Μιχάλης Ψυχής
el22094, michalis.psychis@gmail.com
2η Σειρά Ασκήσεων, Αρχιτεκτονική Υπολογιστών. Τμήμα 3

Μέρος Α

1) Η εντολή branch_on_not_equal ελέγχει εάν το περιεχόμενο δύο καταχωρητών δεν είναι ίδιο, και αν ισχύει, κάνει jump στο label.

bne $rs,$rt,label => if($rs != $rt) go to PC+4+label

opcode (6 bits)	rs (5 bits)	rt (5 bits)	label (16 bits)

Το opcode της εντολής bne είναι *000101.* Η control unit για αυτήν την είσοδο, εξάγει τα εξής Bit σε κάθε path:

Instruction	Opcode (6-bit)	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
BNE (Branch)	000101	X	0	1	0	X	01	0	0	0

Ας δούμε ένα παράδειγμα. Έστω ότι ισχύει rs != rt. Τότε η αναμενόμενη έξοδος είναι PC+4+label. Το ALUop είναι 01, οπότε εκτελείται η πράξη rs - rt στην ALU. Ισχύει *rs - rt != 0,* επομένως το path zero της ALU είναι 0. Άρα η LOGICAL AND GATE, θα έχει έξοδο 0 και στο MUX θα επιλεγεί το PC+4. Υπάρχει πρόβλημα διότι αναμέναμε το PC+4+label, δηλαδή στο MUX να επιλεγεί 1.

Αντίστοιχα, ακόμη και αν ίσχυε rs = rt, θα είχαμε έξοδο PC+4+label (MUX = 1), ενώ αναμέναμε PC+4 (MUX = 0).

2) Πως θα μπορούσαμε να επεκτείνουμε και τροποποιήσουμε το datapath, για να μπορούμε να υποστηρίξουμε την bne;

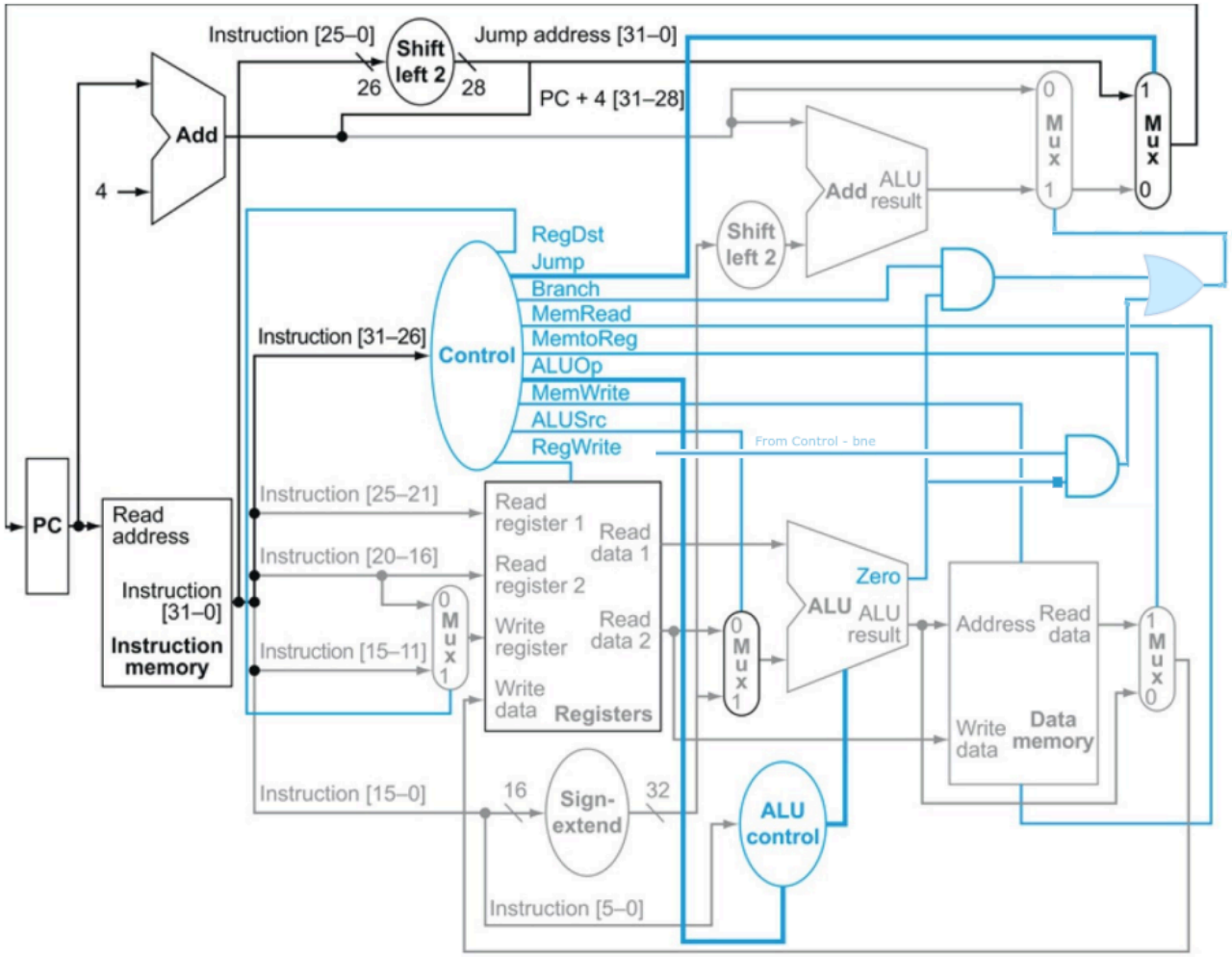
Αρχικά θα πρέπει να προσθέσουμε ένα ακόμη output path στο control unit. Θα το ονομάσουμε BranchNotEqual και θα αντιστοιχεί σε 0 όταν δεν έχουμε bne (op != 000101), ενώ σε 1 όταν έχουμε bne (op = 000101).

Προσθέσαμε μια LOGICAL AND GATE, με είσοδο το *BranchNotEqual* και το *NOT Zero.* Η έξοδος πηγαίνει σε μια OR GATE, με την άλλη είσοδο να είναι η έξοδος της AND GATE που αντιστοιχεί στο branch. Η έξοδος της OR, καθορίζει την επιλογή στο MUX. Γιατί όμως αυτή η συνδεσμολογία είναι σωστή?

Αρχικά εάν δεν έχουμε bne (BranchNotEqual = 0), τότε η AND GATE του BranchNotEqual, έχει έξοδο 0, οπότε δεν επηρεάζει την OR GATE. Επομένως η λειτουργία είναι αντίστοιχη με πριν. Αν έχουμε bne (BranchNotEqual = 1 και Branch = 0), τότε η AND GATE του Branch, έχει έξοδο 0. Άρα ουσιαστικά η είσοδος του MUX καθορίζεται από την AND GATE του BranchNotEqual.
Συγκεκριμένα:

- Εάν $rs != $rt θέλουμε PC+4+label (MUX = 1) - Πράγματι zero = 0, άρα (NOT zero) AND (BranchNotEqual) = 1, επομένως MUX = 1

- Εάν $rs = $rt θέλουμε PC+4 (MUX = 0) - Πράγματι zero = 1, άρα (NOT zero) AND (BranchNotEqual) = 0, επομένως MUX = 0



Το σχεδιάγραμμα του νέου datapath

Μέρος Β

1)

Κύκλος	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
LW $t2, 8($t1)	IF	ID	EX	MEM	WB																								
ADD $t2, $t2, $t1		IF	ID			EX	MEM	WB																					
SW $t2, 8($t1)			IF			ID			EX	MEM	WB																		
ADDI $t1, $t1, 8						IF			ID	EX	MEM	WB																	
LW $t6, 8($t1)									IF	ID			EX	MEM	WB														
LW $t5, 16($t1)										IF			ID	EX	MEM	WB													
ADD $t2, $t6, $t5													IF	ID			EX	MEM	WB										
SW $t2, 8($t1)														IF			ID			EX	MEM	WB							
ADDI $t4, $t4, -4																	IF			ID	EX	MEM	WB						
BNE $t9, $t4, LOOP																				IF	ID			EX	MEM	WB			
LW $t2, 8($t1)																									IF	ID	EX	MEM	WB

Έχουμε συνολικά 6 hazards. Είναι όλα RAW.

Συγκεκριμένα:

- **ADD $t2, $t2, $t1** : Χρησιμοποιούμε τον register $t2 που κάνουμε load στην προηγούμενη εντολή.
- **SW $t2, 8($t1)** : Κάνουμε store τον register $t2 στην Μνήμη 8 + $t1, που κάνουμε add στην προηγούμενη εντολή.
- **LW $t6, 8($t1)** : Κάνουμε load στον $t6 χρησιμοποιώντας τον $t1. Τον $t1 τον κάνουμε addi στην προηγούμενη εντολή.
- **ADD $t2, $t6, $t5** : Κάνουμε add τους $t5 και $t6, που κάναμε load στις αμέσως 2 προηγούμενες εντολές.
- **SW $t2, 8($t1)** : Κάνουμε store τον register $t2 στην Μνήμη 8 + $t1, που κάνουμε add στην προηγούμενη εντολή.
- **BNE $t9, $t4, LOOP** : Χρησιμοποιούμε τον $t4 στην branch not equal, που κάναμε addi στην προηγούμενη εντολή.

Ακόμη, με αρχικά arguments $t9=0x1000 και $t4=0x1400, μπορούμε να δούμε πόσα loops θα πραγματοποιηθούν. Ουσιαστικά ο κώδικας αφαιρεί 4 από τον $t4, μέχρι να ισούται με την τιμή του $t9. Επομένως (0x1400 - 0x1000)/4 loops = (5120 - 4096)/4 = 1024/4 = 256 loops. Κάθε loop διαρκεί 24 ClockCycles, εκτός από το τελευταίο που διαρκεί 26cc. Άρα συνολικά 24*255+26 = 6146 cc.

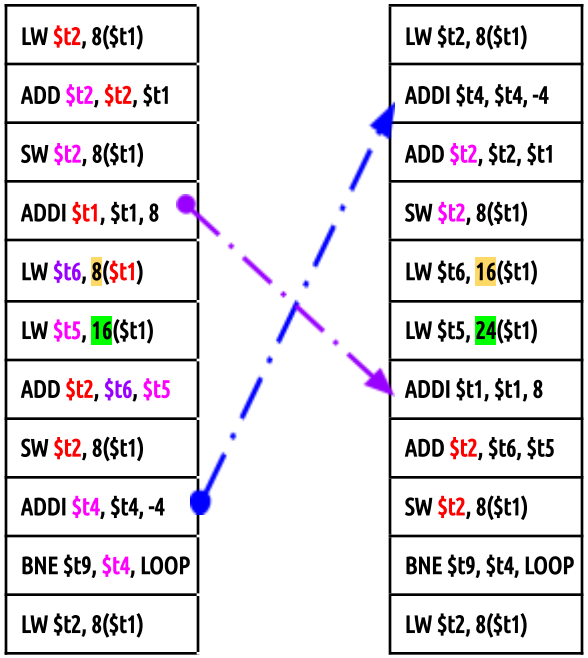
2)

Κύκλος	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
LW $t2, 8($t1)	IF	ID	EX	MEM	WB														
ADD $t2, $t2, $t1		IF	ID		EX	MEM	WB												
SW $t2, 8($t1)			IF		ID	EX	MEM	WB											
ADDI $t1, $t1, 8					IF	ID	EX	MEM	WB										
LW $t6, 8($t1)						IF	ID	EX	MEM	WB									
LW $t5, 16($t1)							IF	ID	EX	MEM	WB								
ADD $t2, $t6, $t5								IF	ID		EX	MEM	WB						
SW $t2, 8($t1)									IF		ID	EX	MEM	WB					
ADDI $t4, $t4, -4											IF	ID	EX	MEM	WB				
BNE $t9, $t4, LOOP												IF	ID	EX	MEM	WB			
LW $t2, 8($t1)															IF	ID	EX	MEM	WB

Παρατηρούμε ότι με προωθήσεις, απαιτούνται 14 ClockCycles για κάθε loop. Επομένως χρειάζονται 14*255 + 16 cc = 3568 cc.

Έχουμε μόνο δύο hazards, όταν πραγματοποιείτε MEM to EX forwarding, καθώς θα πρέπει να περιμένουμε το αποτέλεσμα από την πρόσβαση της μνήμης.

3)



Πρέπει να επιλύσουμε τα hazards στις γραμμές 2 και 7. Αυτό μπορεί να γίνει με τον εξής τρόπο:

Αρχικά μεταφέρουμε την εντολή **ADDI $t4, $t4, -4** στην 2η σειρά. Κατά αυτόν τον τρόπο μετά την lw, εκτελείται αυτή η εντολή και δίνεται χρόνος ούτως ώστε να μην χρειάζεται να γίνει stall για έναν cc. Πρέπει όμως ακόμη να γίνει MEM to EX forwarding από την lw στην add, καθώς το wb της lw γίνεται ταυτόχρονα με το ex της add. Ένα ακόμη πλεονέκτημα είναι ότι αποφεύγουμε το EX to EX forwarding μεταξύ της addi και της bne.

Για το δεύτερο stall, μπορούμε να μεταφέρουμε την εντολή **ADDI $t1, $t1, 8** στην 7η σειρά. Έτσι παρεμβάλουμε μια εντολή μεταξύ των lw και της add, η οποία δημιουργεί μια καθυστέρηση ενός cc. Επομένως δεν απαιτείται να γίνει stall. Πρέπει όμως ακόμη να γίνει MEM to EX forwarding από την lw στην add, καθώς το WB της lw γίνεται ταυτόχρονα με το EX της add. Ένα ακόμη πλεονέκτημα είναι ότι δεν χρειάζεται πλέον να γίνει EX to EX forwarding μεταξύ της addi και της lw. Άρα μειώσαμε τα forwards από 6 σε 4.

Μια σημαντική σημείωση είναι πως για να λειτουργεί ορθά το πρόγραμμα, θα πρέπει να διασφαλίσουμε ότι οι αλλαγές που κάναμε δεν επηρεάζουν την λογική του. Η μετακίνηση της **ADDI $t4, $t4, -4** δεν δημιουργεί κανένα πρόβλημα καθώς ο $t4 χρησιμοποιείται μόνο στην bne. Η μετακίνηση όμως της **ADDI $t1, $t1, 8** δημιουργεί πρόβλημα στις lw, καθώς χρησιμοποιούν τον $t1 για να έχουν πρόσβαση στη μνήμη. Για αυτό τον λόγο αυξήσαμε κατά 8 τα offset των lw.

Κύκλος	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LW $t2, 8($t1)	IF	ID	EX	MEM	WB												
ADDI $t4, $t4, -4		IF	ID	EX	MEM	WB											
ADD $t2, $t2, $t1			IF	ID	EX	MEM	WB										
SW $t2, 8($t1)				IF	ID	EX	MEM	WB									
LW $t6, 16($t1)					IF	ID	EX	MEM	WB								
LW $t5, 24($t1)						IF	ID	EX	MEM	WB							
ADDI $t1, $t1, 8							IF	ID	EX	MEM	WB						
ADD $t2, $t6, $t5								IF	ID	EX	MEM	WB					
SW $t2, 8($t1)									IF	ID	EX	MEM	WB				
BNE $t9, $t4, LOOP										IF	ID	EX	MEM	WB			
LW $t2, 8($t1)													IF	ID	EX	MEM	WB

Παρατηρούμε ότι με προωθήσεις και αναδιάταξη του κώδικα, απαιτούνται 12 ClockCycles για κάθε loop. Επομένως χρειάζονται 12*255 + 14 cc = 3074 cc.

4)

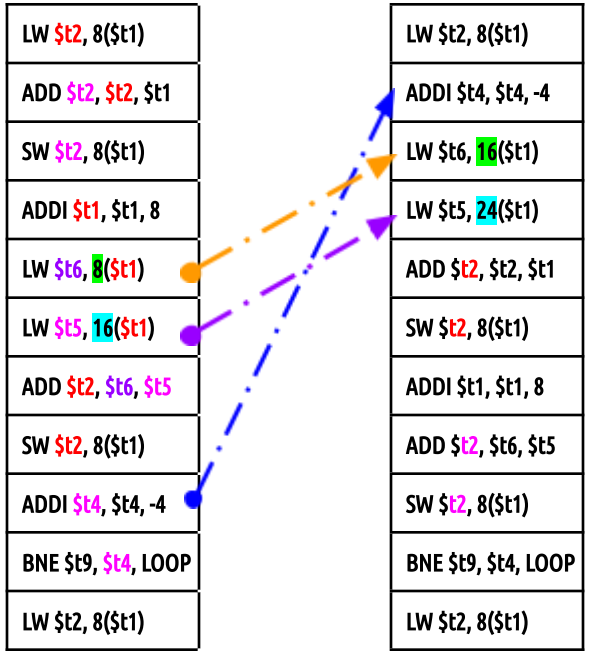
Κύκλος	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LW $t2, 8($t1)	IF	ID	EX	M1	M2	WB																
ADD $t2, $t2, $t1		IF	ID			EX	M1	M2	WB													
SW $t2, 8($t1)			IF			ID	EX	M1	M2	WB												
ADDI $t1, $t1, 8						IF	ID	EX	M1	M2	WB											
LW $t6, 8($t1)							IF	ID	EX	M1	M2	WB										
LW $t5, 16($t1)								IF	ID	EX	M1	M2	WB									
ADD $t2, $t6, $t5									IF	ID			EX	M1	M2	WB						
SW $t2, 8($t1)										IF			ID	EX	M1	M2	WB					
ADDI $t4, $t4, -4													IF	ID	EX	M1	M2	WB				
BNE $t9, $t4, LOOP														IF	ID	EX	M1	M2	WB			
LW $t2, 8($t1)																	IF	ID	EX	M1	M2	WB

Παρατηρούμε ότι, χρησιμοποιώντας δύο στάδια MEM, με προωθήσεις, απαιτούνται 16 ClockCycles για κάθε loop. Επομένως χρειάζονται 16*255 + 19 = 4099cc. Δεν μπορούμε να συμπεράνουμε εάν βελτιώθηκε η επίδοση του επεξεργαστή, καθώς δεν γνωρίζουμε τους Clock Cycles στις δύο περιπτώσεις. Βλέπουμε όμως ότι με ένα στάδιο MEM απαιτούνται 3568cc, ενώ με 2 στάδια MEM απαιτούνται 531cc παραπάνω.

5) Οι χρόνοι κάθε σταδίου είναι οι εξής: **IF: 240ps, ID: 120ps, EX: Tps, MEM: 400ps, WB: 120ps** . Επομένως ο κύκλος του ρολογιού θα πρέπει να είναι ίσος με το πιο αργό στάδιο, δηλαδή max(Tps, 400ps). Εάν ισχύει max(Tps, 400ps) = Tps, τότε αυτομάτως οποιαδήποτε αλλαγή στα στάδια του MEM δεν επηρεάζει το ClockCycle, οπότε η αλλαγή του κώδικα είναι πιο αποδοτική καθώς πραγματοποιούνται λιγότεροι κύκλοι. Επομένως πρέπει Tps < 400ps, για να είναι βέλτιστα τα δύο στάδια MEM. Έτσι γνωρίζουμε ότι το ClockCycle του datapth με αλλαγή κώδικα είναι 400ps και ο συνολικός χρόνος t1 = ClockCycle * Cycles = 400ps *3074cc = **1,2296μs.**

Χρησιμοποιώντας 2 στάδια MEM, οι νέοι χρόνοι κάθε σταδίου είναι οι εξής: **IF: 240ps, ID: 120ps, EX: Tps, MEM1: 200ps, MEM2: 200ps, WB: 120ps .** Ο χρόνος του ρολογιού θα είναι ίσως με το πιο αργό στάδιο, δηλαδή max(Tps, 240ps). Υποθέτουμε ότι Tps > 240ps. Άρα ClockCycle = Tps. Για να είναι βέλτιστα τα δύο στάδια MEM, θα πρέπει t2 < 1.2296μs => ClockCycle * Cycles < 1.2296μs => Tps * 4099cc < 1.2296μs => T < 299,97ps => **Tmax = 299,97ps .** Πράγματι είναι μεγαλύτερο από 240ps.

6)



Κάνοντας τις εξής αναδιατάξεις στον κώδικα, καταφέραμε να έχουμε μηδενικά stalls, χρησιμοποιώντας μόνο 2 forwards. Για να μην επηρεαστεί η λογική του κώδικα, θα πρέπει να αυξήσουμε κατά 8 τα offset των LoadWord. Με αυτές τις αλλαγές, ένα loop αποτελείται από 12 ClockCycles. Έτσι το πρόγραμμα μας χρειάζεται συνολικά 12*255 + 15 = 3075cc.

Οι χρόνοι κάθε σταδίου είναι οι εξής: **IF: 240ps, ID: 120ps, EX: Tps, MEM: 400ps, WB: 120ps** . Επομένως ο κύκλος του ρολογιού θα είναι ίσος με max(Tps, 400ps). Εάν ισχύει max(Tps, 400ps) = Tps, τότε αυτομάτως οποιαδήποτε αλλαγή στα στάδια του MEM δεν επηρεάζει το ClockCycle, οπότε είναι πιο αποδοτικό να διατηρήσουμε ένα στάδιο MEM, και να αλλάξουμε τον κώδικα καθώς πραγματοποιείται ένας λιγότερος κύκλος. Επομένως πρέπει Tps < 400ps, για να είναι βέλτιστα τα δύο στάδια MEM. Έτσι γνωρίζουμε ότι το ClockCycle του datapth ενός σταδίου, με αλλαγή κώδικα είναι 400ps και ο συνολικός χρόνος t1 = ClockCycle * Cycles = 400ps *3074cc = **1,2296μs.**

Χρησιμοποιώντας 2 στάδια MEM, οι νέοι χρόνοι κάθε σταδίου είναι οι εξής: **IF: 240ps, ID: 120ps, EX: Tps, MEM1: 200ps, MEM2: 200ps, WB: 120ps .** Ο χρόνος του ρολογιού θα είναι ίσως με το πιο αργό στάδιο, δηλαδή max(Tps, 240ps). Υποθέτουμε ότι Tps > 240ps. Άρα ClockCycle = Tps. Για να είναι βέλτιστα τα δύο στάδια MEM, θα πρέπει t2 < 1.2296μs => ClockCycle * Cycles < 1.2296μs => Tps * 3075cc < 1.2296μs => T < 399,87ps => **Tmax 399,87ps .** Πράγματι είναι μεγαλύτερο από 240ps.

Κύκλος	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LW $t2, 8($t1)	IF	ID	EX	M1	M2	WB												
ADDI $t4, $t4, -4		IF	ID	EX	M1	M2	WB											
LW $t6, 16($t1)			IF	ID	EX	M1	M2	WB										
LW $t5, 24($t1)				IF	ID	EX	M1	M2	WB									
ADD $t2, $t2, $t1					IF	ID	EX	M1	M2	WB								
SW $t2, 8($t1)						IF	ID	EX	M1	M2	WB							
ADDI $t1, $t1, 8							IF	ID	EX	M1	M2	WB						
ADD $t2, $t6, $t5								IF	ID	EX	M1	M2	WB					
SW $t2, 8($t1)									IF	ID	EX	M1	M2	WB				
BNE $t9, $t4, LOOP										IF	ID	EX	M1	M2	WB			
LW $t2, 8($t1)													IF	ID	EX	M1	M2	WB