Πολυτεχνική Σχολή ΑΠΘ

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής Ι

7ο εξάμηνο

Υλοποίηση επεξεργαστή RISC-V σε γλώσσα περιγραφής υλικού Verilog.

Περιβάλλον ανάπτυξης: EDA Playground IDE

Ιανουάριος 2025

Ιωάννα Ινώ Χαριτίδη, 10043

Άσκηση 1: Υλοποίηση ενός Arithmetic Logic Unit (ALU) με χρήση Verilog

Η ALU που σχεδιάστηκε στα πλαίσια αυτής της άσκησης δέχεται ως είσοδο δύο τελεστέους (op1 και op2 , μήκους 32 bit) και δίνει ως έξοδο το αποτέλεσμα εφαρμογής διάφορων πράξεων πάνω σε αυτούς (result, μήκους 32 bit).

Ένα επιπλέον σήμα εξόδου (zero , μήκους 1 bit) λαμβάνει τις τιμές 1 και 0, και δηλώνει εάν το αποτέλεσμα είναι μηδενικό ή όχι.

Το είδος πράξης που εφαρμόζεται στους δύο τελεστέους op1 και op2 καθορίζεται από το σήμα εισόδου alu_op, το οποίο έχει μήκος 4 bit για να μπορεί να αναπαραστήσει τουλάχιστον 9 τιμές.

Συνολικά η ALU έχει 3 εισόδους και 2 εξόδους.

Η ALU υλοποιεί τις ακόλουθες 9 πράξεις. Η κάθε μία σηματοδοτείται από διαφορετική τιμή της alu_op:

- προσημασμένη πρόσθεση (alu_op=0010),
- προσημασμένη αφαίρεση (alu_op=0110),
- λογικό AND (alu_op=0000) ,
- λογικό OR (alu_op=0001),
- λογικό XOR (alu_op=0101),
- σύγκριση "Μικρότερο από" πάνω σε προσημασμένους αριθμούς (alu_op=0100),
- Λογική ολίσθηση δεξιά κατά op2 bits (alu_op=1000),
- Λογική ολίσθηση αριστερά κατά op2 bits (alu_op=1001) και
- Αριθμητική ολίσθηση δεξιά κατά op2 bits (alu_op=1010)

Για ευκολία, οι πιθανές τιμές της alu_op ορίζονται ως παράμετροι εντός του αρχείου alu.v, με χρήση της οδηγίας parameter και ονόματα που περιγράφουν την αντίστοιχη λειτουργία (πχ parameter[3:0] ALUOP_SUB = 4'b0110;).

Η επιλογή του είδους πράξης πραγματοποιείται από έναν πολυπλέκτη, ο οποίος υλοποιείται με χρήση ενός γενόμενου υπό όρους τελεστή. Συγκεκριμένα χρησιμοποιείται ο τριαδικός τελεστής ***? :*** της Verilog (ternary operator).

Το κύκλωμα της ALU είναι συνδυαστικό, πράγμα που σημαίνει ότι οι έξοδοι αλλάζουν άμεσα όποτε αλλάξει μία από τις εισόδους. Αυτή η συμπεριφορά προσομοιώνεται με χρήση συνεχούς ανάθεσης assign στις εξόδους του κυκλώματος.

```
module ALU(
  output [31:0] result,
  output zero,
  input [31:0] op1,
  input [31:0] op2,
  input [3:0] alu_op);

  parameter[3:0] ALUOP_AND = 4'b0000;
  parameter[3:0] ALUOP_OR = 4'b0001;
  parameter[3:0] ALUOP_ADD = 4'b0010;
  parameter[3:0] ALUOP_SUB = 4'b0110;
  parameter[3:0] ALUOP_LT = 4'b0100;
  parameter[3:0] ALUOP_Rsh = 4'b1000;
  parameter[3:0] ALUOP_Lsh = 4'b1001;
  parameter[3:0] ALUOP_NRsh = 4'b1010;
  parameter[3:0] ALUOP_XOR = 4'b0101;

assign result=(alu_op == ALUOP_AND) ? (op1&op2):
         (alu_op == ALUOP_OR) ? (op1|op2):
         (alu_op == ALUOP_ADD) ? (op1+op2):
         (alu_op == ALUOP_SUB) ? (op1-op2):
         (alu_op == ALUOP_LT) ? ($signed(op1)<$signed(op2)):
         (alu_op == ALUOP_Rsh) ? (op1>>op2[4:0]):
         (alu_op == ALUOP_Lsh) ? (op1<<op2[4:0]):
         (alu_op == ALUOP_NRsh) ? ($unsigned($signed(op1)>>>op2[4:0])):
         (alu_op == ALUOP_XOR) ? (op1^op2):
         (1'bX);
  assign zero=(result == 32'b0) ? (1'b1):(1'b0);

endmodule
```

Figure 1: Μονάδα ALU

Άσκηση 2: Κύκλωμα Αριθμομηχανής

```
`include "ALU.v"
`include "calc_enc.v"

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

  wire [3:0] n1;//internal wires
  wire [31:0] n2;
  wire [31:0] n3;
  wire [15:0] n4;
  wire [31:0] n5;

  sign_extend ex1(.ext_bit32(n5), .bit16(n4));
  sign_extend ex2(.ext_bit32(n2), .bit16(sw));

  calc_enc ALU_CONTROL(.alu_op(n1), .btnl(btnl), .btnc(btnc), .btnr(btnr));

  ALU alu( .result(n3), .zero(z), .op1(n5), .op2(n2), .alu_op(n1) );

  accumulator ACC( .acc_out(n4), .clc(clc), .btnu(btnu), .btnd(btnd), .data_in(n3[15:0]) );

  assign led = n4;

endmodule
```

Figure 2: Υλοποίηση κυκλώματος αριθμομηχανής σε Verilog. Οι υπο-μονάδες sign_extend, calc_enc και accumulator περιγράφονται παρακάτω. Η υπο-μονάδα ALU είναι αυτή που παρουσιάστηκε στην 1η άσκηση.

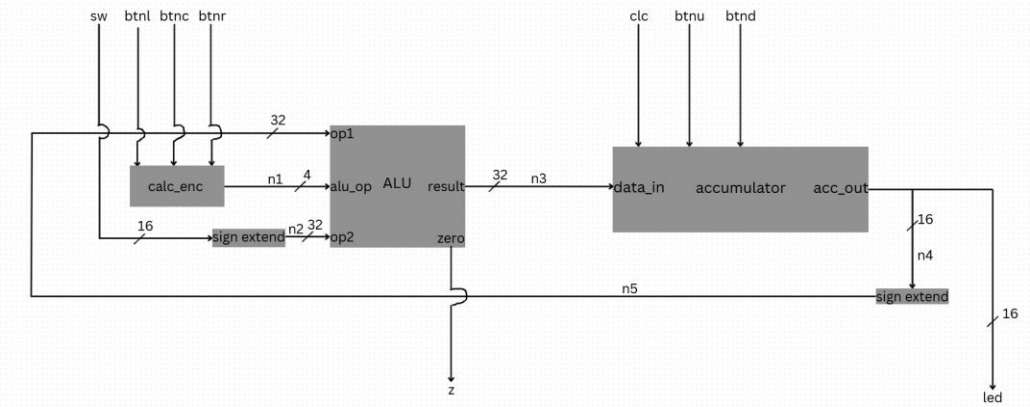


Figure 3: Σχηματική αναπαράσταση του calc.v. Κάθε ένα από τα γκρι πλαίσια αναπαριστά μία μονάδα.

Η εν λόγω αριθμομηχανή διατηρεί μια τρέχουσα τιμή της σε ένα συσσωρευτή 16-bit καταχωρητή και επιτρέπει στο χρήστη να ενημερώνει την τιμή υλοποιώντας οποιαδήποτε από τις αριθμητικές και λογικές συναρτήσεις που παρέχει η ALU.

Υλοποιείται με τη μονάδα calc, στο αρχείο calc.v . Το αρχείο περιέχει, πέρα από την calc, τις βοηθητικές μονάδες accumulator (υλοποίηση του συσσωρευτή 16-bit) και sign_extend (επέκταση προσήμου σημάτων) . Επίσης κάνει include τα εξής αρχεία:

- ALU.v , το οποίο περιέχει την ALU της άσκησης 1, και
- calc_enc.v, το οποίο περιγράφεται παρακάτω.

```
33
34 module accumulator (
35   output reg [15:0] acc_out,
36   input wire clc,
37   input wire btnu,
38   input wire btnd,
39   input wire [15:0] data_in);
40
41   always @(posedge clc) begin
42     if(btnu)
43        acc_out <= 16'b0;
44     else if(btnd)
45       acc_out<=data_in;
46   end
47
48 endmodule
49
```

Figure 4: Βοηθητική μονάδα accumulator: αναπαράσταση με Verilog ενός καταχωρητή 16 bit

Ο accumulator δέχεται ως είσοδο (data_in) τα 16 χαμηλότερα bit της εξόδου αποτελέσματος 32 bit της ALU. Μηδενίζεται σύγχρονα (στην ανιούσα ακμή του σήματος clc) με το πάτημα του btnu, και ενημερώνεται σύγχρονα (επίσης στην ανιούσα ακμή) κάθε φορά που πατιέται το btnd.

Η τιμή του καταχωρητή (acc_out) συνδέεται με τις εξόδους led της αριθμομηχανής, αλλά και με την είσοδο op1 της ALU, αφού υποστεί επέκταση προσήμου χρησιμοποιώντας τον τελεστή concatenation της Verilog, για να μετετραπεί σε σήμα 32 bit.

Η είσοδος op2 της ALU συνδέεται με μια έκδοση με επέκταση προσήμου των εισόδων του διακόπτη 16-bit (sw).

```
49
50 module sign_extend(
51   output wire [31:0] ext_bit32,
52   input wire [15:0] bit16);
53
54    assign ext_bit32= { {16{bit16[15]}},bit16 };
55
56 endmodule
57
```

Figure 5: Βοηθητική μονάδα επέκτασης προσήμου, για τη δημιουργία σημάτων 32 bits από σήματα των 16 bits.

Η επιλογή της λειτουργίας που θα εκτελέσει η ALU γίνεται με βάση τα πλήκτρα btnl, btnc και btnr. Αυτά τα τρία πλήκτρα αποτελούν εισόδους στο συνδυαστικό κύκλωμα calc_enc.v , το οποίο υλοποιείται σε structural Verilog. Η έξοδός του συνδέεται με την είσοδο alu_op της ALU.

Η λογική επιλογής λειτουργίας υλοποιείται στο αρχείο calc_enc.v, με τη μονάδα calc_enc και τις βοηθητικές μονάδες bit0, bit1, bit2, bit3.

```
83 module calc_enc(
84   output wire [3:0] alu_op,
85   input wire btnl,
86   input wire btnc,
87   input wire btnr);
88
89   bit0 zero(.zeroBit(alu_op[0]), .C(btnc), .R(btnr), .L(btnl));
90   bit1 one(.oneBit(alu_op[1]), .C(btnc), .R(btnr), .L(btnl));
91   bit2 two(.twoBit(alu_op[2]), .C(btnc), .R(btnr), .L(btnl));
92   bit3 three(.threeBit(alu_op[3]), .C(btnc), .R(btnr), .L(btnl));
93
94 endmodule
```

Figure 6: Μονάδα calc_enc

Τα sub-modules bit0, bit1, bit2, bit3 δέχονται ως είσοδο τα btnr, btnl, btnc , και παράγουν ως έξοδο τα alu_op[0] (zeroBit) , alu_op[1] (oneBit) , alu_op[2] (twoBit), alu_op[3] (threeBit) αντίστοιχα:

```
1  module bit0(
2    output wire zeroBit,
3    input wire C,
4    input wire R,
5    input wire L);
6
7    wire Cn;
8    wire m1;
9    wire m2;
10
11   not N0(Cn,C);
12   and A0_1(m1,Cn,R);
13   and A0_2(m2,L,R);
14   or O0(zeroBit,m1,m2);
15
16 endmodule
```

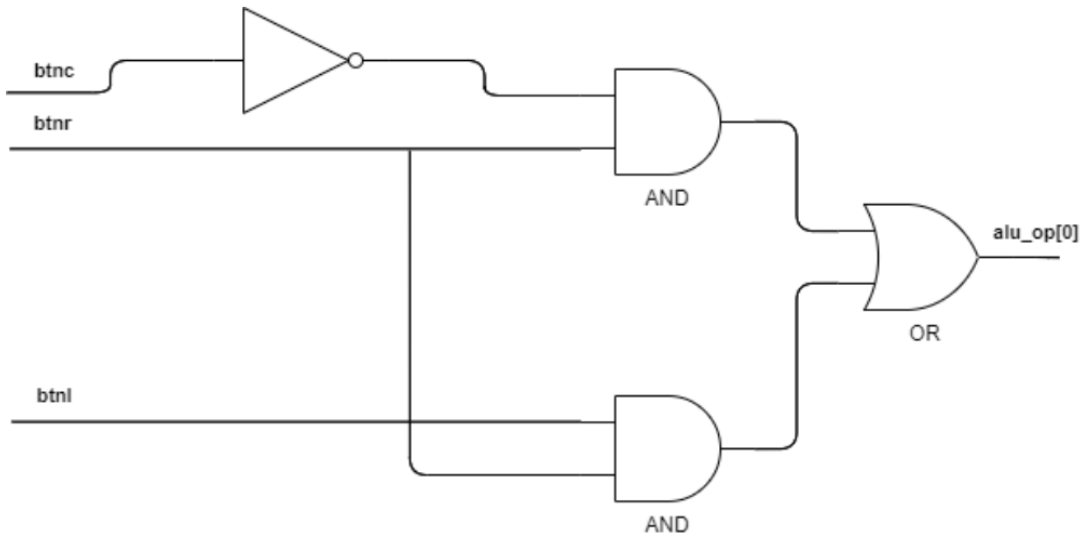


Figure 7: Βοηθητική μονάδα bit0, και σχηματική αναπαράσταση παραγωγής του alu_op[0] μέσω των btnr, btnl, btnc

```
18 module bit1(
19   output wire oneBit,
20   input wire C,
21   input wire R,
22   input wire L);
23
24   wire Ln;
25   wire Rn;
26   wire m3;
27   wire m4;
28
29   not N1_1(Ln,L);
30   not N1_2(Rn,R);
31   and A1_1(m3,Ln,C);
32   and A1_2(m4,C,Rn);
33   or O1(oneBit,m3,m4);
34
35 endmodule
```

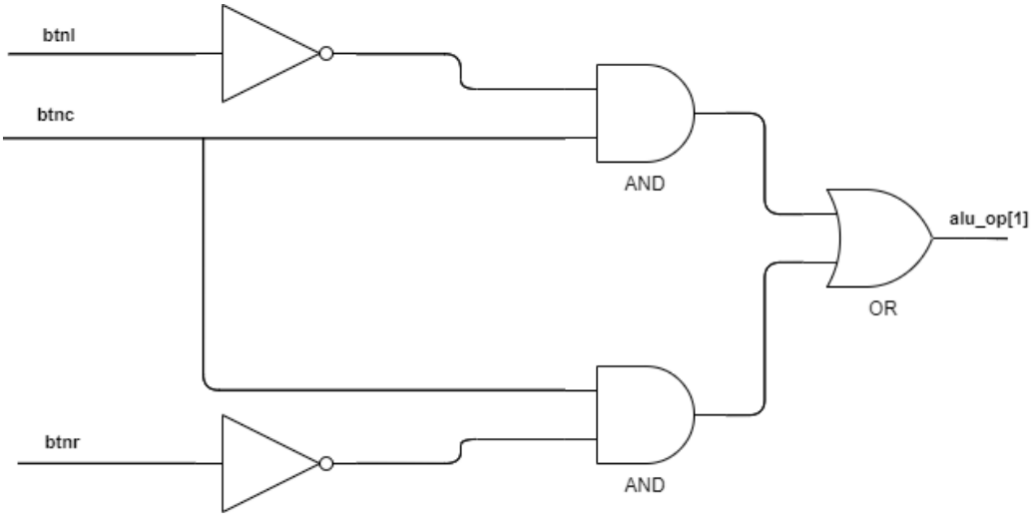


Figure 8: Βοηθητική μονάδα bit1, και σχηματική αναπαράσταση παραγωγής του alu_op[1] μέσω των btnr, btnl, btnc

```
37 module bit2(
38   output wire twoBit,
39   input wire C,
40   input wire R,
41   input wire L);
42
43   wire Cn;
44   wire Rn;
45   wire m5;
46   wire m6;
47   wire m7;
48
49   not N2_1(Cn,C);
50   not N2_2(Rn,R);
51   and A2_1(m5,C,R);
52   and A2_2(m6,L,Cn);
53   and A2_3(m7,m6,Rn);
54
55   or O2(twoBit,m5,m7);
56
57 endmodule
```

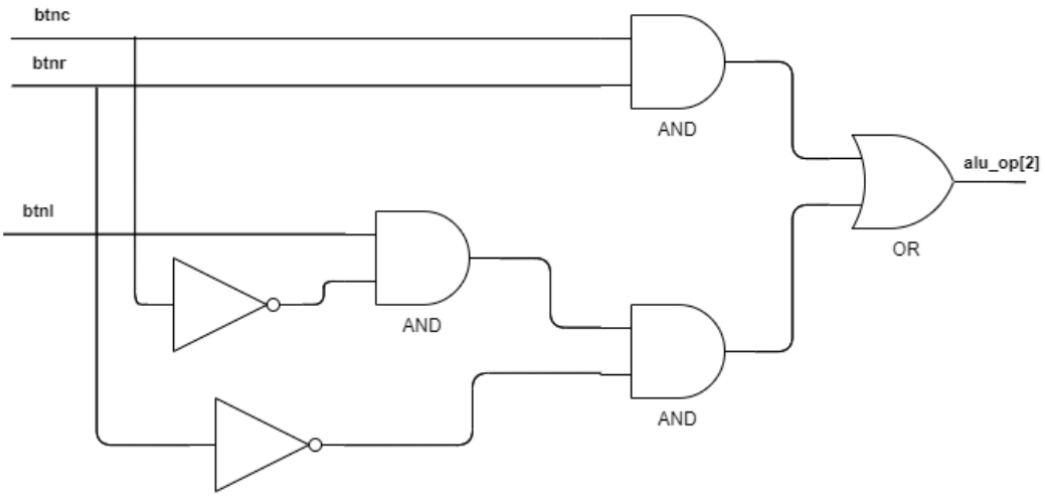


Figure 9: Βοηθητική μονάδα bit2, και σχηματική αναπαράσταση παραγωγής του alu_op[2] μέσω των btnr, btnl, btnc

```
59 module bit3(
60   output wire threeBit,
61   input wire C,
62   input wire R,
63   input wire L);
64
65   wire Cn;
66   wire Rn;
67   wire m8;
68   wire m9;
69   wire m10;
70   wire m11;
71
72   not N3_1(Cn,C);
73   not N3_2(Rn,R);
74   and A3_1(m8,L,Cn);
75   and A3_2(m11,m8,R);
76   and A3_3(m9,L,C);
77   and A3_4(m10,m9,Rn);
78
79   or O3(threeBit,m11,m10);
80
81 endmodule
```

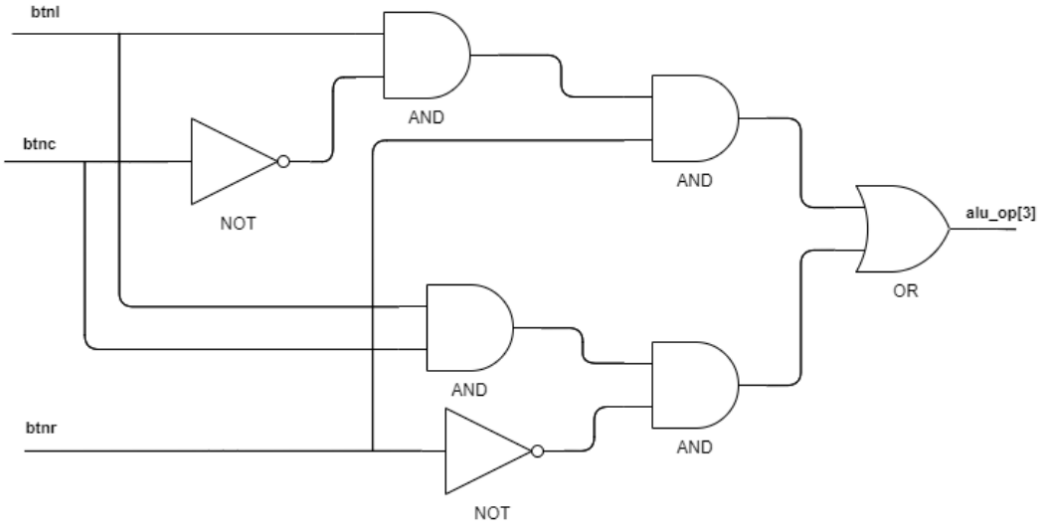


Figure 10: Βοηθητική μονάδα bit3, και σχηματική αναπαράσταση παραγωγής του alu_op[3] μέσω των btnr, btnl, btnc

Για τον έλεγχο της ορθής λειτουργίας της αριθμομηχανής και της ALU, δημιουργείται το παρακάτω testbench για το module calc :

```
// Code your testbench here
// or browse Examples
`timescale 1ns / 1ps

module calc_tb;
  wire [15:0] LED;
  wire Z;
  reg CLC;
  reg C;
  reg L;
  reg U;
  reg R;
  reg D;
  reg [15:0] SW;

  calc calc_tb(LED,Z,CLC,C,L,U,R,D,SW);

  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0,calc_tb);

    CLC=1'b0;

    U=1'b1; //RESET. UP BUTTON IS PRESSED
    //doesn't matter what the rest are. x.

    #20
    L=1'b0;
    C=1'b1;
    R=1'b0;
    U=1'b0;
    D=1'b1;
    SW=16'h354a;

    #20
    L=1'b0;
    C=1'b1;
    R=1'b1;
    U=1'b0;
    D=1'b1;
    SW=16'h1234;

    #20
    L=1'b0;
    C=1'b0;
    R=1'b1;
    U=1'b0;
    D=1'b1;
    SW=16'h1001;

    #20
    L=1'b0;
    C=1'b0;
    R=1'b0;
    U=1'b0;
    D=1'b1;
    SW=16'hf0f0;

    #20
    L=1'b1;
    C=1'b1;
    R=1'b1;
    U=1'b0;
    D=1'b1;
    SW=16'h1fa2;

    #20
    L=1'b0;
    C=1'b1;
    R=1'b0;
    U=1'b0;
    D=1'b1;
    SW=16'h6aa2;

    #20
    L=1'b1;
    C=1'b0;
    R=1'b1;
    U=1'b0;
    D=1'b1;
    SW=16'h0004;
```

```
83      #20
84      L=1'b1;
85      C=1'b1;
86      R=1'b0;
87      U=1'b0;
88      D=1'b1;
89      SW=16'h0001;
90
91      #20
92      L=1'b1;
93      C=1'b0;
94      R=1'b0;
95      U=1'b0;
96      D=1'b1;
97      SW=16'h46ff;
98
99
100   end
101
102   initial #200 $finish;
103   always #10 CLC=~CLC;
104
105 endmodule
106
```

Figure 11: Testbench calc_tb

Μετά από την εφαρμογή του στο κύκλωμα, προκύπτουν αποτελέσματα τα οποία ταυτίζονται με τις αναμενόμενες τιμές που δίνονται από εκφώνηση:

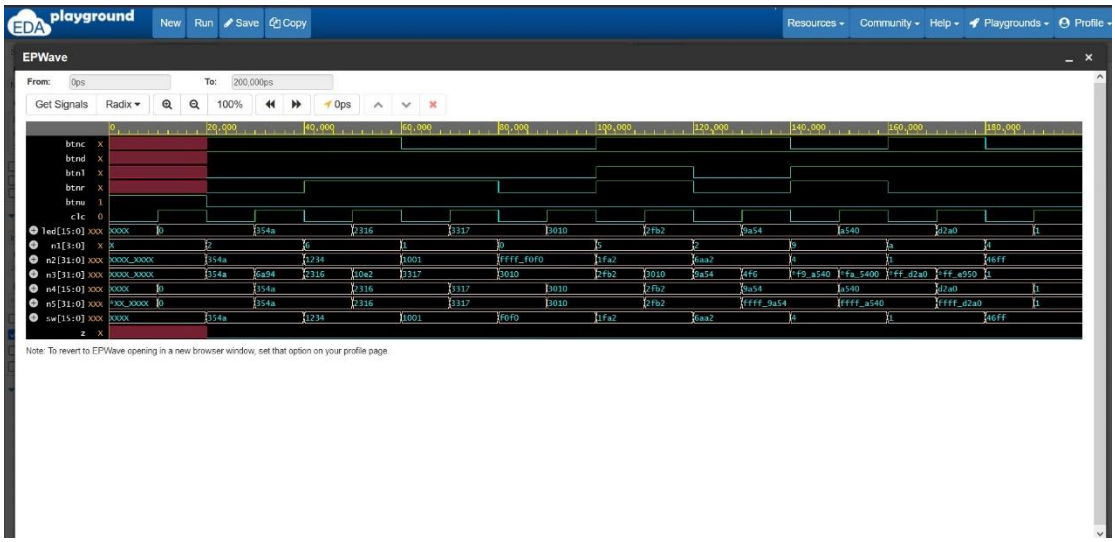


Figure 12: Κυματομορφές. Οι δοκιμαστικές τιμές των sw, btnc, btnd, btnl, btnr, btnu δίνονται από την εκφώνηση, όπως και οι αναμενόμενες τιμές της εξόδου led.

Άσκηση 3: Αρχείο Καταχωρητών

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

  parameter DATAWIDTH = 32;
  reg [DATAWIDTH-1 :0] register [31:0];

  initial begin: initialize_registers   //it is required to name the block when declaring local variables
    integer i;
    for (i = 0; i < 32; i = i + 1) begin
      register[i] = 0;
    end
  end

  // Write
  always @(posedge clk) begin
        if (write) begin
          register[writeReg] <= writeData;
        end
  end

  // Read
   always @(posedge clk) begin
        // Prioritize write over read if readReg1 is the same as writeReg
        if (write && (writeReg == readReg1)) begin
            readData1 <= writeData;  // If we are writing to the same register as readReg1, return the written data
        end else begin
            readData1 <= register[readReg1];  // Else, just read the register
        end
        // Prioritize write over read if readReg2 is the same as writeReg
        if (write && (writeReg == readReg2)) begin
            readData2 <= writeData;  // If we are writing to the same register as readReg2, return the written data
        end else begin
            readData2 <= register[readReg2];  // Else, just read the register
        end
    end
endmodule
```

Figure 13: Μονάδα regfile

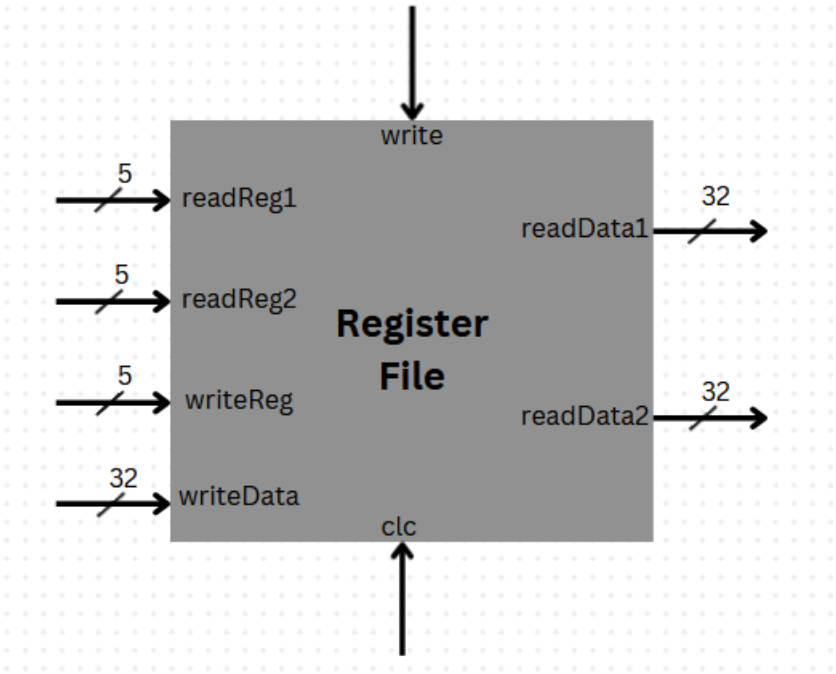


Figure 14: Σχηματική αναπαράσταση του regfile.v.

Σε αυτή την άσκηση υλοποιείται ένα αρχείο 32 καταχωρητών, μέσω της μονάδας regfile. Ο κάθε καταχωρητής έχει μήκος 32 bit .

Η μονάδα regfile έχει 2 εξόδους και 6 εισόδους. Οι έξοδοι, readData1 και readData2, είναι τύπου reg, και περιέχουν τις τιμές των καταχωρητών, των οποίων η διεύθυνση δόθηκε από τις εισόδους readReg1 και readReg2 αντίστοιχα. Οι έξοδοι ενημερώνονται σύγχρονα με την ανιούσα ακμή του σήματος εισόδου clc.

Επιπλέον υπάρχει δυνατότητα εγγραφής νέων δεδομένων στους καταχωρητές, μέσω των σημάτων εισόδου writeReg, writeData και write. Το σήμα writeReg αντιπροσωπεύει τη διεύθυνση του καταχωρητή στον οποίο θα γίνει η εγγραφή, το writeData περιέχει τα νέα δεδομένα, και το write είναι σήμα ελέγχου. Οι εγγραφές πραγματοποιούνται σύγχρονα με την ανιούσα ακμή του clc, και μόνο εφόσον το write είναι ενεργό.

Περιγραφή Κώδικα:

Οι καταχωρητές (reg [DATAWIDTH-1 : 0] register[31:0], όπου DATAWIDTH παράμετρος ίση με 32 από προεπιλογή), αρχικοποιούνται με μηδενικά μέσω ενός initial block και ενός βρόχου for.

```
12  parameter DATAWIDTH = 32;
13  reg [DATAWIDTH-1 :0] register [31:0];
14
15  initial begin: initialize_registers   //it is required to name the block when declaring local variables
16    integer i;
17    for (i = 0; i < 32; i = i + 1) begin
18      register[i] = 0;
19    end
20  end
```

Η λειτουργικότητα εγγραφής υλοποιείται με ένα ακολουθιακό always block (δηλαδή ένα always block με το σήμα ρολογιού στη λίστα ευαισθησίας του). Εάν το σήμα write είναι ενεργό (συνθήκη if(write)), ανατίθεται στον καταχωρητή με διεύθυνση writeReg η τιμή που περιέχεται στο σήμα writeData, με χρήση non-blocking εντολής.

```
22  // Write
23  always @(posedge clk) begin
24        if (write) begin
25          register[writeReg] <= writeData;
26        end
27  end
```

Η λειτουργία ανάγνωσης υλοποιείται σε ένα δεύτερο, επίσης ακολουθιακό always block, με χρήση εντολών συνθήκης if-else. Υπάρχει ειδική μέριμνα για την περίπτωση που η διεύθυνση εγγραφής (writeReg) είναι ίδια με κάποια από τις διευθύνσεις ανάγνωσης (readReg1 ή readReg2).

Εάν το σήμα write είναι ενεργό (οπότε και πραγματοποιείται εγγραφή εκείνη τη στιγμή), και η readReg1 ή readReg2 είναι ίδια με τη writeReg, η εγγραφή πρέπει να πάρει προτεραιότητα. Επομένως τα δεδομένα που δρομολογούνται στις θύρες ανάγνωσης είναι τα δεδομένα που προορίζονται για εγγραφή στο συγκεκριμένο καταχωρητή, και όχι αυτά που ο καταχωρητής περιείχε μέχρι τώρα.

Διαφορετικά, τα δεδομένα που διαβάζονται είναι τα δεδομένα που περιέχει ο εκάστοτε καταχωρητής.

```
// Read
 always @(posedge clk) begin
      // Prioritize write over read if readReg1 is the same as writeReg
      if (write && (writeReg == readReg1)) begin
          readData1 <= writeData;  // If we are writing to the same register as readReg1, return the written data
      end else begin
          readData1 <= register[readReg1];  // Else, just read the register
      end
      // Prioritize write over read if readReg2 is the same as writeReg
      if (write && (writeReg == readReg2)) begin
          readData2 <= writeData;  // If we are writing to the same register as readReg2, return the written data
      end else begin
          readData2 <= register[readReg2];  // Else, just read the register
      end
  end
```

Άσκηση 4: Διαδρομή Δεδομένων

```
`include "ALU.v"
`include "regfile.v"

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

  parameter INITIAL_PC=32'h00400000; //0x00400000
  parameter opcode_R=7'b0110011;
  parameter opcode_I=7'b0010011;
  parameter opcode_S=7'b0100011;
  parameter opcode_B=7'b1100011;
  parameter opcode_LW=7'b0000011;//LW is type Immediate, but has different opcode



  //internal nets

  reg [4:0] i1; //regfile input, readReg1
  reg [4:0] i2; //regfile input, readReg2
  reg [4:0] i3;  //regfile input, writeReg
  reg [11:0] imm12;  //immediate value
  reg [31:0] imm32;
  wire [31:0] n1;  //connects readData1 with op1 of ALU
  wire [31:0] n2;  //connects readData2 with the mux that decides whether ALU's second operant is an immediate value or a register's value, and with dWriteData of RAM.
  wire [31:0] n3;  //connects the output of the mux with op2 of ALU
  wire [31:0] n4;   //connects the ALU's result with dAddress of RAM, and with the mux that decides the value of WriteBackData
  //



  initial begin
    PC=INITIAL_PC;
  end

  always @(instr) begin


    i1=instr[19:15]; //internal net that connects to readReg1
    i2=instr[24:20]; //internal net that connects to readReg2
    i3=instr[11:7]; //internal net that connects to writeReg

    if(instr[6:0]==opcode_I || instr[6:0]==opcode_LW) //imm gen
      imm12=instr[31:20]; //for I types
    else if(instr[6:0]==opcode_S)
      imm12={instr[31:25], instr[11:7]}; //for S types
    else if(instr[6:0]==opcode_B)
      imm12={instr[31],instr[7],instr[30:25],instr[11:8]}; //for B types
    else if(instr[6:0]==opcode_R)
      imm12=12'b000000000000;   //if it's an R type, no immediate value is used, so it doesn't matter what
    else begin                          //imm12 is. Set it to 0 instead of x to avoid unpredictable behaviour.
      $display ("Invalid Opcode at Imm Gen");
      imm12=12'b000000000000;//new addition to the code
    end
    imm32={ {20{imm12[11]}},imm12 };

  end

  regfile REGFL(.readData1(n1), .readData2(n2), .clk(clk), .readReg1(i1), .readReg2(i2), .writeReg(i3) , .writeData(WriteBackData), .write(RegWrite));
  mux MUX1(.chosen(n3), .option0(n2), .option1(imm32), .control(ALUSrc));//If ALUSrc=0, ALU gets rs2, otherwise ALU gets immediate data
  ALU ALU_DATAPATH (.result(n4), .zero(Zero), .op1(n1), .op2(n3), .alu_op(ALUCtrl) );

  assign dWriteData=n2; //wires can't be assigned values inside always blocks
  assign dAddress=n4;       //n4 is the result output of the ALU


  mux MUX2(.chosen(WriteBackData), .option0(n4), .option1(dReadData), .control(MemToReg));



  always @(posedge clk) begin
    if (!rst)
      PC<=INITIAL_PC;

    else if(loadPC) begin
      if(PCSrc) //multiplexer
        PC<=PC+(imm32<<1);//PC + branch_offset (Branch Target)
      else
        PC<=PC+4;//PC + 4 (όταν το πρόγραμμα προχωρά στην επόμενη εντολή στη μνήμη)
    end
  end

endmodule


module mux(
  output wire [31:0] chosen,
  input wire [31:0] option0,
  input wire [31:0] option1,
  input wire control);

  assign chosen=(control) ? (option1) : (option0);

endmodule
```

Figure 15: Μονάδα datapath, και βοηθητική μονάδα mux

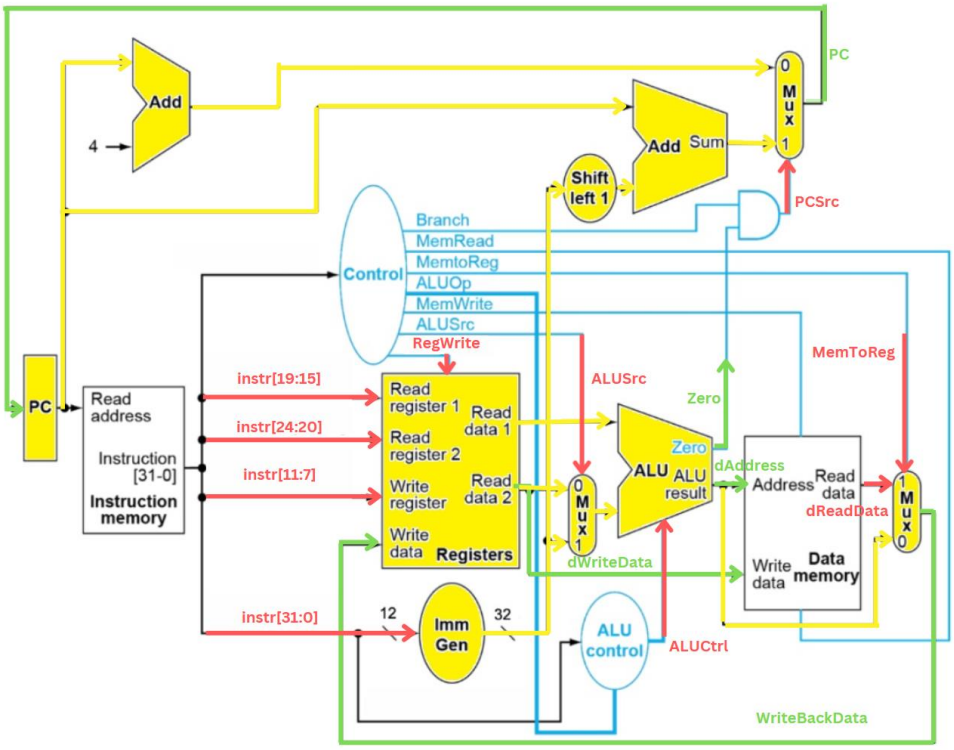


Figure 16: Διάγραμμα υλοποίησης για τη διαδρομή δεδομένων. Τα κομμάτια που αποτελούν τη μονάδα datapath.v, καθώς και τα internal nets της έχουν χρωματιστεί με κίτρινο. Οι είσοδοι της μονάδας έχουν κόκκινο χρώμα, ενώ οι έξοδοι πράσινο.

Το αντικείμενο αυτής της άσκησης είναι η διαδρομή δεδομένων του επεξεργαστή RISC-V. Περιέχει τις εσωτερικές λειτουργικές μονάδες, τους καταχωρητές και τους πολυπλέκτες που χρησιμοποιούνται για την υλοποίηση μεμονωμένων εντολών.

Η εκφώνηση παρέχει έτοιμη μία μνήμη εντολών και μία μνήμη δεδομένων, οι οποίες θα αξιοποιηθούν στην επόμενη άσκηση.

Η διαδρομή δεδομένων υλοποιείται με τη μονάδα datapath, τα βασικά τμήματα της οποίας είναι:

- Το αρχείο καταχωρητών που υλοποιήθηκε και περιγράφηκε στην 3η άσκηση,
- Η αριθμητική λογική μονάδα (ALU) που υλοποιήθηκε και περιγράφηκε στην 1η άσκηση,
- Ένα συνδυαστικό always block που δημιουργεί τις εισόδους του αρχείου καταχωρητών, καθώς και πιθανές εισόδους για τη δεύτερη θύρα ανάγνωσης της ALU, από το σήμα εισόδου instr του datapath (Immediate Generation),
- ο καταχωρητής PC (Program Counter), ο οποίος δείχνει τη θέση στη μνήμη εντολών όπου βρίσκεται η τρέχουσα εντολή που εκτελείται
- Η λογική διακλάδωσης, για τον υπολογισμό της νέας τιμής του PC σε περίπτωση που εκτελεστεί εντολή τύπου Branch, και
- Η λογική “εγγραφής προς τα πίσω”, που καθορίζει εάν η τιμή που εγγράφεται στο αρχείο καταχωρητών είναι το αποτέλεσμα της ALU (για τις συμβατικές αριθμητικές και λογικές εντολές) ή το αποτέλεσμα της ανάγνωσης μνήμης (dReadData). Η επιλογή γίνεται από έναν πολυπλέκτη.

Θα ξεκινήσουμε την περιγραφή της μονάδας datapath με το αρχείο καταχωρητών. Οι τιμές στις θύρες διευθύνσεων της υπομονάδας regfile , readReg1, readReg2, και writeReg, είναι τα bits 19-15, 24-20 και 11-7 αντίστοιχα της εισόδου instr της datapath. Αναπαριστώνται από τα εσωτερικά σήματα i1, i2, i3, στα οποία γίνεται συνεχής ανάθεση εντός ενός συνδυαστικού always block.

```
49      i1=instr[19:15]; //internal net that connects to readReg1
50      i2=instr[24:20]; //internal net that connects to readReg2
51      i3=instr[11:7]; //internal net that connects to writeReg
```

```
68  regfile REGFL(.readData1(n1), .readData2(n2), .clk(clk), .readReg1(i1), .readReg2(i2), .writeReg(i3) , .writeData(WriteBackData), .write(RegWrite));
```

Το σήμα instr έχει μήκος έχει μήκος 32 bit και αναπαριστά μία εντολή RISC-V σε δυαδική μορφή. Το σύνολο εντολών που υποστηρίζει το προς υλοποίηση datapath χωρίζεται σε 4 τύπους: R, I, S και B. Κάθε τύπος εντολής έχει διαφορετική κωδικοποίηση, δηλαδή χωρίζει τα 32 bit σε διαφορετικά πεδία. Τρία πεδία είναι κοινά ως προς τη θέση και για τους 4 τύπους:

- το πεδίο opcode (instr[6:0]), που αντιπροσωπεύει τον τύπο της εντολής. Οι πιθανές του τιμές ορίζονται ως παράμετροι εντός της μονάδας (opcode_R= 7'b0110011, opcode_I =7'b0010011, opcode_S 7'b0100011 και opcode_B 7'b1100011 για τύπους R,I,S,B αντίστοιχα)
- το πεδίο funct3 (instr[12:14]), με βάση το οποίο ξεχωρίζουν οι διαφορετικές εντολές ίδιου τύπου
- το πεδίο rs1 (instr[15:19]), που περιέχει τη διεύθυνση του πρώτου καταχωρητή πηγής. Κάθε μία από τις εντολές που υποστηρίζονται χρησιμοποιούν τα περιεχόμενα τουλάχιστον ενός καταχωρητή ως τελεστή, επομένως αυτό το πεδίο είναι κοινό για όλες τους.

Τα εναπομείναντα bit του σήματος instr αντιπροσωπεύουν διαφορετικά πράγματα, ανάλογα με την εντολή. Για παράδειγμα, εντολές τύπου R ,S και B χρησιμοποιούν ως δεύτερο τελεστή τα περιεχόμενα ενός δεύτερου καταχωρητή πηγής , τη διεύθυνση τη οποίου αντιπροσωπεύουν τα bit 24-20 (πεδίο rs2). Αντίθετα, κάποιες εντολές τύπου I χρησιμοποιούν μία σταθερή τιμή ως δεύτερο τελεστή, επομένως τα bit που σε άλλες εντολές αποτελούν το πεδίο rs2 χρησιμοποιούνται (μαζί με άλλα bits) για να αναπαραστήσουν τη σταθερή αυτή τιμή.

Επομένως, ενώ η τιμή στην είσοδο readReg1 είναι πάντα έγκυρη, και η έξοδος readData1 χρησιμοποιείται πάντα παρακάτω στο datapath, η τιμή στην είσοδο readReg2 μπορεί να μην αντιπροσωπεύει έναν καταχωρητή πηγής. Αυτό δεν αποτελεί πρόβλημα, διότι η έξοδος readData2 του αρχείου καταχωρητών οδηγείται σε έναν πολυπλέκτη, ο οποίος τη διαβιβάζει στο επόμενο στοιχείο μόνο εφόσον είναι έγκυρη. Αυτός ο πολυπλέκτης ελέγχεται από το σήμα ALUSrc, η πηγή του οποίου αποτελεί αντικείμενο της επόμενης άσκησης.

Όμοια, η τιμή στην είσοδο writeReg δεν αντιπροσωπεύει απαραίτητα έναν καταχωρητή προορισμού (rd), γιατί μόνο οι εντολές τύπου R και I παράγουν

αποτελέσματα που πρέπει να αποθηκευτούν στο αρχείο καταχωρητών (και επομένως μόνο αυτές έχουν πεδίο rd). Όμως, το αρχείο καταχωρητών πραγματοποιεί εγγραφές μόνο εάν το σήμα ελέγχου του, write, είναι ενεργό. Εάν η εντολή που εκτελείται δεν είναι τύπου R ή I, το σήμα ελέγχου write θα απενεργοποιηθεί, και η τιμή στην είσοδο writeReg δε θα έχει καμία επίπτωση. Το write προέρχεται από την ίδια πηγή με το ALUSrc.

Οι εντολές τύπου I, S και B χρησιμοποιούν μία σταθερή τιμή κατά την εκτέλεσή τους, αλλά δεν χρησιμοποιούν τα ίδια bit για να την κωδικοποιήσουν. Στις εντολές I, η σταθερή τιμή αναπαρίσταται από τα bit 31-20 , στις εντολές S από τα bit 31-25 και 11-7, ενώ στις εντολές B χρησιμοποιούνται τα bit 31, 7, 30-25 και 11-8.

Στο επόμενο κομμάτι του datapath, που ονομάζεται Immediate Generation, ένα συνδυαστικό always block αναθέτει στο εσωτερικό σήμα imm12 την σταθερή τιμή, ανάλογα με το πεδίο opcode.

```
always @(instr) begin


  i1=instr[19:15]; //internal net that connects to readReg1
  i2=instr[24:20]; //internal net that connects to readReg2
  i3=instr[11:7]; //internal net that connects to writeReg

  if(instr[6:0]==opcode_I || instr[6:0]==opcode_LW) //imm gen
    imm12=instr[31:20]; //for I types
  else if(instr[6:0]==opcode_S)
    imm12={instr[31:25], instr[11:7]}; //for S types
  else if(instr[6:0]==opcode_B)
    imm12={instr[31],instr[7],instr[30:25],instr[11:8]}; //for B types
  else if(instr[6:0]==opcode_R)
    imm12=12'b000000000000;   //if it's an R type, no immediate value is used, so it doesn't matter what
  else begin                          //imm12 is. Set it to 0 instead of x to avoid unpredictable behaviour.
    $display ("Invalid Opcode at Imm Gen");
    imm12=12'b000000000000;//new addition to the code
  end
  imm32={ {20{imm12[11]}},imm12 };

end
```

Έπειτα το σήμα imm32, που προκύπτει από το imm12 με επέκταση προσήμου, χρησιμοποιείται παρακάτω στο datapath, από την ALU ή από τη λογική διακλάδωσης.

Η ALU είναι το επόμενο θεμελιώδες κομμάτι του datapath. Η πρώτη της είσοδος συνδέεται με την έξοδο readData1 του regfile, μέσω του εσωτερικού σήματος n1. Η δεύτερη είσοδος συνδέεται με την έξοδο ενός πολυπλέκτη, μέσω του n3. Αυτός ο πολυπλέκτης επιλέγει εάν η ALU θα χρησιμοποιήσει την έξοδο του regfile readData2 (με την οποία συνδέεται μέσω του n2), ή το σήμα imm32, με βάση το σήμα ALUSrc.

```

regfile REGFL(.readData1(n1), .readData2(n2), .clk(clk), .readReg1(i1), .readReg2(i2), .writeReg(i3) , .writeData(WriteBackData), .write(RegWrite));
mux MUX1(.chosen(n3), .option0(n2), .option1(imm32), .control(ALUSrc));//If ALUSrc=0, ALU gets rs2, otherwise ALU gets immediate data
ALU ALU_DATAPATH (.result(n4), .zero(Zero), .op1(n1), .op2(n3), .alu_op(ALUCtrl) );

```

```

module mux(
  output wire [31:0] chosen,
  input wire [31:0] option0,
  input wire [31:0] option1,
  input wire control);

  assign chosen=(control) ? (option1) : (option0);

endmodule

```

Το σήμα που συνδέεται στην είσοδο alu_op της ALU, και υπαγορεύει το είδος της πράξης που θα εκτελεστεί , ονομάζεται ALUCtrl, και είναι ένα από τα σήματα εισόδου της μονάδας regfile. Προέρχεται από μία μονάδα ελέγχου που θα υλοποιηθεί στα πλαίσια της άσκησης 5.

Η ALU παράγει 2 εξόδους, την zero και την result. Η λογική “εγγραφής προς τα πίσω” καθορίζει εάν η έξοδος WriteBackData του datapath, που εγγράφεται στο αρχείο καταχωρητών, θα περιέχει το αποτέλεσμα result της ALU ή την έξοδο της μνήμης δεδομένων.

Η result συνδέεται με την έξοδο dAddress της μονάδας datapath. Επιπλέον αποτελεί την δεύτερη είσοδο ενός πολυπλέκτη, ο οποίος λαμβάνει ως πρώτη είσοδο και ως σήμα ελέγχου τα σήματα εισόδου dReadData και MemToReg της datapath αντίστοιχα.

Ανάλογα με την εντολή που εκτελείται, η dAddress μπορεί να περιέχει μία διεύθυνση, και εάν εισαχθεί στη μνήμη δεδομένων, η μνήμη δίνει στη έξοδό της τα αντίστοιχα περιεχόμενα (η μνήμη χρησιμοποιεί ως είσοδο τα 9 χαμηλότερα bits. Το περιβάλλον ανάπτυξης αποκόπτει αυτόματα τα 23 υψηλότερα bits). Αυτή είναι και η προέλευση του σήματος dReadData, και, εάν το σήμα MemToReg είναι ενεργό, τα περιεχόμενα του dReadData δρομολογούνται προς εγγραφή στο αρχείο καταχωρητών.

Σε περίπτωση που εκτελείται συμβατική αριθμητική ή λογική εντολή, το αποτέλεσμα που περιέχει η dAddress δεν είναι διεύθυνση. Συνεπώς το σήμα dReadData δεν είναι έγκυρο. Σε αυτή την περίπτωση, το σήμα MemToReg απενεργοποιείται, και ο πολυπλέκτης επιλέγει τη δεύτερη είσοδό του, δηλαδή το αποτέλεσμα της ALU.

Η δεύτερη έξοδος της ALU, zero, συνδέεται με το σήμα εξόδου Zero του datapath. Όπως θα γίνει εμφανές στην περιγραφή της άσκησης 5, το σήμα Zero συμβάλλει, μέσω μιας μονάδας ελέγχου, στη δημιουργία του σήματος PCSrc, το οποίο αποτελεί είσοδο του datapath.

Το PCSrc υπαγορεύει εάν το πρόγραμμα θα εκτελέσει την αμέσως επόμενη εντολή, ή εάν θα πραγματοποιήσει διακλάδωση. Αυτό επιτυγχάνεται με τον έλεγχο της τιμής ενός βασικού στοιχείου της διαδρομής δεδομένων, του καταχωρητή PC.

Στη μονάδα datapath, ο καταχωρητής PC (Program Counter, μετρητής προγράμματος) δηλώνεται ως έξοδος ,τύπου reg και μεγέθους 32 bit. Περιέχει την τιμή της διεύθυνσης μνήμης εντολών, στην οποία βρίσκεται η τρέχουσα εντολή που εκτελείται.

Η μνήμη εντολών δίνεται έτοιμη από εκφώνηση, και δεν αποτελεί μέρος της μονάδας datapath, που υλοποιείται στη συγκεκριμένη άσκηση. Δέχεται όμως ως είσοδο την έξοδο PC της μονάδας datapath (χρησιμοποιεί μόνο τα 9 χαμηλότερα bits. Το περιβάλλον ανάπτυξης αποκόπτει αυτόματα τα 23 υψηλότερα bits), και παράγει ως έξοδο μία από της εισόδους της. Αποτελείται από 512 καταχωρητές

μήκους 8 bit, που περιέχουν 128 εντολές RISC-V σε δυαδική μορφή (1 εντολή έχει μήκος 32 bit, και αναπαρίσταται από 4 διαδοχικούς καταχωρητές). Δέχεται ως είσοδο addr την τιμή που περιέχει ο PC, και στην ανιούσα ακμή ενός σήματος εισόδου clk, δίνει ως έξοδο dout τους 4 (συνενωμένους) καταχωρητές με διευθύνσεις addr, addr+1, addr+2 και addr+3.

Η αρχική τιμή του PC, στην οποία και επανέρχεται εφόσον ενεργοποιηθεί το σήμα rst (reset), είναι 0x00400000. Ορίζεται εντός της μονάδας datapath η παράμετρος INITIAL_PC, που περιέχει αυτή την αρχική τιμή. Ο PC αρχικοποιείται με ένα initial block, και οι μετέπειτα τιμές του καθορίζονται με χρήση ενός always block, σύγχρονα με την ανιούσα ακμή του σήματος εισόδου clk του datapath.

```
42    initial begin
43      PC=INITIAL_PC;
44    end
45
```

```
82  always @(posedge clk) begin
83      if (rst)
84        PC<=INITIAL_PC;
85
86      else if(loadPC) begin
87        if(PCSrc) //multiplexer
88          PC<=PC+(imm32<<1);//PC + branch_offset (Branch Target)
89        else
90          PC<=PC+4;//PC + 4 (όταν το πρόγραμμα προχωρά στην επόμενη εντολή στη μνήμη)
91      end
92    end
```

Το σήμα imm32 υπέκεινται σε λογική ολίσθηση κατά 1 bit αριστερά, και οδηγείται σε έναν αθροιστή μαζί με την τρέχουσα τιμή του PC. Ένας άλλος αθροιστής υπολογίζει το άθροισμα της τρέχουσας τιμής του PC με το 4. Τα αποτελέσματα αυτών των δύο προσθέσεων οδηγούνται σε έναν πολυπλέκτη που ελέγχεται από το PCSrc. Ο PC ενημερώνεται με το αποτέλεσμα του πολυπλέκτη, εφόσον το σήμα εισόδου του datapath, loadPC, είναι ενεργό.

Η διεύθυνση που περιέχει ο μετρητής προγράμματος PC διοχετεύεται στην είσοδο της δοσμένης από εκφώνηση μνήμης εντολών. Η έξοδος αυτής της μνήμης είναι το σήμα instr, η χρήση του οποίου αναλύθηκε στην αρχή.

Άσκηση 5: Ελεγκτής πολλαπλών κύκλων

Το αντικείμενο αυτής της άσκησης είναι η σχεδίαση μίας μονάδας ελέγχου για τη διαδρομή δεδομένων της άσκησης 4, καθώς και μίας μηχανής κατάστασης (FSM). Υλοποιούνται με τη μονάδα top_proc.

Η διαδρομή δεδομένων χωρίζεται σε 5 καταστάσεις :

- IF (Instruction Fetch): Παροχή του PC στη μνήμη εντολών
- ID (Instruction Decode): Αποκωδικοποίηση της ληφθείσας εντολής και έναρξη πρόσβασης στους καταχωρητές
- EX (Execute): Εκτέλεση της λειτουργίας στην ALU
- MEM (Memory): Εκτέλεση πρόσβασης στη μνήμη (για lw/sw)
- WB (Write Back): Εγγραφή νέων δεδομένων στους καταχωρητές

Κάποια από τα σήματα ελέγχου πρέπει να τίθονται αποκλειστικά κατά τη διάρκεια συγκεκριμένων καταστάσεων, ενώ άλλα είναι ανεξάρτητα από την τρέχουσα κατάσταση.

Τα σήματα ALUOp, ALUSrc, Branch είναι ανεξάρτητα της κατάστασης. Δηλώνονται στη μονάδα ως εσωτερικά σήματα, και η τιμή τους εξαρτάται από το opcode της τρέχουσας εντολής.

Το σήμα ALUCtrl παράγεται από το ALUOp και από τα πεδία funct3 και funct7 της κάθε εντολής.

```
always @(opcode) begin    //CONTROL UNIT (signals independent of the FSM state)
  case (opcode)
    opcode_R : begin
      ALUOp=3'b000;
      ALUSrc=1'b0;
      Branch=1'b0;
    endS
    opcode_I : begin
      ALUOp=3'b001;
      ALUSrc=1'b1;
      Branch=1'b0;
    end
    opcode_S : begin
      ALUOp=3'b010;
      ALUSrc=1'b1;
      Branch=1'b0;
    end
    opcode_B : begin
      ALUOp=3'b011;
      ALUSrc=1'b0;
      Branch=1'b1;
    end
    opcode_LW : begin
      ALUOp=3'b100;
      ALUSrc=1'b1;
      Branch=1'b0;
    end
    default : begin
      ALUOp=3'b001;   //invalid opcodes are handled by the ALU like type I, with an imm field of 0 (imm gen in datapath assigns zero to imm12 if opcode is invalid)
      ALUSrc=1'b1;
      Branch=1'b0;
    end
  endcase

end
```

```
 always @(ALUOp,funct7,funct3) begin  //ALU CONTROL
   casex({ALUOp,funct7,funct3})
     12'b0000111, 12'b001x111 : ALUCtrl = 4'b0000; //AND
     12'b0000110, 12'b001x110 : ALUCtrl = 4'b0001; //OR
     12'b011x000, 12'b0001000 : ALUCtrl = 4'b0110; //SUBTRACTION
     12'b0000010, 12'b001x010 : ALUCtrl = 4'b0100; //LESS THAN
     12'b0000101, 12'b0010101 : ALUCtrl = 4'b1000; //SHIFT RIGHT LOGICAL
     12'b0000001, 12'b0010001 : ALUCtrl = 4'b1001; //SHIFT LEFT LOGICAL
     12'b0001101, 12'b0011101 : ALUCtrl = 4'b1010; //SHIFT RIGHT ARITHMETIC
     12'b0000100, 12'b001x100 : ALUCtrl = 4'b0101; //XOR
     12'b100x010: ALUCtrl=4'b0010;                 //LW requires addition
     default : ALUCtrl = 4'b0010;                  //ADDITION (anything valid left requires addition. Anything invalid needs a default.)
   endcase                          //Invalid commands are treated like a type I with imm32=0. it's important for MemWrite and RegWrite to be low, so no changes
 end                                //are made to the regfile or the data memory by a nonsense command
```

Τα υπόλοιπα σήματα ελέγχου ενεργοποιούνται μόνο σε ορισμένες καταστάσεις. Γι΄ αυτό το λόγο οι τιμές τους καθορίζονται μέσα στο OUTPUT_LOGIC always block της μηχανής πεπερασμένης κατάστασης (FSM). Συνολικά η FSM απαρτίζεται από 3 always blocks.

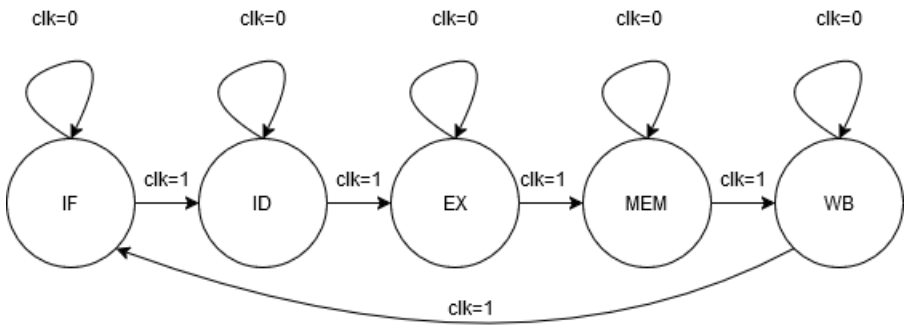
```
132   always @(current_state or instr or dReadData)
133     begin: OUTPUT_LOGIC         //what are the outputs of each state?
134       case (current_state)
135         IF: begin
136           loadPC=1'b0;
137           MemRead=1'b0;
138           MemWrite=1'b0;
139           MemToReg=1'b0;
140           RegWrite=1'b0;
141         end
142
143         ID: begin
144           loadPC=1'b0;
145           MemRead=1'b0;
146           MemWrite=1'b0;
147           MemToReg=1'b0;
148           RegWrite=1'b0;
149         end
150
151         EX:begin
152           loadPC=1'b0;
153           MemRead=1'b0;
154           MemWrite=1'b0;
155           MemToReg=1'b0;
156           RegWrite=1'b0;
157         end
158
159         MEM: begin
160           if(opcode==opcode_LW) //set MemRead to high for load instructions during MEM state
161             MemRead=1'b1;
162           else
163             MemRead=1'b0;
164           if (opcode==opcode_S) //set MemWrite to high for store instructions during MEM state
165             MemWrite=1'b1;
166           else
167             MemWrite=1'b0;
168
169           MemToReg=1'b0;
170           loadPC=1'b0;
171           RegWrite=1'b0;
172         end
173
173
174         WB: begin
175           if(opcode==opcode_LW || opcode==opcode_I || opcode==opcode_R)
176             RegWrite=1'b1;  //set RegWrite to high for instructions with a destination register (rd)
177           else              //during SW state
178             RegWrite=1'b0;  //set RegWrite to low for other instructions, or if the opcode is invalid
179           if(opcode==opcode_LW)
180             MemToReg=1'b1;
181           else
182             MemToReg=1'b0;
183
184           loadPC=1'b1;
185           MemRead=1'b0;
186           MemWrite=1'b0;
187         end
188
189         default:begin
190           loadPC=1'b0;
191           MemRead=1'b0;
192           MemWrite=1'b0;
193           MemToReg=1'b0;
194           RegWrite=1'b0;
195         end
196
197       endcase
198     end
199
```

Η αποθήκευση τρέχουσας κατάστασης και η μετάβαση από τη μία κατάσταση στην άλλη υπαγορεύεται από τα STATE_MEMORY και NEXT_STATE_LOGIC always blocks της FSM.

```
107  //FSM
108  always @(posedge clk)
109    begin: STATE_MEMORY
110      if(!rst)  //synchronous reset, δεν ειναι στη λιστα ευαισθησιας
111        current_state<=IF;
112      else
113        current_state<=next_state;
114    end
115
116  always @(current_state or instr or dReadData)
117    begin: NEXT_STATE_LOGIC     //what needs to happen for the state to change? inputs? clock?
118      case (current_state)
119        IF: next_state=ID;
120        ID: next_state=EX;
121        EX: begin
122          next_state=MEM;
123          //if (instr[6:0]~=opcode_S && instr[6:0]~=opcode_LW) next_state=WB; //only LW, SW use the MEM state
124          //else next_state=MEM;     //do we have to skip it or can the control signals ensure correct function?
125        end                         //This shaves off 1 cycle for all commands except LW, SW
126        MEM: next_state=WB;
127        WB: next_state=IF;
128        default: next_state=IF;
129      endcase
130    end
131
```

Η μετάβαση από τη μία κατάσταση στην επόμενη γίνεται κάθε φορά που αλλάζει το current_state. Αυτό συμβαίνει όταν ολοκληρώνεται ένας κύκλος ρολογιού.

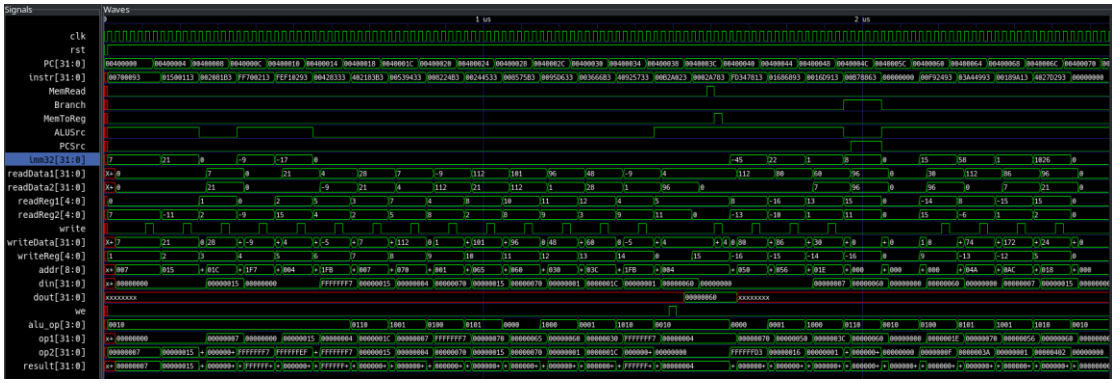
Οι καταστάσεις αλλάζουν διαδοχικά με τη σειρά ως εξής: IF->ID->EX->MEM->WB.



Κάθε κατάσταση διαρκεί 1 κύκλο ρολογιού, επομένως μία εντολή χρειάζεται 5 κύκλους ρολογιού για να εκτελεστεί.

Τα σήματα ελέγχου διασφαλίζουν την ορθή λειτουργία, διαβιβάζοντας μόνο έγκυρες τιμές στα επόμενα στάδια. Για παράδειγμα, η κατάσταση MEM αφορά μόνο τις εντολές LW και SW. Για οποιαδήποτε άλλη εντολή, το αποτέλεσμα της ALU και η δεύτερη έξοδος του regfile δεν αντιπροσωπεύουν κάποια διεύθυνση μνήμης, ή δεδομένα που πρέπει να εγγραφούν στη μνήμη δεδομένων. Επομένως, στην κατάσταση MEM, αυτές οι τιμές είναι μη έγκυρες είσοδοι για τη μνήμη. Όμως, χάρη στα σήματα MemToReg και MemWrite, τα οποία τίθενται μόνο για τις εντολές LW και SW, τα μη έγκυρα δεδομένα που προκύπτουν στην έξοδο της μνήμης δε χρησιμοποιούνται, και η μνήμη δεν επιτρέπει εγγραφή.

Μία άλλη εναλλακτική είναι να παραλείπεται το στάδιο MEM για όλες τις εντολές εκτός από τις LW και SW. Με αυτό τον τρόπο εξοικονομείται ένας κύκλος ρολογιού για κάθε εντολή που δεν απαιτεί πρόσβαση στη μνήμη.



(Σημείωση: οι δυνατότητες του EDA Playground IDE αποδείχθηκαν περιορισμένες. Για την απεικόνιση των κυματομορφών αυτής της άσκησης, χρησιμοποιήθηκε το GTKWave σε δευτερεύον υπολογιστή.)

Οι τιμές του σήματος instr[31:0] αλλάζουν κάθε 5 κύκλους, και εάν εισαχθούν στον αποκωδικοποιητή εντολών https://luplab.gitlab.io/rvcodecjs/ , δίνουν τις παρακάτω εντολές:

1. addi x1, x0, 7 (δεκαεξαδική μορφή: 0070_0093)
2. addi x2, x0, 21 (δεκαεξαδική μορφή: 0150_0113)
3. add x3, x1, x2 (δεκαεξαδική μορφή: 0020_81b3)
4. addi x4, x0, -9 (δεκαεξαδική μορφή: ff70_0213)
5. addi x5, x2, -17 (δεκαεξαδική μορφή: fef1_0293)
6. add x6, x5, x4 (δεκαεξαδική μορφή: 0042_8333)
7. sub x7, x3, x2 (δεκαεξαδική μορφή: 4021_83b3)
8. sll x8, x7, x5 (δεκαεξαδική μορφή: 0053_9433)
9. slt x9, x4, x8 (δεκαεξαδική μορφή: 0082_24b3)
10. xor x10, x8, x2 (δεκαεξαδική μορφή: 0024_4533)
11. and x11, x10, x8 (δεκαεξαδική μορφή: 0085_75b3)
12. srl x12, x11, x9 (δεκαεξαδική μορφή: 0095_d633)
13. or x13, x12, x3 (δεκαεξαδική μορφή: 0036_66b3)
14. sra x14, x4, x9 (δεκαεξαδική μορφή: 4092_5733)
15. sw x11, 0(x5) (δεκαεξαδική μορφή: 00b2_a023)
16. lw x15, 0(x5) (δεκαεξαδική μορφή: 0002_a783)
17. andi x16, x8, -45 (δεκαεξαδική μορφή: fd34_7813)
18. ori x17, x16, 22 (δεκαεξαδική μορφή: 0168_6893
19. srli x18, x13, 1 (δεκαεξαδική μορφή: 0016_d913
20. beq x15, x11, 16 (δεκαεξαδική μορφή: 00b7_8863
21. hex 0000_0000 (αυτή η μορφή δεν αντιστοιχεί σε καμία εντολή. Παραλείπεται λόγω BEQ)
22. hex 0000_0000 (αυτή η μορφή δεν αντιστοιχεί σε καμία εντολή. Παραλείπεται λόγω BEQ)
23. hex 0000_0000 (αυτή η μορφή δεν αντιστοιχεί σε καμία εντολή. Παραλείπεται λόγω BEQ)
24. hex 0000_0000 (αυτή η μορφή δεν αντιστοιχεί σε καμία εντολή. Αντιμετωπίζεται από το παρόν datapath ως τύπος I με μηδενικό σταθερό μέρος, και η ALU εκτελεί πρόσθεση 0+0. Ουσιαστικά αντιμετωπίζεται ως addi

x0, x0, 0 (εντολή do-nothing), με ανενεργά σήματα ελέγχου για να μην πραγματοποιηθεί καμία χρήση ή εγγραφή)

25. slti x9, x18, 15 //hex: 00f9_2493
26. xori x19, x8, 58 //hex: 03a4_4993
27. slli x20, x17, 1 //hex: 0018_9a13
28. srai x5, x15, 2 //hex: 4027_d293

29-128. hex 0000_0000 (αντιμετωπίζονται ως do-nothing εντολές)

Εάν χρησιμοποιηθεί ο ίδιος αποκωδικοποιητής για τις δυαδικές εντολές που δίνονται στο αρχείο rom_bytes, προκύπτουν τα ίδια αποτελέσματα.

Παρατηρείται ότι τα σήματα ελέγχου παίρνουν τις αναμενόμενες τιμές.

Οι τιμές καταχωρητών και της μνήμης που προκύπτουν είναι επίσης ορθές. Ο έλεγχός τους έγινε με τον διερμηνέα RISC-V https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/ με αντικατάσταση των εντολών 0000_0000 από τις addi x0, x0, 0 .

Memory Address	Decimal	Hex	Binary
0x00000000	0	0x00000000	0b00000000000000000000000000000000
0x00000004	96	0x00000060	0b00000000000000000000000001100000
0x00000008	0	0x00000000	0b00000000000000000000000000000000
0x0000000c	0	0x00000000	0b00000000000000000000000000000000
0x00000010	0	0x00000000	0b00000000000000000000000000000000
0x00000014	0	0x00000000	0b00000000000000000000000000000000
0x00000018	0	0x00000000	0b00000000000000000000000000000000
0x0000001c	0	0x00000000	0b00000000000000000000000000000000
0x00000020	0	0x00000000	0b00000000000000000000000000000000
0x00000024	0	0x00000000	0b00000000000000000000000000000000

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	7	0x00000007	0b00000000000000000000000000000111
0	x2 (sp)	21	0x00000015	0b00000000000000000000000000010101
0	x3 (gp)	28	0x0000001c	0b00000000000000000000000000011100
0	x4 (tp)	-9	0xfffffff7	0b11111111111111111111111111110111
0	x5 (t0)	24	0x00000018	0b00000000000000000000000000011000
0	x6 (t1)	-5	0xfffffffb	0b11111111111111111111111111111011
0	x7 (t2)	7	0x00000007	0b00000000000000000000000000000111
0	x8 (s0/fp)	112	0x00000070	0b00000000000000000000000001110000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	101	0x00000065	0b00000000000000000000000001100101
0	x11 (a1)	96	0x00000060	0b00000000000000000000000001100000
0	x12 (a2)	48	0x00000030	0b00000000000000000000000000110000
0	x13 (a3)	60	0x0000003c	0b00000000000000000000000000111100
0	x14 (a4)	-5	0xfffffffb	0b11111111111111111111111111111011
0	x15 (a5)	96	0x00000060	0b00000000000000000000000001100000
0	x16 (a6)	80	0x00000050	0b00000000000000000000000001010000
0	x17 (a7)	86	0x00000056	0b00000000000000000000000001010110
0	x18 (s2)	30	0x0000001e	0b00000000000000000000000000011110
0	x19 (s3)	74	0x0000004a	0b00000000000000000000000001001010
0	x20 (s4)	172	0x000000ac	0b00000000000000000000000010101100
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000

0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
0	x28 (t3)	0	0x00000000	0b00000000000000000000000000000000
0	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000