Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής 1

Εργασία 7ου εξαμήνου (2023-2024)

Τσαράβα Χριστίνα

ΑΕΜ: 10592

ctsarava@ece.auth.gr

Για την εργασία χρησιμοποίησα την πλατφόρμα EDA Playground.

Σε όλες τις ασκήσεις ακολούθησα κατά γράμμα και με τη σειρά όλες τις οδηγίες που μας δόθηκαν, γι'αυτό παρακάτω θα αναφέρω περιληπτικά απλώς κάποια σημεία στους κώδικές μου. Έχω προσθέσει, επίσης, σχόλια σε όλους τους κώδικές μου για την ευκολότερη κατανόησή τους.

Άσκηση 1: Σχεδιασμός και υλοποίηση 32-bit ALU

Στην Άσκηση 1, ο στόχος ήταν η κατασκευή μιας Αριθμητικής Λογικής Μονάδας 32-bit (ALU) για ένα σύστημα επεξεργαστή RISC-V, ικανό να εκτελεί διάφορες αριθμητικές και λογικές πράξεις.

alu.v

Η ALU που δημιούργησα υποστηρίζει προσημασμένη πρόσθεση, προσημασμένη αφαίρεση, λογική AND, λογική OR, λογική XOR, Less Than σύγκριση και τρεις διαφορετικές λειτουργίες ολίσθησης(shift). Κάθε λειτουργία προσδιορίζεται μοναδικά από έναν κωδικό 4-bit, διευκολύνοντας την πολυπλεξία των λειτουργιών.

Η ALU εκτελεί τις παραπάνω λειτουργίες με βάση το σήμα εισόδου alu_op. Γι'αυτό επέλεξα να χρησιμοποιήσω μια "case" δήλωση για την επιλογή της κατάλληλης λειτουργίας με βάση την είσοδο alu_op.

Επίσης, επέλεξα ένα block always @* , το οποίο χρησιμοποιείται για τη δημιουργία ενός συνδυαστικού κυκλώματος. Αυτή η επιλογή σχεδίασης διασφαλίζει ότι η έξοδος της ALU εξαρτάται καθαρά από τις εισόδους op1, op2, alu_op, χωρίς την ανάγκη clock λογικής.

Άσκηση 2: Σχεδιασμός κυκλώματος αριθμομηχανής με χρήση της ALU

Η άσκηση 2 περιλάμβανε το σχεδιασμό και την υλοποίηση ενός κυκλώματος αριθμομηχανής που χρησιμοποιεί την Αριθμητική Λογική Μονάδα 32-bit (ALU) από την Άσκηση 1. Τα κύρια στοιχεία του κυκλώματος είναι ένας συσσωρευτής 16-bit για την αποθήκευση της τρέχουσας τιμής και η προηγουμένως σχεδιασμένη ALU.

calc.v

Χρησιμοποίησα τον τελεστή Concatenation της Verilog ώστε να δημιουργήσω ένα σήμα 32 bit, op1_ex, που είναι εκτεταμένη έκδοση του συσσωρευτή 16 bit και συνδέεται με την είσοδο 'op1' της ALU και ένα άλλο σήμα 32-bit, το op2_ex, που είναι εκτεταμένη έκδοση των εισόδων του sw 16-bit και συνδέεται με την είσοδο 'op2' της ALU.

Δημιούργησα ένα σήμα 4-bit, το alu_bit, που καθορίζεται από τις τιμές των: btnl, btnc και btnr. Αυτό το σήμα καθορίζει τη λειτουργία ALU που θα εκτελεστεί. Το σήμα alu_bit παράγεται από τον αποκωδικοποιητή που υλοποίησα (decoder.v), που συνδέεται με την είσοδο 'alu_op' του ALU.

Δημιούργησα τον συσσωρευτή (accumulator) ώστε να μηδενίζεται ταυτόχρονα με το πάτημα του πλήκτρου 'Up' (btnu) και να ενημερώνεται με τα χαμηλότερα 16 bit του αποτελέσματος ALU όταν πατηθεί το πλήκτρο 'Down' (btnd).

Επίσης σύνδεσα τις εξόδους LED στα 16 λιγότερο σημαντικά bit του συσσωρευτή, τα οποία παρέχουν μια οπτική αναπαράσταση της τρέχουσας τιμής της αριθμομηχανής.

Ιδιαίτερη προσοχή δόθηκε στην απόφαση χρήσης blocking ή non-blocking αναθέσεων.

Οι blocking αναθέσεις (=) χρησιμοποιούνται όταν θέλουμε να εκχωρήσουμε τιμές ταυτόχρονα στο procedural block. Στον κώδικά μου, χρησιμοποίησα = στο always block που είναι υπεύθυνο για την ενημέρωση του συσσωρευτή. Εδώ, θεώρησα το = κατάλληλο επειδή οι αναθέσεις βασίζονται στην κατάσταση των σημάτων btnu και btnd και προορίζονται να εκτελεστούν ταυτόχρονα με βάση τα posedge clk ή posedge btnu.

```
// Update the accumulator with the lower 16 bits of the ALU result and
// update the LED with the lower 16 bits of the accumulator

always @ (posedge clk or posedge btnu)
begin
if(btnu) begin
    accumulator = 0;
    led=0;
end
else if(btnd) begin
    accumulator = alu_result[15:0];
    led = accumulator;
end
else begin
    accumulator <= accumulator;
end

end
```

Στο παραπάνω always block, ο accumulator ενημερώνεται με βάση την κατάσταση των σημάτων btnu και btnd. Η χρήση της non-blocking ανάθεσης (<=) διασφαλίζει ότι η ενημέρωση του συσσωρευτή πραγματοποιείται με διαδοχικό τρόπο στη θετική ακμή του ρολογιού (posedge clk). Οι non-blocking αναθέσεις είναι κατάλληλες για τη μοντελοποίηση της διαδοχικής συμπεριφοράς, καθώς αντικατοπτρίζουν τη σειρά με την οποία θα εκτελούνταν οι εντολές σε ένα σχέδιο hardware.

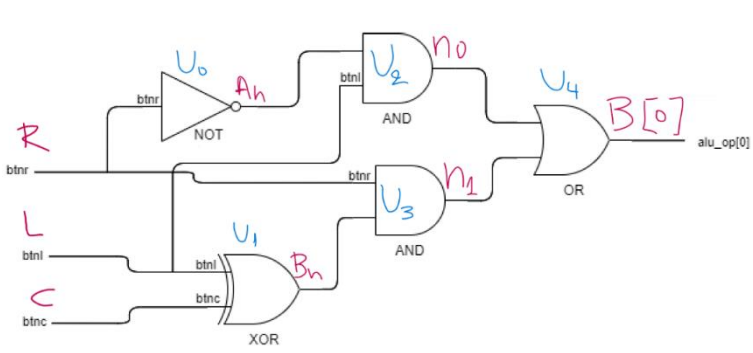
Τα op1_ex και op2_ex προέρχονται από τα σήματα του accumulator και sw, αντίστοιχα. Αυτά τα σήματα χρησιμοποιούνται στη συνέχεια ως είσοδοι στην ALU. Η χρήση non-blocking αναθέσεων διασφαλίζει ότι οι υπολογισμοί των op1_ex και op2_ex βασίζονται στις τιμές του accumulator και sw στον τρέχοντα χρόνο προσομοίωσης χωρίς άμεσο αντίκτυπο στις επόμενες δηλώσεις.

```
always @(accumulator or sw)
begin
    // Concatenation to create sign-extended versions of op1 and op2
    op1_ex <= {{16{accumulator[15]}},accumulator};
    op2_ex <= {{16{sw[15]}},sw};
end
```

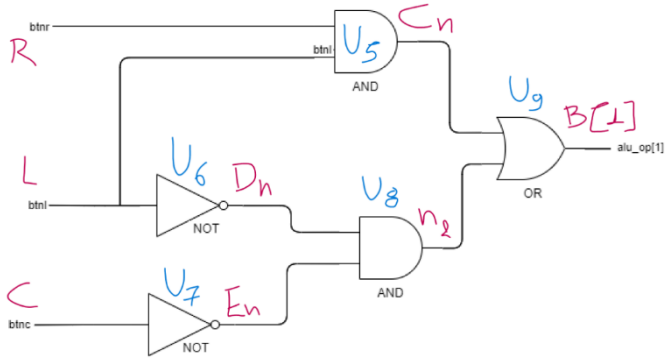
decoder.v

Στο decoder.v πρόσεξα ώστε να χρησιμοποιήσω structural Verilog όπως ζητήθηκε.

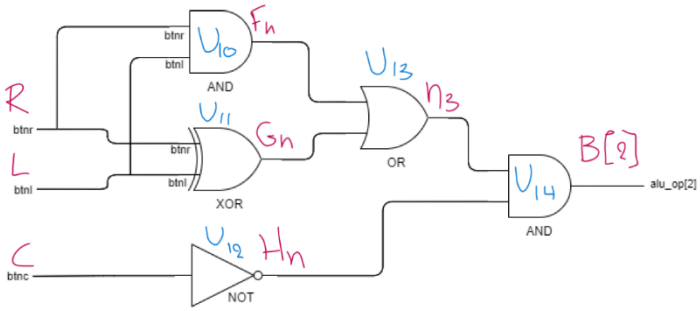
Παρακάτω φαίνονται τα σχήματα 2,3,4,5 που με βοήθησαν να υλοποιήσω το decoder.v:



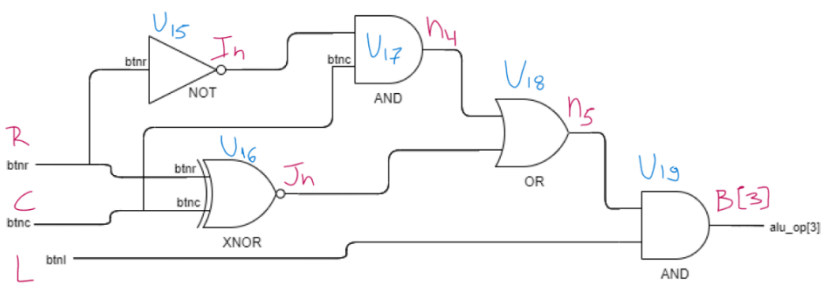
Σχ. 2: Παραγωγή του alu_op[0] μέσω των btnr, btnl, btnc.



Σχ. 3: Παραγωγή του alu_op[1] μέσω των btnr, btnl, btnc.



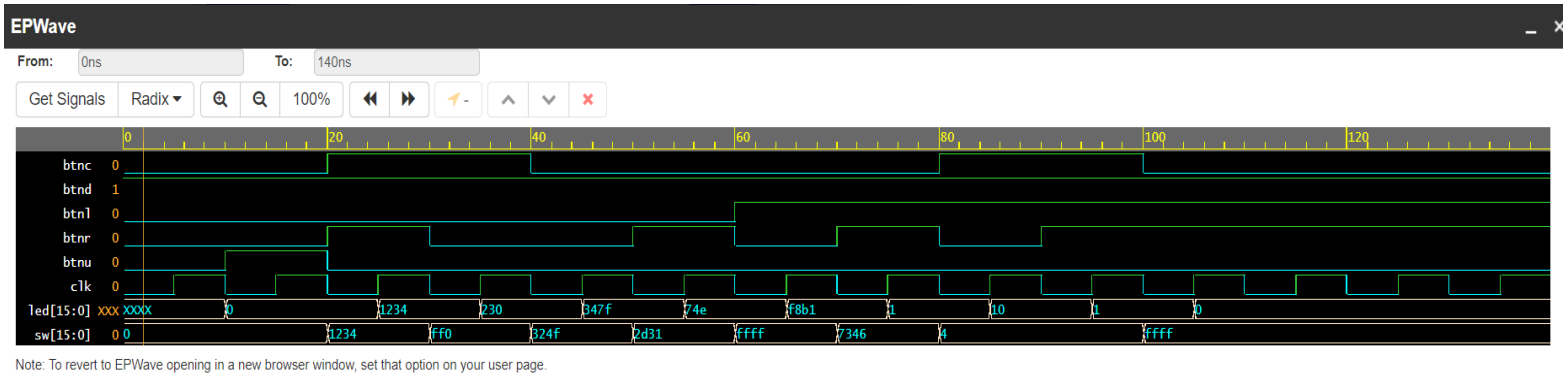
Σχ. 4: Παραγωγή του alu_op[2] μέσω των btnr, btnl, btnc.



Σχ. 5: Παραγωγή του alu_op[3] μέσω των btnr, btnl, btnc.

calc_tb.v

Κάνοντας RUN στο EDA Playground παίρνουμε τις εξής κυματομορφές των σημάτων του testbench:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

led: Αυτό το σήμα αντιπροσωπεύει την έξοδο του κυκλώματος της αριθμομηχανής μου, συγκεκριμένα την τιμή που εμφανίζεται ως αποτέλεσμα. Είναι το σήμα που με ενδιαφέρει για να επαληθεύσω εάν η αριθμομηχανή παράγει τα σωστά αποτελέσματα.

sw: Αυτό το σήμα αντιπροσωπεύει τους διακόπτες εισόδου στην αριθμομηχανή μου. Υποδεικνύει τα δεδομένα που εισάγονται στην αριθμομηχανή.

Παρατηρώντας, λοιπόν, τα σήματα led και sw και τον πίνακα της εκφώνησης της άσκησης , συμπεραίνω ότι το σήμα led παίρνει τις τιμές της στήλης “expected result” όταν το sw παίρνει τις αντίστοιχες τιμές της στήλης «Switches (input)”. Επομένως, η αριθμομηχανή μου λειτουργεί σωστά.

Άσκηση 3: Σχεδιασμός και Υλοποίηση αρχείου καταχωρητών 32x32 bit

regfile.v

Δημιούργησα αυτό το module ώστε να διαθέτει θύρες ανάγνωσης και εγγραφής με έλεγχο ρολογιού και ξεχωριστές διευθύνσεις για ανάγνωση και γραφή. Στο module αρχικοποιώ τους καταχωρητές στο μηδέν και παρέχω λογική ανάγνωσης και εγγραφής στη θετική ακμή του ρολογιού.

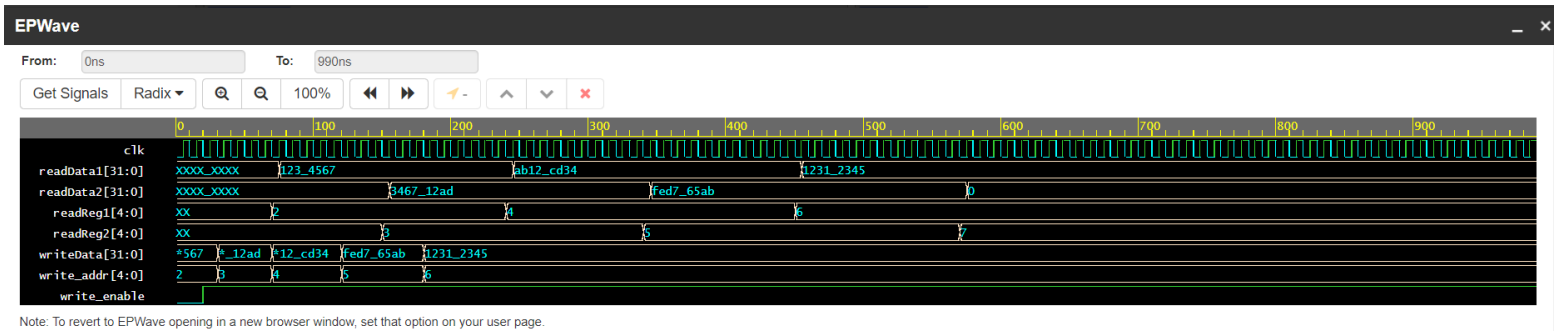
Το always block ενεργοποιείται στη θετική ακμή του ρολογιού. Φρόντισα ώστε να διαβάζει από τους καθορισμένους καταχωρητές για λειτουργίες ανάγνωσης και να γράφει στον καθορισμένο καταχωρητή όταν επιβεβαιώνεται το σήμα ελέγχου εγγραφής.

Ο πιθανός κίνδυνος ταυτόχρονης ανάγνωσης και εγγραφής στην ίδια διεύθυνση αντιμετωπίζεται μέσω της χρήσης μιας non-blocking ανάθεσης (<=) στο always block (registers[writeReg] <= writeData). Αυτό σημαίνει ότι η λειτουργία εγγραφής θα εφαρμοστεί στην επόμενη θετική ακμή του ρολογιού. Αυτό διασφαλίζει ότι ακόμα κι αν η διεύθυνση εγγραφής (writeReg) ταιριάζει με μια από τις διευθύνσεις ανάγνωσης (readReg1 ή readReg2), οι λειτουργίες ανάγνωσης θα εξακολουθούν να χρησιμοποιούν τις τρέχουσες τιμές από τους καταχωρητές πριν τεθεί σε ισχύ η λειτουργία εγγραφής.

regfile_tb.v

Δημιούργησα ένα testbench παρόλο που δεν ζητήθηκε ώστε να ελέγξω την ορθότητα του regfile.v. Έχω ανεβάσει το αρχείο του testbench μαζί με τα υπόλοιπα.

Έτρεξα το testbench στο EDA Playground και τα αποτελέσματα φαίνονται παρακάτω:



Φαίνεται, λοιπόν, ότι το regfile.v έχει υλοποιηθεί σωστά, καθώς βλέπουμε στα σήματα readData1 και readData2 τα δεδομένα που γράψαμε και φαίνονται στο writeData στις αντίστοιχες διευθύνσεις (readReg1, readReg2) που ορίστηκαν από το write_addr. Φαίνεται, ακόμα, ότι όλα αυτά γίνονται τις αναμενόμενες χρονικές στιγμές (σύμφωνα με το testbench).

Άσκηση 4: Σχεδιασμός Datapath

datapath.v

Από το pdf για τον riscv-spec βρήκα τα opcode για τις εντολές διαφορετικού τύπου:

LW → 0000011

SW → 0100011

IMM →0010011

“always @(instr)” Block:

Αυτό το μπλοκ είναι ευαίσθητο σε αλλαγές στο σήμα instr, υποδεικνύοντας ότι θα πρέπει να εκτελείται κάθε φορά που υπάρχει αλλαγή στην είσοδο εντολών.

Το σήμα instr αντιπροσωπεύει τα δεδομένα εντολών από τη μνήμη εντολών. Κάνοντας την decode διαδικασία ευαίσθητη στις αλλαγές σε αυτό το σήμα, διασφαλίζω ότι η λογική αποκωδικοποίησης εντολών ενεργοποιείται κάθε φορά που λαμβάνεται μια νέα εντολή.

1ο “always @(*)” Block:

Οι immediate τιμές (immediateTypeI, immediateStore, branchOffsetEx, κτλ.) υπολογίζονται με βάση σε ποια ομάδα εντολών ανήκουν. Η χρήση non-blocking αναθέσεων (<=) διασφαλίζει ότι αυτές οι τιμές υπολογίζονται διαδοχικά και όχι ταυτόχρονα, κάτι που συμφωνεί με την προβλεπόμενη διαδικαστική ροή.

2ο “always @(*)” Block:

Το MemToReg είναι ένα σήμα ελέγχου που υποδεικνύει εάν τα δεδομένα που θα εγγραφούν πίσω στο αρχείο καταχωρητή πρέπει να προέρχονται από τη μνήμη (dReadData) ή από το αποτέλεσμα της ALU (aluResult).

Εάν MemToReg == 1, επιλέγονται τα δεδομένα από τη μνήμη (dReadData).

Εάν MemToReg == 0, επιλέγεται το αποτέλεσμα από το ALU (aluResult).

“always @(posedge clk)” Block:

Στη θετική ακμή του ρολογιού (posedge clk), ο κώδικας ελέγχει εάν το σήμα επαναφοράς (rst) έχει δηλωθεί. Εάν δηλώνεται rst, ορίζει τον μετρητή προγράμματος (PC) στην αρχική τιμή (INITIAL_PC). Αυτό διασφαλίζει ότι όταν πραγματοποιείται επαναφορά, ο υπολογιστής αρχικοποιείται στην καθορισμένη τιμή.

Εάν δεν υπάρχει επαναφορά ελέγχει εάν έχει δηλωθεί το σήμα loadPC. Εάν δηλωθεί loadPC, ελέγχει το σήμα PCSrc.

Εάν δηλωθεί PCSrc, αυτό σημαίνει ότι ο υπολογιστής πρέπει να ενημερωθεί με μια τιμή branch offset.

Το branch offset προστίθεται στην τρέχουσα τιμή υπολογιστή (PC + branchOffset).

Εάν το PCSrc απενεργοποιηθεί, σημαίνει ότι ο υπολογιστής πρέπει να ενημερωθεί συμβατικά (αυξάνεται κατά 4 για την επόμενη οδηγία).

Η λογική branch offset ευθυγραμμίζεται με την αρχιτεκτονική RISC-V, όπου οι οδηγίες branch τροποποιούν τον υπολογιστή για να ανακατευθύνει τη ροή ελέγχου.

Άσκηση 5: Υλοποίηση multicycle επεξεργαστή

multicycle.v

Η μονάδα multicycle.v αντιπροσωπεύει έναν επεξεργαστή πολλαπλών κύκλων που έχει σχεδιαστεί για να εκτελεί εντολές RISC-V. Περιλαμβάνει μια μηχανή πεπερασμένης κατάστασης (FSM) που συντονίζει τη ροή της εκτέλεσης εντολών μέσω διαφορετικών σταδίων: Ανάκτηση εντολών (IF), Αποκωδικοποίηση εντολών (ID), Εκτέλεση (EX), Πρόσβαση στη μνήμη (MEM) και Εγγραφή (WB). Η μονάδα διασυνδέεται με μια διαδρομή δεδομένων και μονάδες μνήμης για την εκτέλεση εντολών σε πολλαπλούς κύκλους ρολογιού.

Όρισα, αρχικά, παραμέτρους για την αρχική τιμή PC και για τους διαφορετικούς τύπους εντολών. Υλοποίησα μια μηχανή FSM για τον έλεγχο της λειτουργίας του multicycle επεξεργαστή. Όρισα τις καταστάσεις (IF, ID, EX, MEM, WB) με παραμέτρους και την λογική εξόδου με βάση την τρέχουσα κατάσταση του FSM. Διαφορετικές έξοδοι ελέγχονται σε διαφορετικές καταστάσεις. Υλοποίησα την λογική ελέγχου της ALU με βάση το opcode της εντολής και όρισα την λογική της μετάβασης κατάστασης με βάση την τρέχουσα κατάσταση του FSM. Εφάρμοσα την clk λογική για τη μνήμη κατάστασης και τη λογική επόμενης κατάστασης. Αρχικοποίησα μεταβλητές όπως aluSource, aluControl, pcLoad, κ.λπ., πριν χρησιμοποιηθούν. Το μπλοκ OUTPUT_LOGIC ελέγχει διάφορες εξόδους με βάση την τρέχουσα κατάσταση και τον τύπο εντολής.

multicycle_tb

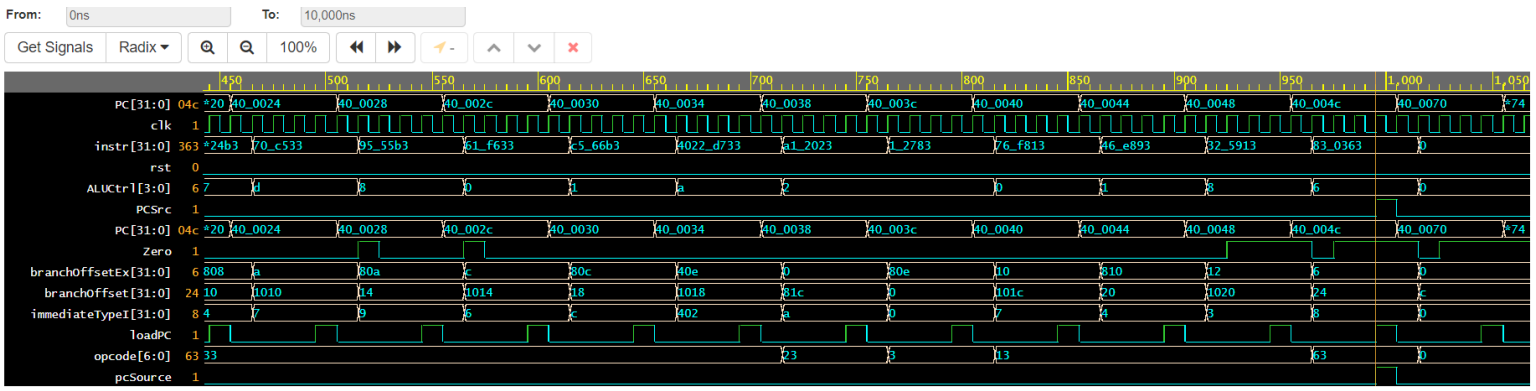
Δημιούργησα ένα testbench για να ελέγξω (μαζί με τη βοήθεια του Σχήματος 7 της εκφώνησης της εργασίας) όλες τις εντολές που υποστηρίζονται.

Παρακάτω δίνω δύο παραδείγματα της διαδικασίας που ακολούθησα:

- Αρχικά, βλέποντας το Σχήμα 7 που υπάρχει στην εκφώνηση της εργασίας βγάζω τα εξής συμπεράσματα:

Έστω ότι θέλω να τσεκάρω το BEQ . Το PC εξαρτάται από το pcSource και από το branchOffset. Το pcSource εξαρτάται από το αν είναι το opcode ίσο με το opcode του BEQ και αν zero=1. Το branchOffset εξαρτάται από το immediate generation του B σήματος, αφού παίρνω το σήμα B και το κάνω shift left κατά 1. Το immediate generation εξαρτάται από το opcode και το opcode εξαρτάται από το instr , αφού το opcode είναι τα 7 LSB του instr.

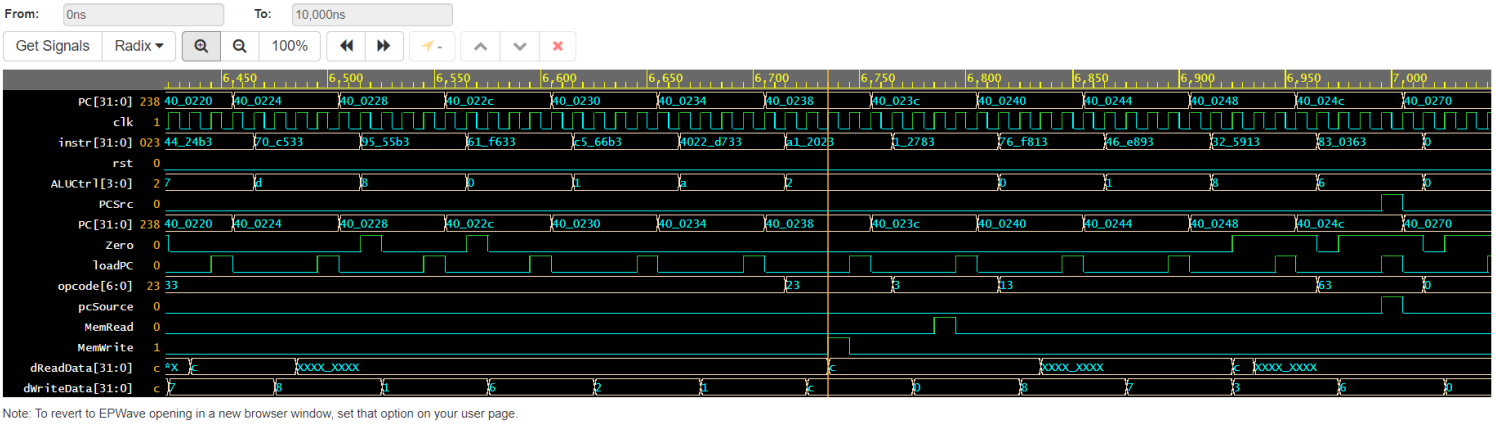
Τρέχοντας το simulation, κάνοντας zoom στο κατάλληλο σημείο και παρατηρώντας τα σήματα opcode, pcSource, Zero, PC για χρόνο 995ns βλέπουμε ότι: το opcode=63 (δεκαεξαδικό) ή αλλιώς opcode=1100011 (δυαδικό). Το opcode, λοιπόν, είναι ίδιο με αυτό των BEQ εντολών, άρα έχουμε branch εντολή, άρα Branch=1 (το μπλε σήμα στο σχήμα). Επίσης zero=1. Άρα η έξοδος της λογικής πύλης AND στο σχήμα θα είναι 1 και όντως από το simulation pcSource=1. Από το σχήμα βλέπουμε ότι όταν ο πολυπλέκτης δέχεται σήμα 1 , το



PC κάνει shift left κατά 1 και δεν του προστίθεται το +4. Πράγματι, από το simulation παρατηρούμε ότι το PC είχε την τιμή 40_0048 και η τιμή του έγινε 40_004c, άρα είχαμε left shift κατά 1.

Αφού, λοιπόν, τα αποτελέσματα της προσομοίωσης επιβεβαιώνουν την λογική του σχήματος του datapath, ο κώδικας μου τρέχει σωστά.

- Στο παρακάτω απόσπασμα της προσομοίωσης παρατηρούμε τα εξής: στα 6,735ns το opcode είναι 23 (δεκαεξαδικό) άρα 0100011 (δυαδικό) , δηλαδή είναι το opcode για SW εντολές (store). Την ίδια χρονική στιγμή το MemWrite = 1 και το dWriteData = c, που σημαίνει ότι γίνεται store στη ram η τιμή c. Μετά από λίγο χρόνο (στα

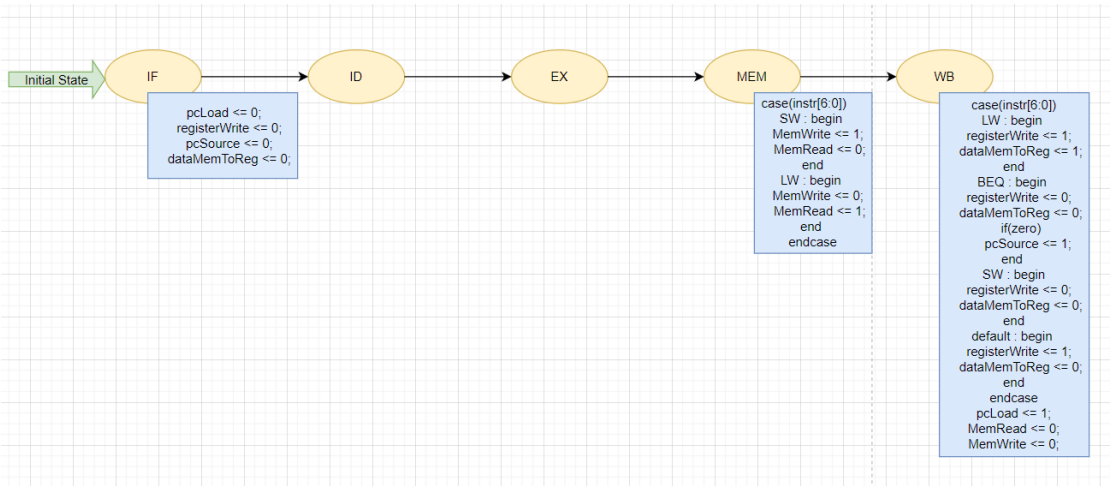


Note: To revert to EPWave opening in a new browser window, set that option on your user page.

6,785ns) το opcode έχει γίνει 3 (δεκαεξαδικό), δηλαδή 0000011 (δυαδικό), είναι επομένως το opcode της εντολής LW (load). Την ίδια στιγμή MemRead = 1 και dReadData = c. Δηλαδή, η τιμή που είχαμε κάνει store στη ram έγινε με επιτυχία loaded. Άρα ο κώδικάς μου έχει την αναμενόμενη συμπεριφορά.

Με τον ίδιο τρόπο σκέψης έλεγξα και τις υπόλοιπες περιπτώσεις εντολών και κατέληξα ότι όλες λειτουργούν με τον σωστό και αναμενόμενο τρόπο.

Σχηματικό διάγραμμα του FSM:



Το FSM που χρησιμοποιείται στην multicycle αρχιτεκτονική σχεδιάστηκε με 5 καταστάσεις:

Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) και Write Back (WB).

Κάθε κατάσταση εξυπηρετεί έναν συγκεκριμένο σκοπό στην εκτέλεση των εντολών.

- IF (Instruction Fetch): Ανάκτηση της επόμενης εντολής από τη μνήμη εντολών. Σε αυτή την κατάσταση, ο μετρητής προγράμματος (PC) φορτώνεται με την τρέχουσα διεύθυνση εντολών και η εντολή ανακτάται από τη μνήμη εντολών.
- ID (Instruction Decode): Αποκωδικοποιεί την ληφθείσα εντολή και καθορίζει τα σήματα ελέγχου.Η εντολή αναλύεται για τη δημιουργία σημάτων ελέγχου που υπαγορεύουν τη συμπεριφορά των επόμενων σταδίων. Το FSM καθορίζει τον τύπο της εντολής (π.χ. load, store, branch, arithmetic) και διαμορφώνει ανάλογα τη διαδρομή δεδομένων.
- EX (Execute): Εκτελεί αριθμητικές ή λογικές πράξεις με βάση την αποκωδικοποιημένη εντολή. Σε αυτήν την κατάσταση, η ALU εκτελεί πράξεις όπως πρόσθεση, αφαίρεση ή λογικές πράξεις. Το αποτέλεσμα χρησιμοποιείται στη συνέχεια για περαιτέρω επεξεργασία ή ως τελεστής για επόμενες εντολές.
- MEM (Memory Access): Χειρίζεται λειτουργίες μνήμης (φόρτωση και αποθήκευση). Για οδηγίες φόρτωσης, τα δεδομένα διαβάζονται από τη μνήμη δεδομένων. Για οδηγίες αποθήκευσης, τα δεδομένα εγγράφονται στη μνήμη δεδομένων. Αυτή η κατάσταση διευκολύνει την αλληλεπίδραση μεταξύ του επεξεργαστή και της μνήμης.
- WB (Write Back): Γράφει το αποτέλεσμα της εκτελεσθείσας εντολής πίσω στους καταχωρητές. Το τελικό αποτέλεσμα της εκτέλεσης της εντολής γράφεται πίσω στο αρχείο μητρώου. Αυτή η κατάσταση είναι ζωτικής σημασίας για τη διατήρηση του αποτελέσματος των λειτουργιών και την ενημέρωση των τιμών του μητρώου.

Επίσης, το FSM μου είναι μηχανή Moore, καθώς οι έξοδοι καθορίζονται αποκλειστικά από την τρέχουσα κατάσταση.

Σημείωση: Επειδή χρησιμοποίησα το EDA Playground στα testbench δεν χρειάστηκε η εντολή “include”. Επίσης, μαζί με τους κώδικες ανέβασα και το regfile_tb παρόλο που δεν ζητήθηκε.