Ψηφιακά Συστήματα Hardware σε Υψηλά Επίπεδα Λογικής 1

Αναφορά Εργασίας

Σάββας Τζανέτης
10889
stzanetis@ece.auth.gr

15 Ιανουαρίου 2025

Εισαγωγή

Αυτή η αναφορά είναι μέρος της εργασίας του μαθήματος **Ψηφιακά Συστήματα Hardware σε Χαμηλά Επίπεδα Λογικής 1** του **Αριστοτέλιου πανεπιστημίου Θεσσαλονίκης**.
Σε αυτή την εργασία, μας ζητήθηκε να υλοποιήσουμε μια αριθμομηχανή, καθώς και έναν επεξεργαστή **RISC-V** με την βοήθεια των παρακάτω ασκήσεων σε γλώσσα **Verilog**.
Στην αναφορά θα γίνει μια σύντομη επεξήγηση των ασκήσεων, σχολιασμός του κώδικα και παρουσίαση των ζητούμενων διαγραμμάτων και κυματομορφών των προσομοιώσεων.

1 Πρώτη Άσκηση

Στη πρώτη άσκηση μας ζητήθηκε να υλοποιήσουμε μια ¨Αριθμητική και Λογική Μονάδα¨ **(ALU)**.

Η υλοποίηση μιας **ALU** σε **Verilog** είναι αρκετά απλή, καθώς αποτελείται από έναν **multiplexer** που υλοποιήθηκε κάνοντας assign στην έξοδο result το ανάλογο αποτέλεσμα μετά από σύγκριση των παραμέτρων που έχουν οριστεί στην αρχή του κωδικά με τις διάφορες λειτουργίες που θέλουμε να εκτελεί η ALU. Η σύγκριση γίνεται με την χρίση λογικόν πράξεων που προσομοιωνόσουν μια σειρά από else if εντολές. Στη συνεχεία μετά από την επιλογή της ανάλογης λειτουργίας πραγματοποιείται η κατάλληλη πράξη μεταξύ των εισόδων **op1** και **op2**. Να σημειωθεί επίσης, πως στις ειδικές περιπτώσεις των πράξεων ¨**less than**¨ και ¨**arithmetic shift right**¨ οι είσοδοι **op1** και **op2** μετατρέπονται σε προσημασμένες τιμές για την σωστή λειτουργία των πράξεων.

```
assign result = (alu_op == ALUOP_AND) ? (op1 & op2):
    (alu_op == ALUOP_OR)          ? (op1 | op2)
    (alu_op == ALUOP_ADD)         ? (op1 + op2):
    (alu_op == ALUOP_SUB)         ? (op1 - op2):
    (alu_op == ALUOP_LTHAN)       ? ($signed(op1) < $signed(op2)):
    (alu_op == ALUOP_LSHIFTR)     ? (op1 >> op2[4:0]):
    (alu_op == ALUOP_LSHIFTL)     ? (op1 << op2[4:0]):
    (alu_op == ALUOP_ASHIFTR)     ? ($signed(op1) >>> op2[4:0]):
    (alu_op == ALUOP_XOR)         ? (op1 ^ op2):
    32'b0;
```

2 Δεύτερη Άσκηση

Η δεύτερη άσκηση έχει ως ζητούμενο, την υλοποίηση της αριθμομηχανής με την βοήθεια του **alu.v** module της πρώτης άσκησης, αλλά και ενός **testbench** για την επαλήθευση της σωστής λειτουργίας της αριθμομηχανής. Για την μοντελοποίηση της αριθμομηχανής μας ζητήθηκαν δυο αρχεία, το αρχείο **calc.v** το οποίο άφορα την γενική λειτουργιά της αριθμομηχανής, και το αρχείο **calc_enc.v** το οποίο είναι υπεύθυνο για την παράγωγη του ανάλογου σήματος **alu_op** το οποίο θα εισαχθεί στο **ALU** module της πρώτης άσκησης.

Ο κωδικοποιητής **calc_enc.v**, παράγει το κατάλληλο σήμα με την χρήση **Structural Verilog**, ακολουθώντας τα σχήματα 2 έως 5 της εκφώνησης της εργασίας για την δημιουργία των κατάλληλων συνδυαστικών κυκλωμάτων. Η κωδικοποίηση αυτών των σημάτων γίνεται με την εισαγωγή τριών σημάτων εισόδου, τα οποία λειτουργούν ως 'κουμπιά' και τα ονομάζουμε **btnc**, **btnl** και **btnr**.

Στο αρχείο **calc.v** δημιουργούμε την βασική λειτουργιά της αριθμομηχανής συμφωνά με το Σχήμα 2.1 της εκφώνησης. Συγκεκριμένα:

- Υλοποιείται ο συσσωρευτής με την χρήση ενός always block.

```
always @(posedge clk) begin
    if(btnu)
        accumulator <= 16'b0;
    else if(btnd)
        accumulator <= alu_out[15:0];
end
```

- Ή επέκταση πρόσημου.

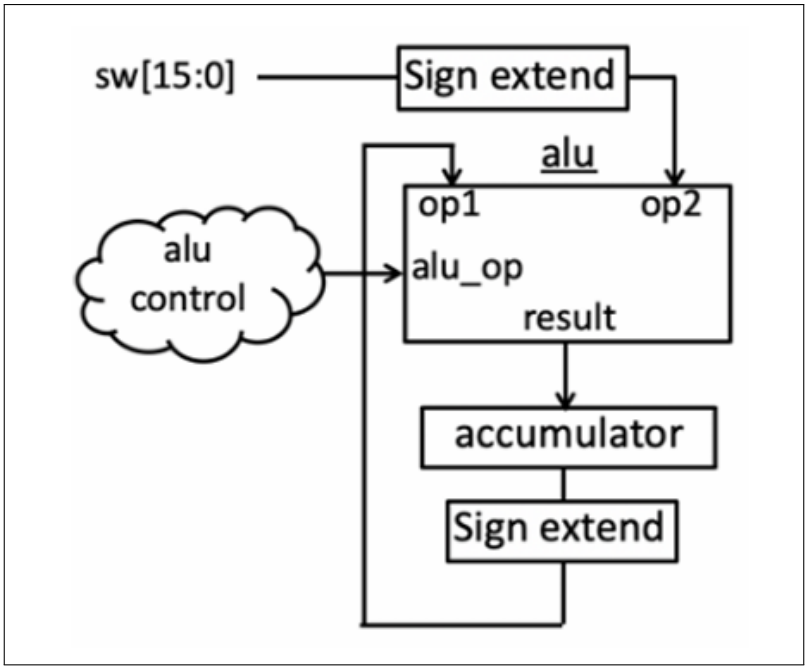
```
wire[31:0] op1_extended = {{16{accumulator[15]}}, accumulator};
wire[31:0] op2_extended = {{16{sw[15]}}, sw};
```

- Συνδέεται η ALU που έχουμε ήδη υλοποιήσει.

```
alu my_alu (
    .op1(op1_extended),
    .op2(op2_extended),
    .alu_op(alu_op),
    .result(alu_out),
    .zero(zero)
);
```

- Το σύστημα έλεγχου της ALU, δηλαδή το calc_enc μοδυλε.

```
calc_enc encoder (
    .btnc(btnc),
    .btnl(btnl),
    .btnr(btnr),
    .alu_op(alu_op)
);
```

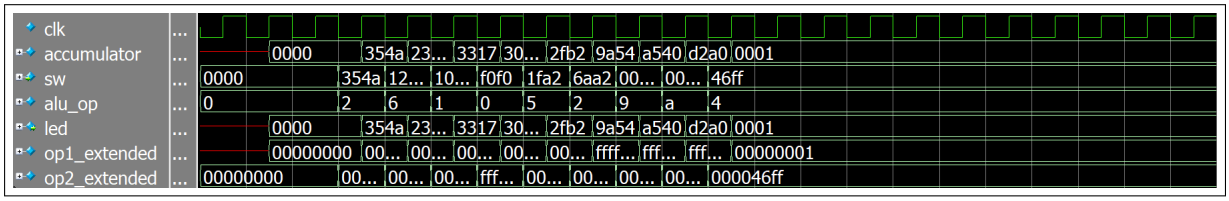


Σχήμα 2.1

Το **calc_tb** έχει έναν πολύ σημαντικό σκοπό. Με την βοήθεια αυτού του **testbench** αρχείου γίνεται ο έλεγχος των παραπάνω υλοποιήσεων. Η εκφώνηση μας ζητάει να πραγματοποιήσουμε κάποια συγκεκριμένα test, εκτελώντας συγκεκριμένες πράξεις με συγκεκριμένες τιμές ώστε να μπορέσουμε να συγκρίνουμε αργότερα τα αποτελέσματα με αυτά που μας έχουν δοθεί από την εκφώνηση. Τα αποτελέσματα αυτών των test είναι:

```
# Reset: LED = 0000 (Expected: 0x0)
# ADD: LED = 354a (Expected: 0x354A)
# SUB: LED = 2316 (Expected: 0x2316)
# OR: LED = 3317 (Expected: 0x3317)
# AND: LED = 3010 (Expected: 0x3010)
# XOR: LED = 2fb2 (Expected: 0x2FB2)
# ADD: LED = 9a54 (Expected: 0x9A54)
# LogicalShiftLeft: LED = a540 (Expected: 0xA540)
# ShiftRight Arithmetic: LED = d2a0 (Expected: 0xD2A0)
# LessThan: LED = 0001 (Expected: 0x0001)
```

Ενώ οι κυματομορφες που παράγονται από την προσομοίωση εμφανίζονται παρακάτω.



Σχήμα 2.2

3 Τρίτη Άσκηση

Σε αυτή την άσκηση της εργασίας, πρέπει να υλοποιήσουμε ένα **αρχείο καταχωρητών**, το οποίο θα είναι υπεύθυνο για την αποθήκευση των τιμών των καταχωρητών που θα χρησιμοποιούνται από τον **RISC-V** επεξεργαστή μας.

Στην την υλοποίηση του αυτού του module, η πρώτη διεργασία είναι η αρχικοποίηση των καταχωρητών με την τιμή μηδέν, εξασφαλίζοντας μια γνωστή αρχική κατάσταση. Η λειτουργία εγγραφής συγχρονίζεται με το σήμα ρολογιού και τα δεδομένα εγγράφονται στον καθορισμένο καταχωρητή εάν το σήμα εγγραφής είναι ενεργό και η διεύθυνση του καταχωρητή δεν είναι μηδέν. Επίσης, στην υλοποίηση χρησιμοποιήθηκαν **Non-Blocking** εντολές, αποτρέποντας την εσφαλμένη χρίση των καταχωρητών με την καταγραφή λανθασμένων τιμών στην περίπτωση που χρησιμοποιηθεί η ίδια διεύθυνση για εγγραφή και ανάγνωση.

4 Τέταρτη Άσκηση

Στόχος της άσκησης 4 είναι να σχεδιάσουμε ένα **datapath** module, μια μονάδα διαδρομής δεδομένων. Η μονάδα αυτί, μαζί με το module της επόμενης άσκησης, καθώς και τη μνήμη εντολών και δεδομένων που μας δίνονται έτοιμες, θα ολοκληρώσουν την λειτουργιά του **RISC-V** επεξεργαστή που θέλουμε να υλοποιήσουμε.

Η άσκηση αυτή, χωρίζεται σε πέντε τμήματα όπως θα δούμε και στο παρακάτω διάγραμμα. Πιο συγκεκριμένα τα τμήματα αυτά είναι:

- Το Program Counter (PC)

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

- Το Register File που θα πρέπει να αρχικοποιήσουμε.

```
regfile my_regfile (
    .clk(clk),
    .write(RegWrite),
    .readReg1(instr[19:15]),
    .readReg2(instr[24:20]),
    .writeReg(instr[11:7]),
    .writeData(write_back_data),
    .readData1(alu_op1),
    .readData2(alu_op2)
);
```

- Immediate Generation.

```
assign input_i_addi = instr[31:20];
assign output_i_addi = {{20{input_i_addi[11]}}, input_i_addi};
assign input_i_sw = {instr[31:25], instr[11:7]};
assign output_i_sw = {{20{input_i_sw[11]}}, input_i_sw};
assign input_i_beq = {instr[31], instr[7], instr[30:25],
   instr[11:8]};
assign output_i_beq = {{19{input_i_beq[11]}}, input_i_beq,
   1'b0};
```

- Το ALU που έχουμε υλοποιήσει στη πρώτη άσκηση και θα πρέπει να αρχικοποιήσουμε.

```
alu my_alu (
        .op1(alu_op1),
    .op2(mux_result_op2),
    .alu_op(ALUCtrl),
    .result(alu_result),
        .zero(Zero)
);
```

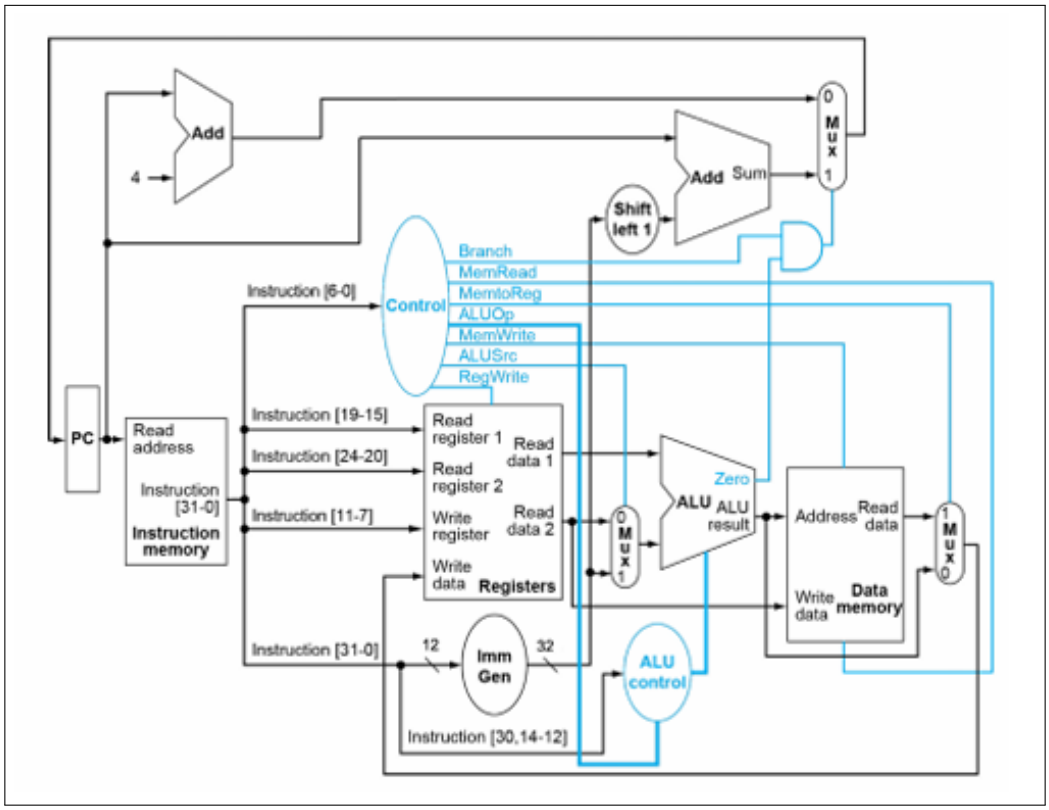
- Το Branch Target.

```
assign left_add = output_i_beq << 1;
assign sum_pc_i = left_add + PC;
```

- Write Back.

```
assign write_back_data = MemToReg ? dReadData : alu_result;
```

Επιπλέον, αξίζει να σημειωθεί πως σε αρκετές περιπτώσεις, για την δημιουργία πολυπλεκτών, χρησιμοποιήσαμε εντολές assign αντί ενός always block. Αυτό έγινε για λόγους απλοποίησης, αλλά και για την αποτροπή σφαλμάτων λόγω καθυστερήσεων με την χρήση always block.



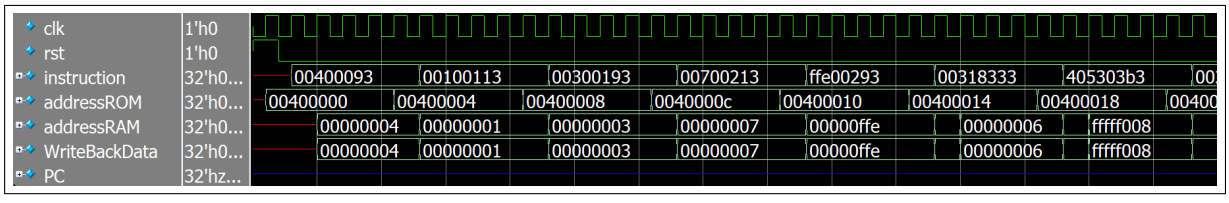
Σχήμα 4.1

5 Πέμπτη Άσκηση

Στην πέμπτη και τελευταία άσκηση, θα υλοποιήσουμε το τελευταίο στοιχείο του επεξεργαστή μας, την μονάδα έλεγχου, καθώς και θα υλοποιήσουμε ένα τεστβενση για τον έλεγχο των λειτουργιών του. Θα δημιουργήσουμε δυο αρχεία, το **top_proc.v** και το **top_proc_tb.v**.

Για το αρχειο **top_proc.v**, έπρεπε και πάλι να υλοποιήσουμε 7 τμήματα. Το πιο σημαντικό από αυτά όμως ήταν η υλοποίηση ενός **FSM (Finite State Machine)**, για το οποίο έπρεπε να υλοποιήσουμε 3 Procedural Blocks. Την Αποθήκευση κατάστασης, την επόμενη κατάσταση και τέλος την κατάσταση εξόδου. Τα 3 αυτά block υλοποιήθηκαν χρησιμοποιώντας διάφορα always block, δημιουργώντας ουσιαστικά διάφορους πολυπλέκτες για κάθε μια από τις πιθανές εντολές της εισόδου **instr**.

Στο **testbench** μας, συνδέουμε τα αρχεία **rom.v** και **ram.v** τα οποία μας έχουν δοθεί και δημιουργούμε ένα ρολόι για την εκτέλεση των απαραίτητων προσομοιώσεων. Στο αρχείο **rom_bytes.data** βρίσκονται οι εντολές **32-bit** που θα πρέπει να εκτελέσει ο επεξεργαστής για να επαληθεύει η ορθή λειτουργία του.



Σχήμα 5.1

Οι παραπάνω κυματομορφες αποτελούν ένα μικρό κομμάτι των κυματομορφών που παράχθηκαν από την προσομοίωση για λόγους διευκόλυνσης.