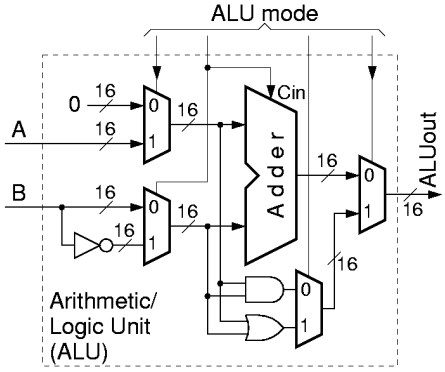
ΗΥ-120: Ψηφιακή Σχεδίαση
Φθινόπωρο 2003

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Σημειώσεις 11:
Ένας απλός Υπολογιστής
και περί Ταχύτητας και Κατανάλωσης Ενέργειας των Κυκλωμάτων CMOS

Στο τελευταίο αυτό μέρος του μαθήματος θα δούμε πώς μπορούμε να φτιάξουμε έναν απλό υπολογιστή χρησιμοποιώντας μόνο λίγους βασικούς δομικούς λίθους αυτού του μαθήματος: καταχωρητές, μνήμες, αθροιστές, πολυπλέκτες, και ένα απλό συνδυαστικό κύκλωμα ή μιά απλή FSM γιά να ελέγχει τα παραπάνω. Παρ' ότι πολύ απλός (απλοϊκός) --και εξαιρετικά αργός!-- θα είναι ένας κανονικός υπολογιστής, ικανός να εκτελεί (σχεδόν) το κάθε πρόγραμμα· τα μόνα που θα του λείπουν θα είναι σχετικά δευτερεύουσες λεπτομέρειες, και όχι κάτι εννοιολογικά κεντρικό. Ο υπολογιστής μας θα είναι δεκαεξάμπιτος (16-bit), δηλαδή θα δουλεύει με λέξεις των 16 bits· δεν θα ασχοληθούμε καθόλου με bytes (8 bits).

11.1 Μία απλή Αριθμητική/Λογική Μονάδα (ALU):



Στην καρδιά ενός υπολογιστή είναι μιά μονάδα που εκτελεί αριθμητικές και λογικές πράξεις. Από αριθμητικές πράξεις, θα έχουμε μόνο πρόσθεση και αφαίρεση ακεραίων: γιά λόγους απλότητας δεν θα έχουμε πολλαπλασιασμό (μοιάζει με πολλές προσθέσεις), διαίρεση (μοιάζει με επανειλημμένες αφαιρέσεις), ή αριθμούς κινητής υποδιαστολής (οι σχετικές πράξεις ανάγονται τελικά σε πράξεις ακεραίων). Θα χρησιμοποιήσουμε σαν "Αριθμητική/Λογική Μονάδα" (Arithmetic/Logic Unit - **ALU**) το κύκλωμα δεξιά, που χρησιμοποιεί έναν αθροιστή (adder). Όπως είδαμε στην §6.7, η αφαίρεση A-B γίνεται μέσω της πρόσθεσης A+(B')+1, όπου A και B είναι προσημασμένοι ακέραιοι αριθμοί σε κωδικοποίηση συμπληρώματος ως προς 2, και (B') είναι ο αριθμός που προκύπτει από τον B αν αντιστρέψουμε το κάθε bit του (συμπλήρωμα ως προς 1, δηλ. λογικό-ΟΧΙ). Την ιδιότητα αυτή εκμεταλλευόμαστε, με τον αριστερό-κάτω πολυπλέκτη, γιά να κάνει η ALU μας και αφαίρεση A-B· το "+1" προκύπτει φροντίζοντας να είναι 1 το κρατούμενο εισόδου, Cin, του αθροιστή όποτε ο πολυπλέκτης επιλέγει το (B'), δηλαδή όποτε κάνουμε αφαίρεση.

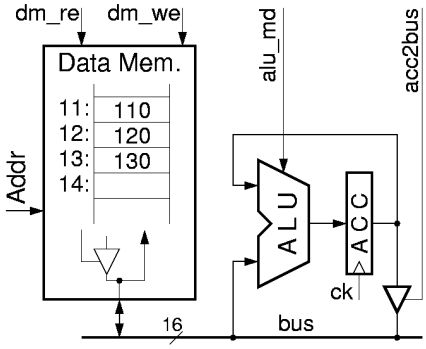
Εκτός από τον δεκαεξάμπιτο αθροιστή, η ALU θα έχει και 16 πύλες ΚΑΙ και 16 πύλες Ή γιά να μπορεί να κάνει τις αντίστοιχες λογικές πράξεις πάνω στις λέξεις εισόδου· πρόκειται γιά λογικές πράξεις bit-προς-bit (bitwise operations), δηλαδή το bit *i* της εξόδου θα είναι το λογικό ΚΑΙ ή το λογικό Ή του bit *i* της εισόδου A και του bit *i* της εισόδου B. Κάθε τέτοια δεκαεξάδα πυλών φαίνεται σχεδιασμένη σαν μία πύλη κάτω από τον αθροιστή. Ο πολυπλέκτης δεξιά επιλέγει αν επιθυμούμε η έξοδος της ALU να είναι το αποτέλεσμα της αριθμητικής πράξης που κάνει ο αθροιστής, ή το αποτέλεσμα της λογικής πράξης που κάνουν οι πύλες· ο πολυπλέκτης κάτω επιλέγει αν το αποτέλεσμα της λογικής πράξης θα είναι το λογικό ΚΑΙ ή το λογικό Ή. Τέλος, ο αριστερός επάνω πολυπλέκτης επιλέγει αν η πράξη θα γίνει με την πρώτη είσοδο της ALU, A, ή με τον αριθμό μηδέν· εάν επιλέξουμε τον αριθμό 0, ο αθροιστής θα δώσει έξοδο 0+B = B ή 0+(B')+1 = -B, οι πύλες ΚΑΙ θα δώσουν έξοδο 0, και οι πύλες Ή θα δώσουν έξοδο B ή B' (ΟΧΙ B). Έτσι, συνολικά, διαπιστώνουμε ότι η ALU μπορεί να κάνει τις παρακάτω πράξεις, ανάλογα με την εκάστοτε τιμή των τεσσάρων σημάτων ελέγχου της, *mode*. Η ένδειξη "x" στο mode σημαίνει ότι η ALU κάνει την ίδια πράξη όποια τιμή και να έχει το αντίστοιχο bit του mode (συνθήκη αδιαφορίας - §4.3). Οι πράξεις που μας ενδιαφέρουν να χρησιμοποιήσουμε στον υπολογιστή μας σημειώνονται με παχειά γράμματα.

mode:	*Πράξη:*	*(όνομα)*			
			0x01	ALUout = 0	
			0011	ALUout = B	
00x0	**ALUout = B**	**(passB)**	**0111**	**ALUout = *NOT* B**	**(not)**
01x0	ALUout = -B		**1001**	**ALUout = A *AND* B**	**(and)**
10x0	**ALUout = A+B**	**(add)**	**1011**	**ALUout = A *OR* B**	**(or)**
11x0	**ALUout = A-B**	**(sub)**	1101	ALUout = A *AND* (*NOT* B)	
			1111	ALUout = A *OR* (*NOT* B)	

11.2 Αριθμητικές/Λογικές Πράξεις σ' έναν Επεξεργαστή Τύπου Συσσωρευτή (Accumulator):

Τώρα που έχουμε την ALU όπου θα εκτελούνται οι πράξεις, το επόμενο θέμα είναι πάνω σε ποιούς αριθμούς θα γίνονται αυτές οι πράξεις, πού θα βρίσκονται αυτοί οι αριθμοί, και πώς θα τους επιλέγουμε. Δεδομένου ότι οι υπολογιστές προορίζονται γιά την επεξεργασία μεγάλου όγκου δεδομένων, προφανώς θα χρειαστούμε μία (μεγάλη) μνήμη (§9.3) όπου θα κρατιούνται αυτά τα δεδομένα. Αυτή τη λέμε "**Μνήμη Δεδομένων**" (Data Memory), και φαίνεται στο παρακάτω σχήμα. Γιά να γίνει μία πράξη (π.χ. πρόσθεση), χρειαζόμαστε δύο αριθμούς πάνω στους οποίους θα γίνει η πράξη, και χρειάζεται να τοποθετηθεί κάπου και το αποτέλεσμα της πράξης. Υπάρχουν υπολογιστές που γιά να γίνει αυτό διαβάζουν δύο αριθμούς από τη μνήμη δεδομένων, και γράφουν το αποτέλεσμα επίσης σε κάποια θέση της μνήμης δεδομένων. Γιά μας όμως, κάτι τέτοιο θα ήταν πολύπλοκο, διότι θα απαιτούσε τρείς χωριστές προσπελάσεις στη μνήμη --δύο αναγνώσεις και μία εγγραφή. Εμείς θα ακολουθήσουμε μιάν άλλη "αρχιτεκτονική" υπολογιστή --την απλούστερη που έχει υπάρξει ιστορικά: την *αρχιτεκτονική συσσωρευτή* (accumulator architecture). Στην αρχιτεκτονική αυτή, υπάρχει ένας ειδικός καταχωρητής, έξω από τη μνήμη δεδομένων, ο οποίος κρατάει έναν αριθμό --το αποτέλεσμα της πιό πρόσφατης πράξης. Κάθε καινούργια πράξη γίνεται ανάμεσα σε αυτό το πιό πρόσφατο αποτέλεσμα και σ' ένα καινούργιο αριθμό που διαβάζουμε από τη μνήμη, και αφήνει το αποτέλεσμά της πάλι σε αυτόν τον ειδικό καταχωρητή. Έτσι, π.χ., αν κάνουμε συνεχείς προσθέσεις, συσσωρεύεται σε αυτόν

τον καταχωρητή το άθροισμα όλων των αριθμών που διαβάσαμε από τη μνήμη και προσθέσαμε, και γι' αυτό ο καταχωρητής αυτός ονομάστηκε " **συσσωρευτής** " (accumulator). Οι σημερινοί υπολογιστές έχουν αρκετούς τέτοιους καταχωρητές --συνήθως 32· τους ονομάζουμε "καταχωρητές γενικού σκοπού" (general-purpose registers), και όχι συσσωρευτές.



Στο σχήμα φαίνεται η απαιτούμενη συνδεσμολογιά γιά να γίνονται οι πράξεις όπως τις περιγράψαμε. Ο συσσωρευτής είναι ένας ακμοπυροδότητος καταχωρητής (§8.4), που σημειώνεται σαν "ACC". Ένα εξωτερικό κύκλωμα, που θα δούμε σε λίγο, τροφοδοτεί τη διεύθυνση Addr στη μνήμη δεδομένων, καθώς και τα σήματα ελέγχου ανάγνωσης (*dm_re* - read enable) και εγγραφής (*dm_we* - write enable), προκαλώντας την ανάγνωση μιάς λέξης (δηλ. ενός αριθμού) από τη θέση μνήμης Addr. Η ALU παίρνει το περιεχόμενο του συσσωρευτή ACC στη μιά της είσοδο, και τον αριθμό που διαβάσαμε από τη μνήμη στην άλλη· το εξωτερικό κύκλωμα ελέγχου προσδιορίζει, μέσω του *alu_md*, το είδος της πράξης που πρέπει να γίνει, και το αποτέλεσμα της πράξης δίδεται σαν είσοδος στο συσσωρευτή. Όταν έλθει η ενεργή ακμή του ρολογιού, το αποτέλεσμα αυτής της πράξης αντικατθιστά το παλαιό περιεχόμενο του συσσωρευτή. Κατά καιρούς, πρέπει το αποτέλεσμα των πράξεων να αποθηκεύεται (γράφεται) σε μιά επιθυμητή θέση (λέξη) μνήμης, προκειμένου μετά να ξεκινήσει κάποια νέα σειρά πράξεων στο συσσωρευτή. Γιά να γίνεται η αποθήκευση αυτή προβλέφτηκε ένας τρικατάστατος οδηγητής, δεξιά κάτω, ο οποίος τοποθετεί το περιεχόμενο του ACC πάνω στη λεωφόρο (bus), απ' όπου και το παραλαμβάνει η μνήμη γιά να γίνει η εγγραφή. Τα σήματα ελέγχου που πρέπει να ενεργοποιηθούν είναι τα *acc2bus* (ACC προς bus - ACC to bus, όπου το "2" είναι ομόηχο με το "to") και *dm_we*.

11.3 Πρόγραμμα και Εντολές: οι Οδηγίες γιά τις Πράξεις

Γιά να λειτουργήσει το παραπάνω κύκλωμα και να γίνουν οι επιθυμητές πράξεις, πρέπει κάποιος να τροφοδοτεί τις διευθύνσεις, *Addr*, και τα σήματα ελέγχου των πράξεων, *dm_re, alu_md, acc2bus,* και *dm_we*. Τη δουλειά αυτή αναλαμβάνει ένα άλλο κύκλωμα, που θα δούμε παρακάτω, το οποίο ακολουθεί πιστά τις σχετικές οδηγίες που έχει γράψει ένας άνθρωπος (με τη βοήθεια κάποιου υπολογιστή) και οι οποίες είναι αποθηκευμένες σε μιά μνήμη. Κάθε οδηγία γιά μιά συγκεκριμένη πράξη ή ενέργεια λέγεται **εντολή** (instruction), και το σύνολο των εντολών που έχουν δοθεί σ' έναν υπολογιστή (έχουν γραφτεί στη μνήμη του) και τις οποίες αυτός ακολουθεί σε δεδομένη στιγμή λέμε ότι αποτελούν το **πρόγραμμα** (program) που αυτός "εκτελεί" (executes) ή "τρέχει" (runs) τη δεδομένη στιγμή. Τα κυκλώματα αποτελούν το **υλικό** (hardware) του υπολογιστή, και τα προγράμματα που τρέχουν ή μπορούν να τρέξουν σε αυτόν αποτελούν το **λογισμικό** του (software).

Ο κάθε υπολογιστής "καταλαβαίνει", δηλαδή μπορεί να αναγνωρίσει και να εκτελέσει, ορισμένες μόνο, συγκεκριμένες εντολές ή τύπους εντολών· αυτές τις ονομάζουμε σύνολο ή **ρεπερτόριο εντολών** (instruction set) του υπολογιστή. Οι εντολές του δικού μας υπολογιστή θα αποτελούνται από δύο κομάτια καθεμία: έναν "κώδικα πράξης" (operation code, ή **opcode** εν συντομία), και μιά **διεύθυνση** Addr. Κάθε εντολή μας θα είναι 16 bits, από τα οποία τα 4 MS bits θα είναι ο opcode και τα 12 LS bits θα είναι η διεύθυνση. Γιά να μπορεί ο υπολογιστής μας να εκτελεί τις αριθμητικές και λογικές πράξεις που περιγράψαμε παραπάνω, το ρεπερτόριό του πρέπει να περιλαμβάνει τις εξής εντολές:

```
load  Addr    ACC <- DM[Addr]              not  Addr   ACC <- NOT DM[Addr]
add   Addr    ACC <- ACC + DM[Addr]        and  Addr   ACC <- ACC AND DM[Addr]
sub   Addr    ACC <- ACC - DM[Addr]        or   Addr   ACC <- ACC OR DM[Addr]
store Addr    DM[Addr] <- ACC
```

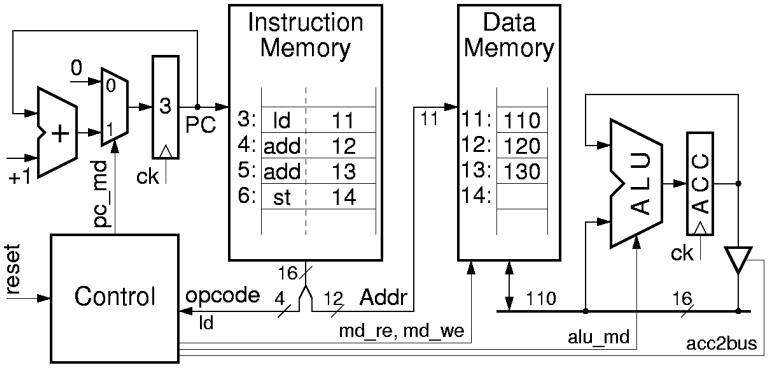
Οι δύο λέξεις αριστερά, με τα πλάγια γράμματα, δείχνουν τη συμβολική γραφή της εντολής: η πρώτη λέξη είναι το σύμβολο του opcode, ενώ το *Addr* αντικαθίσταται κάθε φορά από τη συγκεκριμένη διεύθυνση που επιθυμούμε να χρησιμοποιήσουμε --έναν αριθμό από 0 μέχρι 4095, αφού οι διευθύνσεις είναι δωδεκάμπιτες στον υπολογιστή μας. Ένα πρόγραμμα με τις εντολές του γραμμένες με αυτό το συμβολισμό λέμε ότι είναι γραμμένο σε "**Γλώσσα Assembly**". Στη μνήμη του υπολογιστή, φυσικά, το μόνο που υπάρχει είναι άσσοι και μηδενικά, άρα γιά να εκτελεστεί ένα πρόγραμμα Assembly πρέπει πρώτα να μετατραπεί στην δυαδική του αναπαράσταση, που λέγεται " **Γλώσσα Μηχανής** (machine language, ή object code, ή binary code). Η μετατροπή αυτή είναι πολύ απλή: κάθε συμβολικός opcode αντικαθίσταται με τον αντίστοιχο δυαδικό του κώδικα (βλ. παρακάτω), και κάθε διεύθυνση μετατρέπεται από το δεκαδικό στο δυαδικό. Τη μετατροπή αυτή (και μερικές άλλες σχετικές βοηθητικές εργασίες) την κάνει συνήθως ένα μικρό πρόγραμμα υπολογιστή που ονομάζεται **Assembler**.

Δίπλα στο συμβολισμό Assembly της κάθε εντολής, στο παραπάνω πινακάκι, υπάρχει μία "εξίσωση μεταφοράς καταχωρητών" (register transfer equation), η οποία περιγράφει τις ενέργειες που πρέπει να γίνουν προκειμένου ο υπολογιστής να εκτελέσει την εντολή. Σε αυτές τις εξισώσεις, το αριστερό βέλος υποδεικνύει μεταφορά και εγγραφή πληροφορίας (εκχώρηση - assignment). Ο συμβολισμός "DM[Addr]" σημαίνει τη θέση (λέξη) της μνήμης δεδομένων στη διεύθυνση Addr. Όταν ο συσσωρευτής, ACC, εμφανίζεται και δεξιά και αριστερά από το βέλος, τότε δεξιά μεν σημαίνει το παλαιό περιεχόμενό του (πριν την ακμή ρολογιού), αριστερά δε σημαίνει τη νέα τιμή του (μετά την ακμή ρολογιού).

Έτσι, η εντολή `load Addr` (φόρτωσε) διαβάζει τον αριθμό που περιέχεται στη διεύθυνση *Addr* της μνήμης, δηλαδή διαβάζει το `DM[Addr]`, και το αντιγράφει στο συσσωρευτή. Η εντολή `add Addr` προσθέτει το παλαιό περιεχόμενο του ACC με τον αριθμό που περιέχεται στη διεύθυνση *Addr* της μνήμης, και γράφει το αποτέλεσμα στον ACC. Αντίστοιχα, οι εντολές `sub, not, and, or` κάνουν άλλες παρόμοιες πράξεις. Τέλος, η εντολή `store Addr` (αποθήκευσε) αντιγράφει το περιεχόμενο του ACC στη θέση (λέξη) μνήμης με διεύθυνση *Addr*. Γιά παράδειγμα, λοιπόν, αν η μνήμη δεδομένων περιέχει τους αριθμούς που φαίνονται στο παραπάνω σχήμα, τότε το πρόγραμμα: "`load 11; add 12; add 13; store 14;`" θα προκαλέσει τις εξής

ενέργειες. Πρώτα θα διαβαστεί ο αριθμός 110 από τη θέση 11 και θα γραφτεί στο συσσωρευτή· μετά, θα διαβαστεί ο αριθμός 120 και θα προστεθεί στον 110, και το αποτέλεσμα 230 θα γραφτεί στο συσσωρευτή· εν συνεχεία, θα διαβαστεί το 130 από τη θέση 13, θα προστεθεί στο 230, και το αποτέλεσμα 360 θα μείνει στο συσσωρευτή· και τέλος, το περιεχόμενο 360 του συσσωρευτή θα γραφτεί στη θέση μνήμης 14.

11.4 Ανάγνωση & Εκτέλεση Εντολών:



Γιά να μπορέσει ο υπολογιστής να εκτελέσει τις εντολές του προγράμματος, πρέπει αυτές να είναι κάπου γραμμένες, και από εκεί να τις διαβάζει μία-μία και να τις εκτελεί. Αυτός είναι ο ρόλος των κυκλωμάτων που θα προσθέσουμε εδώ στο κύκλωμα της §11.2, και τα οποία φαίνονται στο διπλανό σχήμα - αριστερό ήμισυ. Το πρόγραμμα είναι αποθηκευμένο στη "**Μνήμη Εντολών**" (Instruction Memory). Κανονικά, οι υπολογιστές χρησιμοποιούν την ίδια μνήμη γιά να αποθηκεύουν τόσο τα δεδομένα όσο και το πρόγραμμα (σε χωριστές διευθύνσεις το καθένα)· με τον τρόπο αυτό, αν έχουμε ένα μικρό πρόγραμμα υπάρχει χώρος γιά πολλά δεδομένα, και αντιστρόφως, αν έχουμε λίγα δεδομένα μπορεί το πρόγραμμα να είναι μεγάλο. Εμείς εδώ χρησιμοποιούμε δύο χωριστές μνήμες, μιά γιά δεδομένα και μιά γιά εντολές, γιά να απλοποιηθεί το κύκλωμα και η λειτουργία του.

Γιά να μπορέσει το κύκλωμά μας να εκτελέσει μιάν εντολή, πρέπει να την διαβάσει από τη μνήμη εντολών, και γιά το σκοπό αυτό χρειάζεται τη διεύθυνση της μνήμης αυτής όπου βρίσκεται η επιθυμητή εντολή. Αυτός είναι ο ρόλος του καταχωρητή PC: να θυμάται τη διεύθυνση αυτή. Στο παράδειγμα του σχήματος, ο PC περιέχει τον αριθμό 3, ο οποίος δίδεται σα διεύθυνση στη μνήμη εντολών· η μνήμη εντολών διαβάζει και μας δίνει το περιεχόμενο της θέσης 3, το οποίο εδώ τυχαίνει να είναι η εντολή `load 11` --κωδικοποιημένη σε γλώσσα μηχανής φυσικά, δηλαδή "0000000000001011" όπως θα δούμε σε λίγο. Κάθε λέξη της μνήμης εντολών είναι 16 bits, και περιέχει μιάν εντολή. Τα 16 σύρματα ανάγνωσης που βγαίνουν από τη μνήμη αυτή, τα χωρίζουμε σε 4 αριστερά (MS) σύρματα που πηγαίνουν στο κύκλωμα ελέγχου, και 12 δεξιά (LS) σύρματα που πηγαίνουν σαν διεύθυνση στη μνήμη δεδομένων. Αφού όλες οι εντολές του υπολογιστή μας αποτελούνται από έναν opcode στα 4 MS bits και μία διεύθυνση στα 12 LS bits, με τη συνδεσμολογία αυτή πηγαίνει ο opcode στο κύκλωμα ελέγχου και η διεύθυνση στη μνήμη δεδομένων. Στο παράδειγμά μας, ο opcode είναι 0000 (που σημαίνει `load`), και η διεύθυνση είναι 000000001011 (δηλ. 11 δεκαδικό). Το κύκλωμα ελέγχου, βλέποντας την εντολή `load`, θα ζητήσει ανάγνωση από τη μνήμη δεδομένων (*dm_re=1, dm_we=0, acc2bus=0*) και θα θέσει την ALU σε λειτουργία *passB* (*alu_md=0000* ή *0010*). Η μνήμη δεδομένων, βλέποντας τη διεύθυνση 11 και ότι της ζητείται ανάγνωση, θα τοποθετήσει τον αριθμό 110 στη λεωφόρο· η ALU, εκτελώντας λειτουργία *passB*, θα περάσει τον αριθμό 110 στην έξοδό της· στην ενεργή ακμή του ρολογιού, ο αριθμός αυτός θα γραφτεί στο συσσωρευτή ACC, ολοκληρώνοντας έτσι την εκτέλεση της εντολής `load 11`.

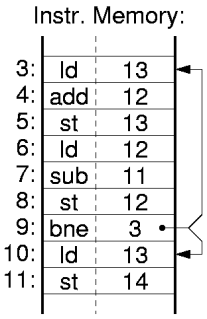
Μετά την εκτέλεση της εντολής `load 11` από τη θέση 3 της μνήμης εντολών, πρέπει να εκτελεστεί η επόμενη εντολή. Κατά πάγια σύμβαση, εκτός ειδικών εξαιρέσεων που θα δούμε σε λίγο, η επόμενη εντολή βρίσκεται γραμμένη στην ακριβώς επόμενη θέση μνήμης --εδώ, στη διεύθυνση 4. Γιά να προκύψει η επόμενη αυτή διεύθυνση γιά τη μνήμη εντολών, χρησιμοποιούμε τον αυξητή (incrementor) που φαίνεται αριστερά στο σχήμα, δηλαδή έναν αθροιστή με δεύτερη είσοδο το +1. Έτσι, στην ίδια παραπάνω ενεργή ακμή του ρολογιού που γράφεται ο αριθμός 110 στο συσσωρευτή ACC, γράφεται και το αποτέλεσμα της πρόσθεσης 3+1=4 στον καταχωρητή PC. Το αποτέλεσμα είναι ότι στον επόμενο κύκλο ρολογιού ο PC θα περιέχει 4· η μνήμη εντολών θα διαβάσει και θα μας δώσει το περιεχόμενο "0010000000001100" της θέσης 4, δηλαδή την εντολή `add 12`· το κύκλωμα ελέγχου, βλέποντας opcode=0010 (`add`), θα δώσει *dm_re=1, dm_we=0, acc2bus=0,* και *alu_md=10x0* (δηλ. add)· η μνήμη δεδομένων, βλέποντας διεύθυνση 12 και *dm_re=1*, θα διαβάσει και θα δώσει στη λεωφόρο τον αριθμό 120· η ALU, βλέποντας ACC=110, στη λεωφόρο το 120, και *alu_md=add*, θα προσθέσει 110+120 και θα δώσει στην έξοδό της 230· και ο αθροιστής/αυξητής αριστερά, βλέποντας PC=4, θα δώσει στην έξοδό του 4+1=5. Μόλις έλθει η επόμενη ενεργή ακμή ρολογιού, το 230 θα μπεί στον ACC, και το 5 θα μπεί στον PC, ολοκληρώνοντας έτσι την εκτέλεση της εντολής `add 12`, και προετοιμάζοντάς μας γιά την επόμενη εντολή, `add 13`, από τη θέση 5.

Ο καταχωρητής PC ονομάζεται "**Μετρητής Προγράμματος**" (Program Counter - PC), επειδή είναι κατά βάση ένας μετρητής που αυξάνεται κατά 1 στο τέλος της εκτέλεσης κάθε εντολής γιά να μας δώσει τη διεύθυνση της επόμενης εντολής· ο πολυπλέκτης που υπάρχει στην είσοδό του προορίζεται γιά την αρχικοποίησή του, όταν δίδεται σήμα reset. Το κύκλωμα **ελέγχου** (control) είναι υπεύθυνο γιά τη δημιουργία όλων των σημάτων ελέγχου που λένε σε κάθε μονάδα τι να κάνει κάθε φορά. Όλες οι εντολές που είδαμε μέχρι στιγμής διαβάζονται και εκτελούνται σε έναν κύκλο ρολογιού η καθεμία, και γι' αυτό το κύκλωμα ελέγχου, μέχρι στιγμής, είναι ένα απλό συνδυαστικό κύκλωμα. Ο πίνακας αληθείας του προκύπτει αν σκεφτούμε τι εργασίες πρέπει να γίνουν γιά την εκτέλεση κάθε εντολής:

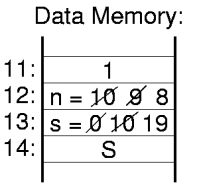
reset:	*opcode:*	*md_re:*	*md_we:*	*alu_md:*	*acc2bus:*	*pc_md:*
1	xxxx	0	0	0x01 (zero)	0	0
0	0000 (load)	1	0	00x0 (passB)	0	1
0	0001 (store)	0	1	00x0 (passB)	1	1
0	0010 (add)	1	0	10x0 (add)	0	1
0	0011 (sub)	1	0	11x0 (sub)	0	1
0	0100 (and)	1	0	1001 (and)	0	1
0	0101 (or)	1	0	1011 (or)	0	1
0	0111 (not)	1	0	0111 (not)	0	1

Το σήμα reset επαναφέρει τον υπολογιστή στην αρχική κατάσταση εκκίνησης, ό,τι κι αν έκανε αυτός πριν (opcode=xxxx): αρχικοποιεί τον PC στο 0, γιά να αρχίσει να εκτελεί εντολές από την αρχή της μνήμης εντολών. Οι εντολές `load` και `add` εκτελούνται όπως περιγράψαμε παραπάνω. Οι εντολές `sub`, `and`, `or`, και `not` εκτελούνται κατά εντελώς ανάλογο τρόπο --απλώς αλλάζει το mode της ALU. Η εντολή `store` διαφέρει λίγο: ανάβοντας το *acc2bus=1* (με *md_re=0*, φυσικά), τοποθετεί την τιμή του συσσωρευτή στο bus· ενεργοποιώντας το *md_we=1*, η τιμή αυτή από το bus εγγράφεται στη μνήμη δεδομένων· επίσης, θέτοντας την ALU σε mode *passB*, η τιμή από το bus επανεγγράφεται στον ACC, άρα διατηρείται εκεί αμετάβλητη. Υπάρχει μιά λεπτομέρεια που δεν είναι σωστή σε αυτό το συνδυαστικό τρόπο γέννησης του σήματος *md_we*: δεν υπάρχει εγγύηση ότι το σήμα αυτό θα ανάψει *μετά* τη σταθεροποίηση της διεύθυνσης της μνήμης δεδομένων, όπως πρέπει να γίνει· το πρόβλημα αυτό δεν μπορεί να διορθωθεί παρά μόνο αν αλλάξει το κύκλωμα ελέγχου και γίνει ακολουθιακό (FSM).

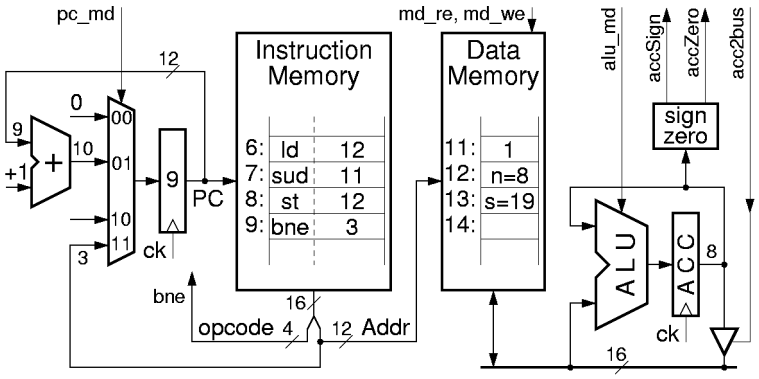
11.5 Επανεκτέλεση Εντολών: Διακλαδώσεις υπό Συνθήκη, Άλματα



Γιά όλες τις εντολές που είδαμε μέχρι στιγμής, η επόμενή τους προς εκτέλεση εντολή ήταν αυτή που είναι γραμμένη στην αμέσως "από κάτω" θέση μνήμης. 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). Μιά 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 10+9+8+...+3+2+1 και το γράφει στη θέση 14 της μνήμης δεδομένων. Ο υπολογισμός γίνεται χρησιμοποιώντας κυρίως τις θέσεις 12 και 13 της μνήμης δεδομένων (κάτω μέρος σχήματος), οι οποίες έχουν αρχική τιμή 10 (η μεταβλητή "n") και 0 (η μεταβλητή "s") αντίστοιχα. Οι τρείς πρώτες εντολές του προγράμματος (θέσεις 3, 4, και 5) διαβάζουν την τρέχουσα τιμή της μεταβλητής s, της προσθέτουν την τρέχουσα τιμή της μεταβλητής n, και γράφουν το αποτέλεσμα πίσω στην s. Την πρώτη φορά που εκτελούνται αυτές οι εντολές, αυξάνουν το s από 0 σε 10. Οι τρείς επόμενες εντολές (θέσεις 6, 7, και 8) ελάττώνουν τη μεταβλητή n κατά 1· την πρώτη φορά που εκτελούνται, αλλάζουν το n από 10 σε 9. Στη συνέχεια εκτελείται η εντολή `bne 3` από τη θέση 9· η εντολή αυτή σημαίνει *εάν ο συσσωρευτής* ***δεν*** *ισούται με μηδέν, διακλαδώσου (πήγαινε) στην εντολή 3* (branch if ACC not equal to zero - brach not equal - bne). Επειδή εκείνη την ώρα ο συσσωρευτής περιέχει το n=9, που είναι διάφορο του μηδενός, η συνθήκη της διακλάδωσης είναι αληθής και η διακλάδωση επιτυγχάνει (πραγματοποιείται). Έτσι, επόμενες εντολές εκτελούνται οι εντολές 3, 4, και 5, αυξάνοντας το s από 10 σε 19, και μετά οι 6, 7, και 8, μειώνοντας το n από 9 σε 8. Μετά, ξαναεκτελείται η `bne 3`· επειδή ο συσσωρευτής περιέχει το 8, η διακλάδωση επιτυγχάνει και πάλι. Έτσι, οι εντολές 3 έως και 9 θα ξαναεκτελεστούν κάμποσες φορές ακόμα, αυξάνοντας διαδοχικά το s κατά 8, 7, ..., 2, και 1, και μειώνοντας το n διαδοχικά σε 7, 6, ..., 1, και 0. Την τελευταία φορά, στο συσσωρευτή θα έχει μείνει n=0. Τότε, η διακλάδωση `bne 3` θα αποτύχει, διότι ο συσσωρευτής δεν είναι πλέον διάφορος του μηδενός· έτσι, η επόμενη εντολή δεν θα διαβαστεί από τη θέση 3 όπως πρίν, αλλά από τη θέση 10, δηλαδή από την "από κάτω" θέση, όπως κάνουν και όλες οι άλλες εντολές που δεν είναι διακλαδώσεις. Τώρα, οι εντολές 10 και 11 θα αντιγράψουν το τελικό αποτέλεσμα 10+9+8+...+3+2+1 = 55 από τη θέση 13 (s) στη θέση 14 (S) και ο στόχος του προγράμματος θα έχει επιτευχθεί.



Γιά να μπορέσει ο υπολογιστής μας να εκτελεί εντολές διακλάδωσης (υπό συνθήκη) και άλματος (χωρίς συνθήκη), απαιτείται μιά προσθήκη στο κύκλωμα του PC, η οποία φαίνεται στο σχήμα δίπλα, και μιά αλλαγή στο κύκλωμα ελέγχου. Στον PC, μεγαλώσαμε τον πολυπλέκτη εισόδου του απο 2-σε-1 σε 4-σε-1, ούτως ώστε να προσφέρουμε περισσότερες επιλογές γιά τον τρόπο προσδιορισμού της επόμενης τιμής του PC, δηλαδή του ποιά θα είναι (από ποιά διεύθυνση) η επόμενη εντολή. Η νέα επιλογή που προσθέτουμε είναι το πεδίο *Addr* της παρούσας εντολής· έτσι, όταν εκτελούμε μιάν εντολή διακλάδωσης ή άλματος, όπως `bne 3`, το πεδίο διεύθυνσής της (εδώ το "3") μπορεί να αποτελέσει την επόμενη τιμή του PC όταν η εντολή είναι άλμα ή επιτυχημένη διακλάδωση. Η είσοδος "10" του πολυπλέκτη προορίζεται γιά τις προσθήκες της επομένης παραγράφου. Όσον αφορά το κύκλωμα ελέγχου, αυτό χρειάζεται περισσότερες εισόδους τώρα, γιά να ξέρει να ελέγξει σωστά τη λειτουργία του υπολογιστή: δεν αρκεί πλέον να ξέρει μόνο τον opcode της παρούσας εντολής (και το αν κάνουμε reset ή όχι), αλλά πρέπει να ξέρει και το αν ο συσσωρευτής είναι μηδέν ή όχι, δεδομένου ότι από αυτό εξαρτάται η έκβαση της εντολής `bne`. Αυτός είναι ο ρόλος της προσθήκης "sign, zero" (πρόσημο, μηδέν) που κοιτάζει το περιεχόμενο του συσσωρευτή, ACC.

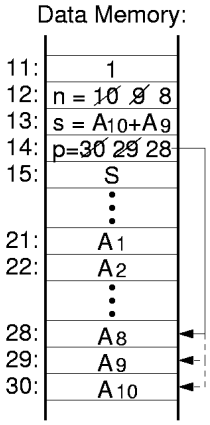
Ο υπολογιστής μας θα έχει τέσσερεις εντολές διακλάδωσης υπό συνθήκη: (α) `bne` (branch not equal): διακλαδώσου εάν ACC διάφορος του μηδενός· (β) `beq` (branch equal): διακλαδώσου εάν ACC ίσος με το μηδέν· (γ) `bge` (branch greater or equal): διακλαδώσου εάν ACC μεγαλύτερος ή ίσος του μηδενός· και (δ) `blt` (branch less than): διακλαδώσου εάν ACC μικρότερος

του μηδενός· επίσης θα έχει μιάν 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. 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 μίαν 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 16 εισόδων, και παρέχουμε στο κύκλωμα ελέγχου αυτά τα δύο σήματα σαν εισόδους του. Τώρα, ο πίνακας αληθείας του κυκλώματος ελέγχου πρέπει να μετατραπεί ως εξής, σε σχέση με αυτόν που είδαμε στο τέλος της §11.4:

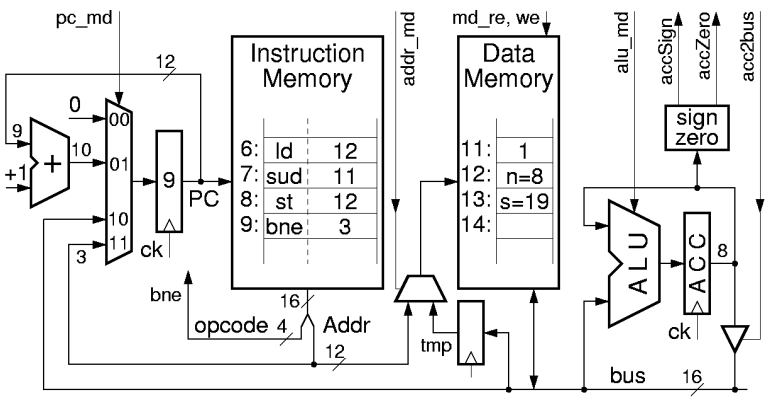
reset:	*opcode:*	*accZero:*	*accSign:*	*md_re,we:*	*alu_md:*	*acc2bus:*	*pc_md:*
1	xxxx	x	x	0 0	zero	0	00
0	0000 (load)	x	x	1 0	passB	0	01
0	0001 (store)	x	x	0 1	passB	1	01
0	... (add,...)	x	x	1 0	add,...	0	01
0	1000 (bne)	0	x	0 0	passB	1	11
0	1000 (bne)	1	x	0 0	passB	1	01
0	1001 (beq)	0	x	0 0	passB	1	01
0	1001 (beq)	1	x	0 0	passB	1	11
0	1010 (bge)	x	0	0 0	passB	1	11
0	1010 (bge)	x	1	0 0	passB	1	01
0	1011 (blt)	x	0	0 0	passB	1	01
0	1011 (blt)	x	1	0 0	passB	1	11
0	1100 (jump)	x	x	0 0	passB	1	11

Γιά τις εντολές που προϋπήρχαν, τα σήματα ελέγχου δεν εξαρτώνται από τις νέες εισόδους, *accZero* και *accSign* , όπως υποδεικνύουν τα "x" στις αντίστοιχες γραμμές και στήλες· το ίδιο ισχύει και γιά την εντολή `jump` . Γιά τις εντολές διακλάδωσης, όμως, το σήμα *pc_md* που ελέγχει το ποιά θα είναι η επόμενη εντολή καθορίζεται από τον τύπο της διακλάδωσης και από τις εισόδους aluZero και aluSign. Κατά την εκτέλεση των εντολών διακλάδωσης και άλματος, οδηγούμε το συσσωρευτή ACC στη λεωφόρο bus (acc2bus=1, md_re=0), και θέτουμε την ALU σε λειτουργία passB προκειμένου η τιμή που υπήρχε στο συσσωρευτή να ανακυκλώνεται εκεί, και επομένως να διατηρείται αμετάβλητη.

11.6 Δομές Δεδομένων και Διαδικασίες: Μεταβλητές Διευθύνσεις Μνήμης



Σε όλες τις εντολές που είδαμε μέχρι τώρα, οι διευθύνσεις των δεδομένων καθορίζονταν οριστικά και αμετάκλητα από το πρόγραμμα: ήταν γραμμένες μέσα του. 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 π.χ. να βρούμε το άθροισμα κάμποσων αριθμών που είναι γραμμένοι στη μνήμη, όπως ας πούμε των 10 αριθμών $A_{10}+A_9+A_8+...+A_2+A_1$ στις θέσεις 30, 29, 28, ..., 22, και 21 στο διπλανό σχήμα, με ένα πρόγραμμα παρόμοιο με αυτό της § 11.5 . Γιά να πετύχουμε ένα τέτοιο στόχο, χρειαζόμαστε μιά παραλλαγή της εντολής `add` η οποία να προσθέτει στο 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 το ίδιο το πρόγραμμα την ώρα που τρέχει! Ας ονομάσουμε αυτή την εντολή ***addx*** (add indexed - πρόσθεση με δείκτη), και ας θεωρήσουμε ότι το πρόγραμμά μας τώρα αρχίζει με τις εντολές: `load 13; addx 14; store 13`. Η πρώτη εντολή φέρνει στο συσσωρευτή την προηγούμενη τιμή του *s*· η εντολή `addx 14` πηγαίνει στη θέση 14, διαβάζει το περιεχόμενό της, και στη συνέχεια χρησιμοποιεί αυτό το περιεχόμενο σαν νέα διεύθυνση της μνήμης δεδομένων και πηγαίνει εκεί να διαβάσει έναν αριθμό και να τον προσθέσει στο συσσωρευτή. Την πρώτη φορά που θα εκτελεστεί η εντολή `addx`, βρίσκει στη θέση 14 τον αριθμό 30, και επομένως διαβάζει από τη θέση 30 τον αριθμό A_{10} και τον προσθέτει στο συσσωρευτή. Τη δεύτερη φορά που θα γυρίσουμε να ξαναεκτελέσουμε την εντολή `addx`, φροντίζουμε να έχουμε ήδη μειώσει κατά 1 το περιεχόμενο της θέσης 14, με τον ίδιο τρόπο που μειώσαμε κατά 1 και το *n* στη θέση 12. Έτσι, τώρα, η εντολή `addx` θα βρεί στη θέση 14 τον αριθμό 29 και θα προσθέσει τον A_9 από τη θέση 29 στο συσσωρευτή, κ.ο.κ.



Γιά να μπορεί να εκτελεί εντολές όπως η παραπάνω `addx`, ο υπολογιστής μας χρειάζεται τις προσθήκες που φαίνονται στο επόμενο σχήμα: ένα πολυπλέκτη στην είσοδο διευθύνσεων της μνήμης δεδομένων, προκειμένου η διεύθυνση αυτή να μπορεί να προέρχεται είτε από την εντολή, είτε από ένα δεδομένο που διαβάσαμε από την ίδια τη μνήμη δεδομένων. 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. Το κύκλωμα ελέγχου, τώρα, δεν μπορεί πλέον να είναι συνδυαστικό: επειδή η εντολή `addx` χρειάζεται δύο κύκλους ρολογιού, το κύκλωμα ελέγχου πρέπει να περιέχει και 1 bit κατάστασης που να μας πληροφορεί αν τώρα βρισκόμαστε στον πρώτο ή στο δεύτερο κύκλο της εκτέλεσης· επομένως, το κύκλωμα ελέγχου τώρα θα είναι μιά μικρή FSM δύο καταστάσεων. Επίσης, χρειαζόμαστε μιά μέθοδο ώστε οι καταχωρητές ACC και PC να μπορούν να διατηρούν την τιμή τους αμετάβλητη τον πρώτο από τους δύο αυτούς κύκλους ρολογιού. Η λύση είναι η χρήση καταχωρητών με ένα μικρό εσωτερικό πολυπλέκτη ανακύκλωσης· αυτοί ονομάζονται καταχωρητές με είσοδο ελέγχου *επίτρεψης φόρτωσης* (load enable).

Παρόμοια με την εντολή `addx`, θα χρειαστούμε σίγουρα και μιάν εντολή `storex`, και βολική θα είναι και μία `loadx`. Μιά άλλη δυνατότητα που δεν είχε η προηγούμενη μορφή του υπολογιστή μας, και που προστέθηκε εδώ, είναι η δυνατότητα εντολής άλματος **`jumpx`** `Addr` με μεταβλητή διεύθυνση προορισμού: η εντολή αυτή διαβάζει έναν αριθμό από τη θέση *Addr* της μνήμης δεδομένων και τον γράφει στον PC. Με τον τρόπο αυτό, η επόμενη εντολή που θα εκτελεστεί θα είναι μιά εντολή που μπορεί να την επιλέξει και να τη μεταβάλει το ίδιο το πρόγραμμα την ώρα που τρέχει! Όπως θα δούμε στο μάθημα "Οργάνωση Υπολογιστών" (HY-225), με την εντολή αυτή μπορούμε να υλοποιούμε διαδικασίες (procedures) και άλλες προηγμένες δυνατότητες των οντοκεντρικών γλωσσών προγραμματισμού.

11.7 Ταχύτητα και Κατανάλωση Ενέργειας των Κυκλωμάτων CMOS:

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, 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 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. 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). Το ρεύμα αυτό 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, 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. Το κακό με το μεγάλωμα της τάσης οδήγησης των transistors, δηλαδή της τάσης τροφοδοσίας, είναι ότι αυτή αυξάνει και το ΔV της εξόδου που λέγαμε παραπάνω, άρα είναι "δώρον άδωρο".

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). Η μεγαλύτερη συνηθισμένη πηγή κατανάλωσης ισχύος 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, ηλεκτρικοι 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 να δουλεύει αργά (λιγότερα ανεβοκατεβάσματα ανά δευτερόλεπτο), ή να δουλεύει γιά λίγο και μετά να κάθεται (να μην αλλάζει κατάσταση), ή τις περισσότερες φορές να εργάζεται ένα μικρό μόνο μέρος του κυκλώματος και το υπόλοιπο να κάθεται.

Up to the Home Page of CS-120

Last updated: 09 Jan. 2004, by M. Katevenis.