1.1 ΑΣΚΗΣΗ

i) Έστω ότι οι εντολές κινητής υποδιαστολής ευθύνονται για το 25% του χρόνου εκτέλεσης ενός προγράμματος σε ένα μηχάνημα. Προτείνεται να βελτιωθεί το υλικό που σχετίζεται με αριθμούς κινητής υποδιαστολής, ώστε οι λειτουργίες κινητής υποδιαστολής να γίνονται 5 φορές ταχύτερα. Ποιο θα είναι το συνολικό speedup για το πρόγραμμα όταν θα χρησιμοποιηθεί το νέο υλικό;

ii) Θεωρήστε ότι έχουν πραγματοποιηθεί οι παρακάτω μετρήσεις για το ίδιο πρόγραμμα και μηχανή :

- Συχνότητα λειτουργιών κινητής υποδιαστολής = 40%
- Μέσο CPI των εντολών κινητής υποδιαστολής = 4
- Μέσο CPI των υπόλοιπων εντολών = 1.5

Μία εναλλακτική για να βελτιώσουμε την απόδοση είναι να μειώσουμε το μέσο CPI των εντολών κινητής υποδιαστολής σε 3. Ποιο θα είναι το speedup του προγράμματος σε μια τέτοια υλοποίηση;

iii) Στο ερώτημα ii) ποιο μέσο CPI των εντολών κινητής υποδιαστολής θα επιτύχει το ίδιο speedup με αυτό του ερωτήματος i);

ΛΥΣΗ

a) Οι FP-> 25% New FP 5 φορές πιο γρήγορα. Άρα :
   $fraction_{en} = 0.25$, $speedup_{en} = 5$. Όμως :
   Speedup = (exec time old) / (exec time new) = $1 / [(1-fraction_{en})+(fraction_{en}/speedup_{en})]$ ⇔
   Speedup = 1 / [(1-0,25)+(0,25/5)] ⇔ Speedup = 1,25
b) $exec_{time}$ = IC * CPI * CC. Μετά την αλλαγή ο αριθμός εντολών (IC) καθώς και ο κύκλος της μηχανής (CC) παραμένουν τα ίδια. Οπότε έχουμε :
   speedup = (exec time old) / (exec time new) = CPI_{old} / CPI_{new} = (0,4*4 + 0,6 * 1,5) / (0,4*3+0,6*1,5) = 1,19
c) Έστω X το μέσο CPI, τότε : speedup = (0,4*X+0,6*1,5) / (0,4*3+0,6*1,5), όπου speedup = 1,25

2.1 ΑΣΚΗΣΗ

Πειραματίζεστε με την αρχική και μια δεύτερη υλοποίηση ενός υπολογιστή και καταλήγετε στα εξής :

- Το clock rate της αρχικής υλοποίησης είναι 5% υψηλότερο.
- 30% των εντολών στην αρχική υλοποίηση είναι load και stores
- Η δεύτερη υλοποίηση εκτελεί τα 2/3 των loads και stores της αρχικής υλοποίησης. Οι υπόλοιπες εντολές μένουν ανέπαφες.
- Όλες οι εντολές (συμπεριλαμβανομένων των load και store) χρειάζονται ένα κύκλο μηχανής.

Θεωρώντας ότι η δεύτερη υλοποίηση μειώνει μόνο τα load και store ποια έκδοση είναι ταχύτερη και πόσο;

ΛΥΣΗ

$CC_{new} = 1,05\ CC_{old}$ και $CPI_{old} = CPI_{new}$. Οπότε :
$IC_{new} = 0,7IC_{old} + 2/3*0.3*IC_{old}$ ⇔ $IC_{new} = 0,9IC_{old}$.
Άρα : Speedup = $(exec_{old} / exec_{new}) = (IC_{old}*CPI*CC_{old}) / (IC_{new}*CPI*CC_{new}) =$
$(IC_{old}*CPI*CC_{old}) / (0,9IC_{old}*CPI*1.05CC_{old})$ ⇔ speedup = 1.058.
Αφού το speedup > 1, το καινούργιο (αρχικό) είναι πιο γρήγορο.

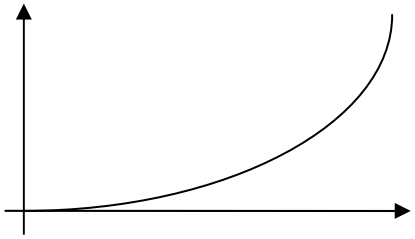
3.1 ΑΣΚΗΣΗ

Έστω ότι σκοπεύουμε να βελτιώσουμε ένα μηχάνημα προσθέτοντάς του ένα vector mode. Όταν ένας υπολογισμός τρέχει στο καινούριο αυτό mode είναι 20 φορές ταχύτερος από την εκτέλεση σε normal mode.

i) Σχεδιάστε τη γραφική παράσταση του speedup ως ποσοστό των υπολογισμών που συμβαίνουν στο vector mode.
ii) Τι ποσοστό των υπολογισμών πρέπει να υλοποιούνται στο vector mode για να επιτύχουμε speedup 2;
iii) Τι ποσοστό των υπολογισμών πρέπει να υλοποιούνται στο vector mode για να επιτύχουμε το μισό του μέγιστου δυνατού speedup;
iv) Έστω ότι το ποσοστό των υπολογισμών που υλοποιούνται στο vector mode είναι 70%. Η ομάδα σχεδιασμού hardware ανακοινώνει ότι μπορεί να διπλασιάσει την ταχύτητα του vector mode. Πόση αύξηση του ποσοστού των υπολογισμών που υλοποιούνται στο vector mode, όπως υφίσταται, θα μπορούσε να πετύχει το speedup που θα έχουμε στην περίπτωση που προτείνουν οι μηχανικοί; Ποιά υλοποίηση θα προτείνατε ;

ΛΥΣΗ

a) Έστω ότι α είναι το ποσοστό των υπολογισμών που συμβαίνουν σε vector mode. Τότε θα έχουμε : speedup = 1 / ((1-a)+(a/20)) => α = (speedup-1) / 0.95*speedup. Η γραφική παράσταση θα είναι κάπως έτσι :



b) speedup = 1 / ((1-a)+(a/20)) = 2 ⇔ a = 0.526
c) 10 = 1 / ((1-a)+(a/20)) => a = 0.95
d) Αν διπλασιάσουμε την ταχύτητα του vector mode τότε θα είναι 2*20 = 40 ($speedup_{en}$), φορές γρηγορότερος. Το α θα γίνει 0.7 και σ' αυτή την περίπτωση θα έχουμε :
$Speedup_{new}$ = 1 / ((1-0,7)+(0,7/40)) = 3,1.
Στην παλιά μηχανή αν είχαμε speedup = 3,1 τότε το α θα ήταν : a = (speedup-1) / 0.95*speedup =>
a = 0.73. Άρα με μόνο 3% αύξηση στο α, με τον παλιό σχεδιασμό (20 φορές ταχύτερος ο υπολογισμός) θα πετυχαίναμε το ίδιο σχεδόν speedup. Είναι λοιπόν και θέμα software η αύξηση της απόδοσης.

4.1 ΑΣΚΗΣΗ

Έστω ένα load-store σύστημα με τέλεια cache που συμπεριφέρεται σύμφωνα με τον πίνακα που ακολουθεί :

Operation	Frequency	Clock cycles
ALU ops	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

Στην περίπτωση που η cache δεν είναι τέλεια, έχουν παρατηρηθεί τα εξής : miss rate εντολών = 5%, miss rate αναφορών σε δεδομένα = 10%, miss penalty = 40 κύκλοι. Βρείτε το CPI για κάθε τύπο εντολής για την περίπτωση ύπαρξης cache misses και υπολογίστε πόσο γρηγορότερος είναι ο υπολογιστής χωρίς cache misses σε σχέση με αυτόν στον οποίο παρατηρούνται cache misses.

ΛΥΣΗ

Ένα σύστημα είναι load-store όταν τη μνήμη μπορούν να προσπελάσουν μόνο η load και η store. Με τα δεδομένα μας, ο πίνακας γίνεται :

Operations	Frequency	Clock Cycles	Instructions Accesses	Data Accesses
ALU Ops	43%	1	1	0
Loads	21%	2	1	1
Stores	12%	2	1	1
Branches	24%	2	1	0

Οπότε :

$CPI = CPI_{ideal} + CPI_{stull}$

CPI_{stull} = MemREFperInstr*MissRate*MissPenalty (MR: δεν το βρίσκει, MP: επιστροφή)

CPI_{ideal} = CPUclockcycles / InstructionCount

Οπότε :

$CPI_{stullALU} = 1*0,05*40+0 = 2$ (0: data accesses)

$CPI_{stullLOAD} = 1*0,05*40+1*0,1*40 = 6$

$CPI_{stullSTORE} = 1*0,05*40+1*0,1*40 = 6$

$CPI_{stullBRANCH} = 1*0,05*40+0 = 2.$

Άρα :

$CPI_{ALU} = CPI_{ideal} + CPI_{stullALU} = 1+2 = 3$

$CPI_{LOAD} = 2+6 = 8$

$CPI_{STORE} = 2+6 = 8$

$CPI_{BRANCH} = 2+2 = 4$

$Speedup = CPI_{notideal} / CPI_{ideal} = (0,43*3+0,21*8+0,12*8+0,24*4) / (0,43*1+0,21*2+0,12*2+0,24*2) = 3.1$

1.2 ΑΣΚΗΣΗ

Σε μια αρχιτεκτονική προσθέτουμε μια νέα register-memory εντολή με σκοπό να αντικαταστήσουμε ζεύγη εντολών της μορφής :

LOAD R1,0(Rb)
ADD R2,R2,R1
by
ADD R2,0(Rb)

Υποθέστε ότι η νέα εντολή θα αυξήσει τον κύκλο ρολογιού κατά 10% αλλά δεν επηρεάζει το CPI. Επίσης γνωρίζουμε ότι το ποσοστό των LOADS σε ένα πρόγραμμα είναι 25,1%

Α. Τι ποσοστό από LOADS πρέπει να εκλείψει με την είσοδο της νέας εντολής, ώστε η απόδοση της αρχιτεκτονικής να παραμείνει ίδια ;

Β. Δείξτε μια κατάσταση όπου ένα LOAD του R1 ακολουθούμενο από μια δεύτερη χρήση του R1 δεν μπορεί να αντικατασταθεί από την νέα εντολή.

ΛΥΣΗ

a) $CPU_{time} = CPI*CC*IC$, όμως :
   $CC_{new} = 1,1CC_{old}$
   $IC_{new} = IC_{old} – R$ => load που αντικαταστάθηκε και από την εκφώνηση έχουμε ότι $CPI_{old} = CPI_{new}$
   Για να έχουμε την ίδια απόδοση με το παλιό σύστημα στο νέο, πρέπει να ισχύει :
   $CPU_{timeold} = CPU_{timenew}$ => $CPI*CC_{old}*IC_{old} = CPI*1,1CC_{old}*(IC_{old} - R)$ => $IC_{old} = 1,1IC_{old} – 1,1R$ =>
   $R = 0,09IC_{old}$

Ή R = 9% του IC_{old}. Συνεπώς το % των LOAD που πρέπει να εκλείψουν από τη νέα μηχανή για να έχουμε την ίδια απόδοση, είναι 9%.

b)

1ο ζεύγος εντολών	**2ο ζεύγος εντολών**
LOAD R1,0(R1)	ADD R1,0(R1)
ADD R1,R1,R1	

Αν θεωρήσουμε R1 = 47 και Mem[47] = 4 τότε τα τελικά αποτελέσματα θα είναι τα εξής :

1ο ζεύγος εντολών	**2ο ζεύγος εντολών**
R1 = 8	R1=51

Σ' αυτή την περίπτωση το πρώτο ζεύγος εντολών δε μπορεί να αντικατασταθεί απ' αυτή τη μία ADD.

2.2 ΑΣΚΗΣΗ

Instruction	gap	gcc	gzip	mcf	perl	Integer average
load	26.5%	25.1%	20.1%	30.3%	28.7%	26%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	21.1%	19.0%	26.9%	10.1%	16.7%	19%
sub	1.7%	2.2%	5.1%	3.7%	2.5%	3%
mul	1.4%	0.1%				0%
compare	2.8%	6.1%	6.6%	6.3%	3.8%	5%
load imm	4.8%	2.5%	1.5%	0.1%	1.7%	2%
cond branch	9.3%	12.1%	11.0%	17.5%	10.9%	12%
cond move	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	2.1%	1.1%	0.5%	2%
and	4.3%	4.6%	9.4%	0.2%	1.2%	4%
or	7.9%	8.5%	4.8%	17.6%	8.7%	9%
xor	1.8%	2.1%	4.4%	1.5%	2.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compare FP						0%
cond mov FP						0%
other FP						0%

Figure 2.32 MIPS dynamic instruction mix for five SPECint2000 programs. Note that integer register-register move instructions are included in the or instruction. Blank entries have the value 0.0%.

Υπολογίστε το effective CPI για τον MIPS χρησιμοποιώντας το figure 2.32. Υποθέστε τις ακόλουθες μετρήσεις για το μέσο CPI εντολών :

Instruction	Clock Cycles
All ALU	1.0
Load-Stores	1.4
Contitional Branches	
Taken	2.0
Not taken	1.5
Jumps	1.2

Υποθέστε επίσης ότι το 60% των conditional branches είναι Taken, οι LOAD imm ανήκουν στις ALU όπως και οι others.

ΛΥΣΗ

EffectiveCPI = Total_Program_Cycles / IC = $\sum_{1}^{n} CPI * ICi$ / IC = $\sum_{1}^{n} CPI * ICi / IC$

όπου ο λόγος ICi / IC υποδηλώνει τη συχνότητα εμφάνισης του τύπου εντολών i. Οπότε :
ALU_{ops} = 19+2,2+0,1+6,1+2,5+0,4+1,1+4,6+8,5+2,1 = 46,6
$LD\text{-}ST_{ops}$ = 25,1+13,2 = 38,3
$Cond\_BR_{ops}$ = 12,1+0,6 = 12,7
$JUMP_{ops}$ = 1,2+0,6+0,6 = 2,4
Άρα :
EffectiveCPI = (1*46,6+1,4*38,3+2*(60%)*12,7+1,5*(40%)*12,7+1,2*2,4)% = 151% = 1,5188

3.2 ΑΣΚΗΣΗ

Έστω ότι προσθέτουμε έναν νέο τρόπο διευθυνσιοδότησης στον MIPS σύμφωνα με τον οποίο προστίθενται 2 registers και ένα 11-bit signed offset για να βρεθεί η διεύθυνση προορισμού. Ο μεταγλωττιστής θα αλλάξει ζεύγη εντολών της μορφής :

ADD R1, R1, R2
LW Rd, 0(R1) (or store)
Με την εντολή
LW Rd, X(R1,R2)

Ο μέσος αριθμός LOADs στον MIPS είναι 26%, ενώ τα STOREs είναι 10%.
A. Εάν ο νέος τρόπος διευθυνσιοδότησης μπορεί να χρησιμοποιηθεί στο 10% των Loads & Stores, ποιος είναι ο λόγος του μέσου συνολικού αριθμού εντολών του νέου MIPS προς τον παλιό ;
B. Αν ο νέος τρόπος διευθυνσιοδότησης αυξάνει τον κύκλο ρολογιού κατά 5%, ποιά μηχανή είναι πιο γρήγορη και πόσο.

ΛΥΣΗ

a) LOADS => στο 10% του 26% φεύγει μια εντολή => 2,6% των εντολών IC
   STORES => στο 10% του 10% φεύγει μια εντολή => 1% των εντολών IC
   Οπότε :
   $\lambda = IC_{new} / IC_{old} = (IC_{old} - 0{,}036IC_{old}) / IC_{old} = 0.964$ ή 96,4% των παλιών εντολών εκτελούνται στο νέο MIPS.
b) $Speedup = exectime_{old} / exectime_{new} = (CPI*IC_{old}*CC_{old}) / (CPI*0{,}964IC_{old}*1{,}05CC_{old}) = 0{,}988$. Άρα το πιο παλιό είναι κατά 1,22% πιο γρήγορο από το νέο.

4.2 ΑΣΚΗΣΗ

Σε μια μηχανή 64-bit θέλουμε να εκτελέσουμε τον παρακάτω μετασχηματισμό μιας εικόνας από RGB σε YUV.

$$Y = (9798*R + 19235*G + 3736*B)/32768$$
$$U = (-4784*R - 9437*G + 4221*B)/32678 + 128$$
$$V = (20218*R – 16941*G + 3277*B)/32768 + 128$$

A. Να υπολογιστεί ο αριθμός των εντολών του προγράμματος μετασχηματισμού εικόνας 16X16 pixels.
B. Να βρεθεί το speed-up για την περίπτωση που η μηχανή υποστηρίζει την εντολή multiply-add.
Γ. Να βρεθεί το speed-up για την περίπτωση που η μηχανή υποστηρίζει SIMD, και stridded addressing.

ΛΥΣΗ

CPI = σταθ = 1, δεν συμπεριλαμβάνονται εντολές διαχείρισης τυχόν επαναλήψεων, δεν υπάρχει DIV

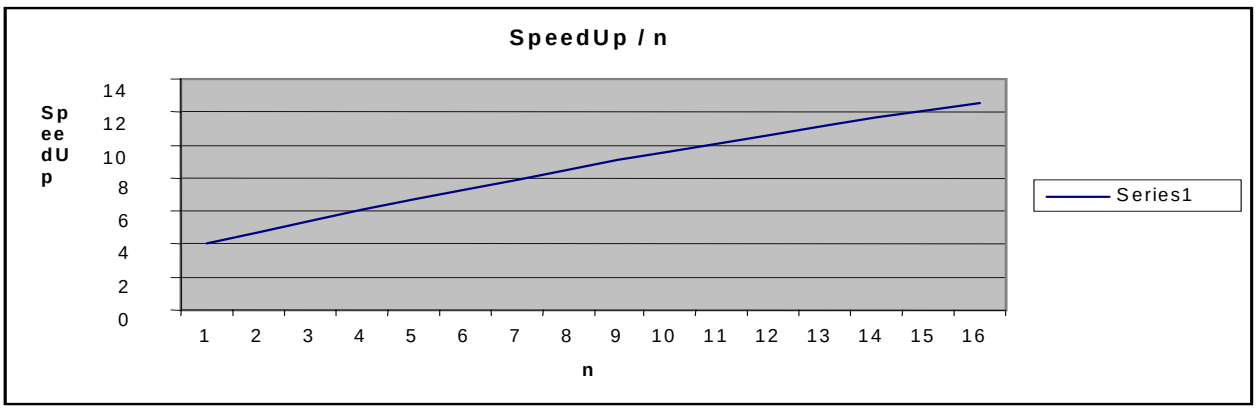
a) Για κάθε R, G, B έχουμε 3 LOADS, 9 MULS, 8 ADDS, 3 STORES, 3 SHIFTS. Δηλαδή έχουμε 26 εντολές για ένα pixel. Οπότε θα χρειαστούμε συνολικά : 16*16*26 = 6656 εντολές μετασχηματισμού εικόνας.
b) Άν υπάρχει MULTIPLE-ADD τότε θα έχουμε για κάθε R, G, B : 3 LOADS, 3 MULS, 6 MUL-ADDS, 2 ADDS, 3 STORES, 3 SHIFTS = 20 εντολές για ένα pixel. Οπότε θα είναι :
   Speedup = IC_{old} / IC_{new} = 26 / 20 = 1,3
c) Θέλουμε να φτιάξουμε όλο 8άδες R, 8άδες G, 8άδες B. Τότε :
   Speedup = IC_{old} / IC_{new} = IC_{old} / (IC_{old}/8) = 16*16*20 / (16*16*20/8) = 8

1.3 ΑΣΚΗΣΗ

Έστω υπολογιστικό σύστημα χωρίς pipeline με clock cycle = 10ns, CPIALU = 4, CPIBranch = 4 και CPILoad / Store = 5, το οποίο εκτελεί προγράμματα με το ακόλουθο μίγμα εντολών : ALU 40%, Branches 20%, Load/Store 40%. Ένα δεύτερο υπολογιστικό σύστημα, με pipeline, χρειάζεται 50ns για την ολοκλήρωση μιας εντολής ανεξάρτητα από τον αριθμό των σταδίων του pipeline. Επιπλέον λόγω του μηχανισμού pipelining προστίθεται μια σταθερή επιβάρυνση 1ns ανά εντολή. Σχεδιάστε το διάγραμμα της επιτάχυνσης που επιτυγχάνεται σε σχέση με το σύστημα χωρίς pipeline, όταν το pipeline έχει 5 έως 20 βαθμίδες. Ποιό θα ήταν το τέλειο speedup σε κάθε περίπτωση ;

ΛΥΣΗ

Γνωρίζουμε ότι CPI_B = 1, αφού σε κάθε κύκλο μηχανής εκτελείται μια εντολή. Έτσι θα έχουμε :
Speedup = CPU_{timeA} / CPU_{timeB} = (IC*CC_A*CPI_A) / (IC*(CC_B*CPI_B+Overhead)). Όμως :
CPI_A = CPIALU*freqALU + CPIBR*freqBR + CPIL/S*freqL/S = 4*0,4+4*0,2+5*0,4 = 4,4
CC_B = 50 / n (pipeline levels) και CC_A = 10 από την εκφώνηση
Οπότε :
Speedup = [10*4,4/((50/n)+1)]. Για n = 5, 10, 15, 20 το speedup = 4, 7.3, 10.15, 12.57 αντίστοιχα. Το γράφημα είναι κάπως έτσι :



Το ιδανικό speedup είναι αυτό που δεν έχει overhead. Θα είχε πλήρως γραμμική γραφική παράσταση και όχι κυρτή όπως είναι τώρα.

2.3 ΑΣΚΗΣΗ

Έστω το επόμενο τμήμα κώδικα :

```
Loop: LW R1, 0(R2)
ADDI R1, R1, #1
SW R1, 0(R2)
ADDI R2, R2, #4
SUB R4, R3, R2
BNEZ R4, Loop
```

Υποθέστε ότι η αρχική τιμή του R3 είναι R2 + 396. Υποθέστε επίσης ότι όλες οι προσπελάσεις στη μνήμη είναι cache hits.

Α. Δώστε το χρονικό διάγραμμα εκτέλεσης των εντολών σε μια μηχανή με pipeline 5 σταδίων, η οποία δεν υποστηρίζει μηχανισμούς αντιμετώπισης των hazards. Πόσοι κύκλοι μηχανής χρειάζονται για την πλήρη εκτέλεση του βρόχου ;

Β. Εάν η μηχανή υποστηρίζει μηχανισμούς forwarding για την αντιμετώπιση των data hazards, πως αλλάζει το χρονικό διάγραμμα εκτέλεσης των εντολών. Πόσοι κύκλοι μηχανής χρειάζονται για την εκτέλεση του βρόχου στη βελτιωμένη μηχανή ;

Γ. Στην μηχανή του ερωτήματος Β, προσθέτουμε έναν μηχανισμό "single-cycle delayed branch" για την αντιμετώπιση των control hazards. Υποθέστε ότι ο υπολογισμός του PC γίνεται κατά το στάδιο ID. Αναδιατάξατε τις εντολές του βρόχου ή/και αλλάξτε τον τρόπο διευθυνσιοδότησης ώστε να βελτιστοποιήσετε την απόδοση της νέας μηχανής. Δώστε το νέο χρονικό διάγραμμα εκτέλεσης και υπολογίστε το speedup σε σχέση με την μηχανή του ερωτήματος Α.

ΛΥΣΗ

a) IF-ID-EX-MEM-WB

LW	IF	ID	EX	MEM	WB													
ADDI		IF	st	st	ID	EX	MEM	WB										
SW			st	st	IF	st	st	ID	EX	MEM	WB							
ADDI				st	st	st	st	IF	ID	EX	MEM	WB						
SUB									IF	st	st	ID	EX	MEM	WB			
BNEZ												IF	st	st	ID	EX	MEM	WB
																		IF

Στο 3^{o} stage θα παγώσει. Στο 18^{o} stage θα γίνει η ενημέρωση του PC ώστε να αποφασιστεί ποιά εντολή θα κάνει fetch. Συνολικά έχουμε 396 / 4 = 99 loops. Όταν κάθε loop φτάνει στην τελευταία εντολή, δεν εκτελεί το WB (δηλαδή συμβαίνει 98 φορές αυτό με 17 στάδια η φορά) και στο τελευταίο loop έχουμε πλήρη εκτέλεση και των 6 εντολών μας. Άρα χρειάζονται : 98*17+18 = 1684 cycles.

b)

LW	IF	ID	EX	MEM	WB						
ADDI		IF	ID	st	EX	MEM	WB				
SW			IF	st	ID	EX	MEM	WB			
ADDI					IF	ID	EX	MEM	WB		
SUB						IF	ID	EX	MEM	WB	
BNEZ							IF	ID	EX	MEM	WB

Στο μηχανισμό forwarding δε χρειάζεται να περιμένει το WB. Οπότε αντίστοιχα με πρίν, εδώ θα έχουμε 98 loops των 10 σταδίων το καθένα και το τελευταίο με 11 στάδια, δηλαδή : 98*10 + 11 = 991 cycles.

c) Τροποποιούμε το πρόγραμμα ως εξής : βρίσκω εντολή που δεν χρησιμοποιεί το R1

Loop: LW R1, O(R2)
ADDI R2, R2, #4
ADDI R1, R1, #1
SUB R4, R3, R2
BNEZ R4, Loop
SW R1, -4(R2)

Οπότε θα έχουμε :

LW	IF	ID	EX	MEM	WB					
ADDI		IF	ID	EX	MEM	WB				
ADDI			IF	ID	EX	MEM	WB			
SUB				IF	ID	EX	MEM	WB		
BNEZ					IF	ID	EX	MEM	WB	
SW						IF	ID	EX	MEM	WB

Εδώ επιτυγχάνουμε τη βέλτιστη υλοποίηση χωρίς ούτε ένα stall. Χρειάζονται : 6*98 + 10 = 598
Οπότε : speedup = 1684 / 598 = 2,816

3.3 ΑΣΚΗΣΗ

Θέλουμε να συγκρίνουμε 2 μηχανές με τα εξής χαρακτηριστικά : Η πρώτη μηχανή έχει 5 στάδια pipeline (IF, ID, EX, MEM, WB) ενώ η δεύτερη έχει 4 (IF, ID, EX+MEM, WB) ενοποιώντας τη φάση εκτέλεσης με τη φάση προσπέλασης στη μνήμη σε ένα στάδιο. Επίσης η δεύτερη μηχανή έχει 50% μεγαλύτερο κύκλο ρολογιού. Το ποσοστό εντολών που παρεμποδίζονται από stalls είναι 5% λόγω branch stalls και 4% λόγω load stalls. Βρείτε ποιά μηχανή είναι πιο γρήγορη και πόσο, εάν υποθέσουμε ότι στη δεύτερη μηχανή εξαλείφονται τα load stalls.

ΛΥΣΗ

LOAD STALLS :

A LD ⇔ IF ID EX MEM WB
IF ID ----- EX MEM

B LD ⇔ IF ID EX-MEM WB
IF ID EX-MEM WB

BRANCH STALLS :

A BR ⇔ IF ID EX MEM WB
IF IF ID EX MEM

Οπότε :

Speedup = $(IC*CC_A*CPI_A) / (IC*1,5CC_A*CPI_B)$

$CPI_A = CPI_{ideal} + CPI_{stall} = 1 + 1*0,05+1*0,04 = 1,09$

$CPI_B = CPI_{ideal} + CPI_{stall} = 1+1*0,05 = 1,05$

Άρα :

Speedup = 1,09 / 1,5*1,05 = 0,692. Άν το κάνουμε B / A (αντιστροφή) έχουμε speedup = 1.445

1.4 ΑΣΚΗΣΗ

Ονοματίστε όλες τις εξαρτήσεις (output, anti, true) που υπάρχουν στο παρακάτω τμήμα κώδικα. Βρείτε ποιες από αυτές είναι loop-carried :

```
for (i=1;i<100;i++){
  a[i]=b[i]+c[i]; //s1
  b[i]=a[i]+d[i]; //s2
a[i+1]=a[i]+e[i]; //s3
}
```

ΛΥΣΗ

Οι εξαρτήσεις σχετίζονται άμεσα με τα hazards.
Output dependency έχουμε όταν συμβεί ένα WriteAfterWrite hazard
Anti dependency έχουμε όταν συμβεί ένα WriteAfterRead hazard
True dependency έχουμε όταν συμβεί ένα ReadAfterWrite hazard

Παρατηρούμε ότι στην s2 εντολή, το α[i] διαβάζεται ενώ στην s1 το α[i] γράφεται, οπότε έχουμε ένα RAW hazard και συνεπώς έχουμε μια true dependency. (η s2 εξαρτάται από την s1 λόγω του α)
Όμοια η s2 εξαρτάται από την s1 λόγω του b. Εδώ έχουμε ένα WAR hazard (anti dependency).
Η s3 έχει το α[i+1] και η s2 το α[i]. Εδώ έχουμε WAW hazard (output dependency). Η s3 είναι επίσης και loop-carried επειδή αναμειγνύονται δύο iterations.
Η s2 εξαρτάται από την s3 λόγω του α (RAW loop-carried => true dependency)
Η s3 εξαρτάται από την s3 λόγω του α (RAW loop-carried => true dependency)
Η s3 εξαρτάται από την s1 λόγω του α (RAW => true dependency?)

2.4 ΑΣΚΗΣΗ

Ο επόμενος βρόχος υλοποιεί την πράξη Y[i]=a*X[i]+Y[i] :

```
Loop: L.D F0, 0(R1)
MUL.D F0, F0, F2
L.D F4, 0(R2)
ADD.D F0, F0, F4
S.D 0(R2), F0
SUB R1, R1, #8
SUB R2, R2, #8
BNEZ R1, Loop
```

Υποθέστε τις καθυστερήσεις που δίνονται στον πίνακα 1 για το pipeline και θεωρήστε επίσης ότι τα branches δημιουργούν καθυστέρηση ενός κύκλου.
Α. Υποθέστε μια single issue μηχανή. Ξαναγράψτε τον κώδικα, προσθέτοντας τα stalls που συμβαίνουν λόγω του pipeline. Ποιός είναι ο χρόνος εκτέλεσης ανά αποτέλεσμα ;

Πίνακας 1

Instr. producing	Instr. using	Latency
FP ALU	FP ALU	3
FP ALU	Store Double	2
Load Double	FP ALU	1
INT ALU	Branch	1

Β. Ξεδιπλώστε το βρόχο και αναδιανείμετε τις εντολές με στόχο την εξάλειψη των stalls. Υποθέστε ότι η μηχανή υποστηρίζει 1-cycle delayed branch. Ποιός είναι ο νέος χρόνος εκτέλεσης ανά αποτέλεσμα ;
Γ. Έστω ότι ο παραπάνω κώδικας εκτελείται σε μια dual-issue superscalar μηχανή (INT / LD / ST instr. + FP instr.). Επαναλάβετε το ερώτημα Β για τη νέα μηχανή. Τι ποσοστό από issue slots παραμένει αχρησιμοποίητο ;

ΛΥΣΗ

A	B	Γ(INT INSTR)	Γ(FP INSTR)
L.D F0, 0(R1)	L.D F0, 0(R1)	LD F0, 0(R1)	
stall	L.D F8, -8(R1) (για να διαβάσει σωστή τιμή)	LD F8, -8(R1)	
MUL.D F0, F0, F2	MUL.D F0, F0, F2	LD F16, -16(R1)	MULD F0, F0, F2
L.D F4, 0(R1)	MUL.D F8, F8, F2	LD F24, -24(R1)	MULD F8, F8, F2
stall	L.D F4, 0(R2)	LD F4, 0(R2)	MULD F16, F16, F2
stall	L.D F12, 0(R2)	LD F12, -8(R2)	MULD F24, F24, F2
ADD.D F0, F0, F4	ADD.D F0, F0, F4	LD F20, -16(R2)	ADD F0, F0, F4
stall	ADD.D F8, F8, F12	LD F28, -24(R2)	ADD F8, F8, F12
stall	SUB R1, R1, #16	SUB R1, R1, #32	ADD F16, F16, F20
S.D 0(R2), F0	S.D 0(R2), F0	SD 0(R2), F0	ADD F24, F24, F28
SUB R1, R1, #8	SUB R2, R2, #16	SD -8(R2), F8	
SUB R2, R2, #8	BNEZ R1, loop	SD -16(R2), F16	
BNEZ R1, Loop	S.D -8(R2), F8	SUB R2, R2, #32	
stall		BNEZ R1, loop	
		SD 8(R2), F24	

Στο Α έχουμε συνολικά 14 cycles per iteration, όπως φαίνεται από τον πίνακα.
Στο Β γράφω μόνο τις εντολές :

L.D F0, 0(R1)	MUL.D F0, F0, F2	L.D F4, 0(R2)	ADD.D F0, F0, F4	S.D 0(R2), F0

Στη συνέχεια προσθέτουμε 8 στους registers (εκτός από το F2 γιατί είναι read) και έχουμε :

L.D F8, -8(R1)	MUL.D F8, F8, F2	L.D F12, 0(R2)	ADD.D F8, F8, F12	S.D -8(R2), F8	SUB R1, R1, #16	SUB R2, R2, #16	BNEZ R1, loop

Οπότε το πρόγραμμα γίνεται όπως φαίνεται στον 1^{ο} πίνακα. Η υλοποίηση αυτή γεμίζει τα stalls. Οπότε έχουμε 13 cycles αλλά 2 iterations. Επομένως έχουμε 13 / 2 = 6.5 cycles per iteration .
Στο Γ έχουμε 15 cycles αλλά 4 iterations, οπότε έχουμε 15 / 4 = 3.75 cycles per iteration. Στο FP instr παρατηρούμε 7 κενά, οπότε το ποσοστό των issue slots που παραμένει αχρησιμοποίητο είναι : 7 / 30 = 23.3%.

3.4 ΑΣΚΗΣΗ

Για την επίλυση της άσκησης χρησιμοποιείστε τον πίνακα 1 της προηγούμενης που αναφέρει τα pipeline latencies. Υποθέστε επίσης ότι η single-issue μηχανή υποστηρίζει 1-cycle delayed branch. Δίνεται ο κώδικας :
Loop : L.D F2, 0(R1) MUL.D F4, F2, F0 L.D F6, 0(R2) ADD.D F6, F4, F6 S.D 0(R2), F6 ADD R1, R1, #8 ADD R2, R2, #8 SUB R3, R1, #800 BNEZ R3, Loop
Α. Ξαναγράψτε τον κώδικα όπως αυτός θα έτρεχε σε μια pipelined μηχανή συμπληρώνοντας stalls όπου αυτά χρειάζονται και υπολογίστε το χρόνο εκτέλεσης ανά αποτέλεσμα. Αναδιατάξτε τις εντολές με στόχο την

εξάλειψη των stalls και υπολογίστε το βελτιστοποιημένο χρόνο εκτέλεσης ανά αποτέλεσμα. Πόσο πιο γρήγορο θα έπρεπε να είναι το ρολόϊ στη μη βελτιστοποιημένη εκτέλεση ώστε να επιτυγχάνουμε τον ίδιο χρόνο εκτέλεσης με τη βελτιστοποιημένη ;
Β. ξεδιπλώστε το βρόχο και αναδιανείμετε τις εντολές με στόχο την εξάλειψη των stalls. Ποιός είναι ο χρόνος εκτέλεσης ανα αποτέλεσμα ;
Γ. Υποθέστε μια κλασική VLIW μηχανή η οποία μπορεί να εκτελέσει 2 προσπελάσεις μνήμης, 2 πράξεις FP, και 1 πράξη INT ή branch σε κάθε κύκλο μηχανής. Ξεδιπλώστε το βρόχο 4 φορές και ξαναγράψτε το πρόγραμμα για την VLIW μηχανή χωρίς να αφήνετε τελείως κενούς κύκλους μηχανής. Αγνοήστε το stall που προέρχεται από το branch. Υπολογίστε το χρόνο εκτέλεσης ανα αποτέλεσμα. Ποιό ποσοστό slots παραμένει αχρησιμοποίητο ;

ΛΥΣΗ

a) Για την unscheduled έχω 16 cycles per iteration. Η scheduled είναι :
   Loop : L.D F2, 0(R1)
   L.D F6, 0(R2)
   MUL.D F4, F2, F0
   ADD R1, R1, #8
   ADD R2, R2, #8
   SUB R3, R1, #800
   ADD.D F6, F4, F6
   Stall
   BNEZ R3, Loop
   S.D –8(R2), F6
   Θέλουμε να αυξήσουμε την ταχύτητα ρολογιού στην unscheduled. Ισχύει :
   $IC_{UNS}*CC_{UNS} = IC_{SCH}*CC_{SCH} \Rightarrow CC_{SCH} / CC_{UNS} = 16 / 10 = 1.6 \Rightarrow$ 60% πιο γρήγορο, όπου IC εννοείται Instruction Count per Iteration (στην ουσία είναι execution).
b) Όμοια με πρίν
c) Έχουμε :

Mem1	Mem2	FP1	FP2	Int or Branch
L.D F2, 0(R1)	L.D F8, 8(R1)			
L.D F14, 16(R1)	L.D F20, 24(R1)			
		MUL.D F4, F2, F0	MUL.D F10, F8, F0	
		MUL.D F16, F14, F0	MUL.D F22, F20, F0	
L.D F6, 0(R2)	L.D F12, 8(R2)			ADD R1, R1, #32
L.D F18, 16(R2)	L.D F24, 24(R2)			
		ADD.D F6, F4, F6	ADD.D F12, F10, F12	ADD R2, R2, #32
		ADD.D F6, F4, F6	ADD.D F24, F22, F24	
				SUB R3, R1, #800
S.D –8(R2), F6	S.D –8(R2), F6			
S.D –8(R2), F6	S.D –8(R2), F6			BNEZ R3, Loop

Οπότε έχουμε 11 cycles αλλά 4 iterations : 11 / 4 = 2.75 cycles per iteration.

1.5 ΑΣΚΗΣΗ

Έστω υπολογιστικό σύστημα με τα παρακάτω χαρακτηριστικά :

- Το ποσοστό επιτυχίας της cache είναι 95%.
- Κάθε block της cache αποτελείται από 2 λέξεις, και στην περίπτωση miss διαβάζεται ολόκληρο το block.
- Ο δίαυλος είναι εύρους 1 λέξης.
- Το 25% των αναφορών στη μνήμη είναι εγγραφές.
- Σε οποιαδήποτε χρονική στιγμή, το 30% των blocks στην cache έχουν αλλαχθεί.
- Η cache ακολουθεί την πολιτική write allocate στα write misses.

Υπολογίστε το μέσο αριθμό προσπελάσεων στην κύρια μνήμη που αντιστοιχεί σε μια προσπέλαση στην cache στην περίπτωση όπου η cache είναι α) write-through και β) write-back.

ΛΥΣΗ

	HIT	Type	Freq	Accesses στην κύρια μνήμη
WRITE THROUGH	Yes	READ	95%*75% = 71.3%	0
	Yes	WRITE	95%*25% = 23.8%	1
	No	READ	5%*75% = 3.8%	2
	No	WRITE	5%*25% = 1.3%	3 (2+1) (σύμφωνα με την πολιτική write allocate στα write misses τα φέρνει πρώτα στην cache).

$MemAccess_{AVG}$ = 71,3%*0 + 23,8%*1 + 3,8*2 + 1.3%*3 = 0,35.

Άρα για κάθε ένα access στην cache έχουμε 0,35 στη μνήμη.

	HIT	Dirty	Freq	Accesses στην κύρια μνήμη
WRITE BACK	Yes	No	95%*70% = 66.5%	0
	Yes	Yes	95%*30% = 28.5%	0
	No	No	5%*70% = 3.5%	2 (1 για κάθε word)
	No	Yes	5%*30% = 1.5%	4 (2+2) (2 για να πάμε το dirty bit στην κ. μνήμη και 2 για να φέρουμε το νέο block στην cache)

$MemAccess_{AVG}$ = 66,5%*0 + 28,5%*0 + 3,5%*2 + 1,5%*4 = 0,13

Άρα με την write back τεχνική έχουμε λιγότερα miss penalties.

2.5 ΑΣΚΗΣΗ

Έστω υπολογιστική μηχανή, για την οποία ισχύουν τα εξής :

- Το memory latency είναι 40 κύκλοι μηχανής
- Κάθε cache block είναι ίσο με 32 bytes και στην περίπτωση miss μεταφέρεται ολόκληρο το block
- Το 20% των εντολών αφορούν εντολές μεταφοράς δεδομένων
- Ο ρυθμός μετάδοσης είναι 4 bytes ανα κύκλο μηχανής
- Το 50% των μεταφορών αφορούν dirty blocks
- Δεν υπάρχει write buffer
- Άν αγνοήσουμε το σύστημα μνήμης τότε το CPI είναι ίσο με 1.5

α) Υπολογίστε το πραγματικό CPI της μηχανής για 3 διαφορετικά είδη write-back, unified cache όπως αυτά δίνονται στον παρακάτω πίνακα :

Είδος cache	Miss Rate
16-KB direct-mapped, unified	0.029
16-KB two-way set-associative, unified	0.022
32-KB direct-mapped, unified	0.020

β) Έστω ένας σχεδιασμός cache που αποτελείται από μια 16-KB instruction cache και μια 16-KB data cache με miss rate 0.004 και 0.05 αντίστοιχα. Υπολογίστε το πραγματικό CPI σε αυτή την περίπτωση. Τι ποσοστό των εντολών πρέπει να είναι Load / Store ώστε ο σχεδιασμός αυτός να παρουσιάζει μεγαλύτερο πραγματικό CPI σε σχέση με αυτό μιας 32-KB direct-mapped, unfied cache ; Υποθέστε ότι όλες οι cache είναι write-back.

ΛΥΣΗ

a) Ισχύει : $CPI_{πραγματικό} = CPI_{exec}$ + MemStalls / Instruction

MemStalls / Instruction = (InstrFetchStalls + DataStalls) / Instruction (1)
Όμως γενικά ισχύει ότι :
StallCycles / Instruction = MemAccesses / Instruction *MissRate * MissPenalty
Οπότε η (1) γίνεται :
MemStalls / Instruction = $MemAccesses_{InstrFetch}$ / Instruction *MR_I * MP_I + $MemAccesses_{DATA}$ / Instruction *MR_D * MP_D
Επομένως : $CPI_{πραγματικο} = 1.5 + 1*MR*MP + 0.2MR*MP = 1.5 +1.2*MR*MP$ (2)
Για τα MissPenalties :
MP = (MemLatency + BlockTransfers) (1+%dirty) = (40 + 32/4) (1 + 0.5) = 72 cycles
Οπότε από τη (2) παίρνουμε :

Είδος cache	MR	CPI
16-KB direct-mapped, unified	0.029	4
16-KB two-way set-associative, unified	0.022	3.4
32-KB direct-mapped, unified	0.020	3.2

b) Στην περίπτωση αυτή θα ισχύει ότι : $MR_I \neq MR_D$. Άρα :
MemStalls / Instruction = 1*0.004*72 + 0.2*0.05*72 = 1.5
Οπότε $CPI_{πραγματικο} = 1.5 + 1.5 = 3$
Το ποσοστό των εντολών που πρέπει να είναι LOAD / STORE για να ισχύει $CPI_{16KB} > CPI_{32KB}$ πρέπει να είναι :
$1.5 + 1*0.004*72 + X*0.05*72 > 1.5 + 1*0.02*72 + X*0.02*72 \Rightarrow X > 0.53 \Rightarrow X > 53\%$

Δηλαδή οι εντολές LOAD-STORE πρέπει να είναι περισσότερες από το 53% για να είναι ο σχεδιασμός της 32-KB direct-mapped, unified χειρότερος.