Ψηφιακά Συστήματα HW-1

ΕΡΓΑΣΤΗΡΙΑΚΕΣ ΑΣΚΗΣΕΙΣ

Εργασία της Πλευρίδη Βασιλική Βαρβάρα (ΑΕΜ:10454)

Η εργαστηριακή άσκηση αυτή έχει ως τελικό στόχο την υλοποίηση μιας αριθμομηχανής και ενός επεξεργαστή RISC-V, τα οποία χτίζονται βήμα βήμα με την βοήθεια των παρακάτω ασκήσεων. Για την υλοποίηση όλων των παραπάνω χρησιμοποιήθηκε ως εργαλείο προσομοίωσης το Icarus Verilog. Παρακάτω θα γίνει ένας μικρός σχολιασμός του κώδικα της Verilog για κάθε άσκηση ξεχωριστά αλλά και ζητούμενα διαγράμματα και κυματομορφές προσομοίωσης.

Άσκηση 1

Σκοπός αυτής της άσκησης είναι η υλοποίηση μίας αριθμητικής/ λογικής μονάδας (ALU). Το αρχείο υλοποίησης του είναι το alu.v ενώ το module ονομάζεται alu.

Σχολιασμός κώδικα

Η υλοποίηση της ALU είναι αρκετά απλή και αποτελείται από ένα always block στο οποίο, με την βοήθεια μίας εντολής case, γίνεται η αντιστοίχιση των παραμέτρων που έχουν οριστεί στην αρχή του κώδικα με τις 9 διαφορετικές λειτουργίες της ALU. Σημαντικό είναι να σχολιαστεί ότι τα inputs op1 και op2 τίθενται παντού προσημασμένα για την σωστή λειτουργία των πράξεων την πρόσθεσης, αφαίρεσης και «μικρότερο από». Η ύπαρξη της θύρας zero γίνεται καλύτερα κατανοητή στις ασκήσεις 4 και 5 και μέσα από τα σχηματικά διαγράμματα υλοποίησης του επεξεργαστή RISC-V. Ουσιαστικά χρησιμοποιείται στην περίπτωση του branch έτσι ώστε να γίνεται η κατάλληλη πρόσθεση του branch offset στο Program Counter.

Άσκηση 2

Στην άσκηση αυτή, πραγματοποιείται η υλοποίηση της αριθμομηχανής αλλά και η επαλήθευση της σωστής λειτουργίας της με ένα testbench. Τα αρχεία της μοντελοποίησής της είναι δύο, το αρχείο calc.v το οποίο επικεντρώνεται στον σχεδιασμό ενός accumulator και το αρχείο calc_env.v το οποίο παράγει το σήμα alu_op που καθορίζει την λειτουργία της ALU. Πέρα από αυτά, υπάρχει και το αρχείο calc_tb.v το οποίο όπως προαναφέρθηκε ελέγχει την ορθή λειτουργία της αριθμομηχανής και εμμέσως και της ALU.

Σχολιασμός κώδικα

Σχετικά με το αρχείο **calc_enc.v**:

Η υλοποίηση αυτού του αρχείου γίνεται σε Structural Verilog. Έτσι, προσέχοντας πάντα την σειρά της υλοποίησης των πράξεων και με την βοήθεια των σχημάτων 2-5 από την εκφώνηση της άσκησης, παράγεται κάθε bit του alu_op ξεχωριστά. Για την πραγματοποίηση του παραπάνω, χρησιμοποιήθηκαν γνωστά πρότυπα κυκλώματα.

Σχετικά με το αρχείο **calc.v**:

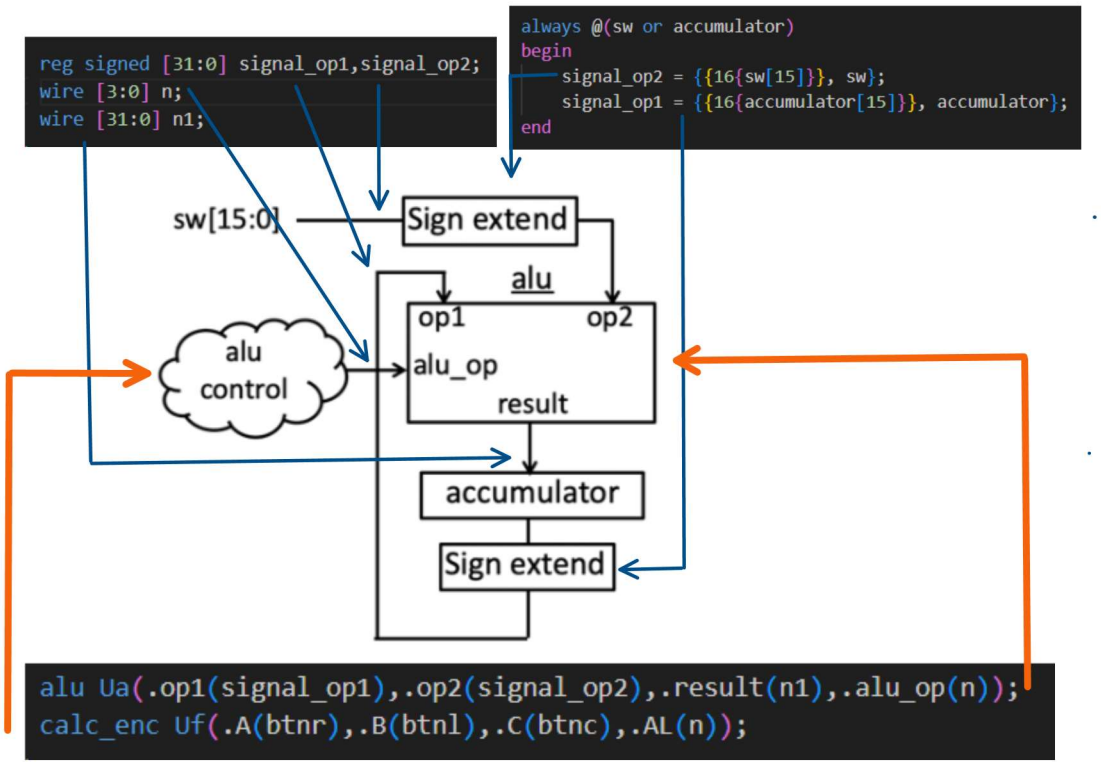
Σε αυτό το αρχείο υλοποιείται ο συσσωρευτής, όπως αυτός περιεγράφηκε και ζητήθηκε στην εκφώνηση της άσκησης. Επίσης, γίνεται ρητή αντιστοίχιση θυρών εισόδου και εξόδου των modules alu και calc_enc. Για την σύνδεση αυτή, απαιτείται και η εντολή include,

έτσι ώστε να υπάρχει πρόσβαση στα άλλα υλοποιημένα αρχεία, όπως παρουσιάζεται παρακάτω:

```
`include  "alu.v"
`include "calc_enc.v"
```

Στην συνέχεια, δημιουργείται ένα always block για την επέκταση προσήμου των σημάτων εισόδου της ALU. Συγκεκριμένα, τα σήματα sw (σήμα εισαγωγής δεδομένων) και η έξοδος του accumulator είναι 16 bit και άρα επεκτείνονται κατά 16 bit ακόμα χρησιμοποιώντας τον τελεστή concatenation της Verilog, ώστε να αποτελέσουν σήμα εισόδου 32 bit στην ALU όπως πρέπει. Τέλος, για την σύνδεση του accumulator με τις εξόδους LED υλοποιείται ένα ακόμα always block με τις περιπτώσεις απλής ενημέρωσης (πάτημα btnd) και μηδενισμού (πάτημα btnu). Τα πλήκτρα btnd και btnu, όπως είναι λογικό, μαζί με το clock (clk) εμπεριέχονται στην λίστα ευαισθησίας του συγκεκριμένου always block.

Παρακάτω παρουσιάζεται το διάγραμμα ροής της αριθμομηχανής με επεξηγηματικές αντιστοιχήσεις με τον κώδικα στην Verilog:



Σχετικά με το αρχείο **calc_tb**:

Με την βοήθεια αυτού του αρχείου γίνεται ο έλεγχος λειτουργίας των παραπάνω. Για την δημιουργία ενός testbench λοιπόν, προφανώς απαιτείται η εντολή include για το κατάλληλο αρχείο και σύνδεση των θυρών ανάλογα αλλά και ο ορισμός ενός timescale προσαρμοσμένο στην άσκηση. Στην συνέχεια δημιουργείται το ρολόι, αρχικοποιώντας το στην τιμή 0 μέσα σε ένα initial block και ακολούθως, μέσα σε ένα always block, αλλάζοντας συνεχώς την τιμή του στο αντίθετο της τρέχουσας ανά, όσες έχουν οριστεί (στην περίπτωσή μας 10), μονάδες χρόνου. Πέρα από το σήμα clk (ρολόι), ακριβώς την ίδια συμπεριφορά ακολουθεί και το πλήκτρο btnd, το οποίο θέλουμε να είναι συγχρονισμένο με το ρολόι έτσι ώστε να μην υπάρχει καθυστέρηση στην ανανέωση των τιμών του accumulator. Το initial

block που ακολουθεί αποτελεί το κύριο κομμάτι του testbench καθώς εδώ ορίζονται τα διάφορα διαδοχικά inputs τα οποία θα χρησιμοποιηθούν για τον έλεγχο της αριθμομηχανής.

Πέρα όμως από αυτό, αξίζουν να σχολιαστούν οι εξής εντολές:

```
$dumpfile("calc_tb.vcd");
```

Η εντολή αυτή ορίζει το όνομα του αρχείου vcd (Value Change Dump) στο οποίο θα καταγράφονται οι αλλαγές των σημάτων κατά τη διάρκεια της προσομοίωσης.

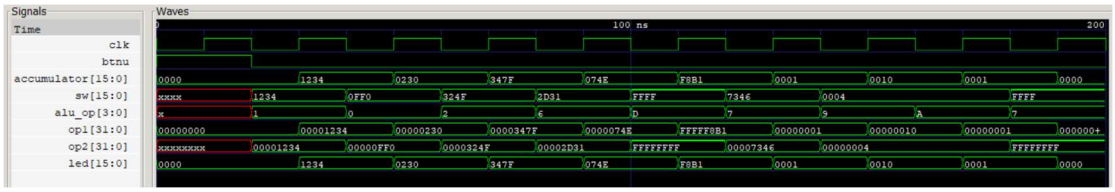
```
$dumpvars(0,calc_tb);
```

Το 0, στην εντολή αυτή, καθορίζει την μορφή καταγραφής των δεδομένων σε ASCII ενώ το calc_tb καθορίζει το πεδίο των μεταβλητών που θα καταγραφούν και συνήθως αποτελεί το όνομα του ίδιου του testbench, όπως και στην περίπτωσή μας.

```
 $finish;
```

Η εντολή αυτή τερματίζει την προσομοίωση.

Κυματομορφές προσομοίωσης



Άσκηση 3

Σε αυτή την άσκηση σχεδιάζεται και υλοποιείται ένα αρχείο καταχωρητών το οποίο θα χρησιμοποιηθεί στην συνέχεια για την αποθήκευση των τιμών των καταχωρητών που χρησιμοπούνται από την επεξεργαστή RISC-V.

Σχολιασμός κώδικα

Η εκφώνηση αυτής της άσκησης έχει σαφείς οδηγίες για την υλοποίηση του κώδικα οπότε δεν χρειάζεται κάποια παραπάνω επεξήγηση. Αυτό που παρουσιάζει ενδιαφέρον και είναι άξιο σχολιασμού είναι ότι η υλοποίηση της ανάγνωσης των τιμών των καταχωρητών από κάθε έξοδο αλλά και της εγγραφής των δεδομένων στην αντίστοιχη διεύθυνση εισόδου, απαιτεί την χρήση non-Blocking εντολών. Ο τελεστής <= μας εξασφαλίζει ότι όλες οι νέες τιμές θα έχουν ανανεωθεί πριν από την χρήση τους και έτσι ότι στην περίπτωση ίδιας διεύθυνσης ανάγνωσης και εγγραφής θα αποφευχθεί η καταγραφή εσφαλμένων τιμών.

Άσκηση 4

Η άσκηση αυτή έχει ως στόχο τον σχεδιασμό της διαδρομής δεδομένων ενός επεξεργαστή RISC-V που μαζί με την μονάδα ελέγχου που θα σχεδιαστεί στην άσκηση 5 και την μνήμη εντολών και την μνήμη δεδομένων που δίνονται έτοιμα, θα οδηγήσουν στην ολοκλήρωση της λειτουργίας αυτού.

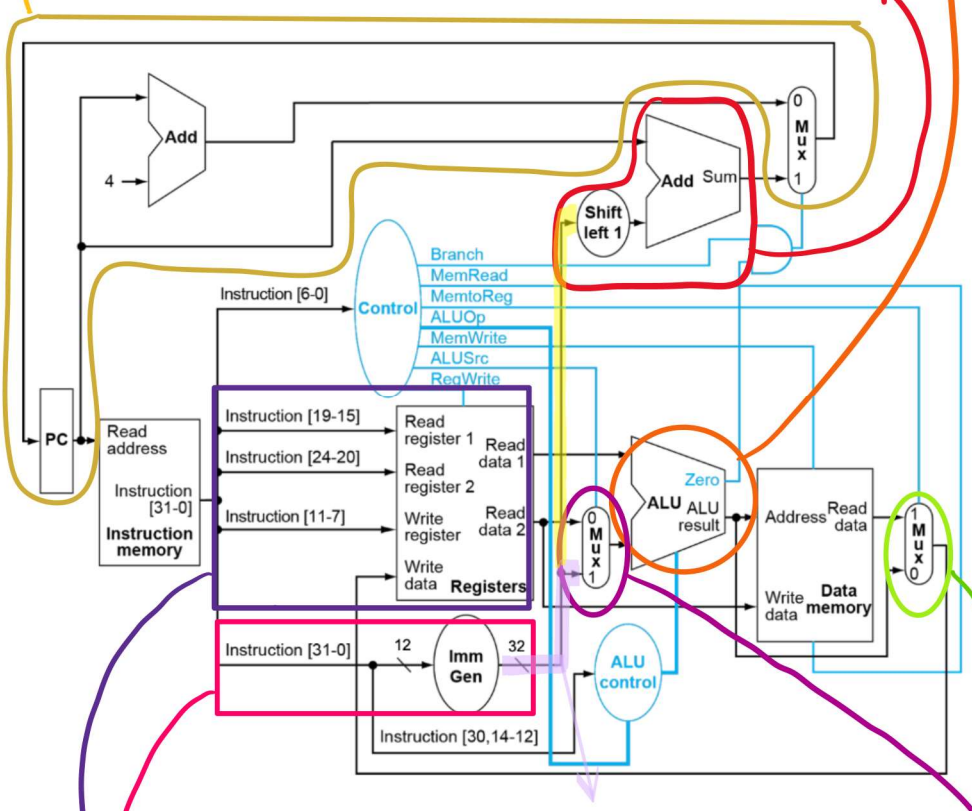
Σχολιασμός κώδικα

Παρακάτω συμπεριλήφθη το διάγραμμα υλοποίησης του datapath εμπλουτισμένο με αναφορές στον κώδικα σε Verilog:

```
//ALU
wire [31:0] alu_result;
alu u2(.op1(op1_readdata1),.op2(mux_result_op2),.alu_op(ALUCtrl),.result(alu_result),.zero(Zero));
```

```
//PC
always @(posedge clk)
begin
    if (rst)
        PC<=INITIAL_PC;
    else if (loadPC)begin
        if (PCSrc)
            PC<=sum_pc_immgen;
        else
            PC<=PC+4;
    end
end
```

```
//Branch Target
wire [31:0] left_input_add,sum_pc_immgen;
assign left_input_add=signed_output_imm_gen_beq<<1;
assign sum_pc_immgen=left_input_add+PC;
```




```
//MUX for choosing command type (S or I)
wire [31:0] signed_output_imm_gen;
assign signed_output_imm_gen=(instr[6:0]==7'b0100011)?signed_output_imm_gen_sw:signed_output_imm_gen_addi;
```

```
//MUX
wire [31:0] mux_result_op2;
assign mux_result_op2= ALUSrc ? signed_output_imm_gen :op2_readdata2;
```

```
//Write Back -MUX
assign write_back_mux= MemToReg ? dReadData : alu_result;
```

```
//Register File
wire [31:0] write_back_mux;
wire [31:0] op1_readdata1,op2_readdata2;
regfile u1(.readReg1(instr[19:15]),.readReg2(instr[24:20]),.writeReg(instr[11:7]),.write(RegWrite),
.readData1(op1_readdata1),.readData2(op2_readdata2),.writeData(write_back_mux),.clk(clk));
```

```
//Immediate Generation for immediate(addi)
wire [11:0] input_imm_gen_addi;
assign input_imm_gen_addi=instr[31:20];
wire [31:0] signed_output_imm_gen_addi;
assign signed_output_imm_gen_addi={{20{input_imm_gen_addi[11]}},input_imm_gen_addi};

//Immediate Generation for store(sw)
wire [11:0] input_imm_gen_sw;
assign input_imm_gen_sw={instr[31:25],instr[11:7]};
wire [31:0] signed_output_imm_gen_sw;
assign signed_output_imm_gen_sw={{20{input_imm_gen_sw[11]}},input_imm_gen_sw};

//Immediate Generation for branch(beq)
wire [11:0] input_imm_gen_beq;
assign input_imm_gen_beq={instr[31],instr[7],instr[30:25],instr[11:8]};
wire [31:0] signed_output_imm_gen_beq;
assign signed_output_imm_gen_beq={{19{input_imm_gen_beq[11]}},input_imm_gen_beq,1'b0};
```

Το παραπάνω διάγραμμα είναι αρκετά επεξηγηματικό όσον αφορά την υλοποίηση του κώδικα σε Verilog αλλά αξίζει να τονιστούν περαιτέρω τα εξής σημεία:

Χρήση εντολής **assign**:

Για τον ορισμό διάφορων σημάτων και την δημιουργία των ζητούμενων πολυπλεκτών προτιμάται η χρήση της εντολής assign, αντί του always block. Η επιλογή αυτή έγινε γιατί θέλουμε σταθερό ορισμό των σημάτων οπότε η λίστα ευαισθησίας της εντολής always δεν κρίνεται απαραίτητη. Άλλωστε κατά την προσομοίωση των κυματομορφών στην άσκηση 5, παρουσιάστηκαν θέματα χρονισμού και καθυστέρησης με την χρήση του always block και για αυτό αντικαταστάθηκαν όπου αυτό ήταν δυνατό με την εντολή assign.

Immediate Generation-Περίπτωση branch:

Ιδιαίτερη προσοχή χρειάστηκε στην πηγή δεδομένων που εισέρχεται στην δεύτερη θύρα ανάγνωσης της ALU. Αν παρατηρηθεί καλά το σχεδιάγραμμα, σε συνδυασμό με την εκφώνηση της άσκησης, καταλαβαίνει κανείς ότι στην περίπτωση εντολής τύπου B (BEQ), δεν μας απασχολεί τι θα πάει στην είσοδο του MUX καθώς σε αυτή την περίπτωση το σήμα ελέγχου ALUSrc θα είναι ίσο με 0 και έτσι η έξοδος του MUX θα προέρχεται από το αρχείο καταχωρητών.

Immediate Generation-Περίπτωση εντολής τύπου S ή I:

Στις περιπτώσεις εντολών τύπου S και I, όπως φαίνεται και στο διάγραμμα, απαιτείται η δημιουργία ενός απλού MUX το οποίο ανάλογα τον τύπο της εντολής που θα έρχεται στον επεξεργαστή, θα βγάζει ως έξοδο την ανάλογη άμεση τιμή εντολών (S ή I), η οποία θα αποτελέσει την δεύτερη θύρα εισόδου της ALU.

Οι άμεσες τιμές εντολών R,S,I και B πάρθηκαν από το αρχείο «The RISC-V Instruction Set Manual» και εντάσσονται και παρακάτω:

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

Άσκηση 5

Στην άσκηση αυτή, όπως προαναφέρθηκε, υλοποιείται το τελευταίο στοιχείο του επεξεργαστή RISC-V το οποίο είναι η μονάδα ελέγχου, η σύνδεση όλων μεταξύ τους και εν τέλει ο έλεγχος σωστής λειτουργίας αυτού.

Σχολιασμός κώδικα

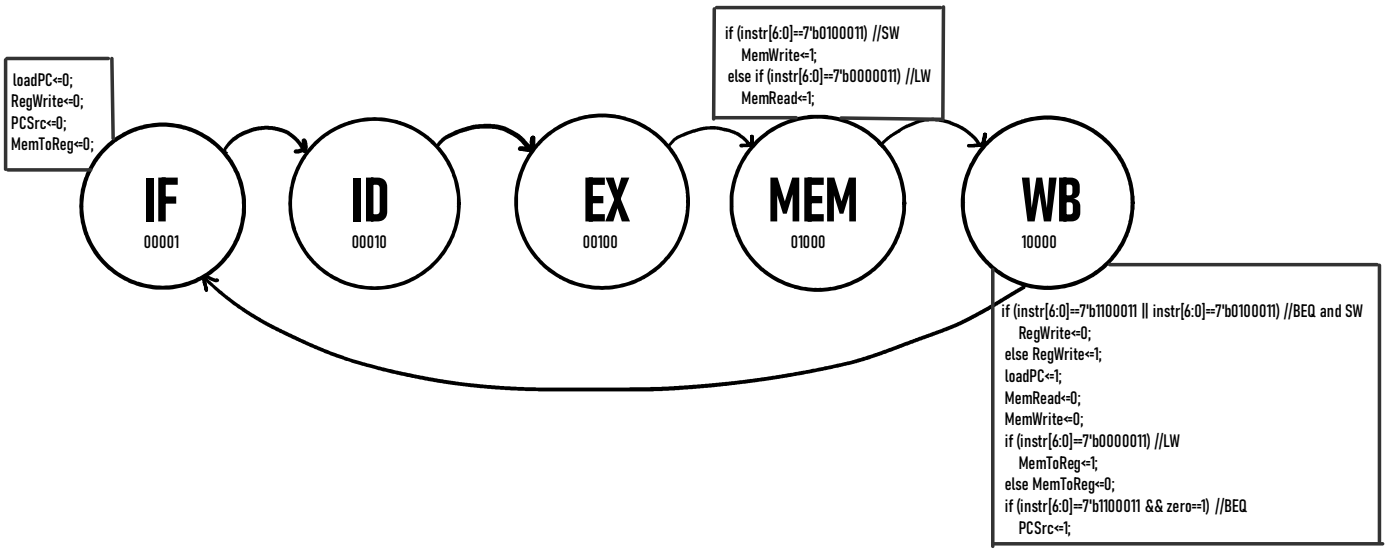
Σχετικά με το αρχείο **top_proc.v**:

Στο αρχείο αυτό, το κύριο στοιχείο που υλοποιείται είναι το FSM (Finite State Machine). Για την υλοποίηση του απαιτείται η δημιουργία 3 Procedural Blocks: **Αποθήκευσης Κατάστασης** (ακολουθιακή λογική => στην λίστα ευαισθησίας έχουμε το clk), **Επόμενης Κατάστασης** και **για τις Εξόδους** (συνδυαστική λογική και στα δύο => δεν εμπλέκεται η μνήμη και άρα δεν χρειάζεται το clk).

Πριν την παρουσίαση του διαγράμματος καταστάσεων σημαντικό είναι να σχολιαστεί ότι οι 5 καταστάσεις IF, ID, EX, MEM και WB, κωδικοποιήθηκαν έτσι ώστε να ακολουθούν την κωδικοποίηση “One-Hot Encoding” για διευκόλυνση εύρεσης σφάλματος και ευκολία στον έλεγχο.

```
parameter [4:0]  IF=5'b00001, ID=5'b00010, EX=5'b00100, MEM=5'b01000,
WB=5'b10000;
```

Σχηματικό Διάγραμμα καταστάσεων του FSM



Πέρα από το FSM, στο αρχείο αυτό υλοποιείται το σήμα ALUCtrl μέσα σε ένα always block και το σήμα ALUSrc με ένα assign, σύμφωνα με τις οδηγίες της εκφώνησης και σε συνδυασμό με τα δεδομένα από το αρχείο «The RISC-V Instruction Set Manual» που αναφέρθηκε και πριν.

Σχετικά με το αρχείο **top_proc_tb**:

Το αρχείο αποτελεί το testbench για την επαλήθευση ορθής λειτουργίας ολόκληρου του επεξεργαστή. Γίνονται οι κατάλληλες συνδέσεις με τα αρχεία rom.v και ram.v τα οποία αποτελούν την μνήμη εντολών και την μνήμη δεδομένων αντίστοιχα, και δημιουργείται το ρολόι έτσι ώστε να γίνουν οι ζητούμενες προσομοιώσεις. Σχετικά με το αρχείο rom_bytes.data εμπεριέχει 32bites εντολές που καλεί να εκτελέσει ο επεξεργαστής. Για να επαληθευτεί η σωστή ή όχι λειτουργία του επεξεργαστή, τα δεδομένα μετατράπηκαν σε κώδικα Assembly, ο οποίος παρουσιάζεται παρακάτω. Στην πρώτη στήλη είναι οι δοσμένες εντολές και δίπλα η αντίστοιχη εντολή σε assembly μαζί με κάποιο επεξηγηματικό σχόλιο αυτής. Τα μηδενικά που υπάρχουν στην μέση και στο τέλος του κώδικα έχουν παραλειφθεί καθώς δεν οδηγούν σε κάποιο αποτέλεσμα.

Binary Code	Assembly Code	Comments
00000000010000000000000010010011	addi $1, $0, 100	# $1 = $0 + 100(4) = 0 + 4 = 4
00000000000010000000000100010011	addi $2, $0, 1	# $2 = $0 + 1 = 0 + 1 = 1
00000000001100000000000110010011	addi $3, $0, 3	# $3 = $0 + 3 = 3
00000000011100000000001000010011	addi $4, $0, 7	# $4 = $0 + 7 = 7
11111111111000000000001010010011	addi $5, $0, -2	# $5 = $0 - 2 = -2
00000000001100011000001100110011	add $6, $3, $3	# $6 = $3 + $3 = 6
01000000010100110000001110110011	sub $7, $6, $5	# $7 = $6 - $5 = 6 - (-2) = 8
00000000001000011001010000110011	sll $8, $3, $2	# $8 = $3 << $2 = 6
00000000010001000010010010110011	slt $9, $8, $4	# $9 = ($8 < $4) = (6 < 7) = 1
00000000011100001100010100110011	xor $10, $1, $7	# $10 = $1 ^ $7 = 4 ^ 8 = 12
00000000100101010101010110110011	srl $11, $10, $9	# $11 = $10 >> $9 = 12 >> 1 = 6
00000000011000011111011000110011	and $12, $3, $6	# $12 = $3 & $6 = 3 & 6 = 2
00000000110001010110011010110011	or $13, $10, $12	# $13 = $10 | $12 = 12 | 2 = 14
01000000001000101101011100110011	sra $14, $5, $2	# $14 = $5 >> $2 = -2 >> 1 = -1
00000000101000010010000000100011	sw $13, 0($2)	# Memory[$2] = $13 => Memory[1] = 14
00000000000000001001001111000011	lw $15, 0($1)	# $15 = Memory[$1] => $15 = Memory[4] = 12
00000000011101101111100000010011	andi $16, $13, 7	# $16 = $13 & 7 = 6 & 7 = 6
00000000010001101110100010010011	ori $17, $13, 4	# $17 = $13 | 4 = 14 | 4 = 14
00000000001100100101100100010011	srli $18, $4, 3	# $18 = $4 >> 3 = 0
00000000100000110000001101100011	beq $8, $6, offset:3	# Branch if $8 = $6 to offset 3 (if $8 == $6)
00000000110001011100100110010011	xori $19, $11, 12	# $19 = $11 ^ 12 = 6 ^ 12 = 10
00000000000110011001101000010011	slli $20, $19, 1	# $20 = $19 << 1 = 10 << 1 = 20
01000000001000101101101010010011	srai $21, $5, $2	# $21 = $5 >> 2 = -2 >> 2 ^ (11 + 2) = -1
00000001110010100010101100010011	slti $22, $20, 28	# $22 = ($20 < 28) = (20 < 28) = 1

Κυματομορφές προσομοίωσης

Επειδή οι κυματομορφές αυτές έχουν μεγάλο μήκος και δεν μπορούν να ενσωματωθούν εύκολα στο παρόν αρχείο, παρουσιάζονται επιλεκτικά οι πρώτες 6 εντολές. Για παρατήρηση ολόκληρων των κυματομορφών και για ποικίλα σήματα, μπορεί να χρησιμοποιηθεί το αρχείο top_proc_tb.vcd που θα δημιουργείται αυτόματα με την εκτέλεση του αρχείου top_proc_tb.v.