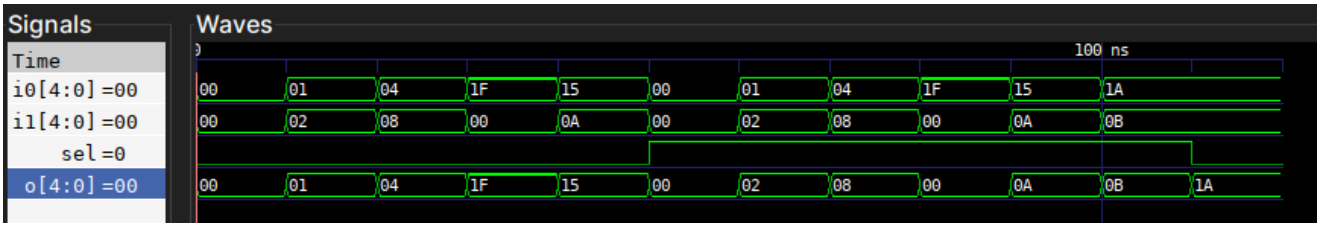
```
-- 5Mux2to1.vhd
-- Implementation of 5-bit MUX 2:1

library ieee;
use ieee.std_logic_1164.all;

entity mux2_5 is
port (
  i0, i1: in std_logic_vector(4 downto 0);
  o: out std_logic_vector(4 downto 0);
  sel: in std_logic
);
end entity mux2_5;

architecture behavioral of mux2_5 is
begin
  o <= i0 when sel = '0' else i1;
end architecture behavioral;
```

Test Bench

```
-- Mux2to1_5_tb.vhd
-- Test Bench of 5-bit 2:1 MUX

library ieee;
use ieee.std_logic_1164.all;

entity mux2_5_tb is end entity mux2_5_tb;

architecture behavioral of mux2_5_tb is
  component mux2_5
  port (
    i0, i1: in std_logic_vector(4 downto 0);
    o: out std_logic_vector(4 downto 0);
    sel: in std_logic
  );
  end component;

  signal i0, i1, o: std_logic_vector(4 downto 0);
  signal sel: std_logic;

begin
  uut: mux2_5 port map (
    i0 => i0,
    i1 => i1,
    o => o,
    sel => sel
  );

  stim_proc: process
    type pattern_type is record
      i0, i1: std_logic_vector(4 downto 0);
      sel: std_logic;
      o: std_logic_vector(4 downto 0);
    end record;

    type pattern_array is array (natural range <>) of pattern_type;
    constant patterns: pattern_array := (
      -- sel = '0', output should be i0
      ("00000", "00000", '0', "00000"),
      ("00001", "00010", '0', "00001"),
      ("00100", "01000", '0', "00100"),
      ("11111", "00000", '0', "11111"),
      ("10101", "01010", '0', "10101"),
      -- sel = '1', output should be i1
      ("00000", "00000", '1', "00000"),
      ("00001", "00010", '1', "00010"),
      ("00100", "01000", '1', "01000"),
      ("11111", "00000", '1', "00000"),
      ("10101", "01010", '1', "01010"),

      -- Required test cases
```

```
        ("11010", "01011", '1', "01011"), -- sel = '0', output should be i0
        ("11010", "01011", '0', "11010")  -- sel = '1', output should be i1
      );
    begin
      for i in patterns'range loop

        i0 <= patterns(i).i0;
        i1 <= patterns(i).i1;
        sel <= patterns(i).sel;

        wait for 10 ns;

        assert o = patterns(i).o
        report "test failed for pattern " & integer'image(i)
        severity error;

      end loop;

      report "end of simulation"
      severity note;
      wait;

    end process;

  end architecture behavioral;
```

Wave



Εικόνα 3: Κυματομορφή του 5-bit MUX 2:1

32-bit MUX 2:1

Entity & Architecture

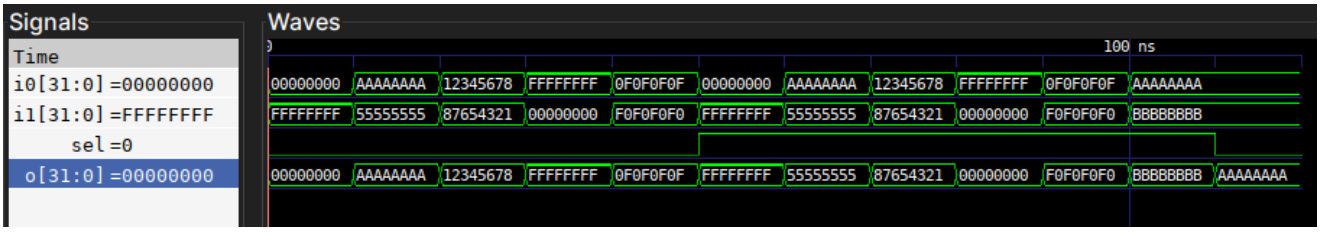
```
-- 32Mux2to1.vhd
-- Implementation of 32-bit MUX 2:1

library ieee;
use ieee.std_logic_1164.all;

entity mux2_32 is
port (
  i0, i1: in std_logic_vector(31 downto 0);
  o: out std_logic_vector(31 downto 0);
  sel: in std_logic
);
end entity mux2_32;

architecture behavioral of mux2_32 is
begin
  o <= i0 when sel = '0' else i1;
end architecture behavioral;
```

Test Bench

```
-- 32Mux2to1_tb.vhd
-- Test Bench for 32-bit 2:1 MUX

library ieee;
use ieee.std_logic_1164.all;

entity mux2_32_tb is end entity mux2_32_tb;

architecture behavioral of mux2_32_tb is
  component mux2_32
  port (
    i0, i1: in std_logic_vector(31 downto 0);
    o: out std_logic_vector(31 downto 0);
    sel: in std_logic
  );
  end component;

  signal i0, i1, o: std_logic_vector(31 downto 0);
  signal sel: std_logic;

begin
  uut: mux2_32 port map (
    i0 => i0,
    i1 => i1,
    o => o,
    sel => sel
  );

  stim_proc: process
    type pattern_type is record
      i0, i1: std_logic_vector(31 downto 0);
      sel: std_logic;
      o: std_logic_vector(31 downto 0);
    end record;

    type pattern_array is array (natural range <>) of pattern_type;
    constant patterns: pattern_array := (
      -- sel = '0', output should be i0
      (x"00000000", x"FFFFFFFF", '0', x"00000000"),
      (x"AAAAAAAA", x"55555555", '0', x"AAAAAAAA"),
      (x"12345678", x"87654321", '0', x"12345678"),
      (x"FFFFFFFF", x"00000000", '0', x"FFFFFFFF"),
      (x"0F0F0F0F", x"F0F0F0F0", '0', x"0F0F0F0F"),
      -- sel = '1', output should be i1
      (x"00000000", x"FFFFFFFF", '1', x"FFFFFFFF"),
      (x"AAAAAAAA", x"55555555", '1', x"55555555"),
      (x"12345678", x"87654321", '1', x"87654321"),
      (x"FFFFFFFF", x"00000000", '1', x"00000000"),
      (x"0F0F0F0F", x"F0F0F0F0", '1', x"F0F0F0F0"),

      -- Required test cases
```

```
      (x"AAAAAAAA", x"BBBBBBBB", '1', x"BBBBBBBB"), -- sel = '1', output should
be i1
      (x"AAAAAAAA", x"BBBBBBBB", '0', x"AAAAAAAA")  -- sel = '0', output should
be i0
    );

  begin
    for i in patterns'range loop

      i0 <= patterns(i).i0;
      i1 <= patterns(i).i1;
      sel <= patterns(i).sel;

      wait for 10 ns;

      assert o = patterns(i).o
      report "test failed for pattern " & integer'image(i)
      severity error;

    end loop;

    report "end of simulation"
    severity note;
    wait;

  end process;

end architecture behavioral;
```

Wave



Εικόνα 4: Κυματομορφή του 5-bit MUX 2:1

Πλήρης Αθροιστής - Full Adder

Περιγραφή και Χρήση του Κυκλώματος

Το κύκλωμα του Αθροιστή (Full Adder) είναι ένα από τα πιο κρίσιμα και βασικά κυκλώματα στην Ψηφιακή Σχεδίαση. Χρησιμοποιείται στην Αριθμητική και Λογική Μονάδα (ALU) (για την υλοποίηση των τεσσάρων βασικών πράξεων τις πρόσθεσης, αφαίρεσης, πολλαπλασιασμού και διαίρεσης. Εντολές `add`, `addi`, `sub`), αλλά και για τον υπολογισμό διευθύνσεων. *Effective Address Calculation* για εντολές τύπου load/store όπως `lw` / `sw`. *Branch Address Calculation* για εντολές διακλάδωσης όπως `beq`, `bne`.

Entity & Architecture

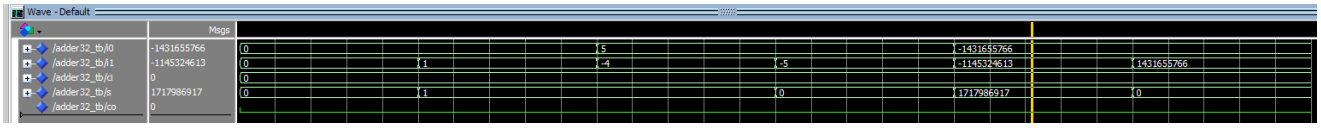
```
-- Adder32.vhd
-- Implementation of 32-bit Full Adder

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder32 is
port (
  i0, i1: in std_logic_vector(31 downto 0);
  ci: in std_logic_vector(0 downto 0);
  s: out std_logic_vector(31 downto 0);
  co: out std_logic
);
end entity adder32;

architecture behavioral of adder32 is

  signal ts: std_logic_vector(32 downto 0);

begin

    ts <= std_logic_vector(
      to_signed(
        to_integer(signed(i0)) + to_integer(signed(i1)) +
to_integer(signed(ci)),
        33
      )
    );
    s <= ts(31 downto 0);
    co <= ts(32);

  end architecture behavioral;
```

Test Bench

```
-- Adder32_tb.vhd
-- Test bench for 32-bit Full Adder

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder32_tb is end entity adder32_tb;

architecture behavioral of adder32_tb is
  component adder32
    port (
      i0, i1: in std_logic_vector(31 downto 0);
      ci: in std_logic_vector(0 downto 0);
      s: out std_logic_vector(31 downto 0);
      co: out std_logic
    );
  end component;

  signal i0, i1: std_logic_vector(31 downto 0);
  signal ci: std_logic_vector(0 downto 0);
  signal s: std_logic_vector(31 downto 0);
  signal co: std_logic;

begin
  uut: adder32 port map (
    i0 => i0,
    i1 => i1,
    ci => ci,
    s => s,
    co => co
  );

  stim_proc: process

    type pattern_type is record
      i0, i1: std_logic_vector(31 downto 0);
      ci: std_logic_vector(0 downto 0);
      s: std_logic_vector(31 downto 0);
      co: std_logic;
    end record;

    type pattern_array is array (natural range <>) of pattern_type;

    constant patterns: pattern_array := (
      -- Basic
      (x"00000000", x"00000000", "0", x"00000000", '0'),
      (x"00000000", x"00000001", "0", x"00000001", '0'),

      -- ALU test cases
```

```
        (x"00000005", x"FFFFFFFC", "0", x"00000001", '0'),
        (x"00000005", x"FFFFFFFB", "0", x"00000000", '0'),

        -- Required test cases
        (x"AAAAAAAA", x"BBBBBBBB", "0", x"66666665", '1'),
        (x"AAAAAAAA", x"55555556", "0", x"FFFFFFFC", '1')
    );

  begin

    i0 <= (others => '0');
    i1 <= (others => '0');
    ci <= "0";


    for i in patterns'range loop

      i0 <= patterns(i).i0;
      i1 <= patterns(i).i1;
      ci <= patterns(i).ci;

      wait for 10 ns;

      assert s = patterns(i).s
        report "unexpected sum value for index" & integer'image(i)
        severity error;

      assert co = patterns(i).co
      report "unexpected carry-out value for index " & integer'image(i)
        severity error;

    end loop;

    -- finish the simulation
    report "end of simulation"
    severity note;
    wait;

  end process;

end architecture behavioral;
```

Wave



Εικόνα 4: Κυματομορφή του 32-bit Full Adder

Μονάδας Ολίσθησης Αριστερά κατά 2

Περιγραφή και Χρήση του Κυκλώματος

Η μονάδα Ολίσθησης Αριστερά κατά 2 παραλείπεται στην συγκεκριμένη υλοποίηση

Entity & Architecture

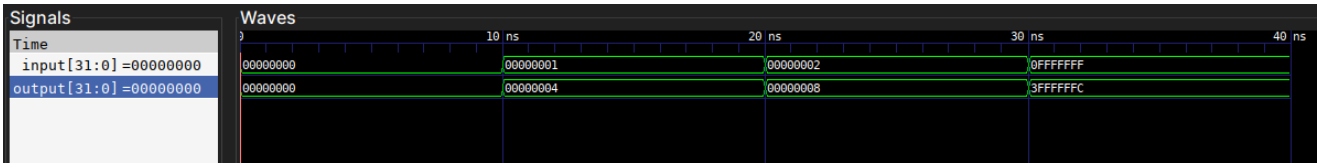
```
-- LeftShift.vhd
-- Implementation of 32-bit Left Shifter by 2

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lshifter32 is
port (
  input: in std_logic_vector(31 downto 0);
  output: out std_logic_vector(31 downto 0)
);
end entity lshifter32;

architecture behavioral of lshifter32 is
  signal tmp: unsigned(31 downto 0);
begin
  tmp <= to_unsigned(to_integer(signed(input)), tmp'length) sll 2;
  output <= std_logic_vector(to_signed(to_integer(tmp), output'length));
end architecture behavioral;
```

Test Bench

```
-- LeftShift_tb.vhd
-- Test bench of 32-bit Left Shifter by 2

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lshifter32_tb is end entity lshifter32_tb;

architecture behavioral of lshifter32_tb is
  component lshifter32
  port (
    input: in std_logic_vector(31 downto 0);
    output: out std_logic_vector(31 downto 0)
  );
  end component;

  signal input : std_logic_vector(31 downto 0);
  signal output : std_logic_vector(31 downto 0);

  type pattern_type is record
    input : std_logic_vector(31 downto 0);
    output : std_logic_vector(31 downto 0);
  end record;

  type pattern_array is array (natural range <>) of pattern_type;

  constant patterns : pattern_array := (
    (input => x"00000000", output => x"00000000"), -- 0 << 2 = 0
    (input => x"00000001", output => x"00000004"), -- 1 << 2 = 4
    (input => x"00000002", output => x"00000008"), -- 2 << 2 = 8

    -- Required test case
    (input => x"0FFFFFFF", output => x"3FFFFFFC")

  );

begin
  uut: lshifter32 port map (
    input => input,
    output => output
  );

  stim_proc: process
  begin
    for i in patterns'range loop

      input <= patterns(i).input;

      wait for 10 ns;
```

```
      -- Check the results
      assert output = patterns(i).output
        report "test failed for pattern " & integer'image(i)
        severity error;
    end loop;

    -- Finish the simulation
    report "end of simulation"
    severity note;
    wait;

  end process;

end architecture behavioral;
```

Wave



Εικόνα 5: Κυματομορφή του 32-bit Left Shifter

Μονάδα Επέκτασης Προσήμου - Sign Extension Unit

Περιγραφή και Χρήση του Κυκλώματος

Η μονάδα επέκτασης προσήμου (Sign Extension Unit) σε έναν επεξεραστή MIPS είναι υπεύθυνη για τη διατήρηση του προσήμου ενός αριθμού κατά την επέκταση του από μικρότερο σε μεγαλύτερο μήκος. Συγκεκριμένα, όταν ένας αριθμός επεκτείνεται από 16-bit σε 32-bit, η μονάδα επέκτασης προσήμου διασφαλίζει ότι το προσήμο του αρχικού αριθμού διατηρείται σωστά.

Για παράδειγμα, ας υποθέσουμε ότι έχουμε έναν 16-bit αριθμό στην αναπαράσταση δυαδικού συμπληρώματος 2 (two's complement):

```
Original 16-bit number: 1010 1010 1010 1010 (σε δυαδική μορφή)
```

Αν τον επεκτείνουμε σε έναν 32-bit αριθμό, η μονάδα επέκτασης προσήμου θα προσθέσει τα απαιτούμενα bits για να διατηρήσει το προσήμο:

```
Extended 32-bit number: 1111 1111 1010 1010 1010 1010 1010 1010 (σε δυαδική μορφή)
```

Εδώ, τα πρόσθετα bits (στο συγκεκριμένο παράδειγμα, το 16ο bit και τα επόμενα) είναι αντίγραφα του προσήμου του αρχικού αριθμού (το πρώτο bit). Αυτό εξασφαλίζει ότι η αναπαράσταση σε 32-bit διατηρεί την ίδια αριθμητική τιμή όπως και η αρχική αναπαράσταση σε μικρότερο μήκος, διατηρώντας παράλληλα το σωστό προσήμο.

Η μονάδα επέκτασης προσήμου είναι σημαντική για την εξασφάλιση της συνέπειας των αριθμητικών λειτουργιών και την αποφυγή προβλημάτων.

Entity & Architecture

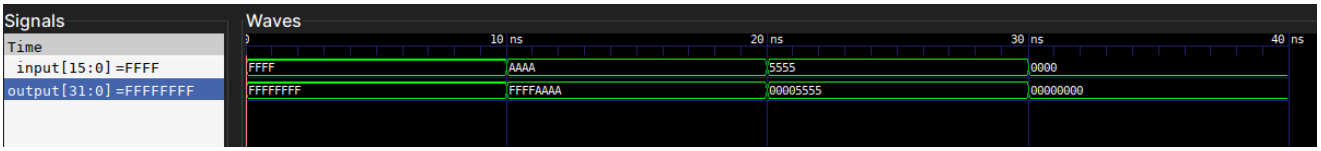
```
-- Signextension.vhd
-- Implementation of Sign Extension Unit

library ieee;
use ieee.std_logic_1164.all;

entity signext is
port (
  input: in std_logic_vector(15 downto 0);
  output: out std_logic_vector(31 downto 0)
);
end entity signext;

architecture behavioral of signext is
begin
  output <= (15 downto 0 => input(15)) & input;
end architecture behavioral;
```

Test Bench

```
-- Signextension_tb.vhd
-- Test Bench of Sign Extension Unit

library ieee;
use ieee.std_logic_1164.all;

entity signext_tb is end entity signext_tb;

architecture behavioral of signext_tb is
  component signext is
  port (
    input: in std_logic_vector(15 downto 0);
    output: out std_logic_vector(31 downto 0)
  );
  end component signext;

  signal input: std_logic_vector(15 downto 0);
  signal output: std_logic_vector(31 downto 0);

  type pattern_type is record
    input: std_logic_vector(15 downto 0);
    output: std_logic_vector(31 downto 0);
  end record;

  type pattern_array is array (natural range <>) of pattern_type;

  constant patterns: pattern_array := (
    (x"FFFF", x"FFFFFFFF"),
    (x"AAAA", x"FFFFAAAA"),
    (x"5555", x"00005555"),
    (x"0000", x"00000000")
  );

begin
 uut: signext
  port map(
     input => input,
     output => output
 );

 stim_proc: process
 begin
  for i in patterns'range loop
    input <= patterns(i).input;

    wait for 10 ns;

    assert output = patterns(i).output
    report "unexpected output value for input at index " & integer'image(i)
    severity error;
```

```
    end loop;

    assert false
      report "end of simulation"
      severity note;
    wait;

  end process stim_proc;

end architecture behavioral;
```

Wave



Εικόνα 6: Κυματομορφή του Sign Extension Unit

Μετρήτης Προγράμματος - Program Counter

Περιγραφή και Χρήση του Κυκλώματος

Ο μετρητής προγράμματος (Program Counter, PC) είναι ένας εξαιρετικά σημαντικός και βασικός καταχωρητής σε κάθε επεξεργαστή.
Η λειτουργία του PC είναι να δείχνει την διεύθυνση της επόμενης εντολής που πρέπει να εκτελεστεί στο πρόγραμμα.

Στην αρχή κάθε εκτέλεσης προγράμματος, ο PC αρχικοποιείται με τη διεύθυνση της πρώτης εντολής (συνήθως αρχή του προγράμματος).
Καθώς η CPU εκτελεί κάθε εντολή, ο PC αυξάνεται κατά το μήκος της τρέχουσας εντολής, ώστε να δείχνει στη διεύθυνση της επόμενης εντολής στη σειρά.

Οι βασικές λειτουργίες του PC στους επεξεργαστές MIPS συμπεριλαμβάνουν:

1. **Εκτέλεση Εντολών:** Ο PC δείχνει την τρέχουσα εντολή που πρέπει να εκτελεστεί. Κάθε φορά που μια εντολή ολοκληρώνεται, ο PC αυξάνεται κατά το μήκος της εντολής για να δείξει στην επόμενη.
2. **Αλλαγή Ροής Προγράμματος:** Ο PC είναι υπεύθυνος για την αλλαγή της ροής του προγράμματος, διατηρώντας την σειρά εκτέλεσης των εντολών. Όταν υπάρχει άλμα (jump) σε μια άλλη διεύθυνση, ο PC ενημερώνεται με τη νέα διεύθυνση που πρέπει να εκτελεστεί η επόμενη εντολή.
3. **Αποθήκευση και Επαναφορά Κατάστασης:** Ο PC μπορεί να αποθηκεύεται (push) και να επαναφέρεται (pop) στην στοίβα κατά την εκτέλεση κλήσεων συναρτήσεων (function calls) ή κατά την επιστροφή από αυτές, επιτρέποντας τη διατήρηση της σωστής ροής εκτέλεσης του προγράμματος.

Entity & Architecture

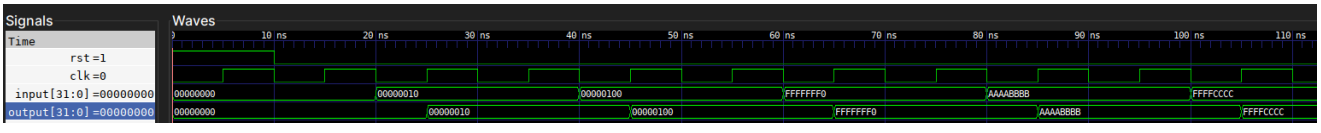
```vhdl
-- PC.vhd
-- Implementation of Program Counter

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pc is
port (
  input: in std_logic_vector(31 downto 0);
  clk, rst: in std_logic;
  output: out std_logic_vector(31 downto 0)
);
end entity pc;

architecture behavioral of pc is
  signal pc_internal: signed(31 downto 0);

begin

    process (clk, rst)
    begin
      if rst = '1' then
        pc_internal <= to_signed(0, pc_internal'length);
      elsif rising_edge(clk) then
        pc_internal <= signed(input);
      end if;
    end process;

    output <= std_logic_vector(pc_internal);

end architecture behavioral;
```

Test Bench

```
-- PC_tb.vhd
-- Test bench of Program Counter (PC)

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pc_tb is
end entity pc_tb;

architecture behavioral of pc_tb is
  component pc
    port (
      input: in std_logic_vector(31 downto 0);
      clk, rst: in std_logic;
      output: out std_logic_vector(31 downto 0)
    );
  end component pc;

  signal clk: std_logic := '0';
  signal rst: std_logic := '0';
  signal input: std_logic_vector(31 downto 0) := (others => '0');
  signal output: std_logic_vector(31 downto 0);

  constant CLOCK_PERIOD: time := 10 ns;

begin

  uut: pc port map (
    input => input,
    clk => clk,
    rst => rst,
    output => output
  );

  clk_proc: process
  begin
    while now < 200 ns loop
      clk <= '0';
      wait for CLOCK_PERIOD / 2;
      clk <= '1';
      wait for CLOCK_PERIOD / 2;
    end loop;
    wait;
  end process clk_proc;

  stim_proc: process

      type pattern_type is record
        input: std_logic_vector(31 downto 0);
```

```
        output: std_logic_vector(31 downto 0);
    end record;

    type pattern_array is array (natural range <>) of pattern_type;

    constant patterns: pattern_array := (
    (x"00000010", x"00000010"),  -- Load PC with 0x00000010
    (x"00000100", x"00000100"),  -- Load PC with 0x00000100
    (x"FFFFFFF0", x"FFFFFFF0"),  -- Load PC with 0xFFFFFFF0

    -- Required test cases
    (x"AAAA_BBBB", x"AAAA_BBBB"),
    (x"FFFF_CCCC", x"FFFF_CCCC")
    );
begin
    rst <= '1';
    wait for CLOCK_PERIOD;
    rst <= '0';
    wait for CLOCK_PERIOD;

    for i in patterns'range loop
      input <= patterns(i).input;
      wait for CLOCK_PERIOD;

      assert output = patterns(i).output
      report "test failed for pattern " & integer'image(i)
      severity error;

      wait for CLOCK_PERIOD;
    end loop;

    report "end of simulation"
    severity note;
    wait;

end process stim_proc;

end architecture behavioral;
```

Wave



Εικόνα 7: Κυματομορφή του Program Counter

Αρχείο Καταχωρητών - Register File

Περιγραφή και Χρήση του Κυκλώματος

Το αρχείο καταχωρητών αποθηκεύει προσωρινά δεδομένα που χρησιμοποιούνται από τον επεξεργαστή κατά την εκτέλεση εντολών. Απαρτίζεται από έναν αριθμό καταχωρητών (στην συγκεκριμένη περίπτωση 16), όπου ο κάθε καταχωρητής έχει συγκεκριμένο μέγεθος (στην συγκεκριμένη περίπτωση 32 bit).

Οι καταχωρητές χρησιμοποιούνται για την αποθήκευση ενδιαμέσων αποτελεσμάτων (π.χ `lw`, `sw`, `add`) και την γρήγορα πρόσβαση σε δεδομένα χωρίς να την προσπέλαση σε μονάδα δεδομένων (RAM).

Δομή του Αρχείου Καταχωρητών

- Καταχωρητές
  Πλήθος των καταχωρητών (16) συγκεκριμένου μεγέθους (32).
- Λειτουργίες Πρόσβασης
  - Ανάγνωση: Δύο καταχωρητές μπορούν να αναγνωστούν ταυτόχρονα.
  - Εγγραφή: Ένας καταχωρητής μπορεί να γραφτεί σε κάθε κύκλο ρολογιού.
- Σήματα Ελέγχου:
  - Ενεργοπόιηση της εγγραφής.
  - Διευθύνσεις καταχωρητών για ανάγνωση και εγγραφή.
  - Δεδομένα προς εγγραφή και δεδομένα ανάγνωσης.

Entity & Architecture

```
-- Registerfile.vhd
-- Implementation of Register File
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity register_file is
port (
  RegIn1, RegIn2, RegToWrite: in std_logic_vector(4 downto 0);
  RegWrite: in std_logic;
  DataToWrite: in std_logic_vector(31 downto 0);
  RegOut1, RegOut2: out std_logic_vector(31 downto 0)
);
end entity register_file;

architecture behavioral of register_file is
  type registers is array (0 to 15) of std_logic_vector(31 downto 0);

  signal rs: registers := (others => (others => '0'));
  signal x: std_logic_vector(31 downto 0) := (others => 'X');
begin

  process(RegIn1, RegIn2, RegWrite, RegToWrite, DataToWrite)
  begin
    if RegWrite = '1' and DataToWrite /= x then
      rs(to_integer(unsigned(RegToWrite))) <= DataToWrite;
    else
      RegOut1 <= rs(to_integer(unsigned(RegIn1)));
      RegOut2 <= rs(to_integer(unsigned(RegIn2)));
    end if;

  end process;

end architecture behavioral;
```

Test Bench

```
-- Registerfile_tb.vhd
-- Test Bench for Register File
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity register_file_tb is end entity register_file_tb;

architecture behavioral of register_file_tb is

  component register_file is
  port (
    RegIn1, RegIn2, RegToWrite: in std_logic_vector(4 downto 0);
    RegWrite: in std_logic;
    DataToWrite: in std_logic_vector(31 downto 0);
    RegOut1, RegOut2: out std_logic_vector(31 downto 0)
  );
  end component;

  signal RegIn1, RegIn2, RegToWrite: std_logic_vector(4 downto 0) := (others =>
'0');
  signal DataToWrite: std_logic_vector(31 downto 0) := (others => '0');
  signal RegWrite: std_logic;
  signal RegOut1, RegOut2: std_logic_vector(31 downto 0);

begin

  uut: register_file
    port map (
      RegIn1 => RegIn1,
      RegIn2 => RegIn2,
      RegToWrite => RegToWrite,
      DataToWrite => DataToWrite,
      RegWrite => RegWrite,
      RegOut1 => RegOut1,
      RegOut2 => RegOut2
    );

  stim_proc: process

    type pattern_type is record
      DataToWrite: std_logic_vector(31 downto 0);
      RegToWrite: std_logic_vector(4 downto 0);
      RegWrite: std_logic;
      RegIn1, RegIn2: std_logic_vector(4 downto 0);
      expected_RegOut1, expected_RegOut2: std_logic_vector(31 downto 0);
    end record;

    type pattern_array is array (natural range <>) of pattern_type;
```

```
    constant patterns: pattern_array := (
      -- Required cases

      -- Write operations
      (
        DataToWrite => x"00000005",
        RegToWrite => "00011",
        RegWrite => '1',
        RegIn1 => "00000",
        RegIn2 => "00000",
        expected_RegOut1 => (others => '0'),
        expected_RegOut2 => (others => '0')
      ),
      (
        DataToWrite => x"00000007",
        RegToWrite => "00100",
        RegWrite => '1',
        RegIn1 => "00000",
        RegIn2 => "00000",
        expected_RegOut1 => (others => '0'),
        expected_RegOut2 => (others => '0')
      ),
      (
        DataToWrite => x"00000009",
        RegToWrite => "00101",
        RegWrite => '1',
        RegIn1 => "00000",
        RegIn2 => "00000",
        expected_RegOut1 => (others => '0'),
        expected_RegOut2 => (others => '0')
      ),

      -- Read operations
      (
        DataToWrite => (others => '0'),
        RegToWrite => (others => '0'),
        RegWrite => '0',
        RegIn1 => "00011",
        RegIn2 => "00100",
        expected_RegOut1 => x"00000005",
        expected_RegOut2 => x"00000007"
      )
    );

  begin
    for i in patterns'range loop

      DataToWrite <= patterns(i).DataToWrite;
      RegToWrite <= patterns(i).RegToWrite;
      RegWrite <= patterns(i).RegWrite;
      RegIn1 <= patterns(i).RegIn1;
      RegIn2 <= patterns(i).RegIn2;
```

```
        wait for 2 ns;

        assert RegOut1 = patterns(i).expected_RegOut1
        report "test failed for pattern " & integer'image(i) & " (RegOut1)"
        severity error;

        assert RegOut2 = patterns(i).expected_RegOut2
        report "test failed for pattern " & integer'image(i) & " (RegOut2)"
        severity error;
      end loop;

      report "end of simulation"
      severity note;
      wait;
    end process stim_proc;

end architecture behavioral;
```

Wave

Εικόνα 8: Κυματομορφή του Register File

Μνήμη Δεδομένων - Data Memory (RAM)

Περιγραφή και Χρήση του Κυκλώματος

Η μνήμη δεδομένων RAM (Random Access Memory) έχει καθοριστικό ρόλο στη αποθήκευση και ανάκτηση δεδομένων.
Σε έναν επεξεραστή MIPS η χρήση της RAM γίνεται μέσω των εντολών `lw` (load word) για ανάκτηση και `sw` (store word) για αποθήκευση.

Entity & Architecture

```vhdl
-- Datamem.vhd
-- Implementation of Data Memory (RAM)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_mem is
port (
  clk, rst: in std_logic;
  input, WriteData: in std_logic_vector(31 downto 0);
  MemRead, MemWrite: in std_logic;
  output: out std_logic_vector(31 downto 0)
);
end entity data_mem;

architecture behavioral of data_mem is

  type ram_type is array (natural range <>) of std_logic_vector(31 downto 0);
  signal ram: ram_type(0 to 15) := (others => (others => '0'));

begin

  main: process(clk)
    variable addr: integer;
  begin

    if rst = '1' then
      -- clear RAM on reset
      ram <= (others => (others => '0'));
      output <= (others => '0');
    elsif rising_edge(clk) then
      addr := to_integer(unsigned(input));

      if MemWrite = '1' and addr <= 15 then
        ram(addr) <= WriteData;
      end if;

      if MemRead = '1' and addr <= 15 then
        output <= ram(addr);
      else
        output <= (others => '0');
      end if;
    end if;
  end process;

end architecture behavioral;
```

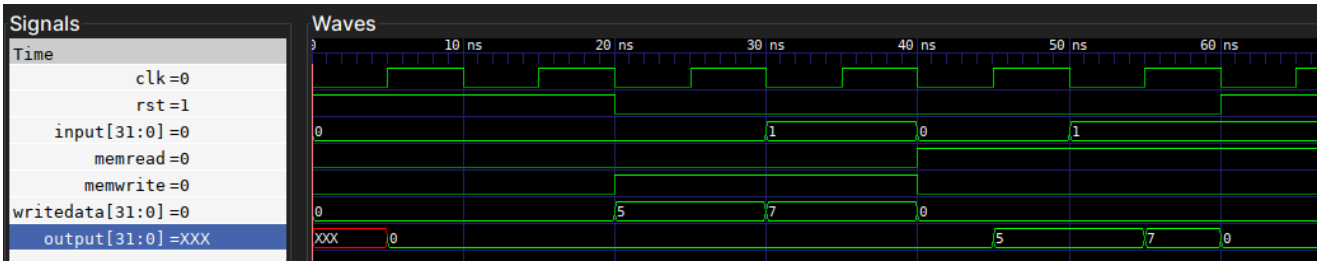
Test Bench

```
-- Datamem_tb.vhd
-- Test bench for Data Memory (RAM)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_mem_tb is end entity data_mem_tb;

architecture behavioral of data_mem_tb is

  component data_mem
  port (
    clk, rst: in std_logic;
    input, WriteData: in std_logic_vector(31 downto 0);
    MemRead, MemWrite: in std_logic;
    output: out std_logic_vector(31 downto 0)
  );
  end component;

  signal clk, rst: std_logic := '0';
  signal input, WriteData: std_logic_vector(31 downto 0) := (others => '0');
  signal MemRead, MemWrite: std_logic := '0';
  signal output: std_logic_vector(31 downto 0);

  constant clk_period: time := 10 ns;

  type pattern_type is record
    input: std_logic_vector(31 downto 0);
    WriteData: std_logic_vector(31 downto 0);
    MemRead, MemWrite: std_logic;
    expected_output: std_logic_vector(31 downto 0);
  end record;

  type pattern_array is array (natural range <>) of pattern_type;

  constant patterns: pattern_array := (
    -- Write operations
    (
      input => x"00000000",
      WriteData => x"00000005",
      MemRead => '0',
      MemWrite => '1',
      expected_output => (others => '0')
    ),
    (
      input => x"00000001",
      WriteData => x"00000007",
      MemRead => '0',
      MemWrite => '1',
      expected_output => (others => '0')
```

```
    ),

    -- Read operations
    (
      input => x"00000000",
      WriteData => (others => '0'),
      MemRead => '1',
      MemWrite => '0',
      expected_output => x"00000005"
    ),
    (
      input => x"00000001",
      WriteData => (others => '0'),
      MemRead => '1',
      MemWrite => '0',
      expected_output => x"00000007"
    )
  );

begin

  uut: data_mem
  port map (
      clk => clk,
      rst => rst,
      input => input,
      WriteData => WriteData,
      MemRead => MemRead,
      MemWrite => MemWrite,
      output => output
  );

  clk_proc: process
  begin
    while now < 1000 ns loop
      clk <= '0';
      wait for clk_period / 2;
      clk <= '1';
      wait for clk_period / 2;
    end loop;
    wait;
  end process;

  stim_proc: process
  begin
    rst <= '1';
    wait for 20 ns;
    rst <= '0';

    for i in patterns'range loop
      input <= patterns(i).input;
      WriteData <= patterns(i).WriteData;
      MemRead <= patterns(i).MemRead;
```

```
        MemWrite <= patterns(i).MemWrite;

        wait for clk_period;

        assert output = patterns(i).expected_output
            report "test failed for pattern " & integer'image(i)
            severity error;
    end loop;

    -- observe clearing of RAM
    rst <= '1';
    wait for 20 ns;

    report "end of simulation"
    severity note;
    wait;

  end process;

end architecture behavioral;
```

Wave



Εικόνα 9: Κυματομορφή της Data Memory

Μνήμη Εντολών - Instruction Memory (ROM)

Η μνήμη εντολών - instruction memory (Read Only Memory) είναι μια ειδική περίπτωση μνήμης όπου επιτρέπει μόνο την λειτουργία της ανάγνωση καθώς αποθηκεύει τις εντολές ενός προγράμματος.

Δίνεται το ακόλουθο πρόγραμμα σε MIPS Assembly για τον έλεγχο της Μνήμης Εντολών.

```
addi $0, $0, 0  # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
addi $2, $2, 0  # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
addi $2, $4, 0  # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
addi $3, $0, 1
addi $5, $0, 3
L1: add $6, $3, $0
    sw $6, 0($4)
    addi $3, $3, 1
    addi $4, $4, 1
    addi $5, $5, -1
bne $5,$0,L1
```

Κώδικας

Με την χρήση του προσομιώτη MARS, παίρνουμε το Instruction Code της κάθε εντολής

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20000000	addi $0,$0,0x00000000	1: addi $0, $0, 0 # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
	0x00400004	0x20420000	addi $2,$2,0x00000000	2: addi $2, $2, 0 # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
	0x00400008	0x20820000	addi $2,$4,0x00000000	3: addi $2, $4, 0 # δε χρειάζεται-οι καταχωρητές έχουν μηδενιστεί
	0x0040000c	0x20030001	addi $3,$0,0x00000001	4: addi $3, $0, 1
	0x00400010	0x20050003	addi $5,$0,0x00000003	5: addi $5, $0, 3
	0x00400014	0x00603020	add $6,$3,$0	6: L1: add $6, $3, $0
	0x00400018	0xac860000	sw $6,0x00000000($4)	7: sw $6, 0($4)
	0x0040001c	0x20630001	addi $3,$3,0x00000001	8: addi $3, $3, 1
	0x00400020	0x20840001	addi $4,$4,0x00000001	9: addi $4, $4, 1
	0x00400024	0x20a5ffff	addi $5,$5,0xffffffff	10: addi $5, $5, -1
	0x00400028	0x14a0fffa	bne $5,$0,0xfffffffa	11: bne $5,$0,L1

Εικόνα 10: Κωδικοί εντολών για τον Κώδικα

```
0x20000000
0x20420000
0x20820000
0x20030001
0x20050003
0x00603020
0xac860000
0x20630001
0x20840001
0x20a5ffff
0x14a0fffa
```

Entity & Architecture

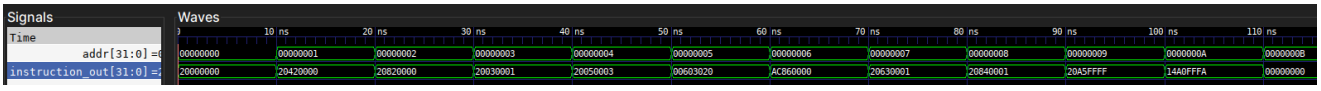
```
-- Imem.vhd
-- Implementation of Instruction Memory (ROM)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity imem is
port (
  addr: in std_logic_vector(31 downto 0);
  instruction: out std_logic_vector(31 downto 0)
);
end entity imem;

architecture behavioral of imem is
  type mem_type is array (0 to 15) of std_logic_vector(31 downto 0);
  signal mem : mem_type := (
    0 => x"20000000", -- addi $0, $0, 0
    1 => x"20420000", -- addi $2, $2, 0
    2 => x"20820000", -- addi $2, $4, 0
    3 => x"20030001", -- addi $3, $0, 1
    4 => x"20050003", -- addi $5, $0, 3
    5 => x"00603020", -- add $6, $3, $0
    6 => x"ac860000", -- sw $6, 0($4)
    7 => x"20630001", -- addi $3, $3, 1
    8 => x"20840001", -- addi $4, $4, 1
    9 => x"20a5ffff", -- addi $5, $5, -1
   10 => x"14a0fffa", -- bne $5, $0, L1
   others => (others => '0')
  );
begin

  main: process (addr)
    variable a: integer;
  begin
    a := to_integer(unsigned(addr));
    if a < 0 or a > 15 then
      instruction <= (others => 'X');
    else
      instruction <= mem(a);
    end if;
  end process;

end architecture behavioral;
```

Test Bench

```
-- Imem_tb.vhd
-- Test Bench for Instruction Memory (ROM)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity imem_tb is end entity imem_tb;

architecture behavioral of imem_tb is

  component imem is
  port (
    addr: in std_logic_vector(31 downto 0);
    instruction: out std_logic_vector(31 downto 0)
  );
  end component imem;

  constant CLK_PERIOD : time := 10 ns;

  signal addr : std_logic_vector(31 downto 0) := (others => '0');
  signal instruction_out : std_logic_vector(31 downto 0) := (others => '0');

  type mem_type is array (0 to 15) of std_logic_vector(31 downto 0);
  signal mem : mem_type := (
    0 => x"20000000", -- addi $0, $0, 0
    1 => x"20420000", -- addi $2, $2, 0
    2 => x"20820000", -- addi $2, $4, 0
    3 => x"20030001", -- addi $3, $0, 1
    4 => x"20050003", -- addi $5, $0, 3
    5 => x"00603020", -- add $6, $3, $0
    6 => x"ac860000", -- sw $6, 0($4)
    7 => x"20630001", -- addi $3, $3, 1
    8 => x"20840001", -- addi $4, $4, 1
    9 => x"20a5ffff", -- addi $5, $5, -1
   10 => x"14a0fffa", -- bne $5, $0, L1
   others => (others => '0')
  );

begin

  uut: imem port map (
    addr => addr,
    instruction => instruction_out
  );

  stim_proc : process
  begin
    for i in mem'range loop
      addr <= std_logic_vector(to_unsigned(i, 32));
      wait for CLK_PERIOD;
```

```
        assert instruction_out = mem(i)
          report "instruction mismatch at address " & integer'image(i)
          severity error;
      end loop;

      report "end of simulation"
      severity note;
      wait;

    end process stim_proc;

end architecture behavioral;
```

Wave



Εικόνα 11: Κυματομορφή της Instruction Memory

Αριθμητική και Λογική Μονάδα - ALU

Περιγραφή και Χρήση του Κυκλώματος

Η Αριθμητική και Λογική Μονάδα (Arithmetic and Logic Unit - ALU) είναι ένα από τα κεντρικά στοιχεία ενός επεξεργαστή.
Ο σκοπός και η χρήση της ALU περιλαμβάνει τα εξής:

- Εκτέλεση Αριθμητικών Λειτουργιών
  Εντολές `add`, `sub`, `addi`
- Εκτέλεση Λογικών Λειτουργιών
  Εντολές `and`, `or`, `xor`, `not`
- Εκτέλεση Λειτουργιών Μετατόπισης
  Εντολές `sll`, `srl`
- Εκτέλεση Λειτουργιών Διακλάδωσης και Σύγκρισης
  Εντολές `beq`, `bne`, `slt`
- Εκτέλεση Λειτουργιών Μνήμης
  Υπολογισμός της διεύθυνσης μνήμης με βάση την βασική διεύθυνση και το offset για τις εντολές `lw`, `sw`

Entity & Architecture

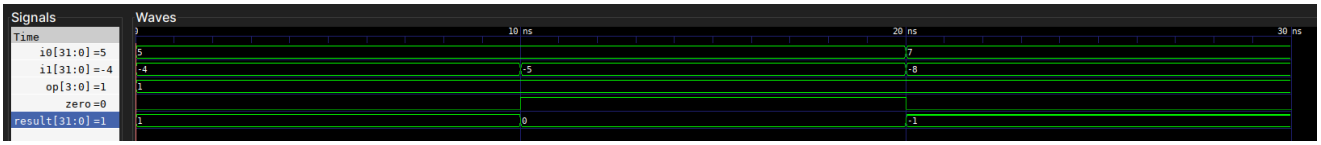
```
-- ALU.vhd
-- Implementation of Arithmetic and Logic Unit (ALU)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu is port (
  i0, i1: in std_logic_vector(31 downto 0);
  op: in std_logic_vector(3 downto 0);
  zero: out std_logic;
  result: out std_logic_vector(31 downto 0)
);
end entity alu;

architecture behavioral of alu is
  component adder32
  port (
    i0, i1: in std_logic_vector(31 downto 0);
    ci: in std_logic_vector(0 downto 0);
    s: out std_logic_vector(31 downto 0);
    co: out std_logic
  );
  end component;

  signal fa_out: std_logic_vector(31 downto 0);
  signal x: std_logic_vector(31 downto 0) := (others => 'X');
  signal co: std_logic;
begin

  fa: adder32 port map (
    i0 => i0,
    i1 => i1,
    ci => "0",
    co => co,
    s => fa_out
  );

  with op select result <=
    fa_out when "0001",
    fa_out when "1101",
    x when others;

  zero <=
    '1' when fa_out = std_logic_vector(to_signed(0, 32)) else
    '1' when ((op = "1011") and (i0 /= i1)) else -- bne
    '1' when ((op = "1010") and (i0 = i0)) else -- beq
    '0';

end architecture behavioral;
```

Test Bench

```
-- ALU_tb.vhd
-- Test Bench of Arithmetic and Logic Unit (ALU)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_tb is end entity alu_tb;

architecture behavioral of alu_tb is
  component alu is port (
    i0, i1: in std_logic_vector(31 downto 0);
    op: in std_logic_vector(3 downto 0);

    zero: out std_logic;
    result: out std_logic_vector(31 downto 0)
  );
  end component alu;

  signal i0, i1: std_logic_vector(31 downto 0);
  signal op: std_logic_vector(3 downto 0);
  signal zero: std_logic;
  signal result: std_logic_vector(31 downto 0);

  constant CLK_PERIOD: time := 10 ns;
begin

  uut: alu port map (
    i0 => i0,
    i1 => i1,
    op => op,
    zero => zero,
    result => result
  );

  stim_proc: process

    type pattern_type is record
      i0, i1: std_logic_vector(31 downto 0);
      op: std_logic_vector(3 downto 0);
      o: std_logic_vector(31 downto 0);
      zero: std_logic;
    end record;

    type pattern_array is array (natural range <>) of pattern_type;

    constant patterns: pattern_array := (
      -- Test 5 + (-4)
      (
        std_logic_vector(to_signed(5, 32)),
        std_logic_vector(to_signed(-4, 32)),
```

```
        "0001",
        std_logic_vector(to_signed(1, 32)),
        '0'
      ),
      -- Test 5 + (-5)
      (
        std_logic_vector(to_signed(5, 32)),
        std_logic_vector(to_signed(-5, 32)),
        "0001",
        std_logic_vector(to_signed(0, 32)),
        '1'
      ),
      -- Test 7 - 8
      (
        std_logic_vector(to_signed(7, 32)),
        std_logic_vector(to_signed(-8, 32)),
        "0001",
        std_logic_vector(to_signed(-1, 32)),
        '0'
      )
    );

  begin
    for i in patterns'range loop
      i0 <= patterns(i).i0;
      i1 <= patterns(i).i1;
      op <= patterns(i).op;

      wait for CLK_PERIOD;

      assert result = patterns(i).o
        report "unexpected output value for index " & integer'image(i)
        severity error;

      assert zero = patterns(i).zero
        report "unexpected zero flag value for index " & integer'image(i)
        severity error;
    end loop;

    report "end of simulation"
    severity note;
    wait;

  end process stim_proc;

end architecture behavioral;
```

Wave



Εικόνα 12: Κυματομορφή της ALU

Μονάδα ελέγχου ALU - ALU Control Unit

Περιγραφή και Χρήση του Κυκλώματος

Σκοπός της ALU Control Unit

Η μονάδα ελέγχου της ALU λαμβάνει τα σήματα από την κύρια μονάδα ελέγχου του επεξεργαστή και τα μετατρέπει σε συγκεκριμένες εντολές που καθορίζουν την ακριβή πράξη που πρέπει να εκτελέσει η ALU.

Λειτουργία της ALU Control Unit

1. *Λήψη Σημάτων Ελέγχου από την Κύρια Μονάδα Ελέγχου*:

   Κατά την ανάγνωση μιας εντολής από τη μνήμη, η κύρια μονάδα ελέγχου αναγνωρίζει τον τύπο της εντολής μέσω του κωδικού λειτουργίας (`opcode`) και, για εντολές τύπου R, του πεδίου λειτουργίας (`funct`).
2. *Δημιουργία Σημάτων Ελέγχου ALU*:

   Η ALU Control Unit λαμβάνει τα σήματα από την κύρια μονάδα ελέγχου και, βάσει αυτών, δημιουργεί τα κατάλληλα σήματα ελέγχου για την ALU. Αυτά τα σήματα υποδεικνύουν στην ALU ποια συγκεκριμένη λειτουργία να εκτελέσει (π.χ., πρόσθεση, αφαίρεση, λογικές πράξεις).
3. *Μεταφορά των Σημάτων στην ALU*:

   Τα σήματα ελέγχου που παράγονται από την ALU Control Unit στέλνονται στην ALU, η οποία στη συνέχεια εκτελεί την καθορισμένη λειτουργία στους καταχωρητές εισόδου.

Entity & Architecture

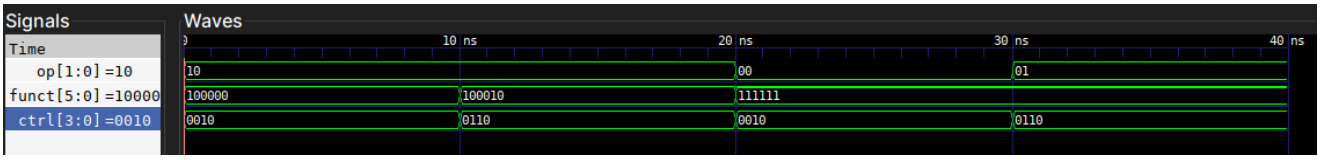
```
-- ALUcontrol.vhd
-- Implementation of ALU Control Unit

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_control is
port (
  op : in std_logic_vector(1 downto 0);
  funct : in std_logic_vector(5 downto 0);
  ctrl : out std_logic_vector(3 downto 0)
);
end entity alu_control;

architecture behavioral of alu_control is
begin

  process(funct, op)
  begin

    case op is

    when "00" => -- lw, sw, addi
      ctrl <= "0010";
    when "01" => -- beq, bne
      ctrl <= "0110";
    when "10" => -- R-type

      case funct is
      when "100000" => -- add
        ctrl <= "0010";
      when "100010" => -- sub
        ctrl <= "0110";
      when others =>
        ctrl <= (others => '0');
      end case;

    when others =>
      ctrl <= "0000";
    end case;

  end process;

end architecture behavioral;
```

Test Bench

```
-- ALUcontrol_tb.vhd
-- Test Bench of ALU Control Unit
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_control_tb is end entity alu_control_tb;

architecture behavioral of alu_control_tb is
  signal op : std_logic_vector(1 downto 0);
  signal funct : std_logic_vector(5 downto 0);
  signal ctrl : std_logic_vector(3 downto 0);

  constant CLK_PERIOD : time := 10 ns;

component alu_control is
port (
  op : in std_logic_vector(1 downto 0);
  funct : in std_logic_vector(5 downto 0);
  ctrl : out std_logic_vector(3 downto 0)
);
end component alu_control;

begin
  uut: alu_control port map (
    op => op,
    funct => funct,
    ctrl => ctrl
  );

  stim_proc: process
    type pattern_type is record
      funct : std_logic_vector(5 downto 0);
      op : std_logic_vector(1 downto 0);
      expected_ctrl : std_logic_vector(3 downto 0);
    end record;

    type pattern_array is array (natural range <>) of pattern_type;

    constant patterns: pattern_array := (
      (funct => "100000", op => "10", expected_ctrl => "0010"),
      (funct => "100010", op => "10", expected_ctrl => "0110"),
      (funct => "111111", op => "00", expected_ctrl => "0010"),
      (funct => "111111", op => "01", expected_ctrl => "0110")
    );

  begin

    for i in patterns'range loop
```

```
        funct <= patterns(i).funct;
        op <= patterns(i).op;

        wait for CLK_PERIOD;

        assert ctrl = patterns(i).expected_ctrl
        report "unexpected control value for pattern " & integer'image(i)
        severity error;

      end loop;

      report "end of simulation"
      severity note;
      wait;

    end process stim_proc;

end architecture behavioral;
```

Wave



Εικόνα 13: Κυματομορφή του ALU Control Unit

Μονάδα ελέγχου - Control Unit

Περιγραφή και Χρήση του Κυκλώματος

Ο σκοπός της μονάδας ελέγχου είναι να διευθύνει και να συντονίζει τις δραστηριότητες του επεξεργαστή, ώστε να εκτελεί σωστά τις εντολές ενός προγράμματος. Ακολουθούν οι λεπτομέρειες για τον σκοπό και τη χρήση της μονάδας ελέγχου σε έναν MIPS επεξεργαστή.

Σκοπός της Μονάδας Ελέγχου

1. *Διερμηνεία Εντολών*:

   Η μονάδα ελέγχου αναγνωρίζει και διερμηνεύει τις εντολές που ανακτώνται από τη μνήμη. Αναλύει τον κωδικό λειτουργίας (opcode) κάθε εντολής και καθορίζει ποια λειτουργία πρέπει να εκτελεστεί.
2. *Γένεση Σημάτων Ελέγχου*:

   Η μονάδα ελέγχου παράγει σήματα ελέγχου που χρησιμοποιούνται για να κατευθύνουν τα άλλα μέρη του επεξεργαστή. Αυτά τα σήματα ελέγχου καθορίζουν τις ενέργειες που πρέπει να ληφθούν, όπως η επιλογή της σωστής πηγής δεδομένων, ο καθορισμός του τύπου της αριθμητικής ή λογικής λειτουργίας που θα εκτελεστεί από την ALU, και άλλες παρόμοιες λειτουργίες.
3. *Διαχείριση Ακολουθίας Εκτέλεσης*:

   Η μονάδα ελέγχου διαχειρίζεται τη σειρά με την οποία εκτελούνται οι εντολές. Διασφαλίζει ότι οι εντολές εκτελούνται στη σωστή σειρά και ότι οι κατάλληλοι πόροι είναι διαθέσιμοι για κάθε εντολή.
4. *Συντονισμός Δεδομένων και Σημάτων*:

   Συντονίζει τη ροή των δεδομένων και των σημάτων εντός του επεξεργαστή, διασφαλίζοντας ότι οι κατάλληλες μονάδες ενεργοποιούνται και λειτουργούν συγχρονισμένα.

Entity & Architecture

```
-- Control.vhd
-- Implementation of Control Unit
library ieee;
use ieee.std_logic_1164.all;

entity control is
port (
  clk: in std_logic;
  input: in std_logic_vector(5 downto 0);
  AluOp: out std_logic_vector(1 downto 0);
  AluSrc,
  RegDst,
  RegWrite,
  MemRead,
  MemWrite,
  MemToReg,
  Branch,
  Jump: out std_logic
);
end entity control;

architecture behavioral of control is
  constant lw: std_logic_vector(5 downto 0) := "100011";
  constant sw: std_logic_vector(5 downto 0) := "101011";
  constant add: std_logic_vector(5 downto 0) := "000000";
  constant addi: std_logic_vector(5 downto 0) := "001000";
  constant beq: std_logic_vector(5 downto 0) := "000100";
  constant bne: std_logic_vector(5 downto 0) := "000101";
  constant j: std_logic_vector(5 downto 0) := "000010";
begin

  main: process(clk, input)
  begin

    case input is
    when lw =>
      AluOp <= "00";
      RegDst <= '0';
      AluSrc <= '1';
      MemToReg <= '1';
      RegWrite <= '1';
      MemRead <= '1';
      MemWrite <= '0';
      Branch <= '0';
      Jump <= '0';
    when sw =>
      AluOp <= "00";
      RegDst <= '0';
      AluSrc <= '1';
      MemToReg <= '0';
```

```
      RegWrite <= '0';
      MemRead <= '0';
      MemWrite <= '1';
      Branch <= '0';
      Jump <= '0';
    when add =>
      AluOp <= "10";
      RegDst <= '1';
      AluSrc <= '0';
      MemToReg <= '0';
      RegWrite <= '1';
      MemRead <= '0';
      MemWrite <= '0';
      Branch <= '0';
      Jump <= '0';
    when addi =>
      AluOp <= "00";
      RegDst <= '0';
      AluSrc <= '1';
      MemToReg <= '0';
      RegWrite <= '1';
      MemRead <= '0';
      MemWrite <= '0';
      Branch <= '0';
      Jump <= '0';
    when beq =>
      AluOp <= "01";
      RegDst <= '0';
      AluSrc <= '0';
      MemToReg <= '0';
      RegWrite <= '0';
      MemRead <= '0';
      MemWrite <= '0';
      Branch <= '1';
      Jump <= '0';
    when bne =>
      AluOp <= "01";
      RegDst <= '0';
      AluSrc <= '0';
      MemToReg <= '0';
      RegWrite <= '0';
      MemRead <= '0';
      MemWrite <= '0';
      Branch <= '1';
      Jump <= '0';
    when j =>
      AluOp <= "XX";
      RegDst <= 'X';
      AluSrc <= 'X';
      MemToReg <= 'X';
      RegWrite <= 'X';
      MemRead <= 'X';
      MemWrite <= 'X';
```

```
      Branch <= 'X';
      Jump <= '1';
    when others =>
      AluOp <= "00";
      AluSrc <= '0';
      RegDst <= '0';
      RegWrite <= '0';
      MemRead <= '0';
      MemWrite <= '0';
      MemToReg <= '0';
      Branch <= '0';
      Jump <= '0';
    end case;
  end process main;

end architecture behavioral;
```

Test Bench

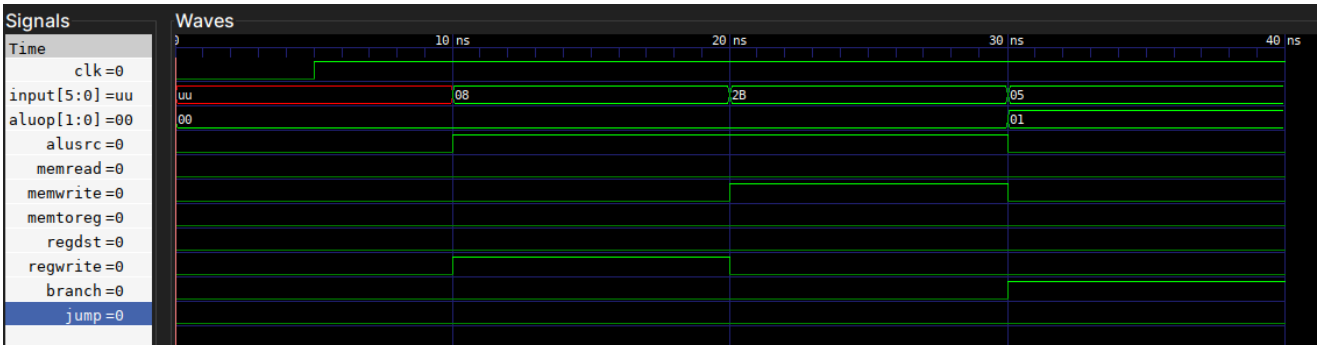
```
-- Control_tb.vhd
-- Test Bench of Control Unit
library ieee;
use ieee.std_logic_1164.all;

entity control_tb is end entity control_tb;

architecture behavioral of control_tb is

  component control
  port (
    clk: in std_logic;
    input: in std_logic_vector(5 downto 0);
    RegWrite: out std_logic;
    AluOp: out std_logic_vector(1 downto 0);
    AluSrc,
    MemWrite,
    MemRead,
    RegDst,
    MemToReg,
    Jump,
    Branch: out std_logic
  );
  end component;

  signal clk: std_logic := '0';
  signal input: std_logic_vector(5 downto 0);
  signal RegWrite: std_logic;
  signal AluOp: std_logic_vector(1 downto 0);
  signal
    AluSrc,
    MemWrite,
    MemRead,
    RegDst,
    MemToReg,
    Jump,
    Branch: std_logic;

begin
  uut: control port map (
    clk => clk,
    input => input,
    RegWrite => RegWrite,
    AluOp => AluOp,
    AluSrc => AluSrc,
    MemWrite => MemWrite,
    MemRead => MemRead,
    RegDst => RegDst,
    MemToReg => MemToReg,
    Jump => Jump,
```

```
    Branch => Branch
    );

  stim_proc: process

    type pattern_type is record
      input: std_logic_vector(5 downto 0);
      RegWrite: std_logic;
      AluOp: std_logic_vector(1 downto 0);
      AluSrc: std_logic;
      MemWrite: std_logic;
      MemRead: std_logic;
      RegDst: std_logic;
      MemToReg: std_logic;
      Jump: std_logic;
      Branch: std_logic;
    end record;

    type pattern_array is array (natural range <>) of pattern_type;

    constant patterns: pattern_array := (
      -- addi $0, $0, 0 (opcode = 001000)
      (
        input => "001000",
        RegWrite => '1',
        AluOp => "00",
        AluSrc => '1',
        MemWrite => '0',
        MemRead => '0',
        RegDst => '0',
        MemToReg => '0',
        Jump => '0',
        Branch => '0'
      ),
      -- sw $6, 0($4) (opcode = 101011)
      (
        input => "101011",
        RegWrite => '0',
        AluOp => "00",
        AluSrc => '1',
        MemWrite => '1',
        MemRead => '0',
        RegDst => '0',
        MemToReg => '0',
        Jump => '0',
        Branch => '0'
      ),
      -- bne $5, $0, L1 (opcode = 000101)
      (
        input => "000101",
        RegWrite => '0',
        AluOp => "01",
        AluSrc => '0',
```

```
        MemWrite => '0',
        MemRead => '0',
        RegDst => '0',
        MemToReg => '0',
        Jump => '0',
        Branch => '1'
      )
    );

  begin
    clk <= '0';
    wait for 5 ns;
    clk <= '1';
    wait for 5 ns;

    for i in patterns'range loop
      input <= patterns(i).input;

      wait for 10 ns;

      assert RegWrite = patterns(i).RegWrite
      report "unexpected RegWrite value for index " & integer'image(i)
      severity error;

      assert AluOp = patterns(i).AluOp
      report "unexpected AluOp value for index " & integer'image(i)
      severity error;

      assert AluSrc = patterns(i).AluSrc
      report "unexpected AluSrc value for index " & integer'image(i)
      severity error;

      assert MemWrite = patterns(i).MemWrite
      report "unexpected MemWrite value for index " & integer'image(i)
      severity error;

      assert MemRead = patterns(i).MemRead
      report "unexpected MemRead value for index " & integer'image(i)
      severity error;

      assert RegDst = patterns(i).RegDst
      report "unexpected RegDst value for index " & integer'image(i)
      severity error;

      assert MemToReg = patterns(i).MemToReg
      report "unexpected MemToReg value for index " & integer'image(i)
      severity error;

      assert Jump = patterns(i).Jump
      report "unexpected Jump value for index " & integer'image(i)
      severity error;

      assert Branch = patterns(i).Branch
```

```
        report "unexpected Branch value for index " & integer'image(i)
        severity error;
      end loop;

      report "end of simulation"
      severity note;
      wait;

    end process;

end architecture behavioral;
```

Wave



Εικόνα 14: Κυματομορφή του Control Unit

MIPS

Entity & Architecture

```
-- 20390068_ZOUMAS_02_MIPS.vhd
-- Implementation of MIPS processor
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mips is
    port (
        clk, rst: in std_logic
    );
end entity mips;

architecture behavioral of mips is

    component adder32 is port (
      i0, i1: in std_logic_vector(31 downto 0);
      ci: in std_logic_vector(0 downto 0);
      s: out std_logic_vector(31 downto 0)
    );
    end component adder32;

    component alu is port (
      i0, i1: in std_logic_vector(31 downto 0);
      op: in std_logic_vector(3 downto 0);
      zero: out std_logic;
      result: out std_logic_vector(31 downto 0)
    );
    end component alu;

    component pc is port (
      input: in std_logic_vector(31 downto 0);
      clk, rst: in std_logic;
      output: out std_logic_vector(31 downto 0)
    );
    end component pc;

    component imem is port (
      addr: in std_logic_vector(31 downto 0);
      instruction: out std_logic_vector(31 downto 0)
    );
    end component imem;

    component register_file is port (
      RegIn1, RegIn2, RegToWrite: in std_logic_vector(4 downto 0);
      RegWrite: in std_logic;
      DataToWrite: in std_logic_vector(31 downto 0);
      RegOut1, RegOut2: out std_logic_vector(31 downto 0)
    );
```

```
    end component register_file;

    component alu_control is port (
      op : in std_logic_vector(1 downto 0);
      funct : in std_logic_vector(5 downto 0);
      ctrl : out std_logic_vector(3 downto 0)
    );
    end component alu_control;

    component control is port (
      clk: in std_logic;
      input: in std_logic_vector(5 downto 0);
      AluOp: out std_logic_vector(1 downto 0);
      AluSrc,
      MemWrite,
      MemRead,
      RegDst,
      RegWrite,
      MemToReg,
      Jump,
      Branch: out std_logic
    );
    end component control;

    component data_mem is port (
      clk, rst: in std_logic;
      input, WriteData: in std_logic_vector(31 downto 0);
      MemRead, MemWrite: in std_logic;
      output: out std_logic_vector(31 downto 0)
    );
    end component data_mem;

    component mux2_5 is port (
      i0, i1: in std_logic_vector(4 downto 0);
      o: out std_logic_vector(4 downto 0);
      sel: in std_logic
    );
    end component mux2_5;

    component mux2_32 is port (
      i0, i1: in std_logic_vector(31 downto 0);
      o: out std_logic_vector(31 downto 0);
      sel: in std_logic
    );
    end component mux2_32;

    component signext is port (
      input: in std_logic_vector(15 downto 0);
      output: out std_logic_vector(31 downto 0)
    );
    end component signext;

    signal
```

```
        RegWrite,
        ALUSrc,
        MemWrite,
        MemRead,
        RegDst,
        MemToReg,
        Jump,
        Zero,
        Branch,
        BranchTaken: std_logic;

    signal AluOp: std_logic_vector(1 downto 0);
    signal ALUControl_OUT: std_logic_vector(3 downto 0);
    signal MUX_REG_OUT: std_logic_vector(4 downto 0);

    signal
        PC_FA_IM,
        FA_PC_OUT,
        IM_OUT,
        ONE,
        ALU_OUT,
        RegOut1,
        RegOut2,
        DataToWrite,
        MUX_ALU_OUT,
        SignExt_OUT,
        RAM_OUT,
        ShiftJump2MuxJump,
        MuxJump2PC,
        MuxBranch2MuxJump,
        ALU_Branch_OUT: std_logic_vector(31 downto 0);

begin
    ONE <= std_logic_vector(to_unsigned(1, 32));

    FA_PC: adder32 port map(
      i0 => PC_FA_IM,
      i1 => ONE,
      ci => "0",
      s => FA_PC_OUT
    );

    FA_BRANCH: adder32 port map(
      i0 => FA_PC_OUT,
      i1 => SignExt_OUT,
      ci => "0",
      s => ALU_Branch_OUT
    );

    pc_inst: pc port map(
      input => MuxJump2PC,
      clk => clk,
      rst => rst,
```

```
    output => PC_FA_IM
  );

  imem_inst: imem port map(
    addr => PC_FA_IM,
    instruction => IM_OUT
  );

  register_file_inst: register_file port map(
    RegIn1 => IM_OUT(25 downto 21),
    RegIn2 => IM_OUT(20 downto 16),
    RegToWrite => MUX_REG_OUT,
    RegWrite => RegWrite,
    DataToWrite => DataToWrite,
    RegOut1 => RegOut1,
    RegOut2 => RegOut2
  );

  control_inst: control port map(
    clk => clk,
    input => IM_OUT(31 downto 26),
    RegWrite => RegWrite,
    AluOp => AluOp,
    AluSrc => ALUSrc,
    MemWrite => MemWrite,
    MemRead => MemRead,
    RegDst => RegDst,
    MemToReg => MemToReg,
    Jump => Jump,
    Branch => Branch
  );

  alu_control_inst: alu_control port map(
    op => AluOp,
    funct => IM_OUT(5 downto 0),
    ctrl => ALUControl_OUT
  );

  alu_inst: alu port map(
    i0 => RegOut1,
    i1 => MUX_ALU_OUT,
    op => ALUControl_OUT,
    zero => Zero,
    result => ALU_OUT
  );

  data_mem_inst: data_mem port map(
    clk => clk,
    rst => rst,
    input => ALU_OUT,
    WriteData => RegOut2,
    MemRead => MemRead,
    MemWrite => MemWrite,
```

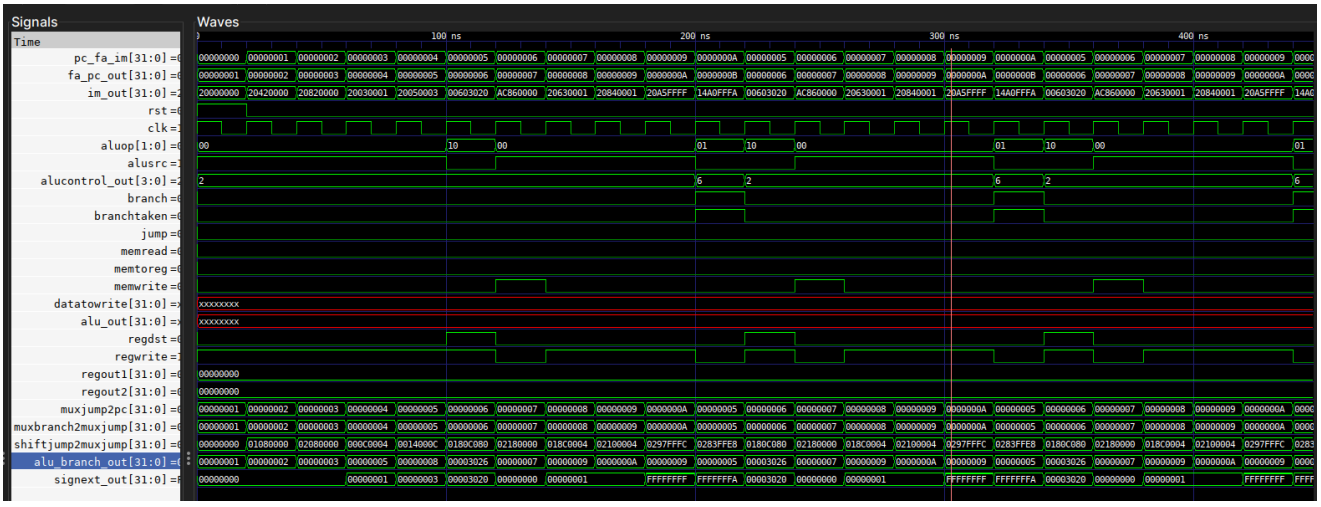
```
      output => RAM_OUT
    );

    MUX_REG: mux2_5 port map(
      i0 => IM_OUT(20 downto 16),
      i1 => IM_OUT(15 downto 11),
      o => MUX_REG_OUT,
      sel => RegDst
    );

    MUX_ALU_IN: mux2_32 port map(
      i0 => RegOut2,
      i1 => SignExt_OUT,
      o => MUX_ALU_OUT,
      sel => ALUSrc
    );

    MUX_RAM: mux2_32 port map(
      i0 => ALU_OUT,
      i1 => RAM_OUT,
      o => DataToWrite,
      sel => MemToReg
    );

    ShiftJump2MuxJump <= FA_PC_OUT(31 downto 28) & IM_OUT(25 downto 0) & "00";

    MUX_JUMP: mux2_32 port map(
      i0 => MuxBranch2MuxJump,
      i1 => ShiftJump2MuxJump,
      o => MuxJump2PC,
      sel => Jump
    );

    BranchTaken <= Branch and Zero;

    MUX_BRANCH: mux2_32 port map(
      i0 => FA_PC_OUT,
      i1 => ALU_Branch_OUT,
      o => MuxBranch2MuxJump,
      sel => BranchTaken
    );

    signext_inst: signext port map(
      input => IM_OUT(15 downto 0),
      output => SignExt_OUT
    );

end architecture behavioral;
```

Test Bench

```
-- 20390068_ZOUMAS_03_testbench.vhd
-- Test Bench of MIPS processor
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mips_tb is
end entity mips_tb;

architecture testbench of mips_tb is

  signal clk, rst: std_logic := '0';

  component mips
  port (
    clk: in std_logic;
    rst: in std_logic
  );
  end component mips;

  signal simulation_done: boolean := false;

begin

  uut: mips port map (
    clk => clk,
    rst => rst
  );

  clk_process: process
  begin
    while not simulation_done loop
      clk <= '1';
      wait for 10 ns;
      clk <= '0';
      wait for 10 ns;
    end loop;
    wait;
  end process clk_process;


  stim_proc: process
  begin

    rst <= '1';
    wait for 20 ns;
    rst <= '0';

    wait for 10000 ns;
```

```
    simulation_done <= true;
    wait;
  end process stim_proc;

end architecture testbench;
```

Wave



Εικόνα 15: Κυματομορφή του MIPS