HY255 Εργαστήριο Λογισμικού – L16

x86

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2022
Άγγελος Μπίλας

C source - Assembly - Machine code

- Στόχος: Να μπορούμε να καταλάβουμε ποια είναι η μορφή του προγράμματος μας σε γλώσσα assembly/machine code όταν εκτελείται;

```
gcc –S main.c → main.s (assembly)
gcc main.c → a.out (machine code)
```

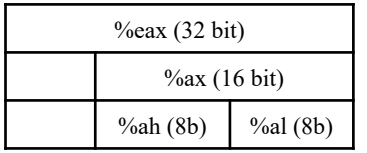
- Η γλώσσες assembly και machine code θεωρούμε ότι είναι πολύ κοντά η μία στην άλλη με σημαντικότερη διαφορά την αναπαράσταση
  - Assembly: χαρακτήρες (symbolic)
  - Machine code: 0,1 (binary)
- Η μετάφραση από την μια στην άλλη είναι σχεδόν 1-1 για κάθε εντολή και μπορεί να γίνει και προς τις δύο κατευθύνσεις
  - Assemble: μετατροπή ενός προγράμματος από machine code σε assembly
  - Disassemble: το αντίστροφο
- Η assembly γλώσσα καθορίζεται από τον επεξεργαστή
  - Σε αντίθεση με τις γλώσσες που μεταφράζονται σε assembly (C, Java, etc) που είναι ίδιες για όλους τους επεξεργαστές

x86

- x86 είναι η 32-bit αρχιτεκτονική (το interface ανάμεσα στον επεξεργαστή και στα προγράμματα, που πιο σωστά το ονομάζουμε Instruction Set Architecture ή ISA) των επεξεργαστών Intel/AMD
  - x86_64 είναι η 64-bit αρχιτεκτονική
  - Εμείς θα εξετάσουμε την 32-bit γιατί μας βολεύει καλύτερα για τους σκοπούς των ασκήσεων 5,6
- Τα βασικά στοιχεία της ISA ενός επεξεργαστή είναι
  - (1) Τα ονόματα των εντολών και καταχωρητών
  - (2) Οι τρόποι προσπέλασης (δεικτιοδότησης) της μνήμης
  - (3) Η μορφή της στοίβας
  - (4) Ορισμένα άλλα στοιχεία που αφορούν στο λειτουργικό σύστημα και δεν απασχολούν εν γένη τα προγράμματα

Εντολές, καταχωρητές x86

- Οι επεξεργαστές της οικογένειας αυτής διαθέτουν πάρα πολλές εντολές και παραλλαγές τους. Ωστόσο για να γράψουμε τα πιο πολλά προγράμματα δεν χρειαζόμαστε πολλές εντολές
  - mov: αντιγράφει μια ποσότητα σε μια νέα θέση, π.χ. μια σταθερή τιμή σε μια θέση μνήμης
  - add, και άλλες αριθμητικές πράξεις
  - call/ret: καλεί/επιστρέφει από μια συνάρτηση
  - jmp: μεταφέρει τη ροή του προγράμματος σε άλλο σημείο
  - cmp: συγκρίνει δύο τιμές
  - je, jg, jl, etc: μεταφέρουν τη ροή με βάση κάποια συνθήκη (conditional jump)
  - lea (load effective address): μια «βοηθητική» εντολή που υπολογίζει μια διεύθυνση (χωρίς να διαβάζει/γράφει τα περιεχόμενα)
- Κάθε εντολή δέχεται παραμέτρους που γενικά μπορεί να είναι
  - Ένας αριθμός (constant): συμβολίζεται με το $, π.χ. $1, $1000
  - Μια θέση μνήμης: συμβολίζεται με έναν αριθμό χωρίς $, π.χ. 1000 είναι η θέση μνήμης 1000 ($1000 είναι ο αριθμός 1000)
    Επίσης, συμβολίζεται με έναν καταχωρητή μέσα σε παρενθέσεις, που σημαίνει την διεύθυνση στην οποία δείχνει ο καταχωρητής (indirect addressing)
  - Ένας καταχωρητής
- Οι καταχωρητές είναι λίγες θέσεις μνήμης που βρίσκονται μέσα στον επεξεργαστή και τις χρησιμοποιούμε βοηθητικά για την εκτέλεση των διάφορων εντολών
  - Στον x86, μια ιδιαιτερότητα είναι ότι πολλές εντολές μπορούν να χρησιμοποιούν απ'ευθείας θέσης μνήμης και επομένως η χρήση των καταχωρητών σαν παράμετροι εντολών είναι πιο περιορισμένη από ότι σε άλλες οικογένειες επεξεργαστών
- Οι x86 έχουν (ιστορικά) λίγους καταχωρητές ως μέρος της ISA (και περισσότερους physical που όμως δεν φαίνονται στο πρόγραμμα, δεν είναι μέρος του interface)
  - Γενικού σκοπού: %eax (%ax,%ah,%al), %ebx (%bx,%bh,%bl), %ecx (%cx,%ch,%cl), %edx (%dx,%dh,%dl), %edi, %esi
    Οι καταχωρητές αυτοί είναι προσπελάσιμοι και σε 16 ή 8 bit ποσότητες με τα ονόματα στις παρενθέσεις. Π.χ. για τον %eax:
  - Ειδικού σκοπού: %ebp, %esp: για την χρήση της στοίβας
    %eip: intruction pointer
    %eflags: περιέχει flags (π.χ. carry) για την υλοποίηση των π.χ. conditional branches



Σύνταξη

- Το destination argument στις εντολές είναι το τελευταίο (ακολουθεί το source), π.χ.
  - movl %eax, %ebx → αντιγράφει τα περιεχόμενα του %eax στον %ebx
  - Διαφορετικοί assemblers (π.χ. AT&T vs. Intel) έχουν διαφορετικές συμβάσεις για διάφορα θέματα σύνταξης, εμείς χρησιμοποιούμε τον GNU assembler που ακολουθεί γενικά τον AT&T
- Οι εντολές μπορεί να αναφέρονται σε ποσότητες 8 (b - byte), 16 (s - short), ή 32 (l - long) bits
  - movb, movs, movl
  - addb, adds, addl
  - Οι πιο πολλές εντολές που θα χρειαστούμε εμείς αναφέρονται σε ποσότητες 32 bits / 4 bytes

Παραδείγματα

```
movl $0, %eax                # eax ← 0
movl 0, %eax                 # eax ← MEM[0]
movl %ebx, %eax              # eax ← ebx
movl (%ebx), %eax            # eax ← MEM[ebx]
movl 10(%ebx), %eax          # eax ← MEM[ebx+10]
leal  10(%eax), %ebx         # ebx ← eax + 10
addl %eax, %ebx              # ebx ← eax + ebx
cmpl %eax, %ebx              # σύγκρινε το 2ο op με το 1ο και κράτα στα
                             # %eflags το αποτέλεσμα της σύγκρισης
je label                     # μετέφερε την ροή στο label αν το αποτέλεσμα της
                             # προηγούμενης σύγκρισης (cmp) ήταν "="
jmp label                    # μετέφερε την ροή στο label (unconditional)
jmp %eax                     # μετέφερε την ροή στη διεύθυνση που περιέχει ο eax
```

Παράδειγμα: Μέγιστος δύο αριθμών

max.c

```
int a = 10;
int b = 12;
int main(void){
    int t;
    if (a<=b)
        t = b;
    else
        t = a;
    printf("max is %d\n", t);
    return 0;
}
```

max.s

```
.section .data
a: .long 10
b: .long 12
msg: .ascii "max is %d\n\0"
.globl main
main:
    # t in %eax
    movl a, %eax
    movl b, %ebx
    cmpl %eax, %ebx      # (%ebx - %eax)
    jgt Lb>a
    movl %eax, %ecx      # t = a
    jmp Lprint
Lb>a
    mov %ebx, %ecx       # t = b
Lprint
    pushl %ecx
    pushl msg
    call printf
    popl %ecx
    popl %ecx
    movl $0, %eax
    ret
```

Assembly in max.s

gcc –Wall –pedantic –ansi –S –O2 max.c

Το –O2 παράγει κώδικα assembly πιο κοντά σε αυτό που θα γράφαμε εμείς (σε σχέση με –O0,1

Executable a.out with gdb info

$gcc –Wall –pedantic –ansi –g max.c

Execute

$./a.out

In gdb

- gdb a.out
- gdb> b main
- gdb> run
- gdb> x/9i main [assembly instructions]

```
0x80483fb <main>:     lea    0x4(%esp),%ecx
0x80483ff <main+4>:   and    $0xfffffff0,%esp
0x8048402 <main+7>:   pushl  -0x4(%ecx)
0x8048405 <main+10>:  push   %ebp
0x8048406 <main+11>:  mov    %esp,%ebp
0x8048408 <main+13>:  push   %ecx
0x8048409 <main+14>:  sub    $0x14,%esp
0x804840c <main+17>:  mov    0x80496ec,%edx
0x8048412 <main+23>:  mov    0x80496f0,%eax
```

- gdb> print a
  - a = 10
- gdb> set a = 100
- gdb> print a
  - a = 100
- gdb> set *main = 0xabcdabcd ← ?
- gdb> x/10i main
  - ?

gdb> x/9x main [machine code]

```
0x80483fb <main>:    0x04244c8d
                     0xfff0e483
                     0x8955fc71
                     0xec8351e5
0x804840b <main+16>:0xec158b14
                     0xa1080496
                     0x080496f0
                     0x0a7fc239
0x804841b <main+32>:0x0496f0a1
```

Reading

- Chapter 3 and Appendix B, Programming from the Ground Up, Jonathan Bartlett 2004
- Δεν θα σας χρειαστεί, ωστόσο, όλη η λεπτομέρεια για x86-32 υπάρχει στο Intel® 64 and IA-32 Architectures Software Developer Manuals: https://cdrdv2.intel.com/v1/dl/getContent/671200 (combined 4 volumes)