Ερώτηση 1: (20 Μονάδες) Ας υποθέσουμε την «κλασσική» αρχιτεκτονική του MIPS με πέντε επίπεδα pipeling (fetch, decode, execute, memory, write-back,) και τον παρακάτω κώδικα

```
            addi r5, r0, 1
            addi r6, r0, 4
            addi r3, r0, 48
            add r2, r0, r0
Loop:       lw r1,10(r2)
            add r1, r1, r5
            sw r1,0(r2)
            add r2, r2, r6
            sub r4, r3, r2
            bnz r4, Loop
```

Όλες οι εντολές παίρνουν έναν κύκλο εκτός από τις LW και SW, που θέλουν έναν κύκλο για εκτέλεση και 2 κύκλους καθυστέρησης σε περίπτωση data hazard, και τις εντολές διακλάδωσης (branch) που έχουν 2 κύκλους καθυστέρηση. Για απλότητα ο επεξεργαστής δεν υποστηρίζει forwarding .

1) Πόσοι κύκλοι ρολογιού ανά επανάληψη χάνονται λόγω της διακλάδωσης και πόσοι συνολικά για το πρόγραμμα;

2) Ας υποθέσουμε ότι έχουμε έναν static branch predictor που υποθέτει ότι όλες οι διακλαδώσεις προς μικρότερες διευθύνσεις ("προς τα πίσω") είναι επιτυχημένες. Όποτε η πρόβλεψη είναι σωστή η διακλάδωση εκτελείται σε έναν κύκλο ενώ όταν είναι λάθος η «σωστή» εντολή εκκινεί 3 κύκλους μετά την εκκίνηση της εντολής branch. Τώρα πόσους κύκλους ρολογιού χάνουμε λόγω της διακλάδωσης συνολικά για όλο το πρόγραμμα ;

3) Ας υποθέσουμε ότι έχουμε έναν dynamic branch predictor που «θυμάται» τι συνέβη στα 2 τελευταία branches και με βάση αυτά παίρνει μια απόφαση. Όποτε η πρόβλεψη είναι σωστή η διακλάδωση εκτελείται σε έναν κύκλο ενώ όταν είναι λάθος η «σωστή» εντολή εκκινεί 3 κύκλους μετά την εκκίνηση της εντολής branch. Τώρα πόσους κύκλους ρολογιού χάνουμε λόγω της διακλάδωσης σε όλο το πρόγραμμα ; Κάντε ότι υπόθεση θέλετε (και γράψτε την) για την απόφαση του predictor όταν το history buffer του είναι άδειο (κοινώς τις πρώτες φορές που θα εκτελεστεί το branch)

Ερώτηση 2: (50 Μονάδες) Έστω ότι ένα υπολογιστικό σύστημα υλοποιεί τον αλγόριθμο Tomasulo και εκτελεί τον παρακάτω κώδικα :

```
Loop:   fld F8,0(r1) ;
        fmul.d F2,F8,F4 ;
        fld F10,0(r2) ;
        fadd.d F10,F2,F10 ;
        fsd F10,0(r2) ;
        addi r1,r1,#8 ;
        addi r2,r2,#8 ;
        sltu r3,r1,r4 ;
        bnez r3, Loop ;
```

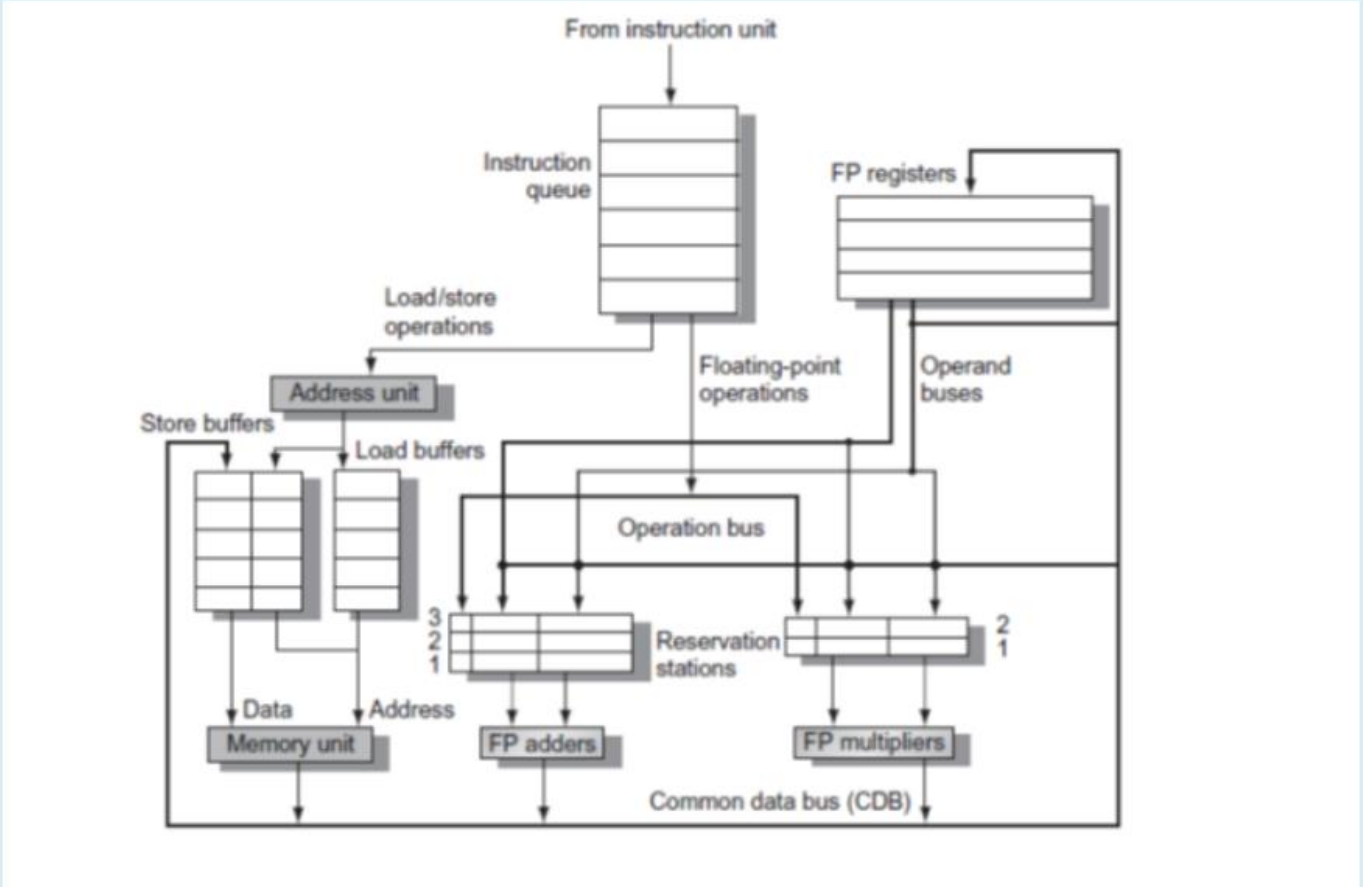
Και έστω οι παρακάτω καθυστερήσεις για τις λειτουργικές μονάδες

Μονάδα	**Κύκλους στο στάδιο EX**	**Αριθμός Μονάδων**	**Αριθμός reservation stations**
Αριθμητική ακεραίων (Integer)	**1**	**1**	**5**
Floating Point Πρόσθεσης (Adder)	**10**	**1**	**3**
Floating Point Πολ/σμου (Multiplier)	**15**	**1**	**2**

Επίσης υποθέτουμε ότι:

- Οι λειτουργικές μονάδες δεν είναι pipelined.
- Δεν υπάρχει προώθηση (forwarding) μεταξύ λειτουργικών μονάδων οπότε τα αποτελέσματα κοινοποιούνται μόνο μέσω του Common Data Bus (CDB).
- Το pipeline του επεξεργαστή είναι διαφοροποιημένο σε σχέση με του «κλασσικό» MIPS αφού στον 3^{o} κύκλο κάνει το issue και υπολογίζει την διεύθυνση για τις προσβάσεις μνήμης και στον 4ο κύκλο κάνει τόσο την πρόσβαση της μνήμης όσο και την εκτέλεση των πράξεων. Δηλαδή το pipeline είναι Fetch / Decode / Issue / Execute+Memory Access / Write Back.
- Τα loads απαιτούν έναν κύκλο ρολογιού.
- Τα στάδια Issue και Write Back απαιτούν το καθένα έναν κύκλο ρολογιού.
- Υπάρχουν πέντε load buffer slots και πέντε store buffer slots.
- Υποθέτουμε ότι η εντολή Branch on Not Equal to Zero (BNEZ) απαιτεί ένα κύκλο ρολογιού

Προς διευκόλυνση σας παρακάτω το σχήμα για το single issue Tomasulo που παρουσιάστηκε και στο μάθημα



Α) Δείξτε του κύκλους εκτέλεσης των εντολών **για 3 πλήρεις εκτελέσεις του loop**. Για να το επιδείξετε αυτό με λεπτομέρεια, σχεδιάστε έναν πίνακα όπως αυτόν που κάναμε στις διαλέξεις που να περιέχει τις παρακάτω στήλες. Υποθέστε ότι ο πρώτος κύκλος που εκτελείται η $1^{η}$ εντολή του προγράμματος είναι ο κύκλος 1

- Αριθμός επανάληψης
- Σε ποια εντολή αναφέρεστε
- Κύκλος που ξεκινάει το issue
- Κύκλος που ξεκινάει το execute
- Κύκλος που ξεκινάει η πρόσβαση μνήμης
- Κύκλος που γράφεται το δεδομένο στο CDB

Β) Σε πόσους κύκλους ρολογιού εκτελείται το κάθε loop (δείξτε πως φτάσατε στο αποτέλεσμα);

Γ) Επαναλάβετε το μέρος (α), αλλά αυτή τη φορά υποθέστε ότι η υλοποίηση του Tomasulo γίνεται two-issue (δηλαδή μπορούν να ξεκινάνε 2 εντολές ταυτόχρονα) και οι μονάδες floatingpoint είναι fullpipelined.

Δ) Ο αλγόριθμος Tomasulo έχει ένα μειονέκτημα: μόνο ένα αποτέλεσμα μπορεί να γραφτεί ανά κύκλο ρολογιού στο CDB. Χρησιμοποιήστε τη διαμόρφωση υλικού (hardware) και τις καθυστερήσεις από την ερώτηση Α και βρείτε μια ακολουθία εντολών, που δεν υπερβαίνει τις 10 εντολές, όπου ο αλγόριθμος Tomasulo να προξενεί stall λόγω της συμφόρησης στο CDB. Δείξτε σε ποια εντολή και σε ποιόν κύκλο δημιουργείται η συμφόρηση.

Ερώτηση 3: (30 Μονάδες) Έστω το παρακάτω κομμάτι κώδικα όπου οι αριθμοί είναι floating point

```
for (i =0 ; i < 500; i++)
{
        c[i] = a[i] * b[i] – d[i] * e[i];
        f[i] = a[i] * e [i] – d[i] * b[i];
}
```

Υποθέστε ότι ο επεξεργαστής λειτουργεί στα 700MHz και έχει μέγιστο μήκος διανύσματος (vector length) 64. Η μονάδα load/store χρειάζεται αρχικά 15 κύκλους για να «γεμίσει» και να αρχίσει την εκτέλεση, η μονάδα πολλαπλασιασμού 8 κύκλους και η μονάδα πρόσθεσης/αφαίρεσης χρειάζεται 5 κύκλους. Κατόπιν όλα λειτουργούν σε έναν κύκλο ρολογιού (fully pipelined).

Α) Ποιο είναι το arithmetic intensity του κώδικα ; Γράψτε αναλυτικά το πώς το υπολογίσατε

Β) Μετατρέψτε το πρόγραμμα σε assembly για RV64V (παρακάτω οι εντολές του RV64V)

Γ) Πόσα chimes περιέχει ο κώδικας σας (με βάση τα «κλασσικά» chimes του RV64V; Με βάση και τις αρχικές καθυστερήσεις που αναφέρονται παραπάνω και έστω ότι υπάρχει μόνο μια μνήμη για τα load και store (άρα κάθε memory access χρειάζεται έναν κύκλο), σε πόσους κύκλους υπολογίζεται το κάθε ζευγάρι τιμών c,f ; Σε πόσο χρόνο θα ολοκληρωθεί το πρόγραμμα σας ;

Δ) Έστω ότι ο επεξεργαστής υποστηρίζει chaining και και μπορεί να κάνει ταυτόχρονα 4 memory accesses σε πόσο χρόνο θα υπολογίζεται το κάθε ζευγάρι τιμών c,f;

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter