Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής Ι

ΚΑΡΓΙΩΤΗΣ ΑΛΕΞΑΝΔΡΟΣ 10662

16-01-2024

ΑΣΚΗΣΗ 1

Σε αυτή την άσκηση υλοποιήθηκε ένα module ALU (Arithmetic Logic Unit) η οποία θα υλοποιεί τις ακόλουθες πράξεις: προσημασμένη πρόσθεση, προσημασμένη αφαίρεση, λογικό AND, λογικό OR, λογικό XOR, σύγκριση "Μικρότερο από" και τρεις διαφορετικές πράξεις ολίσθησης.

Το module ζητήθηκε να είναι ακολουθιακό και να έχει τις ακόλουθες θύρες:

Όνομα θύρας	**Κατεύθυνση**	**Σκοπός**
op1	Είσοδος	Τελεστής πράξης 1 σε συμπλήρωμα ως προς 2
op2	Είσοδος	Τελεστής πράξης 2 σε συμπλήρωμα ως προς 2
alu_op	Είσοδος	Δείχνει ποία λειτουργία θα εκτελεστεί
zero	Έξοδος	Δείχνει πότε το αποτέλεσμα της ALU είναι μηδέν
result	Έξοδος	Αποτέλεσμα

```
module alu(
        input [31:0] op1,               // operand 1, two's complement
        input [31:0] op2,               // operand 2, two's complement
        input [3:0] alu_op,             // shows which instruction to execute
        output zero,                    // = 1 when result is 0
        output reg [31:0] result );

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_SLT = 4'b0100;
parameter [3:0] ALUOP_SRL = 4'b1000;
parameter [3:0] ALUOP_SLL = 4'b1001;
parameter [3:0] ALUOP_SRA = 4'b1010;
parameter [3:0] ALUOP_XOR = 4'b0101;


always @ (*)    // (*) includes all signals used inside the always block
        begin
                case(alu_op)
                        ALUOP_AND: result = op1 & op2;                                  // Bitwise AND
                        ALUOP_OR: result = op1 | op2;                                   // Bitwise OR
                        ALUOP_ADD: result = op1 + op2;
                        ALUOP_SUB: result = op1 - op2;
                        ALUOP_SLT: result = ($signed(op1) < $signed(op2));              // result = 1 if ((signed)op1 < (signed)op2), else result = 0
                        ALUOP_SRL: result = op1 >> op2[4:0];                            // Logical Shift Right
                        ALUOP_SLL: result = op1 << op2[4:0];                            // Logical Shift Left
                        ALUOP_SRA: result = $unsigned($signed(op1) >>> op2[4:0]);       // Arithmetic Shift Right
                        ALUOP_XOR: result = op1 ^ op2;                                  // Logical XOR (Bitwise)
                endcase

        end
assign zero = (result == 32'b0) ? 1'b1 : 1'b0; // zero = 1 if result = 0

endmodule
```

Figure 1: Ο κώδικας της alu.v

Δόθηκε προσοχή στο να γίνονται οι πράξεις πάνω σε προσημασμένους αριθμούς. Ο κώδικας της άσκησης είναι απλος, και οι οδηγίες αρκετά επεξηγηματικέ, ώστε να μην χρειάστεί περαιτέρω σχολιασμός.

ΑΣΚΗΣΗ 2

Σε αυτή την άσκηση ζητήθηκε ο σχεδιασμός ενός κυκλώματος αριθμομηχανής που χρησιμοποιεί την ALU που δημιούργησα. Αυτό το κύκλωμα 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. Τα δύο βασικά στοιχεία του κυκλώματος είναι ένας accumulator 16-bit για να κρατά την τρέχουσα τιμή της αριθμομηχανής και η ALU που δημιουργήσατε

Το module calc ζητήθηκε να έχει τις ακόλουθες θύρες:

Όνομα θύρας	**Κατεύθυνση**	**Σκοπός**
clk	Είσοδος	Ρολόι
btnc	Είσοδος	Κεντρικό πλήκτρο
btnl	Είσοδος	Αριστερό πλήκτρο
btnu	Είσοδος	Πάνω πλήκτρο
btnr	Είσοδος	Δεξιά πλήκτρο
btnd	Είσοδος	Κάτω πλήκτρο
sw	Είσοδος	Διακόπτες για την εισαγωγή δεδομένων
led	Έξοδος	LED για την έξοδο του accumulator

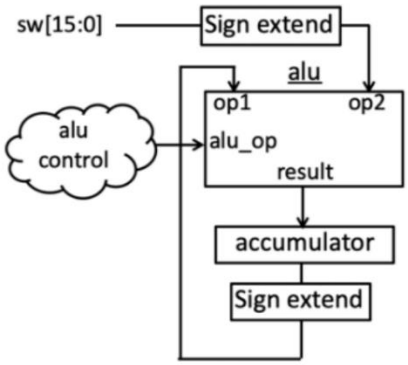


Figure 2: Διάγραμμα ροής της αριθμομηχανής

```
module calc(input clk,          // clock
        input btnc,             // central button
        input btnl,             // left button
        input btnu,             // up button, resets accumulator
        input btnr,             // right button
        input btnd,             // down button
        input [15:0] sw,        // switches for data input
        output reg [15:0] led   // LED for accumulator output
);
        // Internal signals
        reg [15:0] accumulator;         // stores 16 bits of ALU result
        wire [31:0] op1_extended;       // Sign-extended accumulator to 32-bit
        wire [31:0] op2_extended;       // Sign-extended switch input to 32-bit
        wire [31:0] result;             // ALU result
        wire zero;                      // Zero flag from ALU
        wire [3:0] alu_op;              // ALU operation

assign op1_extended = {{16{accumulator[15]}}, accumulator};     // Sign extension of accumulator to 32 bits
assign op2_extended = {{16{sw[15]}}, sw};                       // Sign extension of switch input to 32 bits

// Calc_enc instance
calc_enc my_calc_enc ( .btnc(btnc), .btnl(btnl), .btnr(btnr), .alu_op(alu_op) );

// ALU instance
alu my_alu ( .op1(op1_extended), .op2(op2_extended), .alu_op(alu_op), .result(result), .zero(zero) );

// Accumulator logic
always @(posedge clk) begin
        if(btnu) begin
                accumulator <= 16'b0;   // Reset when btnu is presed
        end
        else if(btnd) begin
                accumulator <= result[15:0];    // Take lower 16 bits of ALU result
        end

        led <= accumulator; // outputs accumulator value to led

end

endmodule
```

Figure 3: Ο κώδικας της calc.v

Επίσης ζητήθηκε να η δημιουργία ενός module calc_enc.v το οποίο θα δημιουργέι το σήμα alu_op βάσει των btnc, btnl, btnr.

```
module calc_enc(input btnc,     // central button
        input btnl,             // left button
        input btnr,             // right button
        output [3:0] alu_op     // specifies with ALU instruction to execute
);

// alu_op bits are assigned based on given tree
assign alu_op[0] = ( ~btnc & btnr ) | ( btnl & btnr );
assign alu_op[1] = ( !btnl & btnc ) | ( btnc & ~btnr );
assign alu_op[2] = ( btnc & btnr ) | ( ( btnl & ~btnc ) & ~btnr );
assign alu_op[3] = ( ( btnl & ~btnc ) & btnr ) | ( ( btnl & btnc ) & ~btnr );

endmodule
```

Figure 4: Ο κώδικας της calc_enc.v

Στην calc_enc κάθε bit της alu_op δημιουργείται βάσει του συνδυασμού btnc, btnl, btnr που επιλεγεται

Στην calc, αρχικοποιείται η calc_enc και η alu και γινεται sign extension των op1 kai op2. Επίσης υλοποιείται η λογική του accumulator σύμφωνα με τα δεδομένα που δίνονται και το Figure 2.

Τέλος, υλοποιήθηκε το testbench, calc_tb:

```
module calc_tb;

        reg clk, btnc, btnl, btnu, btnr, btnd;
        reg [15:0] sw;
        wire [15:0] led;

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

 always #5 clk = ~clk;   // Clock generation: Toggle every 5 time units

initial begin   // Test sequence
        // Initialize inputs
        clk = 0; btnc = 0; btnl = 0; btnu = 0; btnr = 0; btnd = 0; sw = 16'b0;

        // Display header for test results
        $display("Time\t btnl btnc btnr btnd btnu sw (hex)\t LED (hex)");
        $monitor("%0t\t %b   %b   %b   %b   %b   %h\t %h", $time, btnl, btnc, btnr, btnd, btnu, sw, led);

        // Test 1: Reset accumulator
        btnu = 1;
        #10; btnu = 0; #10;             // Press btnu ==> reset operation
                                        // Press btnd ==> apply operation

        // Test 2: ADD operation
        btnl = 0; btnc = 1; btnr = 0; sw = 16'h354a;
        btnd = 1;
        #10; btnd = 0; #20;

        // Test 3: SUBTRACT operation
        btnl = 0; btnc = 1; btnr = 1; sw = 16'h1234;
        btnd = 1;
        #10; btnd = 0; #20;
```

```
        // Test 4: OR operation
        btnl = 0; btnc = 0; btnr = 1; sw = 16'h1001;
        btnd = 1;
        #10; btnd = 0; #20;

        // Test 5: AND operation
        btnl = 0; btnc = 0; btnr = 0; sw = 16'hf0f0;
        btnd = 1;
        #10; btnd = 0; #20;

        // Test 6: XOR operation
        btnl = 1; btnc = 1; btnr = 1; sw = 16'h1fa2;
        btnd = 1;
        #10; btnd = 0; #20;

        // Test 7: ADD operation
        btnl = 0; btnc = 1; btnr = 0; sw = 16'h6aa2;
        btnd = 1;         // Press btnd (apply operation)
        #10; btnd = 0; #20;

        // Test 8: Logical Shift Left
        btnl = 1; btnc = 0; btnr = 1; sw = 16'h0004;
        btnd = 1;
        #10; btnd = 0; #20;

        // Test 9: Arithmetic Shift Right
        btnl = 1; btnc = 1; btnr = 0; sw = 16'h0001;
        btnd = 1;         // Press btnd (apply operation)
        #10; btnd = 0; #20;

        // Test 10: SLT operation (Set Less Than)
        btnl = 1; btnc = 0; btnr = 0; sw = 16'h46ff;
        btnd = 1;         // Press btnd (apply operation)
        #10; btnd = 0; #20;

        // End the simulation
        $stop;
    end
endmodule
```

Figure 5: Ο κώδικας του calc_tb

Στο testbench υλοποιήθηκαν ολες οι πράξεις που ζητούνταν στην εργασία με την σειρά που δίνονταν.

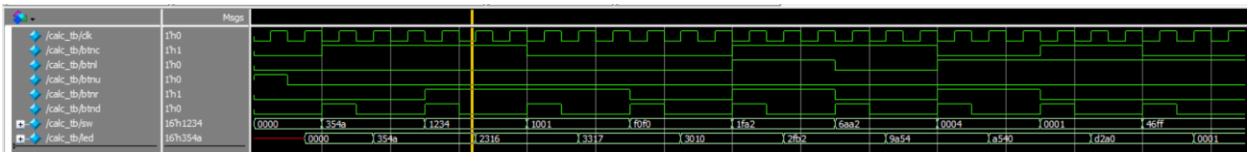


Figure 6: Κυματομορφή Προσομοίωσης άσκησης 2

Το αποτέλεσμα της ALU, που φαίνεται μέσω της μεταβλητής led, για κάθε πράξη είναι ίδιο με το αναμενόμενο αποτελέσμα.

ΑΣΚΗΣΗ 3

Ζητήθηκε ο σχεδιασμός ενός αρχείου καταχωρητών το οποίο θα αποθηκεύει τις τιμές των καταχωρητών που χρησιμοποιούνται από τον επεξεργαστή RISC-V. Το αρχείο περιέχει 32 καταχωρητές και ζητείται να έχει τις παρακάτω θύρες:

Όνομα θύρας	**Κατεύθυνση**	**Σκοπός**
clk	Είσοδος	Ρολόι
readReg1	Είσοδος	Διεύθυνση για την θύρα ανάγνωσης 1
readReg2	Είσοδος	Διεύθυνση για την θύρα ανάγνωσης 2
writeReg	Είσοδος	Διεύθυνση για την θύρα εγγραφής
writeData	Είσοδος	Δεδομένα προς εγγραφή
write	Είσοδος	Σήμα ελέγχου που υποδεικνύει την εγγραφή
readData1	Έξοδος	Δεδομένα ανάγνωσης από τη θύρα 1
readData2	Έξοδος	Δεδομένα ανάγνωσης από τη θύρα 2

Στον κώδικα, η εγγραφή δεδομένων γίνεται σύγχρονα, ενώ η ανάγνωση ασύγχρονα. Έγινε αρχικά προσπάθεια σύγχρονης ανάγνωσης, όμως η Verilog δεν επιτρέπει την χρήση reg στα outputs ενός module. Καθώς είναι συνήθης πρακτική στη σχεδίαση τέτοιων κυκλωμάτων, επιλέχθηκε η απλή λύση της ασύγχρονης εγγραφής. Επίσης δόθηκε βάση στη υλοποίηση του module έτσι ώστε να δίνεται προτεραιότητα στην εγγραφή σε περίπτωση ταυτόχρονης εγγραφής και ανάγνωσης στον ίδιο καταχωρητή.

```
module regfile #(parameter DATAWIDTH = 32)
(
        input clk,                              // clock
        input [4:0] readReg1,                   // address of read port 1
        input [4:0] readReg2,                   // address of read port 2
        input [4:0] writeReg,                   // address of write port
        input [DATAWIDTH-1:0] writeData,        // data to write
        input write,                            // if = 1, data is written on register[writeReg]
        output [DATAWIDTH-1:0] readData1,       // data to be read port 1
        output [DATAWIDTH-1:0] readData2        // data to be read port 2
);
// Create reg array of 32 DATAWIDTH-bit registers and initialize to 0
reg [DATAWIDTH-1:0] register[0:31];
integer i;
initial begin
        for(i=0;i<DATAWIDTH;i=i+1) begin
                register[i] = 0;
        end
end

// Write data to register if allowed and read data from register
always @(posedge clk) begin

        if(write) begin
                register[writeReg] <= writeData;
        end

end

// If there is a write signal to the same register there is a read signal, write first then read
assign readData1 = (write && (writeReg == readReg1)) ? writeData : register[readReg1];
assign readData2 = (write && (writeReg == readReg2)) ? writeData : register[readReg2];

endmodule
```

Figure 7: Ο κώδικας του regfile

ΑΣΚΗΣΗ 4

Ζητείται να σχεδιαστει η μονάδα διαδρομής δεδομένων datapath του επεξεργαστή. Ο σκοπός του μονοπατιού δεδομένων είναι να λαμβάνει μια εντολή (instr), να αποκωδικοποιεί την εντολή αναγνωρίζοντας τον τύπο εντολής και τα πεδία εντολής για να ρυθμίζει κατάλληλα τους τελεστές της και στη συνέχεια να τροφοδοτεί τα σήματα ελέγχου τα οποία δίνονται στις θύρες εισόδου, όπου είναι απαραίτητο.

Το αρχείο περιέχει τις παρακάτω θύρες:

Όνομα θύρας	Κατεύθυνση	Σκοπός
clk	Είσοδος	Ρολόι
rst	Είσοδος	Σύγχρονο reset
instr	Είσοδος	Δεδομένα εντολών από την ROM
PCSrc	Είσοδος	Πηγή του PC
ALUSrc	Είσοδος	Πηγη του 2nd op της ALU
RegWrite	Είσοδος	Ενεργοποίηση εγγραφής δεδομένων στο regfile.v
MemToReg	Είσοδος	Πολυπλέκτης εισόδου στο regfile.v (RAM ή ALU)
ALUCtrl	Είσοδος	Δείχνει ποια λειτουργία πρέπει να εκτελέσει η ALU
loadPC	Είσοδος	Ενημέρωση του PC με νεα τιμή (PC+4 ή branch)
PC	Έξοδος	Program Counter
Zero	Έξοδος	Ένδειξη μηδενισμού ALU, για συνθήκες διακλάδωσης
dAddress	Έξοδος	Διεύθυνση δεδομένων στην RAM

dWriteData	Έξοδος	Δεδομένα που εγγράφονται στη RAM
dReadData	Είσοδος	Δεδομένα που διαβάζονται από την RAM
WriteBackData	Έξοδος	WriteBack δεδομένα που επιστρέφουν στο regfile.v

```
module datapath #(parameter INITIAL_PC = 32'h400000)
(       input clk,
        input rst,              // synchronous reset
        input [31:0] instr,     // instruction data from instuction memory
        input PCSrc,            // PC source
        input ALUSrc,           // source of 2nd ALU operand
        input RegWrite,         // writes data to Registers
        input MemToReg,         // input mux to registers
        input [3:0] ALUCtrl,    // ALU control signal
        input loadPC,           // updates PC
        output [31:0] PC,       // Program Counter
        output Zero,            // 1 if (ALU zero == 1)
        output [31:0] dAddress,         // address of data memory
        output [31:0] dWriteData,       // data to write in data memory
        input [31:0] dReadData,         // data to read from data memory
        output [31:0] WriteBackData     // WriteBack data which return to registers
);
//Internal signals
reg [31:0] imm_out;
wire [31:0] alu_op1;
wire [31:0] alu_op2;
wire [31:0] alu_result;
wire [31:0] branch_target;

// PC Logic
reg [31:0] tempPC;
always @(posedge clk) begin
        $display("ALUSrc: %d and immediate: %d", ALUSrc, imm_out);
        if (rst) begin
                tempPC <= INITIAL_PC;           // reset
                $display("TempPc: %d and PC value: %d", tempPC, PC);
        end else if (loadPC) begin
                if (PCSrc) begin
                        tempPC <= branch_target ;       // branch
                end else begin
                        tempPC <= PC + 4;               // next instruction
                end
                $display("Updating PC. Instr: %d", instr);
        end
end
assign PC = tempPC;
```

```
42
43  // Register variables
44  wire [31:0] readData1, readData2;
45  //Regfile instantiation
46  regfile datapath_regfile (
47          .clk(clk),
48          .readReg1(instr[19:15]),        // rs1
49          .readReg2(instr[24:20]),        // rs2
50          .writeReg(instr[11:7]),         // rd
51          .writeData(WriteBackData),      // Data to write back
52          .write(RegWrite),               // Write enable signal
53          .readData1(readData1),          // Output for register 1 data
54          .readData2(readData2)           // Output for register 2 data
55  );
56
57  //Immediate Generation
58  always @(*) begin
59          case (instr[6:0])
60              7'b0010011: imm_out = {{20{instr[31]}}, instr[31:20]};  // I-type
61              7'b0100011: imm_out = {{20{instr[31]}}, instr[31:25], instr[11:7]};  // S-type
62              7'b1100011: imm_out = {{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}; // B-type
63          endcase
64      end
65
66  // ALU operands
67  assign alu_op1 = readData1;
68  assign alu_op2 = (ALUSrc) ? imm_out : readData2;        // If (ALUSrc): 2nd operand = imm_out
69                                                          //else: 2nd operand = ReadData2
70  // ALU instantation
71  alu datapath_alu (
72          .op1(alu_op1),          // First ALU operand (from register file)
73          .op2(alu_op2),          // Second ALU operand (from immediate or register)
74          .alu_op(ALUCtrl),       // ALU control signal
75          .result(alu_result),    // ALU result
76          .zero(Zero)             // Zero flag
77  );
78
79
80  // Branch target
81  assign branch_target = PC + {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
82
```

```
83  // combinational logic preperation for dAddress and WriteBackData
84  reg [31:0] alu_result_wb;
85  always @(posedge clk or posedge rst) begin
86          if (rst) begin
87              alu_result_wb <= 32'b0;  // Reset
88          end else begin
89              alu_result_wb <= alu_result;
90          end
91      end
92
93  //Write Back
94  assign WriteBackData = (MemToReg) ? dReadData : alu_result_wb;          // Select between ALU result or data from memory
95
96
97  //Memory Addressing
98  assign dAddress = alu_result_wb;        // ALU result gives memory address
99  assign dWriteData = readData2;  // Data to be written to memory
100
101 endmodule
```

Figure 8: Ο κώδικας του datapath

Στο module datapath.v αρχικοποιείται η λογικη του Program Counter (PC), όπου αυξάνεται κατά 4 bytes σε κύκλο, και όταν εκτελείται B-type instruction, υπολογίζει το branch offset και αλλάζει το PC ανάλογα. Επίσης υπολογίζεται η 32-bit μεταβλητή immediate, σύμφωνα με το instruction set manual του RISC-V. Τέλος, υλοποιείται ένα combinational κύκλωμα, ώστε το αποτέλεσμα της ALU να διατηρείται σε κάθε κύκλο.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Εικόνα 1: Risc-v ISA

ΑΣΚΗΣΗ 5

Ζητείται η δημιουργία ενός ελεγκτή πολλαπλών κύκλων που εκτελεί κάθε εντολή που του δίνεται από την rom σε πέντε κύκλους ρολογιού. Ο στόχος είναι η σχεδίαση και υλοποίηση ενός ελεγκτή που θα κατευθύνει το datapath στον επεξεργαστή και θα διαχειρίζεται την εκτέλεση των εντολών μέσω πέντε διαδοχικών κύκλων ρολογιού, καθορίζοντας τις απαραίτητες ενέργειες για κάθε στάδιο της διαδικασίας

Το αρχείο περιέχει τις παρακάτω θύρες :

Όνομα θύρας	**Κατεύθυνση**	**Σκοπός**
clk	Είσοδος	Ρολόι

rst	Είσοδος	Σύγχρονο Reset
instr	Είσοδος	Δεδομένα εντολών από την ROM
dReadData	Είσοδος	Ανάγνωση δεδομένων από την RAM
PC	Έξοδος	Program Counter
dAddress	Έξοδος	Διεύθυνση δεδομένων στην RAM
dWriteData	Έξοδος	Δεδομένα που εγγράφονται στη RAM
MemRead	Έξοδος	Σήμα ελέγχου που υποδεικνύει ανάγνωση μνήμης
MemWrite	Έξοδος	Σήμα ελέγχου που υποδεικνύει εγγραφή μνήμης
WriteBackData	Έξοδος	Δεδομένα που εγγράφονται στο regfile.v

Ο ελεγκτής εμπεριέχει ένα Finite State Machine (FSM), με τα ακόλουθα βήματα:

Κατάσταση	**Σκοπος**
Instruction Fetch (IF)	Παροχή του PC στη μνήμη εντολών
Instruction Decode (ID)	Αποκωδικοποίηση της ληφθείσας εντολής και έναρξη πρόσβασης στο regfile.v
Execute (EX)	Εκτέλεση της λειτουργίας στην ALU
Memory (MEM)	Εκτέλεση της λειτουργίας στην ALU (για lw/sw)
Write Back (WB)	Εγγραφή νέων δεδομένων στο regfile.v

Η υλοποίηση ακολούθησε το παρακάτω FSM Graph:

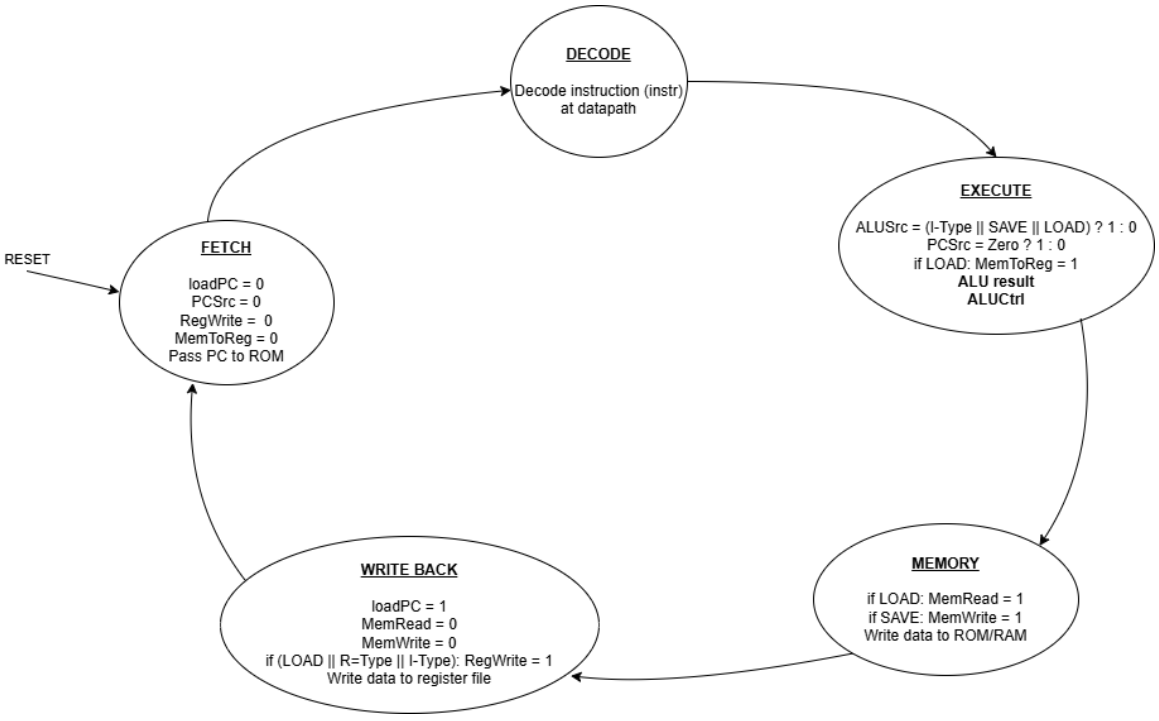


Figure 9: Σχηματικό Διάγραμμα του FSM

Να σημειωθεί πως το reset αρχικοποιεί το FSM στο state IF.

Στο top_proc.v αρχικοποιείται το datapath.v αρχικά. Έπειτα δημιουργούνται 2 always blocks, ένα που να εναλλάσσει το state σε κάθε clock cycle, και ένα που να υλοποιεί την λογική πίσω από κάθε state.

Ιδιαίτερη προσοχή δόθηκε στην αρχικοποίηση συγκεκριμένων τιμών στο 0 σε συγκεκριμένα στάδια, σύμφωνα με τις οδηγίες που δόθηκαν. Ιδιαίτερη προσοχή δόθηκε επίσης στο ALUCtrl, το οποίο μηδενίζεται σε κάθε νέο FSM cycle, ώστε οι τιμές του WriteBackData να μην παίζουν στα στάδια ID και EX, όπου το instruction εχει αλλάξει, αλλα η νέα τιμή δεν έχει ακόμη υπολογιστεί. Το ALUCtrl στο στάδιο EX, βρίσκεται αποκωδικοποιώντας το instruction. Πιο συγκεκριμένα, από τα funct3, funct7 και opcode των εντολών όπως φαίνεται στην Εικόνα 1.

Τα σήματα MemRead και MemWrite τίθενται για την ανάγνωση και την εγγραφή στην μνήμη RAM, αντίστοιχα, μόνο κατά το στάδιο MEM, αν χρειαστεί. Να σημειωθεί ότι το MemRead δεν χρησιμοποιήθηκε στα πλαίσια της άσκησης

Το σήμα RegWrite επιτρέπει την εγγραφή στους registers του regfile στο στάδιο WB.

Το σήμα MemtoReg βασίζεται στην τρέχουσα εντολή και τίθεται σε 1 μόνο κατά την εκτέλεση μιας εντολής load, υποδεικνύοντας την εγγραφή σε κάποιον register του regfile.v από την μνήμη RAM. Το σήμα δεν εξαρτάται από την τρέχουσα κατάσταση.

Το σήμα ALUSrc υποδεικνύει ποια θα είναι η είσοδος του op2 της ALU. Αν ‘ALUSrc'== 0, το readReg2 2 χρησιμοποιείται ως είσοδος στον op2, ενώ όταν 'ALUSrc'== 1, χρησιμοποιείται η imm_out (immediate data). Άμεσα δεδομένα απαιτούνται μόνο για τις εντολές 'load', 'store' και 'ALU Immediate'. Το ALUSrc προσδιορίζεται απευθείας από τα bits του opcode της εντολής.

Το σήμα loadPC τίθεται σε 1 σε κάθε στάδιο WB, ώστε να ενημερωθεί στο επόμενο στάδιο το PC με μια νέα τιμή, ενώ το σήμα PCSrc χρησιμοποιείται για να υποδείξει ποια τιμή πρέπει να φορτωθεί στον PC: είτε PC+4 για κανονικές εντολές, είτε PC+branch_offset για πράξεις που λαμβάνονται με διακλάδωση. Αυτό το σήμα τίθεται σε 1 όταν υπάρχουν οι ακόλουθες συνθήκες: (1) η τρέχουσα εντολή είναι μια λειτουργία BEQ και (2) το Zero είναι ίσο με 1.

```
module top_proc #(
    parameter INITIAL_PC = 32'h00400000
)(
        input clk,                  // Clock
        input rst,                  // Reset
        input [31:0] instr,         // Instruction fetched from instruction memory
        input [31:0] dReadData,     // Data read from data memory
        output [31:0] PC,           // Program Counter
        output [31:0] dAddress,     // Data memory address
        output [31:0] dWriteData,   // Data to write to memory
        output reg MemRead,         // Memory read enable
        output reg MemWrite,        // Memory write enable
        output [31:0] WriteBackData // Data written back to registers
);

    // FSM States
    localparam [2:0] IF  = 3'b000,
                     ID  = 3'b001,
                     EX  = 3'b010,
                     MEM = 3'b011,
                     WB  = 3'b100;

    // Current state and next state registers
    reg [2:0] state;

    // Internal signals
    reg loadPC, PCSrc, ALUSrc, RegWrite, MemToReg;
    wire Zero;
    reg [3:0] ALUCtrl;
    wire [31:0] alu_result;

    // Datapath Instantiation
    datapath dp (
        .clk(clk),
        .rst(rst),
        .instr(instr),              // Instruction fetched from instruction memory
        .PCSrc(PCSrc),              // Branch decision signal
        .ALUSrc(ALUSrc),            // Selects between register and immediate for ALU
        .RegWrite(RegWrite),        // Enables writing to the register file
        .MemToReg(MemToReg),        // Selects data memory output for write-back
        .ALUCtrl(ALUCtrl),          // ALU operation control signal
        .loadPC(loadPC),            // Updates the PC
        .PC(PC),                    // Program Counter output
        .Zero(Zero),                // ALU Zero flag
        .dAddress(dAddress),        // Address for accessing data memory
        .dWriteData(dWriteData),    // Data to be written to memory
        .dReadData(dReadData),      // Data read from memory
        .WriteBackData(WriteBackData) // Data written back to registers
    );

// FSM Logic
always @(posedge clk) begin
        if (rst) begin
            state <= IF;
        end else begin
        case (state)
                IF: begin
                        state <= ID;
                end
                ID: begin
                        state <= EX;
                end
                EX: begin
                        state <= MEM;
                end
                MEM: begin
                        state <= WB;
                end
                WB: begin
                        state <= IF;
                end
                default: state <= IF;
        endcase
    end
end
```

```
76
77      always @(*) begin
78          // Default values
79          MemRead <= 0; MemWrite <= 0;
80
81          $display("Entering state.   State: %d, instr, PC: %d : %d", state, instr, PC);
82          case (state)
83              IF: begin
84                  ALUCtrl <=-1;
85                  PCSrc <= 0; // PCSrc can be 1 only when current instruction is BEQ AND Zero == 1
86                  loadPC <= 0;
87                  RegWrite <= 0;
88                  MemToReg <= 0;
89              end
90              ID: begin
91                  // Decode implicitly happens at datapath
92              end
93              EX: begin
94                  case (instr[6:0])
95                          7'b0110011: begin // R-Type
96                                  ALUSrc <= 0;
97                                  case (instr[14:12])
98                                          3'b000: ALUCtrl <= (instr[30] ? 4'b0110 : 4'b0010); // SUB or ADD
99                                          3'b001: ALUCtrl <= 4'b1001; // SLL
100                                         3'b010: ALUCtrl <= 4'b0100; // SLT
101                                         3'b100: ALUCtrl <= 4'b0101; // XOR
102                                         3'b101: ALUCtrl <= (instr[30] ? 4'b1010 : 4'b1000); // SRA or SRL
103                                         3'b110: ALUCtrl <= 4'b0001; // OR
104                                         3'b111: ALUCtrl <= 4'b0000; // AND
105                                 endcase
106                         end
107                         7'b0010011: begin // I-Type
108                                 ALUSrc = 1; // Use immediate
109                                 case (instr[14:12])
110                                         3'b000: ALUCtrl <= 4'b0010; // ADDI
111                                         3'b010: ALUCtrl <= 4'b0100; // SLTI
112                                         3'b111: ALUCtrl <= 4'b0000; // ANDI
113                                         3'b110: ALUCtrl <= 4'b0001; // ORI
114                                         3'b100: ALUCtrl <= 4'b0101; // XORI
115                                         3'b001: ALUCtrl <= 4'b1001; // SLLI
116                                         3'b101: ALUCtrl <= (instr[30] ? 4'b1010 : 4'b1000); // SRAI or SRLI
117                                 endcase
118                                 MemToReg = 1; // Write from memory data to register
119                         end
120                         7'b0100011: begin // SW
121                                 ALUCtrl = 4'b0010; // ADD
122                         end
123                         7'b1100011: begin // BEQ
124                                 ALUCtrl = 4'b0110; // SUB
125                         end
126                 endcase
127             end
128             EX: begin
129                 case (instr[6:0])
130                         7'b0110011: begin // R-Type
131                                 ALUSrc = 0;
132                         end
133                         7'b0010011: begin // I-Type
134                                 ALUSrc = 1; // Use immediate
135                         end
136                         7'b0000011, 7'b0100011: begin // LW or SW
137                                 ALUSrc = 1; // Use immediate
138                         end
139                         7'b1100011: begin // BEQ
140                                 ALUSrc = 0;
141                                 PCSrc = Zero;   //Branch if ALU result is zero
142                         end
143                 endcase
144             end
145             MEM: begin
146                 case (instr[6:0])
147                     7'b0000011: begin // LW
148                         MemRead = 1; // Enable memory read
149                     end
150                     7'b0100011: begin // SW
151                         MemWrite = 1; // Enable memory write
152                     end
153                 endcase
154             end
155             WB: begin
156                 MemRead = 0;
157                 MemRead = 0;
158                 loadPC = 1; // Load new PC value before transitioning to IF
158                   loadPC = 1; // Load new PC value before transitioning to IF
159                   RegWrite = 1; // Enable register write
160               end
161           endcase
162       end
163
164   endmodule
165
166
```

Figure 7: Ο κώδικας του top_proc

```
module top_proc_tb;


    reg clk;
    reg rst;

    // Inputs and outputs related to 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

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

    DATA_MEMORY ram (
        .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
    ) proc (
        .clk(clk),
        .rst(rst),
        .instr(instr),
        .dReadData(dReadData),
        .PC(PC),
        .dAddress(dAddress),
        .dWriteData(dWriteData),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .WriteBackData(WriteBackData)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10ns clock period
    end

    // Test sequence
    initial begin
        // Initialize reset
        rst = 1;
        #6;
        rst = 0;

        // Wait for simulation to complete
        #2000;  // Run the simulation for a sufficient amount of time
        $finish;
    end


endmodule
```

Figure 11: ο κώδικας του top_proc_tb

Στο testbench, δημιουργείται το clock και αρχικοποιούνται οι RAM, ROM και το top_proc.v.

Η σειρά με την οποία τρέχουν οι εντολές είναι η εξής:

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

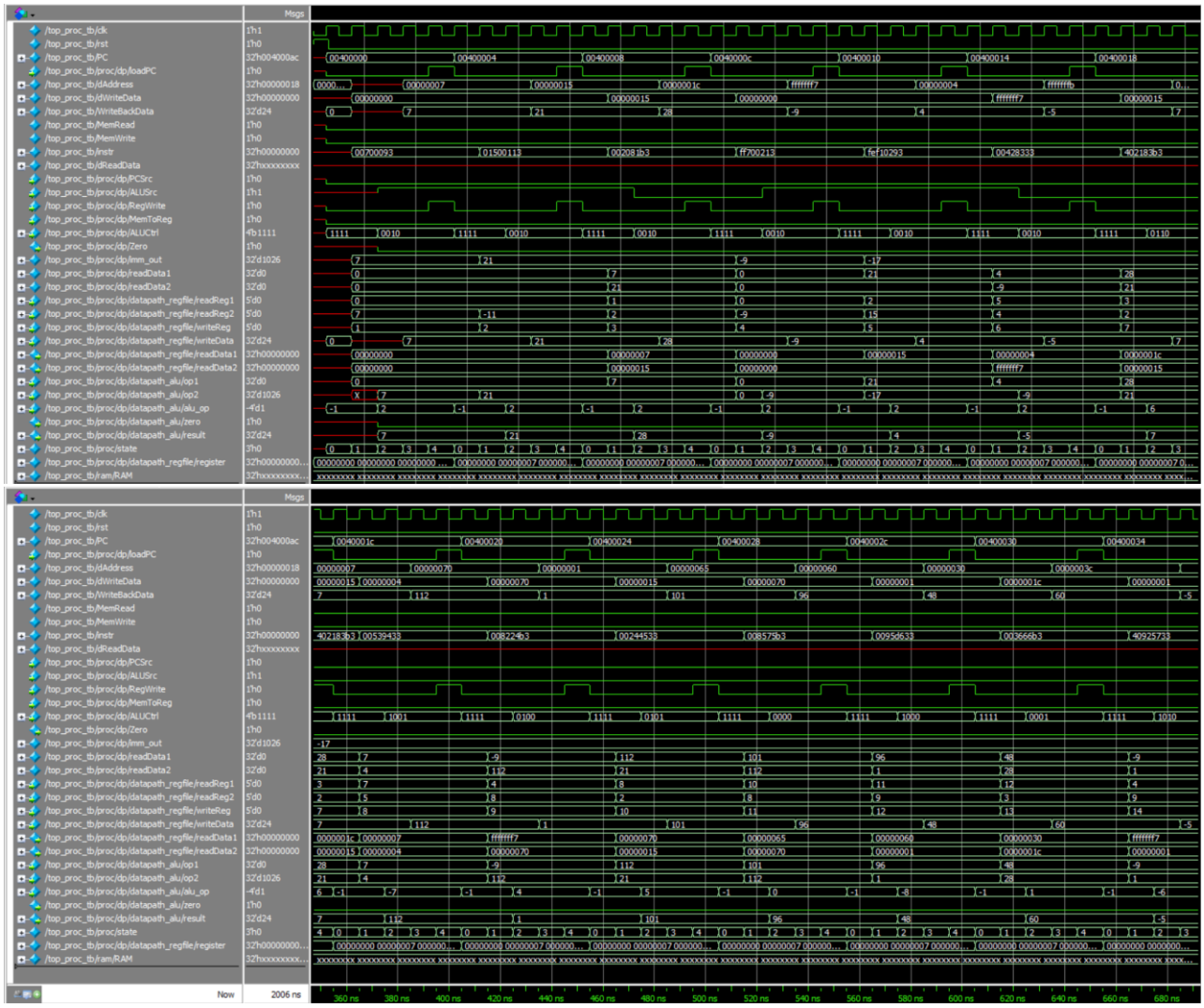
77: beq x15, x11, 16 Expected result: PC = PC + 16

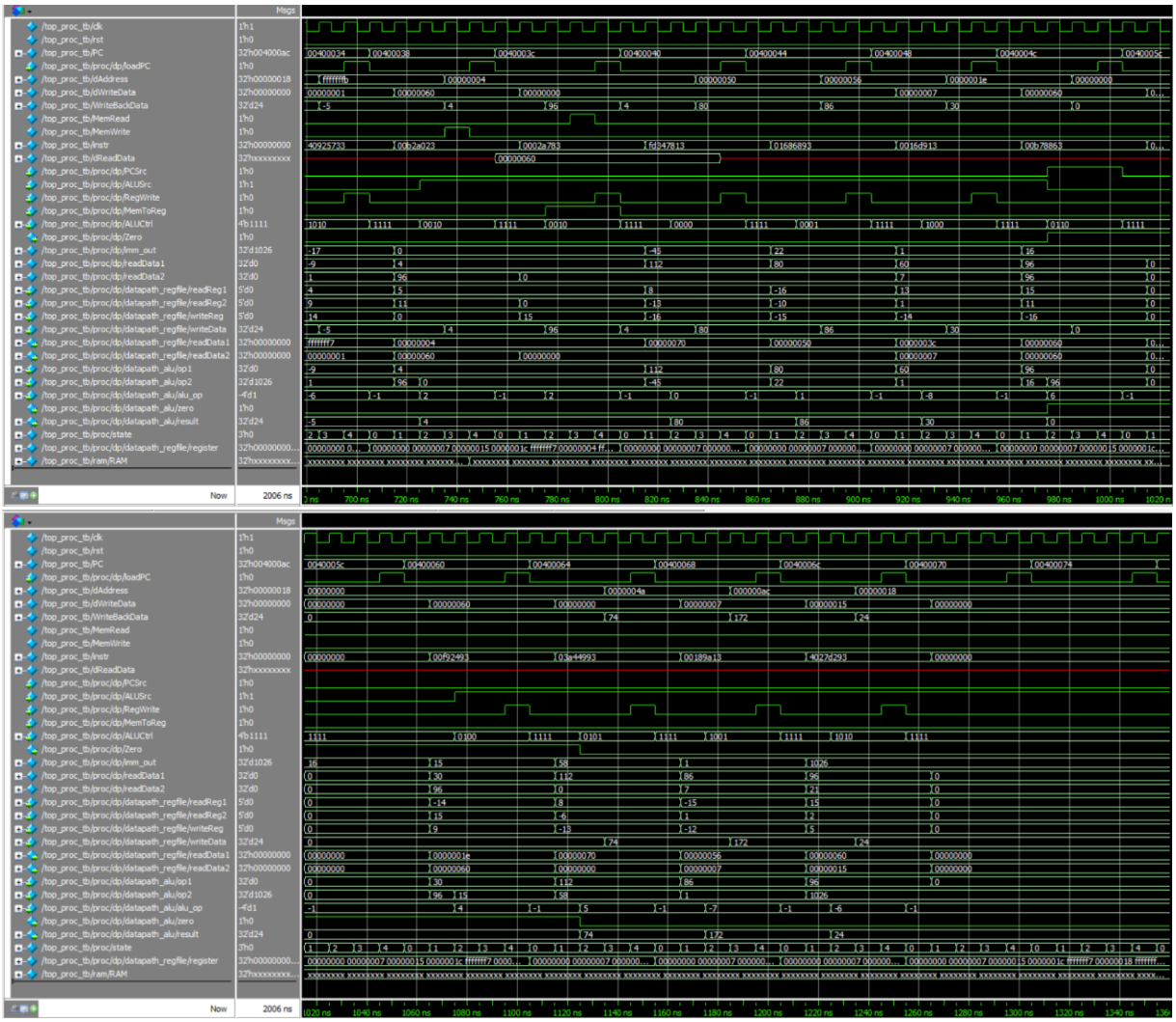
97: slti x9, x18, 15 Expected result: x9 = 0

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


Msgs
/top_proc_tb/clk
/top_proc_tb/rst
/top_proc_tb/PC
/top_proc_tb/proc/dp/loadPC
/top_proc_tb/dAddress
/top_proc_tb/dWriteData
/top_proc_tb/WriteBackData
/top_proc_tb/MemRead
/top_proc_tb/MemWrite
/top_proc_tb/instr
/top_proc_tb/dReadData
/top_proc_tb/proc/dp/PCSrc
/top_proc_tb/proc/dp/ALUSrc
/top_proc_tb/proc/dp/RegWrite
/top_proc_tb/proc/dp/MemToReg
/top_proc_tb/proc/dp/ALUCtrl
/top_proc_tb/proc/dp/Zero
/top_proc_tb/proc/dp/imm_out
/top_proc_tb/proc/dp/readData1
/top_proc_tb/proc/dp/readData2
/top_proc_tb/proc/dp/datapath_regfile/readReg1
/top_proc_tb/proc/dp/datapath_regfile/readReg2
/top_proc_tb/proc/dp/datapath_regfile/writeReg
/top_proc_tb/proc/dp/datapath_regfile/writeData
/top_proc_tb/proc/dp/datapath_regfile/readData1
/top_proc_tb/proc/dp/datapath_regfile/readData2
/top_proc_tb/proc/dp/datapath_alu/op1
/top_proc_tb/proc/dp/datapath_alu/op2
/top_proc_tb/proc/dp/datapath_alu/alu_op
/top_proc_tb/proc/dp/datapath_alu/zero
/top_proc_tb/proc/dp/datapath_alu/result
/top_proc_tb/proc/state
/top_proc_tb/proc/dp/datapath_regfile/register
/top_proc_tb/ram/RAM
Now
2006 ns


Msgs
/top_proc_tb/clk
/top_proc_tb/rst
/top_proc_tb/PC
/top_proc_tb/proc/dp/loadPC
/top_proc_tb/dAddress
/top_proc_tb/dWriteData
/top_proc_tb/WriteBackData
/top_proc_tb/MemRead
/top_proc_tb/MemWrite
/top_proc_tb/instr
/top_proc_tb/dReadData
/top_proc_tb/proc/dp/PCSrc
/top_proc_tb/proc/dp/ALUSrc
/top_proc_tb/proc/dp/RegWrite
/top_proc_tb/proc/dp/MemToReg
/top_proc_tb/proc/dp/ALUCtrl
/top_proc_tb/proc/dp/Zero
/top_proc_tb/proc/dp/imm_out
/top_proc_tb/proc/dp/readData1
/top_proc_tb/proc/dp/readData2
/top_proc_tb/proc/dp/datapath_regfile/readReg1
/top_proc_tb/proc/dp/datapath_regfile/readReg2
/top_proc_tb/proc/dp/datapath_regfile/writeReg
/top_proc_tb/proc/dp/datapath_regfile/writeData
/top_proc_tb/proc/dp/datapath_regfile/readData1
/top_proc_tb/proc/dp/datapath_regfile/readData2
/top_proc_tb/proc/dp/datapath_alu/op1
/top_proc_tb/proc/dp/datapath_alu/op2
/top_proc_tb/proc/dp/datapath_alu/alu_op
/top_proc_tb/proc/dp/datapath_alu/zero
/top_proc_tb/proc/dp/datapath_alu/result
/top_proc_tb/proc/state
/top_proc_tb/proc/dp/datapath_regfile/register
/top_proc_tb/ram/RAM
Now
2006 ns
Msgs
/top_proc_tb/clk
/top_proc_tb/rst
/top_proc_tb/PC
/top_proc_tb/proc/dp/loadPC
/top_proc_tb/dAddress
/top_proc_tb/dWriteData
/top_proc_tb/WriteBackData
/top_proc_tb/MemRead
/top_proc_tb/MemWrite
/top_proc_tb/instr
/top_proc_tb/dReadData
/top_proc_tb/proc/dp/PCSrc
/top_proc_tb/proc/dp/ALUSrc
/top_proc_tb/proc/dp/RegWrite
/top_proc_tb/proc/dp/MemToReg
/top_proc_tb/proc/dp/ALUCtrl
/top_proc_tb/proc/dp/Zero
/top_proc_tb/proc/dp/imm_out
/top_proc_tb/proc/dp/readData1
/top_proc_tb/proc/dp/readData2
/top_proc_tb/proc/dp/datapath_regfile/readReg1
/top_proc_tb/proc/dp/datapath_regfile/readReg2
/top_proc_tb/proc/dp/datapath_regfile/writeReg
/top_proc_tb/proc/dp/datapath_regfile/writeData
/top_proc_tb/proc/dp/datapath_regfile/readData1
/top_proc_tb/proc/dp/datapath_regfile/readData2
/top_proc_tb/proc/dp/datapath_alu/op1
/top_proc_tb/proc/dp/datapath_alu/op2
/top_proc_tb/proc/dp/datapath_alu/alu_op
/top_proc_tb/proc/dp/datapath_alu/zero
/top_proc_tb/proc/dp/datapath_alu/result
/top_proc_tb/proc/state
/top_proc_tb/proc/dp/datapath_regfile/register
/top_proc_tb/ram/RAM
Now
2006 ns

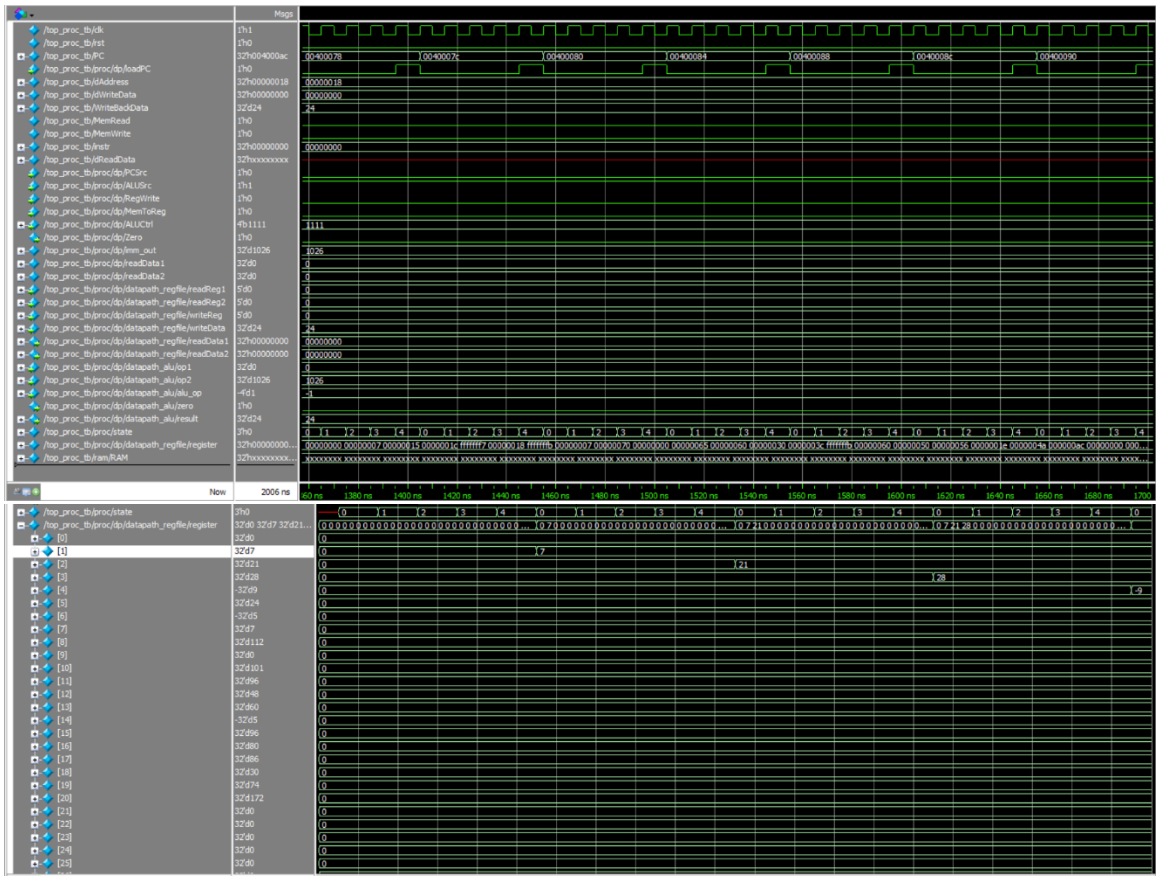


Figure 11: Κυματομορφή άσκησης 2. Η τελευταία εικόνα δείχνει ένα απόσπασμα από τον τρόπο αποθήκευσης δεδομένων στους registers