Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής Ι

Ακαδημαϊκό Έτος 2024-2025
Υποχρεωτική Εργασία

ΠΑΠΑΔΟΠΟΥΛΟΣ ΠΑΝΑΓΙΩΤΗΣ 10697

• Άσκηση 1 - ALU

Εισαγωγή

Στην πρώτη άσκηση της εργασίας, αναπτύχθηκε μια Αριθμητική/Λογική Μονάδα (ALU) για την υλοποίηση στον επεξεργαστή $RISC-V$. Η ALU είναι σχεδιασμένη για την εκτέλεση βασικών αριθμητικών και λογικών λειτουργιών.

Σχεδίαση ALU

Η ALU σχεδιάστηκε για να υλοποιεί τις εξής πράξεις:
- Προσημασμένη πρόσθεση
- Προσημασμένη αφαίρεση
- Λογικό AND
- Λογικό OR
- Λογικό XOR
- Σύγκριση “Μικρότερο από”
- Τρεις διαφορετικές πράξεις ολίσθησης

Το $Verilog\ module\ alu$ ορίστηκε με τις ακόλουθες θύρες:
- **op1** και **op2**: Οι τελεστές για τις πράξεις
- **alu_op**: Οδηγίες ελέγχου για τον τύπο πράξης
- **zero**: Έξοδος που δείχνει εάν το αποτέλεσμα είναι μηδέν
- **result**: Το αποτέλεσμα της ALU

```
module alu(
    input [31:0] op1,        // First operand
    input [31:0] op2,        // Second operand
    input [3:0] alu_op,      // ALU operation selector
    output reg zero,         // Zero flag
    output reg [31:0] result // Result of ALU operation
);

    // Constants for ALU operations
    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_LESS    = 4'b0100;
```

```
    parameter [3:0] ALUOP_RSHIFT  = 4'b1000;
    parameter [3:0] ALUOP_LSHIFT  = 4'b1001;
    parameter [3:0] ALUOP_NRSHIFT = 4'b1010;
    parameter [3:0] ALUOP_XOR     = 4'b0101;

    // ALU combinational logic
    always @(*) begin
        case (alu_op)
            ALUOP_AND: result = op1 & op2;                                  // AND operation
            ALUOP_OR: result = op1 | op2;                                   // OR operation
            ALUOP_ADD: result = op1 + op2;                                  // Addition
            ALUOP_SUB: result = op1 - op2;                                  // Subtraction
            ALUOP_LESS: result = ($signed(op1) < $signed(op2)) ? 1 : 0;     // Set less than (signed)
            ALUOP_RSHIFT: result = op1 >> op2[4:0];                         // Logical right shift
            ALUOP_LSHIFT: result = op1 << op2[4:0];                         // Logical left shift
            ALUOP_NRSHIFT: result = $unsigned($signed(op1) >>> op2[4:0]);   // Arithmetic right shift
            ALUOP_XOR: result = op1 ^ op2;                                  // XOR operation
            default: result = 32'b0;                                        // Default case
        endcase

        // Set zero flag if result is zero
        if (result == 32'b0)
            zero = 1'b1;
        else
            zero = 1'b0;

    end

endmodule
```

Υλοποίηση

Ο κώδικας $Verilog$ αναπτύχθηκε για να ανταποκρίνεται στις προδιαγραφές της άσκησης. Η λειτουργία της ALU βασίζεται στις τιμές του alu_op για να επιλέξει την κατάλληλη πράξη, συγκρίνοντας την είσοδο με τις προκαθορισμένες παραμέτρους όπως ζητήθηκε. Ένας σημαντικός σχεδιαστικός θεματικός άξονας ήταν η διασφάλιση ότι οι πράξεις που εξαρτώνται από το πρόσημο λειτουργούν σωστά σε προσημασμένες εισόδους.

Συμπεράσματα

Η ALU αποδείχθηκε αποτελεσματική στην εκτέλεση των καθορισμένων λειτουργιών και έτοιμη για ενσωμάτωση στον επεξεργαστή $RISC-V$ που θα αναπτυχθεί σε επόμενες ασκήσεις. Η διαδικασία αυτή παρέχει μια σταθερή βάση για περαιτέρω επέκταση και τεστάρισμα.

• Άσκηση 2 - Επέκταση της ALU

Εισαγωγή

Για την Άσκηση 2, το παραδοτέο αφορά την υλοποίηση μιας αριθμομηχανής που βασίζεται στην ALU από την Άσκηση 1 και περιλαμβάνει τη χρήση ενός συσσωρευτή 16 bit, μαζί με τη σύνδεση με κουμπιά και διακόπτες για τον έλεγχο των πράξεων. Η αριθμομηχανή που θα υλοποιηθεί έχει σκοπό να εκτελεί αριθμητικές και λογικές πράξεις με βάση τις λειτουργίες της ALU. Ο συσσωρευτής ($accumulator$) διατηρεί την τρέχουσα τιμή και ενημερώνεται από το αποτέλεσμα της ALU. Τα δεδομένα εισόδου προέρχονται από τους διακόπτες (sw) και ο έλεγχος των λειτουργιών γίνεται με κουμπιά ($btnc$, $btnl$, $btnr$, $btnd$, $btnu$).

Σχεδίαση Αριθμομηχανής

Η σχεδίαση βασίζεται στη χρήση της ALU που υλοποιήθηκε στην πρώτη άσκηση, ενσωματώνοντας επιπλέον έναν συσσωρευτή για την αποθήκευση και προβολή των αποτελεσμάτων. Οι λειτουργίες της αριθμομηχανής περιλαμβάνουν:

- Προσημασμένη πρόσθεση και αφαίρεση μεταξύ του συσσωρευτή και του εισερχόμενου τελεστή.
- Λογικές πράξεις όπως AND, OR και XOR.
- Πράξεις ολίσθησης (δεξιά, αριστερά, αριθμητική δεξιά).
- Σύγκριση “Μικρότερο από” για προσημασμένους αριθμούς.
- Αρχικοποίηση του συσσωρευτή μέσω κουμπιού.

Το $Verilog\ module\ calc$ ορίστηκε με τις ακόλουθες θύρες:

- **sw**: Οι διακόπτες που παρέχουν την είσοδο στη δεύτερη ALU παράμετρο.
- **btnc**, **btnl**, **btnr**, **btnd**, **btnu**: Τα κουμπιά που καθορίζουν τις πράξεις και τις λειτουργίες ελέγχου (reset του συσσωρευτή, εκτέλεση πράξης κ.λπ.).
- **led**: Τα LED που απεικονίζουν την τρέχουσα τιμή του συσσωρευτή.
- **clk**: Σήμα ρολογιού για τον συγχρονισμό της λειτουργίας.

```
module calc(
  input clk,
  input reg btnc,
  input reg btnl,
  input reg btnu,
  input reg btnr,
  input reg btnd,
  input reg [15:0] sw,
  output[15:0] led
);
  genvar i;
  reg [3:0] alu_op;
  reg [31:0] res, p1, p2;
  wire btnc_w = btnc;
  calc_enc calc_enc(btnc_w, btnr, btnl, alu_op);
  alu alu(p1, p2, alu_op, z, res);
  reg[15:0] accumulator;

  for(i=0; i<16; i = i + 1)
    begin
     assign led[i] = accumulator[i];
    end

  assign p1 = $signed(accumulator);
  assign p2 = $signed(sw);

  always @(posedge clk)
    begin
      if(btnu) accumulator = 15'b0;
      else if(btnd) accumulator = res;
    end

endmodule
```

Υλοποίηση

Η λειτουργία της αριθμομηχανής καθορίζεται από τα σήματα ελέγχου που προέρχονται από τα κουμπιά και τους διακόπτες:

- Τα κουμπιά καθορίζουν την πράξη που θα εκτελεστεί ή επανεκκινούν τη συσκευή.
- Οι διακόπτες εισάγουν τη δεύτερη παράμετρο για την ALU.
- Το αποτέλεσμα της ALU αποθηκεύεται στον συσσωρευτή όταν πατηθεί το κουμπί εκτέλεσης.

Ένα βασικό στοιχείο ήταν η σωστή ενσωμάτωση του συσσωρευτή και η χρήση του αποτελέσματος της ALU για την ενημέρωσή του. Ο σχεδιασμός εξασφαλίζει την εκτέλεση όλων των βασικών αριθμητικών και λογικών πράξεων.

Κύκλωμα Σήματος ALU

Το κύκλωμα για την επιλογή της λειτουργίας της αριθμητικής/λογικής μονάδας υλοποιήθηκε σε $Structural\ Verilog$, όπως ζητήθηκε. Η είσοδος του κυκλώματος είναι τα κουμπιά $btnl$, $btnc$ και $btnr$ με έξοδο τα 4 ψηφία της μεταβλητής alu_op. Η υλοποίηση ακολουθεί.

```
module calc_enc(
  input wire btnc,
  input wire btnr,
  input wire btnl,
  output[3:0] alu_op
);

  not(btncNOT, btnc);
  and(w12, btncNOT, btnr);
  and(w13, btnr, btnl);
  or(alu_op[0], w12, w13);

  not(btnlNOT, btnl);
  and(w22, btnc, btnlNOT);
  not(btnrNOT, btnr);
  and(w23, btnc, btnrNOT);
  or(alu_op[1], w22, w23);

  and(w31, btnl, btncNOT);
  and(w32, btnc, btnr);
  and(w33, w31, btnrNOT);
  or(alu_op[2], w32, w33);

  and(w41, btnl, btncNOT);
  and(w42, btnl, btnc);
  and(w43, w41, btnr);
  and(w44, w42, btnrNOT);
  or(alu_op[3], w43, w44);

endmodule
```

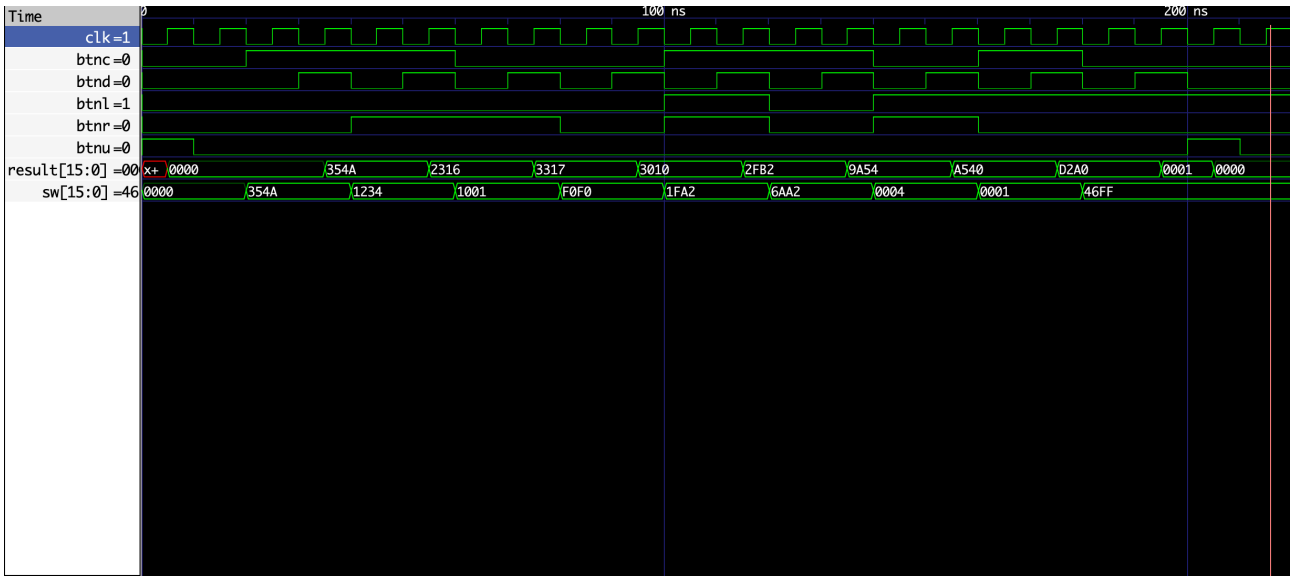
Έλεγχος

Χρησιμοποιώντας τα παραπάνω $modules$, δημιουργήθηκε ένα $testbench$ για τον έλεγχο της ορθής λειτουργίας της αριθμομηχανής και της ALU. Δόθηκαν συγκεκριμένες τιμές με σκοπό να αναδειχθούν τα αποτελέσματα με τις τιμές που δόθηκαν, και προέκυψε το παρακάτω.

```
`include "calc.v"
`timescale 1ns / 1ps

module calc_tb;

    // Testbench signals
    reg clk;
    reg btnc, btnl, btnr, btnu, btnd;
    reg [15:0] sw; // Switch input for data
    wire [15:0] result; // Output result from calculator

    // Instantiate the calculator module
    calc uut (
        .clk(clk),
        .btnc(btnc),
        .btnl(btnl),
        .btnr(btnr),
        .btnu(btnu),
        .btnd(btnd),
        .sw(sw),
        .led(result)
    );

    // Clock generation
```

```
    initial begin
        $dumpfile("calc.vcd");
        $dumpvars;
        clk = 0;
        forever #5 clk = ~clk; // 10ns clock period
    end

    // Test sequence
    initial begin
        // Initialize inputs
        btnc = 0; btnl = 0; btnr = 0; btnu = 0; btnd = 0; sw = 16'h0;

        // Reset the calculator
        btnu = 1; #10; btnu = 0; #10;

        // Test addition (ADD)
        sw = 16'h354a; btnl = 0; btnc = 1; btnr = 0; #10; btnd = 1; #10; btnd = 0;

        // Test subtraction (SUB)
        sw = 16'h1234; btnl = 0; btnc = 1; btnr = 1; #10; btnd = 1; #10; btnd = 0;

        // Test OR operation
        sw = 16'h1001; btnl = 0; btnc = 0; btnr = 1; #10; btnd = 1; #10; btnd = 0;

        // Test AND operation
        sw = 16'hf0f0; btnl = 0; btnc = 0; btnr = 0; #10; btnd = 1; #10; btnd = 0;

        // Test XOR operation
        sw = 16'h1fa2; btnl = 1; btnc = 1; btnr = 1; #10; btnd = 1; #10; btnd = 0;

        // Test addition (ADD)
        sw = 16'h6aa2; btnl = 0; btnc = 1; btnr = 0; #10; btnd = 1; #10; btnd = 0;

        // Test Logical Shift Left (LSL)
        sw = 16'h0004; btnl = 1; btnc = 0; btnr = 1; #10; btnd = 1; #10; btnd = 0;

        // Test Shift Right Arithmetic (SRA)
        sw = 16'h0001; btnl = 1; btnc = 1; btnr = 0; #10; btnd = 1; #10; btnd = 0;

        // Test Less Than (LT)
        sw = 16'h46ff; btnl = 1; btnc = 0; btnr = 0; #10; btnd = 1; #10; btnd = 0;

        // Test reset with btnu
        btnu = 1; #10; btnu = 0; #10;

        // End simulation
        $finish;
    end

endmodule
```

Με την εκτέλεσή του προκύπτουν οι κυματομορφές προσομοίωσης:



Κυματομορφές Προσομοίωσης Άσκησης 2

Είναι εμφανές ότι τα αποτελέσματα της προσομοίωσης αντιστοιχούν με τα προσδοκούμενα αποτελέσματα.

Συμπεράσματα

Η αριθμομηχανή λειτουργεί όπως απαιτείται από την εκφώνηση. Όλες οι πράξεις της ALU έχουν ελεγχθεί μέσω προσομοιώσεων και επιβεβαιώθηκε η σωστή ενημέρωση του $accumulator$. Οι απαιτούμενες λειτουργίες προσθήκης, αφαίρεσης, λογικών πράξεων και ολίσθησης λειτουργούν ορθά, με τα αποτελέσματα να είναι ακριβή και τα LED να εμφανίζουν την τρέχουσα τιμή.

• Άσκηση 3 - Καταχωρητές

Εισαγωγή

Η τρίτη άσκηση επικεντρώνεται στον σχεδιασμό και την υλοποίηση ενός αρχείου καταχωρητών ($register\ file$), που αποτελεί ένα κρίσιμο στοιχείο κάθε επεξεργαστή $RISC-V$. Το αρχείο καταχωρητών είναι υπεύθυνο για την αποθήκευση των τιμών των καταχωρητών, οι οποίες χρησιμοποιούνται στις περισσότερες εντολές που εκτελεί ο επεξεργαστής. Η λειτουργία του αρχείου καταχωρητών διασφαλίζει τη σωστή ανάγνωση και εγγραφή δεδομένων, προσφέροντας έτσι τον απαιτούμενο μηχανισμό υποστήριξης για τη λειτουργία του επεξεργαστή.

Σχεδιασμός Καταχωρητών

Η σχεδίαση του αρχείου καταχωρητών βασίζεται στη δημιουργία μιας μονάδας $Verilog$ που περιλαμβάνει 32 καταχωρητές των 32 bits. Οι διευθύνσεις εισόδου επιτρέπουν την ανάγνωση δύο καταχωρητών και την εγγραφή σε έναν καταχωρητή. Η λειτουργία ανάγνωσης είναι ασύγχρονη, ενώ η εγγραφή πραγματοποιείται συγχρονισμένα με το σήμα ρολογιού. Οι καταχωρητές αρχικοποιούνται σε μηδενικές τιμές κατά την εκκίνηση. Εάν η διεύθυνση εγγραφής συμπίπτει με κάποια από τις διευθύνσεις ανάγνωσης, δίνεται προτεραιότητα στην εγγραφή για τη διασφάλιση της σωστής λειτουργίας. Ο μηδενικός καταχωρητής (x_0) θα πρέπει να παραμένει πάντα μηδενικός. Ωστόσο, δεν προβλέπεται ενέργεια ενάντια σε απόπειρα εγγραφής σε αυτόν, εφόσον δεν αναφέρεται στην εκφώνηση της άσκησης.

Η σχεδίαση του αρχείου καταχωρητών περιλαμβάνει τη δημιουργία μιας μονάδας $Verilog$ με τις εξής θύρες:

- **clk**: Είσοδος 1 bit, χρησιμοποιείται ως σήμα ρολογιού για συγχρονισμένες λειτουργίες εγγραφής.
- **readReg1** και **readReg2**: Είσοδοι 5 bits, χρησιμοποιούνται για την παροχή διευθύνσεων των δύο καταχωρητών που θα αναγνωστούν.
- **writeReg**: Είσοδος 5 bits, χρησιμοποιείται για την παροχή της διεύθυνσης του καταχωρητή στον οποίο θα γίνει εγγραφή.
- **writeData**: Είσοδος 32 bits ($DATAWIDTH$), τα δεδομένα που θα εγγραφούν στον καταχωρητή.
- **write**: Είσοδος 1 bit, σήμα ελέγχου που ενεργοποιεί την εγγραφή στον καταχωρητή.

- **readData1** και readData2: Έξοδοι 32 bits ($DATAWIDTH$), παρέχουν τα δεδομένα των καταχωρητών που αναγνώστηκαν από τις αντίστοιχες διευθύνσεις.

```
module regfile #(parameter DATAWIDTH = 32)
  (
  input clk,
  input [4:0] readReg1,
  input [4:0] readReg2,
  input [4:0] writeReg,
  input [DATAWIDTH-1:0] writeData,
  input write,
  output wire [DATAWIDTH-1:0] readData1,
  output wire [DATAWIDTH-1:0] readData2
);

  reg [DATAWIDTH-1:0] registers [0:31];
  integer i;

  initial begin
    for(i=0; i<32; i++)
      begin
        registers[i] <= 0;
      end


  end

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

  assign readData1 = (write && (writeReg == readReg1)) ? writeData : registers[readReg1];
  assign readData2 = (write && (writeReg == readReg2)) ? writeData : registers[readReg2];

endmodule
```

Υλοποίηση

Η υλοποίηση του αρχείου καταχωρητών ακολουθεί πιστά τις προδιαγραφές της εκφώνησης. Περιλαμβάνει 32 καταχωρητές των 32 bits, οι οποίοι αρχικοποιούνται σε μηδενικά μέσω ενός $initial\ block$. Οι θύρες του $module$ επιτρέπουν την ασύγχρονη ανάγνωση δύο καταχωρητών και τη συγχρονισμένη εγγραφή ενός καταχωρητή με προτεραιότητα στην εγγραφή σε περίπτωση σύγκρουσης διευθύνσεων. Επιπλέον, η χρήση συνθήκης εξασφαλίζει τη σωστή διαχείριση της εγγραφής όταν η διεύθυνση εγγραφής ταυτίζεται με κάποια από τις διευθύνσεις ανάγνωσης, όπως ορίζεται στην άσκηση.

• Άσκηση 4 - Μονάδα Διαδρομής Δεδομένων

Εισαγωγή

Στην τέταρτη άσκηση, το επίκεντρο είναι η υλοποίηση και πλήρης λειτουργία της διαδρομής δεδομένων ($datapath$) του επεξεργαστή $RISC-V$. Η διαδρομή δεδομένων είναι ουσιαστικά το σύνολο των συνιστωσών που εκτελούν τις εντολές μεταξύ των καταχωρητών, της μνήμης και της ALU. Αυτή η άσκηση απαιτεί την κατασκευή και ενσωμάτωση των κύριων λειτουργικών μερών όπως η μονάδα ελέγχου, η μονάδα λογικής, οι καταχωρητές, καθώς και οι μηχανισμοί διαχείρισης και επικοινωνίας μεταξύ των διάφορων μερών του επεξεργαστή.

Σχεδιασμός Μονάδας Διαδρομής Δεδομένων

Στην παρούσα άσκηση 4 καλούμαστε να υλοποιήσουμε βασικές μονάδες και λειτουργίες της διαδρομής δεδομένων για τον επεξεργαστή $RISC-V$. Ο $Program\ Counter$ (PC) είναι υπεύθυνος για την παρακολούθηση της διεύθυνσης της επόμενης εντολής που

πρόκειται να εκτελεστεί. Ο PC αυξάνεται κατά 4 για κάθε νέα εντολή, εκτός αν υπάρχει εντολή διακλάδωσης, οπότε ο PC αλλάζει σύμφωνα με το $offset$ της εντολής. Στη συνέχεια, η υλοποίηση του καταχωρητή ($Register\ File$) απαιτεί την αποθήκευση και ανάγνωση των δεδομένων από τους 32 καταχωρητές του επεξεργαστή. Η διαχείριση των διευθύνσεων ανάγνωσης και εγγραφής είναι ζωτικής σημασίας, ειδικά όταν η ίδια διεύθυνση χρησιμοποιείται για ανάγνωση και εγγραφή, ώστε να διασφαλιστεί η σωστή προτεραιότητα στην εγγραφή των δεδομένων.

Η ενότητα για τη δημιουργία των άμεσων τιμών ($Immediate\ Generation$) είναι επίσης καθοριστική, καθώς παράγει τις άμεσες τιμές που απαιτούνται από τις εντολές τύπου I, S και B, όπως οι $ADDI$, BEQ και άλλες. Αυτές οι τιμές πρέπει να παράγονται από τα κατάλληλα πεδία της εντολής, και η σωστή αναγνώρισή τους είναι κρίσιμη για την ορθή λειτουργία του επεξεργαστή. Η ALU εκτελεί τις αριθμητικές και λογικές πράξεις, όπως η πρόσθεση, η αφαίρεση, οι λογικές πράξεις AND, OR και XOR, καθώς και τις πράξεις μετατόπισης και σύγκρισης. Ειδική προσοχή πρέπει να δοθεί στην υλοποίηση των πράξεων προσημασμένων και μη προσημασμένων αριθμών, ώστε να διασφαλιστεί η ακρίβεια των υπολογισμών και των συγκρίσεων.

Για τις εντολές διακλάδωσης, όπως η BEQ, είναι απαραίτητος ο υπολογισμός της διεύθυνσης στην οποία θα πρέπει να μεταβεί ο PC αν πληρούνται οι συνθήκες διακλάδωσης. Ο σωστός υπολογισμός του branch offset είναι σημαντικός για την ορθή εκτέλεση των εντολών διακλάδωσης, και πρέπει να γίνει με βάση τα δεδομένα της εντολής.

Τέλος, η λογική $Write\ Back$ είναι υπεύθυνη για την εγγραφή των αποτελεσμάτων στους καταχωρητές μετά την εκτέλεση των εντολών. Στις εντολές τύπου $Load$, το αποτέλεσμα από τη μνήμη εγγράφεται στους καταχωρητές, ενώ για τις άλλες εντολές, το αποτέλεσμα της ALU γράφεται στους καταχωρητές.

Η σχεδίαση του αρχείου διαδρομής δεδομένων περιλαμβάνει τη δημιουργία μιας μονάδας $Verilog$ με τις εξής θύρες:

- **clk**: είναι απαραίτητη για τη συγχρονικότητα του επεξεργαστή και τις λειτουργίες του. Όλες οι αλλαγές στις θύρες και οι ενέργειες του επεξεργαστή γίνονται με την άνοδο του ρολογιού.
- **rst**: χρησιμοποιείται για να εκκινήσει ή να επαναφέρει την κατάσταση του συστήματος. Όταν το rst είναι ενεργό, το σύστημα επανέρχεται στην αρχική του κατάσταση, επιτρέποντας την επανεκκίνηση του προγράμματος.
- **instr**: είναι υπεύθυνη για τη λήψη της εντολής που διαβάζεται από τη μνήμη εντολών. Η εντολή αυτή καθορίζει την πράξη που πρέπει να εκτελέσει ο επεξεργαστής.
- **PCSrc**: χρησιμοποιείται για να αποφασίσει αν ο $Program\ Counter$ (PC) θα αυξηθεί κατά 4 ή αν θα ενημερωθεί για να πραγματοποιηθεί $branching$ (διακλάδωση). Έτσι, καθορίζει την κατεύθυνση του επόμενου βήματος του προγράμματος.
- **ALUSrc**: καθορίζει την πηγή του δεύτερου τελεστή της ALU. Αν είναι ενεργοποιημένη, η ALU θα χρησιμοποιήσει μια άμεση τιμή ως δεύτερο όρισμα, αλλιώς θα χρησιμοποιήσει έναν καταχωρητή.
- **RegWrite**: επιτρέπει την εγγραφή αποτελέσματος σε έναν καταχωρητή. Όταν είναι ενεργή, ο επεξεργαστής γράφει το αποτέλεσμα της ALU ή της μνήμης στον καταχωρητή που έχει καθοριστεί.

- **MemToReg**: καθορίζει αν το δεδομένο που γράφεται στον καταχωρητή θα προέρχεται από τη μνήμη ή από την ALU. Αν είναι ενεργό, ο καταχωρητής θα λάβει δεδομένα από τη μνήμη, ενώ αν είναι ανενεργό, τα δεδομένα θα προέρχονται από το αποτέλεσμα της ALU.
- **ALUCtrl**: είναι το σήμα ελέγχου που καθορίζει τη λειτουργία της ALU, επιλέγοντας την πράξη που θα εκτελέσει (π.χ. πρόσθεση, αφαίρεση, κ.λπ.) με βάση τις απαιτήσεις της εντολής.
- **loadPC**: χρησιμοποιείται για να ενεργοποιήσει την ενημέρωση του $Program\ Counter$ (PC) κατά την εκτέλεση του προγράμματος. Αν είναι ενεργή, ο PC θα αυξηθεί σύμφωνα με το τι έχει οριστεί στην εντολή.
- **dReadData**: παρέχει τα δεδομένα που έχουν διαβαστεί από τη μνήμη δεδομένων και χρησιμοποιούνται για την εκτέλεση της εντολής ή την εγγραφή στους καταχωρητές.
- **PC**: είναι η έξοδος του $Program\ Counter$ και δείχνει την τρέχουσα διεύθυνση της εντολής που εκτελείται, η οποία ενημερώνεται με βάση τις εντολές ή τις $branching$ αυξήσεις.
- **dAddress**: καθορίζει τη διεύθυνση μνήμης για ανάγνωση ή εγγραφή δεδομένων
- **dWriteData**: περιέχει τα δεδομένα που πρέπει να γράψουν στη μνήμη
- **WriteBackData**: χρησιμοποιείται για να γράψει τα δεδομένα πίσω στους καταχωρητές είτε από τη μνήμη είτε από την ALU.

```
module datapath #(
    parameter INITIAL_PC = 32'h00400000  // Αρχική διεύθυνση του PC
)(
    input wire clk,                      // Ρολόι
    input wire rst,                      // Σύγχρονο reset
    input wire [31:0] instr,             // Εντολές από τη μνήμη εντολών
    input wire PCSrc,                    // Επιλογή της πηγής για το PC
    input wire ALUSrc,                   // Επιλογή της πηγής για τον δεύτερο τελεστή της ALU
    input wire RegWrite,                 // Ενεργοποίηση εγγραφής στους καταχωρητές
    input wire MemToReg,                 // Επιλογή του δεδομένου εγγραφής στους καταχωρητές
    input wire [3:0] ALUCtrl,            // Σήμα ελέγχου της ALU
    input wire loadPC,                   // Ενεργοποίηση ενημέρωσης του PC
    input wire [31:0] dReadData,         // Δεδομένα από τη μνήμη δεδομένων
    output wire [31:0] PC,               // Program Counter
    output wire Zero,                    // Έξοδος μηδενισμού ALU
    output wire [31:0] dAddress,         // Διεύθυνση για τη μνήμη δεδομένων
    output wire [31:0] dWriteData,       // Δεδομένα προς εγγραφή στη μνήμη δεδομένων
    output wire [31:0] WriteBackData     // Δεδομένα για εγγραφή στους καταχωρητές
);

     // Operation Types
    localparam RTYPE = 7'b0110011,
               ITYPE = 7'b0010011,
               STYPE = 7'b0100011,
               BTYPE = 7'b1100011,
               LOAD = 7'b0000011;

    // 1. Καταχωρητής Program Counter (PC)
    reg [31:0] currentPC;
    always @(posedge clk or posedge rst) begin
        if (rst)
            currentPC <= INITIAL_PC; // Reset στην αρχική διεύθυνση
        else if (loadPC) begin
            currentPC <= PCSrc ? (currentPC + {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0}) :
(currentPC + 4);
        end
    end
    assign PC = currentPC;

    // 2. Register File
    wire [4:0] readReg1 = instr[19:15];  // Rs1
    wire [4:0] readReg2 = (instr[7:0] == ITYPE) ? 5'bxxxxx : instr[24:20];  // Rs2
    wire [4:0] writeReg = instr[11:7];   // Rd
    wire [31:0] readData1, readData2;

    regfile register_file (
        .clk(clk),
        .readReg1(readReg1),
        .readReg2(readReg2),
        .writeReg(writeReg),
        .writeData(WriteBackData),
        .write(RegWrite),
        .readData1(readData1),
```

```
        .readData2(readData2)
    );

    // 3. Immediate Generation
    wire [31:0] imm_I_L = {{20{instr[31]}}, instr[31:20]}; // Immediate για τύπο I
    wire [31:0] imm_S = {{20{instr[31]}}, instr[31:25], instr[11:7]}; // Immediate για τύπο S
    wire [31:0] imm_B = {{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}; // Immediate για
τύπο B
    wire [31:0] imm = (STYPE == instr[6:0]) ? imm_S : (BTYPE == instr[6:0]) ? imm_B : imm_I_L;

    // 4. ALU
    wire [31:0] alu_op1 = readData1; // Πρώτος τελεστής ALU
    wire [31:0] alu_op2 = ALUSrc ? imm : readData2; // Επιλογή δεύτερου τελεστή ALU
    wire [31:0] alu_result;
    reg [31:0] EX_MEM, MEM_WB;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            MEM_WB <= 32'b0;  // Reset
            EX_MEM <= 32'b0;  // Reset
        end else begin
            MEM_WB <= EX_MEM;
            EX_MEM <= alu_result;  // Latch ALU result for R-type or I-type
        end
    end

    alu alu_unit (
        .op1(alu_op1),
        .op2(alu_op2),
        .alu_op(ALUCtrl),
        .result(alu_result),
        .zero(Zero)
    );

    // 5. Μνήμη Δεδομένων
    assign dAddress = EX_MEM;   // Διεύθυνση δεδομένων στη μνήμη
    assign dWriteData = readData2; // Δεδομένα προς εγγραφή στη μνήμη

    // 6. Write Back
    assign WriteBackData = MemToReg ? dReadData : MEM_WB;

endmodule
```

Υλοποίηση

Η υλοποίηση της μονάδας διαδρομής δεδομένων ($datapath$) ακολουθεί τις απαιτήσεις της εκφώνησης και προσφέρει μια λειτουργική και ολοκληρωμένη λύση για τον επεξεργαστή $RISC-V$.

Η μονάδα διαδρομής δεδομένων περιλαμβάνει τον $Program\ Counter$ (PC), ο οποίος ενημερώνεται συγχρονισμένα με το ρολόι και διαχειρίζεται την εκτέλεση των εντολών, ελέγχοντας τον επόμενο δείκτη εντολής. Το $Register\ File$ επιτρέπει την ανάγνωση και εγγραφή στους καταχωρητές, παρέχοντας δεδομένα στις εντολές για εκτέλεση. Η ALU εκτελεί τις αριθμητικές και λογικές πράξεις, συμπεριλαμβανομένων των πράξεων πρόσθεσης, αφαίρεσης, και συγκρίσεων, ενώ η $Immediate\ Generation$ παράγει τα κατάλληλα άμεσα δεδομένα για χρήση στις εντολές τύπου I και S. Ο $Branch\ Target$ υπολογίζει τη διεύθυνση στόχου για τις εντολές διακλάδωσης, ενώ η μονάδα $Write\ Back$ διασφαλίζει ότι το αποτέλεσμα της ALU ή τα δεδομένα από τη μνήμη γράφονται στους καταχωρητές κατά την εκτέλεση της αντίστοιχης εντολής.

Στην παραπάνω υλοποίηση, η ALU χρησιμοποιεί $pipeline\ registers$ για την αποθήκευση των ενδιάμεσων αποτελεσμάτων μέχρι το τελικό στάδιο του $datapath$. Συγκεκριμένα, τα αποτελέσματα της ALU αποθηκεύονται προσωρινά σε αυτά τους καταχωρητές, μεταφέρονται από τον έναν καταχωρητή στον επόμενο ταυτόχρονα με την μετάβαση των καταστάσεων ($'ripple'$) και μόνο στο στάδιο $WRITE\_BACK$ καταγράφονται στους καταχωρητές του $regfile$, εφόσον απαιτείται.

Ο ίδιος ο επεξεργαστής $RISC-V$ δεν ορίζει ένα συγκεκριμένο σχέδιο για τα $pipeline\ registers$. Τα στάδια του $pipeline$ και το πώς αποθηκεύονται τα ενδιάμεσα αποτελέσματα μεταξύ τους (δηλαδή η ακριβής χρήση των $pipeline\ registers$) εξαρτώνται από την συγκεκριμένη υλοποίηση του επεξεργαστή $RISC-V$ και τον αριθμό των σταδίων που έχει σχεδιαστεί να υποστηρίζει ο επεξεργαστής.
Συνολικά, η υλοποίηση της μονάδας διαδρομής δεδομένων προσφέρει τις ίδιες λειτουργίες με έναν πραγματικό επεξεργαστή $RISC-V$, προσαρμοσμένες στην εκφώνηση της άσκησης. Όλα τα στάδια της εκτέλεσης της εντολής καλύπτονται με τον σωστό συγχρονισμό, εξασφαλίζοντας την ομαλή ροή δεδομένων μέσα στον επεξεργαστή.

• Άσκηση 5 - Ελεγκτής Πολλαπλών Κύκλων

Εισαγωγή

Η Άσκηση 5 αφορά τη δημιουργία ενός ελεγκτή πολλαπλών κύκλων ($multi-cycle\ controller$) για έναν επεξεργαστή $RISC-V$ που θα εκτελεί κάθε εντολή σε πέντε κύκλους ρολογιού. Στόχος είναι να σχεδιαστεί και να υλοποιηθεί ο ελεγκτής που θα καθοδηγεί τη διαδρομή δεδομένων ($datapath$) στον επεξεργαστή και θα διαχειρίζεται τις εντολές μέσα από πέντε διαδοχικούς κύκλους ρολογιού, καθορίζοντας τις ενέργειες που πρέπει να εκτελούνται σε κάθε φάση της εκτέλεσης της εντολής.

Σχεδιασμός Ελεγκτή Πολλαπλών Κύκλων

Ο ελεγκτής θα περιλαμβάνει μια μηχανή καταστάσεων (FSM) που θα χωρίζει τη διαδικασία εκτέλεσης της εντολής σε πέντε βασικά στάδια: IF ($Instruction\ Fetch$), ID ($Instruction\ Decode$), EX ($Execute$), MEM ($Memory$), και WB ($Write\ Back$). Σε κάθε στάδιο, ο ελεγκτής θα ενεργοποιεί τα κατάλληλα σήματα για να καθοδηγήσει τη διαδρομή δεδομένων και να ελέγξει τις ενέργειες που πρέπει να γίνουν για κάθε εντολή.

Το κύκλωμα αυτό απαιτεί την ακριβή υλοποίηση των ακόλουθων λειτουργιών:
- Στη φάση IF ($Instruction\ Fetch$), ο μετρητής προγράμματος (PC) πρέπει να φορτώνει την επόμενη εντολή από τη μνήμη εντολών.
- Στη φάση ID ($Instruction\ Decode$), η εντολή πρέπει να αποκωδικοποιείται για να διαβαστούν οι καταχωρητές και να προσδιοριστούν τα δεδομένα που θα χρησιμοποιηθούν.
- Στη φάση EX ($Execute$), η ALU πρέπει να εκτελεί την αριθμητική ή λογική πράξη.
- Στη φάση MEM ($Memory$), η μνήμη πρέπει να διαβάσει ή να γράψει δεδομένα.
- Στη φάση WB ($Write\ Back$), τα αποτελέσματα πρέπει να εγγραφούν στους καταχωρητές.

Ο έλεγχος του επεξεργαστή για κάθε στάδιο απαιτεί την υλοποίηση των κατάλληλων σημάτων ελέγχου για κάθε μονάδα της διαδρομής δεδομένων ($datapath$), όπως οι καταχωρητές, η ALU, η μνήμη, κλπ.

Θύρες Εισόδου:

clk: Η θύρα αυτή μεταφέρει το σήμα ρολογιού και είναι απαραίτητη για τη συγχρονισμένη λειτουργία του ελεγκτή, καθώς και για τον έλεγχο των χρονικών φάσεων των υποσυστημάτων του επεξεργαστή.
rst: Το σήμα επαναφοράς ($reset$) ενεργοποιείται για να επαναφέρει τα υποσυστήματα του επεξεργαστή στην αρχική τους κατάσταση. Είναι ένα σήμα υψηλής προτεραιότητας, το οποίο επηρεάζει τις βασικές διεργασίες κατά την εκκίνηση ή την επαναφορά του συστήματος.
instr: Αυτή η θύρα μεταφέρει την εντολή από τη μνήμη και είναι κρίσιμη για τη διαδικασία αποκωδικοποίησης των εντολών και την επιλογή των κατάλληλων σημάτων ελέγχου για κάθε τύπο εντολής.
PCSrc: Αυτό το σήμα ελέγχει αν η διεύθυνση του PC θα ενημερωθεί με την κανονική τιμή ή αν θα γίνει $branch$, αναλόγως της συνθήκης της διακλάδωσης.
ALUSrc: Αυτό το σήμα αποφασίζει αν ο δεύτερος τελεστής της ALU θα προέρχεται από τον καταχωρητή ή από το άμεσο ($immediate$) πεδίο της εντολής.
RegWrite: Ενεργοποιεί τη διαδικασία εγγραφής στους καταχωρητές μετά από την εκτέλεση της εντολής, είτε από την ALU είτε από τη μνήμη.
MemToReg: Καθορίζει αν τα δεδομένα που γράφονται στους καταχωρητές προέρχονται από τη μνήμη ή από την ALU.
ALUCtrl: Ελέγχει την λειτουργία της ALU (όπως ADD, SUB, AND, OR, κλπ.), δίνοντας τις κατάλληλες οδηγίες για τις αριθμητικές και λογικές πράξεις.
loadPC: Ενεργοποιεί την ενημέρωση του PC στην εκτέλεση της επόμενης εντολής, είτε μέσω της κανονικής αύξησης του ή μέσω της διακλάδωσης.
dReadData: Τα δεδομένα που διαβάζονται από τη μνήμη και ενδέχεται να χρειαστούν για εγγραφή στους καταχωρητές.

Θύρες Εξόδου:
PC: Η έξοδος του προγράμματος μετρητή (PC), που χρησιμοποιείται για την αποθήκευση και ενημέρωση της διεύθυνσης της επόμενης εντολής που θα εκτελεστεί.
Zero: Το σήμα μηδενισμού της ALU, το οποίο υποδεικνύει αν το αποτέλεσμα της ALU είναι μηδέν. Χρησιμοποιείται για τις συνθήκες διακλάδωσης, όπως στο BEQ ($branch\ if\ equal$).
dAddress: Η διεύθυνση για τη μνήμη δεδομένων, η οποία υπολογίζεται στην ALU ή από το άμεσο ($immediate$) πεδίο και καθορίζει που θα πραγματοποιηθεί η ανάγνωση ή η εγγραφή δεδομένων στη μνήμη.
dWriteData: Τα δεδομένα που πρέπει να γράφουν στη μνήμη δεδομένων, τα οποία προέρχονται είτε από τους καταχωρητές είτε από την ALU.
WriteBackData: Τα δεδομένα που προορίζονται για εγγραφή στους καταχωρητές, τα οποία μπορεί να προέρχονται από την ALU ή από τη μνήμη, αναλόγως του αν η εντολή είναι τύπου $Load$ ή άλλου τύπου εντολής.

```
module top_proc #(
    parameter INITIAL_PC = 32'h00400000  // Initial PC value
)(
    input wire clk,                      // Clock signal
    input wire rst,                      // Synchronous reset signal
    input wire [31:0] instr,             // Instruction fetched from instruction memory
    input wire [31:0] dReadData,         // Data read from data memory
    output [31:0] PC,                 // Program Counter
    output wire [31:0] dAddress,         // Address sent to data memory
    output wire [31:0] dWriteData,       // Data to write to memory
    output reg MemRead,                  // Memory read enable
    output reg MemWrite,                 // Memory write enable
```

```
    output wire [31:0] WriteBackData     // Data written back to register file
);

    // Control Signals
    reg PCSrc, ALUSrc, RegWrite, MemToReg, loadPC;
    reg [3:0] ALUCtrl;

    // Wires for datapath
    wire Zero;
    wire [31:0] alu_result;

    // Internal state register for FSM
    reg [2:0] state;  // FSM state encoding

    // FSM states
    localparam FETCH   = 3'b000,            // 0
               DECODE  = 3'b001,            // 1
               EXECUTE = 3'b010,            // 2
               MEMORY  = 3'b011,            // 3
               WRITE_BACK = 3'b100,         // 4
               NONE = 3'b111;               // 7

    // Operation Types
    localparam RTYPE = 7'b0110011,
               ITYPE = 7'b0010011,
               STYPE = 7'b0100011,
               BTYPE = 7'b1100011,
               LOAD = 7'b0000011;

    // Datapath instantiation
    datapath #(
        .INITIAL_PC(INITIAL_PC)
    ) dp (
        .clk(clk),
        .rst(rst),
        .instr(instr),
        .PCSrc(PCSrc),
        .ALUSrc(ALUSrc),
        .RegWrite(RegWrite),
        .MemToReg(MemToReg),
        .ALUCtrl(ALUCtrl),
        .loadPC(loadPC),
        .dReadData(dReadData),
        .PC(PC),
        .Zero(Zero),
        .dAddress(dAddress),
        .dWriteData(dWriteData),
        .WriteBackData(WriteBackData)
    );

    // FSM: Sequential logic to manage states
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= DECODE;  // Start in the FETCH state after reset
        end else begin
            case (state)
                FETCH: begin
                    state <= DECODE;  // Move to DECODE after fetching the instruction
                end
                DECODE: begin
                    state <= EXECUTE;  // Decode the instruction and prepare for execution
                end
                EXECUTE: begin
                    // Move to MEMORY for load/store or WRITE_BACK for arithmetic/logical instructions

                        state <= MEMORY;

                end
                MEMORY: begin
                    state <= WRITE_BACK;  // Move to WRITE_BACK after memory access
                end
                WRITE_BACK: begin
                    state <= FETCH;  // After writing back, go back to FETCH
                end
                NONE: begin
                    state <= FETCH;
                end
                default: state <= FETCH;  // Default to FETCH state
            endcase
        end
    end

    // FSM: Combinational logic for control signals
    always @(*) begin
        // Default control signal values
       RegWrite = 0;
       MemToReg = 0;
       MemRead = 0;
       MemWrite = 0;
       ALUCtrl = 4'b0000;
```

```
        case (state)
            FETCH: begin
                loadPC = 0;
            end
            DECODE: begin
                PCSrc = 0;
                // No additional controls in this state; decode happens implicitly in the datapath
            end
            EXECUTE: begin
                // ALU operations depend on the instruction type
                case (instr[6:0])
                    RTYPE: begin  // R-type (ADD, SUB, etc.)
                       case ({instr[30], instr[14:12]})
                       4'b0111: begin
                            ALUCtrl = 4'b0000;
                        end
                        4'b0110: begin
                            ALUCtrl = 4'b0001;
                        end
                        4'b0000: begin
                            ALUCtrl = 4'b0010;
                        end
                        4'b1000: begin
                            ALUCtrl = 4'b0110;
                        end
                        4'b0010: begin
                            ALUCtrl = 4'b0100;
                        end
                        4'b0101: begin
                            ALUCtrl = 4'b1000;
                        end
                        4'b0001: begin
                            ALUCtrl = 4'b1001;
                        end
                        4'b1101: begin
                            ALUCtrl = 4'b1010;
                        end
                        4'b0100: begin
                            ALUCtrl = 4'b0101;
                        end
                        endcase
                    end
                     ITYPE: begin  // I-type (ADDI, ANDI, etc.)
                        ALUSrc = 1;  // Use immediate as the second operand
                        case (instr[14:12])
                        3'b111: begin
                            ALUCtrl = 4'b0000;
                        end
                        3'b110: begin
                            ALUCtrl = 4'b0001;
                        end
                        3'b000: begin
                            ALUCtrl = 4'b0010;
                        end
                        3'b010: begin
                            ALUCtrl = 4'b0100;
                        end
                        3'b101: begin
                            ALUCtrl = instr[30] ? 4'b1010 : 4'b1000;
                       end
                        3'b001: begin
                            ALUCtrl = 4'b1001;
                        end
                        3'b100: begin
                            ALUCtrl = 4'b0101;
                        end

                        default: begin
                            ALUCtrl = 4'b1111;
                        end
                        endcase
                    end
                     BTYPE: begin  // Branch (BEQ, etc.)
                        ALUCtrl = 4'b0110;  // SUB for comparison
                        PCSrc = (Zero) ? 1 : 0;  // Branch if condition is met
                     end
                     LOAD, STYPE: begin
                        ALUCtrl = 4'b0010;
                        ALUSrc = 1;
                     end
                endcase
            end
             MEMORY: begin
                case (instr[6:0])
                    LOAD: begin  // Load (LW)
                        MemRead = 1;  // Enable memory read
                    end
                    STYPE: begin  // Store (SW)
                        MemWrite = 1;  // Enable memory write
                    end
                endcase
```

```
            end
            WRITE_BACK: begin
                loadPC = 1;
                ALUSrc = 0;
                case (instr[6:0])
                    LOAD: begin  // Load (LW)
                        RegWrite = 1;  // Write memory data to the register file
                        MemToReg = 1;  // Select memory as the data source
                    end
                    RTYPE, ITYPE: begin  // R-type and I-type instructions
                        RegWrite = 1;  // Write ALU result to the register file
                        MemToReg = 0;  // Select ALU result as the data source
                    end
                endcase
            end
        endcase
    end

endmodule
```

Υλοποίηση

Η παραπάνω υλοποίηση αποτελεί έναν ελεγκτή πολλαπλών κύκλων που εκτελεί κάθε εντολή σε πέντε κύκλους ρολογιού. Για να ικανοποιηθεί η εκφώνηση, το πρώτο βήμα είναι η αρχικοποίηση του $datapath$ που δημιουργήθηκε στην προηγούμενη άσκηση. Αυτό περιλαμβάνει τη σύνδεση της παραμέτρου $INITIAL\_PC$ του $module\ top\_proc$ στην μονάδα $datapath$, η οποία καθορίζει την αρχική τιμή του προγράμματος που εκτελείται.

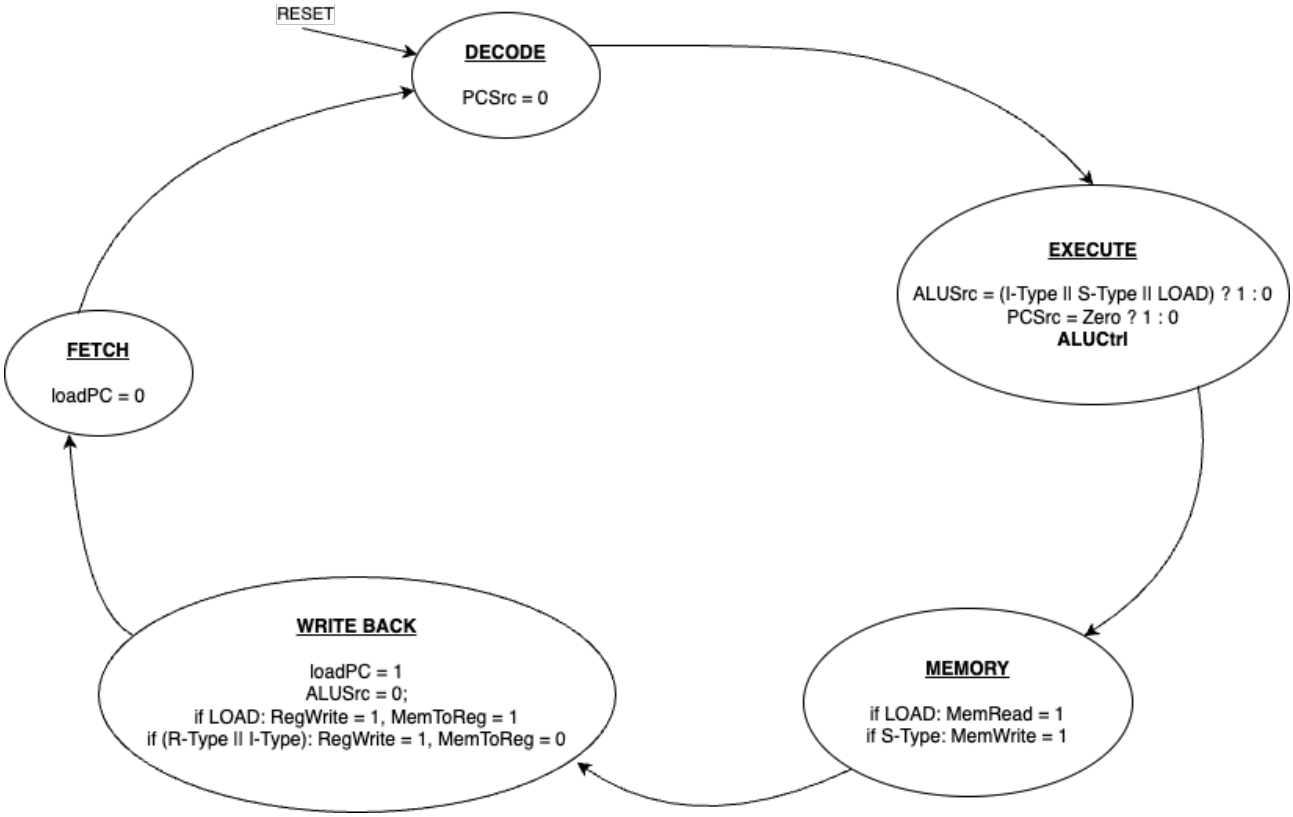
Το επόμενο βήμα είναι η δημιουργία μιας μηχανής καταστάσεων (FSM) που διαχειρίζεται τις πέντε καταστάσεις εκτέλεσης της εντολής: IF ($Instruction\ Fetch$), ID ($Instruction\ Decode$), EX ($Execute$), MEM ($Memory$), και WB ($Write\ Back$). Κάθε κατάσταση αντιστοιχεί σε έναν συγκεκριμένο κύκλο του επεξεργαστή, και ο έλεγχος των σημάτων ελέγχου για τη διαδρομή δεδομένων καθορίζεται ανάλογα με την κατάσταση στην οποία βρίσκεται ο επεξεργαστής.

Η δημιουργία του $ALUCtrl$ απαιτεί την αποκωδικοποίηση της εντολής με βάση τα $bits\ opcode$, $funct3$ και $funct7$. Επιπλέον, τα σήματα $MemRead$ και $MemWrite$ χρησιμοποιούνται για τις εντολές που αφορούν τη μνήμη ($LOAD$, $STORE$), και το σήμα $RegWrite$ ελέγχει την εγγραφή δεδομένων στο αρχείο καταχωρητών μόνο στην κατάσταση $Write\ Back$ (WB).

Το σήμα $PCSrc$ υποδεικνύει αν πρέπει να γίνει διακλάδωση και υπολογίζεται με βάση το αποτέλεσμα της ALU και τη συνθήκη μηδενισμού ($Zero$). Επίσης, κατά τη διάρκεια του σταδίου $Memory$, ενεργοποιούνται τα σήματα $MemRead$ και $MemWrite$ ανάλογα με την εντολή.

Αυτός ο σχεδιασμός περιλαμβάνει τα απαραίτητα σήματα ελέγχου για την εκτέλεση κάθε τύπου εντολής ($R-type$, $I-type$, $S-type$, $B-type$) και διασφαλίζει τη σωστή λειτουργία του επεξεργαστή με πέντε καταστάσεις.

Διάγραμμα FSM



Η αρχική κατάσταση του FSM είναι η κατάσταση $DECODE$. Προσπερνάται η κατάσταση $FETCH$ διότι ο επεξεργαστής ξεκινά με φορτωμένη την πρώτη εντολή. Η μετάβαση στην επόμενη κατάσταση γίνεται αυτόματα. Σε κάθε άνοδο του παλμού του ρολογιού η μηχανή περνάει στην επόμενη κατάσταση. Δεν υπάρχει σήμα που να αποτρέπει την ροή αυτή, υπάρχουν ωστόσο σήματα τα οποία είναι χαρακτηριστικά για κάθε κατάσταση, όπως ακριβώς περιγράφονται στην εκφώνηση της άσκησης. Στην κατάσταση $EXECUTE$ το σήμα $ALUSrc$ ενεργοποιείται αν η εντολή που εκτελείται είναι $R-TYPE$, $I-TYPE$ ή $LOAD$. Ακόμη, το $PCSrc$ ενεργοποιείται αν η έξοδος $ZERO$ της ALU είναι κι αυτή ενεργή σε εντολή τύπου $B-TYPE$, σημαίνοντας ότι πρόκειται για βρόγχο. Στην κατάσταση $MEMORY$ ενεργοποιείται το σήμα $MemRead$ για εντολή $LOAD$, ενώ για εντολή $STORE$ ενεργοποιείται το σήμα $MemWrite$. Στην κατάσταση $WRITE\ BACK$ ενεργοποιείται το σήμα $loadPC$ προκειμένου να φορτώσει ο επεξεργαστής την νέα εντολή από την μνήμη, 'καθαρίζεται' το σήμα $ALUSrc$ και υπάρχουν δυο επιλογές για την εγγραφή. Αν πρόκειται για εγγραφή από τη μνήμη ενεργοποιείται το σήμα $MemToReg$, ενώ αν πρόκειται για εγγραφή σε καταχωρητή ενεργοποιείται το σήμα $RegWrite$. Στην κατάσταση $FETCH$ απενεργοποιείται το σήμα $loadPC$ και η εντολή θα είναι διαθέσιμη στον επόμενο κύκλο. Στην κατάσταση $DECODE$ 'καθαρίζεται' το σήμα $PCSrc$ και το FSM περνάει στη επόμενη κατάσταση ολοκληρώνοντας τον κύκλο και την επεξεργασία της εντολής.

Έλεγχος

Δημιουργήθηκε ένα $module\ \ top\_proc\_tb.v$ όπως ζητήθηκε στην εκφώνηση το οποίο εκτελεί τις εντολές που είναι αποθηκευμένες στο αρχείο $rom\_bytes.data$.

```
`include "top_proc.v"
`include "rom.v"
```

```
`include "ram.v"
`include "datapath.v"
`include "alu.v"
`include "regfile.v"

`timescale 1ns/1ns

module top_proc_tb;

    // Clock and reset signals
    reg clk;
    reg rst;

    // Inputs and outputs to/from top_proc
    wire [31:0] PC;
    wire [31:0] dAddress;
    wire [31:0] dWriteData;
    wire [31:0] WriteBackData;
    wire MemRead;
    wire MemWrite;
    wire [31:0] instr;       // From ROM
    wire [31:0] dReadData;  // From RAM

    // Instantiate ROM and RAM
    INSTRUCTION_MEMORY rom_inst (
        .clk(clk),
        .addr(PC[8:0]),   // Instruction address (word-aligned)
        .dout(instr)       // Fetched instruction
    );

    DATA_MEMORY ram_inst (
        .clk(clk),
        .we(MemWrite),
        .addr(dAddress[8:0]), // Data address (word-aligned)
        .din(dWriteData),
        .dout(dReadData)
    );

    // Instantiate top_proc
    top_proc #(
        .INITIAL_PC(32'h00400000)  // Initial program counter
    ) uut (
        .clk(clk),
        .rst(rst),
        .instr(instr),
        .dReadData(dReadData),
        .PC(PC),
        .dAddress(dAddress),
        .dWriteData(dWriteData),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .WriteBackData(WriteBackData)
    );

    // Clock generation
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        clk = 0;
        forever #5 clk = ~clk;  // 10ns clock period
    end

    // Test sequence
    initial begin
        // Initialize reset
        rst = 1;
        #2;
        rst = 0;

        // Wait for simulation to complete
        #1050;  // Run the simulation for a sufficient amount of time
        $finish;
    end

endmodule
```

Παρακάτω είναι οι αποκωδικοποιημένες εντολές που βρίσκονται στο αρχείο $rom\_bytes.data$.

```
1:    addi x1, x0, 7          Expected result: x1 = 7
5:    addi x2, x0, 21         Expected result: x2 = 21
9:    add x3, x1, x2          Expected result: x3 = 28
13:   addi x4, x0, -9         Expected result: x4 = -9
17:   addi x5, x2, -17        Expected result: x5 = 4
21:   add x6, x5, x4          Expected result: x6 = -5
```

```
25:   sub x7, x3, x2          Expected result: x7 = 7
29:   sll x8, x7, x5          Expected result: x8 = 112
33:   slt x9, x4, x8          Expected result: x9 = 1
37:   xor x10, x8, x2         Expected result: x10 = 101
41:   and x11, x10, x8        Expected result: x11 = 96
45:   srl x12, x11, x9        Expected result: x12 = 48
49:   or x13, x12, x3         Expected result: x13 = 60
53:   sra x14, x4, x9         Expected result: x14 = -5
57:   sw x11, 0(x5)           Expected result: RAM[4] = 96
61:   lw x15, 0(x5)           Expected result: x15 = 96
65:   andi x16, x8, -45       Expected result: x16 = 80
69:   ori x17, x16, 22        Expected result: x17 = 86
73:   srli x18, x13, 1        Expected result: x18 = 30
77:   beq x15, x11, 16       Expected result: PC = PC + 16
----
97:   slti x9, x18, 15        Expected result: x9 = 0
101:  xori x19, x8, 58        Expected result: x19 = 74
105:  slli x20, x17, 1        Expected result: x20 = 172
109:  srai x5, x15, 2         Expected result: x5 = 24
```

Ακολουθούν οι κυματομορφές για την εκτέλεση των παραπάνω εντολών.