Ψηφιακά Συστήματα HW-I

Σχεδίαση επεξεργαστή RISC-V

Υπεύθυνη εργασίας: Μανωλίδου Ανατολή

Email: amanolid@ece.auth.gr

ΑΕΜ: 10874

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών ΑΠΘ

7ο εξάμηνο

Θεσσαλονίκη, Οκτώβριος 2024

Περιεχόμενα

Άσκηση 1

Ζητούμενο της πρώτης άσκησης, ήταν η δημιουργία μιας αριθμητικής/λογικής μονάδας (Arithmetic Logic Unit (ALU)). Αυτή η μονάδα είναι υπεύθυνη για την εκτέλεση τόσο των αριθμητικών λειτουργιών καθώς και των λογικών λειτουργιών.

Όσο αφορά τώρα το κώδικα:

- Στην αρχή ορίζονται τα bits όλων των λειτουργιών ως παράμετροι.
- Έπειτα, μέσω ενός always block για κάθε μια λειτουργία ορίζεται και η σωστή πράξη μεταξύ των input ports (op1 και op2). Για το always block που προαναφέρθηκε, χρησιμοποίησα sensitivity list τέτοιο ώστε οποιαδήποτε αλλαγή στα σήματα op1, op2 ή alu_op να ενεργοποιεί το block (always @ (*) begin).
- Όσο αφορά τη πράξη ‘’μικρότερο από’’ χρησιμοποιήθηκε το cast $signed και για το op1 αλλά και για το op2, ώστε και οι δύο να είναι προσημασμένοι.
- Επίσης και για τη πράξη ‘’αριθμητική ολίσθηση δεξιά κατά op2 bits’’ και πάλι χρησιμοποιήθηκε το cast $signed για το op1 ενώ το αποτέλεσμα μετατρέπεται σε μη προσημασμένο μέσω του cast $unsigned.

Άσκηση 2

Ζητούμενο της δεύτερης άσκησης ήταν η δημιουργία ενός κυκλώματος αριθμομηχανής αλλά και ενός testbench για να ελέγξουμε εάν λειτουργεί σωστά.

Ξεκινώντας από τον κώδικα του calc.v, εδώ έπρεπε να ασχοληθούμε κυρίως με τη λογική του accumulator.

- Έτσι μέσω ενός always block, αναθέτουμε στη μεταβλητή led τη τιμή του accumulator.
- Έπειτα ανάλογα με το αν έχει πατηθεί το btnu ή το btnd κουμπί, η τιμή του accumulator γίνεται 0 ή ίση με το result της ALU.
- Όπως είχε ζητηθεί και στην εκφώνηση της εργασίας, στα σήματα op1 και op2 έπρεπε να συνδεθούν με τα σήματα accumulator και sw αντίστοιχα, αφού όμως είχε γίνει πρώτα επέκταση προσήμου σε 32 bits. Γι’ αυτό το λόγο, στο σημείο του κώδικα όπου γίνεται το instantiation της ALU με χρήση της τεχνικής concatenation έγινε η επέκταση προσήμου. Παρατίθεται και το σχετικό κομμάτι του κώδικα:

```
.op1({{16{accumulator[15]}}, accumulator})
.op2({{16{sw[15]}}, sw})
```

Επιπλέον, έπρεπε μέσω κάποιων κυκλωμάτων να παράγουμε το κατάλληλο σήμα alu_op. Αυτό υλοποιήθηκε στο αρχείο calc_enc.v και συγκεκριμένη με τη χρήση της Structural Verilog. Πιο συγκεκριμένα, έγινε περιγραφή κάθε πύλης ξεχωριστά και η σύνδεση μεταξύ των πυλών έγινε μέσω wires.

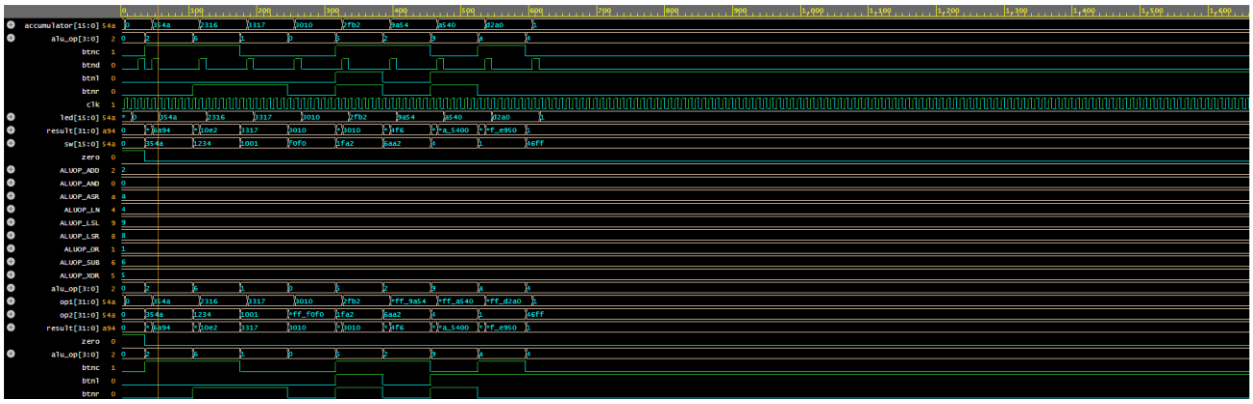
Τέλος, όσο αφορά το testbench, έβαλα να δοκιμάσω όλα τα operations τα οποία υποστηρίζει η αριθμομηχανή.

- Αρχικά, έγινε το instantiation του calc module.
- Δημιούργησα ένα σήμα ρολογιού με περίοδο 10 time units.
- Έπειτα, έκανα initialize όλα τα σήματα με τη τιμή 0.
- Υλοποίησα, test cases για όλα τα operations ορίζοντας κατάλληλα τα σήματα btnc, btnr και btnl. Για να ελέγξω και τη σωστή λειτουργία έβαλα να γίνεται display η αναμενόμενη τιμή αλλά και η τιμή που όντως παίρνουμε. Στιγμιότυπο φαίνεται παρακάτω:

```
# Expected value: 0x0, Got: 0000
# Expected value: 0x0354a, Got: 354a
# Expected value: 0x2316, Got: 2316
# Expected value: 0x3317, Got: 3317
# Expected value: 0x3010, Got: 3010
# Expected value: 0x2fb2, Got: 2fb2
# Expected value: 0x9a54, Got: 9a54
# Expected value: 0xa540, Got: a540
# Expected value: 0xd2a0, Got: d2a0
# Expected value: 0x0001, Got: 0001
```

Εικόνα 1: Τα αποτελέσματα της Άσκησης 2.

Τέλος, αυτές είναι οι κυματομορφές της άσκησης 2:



Εικόνα 2: Οι κυματομορφές από την Άσκηση 2.

Άσκηση 3

Ζητούμενο της τρίτης άσκησης ήταν ο σχεδιασμός του αρχείου καταχωρητών (Register File). Όσο αφορά το κώδικα του regfile.v:

- Αρχικά μέσω ενός for loop αρχικοποιήθηκαν όλοι οι καταχωρητές με μηδενικά.
- Στη συνέχεια, μέσω ενός always block διαβάζουμε τις τιμές των καταχωρητών από κάθε έξοδο και έπειτα σύμφωνα με το σήμα write περνάμε τα δεδομένα από το writeData στη σωστή διεύθυνση. Όπως επισημάνθηκε και στην εκφώνηση, έπρεπε να δοθεί ιδιαίτερη προσοχή στις περιπτώσεις όπου η διεύθυνση εγγραφής συμπίπτει με διεύθυνση ανάγνωσης και έτσι προστέθηκε επιπλέον έλεγχος ώστε να μην υπάρξει λανθασμένη εγγραφή.

Άσκηση 4

Σε αυτή την άσκηση, έπρεπε να σχεδιάσουμε τη μονάδα διαδρομής δεδομένων (Datapath). Όσο αφορά το κώδικα, έχουμε τα εξής:

- Ορίστηκαν όλες τις απαραίτητες παραμέτρους (INITIAL_PC και τα opcodes για τους διάφορους τύπους εντολών) καθώς επίσης και τα inputs και outputs όπως αυτά αναγράφονται στην εκφώνηση.
- Όσο αφορά τον branch offset, έκανα το decoding σύμφωνα με το instruction manual του RISK-V που μας δόθηκε. Πιο συγκεκριμένα, έγινε επέκταση προσήμου για το MSB (instr[31]) ενώ χρησιμοποιήθηκαν τα bits instr[31], instr[7], instr[30:25], instr[11:8] και τέλος προστέθηκε ένα επιπλέον μηδενικό bit (1'b0) για στοίχιση, καθώς το branch offset είναι word-aligned.
- Έπειτα, για την υλοποίηση του Program Counter χρησιμοποιήθηκε ένα always block με sensitivity list τη θετική ακμή του ρολογιού, ενώ ακολούθησα τις οδηγίες όπως αυτές δόθηκαν. Συγκεκριμένα, όταν ενεργοποιείται το rst σήμα το PC παίρνει τη τιμή της παραμέτρου INITIAL_PC αλλιώς όταν ενεργοποιείται το loadPC, το PC αυξάνει κατά τη τιμή του branch offset όταν ενεργοποιείται το PCSrc σήμα ενώ αλλιώς το PC αυξάνεται κατά 4.
- Μετά έγιναν τα instantiations των regfile και alu modules.
- Και πάλι σύμφωνα με το instruction manual έθεσα τα immediate για όλες τις εντολές (εκτός από τις τύπου R που δεν έχουν immediate field). Για τις εντολές τύπου I, η τιμή προέρχεται από τα 12 bits instr[31:20] με επέκταση προσήμου. Για τις εντολές τύπου S, η τιμή προέρχεται από τα bits instr[31:25] και instr[11:7], τα οποία ενώνονται και έπειτα γίνεται η επέκταση προσήμου. Για τις εντολές τύπου B έγινε αναφορά παραπάνω.
- Για την υλοποίηση της ALU, χρησιμοποίησα ένα always block ώστε να ορίζεται η τιμή του σήματος alu_op2 σύμφωνα πάντα με τις εντολές που μας δόθηκαν από την εκφώνηση.
- Τέλος, για την υλοποίηση του Write Back, μέσω ενός always block, με sensitivity list τέτοιο ώστε οποιαδήποτε αλλαγή στα σήματα MemToReg, dReadData ή alu_result να

ενεργοποιεί το block, εάν το MemtoReg σήμα είναι ίσο με 1, η τιμή που θα καταγραφεί θα είναι από τη μνήμη (dReadData) αλλιώς θα είναι από το αποτέλεσμα της ALU (alu_result). Επίσης, τα δεδομένα που γράφονται στους καταχωρητές συνδέονται στην έξοδο WriteBackData του κυκλώματος.

Άσκηση 5

Ζητούμενο της πέμπτης άσκησης ήταν ο σχεδιασμός της μονάδας ελέγχου του RISK-V. Δομικό στοιχείο της μονάδας αποτελεί το FSM (Finite State Machine). Το συγκεκριμένα FSM αποτελείται από τα παρακάτω stages:

Fetch

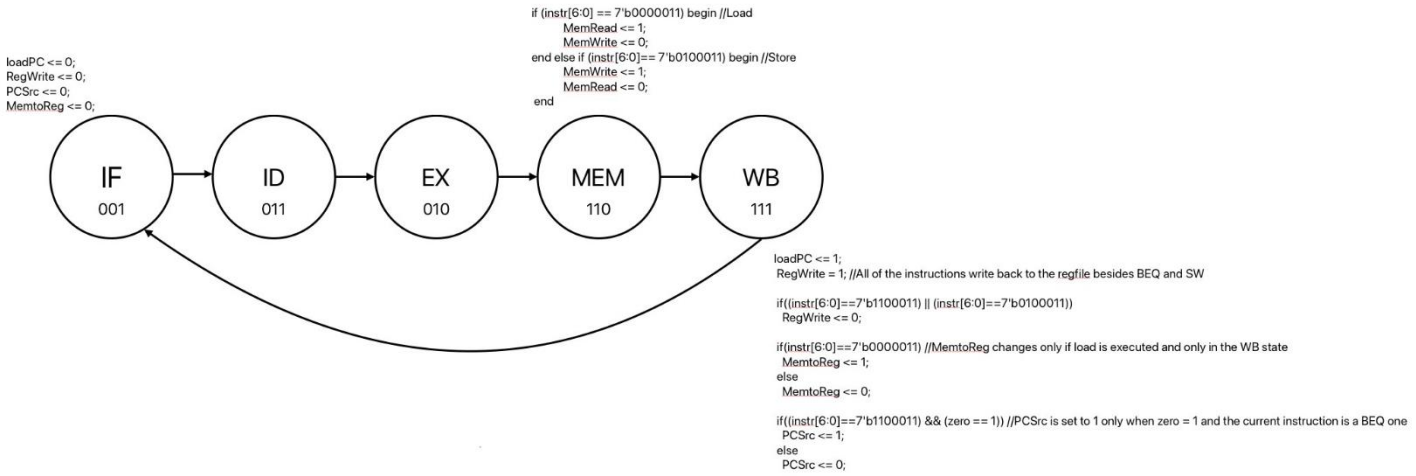
Decode

Execute

Memory Access

Write Back

Ενώ παρακάτω φαίνεται το σχεδιάγραμμα καταστάσεων του FSM:



Εικόνα 3: Διάγραμμα καταστάσεων του FSM.

Όπως φαίνεται και στο διάγραμμα, για τις καταστάσεις χρησιμοποιήθηκε ο κώδικας Gray για 3bits.

Όσο αφορά τον υπόλοιπο κώδικα του top_proc.v, σύμφωνα και με την εκφώνηση έχουμε τα εξής:

- Αρχικά, όρισα τη παράμερο INITIAL_PC καθώς επίσης και τα inputs και outputs όπως δόθηκαν από την εκφώνηση.
- Έγινε το instantiation του datapath.
- Για την υλοποίηση του ALUCtrl χρησιμοποιήθηκε ένα always block με sensitivity list το σήμα instr (καθώς ανάλογα με τη τιμή του, θέτουμε και την ανάλογη εντολή στο ALUCtrl). Έτσι σύμφωνα και με το instruction manual αντιστοίχισα τη κάθε εντολή και την έθεσα στο ALUCtrl.
- Το σήμα ALUSrc καθορίζει αν ο δεύτερος τελεστής της ALU θα προέρχεται από καταχωρητή ή από immediate field της εντολής. Αυτό έγινε με χρήση των bits του opcode της εντολής.
- Τέλος, έχουμε την υλοποίηση του FSM. Όπως αναφέρθηκε και παραπάνω, χρησιμοποιήθηκε κώδικας gray για να οριστούν οι καταστάσεις. Έπειτα, έχουμε το State Transition Logic όπου με ένα always block (με sensitivity list τη θετική ακμή του ρολογιού) θέτουμε ως επόμενη κατάσταση την IF στη περίπτωση όπου έχουμε ενεργοποίηση του rst σήματος. Αλλιώς, η επόμενη κατάσταση ορίζεται όπως παρουσιάστηκε πιο πάνω από το διάγραμμα καταστάσεων. Όσο αφορά το Output Logic των καταστάσεων, στη κατάσταση IF θέτουμε τα σήματα loadPC, RegWrite, PCSrc και MemtoReg ίσα με 0. Στη κατάσταση MEM, θέτουμε το σήμα MemRead ίσο με 1 όταν έχουμε εντολές Load, ενώ το σήμα MemWrite το θέτουμε ίσο με 1 όταν έχουμε εντολές Store. Για την κατάσταση WB, θέτουμε το σήμα loadPc και RegWrite ίσα με 1, όταν όμως έχουμε εντολές BEQ και SW το σήμα RegWrite το θέτουμε ίσο με 0. Όσο αφορά το σήμα MemtoReg αυτό το θέτουμε ίσο με 1 μόνο όταν έχουμε εντολές Load. Για το σήμα PCSrc, αυτό το θέτουμε ίσο με 1 μόνο όταν έχουμε εντολή τύπου BEQ και το σήμα zero είναι ενεργοποιημένο.

Για το top_proc_tb.v, δηλαδή το testbench έχουμε τα εξής:

- Αρχικά, έγιναν τα instantiations τόσο του top_proc module όσο και των DATA_MEMORY (ram.v) και INSTRUCTION_MEMORY(rom.v).
- Στη συνέχεια ακολουθεί το clock generation block όπου δημιουργείται ρολόι με περίοδο 20 time units.
- Μετά, το σήμα rst το θέτουμε ίσο με 1, κάνουμε δηλαδή reset τον processor στο initial state.
- Έπειτα, βάζουμε ένα delay 20 time units και θέτουμε το rst σήμα ίσο με 0.
- Τέλος, βάζουμε ένα delay ίσο με 2700 time units, ώστε να αφήσουμε περιθώριο στον processor να επεξεργαστεί τα instructions. Το νούμερο αυτό προέκυψε από

το γεγονός πως η πρώτη εντολή χρειάζεται 5 κύκλους για να εκτελεστεί ενώ κάθε επόμενη εντολή χρειάζεται συν 1 ακόμη κύκλο δηλαδή συνολικά 132 κύκλους. Αφού έχουμε θέσει το ρολόι στα 20 time units, συνολικά θέλουμε $(132 * 20) + 20 = 2660$ time units, όπου τα 20 time units που προσθέσαμε είναι το delay πριν ξεκινήσει ο processor να εκτελεί τα instructions.

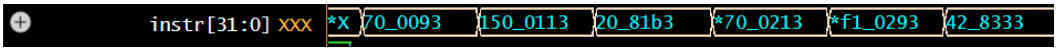
Όσο αφορά το αρχείο rom_bytes.data, αυτό περιέχει 512 σειρές το οποίο σημαίνει πως συνολικά περιλαμβάνει 128 εντολές. Αυτό προκύπτει από το γεγονός πως το κάθε instruction είναι 32 bit ενώ κάθε σειρά περιλαμβάνει ένα αριθμό των 8 bits.

Αφού τρέξουμε τους κώδικες στη συνέχεια πρέπει να παρατηρήσουμε τις κυματομορφές σε συνδυασμό με τα instructions από το rom_bytes.data αρχείο. Κάνοντας decoding τη πρώτη εντολή έχουμε τα εξής αποτελέσματα:

```
       Assembly =   addi x1, x0, 7
         Binary =   0000 0000 0111 0000 0000 0000 1001 0011
    Hexadecimal =   0x00700093
         Format =   I-type
Instruction set =   RV32I
         Manual =   addi
```

Εικόνα 4: Decoding 1ου instruction

Ενώ από τις κυματομορφές έχουμε τα εξής:



Εικόνα 5: Στιγμιότυπο από τις κυματομορφές, εδώ φαίνεται το σήμα instr.

Παρατηρούμε πως ο processor έχει κάνει σωστά fetch το πρώτο instruction από τη μνήμη. Έπειτα για τη δεύτερη εντολή έχουμε το παρακάτω decoding:

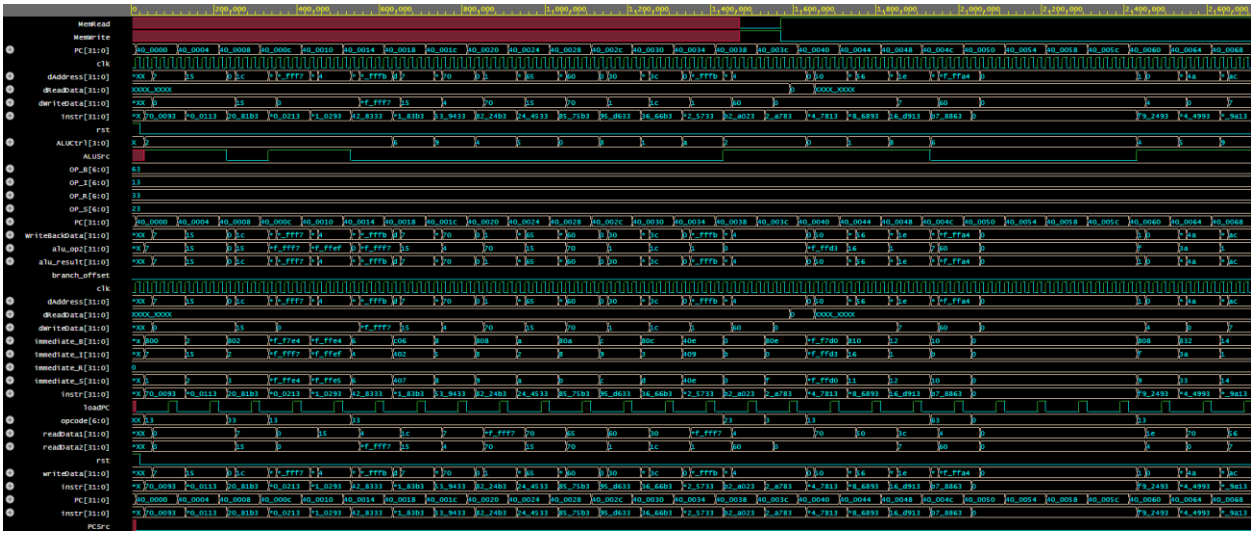
```
       Assembly =   addi x2, x0, 21
         Binary =   0000 0001 0101 0000 0000 0001 0001 0011
    Hexadecimal =   0x01500113
         Format =   I-type
Instruction set =   RV32I
         Manual =   addi
```

Εικόνα 6: Decoding 2ου instruction

Παρατηρώντας και τις κυματομορφές, βλέπουμε πως και πάλι πως ο processor έχει κάνει σωστά fetch το δεύτερο instruction από τη μνήμη. Με την ίδια λογική ελέγχθηκαν και οι υπόλοιπες εντολές, οι οποίες παρουσιάζονται αναλυτικά στο παρακάτω πίνακα:

Binary	Hexadecimal	Assembly
00000000011100000000000010010011	0x00700093	addi x1, x0, 7
00000001010100000000000100010011	0x01500113	addi x2, x0, 21
00000000001000001000000110110011	0x002081b3	add x3, x1, x2
11111111011100000000001000010011	0xff700213	addi x4, x0, -9
11111110111100010000001010010011	0xfef10293	addi x5, x2, -17
00000000010000101000001100110011	0x00428333	add x6, x5, x4
01000000001000011000001110110011	0x402183b3	sub x7, x3, x2
00000000010100111001010000110011	0x00539433	sll x8, x7, x5
00000000100000100010010010110011	0x008224b3	slt x9, x4, x8
00000000001001000100010100110011	0x00244533	xor x10, x8, x2
00000000100001010111010110110011	0x008575b3	and x11, x10, x8
00000000100101011101011000110011	0x0095d633	srl x12, x11, x9
00000000001101100110011010110011	0x003666b3	or x13, x12, x3
01000000100100100101011100110011	0x40925733	sra x14, x4, x9
00000000101100101010000000100011	0x00b2a023	sw x11, 0(x5)
00000000000000010101001111000011	0x0002a783	lw x15, 0(x5)
11111101001101000111100000010011	0xfd347813	andi x16, x8, -45
00000001011010000110100010010011	0x01686893	ori x17, x16, 22
00000000000101101101100100010011	0x0016d913	srli x18, x13, 1
00000000101101111000100001100011	0x00b78863	beq x15, x11, 16
00000000111111001001001001010011	0x00f92493	slti x9, x18, 15
00000011101001000100100110010011	0x03a44993	xori x19, x8, 58
00000000000110001001101000010011	0x00189a13	slli x20, x17, 1
01000000001001111101001010010011	0x4027d293	srai x5, x15, 2

Παρακάτω, παρατίθενται και ένα επιπλέον στιγμιότυπο από τις κυματομορφές τις άσκησης 5.



Εικόνα 7: Στιγμιότυπο κυματομορφών άσκησης 5