ΗΥ-120: Ψηφιακή Σχεδίαση
Φθινόπωρο 2021

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Εργαστήριο 12: Διακλαδώσεις, Έμμεσες Προσπελάσεις (Pointers), Ταχύτητα και Κατανάλωσης Ενέργειας των Κυκλωμάτων CMOS

14 - 19 Ιανουαρίου 2022

12.1 Επανεκτέλεση Εντολών: Διακλαδώσεις, Άλματα

```
s = 0;  n = input();
do { s = s+n;  n = n−1;
     } while (n != 0)
S = s;  /* print sum */
```

Instr. Memory:

	1E:	ld	00
	1F:	st	19
	20:	inp	18
n=n−1; s=s+n;	21:	ld	19
	22:	add	18
	23:	st	19
	24:	ld	18
	25:	sub	01
	26:	st	18
	27:	bne	21 •
	28:	ld	19
	29:	st	1A
	2A:	jmp	1E

Data Memory:

00:	0
01:	1
	• • •
18:	n = ~~8~~ ~~7~~ 6
19:	s = ~~0~~ ~~8~~ F
1A:	S

Γιά τις εντολές πράξεων και φόρτωσης/αποθήκευσης που είδαμε στην §11.3 η επόμενή τους προς εκτέλεση εντολή ήταν αυτή που είναι γραμμένη στην αμέσως "από κάτω" θέση μνήμης (§11.4). Αν ήταν έτσι όλες οι εντολές ενός προγράμματος, οι εντολές αυτές θα εκτελούντο ακριβώς μία φορά η καθεμιά, από την πρώτη μέχρι την τελευταία, και το πρόγραμμα θα τελείωνε πολύ γρήγορα. Όπως ξέρουμε, όμως, οι υπολογιστές αντλούν τη δύναμη και την ευελιξία τους από τη δυνατότητά τους να *επανεκτελούν* πολλαπλές φορές την ίδια σειρά εντολών (την ίδια δουλειά - τον ίδιο *αλγόριθμο*) πάνω σε διαφορετικά κάθε φορά δεδομένα, ούτως ώστε τελικά να επιτυγχάνουν μακρές και πολύπλοκες επεξεργασίες πληροφοριών. Τη δυνατότητα αυτή την αποκτούν με τις **εντολές διακλάδωσης** (branch) ή άλματος (jump). Όπως είδαμε στην §11.9, η εντολή άλματος κάνει ώστε η επόμενη εντολή που θα εκτελεστεί να είναι η εντολή που βρίσκεται σε ορισμένη θέση (διεύθυνση) *διαφορετική* από την "από κάτω" θέση. Μιά εντολή διακλάδωσης κάνει το ίδιο, αλλά **υπό συνθήκη**, δηλαδή μερικές φορές η επόμενη εντολή είναι η "άλλη", και μερικές φορές θα είναι η "από κάτω", ανάλογα αν ισχύει ή όχι μιά δοσμένη συνθήκη κάθε φορά.

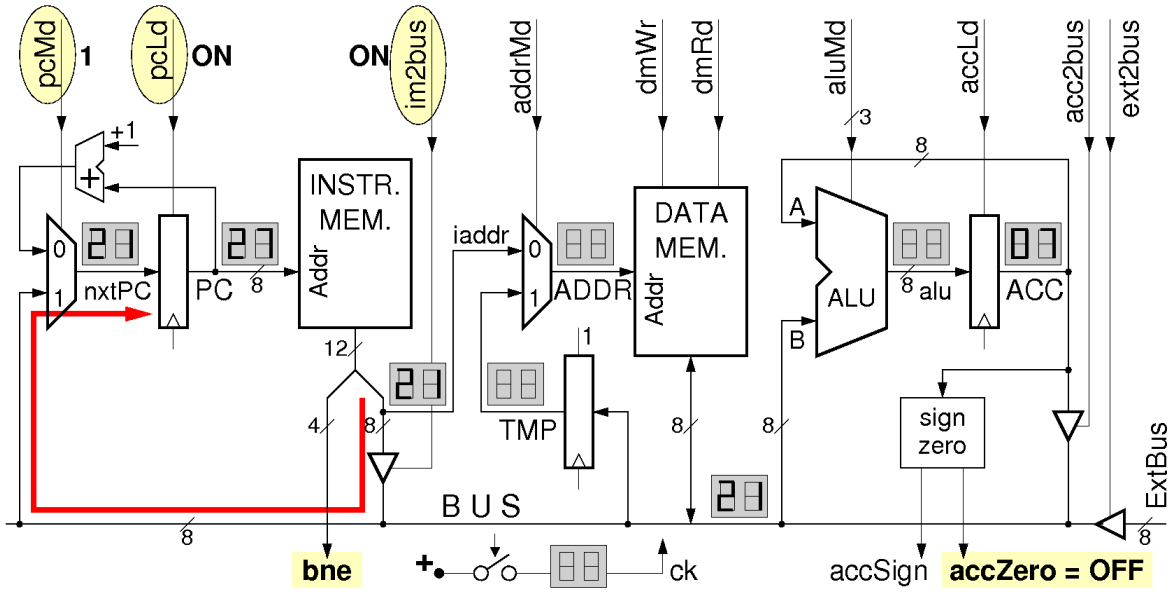
Στο σχήμα βλέπουμε ένα παράδειγμα προγράμματος που υπολογίζει το άθροισμα n+(n-1)+(n-2)+...+3+2+1 και το γράφει στη θέση 1A της μνήμης δεδομένων. Ξεκινάμε μηδενίζοντας τη μεταβλητή "s" (θέση μνήμης 19), αντιγράφοντας εκεί από τη θέση 00 που σε εμάς περιέχει τον αριθμό 0 που φροντίζουμε να μην αλλάζει ποτέ. Τον αριθμό n, στη θέση μνήμης 18, τον διαβάζει η επόμενη εντολή (στη διευθ. 20) από το πληκτρολόγιο: `input 18`· στο σχήμα υποτίθεται ότι ο αριθμός αυτός είναι 8. Οι τρείς αρχικές εντολές του "βρόχου", στις θέσεις 21, 22, και 23, διαβάζουν την τρέχουσα τιμή της μεταβλητής s, της προσθέτουν την τρέχουσα τιμή της μεταβλητής n, και γράφουν το αποτέλεσμα πίσω στην s (s=s+n). Στο παράδειγμα, την πρώτη φορά που εκτελούνται αυτές οι εντολές, αυξάνουν το s από 0 σε 8. Οι τρείς επόμενες εντολές (θέσεις 24, 25, και 26) ελαττώνουν τη μεταβλητή n κατά 1· στο παράδειγμα, την πρώτη φορά που εκτελούνται, αλλάζουν το n από 8 σε 7.

Στη συνέχεια εκτελείται η εντολή `bne 21` από τη θέση 27· η εντολή αυτή σημαίνει *εάν ο συσσωρευτής **δεν** ισούται με μηδέν, διακλαδώσου (πήγαινε) στην εντολή 21* (branch if ACC not equal to zero - brach not equal - bne). Επειδή εκείνη την ώρα ο συσσωρευτής περιέχει το n=7, που είναι διάφορο του μηδενός, η συνθήκη της διακλάδωσης είναι αληθής και η διακλάδωση επιτυγχάνει (πραγματοποιείται). Έτσι, επόμενες εντολές εκτελούνται οι εντολές 21, 22, και 23, αυξάνοντας το s από 8 σε F (=15 στο δεκαδικό), και μετά οι 24, 25, και 26, μειώνοντας το n από 7 σε 6. Μετά, ξαναεκτελείται η `bne 21`· επειδή ο συσσωρευτής περιέχει το 6, η διακλάδωση επιτυγχάνει και πάλι. Έτσι, οι εντολές 21 έως και 27 θα ξαναεκτελεστούν κάμποσες φορές ακόμα, αυξάνοντας διαδοχικά το s κατά 6, 5, ..., 2, και 1, και μειώνοντας το n διαδοχικά σε 5, 4, ..., 1, και 0.

Την τελευταία φορά, στο συσσωρευτή θα έχει μείνει n=0. Τότε, η διακλάδωση `bne 21` θα αποτύχει, διότι ο συσσωρευτής δεν είναι πλέον διάφορος του μηδενός· έτσι, η επόμενη εντολή δεν θα διαβαστεί από τη θέση 21 όπως πρίν, αλλά από τη θέση 28, δηλαδή από την "από κάτω" θέση, όπως κάνουν και όλες οι άλλες εντολές που δεν είναι διακλαδώσεις. Τώρα, οι εντολές 28 και 29 θα αντιγράψουν το τελικό αποτέλεσμα 8+7+6+...+2+1 από τη θέση 19 (s) στη θέση 1A (S) και ο στόχος του προγράμματος θα έχει επιτευχθεί.

12.2 Υλοποίηση στον Υπολογιστή μας των Διακλαδώσεων υπό Συνθήκη

Γιά να εκτελέσει ο υπολογιστής μας τις εντολές διακλάδωσης (υπό συνθήκη) και άλματος (χωρίς συνθήκη) απαιτείται ενεργοποίηση των δρόμων που φαίνονται στο παρακάτω σχήμα· πρόκειται ακριβώς γιά τους ίδιους δρόμους που είδαμε και στην §11.9 γιά την εντολή άλματος, jump. Ο πολυπλέκτης που τροφοδοτεί τον PC δέχεται στις δύο εισόδους του τις διευθύνσεις των δύο υποψήφιων επόμενων εντολών: της "από κάτω" (PC+1, στην είσοδο 0) γιά τις αποτυχημένες διακλαδώσεις, και της εντολής προορισμού (ADDR, στην είσοδο 1) γιά τις επιτυχημένες διακλαδώσεις. Η διεύθυνση ADDR φτάνει στην είσοδο 1 του πολυπλέκτη από το BUS, όπου την βάζει ο τρικατάστατος οδηγητής που ανάβει από το σήμα im2bus. Το μόνο που μένει είναι να ελεγχθεί σωστά ο πολυπλέκτης του PC ώστε να επιλέξει την σωστή είσοδό του ανάλογα με το αν η εκάστοτε διακλάδωση είναι επιτυχημένη ή αποτυχημένη (σημειωτέον ότι η εντολή άλματος (jump) συμπεριφέρεται σαν μία πάντα επιτυχημένη διακλάδωση).



Γιά να ξέρει το κύκλωμα ελέγχου αν μιά εντολή διακλάδωσης είναι επιτυχημένη ή αποτυχημένη δεν του αρκούν μόνο τα 4 bits του opcode σαν είσοδος --χρειάζεται εισόδους και από τα δεδομένα, και συγκεκριμένα να ξέρει τι είδους αριθμός βρίσκεται αυτή τη στιγμή στον συσσωρευτή (ACC): θετικός, αρνητικός, μηδέν, κλπ. Ο υπολογιστής μας θα έχει τέσσερεις εντολές διακλάδωσης υπό συνθήκη: **(α)** `bne` (branch not equal): διακλαδώσου εάν ACC διάφορος του μηδενός· **(β)** `beq` (branch equal): διακλαδώσου εάν ACC ίσος με το μηδέν· **(γ)** `bge` (branch greater or equal): διακλαδώσου εάν ACC μεγαλύτερος ή ίσος του μηδενός· και **(δ)** `blt` (branch less than): διακλαδώσου εάν ACC μικρότερος του μηδενός· επίσης **(ε)** έχει την εντολή άλματος χωρίς συνθήκη, `jump`, η οποία αλλάζει πάντοτε την επόμενη εντολή. Γιά να εκτελεστούν οι εντολές διακλάδωσης πρέπει να ξέρουμε το πρόσημο του συσσωρευτή καθώς και αν αυτός είναι μηδέν ή όχι. Αυτός είναι ο ρόλος των σημάτων accSign και accZero (πρόσημο, μηδέν) που φαίνονται στο datapath να γεννιούνται από το block συνδυαστικής λογικής "sign/zero" που κοιτάζει το περιεχόμενο του συσσωρευτή, ACC.

Όπως είδαμε στην §6.3, το πρόσημο ενός αριθμού κωδικοποιημένου σε συμπλήρωμα ως προς 2 είναι το περισσότερο σημαντικό (MS) bit του (το αριστερότερο bit): όταν αυτό είναι 1 τότε ο αριθμός είναι αρνητικός, ενώ όταν το MS bit είναι 0 ο αριθμός είναι θετικός ή μηδέν. Η ανίχνευση του εάν ο αριθμός είναι μηδέν ή διάφορος του μηδενός απαιτεί μιά πύλη NOR με τόσες εισόδους όσα τα bits του αριθμού. Αφού η έξοδος μιάς πύλης NOR είναι 1 όταν και μόνον όταν όλες οι είσοδοί της είναι μηδέν, συνδέοντας κάθε bit του συσσωρευτή σε μία είσοδο της NOR έχουμε την έξοδό της να ανάβει όταν και μόνον όταν όλα τα bits του συσσωρευτή είναι μηδέν, δηλαδή όταν ο ACC περιέχει τον αριθμό μηδέν. Ονομάζουμε *accSign* το MS bit του ACC, ονομάζουμε *accZero* την έξοδο της πύλης NOR 8 εισόδων, και παρέχουμε στο κύκλωμα ελέγχου αυτά τα δύο σήματα σαν εισόδους του.

Τώρα, ο πίνακας αληθείας του κυκλώματος ελέγχου που είδαμε στη §11.10 πρέπει να συμπληρωθεί όπως παρακάτω. Πάντα γιά όλες αυτές τις εντολές: pcLd=1 και addrMd=0. Η εντολή `jumpx` που φαίνεται ακριβώς κάτω από την `jump` είναι νέα και θα συζητηθεί στην επόμενη §12.3 Γιά τις εντολές που προϋπήρχαν, περιλαμβανόμενης και της `jump`, τα σήματα ελέγχου δεν εξαρτώνται από

τις νέες εισόδους, *accZero* και *accSign*, όπως υποδεικνύουν τα "x" στις αντίστοιχες γραμμές και στήλες. Γιά τις εντολές διακλάδωσης, όμως, το σήμα *pcMd* που ελέγχει το ποιά θα είναι η επόμενη εντολή καθορίζεται από τον τύπο της διακλάδωσης και από τις εισόδους *accZero* και *accSign*. Παρατηρήστε ότι γιά τις beq/bne (opcode = 100x), το pcMd είναι το αποκλειστικό-Ή (XOR) του accZero με το LS bit του opcode· γιά τις blt/bge (opcode = 101x), το pcMd είναι το XOR του accSign με το LS bit του opcode. Γιά τις αποτυχημένες διακλαδώσεις (opcode = 10xx και pcMd = 0), το im2bus μπορεί να είναι είτε 0 είτε 1, δεδομένου ότι ο πολυπλέκτης αγνοεί ούτως ή άλλως την τιμή του BUS· αν επιλέξουμε 0, τότε im2bus = pcMd, ενώ αν επιλέξουμε 1, τότε απλοποιείται ο χάρτης Karnaugh του im2bus.

accZero:		*Λειτουργία:*	*aluMd*	*acc2bus*		*dmWr*	*pcMd*	
opcode:	*accSign:*		*dmRd*		*accLd*	*ext2bus*		*im2bus*
0000 (add)	x x	ACC:=ACC+DM[A]	1	000	1	0	0	0 0
0001 (sub)	x x	ACC:=ACC-DM[A]	1	001	1	0	0	0 0
0010 (and)	x x	ACC:=ACCandDM[A]	1	010	1	0	0	0 0
0011 (nor)	x x	ACC:=ACCnorDM[A]	1	011	1	0	0	0 0
0100 (inp)	x x	DM[A]:=IO_BUS	0	xxx	0	0	1	1 0
0101 (ld)	x x	ACC:=DM[A]	1	1xx	1	0	0	0 0
0110 (st)	x x	DM[A]:=ACC	0	xxx	0	1	0	1 0
0111 (jmp)	x x	PC := A	0	xxx	0	0	0	0 1
1111 (jmpx)	x x	PC := DM[A]	1	xxx	0	0	0	0 1
1000 (beq)	0 x	PC := PC+1	0	xxx	0	0	0	0 x
1000	1 x	PC := A	0	xxx	0	0	0	0 1
1001 (bne)	0 x	PC := A	0	xxx	0	0	0	0 1
1001	1 x	PC := PC+1	0	xxx	0	0	0	0 x
1010 (blt)	x 0	PC := PC+1	0	xxx	0	0	0	0 x
1010	x 1	PC := A	0	xxx	0	0	0	0 1
1011 (bge)	x 0	PC := A	0	xxx	0	0	0	0 1
1011	x 1	PC := PC+1	0	xxx	0	0	0	0 x

12.3 Διαδικασίες: Άλματα σε Μεταβλητές Διευθύνσεις

Όλες οι εντολές διακλάδωσης, καθώς και η εντολή απλού άλματος (jump) που είδαμε μέχρι τώρα, έχουν τη διεύθυνση προορισμού τους καθορισμένη μέσα στην εντολή αυτή καθ'εαυτή --γραμμένη μέσα στη μνήμη εντολών (διεύθυνση προορισμού είναι η διεύθυνση της "άλλης" επόμενης εντολής --όχι της "από κάτω"). Στον υπολογιστή μας, κάθε τι που είναι γραμμένο στη μνήμη εντολών **δεν** μπορεί να αλλάξει την ώρα που τρέχει το πρόγραμμα, διότι δεν υπάρχουν εντολές που να γράφουν στην μνήμη εντολών. Το ίδιο συμβαίνει και στους αληθινούς, εμπορικούς υπολογιστές, γιά τη συντριπτική πλειοψηφία του σημερινού λογισμικού: παρ' ότι οι μνήμες εντολών και δεδομένων δεν είναι χωριστές, όμως **δεν** γράφουμε πάνω στις εντολές, ούτως ώστε να αποφύγουμε ένα σωρό πονοκεφάλους κατά την αποσφαλμάτωση (debugging) των προγραμμάτων. Έτσι προκύπτει ότι η διεύθυνση προορισμού μιάς διακλάδωσης ή ενός απλού άλματος δεν μπορεί να αλλάξει την ώρα που τρέχει το πρόγραμμα, δηλαδή είναι μία και μοναδική, πάντα η ίδια, αυτή που καθόρισε ("στατικά") ο προγραμματιστής όταν έγραφε το πρόγραμμα. Αυτό επαρκεί γιά τους περισσότερους αλλά όχι γιά όλους τους σκοπούς.

Γιά να μπορεί ο υπολογιστής μας να εκτελεί *διαδικασίες* (procedures), πρέπει να έχουμε έναν τρόπο ώστε, όταν τελειώνει η εκτέλεση της διαδικασίας, να "πηγαίνει" το πρόγραμμα πίσω σε αυτόν που την κάλεσε, όποιος κι αν ήταν αυτός. Το θέμα είναι ότι την ίδια διαδικασία μπορεί να την καλέσουμε από πολλά και διαφορετικά μέρη ενός προγράμματος· πώς θα μπορέσει αυτή να "επιστρέψει" άλλοτε στο ένα και άλλοτε στο άλλο από αυτά, εάν η τελευταία της εντολή είναι ένα απλό άλμα που μας πηγαίνει πάντα σε ένα, σταθερό σημείο; Φυσικά, όποιος καλεί τη διαδικασία μπορεί να γράφει κάπου (δηλαδή στη μνήμη δεδομένων, όπου και μόνο μπορούν να γράφουν τα προγράμματα την ώρα που τρέχουν) τη διεύθυνσή του, δηλαδή τη διεύθυνση της εντολής όπου επιθυμεί να επιστρέψει η διαδικασία. Όμως το ζητούμενο είναι να έχουμε μιάν ειδική εντολή άλματος η οποία να μπορεί να μας στείλει σε **μεταβλητή** διεύθυνση προορισμού, δηλαδή σε προορισμούς που να ποικίλλουν την ώρα που τρέχει το πρόγραμμα. Αν την έχουμε αυτήν, τότε κάθε διαδικασία μπορεί να τελειώνει με μιά τέτοια εντολή, η οποία θα βρίσκει την επόμενή της εντολή διαβάζοντας τη διεύθυνση αυτής της επόμενης εντολής από την ειδική εκείνη θέση της μνήμης δεδομένων όπου αυτός που κάλεσε την διαδικασία έγραψε την εκάστοτε επιθυμητή διεύθυνση επιστροφής.

Την ειδική αυτή εντολή άλματος την ονομάζουμε *jmpx ADDR* (jump indexed - άλμα με δείκτη, ή άλλοτε jump indirect - έμμεσο άλμα), όπως αυτή φαίνονταν στο παραπάνω πίνακα αμέσως κάτω από την απλή εντολή άλματος *jmp ADDR*. Σαν σημασία και σαν υλοποίηση, η νέα εντολή είναι ένα κράμα του απλού άλματος και της εντολή φορτώματος (load) του συσσωρευτή. Σαν το απλό άλμα, το έμμεσο άλμα γράφει μιά νέα τιμή στον PC, διαφορετική από PC+1 (και γιά το σκοπό αυτό κάνει pcMd=1)· σαν την εντολή load όμως, αυτό που φορτώνει στον PC (αντί στον ACC) είναι το *DM[ADDR]*, αντί του απλού *ADDR* (και γι' αυτό ανάβει το dmRd αντί του im2bus): η νέα τιμή του PC είναι η (εν δυνάμει μεταβλητή) τιμή *DM[ADDR]* που διαβάζουμε από την μνήμη δεδομένων (στη θέση που μας λέει η εντολή, μέσω του *ADDR*), αντί της σταθερής τιμής *ADDR* που διαβάζουμε από την μνήμη εντολών.

΄Ενα άλλο παράδειγμα χρήσης της νέας εντολής έμμεσου άλματος έχουμε στις πρώτες δύο θέσεις της μνήμης εντολών του υπολογιστή μας, όπως τις είδαμε στην §11.11: ο υπολογιστής μας αρχίζει πάντα να εκτελεί εντολές από τη διεύθυνση 00, όπου βρίσκει σαν δύο πρώτες εντολές τις "input 10; jmpx 10" (στο εργαστήριο 11.11 εκτελούσατε την jmpx σαν απλή jmp, αλλά εδώ τα πράγματα αλλάζουν). Η πρώτη εντολή, input 10, παραλαμβάνει έναν αριθμό από την περιφερειακή συσκευή (πληκτρολόγιο), και τον γράφει στη θέση 10 της μνήμης δεδομένων (DM[10]). Η δεύτερη εντολή, jmpx 10, εκτελεί ένα (έμμεσο!) άλμα --όχι στη διεύθυνση 10, αλλά στη διεύθυνση που διαβάζει από την παραπάνω θέση 10 της μνήμης δεδομένων, DM[10]. ΄Ετσι τελικά, σαν τρίτη εντολή εκτελείται η εντολή από τη διεύθυνση που ο χρήστης έδωσε από το πληκτρολόγιο, δηλαδή μιά εντολή που μπορεί να αλλάζει κάθε φορά που εκτελείται το πρόγραμμα! Στο εργαστήριο, όταν θέλετε να εκτελέσετε το πρόγραμμα της §12.1, πληκτρολογήστε "0020" (δεκαεξαδικό) όταν ανάβει ο υπολογιστής: η πρώτη εντολή "input 10" θα προκαλέσει "DM[10] := 20", και η δεύτερη εντολή "jmpx 10" θα προκαλέσει "PC := DM[10] = 20")· έτσι, σαν τρίτη εντολή θα εκτελεστεί η εντολή 20, που είναι η πρώτη εντολή του προγράμματος §12.1. Αν πάλι θέλετε να εκτελέσετε το πρόγραμμα της παρακάτω §12.5, που αρχίζει στη διεύθυνση 30 (δεκαεξαδικό) της μνήμης εντολών, πληκτρολογήστε "0030" όταν ανάβει ο υπολογιστής: η "input 10" θα κάνει "DM[10] := 30", και η "jmpx 10" θα κάνει "PC := DM[10] = 30".

Άσκηση (Πείραμα) 12.4 Εντολές Διακλάδωσης

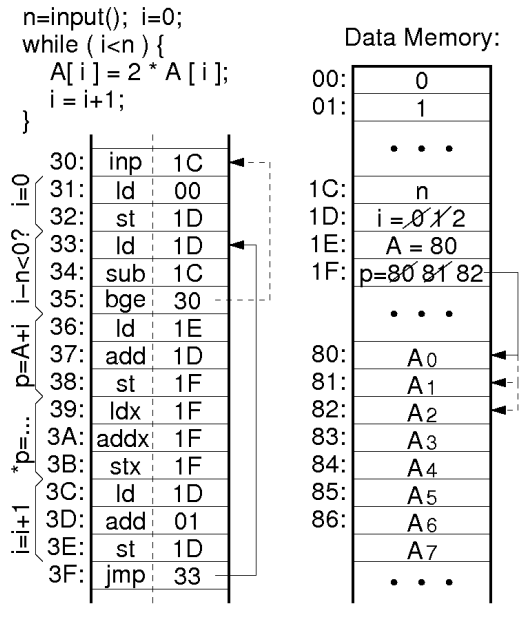
Μεταφράστε τον παραπάνω πίνακα αληθείας της παραγράφου 12.2 σε (απλοποιημένες) εξισώσεις Boole που περιγράφουν κάθε έξοδο του κυκλώματος ελέγχου (δηλ. κάθε σήμα ελέγχου) σαν συνάρτηση των εισόδων του κυκλώματος ελέγχου, opcode, accZero, και accSign. Παρατηρήστε ότι το σήμα pcMd μπορεί να εκφραστεί σαν: [(opcode=x111)] OR [(opcode=100x) AND (opcode[0] XOR accZero)] OR [(opcode=101x) AND (opcode[0] XOR accSign)], όπου XOR είναι το αποκλειστικό-΄Η, opcode[0] είναι το LS bit του opcode, "opcode=x111" παριστάνει τη συνάρτηση KAI των τριών LS bits του opcode ή των συμπληρωμάτων τους που ανάβει όποτε ο opcode τελειώνει σε 111, "opcode=100x" παριστάνει την συνάρτηση KAI των τριών MS bits του opcode ή των συμπληρωμάτων τους που ανάβει όποτε ο opcode αρχίζει με 100, και αντίστοιχα γιά "opcode=101x" όποτε αυτός αρχίζει με 101.

Στο εργαστήριο, εάν μεν προλαβαίνετε να κατασκευάσετε το κύκλωμα της §12.7, τότε προσπεράστε αυτό εδώ το κύκλωμα (που είναι υποσύνολο εκείνου). Αλλοιώς, κατασκευάστε απλά και μόνο αυτό εδώ το κύκλωμα ελέγχου και δοκιμάστε το με τα προγράμματα από τις διευθύνσεις 10 (§11.11) και 20 (§12.1). Πηγαίνετε στο πρόγραμμα που επιθυμείτε να εκτελέσετε με την μέθοδο που εξηγήσαμε στο τέλος της §12.3.

12.5 Δομές Δεδομένων: Μεταβλητές Διευθύνσεις Μνήμης

Σε όλες τις εντολές πράξεων (load, store, add, sub,...) που είδαμε μέχρι τώρα, οι διευθύνσεις των δεδομένων καθορίζονταν οριστικά και αμετάκλητα από το πρόγραμμα: ήταν γραμμένες μέσα του (στη μνήμη εντολών, όπου δεν αλλάζουν οι πληροφορίες την ώρα που τρέχει το πρόγραμμα). Μιά τέτοια εντολή κάνει πάντα την πράξη της στην ίδια μεταβλητή --στην ίδια θέση (διεύθυνση) της μνήμης δεδομένων. Με τέτοιες εντολές και μόνο δεν μπορούμε να επεξεργαστούμε *δομές δεδομένων* (πίνακες, λίστες, δένδρα, κλπ): δεν μπορούμε να γράψουμε ένα κομάτι προγράμματος μέσα σ' ένα βρόχο που κάθε φορά που θα επανεκτελείται να προσπελαύνει *διαφορετικά* στοιχεία της δομής δεδομένων, σε διαφορετικές διευθύνσεις της μνήμης δεδομένων κάθε φορά. Γιά να αποκτήσει ο υπολογιστής μας και αυτή τη δυνατότητα --την τελευταία που μας μένει γιά να γίνει πραγματικός υπολογιστής!-- χρειάζεται δυό-τρείς εντολές ακόμα, όπως θα αναλύσουμε σε αυτή την παράγραφο.

Στο παράδειγμα δεξιά, υπάρχει ένας πίνακας (array) από αριθμούς, A_0, A_1, A_2, A_3,... αποθηκευμένος στις διευθύνσεις μνήμης (δεδομένων) 80, 81, 82, 83,.... Παρατηρήστε ότι το στοιχείο A_i βρίσκεται στη διεύθυνση *(80+i)*. Ο αριθμός A = 80 λέγεται *διεύθυνση βάσης* του πίνακα. Ας πούμε ότι θέλουμε να διπλασιάσουμε τα *n* πρώτα στοιχεία αυτού του πίνακα, A_0, ..., A_{n-1}, όπου *n* είναι ένας αριθμός που δίδεται σαν είσοδος. Αυτό θα το κάνουμε με το βρόχο που φαίνεται γραμμένος σε γλώσσα C στο σχήμα πάνω αριστερά· το πρόγραμμα αυτό βρίσκεται γραμμένο στις θέσεις 30 - 3C (μνήμη εντολών) του υπολογιστή στο εργαστήριο, και φαίνεται δεξιά σε γλώσσα Assembly. Η εντολή `input 1C` (θέση 30) διαβάζει την τιμή *n* και την γράφει στη θέση 1C της μνήμης δεδομένων. Οι δύο επόμενες εντολές αρχικοποιούν τη μεταβλητή *i* του προγράμματος, που κρατιέται στη διεύθυνση 1D, χρησιμοποιώντας την τιμή 0 από τη θέση 00. Την μεταβλητή *i* θα χρησιμοποιήσουμε σαν μετρητή του βρόχου μας, γιά να επεξεργαζόμαστε κάθε φορά το στοιχείο A_i του πίνακα.



Ο βρόχος αρχίζει στη διεύθυνση 33, ελέγχοντας αν το *i* είναι μικρότερο του *n*· αν δεν είναι ο βρόχος δεν εκτελείται άλλο (γιά *n* αρνητικό ή μηδέν, δεν θα εκτελεστεί ούτε μιά φορά). Η σύγκριση γίνεται μέσω της ισοδύναμης έκφρασης *i-n<0*: αφού υπολογιστεί το *i-n*, φεύγουμε από το βρόχο όταν η έκφραση αυτή είναι ψευδής, δηλαδή όταν *i-n* είναι μεγαλύτερο ή ίσο με μηδέν (εντολή `bge`)· η έξοδος από το βρόχο, εδώ, είναι επιστροφή στην αρχή του προγράμματος γιά να ζητηθεί νέα τιμή *n*. Όταν μπούμε στο βρόχο, αρχικά υπολογίζουμε την διεύθυνση *p* του στοιχείου A_i του πίνακα, διότι μόνον αν ξέρουμε τη διεύθυνσή του μπορούμε να το βρούμε και να το επεξεργαστούμε· η διεύθυνση *p* του δεδομένου A_i λέγεται *pointer* (δείκτης) σε αυτό το στοιχείο. Όπως παρατηρήσαμε παραπάνω, η διεύθυνση αυτή είναι *p=(80+i)*, και την υπολογίζουμε με τις εντολές 36-38, οι οποίες και την γράφουν στη θέση 1F της μνήμης δεδομένων. Κατά την πρώτη επανάληψη του βρόχου, *i=0* και *p=80* δείχνει στο πρώτο στοιχείο, A_0· στην δεύτερη επανάληψη, *i=1* και *p=81* δείχνει στο A_1· την τρίτη φορά, όπως στο σχήμα, *i=2* και *p=82* δείχνει το A_2.

Οι τρείς νέες εντολές του υπολογιστή μας φαίνονται στις θέσεις 39-3B του προγράμματος. Ενώ η απλή εντολή `ld ADDR` κάνει `ACC:=DM[ADDR]` (δηλαδή φορτώνει τον καταχωρητή από τη θέση `ADDR`), η νέα εντολή, *load indexed*, `ldx ADDR`, κάνει `ACC:=DM[DM[ADDR]]`, δηλαδή φορτώνει τον καταχωρητή από τη θέση της μνήμης δεδομένων την οποία ορίζει η θέση `ADDR` --όπως λέμε, έχουμε μιαν *έμμεση πρόσβαση* (indirect access). Στο παράδειγμά μας, ενώ η απλή εντολή `ld 1F` θα έφερνε στον συσσωρευτή τα δεδομένα της θέσης 1F, δηλαδή τον αριθμό 82, η νέα εντολή `ldx 1F` διαβάζει από τη θέση 1F τον αριθμό 82, και χρησιμοποιεί αυτό τον αριθμό σα διεύθυνση του πραγματικού δεδομένου γιά να φέρει τελικά στο συσσωρευτή τον αριθμό A_2 από τη θέση 82. Εάν η εντολή ήταν μιά απλή *load*, τότε κάθε φορά που θα επαναλαμβάνονταν ο βρόχος, πάντα η εντολή αυτή θα διάβαζε την *ίδια θέση* μνήμης, 1F. Τώρα που η εντολή είναι *load indexed*, σε κάθε επανάληψη του βρόχου η θέση μνήμης που τελικά διαβάζουμε *αλλάζει*: στην αρχή ήταν η 80, μετά η 81, ύστερα η 82, κ.ο.κ. έτσι πετυχαίνουμε να επεξεργαζόμαστε διαφορετικό στοιχείο της δομής δεδομένων κάθε φορά!

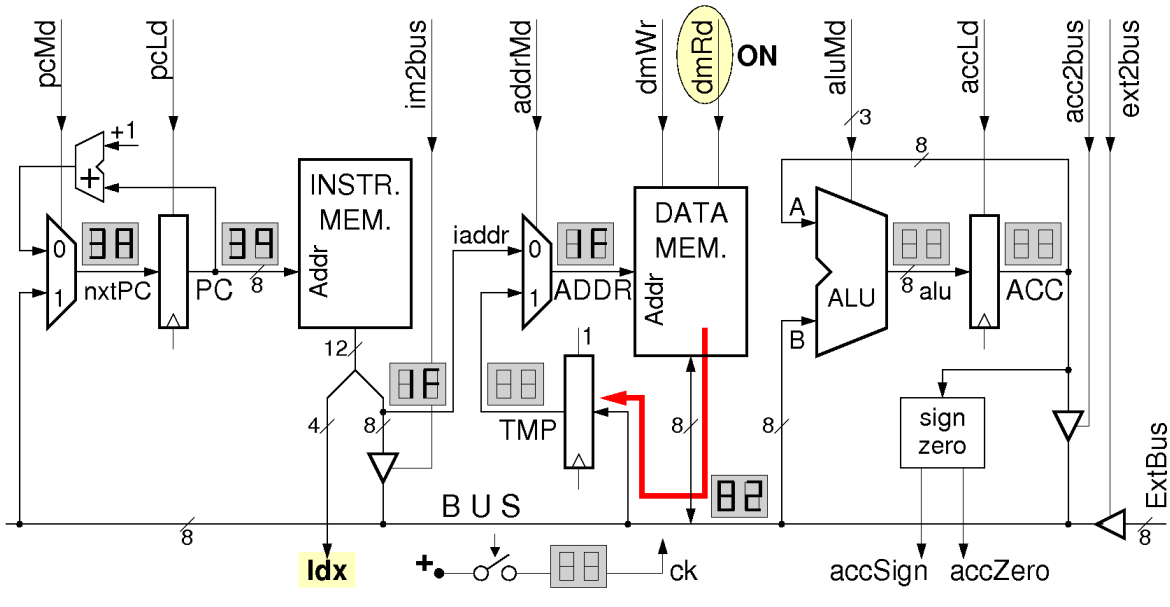
Κατ' αναλογία προς την `ldx 1F`, η *add indexed* `addx 1F` διαβάζει από τη θέση 1F τον αριθμό 82, και χρησιμοποιεί αυτό τον αριθμό σα διεύθυνση του πραγματικού δεδομένου γιά να προσθέσει τελικά στο συσσωρευτή τον αριθμό A_2 από τη θέση 82. Αφού ο συσσωρευτής περιείχε ήδη τον αριθμό A_2, το άθροισμα που προκύπτει ισούται με το επιθυμητό διπλάσιο, $2*A_2$. Κατ' αναλογία προς τις `ldx` και `addx`, η *store indexed* `stx 1F` διαβάζει από τη θέση 1F τον αριθμό 82, και χρησιμοποιεί αυτό τον αριθμό σα διεύθυνση της πραγματικής θέσης όπου θα αποθηκεύσει το περιεχόμενο του συσσωρευτή· έτσι, στο παράδειγμά μας, η `stx 1F` θα γράψει τελικά στη θέση 82 (στην προηγούμενη ανακύκλωση είχε γράψει στη θέση 81, και στην προ-προηγούμενη στη θέση 80). Άρα, τελικά, το αρχικό A_2 στη θέση 82 αντικαθίσταται από το διπλάσιό του. Ο βρόχος ολοκληρώνεται αυξάνοντας το *i* κατά 1, και επιστρέφοντας στην εντολή 33 γιά να συγκρίνουμε τη νέα τιμή του *i* με

το *n*, κ.ο.κ.

Όπως είδαμε, οι νέες εντολές μεταφέρουν μεταξύ μνήμης και συσσωρευτή όχι το περιεχόμενο μιάς σταθερής --πάντα της ίδιας-- θέσης (διεύθυνσης) μνήμης, αλλά το περιεχόμενο μιάς **μεταβλητής** θέσης μνήμης --μιάς θέσης που τη διεύθυνσή της να μπορεί να την υπολογίζει και να την αλλάζει κατά βούληση το ίδιο το πρόγραμμα την ώρα που τρέχει! Γιά να κάνουμε και αφαιρέσεις και λογικές πράξεις με στοιχεία δομών δεδομένων, θα μας βόλευαν και οι αντίστοιχες εντολές subx, andx, norx. Πάντως, μπορούμε να κάνουμε και χωρίς αυτές (ακομή και χωρίς την `addx`), αρκεί να έχουμε τις `ldx` και `stx`: Αντιγράφουμε το επιθυμητό στοιχείο της δομής δεδομένων σε μιά σταθερή θέση tmp (π.χ. `ldx p; st tmp`), κάνουμε τις πράξεις που θέλουμε πάνω σε τέτοιες σταθερές προσωρινές θέσεις (π.χ. `ld tmp2; sub tmp`), και στο τέλος στέλνουμε το αποτέλεσμα στη θέση της δομής δεδομένων που θέλουμε (π.χ. `stx p`).

12.6 Υλοποίηση Εντολών Indexed

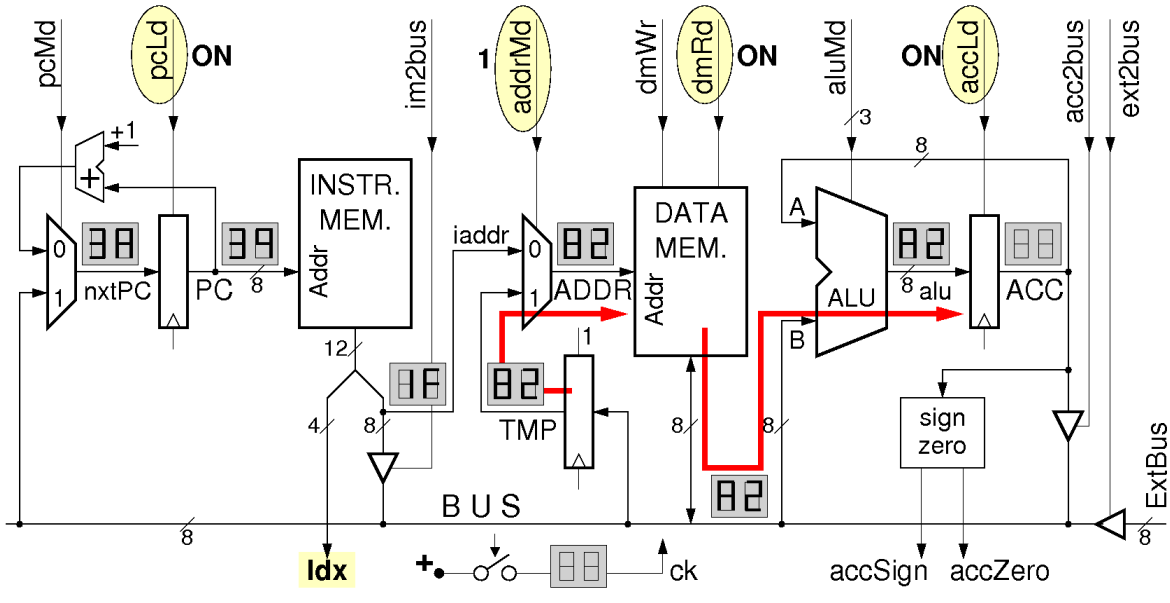
Γιά να μπορεί να εκτελεί τις εντολές `ldx`, `stx`, και `addx`, ο υπολογιστής μας χρειάζεται τα δύο κυκλώματα στο μέσον του datapath που δεν είχαν χρησιμοποιηθεί μέχρι στιγμής. Πρώτον, χρειάζεται ένα πολυπλέκτη στην είσοδο διευθύνσεων της μνήμης δεδομένων, προκειμένου η διεύθυνση αυτή να μπορεί να προέρχεται είτε από την εντολή --γιά τις απλές περιπτώσεις-- είτε από ένα δεδομένο που διαβάσαμε από την ίδια τη μνήμη δεδομένων --γιά τη δεύτερη φάση των εντολών indexed. Δεύτερον, επειδή πρέπει να γίνουν δύο αναγνώσεις από τη μνήμη δεδομένων, η μία μετά την άλλη, χρειαζόμαστε και έναν καταχωρητή, *TMP*, που να κρατάει το αποτέλεσμα της πρώτης ανάγνωσης διαθέσιμο γιά τη δεύτερη.



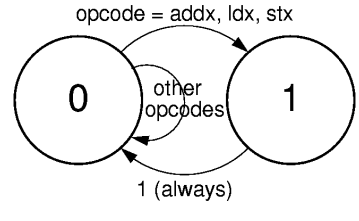
Γιά τον τελευταίο αυτό λόγο, οι εντολές indexed δεν μπορούν να εκτελεστούν σε ένα μόνο κύκλο ρολογιού, όπως οι υπόλοιπες εντολές: χρειάζεται μιά ενδιάμεση ακμή ρολογιού η οποία να τερματίσει την πρώτη ανάγνωση από τη μνήμη δεδομένων και να ξεκινήσει την δεύτερη. Έτσι, και το κύκλωμα ελέγχου δεν μπορεί πιά να είναι συνδυαστικό: χρειάζεται και 1 bit κατάστασης που να μας πληροφορεί αν τώρα βρισκόμαστε στον πρώτο ή στο δεύτερο κύκλο της εκτέλεσης. Στο πρώτο διάγραμμα, πιό πάνω, φαίνονται οι πράξεις και μεταφορές καταχωρητών που πρέπει να γίνουν στο datapath κατά τον πρώτο κύκλο εκτέλεσης των τριών νέων εντολών. Πρέπει να γίνει ανάγνωση από τη μνήμη δεδομένων (ενεργοποίηση *dmRd*), αλλά το προϊόν αυτής της ανάγνωσης είναι ένα ενδιάμεσο αποτέλεσμα --μία νέα διεύθυνση-- και όχι το τελικό αποτέλεσμα της εντολής. Γιά το λόγο αυτό η έξοδος της μνήμης δεδομένων, από το BUS, οδηγείται να αποθηκευτεί στον προσωρινό καταχωρητή *TMP* (η είσοδος επίτρεψης φόρτωσης, tmpLd, αυτού του καταχωρητή είναι πάντα ενεργοποιημένη, επειδή το περιεχόμενο του *TMP* το καταναλώνουμε πάντα μέσα στον πρώτο κύκλο μετά το φόρτωμά του, άρα δεν έχουμε ανάγκη αυτό να διατηρείται γιά μακρότερο). Το *addrMd* πρέπει να έχει την κανονική του τιμή (0), διότι η πρώτη αυτή ανάγνωση από την μνήμη δεδομένων γίνεται από την θέση που υποδεικνύει το πεδίο *ADDR* της εντολής. Πρέπει να κρατηθούν σβηστά τα *pcLd* και *accLd* διότι δεν πρέπει να πειραχτούν οι αντίστοιχοι καταχωρητές: ο PC δεν πρέπει να αλλάξει διότι δεν τέλειωσε η εκτέλεση της εντολής ακόμα (μόλις αλλάξει ο PC αλλάζει και ο opcode στην έξοδο της μνήμης εντολών), και ο ACC δεν πρέπει να αλλάξει διότι δεν είμαστε ακόμη έτοιμοι

να κάνουμε την επιθυμητή πράξη πάνω του (π.χ. πρόσθεση νέων δεδομένων σε αυτόν (`addx`), ή αποθήκευσή του (`stx`)).

Στο δεύτερο διάγραμμα, παρακάτω, φαίνονται οι λειτουργίες που πρέπει να γίνουν κατά τον δεύτερο κύκλο εκτέλεσης μιάς εντολής `ldx`. Πρέπει να γίνει η δεύτερη ανάγνωση από τη μνήμη δεδομένων --αυτή τη φορά των δεδομένων αυτών καθ'εαυτών που θέλουμε να φορτώσουμε στο συσσωρευτή. Η ανάγνωση αυτή *δεν* γίνεται από τη διεύθυνση *ADDR* που περιέχει η εντολή, αλλά από τη διεύθυνση *TMP* που διαβάσαμε κατά τον πρώτο κύκλο από τη μνήμη δεδομένων· γιά να γίνει αυτή η επιλογή πρέπει να σήμα *addrMd* να πάρει την μη συνήθη τιμή του, 1. Τα δεδομένα που διαβάζονται πρέπει να περάσουν στο συσσωρευτή κατά τρόπο εντελώς ανάλογο με την απλή εντολή `ld`: *dmRd*=1, *aluMd*=`passB`, *accLd*=1. Επίσης, πρέπει να ανάψουμε το σήμα *pcLd*, αφού αυτός είναι ο τελευταίος κύκλος εκτέλεσης της εντολής: στον επόμενο κύκλο, ο PC πρέπει να έχει τη νέα του τιμή, γιά να εκτελεστεί η επόμενη εντολή.



Γιά τις εντολές `addx`, οι λειτουργίες κατά τον δεύτερο κύκλο εκτέλεσης είναι παρόμοιες: το μόνο που αλλάζει είναι το *aluMd*=`add`. Γιά τις εντολές `stx`, οι λειτουργίες του δεύτερου κύκλου εκτέλεσης θυμίζουν εκείνες της απλής `st`: ο ACC οδηγεί το BUS (*acc2bus*=1) και τα δεδομένα αυτά γράφονται στη μνήμη δεδομένων (*dmWr*=1). Η διαφορά από την απλή `st` είναι στο *addrMd*: εκεί ήταν 0, οπότε γράφαμε στη διεύθυνση *ADDR* που περιέχει η εντολή, ενώ εδώ είναι 1, οπότε γράφουμε στη διεύθυνση *TMP* που διαβάσαμε κατά τον πρώτο κύκλο από την ίδια τη μνήμη δεδομένων. Θυμηθήτε ότι, και γιά τις τρείς `ldx`, `addx`, `stx`, ο πρώτος κύκλος εκτέλεσης είναι ίδιος.



Όπως είπαμε παραπάνω, αφού μερικές εντολές του τελικού υπολογιστή μας χρειάζονται δύο κύκλους ρολογιού γιά την εκτέλεσή τους, το κύκλωμα ελέγχου δεν μπορεί πιά να είναι συνδυαστικό: χρειάζεται και 1 bit κατάστασης που να μας πληροφορεί αν τώρα βρισκόμαστε στον πρώτο ή στο δεύτερο κύκλο της εκτέλεσης. Η FSM του κυκλώματος ελέγχου που προκύπτει θα έχει το διάγραμμα καταστάσεων που φαίνεται δεξιά. Έστω *S* το bit κατάστασης της FSM. Όταν *S*=0 θα σημαίνει ότι βρισκόμαστε στον πρώτο ή και μοναδικό κύκλο εκτέλεσης της εντολής. Μετά από έναν κύκλο *S*=0, μόνον οι εντολές `ldx`, `addx`, `stx` πηγαίνουν στην κατάσταση *S*=1, δηλαδή σε δεύτερο κύκλο εκτέλεσης --όλες οι άλλες παραμένουν στην κατάσταση *S*=0, δηλαδή στον πρώτο κύκλο εκτέλεσης, φυσικά της επόμενης εντολής. Οι εντολές `ldx`, `addx`, `stx`, μετά την κατάσταση *S*=1, πηγαίνουν πάντα στην *S*=0, δηλαδή στον πρώτο κύκλο εκτέλεσης --φυσικά της επόμενης εντολής.

Κανονικά, χρειαζόμαστε ένα σήμα αρχικοποιήσης (reset) της FSM, που να την φέρνει στην κατάσταση *S*=0. Γιά λόγους απλοποίησης δεν θα χρησιμοποιήσουμε τέτοιο σήμα, βασιζόμενοι στο ότι αν η FSM εκκινήσει σε *S*=1, τότε πάντοτε τον επόμενο κύκλο θα μεταβεί σε *S*=0, σύμφωνα με το διάγραμμα μετάβασης καταστάσεών της. Επίσης, πάλι γιά απλοποίηση, θα κάνουμε τις εντολές που εκτελούνται σε ένα κύκλο να αγνοούν την τιμή του bit κατάστασης, *S*. Έτσι, εάν η εντολή που εκτελείται τον πρώτο κύκλο ρολογιού δεν είναι `ldx`, `addx`, `stx`, αυτή θα εκτελεστεί σωστά ακόμα κι

αν το bit κατάστασης *S*=1 κατά τον πρώτο εκείνο κύκλο, λόγω έλλειψης reset. Σε μας αυτό εξασφαλίζεται, γιατί η εντολή στη θέση 00 είναι η `input 10`. Φυσικά, το σήμα reset δεν μπορούμε να το γλυτώσουμε εντελώς, διότι πρέπει κάποιος να αρχικοποιήσει τον PC=0· γιά απλοποίηση των κυκλωμάτων σας, αυτό το κάνει μόνη της η πλακέτα του υπλογιστή, με το δικό της κουμπί reset (αριστερά κάτω) ή μόλις ανάβει η τροφοδοσία.

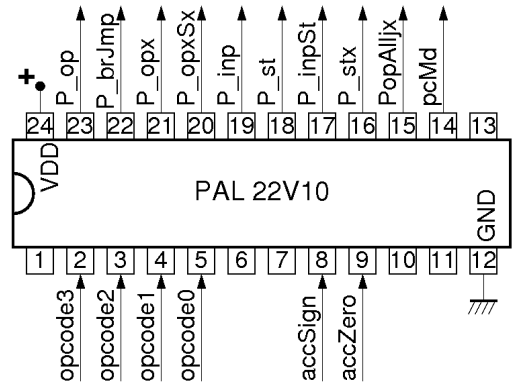
Βάσει των παραπάνω, ο πίνακας αληθείας της §12.2 επεκτείνεται με τις 3 νέες εντολές όπως φαίνεται παρακάτω. Αυτό είναι και το πλήρες κύκλωμα ελέγχου του υπολογιστή μας. Η είσοδος *S* είναι το bit κατάστασης· η έξοδος *nS* είναι η επόμενη κατάσταση. Το σήμα *pcMd* γιά τις εντολές διακλάδωσης εμφανίζεται σαν απλός αστερίσκος (*), παραπέμποντας στον πλήρη ορισμό του στην παραπάνω §12.2 ή στήν ισοδύναμη εξίσωση στην §12.4· γιά τον ίδιο λόγο, οι είσοδοι *accZero, accSign* έχουν παραληφθεί από τον πίνακα. Παρατηρήστε ότι το σήμα *pcLd* είναι το συμπλήρωμα του *nS* (φορτώνουμε πάντα τον PC αμέσως πριν μεταβούμε στην κατάσταση 0 γιά να εκτελέσουμε μιάν επόμενη εντολή).

opcode:		*Λειτουργία:*	*addrMd*	*aluMd*		*acc2bus*	*dmWr*	*pcMd*			
	S:		*nS:*	*dmRd*		*accLd*	*ext2bus*		*im2bus*		*pcLd*
0000 (add)	x	ACC:=ACC+DM[A]	0	0	1	000	1	0	0	0	0	0	1
0001 (sub)	x	ACC:=ACC-DM[A]	0	0	1	001	1	0	0	0	0	0	1
0010 (and)	x	ACC:=ACCandDM[A]	0	0	1	010	1	0	0	0	0	0	1
0011 (nor)	x	ACC:=ACCnorDM[A]	0	0	1	011	1	0	0	0	0	0	1
0100 (inp)	x	DM[A]:=IO_BUS	0	0	0	xxx	0	0	1	1	0	0	1
0101 (ld)	x	ACC:=DM[A]	0	0	1	1xx	1	0	0	0	0	0	1
0110 (st)	x	DM[A]:=ACC	0	0	0	xxx	0	1	0	1	0	0	1
0111 (jmp)	x	PC := A	0	0	0	xxx	0	0	0	0	1	1	1
1000 (beq)	x	PC := A or PC+1	0	0	0	xxx	0	0	0	0	1	(*)	1
1001 (bne)	x	PC := A or PC+1	0	0	0	xxx	0	0	0	0	1	(*)	1
1010 (blt)	x	PC := A or PC+1	0	0	0	xxx	0	0	0	0	1	(*)	1
1011 (bge)	x	PC := A or PC+1	0	0	0	xxx	0	0	0	0	1	(*)	1
1100 (addx)	0	TMP:=DM[A]	1	0	1	xxx	0	0	0	0	0	x	0
1100	1	ACC:=ACC+DM[TMP]	0	1	1	000	1	0	0	0	0	0	1
1101 (ldx)	0	TMP:=DM[A]	1	0	1	xxx	0	0	0	0	0	x	0
1101	1	ACC:= DM[TMP]	0	1	1	1xx	1	0	0	0	0	0	1
1110 (stx)	0	TMP:=DM[A]	1	0	1	xxx	0	0	0	0	0	x	0
1110	1	DM[TMP]:=ACC	0	1	0	xxx	0	1	0	1	0	0	1
1111 (jmpx)	x	PC := DM[A]	0	0	1	xxx	0	0	0	0	0	1	1

Πείραμα 12.7 Συνολικό Κύκλωμα Ελέγχου

Σχεδιάστε το πλήρες κύκλωμα ελέγχου, βασισμένοι στον παραπάνω πίνακα αληθείας του. Επειδή το κύκλωμα αυτό είναι πολύ μεγάλο γιά να το φτιάξετε όλο με τις λίγες απλές πύλες AND και OR των 2 εισόδων που έχετε στα chips σας, θα βρείτε στις πλακέτες (breadboard) ένα επιπλέον chip που θα περιέχει έτοιμες αρκετές από τις λογικές συναρτήσεις που θα σας χρειαστούν. Πρόκειται γιά το chip PAL CE 22V10 H-15PC/4, που είναι ένα chip προγραμματιζόμενης λογικής (PAL - Programmable Array Logic). Οι ακροδέκτες της PAL αυτής, με τον τρόπο που εσείς θα τις χρησιμοποιήστε, φαίνονται στο πιό κάτω σχήμα. Θα τροφοδοτήστε στην PAL 6 εισόδους: τα 4 bits του opcode και τα 2 bits ελέγχου προσήμου/μηδενισμού του συσσωρευτή. Τα pins 1, 6, 7, 10, 11, και 13 είναι αχρησιμοποιήτες είσοδοι, και μπορείτε να τα αφήσετε ασύνδετα επειδή η PAL έχει προγραμματιστεί να αγνοεί την τιμή τους. Η PAL γεννά 10 εξόδους γιά σας, τις εξής:

- **`P_op`**: ανάβει όποτε ο opcode υποδεικνύει εντολή πράξης (*op*eration), απλή (όχι indexed), η οποία μεταβάλει το συσσωρευτή, δηλαδή όταν ο opcode είναι ένας από τους { add, sub, and, nor, ld }, ή αλλοιώς όποτε opcode==00xx ή opcode==0101.
- **`P_brJmp`**: ανάβει όποτε ο opcode υποδεικνύει εντολή διακλάδωσης ή απλού (όχι indexed) άλματος, δηλαδή όταν ο opcode είναι ένας από τους { beq, bne, blt, bge, jmp }, ή αλλοιώς όποτε opcode==10xx ή opcode==0111.
- **`P_opx`**: ανάβει όποτε ο opcode υποδεικνύει εντολή πράξης indexed η οποία μεταβάλει το συσσωρευτή, δηλαδή όταν ο opcode είναι ένας από τους { addx, ldx }, ή αλλοιώς όποτε opcode==110x
- **`P_opxSx`**: ανάβει όποτε ο opcode υποδεικνύει εντολή πράξης ή αποθήκευσης indexed, δηλαδή



όταν ο opcode είναι ένας από τους { addx, ldx, stx }, ή αλλοιώς όποτε opcode==110x ή opcode==1110.

- **`P_inp`**: ανάβει όταν και μόνον όταν opcode==0100, δηλαδή είναι η εντολή *input*.
- **`P_st`**: ανάβει όταν και μόνον όταν opcode==0110, δηλαδή είναι η εντολή *store*.
- **`P_inpSt`**: ανάβει όταν opcode==01x0, δηλαδή όταν η εντολή είναι *input* ή *store*.
- **`P_stx`**: ανάβει όταν και μόνον όταν opcode==1110, δηλαδή είναι η εντολή *store indexed (stx)*.
- **`PopAlljx`**: ανάβει όποτε ο opcode υποδεικνύει εντολή πράξης, απλής ή indexed, η οποία μεταβάλει το συσσωρευτή, ή υποδεικνύει την εντολή jump indexed, δηλαδή όταν ο opcode είναι ένας από τους { add, sub, and, nor, ld, addx, ldx, jmpx }, ή αλλοιώς όποτε ο opcode είναι 00xx ή x101 ή 110x ή 11x1.
- **`pcMd`** = [(opcode==x111)] OR [(opcode==100x) AND (opcode[0] XOR accZero)] OR [(opcode==101x) AND (opcode[0] XOR accSign)], δηλαδή είναι έτοιμο το σήμα που πρέπει να ελέγξει τον πολυπλέκτη στην είσοδο του PC.

Στο εργαστήριο, κατασκευάστε το κύκλωμα ελέγχου του απλού υπολογιστή, χρησιμοποιώντας την παραπάνω PAL καθώς και δικά σας chips γιά το (ένα) flip-flop κατάστασης, γιά έναν αντιστροφέα γιά το bit κατάστασης, και γιά μερικές πύλες AND και OR που θα χρειαστείτε. Θα χρειαστείτε ένα **κοινό ρολόϊ** που να τροφοδοτεί τόσο το δικό σας flip-flop κατάστασης όσο και το datapath του απλού υπολογιστή. Χρησιμοποιήστε το διακόπτη A, από την πλακέτα εισόδων/εξόδων, δεξιά, γιά το σκοπό αυτό. Την έξοδο αυτού του διακόπτη, από την δεξιά καλωδιοταινία, δώστε την τόσο στο δικό σας chip με το flip-flop κατάστασης όσο και στο datapath του υπολογιστή μέσω της αριστερής καλωδιοταινίας. Γιά τον τελευταίο αυτό σκοπό, τροφοδοτήστε το σήμα από τον διακόπτη A στον ακροδέκτη **clock2p** (τελευταίος δεξιά) της καλωδιοταινίας από/προς το datapath. Εσωτερικά, ο απλός υπολογιστής χρησιμοποιεί σαν ρολόϊ του το λογικό OR αυτής της εισόδου **clock2p** και του δικού του διακόπτη· άρα τελικά, πατώντας μόνο τον διακόπτη A και όχι τον διακόπτη ρολογιού του datapath, θα έχετε ένα ισοδύναμο ρολόϊ γιά το datapath που ταυτίζεται με την έξοδο του διακόπτη A.

Ελέγξτε το κύκλωμά σας και τον υπολογιστή εκτελώντας τα παραπάνω προγράμματα στις διευθύνσεις 20 και 30 γιά διάφορες τιμές εισόδου. Ο πίνακας στοιχείων A_0, A_1, ..., A_{127} στις διευθύνσεις 80, 81, ..., FF της μνήμης δεδομένων αρχικοποιείται γιά σας σε κάθε reset στις τιμές: 01, FF, 02, FE, 03, FD, 04, FC, ..., 3F, C1, 40, C0, κατά σειρά.

Προσομοιωτής του Datapath: *Προαιρετικά και εάν σας ενδιαφέρει* μπορείτε να βρείτε έναν προσομοιωτή του datapath του απλού υπολογιστή των Εργαστηρίων 11 και 12, ευγενή συνεισφορά του κ. Εμμανουήλ Α. Νεονάκη, βασισμένο στο σύστημα *Hamburg Design System (HADES)* (tams-www.informatik.uni-hamburg.de/applets/hades/html/hades.html), στο directory **`~hy120/dpath_hades_EAN11/`** (δηλαδή: `/home/misc/courses/hy120/dpath_hades_EAN11/`) των υπολογιστών του Τμήματος (μπορείτε να κατεβάσετε όλα τα περιεχόμενα αυτού του directory αντιγράφοντας και αποσυμπιέζοντας το αρχείο: `~hy120/dpath_hades_EAN11.zip`). Το πρόγραμμα Java που πρέπει να ανοίξετε και εκτελέσετε είναι το `hades.jar` και στη συνέχεια από το μενού του hades: File/Open επιλέξτε το αρχείο DataPath.hds

Επιστροφή των Chips του Εργαστηρίου: Μετά τη λήξη του εργαστηρίου 12 υποχρεούστε να επιστρέψετε το σακκουλάκι με τα chips που σας δόθηκε γιά τις εργαστηριακές ασκήσεις αυτού του μαθήματος. Επιστρέψτε τα στον κ. Νίκο Κρασσά, τηλ. 2810.393595, στο γραφείο B217, ώρες **9:30 - 14:30**, μέχρι την Παρασκευή 11 Ιανουαρίου 2019. Η επιστροφή των chips είναι απαραίτητη προϋπόθεση γιά να περάσετε το μάθημα.

12.8 Ταχύτητα και Κατανάλωση Ενέργειας των Κυκλωμάτων CMOS:

Βιβλίο Dally: το **κεφάλαιο 5** (σελ. 88-113) είναι εξαιρετικό και περιέχει την ύλη αυτής της παραγράφου με πολύ-πολύ περισσότερα στοιχεία και πληροφορίες –διαβάστε το εγκυκλοπαιδικά και μόνο, εάν ενδιαφέρεστε. Τα άλλα βιβλία δεν καλύπτουν το θέμα αυτό.

Η κυριότερη πηγή καθυστερήσεων στα μικροηλεκτρονικά chips είναι ο χρόνος Δt που ένα ρεύμα I χρειάζεται γιά να φορτίσει ή να εκφορτίσει μιά **παρασιτική χωρητικότητα** C αλλάζοντας την τάση της κατά ΔV: $\Delta t = \Delta Q / I = C \cdot \Delta V / I$ (όπου ΔQ είναι το ηλεκτρικό φορτίο που δίνουμε ή παίρνουμε από τη χωρητικότητα). Από την εξίσωση αυτή προκύπτει ότι γιά να έχουμε γρηγορότερα κυκλώματα, δηλαδή γιά να ελαττώσουμε την καθυστέρηση Δt της κάθε πύλης, πρέπει:

- Να ελαττώσουμε την παρασιτική χωρητικότητα "φορτίου" C που είναι συνδεδεμένη στην έξοδο της πύλης, και που επομένως η πύλη αυτή πρέπει να την φορτίζει και να την εκφορτίζει κάθε φορά που χρειάζεται να αλλάξει την τάση (λογική τιμή) της εξόδου της. Η χωρητικότητα φορτίου μιάς πύλης καθορίζεται από το πόσο πολλά και πόσο μεγάλα πράγματα συνδέονται στην έξοδό της. Όσο περισσότερες άλλες πύλες έχουμε συνδέσει στην έξοδό της ("fan-out"), τόσο μεγαλώνει αυτή η χωρητικότητα φορτίου. Επίσης, όσο μεγαλύτερα transistors έχουν αυτές οι άλλες πύλες, πάλι τόσο μεγαλώνει η χωρητικότητα φορτίου. Ακόμα, εάν υπάρχουν μακρυά καλώδια συνδεδεμένα στην έξοδό μας, και αυτά προσθέτουν χωρητικότητα. Άρα, γιά γρήγορα κυκλώματα, πρέπει κάθε πύλη να οδηγεί **λίγες** άλλες, **μικρές** πύλες, σε **κοντινές** αποστάσεις: *όσο μικρότερο και απλούστερο είναι ένα κύκλωμα, τόσο γρηγορότερα δουλεύει!*
- Να ελαττώσουμε τις αλλαγές τάσης ΔV της χωρητικότητας φορτίου της πύλης. Συνήθως, αυτές είναι αλλαγές μεταξύ των δύο λογικών επιπέδων, "0" και "1", δηλ. μεταξύ 0 Volt και της τάσης τροφοδοσίας. Έτσι, νεότερες γενιές chips έχουν χαμηλότερη τάση τροφοδοσίας (π.χ. 2.5 V, 1.8 V, 1.2 V) από τις παλαιότερες (αλλά δυστυχώς αυτό μειώνει και το ρεύμα I των transistors, κι έτσι το κάνουν κυρίως γιά μείωση της κατανάλωσης ενέργειας, και όχι τόσο γιά αύξηση ταχύτητας). Ένα μειονέκτημα της μείωσης αυτής είναι ότι τα κυκλώματα γίνονται πιό ευαίσθητα στο θόρυβο, π.χ. θόρυβος 1.5 Volt είναι αδιάφορος σε κυκλώματα των 5 Volt, ενώ είναι καταστροφικός σε κυκλώματα με τροφοδοσία 1.2 Volt.
- Να αυξήσουμε το ρεύμα I που η πύλη μπορεί να δώσει στην έξοδό της προκειμένου να κάνει τη χωρητικότητα φορτίου να αλλάξει τάση (λογική τιμή). Τό I καθορίζεται από τρείς παράγοντες:
  - Όσο λιγότερα transistors εν σειρά έχει το κύκλωμα μιάς πύλης, τόσο μεγαλύτερο ρεύμα μπορεί να περάσει μέσα από αυτά. Αν θυμηθούμε ότι μιά πύλη έχει τόσα transistors εν σειρά όσες και οι είσοδοί της, προκύπτει ότι γιά γρήγορα κυκλώματα πρέπει οι πύλες να έχουν **λίγες εισόδους** η κάθε μία (μικρό "fan-in"), άρα και πάλι *όσο απλούστερο είναι ένα κύκλωμα, τόσο γρηγορότερα δουλεύει!*
  - Όσο μεγαλύτερο είναι ένα transistor (όσο φαρδύτερο είναι το κανάλι του), τόσο μεγαλύτερο ρεύμα δίνει. Το κακό με κάθε μεγάλο transistor είναι ότι αυξάνει ανάλογα και η παρασιτική του χωρητικότητα, επομένως η *προηγούμενη* πύλη που οδηγεί αυτό το transistor βλέπει μεγαλύτερη χωρητικότητα φορτίου και γίνεται πιό αργή!
  - Όσο μεγαλύτερη είναι η τάση οδήγησης ενός transistor τόσο μεγαλύτερο ρεύμα δίνει αυτό –και μάλιστα ανάλογο προς το τετράγωνο της τάσης, στην περιοχή λειτουργίας που μας ενδιαφέρει περισσότερο. Από την άλλη, το μεγάλωμα της τάσης οδήγησης των transistors, δηλαδή της τάσης τροφοδοσίας, αυξάνει και το ΔV της εξόδου που λέγαμε παραπάνω –αλλά γραμμικά μόνον– άρα τελικά η ταχύτητα αυξάνει περίπου γραμμικά με την τάση τροφοδοσίας.

Εκτός από την ταχύτητα ενός κυκλώματος, το άλλο σημαντικό χαρακτηριστικό που μας απασχολεί είναι η κατανάλωση ενέργειας ανά μονάδα χρόνου (ηλεκτρική ισχύς): πόσο γρήγορα ξοδεύει την μπαταρία, ή πόσο πολύ ζεσταίνεται (πόσους ανεμιστήρες ή κλιματιστικά χρειάζεται). Συνήθως η σημαντικότερη αιτία κατανάλωσης των chips CMOS είναι η ενέργεια φόρτισης και εκφόρτισης των παρασιτικών χωρητικοτήτων που αυτά έχουν μέσα τους. Κάθε φορά που η λογική τιμή ενός ηλεκτρικού κόμβου ανεβαίνει από το 0 στο 1 και μετά ξαναπέφτει στο 0, χάνεται (μετατρέπεται σε θερμότητα) ποσότητα ενέργειας ίση πρός $C \cdot V^2$, όπου C η παρασιτική χωρητικότητα του κόμβου και V η τάση τροφοδοσίας. Αυτός είναι ο κύριος λόγος γιά τον οποίο οι νεότερες γενιές chips έχουν χαμηλότερη τάση τροφοδοσίας (π.χ. 2.5 V, 1.8 V, 1.2 V) από τις παλαιότερες. Γιά δοσμένη τάση τροφοδοσίας, όσο περισσότεροι, και μεγαλύτερης χωρητικότητας, ηλεκτρικοί κόμβοι ανεβοκατεβαίνουν (αναβοσβήνουν) μέσα σε ένα chip, και όσο περισσότερες φορές ανά δευτερόλεπτο το κάνουν αυτό, τόσο μεγαλύτερη ισχύ καταναλώνει το chip αυτό. Άρα, γιά μικρή κατανάλωση, πρέπει ένα κύκλωμα να είναι μικρό (λίγες πύλες, μικρή χωρητικότητα), ή και να δουλεύει αργά (λιγότερα ανεβοκατεβάσματα ανά δευτερόλεπτο) και να δουλεύει με όσο γίνεται χαμηλότερη τάση τροφοδοσίας, ή και να δουλεύει γιά λίγο και μετά να κάθεται (να μην αλλάζει κατάσταση), ή και τις περισσότερες φορές να εργάζεται ένα μικρό μόνο μέρος του κυκλώματος και το υπόλοιπο να κάθεται.