# 5

# Συμβολική Γλώσσα και Υπορουτίνες

## 5.1. Η Ανάγκη Εισαγωγής της Συμβολικής Γλώσσας

Τα προγράμματα μπορούν να γραφτούν σε ένα από τα παρακάτω τρία είδη γλώσσας προγραμματισμού: γλώσσα μηχανής (machine language), συμβολική γλώσσα (assembly language) ή σε κάποια γλώσσα υψηλού επιπέδου (high level language). Στις δύο τελευταίες περιπτώσεις για να εκτελεστούν πρέπει πρώτα να μεταφραστούν σε γλώσσα μηχανής.

Ως γνωστό ένα πρόγραμμα σε γλώσσα μηχανής αποτελείται από μια σκολουθία από 0 και 1. Ένα τέτοιο παράδειγμα φαίνεται στον πίνακα 5.1, όπου χρησιμοποιείται ο δυαδικός κώδικας για τον μΕ 8035Α.

**Πίνακας 5.1.** Πρόγραμμα σε δυαδικό και δεκαεξαδικό κώδικα

| Δυαδικός Κώδικας | Δεκαεξαδικός |
|---|---|
| 00111010 | 3A |
| 00001010 | 0A |
| 00000000 | 00 |
| 01001111 | 4F |
| 00111010 | 3A |
| 00001011 | 0B |
| 00000000 | 00 |
| 10000001 | 81 |
| 00110010 | 32 |
| 00001010 | 0A |
| 00000000 | 00 |
| 01110110 | 76 |

Το πρόγραμμα αυτού του πίνακα μεταφέρει δύο αριθμούς από τις θέσεις 10, 11 της μνήμης, τους προσθέτει και φυλάει το αποτέλεσμα πάλι στη μνήμη στη θέση 10.

Όπως φαίνεται από το παράδειγμα ο προγραμματισμός σε γλώσσα μηχανής παρουσιάζει δύο σοβαρά μειονεκτήματα: πρώτον είναι δύσκολο να διαβαστεί, να ελεγχθεί και να γραφεί ένα πρόγραμμα και δεύτερον χρειάζονται πολλές γραμμές προγράμματος για απλές λειτουργίες.

Ένα βήμα για να είναι τα προγράμματα πιο ευανάγνωστα και για να ελαττώσουμε τον όγκο τους είναι η χρησιμοποίηση του δεκαεξαδικού συστήματος αντί του δυαδικού.

Όπως φαίνεται το πρόγραμμα μειώθηκε σε όγκο όμως παραμένει δύσκολο στην εξήγηση της λειτουργίας του ακόμη και για κάποιον που έχει σχετική εξοικείωση με τη γλώσσα μηχανής. Επίσης ο συνολικός αριθμός των γραμμών του προγράμματος δεν μειώθηκε.

Μια πρώτη λύση στα προβλήματα αυτά δόθηκε με την εισαγωγή της συμβολικής (assembly) γλώσσας. Στην συμβολική γλώσσα υπάρχουν συμβολικά ονόματα για τις εντολές, τα δεδομένα καθώς και τις διευθύνσεις. Έτσι το προηγούμενο παράδειγμα γραμμένο σε συμβολική γλώσσα για τον μΕ 8085Α φαίνεται στον πίνακα 5.2.

**Πίνακας 5.2** Αντιστοιχία γλώσσας υψηλού επιπέδου με συμβολική και με γλώσσα μηχανής

| Γλώσσα Υψηλού Επιπέδου | Συμβολική Γλώσσα | HEX Κώδικας | Δυαδικός Κώδικας |
|---|---|---|---|
| A = A + K | LDA SUMA | 3A | 00111010 |
| | | 0A | 00001010 |
| | | 00 | 00000000 |
| | MOV C,A | 4F | 01001111 |
| | LDA SUMK | 3A | 00111010 |
| | | 0B | 00001011 |
| | | 00 | 00000000 |
| | ADD C | 81 | 10000001 |
| | STA SUMA | 32 | 00110010 |
| | | 0A | 00001010 |
| | | 00 | 00000000 |
| | HLT | 76 | 01110110 |

Στο παραπάνω πρόγραμμα η πρόσθεση συμβολίζεται με **ADD** και τα **SUMA, SUMK** είναι συμβολικά ονόματα για τις θέσεις μνήμης $10_{10}$ και $11_{10}$ αντίστοιχα.

Εδώ βλέπουμε ένα ακόμη σημείο υπεροχής του προγραμματισμού σε συμβολική γλώσσα έναντι αυτού σε γλώσσα μηχανής. Στον προγραμματισμό σε γλώσσα μηχανής όλες οι αναφορές σε διευθύνσεις μνήμης για οποιαδήποτε διεργασία (π.χ. μεταφορά δεδομένων) γίνεται σε απόλυτες διευθύνσεις μνήμης σε αντίθεση με τον προγραμματισμό σε συμβολική γλώσσα που όπως είδαμε γίνεται με συμβολικά ονόματα.

Τέλος ο προγραμματισμός γίνεται ακόμη πιό εύκολος με τη χρησιμοποίηση μιάς γλώσσας υψηλού επιπέδου. Εδώ η εντολή A=A+K αντιστοιχεί σε πολλές εντολές γλώσσας μηχανής όπως φαίνεται και από το παραπάνω παράδειγμα.

Τα προγράμματα σε συμβολική γλώσσα καθώς και σε κάποια γλώσσα υψηλού επιπέδου πρέπει να μεταφραστούν σε κώδικα μηχανής για να εκτελεστούν σε έναν μΕ. Οι γλώσσες υψηλού επιπέδου είναι πιό εύκολες

στη χρήση και είναι ανεξάρτητες από τον τύπο του μΕ που χρησιμοποιούμε. Ομως για τη χρήση τους χρειάζεται να δεσμεύσουμε ένα τμήμα της μνήμης του μΕ γιά τον μεταφραστή της γλώσσας. Επίσης ένα τέτοιο πρόγραμμα συνήθως είναι σημαντικά μεγαλύτερο σε όγκο και αργότερο όταν τρέχει από το ισοδύναμο του σε συμβολική γλώσσα.

Σημειώνουμε εδώ πως ένα πρόγραμμα γραμμένο σε γλώσσα μηχανής περιέχει ουσιαστικά μιά σειρά από bytes, μερικά από τα οποία μπορεί να είναι εντολές, άλλα δεδομένα και άλλα διευθύνσεις άλματος. Ειδικά κυκλώματα στον μΕ (τμήμα ελέγχου) στη φάση της αποκωδικοποίησης της εντολής ορίζουν πότε αυτή πρέπει να ακολουθείται από δεδομένα και πότε από διεύθυνση. Αυτά με την προϋπόθεση ότι δίνεται σωστά το σημείο έναρξης του κώδικα της εντολής. Αν γίνει για παράδειγμα διακλάδωση σε δεδομένα αυτά θεωρούνται ως εντολές από το μΕ.

Η συμβολική γλώσσα χρησιμοποιείται ευρέως γιά τον προγραμματισμό των μΕ. Γενικά είναι δυσκολότερο να γραφεί ένα πρόγραμμα σε συμβολική γλώσσα από ότι είναι να γραφεί σε μιά γλώσσα υψηλού επιπέδου. Είναι ευκολότερο όμως να μεταφράσουμε από συμβολική γλώσσα σε γλώσσα μηχανής παρά από μιά γλώσσα υψηλού επιπέδου σε γλώσσα μηχανής. Επίσης σε εφαρμογές στις οποίες ενδιαφερόμαστε γιά την ταχύτητα ή για την εξοικονόμηση μνήμης η συμβολική γλώσσα είναι η καλύτερη επιλογή. Αυτό οφείλεται στο γεγονός ότι η συμβολικής γλώσσα είναι πιο κοντά στο μΕ και αξιοποιεί καλύτερα τις δυνατότητές του.

## 5.2. Η Ανάπτυξη του Λογικού σε μΥ-Σ

Η ανάπτυξη του λογικού (software development) αφορά στην παραγωγή μιάς ακολουθίας από 0 και 1 η οποία όταν τοποθετηθεί στις κατάλληλες διευθύνσεις μνήμης ενός μΕ και εκτελεστεί φέρνει το σύστημα σε κάποια επιθυμητή κατάσταση. Γενικά αυτό συνίσταται σε τέσσερα βήματα:

i) ***Σχεδίαση (Design)***. Είναι η επιλογή της όλης δομής του προγράμματος και των δεδομένων καθώς και του αλγόριθμου που θα χρησιμοποιηθεί.
ii) ***Κωδικοποίηση (Coding)***. Είναι το γράψιμο των εντολών σε κάποια γλώσσα προγραμματισμού που υλοποιούν τον αλγόριθμο.
iii) ***Μετάφραση (Translation)***. Είναι η παραγωγή του δυαδικού κώδικα (object code) από το πηγαίο πρόγραμμα (source program) του βήματος ii.
iv) ***Ελεγχος-Διόρθωση (Testing-Debugging)***. Είναι ο έλεγχος καλής λειτουργίας (έλεγχουμε αν ο κώδικας όταν εκτελείται δίνει γιά το σύστημα το επιθυμητό αποτέλεσμα) και η διόρθωση του προγράμματος από τα

λάθη οπότε έχουμε επανάληψη των βημάτων ξεκινώντας από το βήμα i ή το ii.

Αν και τα βήματα αυτά μπορούν να εκτελεστούν με το "χέρι", πρακτικά αυτό γίνεται με τη βοήθεια υπολογιστών εκτός ίσως από πολύ μικρά προγράμματα. Γιά το λόγο αυτό έχουν αναπτυχθεί αρκετά προγράμματα εφαρμογών μερικά από τα οποία εξετάζουμε αμέσως.

**i) Πρόγραμμα εποπτείας (Monitor)**

Το monitor είναι ένα πρόγραμμα που μπορούμε να χαρακτηρίσουμε καρδιά γιά το λογισμικό του μΕπεξεργαστή. Αυτό επιτρέπει στο χρήστη να επικοινωνεί με το σύστημα, να καλεί άλλα λογικά συστήματα, να χρονοδρομολογεί την εκτέλεση προγραμμάτων και να διανέμει αποδοτικά τους "πόρους" του συστήματος (ΚΜΕ, I/O κλπ.).

**ii) Οδηγούς εισόδου εξόδου (I/O driver)**

Ο οδηγός εισόδου/εξόδου περιέχει ρουτίνες που χειρίζονται την επικοινωνία μεταξύ του μΕπεξεργαστή και της περιφερειακής μονάδας. Η ύπαρξη του είναι προαιρετική.

**iii) Διαχείριση δεδομένων ή σύστημα αρχείων**

Η διαχείριση δεδομένων ή σύστημα αρχείων επιβλέπει τη χρήση της βοηθητικής μνήμης όπως π.χ. την οργάνωση, ενημέρωση και διανομή της σε αρχεία. Η ύπαρξη του είναι προαιρετική.

**iv) Εκδότης κειμένου (Text editor)**

Ο text editor χρησιμεύει στη δημιουργία αλλά και την ανάκτηση από το σύστημα και αλλαγή ενός αρχείου με κάποιο πρόγραμμα ή δεδομένα. Κατόπιν με τον editor μπορούμε να ξανασώσουμε το αρχείο σε κάποιο δίσκο.

**v) Μεταφραστής (Translator)**

Ο Μεταφραστής μετατρέπει το πηγαίο πρόγραμμα (source program) σε εντολές γλώσσας μηχανής και παράγει έτσι τον τελικό αντικειμενικό κώδικα (object code). Υπάρχουν πολλά είδη μεταφραστών όπως assemblers, cross assemblers, self assemblers, compilers και interpreters. Οι assemblers δέχονται το πηγαίο πρόγραμμα σε συμβολική γλώσσα και παράγουν τον κώδικα γιά τον μΕπεξεργαστή.

Οι assemblers φτιάχνουν και τρέχουν κώδικες γιά τον ίδιο επεξεργαστή. Οι cross assemblers φτιάχνουν κώδικες για προγράμματα γραμμένα γιά άλλους επεξεργαστές από τον οποίο τρέχουν. Οι cross assemblers είναι γραμμένοι σε γλώσσα υψηλού επιπέδου και είναι πιο σύνθετοι από τους assemblers. Οι compilers και οι interpreters χρησιμοποιούνται όταν το πηγαίο πρόγραμμα είναι γραμμένο σε γλώσσα υψηλού επιπέδου.

**vi) Φορτωτής (Loader)**

Ο φορτωτής μεταφέρει τον κώδικα που έχει κατασκευαστεί, από την περιφερειακή μονάδα που βρίσκεται και τον τοποθετεί σε κάποιες ελεύθερες θέσεις στην κύρια μνήμη, με σκοπό να τρέξει το πρόγραμμα.

**vii) Εξομοιωτής (Simulator)**

Ο εξομοιωτής είναι ένα πρόγραμμα που τρέχει σε μεγάλους υπολογιστές γενικού σκοπού και στο οποίο προσομοιώνονται οι καταχωρητές και η μνήμη του μΕ (μΕπεξεργαστή).

**viii) Διορθωτής (Debugger)**

Ο διορθωτής είναι επίσης ένα πρόγραμμα που διευκολύνει τον έλεγχο της εκτέλεσης του κώδικα σε ένα μΕ. Όταν υπάρχει ένας εξομοιωτής τότε ο διορθωτής είναι μέρος του. Σε πολλά συστήματα εξάλλου οι χρήσεις του διορθωτή περιλαμβάνονται στο monitor πρόγραμμα.

Γιά την ανάπτυξη του λογισμικού, όπως είδαμε, μπορούμε να χρησιμοποιήσουμε είτε έναν υπολογιστή (οπότε θα έχουμε "εικονικό" τρέξιμο των προγραμμάτων με τη χρήση του εξομοιωτή είτε ένα μΥ-Σ που βασίζεται στον ίδιο μΕ (οπότε έχομε πραγματικό τρέξιμο προγραμμάτων). Εχουμε δηλαδή διαφορά στο τέταρτο βήμα (αυτό του debugging). Στη συνέχεια θα εξετάσουμε τα βήματα της ανάπτυξης του λογισμικού εάν χρησιμοποιήσουμε έναν cross assembler και έναν εξομοιωτή σε υπολογιστή γενικού σκοπού.

Το πηγαίο πρόγραμμα οδηγείται στον assembler όπου παράγεται ο ισοδύναμος κώδικας αφού το πρόγραμμα ελεγχθεί γιά συντακτικά λάθη (syntactic errors). Αν υπάρχουν τέτοια το πρόγραμμα διορθώνεται με τον εκδότη και ξαναοδηγείται στον assembler. Όταν όλα τα συντακτικά λάθη διορθωθούν ο κώδικας εκτελείται με τη χρήση του εξομοιωτή. Εδώ γίνεται έλεγχος γιά τα λογικά λάθη (logical errors).

Μερικούς τρόπους με τους οποίους γίνεται αυτός ο έλεγχος θα δούμε πιό κάτω στην παράγραφο 5.7. Αν υπάρχουν τέτοια λάθη το πρόγραμμα διορθώνεται με τον εκδότη, ξαναοδηγείται στον assembler και κατόπιν στον εξομοιωτή.

Όταν όλα τα λάθη διορθωθούν ο κώδικας φορτώνεται (download) στη μνήμη του μΥ-Σ και το λογισμικό με το υλικό δοκιμάζονται μαζί για τη συγκεκριμένη λειτουργία του συστήματος.

## 5.3. Η Σύνταξη των Εντολών στη Συμβολική Γλώσσα

Σε ένα πρόγραμμα γραμμένο σε συμβολική (assembly) γλώσσα μπορούμε να διακρίνουμε:

i) **Τις εντολές του μΕ.**

ii) **Τις οδηγίες** (directives statements) ή **ψευδοεντολές** (pseudo instructions)

προς τον assembler.

**iii) Τα σχόλια** (comment statements) που δεν επηρεάζουν τη μετάφραση του προγράμματος, αλλά βοηθούν στην κατανόηση του.

Σε αυτήν την παράγραφο θα ασχοληθούμε με τον τρόπο με τον οποίο πρέπει να γράφουμε μία εντολή του μΕ, δηλαδή με την τυπική μορφή της (format).

Μία εντολή λοιπόν αποτελείται από τα παρακάτω τέσσερα τμήματα, από τα οποία ένα ή περισσότερα μπορεί να είναι κενό.

```
ΕΠΙΓΡΑΦΗ:     ΛΕΙΤΟΥΡΓΙΑ     ΟΡΙΣΜΑ(ΤΑ)          ;Σχόλια
```

Έχουμε λοιπόν γιά κάθε ένα τμήμα χωριστά:

**(α)Τμήμα Επιγραφής (Label field)**

Στο πεδίο αυτό μπορούμε να ονομάσουμε μιά εντολή με κάποιο συμβολικό όνομα. Ολες οι επιγραφές πρέπει να αρχίζουν με αλφαβητικό χαρακτήρα στην πρώτη στήλη της γραμμής. Απαγορεύεται το ίδιο όνομα γιά επιγραφή να χρησιμοποιηθεί πάνω από μία φορά. Το τμήμα τελειώνει με δύο τελείες (άνω και κάτω) οι οποίες όμως είναι προαιρετικές. Αντίθετα είναι απαραίτητο ένα τουλάχιστον κενό να χωρίζει την επιγραφή από την υπόλοιπη γραμμή.

**(β)Τμήμα Λειτουργίας (Operation field)**

Εδώ έχομε το μνημονικό όνομα είτε της εντολής είτε της ψευδοεντολής (ή οδηγίας γιά τις οποίες θα αναφερθούμε στην παράγραφο 5.4) Για το λόγο αυτό το τμήμα αυτό αναφέρεται και σαν τμήμα Μνημονικού ονόματος (mnemonic label). Το όνομα της εντολής δεν πρέπει να αρχίζει από την πρώτη στήλη της γραμμής (ακόμη κι αν δεν υπάρχει τμήμα επιγραφής) και πρέπει να ακολουθείται από ένα τουλάχιστον κενό.

**(γ)Τμήμα Ορίσματος (Operand field)**

Τα ορίσματα εξαρτώνται από την εντολή και τη σύνταξή της. Έτσι μπορεί να έχομε μηδέν, ένα ή περισσότερα ορίσματα. Η σειρά των ορισμάτων καθορίζεται με την χρήση των όρων "πρωτεύον" (primary) γιά το πρώτο όρισμα και "δευτερεύον" (secondary) γιά τα υπόλοιπα.

**(δ)Τμήμα Σχολίων (Comments field)**

Το πεδίο με τα σχόλια μπορεί να τοποθετηθεί στην ίδια ή σε διαφορετική γραμμή. Η χρήση του είναι προαιρετική όμως απαραίτητη γιά την κατανόηση του προγράμματος ακόμη και από αυτόν που το έγραψε. *Ο μόνος κανόνας γιά το τμήμα σχολίων είναι πως πρέπει να αρχίζει με ένα* Ελληνικό ερωτηματικό (;).

Μία καλή ιδέα όταν ξεκινάμε να γράψουμε ένα πρόγραμμα σε συμβολική γλώσσα είναι να αφήνουμε ένα σταθερό χώρο γιά κάθε τμήμα έτσι ώστε στο τέλος το πρόγραμμα μας να είναι πιό ευανάγνωστο από εμάς και τους άλλους.

## 5.4. Οι Ψευδοεντολές ή Οδηγίες

Οι ψευδοεντολές συντάσσονται όπως και οι εντολές αλλά έχουν σημαντικές διαφορές από αυτές. Πρώτον δεν ανήκουν στο σύνολο των εντολών του μΕ (μικροεπεξεργαστή) και δεύτερον δεν μεταφράζονται σε κώδικα. Η χρησιμότητα τους έγκειται στο να δίνουν κάποιες πληροφορίες προς τον assembler όταν αυτός δημιουργεί τον κώδικα. Το τι είδους πληροφορίες είναι αυτές θα φανεί αμέσως με την εξέταση των κυριωτέρων ψευδοεντολών. Στην παρουσίαση της σύνταξης των ψευδοεντολών ότι παρουσιάζεται σε < > είναι προαιρετικό.

**ORG (origin)**

*Σύνταξη:* ORG έκφραση

Η ψευδοεντολή αυτή ακολουθείται από έναν αριθμό που δείχνει την απόλυτη θέση μνήμης όπου πρέπει να τοποθετηθεί η πρώτη εκτελέσιμη εντολή του προγράμματος. Αν δεν υπάρχει η ψευδοεντολή ORG πρίν την πρώτη εντολή του προγράμματος τότε η εντολή τοποθετείται στη θέση 0 της μνήμης. Μπορούμε να έχουμε περισσότερες από μία ψευδοεντολές ORG οι οποίες θα δείχνουν τη θέση της μνήμης γιά την πρώτη εκτελέσιμη εντολή που τις ακολουθεί. Παράδειγμα στις παρακάτω εντολές ενός προγράμματος έχουμε:

```
ORG 0100H       ;Ο κώδικας των εντολών αυτών αρχίζει από τη θέση
                ;μνήμης 0100H
ΕΝΤΟΛΕΣ ASSEMBLY
ORG 0200H       ;Ο κώδικας των εντολών αυτών αρχίζει από τη θέση
                ;μνήμης 0200H
ΕΝΤΟΛΕΣ ASSEMBLY
```

**END**

*Σύνταξη:* **END** *<έκφραση>*

Η ψευδοεντολή αυτή δηλώνει το φυσικό τέλος του προγράμματος και πρέπει να είναι μοναδική γιά κάθε πρόγραμμα. Όταν ο assembler αναγνωρίσει το φυσικό τέλος του προγράμματος αρχίζει τη δημιουργία του κώδικα και (πιθανώς) της λίστας του πηγαίου προγράμματος.

**DS (define storage)**

*Σύνταξη:* <επιγραφή:> DS έκφραση

Με την ψευδοεντολή αυτή φυλάσσεται ένας αριθμός από θέσεις μνήμης γιά αποθήκευση δεδομένων. Η πρώτη από τις θέσεις μπορεί να αναφερθεί με τη συμβολική επιγραφή. Προσοχή πρέπει να δώσουμε στις δύο τελείες μετά την επιγραφή. Ο αριθμός των bytes που δεσμεύονται είναι ίσος με την τιμή της έκφρασης. Γιά παράδειγμα αν θέλουμε να κρατήσουμε

δέκα bytes γιά αποθήκευση δεδομένων με το όνομα DEKA δηλώνουμε:

```
DEKA:    DS   10
```

**DB, DW, DD**

define byte, define word, define double word

*Σύνταξη:* <επιγραφή:> `DB` λίστα
<επιγραφή:> `DW` λίστα
<επιγραφή:> `DD` λίστα

Με τον όρο λίστα εννοούμε τιμές δεδομένων ή εκφράσεις.

Οι ψευδοεντολές αυτές έχουν σαν αποτέλεσμα να αποθηκεύει ο assembler σε μία ή περισσότερες θέσεις μνήμης τις συγκεκριμένες τιμές αρχίζοντας από τη διεύθυνση της επιγραφής. Οι θέσεις αυτές μπορεί να είναι ομάδες από bytes (8 bits), λέξεις (16 bits) ή διπλές λέξεις (32 bits). Ο όρος έκφραση εδώ αναφέρεται σε μιά ή περισσότερες αριθμητικές ή λογικές εκφράσεις. Επίσης στην περίπτωση της ψευδοεντολής DB μπορούμε στη λίστα να περιλάβουμε και χαρακτήρες σε εισαγωγικά. Ο asssembler αναλαμβάνει την αντικατάστασή τους σε ASCII μορφή.

**EQU (equate)**

*Σύνταξη:* όνομα `EQU` έκφραση

Με την ψευδοεντολή αυτή ένα συμβολικό όνομα αντιστοιχείται με μιά σταθερά ή διεύθυνση ή γενικότερα έκφραση. Η εντολή αυτή είναι ισοδύναμη με την εντολή const της Pascal και έχει σαν αποτέλεσμα όποτε συναντάται το συμβολικό όνομα στο πρόγραμμα να χρησιμοποιείται η έκφραση στην οποίαν ισοδυναμεί. Πρέπει να σημειώσουμε πως εδώ το όνομα δεν είναι επιγραφή και δεν ακολουθείται από δύο τελείες (:) αν και βρίσκεται στο πεδίο της επιγραφής. Επίσης η τιμή του ονόματος δεν μπορεί να αλλάξει με το πρόγραμμα.

**SET**

*Σύνταξη:* όνομα `SET` έκφραση

Εχει τα ίδια αποτελέσματα με την `EQU`. Η διαφορά της βρίσκεται στο ότι μπορεί η τιμή του συμβολικού ονόματος να αλλάξει μέσα στο πηγαίο πρόγραμμα. Έτσι πολλές `SET` μπορεί να αναφέρονται στο ίδιο συμβολικό όνομα μέσα στο ίδιο πρόγραμμα.

**IF, ENDIF**

*Σύνταξη:*

```
IF έκφραση
        εντολές assembly
ENDIF
```

Εάν η τιμή της έκφρασης είναι 0 οι εντολές έως την `ENDIF`

σημαίνουν ενώ εάν είναι 1 οι εντολές μεταφράζονται από τον assembler.

Υπάρχει ακόμη ένας σημαντικός αριθμός από ψευδοεντολές που όμως αλλάζουν από υλοποίηση σε υλοποίηση. Μπορούμε να αναφέρουμε ενδεικτικά γιά τον MASM 8086 τις **SEGMENT/ENDS, ASSUME, GROUP, LABEL** και **PROG/ENDP,** που θα αναφερθούν και στο κεφάλαιο 9 του παρόντος βιβλίου.

Τελειώνοντας αναφέρουμε πως τις ψευδοεντολές (pseudo instructions) μπορούμε να συναντήσουμε στην βιβλιογραφία και ως assembler directives, nongenerative instructions ή declaratives.

## 5.5. Assemblers δύο Περασμάτων

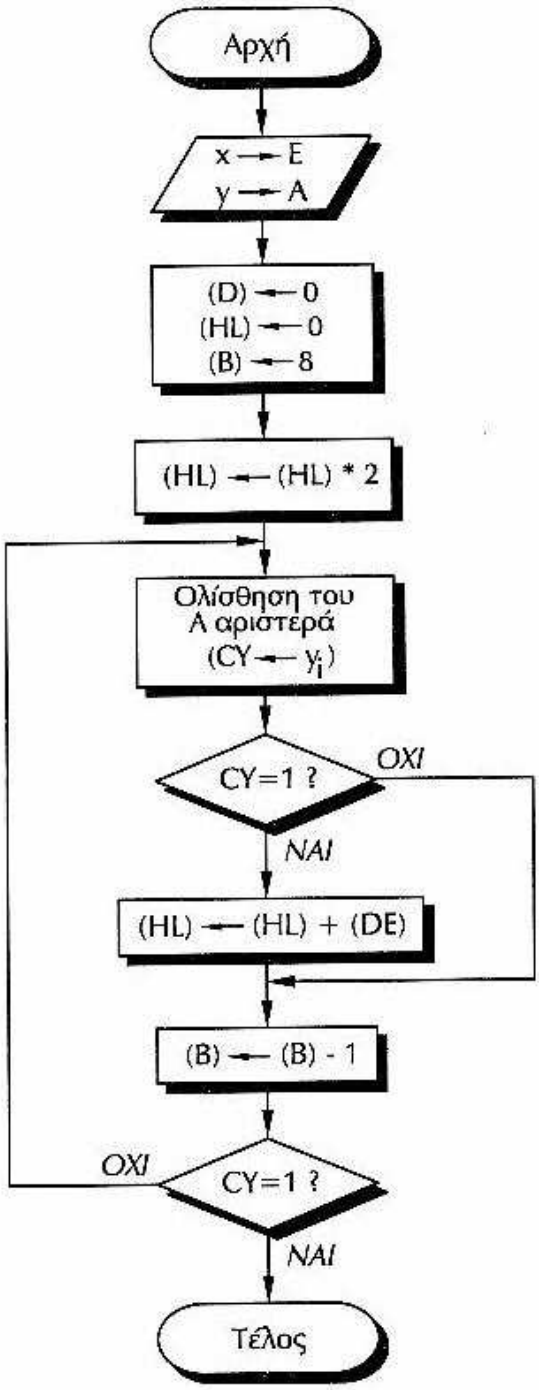
Γιά να φανεί η χρησιμότητα ενός assembler δύο περασμάτων (two pass assembler) καθώς και η διαφορά του από έναν assemblers απλού περάσματος θα δούμε ένα παράδειγμα. Στο σχήμα 5.1 φαίνεται το διάγραμμα ροής του πρόγραμματος το οποίο εκτελεί τον πολλαπλασιασμό δύο δυαδικών αριθμών των 8 bits. Ο πολλαπλασιαστέος και ο πολλαπλασιαστής βρίσκονται στους καταχωρητές E και A αντίστοιχα. Το γινόμενο είναι δυαδικός αριθμός των 16 bits και επιστρέφεται μέσα από τους καταχωρητές H, L. Το πρόγραμμα σε πηγαία (assembly) μορφή που υλοποιεί τον αλγόριθμο του πολλαπλασιασμού φαίνεται στην αντίστοιχη στήλη του πίνακα 5.3. Η εργασία του assembler είναι με βάση το πηγαίο να παράγει τον αντικειμενικό κώδικα που στη συγκεκριμένη περίπτωση αντιστοιχεί στις δύο πρώτες στήλες του πίνακα.

Ένας απλός assember μπορεί να μεταφράσει ένα πρόγραμμα σε ένα πέρασμα εάν αυτό δεν περιέχει επιγραφές ή εάν εκτελούνται εντολές άλματος μόνο προς τα πίσω, δηλαδή σε επιγραφές που η διεύθυνση έχει ήδη εξεταστεί και είναι γνωστή. Στην περίπτωση που γίνει άλμα σε μία διεύθυνση που βρίσκεται μετά την εντολή άλματος, ο απλός assember δεν μπορεί να αντικαταστήσει την επιγραφή με συγκεκριμένη τιμή.

Για παράδειγμα στην εντολή **JNC CHCNT** (7η γραμμή του πίνακα 5.3) ο assember δεν μπορεί να αντικαταστήσει το **CHCNT** με τον αντίστοιχο δυαδικό κώδικα γιατί δεν ξέρει ακόμη σε ποιά διεύθυνση αντιστοιχεί.

Το πρόβλημα αυτό λύνεται με τον assember δύο περασμάτων. Στο πρώτο πέρασμα ο assember διαβάζει όλο το πρόγραμμα και μεταφράζει τις εντολές στον αντίστοιχο κώδικα μηχανής, χρησιμοποιώντας έναν πίνακα με κώδικες εντολών (operation code table) που περιέχει γιά κάθε όνομα εντολής, τον αντίστοιχο κώδικα της και ένα "περιγραφητή" (descriptor word) ο οποίος δείχνει τον αριθμό των bytes της εντολής και το format των δεδομένων στο όρισμα της εντολής. Ένας απαριθμητής θέσης που έχει αρχικοποιηθεί από την εντολή **ORG**, αυξάνει κάθε φορά κατά το

μήκος της εντολής. Έτσι στο πρώτο πέρασμα προσδιορίζεται η τιμή κάθε επιγραφής και συμβολικού δεδομένου. Δηλαδή αντιστοιχείται σε κάθε σύμβολο, είτε αυτό είναι επιγραφή (συμβολική διεύθυνση) είτε συμβολικό δεδομένο σε ψευδοεντολή, η τιμή που αντιπροσωπεύει (όπου υπάρχει επιγραφή αντιστοιχείται με την διεύθυνση που συμβολίζει και όπου υπάρχει σύμβολο με την τιμή του). Στη συνέχεια οι τιμές αυτές αποθηκεύονται στον πίνακα συμβόλων.




**Σχήμα 5.1** Διάγραμμα ροής πολλαπλασιασμού

Στο δεύτερο πέρασμα τα συμβολικά δεδομένα και οι διευθύνσεις αντικαθίστανται από τις πραγματικές τους τιμές και προκύπτει ο

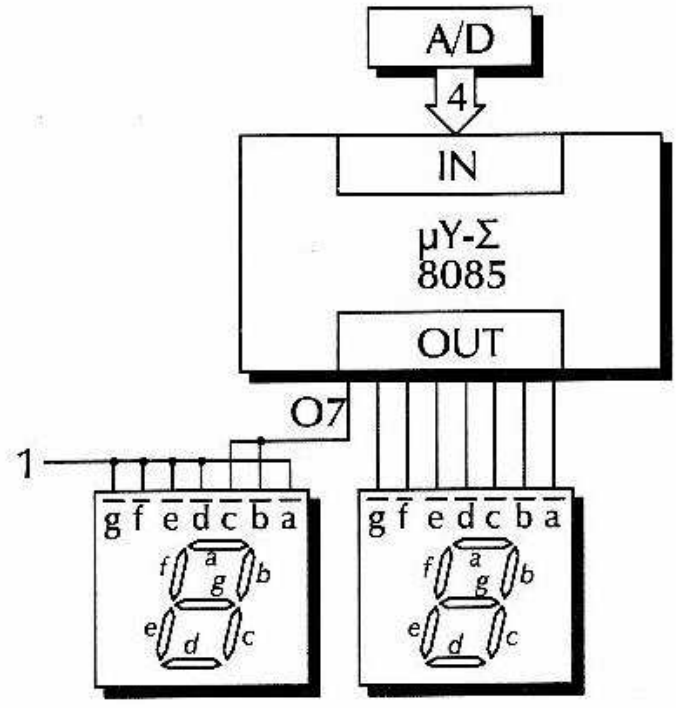
αντικειμενικός κώδικας (object code).

Στον πίνακα 5.3 φαίνεται σε πρόγραμμα η υλοποίηση του διαγράμματος ροής του προγράμματος πολλαπλασιασμού του προηγούμενου σχήματος. Η λίστα του προγράμματος είναι όπως δίνεται από τον assembler και δείχνει την αντιστοιχία ανάμεσα στον πηγαίο κώδικα και τον αντικειμενικό. Στον πίνακα έχουν υπογραμμισθεί οι πραγματικές τιμές των συμβόλων, που μπορούν να τοποθετηθούν μόνο κατά το δεύτερο πέρασμα του assembler.

**Πίνακας 5.3** Το αρχείο με τη λίστα του προγράμματος

| Διεύθυνση | OBJ | Γραμμή | Πηγαίο | Σχόλια |
|---|---|---|---|---|
| | | 1 | COUNT EQU 8 | |
| | | 2 | ORG 0100H | |
| 0100 | 16 00 | 3 | MVI D,0 | ; (DE) = Πολ/τέος |
| 0102 | 21 00 00 | 4 | LXI H,0 | ; (HL) = Γινόμενο |
| 0105 | 06 08 | 5 | MVI B,COUNT | ; (B) = Μετρητής |
| 0107 | 29 | 6 | MULT:DAD H | ; Γιν. = 2×Γινόμ. |
| 0108 | 17 | 7 | RAL | ;Ολίσθηση Αριστερά |
| 0109 | D2 0D 01 | 8 | JNC CHCNT | ; Είναι CY = 1 |
| 010C | 19 | 9 | DAD D | ; Γιν. = Γιν. + Πολ. |
| 010D | 05 | 10 | CHCNT:DCR B | ; B = B - 1 |
| 010E | C2 07 01 | 11 | JNZ MULT | ; Αν B>0 ξανά |
| 0111 | 76 | 12 | HLT | ; Τέλος |
| | | | END | |

Στη συνέχεια δίνεται ένα δεύτερο παράδειγμα προγράμματος σε assembly.




**Σχήμα 5.2.** Το μΥ-Σ για την απεικόνιση δεδομένων των 4 bit σε δύο ενδείκτες 7 τμημάτων

Όπως φαίνεται στο σχήμα 5.2, ένα μΥ σύστημα με τον 8085 συνδέεται και λαμβάνει δεδομένα των 4 bits από έναν μετατατροπέα A/D. Στη συνέχεια τα τυπώνει σε δύο ενδείκτες 7 τμημάτων (7-segment display) μέσω μιας πόρτας εξόδου. Η σύνδεση των ακίδων της πόρτας εξόδου με τα τμήματα του ενδείκτη είναι η ακόλουθη:

$O_7 \rightarrow$ b′ και c′, $O_6 \rightarrow$ g, $O_5 \rightarrow$ f, $O_4 \rightarrow$ e, $O_3 \rightarrow$ d, $O_2 \rightarrow$ c, $O_1 \rightarrow$ b και $O_0 \rightarrow$ a

Τα τονούμενα τμήματα b′, c′ αφορούν το δεύτερο ενδείκτη 7 τμημάτων. Για να εμφανιστεί η κατάλληλη τιμή (από 0 έως 15) στους ενδείκτες ο 8085 τοποθετεί την κατάλληλη τιμή στην πόρτα εξόδου ώστε να φωτιστούν τα κατάλληλα τμήματα των δύο ενδεικτών. Οι τιμές αυτές είναι τοποθετημένες σε έναν πίνακα στη μνήμη. Έτσι το πρόγραμμα που ακολουθεί ανάλογα με την τιμή που δέχεται στην πόρτα εισόδου, διαβάζει το κατάλληλο στοιχείο του πίνακα και το στέλνει την πόρτα εξόδου.

```
PORT_IN   EQU  20H
PORT_OUT  EQU  40H
          ORG  100H
CONTINUE: IN   PORT_IN          ;Διάβασε το δεδομένο στην είσοδο
          MOV  E,A
          MVI  D,0
          LXI  H,SEVEN_SEG      ;Φόρτωσε τη διεύθυνση του πίνακα
                                ;στο ζεύγος H-L
          DAD  D                ;Προσδιόρισε τη διεύθυνση του στοιχείου
                                ;στον πίνακα SEVEN_SEG
          MOV  A,M              ;Φόρτωσε το σχήμα του ψηφίου
          OUT  PORT_OUT         ;Στείλε την τιμή στην πόρτα εξόδου
          JMP  CONTINUE         ;Συνεχής απεικόνιση
;Πίνακας τιμών
SEVEN_SEG:
          DB   COH,F9H,A4H,B0H,99H,92H,82H,F8H
          DB   80H,90H 40H,79H,24H,30H,19H,12H
          END
```

Αν δώσουμε το παραπάνω πρόγραμμα με το όνομα DISPLAY.ASM σαν είσοδο σε έναν μεταφραστή (assembler), δίνει σαν έξοδο τα αρχεία, DISPLAY.OBJ και DISPLAY.LST. Το πρώτο επειδή είναι αρχείο σε δυαδική μορφή δεν μπορούμε να το παρουσιάσουμε, ενώ το δεύτερο είναι αρχείο κειμένου και φαίνεται παρακάτω. Το αρχείο αυτό περιέχει τον πηγαίο κώδικα (SOURCE CODE) συνοδευόμενο από τους αριθμούς γραμμών του πηγαίου αρχείου (LINE) DISPLAY.ASM και τον αντικειμενικό κώδικα (OBJ) με τις αντίστοιχες διευθύνσεις (ADDR), που παράγει ο assembler.

```
                INPUT FILE NAME : DISPLAY.ASM
                OUTPUT FILE NAME: DISPLAY.LST

ADDR   OBJ          LINE    SOURCE CODE

0020                  1     PORT_IN   EQU      20H
0040                  2     PORT_OUT  EQU      40H
0100                  3     ORG  100H
0100   DB 20          4  CONTINUE:IN  PORT_IN
0102   5F             5     MOV  E,A
0103   16 00          6     MVI  D,0
0105   21 0F 01       7     LXI  H,SEVEN_SEG
                      8
0108   19             9     DAD  D
                      10
0109   7E             11    MOV  A,M
010A   D3 40          12    OUT  PORT_OUT
010C   C3 00 01       13    JMP  CONTINUE
010F                  14
010F                  15  SEVEN_SEG:
010F   C0 F9 A4 B0    16    DB C0H,F9H,A4H,B0H,99H,92H,82H,F8H
0113   99 92 82 F8
0117   80 90 40 79    17    DB 80H,90H,40H,79H,24H,30H,19H,12H
011B   24 30 19 12
011F                  18    END

       S Y M B O L I C   R E F E R E N C E   T A B L E

CONTINUE   0100     PORT_IN   = 0020     PORT_OUT  = 0040
SEVEN_SEG  010F

LINES ASSEMBLED :  18              ASSEMBLY ERRORS :  0
```

## 5.6 Μακροεντολές

Πολλές φορές γράφοντας ένα πρόγραμμα παρατηρούμε πως ένα ή περισσότερα τμήματα του επαναλαμβάνονται πάνω από μία φορά. Όπως οι περισσότερες γλώσσες προγραμματισμού έτσι και η συμβολική γλώσσα μας δίνει τη δυνατότητα να χρησιμοποιήσουμε όσες φορές θέλομε ένα τμήμα προγράμματος χωρίς να είμαστε αναγκασμένοι να το γράφουμε κάθε φορά που το χρειαζόμαστε. Η δυνατότητα αυτή μας παρέχει την ευκολία να χρησιμοποιήσουμε τμήματα κάποιου άλλου προγράμματος ή προγραμματιστή ή και να δημιουργήσουμε μιά βιβλιοθήκη με τα υποπρογράμματα (subprograms) που συνήθως χρησιμοποιούμε. Υπάρχουν δύο είδη υποπρογραμμάτων, οι μακροεντολές (macros) και οι υπορουτίνες (subroutines).

***i) Μακροεντολές (ή ανοικτό υποπρόγραμμα)***

*Σύνταξη:* όνομα **MACRO** <παράμετρος>
Σώμα της macro
**ENDM**

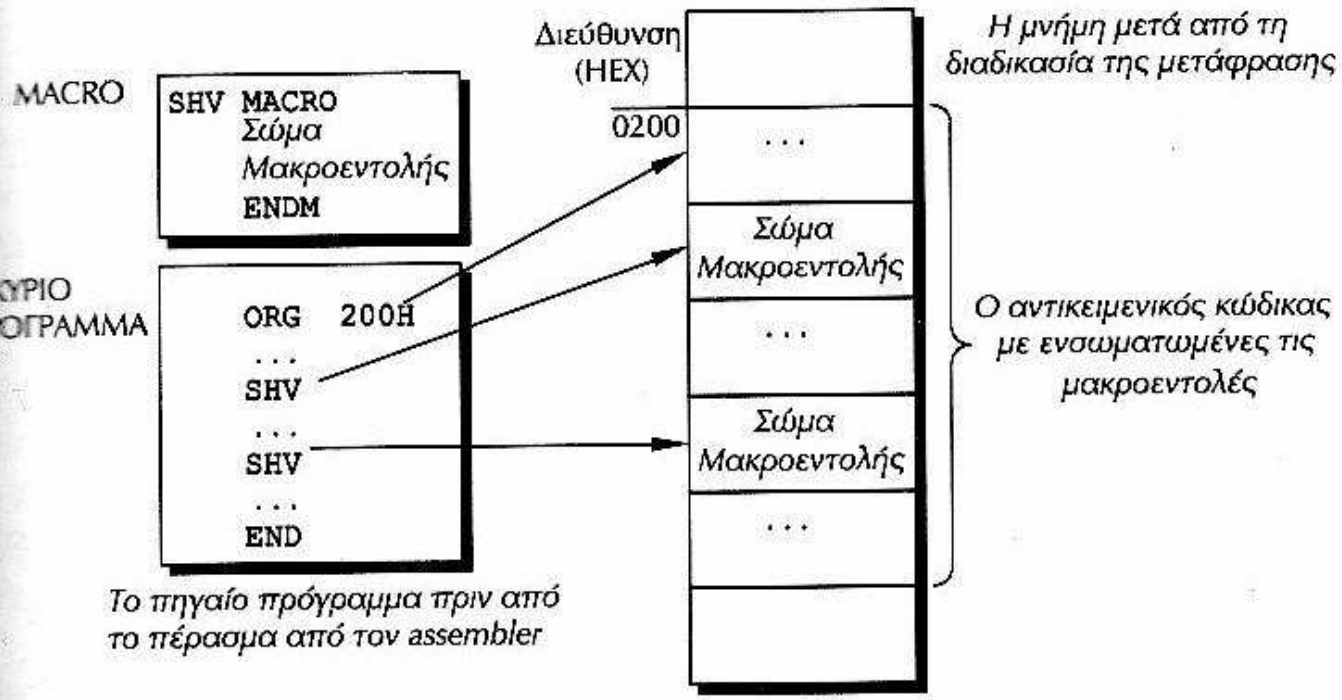
Γιά να καλέσουμε τη μακροεντολή στο πρόγραμμα μας απλά γράφουμε το όνομα της με την τιμή της παραμέτρου αν έχει, στη θέση που θέλουμε τις εντολές από τις οποίες αποτελείται. Συνήθως ο ορισμός των μακροεντολών προηγείται του κυρίως προγράμματος. Κατά τη δημιουργία του κώδικα όταν ο assembler συναντά μιά μακροεντολή τοποθετεί στη θέση της όλο το σώμα της εντολής αυτής. Έτσι εάν γιά παράδειγμα σε ένα πρόγραμμα καλείται επτά φορές η ίδια μακροεντολή στον αντικειμενικό κώδικα του προγράμματος θα περιλαμβάνεται επτά φορές ο κώδικας της μακροεντολής. Αυτό φαίνεται καλύτερα στο σχήμα 5.3, όπου **SHV** είναι το όνομα της μακροεντολής.

***ii) Υπορουτίνες (ή κλειστό υποπρόγραμμα)***

Στις υπορουτίνες θα αναφερθούμε με λεπτομέρεια σε επόμενη παράγραφο. Εδώ απλώς θα σημειώσουμε πως είναι ένα τμήμα προγράμματος που αφού δημιουργηθεί ο κώδικάς του, τοποθετείται σε μια περιοχή στη μνήμη έξω από τον κώδικα του κυρίως προγράμματος. Όταν καλείται η υπορουτίνα τότε διακόπτεται η εκτέλεση του κυρίως προγράμματος και ο έλεγχος μεταφέρεται στην πρώτη εντολή της υπορουτίνας.

Όταν τελειώσει η υπορουτίνα (η τελευταία εντολή της είναι πάντα η **RETURN**) ο έλεγχος αυτόματα επιστρέφει στο σημείο του κυρίως προγράμματος που είχε διακοπεί. Έτσι εάν μιά υπορουτίνα κληθεί επτά φορές αυτό δεν σημαίνει πως στον κώδικα του προγράμματος θα έχουμε τις εντολές της υπορουτίνας επτά φορές.

Παρατηρούμε αμέσως τη διαφορά μεταξύ μακροεντολών και υπορουτίνων. Με τη χρήση υπορουτινών έχουμε οικονομία στη μνήμη. Επίσης με τη χρήση μακροεντολών οι εντολές τους εισέρχονται στο κυρίως πρόγραμμα πριν το χρόνο μετάφρασης ενώ με τις υπορουτίνες αυτό γίνεται κατά την εκτέλεση του προγράμματος. Από την άποψη της ταχύτητας οι μακροεντολές δίνουν καλύτερα αποτελέσματα αφού δεν επιβαρύνουν το πρόγραμμα με εντολές κλήσης και επιστροφής υπορουτινών.




**Σχήμα 5.3** Η μακροεντολή SHV καλείται δύο φορές στο πρόγραμμα

## 5.7. Έλεγχος ενός Προγράμματος

Σε κάθε προγραμματιστή έχει τύχει κάποιο πρόγραμμά του να μην έχει συντακτικά λάθη και όμως να μην δίνει το επιθυμητό αποτέλεσμα. Λέμε τότε πως το πρόγραμμα έχει λογικά λάθη. Η ανάγκη γιά γρηγορότερο εντοπισμό των λαθών οδήγησε τους κατασκευαστές στη δημιουργία ειδικών εντολών γιά το σκοπό αυτό.

Μία τέτοια εντολή είναι η TRACE και μας δίνει τη δυνατότητα να τρέξουμε ένα πρόγραμμα εντολή προς εντολή και να κάνουμε έλεγχο των θέσεων μνήμης.

Πολλές φορές πάλι είναι επιθυμητό να ξέρουμε τις τιμές που έχουν οι καταχωρητές σε κάποιο συγκεκριμένο σημείο του προγράμματος ή να ελέγξουμε την ορθότητα του προγράμματος μέχρι ένα συγκεκριμένο σημείο. Αυτό επιτυγχάνεται με τη χρησιμοποίηση σημείων διακοπής (breakpoint). Όταν ένα τέτοιο σημείο αναγνωριστεί (ανάλογα με την υλοποίηση θα υπάρχει κάποια εντολή γιά το σκοπό αυτό) η εκτέλεση του προγράμματος σταματά και ο έλεγχος γυρνά στο πρόγραμμα εποπτείας (monitor). Από εκεί μπορούμε να δώσουμε την κατάλληλη εντολή στο πρόγραμμα εποπτείας ανάλογα με τη λειτουργία ή τον έλεγχο που θέλουμε να εκτελεστεί π.χ. έλεγχος των καταχωρητών, συνέχιση της εκτέλεσης του προγράμματος κ.α.

## 5.8. Η Στοίβα (Stack) και οι λειτουργίες της

Η ανάγκη αποθήκευσης μιας πληροφορίας προσωρινά σε κάποιο σημείο ενός προγράμματος, καθώς και η ανάγνωσή της όταν απαιτηθεί, εξυπηρετείται από μια δομή φύλαξης, τη στοίβα (ή σωρό). Η στοίβα είναι μια δομή LIFO (Last In - First Out), δηλαδή το δεδομένο με τη μεγαλύτερη προτεραιότητα ανάγνωσης (δηλαδή το πρώτο που μπορούμε να διαβάσουμε) είναι αυτό που αποθηκεύτηκε τελευταίο.

Οι εντολές εγγραφής και ανάγνωσης, από την κορυφή της στοίβας πάντα, είναι οι **PUSH** και **POP** αντίστοιχα. Στον μικροεπεξεργαστή 8085A ένας καταχωρητής των 16 bits ονομάζομενος SP (stack pointer), δηλαδή δείκτης στοίβας, δείχνει την τρέχουσα διεύθυνση της κορυφής της στοίβας, όπου έχει αποθηκευθεί το πιο πρόσφατο δεδομένο. Η διαθέσιμη για τη στοίβα μνήμη εξαρτάται από τις σχεδιαστικές ανάγκες του συστήματος.

Οι εντολές **PUSH rp** και **POP rp** αναφέρονται σε ζεύγη καταχωρητών και κάνουν τις εξής λειτουργίες:

**PUSH rp**

((SP)-1) ← (rh)
((SP)-2) ← (rl)
(SP) ← (SP)-2

Αν το ζεύγος είναι πχ. το B-C στη στοίβα έχουμε πρώτα το περιεχόμενο του καταχωρητή B και μετά του καταχωρητή C.

**POP rp**

(rl) ← ((SP))
(rh) ← ((SP)+1)
(SP) ← (SP)+2

Αντίστροφα εδώ με την εντολή **POP B** λαμβάνεται πρώτα ο καταχωρητής C και μετά ο καταχωρητής B.

Ένα παράδειγμα για την λειτουργία των εντολών **PUSH-POP** του 8085 είναι η ανταλλαγή των περιεχομένων των καταχωρητών BC και HL, χρησιμοποιώντας τη στοίβα για προσωρινή φύλαξη. Σημειώνεται οτι πριν εκτελεστεί κάποια από τις εντολές **PUSH-POP** που αφορούν τη στοίβα, πρέπει προηγουμένως να ορίσουμε μια αρχική διεύθυνση, στην οποία να δείχνει ο SP.
Αυτό γίνεται με την εντολή:

```
LXI  SP,data16      ;Φορτώνει άμεσα στον καταχωρητή SP
                    ;Το δεδομένο data των 16 bits
```

Αλλος τρόπος είναι η εντολή **SPHL** (με λειτουργία (SP)←(H)(L)) όπου

τα περιεχόμενα του διπλού καταχωρητή H-L μεταφέρονται στον SP. Οι εντολές που απαιτούνται για την ανταλλαγή των δεδομένων των καταχωρητών B-C και H-L, είναι:

```
PUSH H          ;τοποθετεί το ζεύγος HL στη στοίβα
PUSH B          ;τοποθετεί το ζεύγος BC στη στοίβα
POP  H          ;αφαιρεί το ζεύγος BC από τη στοίβα και το βάζει στο HL
POP  B          ;αφαιρεί το ζεύγος HL από τη στοίβα και το βάζει στο BC
```

Επίσης υπάρχουν και ειδικές εντολές PUSH και POP, που τοποθετούν ή αφαιρούν από τη στοίβα , τα περιεχόμενα του καταχωρητή A και του καταχωρητή σημαίας F.

Η εντολή **PUSH PSW** τοποθετεί στη στοίβα τα περιεχόμενα του ζεύγους καταχωρητών A-F. Ο καταχωρητής A-F ονομάζεται και processor status word και αποτελείται από τον συσσωρευτή A και τον καταχωρητή σημαιών, που όπως έχει ήδη αναφερθεί στο κεφάλαιο 2 περιέχει τις πέντε σημαίες S, Z, AC, P, CY, στην εξής μορφή:

| S | Z | x | AC | x | P | x | CY |
|---|---|---|---|---|---|---|---|

όπου *x=δεν χρησιμοποιείται*

Οι πληροφορίες που περιέχονται στις σημαίες είναι απαραίτητες όταν επιστρέφουμε σε ένα πρόγραμμα στην περίπτωση, που αυτές χρησιμοποιούνται στη συνέχεια για την διακλάδωση του προγράμματος. Με την εντολή που ακολουθεί σώζονται οι πληροφορίες αυτές.

**PUSH PSW**

(SP)-1) ← (A)
(SP)-2) ← (F)
(SP) ← (SP)-2

Παρατηρούμε οτι μετά την αποθήκευση των δύο καταχωρητών στο σωρό, ο δείκτης ελαττώνεται κατά δύο, δείχνοντας έτσι το γέμισμα του σωρού. Τα ίδια φυσικά ισχύουν και για τα άλλα ζεύγη καταχωρητών.

Η εντολή, η οποία μεταφέρει τα περιεχόμενα της κορυφής της στοίβας στους καταχωρητές A και F, είναι η **POP PSW**.

**POP PSW**

(F) ← ((SP))
(A) ← ((SP)+1)
(SP) ← (SP)+2

Αντίθετα εδώ η λήψη δύο δεδομένων από το σωρό αυξάνει κατά δύο την τιμή δείχνοντας έτσι το άδειασμα του σωρού. Επίσης με την εντολή

**XTHL**

(exchange top of stack with H and L)
(L) ↔ ((SP))
(H) ↔ ((SP)+1)

Ανταλλάσσουμε τα δεδομένα της κορυφής της στοίβας με τα περιεχόμενα των καταχωρητών H και L.

## 5.9 Υπορουτίνες

Ένα φαινόμενο, που παρατηρείται συχνά κατά την εκτέλεση προγραμμάτων, είναι η ανάγκη να εκτελείται η ίδια ακριβώς σειρά εντολών σε διαφορετικά σημεία ενός ή και περισσοτέρων προγραμμάτων. Με τη δημιουργία μακροεντολών μπορούμε να απλοποιήσουμε τη γραφή ενός τέτοιου προγράμματος. Ομως κάτι τέτοιο απαιτεί αρκετή μνήμη όταν έχουμε πολλές αναφορές στη μακροεντολή.

Στην περίπτωση αυτή ο assembler αντικαθιστά κάθε φορά στη θέση της μακροεντολή τον αντίστοιχο κώδικα της σειράς των εντολών. Η ανάγκη λοιπόν που προκύπτει είναι να αντικαθίσταται το σύνολο των εντολών από τον αντίστοιχο κώδικα μόνο μια φορά και να μπορεί να καλείται από οποιοδήποτε σημείο του προγράμματος. Η χρήση υπορουτινών δίνει ακριβώς αυτή τη δυνατότητα. Οι υπορουτίνες δηλώνονται με μία χαρακτηριστική για κάθε περίπτωση ονομασία (επιγραφή) που δείχνει τη διεύθυνσή τους.

Η εντολή με την οποία μπορούμε να έχουμε πρόσβαση σε μια υπορουτίνα είναι η **CALL** και η **Συνθήκη**.

Με την εντολή **CALL** στο κυρίως πρόγραμμα ουσιαστικά γίνεται μια λειτουργία **PUSH** του PC (μετρητής προγράμματος) και φορτώνεται σε αυτόν η διεύθυνση της υπορουτίνας. Αναλυτικότερα γίνονται τα ακόλουθα:

**CALL Address**

((SP) - 1) ← (PCH)
((SP) - 2) ← (PCL)
(SP) ← (SP) - 2
(PC) ← Address

Η εντολή CALL αποτελείται από 3 bytes, το πρώτο περιέχει τον κώδικα της εντολής, το δεύτερο το λιγότερο σημαντικό byte της διεύθυνσης της

υπορουτίνας και το τρίτο το πιο σημαντικό byte αυτής της διεύθυνσης.

Τα 8 bits υψηλής τάξης της διεύθυνσης της επόμενης εντολής μετακινούνται στη θέση μνήμης της οποίας η διεύθυνση είναι κατά 1 μικρότερη από το περιεχόμενο του καταχωρητή SP. Τα 8 bits χαμηλής τάξης της διεύθυνσης της επόμενης εντολής μετακινούνται στη θέση μνήμης της οποίας η διεύθυνση είναι κατά 2 μικρότερη από το περιεχόμενο του καταχωρητή SP. Το περιεχόμενο του SP μειώνεται συνολικά κατά 2. Ο έλεγχος μεταφέρεται στην εντολή της οποίας η διεύθυνση καθορίζεται στα byte 3 και byte 2 της τρέχουσας εντολής.

Δηλαδή, φυλάσσεται στην κορυφή μιας στοίβας η διεύθυνση της εντολής που ακολουθεί την **CALL** και στη συνέχεια δίνεται στο μετρητή προγράμματος η διεύθυνση της πρώτης εντολής της υπορουτίνας, ώστε να γίνει το απαραίτητο άλμα. Η τελευταία εντολή κάθε υπορουτίνας πρέπει να είναι μια εντολή **RET**urn, έτσι ώστε να επιστρέφει ο έλεγχος στο κυρίως πρόγραμμα.

Με την εντολή **RET**, η οποία είναι μια εντολή επιστροφής από ρουτίνα χωρίς συνθήκη, εκτελείται μια λειτουργία **POP** του καταχωρητή PC, δηλαδή μεταφέρεται στον PC η διεύθυνση της εντολής που ακολουθεί την εντολή **CALL** και συνεχίζεται έτσι η εκτέλεση του κυρίως προγράμματος, που διακόπηκε προσωρινά. Αναλυτικότερα γίνονται τα ακόλουθα:

**RET**

(PCL) ← (SP)
(PCH) ← ((SP)+1)
(SP) ← (SP) + 2

Στην περίπτωση που καλείται μια υπορουτίνα μέσα από μια άλλη, η οποία δεν έχει τελειώσει, δημιουργείται μια φωλιασμένη υπορουτίνα. Το επίπεδο φωλιάσματος αυξάνει κατά ένα, για κάθε διαδοχική εντολή **CALL** που υπάρχει, χωρίς να παρεμβάλλεται η εντολή **RET**. Ο αριθμός των φωλιασμένων υπορουτινών περιορίζεται από τη μέγιστη διαθέσιμη μνήμη του συστήματος.

Ένα πράγμα που θα πρέπει να προσεχθεί ιδιαίτερα, είναι η χρήση εντολών **PUSH** μέσα στην υπορουτίνα. Αυτές οι εντολές πρέπει να ακολουθούνται από τις αντίστοιχες εντολές **POP**, πριν την εντολή **RET** της συγκεκριμένης υπορουτίνας. Με αυτό τον τρόπο εξασφαλίζεται ότι, με την εκτέλεση της εντολής **RET**, το περιεχόμενο της κορυφής της στοίβας είναι η σωστή διεύθυνση επιστροφής στο κυρίως πρόγραμμα.

Επίσης, στην περίπτωση που τα περιεχόμενα των καταχωρητών χρειάζονται και μετά την εκτέλεση της υπορουτίνας, φροντίζουμε έτσι ώστε αυτά να σωθούν προσωρινά στη στοίβα είτε στο κυρίως πρόγραμμα,

ή στην ίδια την υπορουτίνα και στη συνέχεια να ανακληθούν.

Στο παράδειγμα που ακολουθεί, σώζονται τα περιεχόμενα των καταχωρητών του μικροεπεξεργαστή στην αρχή μιας υπορουτίνας, έτσι ώστε να είναι σε θέση να χρησιμοποιηθούν μετά το πέρας αυτής.

```
0000            LXI  SP,610H
                Κύριο Πρόγραμμα
                .....
0180            CALL SUBR_A
0183            (επόμενη εντολή)
                Υπόλοιπο Κυρίου Προγράμματος
                .....
                END

;Υπορουτίνα A
0400  SUBR_A:  PUSH H
0401           PUSH D
0402           PUSH B
0403           PUSH PSW
               Κύριο Σώμα Υπορουτίνας A
               .....
0420           CALL SUBR_B
               Υπόλοιπες εντολές υπορουτίνας A
               ....
0436           POP  PSW
0437           POP  B
0438           POP  D
0439           POP  H
0440           RET

;Υπορουτίνα B
0510  SUBR_B:  PUSH H
0511           PUSH D
0512           PUSH B
0513           PUSH PSW
               Κύριο Σώμα Υπορουτίνας B
               .....
0536           POP  PSW
0537           POP  B
0538           POP  D
0539           POP  H
0540           RET
```

Στο παραπάνω πρόγραμμα όταν εκτελεστεί η εντολή **CALL SUBR_A** ο δείκτης στοίβας γίνεται SP=60EH και στις διευθύνσεις 60FH και 60EH φυλάσσονται τα δεδομένα 01 και 83H αντίστοιχα. Ο μετρητής

προγράμματος γίνεται PC=0400 και ο έλεγχος μεταφέρεται στην πρώτη εντολή της υπορουτίνας δηλαδή εκτελείται η εντολή **PUSH H**. Όταν το πρόγραμμα φτάσει στη διεύθυνση 0420H καλείται η υπορουτίνα SUBR_B. Ο δείκτης στοίβας ήταν 0606H και γίνεται 604H και στις διευθύνσεις 605H και 604H φυλάσσονται τα δεδομένα 04 και 23H αντίστοιχα. Στη συνέχεια εκτελούνται οι εντολές της υπορουτίνας SUBR_B και μετά την **RET** ο έλεγχος δίνεται στην εντολή που ακολουθεί την **CALL SUBR_B** της υπορουτίνα SUBR_A (PC=0423H). Κατόπιν εκτελούνται όλες οι εντολές μέχρι και τη **RET**, η εκτέλεση της οποίας μεταφέρει τον έλεγχο στο κυρίως πρόγραμμα στην εντολή που ακολουθεί μετά την **CALL SUBR_A** (PC=0183H).

Είναι δυνατόν η κλήση μιας υπορουτίνας να γίνει υπό συνθήκη. Αυτό επιτυγχάνεται με την εντολή:

**Cσυνθήκη Address** (συνθήκη=NZ,Z,NC,C,PO,PE,P,M βλ. παραρτ. 1)

Αν ισχύει η συνθήκη τότε συμβαίνουν οι παρακάτω ενέργειες:
(SP-1) ← (PCH)
(SP-2) ← (PCL)
(SP) (SP)-2
(PC) ← byte3 byte2
Αλλιώς εκτελείται η επόμενη εντολή.

Δηλαδή εαν ισχύει η συνθήκη της εντολής τότε όλες οι ενέργειες που αναφέρθηκαν στην εντολή **CALL** εκτελούνται. Διαφορετικά εκτελείται η επόμενη εντολή.

Μια ειδική μορφή παρόμοια της εντολής **CALL** είναι η **RST n** (**RESTART n**) η οποία μεταφέρει τον έλεγχο σε μια από οκτώ προκαθορισμένες διευθύνσεις. Ορίζεται ως εξής:

**RST n**

(SP)-1) ← (PCH) Φυλάσσεται στο σωρό
(SP)-2) ← (PCL) ο μετρητής προγράμματος
(SP) ← (SP)-2
(PC) ← 8 x n $0 \leq n \leq 7$

Ο έλεγχος δίνεται σε μια νέα θέση μνήμης συγκεκριμένη για κάθε εντολή. Αναλυτικότερα αυτές είναι:

| Εντολή | Διεύθυνση |
|---|---|
| RST 0 | 0000h |
| RST 1 | 0008h |
| RST 2 | 0010h |
| RST 3 | 0018h |
| RST 4 | 0020h |
| RST 5 | 0028h |
| RST 6 | 0030h |
| RST 7 | 0038h |

Η εντολή **RST** ονομάζεται και λογισμική διακοπή (software interrupt). Στις πιο πάνω θέσεις τοποθετούνται εντολές άλματος για τις αντίστοιχες ρουτίνες εξυπηρέτησης της κάθε μιας λογισμικής διακοπής.

## 5.10 Πέρασμα Παραμέτρων

Μια πολύ χρήσιμη δυνατότητα των υπορουτινών είναι η επεξεργασία δεδομένων που εισέρχονται από το κυρίως πρόγραμμα και στη συνέχεια η επιστροφή τους σ'αυτό. Η διαδικασία αυτή ονομάζεται πέρασμα παραμέτρων και μπορεί να γίνει με πολλούς τρόπους. Οι πιο συνηθισμένοι απ'αυτούς είναι:

**α)** μέσω εσωτερικών καταχωρητών
**β)** μέσω κρατημένων θέσεων μνήμης
**γ)** μέσω στοίβας

Ο πρώτος τρόπος ενδείκνυται στις περιπτώσεις όπου ο αριθμός των μεταφερόμενων δεδομένων είναι μικρότερος του αριθμού των καταχωρητών που μπορούμε να χρησιμοποιήσουμε.

Σαν παράδειγμα θα μπορούσαμε να αναφέρουμε μια υπορουτίνα MULT, που δημιουργούμε για τον 8085A, η οποία να έχει σαν εισόδους δύο 8-bits αριθμούς και να επιστρέφει το 16-bits γινόμενό τους. Το πρόγραμμα πολλαπλασιασμού του πίνακα 5.3, που έχει ήδη παρουσιαστεί, μπορεί να μετατραπεί σε υπορουτίνα αν τερματισθεί με την εντολή **RET** αντί της **HLT**. Η τοποθέτηση του πολλαπλασιαστέου στον καταχωρητή A και του πολλαπλασιαστή στον E πρέπει να γίνει πρίν τη κλήση της υπορουτίνας, ενώ το αποτέλεσμα της πράξης σχηματίζεται στον H-L πριν εκτελεστεί η RETurn. Μετά την επιστροφή στο κυρίως πρόγραμμα το αποτέλεσμα μπορεί να ληφθεί από τον διπλό καταχωρητή H-L.

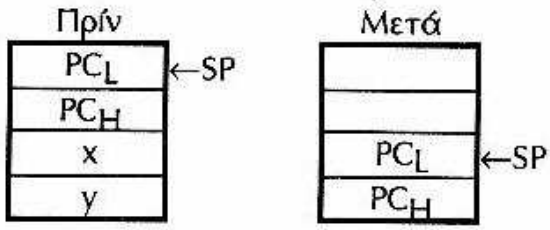
Το πέρασμα παραμέτρων μεταξύ κυρίως προγράμματος και μιας υπορουτίνας ή μεταξύ υπορουτινών μέσω κρατημένων θέσεων μνήμης, γίνεται με τη οδηγία προς τον Assembler DS (define storage), η οποία

δημιουργεί αυτές τις θέσεις. Η παράμετρος αναφέρεται με το συμβολικό της όνομα, δηλαδή την ετικέτα του Assembler, που φροντίζει τη φύλαξή της.

Στην περίπτωση που έχουμε ένα μεγάλο πλήθος Ν παραμέτρων, που πρέπει να περαστούν στην υπορουτίνα για να υπολογιστεί π.χ. ο μέσος όρος των αριθμών αυτών, χρησιμοποιούνται οι καταχωρητές HL και DE. Στον HL φορτώνεται η θέση μνήμης στην οποία υπάρχει η τιμή του Ν, ενώ οι αμέσως επόμενες Ν θέσεις μνήμης περιέχουν τους αριθμούς, των οποίων ο μέσος όρος θα υπολογιστεί. Στον DE βρίσκεται η διεύθυνση στην οποία επιστρέφεται το αποτέλεσμα.

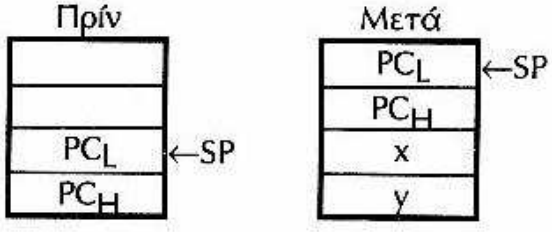
Η χρησιμοποιήση της στοίβας είναι επίσης μια μέθοδος για πέρασμα παραμέτρων. Η εισαγωγή των παραμέτρων στη στοίβα γίνεται στο κυρίως πρόγραμμα, χρησιμοποιώντας διαδοχικές εντολές **PUSH**. Μετά την κλήση της υπορουτίνας ο δείκτης στοίβας δείχνει τη διεύθυνση επιστροφής η οποία ακολουθείται από τις παραμέτρους. Από τη στιγμή που ο έλεγχος περνά στην υπορουτίνα, τα δεδομένα της στοίβας μπορούν να χρησιμοποιηθούν, αφού φύγει από την κορυφή η διεύθυνση επιστροφής και σωθεί σε κάποιο καταχωρητή ή στη μνήμη. Όταν βγούν και επεξεργαστούν από τη στοίβα όλα τα δεδομένα, γίνεται αν χρειαστεί εισαγωγή των αποτελεσμάτων στη στοίβα. Στην συνέχεια τοποθετείται στην κορυφή της στοίβας η διεύθυνση επιστροφής και εκτελείται η εντολή RETurn. Ένα παράδειγμα σε μορφή μακροεντολής που δείχνει τη λήψη ενός 16-bits δεδομένου από το σωρό με επαναφορά της διεύθυνσης επιστροφής μνήμης είναι το παρακάτω:

```
PARAM_IN MACRO
     POP  D     ;DE←PC
     POP  H     ;L←x,H←y
     PUSH D     ;στοίβα←PC
ENDM
```




Αν θέλουμε να επιστρέψουμε μέσα από τη στοίβα δεδομένα που βρίσκονται στον καταχωρητή H-L αυτό μπορεί να γίνει με τη μακροεντολή PARAM_OUT, που δίνεται παρακάτω:

```
PARAM_OUT MACRO
     POP  D     ;DE←PC
     PUSH H     ;στοίβα χ,y
     PUSH D     ;στοίβα←PC
ENDM
```



Υπάρχουν περιπτώσεις που μια υπορουτίνα μπορεί να καλεί τον εαυτό της. Οι ρουτίνες αυτές ονομάζονται αναδρομικές (Reenterant routines). Στην περίπτωση αυτή η υπορουτίνα πρέπει να είναι κατάλληλα δομημένη για να μη δημιουργείται πρόβλημα. Ένα παράδειγμα τέτοιας υπορουτίνας δίνεται στη συνέχεια και αφορά τον υπολογισμό της σειράς S(N)=1+2+...+N με βάση τη σχέση:

*Αν* N>1 *τότε* S(N)=S(N-1)+N *διαφορετικά για* N=1 S(1)=1

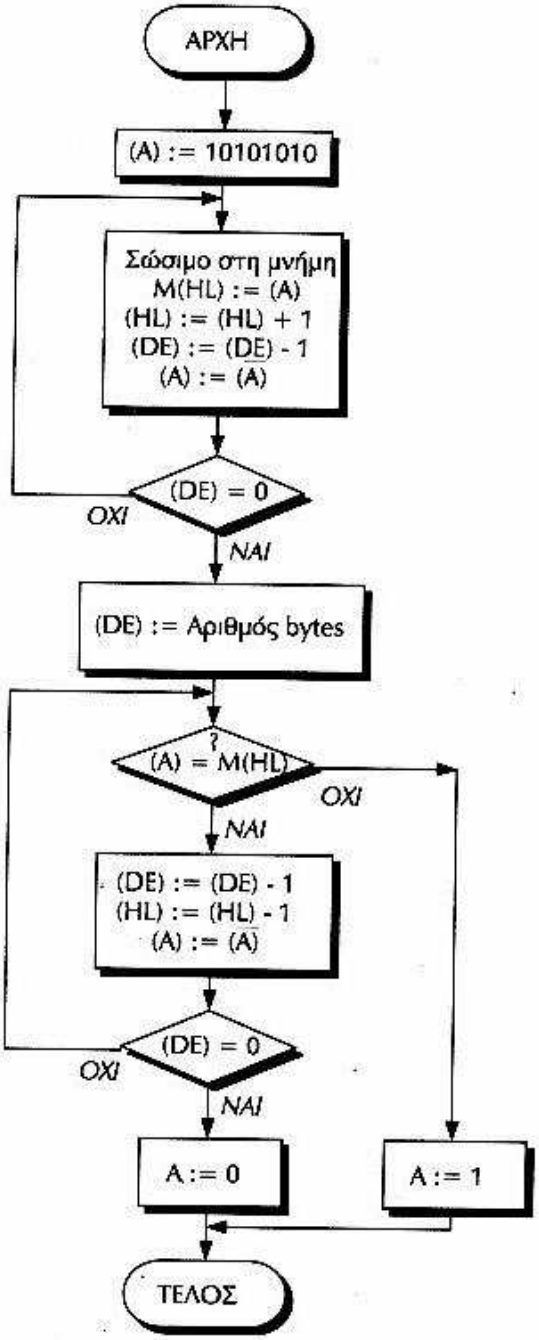
Το πρόγραμμα δίνεται σε μορφή αναδρομικής ρουτίνας που δέχεται σαν παράμετρο μέσω του καταχωρητή E τον αριθμό N και επιστρέφει το αποτέλεσμα από τον HL. Υποθέτουμε πως ο καταχωρητής D είναι μηδέν.

```
SUM:                    ;Αναδρομική ρουτίνα υπολογισμού της σειράς 1+2+...+N
     MOV  A,E
     CPI  1             ;Αν είναι 1 η ρουτίνα τερματίζεται επιστρέφοντας την
     JZ   ADDR          ;τιμή 1
     PUSH D             ;Ο αριθμός N τοποθετείται στη στοίβα για την επόμενη
                        ;κλήση της SUM
     DCR  E             ;N←N-1
     CALL SUM           ;Καλείται αναδρομικά η SUM για τον υπολογισμό του
N_RET:                  ; προηγούμενου όρου S(N-1), συνολικά N-1 φορές
     POP  D             ;Παίρνουμε από το σωρό την παράμετρο N (D=0,E=N)
     DAD  D             ;Πρόσθεση του επόμενου όρου της σειρά
     RET                ;Η εντολή αυτή στέλνει το πρόγραμμα N φορές στη
                        ; θέση N_RET
ADDR:
     LXI  HL,1
     RET
```

## 5.11 Παράδειγμα Υπορουτίνας

Έχοντας αναλύσει στις προηγούμενες παραγράφους τις διάφορες λειτουργίες της στοίβας, τον τρόπο κλήσης και εκτέλεσης υπορουτινών, καθώς και το πέρασμα παραμέτρων μέσα σ'αυτές, παρακάτω δίνεται ένα παράδειγμα ολοκληρωμένης υπορουτίνας.

Με την υπορουτίνα αυτή, γράφεται στις διαδοχικές θέσεις μνήμης RAM (read write memory) κάποιος δυαδικός αριθμός και στη συνέχεια διαβάζεται για επαλήθευση. Συγκεκριμένα, με την κλήση της υπορουτίνας στο ζεύγος καταχωρητών HL, υπάρχει η αρχική διεύθυνση της μνήμης, στην οποία θα γίνει η εγγραφή. Στο ζεύγος DE υπάρχει το πλήθος των bytes που θα γραφούν. Αρχικά σώζονται τα περιεχόμενα των DE στη στοίβα για να χρησιμοποιηθούν αργότερα.




**Σχήμα 5.4** Διάγραμμα ροής ελέγχου μνήμης RAM

Στη συνέχεια, γράφεται στην αρχική διεύθυνση μνήμης ο αριθμός 10101010, στην επόμενη το συμπληρωματικό του 01010101 και επαναλαβάνεται η ίδια διαδικασία μέχρι να μηδενιστεί ο μετρητής των bytes. Μετά τη λήξη της διαδικασίας εγγραφής στη μνήμη η τιμή του πλήθους των bytes ανακτάται από τη στοίβα και τοποθετείται ξανά στους D-E. Αυτή η τιμή χρειάζεται, για να γίνει μια νέα αναδρομή στη μνήμη σε αντίστροφη σειρά για σύγκριση του αριθμού που έχει γραφεί, με αυτόν που έπρεπε να γραφεί. Στην περίπτωση που βρεθεί κάποιο λάθος η υπορουτίνα θέτει τον καταχωρητή A ίσο με 1 και επιστρέφει, έχοντας στους καταχωρητές HL την πρώτη αποτυχούσα διεύθυνση. Αν δεν βρεθεί λάθος, ο καταχωρητής A επιστρέφεται μηδενισμένος.

Στο σχήμα 5.4, φαίνεται το διάγραμμα ροής που περιγράφει τη λειτουργία της υπορουτίνας και παρακάτω παρατίθεται το αντίστοιχο πρόγραμμα assembly. Για την απλοποίηση του προγράμματος εισάγεται η μακροεντολή REG16_FLAG που ενεργοποιεί τη σημαία μηδενισμού (Z) για τον διπλό καταχωρητή DE.

```
REG16_FLAG     MACRO
      MOV  B,A              ;Φυλλάσεται ο καταχωρητής A
      MOV  A,E              ;Οταν και οι δύο καταχωρητές  E και D είναι 0,
      ORA  D                ; Η σημαία Z γίνεται 0
      MOV  A,B
ENDM

;Υπορουτίνα ελέγχου μνήμης
      MVI  A,10101010B      ;Φόρτωση του byte ελέγχου
      PUSH D                ;Σώσιμο του πλήθους των bytes
WRITE:
      MOV  M,A              ;Εγγραφή στη μνήμη
      CMA                   ; Δημιουργία συμπληρωματικού byte ελέγχου
      INX  H                ;Αύξηση του δείκτη της διεύθυνσης μνήμης
      DCX  D                ;Μείωση του μετρητή των bytes
      REG16_FLAG            ;Ελεγχος μηδενισμού καταχ. BC
      JNZ  WRITE            ;Αν δεν είναι 0 ξαναγράψε
      POP  D                ;Φόρτωση του πλήθους των bytes

;Αντίστροφη ανάγνωση και έλεγχος της μνήμης
READ:
      DCX  H                ;Μείωση του δείκτη της διεύθυνσης μνήμης
                            ;για να δείχνει το τελευταίο byte
      CMA                   ;Επαναφορά του byte ελέγχου
      CMP  M                ;Ελεγχος ορθής εγγραφής
      JNZ  ERROR            ;Σε περίπτωση λάθους τερματισμός
      DCX  D                ;Μείωση του μετρητή των bytes
      REG16_FLAG            ;Ελεγχος μηδενισμού καταχ. BC
      JNZ  READ             ;Αν δεν είναι 0 ξαναδιάβασε
OK:
      MVI  A,0              ;Δε βρέθηκε σφάλμα, επιστρέφεται το 0
      RET
ERROR:
      MVI A,1               ;Βρέθηκε σφάλμα, επιστρέφεται το 1
      RET
      END
```

## 5.12 Προγράμματα με Σύνθετες Αριθμητικές και Λογικές Πράξεις

***Παράδειγμα 1ο: Πολλαπλασιασμός***

Θα υλοποιηθεί ο πολλαπλασιασμός δύο δυαδικών αριθμών των 8 bits (χωρίς πρόσημο), που βρίσκονται στις θέσεις 40 και 41 της μνήμης. Το γινόμενο είναι δυαδικός αριθμός των 16 bits πρέπει να τοποθετείται στις θέσεις 42 (το LSByte) και 43 (το MSByte). Θα αποθηκεύσουμε τον πολλαπλασιαστέο στο ζεύγος των καταχωρητών και τον πολλαπλασιαστή στον καταχωρητή Η.

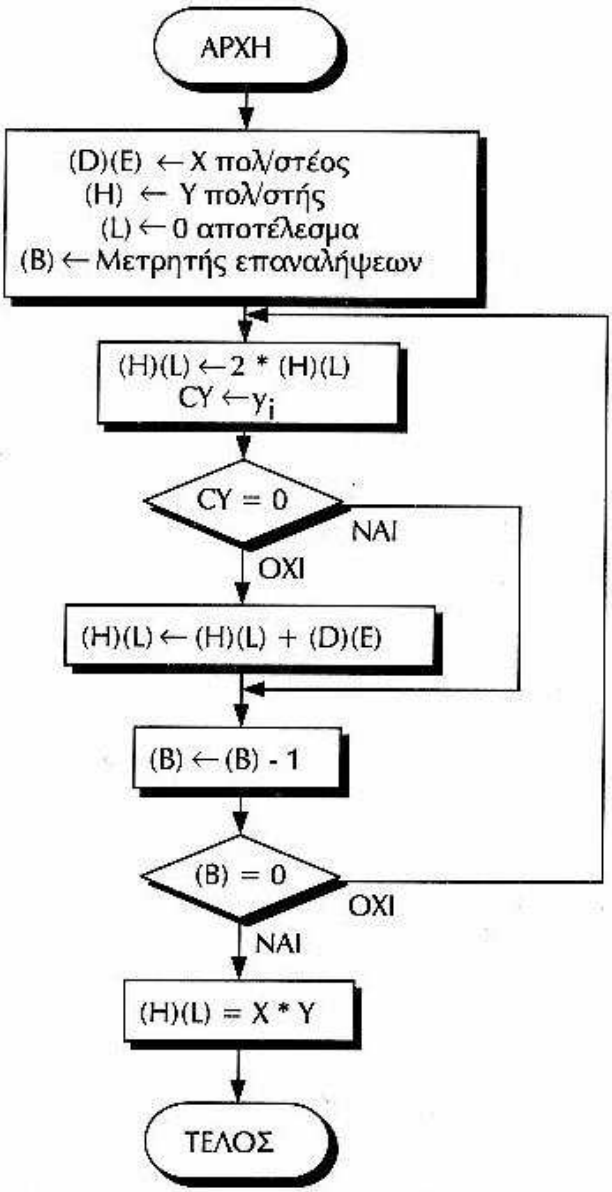
Το γινόμενο, που θα είναι δυαδικός αριθμός των 16 bits θα αποθηκευτεί στο ζεύγος των καταχωρητών. Οι αριθμοί που θα πολλαπλασιαστούν είναι των 8 bits, συνεπώς θα χρειαστούν 8 επαναλήψεις. Ως μετρητής των επαναλήψεων αυτών θα χρησιμοποιηθεί ο καταχωρητής Β. Ο αλγόριθμος αυτός είναι σε πιο βελτιωμένη μορφή σε σχέση με αυτόν του σχήματος 5.1 που παρουσιάστηκε στην αρχή του κεφαλαίου.

Η διαδικασία της πράξης του πολλαπλασιασμού έχει ως εξής: Στην αρχή του βρόγχου διπλασιάζεται το αποτέλεσμα (με ολίσθηση μιας θέσης αριστερά). Στη συνέχεια εξετάζεται το κρατούμενο που αντιστοιχεί στο αριστερότερο ψηφίο του πολλαπλασιαστή και αν διαπιστωθεί οτι αυτό είναι 1, προστίθεται ο πολλαπλασιαστέος στο αποτέλεσμα. Αλλιώς δεν προστίθεται τίποτα.

Ακολουθεί νέος διπλασιασμός του αποτελέσματος και έλεγχος του επόμενου bit. Πρέπει να σημειωθεί ότι ο καταχωρητής Η-L έχει διπλή χρήση. Χρησιμεύει για τη συσσώρευση του αποτελέσματος και ταυτόχρονα σε κάθε βήμα εκταμιεύει ένα ψηφίο του πολλαπλασιαστή στον CY. Σε κάθε βήμα ο καταχωρητής Η-L περιλαμβάνει και ένα τμήμα του πολλαπλασιαστή Υ. Η παραπάνω διαδικασία συνεχίζεται έως ότου εξεταστούν και τα 8 bits του πολλαπλασιαστή. Ακολουθεί το αντίστοιχο πρόγραμμα:

```
      LXI  H, 40H
      MOV  E,M        ;(D)(E)=0, M(40)=πολ/στέος
      MVI  D,0
      INX  H
      MOV  A,M        ;(A) ← M(41)= πολ/στής
      LXI  H,0        ;(H)(L) = 0
      MVI  B,8H       ;αρχικοποίηση του μετρητή επαναλήψεων
MULT: DAD  H          ;(H)(L)← 2× (H)(L), ολίσθηση του
      RAL             ;αποτελέσματος μια θέση αριστερά
```

```
      JNC   ADDR        ;αν A7 = 0
      DAD   D           ;πρόσθεση του πολ/στέου στο ενδιάμεσο αποτέλεσμα
ADDR: DCR   B
      JNZ   MULT
      SHLD  42H         ;M(42) ← (L), M(43) ← (H)
      HLT
```




**Σχήμα 5.5** Διάγραμμα ροής πολλαπλασιασμού

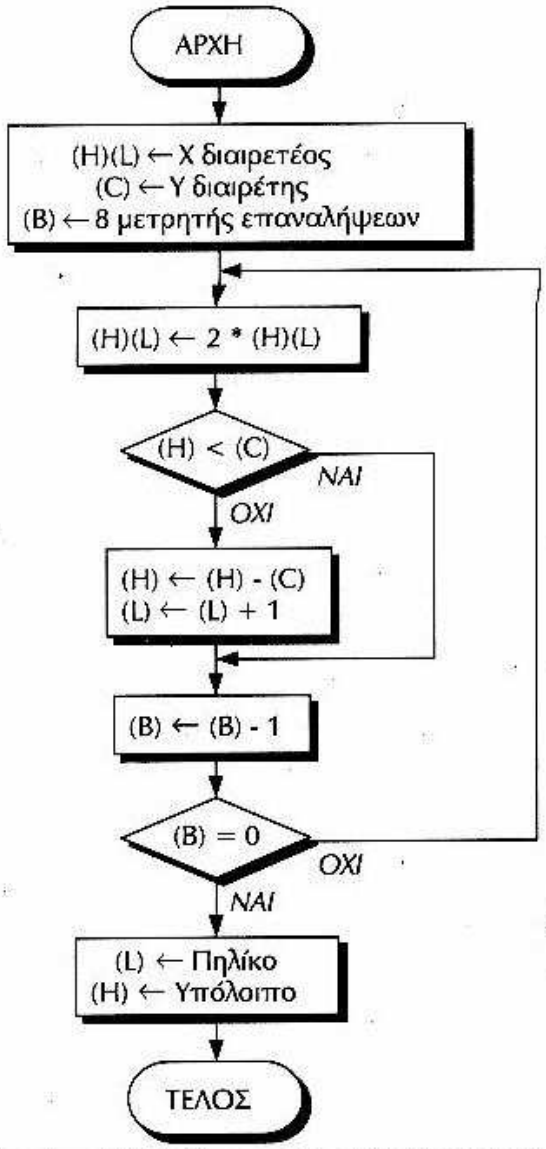
### *Παράδειγμα 2ο: Διαίρεση*

Θα εκτελέσουμε τη διαίρεση αριθμού 16 bits που βρίσκεται στις θέσεις 40 και 41 (το MSByte στη θεση 41) με αριθμό 8 bits που βρίσκεται στη θέση 42. Το πηλίκο αποθηκεύεται στη θέση 43, ενώ το υπόλοιπο στη θεση 44. Υποθέτουμε ότι ο διαιρετέος (Δ) είναι μεγαλύτερος του διαιρέτη (δ) και ότι το MSBit του διαιρετέου είναι 0, ενώ το αμέσως επόμενο ψηφίο

του διαιρέτη είναι 1. Γενικότερα για να μην έχουμε υπερχείλιση στο πηλίκο πρέπει 256δ>Δ.

Θα αποθηκεύσουμε το διαιρετέο στο ζεύγος των καταχωρητών (H)(L) και τον διαιρέτη στον καταχωρητή (C). Το πηλίκο θα δημιουργηθεί στον καταχωρητή (L) και το υπόλοιπο στον καταχωρητή (H).

Χρειάζονται 8 επαναλήψεις. Ως μετρητή των επαναλήψεων θα χρησιμοποιήσουμε τον καταχωρητή (B). Η διαδικασία της πράξης της διαίρεσης είναι η παρακάτω. Στην αρχή του βρόγχου διπλασιάζουμε το αποτέλεσμα (ολίσθηση μια θέση αριστερά).




**Σχήμα 5.6** Διάγραμμα ροής διαίρεσης

Στη συνέχεια εξετάζουμε αν το υπόλοιπο είναι μικρότερο από το διαιρέτη. Αν είναι, μειώνουμε τον μετρητή κατά 1 και επιστρέφουμε στο βρόγχο συνεχίζοντας έως ότου μηδενιστεί ο μετρητής. Εδώ το ψηφίο του πηλίκου είναι μηδέν και έχει ήδη εισαχθεί στον Η με την προηγούμενη ολίσθηση του ξεύγους H-L. Αν το υπόλοιπο είναι μεγαλύτερο από το

διαιρέτη, αφαιρούμε το διαιρέτη από το υπόλοιπο, αυξάνουμε τον καταχωρητή Η κατά 1 και μειώνουμε τον μετρητή κατά 1. Εδώ το ψηφίο του πηλίκου είναι 1. Με την προηγούμενη ολίσθηση του H-L είχε εισαχθεί το 0. Ετσι η αύξηση κατά 1 του καταχωρητή L θέτει το ψηφίο του πηλίκου.

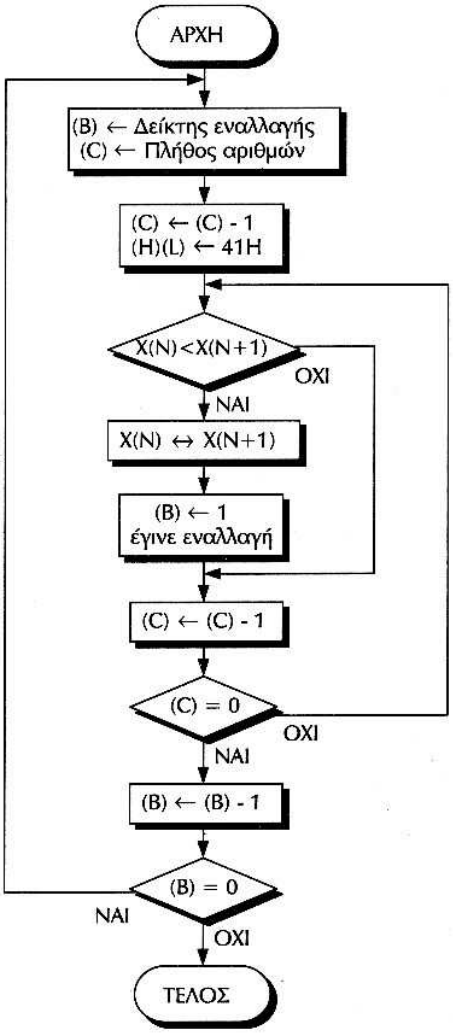
Αν ο μετρητής δεν έχει μηδενιστεί, επιστρέφουμε στο βρόγχο. Διαφορετικά, στον καταχωρητή (L) έχει δημιουργηθεί το πηλίκο, στον καταχωρητή (H) το υπόλοιπο και γίνεται αποθήκευση τους στις θέσεις μνήμης με διευθύνσεις 43 και 44 αντίστοιχα. Το αντίστοιχο πρόγραμμα σε assembly είναι το ακόλουθο:

```
      LHLD 40H        ; (L) ← M(40), (H) ← M(41)
      LDA  42H        ; (A) ← M(42)
      MOV  C,A
      MVI  B,8H
DIV:  DAD  H          ; (H)(L) ← (H)(L) + (H)(L)
      MOV  A,H
      SUB  C          ; (A) ← (A) -(C)
      JC   ADDR       ; αν (H) < (C)
      MOV  H,A
      INR  L
ADDR: DCR  B
      JNZ  DIV
      SHLD 43H        ; M(43) ← (L), M(44) ← (H)
      HLT
```

### *Παράδειγμα 3ο: Κατάταξη αριθμών*

Με το πρόγραμμα αυτό κατατάσσονται σε σειρά φθίνοντος μεγέθους ένα σύνολο αριθμών (χωρίς πρόσημο) που το πλήθος του βρίσκεται στη θέση 40 και η αρχή του είναι η θέση 41.

Προκειμένου να πετύχουμε κατάταξη αριθμών σε σειρά φθίνοντος μεγέθους, εφαρμόζουμε τη μέθοδο των φυσαλλίδων, δηλαδή πραγματοποιούμε διαδοχικά περάσματα στην ακολουθία των αριθμών και αντιστρέφουμε τους διαδοχικούς αριθμούς που δεν είναι σε φθίνουσα σειρά. Η διαδικασία αυτή επαναλαμβάνεται μέχρις ότου να μη χρειαστεί να γίνει εναλλαγή σε κάποιο πέρασμα. Τότε, με το τέλος του περάσματος και με την εντολή DCR B, το (B) γίνεται 1 (δείκτης εναλλαγής) και το πρόγραμμα τελειώνει. Αν όμως σε κάποιο πέρασμα γίνεται εναλλαγή, στο τέλος του περάσματος είναι (B)=0 και γίνεται και νέο πέρασμα. Το διάγραμμα ροής φαίνεται στο σχήμα 5.7.




**Σχήμα 5.7** Διάγραμμα ροής του προγράμματος κατάταξης αριθμών

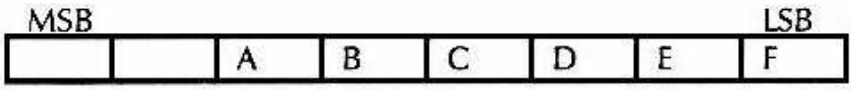
Το αντίστοιχο πρόγραμμα σε assembly είναι το ακόλουθο:

```
SORT:
    MVI  B,0        ;αρχικοποίηση δείκτη εναλλαγών
    LXI  H,40H
    MOV  C,M        ;(C) ← πλήθος αριθμών
    DCR  C
```

```
      INX   H
COMP: MOV   A,M         ;Συγκρίνονται δύο διαδοχικές τιμές του πίνακα
      INX   H
      CMP   M
      JNC   CNT         ;αν M((H)(L)) > M((H)(L)+1) δεν χρειάζεται εναλλαγή
      MOV   D,M
      MOV   M,A         ;Αλλιώς
      DCX   H
      MOV   M,D
      INX   H
      MVI   B,1H        ;έγινε εναλλαγή
CNT:  DCR   C           ;Να σαρωθεί όλος ο πίνακας
      JNZ   COMP
      DCR   B
      JZ    SORT        ;αν έγινε έστω και μία εναλλαγή επανέλαβε
      HLT
```

***Παράδειγμα 4ο: Εξομοίωση λογικής συνάρτησης***

Θα εξομοιωθεί με πρόγραμμα που ακολουθεί η λογική συνάρτηση $G = A\,\overline{B}\,C\,D + A\,\overline{B}\,\overline{C}\,D\,\overline{E}\,F$. Υποθέτουμε ότι οι μεταβλητές εισόδου βρίσκονται στα 6 μικρότερης αξίας bits της θύρας εισόδου A0H ενός μΥ συστήματος 8085. Το αποτέλεσμα παρέχεται στο LSB της θύρας εξόδου 20H της μνήμης. Η ακριβής αντιστοιχία των λογικών μεταβλητών με τα bit της θύρας εισόδου είναι:




**Α' Τρόπος**

Εξετάζουμε τις τιμές των εισόδων για να διαπιστώσουμε αν κάνουν αληθή κάποιον από τους δυο όρους της συνάρτησης. Εξετάζουμε δηλαδή αν στο δεδομένο εισόδου υπάρχει ο αριθμός XX1011XX ή ο αριθμός XX100101 όπου X αδιάφορο bit.

Όσον αφορά τις συνθήκες X, πριν απο κάθε σύγκριση φροντίζουμε να μηδενίσουμε τις αντίστοιχες θέσεις με κατάλληλη μασκα.

```
      IN    0A0H        ;Είσοδος
      MOV   B,A
      ANI   00111100B   ;Τοποθέτηση μάσκας
      CPI   00101100B   ;Σύγκριση για τον 1ο όρο της G
      JZ    ADDR1       ;Αν πληρούται έξοδος 1
      MOV   A,B
      ANI   00111111B   ;Μάσκα
      CPI   00100101B   ;Σύγκριση για τον 2ο όρο της G
```

```
      JZ    ADR1          ;Αν πληρούται έξοδος 1
      XRA   A             ;Αλλιώς μηδενίζω την έξοδο
      OUT   20H
      JMP   END
ADR1: MVI   A,01H
      OUT   20H
END:  HLT
```

**Β' Τρόπος**

Η εξομοίωση της λογικής συνάρτησης G μπορεί να γίνει και με τη χρήση ενός πίνακα αναφοράς 64 τιμών ($2^6$=64), που αποτελεί υλοποίηση του πίνακα αληθείας της συνάρτησης G. Οι αριθμοί που δίνουν G=1 είναι:

Πίνακας Αλήθειας της Συνάρτησης G

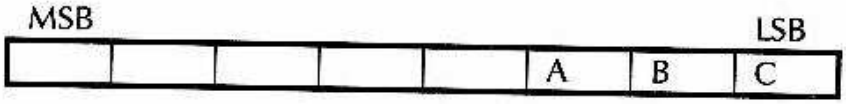
| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0=2CH | 1 |
| 1 | 0 | 1 | 1 | 0 | 1=2DH | 1 |
| 1 | 0 | 1 | 1 | 1 | 0=2EH | 1 |
| 1 | 0 | 1 | 1 | 1 | 1=2FH | 1 |
| 1 | 0 | 0 | 1 | 0 | 1=25H | 1 |
| Για τους υπόλοιπους συνδυασμούς | | | | | | 0 |

```
LOOK_UP MACRO TABLE
      MVI   D,0
      LXI   H,TABLE
      DAD   D
      MOV   A,M
ENDM
ATERMON:
      IN    0A0H
      MOV   E,A
      LOOK_UP   TABLE
      OUT   20H
      JMP   ATERMON
TABLE:
      DB  0,...(σύνολο 37 μηδενικά)..,0
      DB  1,0,0,0,0,0,0,1,1,1,1
      DB  0,...(σύνολο 16 μηδενικά)..,0
      END
```

**Γ' Τρόπος**

Η εξομοίωση μπορεί να γίνει και με τη χρήση των λογικών εντολών του μΕ. Σημειώνεται ότι οι λογικές πράξεις εκτελούνται μεταξύ των ίδιας τάξης δυαδικών ψηφίων. Έτσι απαιτούνται πριν από τις λογικές πράξεις αλισθήσεις για να μεταφερθούν οι μεταβλητές στην ίδια θέση που είναι το

bit εξόδου. Για λόγους απλότητας δίνεται ως παράδειγμα η υλοποίηση της συνάρτησης $F = A\overline{B} + C$. Εδώ η αντιστοιχία των λογικών μεταβλητών με τα bit της θύρας εισόδου A0H είναι:




```
START:
     IN   0A0H
     MOV  D,A            ;Φύλαξη της εισόδου
     ANI  00000001B      ;Τοποθέτηση μάσκας για απομόνωση της C
     MOV  C,A            ;Αποθήκευση της C στον αντίστοιχο καταχ.
     MOV  A,D
     ANI  00000010B      ;Τοποθέτηση μάσκας για απομόνωση της B
     RRC                 ;Ολίσθηση της B στο LSB
     XRI  1              ;Δημιουργία της B'
     MOV  B,A            ;Αποθήκευση της B' στον αντίστοιχο καταχ.
     MOV A,D
     ANI  00000100B      ;Τοποθέτηση μάσκας για απομόνωση της A
     RRC
     RRC                 ;Ολίσθηση της A στο LSB
     ANA  B              ;Δημιουργία του AB'
     ORA  C              ;Δημιουργία του AB'+C
     OUT  20H            ;Εξοδος του αποτελέσματος
     JMP  START
     END
```

***Άσκηση προς λύση:***

Να γραφεί πρόγραμμα σε assembly (για τον 8085) που να διαβάζει 25 αριθμούς (χωρίς πρόσημο) από την πόρτα 10H και να τους αποθηκεύει σε διαδοχικές θέσεις της μνήμης με αρχή τη θέση 300H. Αυτό να γίνεται όταν ο αριθμός είναι μεταξύ 10H και F0H (συμπεριλαμβανομένων). Αλλιώς, στην αντίστοιχη θέση μνήμης θα αποθηκεύεται το μηδέν.

## 5.13 Ρουτίνες Χρονοκαθυστερήσεων

Σε συστήματα αυτοματισμού είναι χρήσιμη η δημιουργία καθυστερήσεων μέσω προγράμματος. Η καθυστέρηση είναι σύμφυτη με την εκτέλεση των προγραμμάτων αφού κάθε εντολή εισάγει και μία καθυστέρηση. Αυτή προσδιορίζεται από τον αριθμό των καταστάσεων που απαιτείται για την εκτέλεση της κάθε εντολής. (βλ. παράρτημα 1). Ο

χρόνος που απαιτείται για κάθε κατάσταση είναι ίσος με την περίοδο του ρολογιού του 8085. Λόγω της μικρής διάρκειας των καθυστερήσεων αυτών (τάξη μεγέθους μsec) για τη δημιουργία καθυστερήσεων που σχετίζονται με φυσικές διαδικασίες απαιτείται η εκτέλεση πολλών εντολών. Αυτή εξασφαλίζεται με τη δημιουργία ανακύκλωσης για την εκτέλεση ενός αριθμού εντολών $2^8=256$ ή $2^{16}=65536$ φορές. Αν δεν αρκεί ο αριθμός αυτός υπάρχει η δυνατότητα ανακύκλωσης μέσα σε ανακύκλωση. Στην περίπτωση αυτή ο αριθμός των ανακυκλώσεων πολλαπλασιάζεται. Στα δύο παραδείγματα που ακολουθούν φαίνονται αναλυτικά οι παραπάνω τεχνικές.

***Παράδειγμα 1ο***

Δίνεται το παρακάτω πρόγραμμα σε assembly (8085 στα 2 Mhz)

```
      MVI   A,DELAY
LOOP:
      DCR   A
      JNZ   LOOP
```

**α.** Ποιός είναι ο ελάχιστος χρόνος καθυστέρησης που μπορεί να επιτευχθεί με το πρόγραμμα αυτό.

**β.** Ποιά είναι η καθυστέρηση σαν συνάρτηση του DELAY; Να βρεθεί η μέγιστη τιμή της. Να υπολογιστεί η τιμή της παραμέτρου DELAY ώστε να έχουμε καθυστέρηση 15 μsec.

**Απαντήσεις**

**α.** Θέτω DELAY=1, τότε ανακύκλωση πραγματοποιείται μια φορά.

```
      MVI   A,DELAY     ;7 κατ. × 500 nsec/κατ.=3.5 μsec
LOOP:
      DCR   A           ;4 κατ. × 500 nsec/κατ. = 2 μsec
      JNZ   LOOP        ;7 κατ. × 500 nsec/κατ. = 3.5  μsec ή 10 κατ.=5 μsec
```

Η εντολή **JNZ** χρειάζεται 7 καταστάσεις όταν δεν εκτελείται άλμα, διαφορετικά απαιτούνται 10 καταστάσεις. Εδώ, η ελάχιστη καθυστέρηση που μπορεί να επιτευχθεί είναι 7+4+7=18 καταστάσεις, δηλαδή 18×0.5 μsec= 9 μsec.

**β.** Η μέγιστη καθυστέρηση παρέχεται για DELAY = 0. Σε αυτή την περίπτωση, μετά την πρώτη μείωση του περιεχομένου του καταχωρητή A, το περιεχόμενο αυτού παίρνει την τιμή FFH, οπότε έως ότου ξαναμηδενιστεί γίνονται συνολικά 256 επαναλήψεις. Στα πρώτα DELAY-1 βήματα έχουμε καθυστέρηση 7+(DELAY-1)= (4+10) καταστάσεις. Στο

τελευταίο βήμα η καθυστέρηση είναι 4+7=11 καταστάσεις. Η συνολική καθυστέρηση δίνεται από τη σχέση T=(14DELAY+4)0.5μsec.
Η μέγιστη καθυστέρηση $T_{max}$=(14×256+4)×0.5μsec=1.794 msec.
Για να έχουμε καθυστέρηση 15 μsec, αρκεί να θέσουμε DELAY=2. Αυτό μπορούμε να το πετύχουμε με την ψευδοεντολή **DELAY EQU 2**.

***Παράδειγμα 2ο***
Δίνεται το παρακάτω πρόγραμμα σε assembly που δημιουργεί καθυστέρηση:

```
     MVI  D,M (8 bit)     ;7 καταστάσεις
DELAY:
     LXI  B,N (16 bit)    ;10 καταστάσεις

;********* Εσωτερική ανακύκλωση ******
LOOP:
     DCX  B               ;6 καταστάσεις
     MOV  A,B             ;4 καταστάσεις
     ORA  C               ;4 καταστάσεις
     JNZ  LOOP            ;7 ή 10 καταστάσεις

;*********************************
     DCR  D               ;4 καταστάσεις
     JNZ  DELAY           ;7 ή 10 καταστάσεις
     RET                  ;10 καταστάσεις
```

Εστω ότι το πρόγραμμα εκτελείται σε ένα μΥ 8085 που λειτουργεί με ρολόι 4 MHz. Να υπολογιστεί η καθυστέρηση που δημιουργεί το παραπάνω πρόγραμμα σαν συνάρτηση των M και N.

Το πρόγραμμα αποτελείται από δύο βρόγχους τον ένα μέσα στον άλλον. Ο εσωτερικός βρόγχος εκτελείται N-1 φορές εκτελώντας το άλμα και δημιουργεί καθυστέρηση ίση με:

(N-1)(6+4+4+10)=24(N-1) καταστάσεις

Στο τελευταίο πέρασμα δεν εκτελείται το άλμα και έχουμε καθυστέρηση ίση με 6+4+4+7=21 καταστάσεις. Η συνολική καθυστέρηση του εσωτερικού βρόγχου είναι 24N-3 καταστάσεις. Ο εξωτερικός βρόγχος με τις επιπλέον εντολές που έχει δημιουργεί καθυστέρηση για τα πρώτα M-1 βήματα (M-1)(10+4+10+24N-3)=(M-1)(24N+21) καταστάσεις. Στο τελευταίο βήμα έχουμε καθυστέρηση 10+4+7+24N-3+10=24N+28.

Επομένως η συνολική καθυστέρηση που δημιουργείται είναι 24MN+21M+14 καταστάσεις. Αν τώρα το ρολόι του συστήματος λειτουργεί στα 4MHz, η κάθε κατάσταση έχει διάρκεια 0.25 μsec, οπότε η καθυστέρηση είναι Delay(M,N)=(24MN+21M+14) 0.25 μsec.

### *Παράδειγμα 3ο: Υλοποίηση χρονομέτρου*

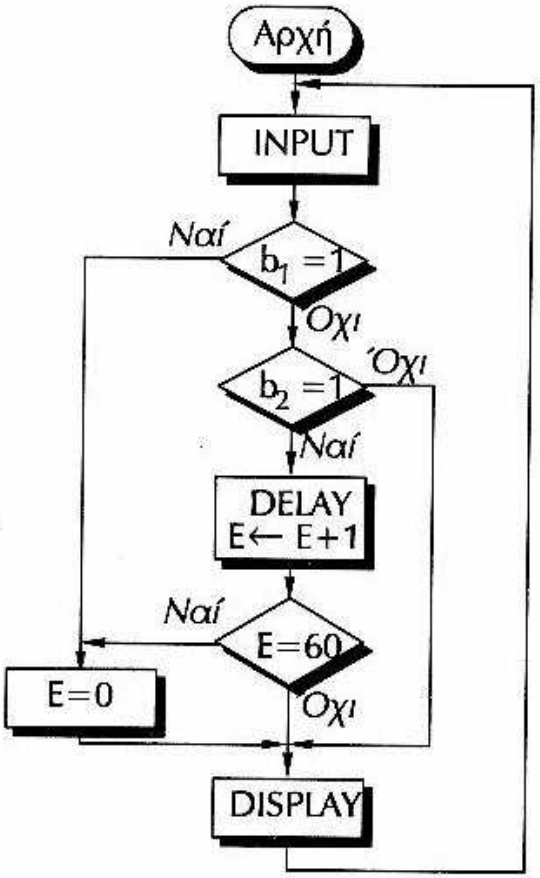
Υλοποιείται χρονόμετρο δευτερολέπτων σε μΥ-Σ 8085, το οποίο διαθέτει πόρτα εισόδου (διεύθυνση 10Η) και πόρτα εξόδου (διεύθυνση 20Η). Το χρονόμετρο μετράει μέχρι το 59 με τον παρακάτω τρόπο:
α) όταν το δεύτερο bit (b2) της εισόδου γίνει 1 να ξεκινά η μέτρηση, ενώ όταν γίνει 0 να "παγώνει" και το χρονόμετρο να συνεχίζει τη μέτρηση όταν το $b_2$ ξαναγίνει 1.
β) Το πρώτο bit ($b_1$) της εισόδου χρησιμοποιείται ως Reset (δηλ. όταν ενεργοποιείται το χρονόμετρο μηδενίζεται). Για να εκτελεστεί η λειτουργία που περιγράφηκε στο α) θα πρέπει το bit αυτό ($b_1$) να είναι 0.

Στην πόρτα εξόδου θεωρούμε συνδεδεμένους δύο μετατροπείς BCD - 7 τμήματα και δύο ενδείκτες 7 τμημάτων για την απεικόνιση της μέτρησης. Το διάγραμμα ροής είναι στο σχήμα 5.8. Το αντίστοιχο πρόγραμμα είναι το ακόλουθο:

```
     MVI  E,00        ;Αρχικοποίηση του μετρητή
INPUT:
     IN   10H         ;Είσοδος από την πόρτα 10Η
     RRC              ;Το bit 1(b1) στο κρατούμενο
     JC   RESET       ;Αν b1=1 το χρονόμετρο γίνεται RESET

     RRC              ;Το bit 2(b2) στο κρατούμενο
     JNC  DISPLAY     ;Αν είναι 1 το χρονόμετρο παγώνει (Παρακάμπτεται η
                      ; αύξηση του μετρητή και εκτελείται μόνο απεικόνιση)
     CALL DELAY       ;Καθυστέρηση 1 sec
     MOV  A,E         ;Μεταφορά του μετρητή στον καταχωρητή Α
     INR  A           ;και αύξηση της ένδειξης
     DAA              ;Προσαρμογή της ένδειξης σε BCD μορφή
     MOV  E,A         ;Ενημέρωση του καταχωρητή Ε
     CPI  60H         ;Εφτασε το χρονόμετρο στη τιμή 60; (Προσοχή 60Hex !)
     JNZ  DISPLAY     ;Αν όχι συνεχίζεται η χρονομέτρηση. Αλλιώς

RESET:
     MOV  E,0         ;μηδενίζεται το χρονόμετρο.
DISPLAY:              ; Για απεικόνιση
     MOV  A,E         ; μεταφορά του μετρητή στο συσσωρευτή Α
     OUT  20H         ;Εμφάνιση της μέτρησης στους ενδείκτες
     JMP  INPUT
     END
```




**Σχήμα 5.8** Διάγραμμα ροής χρονομέτρου

Η ρουτίνα **DELAY** που εξασφαλίζει καθυστέρηση 1 sec είναι η εξής:

```
                        ; Αριθμός  καταστάσεων
DELAY:
      LXI   H,0000      ; 10 καταστάσεις
      MVI   D,02H       ; 7 καταστάσεις
LOOP:
      DCR   L           ; 4 καταστάσεις
      l × NOPs          ; l × 4. Υπονοείται ότι σε αυτό το σημείο
                        ;του προγράμματος θα εισαχθούν l εντολές NOP
      JNZ   LOOP        ; 7/10
      DCR   H           ; 4 καταστάσεις
      m × NOPs          ; m × 4, m το πλήθος εντολές NOP
      JNZ   LOOP        ; 7/10, c το πλήθος εντολές NOP
      DCR   D           ; 4 καταστάσεις
      n × NOPs          ; n × 4
      JNZ   LOOP        ; 7/10 καταστάσεις
      RET               ; 10 καταστάσεις
```

Από τους βρόγχους που έχουμε ο εξωτερικός εκτελείται 2 φορές, ο μεσαίος 256 φορές και ο εσωτερικός επίσης 256 φορές. Ο αριθμός

καταστάσεων του εσωτερικού βρόγχου είναι $K_1=256(4+4l+10)-3$. Ο μεσαίος βρόγχος περιλαμβάνει $K_2=256(K_1+4+4m+10)-3$ το πλήθος καταστάσεις. Τέλος ο εξωτερικός βρόγχος περιλαμβάνει $K_3=2(K_2+4+4n+10)-3$ καταστάσεις. Οπότε ο συνολικός αριθμός είναι:

$$K = 10+7+K_3+10 = 1.840.686 +524.288l+2048m+8n.$$

Για $l=2$, $m=54$ και $n=17$ έχουμε $K=2.999.990$. Με την τοποθέτηση ακόμα μιας εντολής **LXI** πριν από την εντολή **RET** και υποθέτοντας ότι η συχνότητα του ρολογιού είναι 3MHz, η ρουτίνα DELAY θα προκαλεί καθυστέρηση ακριβώς 1 sec.