Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής Ι

Μπαρμπαγιάννος Βασίλειος

ΑΕΜ: 10685

Εξάμηνο: Χειμερινό 2024 – 2025

Υπεύθυνος Καθηγητής: Βασίλης Παυλίδης

Βοηθός Διδασκαλίας: Αριστοτέλης Τσεκούρας

Άσκηση 1

Η πρώτη άσκηση μας ζητάει να υλοποιήσουμε μια 32-bit ALU. Η ALU πρέπει να υλοποιεί τις ακόλουθες πράξεις: προσημασμένη πρόσθεση, προσημασμένη αφαίρεση, λογικό AND, λογικό OR, λογικό XOR, σύγκριση "Μικρότερο από" και τρεις διαφορετικές πράξεις ολίσθησης.

Η πράξη η οποία θα εκτελεστεί μεταξύ των δύο τελεστών ‘op1’ και ‘op2’ θα καθορίζεται από τον πολυπλέκτη, με το σήμα ‘alu_op’.

Όνομα θύρας	**Κατεύθυνση**	**Πλάτος [αρ. bit]**	**Σκοπός**
op1	Είσοδος	32	Τελεστής 1 σε συμπλήρωμα ως προς 2
op2	Είσοδος	32	Τελεστής 2 σε συμπλήρωμα ως προς 2
alu_op	Είσοδος	4	Δείχνει ποια λειτουργία πρέπει να εκτελεστεί
zero	Έξοδος	1	Δείχνει πότε το αποτέλεσμα της ALU είναι μηδέν
result	Έξοδος	32	Αποτέλεσμα

Πίνακας 1.1

Η λειτουργία που εκτελεί η ALU βασίζεται στο σήμα εισόδου 'alu_op' τεσσάρων bit, όπως ορίζεται στον ακόλουθο πίνακα:

alu_op	**Πράξη**	**Αποτέλεσμα**
0000	Λογική AND	op1 & op2
0001	Λογική OR	op1 | op2
0010	Πρόσθεση	op1 + op2
0110	Αφαίρεση	op1 - op2
0100	Μικρότερο από	op1 < op2
1000	Λογική ολίσθηση δεξιά κατά op2 bits	op1 >> op2[4:0]
1001	Λογική ολίσθηση αριστερά κατά op2 bits	op1 << op2[4:0]
1010	Αριθμητική ολίσθηση δεξιά κατά op2 bits	op1 >>> op2[4:0]
0101	Λογική XOR	op1 ^ op2

Πίνακας 1.2

Προσοχή: Η πράξη “Μικρότερο από” πρέπει να γίνει πάνω σε προσημασμένους αριθμούς. Επίσης για την αριθμητική ολίσθηση, το op1 θα πρέπει να μετατραπεί σε προσημασμένο αριθμό και το αποτέλεσμα μετά ξανά σε μη προσημασμένο.

- Ξεκινάμε με τον ορισμό του module ‘alu’.

Οι είσοδοι του module είναι:

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

- op1(32-bit): πρώτος τελεστής
- op2(32-bit): δεύτερος τελεστής
- alu_op(4-bit): επιλογή πράξης

Οι έξοδοι του module είναι:

- result(32-bit): το αποτέλεσμα της πράξης
- zero(1-bit): δείχνει πότε το αποτέλεσμα της ALU είναι 0.

- Τώρα θα ορίσουμε τις σταθερές του σήματος ‘alu_op’ βάσει του πίνακα 1.2:

```
  // Ορίζουμε τις σταθερές για τις πράξεις.
parameter [3:0] ALU_AND = 4'b0000,
                ALU_OR = 4'b0001,
                ALU_ADD = 4'b0010,
                ALU_SUB = 4'b0110,
                ALU_LESS = 4'b0100,
                ALU_SHIFTR = 4'b1000,
                ALU_SHIFTL = 4'b1001,
                ALU_ARITHMR = 4'b1100,
                ALU_XOR = 4'b0101;
```

- Έπειτα θα υλοποιήσουμε τον πολυπλέκτη ο οποίος θα εκτελεί την πράξη με βάση την τιμή του σήματος ‘alu_op’.

```
// Πολυπλέκτης (εντολή case) για επιλογή της πράξης.
always @(*)
  begin
    case (alu_op) // Ανάλογα την τιμή του alu_op γίνεται και
                  // η αντίστοιχη πράξη.
      ALU_AND: result = op1 & op2;
      ALU_OR:  result = op1 | op2;
      ALU_ADD: result = op1 + op2;
      ALU_SUB: result = op1 - op2;
      ALU_LESS:  result = (op1 < op2) ? 32'b1 : 32'b0;
      ALU_SHIFTR: result = op1 >> op2[4:0];
      ALU_SHIFTL: result = op1 << op2[4:0];
      ALU_ARITHMR: result = op1 >>> op2[4:0];
      ALU_XOR: result = op1 ^ op2;
      default: result = 32'b0;
    endcase
  end
```

Μέσα σε ένα always @(*) μπλοκ, χρησιμοποιούμε την εντολή case για την επιλογή της πράξης.

Το default case της ALU είναι να δείχνει 0 (result = 32'b0).

- Τέλος, το zero πρέπει να δείχνει 1 αν το αποτέλεσμα είναι 0, αλλιώς 0.

```
36      // Υπολογισμός του zero.
37    always @(*)
38      begin // Όταν το result είναι 0, το zero είναι λογικό 1.
39        zero = (result == 32'b0) ? 1'b1 : 1'b0;
40      end
41  endmodule
42
```

Άσκηση 2

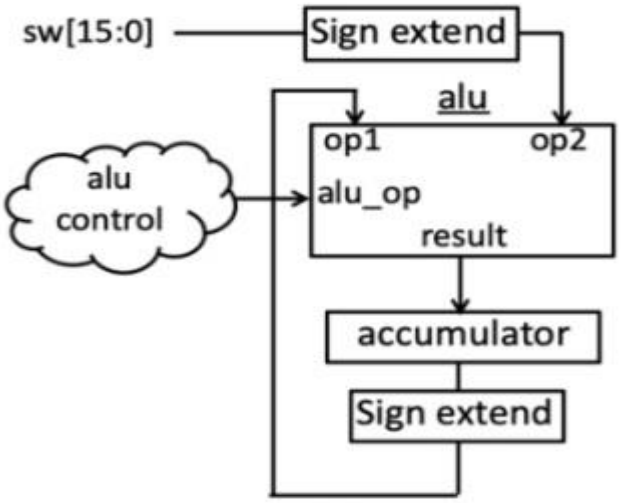
Στην 2η άσκηση θα σχεδιάσουμε μια αριθμομηχανή με τη βοήθεια της ALU της προηγούμενης άσκησης. Η αριθμομηχανή θα διατηρεί μια τρέχουσα τιμή της σε ένα συσσωρευτή 16-bit καταχωρητή και θα επιτρέπει στο χρήστη να ενημερώνει την τιμή υλοποιώντας οποιαδήποτε από τις αριθμητικές και λογικές συναρτήσεις που παρέχει η ALU.

- Το module calc πρέπει να έχει τις κάτωθι θύρες:

Όνομα θύρας	**Κατεύθυνση**	**Πλάτος [αρ. bit]**	**Σκοπός**
clk	Είσοδος	1	Ρολόι
btnc	Είσοδος	1	Κεντρικό πλήκτρο
btnl	Είσοδος	1	Αριστερό πλήκτρο
btnu	Είσοδος	1	Πάνω πλήκτρο
btnr	Είσοδος	1	Δεξί πλήκτρο
btnd	Είσοδος	1	Κάτω πλήκτρο
sw	Είσοδος	16	Διακόπτες για την εισαγωγή δεδομένων
led	Έξοδος	16	LED για την έξοδο του συσσωρευτή

Πίνακας 2.1

- Η σχέση της ALU και του συσσωρευτή πρέπει να είναι όπως φαίνεται στο ακόλουθο σχήμα:



Σχήμα 2.1

❖ Ο καταχωρητής 16-bit ονομάζεται accumulator και πρέπει να σχεδιαστεί ως εξής:

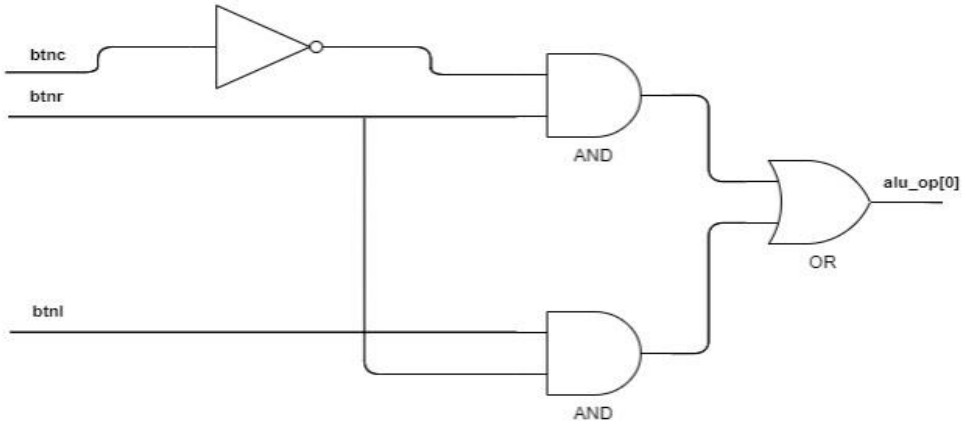
● Ο accumulator θα πρέπει να συνδεθεί με την είσοδο του ρολογιού.
● Ο accumulator θα πρέπει να μηδενίζεται σύγχρονα με το πάτημα του **btnu**.
● Η είσοδος του accumulator θα πρέπει να είναι τα 16 χαμηλότερα bit της εξόδου αποτελέσματος 32 bit της ALU.
● Ο accumulator θα πρέπει να ενημερώνεται σύγχρονα κάθε φορά που πατιέται το "down" **(btnd)**.
● Συνδέστε την τιμή του accumulator με τις εξόδους LED.

❖ Συνδέστε τις εισόδους στην ALU ως εξής:

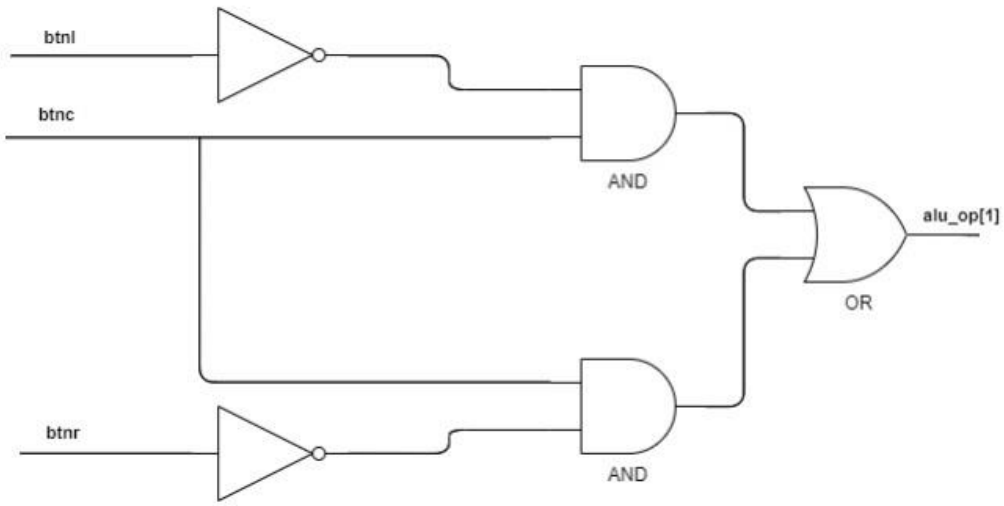
● Δημιουργήστε ένα σήμα 32-bit που είναι μια έκδοση με επέκταση προσήμου του accumulator 16-bit. Συνδέστε αυτό το σήμα στην είσοδο '**op1**' της ALU.
● Δημιουργήστε ένα σήμα 32-bit που είναι μια έκδοση με επέκταση προσήμου των εισόδων του διακόπτη 16-bit. Συνδέστε αυτό το σήμα στην είσοδο '**op2**' της ALU.
● Συνδέστε τα χαμηλότερα 16 bit της εξόδου "**result**" στην είσοδο του accumulator (όπως αναφέρθηκε παραπάνω).
● Δημιουργήστε ένα νέο σήμα για το '**alu_op**' καθώς και τη λογική για αυτό το σήμα όπως αναλύεται παρακάτω (σχήματα 2.2 – 2.5).

Σημείωση: Μπορούμε να δημιουργήσουμε ένα σήμα 32-bit με επέκταση προσήμου χρησιμοποιώντας τον τελεστή concatenation της Verilog (δηλ., επαναλαμβάνουμε το κορυφαίο bit του σήματος που επεκτείνουμε με πρόσημο όσες φορές χρειάζεται).

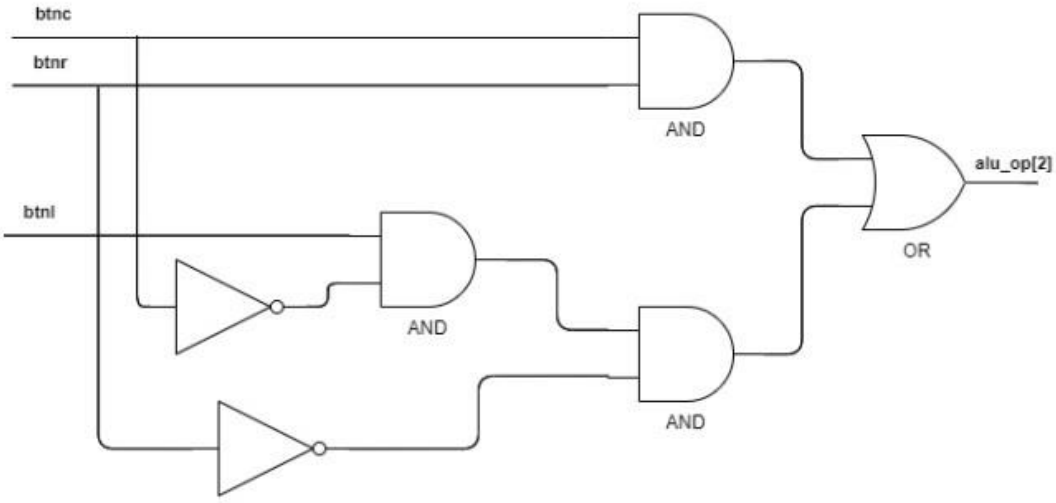
Το σήμα '**alu_op**' καθορίζει ποια πράξη της ALU θα εκτελεστεί. Θα καθορίσουμε ποια λειτουργία θα εκτελέσουμε με βάση την τιμή των τριών πλήκτρων: **btnl, btnc** και **btnr.** Θα πρέπει να δημιουργήσουμε το συνδυαστικό κύκλωμα των σχημάτων 2.2 – 2.5 που να παράγει το κατάλληλο σήμα '**alu_op**' με βάση την τιμή αυτών των τριών πλήκτρων.



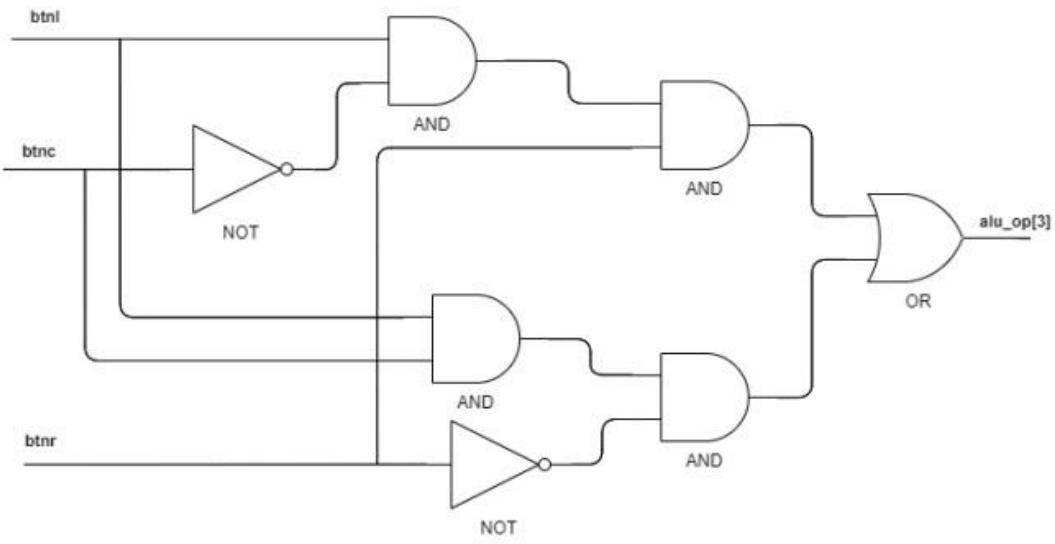
Σχήμα 2.2: Παραγωγή του alu_op[0] μέσω των btnr, btnl, btnc.



Σχήμα 2.3: Παραγωγή του alu_op[1] μέσω των btnr, btnl, btnc.



Σχήμα 2.4: Παραγωγή του alu_op[2] μέσω των btnr, btnl, btnc.



Σχήμα 2.5: Παραγωγή του alu_op[3] μέσω των btnr, btnl, btnc.

- Πρώτα ορίζουμε το module ‘calc’.

```
`timescale 1ns/1ps

module calc (output wire [15:0] led, // LED για την έξοδο του συσσωρευτή.
             input wire clk, // Ρολόι
             input wire btnc, // Κεντρικό πλήκτρο.
             input wire btnl, // Αριστερό πλήκτρο.
             input wire btnu, // Πάνω πλήκτρο.
             input wire btnr, // Δεξιό πλήκτρο.
             input wire btnd, // Κάτω πλήκτρο.
             input wire [15:0] sw); // Διακόπτες για την εισαγωγή δεδομένων.

  reg [15:0] accumulator; // Καταχωρητής 16 bit.
  wire [3:0] alu_op; // Επιλογή πράξης alu.
  wire [31:0] alu_result; // Το αποτέλεσμα της alu.

  wire [31:0] op1_extended, op2_extended;

  // Επέκταση προσήμου.
  assign op1_extended = {{16{accumulator[15]}}, accumulator};
  assign op2_extended = {{16{sw[15]}}, sw};

  // Σύνδεση του calc_enc.
  calc_enc encoder(.btnl(btnl), .btnr(btnr), .btnc(btnc), .alu_op(alu_op));

```

Εντός του module ορίζουμε το σήμα accumulator (16-bit) για να κρατάει (άρα είναι τύπου reg) τα περιεχόμενα της τρέχουσας τιμής της αριθμομηχανής, το σήμα alu_op (4-bit) για την επιλογή της πράξης που θα εκτελέσει η ALU, τα σήματα op1_extended και op_2extended (32-bit) που είναι η έκδοση με επέκταση προσήμου των σημάτων accumulator και sw αντίστοιχα, και τέλος το σήμα alu_result (32-bit) για το αποτέλεσμα της πράξης της ALU.

Η επέκταση του προσήμου πραγματοποιείται με την επανάληψη του πιο σημαντικού bit (δηλαδή το bit 15, MSB), του κάθε σήματος, 16 φορές ώστε να πάρουμε σήμα 32-bit.

Έπειτα συνδέουμε τα modules ‘calc_enc’ (περιγράφεται πιο κάτω) και ‘alu’ (η περιγραφή του δόθηκε στην 1η άσκηση).

```
  // Σύνδεση της alu.
  alu alu(.op1(op1_extended), .op2(op2_extended), .alu_op(alu_op),
.result(alu_result));

  // Λειτουργία του accumulator.
  always @(posedge clk) // Συνδέεται με την είσοδο του ρολογιού.
    begin
      if (btnu) // Μηδενίζεται σύγχρονα με το πάτημα του btnu.
        accumulator <= 16'b0;
      else if (btnd) // Ενημερώνεται σύγχρονα με το πάτημα του btnd.
        accumulator <= alu_result[15:0]; // Τα 16 χαμηλότερα bits του alu_result.
    end

  // Αντιγραφή του accumulator στην έξοδο LED.
  assign led = accumulator;

endmodule

```

Στο module ‘alu’ ως είσοδο στα σήματα op1 και op2 βάζουμε τα επεκτεταμένα σήματα op1_extended και op2_extended αντίστοιχα.

Σε κάθε θετική ακμή του ρολογιού, ο accumulator λειτουργεί ως εξής:

- Αν πατηθεί το πλήκτρο btnu (δηλ. btnu = 1) τότε ο accumulator μηδενίζεται σύγχρονα.
- Αν πατηθεί το πλήκτρο btnd (δηλ. btnd = 1) τότε ο accumulator ενημερώνεται σύγχρονα με τα 16 λιγότερα σημαντικά bit της εξόδου της ALU.

Στο τέλος, απλώς η τιμή του accumulator αντιγράφεται στα LED’s.

- Τώρα θα ορίσουμε το module ‘calc_enc’. Είναι ένα συνδυαστικό κύκλωμα που παίρνει ως είσοδο τα σήματα btnl, btnr και btnc και παράγει ως έξοδο το 4-bit σήμα alu_op, το οποίο και καθορίζει ποια πράξη της ALU θα εκτελεστεί.

```
module calc_enc (output wire [3:0] alu_op, // Έξοδος alu_op.
                 input wire btnl, btnr, btnc); // Είσοδοι πλήκτρων.

  // Υπολογισμός του alu_op[0].
  wire and1_0, and2_0;
  assign and1_0 = ~btnc & btnr;
  assign and2_0 = btnl & btnr;
  assign alu_op[0] = and1_0 | and2_0;

  // Υπολογισμός του alu_op[1].
  wire and1_1, and2_1;
  assign and1_1 = ~btnl & btnc;
  assign and2_1 = btnc & ~btnr;
  assign alu_op[1] = and1_1 | and2_1;

  // Υπολογισμός του alu_op[2].
  wire and1_2, and2_2, and3_2;
  assign and1_2 = btnc & btnr;
  assign and2_2 = btnl & ~btnc
  assign and3_2 = and2_2 & ~btnr
  assign alu_op[2] = and1_2 | and2_3;

  // Υπολογισμός του alu_op[3].
  wire and1_3, and2_3, and3_3, and4_3;
  assign and1_3 = btnl & ~btnc;
  assign and2_3 = and1_3 & btnr;
  assign and3_3 = btnl & btnc
  assign and4_3 = and3_3 & ~btnr
  assign alu_op[3] = and2_3 & and4_3;

endmodule
```

Ο υπόλοιπος κώδικας του module υπολογίζει το κάθε bit του σήματος alu_op σύμφωνα με τη συνδυαστική λογική των σχημάτων 2.2-2.5.

- Ο κώδικας για το testbench σύμφωνα με τον πίνακα 2.2 είναι:

```
`timescale 1ns/1ps

module calc_tb;

    // Είσοδοι
    reg clk;
    reg btnc, btnl, btnu, btnr, btnd;
    reg [15:0] sw;

    // Έξοδος
    wire [15:0] led;

    // Σύνδεση με το DUT (Device Under Test)
    calc dut (
        .clk(clk),
        .btnc(btnc),
        .btnl(btnl),
        .btnu(btnu),
        .btnr(btnr),
        .btnd(btnd),
        .sw(sw),
        .led(led)
    );

    // Ρολόι
    always #5 clk = ~clk; // Περίοδος 10ns (100 MHz)
```

```
    initial begin
        $dumpfile("calc_tb.vcd"); // Αρχείο waveforms
        $dumpvars(0, calc_tb);    // Παρακολούθηση μεταβλητών

        // Αρχικοποίηση
        clk = 0;
        btnc = 0; btnl = 0; btnu = 0; btnr = 0; btnd = 0;
        sw = 16'h0000;
        #10;

        // Reset του accumulator
        btnu = 1; #10;
        btnu = 0; #10;
        $display("Reset: led = %h (αναμενόμενο: 0x0)", led);

        // Test 1: ADD (acc = 0x0 + sw = 0x354a)
        sw = 16'h354a; btnl = 0; btnc = 1; btnr = 0; btnd = 1;
        #10; btnd = 0; #10;
        $display("Test 1 (ADD): led = %h (αναμενόμενο: 354a)", led);

        // Test 2: SUB (acc = 0x354a - sw = 0x1234)
        sw = 16'h1234; btnl = 1; btnc = 1; btnr = 0; btnd = 1;
        #10; btnd = 0; #10;
        $display("Test 2 (SUB): led = %h (αναμενόμενο: 2316)", led);

        // Test 3: OR (acc = 0x2316 | sw = 0x1001)
        sw = 16'h1001; btnl = 0; btnc = 0; btnr = 1; btnd = 1;
        #10; btnd = 0; #10;
        $display("Test 3 (OR): led = %h (αναμενόμενο: 3317)", led);

        // Test 4: AND (acc = 0x3317 & sw = 0x0f0f)
        sw = 16'h0f0f; btnl = 0; btnc = 0; btnr = 0; btnd = 1;
        #10; btnd = 0; #10;
        $display("Test 4 (AND): led = %h (αναμενόμενο: 3010)", led);
```

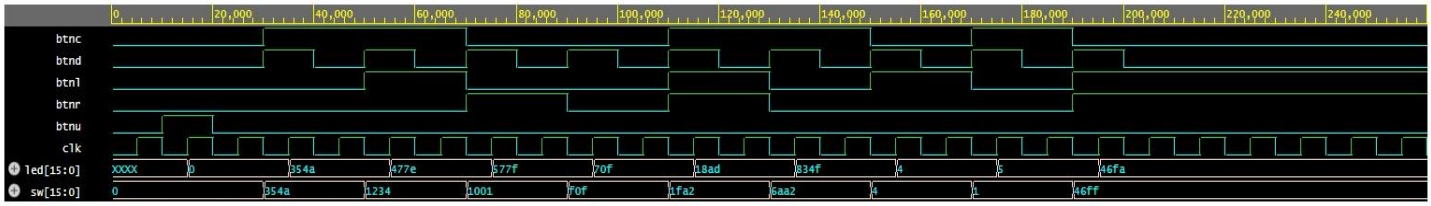
```
62
63          // Test 5: XOR (acc = 0x3010 ^ sw = 0x1fa2)
64          sw = 16'h1fa2; btnl = 1; btnc = 1; btnr = 1; btnd = 1;
65          #10; btnd = 0; #10;
66          $display("Test 5 (XOR): led = %h (αναμενόμενο: 2fb2)", led);
67
68          // Test 6: ADD (acc = 0x2fb2 + sw = 0x6aa2)
69          sw = 16'h6aa2; btnl = 0; btnc = 1; btnr = 0; btnd = 1;
70          #10; btnd = 0; #10;
71          $display("Test 6 (ADD): led = %h (αναμενόμενο: 9a54)", led);
72
73          // Test 7: Logical Shift Left (acc = 0x9a54 << 0x0004)
74          sw = 16'h0004; btnl = 1; btnc = 0; btnr = 0; btnd = 1;
75          #10; btnd = 0; #10;
76          $display("Test 7 (Logical Shift Left): led = %h (αναμενόμενο: a540)",
    led);
77
78          // Test 8: Arithmetic Shift Right (acc = 0xa540 >>> 0x0001)
79          sw = 16'h0001; btnl = 0; btnc = 1; btnr = 0; btnd = 1;
80          #10; btnd = 0; #10;
81          $display("Test 8 (Arithmetic Shift Right): led = %h (αναμενόμενο: d2a0)",
    led);
82
83          // Test 9: Less Than (acc = 0xd2a0 < sw = 0x46ff)
84          sw = 16'h46ff; btnl = 1; btnc = 0; btnr = 1; btnd = 1;
85          #10; btnd = 0; #10;
86          $display("Test 9 (Less Than): led = %h (αναμενόμενο: 0001)", led);
87
88          #50 $finish;
89      end
90
91  endmodule
92
```

btnl, btnc, btnr (input)	Previous value (acc.)	Switches (input)	Function in ALU	Expected Result
(btnu for reset)	xxxx	xxxx	Reset	0x0
0,1,0	0x0	0x354a	ADD	0x354a
0,1,1	0x354a	0x1234	SUB	0x2316
0,0,1	0x2316	0x1001	OR	0x3317
0,0,0	0x3317	0xf0f0	AND	0x3010
1,1,1	0x3010	0x1fa2	XOR	0x2fb2
0,1,0	0x2fb2	0x6aa2	ADD	0x9a54
1,0,1	0x9a54	0x0004	Logical Shift Left	0xa540
1,1,0	0xa540	0x0001	Shift Right Arithmetic	0xd2a0
1,0,0	0xd2a0	0x46ff	Less Than	0x0001

Προσοχή: Η αριθμομηχανή δεν θα υποστηρίζει λογική ολίσθηση προς τα δεξιά.

Πίνακας 2.2

Το παραπάνω testbench παράγει την ακόλουθη κυματομορφή:



Note: To revert to EPWave opening in a new browser window, set that option on your profile page.

Το testbench επίσης εκτυπώνει το εξής αποτέλεσμα:

```
Reset: led = 0000 (??????????: 0x0)
Test 1 (ADD): led = 354a (??????????: 354a)
Test 2 (SUB): led = 477e (??????????: 2316)
Test 3 (OR): led = 577f (??????????: 3317)
Test 4 (AND): led = 070f (??????????: 3010)
Test 5 (XOR): led = 18ad (??????????: 2fb2)
Test 6 (ADD): led = 834f (??????????: 9a54)
Test 7 (Logical Shift Left): led = 0004 (??????????: a540)
Test 8 (Arithmetic Shift Right): led = 0005 (??????????: d2a0)
Test 9 (Less Than): led = 46fa (??????????: 0001)
```

<u>Σημείωση:</u> Δεν ξέρω γιατί εμφανίζει τα σύμβολα ‘?’.

Άσκηση 3

Στην 3η άσκηση της εργασίας δημιουργούμε ένα αρχείο καταχωρητών (register file).

Το αρχείο καταχωρητών πρέπει να έχει τις κάτωθι θύρες.

Όνομα θύρας	Κατεύθυνση	Πλάτος [αρ. bit]	Σκοπός
clk	Είσοδος	1	Ρολόι
readReg1	Είσοδος	5	Διεύθυνση για τη θύρα ανάγνωσης 1
readReg2	Είσοδος	5	Διεύθυνση για τη θύρα ανάγνωσης 2
writeReg	Είσοδος	5	Διεύθυνση για θύρα εγγραφής
writeData	Είσοδος	DATAWIDTH	Δεδομένα προς εγγραφή
write	Είσοδος	1	Σήμα ελέγχου που υποδεικνύει εγγραφή
readData1	Έξοδος	DATAWIDTH	Δεδομένα ανάγνωσης από τη θύρα 1
readData2	Έξοδος	DATAWIDTH	Δεδομένα ανάγνωσης από τη θύρα 2

Πίνακας 3.1

Το αρχείο καταχωρητών μας πρέπει να αποτελείται από 32 καταχωρητές των 32 bit.

- Οπότε ας ορίσουμε το module ‘regfile’ με εισόδους και εξόδους αυτές του πίνακα 3.1.

```
1  // Code your design here
2  module regfile #(parameter DATAWIDTH = 32, // Πλάτος δεδομένων.
3                   parameter COUNT = 32) // Πλήθος καταχωρητών.
4
5    (output reg [DATAWIDTH-1:0] readData1, // Δεδομένα ανάγνωσης από τη θύρα 1.
6     output reg [DATAWIDTH-1:0] readData2, // Δεδομένα ανάγνωσης από τη θύρα 2.
7     input wire clk, // Ρολόι.
8     input wire [4:0] readReg1, // Διεύθυνση για τη θύρα ανάγνωσης 1.
9     input wire [4:0] readReg2, // Διεύθυνση για τη θύρα ανάγνωσης 2.
10    input wire [4:0] writeReg, // Διεύθυνση για θύρα εγγραφής.
11    input wire [DATAWIDTH-1:0] writeData, // Δεδομένα προς εγγραφή.
12    input wire write); // Σήμα ελέγχου που υποδεικνύει εγγραφή.
13
14   reg[DATAWIDTH-1:0] registers [0:COUNT-1]; // 32x32 αρχείο καταχωρητών,
15                                             // 32 καταχωρητές των 32 bit.
16
```

Δηλώνουμε ως παραμέτρους το μήκος (DATAWIDTH) του καταχωρητή και το πλήθος (COUNT) των καταχωρητών του αρχείου.

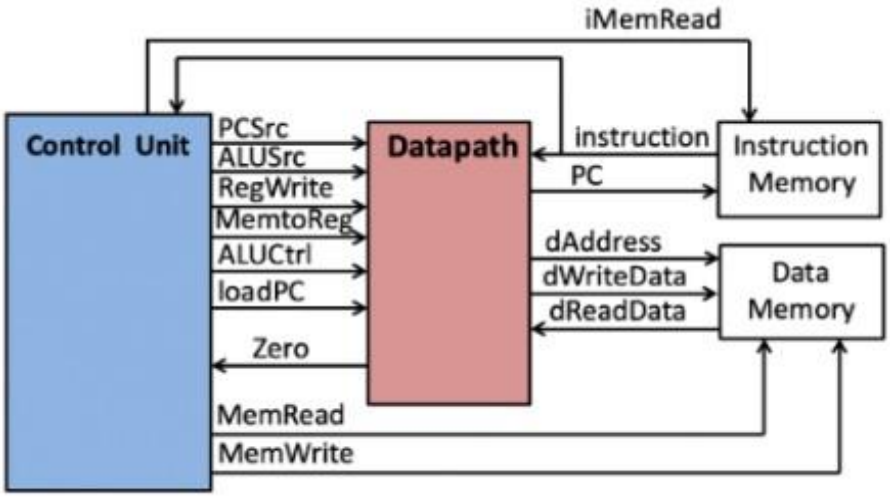
Στη γραμμή 14 του κώδικα ορίζω το 32x32 αρχείο καταχωρητών.

Αρχικοποιώ τους καταχωρητές με μηδενικά μέσω μιας εντολής initial και μιας for. Έπειτα γράφω τα δεδομένα στην κατάλληλη διεύθυνση του καταχωρητή, όταν το σήμα write είναι ενεργοποιημένο. Τέλος, διαβάζω τα δεδομένα από τις θύρες ανάγνωσης.

```
  // Αρχικοποιώ τους καταχωρητές με μηδενικά.
  initial
    begin
      for(integer i = 0; i <= 31; i=i+1)
        registers[i] = {DATAWIDTH{1'b0}}; // Επαναλαμβάνω 32 φορές το bit 0.
    end

  always @(posedge clk)
    begin
      if(write) // Εγγραφή δεδομένων όταν write=1.
        begin
          // Γράφω τα δεδομένα writeData στη διεύθυνση writeReg.
          registers[writeReg] <= writeData;
        end
    end

  // Διαβάζω τα δεδομένα από τη θύρα readReg1.
  assign readData1 = registers[readReg1];
  // Διαβάζω τα δεδομένα από τη θύρα readReg2.
  assign readData2 = registers[readReg2];

endmodule
```

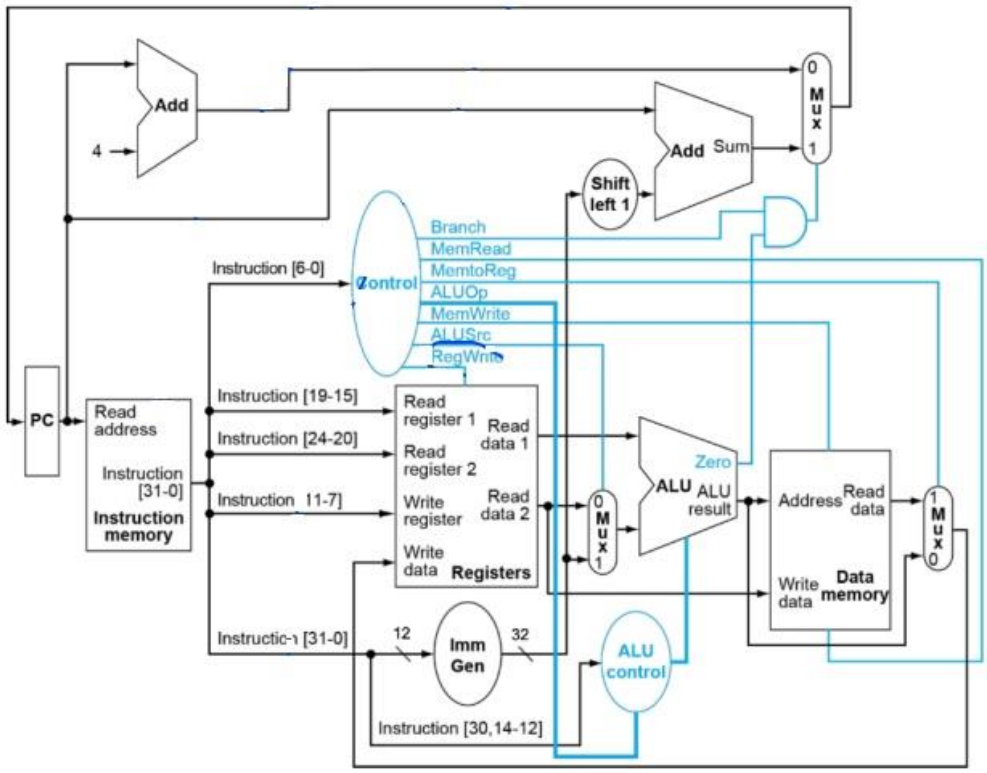
Άσκηση 4

Στην 4η άσκηση της εργασίας θα υλοποιήσουμε τη μονάδα διαδρομής δεδομένων (datapath) του επεξεργαστή RISC-V (βλ. σχήμα 4.1).



Σχήμα 4.1

Το διάγραμμα λειτουργίας του datapath φαίνεται στο σχήμα 4.2.



Σχήμα 4.2

Παράμετρος	**Πλάτος [αρ. bit]**	**Προεπιλεγμένη τιμή**
INITIAL_PC	32	0x00400000

Όνομα Θύρας	**Κατεύθυνση**	**Πλάτος [αρ. bit]**	**Σκοπός**
clk	Είσοδος	1	Ρολόι
rst	Είσοδος	1	Σύγχρονο Reset
instr	Είσοδος	32	Δεδομένα εντολών από τη μνήμη εντολών
PCSrc	Είσοδος	1	Πηγή του PC
ALUSrc	Είσοδος	1	Πηγή του 2ου τελεστή της ALU
RegWrite	Είσοδος	1	Εγγραφή δεδομένων στους καταχωρητές
MemToReg	Είσοδος	1	Πολυπλέκτης εισόδου στους καταχωρητές
ALUCtrl	Είσοδος	4	Δείχνει ποια λειτουργία πρέπει να εκτελέσει η ALU
loadPC	Είσοδος	1	Ενημέρωση του PC με μια νέα τιμή
PC	Έξοδος	32	Program Counter
Zero	Έξοδος	1	Ένδειξη μηδενισμού ALU
dAddress	Έξοδος	32	Διεύθυνση για δεδομένα μνήμης
dWriteData	Έξοδος	32	Δεδομένα προς εγγραφή στη μνήμη δεδομένων
dReadData	Είσοδος	32	Δεδομένα προς ανάγνωση από τη μνήμη δεδομένων
WriteBackData	Έξοδος	32	WriteBack δεδομένα που επιστρέφουν στους καταχωρητές

Πίνακας 4.1

- Ορίζουμε το module ‘datapath’ με τις θύρες και παραμέτρους αυτές του πίνακα 4.1.

```
// Code your design here
`timescale 1ns/1ps
module datapath #(parameter [31:0] INITIAL_PC = 32'h00400000)
(
    output reg [31:0] PC, // Program Counter.
    output wire Zero, // Ένδειξη μηδενισμού ALU.
    output wire [31:0] dAddress, // Διεύθυνση για δεδομένα μνήμης.
    output wire [31:0] dWriteData, // Δεδομένα προς εγγραφή στη μνήμη δεδομένων.
    output wire [31:0] WriteBackData, // WriteBack δεδομένα στους καταχωρητές.
    input wire clk, // Ρολόι.
    input wire rst, // Σύγχρονο Reset.
    input wire [31:0] instr, // Δεδομένα εντολών από τη μνήμη εντολών
    input wire PCSrc, // Πηγή του PC.
    input wire ALUSrc, // Πηγή του 2ου τελεστή της ALU.
    input wire RegWrite, // Εγγραφή δεδομένων στους καταχωρητές.
    input wire MemToReg, // Πολυπλέκτης εισόδου στους καταχωρητές.
    input wire [3:0] ALUCtrl, // Εντολή για την ALU.
    input wire loadPC, // Ενημέρωση του PC με νέα τιμή.
    input wire [31:0] dReadData); // Δεδομένα από τη μνήμη δεδομένων.

    // Αποκωδικοποίηση των bits της εντολής.
    wire [6:0] opcode = instr[6:0];
    wire [4:0] rd = instr[11:7];
    wire [2:0] funct3 = instr[14:12];
    wire [4:0] rs1 = instr[19:15];
    wire [4:0] rs2 = instr[24:20];
    wire [6:0] funct7 = instr[31:25];
```

Αποκωδικοποιούμε την εντολή στα επίπεδά της σύμφωνα με την σελίδα 148 του pdf.

- Υλοποιούμε τη λειτουργία Immediate Generation.

```
    // Immediate Generation με βάση την 148 σελίδα του pdf.
    reg [31:0] imm;
    always @(*) begin
        case (opcode)
            7'b0010011, 7'b0000011, 7'b1100111: // I-type (ADDI, LW, JALR)
                imm = {{20{instr[31]}}, instr[31:20]};
            7'b0100011: // S-type (SW, SH, SB)
                imm = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            7'b1100011: // B-type (BEQ, BNE, etc.)
                imm = {{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8],
1'b0};
            7'b0110111, 7'b0010111: // U-type (LUI, AUIPC)
                imm = {instr[31:12], 12'b0};
            7'b1101111: // J-type (JAL)
                imm = {{11{instr[31]}}, instr[31], instr[19:12], instr[20],
instr[30:21], 1'b0};
            default:
                imm = 32'b0;
        endcase
    end

    // Σύνδεση με το submodule regfile.
    wire [31:0] readData1, readData2;
    regfile #(32, 32) regfile_inst (
      .readData1(readData1),
      .readData2(readData2),
      .clk(clk),
      .readReg1(rs1),
      .readReg2(rs2),
      .writeReg(rd),
      .writeData(WriteBackData),
      .write(RegWrite)
    );
```

Το στοιχείο Immediate Generation τροφοδοτεί την 2η πύλη της ALU, κατόπιν βέβαια από την επιλογή μέσω του πολυπλέκτη.

Ανάλογα το opcode, τα άμεσα δεδομένα (imm) είναι διαφορετικά για κάθε τύπο εντολής.

Επίσης συνδέουμε με το datapath το αρχείο καταχωρητών μας από την 3η άσκηση. Προσέχουμε ιδιαίτερα στη δήλωση των θυρών.

- Ο πολυπλέκτης λοιπόν θα αποφασίσει ποια δεδομένα θα εισέλθουν στη 2η πύλη της ALU μέσω του σήματος 'ALUSrc'. Ο πολυπλέκτης επιλέγει μεταξύ άμεσων δεδομένων imm και δεδομένων από την έξοδο Read data 2 του register file.

```
60
61   // Είσοδος op2 της alu.
62   wire [31:0] alu_op2 = ALUSrc ? imm : readData2;
63
64   // Σύνδεση με το submodule alu.
65   wire [31:0] aluResult;
66   alu alu_inst (
67     .result(aluResult),
68     .zero(Zero),
69     .op1(readData1),
70     .op2(alu_op2),
71     .alu_op(ALUCtrl)
72   );
73
74   // Data Memory.
75   assign dAddress = aluResult;
76   assign dWriteData = readData2;
77
78   // Write Back.
79   assign WriteBackData = MemToReg ? dReadData : aluResult;
80
81   // Program Counter.
82   always @(posedge clk)
83     begin
84       if(rst) // Σύγχρονο Reset.
85         PC <= INITIAL_PC;
86       else if(loadPC) // Ενημέρωση του PC με νέα τιμή (στην επόμενη ακμή του ρολογιού).
87         PC <= PCSrc ? (PC + (imm << 1)) : PC + 4; // Λογική branch target ανάλογα την
   τιμή του PCSrc.
88     end
89
90 endmodule
91
```

Συνδέω επίσης με προσοχή την alu από την 1η άσκηση.

Οι μεταβλητές dAddress και dWriteData είναι οι είσοδοι του στοιχείου Data Memory.

Μέσω ενός πολυπλέκτη ελεγχόμενου από το σήμα 'MemtoReg' αποφασίζω ποια δεδομένα θα στείλω πίσω στο register file, στην είσοδο Write data.

Το στοιχείο Program Counter, σε κάθε θετική ακμή του ρολογιού, εάν πατηθεί το reset παίρνει την αρχική τιμή που ορίσαμε ως παράμετρο, αλλιώς ενημερώνεται εάν «πατηθεί» το σήμα loadPC. Η νέα τιμή που θα πάρει εξαρτάται από το σήμα PCSrcs.

Άσκηση 5

Στην άσκηση 5 δημιουργούμε έναν ελεγκτή πολλαπλών κύκλων που εκτελεί κάθε εντολή σε πέντε κύκλους ρολογιού.

Παράμετρος	**Πλάτος [αρ. bit]**	**Προεπιλεγμένη τιμή**
INITIAL_PC	32	0x00400000

Όνομα Θύρας	**Κατεύθυνση**	**Πλάτος [αρ. bit]**	**Σκοπός**
clk	Είσοδος	1	Ρολόι
rst	Είσοδος	1	Σύγχρονο Reset
instr	Είσοδος	32	Δεδομένα εντολών από τη μνήμη εντολών
dReadData	Είσοδος	32	Ανάγνωση δεδομένων από τη μνήμη δεδομένων
PC	Έξοδος	32	Program Counter
dAddress	Έξοδος	32	Διεύθυνση για δεδομένα μνήμης
dWriteData	Έξοδος	32	Δεδομένα προς εγγραφή στη μνήμη δεδομένων
MemRead	Έξοδος	1	Σήμα ελέγχου που υποδεικνύει ανάγνωση μνήμης
MemWrite	Έξοδος	1	Σήμα ελέγχου που υποδεικνύει εγγραφή στη μνήμη
WriteBackData	Έξοδος	32	Δεδομένα που εγγράφονται σε καταχωρητές (για αποσφαλμάτωση)

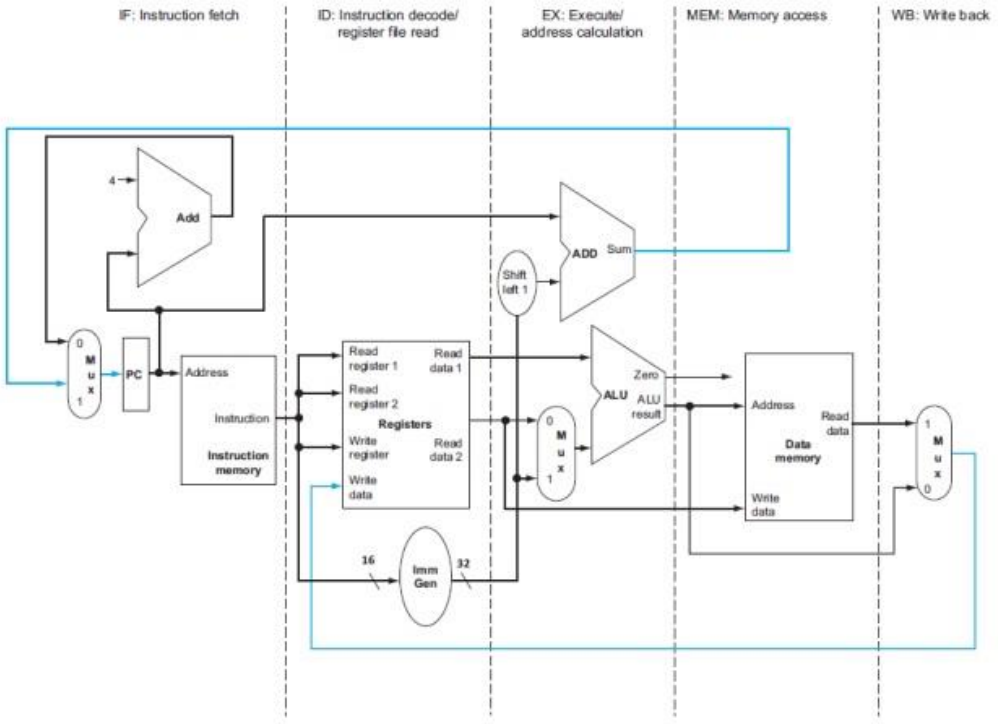
Πίνακας 5.1

- Σύμφωνα με τον πίνακα 5.1 ορίζουμε το module ‘top_proc’:

```
// Code your design here
module top_proc #(parameter [31:0] INITIAL_PC = 32'h00400000)
  (
    output reg [31:0] PC, // Program Counter.
    output wire [31:0] dAddress, // Διεύθυνση για δεδομένα μνήμης.
    output wire [31:0] dWriteData, // Δεδομένα προς εγγραφή στη μνήμη δεδομένων.
    output wire MemRead; // Σήμα ελέγχου που υποδεικνύει ανάγνωση μνήμης.
    output wire MemWrite; // Σήμα ελέγχου που υποδεικνύει εγγραφή στη μνήμη.
    output wire [31:0] // Δεδομένα που εγγράφονται σε καταχωρητές (για
αποσφαλμάτωση).
    input wire clk, // Ρολόι.
    input wire rst, // Σύγχρονο Reset.
    input wire [31:0] instr, // Δεδομένα εντολών από τη μνήμη εντολών.
    input wire [31:0] dReadData); // Ανάγνωση δεδομένων από τη μνήμη δεδομένων.

  // Datapath.
  datapath #(
    .INITIAL_PC(INITIAL_PC) // Κατάλληλη αρχικοποίηση εντός module.
  ) datapath_inst(
    .PC(PC),
    .instr(instr),
    .dAddress(dAddress),
    .dReadData(dReadData),
    .dWriteData(dWriteData)
  )
```

Χρειάζεται να δημιουργήσουμε τη μηχανή 5 καταστάσεων καθώς και τα σήματα ελέγχου για τη μονάδα διαδρομής δεδομένων.

Το datapath διαχωρίζεται στις 5 καταστάσεις ως εξής:



Σχήμα 5.1