# ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ HW-I

ΑΝΑΦΟΡΑ ΕΡΓΑΣΙΑΣ ΕΞΑΜΗΝΟΥ

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ 2023-24

ΑΝΑΣΤΑΣΙΟΣ ΣΤΕΡΓΙΟΥ| 10079
ΤΗΜΜΥ ΑΠΘ | anasster@ece.auth.gr

# ΕΙΣΑΓΩΓΗ

Το παρόν έγγραφο αποτελεί την αναφορά πάνω στην εργασία εξαμήνου του μαθήματος *Ψηφιακά Συστήματα HW – I*. Ως ζητούμενο η εργασία έχει τη σχεδίαση ενός επεξεργαστή σε αρχιτεκτονική *RISC – V*. Η εργασία δομείται σε πέντε ασκήσεις, των οποίων τα αποτελέσματα και ο κώδικας θα παρουσιαστούν. Κατά τη διάρκεια της αναφοράς, θα παρατίθενται τμήματα του κώδικα τα οποία θα είναι σε γραμματοσειρά `Consolas`, σε μαύρο φόντο ως εξής:

```
example code
```

# ΑΣΚΗΣΗ 1

Η άσκηση αυτή είχε ως ζητούμενο την κατασκευή μίας αριθμητικής – λογικής μονάδας (arithmetic – logic unit / ALU). Η μονάδα αυτή έχει ως σκοπό να εκτελεί διάφορες πράξεις μέσα στον επεξεργαστή, μεταξύ δύο τελεστέων, μήκους 32 bits. Οι πράξεις που θα εκτελούνται μέσα στην ALU θα είναι οι εξής:

- AND
- OR
- ADD (Πρόσθεση)
- SUB (Αφαίρεση)
- LT (Μικρότερο από)
- SRL (Shift Right Logical)
- SLL (Shift Left Logical)
- SRA (Shift Right Arithmetic)
- XOR

Το ποια πράξη θα εκτελέσει μεταξύ των τελεστέων η ALU ορίζεται από ένα σήμα 4-bit, με αποτέλεσμα η ALU να λειτουργεί επί της ουσίας σαν ένας πολυπλέκτης, όπου ανάλογα με το σήμα select παράγει το αποτέλεσμα της αντίστοιχης πράξης.

Σε Verilog η ALU υλοποιείται ως εξής:

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

Όπου ως ορίσματα εισόδου έχει τα προαναφερθέντα σήματα (τελεστέοι και σήμα επιλογής πράξης), και σαν έξοδο έχει το αποτέλεσμα της πράξης (επίσης 32 bits), καθώς και ένα σήμα 1 bit, το `zero`, που ειδοποιεί αν το αποτέλεσμα είναι μηδενικό ή όχι. Η υλοποίηση του πολυπλέκτη ALU γίνεται παρακάτω:

```
    case(alu_op)
    ALUOP_AND: result = op1 & op2;
    ALUOP_OR: result = op1 | op2;
    ALUOP_ADD: result = op1 + op2;
    ALUOP_SUB: result = op1 - op2;
    ALUOP_SLT: result = (-op1 < -op2) ? 32'b1 : 32'b0;
    ALUOP_SRL: result = op1 >> op2[4:0];
    ALUOP_SLL: result = op1 << op2[4:0];
    ALUOP_SRA: result = -(-op1 >>> op2[4:0]);
    ALUOP_XOR: result = op1 ^ op2;
    default: result = 32'hzzzzzzzz;
    endcase
```

Όπως φαίνεται, έχουν οριστεί παραμετρικά όλες οι πιθανές τιμές του σήματος επιλογής για κάθε πράξη, σύμφωνα με τις τιμές που δίνονται στην εκφώνηση. Επιπλέον, αν δεν δίνεται κάποια έγκυρη τιμή στο σήμα της πράξης, το αποτέλεσμα λαμβάνει by default την τιμή X (don’t care).

Τέλος, ο έλεγχος για μηδενικό αποτέλεσμα γίνεται ως εξής:

```
assign zero = (result == 32'b0) ? 1'b1 : 1'b0;
```

## ΑΣΚΗΣΗ 2

Η άσκηση αυτή έχει ως ζητούμενο τη δημιουργία μίας αριθμομηχανής που χρησιμοποιεί την ALU της προηγούμενης άσκησης για να κάνει πράξεις, και αποθηκεύει σε έναν καταχωρητή το αποτέλεσμα για να το χρησιμοποιήσει στην επόμενη πράξη.

Το κύκλωμα έχει ως εισόδους έναν διακόπτη 16-bit, πέντε 1-bit σήματα, τα οποία αντιπροσωπεύουν κουμπιά τα οποία πιέζονται για επιλογή πράξης, επαναφορά του καταχωρητή στο 0 και εμφάνιση του αποτελέσματος. Ως έξοδο, έχει ένα σήμα 16-bit το οποίο είναι η πιο πρόσφατη τιμή αποθηκευμένη στον καταχωρητή. Το κύκλωμα ορίζεται ως εξής:

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

Σε πρώτο στάδιο, πραγματοποιείται επέκταση προσήμου στα 16-bit σήματα εισόδου της ALU, ούτως ώστε να έχουν μήκος 32 bits.

```
// Sign extend switch
wire[31:0] sw_ext;
assign sw_ext = {{16{sw[15]}}, sw};

// LED sign extend
wire[31:0] led_ext;
assign led_ext = {{16{led[15]}}, led};
```

Στη συνέχεια, υπολογίζεται το σήμα της επιλογής της πράξης της ALU όπως υποδεικνύεται στην εκφώνηση μέσω ενός αποκωδικοποιητή που υλοποιείται ξεχωριστά:

```
module decoder(
    input wire btnc, btnl, btnr,
    output wire[3:0] alu_op
);
// Module for producing the bits of alu_op
wire op0, op1, op2, op3;

assign op0 = ((~btnr) & btnl) | ((btnc ^ btnl) & btnr); // 1st bit
assign op1 = (btnr & btnl) | ((~btnl) & (~btnc)); // 2nd bit
assign op2 = ((btnr & btnl) | (btnr ^ btnl)) & (~btnc); // 3rd bit
assign op3 = (((~btnr) & btnc) | (btnr ^~ btnc)) & btnl; // 4th bit
assign alu_op = {op3, op2, op1, op0};

endmodule
```

Εφόσον λοιπόν έχουν παραχθεί οι είσοδοι της ALU, υπολογίζεται το αποτέλεσμα της πράξης της ALU και το αποτέλεσμα αποθηκεύεται σε έναν καταχωρητή:

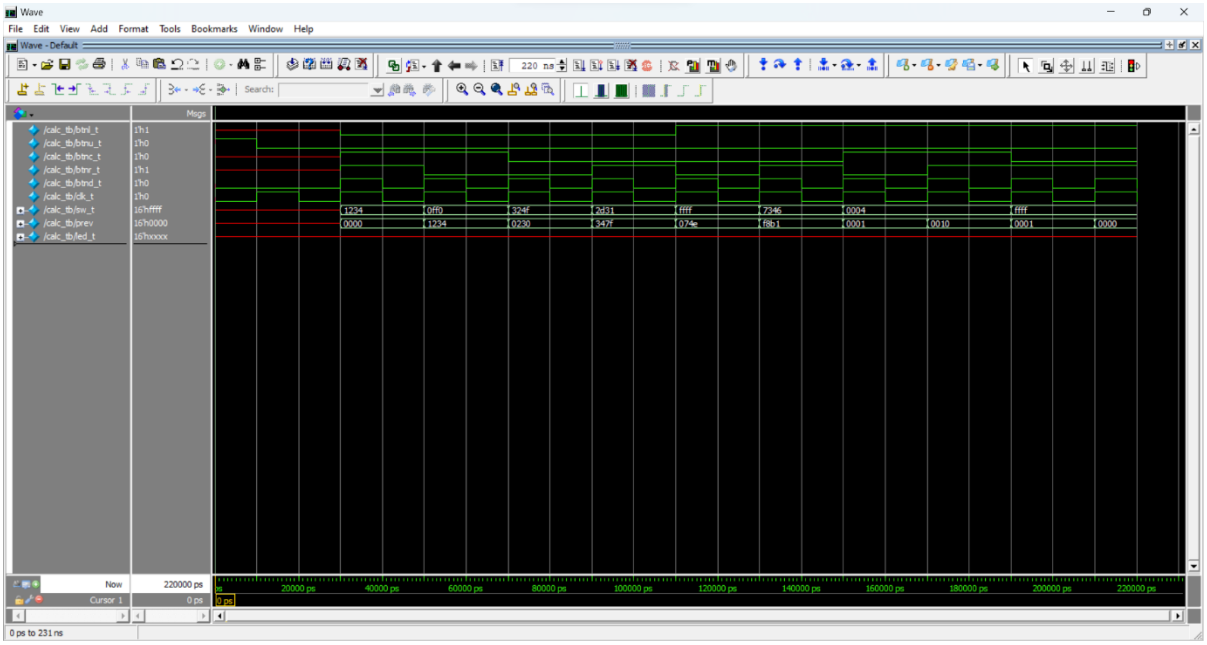
```
// Extract the 16 LSB of the ALU
```

```
wire[15:0] acc_in;
assign acc_in = alu_sub.result[15:0];

// Accumulator register
always @(posedge clk or posedge btnu)
begin
    if(btnu) begin
        // Reset the register at 0 when btnu is pressed
        led = 16'b0;
    end

    else if(btnd) begin
        // Update the register when btnd is pressed
        led = acc_in;
    end
end
```

Για την επαλήθευση του κυκλώματος, παράχθηκε ένα testbench όπου για διάφορες τιμές εισόδου παράγεται το αποτέλεσμα. Πράγματι, η ALU λειτουργεί σωστά καθώς όπως φαίνεται και από τις κυματομορφές:



*Εικόνα 1 - Κυματομορφές του testbench για την Άσκηση 2*

Όπως μπορεί να παρατηρηθεί, οι τιμές της πράξης εμφανίζονται στον επόμενο κύκλο ρολογιού, καθώς κάθε κύκλο ρολογιού ανανεώνουμε το παλαιό αποτέλεσμα ώστε να μπει στην επόμενη πράξη. Η προσομοίωση έγινε για σήμα ρολογιού με περίοδο 20 ns και duty cycle 50%.

# ΑΣΚΗΣΗ 3

Η άσκηση αυτή έχει ως ζητούμενο τη δημιουργία ενός αρχείου καταχωρητών 32×32bits. Σε αυτό το αρχείο θα γίνεται εγγραφή (write) και ανάγνωση των δεδομένων. Το αρχείο υλοποιείται ως εξής:

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

Ως είσοδο έχουμε το σήμα του ρολογιού, τις διευθύνσεις εγγραφής και ανάγνωσης, τα δεδομένα εγγραφής καθώς και το σήμα εγγραφής, ενώ στην έξοδο έχουμε τα δεδομένα ανάγνωσης. Αρχικά, οι καταχωρητές αρχικοποιούνται με μηδενικά όπως φαίνεται παρακάτω:

```
parameter REG_FILE_SIZE = 32;
parameter DATA_WIDTH = 32;

reg[DATA_WIDTH-1:0] registers[0:REG_FILE_SIZE-1]; // 32 x 32-bit registers
integer i;
// Initialize all registers to 0
initial begin
    for(i = 0; i < REG_FILE_SIZE; i = i + 1) begin
        registers[i] = 32'b0;
    end
end
```

Η διαδικασία της εγγραφής και της ανάγνωσης γίνεται:

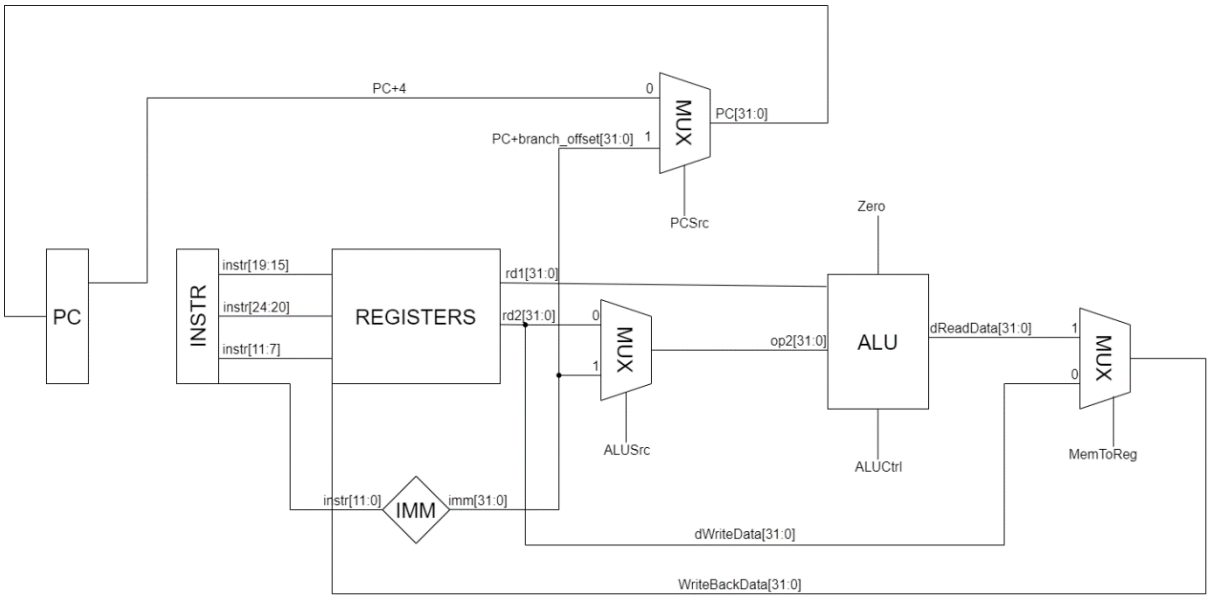
```
always @(posedge clk) begin
    // Read the registers
    readData1 = registers[readReg1];
    readData2 = registers[readReg2];

    // Write to the registers
    if(write) begin
        // Write at next cycle if write and read addresses are the same
        if(writeReg == readReg1 || writeReg == readReg2) begin
            registers[writeReg] <= writeData;
        end
        else begin
            registers[writeReg] = writeData;
        end
    end
end
```

Όπως φαίνεται, όλα γίνονται σύγχρονα με το ρολόι, ενώ στην περίπτωση ταύτισης διεύθυνσης εγγραφής με ανάγνωσης, γίνεται ανάγνωση σύγχρονα με το ρολόι και τα δεδομένα εγγράφονται στον επόμενο κύκλο ρολογιού στη δοσμένη διεύθυνση.

# ΑΣΚΗΣΗ 4

Η άσκηση αυτή, σε συνδυασμό με την επόμενη, έχουν ως σκοπό την κατασκευή του top-level module του επεξεργαστή, το οποίο αποτελείται από τέσσερα τμήματα: τη μνήμη εντολών, τη μνήμη δεδομένων, τη μονάδα ελέγχου καθώς και τη διαδρομή δεδομένων η οποία υλοποιείται και σε αυτήν την άσκηση. Ένα πρόχειρο σχηματικό διάγραμμα της άσκησης 4 με τα σήματα που περιλαμβάνει παρατίθεται στην παρακάτω εικόνα:




*Εικόνα 2 - Σχηματικό διάγραμμα διαδρομής δεδομένων*

Με βάση αυτό το διάγραμμα, υλοποιείται και το module datapath σε Verilog ως εξής:

```
module datapath #(
    parameter INITIAL_PC = 32'h00400000
)
(
    input wire clk, rst, wire[31:0] instr, wire PCSrc, ALUSrc, RegWrite,
MemToReg, wire[3:0] ALUCtrl, wire loadPC,
    output reg[31:0] PC, wire Zero, wire[31:0] dAddress, reg[31:0] dWriteData,
dReadData, WriteBackData
);
```

Όλοι οι καταχωρητές δουλεύουν σύγχρονα με ένα ρολόι `clk` και αρχικοποιούνται σύγχρονα με ένα σήμα επαναφοράς `rst`.

Σε πρώτο στάδιο, αρχικοποιείται το Program Counter (PC) σύμφωνα με την default τιμή που δόθηκε `INITIAL_PC`, και κατόπιν ανανεώνεται η τιμή του ανάλογα με το σήμα `PCSrc`, το οποίο καθορίζει αν θα υπάρχει διακλάδωση ή όχι.

```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        PC <= INITIAL_PC; // Reset at default value
    end
    if(loadPC) begin
        PC <= PCSrc ? PC + branch_offset : PC + 4;
```

```
    end
end
```

Στο επόμενο στάδιο, γίνεται η αρχικοποίηση του αρχείου καταχωρητών.

```
wire[31:0] rd1;
regfile registers(.clk(clk), .readReg1(instr[19:15]), .readReg2(instr[24:20]),
.writeReg(instr[11:7]), .writeData(WriteBackData), .write(RegWrite),
.readData1(rd1), .readData2(dWriteData));
```

Όπως φαίνεται, τα δεδομένα ανάγνωσης από τη δεύτερη θύρα είναι και τα δεδομένα εγγραφής στη μνήμη δεδομένων.

Στη συνέχεια, δημιουργείται το σήμα της άμεσης τιμής (immediate) ανάλογα με τον τύπο εντολής τον οποίο έχουμε. Ο τύπος καθορίζεται από τα bits του opcode του σήματος εντολής `instr`, τα οποία είναι τα 7 LSB, δηλαδή `instr[6:0]`. Όπως φαίνεται και από το manual του επεξεργαστή, οι εντολές που έχουν immediate value είναι οι BEQ, LW, SW και {ADDI, SLTI, XORI, ORI, ANDI}. Οι πέντε τελευταίες μοιράζονται το ίδιο opcode. Κατ' αναλογία λοιπόν με τον πίνακα, έχουμε τη δημιουργία του σήματος immediate:

```
reg[31:0] imm;
always @(instr) begin
    case (instr[6:0]) // Case statement for different instruction types
(According to opcode value)
    7'b1100011: imm = {{19{instr[31]}}, instr[31], instr[7], instr[30:25],
instr[11:8], 1'b0}; // BEQ which has to be shifted left
    7'b0000011: imm = {{20{instr[31]}}, instr[31:20]}; // LW
    7'b0100011: imm = {{20{instr[31]}}, instr[31:25], instr[11:7]}; // SW
    7'b0010011: imm = {{20{instr[31]}}, instr[31:20]}; // I-type
    default: imm = 32'hxxxxxxxx;
    endcase
end
```

Το τέταρτο στάδιο του datapath είναι η δημιουργία της ALU:

```
// ALU
wire[31:0] op2;
assign op2 = ALUSrc ? imm : dWriteData; // Choose operator 2 according to
ALUCtrl signal
alu ALU(.op1(rd1), .op2(op2), .alu_op(ALUCtrl), .zero(Zero),
.result(dAddress));
```

Όπως φαίνεται, με μία λογική πολυπλέκτη επιλέγεται για τον δεύτερο τελεστή είτε η άμεση τιμή, είτε τα δεδομένα ανάγνωσης της δεύτερης θύρας των καταχωρητών, δηλαδή τα δεδομένα εγγραφής στη μνήμη δεδομένων.

Στο προτελευταίο στάδιο του datapath, γίνεται η δημιουργία του branch offset, σε περίπτωση φυσικά όπου έχουμε διακλάδωση (BEQ), όπου και κάνουμε shift left κατά 1 bit την άμεση τιμή.

```
assign branch_offset = (instr[6:0] == 7'b1100011) ? imm << 1 : 32'b0;
```

Τέλος, γίνεται η επανεγγραφή δεδομένων στο αρχείο καταχωρητών, είτε από το αποτέλεσμα της ALU είτε από τα δεδομένα ανάγνωσης της μνήμης δεδομένων:

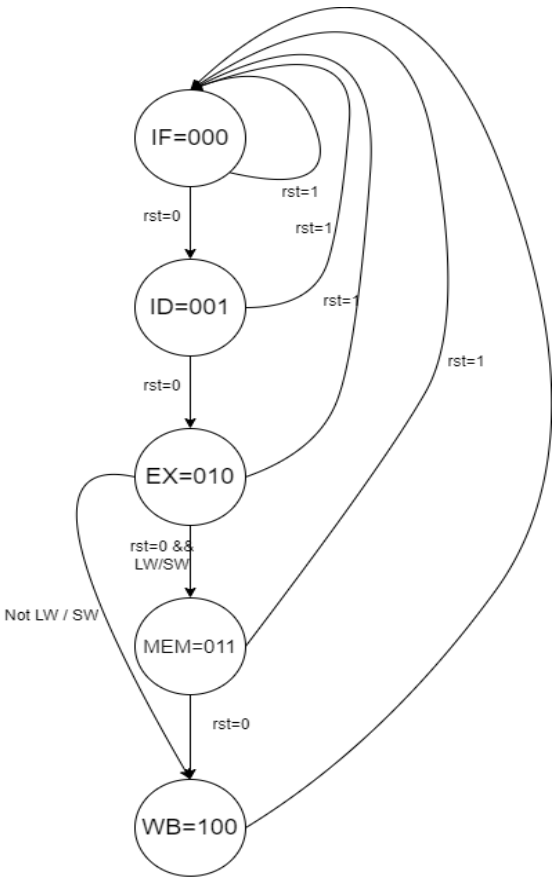
```
assign WriteBackData = MemToReg ? dReadData : ALU.result;
```

# ΑΣΚΗΣΗ 5

Στην πέμπτη και τελευταία άσκηση αυτής της εργασίας, ζητείται η κατασκευή του top-level module του επεξεργαστή RV32-I. Επί της ουσίας, πρέπει να γίνει η δημιουργία ενός FSM το οποίο θα διαχωρίζει τις λειτουργίες του επεξεργαστή σε πέντε διαφορετικές καταστάσεις:

- IF (Instruction Fetch): Η διαδικασία δημιουργίας μίας διεύθυνσης για τη διευθυνσιοδοτούμενη μνήμη εντολών ROM, μέσω του σήματος PC, καθώς και η ανάκτηση της εντολής αυτής.
- ID (Instruction Decode): Η διαδικασία αποκωδικοποίησης της εντολής που ανακτήθηκε από τη μνήμη εντολών, σε διευθύνσεις προς το αρχείο καταχωρητών, καθώς και σε άλλα σήματα με βάση τα opcode, funct3 και funct7 τμήματα της εντολής ούτως ώστε να γίνει αντιληπτό ποια λειτουργία έχει ο επεξεργαστής και να θέσει κάποια σήματα υψηλά ή χαμηλά ανάλογα.
- EX (Execute): Όπως ορίστηκε στο datapath, εκτελείται η λειτουργία της ALU.
- MEM (Memory): Σε περίπτωση που έχουμε εντολές τύπου store ή load, τίθενται υψηλά τα σήματα MemWrite και MemRead αντίστοιχα για κάθε εντολή.
- WB (Write Back): Η διαδικασία εγγραφής στο αρχείο καταχωρητών, καθώς και η δημιουργία ενός νέου PC με σκοπό την εκτέλεση της επόμενης εντολής.

Με βάση τα παραπάνω, το FSM διάγραμμα όπως ζητείται είναι το εξής:




*Εικόνα 3 - Διάγραμμα FSM επεξεργαστή*

Η υλοποίηση αυτού του FSM σε κώδικα γίνεται αρχικά ορίζοντας τις πέντε καταστάσεις του συστήματος:

```
// Create parameters for each state
parameter IF = 3'b000, ID = 3'b001, EX = 3'b010, MEM = 3'b011, WB = 3'b100;
```

Κατόπιν, ορίζεται όπως σε κάθε FSM μία μνήμη καταστάσεων με βάση το σήμα επαναφοράς, ώστε να επαναφέρεται η μηχανή στη default κατάσταση σε περίπτωση ανάγκης.

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

// Create a state memory for the FSM
always @(posedge clk or posedge rst) begin: STATE_MEMORY
    if(rst) begin
        current_state <= IF;
    end
    else begin
        current_state <= next_state;
    end
end
```

Για να μεταβούμε στην επόμενη κατάσταση, όπως φαίνεται και από το διάγραμμα, δεν χρειάζεται να συμβεί τίποτα για τις δύο πρώτες καταστάσεις (IF, ID). Ωστόσο, στην κατάσταση EX πρέπει να ελέγξουμε τι είδους εντολή έχουμε, διότι για όλες τις εντολές πέραν των LW, SW, δεν γίνεται προσπέλαση στη μνήμη RAM των δεδομένων, συνεπώς πάμε κατευθείαν στο στάδιο εγγραφής προς τα πίσω (WB).

```
always @(current_state, rst, instr) begin: NEXT_STATE_LOGIC
    case(current_state)
    IF: next_state = rst ? IF : ID;
    ID: next_state = rst ? IF : EX;
    EX:
    if(rst) begin
        next_state = IF;
    end
    else if((instr[6:2] != 5'b01000) && (instr[6:2] != 5'b00000)) begin // If
we don't have load or store instrudctions, RAM is not accessed
        next_state = WB;
    end
    else begin
        next_state = MEM;
    end
    MEM: next_state = rst ? IF : WB;
    WB: next_state = IF;
    endcase
end
```

Τέλος, κατά τη διάρκεια των πέντε καταστάσεων αλλάζουν οι τιμές των σημάτων `loadPC`, `MemWrite`, `MemRead`, `MemToReg` και `RegWrite`. Σύμφωνα με τις οδηγίες της εκφώνησης για

το τί θα πρέπει να συμβαίνει στα πέντε αυτά σήματα, ορίζεται η λογική της τρέχουσας κατάστασης (current state logic) ως εξής:

```
always @(current_state or instr) begin: CURRENT_STATE_LOGIC
    case(current_state)
    IF: begin loadPC = 1'b0; MemWrite = 1'b0; MemRead = 1'b0; MemToReg = 1'b0;
RegWrite = 1'b0; end
    ID: begin loadPC = 1'b0; MemWrite = 1'b0; MemRead = 1'b0; MemToReg = 1'b0;
RegWrite = 1'b0; end
    EX: begin loadPC = 1'b0; MemWrite = 1'b0; MemRead = 1'b0; MemToReg = 1'b0;
RegWrite = 1'b0; end
    MEM:begin loadPC = 1'b0; MemWrite = (instr[6:2] == 5'b01000) ? 1'b1 :
1'b0; MemRead = (instr[6:2] == 5'b00000) ? 1'b1 : 1'b0; MemToReg = 1'b0;
RegWrite = 1'b0; end
    WB: begin loadPC = 1'b1; MemWrite = 1'b0; MemRead = 1'b1; MemToReg =
instr[6:2] == 5'b00000 ? 1'b1 : 1'b0; RegWrite = (instr[6:0] == 7'b1100011) ||
(instr[6:0] == 7'b0100011) ? 1'b0 : 1'b1; end
    endcase
end
```

Τέλος, στο module αυτό δημιουργήθηκε και μία λογική παραγωγής των σημάτων `ALUCtrl`, `ALUSrc` και `PCSrc`, για την ALU και τους δύο πολυπλέκτες αντίστοιχα. Και πάλι σύμφωνα με την εκφώνηση, ο κώδικας παραγωγής του σήματος για την πράξη της ALU είναι:

```
// Create control logic to find the signal ALUCtrl
always @(instr) begin: CONTROL_LOGIC_ALUCTRL
    // AND - ANDI
    if(((instr[14:12] == 3'b111) && (instr[6:0] == 7'b0110011)) ||
((instr[14:12] == 3'b111) && (instr[6:0] == 7'b0010011))) begin
        ALUCtrl = 4'b0000;
    end
    // OR - ORI
    else if(((instr[14:12] == 3'b110) && (instr[6:0] == 7'b0110011)) ||
((instr[14:12] == 3'b110) && (instr[6:0] == 7'b0010011))) begin
        ALUCtrl = 4'b0001;
    end
    // ADD - ADDI
    else if(((instr[31:25] == 7'b0000000) && (instr[14:12] == 3'b000) &&
instr[6:0] == 7'b0110011) || ((instr[14:12] == 3'b000) && (instr[6:0] ==
7'b0010011))) begin
        ALUCtrl = 4'b0010;
    end
    // BEQ (SUB)
    else if(instr[6:0] == 7'b1100011 && instr[14:12] == 3'b000) begin
        ALUCtrl = 4'b0110;
    end
    // SUB
    else if((instr[31:25] == 7'b0100000) && (instr[14:12] == 3'b000) &&
(instr[6:0] == 7'b0110011)) begin
        ALUCtrl = 4'b0110;
```

```
    end
    // SLT - SLTI
    else if(((instr[31:25] == 7'b0000000) && (instr[14:12] == 3'b010) &&
(instr[6:0] == 7'b0110011)) || (instr[14:12] == 3'b010) && (instr[6:0] ==
7'b0010011)) begin
        ALUCtrl = 4'b0111;
    end
    // SRL - SRLI
    else if(((instr[31:25] == 7'b0000000) && (instr[14:12] == 3'b101) &&
(instr[6:0] == 7'b0110011)) || ((instr[14:12] == 3'b101) && (instr[6:0] ==
7'b0010011))) begin
        ALUCtrl = 4'b1000;
    end
    // SLL - SLLI
    else if(((instr[31:25] == 7'b0000000) && (instr[14:12] == 3'b001) &&
(instr[6:0] == 7'b0110011)) || ((instr[14:12] == 3'b001) && (instr[6:0] ==
7'b0010011))) begin
        ALUCtrl = 4'b1001;
    end
    // SRA - SRAI
    else if(((instr[31:25] == 7'b0100000) && (instr[14:12] == 3'b101) &&
(instr[6:0] == 7'b0110011)) || ((instr[14:12] == 3'b101) && (instr[6:0] ==
7'b0010011))) begin
        ALUCtrl = 4'b1010;
    end
    // XOR - XORI
    else if(((instr[31:25] == 7'b0000000) && (instr[14:12] == 3'b100) &&
(instr[6:0] == 7'b0110011)) || ((instr[14:12] == 3'b100) && (instr[6:0] ==
7'b0010011))) begin
        ALUCtrl = 4'b1101;
    end
    // LW - SW (ADD)
    else if(((instr[14:12] == 3'b010) && (instr[6:0] == 7'b0000011)) ||
((instr[14:12] == 3'b010) && (instr[6:0] == 7'b0100011))) begin
        ALUCtrl = 4'b0010;
    end
end
```

Οι εντολές LW, SW μεταφράζονται ως αφαίρεση, και η BEQ ως πρόσθεση.

Στη συνέχεια, για το σήμα του πολυπλέκτη που αφορά πράξη της ALU με άμεση τιμή:

```
// Create control logic for ALUSrc
always @(instr) begin: CONTROL_LOGIC_ALUSRC
    if(((instr[6:0] == 7'b1100011) && (instr[14:12] == 3'b000)) ||
((instr[6:0] == 7'b0000011) && (instr[14:12] == 3'b010)) || ((instr[6:0] ==
7'b0100011) && (instr[14:12] == 3'b010)) || ((instr[6:0] == 7'b0010011) &&
(instr[14:12] != 3'b011))) begin
        ALUSrc = 1'b1;
    end
```

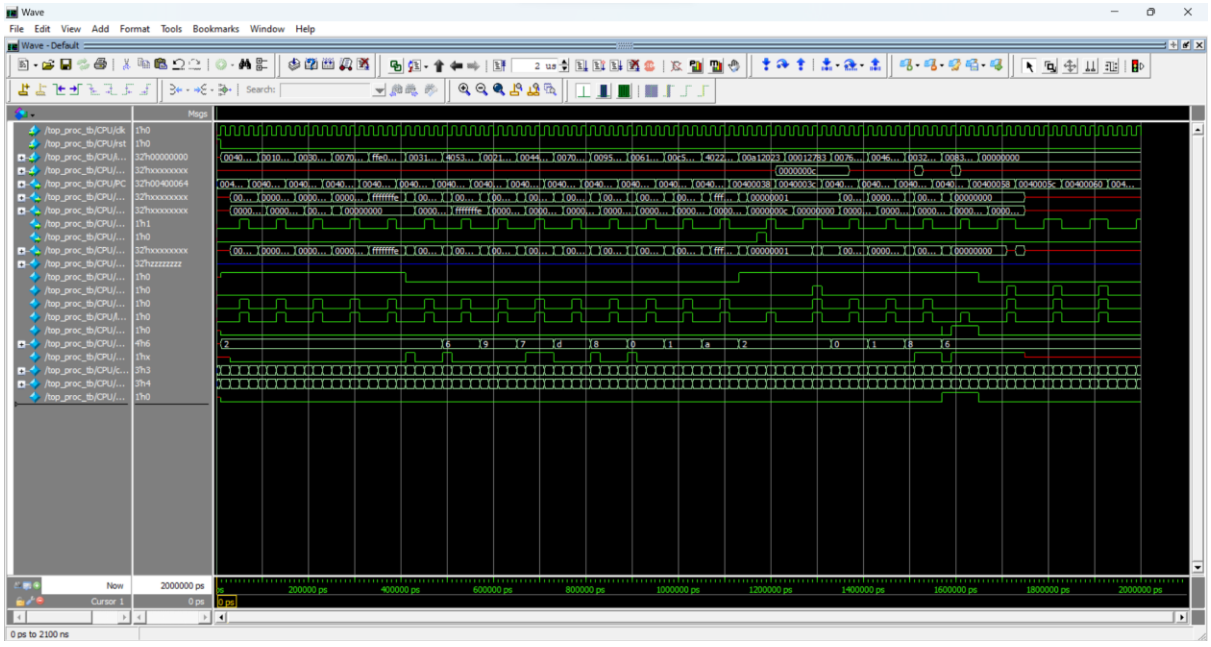
```
    else begin
        ALUSrc = 1'b0;
    end
end
```

Όπως φαίνεται, συμπεριλαμβάνονται όλες οι εντολές τύπου I, καθώς και οι LW, SW και BEQ.

Τέλος, για το σήμα ελέγχου διακλάδωσης ορίζεται ο εξής έλεγχος για εντολή BEQ:

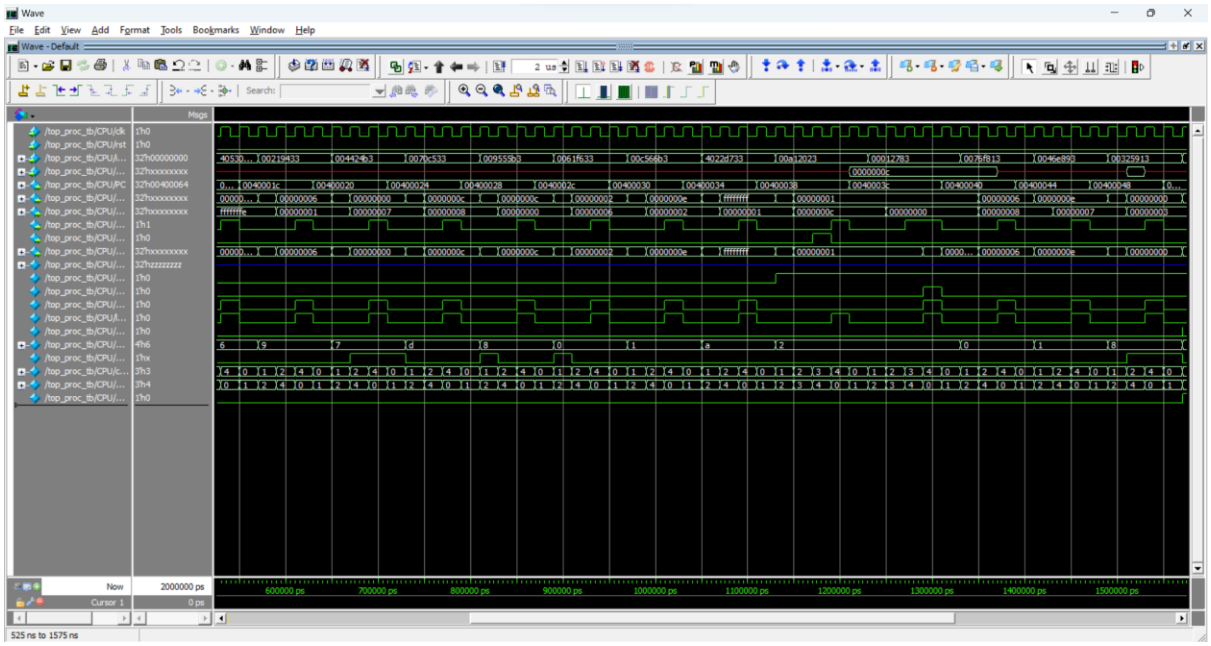
```
// Create control logic for PCSrc
reg Branch;
always @(instr, Zero) begin:CONTROL_LOGIC_PCSRC
    if(instr[6:0] == 7'b1100011 && instr[14:12] == 3'b000) begin
        Branch = 1'b1;
    end
    else begin
        Branch = 1'b0;
    end
    PCSrc = Branch & Zero;
end
```

Για την επαλήθευση της λειτουργίας του κυκλώματος έγινε ένα testbench χρησιμοποιώντας τις έτοιμες υλοποιήσεις RAM και ROM, καθώς και ένα byte stream από ένα αρχείο που μας δόθηκε προκειμένου να δοκιμαστούν όλες οι εντολές. Η προσομοίωση έγινε για 2μs με ρολόι περιόδου 20ns. Τα αποτελέσματα φαίνονται στην παρακάτω κυματομορφή:



*Εικόνα 4 - Αποτελέσματα επεξεργαστή*

Όπως φαίνεται πάνω – πάνω, απεικονίζονται οι κωδικοί της κάθε εντολής που υπαγορεύεται από τον PC σε δεκαεξαδικό σύστημα. Για παράδειγμα, εάν εστιάσουμε σε ένα τμήμα των κυματομορφών του testbench θα φανεί:



*Εικόνα 5 - Δείγμα του testbench*

Ας πάρουμε για παράδειγμα την εντολή με κωδικό $009555b3_{16}=00000000100101010101010110110011_2$. Τα 7 LSB της είναι 0110011 άρα σίγουρα πρόκειται για μία εντολή τύπου R (Register – Register). Από τα bits 14 – 12, φαίνεται ότι έχει πεδίο $funct3=101_2$, και πεδίο $funct7=0000000_2$ δηλαδή είναι η SRL. Πράγματι, το σήμα ALUCtrl που ελέγχει την πράξη της ALU έχει την τιμή $8_{16}=1000_2$, άρα επαληθεύουμε ότι γίνεται αυτή η πράξη. Αντίστοιχα για τις υπόλοιπες, το testbench τυπώνει αυτόματα το όνομα της πράξης ανάλογα με τα σηματικά πεδία της 32-bit εντολής.

# ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ