Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής Ι

7ο εξάμηνο

Μαμουγιώργη Μαρία 10533

21 Ιανουαρίου 2024

Περιεχόμενα:

1. alu.v……3
2. calc.v……4
   decoder.v……5
   calc_tb.v……6
3. regfile.v……7
4. datapath.v……8
5. mylticycle.v……10
   mylticycle_tb.v……12

Υλοποίηση Εργασίας με χρήση Icarus Simulator μέσω Visual Studio Code

ΑΣΚΗΣΗ1: ALU

- Το κύκλωμα είναι συνδυαστικό, καθώς η κατάσταση εξόδου εξαρτάται μόνο από τις εισόδους του (op1, op2, alu_op) και δεν υπάρχει εσωτερική κατάσταση που να αποθηκεύεται για μελλοντικές καταστάσεις.
- Για την υλοποίηση του κυκλώματος alu χρησιμοποίησα ένα always block και μια case. Έτσι, το αποτέλεσμα result καθορίζεται σύμφωνα με την πράξη που δηλώνει η alu_op πως θα πραγματοποιηθεί. Οι έξοδοι μου, result και zero προσδιορίζονται μέσα σε always block επομένως είναι τύπου reg.
- Έθεσα το sensitivity list του always block (*), ώστε οποιαδήποτε αλλαγή των εισόδων μου (op1, op2, alu_op) να μεταβάλλει αντίστοιχα και τις εξόδους μου.
- Έθεσα τις τιμές όλων των 4bit παραμέτρων σύμφωνα με τον πίνακα που δινόταν στην εκφώνηση.
- Η σύγκριση των εισόδων op1 και op2 πραγματοποιείται σε προσημασμένους αριθμούς, για αυτό χρησιμοποίησα την εντολή $signed . Το λογικό αποτέλεσμα της σύγκρισης είναι 1 σε περίπτωση που ισχύει και 0 σε περίπτωση που δεν ισχύει.

```
ALUOP_SLT: result = ($signed(op1)<$signed(op2)) ? 32'b1:32'b0;
```

- Η αριθμητική ολίσθηση απαιτεί την μετατροπή της τιμής που ολισθαίνει σε προσημασμένο αριθμό και έπειτα την μετατροπή του αποτελέσματος πίσω σε μη προσημαμένο.

```
ALUOP_SRA: result = $unsigned($signed(op1) >>> op2[4:0]);
//η τιμή που ολισθαίνει --> μετατροπή σε προσημασμένο αριθμό
//το αποτέλεσμα --> μετατροπή σε μη προσημασμένο αριθμό
```

- Προσωπικά έθεσα το default state της case(alu_op) σε αποτέλεσμα ίσο με 0.

```
default: result = 32'b0;
```

- Για την τιμή της εξόδου zero πραγματοποίησα έλεγχο, αν η τιμή result ισούται με 0, η zero έξοδος να παίρνει την τιμή 1.

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

ΑΣΚΗΣΗ2: Δημιουργία κυκλώματος αριθμομηχανής (calc.v)

- Στο κύκλωμα calc χρησιμοποείται η alu και η decoder. Επειδή το calc module είναι top module, αρχικοποιείται το module alu και το decoder με ρητή αντιστοίχιση θυρών (για ευελιξία) μέσα στην calc.
  Γίνεται `include “alu.v” και `include “decoder.v”, μέσα στο αρχείο αλλά έξω από το module (για το compile).
- Με την δήλωση reg [15:0] accumulator στα internal nets και την αρχικοποίηση του μέσα σε always block με sensitivity list (posedge clk) δηλώνεται ακολουθιακή λογική (δηλώνεται ένας καταχωρητής).
- Ο accumulator αποτελεί έναν καταχωρητή που παίρνει τις τιμές του σύγχρονα, με σύγχρονο σήμα reset(btnu) και αρχικοποιείται μέσω ενός always (posedge clk). Οι εντολές αρχικοποίησης του accumulator είναι non-blocking διότι η λογική του καταχωρητή είναι ακολουθιακή, αποθηκεύονται δηλαδή οι προηγούμενες καταστάσεις του.
- Το btnd πλήκτρο είναι τύπου enable πλήκτρο, το θεωρώ σύχρονο, και κάθε φορά που είναι 1, επιτρέπει να γίνει εγγραφή στον καταχωρητή accumulator. Επίσης συνδέω την τιμή του accumulator με τις εξόδους LED μέσω μιας assign.
- [<MSB_index>:<LSB_index>], επομένως καταλαβαίνω ότι τα χαμηλότερα bit ενός 32bit σήματος είναι:

```
[15:0] result;
//τα 16 χαμηλότερα bit της εξόδου "result" 32 bit της ALU.
```

- Η επέκταση προσήμου των σημάτων, για να χρησιμοποιηθούν ως είσοδοι στην alu, πραγματοποιήθηκε με concatenation, επαναλαμβάνοντας το MSB του σήματος.

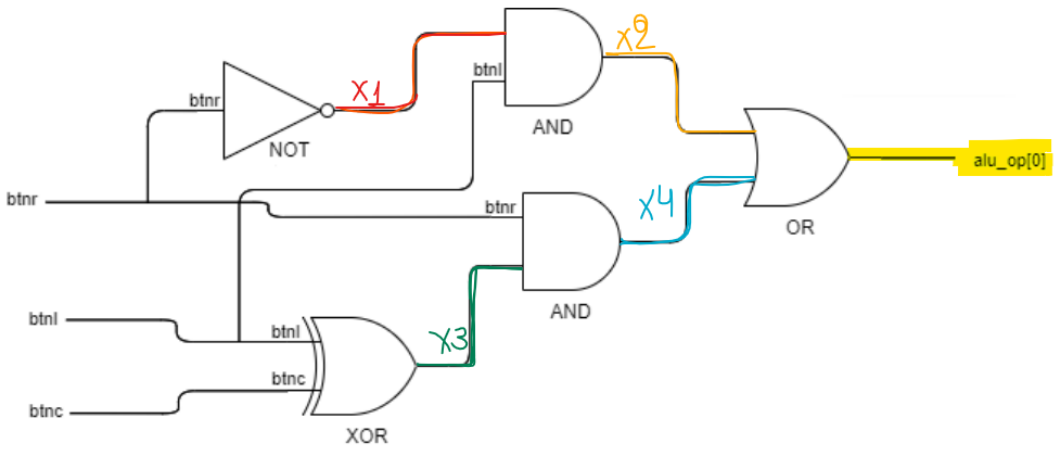
```
//Επέκταση προσήμων
assign alu_op1 = { {16{accumulator[15]}} ,accumulator};
//'op1' της ALU. --> σήμα 32-bit (επέκταση προσήμου) του accumulator 16-
bit.
assign alu_op2 = { {16{sw[15]}} ,sw};
//'op2' της ALU --> σήμα 32-bit (επέκταση προσήμου) των εισόδων του διακόπτη
16-bit.
```

Decoder.v module

- Μέσω του εν λόγω module αρχικοποιούνται τα bits της alu_op.
- Χρήση Structural Verilog, Συνδυαστική Λογική
- Ένα παράδειγμα κατασκευής Structural Verilog είναι οι δομικοί τελεστές (gate-level operators) όπως AND, OR, XOR, NOT, και άλλοι. Η σύνταξη χρησιμοποιείται για να περιγραφούν λογικοί τελεστές στο επίπεδο των πυλών.
  π.χ. and(έξοδος, είσοδος1, είσοδος2) ή not(έξοδος, είσοδος)
- Δηλώνω internal nets για να προσδιορίσω την σύνδεση μεταξύ των gates και να καταλήξω στην έξοδο του εκάστοτε bit.
- Για παράδειγμα η alu_op[0]: (ομοίως και τα υπόλοιπα bits της alu)

```
    //internal nets
    wire x1,x2,x3,x4;

//gate: (output, inputs) --> STRUCTURAL VERILOG
    //alu_op[0]:
    not(x1, btnr);
    and(x2, x1, btnl);
    xor(x3, btnl, btnc);
    and(x4, btnr, x3);
    or (alu_op[0], x2, x4);
```



Σχ. 2: Παραγωγή του alu_op[0] μέσω των btnr, btnl, btnc.

Testbench (calc_tb.v)

- Περιλαμβάνω το αρχείο calc.v, το οποίο θα ελέγξω με το testbench.

```
`include "calc.v"
```

- Στο «test bench» δηλώνεται το προς επαλήθευση σύστημα σαν ένα module στο χαμηλότερο επίπεδο ιεραρχίας. (με ρητή αντιστοίχιση θυρών).

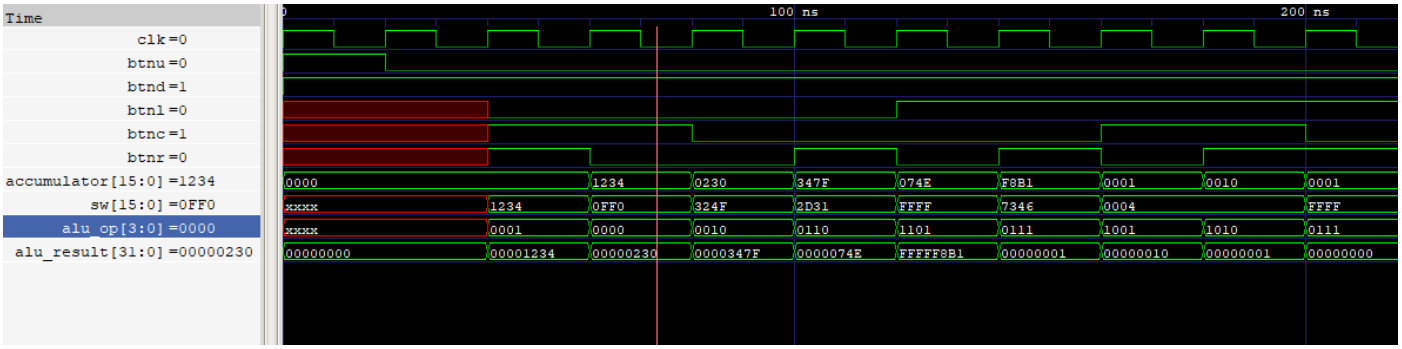
```
 calc TB (.led(led), .clk(clk), .btnc(btnc), .btnl(btnl), .btnu(btnu), .btnr(btnr), .btnd(btnd), .sw(sw));
```

- Δημιουργώ με ένα initial και ένα always block τον παλμό του ρολογιού.

```
    initial
    begin
        clk = 1'b1;
    end
    always
    begin
        #10 clk = ~clk; //παλμός ρολογιού
    end
```

- Μέσα σε ένα initial block γράφω τις καταστάσεις που δίνονται από την εκφώνηση ώστε να ελέγξω τις τιμές εξόδου αν είναι οι αναμενόμενες.

Κυματομορφές προσομοίωσης



btnl, btnc, btnr (input)	Previous value (acc.)	Switches (input)	Function in ALU	Expected Result
(btnu for reset)	xxxx	xxxx	Reset	0x0
0,1,1	0x0	0x1234	OR	0x1234
0,1,0	0x1234	0x0ff0	AND	0x0230
0,0,0	0x0230	0x324f	ADD	0X347f
0,0,1	0x347f	0x2d31	SUB	0x074e
1,0,0	0x074e	0xffff	XOR	0xf8b1

ΑΣΚΗΣΗ3: Δημιουργία αρχείου καταχωρητών (regfile.v)

- Με την δήλωση reg μεταβλητής εξόδου και την αρχικοποίηση της μέσα σε procedural block, όπως always block, δηλώνεται ακολουθιακή λογική (δηλώνεται ένας καταχωρητής). Χρησιμοποιώ non-blocking εντολές γιατί έχω ακολουθιακή λογική.
- Για να υλοποιήσω ένα αρχείο καταχωρητών 32x32, δήλωσα έναν πίνακα καταχωρητών με 32 μεταβλητές τύπου reg των 32bit.

```
reg [31:0] registers [0:31];
```

- Έχω ένα always block με sensitivity list (posedge clk) που ενημερώνεται σε κάθε ανερχόμενη τιμή του clk. Η είσοδος write λειτουργεί σαν enable σήμα για την λειτουργία του regfile.
- Στην περίπτωση που η διεύθυνση εγγραφής είναι ίδια με κάποια από τις διευθύνσεις ανάγνωσης, τα δεδομένα ανάγνωσης από τη θύρα 1 ή 2 (ανάλογα το ποια διεύθυνση ανάγνωσης ταυτίζεται με την διεύθυνση εγγραφής) παίρνουν την τιμή του writeData(δεδομένα προς εγγραφή)

```
if (writeReg == readReg1) //readReg1 --> Διεύθυνση για τη θύρα ανάγνωσης 1
     readData1 <= writeData;
     //readData1 --> Δεδομένα ανάγνωσης από τη θύρα 1
if (writeReg == readReg2)
     readData2  <= writeData;
```

- Γενικά στο εν λόγω module ακολουθήθηκε η εκφώνηση κατά γράμμα.

ΑΣΚΗΣΗ4: Δημιουργία Datapath (datapath.v)

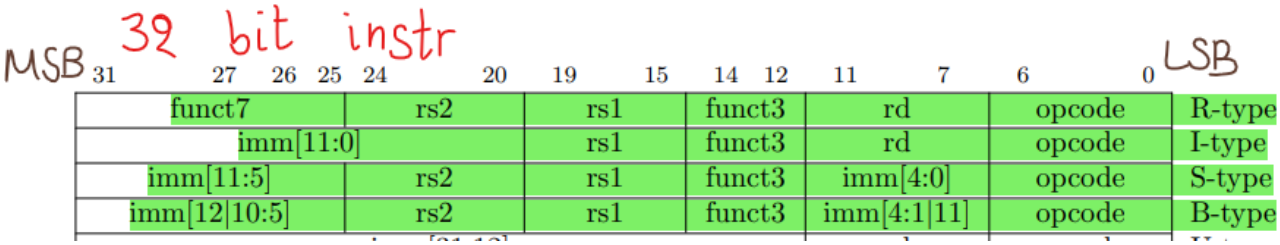
- Αρχικά ακολούθησα το σχήμα 5 και 6 προσεκτικά και πραγματοποίησα την αρχικοποίηση με ρητή αντιστοίχιση θυρών των module alu και regfile. Έθεσα διάφορα internal wires, που έπειτα συνέδεσα με εισόδους και εξόδους module και MUX, ανάλογα τα ζητούμενα της εκφώνησης και των σχημάτων. Τα control signals που δηλώνονται ως είσοδοι στην datapath, χρησιμοποιούνται κανονικά από τους MUX και τα modules, και αρχικοποιούνται στην multicycle.
- Έθεσα σύμφωνα με τους τύπους εντολών risc-spec το opcode ως:

```
assign opcode = instr[6:0];
```

Παρατήρησα ότι οι εντολές, ανάλογα με τον τύπο τους, έχουν ίδιο opcode. (με εξαίρεση την LW που είναι I-type αλλά διαφέρει το opcode της από τις υπόλοιπες). Επομένως έκανα συχνά έλεγχο της τιμής του, για να καταλάβω τι τύπο εντολής είχα. Για αυτό έθεσα παραμέτρους για το opcode του εκάστοτε τύπου εντολής, ώστε να είναι πιο ευανάγνωστος και διαχειρίσιμος ο κώδικας.

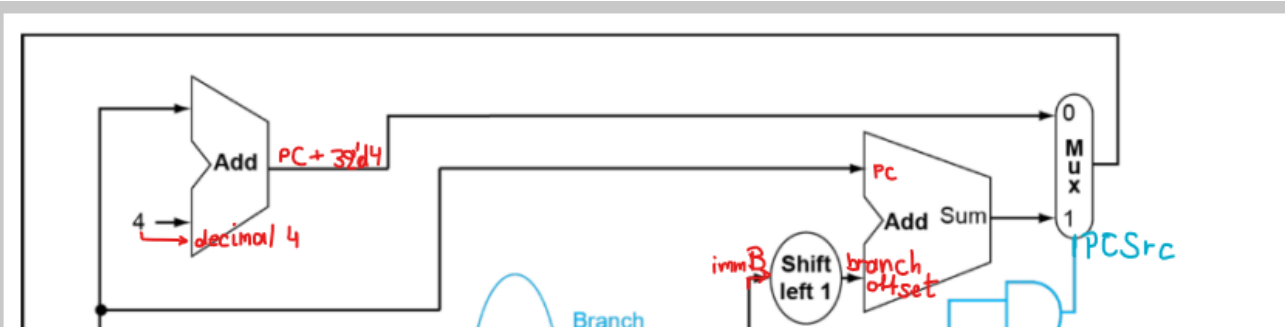
```
    //Parameters of Instruction Types
parameter [6:0] OP_I = 7'b0010011; //I-type
parameter [6:0] OP_LW = 7'b0000011; //LW
parameter [6:0] OP_SW = 7'b0100011; //SW
parameter [6:0] OP_BEQ = 7'b1100011; //BEQ
parameter [6:0] OP_R = 7'b0110011; //R-type
```

- Για την Immediate Generation έθεσα 4 διαφορετικά σήματα imm, ανάλογα τον κάθε τύπο εντολής, καθώς ανάλογα τον τύπο αλλάζει η θέση των 12 bits της imm στα 32bit της instr. Πραγματοποίησα επέκταση προσήμου με concatenation. Δημιούργησα διαφορετικά σήματα για την LW και τις υπόλοιπες I-type εντολές γιατί έχουν διαφορετικό opcode και ήθελα να φαίνεται ξεχωριστά.



- Η υλοποίηση των MUX πραγματοποιήθηκε είτε με always block (επομένως reg σήματα εξόδου των MUX), ώστε οποιαδήποτε αλλαγή στις εισόδους του module, που μπορεί να οδηγήσει σε αλλαγές στις εισόδους του MUX, να αλλάξει αντίστοιχα και η τιμή της εξόδου του πολυπλέκτη. (να υπάρχει με λίγα λόγια μια συνέχεια).

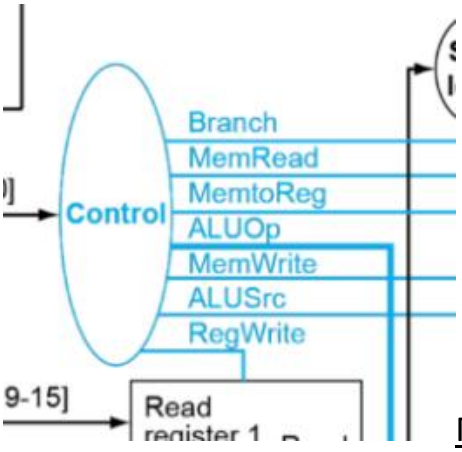
- Ο MUX της εισόδου alu_op2 εξαρτάται από το σήμα ALUSrc, και παίρνει την τιμή της εξόδου immediate generation, όταν είναι 1. Επομένως εξαρτάται από τον τύπο της instr καθώς η τιμή της immediate εξόδου αλλάζει. Για την υλοποίηση αυτού του MUX χρησιμοποιήθηκε μια case(opcode) μέσα σε ένα always block, για να αναγνωρίζεται κάθε φορά ο τύπος εντολής και να χρησιμοποιείται το αντίστοιχο σήμα εξόδου imm.
- Η έξοδος του MUX της εξόδου ram, στέλνει την τιμή του στην είσοδο WriteBack του regfile.(WriteBackData=WriteBack). Ο εν λόγω πολυπλέκτης εξαρτάται από το control signal είσοδο MemToReg.
- Η branch_offset αποτελεί την έξοδο της shift left 1, η οποία πραγματοποιείται μόνο για εντολή διακλάδωσης (BEQ). Ο MUX του PC εξαρτάται από την τιμή της PCSrc για το ποια θα είναι η επόμενη τιμή της εξόδου PC. Το loadPC είναι τύπου enable button για την φόρτωση των επόμενων τιμών του PC. Το rst είναι σύγχρονο, επομένως δεν περιλαμβάνεται στην sensitivity list, αλλάζει σύμφωνα με το ρολόι.



ΑΣΚΗΣΗ5: Δημιουργία ελεγκτή πολλαπλών κύκλων (multicycle.v)

- Η υλοποίηση έγινε με την χρήση ενός FSM με 5 καταστάσεις/στάδια, πηγαίνοντας στην επόμενη κατάσταση κάθε ανερχόμενη ακμή του clk (posedge clk), δηλαδή με ακολουθιακή λογική.
  Τέθηκαν παράμετροι για την κάθε κατάσταση του FSM, με 5bit η καθεμία καθώς οι καταστάσεις του είναι 5 (hot one αποκωδικοποιητής για να είναι πιο ευανάγνωστο).
  Το 5bit state σήμα(τύπου reg), ανάλογα την τιμή του, ενημερώνεται με την επόμενη μέσω non-blocking εντολών(ακολουθιακή λογική).
  Στην περίπτωση θετικού reset, κατά την ανερχόμενη ακμή του (διότι το reset είναι σύγχρονο), πραγματοποιείται reset στην τιμή του state με το αρχικό στάδιο (IF State).
  Υλοποιήθηκαν επίσης κάποια control σήματα (if_enable, id_enable κτλ.) με σκοπό να γνωρίζουμε πότε βρισκόμαστε στο εκάστοτε στάδιο του FSM και να μεταβάλλουμε ανάλογα τα σήματα του Control Unit του Σχήματος 6.
- Το FSM νομίζω είναι τύπου Moore, διότι :
  - Οι έξοδοι του ελεγκτή εξαρτώνται μόνο από την τρέχουσα κατάσταση – state του FSM και όχι από την τωρινή τιμή των εισόδων.
  - Δεν επηρεάζονται από την κάθε αλλαγή των εισόδων άμεσα. (δεν εξαρτώνται απευθείας από τις εισόδους)

- Υλοποιήθηκαν όλα τα σήματα του Control Unit που φαίνεται στο Σχήμα 6 και αρχικοποιήθηκαν σύμφωνα με την εκφώνηση. Μέσω των control σημάτων (στο παράδειγμα wb_enable) του FSM και του opcode, έγινε η αρχικοποίηση των σημάτων του Control Unit ανάλογα με το στάδιο που βρίσκονται και αν επηρεάζονται από αυτό.



Για παράδειγμα:

```
always @*
begin //RegWrite(WB state)
   RegWrite = (wb_enable && ((opcode==OP_I) || (opcode==OP_LW) ||
(opcode==OP_R))) ? 1'b1 : 1'b0;
end
```

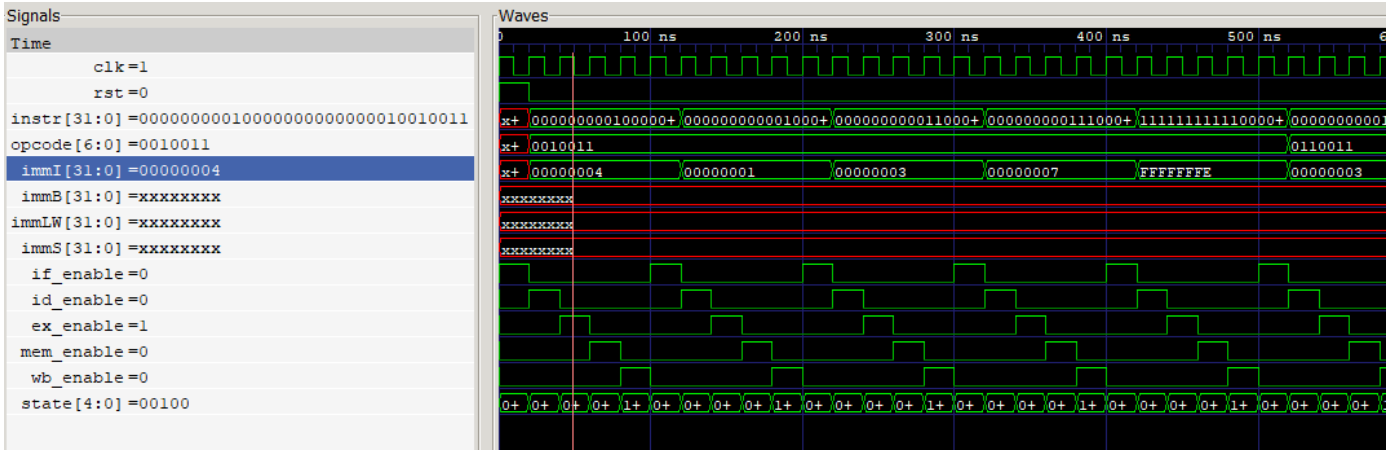
- Τέθηκαν οι 4bit παράμετροι της ALUOP που υπήρχαν στο alu.v, καθώς η ALUCtrl δηλώνει ποια πράξη θα πραγματοποιηθεί στην alu module. (ALUCtrl=alu_op). Το σήμα ALUCtrl εξαρτάται από τον εκάστοτε τύπο εντολής instr, επομένως πραγματοποιείται, μέσα σε ένα always block, διαρκής έλεγχος του είδους εντολής που έχουμε (case(opcode)). Η default είναι για τα R-type και I-type(πέραν της LW). Για να ξεχωρίσουμε συγκεκριμένα την κάθε εντολή μέσα στην default, κάνω έλεγχο funct3. Σε 2 περιπτώσεις που ταυτίζεται η funct3 δύο διαφορετικών εντολών υλοποιείται MUX, όπου το σήμα που καθορίζει την έξοδο του πολυπλέκτη αποτελεί η funct7. Τα σήματα funct3 καθορίζονται από τους I-type και R-type εντολές και το funct7 από την R-type εντολή, σύμφωνα με το risc-spec pdf.

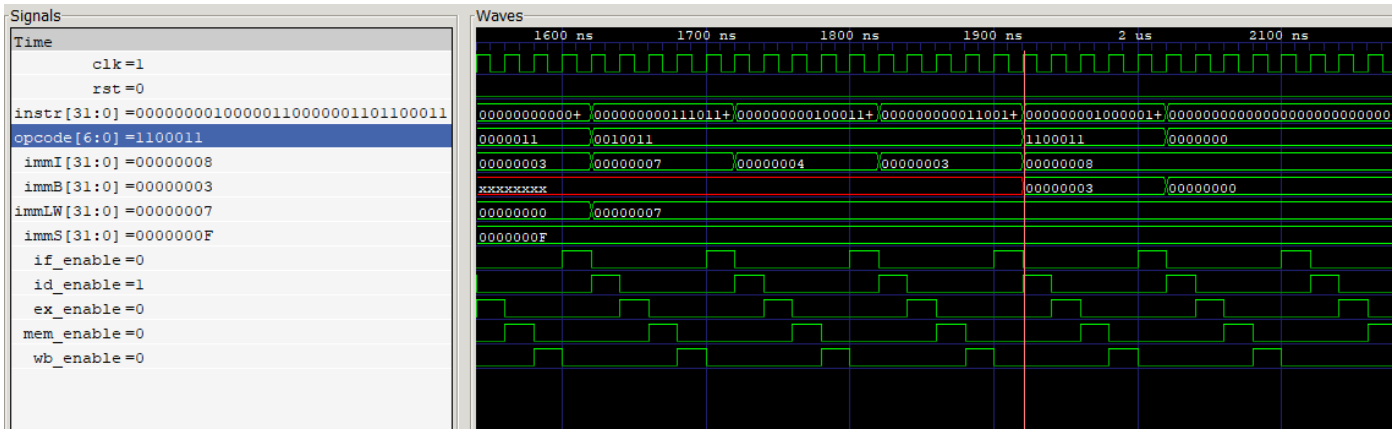
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

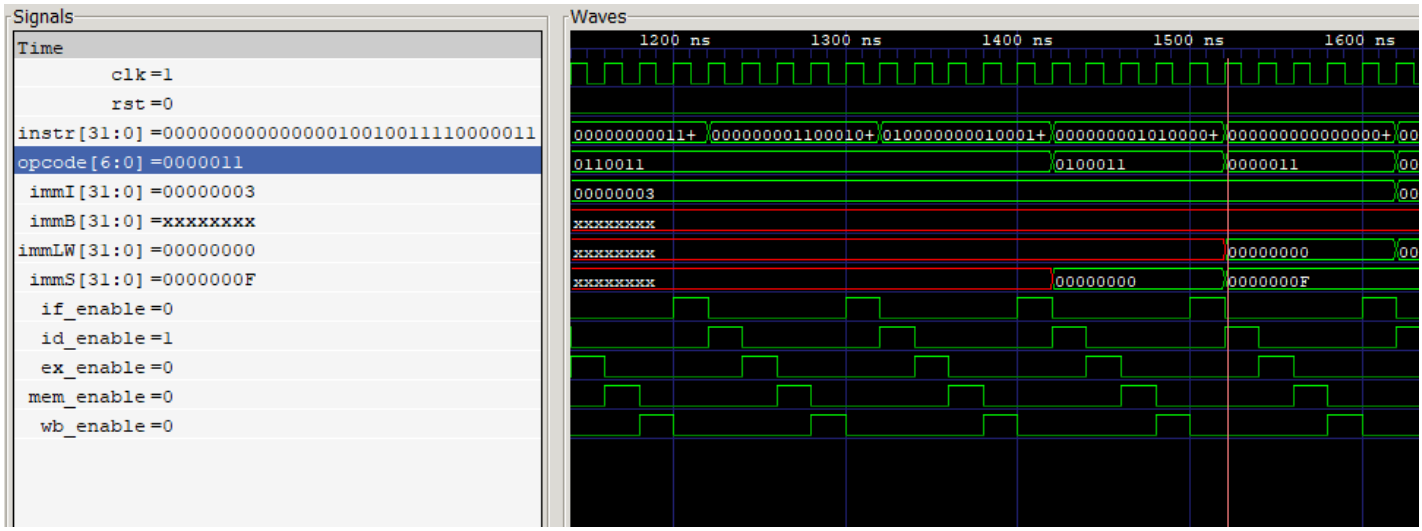
- Το Branch σήμα από το control unit (σχήμα 6) ισχύει μόνο για opcode == OP_BEQ. (τότε, true, παίρνει την τιμή 1)
```
assign Branch = (opcode==OP_BEQ);
```
- Στο στάδιο WB του FSM αρχικοποιείται η RegWrite (εξαρτάται και από τον τύπο εντολής που έχω) και η loadPC γίνεται 1(μόνο σε αυτό το στάδιο). Επίσης μόνο στο εν λόγω στάδιο μας ενδιαφέρει η τιμή της MemToReg (για να φορτωθεί η τιμή από την RAM στην dout της – δλδ. την dReadData, που είναι είσοδος του πολυπλέκτη) και η τιμή της PCSrc (ώστε να υποδείξει την τιμή που θα φορτωθεί στον PC).

Testbench (multicycle_tb.v)

- Πραγματοποίησα instantiation της ram και της rom,όπως και του multicycle module, στο αρχείο του testbench, καθώς ελέγχει ολόκληρη την λειτουργία του RISC-V επεξεργαστή. Επειδή η είσοδος addr είναι 9bit και στα δύο module και συνδέονται με τα 32bit PC (Instruction_Memory) και dAddress (Data_Memory), αντιστοίχισα τα 9 LSB τους. Γενικά στην testbench δημιούργησα μόνο τον παλμό του ρολογιού και στο initial block πραγματοποίησα reset.
- Το σήμα we της ram ταυτίζεται με την τιμή του MemWrite, καθώς μέσα στο αρχείο φαίνεται η σύνδεση της τιμής we με την λειτουργία της MemWrite, όταν πραγματοποιείται store στην data memory (λειτουργεί σαν enable για την εγγραφή στην RAM από την είσοδο din) .
- Το αρχείο “rom_bytes.data” ενημερώνει αυτόματα με την εντολή readmemb, τις τιμές της dout της Instruction Memory. Η instr αποτελείται από 32bit και η κάθε σειρά του αρχείου “rom_bytes.data” αποτελείται από 8 ψηφία. Επομένως:
  (512 γραμμές) / 4 (διότι 4γραμμές η κάθε instr) = 128 εντολές instr
  *(128 εντολές) * (5 κύκλους FSM) = 640 περιόδους* clock
  *(640 περιόδους clock) * (20 ns η περίοδος* clk*) = 12800 ns συνολικά*
- Αρχικά ελέγχω την instr και την opcode σε συνδυασμό με την immediate generation και τις τιμές state και enable του FSM. Προσέχω ότι η immediate αλλάζει τιμή στο 2^{ο} στάδιο του FSM (ID state)
  <u>Ελέγχονται με την σειρά</u>: I-type , B-type, SW & LW type

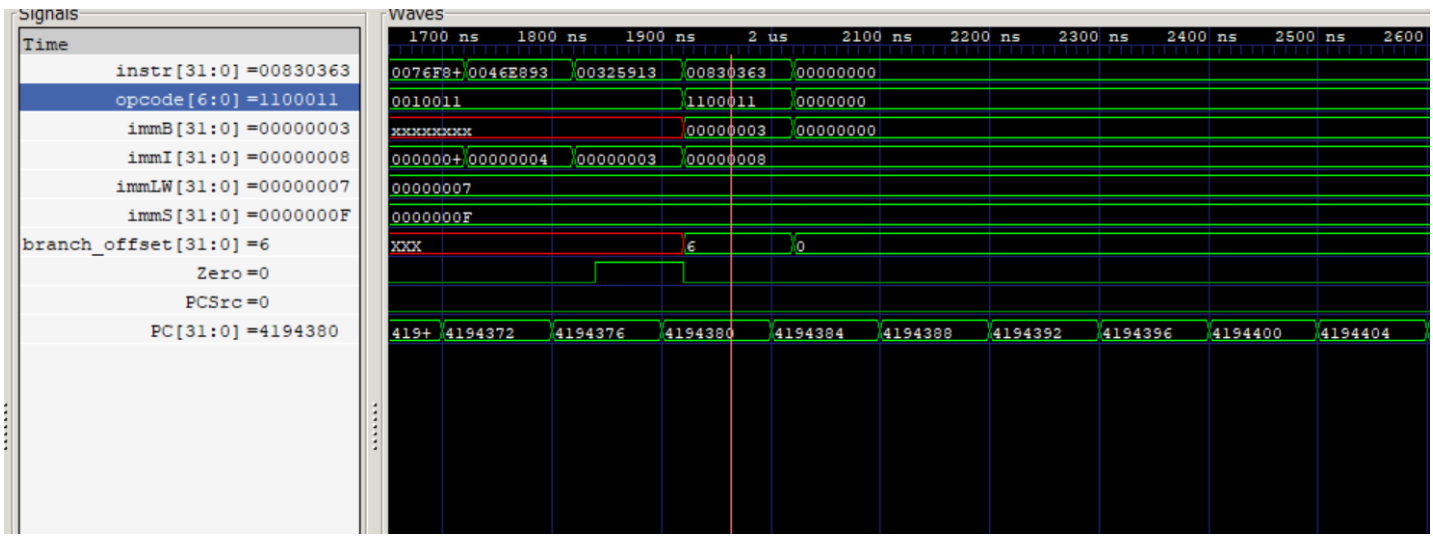




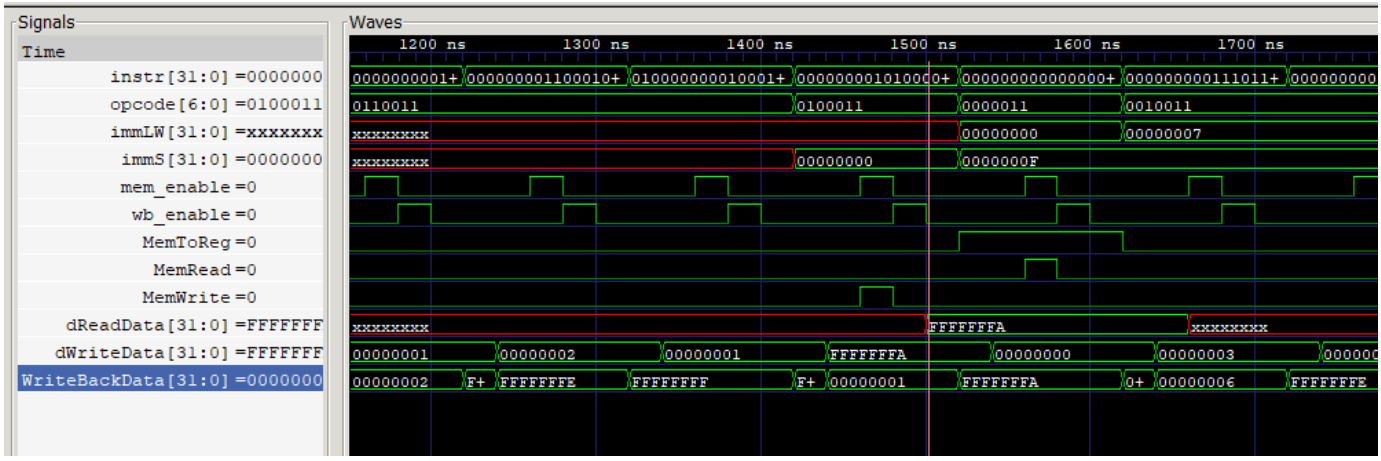


- <u>Παράδειγμα Ελέγχου:</u>
  PC εξαρτάται από το PCSrc:
  - Αν PCSrc → PC <= PC+4 , ενώ αν PCSrc=1 → PC<= PC + branch_offset
  - Ισχύει PCSrc = 1 όταν → (BEQ εντολή && Zero==1)

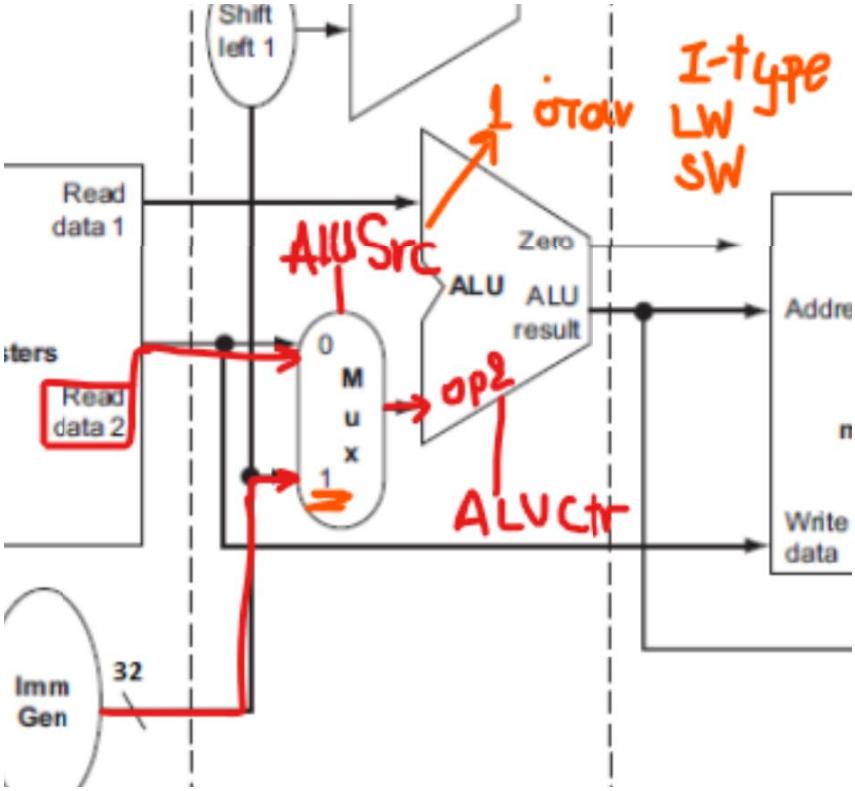
  Εδώ δεν ισχύουν αυτές οι 2 περιπτώσεις. Άρα PC <= PC+4 .

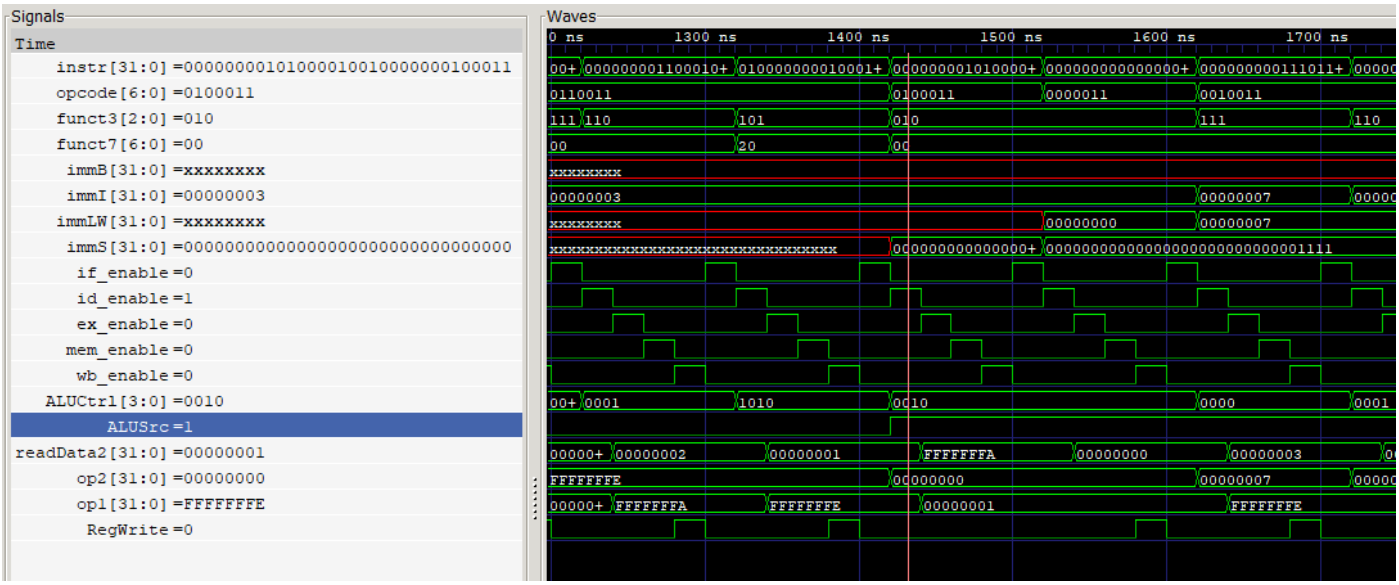


- <u>Παράδειγμα Ελέγχου:</u>
  - Αν opcode==OP_LW → MemToReg=1; → στον MUX η έξοδος WriteData/WriteBackData = dReadData
  - Αν opcode==OP_S (δλδ. SW) → MemWrite =1;
  - Αν opcode==OP_LW → MemRead=1;
  - dReadData παίρνει την τιμή του dWriteData όταν we=0, άρα MemWrite=0, άρα στο WB state, επειδή MemRead και MemWrite τίθενται μόνο κατά τη διάρκεια του σταδίου MEM της FSM.



- <u>Παράδειγμα Ελέγχου</u>: MUX με έξοδο op2





- Με παρόμοια λογική ελέγχθηκαν και τα υπόλοιπα σήματα.

Σχηματικό Διάγραμμα