ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΣΧΕΔΙΑΣΗ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΑΜ : 20390132

ΟΝΟΜΑΤΕΠΩΝΥΜΟ : ΛΟΓΟΘΕΤΗΣ ΑΛΕΞΙΟΣ ΑΝΤΩΝΙΟΣ

4ο ΕΞΑΜΗΝΟ

Κύριος Στόχος Εργασίας.

Ο κύριος στόχος αυτής της εργαστηριακής εργασίας είναι ο σχεδιασμός μιας CPU ενός κύκλου με βάση την αρχιτεκτονική MIPS. Για την επίτευξη αυτού του στόχου, ο φοιτητής χρειάζεται να εφαρμόσει τις γνώσεις που απέκτησε καθ' όλη τη διάρκεια του εξαμήνου σχετικά με το σχεδιασμό και την υλοποίηση συνδυαστικών και διαδοχικών ψηφιακών κυκλωμάτων. Αυτό το εργαστήριο απαιτεί επίσης μια σταθερή κατανόηση της αρχιτεκτονικής MIPS και της αρχιτεκτονικής του συνόλου εντολών, καθώς το μοντέλο CPU που πρέπει να ακολουθηθεί βασίζεται σε μεγάλο βαθμό σε ένα ελάχιστο προσχέδιο MIPS που υλοποιεί ένα σύνολο βασικών οδηγιών από την αρχιτεκτονική. Επιπλέον, για αυτή την εργασία χρειάζεται ο μαθητής να έχει γνώσεις σε μια γλώσσα περιγραφής υλικού όπως η VHDL και να γνωρίζει πώς να χρησιμοποιεί εργαλεία σχεδίασης βοηθημάτων υπολογιστή όπως το ModelSim.

Σχεδιασμός και εκτέλεση.

Η CPU ενός κύκλου που έχουμε σχεδιάσει αποτελείται από έντεκα κύριες μονάδες που θα περιγράψουμε λεπτομερώς ξεχωριστά. Αυτές οι μονάδες, που φαίνονται στο σχήμα 1, είναι: Program Counter, Instruction Memory, Register File, ALU, Data Memory, Control Unit, ALU Control, Adder 32 bit, Sign Extend 16 το 32, Shift Left 2 και Multiplexers. Οι πολλαπλές διαδρομές δεδομένων που παρουσιάζονται στο σχήμα επιτρέπουν την υλοποίηση των περισσότερων από τις οδηγίες που εφαρμόσαμε.

Program Counter.

Ο μετρητής προγράμματος είναι απλώς ένας καταχωρητής πλάτους 32 bit που αποθηκεύει τη διεύθυνση της εντολής που εκτελείται αυτήν τη στιγμή. Παίζει σημαντικό ρόλο στη φάση ανάκτησης εντολών, καθώς παρέχει τη διεύθυνση που απαιτείται για την ανάκτηση μιας εντολής από τη μνήμη εντολών. Ο μετρητής του προγράμματός μας ενεργοποιείται με θετική ακμή ρολογιού. τα περιεχόμενα ενός στοιχείου κατάστασης που ενεργοποιείται από θετική ακμή μπορεί να ενημερωθεί μόνο όταν το σήμα ρολογιού μεταβαίνει από το χαμηλό στο υψηλό.

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

entity ProgramCounter is
GENERIC (n : integer := 32);
port( -- input
      CLK         : in  std_logic;
      reset_neg   : in  std_logic;
      input       : in  std_logic_vector(31 downto 0);

      -- output
      output : out std_logic_vector(31 downto 0) );
end ProgramCounter;

architecture Behavioral of ProgramCounter is

begin

  process(CLK)
  begin
    if reset_neg = '0' then
      output <= (others => '0' );
    elsif rising_edge(CLK) then
      output <= input;
    end if;
  end process;

end Behavioral;
```

Instruction Memory.

Η μνήμη εντολών μας χρησιμοποιείται για να αποθηκεύσουμε τις εντολές του προγράμματος του οποίου θέλουμε να εκτελέσουμε. Η μνήμη εντολών μας έχει μια είσοδο και μια έξοδο. Η είσοδος έρχεται απο τον Program Counter και είναι η διεύθυνση της εντολής η οποία θέλουμε να εκτελεστεί. Η έξοδος μας είναι η ίδια η εντολή σε δυαδική μορφή.

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

entity InstructionMemory is
  port( -- input
        register_addr : in  std_logic_vector(31 downto 0);

        -- output
        instruction   : out std_logic_vector(31 downto 0) );
end InstructionMemory;

architecture Behavioral of InstructionMemory is

type reg is array (0 to 10) of std_logic_vector(31 downto 0);
signal instr_memory: reg := (

    --addi $3,$0,1
    0 => "00100000011000000000000000000001",
    --addi $5,$0,3
    1=>  "00100000101000000000000000000011",
    -- L1:add $6,$3,$0
    2=>  "00000000011000000011000000100000",
    --sw $6,0($4)
    3=>  "10101100100001100000000000000000",
    --addi $3,$3,1
    4=>  "00100000011000110000000000000001",
    --addi $4,$4,1
    5=>  "00100000100000100000000000000001",
    --addi $5,$5,01
    6=>  "00100000101001011111111111111111",
    --bne $5,$0,L1
    7=>  "00010100101000001111111111111010",
    others=> "00000000000000000000000000000000" );

begin
  instruction <= instr_memory(to_integer(unsigned(register_addr)));
end Behavioral;
```

Register File.

Το αρχείο καταχωρητή είναι επίσης ένα στοιχείο κατάστασης που ενεργοποιείται από θετικά άκρα. Αυτό το στοιχείο μνήμης αποτελείται από 32 καταχωρητές πλάτους 32 bit, στους οποίους μπορείτε να έχετε πρόσβαση μέσω διευθύνσεων 5 bit. Καθε καταχωρητής προσδιορίζεται με έναν αριθμό μεταξύ 0 και 31. Επειδή όλες οι εντολές τύπου R MIPS καθορίζουν τρεις τελεστές καταχωρητών, δύο καταχωρητές πηγής (RS και RT) συν έναν καταχωρητή προορισμού (RD), το register file μας μεταφέρεται με τριπλή θύρα. Δύο από τις θύρες χρησιμοποιούνται για τον καθορισμό δύο διευθύνσεων ανάγνωσης, ενώ η υπόλοιπη χρησιμοποιείται για την παροχή μιας ενιαίας διεύθυνσης εγγραφής. Ως αποτέλεσμα, το αρχείο καταχωρητή έχει δύο αποκλειστικούς διαύλους για την έξοδο των περιεχομένων των δύο καταχωρητών που προσδιορίζονται από RS και RT και ένα σήμα εισόδου διαύλου για την παροχή της τιμής 32 bit που πρέπει να γραφτεί στον καταχωρητή που υποδεικνύεται από το RD. Επιπλέον, το σήμα εισόδου RegWrite, η τιμή του οποίου παράγεται από τη μονάδα ελέγχου ανά πάσα στιγμή, καθορίζει πότε το αρχείο καταχωρητή γίνεται εγγράψιμο.

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

entity Registers is
  GENERIC(n : integer := 32);
  port( -- input
        CLK          : in std_logic;
        reset_neg    : in std_logic;
        address_in_1 : in std_logic_vector(4 downto 0);
        address_in_2 : in std_logic_vector(4 downto 0);
        address_out  : in std_logic_vector(4 downto 0);

        write_data   : in std_logic_vector(n - 1 downto 0);
        RegWrite     : in std_logic;  -- signal control

        -- output
        register_1   : out std_logic_vector(n - 1 downto 0);
        register_2   : out std_logic_vector(n - 1 downto 0) );
end Registers;

architecture Behavioral of Registers is
  type   registers_type is array (0 to 31) of std_logic_vector(n - 1 downto 0);
  signal reg : registers_type := ((others => (others => '0')));

begin

  process(CLK)
  begin
    if reset_neg = '0' then -- reset
      reg(to_integer(unsigned(address_out))) <= (others => '0');
      else if rising_edge(CLK) and RegWrite = '1' then
        reg(to_integer(unsigned(address_out))) <= write_data;
      end if;
    end if;
  end process;

  register_1 <= reg(to_integer(unsigned(address_in_1)));  -- read in address 1
  register_2 <= reg(to_integer(unsigned(address_in_2)));  -- read in address 2

end Behavioral;
```

Arithmetical Logic Unit

Η αριθμητική λογική μονάδα λειτουργεί σε ακέραιες τιμές 32 bit και μπορεί να υπολογίσει τις ακόλουθες πράξεις: πρόσθεση, αφαίρεση, bitwise XOR και αριστερή ολίσθηση κατά 2. Οι δύο τελεστές, τους οποίους ονομάζουμε operand_1 και operand_2, υποτίθεται ότι είναι σε μορφή συμπληρώματος δύο. Η λειτουργία της ALU ελέγχεται με έναν κωδικό 2-bit που δημιουργείται από τη μονάδα ελέγχου ALU.Αποφασίσαμε να μην δημιουργήσουμε μια ξεχωριστή μονάδα ελέγχου για την ALU αλλα να την ενσωματόσουμε μαζί με την μονάδα της ALU.

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

entity ALU is
  GENERIC(n : integer := 32);
  port( -- input
        operand_1   : in std_logic_vector(n - 1 downto 0);
        operand_2   : in std_logic_vector(n - 1 downto 0);
        ALU_control : in std_logic_vector(1 downto 0);  -- 4 operations

        -- output
        result      : out std_logic_vector(n - 1 downto 0);
        zero        : out std_logic );
end ALU;

architecture Behavioral of ALU is
  signal temp : std_logic_vector(n - 1 downto 0);

begin

  temp <=
    -- add
    std_logic_vector(unsigned(operand_1) + unsigned(operand_2)) after 1 ns when ALU_control = "00" else
    -- subtract
    std_logic_vector(unsigned(operand_1) - unsigned(operand_2)) after 1 ns when ALU_control = "01" else
    -- XOR
    operand_1 XOR  operand_2 after 1 ns when ALU_control = "11" else
    -- shift left logical
    std_logic_vector(shift_left(unsigned(operand_1), to_integer(unsigned(operand_2(10 downto 6)))))
    after 1 ns when ALU_control = "10" else

    -- in other cases
    (others => '0');

  zero <= '1' when (temp <= "00000000000000000000000000000000") else '0';
  result <= temp;

end Behavioral;
```

Data Memory.

Ακριβώς όπως η μνήμη εντολών και το αρχείο καταχωρητή, η μνήμη δεδομένων μπορεί να γραφτεί μόνο στην ανερχόμενη άκρη του σήματος ρολογιού. Τα σήματα ελέγχου MemWrite και MemRead, των οποίων οι τιμές δημιουργούνται από τη μονάδα ελέγχου, χρησιμοποιούνται για τη διαχείριση αυτού του στοιχείου κατάστασης το οποίο έχει μόνο μία θύρα και για λειτουργίες ανάγνωσης και εγγραφής. Σε κάθε δεδομένη στιγμή, μόνο ένα από αυτά τα δύο σήματα ελέγχου μπορεί να επιβεβαιωθεί.

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

entity DataMemory is
  GENERIC(n : integer := 32);
  port( -- inputs
        CLK            : in std_logic;
        reset_neg      : in std_logic;
        memory_address : in std_logic_vector(n - 1 downto 0);
        MemWrite       : in std_logic;
        MemRead        : in std_logic;
        data_in        : in std_logic_vector(n - 1 downto 0);

        -- output
        data_out       : out std_logic_vector(n - 1 downto 0) );
end DataMemory;

architecture Behavioral of DataMemory is
  type mem_type is array  (127 downto 0) of std_logic_vector(n - 1 downto 0); -- should be 2^30 words
  signal mem: mem_type;

begin

  process(CLK, reset_neg)
  begin
  if reset_neg = '0' then
    mem <= (others => (others=>'0'));
  elsif rising_edge(CLK) and MemWrite = '1' then
    mem(to_integer(unsigned(memory_address))) <= data_in;
  end if;
  end process;

    data_out <= (mem(to_integer(unsigned(memory_address)))) when MemRead = '1' else (others => '0');

end Behavioral;
```

Control Unit.

Η κύρια μονάδα ελέγχου λαμβάνει ως είσοδο τα έξι πιο σημαντικά bit της εξόδου εντολών των 32 bit από τη μνήμη εντολών. Στις διάφορες μορφές στο σύνολο εντολών MIPS, αυτό το τμήμα της εντολής αντιστοιχεί στο πεδίο OPCODE. Αυτό το πεδίο καθορίζει τον τύπο λειτουργίας που κωδικοποιείται σε μια εντολή. Ως έξοδοι, το κύκλωμα έχει πολλαπλές γραμμές ελέγχου που επιβλέπουν τη λειτουργία των διακριτών στοιχείων της CPU.

```
entity ControlUnit is
  port( -- input
        instruction : in std_logic_vector(31 downto 0);
        ZeroCarry   : in std_logic;

        -- output (control signals)
        RegDst      : out std_logic;
        Branch      : out std_logic;
        MemRead     : out std_logic;
        MemToReg    : out std_logic;
        ALUOp       : out std_logic_vector (1 downto 0);  -- 3 operations
        MemWrite    : out std_logic;
        ALUSrc      : out std_logic;
        RegWrite    : out std_logic );
end ControlUnit;
```

```
architecture Behavioral of ControlUnit is
  signal data : std_logic_vector(8 downto 0);  -- used to set the control signals

begin
  -- in according to the standard MIPS32 instruction reference
  -- add
  data <= "100000001" when (instruction(31 downto 26) = "000000" and
                            instruction(10 downto 0)  = "00000000001") else
  -- subtract
  "100001001" when (instruction(31 downto 26) = "000000" and
                    instruction(10 downto 0)  = "00000100010") else
  -- shift left logical
  "100010011" when (instruction(31 downto 26) = "000000" and
                    instruction(5 downto 0)   = "000000") else
  -- add immediate
  "000000011" when instruction(31 downto 26) = "001000" else
  -- load word
  "001100011" when instruction(31 downto 26) = "100011" else
  -- store word
  "000000110" when instruction(31 downto 26) = "101011" else
  -- branch on not equal
  "010011010" when instruction(31 downto 26) = "000101" else
  -- otherwise
  (others =>'0');

  RegDst   <= data(8);
  -- AND port included considering the LSB of beq and bne
  Branch   <= data(7) AND (ZeroCarry XOR instruction(26));
  MemRead  <= data(6);
  MemToReg <= data(5);
  ALUOp    <= data(4 downto 3);  -- 4 operations available
  MemWrite <= data(2);
  ALUSrc   <= data(1);
  RegWrite <= data(0);

end Behavioral;
```

32 Bit Adder.

Αυτό το κύκλωμα είναι απλώς ένας αθροιστής που υπολογίζει το άθροισμα δύο τελεστών 32-bit. Στην υλοποίηση της CPU μας, χρησιμοποιούνται συνολικά δύο αθροιστές. Η πρώτη υπολογίζει την επόμενη διαδοχική διεύθυνση εντολών προσθέτοντας ενα στη διεύθυνση που είναι αποθηκευμένη στον μετρητή προγράμματος. Το δεύτερο, από την άλλη πλευρά, υπολογίζει τη διεύθυνση στόχου διακλάδωσης, η οποία βρίσκεται προσθέτοντας την επόμενη διαδοχική διεύθυνση εντολών (PC + 1) στο προσημασμένο εκτεταμένο immediate πεδίο 16 bit από την εντολή που έχει μετατοπιστεί δύο θέσεις προς τα αριστερά. Αυτοί οι αθροιστές δεν έχουν καμία έξοδο εκτός από το δίαυλο αποτελεσμάτων 32 bit.

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

entity Adder is
  GENERIC(n : integer := 32);
  port( -- input
        operand_1 : in  std_logic_vector(n - 1 downto 0);
        operand_2 : in  std_logic_vector(n - 1 downto 0);

        -- output
        result    : out std_logic_vector(n - 1 downto 0) );
end Adder;

architecture Behavioral of Adder is
begin

  result <= std_logic_vector(unsigned(operand_1) + unsigned(operand_2));

end Behavioral;
```

Shift Left 2.

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

entity ShiftLeft2 is
  port( -- input
        input  : in std_logic_vector(31 downto 0);

        -- output
        output : out std_logic_vector(31 downto 0) );
end ShiftLeft2;

architecture Behavioral of ShiftLeft2 is
begin

  output <= std_logic_vector(unsigned(input) sll 2);

end Behavioral;
```

Sign Extend 16 to 32.

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

entity SignExtend is
  port( -- input
        input  : in std_logic_vector(15 downto 0);

        -- output
        output : out std_logic_vector(31 downto 0) );
end SignExtend;

architecture Behavioral of SignExtend is
begin

  output <= "0000000000000000" & input when (input(15) = '0') else
            "1111111111111111" & input;

end Behavioral;
```

Multiplexer.

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

entity Mux is
  GENERIC(n : integer := 32);
  port( -- input
        input_1    : in std_logic_vector(n - 1 downto 0);
        input_2    : in std_logic_vector(n - 1 downto 0);
        mux_select : in std_logic;

        -- output
        output     : out std_logic_vector(n - 1 downto 0) );
end Mux;

architecture Behavioral of Mux is
begin
with mux_select select
  output <= input_1 when '0', input_2 when others;

end Behavioral;
```

Data Path.

Το DataPath ορίζει τον τρόπο με τον οποίο θα κινηθούν τα δεδομένα μέσα στο κύκλωμα μας από component σε component,για τον λόγο αυτό έχουμε δημιουργήσει όσα σήματα "καλώδια" χρειάστηκαν.

```
entity DataPath is
   GENERIC(n : integer := 32);
   port( -- inputs
         CLK, reset_neg    : in std_logic;
         instruction       : in std_logic_vector(31 downto 0);
         -- control signals
         RegDst            : in std_logic;
         Branch            : in std_logic;
         MemRead           : in std_logic;
         MemToReg          : in std_logic;
         ALUOp             : in std_logic_vector(1 downto 0);
         MemWrite          : in std_logic;
         ALUSrc            : in std_logic;
         RegWrite          : in std_logic;
         -- outputs
         next_instruction  : out std_logic_vector(31 downto 0);
         ZeroCarry         : out std_logic );
end DataPath;
```

```vhdl
architecture Behavioral of DataPath is

component ShiftLeft2 is
  port( -- input
        input : in std_logic_vector(31 downto 0);

        -- output
        output: out std_logic_vector(31 downto 0) );
end component;

component SignExtend is
  port( -- input
        input  : in std_logic_vector(15 downto 0);

        -- output
        output : out std_logic_vector(31 downto 0) );
end component;

component Mux is
  generic(n: integer);
  port( -- input
        input_1    : in std_logic_vector(n - 1 downto 0);
        input_2    : in std_logic_vector(n - 1 downto 0);
        mux_select : in std_logic;

        -- output
        output     : out std_logic_vector(n - 1 downto 0) );
end component;

component DataMemory is
  port( -- inputs
        CLK            : in std_logic;
        reset_neg      : in std_logic;
        memory_address : in std_logic_vector(n - 1 downto 0);
        MemWrite       : in std_logic;
        MemRead        : in std_logic;
        data_in        : in std_logic_vector(n - 1 downto 0);

        -- output
        data_out       : out std_logic_vector(n - 1 downto 0) );
end component;
```

```
component Registers is
  port( -- input
        CLK          : in std_logic;
        reset_neg    : in std_logic;
        address_in_1 : in std_logic_vector(4 downto 0);
        address_in_2 : in std_logic_vector(4 downto 0);
        address_out  : in std_logic_vector(4 downto 0);

        write_data   : in std_logic_vector(n - 1 downto 0);
        RegWrite     : in std_logic;  -- signal control

        -- output
        register_1   : out std_logic_vector(n - 1 downto 0);
        register_2   : out std_logic_vector(n - 1 downto 0) );
end component;

component ProgramCounter is
port( -- input
      CLK        : in  std_logic;
      reset_neg  : in  std_logic;
      input      : in  std_logic_vector(31 downto 0);

      -- output
      output     : out std_logic_vector(31 downto 0) );
end component;

component Adder is
  port( -- input
        operand_1 : in  std_logic_vector(n - 1 downto 0);
        operand_2 : in  std_logic_vector(n - 1 downto 0);

        -- output
        result    : out std_logic_vector(n - 1 downto 0) );
end component;

component ALU is
  port( -- input
        operand_1   : in std_logic_vector(n - 1 downto 0);
        operand_2   : in std_logic_vector(n - 1 downto 0);
        ALU_control : in std_logic_vector(1 downto 0);  -- 4 operations

        -- output
        result      : out std_logic_vector(n - 1 downto 0);
        zero        : out std_logic );
end component;
```

```
constant PC_increment     : std_logic_vector(31 downto 0) := "00000000000000000000000000000001";
signal PC_out             : std_logic_vector(31 downto 0);
signal MuxToWriteRegister : std_logic_vector(4 downto 0);
signal SignExtendToSLL    : std_logic_vector(31 downto 0);
signal SLLToAdder         : std_logic_vector(31 downto 0);
signal ReadData1ToALU     : std_logic_vector(n-1 downto 0);
signal ReadData2ToMux     : std_logic_vector(n-1 downto 0);
signal MuxToALU           : std_logic_vector(n-1 downto 0);
signal ALUToDataMemory    : std_logic_vector(n-1 downto 0);
signal DataMemoryToMux    : std_logic_vector(n-1 downto 0);
signal MuxToWriteData     : std_logic_vector(n-1 downto 0);
signal AdderToMux         : std_logic_vector(31 downto 0);
signal MuxToPC            : std_logic_vector(31 downto 0);
signal Adder1ToMux        : std_logic_vector(31 downto 0);
signal SLLToMux           : std_logic_vector(31 downto 0);
signal SLLOut             : std_logic_vector(31 downto 0);
```

```
begin

  SLLToMux  <= Adder1ToMux(31 downto 28) & SLLOut(27 downto 0);


  Memory        : DataMemory     port map(CLK, reset_neg, ALUToDataMemory, MemWrite, MemRead, ReadData2ToMux, DataMemoryToMux);
  ALogicUnit    : ALU            port map(ReadData1ToALU, MuxToALU, ALUOp, ALUToDataMemory, ZeroCarry);

  MuxAlu        : Mux            generic map(32) port map(ReadData2ToMux, SignExtendToSLL, ALUSrc, MuxToALU);
  MuxReg        : Mux            generic map(5)  port map(instruction(20 downto 16), instruction(15 downto 11), RegDst, MuxToWriteRegister);
  MuxMem        : Mux            generic map(32) port map(ALUToDataMemory, DataMemoryToMux, MemToReg, MuxToWriteData);
  MuxBranch     : Mux            generic map(32) port map(Adder1ToMux, AdderToMux, Branch, MuxToPC);
  AdderPC       : Adder          port map(PC_out, PC_increment, Adder1ToMux);
  AdderBranch   : Adder          port map(Adder1ToMux, SLLToAdder, AdderToMux);
  ShifterBranch : ShiftLeft2     port map(SignExtendToSLL, SLLToAdder);
  ShiftExtend   : SignExtend     port map(instruction(15 downto 0), SignExtendToSLL);
  PC            : ProgramCounter port map(CLK, reset_neg, MuxToPC, PC_out);
  Registers1    : Registers      port map(CLK, reset_neg, instruction(25 downto 21), instruction(20 downto 16),
                                          MuxToWriteRegister, MuxToWriteData, RegWrite, ReadData1ToALU, ReadData2ToMux);

  next_instruction <= PC_out;
end Behavioral;
```

Mips.

Τέλος, παρουσιάζουμε τον Mips έχοντας ενώσει όλα τα components στα οποία προαναφερθήκαμε.

```
entity Mips is
    GENERIC (n : integer := 32);
    port( CLK, reset_neg : in std_logic );
end Mips;
```

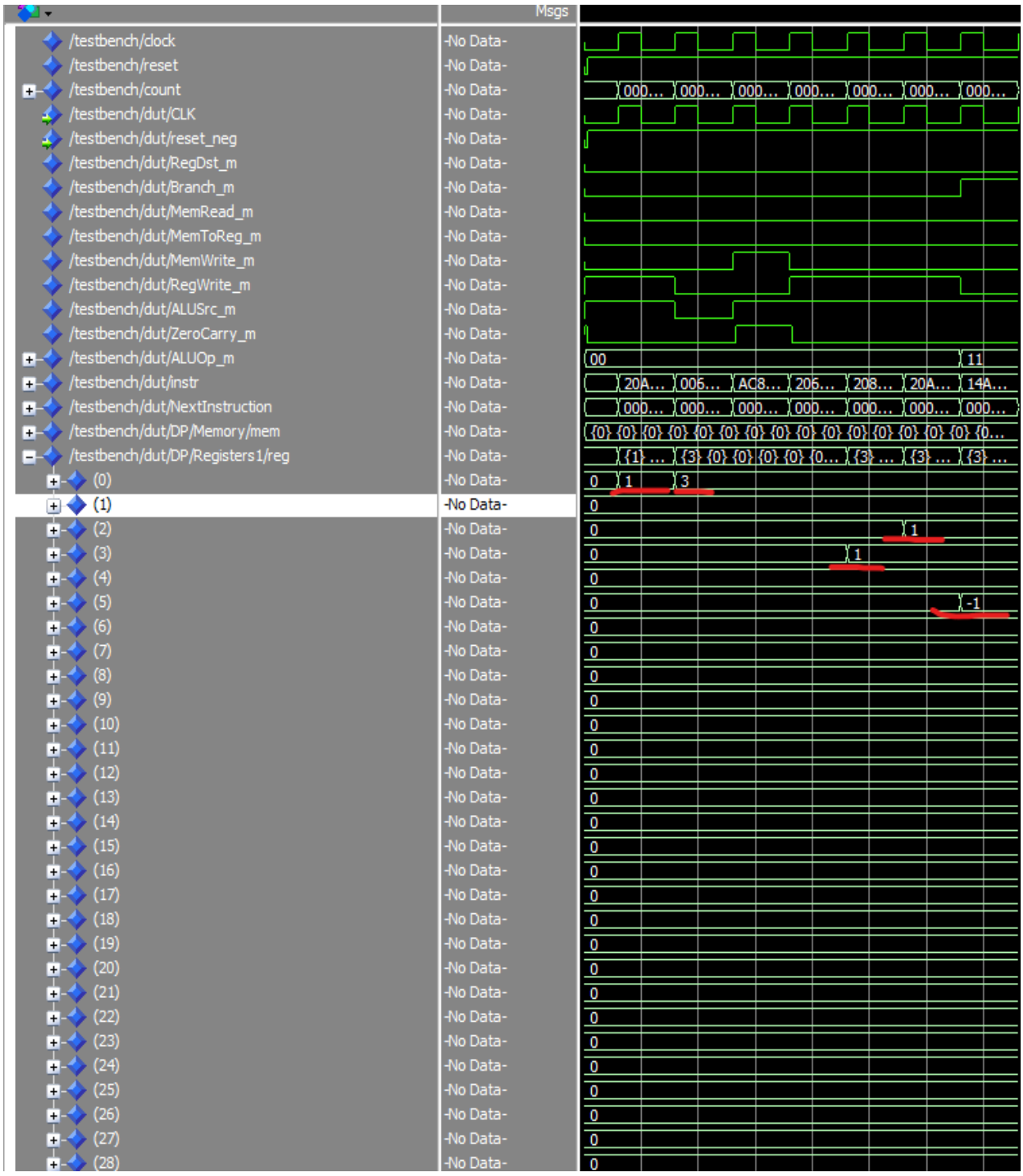
```
architecture Behavioral of Mips is

    component ControlUnit is
    port( -- input
        instruction : in std_logic_vector(31 downto 0);
        ZeroCarry   : in std_logic;

        -- output (control signals)
        RegDst      : out std_logic;
        Branch      : out std_logic;
        MemRead     : out std_logic;
        MemToReg    : out std_logic;
        ALUOp       : out std_logic_vector (1 downto 0);  -- 4 operations
        MemWrite    : out std_logic;
        ALUSrc      : out std_logic;
        RegWrite    : out std_logic );
    end component;

    component DataPath is
    port( -- inputs
    CLK, reset_neg    : in std_logic;
    instruction       : in std_logic_vector(31 downto 0);
    -- control signals
    RegDst            : in std_logic;
    Branch            : in std_logic;
    MemRead           : in std_logic;
    MemToReg          : in std_logic;
    ALUOp             : in std_logic_vector(1 downto 0);
    MemWrite          : in std_logic;
    ALUSrc            : in std_logic;
    RegWrite          : in std_logic;

    -- outputs
    next_instruction  : out std_logic_vector(31 downto 0);
    ZeroCarry         : out std_logic );
    end component;
```

```
component InstructionMemory is
port( -- input
register_addr : in  std_logic_vector(31 downto 0);

-- output
instruction   : out std_logic_vector(31 downto 0) );
end component;

signal RegDst_m        : std_logic;
signal Branch_m        : std_logic;
signal MemRead_m       : std_logic;
signal MemToReg_m      : std_logic;
signal MemWrite_m      : std_logic;
signal RegWrite_m      : std_logic;
signal ALUSrc_m        : std_logic;
signal ZeroCarry_m     : std_logic;
signal ALUOp_m         : std_logic_vector(1 downto 0);
signal instr           : std_logic_vector(31 downto 0);
signal NextInstruction : std_logic_vector(31 downto 0);
```

```
    begin
    CU : ControlUnit  port map(
                        instr,
                        ZeroCarry_m,
                        RegDst_m,
                        Branch_m,
                        MemRead_m,
                        MemToReg_m,
                        ALUOp_m,
                        MemWrite_m,
                        ALUSrc_m,
                        RegWrite_m );

    DP : DataPath     port map(
                        CLK,
                        reset_neg,
                        instr,
                        RegDst_m,
                        Branch_m,
                        MemRead_m,
                        MemToReg_m,
                        ALUOp_m,
                        MemWrite_m,
                        ALUSrc_m,
                        RegWrite_m,
                        NextInstruction,
                        ZeroCarry_m );


    I  : InstructionMemory  port map( NextInstruction, instr );

end Behavioral;
```

Test Bench και Εκτέλεση.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity TestBench is
end TestBench;

architecture Behavioral of TestBench is

   -- constants
   constant CLK_low    : time := 12 ns;
   constant CLK_high   : time := 8 ns;
   constant CLK_period : time := CLK_low + CLK_high;
   constant ResetTime  : time := 1 ns;

   -- dut signals
   signal clock , reset : std_logic;
   signal count         : std_logic_vector(10 downto 0):="00000000000";

component Mips is
   port( CLK, reset_neg  : in std_logic );
end component;

begin
   dut : Mips port map ( clock, reset );

   -- reset
   reset_process : process
   begin
     reset <= '0';
     wait for ResetTime;
     reset <= '1';
     wait;
   end process reset_process;

   clock_process : process
   begin
     if (clock = '0') then
       clock <= '1';
       wait for CLK_high;
     else
       clock <= '0';
       wait for CLK_low;
       count <= std_logic_vector(unsigned(count) + 1);
     end if;
   end process clock_process;
end Behavioral;
```



Οι υπογραμμισμένες με κόκκινο τιμές είναι οι τιμές των καταχωρητών 0,2,3 και 5 αντίστοιχα.

Συμπέρασμα.

Ως αποτέλεσμα αυτής της άσκησης έχουμε σχεδιάσει μια πολύ απλή CPU μονού κύκλου βασισμένη στην αρχιτεκτονική MIPS. Για την επίτευξη αυτού του στόχου, εφαρμόσαμε τις γνώσεις που αποκτήσαμε καθ' όλη τη διάρκεια του εξαμήνου σχετικά με το σχεδιασμό και την ανάπτυξη συνδυαστικών και διαδοχικών ψηφιακών κυκλωμάτων. Η κατανόηση της λειτουργίας κάθε μεμονωμένου στοιχείου στην CPU, και του τρόπου με τον οποίο σχετίζονται με πολλαπλές διαδρομές δεδομένων, ήταν ένα βασικό συστατικό που μας επέτρεψε να ολοκληρώσουμε αυτήν την εργασία. Η χρήση της γλώσσας VHDL και των εργαλείων σχεδίασης με τη βοήθεια υπολογιστή, όπως το ModelSim , απλοποίησαν σημαντικά την εργασία. Στην πορεία, αντιμετωπίσαμε πολλά διλήμματα, όπου χρειάστηκε να συμβιβαστούμε σε ορισμένες πτυχές της εφαρμογής μας για να προχωρήσουμε. Οι αποφάσεις μας, ωστόσο, ήταν κατάλληλες και μας επέτρεψαν να ολοκληρώσουμε ένα σχέδιο CPU.