Εργαστήριο 7

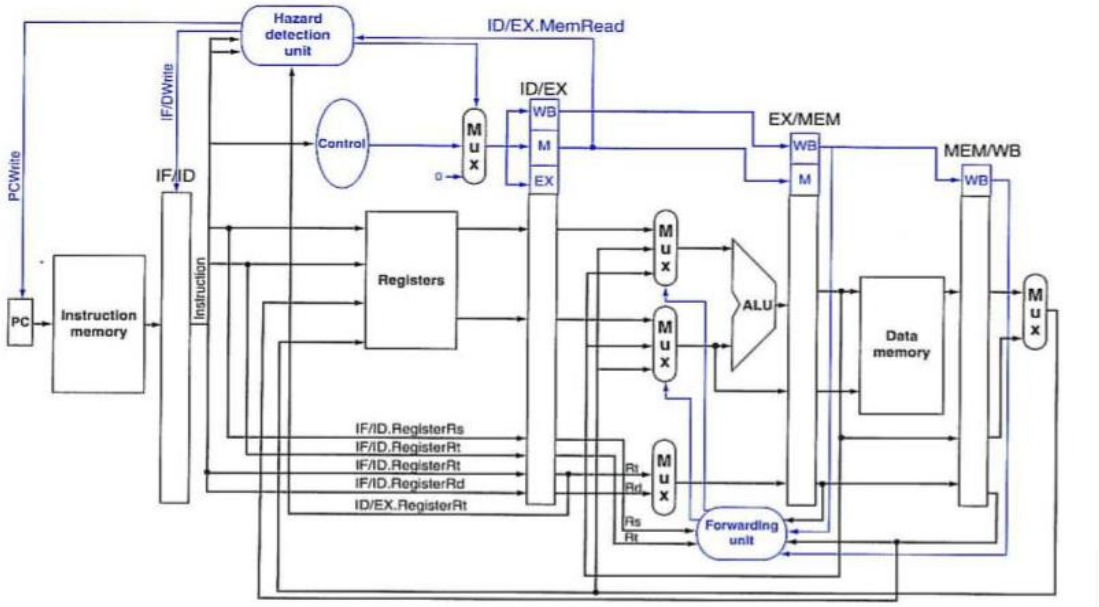
Χειμερινό Εξάμηνο 2015-2016

1. Υλοποίηση του επεξεργαστή MIPS

a. Επέκταση της μικρο-αρχιτεκτονικής διοχέτευσης του MIPS (bypass, stall) (7 μονάδες)

Η χρήση της μικρο-αρχιτεκτονικής διοχέτευσης (pipeline) είναι απαραίτητη σε κάθε σύγχρονο επεξεργαστή, αλλά δημιουργεί μια σειρά από προβλήματα στην σωστή λειτουργία του συστήματος. Στο εργαστήριο αυτό θα σας δοθεί μια αρχική υλοποίηση της μικρο-αρχιτεκτονικής διοχέτευσης του MIPS σε Verilog και εσείς θα πρέπει να υλοποιήσετε κάποιες προσθήκες οι οποίες έχουν σαν σκοπό να διορθώσουν κάποια από αυτά τα προβλήματα.

Η μικρο-αρχιτεκτονική διοχέτευσης μπορεί να εκτελέσει εντολές που ακολουθούν το format-



Εικόνα 1. Αρχιτεκτονική MIPS με μηχανισμό διοχέτευσης (pipeline), μονάδα ανίχνευσης κινδύνου και μονάδα προώθησης (bypass). Η αρχιτεκτονική αυτή μπορεί να εκτελέσει εντολές format-R, καθώς και εντολές φόρτωσης, αποθήκευσης (load/store). Η εικόνα είναι όμοια με την εικόνα 4.60 του βιβλίου.

R καθώς και οι εντολές load/store που έχουμε συναντήσει από τα προηγούμενα εργαστήρια. Το εργαστήριο αυτό σας ζητάει να υλοποιήσετε τις τεχνικές της προώθησης (bypass) και της καθυστέρησης (stall) προκειμένου οι εντολές αυτές να εκτελούνται σωστά σε κάθε περίπτωση. Η θεωρία για το εργαστήριο αυτό καλύπτεται μέχρι και το Κεφ. 4.7 του βιβλίου.

Η εικόνα 1 δείχνει το διάγραμμα της αρχιτεκτονικής που θα πρέπει να υλοποιήσετε, εμπλουτίζοντας το τμήμα δεδομένων (*data path unit* στο *cpu.v*) και το τμήμα ελέγχου (*control unit* στο *control.v*) της μικρο-αρχιτεκτονικής που σας δίδεται. Προτείνουμε την σταδιακή υλοποίηση της μικρο-αρχιτεκτονικής έτσι ώστε να διευκολυνθείτε στην διαδικασία επαλήθευσης της σωστής λειτουργίας. Σε κάθε βήμα θα ήταν καλό να ελέγχετε λεπτομερώς την σωστή λειτουργία του κυκλώματος πριν μεταβείτε στο επόμενο βήμα. Προτείνω να ακολουθήσετε τα παρακάτω βήματα.

a. Μελετήστε τον κώδικα που σας δίνεται τόσο για να καταλάβετε τόσο την λειτουργία του όσο και το σημείο όπου θα πρέπει να τοποθετήσετε το *stall* και το *bypass*. Για την καλύτερη κατανόηση θα πρέπει να προσομοιώσετε τον κώδικα που σας δίνουμε με την χρήση του *Modelsim*. Χρησιμοποιείστε μία ακολουθία εντολών που εξασκεί όσο τον δυνατόν περισσότερα τμήματα του pipeline.

b. Υλοποιήστε την μονάδα προώθησης (bypass). Υπενθυμίζουμε ότι η μονάδα αυτή προωθεί αποτελέσματα που έχουν παραχθεί από την ALU (από εντολή format R) ή που έχουν διαβαστεί από την μνήμη (από μια εντολή **lw**) σε αμέσως επόμενες εντολές που χρειάζονται να διαβάσουν τα αποτελέσματα αυτά. Θα πρέπει να προσέξετε οριακές περιπτώσεις όπως για παράδειγμα αυτές που εμφανίζονται στην παρακάτω ακολουθία εντολών:

```
or $a0, $a0, $t2
add $a0, $a0, $v0
slt $sp, $a0, $t1
```

   Ο καταχωρητής $a0 της τελευταίας εντολής προέρχεται από την εντολή **add** που την στιγμή της προώθησης βρίσκεται στο στάδιο **MEM** της διοχέτευσης και όχι από την εντολή **or** που βρίσκεται στο στάδιο **WB.**

c. Υπάρχει ένα τελευταίο πρόβλημα που δημιουργείται όταν μια εντολή δέχεται δεδομένα από μια αμέσως προηγούμενη εντολή load. Θα πρέπει οι εντολές μετά την load να καθυστερήσουν για έναν κύκλο μηχανής, ενώ θα πρέπει να δημιουργηθεί και ένα bubble στο pipeline μεταξύ της εντολής Load και της αμέσως επόμενης εντολής . Θα πρέπει λοιπόν να υλοποιήσετε μια μονάδα ανίχνευσης κινδύνων η οποία να δημιουργεί αυτό το bubble και να κάνει οτιδήποτε άλλο χρειάζεται για την σωστή εκτέλεση του προγράμματος.

   Για τα ερωτήματα b. και c. θα πρέπει να γράψετε κώδικα στο αρχείο *control.v* για να υλοποιήσετε την λογική ανίχνευσης του bypass και του stall. Επίσης θα πρέπει να γράψετε κώδικα στο αρχείο *cpu.v* για να τοποθετήσετε τα επιπλέον κυκλώματα στο τμήμα δεδομένων.

b. Προσομοίωση για Επαλήθευση Ορθής Λειτουργίας

Το προτεινόμενο πρόγραμμα για τον έλεγχο της διοχέτευσης φαίνεται παρακάτω. Θεωρούμε ότι κάθε καταχωρητής αρχικοποιείται με την τιμή `reg[i] = i`, και δίνουμε σαν σχόλια τα αποτελέσματα μετά από την εκτέλεση κάθε εντολής. Θα πρέπει η υλοποίηση σας να δίνει τα ίδια αποτελέσματα σε κάθε βήμα της εκτέλεσης.

```
label: add $t0, $t0, $s0      # $t0 = $8 = 24 (24 is decimal)
       sw $ra, 4($t2)         # Mem[$t2+4] = 31
       lw $t5, 4($t2)         # $t5 = $13 = 31
       sub $t1, $t1, $a0      # $t1 = $9 = 5
       or $t6, $t7, $t5       # $t6 = $14 = 31
       and $s3, $s0, $s2      # $s3 = $19 = 16
       lw $t6, 4($t2)         # $t6 = $14 = 31
       sw $gp, 8($t2)         # Mem[$t2+8] = 28
       lw $v0, 8($t2)         # $v0 = $2 = 28
       and $a0, $v0, $t5      # $a0 = $4 = 28
       or $a0, $a0, $t0       # $a0 = $4 = 28
       add $t1, $a0, $v0      # $t1 = $9 = 56
       slt $sp, $a0, $t1      # $sp = $29 = 1
```

c. Πρόσθεση επιπλέον εντολής (3 μονάδες)

Το ερώτημα σας ζητά να επεκτείνετε την αρχιτεκτονική του πρώτου ερωτήματος με την εντολή **addi** και τις εντολές αριστερής ολίσθησης **sll** και **sllv.** Προσέξτε ότι και οι δύο εντολές ολίσθησης **sll** και **sllv** ακολουθούν το R-format. Η εντολή **sll** απαιτεί την χρήση του πεδίου *shamt* (bits [10:6] της εντολής) για τον καθορισμό του μεγέθους της ολίσθησης. Συμβουλευτείτε τα opcodes των εντολών αυτών πριν ξεκινήσετε την αποκωδικοποίηση τους σε Verilog.

Τοποθετείστε τις παρακάτω εντολές στο τέλος της προηγούμενης σειράς εντολών για να ελέγξετε την σωστή λειτουργία του τελικού σας κυκλώματος.

```
       lw $v0, 8($t2)         # $v0 = $2 = 28
       sll $s4, $v0, 12       # $s4 = $20 = 0x0001c000
       sllv $s6, $s4, $sp     # $s6 = $22 = 0x00038000
       addi $s6, $s6, -100    # $s6 = $22 = 0x00037f9c
```