Εργασία Ψηφιακά Συστήματα HW-1

ELENI SOURLI

Το παρόν αρχείο αποτελεί ηλεκτρονική αναφορά της εργασίας που ανατέθηκε στο μάθημα "Ψηφιακά Συστήματα HW-1". Στόχος της είναι η εκμάθηση της διαδικασίας σχεδιασμού ενός διαγράμματος υψηλού επιπέδου του επεξεργαστή RISC-V καθώς και ενός κυκλώματος αριθμομηχανής. Απαρτίζεται από πέντε διαφορετικές ασκήσεις άμεσα συνδεδεμένες μεταξύ τους. Συμπεριλαμβάνονται επεξηγήσεις και κυματομορφές προσομοίωσης για τη βέλτιστη περιγραφή των βημάτων που πραγματοποιήθηκαν.

Η εργασία πραγματοποιήθηκε με το διαδικτυακό ολοκληρωμένο περιβάλλον ανάπτυξης (web IDE)EDA playground.

Άσκηση 1

Ζητείται η υλοποίηση μιας αριθμητικής/λογικής μονάδας (Arithmetic Logic Unit (ALU)), η οποία είναι ένα σημαντικό στοιχείο κάθε συστήματος επεξεργαστή RISC-V. Μέσω αυτής της μονάδας εκτελούνται τόσο αριθμητικές λειτουργίες όσο και λογικές λειτουργίες. Η χρησιμότητα της γίνεται φανερή και στον επεξεργαστή RISC-V στη συνέχεια.

Η ALU που σχεδιάστηε ,υλοποιεί τις ακόλουθες πράξεις: προσημασμένη πρόσθεση, προσημασμένη αφαίρεση, λογικό AND, λογικό OR, λογικό XOR, σύγκριση "Μικρότερο από" και τρεις διαφορετικές πράξεις ολίσθησης. ι είσοδοι έχουν πλάτος 32-bit και παράγεται μια έξοδος πλάτους 32-bit.

Υλοποιείται το module alu στο οποίο ορίζονται οι ζητούμενες πράξεις. Δηλώνονται οι τιμές της alu_op που αντιστοιχούν σε κάθε λειτουργία προς ευκολία του χρήστη και καλύτερη κατανόηση.(εντολή parameter [3:0] της Verilog). Οι λειτουργίες λογική AND, λογική OR, πρόσθεση, αφαίρεση και λογική XOR πραγματοποιούνται μέσω των ‘ &, |, +, -, ^ ‘ ανάμεσα στις εισόδους op1 , op2 αντίστοιχα . Οι λειτουργίες λογική ολίσθηση δεξιά κατά op2 bits και λογική ολίσθηση αριστερά κατά op2 bits πραγματοποιείται μέσω των εντολών ‘ >> op2[4:0], << op2[4:0] ‘ αντίστοιχα. Μέσω του op2[4:0] δηλώνεται ότι η ολίσθηση θα γίνει κατά τον αριθμό που προκύπτει από τα πέντε χαμηλότερα ψηφία του op2. Η λειτουργία μικρότερο από πραγματοποιείται με το’<’ το οποίο εφαρμόζεται σε προσημασμένους αριθμούς σύμφωνα με τις προδιαγραφές (εντολή $signed της Verilog). Τέλος , η λειτουργία αριθμητική ολίσθηση δεξιά κατά op2 bits [πραγματοποιείται με το ‘ >>>’. Απαιτεί την μετατροπή του op1 σε προσημασμένο(εντολή $signed της Verilog) . Το αποτέλεσμα που παράγεται είναι προσημασμένο και μετατρέπεται σε μη προσημασμένο (εντολή $unsigned της Verilog).

Το αποτέλεσμα result λαμβάνει την κατάλληλη τιμή μέσω ενός πολυπλέκτη που αποφασίζει ποια λειτουργία εκτελείται κάθε φορά .

Ο κώδικας στη verilog είναι ο ξής :

```
`timescale 1ns/1ps
module alu
#(parameter [3:0] ALUOP_AND = 4'b0000,
 parameter [3:0] ALUOP_OR = 4'b0001,
 parameter [3:0] ALUOP_SUM = 4'b0010,
 parameter [3:0] ALUOP_SUB = 4'b0110,
 parameter [3:0] ALUOP_LESSTHAN = 4'b0111,
 parameter [3:0] ALUOP_LOGSHIFTR = 4'b1000,
 parameter [3:0] ALUOP_LOGSHIFTL = 4'b1001,
 parameter [3:0] ALUOP_SHIFTR = 4'b1010,
 parameter [3:0] ALUOP_XOR = 4'b1101)

(output reg [31:0] result,
 output reg zero,
 input wire [31:0] op1,
 input wire [31:0] op2,
 input wire [3:0] alu_op);

  reg [31:0] m0, m1, m2, m3, m4, m5, m6, m7, m8;

 always @(*)
 begin
   m0 = op1 & op2;
   m1 = op1 | op2;
   m2 = op1 + op2;
   m3 = op1 - op2;
   m4 = ($signed(op1) < $signed(op2));
   m5 = op1 >> op2[4:0];
   m6 = op1 << op2[4:0];
   m7 =  $unsigned($signed( op1) >>> op2[4:0]);
   m8 = (op1 ^ op2);

   result = (alu_op == ALUOP_AND) ? m0 ://AND
   (alu_op == ALUOP_OR) ? m1 ://OR
   (alu_op == ALUOP_SUM) ? m2 ://SUM
   (alu_op == ALUOP_SUB) ? m3 ://SUB
   (alu_op == ALUOP_LESSTHAN) ? m4 ://LESS THAN
   (alu_op == ALUOP_LOGSHIFTR) ? m5 ://LOGICAL SHIFT RIGHT
   (alu_op == ALUOP_LOGSHIFTL) ? m6 ://LOGICAL SHIFT LEFT
   (alu_op == ALUOP_SHIFTR) ? m7 ://SHIFT RIGHT
   (alu_op == ALUOP_XOR) ? m8 ://XOR
```

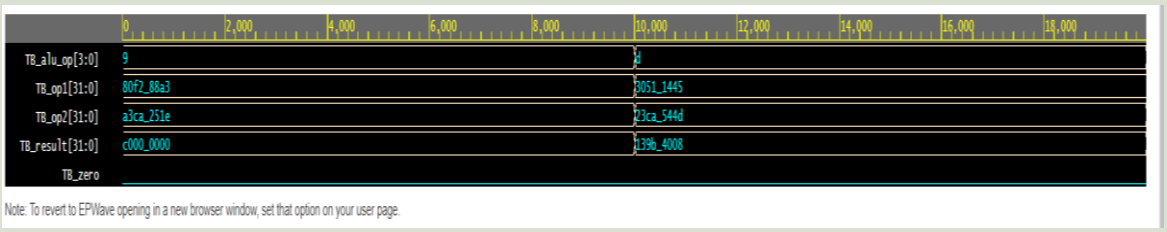
`

```
  parameter [3:0] ALUOP_LESSTHAN = 4'b0111,
  parameter [3:0] ALUOP_LOGSHIFTR = 4'b1000,
  parameter [3:0] ALUOP_LOGSHIFTL = 4'b1001,
  parameter [3:0] ALUOP_SHIFTR = 4'b1010,
  parameter [3:0] ALUOP_XOR = 4'b1101)

 (output reg [31:0] result,
  output reg zero,
  input wire [31:0] op1,
  input wire [31:0] op2,
  input wire [3:0] alu_op);

   reg [31:0] m0, m1, m2, m3, m4, m5, m6, m7, m8;

  always @(*)
  begin
    m0 = op1 & op2;
    m1 = op1 | op2;
    m2 = op1 + op2;
    m3 = op1 - op2;
    m4 = ($signed(op1) < $signed(op2));
    m5 = op1 >> op2[4:0];
    m6 = op1 << op2[4:0];
    m7 =  $unsigned($signed( op1) >>> op2[4:0]);
    m8 = (op1 ^ op2);

    result = (alu_op == ALUOP_AND) ? m0 ://AND
    (alu_op == ALUOP_OR) ? m1 ://OR
    (alu_op == ALUOP_SUM) ? m2 ://SUM
    (alu_op == ALUOP_SUB) ? m3 ://SUB
    (alu_op == ALUOP_LESSTHAN) ? m4 ://LESS THAN
    (alu_op == ALUOP_LOGSHIFTR) ? m5 ://LOGICAL SHIFT RIGHT
    (alu_op == ALUOP_LOGSHIFTL) ? m6 ://LOGICAL SHIFT LEFT
    (alu_op == ALUOP_SHIFTR) ? m7 ://SHIFT RIGHT
    (alu_op == ALUOP_XOR) ? m8 ://XOR
     32'b00000000000000000000000000000000;
    zero = (result == 32'b00000000000000000000000000000000) ? 1'b1 : 1'b0;
  end
endmodule
```

Για την επαλήθευση του module υλοποιήθηκε το ακόλουθο testbench:

```
1  `timescale 1ns/1ps
2  module TB_alu;
3    reg [31:0] TB_op1,TB_op2;
4    reg [3:0]TB_alu_op;
5    wire [31:0]  TB_result;
6    wire TB_zero;
7    alu DUT (.result(TB_result),.zero(TB_zero),
8             .op1(TB_op1),.op2(TB_op2),.alu_op(TB_alu_op));
9      initial
10     begin
11       $dumpfile("dump.vcd"); $dumpvars;
12
13       TB_op1=32'b 10000000111100101000100010100011;
14       TB_op2=32'b 10100011110010100010010100011110;
15       TB_alu_op=4'b1001;
16     #10
17       TB_op1=32'b00110000010100010001010001000101;
18       TB_op2=32'b00100011110010100101010001001101;
19       TB_alu_op=4'b1101;
20       #10TB_op1=32'b00110000010100010001010001000101;
21      TB_op2=32'b 10100011110010100010010100011110;
22       TB_alu_op=4'b1001;
23
24     end
25 endmodule
```

Παρήχθησαν τα ακόλουθα αποτελέσματα τα οποία ήταν και τα αναμενόμενα για τις πράξεις που έγιναν:

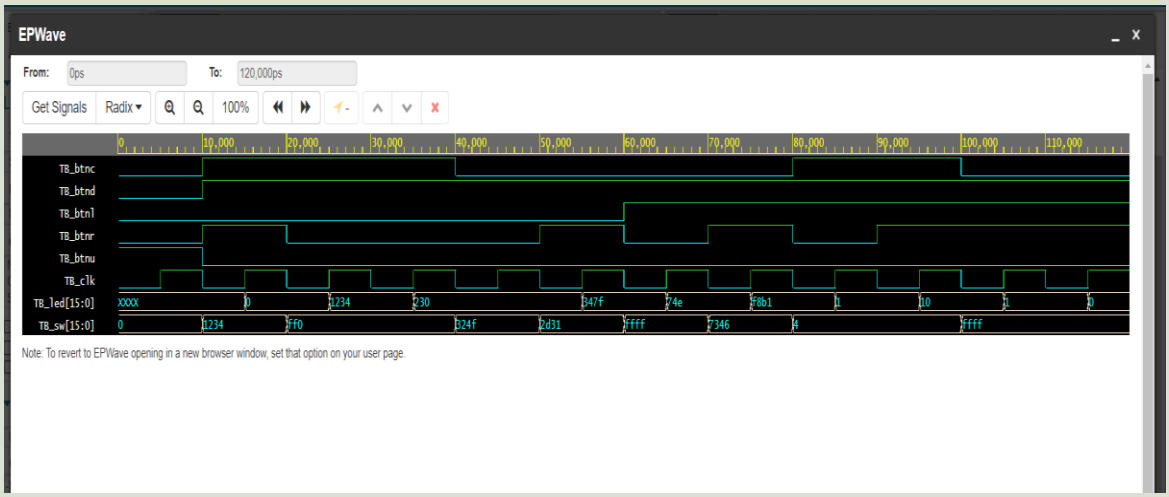


Άσκηση 2

Ζητείται ο σχεδιασμός ενός κυκλώματος αριθμομηχανής, η οποία θα διατηρεί την τιμής της σε έναν συσσωρευτή 16-bit καταχωρητή. Δίνεται η δυνατότητα στο χρήστη να ενημερώνει την τιμή υλοποιώντας οποιαδήποτε από τις αριθμητικές και λογικές συναρτήσεις που περιέχονται στην ALU που δημιουργήθηκε νωρίτερα. Προκειμένου να καθοριστεί η λειτουργία που θα εκτελέσει η αριθμομηχανή δημιουργείται ένα module decoder το οποίο εμπεριέχει συνδυασμούς των πλήκτρων (κεντρικό, δεξί, αριστερό) χρησιμοποιώντας πρότυπες πύλες της verilog. Ανάλογα με τις τιμές του προκύπτει διαφορετικό alu_op κάθε φορά. Κάθε ένα από τα bits της alu_op παράγεται ξεχωριστά μέσα από ένα διαφορετικό κύκλωμα πυλών. Η λογική σχεδιασμού που ακολουθείται είναι ότι εισάγονται δύο τιμές στην ALU και εξάγεται ένα αποτέλεσμα το οποίο αποθηκεύεται στον καταχωρητή . Στη συνέχεια, συνδέεται στην έξοδο LED και αφού υποστεί επέκταση προσήμου (γίνεται 32-bit) οδηγείται στην είσοδο 1 της ALU. Η είσοδος 2 δίνεται από το χρήστη. Αξιοσημείωτο είναι ότι η τιμή του συσσωρευτή ενημερώνεται ή μηδενίζεται σύμφωνα με την τιμή των btnd ,btnu αντίστοιχα σε κάθε ακμή ρολογιού. Επέκταση πρόσημού εφαρμόζεται και στην είσοδο από τον χρήστη διότι η ALU δέχεται ορίσματα 32-bit.

Στο testbench δηλώνονται οι δοσμένες τιμές για τα πλήκτρα και το SW. Σε ένα always ορίζεται η αλλαγή του clk κάθε πέντε μονάδες του χρόνου, ώστε να μεταβάλλονται οι τιμές και να ανανεώνεται ο accumulator.

Μετά την υλοποίηση του testbench που ζητείται προκύπτουν τα ακόλουθα αποτελέσματα, τα οποία ταυτίζονται με τα δοσμένα .



Φαίνεται ότι αρχικά είναι ενεργοποιημένο το btnu και απενεργοποιημένο το btnd οπότε το αναμενόμενο αποτέλεσμα είναι μηδέν. Έπειτα, απενεργοποιείται το btnu και ενεργοποιείται το btnd και ο συσσωρευτής ξεκινά να λαμβάνει τιμές. Οι τιμές του αναμένομενου αποτελέσματος (TB_led) ταυτίζονται με τις δοσμένες γεγονός που μας αποδεικνύει ότι υλοποιήθηκε σωστά το module calc.

Ο κώδικας στη verilog είναι ο ξής :

SV/Verilog Design

```
`timescale 1ns/1ps
`include"alu.v"
`include"decoder.v"

module calc
  (output reg [15:0] led,
   input wire clk,btnc,btnl,btnu,btnr,btnd,
   input wire [15:0] sw);

  reg  [15:0] accumulator;
  wire [3:0] new_alu_op;
  wire [31:0] extended_accumulator;
  wire [31:0] extended_sw;
  wire zero;
  wire [31:0] result;


  assign extended_accumulator ={{16{accumulator[15]}}, accumulator};
  assign extended_sw = {{16{sw[15]}}, sw};

  alu ALU_CALC (.result(result),.zero(zero),.op1(extended_accumulator),
              .op2(extended_sw),.alu_op(new_alu_op));

  decoder DECODER_CALC(.alu_op(new_alu_op),
                    .btnc(btnc),.btnl(btnl),.btnr(btnr));

  always @(posedge clk)
    begin
      if(btnu)//the accumulator gets the value 0
        accumulator <=16'b0;
      else if (btnd)

        accumulator <= result[15:0]; //the accumulator gets the value of alu's
rsult

      led = accumulator;

    end

endmodule

`
```

Ο κώδικας του decoder στη verilog είναι ο ξής :

```
1  `timescale 1ns/1ps
2
3  module decoder
4    (output wire[3:0]alu_op,
5     input wire btnc,btnl,btnr);
6
7    wire m0,m1,m2,m3,m5,m6,m7,m8,m10,m11,m12,m13,m15,m16,m17,m18;
8
9    //alu_op[0]
10
11   not U0(m0,btnr);
12   and U1(m1,m0,btnl);
13   xor U2(m2,btnl,btnc);
14   and U3(m3,m2,btnr);
15   or U4(alu_op[0],m3,m1);
16
17   //alu_op[1]
18
19   and U5(m5,btnl,btnr);
20   not U6(m6,btnl);
21   not U7(m7,btnc);
22   and U8(m8,m6,m7);
23   or U9(alu_op[1],m5,m8);
24
25   //alu_op[2]
26
27   and U10(m10,btnr,btnl);
28   xor U11(m11,btnr,btnl);
29   or U12(m12,m10,m11);
30   not U13(m13,btnc);
31   and U14(alu_op[2],m12,m13);
32
33   //alu_op[3]
34
35   not U15(m15,btnr);
36   and U16(m16,m15,btnc);
37   xnor U17(m17,btnr,btnc);
38   or U18(m18,m16,m17);
39   and U19(alu_op[3],m18,btnl);
40 endmodule
41
```

ο κώδικας του testbench:

```
`timescale 1ns/1ps

module TB_calc;

  reg TB_clk, TB_btnc, TB_btnl, TB_btnu, TB_btnr, TB_btnd;
  reg [15:0] TB_sw;
  wire [15:0] TB_led;


  calc DUT (
    .led(TB_led),
    .clk(TB_clk),
    .btnc(TB_btnc),
    .btnl(TB_btnl),
    .btnu(TB_btnu),
    .btnr(TB_btnr),
    .btnd(TB_btnd),
    .sw(TB_sw)
  );

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, TB_calc);


    TB_clk = 0;
    TB_btnc = 0;
    TB_btnl = 0;
    TB_btnu = 1;
    TB_btnr = 0;
    TB_btnd = 0;
    TB_sw = 16'h0000;
    #120;


    $finish;
  end

  always #5 TB_clk = ~TB_clk;


  initial begin
    #10;
```

```
    TB_btnu = 0;
    TB_btnd = 1;

    TB_btnl = 0;
    TB_btnc = 1;
    TB_btnr = 1;
    TB_sw = 16'h1234;
    #20;
  end

  initial
    begin
    #20;
    TB_btnu = 0;
    TB_btnd = 1;

    TB_btnl = 0;
    TB_btnc = 1;
    TB_btnr = 0;
    TB_sw = 16'h0ff0;
    #40;
  end

  initial
    begin
    #40;
    TB_btnu = 0;
    TB_btnd = 1;

    TB_btnl = 0;
    TB_btnc = 0 ;
    TB_btnr = 0;
    TB_sw = 16'h324f;
    #50;
  end

   initial
    begin
    #50;
    TB_btnu = 0;
    TB_btnd = 1;

    TB_btnl = 0;
```

```
85
86      TB_btnl = 0;
87      TB_btnc = 0 ;
88      TB_btnr = 1;
89      TB_sw = 16'h2d31;
90      #60;
91    end
92
93    initial
94      begin
95      #60;
96      TB_btnu = 0;
97      TB_btnd = 1;
98
99      TB_btnl = 1;
100     TB_btnc = 0 ;
101     TB_btnr = 0;
102     TB_sw = 16'hffff;
103     #70;
104   end
105
106   initial
107     begin
108     #70;
109     TB_btnu = 0;
110     TB_btnd = 1;
111
112     TB_btnl = 1;
113     TB_btnc = 0 ;
114     TB_btnr = 1;
115     TB_sw = 16'h7346 ;
116     #80;
117   end
118
119    initial
120     begin
121     #80;
122     TB_btnu = 0;
123     TB_btnd = 1;
124
125     TB_btnl = 1;
126     TB_btnc = 1 ;
127     TB_btnr = 0;
```

```
85
86      TB_btnl = 0;
87      TB_btnc = 0 ;
88      TB_btnr = 1;
89      TB_sw = 16'h2d31;
90      #60;
91    end
92
93    initial
94      begin
95      #60;
96      TB_btnu = 0;
97      TB_btnd = 1;
98
99      TB_btnl = 1;
100     TB_btnc = 0 ;
101     TB_btnr = 0;
102     TB_sw = 16'hffff;
103     #70;
104   end
105
106   initial
107     begin
108     #70;
109     TB_btnu = 0;
110     TB_btnd = 1;
111
112     TB_btnl = 1;
113     TB_btnc = 0 ;
114     TB_btnr = 1;
115     TB_sw = 16'h7346 ;
116     #80;
117   end
118
119    initial
120     begin
121     #80;
122     TB_btnu = 0;
123     TB_btnd = 1;
124
125     TB_btnl = 1;
126     TB_btnc = 1 ;
127     TB_btnr = 0;
```

```
    TB_btnl = 1;
    TB_btnc = 1 ;
    TB_btnr = 0;
    TB_sw = 16'h0004 ;
    #90;
  end

  initial
    begin
    #90;
    TB_btnu = 0;

    TB_btnd = 1;

    TB_btnl = 1;

    TB_btnc = 1 ;
    TB_btnr = 1;
    TB_sw = 16'h0004 ;
    #100;
  end

  initial
    begin
    #100;
    TB_btnu = 0;
    TB_btnd = 1;

    TB_btnl = 1;
    TB_btnc = 0 ;
    TB_btnr = 1;
    TB_sw = 16'hffff ;
    #120;
  end




endmodule
```

Άσκηση 3

Ζητείται η δημιουργία ενός αρχείου καταχωρητών στο οποίο αποθηκεύονται οι τιμές των καταχωρητών που χρησιμοποιούνται από τον επεξεργαστή RISC-V. Το αρχείο αποτελείται από 32 καταχωρητές των 32-bit ο καθένας. Οι καταχωρητές αρχικοποιούνται με μηδενικά μέσω ενός for και έπειτα λαμβάνουν τιμές. Οι τιμές των εισόδων readReg1,readReg2 υποδεικνύουν τη θέση του καταχωρητή από την οποία πρέπει να γίνει η ανάγνωση δεδομενων. Όταν το σήμα write είναι ενεργοποιημένο εγγράφονται τα δεδομένα της εισόδου στην αντίστοιχη διεύθυνση που υποδεικνύεται από το χρήστη. Σε περίπτωση που η διεύθυνση που δίνεται από τον χρήστη ταυτίζεται με κάποια διεύθυνση θύρας ανάγνωσης τότε τα δεδομένα καταχωρούνται στη θέση των δεδομένων ανάγνωσης(readData1 ή readData2 αντίστοιχα).

Ο κώδικας στη verilog είναι ο ξής :

```
`timescale 1ns/1ps
module regfile
(output reg [31:0] readData1 ,readData2,
   input wire clk,write,
   input wire [4:0] readReg1,readReg2,writeReg,
   input wire [31:0] writeData);


  reg [31:0] register [31:0];
  integer i;

  initial
    begin

      for (i=0;i<32;i=i+1)
    register[i] = 0;
    end

  always@(posedge clk)
    begin
      readData1 <= register[readReg1];
      readData2 <= register[readReg2];
      if (write==1'b1 && writeReg!=5'b00000)
      begin
      register[writeReg] <= writeData;
      if (readReg1 == writeReg)
          readData1 <= writeData;
      if (readReg2 == writeReg)
         readData2 <= writeData;
      end
   end

endmodule
```

Άσκηση 4

Ζητείται η υλοποίηση της διαδρομής δεδομένων του υψηλού επιπέδου του επεξεργαστή RISC-V. Αποτελείται από τις εσωτερικές λειτουργικές μονάδες, τους καταχωρητές και τους πολυπλέκτες που χρησιμοποιούνται για την υλοποίηση μεμονωμένων εντολών. Το αρχείο που δημιουργείται περιλαμβάνει την ALU και το αρχείο καταχωρητών που δημιουργήθηκαν νωρίτερα. Πραγματοποιείται αποκωδικοποίηση των εντολών του RISC-V (R, I, S, B types).Το module datapath επιτυγχάνει την αναγνώριση του τύπου των εντολών και του πεδίου εντολής, ώστε να προβεί στην κατάλληλη ρύθμιση των τελεστών της ALU και την ενεργοποίηση των σημάτων ελέγχου όπου κρίνεται απαραίτητο . Γίνεται αντιστοίχιση των θυρών του datapath με τις θύρες των module alu , regfile είτε απευθείας μέσω των εισόδων και των εξόδων της διαδρομής δεδομένων είτε μέσω εσωτερικών καλωδίων που ορίζονται . Συντάσσονται οι άμεσες τιμές για τους διάφορους τύπους εντολών με χρήση των bits της λέξης εντολής, όπως ορίζεται στον πίνακα του pdf που δίνεται . Η άμεση τιμή της εντολής I type περιλαμβάνει τα 12 MSB της λέξης εντολής([31:20]) . Η άμεση τιμή της εντολής S type περιλαμβάνει τα 25 έως 31 bits και τα 7 έως 11 bits της λέξης εντολής με το MSB να είναι το 31ο bit([31:25],[11:7]) . Η άμεση τιμή της εντολής B type περιλαμβάνει το 31ο bit, το 7ο bit ,τα 25 έως 30 bits και τα 8 έως 11 bits της λέξης εντολής και τέλος ένα μηδενικό με το MSB να είναι το 31ο bit([31],[7],[30:25],[11:8],0). Οι άμεσες αυτές τιμές που ορίστηκαν πρέπει να υποστούν επέκταση πρόσημου προσθέτοντας 20 φορές το 31ο bit για να γίνουν λέξεις των 32-bit και να μπορούν να χρησιμοποιηθούν από την ALU. Σε περίπτωση που πραγματοποιείται διακλάδωση (εντολή BEQ) υπολογίζεται το "branch offset" ,το οποίο ισούται με την επέκταση της άμεσης τιμής της εντολής B type μετατοπισμένη αριστερά κατά 1. Υλοποιείται ένας πολυπλέκτης ο οποίος αποφασίζει ποια άμεση τιμή θα επιλεχθεί σύμφωνα με το δοσμένο opcode (τα πρώτα 7 bits της λέξης εντολής). Η διαδρομή δεδομένων επαναφέρεται με το σύγχρονο σήμα rst και λαμβάνει την προεπιλεγμένη τιμή INITIAL_PC="0x00400000".Ενημερώνεται όταν το σήμα ελέγχου loadPC είναι ενεργοποιημένο , λαμβάνοντας είτε την τιμή που είχε αυξημένη κατά 4 είτε την τιμή που είχε αθροισμένη με το branch offset. Η επιλογή αυτή καθορίζεται από το σήμα ελέγχου PCSrc , το οποίο όταν είναι ενεργοποιημένο υποδεικνύει το άθροισμα του PC με το branch offset. Τέλος, αναπτύσσεται η λογική της "εγγραφής προς τα πίσω" που αφορά την τιμή που εγγράφεται στο αρχείο των καταχωρητών. Η επιλογή της τιμής καθορίζεται από την τιμή του σήματος ελέγχου MemtoReg ,το οποίο όταν είναι ενεργοποιημένο επιλεγεί το αποτέλεσμα της ανάγνωσης μνήμης (dReadData) αλλιώς το αποτέλεσμα της ALU για τις συμβατικές αριθμητικές και λογικές εντολές(dAddress).

Ο κώδικας στη verilog είναι ο ξής :

```
`include"alu.v"
`include"regfile.v"


module datapath
   #(parameter [31:0] INITIAL_PC = 32'h00400000)

  (output reg [31:0] PC,WriteBackData,dWriteData,
   output wire [31:0] dAddress,dReadData,
   output wire Zero,
   input wire clk,rst,PCSrc,ALUSrc ,RegWrite,MemToReg,loadPC,
   input wire [31:0] instr,
   input wire [3:0] ALUCtrl);


  reg [31:0] branch_offset;

  wire [31:0] readData1;
  wire [31:0] readData2;
  reg [31:0] extended_Itype,extended_Stype,extended_Btype;
  wire [6:0] opcode = instr[6:0];
  reg [31:0] ImmGen;
  reg [31:0] k;

  regfile REG_DATAPATH(.readData1(readData1),.readData2(readData2)
              ,.clk(clk),.write(RegWrite),.readReg1(instr[19:15])
              ,.readReg2(instr[24:20]),.writeReg(instr[11:7])
                      ,.writeData(WriteBackData));

 alu ALU_DATAPATH (.op1(readData1), .op2(k), .alu_op(ALUCtrl), .zero(Zero),
.result(dAddress));




  always @(posedge clk)
    begin

        if (rst)
            PC <= INITIAL_PC;
         else
```

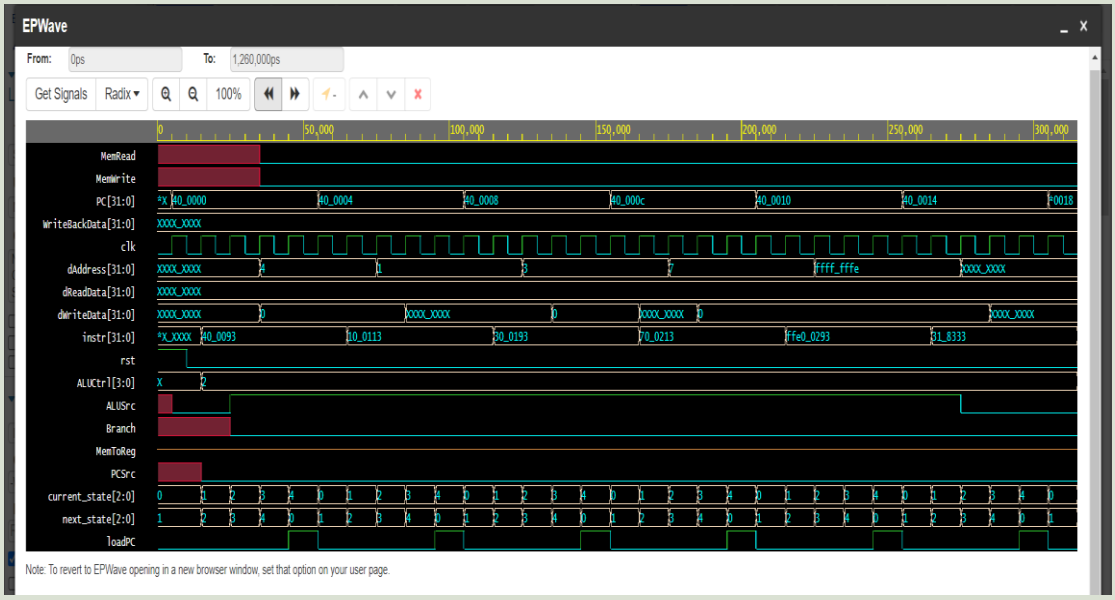
```
37
38          if (rst)
39              PC <= INITIAL_PC;
40           else
41              if (loadPC)
42                begin
43                  if (PCSrc)
44                    PC<= PC +branch_offset;
45                  else
46                    PC<=PC+4;
47                end
48         branch_offset=extended_Btype<<1;
49
50
51
52
53
54     extended_Itype={{20{instr[31]}},instr[31:20]};
55     extended_Stype={{20{instr[31]}},instr[31:25],instr[11:7]};
56     extended_Btype=
{{20{instr[31]}},instr[31],instr[7],instr[30:25],instr[11:8],1'b0};
57
58
59        ImmGen= (opcode==7'b0010011)?extended_Itype:
60       (opcode==7'b1100011)?extended_Btype:
61       (opcode==7'b0100011)?extended_Stype:
62       32'b00000000000000000000000000000000;
63
64
65
66        k=(ALUSrc==0)?readData2:ImmGen;
67
68
69
70     dWriteData =readData2;
71     WriteBackData=(MemToReg==1'b1)?dReadData:dAddress;
72      end
73
74 endmodule
```

Άσκηση 5

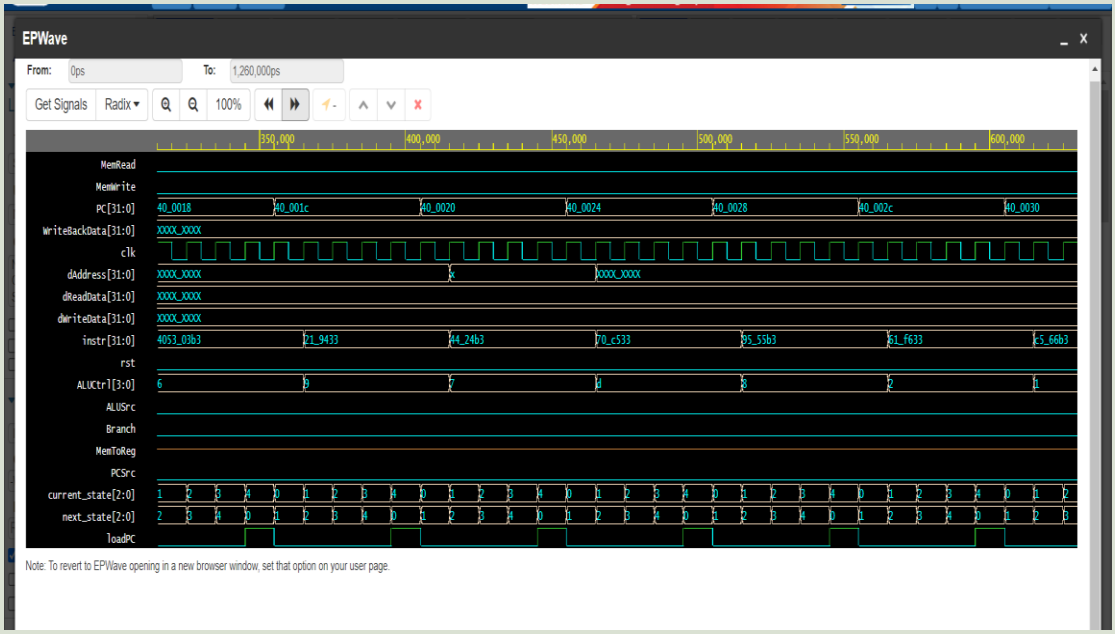
Ζητείται η δημιουργία ενός ελεγκτή πολλαπλών κύκλων που εκτελεί κάθε εντολή σε πέντε κύκλους ρολογιού. Πραγματοποιείται αντιστοίχιση των θυρών του datapath στην αντίστοιχη θύρα του multicycle module και συμπεριλαμβάνεται η παράμετρος INITIAL_PC. Σχεδιάζεται το FSM πέντε σταδίων που θεωρείται το κυριότερο στοιχείο της άσκησης. Κάθε στάδιο αντιστοιχίζεται σε μία τιμή ,η οποία δηλώνεται ως παράμετρος στον κώδικα . Τα στάδια λαμβάνουν τιμές από το 0 έως το 5 με τη σειρά που παρουσιάζονται στην εκφώνηση .Ορίζονται τρία procedural μπλοκ(always blocks) , όπως αναφέρεται στη θεωρία. Το πρώτο, το οποίο αφορά την αποθήκευση της κατάστασης, σε κάθε ακμή του ρολογιού ή του reset(rst) δίνει στην παρούσα κατάσταση είτε την τιμή του σταδίου IF (ενεργοποιημένο rst) είτε την τιμή της επόμενης κατάστασης. Το δεύτερο περιγράφει τη λογική που προσδιορίζει την επόμενη κατάσταση. Κάθε φορά που αλλάζει η τιμή της παρούσας κατάστασης η επόμενη κατάσταση λαμβάνει την τιμή της επομένης με τη σειρά π δηλώνονται στην αρχή (IF,ID,EX,MEM,WB). Εάν το σήμα ελέγχου PCSrc είναι ενεργοποιημένο και η παρούσα κατάσταση βρίσκεται στο τρίτο στάδιο η επόμενη κατάσταση θα είναι το πέμπτο στάδιο της εγγραφής νέων δεδομένων στους καταχωρητές (WB). Τέλος, το τρίτο procedural μπλοκ εμπεριέχει τη λογική που προσδιορίζει τις τιμές των εξόδων. Ενεργοποιούνται τα κατάλληλα σήματα ελέγχου ανάλογα με το στάδιο στο οποίο βρισκόμαστε συμφώνα με τις οδηγίες της εκφώνησης. Στο στάδιο ID καθορίζεται η τιμή του ALUCtrl μέσω εμφωλευμένων case ανάλογα με τις τιμές των opcode, funct3, funct7 όπως δηλώνονται στον πίνακα για τον επεξεργαστή RISCV. Η τιμή του σήματος ελέγχου ALUSrc καθορίζεται μέσω ενός procedural μπλοκ που πυροδοτείται σύγχρονα με το clk και ανάλογα με την τιμή του opcode ,δηλαδή το είδος των εντολών, λαμβάνει τις τιμές ένα και μηδέν.

Στο testbench γίνεται αντιστοίχιση των θυρών του module multicycle με τις θύρες τψν modules INSTRUCTION_MEMORY και DATA_MEMORY , από όπου και λαμβάνονται τα δεδομένα. Σε ένα always ορίζεται η αλλαγή του clk κάθε πέντε μονάδες του χρόνου ,ώστε να να λαμβάνονται νέες τιμές για τη λέξη εντολή και να λειτουργεί ο επεξεργαστής.

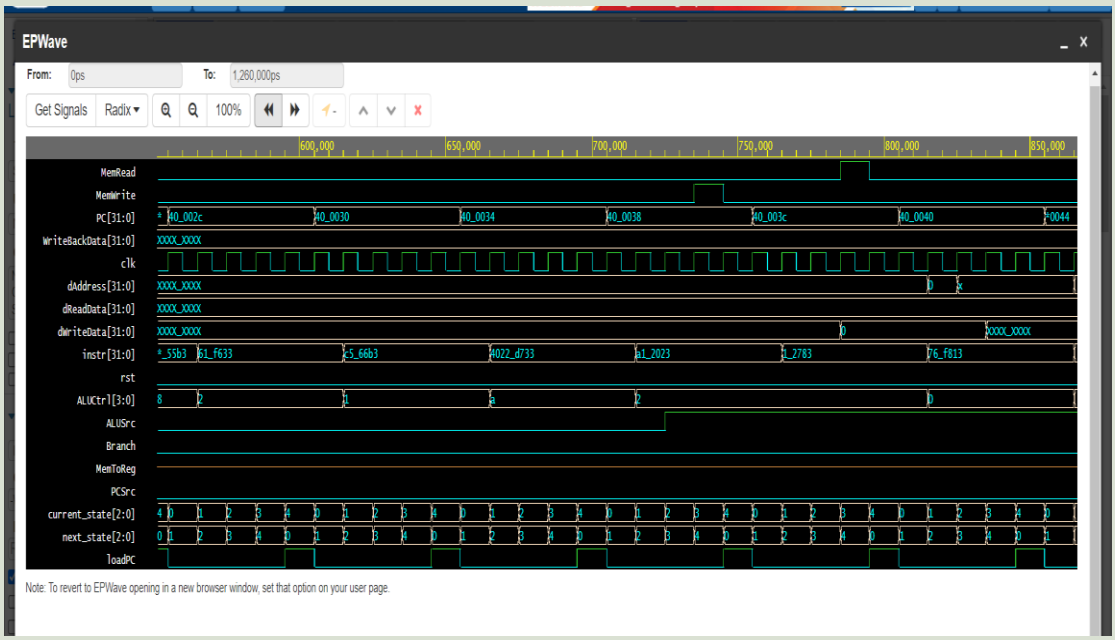
Οι προσομοιώσεις που προκύπτουν από το testbench:



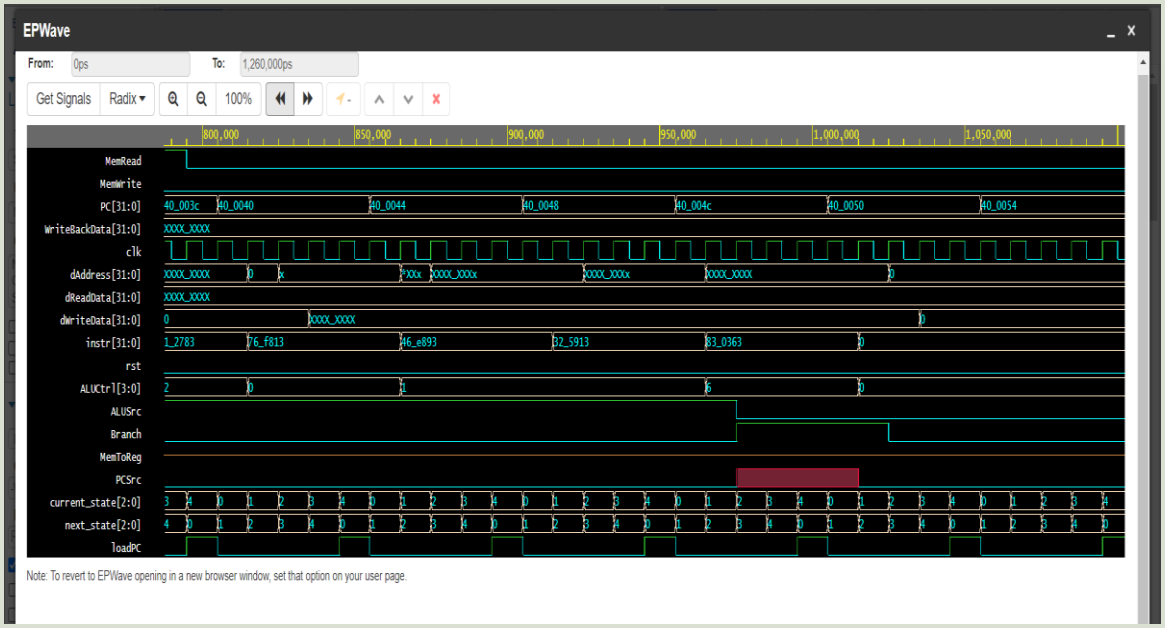
Φαίνεται ότι το PC αρχικοποιείται λαμβάνοντας την τιμή INITIAL_PC , όπως θα έπρεπε. Φορτώνεται τιμή στην λέξη εντολή instr και αποκωδικοποιείται ενεργοποιώντας το ALUSrc ,αφού δεν είναι εντολή BEQ. Γίνεται ομαλή μετάβαση μεταξύ των πέντε σταδίων με σειρά 0,1,2,3,4. Στη συνέχεια , το loadPC γίνεται 1, το instr λαμβάνει νέα τιμή και αφού πάλι δεν έχουμε εντολή BEQ το PC αυξάνεται κατά 4. Το ίδιο συμβαίνει και στις υπόλοιπες τιμές της λέξης εντολής. Η ALUctrl έχει την τιμή 2 δηλαδή λειτουργία ADD.



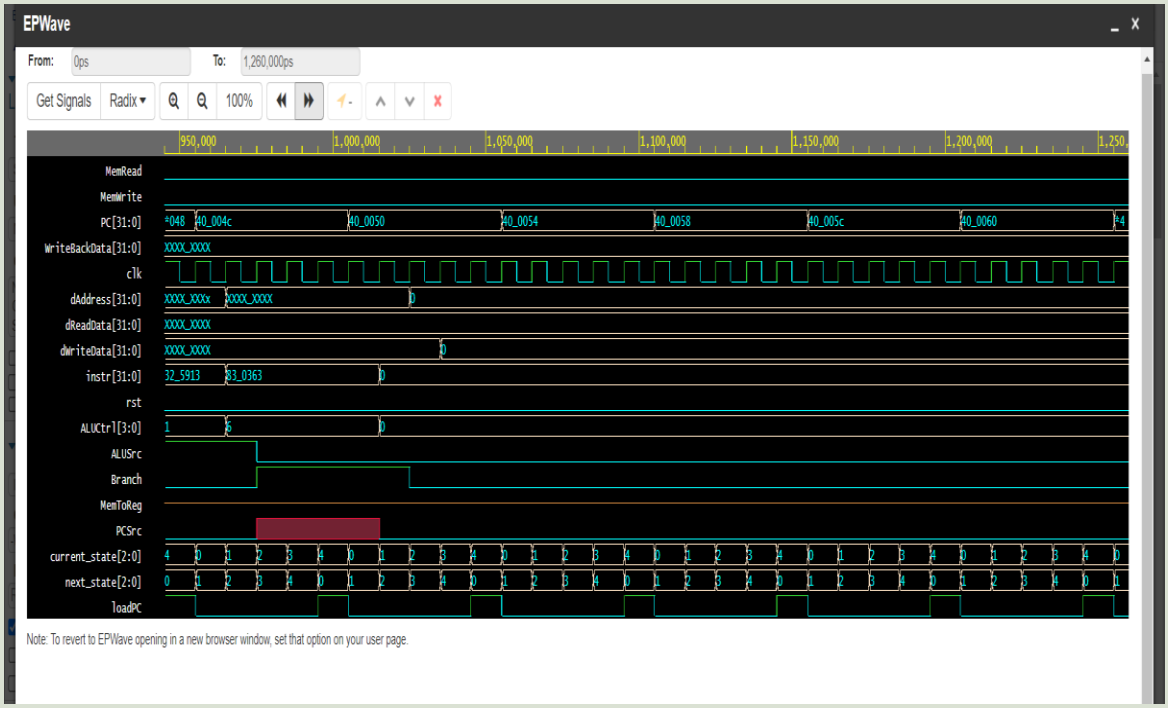
Φαίνεται ότι η λέξη εντολή εξακολουθεί να λαμβάνει τιμές που δεν είναι εντολές τύπου BEQ διότι η τιμή του PC αυξάνεται κάθε 5 κύκλους κατά 4 όταν ενεργοποιείται το loadPC .Παρατηρείται ομαλή μετάβαση μεταξύ των σταδίων, όπως και νωρίτερα . Η τιμή του ALUctrl μεταβάλλεται στις τιμές 6,9,7,13,8,2,1 δηλαδή SUB, SLL, SLT, XOR, SRL, AND, OR.



Φαίνεται ότι η λέξη εντολή εξακολουθεί να λαμβάνει τιμές που δεν είναι εντολές τύπου BEQ διότι η τιμή του PC αυξάνεται κάθε 5 κύκλους κατά 4 όταν ενεργοποιείται το loadPC. Παρατηρείται ομαλή μετάβαση μεταξύ των σταδίων, όπως και νωρίτερα. Η τιμή του ALUctrl μεταβάλλεται στις τιμές 10,2,0 δηλαδή SRA,AND, ANDI. Επίσης, η τιμή του σήματος ελέγχου για τις δύο τελευταίες τιμές της λέξης εντολής γίνεται 1 .



Φαίνεται ότι το PC σταματά να αυξάνεται κατά 4 , όταν λαμβάνει η instr εντολή τύπου BEQ(instr=83_0363). Το ALUSrc τότε γίνεται 0 ,το Branch 1. Ο επεξεργαστής λόγω του τύπου της εντολής μεταβαίνει απευθείας στο τελευταίο στάδιο WB ,όπως θα έπρεπε. Άρα, όλες οι λειτουργίες εκτελούνται σωστά.



Φαίνεται ότι η instr σταματά να λαμβάνει τιμές μετά τα 1050ps την τιμή 0 και σταματούν να μεταβάλλονται οι τιμές των σημάτων ελέγχου και του PC.

Ο κώδικας στη verilog είναι ο ξής:

```
`include"datapath.v"
module multicycle

#(parameter [31:0] INITIAL_PC = 32'h00400000,
  parameter [2:0]  IF=3'b000,
  parameter [2:0]  ID=3'b001,
  parameter [2:0]  EX=3'b010,
  parameter [2:0]  MEM=3'b011,
  parameter [2:0]  WB=3'b100)

 (output wire [31:0] PC ,dAddress,dWriteData,WriteBackData,
  output reg MemRead ,MemWrite,
  input wire clk,rst,
  input wire  [31:0] instr,dReadData);

 wire Zero;
 reg PCSrc,ALUSrc,RegWrite,MemtoReg,loadPC;
 reg [3:0] ALUCtrl;
 reg Branch;


 datapath DO (.PC(PC),.dAddress(dAddress),.dWriteData(dWriteData)
             ,.dReadData(dReadData),.WriteBackData(WriteBackData)
             ,.Zero(Zero),.clk(clk),.rst(rst),.PCSrc(PCSrc)
             ,.ALUSrc(ALUSrc),.RegWrite(RegWrite),.MemToReg(MemToReg)
             ,.loadPC(loadPC),.instr(instr),.ALUCtrl(ALUCtrl));



 always @(posedge clk)
   begin
     case (instr[6:0]) // opcode

          7'b0000011: ALUSrc <= 1; //  load instructions
          7'b0100011: ALUSrc <= 1; //  store instructions
          7'b0010011: ALUSrc <= 1; //  ALU Immediate instructions

          default: ALUSrc = 0;
       endcase
   end
```

```
    reg [2:0] current_state,next_state;

always@(posedge clk or posedge rst)
      begin
        if(rst)
          current_state<=IF;
        else
          current_state<=next_state;
      end


    always@(current_state)
      begin
        case(current_state)
          IF:next_state=ID;
          ID:next_state=EX;
          EX:begin
            if(PCSrc)
              next_state=WB;
            else
              next_state=MEM;
          end
          MEM:next_state=WB;
          WB:next_state=IF;
        endcase
      end




    always@(current_state)
      begin
        case(current_state)
          IF:begin
            loadPC=1'b0;
            RegWrite=1'b0;
          end

          ID:begin
            if(Zero==1'b1 && instr[6:0]==7'b1100011)
              PCSrc=1'b1;
```

```
80            ID:begin                                                SV/Verilog Desig
81              if(Zero==1'b1 && instr[6:0]==7'b1100011)
82                PCSrc=1'b1;
83              else
84                PCSrc=1'b0;
85
86
87      case (instr[6:0]) // opcode
88          7'b0110011: // R-type
89              begin
90                case (instr[31:25]) // funct7
91                  7'b0000000:
92                    begin
93                      case (instr[14:12]) // funct3
94                        3'b000:ALUCtrl=4'b0010; //ADD
95                        3'b001:ALUCtrl=4'b1001; //SLL
96                        3'b010:ALUCtrl=4'b0111; //SLT
97                        3'b100:ALUCtrl=4'b1101; //XOR
98                        3'b101:ALUCtrl=4'b1000; //SRL
99                        3'b110:ALUCtrl=4'b0001; //OR
00                        3'b111:ALUCtrl=4'b0010; //AND
01                      endcase
02                    end
03                  7'b0100000:
04                      begin
05                        case (instr[14:12]) // funct3
06                          3'b000:ALUCtrl=4'b0110; //SUB
07                          3'b101:ALUCtrl=4'b1010; //SRA
08                        endcase
09                      end
10              endcase
11            end
12         7'b0010011://I-type
13              begin
14                case (instr[14:12]) // funct3
15                  3'b000:ALUCtrl=4'b0010; //ADDI
16                  3'b010:ALUCtrl=4'b0111; //SLTI
17                  3'b100:ALUCtrl=4'b1101; //XORI
18                  3'b110:ALUCtrl=4'b0001; //ORI
19                  3'b111:ALUCtrl=4'b0000; //ANDI
20                endcase
```

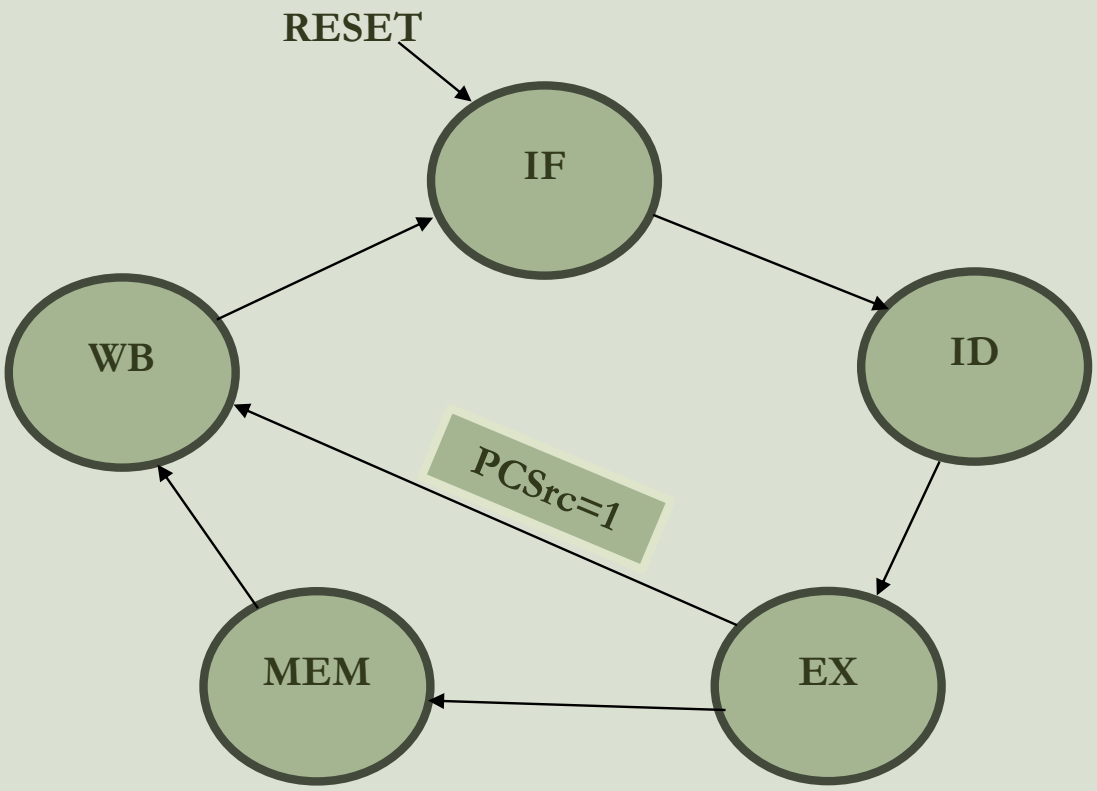
```
121             end                                                    SV/Verilo
122
123
124         7'b0000011: ALUCtrl = 4'b0010; // LW/SW -> ADD(LW)
125         7'b0100011: ALUCtrl = 4'b0010; // LW/SW -> ADD(SW)
126         7'b1100011: ALUCtrl = 4'b0110; // BEQ -> SUB
127
128         default: ALUCtrl = 4'b0000; // Default case
129     endcase
130 end
131             EX:begin
132                Branch=(instr[6:0]==7'b1100011)?1'b1:1'b0;
133
134               PCSrc = (Branch & Zero);
135             end
136
137             MEM:begin
138               if(instr[6:0]==7'b0100011)//store instr
139                 begin
140                    MemWrite=1'b1;
141                    MemRead=1'b0;
142                 end
143
144               else if(instr[6:0]==7'b0000011)//load
145                 begin
146                    MemRead=1'b1;
147                    MemWrite=1'b0;
148                 end
149               else
150                 begin
151                    MemWrite=1'b0;
152                    MemRead=1'b0;
153                 end
154
155                  end
156
157             WB: begin
158                loadPC=1'b1;
159               if(PCSrc==1'b0)
160                 begin
161                   MemWrite=1'b0;
```

```
                end

              else if(instr[6:0]==7'b0000011)//load
                begin
                   MemRead=1'b1;
                   MemWrite=1'b0;
                end
              else
                begin
                   MemWrite=1'b0;
                   MemRead=1'b0;
                end

                 end

            WB: begin
                loadPC=1'b1;
              if(PCSrc==1'b0)
                begin
                  MemWrite=1'b0;
                  MemRead=1'b0;
                  RegWrite=1'b1;
                  if(instr[6:0]==7'b0000011 )//load instr
                    RegWrite=1'b0;
                  if(instr[6:0]==7'b0000011)//load instr
                  MemtoReg=1'b1;
                  else
                    MemtoReg=1'b0;
                end


              end

    endcase


      end

endmodule
```

ο κώδικας του testbench:

```
3
4  module TB_multicycle;
5
6    reg clk;
7    reg rst;
8    wire [31:0] instr;
9    wire [31:0] dReadData;
10   wire [31:0] PC, dAddress, dWriteData, WriteBackData;
11   wire MemRead, MemWrite;
12
13   // Συνδέουμε τον multicycle module με το testbench
14   multicycle DUT (
15     .PC(PC), .dAddress(dAddress), .dWriteData(dWriteData),
16     .WriteBackData(WriteBackData), .clk(clk),
17     .rst(rst), .instr(instr),
18     .MemRead(MemRead), .MemWrite(MemWrite), .dReadData(dReadData)
19   );
20
21   DATA_MEMORY DUT_DATA_MEMORY (
22     .clk(clk), .we(MemWrite), .addr(dAddress[8:0]),
23     .din(dWriteData), .dout(dReadData)
24   );
25
26   INSTRUCTION_MEMORY DUT_INSTRUCTION_MEMORY (
27     .clk(clk), .addr(PC[8:0]), .dout(instr)
28   );
29
30   // Clock generation
31    always #5 clk = ~clk;
32
33     initial begin
34       $dumpfile("TB_multicycle.vcd");
35       $dumpvars(0, TB_multicycle);
36         clk = 0;
37         rst = 1;
38         #10;
39         rst = 0;
40         #210;
41         $finish;
42     end
43 endmodule
```

Δίνεται το σχηματικό διάγραμμα FSM :



Στο σχηματικό διάγραμμα FSM παρουσιάζονται τα πέντε διαφορετικά στάδια.Η μετάβαση από το ένα στάδιο στο επόμενο πραγματοποιείται σε κάθε ακμή του ρολογιού(ένας κύκλος ρολογιού).Σε περίπτωση που υπαρχει εντολή τύπου BEQ ενεργοποιείται το σήμα ελέγχου PCSrc και μεταβαίνουμε απευθείας από το τρίτο στάδιο στο πέμπτο .Με την ενεργοποίση του RESET οδηγούμαστε στο πρώτο στάδιο IF, παροχής του PC στη μνήμη εντολών.