Report: RISC-V CPU

Παναγιώτης Κούτρης

10671

Contents

1 Part 1

Σε αυτή την άσκηση κληθήκαμε να υλοποιήσουμε μια αριθμητική/λογική μονάδα (ALU) για έναν επεξεργαστή RISC-V.

Η διαδικασία που ακολουθήσαμε είναι η εξής:

1. Ορίσαμε τα εισερχόμενα και εξερχόμενα σήματα του module, δηλαδή τα δύο 32-bit δεδομένα εισόδου (`op1` και `op2`) και τον έλεγχο λειτουργίας (`alu_op`), καθώς και τις εξόδους για το αποτέλεσμα (`result`) και την ένδειξη μηδενισμού (`zero`).

2. Καθορίσαμε τις παραμέτρους για κάθε τύπο λειτουργίας της ALU (`AND`, `OR`, `ADD`, `SUB`, `SLT`, `LSR`, `LSL`, `ASR`, `XOR`).

3. Στη συνέχεια, χρησιμοποιήσαμε ένα `always @ (*)` block, το οποίο είναι κατάλληλο για την υλοποίηση συνδυαστικής λογικής. Με βάση την τιμή του σήματος `alu_op`, ελέγχει ποια από τις εντολές της ALU θα εκτελεστεί.

4. Για τις πράξεις αριθμητικής φύσης, χρησιμοποιήσαμε τους `signed` τελεστές για να υποστηρίξουμε την προσημασμένη πρόσθεση, αφαίρεση και σύγκριση.

5. Για τις πράξεις ολίσθησης, χρησιμοποιήσαμε τις αντίστοιχες λογικές και αριθμητικές ολίσθησης.

6. Ελέγξαμε αν το αποτέλεσμα είναι μηδέν και αναθέσαμε την τιμή `1` στην έξοδο `zero` αν ισχύει.

Η ALU υλοποιήθηκε ως συνδυαστικό κύκλωμα, εξασφαλίζοντας ότι το αποτέλεσμα προκύπτει άμεσα από τις εισόδους χωρίς μνήμη ή σήματα ρολογιού, προσφέροντας άμεση απόκριση στις αλλαγές των εισόδων.

2 Part 2

Σε αυτήν την άσκηση κληθήκαμε να υλοποιήσουμε μια αριθμομηχανή που χρησιμοποιεί την αριθμητική/λογική μονάδα (ALU) της προηγούμενης άσκησης, καθώς και έναν 16-bit συσσωρευτή.

Η διαδικασία που ακολουθήσαμε είναι η εξής:

1. Χρησιμοποιήσαμε το module `calc_enc` για να παράγουμε το σήμα ελέγχου `alu_op`, βασισμένο στις καταστάσεις των πλήκτρων `btnl`, `btnc` και `btnr`, σύμφωνα με τη λογική συνδυασμού που ορίζει η εκφώνηση.

2. Επεκτείναμε τα σήματα του συσσωρευτή (`accumulator`) και των διακοπτών (`sw`) σε 32-bit με επέκταση προσήμου, ώστε να συνδεθούν σωστά στις εισόδους της ALU.

3. Συνδέσαμε την ALU, η οποία υπολογίζει το αποτέλεσμα με βάση τους τελεστές `ext_accumulator` και `ext_sw`, καθώς και το `alu_op`.

4. Σχεδιάσαμε έναν σύγχρονο 16-bit συσσωρευτή (`accumulator`), ο οποίος:

   - Μηδενίζεται με το πάτημα του `btnu`.
   - Ενημερώνεται με τα 16 χαμηλότερα bit της εξόδου της ALU όταν πατηθεί το `btnd`.

5. Συνδέσαμε την τιμή του συσσωρευτή με τις εξόδους `led`.

6. Επαληθεύσαμε τη λειτουργία του κυκλώματος με ένα testbench, το οποίο δοκιμάζει διάφορες λειτουργίες της αριθμομηχανής (πρόσθεση, αφαίρεση, λογικές πράξεις, λειτουργίες ολίσθησης κ.ά.) και καταγράφει τα αποτελέσματα.

Η αριθμομηχανή υλοποιήθηκε με τη χρήση συνδυαστικής και ακολουθιακής λογικής. Συγκεκριμένα:

- Χρησιμοποιήσαμε `always @ (*)` blocks για τις συνδυαστικές λογικές λειτουργίες.

- Ο συσσωρευτής σχεδιάστηκε με χρήση `always @(posedge clk)` για να λειτουργεί ως σύγχρονο κύκλωμα με μνήμη.

2.1 Αποτελέσματα

```
[2025-01-21 15:16:47 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile dump.vcd opened for output.
Test 1 Passed: Reset succeeded.
Test 2 Passed: ADD operation succeeded.
Test 3 Passed: SUB operation succeeded.
Test 4 Passed: OR operation succeeded.
Test 5 Passed: AND operation succeeded.
Test 6 Passed: XOR operation succeeded.
Test 7 Passed: ADD operation succeeded.
Test 8 Passed: Logical Shift Left succeeded.
Test 9 Passed: Shift Right Arithmetic succeeded.
Test 10 Passed: Less Than operation succeeded.
testbench.sv:128: $finish called at 210000 (1ps)
Done
```

Figure 1: Αποτελέσματα Log

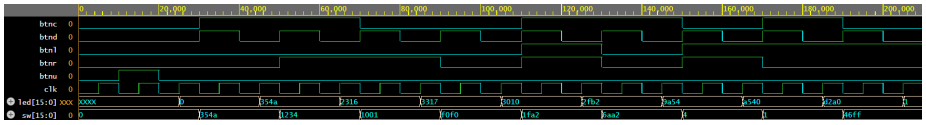

Figure 2: Κυματομορφές Προσομοίωσης

3 Part 3

Σε αυτήν την άσκηση κληθήκαμε να υλοποιήσουμε ένα αρχείο καταχωρητών (register file), το οποίο είναι υπεύθυνο για την αποθήκευση και ανάγνωση δεδομένων από τους 32 καταχωρητές του επεξεργαστή.

Η διαδικασία που ακολουθήσαμε είναι η εξής:

1. Δημιουργήσαμε έναν πίνακα 32 καταχωρητών, με δυνατότητα παραμετροποίησης του πλάτους των δεδομένων (`DATAWIDTH`).
2. Αρχικοποιήσαμε όλους τους καταχωρητές σε μηδενική τιμή χρησιμοποιώντας ένα `initial` block και έναν βρόχο `for`.
3. Υλοποιήσαμε τη λειτουργικότητα εγγραφής και ανάγνωσης μέσω ενός `always @(posedge clk)` block:
   - Όταν το σήμα `write` είναι ενεργό, γίνεται εγγραφή του `writeData` στον καταχωρητή που υποδεικνύεται από τη διεύθυνση `writeReg`.
   - Διαχειριστήκαμε περιπτώσεις επικάλυψης, όπου η διεύθυνση εγγραφής είναι ίδια με μία από τις διευθύνσεις ανάγνωσης (`readReg1` ή `readReg2`), διασφαλίζοντας ότι τα δεδομένα ανάγνωσης ενημερώνονται άμεσα με τα δεδομένα εγγραφής.
   - Αν το σήμα `write` δεν είναι ενεργό, γίνεται μόνο ανάγνωση από τους καταχωρητές στις διευθύνσεις `readReg1` και `readReg2`.
4. Διασφαλίσαμε ότι η υλοποίηση είναι συμβατή με την προδιαγραφή του πλάτους των δεδομένων (`DATAWIDTH`), κάνοντας τον κώδικα παραμετροποιήσιμο.

Το αρχείο καταχωρητών υλοποιήθηκε ως ακολουθιακό κύκλωμα, καθώς χρησιμοποιεί σήμα ρολογιού για τη λειτουργία εγγραφής και ανάγνωσης. Η σχεδίαση αυτή επιτρέπει ασφαλή και συγχρονισμένη διαχείριση των δεδομένων.

4 Part 4

Σε αυτήν την άσκηση κληθήκαμε να υλοποιήσουμε το datapath ενός επεξεργαστή RISC-V,

Η διαδικασία που ακολουθήσαμε είναι η εξής:

1. **Αποκωδικοποίηση εντολών:**
   - Αναλύσαμε την εντολή εισόδου (`instr`) και εξαγάγαμε τα πεδία `opcode`, `funct3`, `funct7`, καθώς και τα `rd`, `rs1`, `rs2` και `imm`.
   - Υλοποιήσαμε έναν αποκωδικοποιητή για την κατηγοριοποίηση των εντολών σε τύπους R, I, S, και B, με χρήση συνδυαστικής λογικής.
2. **Σύνδεση με το αρχείο καταχωρητών:**
   - Συνδέσαμε τις διευθύνσεις `rs1`, `rs2`, και `rd` με το αρχείο καταχωρητών (`regfile`).
   - Υποστηρίξαμε λειτουργίες ανάγνωσης (`readData1`, `readData2`) και εγγραφής (`WriteBackData`) με έλεγχο από το σήμα `RegWrite`.
3. **Σύνδεση με την ALU:**
   - Χρησιμοποιήσαμε τους τελεστές `readData1` και `readData2` από το αρχείο καταχωρητών.
   - Προσθέσαμε επιλογή για την είσοδο `op2` μέσω του σήματος `ALUSrc`, ώστε να υποστηρίζεται και η χρήση άμεσων δεδομένων (`imm`).
   - Συνδέσαμε το αποτέλεσμα της ALU (`ALUResult`) με τα σήματα ελέγχου για τη μνήμη δεδομένων και την εγγραφή στο αρχείο καταχωρητών.
4. **Διαχείριση του μετρητή προγράμματος (PC):**
   - Υλοποιήσαμε έναν σύγχρονο μετρητή προγράμματος που αρχικοποιείται με την παράμετρο `INITIAL_PC`.
   - Χρησιμοποιήσαμε ένα σήμα πολυπλέκτη (`PCSrc`) για την επιλογή της επόμενης διεύθυνσης, υποστηρίζοντας διακλαδώσεις (`branch`).
5. **Ενσωμάτωση μνήμης δεδομένων:**
   - Συνδέσαμε το αποτέλεσμα της ALU (`ALUResult`) ως διεύθυνση (`dAddress`) και το `readData2` ως δεδομένα προς εγγραφή (`dWriteData`).
   - Υλοποιήσαμε πολυπλέκτη για το `WriteBackData`, ώστε να επιλέγεται μεταξύ της μνήμης δεδομένων (`dReadData`) και του αποτελέσματος της ALU.

Το datapath υλοποιήθηκε με συνδυαστική και ακολουθιακή λογική:

- Χρησιμοποιήσαμε `always @ (*)` για τη συνδυαστική λογική (π.χ. αποκωδικοποίηση εντολών και πολυπλέκτες).
- Ο μετρητής προγράμματος και οι λειτουργίες εγγραφής υποστηρίχθηκαν από ακολουθιακή λογική μέσω `always @(posedge clk)`.

5 Part 5

Σε αυτήν την άσκηση κληθήκαμε να υλοποιήσουμε έναν ελεγκτή πολλαπλών κύκλων (multi-cycle controller) για την ολοκλήρωση της λειτουργίας του επεξεργαστή RISC-V. Ο ελεγκτής οργανώνει τη ροή δεδομένων μέσω του datapath και παράγει τα απαραίτητα σήματα ελέγχου για κάθε εντολή.

Η διαδικασία που ακολουθήσαμε είναι η εξής:

1. **Αποκωδικοποίηση εντολών:**
   - Αναλύσαμε την εντολή εισόδου (`instr`) και εξαγάγαμε τα πεδία `opcode`, `funct3`, `funct7`, καθώς και τα `rd`, `rs1`, `rs2` και `imm`.
   - Χρησιμοποιήσαμε το αποκωδικοποιημένο αποτέλεσμα (`command`) για να καθορίσουμε τις απαιτούμενες λειτουργίες της ALU, της μνήμης, και των καταχωρητών.
2. **Σχεδίαση FSM:**
   - Υλοποιήσαμε μία μηχανή κατάστασης πέντε σταδίων (`IF`, `ID`, `EX`, `MEM`, `WB`):
     - Στην κατάσταση `IF` γίνεται ανάγνωση εντολών από τη μνήμη εντολών.
     - Στην κατάσταση `ID` γίνεται αποκωδικοποίηση της εντολής και πρόσβαση στους καταχωρητές.
     - Στην κατάσταση `EX` εκτελείται η λειτουργία της ALU.
     - Στην κατάσταση `MEM` πραγματοποιούνται οι λειτουργίες μνήμης (`LW`/`SW`).
     - Στην κατάσταση `WB` ενημερώνονται οι καταχωρητές και ο μετρητής προγράμματος (`PC`).
3. **Παραγωγή σημάτων ελέγχου:**
   - Καθορίσαμε τα σήματα ελέγχου (`ALUCtrl`, `ALUSrc`, `MemToReg`, `PCSrc`, `RegWrite`, `MemRead`, `MemWrite`) για κάθε στάδιο και εντολή.
   - Προσαρμόσαμε τα σήματα ώστε να υποστηρίζουν όλες τις βασικές κατηγορίες εντολών (τύποι R, I, S, B, LW, SW).
4. **Σύνδεση με το datapath:**
   - Συνδέσαμε τις εξόδους του ελεγκτή με τις εισόδους του datapath για να ελέγξουμε τη ροή δεδομένων, τις πράξεις της ALU, και τις λειτουργίες μνήμης.
   - Χρησιμοποιήσαμε τις τιμές των σημάτων `Zero` και `PCSrc` για την υλοποίηση των διακλαδώσεων (`BEQ`).

5. **Testbench:**

   - Δημιουργήσαμε ένα testbench που προσομοιώνει όλες τις υποστηριζόμενες εντολές, συνδέοντας το datapath με τη μνήμη εντολών (`ROM`) και τη μνήμη δεδομένων (`RAM`).
   - Ελέγξαμε τη λειτουργία του επεξεργαστή μέσω της παρακολούθησης των εξόδων (`PC`, `WriteBackData`, `dAddress`, `dReadData`, `dWriteData`).

Ο ελεγκτής σχεδιάστηκε χρησιμοποιώντας συνδυαστική και ακολουθιακή λογική:

- Χρησιμοποιήσαμε `always @ (*)` για την παραγωγή συνδυαστικών σημάτων (π.χ., `ALUCtrl`, `ALUSrc`, `PCSrc`).
- Υλοποιήσαμε τη μηχανή κατάστασης (`FSM`) με `always @(posedge clk)`, ώστε να διασφαλίζεται η ακολουθιακή εκτέλεση των σταδίων.

5.1 Σχηματικό Διάγραμμα FSM

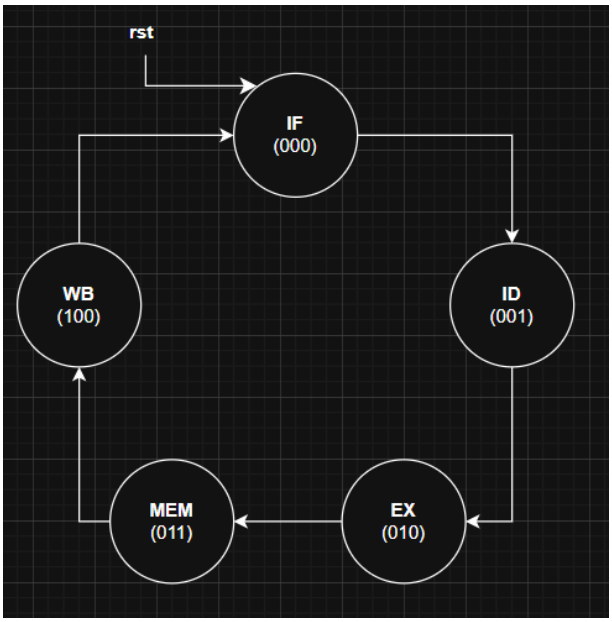

Figure 3: FSM diagram

5.2 Εντολές

Οι εντολές που τρέχει ο επεξεργαστής από τη ROM είναι οι εξής:

Instruction 1: `addi x1, x0, 7`	**Instruction 13:** `or x13, x12, x3`
Instruction 2: `addi x2, x0, 21`	**Instruction 14:** `sra x14, x4, x9`
Instruction 3: `add x3, x1, x2`	**Instruction 15:** `sw x11, 0(x5)`
Instruction 4: `addi x4, x0, -9`	**Instruction 16:** `lw x15, 0(x5)`
Instruction 5: `addi x5, x2, -17`	**Instruction 17:** `andi x16, x8, -45`
Instruction 6: `add x6, x5, x4`	**Instruction 18:** `ori x17, x16, 22`
Instruction 7: `sub x7, x3, x2`	**Instruction 19:** `srli x18, x13, 1`
Instruction 8: `sll x8, x7, x5`	**Instruction 20:** `beq x15, x11, 16`
Instruction 9: `slt x9, x4, x8`	**Instruction 21:** `slti x9, x18, 15`
Instruction 10: `xor x10, x8, x2`	**Instruction 22:** `xori x19, x8, 58`
Instruction 11: `and x11, x10, x8`	**Instruction 23:** `slli x20, x17, 1`
Instruction 12: `srl x12, x11, x9`	**Instruction 24:** `srai x5, x15, 2`

Table 1: List of Instructions

5.3 Αποτελέσματα

Από την ανάλυση των σημάτων που τυπώνονται στο log από το testbench (PC, Instr, dAddress, dWriteData, dReadData, WriteBackData, MemRead και MemWrite), παρατηρήθηκε ότι όλες οι εντολές εκτελούνται σωστά. Το PC ενημερώνεται με τη σωστή τιμή σε κάθε κύκλο, επιβεβαιώνοντας τη σωστή διαχείριση της ροής εντολών. Οι εντολές αποκωδικοποιούνται ορθά, και οι τιμές στα σήματα dAddress, dWriteData, dReadData και WriteBackData είναι συνεπείς με την εκάστοτε εντολή. Επιπλέον, τα σήματα MemRead και MemWrite ενεργοποιούνται μόνο όταν απαιτείται ανάγνωση ή εγγραφή στη μνήμη RAM, επιβεβαιώνοντας την ορθή λειτουργία της μονάδας μνήμης. Η συνολική συμπεριφορά του συστήματος είναι σύμφωνη με τις προσδοκίες, και τα αποτελέσματα συμφωνούν με το θεωρητικό μοντέλο.

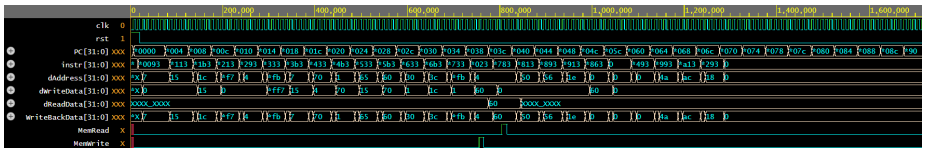

Figure 4: Κυματομορφές Προσομοίωσης