ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ HW-I

ΑΝΑΦΟΡΑ ΕΡΓΑΣΙΑΣ ΕΞΑΜΗΝΟΥ

ΚΩΝΣΤΑΝΤΙΝΟΣ ΒΛΑΧΑΚΟΣ

ΑΕΜ:10403

Περιεχόμενα

Εισαγωγή

Η παρούσα εργασία υλοποιήθηκε στο IDE του Visual Studio Code, ενώ για το compilation και την ανάγνωση των αποτελεσμάτων χρησιμοποιήθηκε το open source λογισμικό Icarus Verilog, σε περιβάλλον bash (Ubuntu Linux).

Τα θέματα που θα αναλυθούν περιλαμβάνουν τη δημιουργία απλών modules με χρήση των τελεστών, τον συνδυασμό αυτών για τη δημιουργία ενός μεγαλύτερου, την υλοποίηση λογικών κυκλωμάτων με structural τρόπο, την προσομοίωση κυκλωμάτων με υλοποίηση και χρήση testbench, την υλοποίηση μηχανών πεπερασμένων καταστάσεων (FSM) και τέλος τον συνδυασμό όλων αυτών με στόχο την δημιουργία ενός (απλουστευμένου μεν πραγματικού δε) πολύκυκλου επεξεργαστή εντολής (χωρίς διοχέτευση) με βάση την αρχιτεκτονική RISC-V.

Μην έχοντας δηλώσει το μάθημα, ευχαριστώ εκ των προτέρων για την αποδοχή στην κοινότητα και στο υλικό του μαθήματος καθώς και για τον χρόνο ανάγνωσης και διόρθωσης της παρούσας εργασίας.

Άσκηση 1

Στην παρούσα άσκηση ζητείται η υλοποίηση της αριθμητικής/λογικής μονάδας βάσει ορισμένων προδιαγραφών όπως τα κατάλληλα operational codes για είσοδο και τον χειρισμό των προσημασμένων περιπτώσεων.

Για τον λόγο αυτό, τα operational codes ορίζονται ως παράμετροι με βάση τις πρώτες 2 στήλες του πίνακα και έπειτα ακολουθεί ο πολυπλέτκης που επιλέγει την πράξη που θα πραγματοποιήσει η ALU.

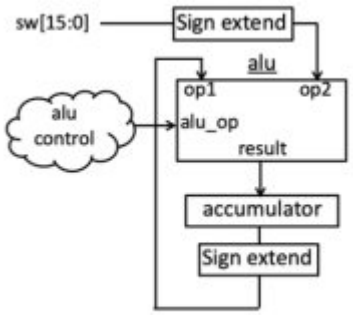
Όντας καθαρά συνδυαστικό κύκλωμα, οι έξοδοι επιλέχθηκαν να παίρνουν τιμή μέσω της εντολής **assign**, ενώ για την περιγραφή του πολυπλέκτη χρησιμοποιήθηκε ο τελεστής **?**.

Για τη διαχείριση των προσημασμένων περιπτώσεων, οι τελεστές της Verilog λειτουργούν ως unsigned by default. Η μόνη περίπτωση που χειρίζονται τους τελεστέους ως signed είναι όταν είναι και οι δύο signed. Συνεπώς, για χρειάζομαι signed και τα 2 σήματα στην εντολή SLT, και signed τον op1 στην εντολή SRA.

Για τον σκοπό αυτό δηλώθηκαν 2 έξτρα wires με το πρόθεμα **signed**, ώστε να χρησιμοποιηθούν στις περιπτώσεις που πρέπει. Μια δεύτερη λύση που βρέθηκε είναι η system function **$signed()** η οποία χειρίζεται τον τελεστέο ως signed ακόμα και αν έχει δηλωθεί unsigned.

Άσκηση 2

Στην παρούσα άσκηση ζητείται η υλοποίηση μιας αριθμομηχανής, η οποία θα δέχεται μια σειρά από switches στον πρώτο τελεστέο, το αποτέλεσμα της προηγούμενης κατάστασής της στον δεύτερο τελεστέο και η πράξη θα ορίζεται μέσα από την αποκωδικοποίηση τριών κουμπιών ελέγχου. Τέλος, θα υπάρχει ένα σύγχρονο κουμπί επαναφοράς της κατάστασης της μηχανής και ένα ασύγχρονο κουμπί που θα επιτρέπει την εμφάνιση και την ενημέρωση του αποτελέσματος. Ακολουθώντας το εξής σχηματικό:



Σχ. 1: Διάγραμμα ροής της αριθμομηχανής

γίνεται το instanciaton της ALU από την πρώτη άσκηση, ορίζονται τα σήματα των τελεστών της ALU και επεκτείνονται κατάλληλα μέσω του τελεστή concatenation.

Έπειτα υλοποιείται η ακολουθιακή λογική του κυκλώματος μέσα σε δύο always blocks όπου στο πρώτο δημιουργείται το ασύγχρονο κουμπί ενημέρωσης του αποτελέσματος, ενώ στο δεύτερο το σύγχρονο κουμπί μηδενισμού. Όντας ακολουθιακά κυκλώματα στα οποία επιθυμώ όλες οι έξοδοι να ενημερώνονται στην άνοδο ορισμένων σημάτων χρησιμοποιήθηκαν non blocking assignments (<=), ενώ για την έξοδο led η οποία προβάλλει για οποιαδήποτε στιγμή το περιεχόμενο του accumulator χρησιμοποιήθηκε blocking assignment (=).

Το decoder για τα σήματα ελέγχου της alu μέσα από τα κουμπιά ελέγχου της μηχανής ζητήθηκε να υλοποιηθεί αυστηρά με structural τρόπο. Αυτό σημαίνει να χρησιμοποιηθούν έτοιμα primitives και να γίνει απλά η σύνδεσή τους σε είσοδο/ έξοδο. Συνεπώς, δεν υπάρχει ανάγκη δήλωσης όλων των ενδιάμεσων σημάτων στα λογικά κυκλώματα και μπορούν να περιγραφούν από απλή λίστα πυλών με μόνη προϋπόθεση να ταιριάζουν τα ονόματα εξόδων/ εισόδων. Για να μην υπάρξει μπέρδεμα στα ενδιάμεσα σήματα, ονοματίστηκαν στο χαρτί πάνω στο λογικό κύκλωμα όλες οι ενδιάμεσες έξοδοι πριν υλοποιηθεί η λίστα των πυλών μέσα στο block. Στη συνέχεια, δημιουργείται ένα instance αυτού μέσα στο calc, με τα σήματα εισόδου να ανατίθενται κατάλληλα.

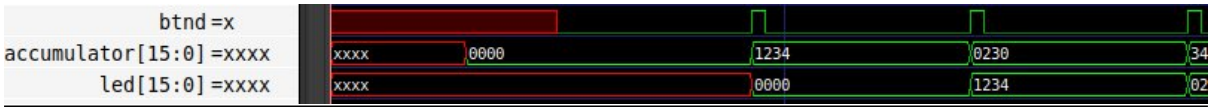
Τέλος, το κύκλωμα δοκιμάζεται στο testbench, του οποίου ο έλεγχος γίνεται από την παρατήρηση των γραφικών παραστάσεων. Οι χρόνοι καθυστέρησης στο testbench προέκυψαν από δοκιμές, των οποίων τα συμπεράσματα θα αναλυθούν καθώς το αποτέλεσμα τους κρίνεται γραφικά. Δεν έγινε κάποια συνάρτηση ή κάποιο task που να χρονίζει επ' ακριβώς και με κανόνα το testbench, συνεπώς η τεκμηρίωση αυτού του επιπέδου της άσκησης ενδέχεται να μην είναι πλήρης.

Για το testbench υλοποιήθηκε ένα vector 3bit τύπου reg, το οποίο μπαίνει ως είσοδος στο instance του calc, για τον καλύτερο έλεγχο των κουμπιών. Στη συνέχεια δημιουργούνται παλμοσειρές ρολογιού (για να αποδειχθεί πως είναι σύγχρονο στο κουμπί reset (btnu)) καθώς και btnd, το οποίο οφείλει να ενεργοποιείται αυστηρά μία φορά σε κάθε έλεγχο του πίνακα του testbench. Η ασυγχρόνιστη ενεργοποίησή του ασυσχέτιστα με τις μεταβάσεις των button_state και switches οδηγεί στην μη επιθυμητή ανάδραση του αποτελέσματος με τον εαυτό του, καταστρέφοντας τα επιθυμητά αποτελέσματα. Το συμπέρασμα αυτό παρατηρήθηκε με δοκιμές. Τέλος, το initial block, στο οποίο ανα αρκετά μεγαλύτερες χρονικές περιόδους μεταβαίνω από τον έναν έλεγχο στον επόμενο

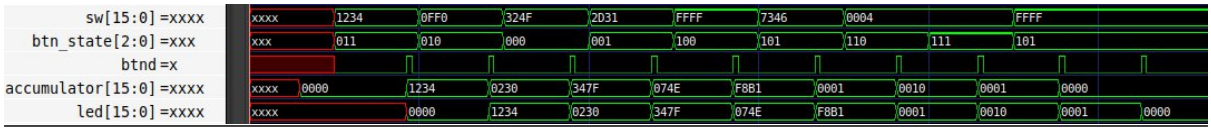
χρησιμοποιώντας τα δεδομένα inputs. Τα αποτελέσματα των γραφημάτων παρατίθενται και στις εξής εικόνες:

Σύγχρονος μηδενισμός του accumulator με το πάτημα του btnu

Ενημέρωση του accumulaltor και των εξόδων LED με το πάτημα του btnd



Παράθεση όλων των δοκιμών και επαλήθευσή τους με τα expected results



(Παρατηρείται ότι ο χρονισμός του btnd δεν είναι τέλειος, ωστόσο τηρείται η βασική προυπόθεση για ένα πάτημα ανά στάθμη (sw,btn_state) που εγγυάται το σωστό αποτέλεσμα. Για τη διευκόλυνση της πραγματοποίησης του, ο παλμός btnd έγινε αισθητά στενότερος).

Άσκηση 3

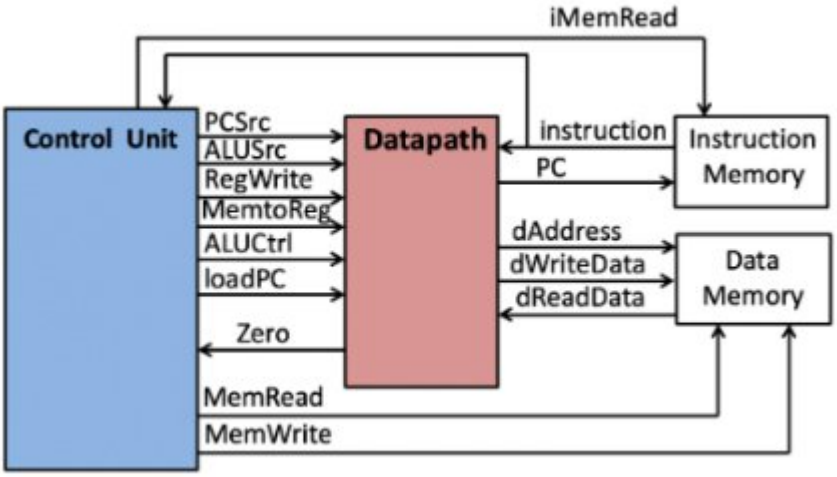
Στο αρχείο καταχωρητών μέσω ενός initial block και μιας for μηδενίζονται οι 32 καταχωρητές που είναι δηλωμένοι. Έπειτα μέσω ενός always block το οποίο λειτουργεί με το ρολόι, εκτελούνται στην ίδια ακμή η γραφή (εφ ‘όσον το σήμα write είναι ψηλά) και η ανάγνωση δεδομένων προς/από διάφορους καταχωρητές.

Το πρόβλημα που προκύπτει είναι στην περίπτωση που η διεύθυνση εγγραφής συμπέσει με τη διεύθυνση ανάγνωσης. Αν για κάποιο λόγο γίνει αυτό (διοχετευμένες εντολές, deadlocks του κώδικα κλπ) τότε θα πρέπει με κάποιο τρόπο να δοθεί προτεραιότητα στην ανάγνωση έναντι της εγγραφής (έστω και με υστέρηση ενός κύκλου ρολογιού), ειδάλλως θα υπάρξει απώλεια πληροφορίας. (Η πληροφορία που υπάρχει ήδη και πρέπει να εγγραφεί θα κάνει overwrite την πληροφορία που δεν υπάρχει ακόμα και πρέπει να διαβαστεί).

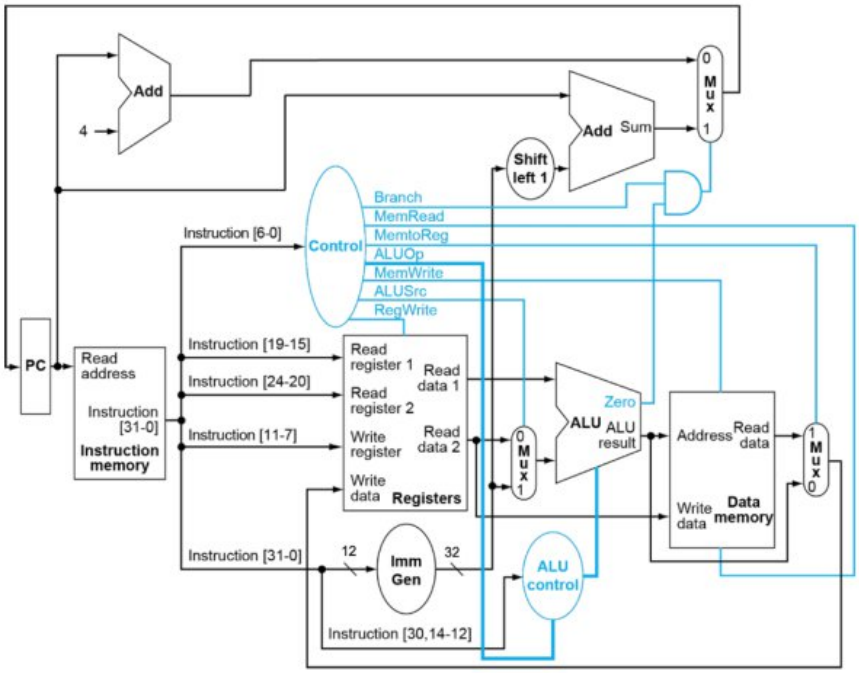
Προκειμένου να υλοποιηθεί αυτή η κατάσταση μέσα στο always block χρησιμοποιώ non blocking assignment (<=) για την εγγραφή και blocking assignment για την ανάγνωση (=). Έτσι, στην περίπτωση της ταυτόχρονης εγγραφής και ανάγνωσης, ο καταχωρητής θα ενημερωθεί με την ολοκλήρωση του block ωστόσο η νέα του τιμή θα εμφανιστεί ως readData στην επόμενη ακμή του ρολογιού. Με αυτόν τον τρόπο, τη δεδομένη στιγμή που έχω ταυτόχρονη εγγραφή και ανάγνωση δίνω προτεραιότητα στην ανάγνωση διαβάζοντας “τα παλιά” και γράφοντας ταυτόχρονα τα καινούργια.

Άσκηση 4

Για την υλοποίηση του datapath το οποίο φαίνεται στο διάγραμμα υψηλού επιπέδου του εξής σχήματος:



αξιοποιήθηκε το αναλυτικό λογικό σχέδιο της άσκησης (απλουστευμένο σε σχέση με το πιο ρεαλιστικό από άλλα βιβλία) καθώς και το πρότυπο του RISC-V για την αποκωδικοποίηση των εντολών και τη δημιουργία ορισμένων blocks. Από το λογικό σχέδιο του datapath εξάγονται τα εξής:



Καθώς βρίσκεται σε επίπεδο ενός κύκλου ρολογιού και ενός στατικού instruction (λαμβάνοντας υπόψη σαν black boxes το instruction memmory, data memory, register file) μπορεί να χρησιμοποιηθεί συνδυαστική λογική για να περιγράψει τις μαύρες διαδρομές του datapath δηλώνοντας wires και συνδέοντας τα παράλληλα με το instruction το οποίο θα έρχεται από τη μνήμη εντολών. Έτσι, η μετάφραση της εντολής δε γίνεται ακολουθιακά (ανάλογα με το opcode να πάει προς μια συγκεκριμένη κατεύθυνση) αλλά μεταφράζεται προς όλα τα σήματα παράλληλα, και τα σήματα ελέγχου είναι εκείνα που καθορίζουν είτε σε επίπεδο κατάστασης (επόμενο στάδιο) είτε σε επίπεδο εντολής το ποια μονοπάτια θα ανοίξουν και ποια όχι. Συνεπώς ακολουθιακή λογική χρησιμοποιήθηκε μόνο για τον

προσδιορισμό και το incrementation του PC μέσα σε ένα απλό always block λαμβάνοντας υπόψη το σύγχρονο reset και την παράμετρο INITIAL_PC.

Έπειτα ακολουθεί το instanciation του αρχείου καταχωρητών όπου με βάση την αποδόμηση της εντολής και το λογικό διάγραμμα, παράλληλα wires από την εντολή συνδέονται με απευθείας αναθέσεις στο instance του regfile. Τον ρόλο του σήματος write έχει σε αυτό το επίπεδο το σήμα RegWrite.

Στη συνέχεια επιχειρείται να φτιαχτεί το block του immediate generator, από το οποίο περνάνε όλες οι εντολές που δεν είναι τύπου R. Παρατηρώντας το πρότυπο των εντολών του RISC-V, γίνεται αντιληπτό ότι υπάρχει ίδιο mapping για τις εντολές IMMEDIATE και LOAD (με εξαίρεση τις εντολές SRLI, SLLI, SRAI) και διαφορειτκά mappings για τις εντολές STORE και BRANCH. Επιπρόσθετα, οι εντολές SRLI και SRAI έχουν κοινό funct3 πέρα από κοινό opcode, ωστόσο χρήζουν διαφορετικής αντιμετώπισης όσον αφορά το sign extension καθώς στην SRAI επιθυμώ ο όρος του immediate shifting να είναι unsigned (από την ALU). Έτσι, χρειάζεται να προστεθούν μηδενικά αντί να γίνει το extend. Για όλες αυτές τις περιπτώσεις, υλοποιείται ο πολυπλέτκης sign_extended_imm ενώ η λογική του ξεκινάει από τις πιο εξειδικευμένες περιπτώσεις προς τις πιο απλές ώστε να αποφευχθεί σε κάθε περίπτωση να γίνει με λάθος τρόπο η αναγνώριση του immediate value.

Έπειτα από το σχηματικό γίνεται η αρχικοποίηση και η σύνδεση της ALU, ενώ δημιουργείται και ο πολυπλέκτης με σήμα ελέγχου το ALUSrc

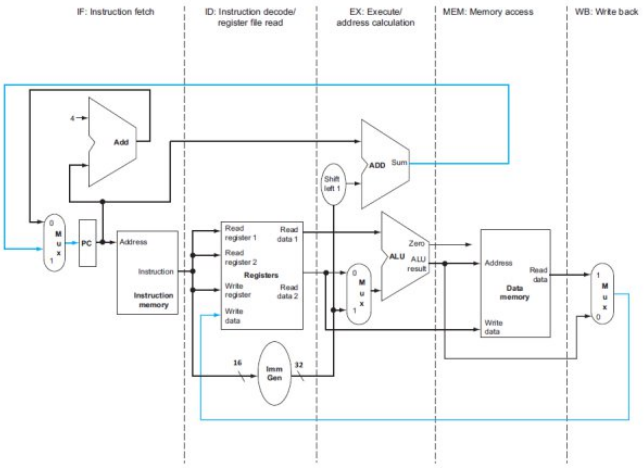
Τέλος υλοποιείται το shifting σε περίπτωση branch, για τη δημιουργία του branch offset που θα προστεθεί στον PC καθώς και ο πολυπλέκτης με σήμα ελέγχου το MemToReg για επικοινωνία και σύνδεση με την μνήμη δεδομένων

Η ανάθεση των υπολοίπων σημάτων ελέγχου (μπλε μονοπάτια) γίνεται στο πολύκυκλο module.

Άσκηση 5

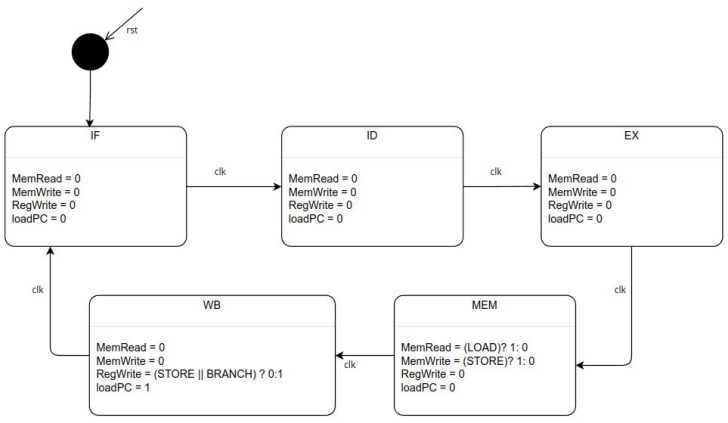
Στην παρούσα άσκηση ζητείται η υλοποίηση μιας μονάδας πολλαπλών κύκλων που θα εκτελεί μια εντολή ανά πέντε κύκλους, αξιοποιόντας το "στατικό" datapath του ενός κύκλου της προηγούμενης άσκησης. Για τη διαδικασία αυτή, απαιτείται η υλοποίηση ενός FSM καθώς και η ολοκλήρωση των αναθέσεων των σημάτων ελέγχου.

Όσο αφορά το FSM από το συγκεκριμένο σχήμα μπορούν να διακριθούν 5 καταστάσεις (IF,ID,EX,MEM,WB):



Όσο αφορά τη λογική μετάβασης από τη μια κατάσταση στην άλλη καθώς και τις τιμές εξόδου, ύστερα από πολύ προβληματισμό προέκυψαν οι εξής σκέψεις: κάθε εντολή που πρέπει να υλοποιηθεί θα περάσει αναγκαστικά και από τα πέντε στάδια με τη συγκεκριμένη διάκριση και σειρά των καταστάσεων (αν μια εντολή έρθει IF, θα πρέπει να αποκωδικοποιηθεί ID, να εκτελεστεί EX, να επιχειρήσει στη μνήμη MEM, να εγγράψει στους καταχωρητές και να φορτώσει νέα εντολή WB -> IF). τα σήματα ελέγχου που απομένουν να υλοποιηθούν δεν είναι όλα μεταβαλλόμενα ανάλογα με την κατάσταση της μηχανής (πχ το ALUCtrl εφ' οσον είναι γνωστό ποια εντολή είναι μέσα στη μηχανή μπορεί να εξαχθεί συνδυαστικά μέσα από το funct3 μέρος του, ανεξάρτητα από την κατάσταση που βρίσκεται).

Έτσι, αποκωδικοποιώντας το περιεχόμενο του κειμένου για τα σήματα ελέγχου, αποφασίστηκε να οριστούν ως έξοδοι του FSM μόνο τα σήματα που είναι μεταβαλλόμενα ανάλογα με την κατάσταση (loadPC,MemRead,MemWrite,RegWrite) ενώ τα υπόλοιπα (ALUCtrl, MemToReg,ALUSrc,PCSrc) να υλοποιηθούν συνδυαστικά, έξω από τη λογική και τα blocks του FSM με συνεχείς αναθέσεις **assign** σε wires. Το σχηματικό διάγραμμα του FSM που προκύπτει με βάση όλες τις πάνω παραδοχές είναι:



Το συγκεκριμένο FSM εφ’ όσον η είσοδος (instruction) έχει απευθείας ρόλο στην έξοδο (στα στάδια που ελέγχω το opcode) και η έξοδος δεν εξαρτάται μόνο από την προηγούμενη κατάσταση θα μπορούσε να χαρακτηριστεί τύπου Mealy.

Με βάση όλα τα παραπάνω η υλοποίηση του πραγματοποιείται σε 3 always blocks, ένα για την αποθήκευση κατάστασης το οποίο είναι ευαίσθητο στο ρολόι και ορίζεται και το σήμα του reset, ένα για την λογική της επόμενης κατάστασης (όπου στην περίπτωση αυτή η επόμενη κατάσταση προσδιορίζεται απευθείας από τον κύκλο) και ένα για την λογική της εξόδου η οποία ακολουθεί το διάγραμμα και το κείμενο περιγραφής των σημάτων.

Το σήμα ALUCtrl υλοποιείται από έναν μεγάλο πολυπλέκτη ο οποίος εξετάζει τα πεδία funct3, funct7 και opcode της εντολής και λειτουργώντας με παρόμοια λογική με το immediate generator της άσκησης 4 (από τα πιο εξειδικευμένα στα πιο γενικά) δημιουργεί το σήμα ALUCtrl το οποίο εισάγεται στο alu_op της ALU. Ειδικότερα, καθώς οι funct3 είναι αυτές που έχουν μοναδικό κωδικό για την πράξη και είναι κοινές για τις τύπου I και τύπου R (κατά κανόνα), το funct7 χρησιμοποιείται περισσότερο για να κάνει το διαχωρισμό απ’ ότι το opcode. Επίσης καθώς οι εντολές LOAD και STORE έχουν ίδιο funct3 με την SLT θα πρέπει να ελεγχθεί το opcode τους σε προγενέστερο στάδιο, ώστε να μην μπερδευτούν ως SLT. Το opcode χρησιμοποιείται επίσης και στη BEQ, τη μοναδική BRANCH εντολή.

Τέλος γίνονται οι αναθέσεις των MemToReg,ALUSrc,PCSrc με βάση το κείμενο και τη λογική του διαγράμματος ελέγχου (μπλε διαδρομές).

Για το testbench χρειάζεται να αξιοποιηθεί η μονάδα πολλαπλών κύκλων ενώ δίνονται έτοιμες η μνήμη εντολών και η μνήμη δεδομένων. Από την ανάγνωση των συγκεκριμένων modules παρατηρώ ότι:

Η μνήμη εντολών αποτελείται από ένα αρχείο .data και για να δημιουργήσει μια εντολή 32 bits κάνει concatenation 4 γραμμές από το αρχείο .data. Αντίστοιχα η μνήμη δεδομένων αποτελείται από 512 θέσεις 32bit καταχωρητών οι οποίοι δεν είναι μηδενισμένοι. Από το σχηματικό διάγραμμα της μονάδας ενώνονται τα instances των instruction memory και data memory και εφ’ όσον οι εντολές είναι ήδη φορτωμένες στο instance, αρκεί ένα reset στη μονάδα και μια παλμοσειρά ρολογιού για να ξεκινήσουν να πέφτουν οι εντολές.

Για τον καλύτερο έλεγχο και debugging, αξιοποιώντας τις γνώσεις από τα Λειτουργικά Συστήματα και την Οργάνωση Υπολογιστών γράφτηκε ένα πρόγραμμα memory_decoder στη γλώσσα C το οποίο όταν καλείται από τη γραμμή εντολών με όρισμα μια 32 bit εντολή κάνει το decoding και τυπώνει σε MIPS την εντολή στην οποία αντιστοιχεί το διάνυσμα. Στη συνέχεια γράφτηκαν 2 bash scripts ένα το οποίο ενώνει τις 4αδες γραμμών για να φτιάξει 32 bit εντολές και ένα το οποίο διαβάζει από αρχείο 32 bit εντολές και αποδίδει όλον τον κώδικα MIPS μαζί με τις τιμές του PC μέσα στη μνήμη. Το tarball μπορεί να βρεθεί και να δοκιμαστεί εδώ: https://drive.google.com/file/d/1aupn-y-FPJE78nNckuJO7G7eWPhFYvMy/view?usp=drive_link

Με βάση όλα τα παραπάνω, από την έξοδο του κώδικα assembly έχω:

```
PC(0x0000) ADDI $1, $0, 4
PC(0x0004) ADDI $2, $0, 1
PC(0x0008) ADDI $3, $0, 3
PC(0x000C) ADDI $4, $0, 7
PC(0x0010) ADDI $5, $0, -2
PC(0x0014) ADD $6, $3, $3
PC(0x0018) SUB $7, $6, $5
PC(0x001C) SLL $8, $3, $2
PC(0x0020) SLT $9, $8, $4
PC(0x0024) XOR $10, $1, $7
PC(0x0028) SRL $11, $10, $9
PC(0x002C) AND $12, $3, $6
PC(0x0030) OR $13, $10, $12
PC(0x0034) SRA $14, $5, $2
PC(0x0038) SW $10, 0($2)
PC(0x003C) LW $15, 0($2)
PC(0x0040) ANDI $16, $13, 7
PC(0x0044) ORI $17, $13, 4
PC(0x0048) SRLI $18, $4, 3
PC(0x004C) BEQ $6, $8, 6
PC(0x0050)
PC(0x0054)
PC(0x0058)
PC(0x005C)
PC(0x0060)
PC(0x0064)
PC(0x0068)
PC(0x006C)
PC(0x0070)
PC(0x0074)
PC(0x0078)
PC(0x007C) XORI $19, $11, 12
PC(0x0080) SLLI $20, $19, 1
PC(0x0084) SRAI $21, $5, 2
PC(0x0088) SLTI $22, $20, 28
```

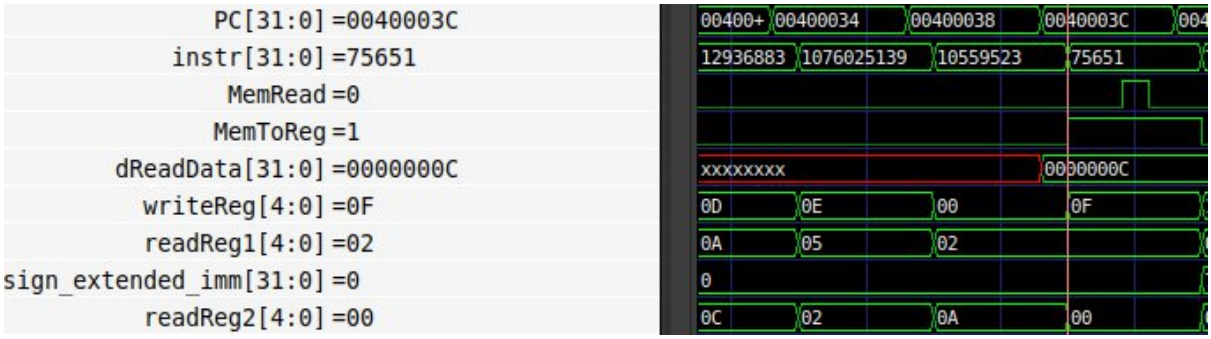
Ενώ δειγματοληπτικά για την πρώτη εντολή: ADDI $1,$0,4 παρατηρώ από το γράφημα:

(immediate τιμή 4, γράψε στον καταχωρητή 1 το περιεχόμενο του καταχωρητή 0 + την τιμή 4, 0010 = ADD στην ALU).

Για την εντολή store word (εγγραφή από το αρχείο καταχωρητών στη μνήμη δεδομένων (που εφ' όσον δεν αρχικοποιείται στο 0 παίρνει τιμές XX στο γράφημα του testbench) έχω το εξής: SW $10, 0($2):

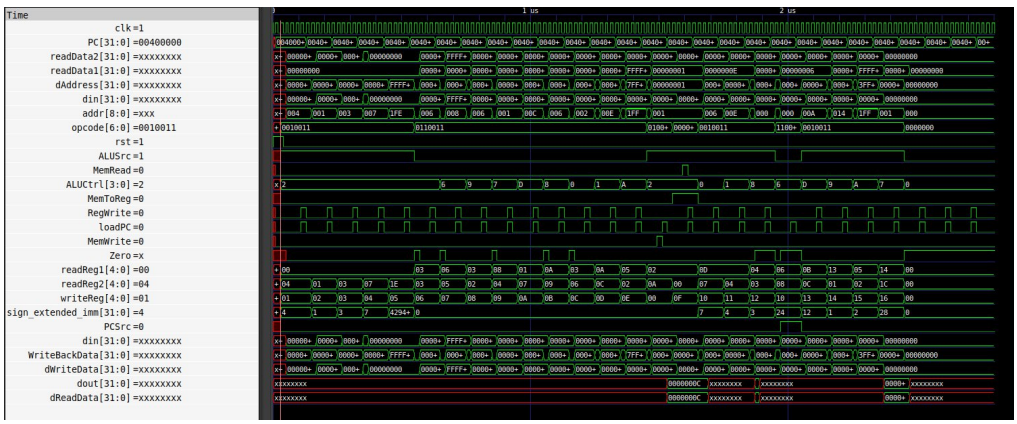
(γράψε στη μνήμη τα δεδομένα του καταχωρητή 10, στη θέση διεύθυνσης του καταχωρητή 2, με immediate offset 0). Παρατηρώ το σήμα MemWrite να σηκώνεται στο 4ο στάδιο του πεντάκυκλου της εντολής καθώς και τη διεύθυνση στην οποία έγραψε να παίρνει πράσινη τιμή (αξιοποιείται απευθείας από την επόμενη εντολή load word).

Για την εντολή load word (εγγραφή από τη μνήμη δεδομένων στον καταχωρητή) έχω το εξής LW $15,0($2):



(γράψε στον καταχωρητή 15 τα δεδομένα που βρίσκονται στη μνήμη με διεύθυνση που βρίσκεται στον καταχωρητή 2 και immediate offset 0).

Με παρόμοιο τρόπο μπορούν να ελεγχθούν και όλες οι υπόλοιπες εντολές επιλέγοντας κάθε φορά τα κατάλληλα σήματα από την αποκωδικοποίηση της εντολής. Μια γενική εικόνα όλων των εντολών που τρέχουν με διάφορα σήματα ανοιγμένα δίνεται από κάτω για λόγους ομορφιάς και πληρότητας:



Υποσημείωση

Το αρχείο rom_bytes.data περιέχει σφάλμα στην εντολή BEQ καθώς επιδιώκει offset 48 για να βρεθεί στην επόμενη εντολή, ωστόσο πραγματοποιεί offset 6. Για τη διόρθωση του σφάλματος απαιτούνται οι εξής αλλαγές:

Γραμμή 77: 0000000 -> 00000010

Γραμμή 79: 0000011 -> 00001000.

Σε περίπτωση μη διόρθωσης με το BEQ πέφτει σε χώρο μηδενικών και χάνει κύκλους ρολογιού έως ότου το PC να βρει τις υπόλοιπες εντολές της μνήμης εντολών. Η εκσφαλμάτωση και η διόρθωση έγινε με το χέρι και με τη βοήθεια των scripts που παρατίθενται, ενώ το τελικό screenshot βρίσκεται πάνω στο διορθωμένο και για αυτό όλες οι εντολές διαδέχονται η μία την άλλη χωρίς να μηδενίζει πουθενά.