CSD1100

Assembler - Basics

Vadim Surov

Introduction

- Most programming nowadays is done in high-level languages such as C/C++ or JavaScript.
- Such languages deliberately hide from a programmer many details concerning HOW his problem actually will be solved by the computer, so
  - Programmers focus on the problem to be solved.
  - Programmers don't have to know very much about how different computers actually works.

Introduction

- Going to lowest level, we must use the computer's own language, called **machine language**.
- This language is specific to the machine's architecture.
- Here's how one instruction for Intel i7 processor:

  A1BC9304080305C0930408A3C0940408
- It means: set register RAX with value 1075. (could be)

Introduction

- It is extremely difficult, tedious (and error-prone) for humans to read and write any code on such low-level machine language.
- So **human readable machine language**, called assembly language or assembler, was invented.
- There are two key ideas behind assembler:
    - **mnemonic opcodes**: employ abbreviations of English language words to denote operations.
    - **symbolic addresses**: invent “meaningful” names for memory storage locations we need.

Introduction

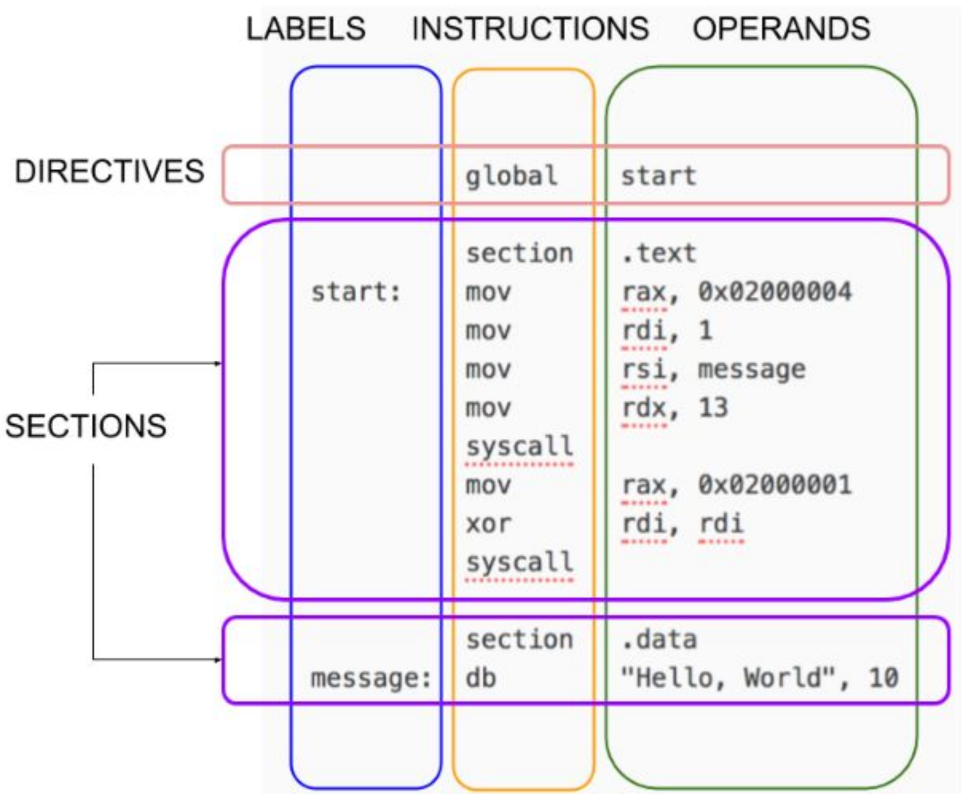
- Advantages of using assembler:
  - **Greater control**. It gives you greater control over how certain functions are implemented at the assembly language level.
  - **Optimization**. Programmers can use assembly language to implement the most performance-sensitive parts of their program's algorithms.
  - **Accessibility**. Access to processor specific instructions.

Tools

- The **Netwide Assembler** (NASM)
  - Is an assembler for the Intel x86 architecture.
  - It can be used to write 16-bit, 32-bit and 64-bit (x86-64) programs.
  - It can output several binary formats, including Executable and Linkable Format (ELF) that we are going to use in labs with compilation option -f elf64.
  - NASM is considered to be one of the most popular assemblers for Linux.

Program structure

- ASCII-character text
- 2 segments: .data and .text
- Program consists of series of ‘statements’
- Each program-statement fits on one line
- Program-statements all have same layout
- Design in 1950s was for IBM punch-cards



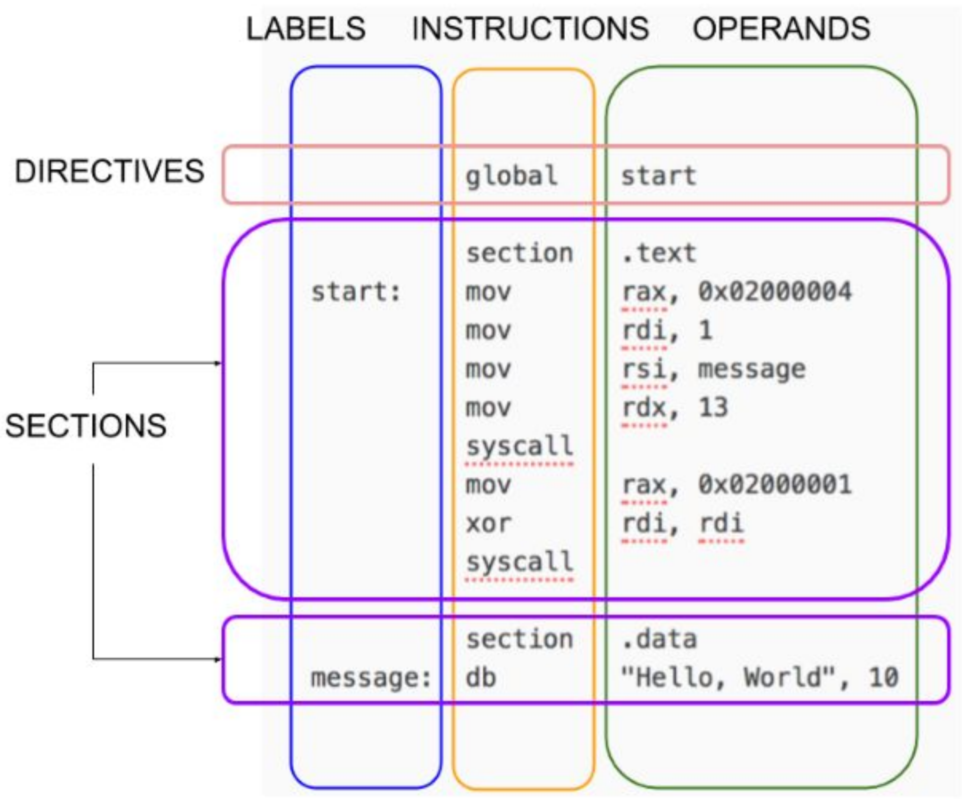
Program structure

- Each ‘statement’ was comprised of four ‘fields’
- Fields appear in a prescribed left-to-right order
- These four fields were named (in order):
  - the ‘label’ field
  - the ‘opcode’ field
  - the ‘operand’ field
  - the ‘comment’ field
- In many cases some fields could be left blank
- Extreme case (very useful): whole line is blank!

```
start:      mov         rax, 0x02000004
            mov         rdi, 1
            mov         rsi, message
```

Segments

- The .data segment
  - contains any global or static variables which have a pre-defined value and can be modified
- The .text segment
  - contains executable instructions



Label

- A label is a ‘symbol’ followed by a colon (‘:’)
- The programmer invents his own ‘symbols’
- Symbols can use letters and digits, plus a very small
- number of ‘special’ characters (‘.’, ‘_’, ‘$’)
  - A ‘symbol’ is allowed to be of arbitrarily length
- Ex: start:

```
start:    mov    rax, 0x02000004
          mov    rdi, 1
          mov    rsi, message
```

Opcode

- Opcodes are predefined symbols that are recognized by the assembler.
- There are two categories of opcodes: Directives and Instructions.
- **Directives** are commands that guide the work of the assembler (e.g., ‘global’)

Opcode

- **Instructions** represent operations that the CPU is able to perform (e.g., ‘add’, ‘inc’).
- Each instruction gets translated by compiler into a machine-language statement that will be fetched and executed by the CPU when the program runs.

```
start:    mov     rax, 0x02000004
          mov     rdi, 1
          mov     rsi, message
          mov     rdx, 13
          syscall
          mov     rax, 0x02000001
          xor     rdi, rdi
          syscall
```

Opcode

- An official list of the instruction codes can be found in NASM manuals, appendix B:
  - https://www.nasm.us/doc/nasmdocb.html
- You can Google to find an ‘unofficial’ short list of most common instructions. It's allowed to print it on one sheet and use on exams as a cheat sheet.

Opcode

- CPU Instructions usually operate on data-items.
- Only certain sizes of data are supported:
  - **byte**: one byte consists of 8 bits
  - **word**: consists of two bytes (16 bits)
  - **long**: uses four bytes (32 bits)
  - **quadword**: uses eight bytes (64 bits)
- With **Intel syntax** (NASM), data-size usually isn't explicit, but is inferred by context (i.e., from operands).
- With **AT&T's syntax** (GCC), an instruction's name also incorporates its effective data-size (as a suffix b/w/l/q).

Operand

- Operands can be of several types:
  - a CPU register may hold the datum,
  - a memory location may hold the datum.
- An instruction can have ‘built-in’ data.
- Frequently there are multiple data-items.
- Sometimes there are no data-items.
- An instruction’s operands usually are ‘explicit’, but in a few cases they also could be ‘implicit’.

Operand

- Some instruction have two operands:
  - mov rbx, 0
  - add rsp, 4
- Some instructions have one operand:
  - inc rax
  - push rax
- The instructions that have no operands:
  - ret
  - syscall

Registers

General Purpose Registers:

Symbol	Meaning	Use
RAX	Accumulator	Arithemtic Operations
RBX	Base	Pointer to Data
RCX	Counter	Shift/Rotate Instructions or Loops
RDX	Data	Arithemtic or I/O Operations
RSI	Source Index	Pointer to Source in Stream Operations
RDI	Destination Index	Pointer to Destination in Stream Operations
RSP	Stack Pointer	Pointer to the Top of the Stack
RBP	Stack Base Pointer	Point to the Base of the Stack

Registers

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Link

Registers

Special Registers:

Name	Meaning	Description
rip (eip, ip)	Instruction Pointer	Points to next address to be executed in control flow
rsp (esp, sp)	Stack Pointer	Points to the top address of the stack
rbp (ebp, bp)	Stack Base Pointer	Points to the bottom of the stack

Comments

- An assembly language code often can be hard for a human being to understand.
- Even a program's author may not be able to recall his programming idea after a while.
- So programmer comments can be vital.
- Comments begin with the '#' character in the beginning of line.
- A single semi-colon ; is used for comments from any position, and works the same as double slash // in C++: the compiler ignores from the semicolon to the next newline.

Constants

```
mov     ax, 200         ; decimal
mov     ax, 0200        ; still decimal
mov     ax, 0200d       ; explicitly decimal
mov     ax, 0d200       ; also decimal
mov     ax, 0c8h        ; hex
mov     ax, $0c8        ; hex again: the 0 is required
mov     ax, 0xc8        ; hex yet again
mov     ax, 0hc8        ; still hex
```

Floating point data types are out of our scope

Constants

```
mov     ax, 310q          ; octal
mov     ax, 310o          ; octal again
mov     ax, 0o310         ; octal yet again
mov     ax, 0q310         ; octal yet again
mov     ax, 11001000b     ; binary
mov     ax, 1100_1000b    ; same binary constant
mov     ax, 1100_1000y    ; same binary constant once more
mov     ax, 0b1100_1000   ; same binary constant yet again
mov     ax, 0y1100_1000   ; same binary constant yet again
```

Defining Data

```
db    0x55                  ; just the byte 0x55
db    0x55,0x56,0x57        ; three bytes in succession
db    'a',0x55              ; character constants are OK
db    'hello',13,10,'$'     ; so are string constants
dw    0x1234                ; 0x34 0x12
dw    'a'                   ; 0x61 0x00 (it's just a number)
dw    'ab'                  ; 0x61 0x62 (character constant)
dw    'abc'                 ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678            ; 0x78 0x56 0x34 0x12
dd    1.234567e20           ; floating-point constant
dq    0x123456789abcdef0  ; eight byte constant
```

Syscall instruction

- Use **syscall** instruction to take advantage of the operating system to perform actions.
- Using syscall avoids having to recreate the functions from scratch to interact with system components like:
  - Display data on the screen
  - Write on disk
  - Etc.

- Use next example to output result of calculations in assignments and for output temp values during debugging.

```
section .data
msg db "Hello world!", 10    ; 10 is the ASCII code for a new line (LF)

section .text
    global _start

_start:
    mov rax, 1      ; syscall number for write
    mov rdi, 1      ; 1st argument is a file descriptor (1 for stdout)
    mov rsi, msg    ; 2nd argument is a pointer to a string
    mov rdx, 13     ; 3rd argument is the number of characters to display
    syscall

    mov rax, 60     ; syscall number for exit
    mov rdi, 0      ; int status
    syscall
```

Output using C's standard library

- Compile and link previous code to see the output:

```
$ nasm -f elf64 output_syscall.asm
$ ld output_syscall.o -o output_syscall
```

- Next code produces the same result bu using C's standard library. Compile it and link with the library.

```
$ nasm -f elf64 output_puts.asm
$ ld -dynamic-linker
/lib64/ld-linux-x86-64.so.2 -lc output_puts.o
-o output_puts
```

```
section .data
msg db "Hello world!",10         ; 10 is the ASCII code for a new
                                  ; line (LF)

section .text
    global _start
    extern puts

_start:
    mov rdi, msg    ; 1st argument is a pointer to a string
    call puts

    mov rax, 60     ; syscall number for exit
    mov rdi, 0      ; int status
    syscall
```

Move

- The **mov** instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory).
- While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

```
section .data
format db 'rsi=%ld data=%ld',10,0
data    dq 5678

section .text
    global _start
    extern printf

_start:
    mov rdi, format     ; 1st argument (by convention)
    mov rsi, 1234       ; 2nd argument
    mov rdx, [data]     ; 3rd argument (value by address)
    xor rax, rax        ; required (no xmm registers used)
    call printf

    mov rax, 60     ; syscall number for exit
    mov rdi, 0      ; int status
    syscall
```

Add

- The **add** instruction adds together its two operands, storing the result in its first operand.
- Second operand can be register, memory or constant.
- First operand is a register or memory.

ADD RAX, *imm32*	Add *imm32 sign-extended to 64-bits* to RAX.
ADD *r/m64, imm32*	Add *imm32 sign-extended to 64-bits* to *r/m64*.
ADD *r/m64, r64*	Add r64 to *r/m64*.
ADD *r64, r/m64*	Add *r/m64* to *r64*.

```
section .data
format db 'x+y=%ld',10,0
x    dq 10
y    dq 42

section .text
    global _start
    extern printf

_start:
    mov rdi, format
    mov rsi, [x]
    add rsi, [y]
    xor rax, rax
    call printf

    mov rax, 60     ; syscall number for exit
    mov rdi, 0      ; int status
    syscall
```

imul — Signed Multiply

Instruction	Description
IMUL *r/m8**	AX← AL ∗ *r/m* byte.
IMUL *r/m16*	DX:AX ← AX ∗ *r/m* word.
IMUL *r/m32*	EDX:EAX ← EAX ∗ *r/m*32.
IMUL *r/m64*	RDX:RAX ← RAX ∗ *r/m*64.
IMUL *r16, r/m16*	word register ← word register ∗ *r/m*16.
IMUL *r32, r/m32*	doubleword register ← doubleword register ∗ *r/m32*.
IMUL *r64, r/m64*	Quadword register ← Quadword register ∗ *r/m64*.
IMUL *r16, r/m16, imm8*	word register ← *r/m16* ∗ sign-extended immediate byte.
IMUL *r32, r/m32, imm8*	doubleword register ← *r/m32* ∗ sign-extended immediate byte.
IMUL *r64, r/m64, imm8*	Quadword register ← *r/m64* ∗ sign-extended immediate byte.
IMUL *r16, r/m16, imm16*	word register ← *r/m16* ∗ immediate word.
IMUL *r32, r/m32, imm32*	doubleword register ← *r/m32* ∗ immediate doubleword.
IMUL *r64, r/m64, imm32*	Quadword register ← *r/m64* ∗ immediate doubleword.

```
section .data
format db 'x*y=%ld',10,0
x    dq 10
y    dq 42

section .text
    global _start
    extern printf

_start:
    mov rdi, format
    mov rsi, [x]
    imul rsi, [y]
    xor rax, rax
    call printf

    mov rax, 60     ; syscall number for exit
    mov rdi, 0      ; int status
    syscall
```

For Lab 10. Passing parameters

- When writing code for 64-bit Linux that integrates with a C library, you must follow the calling conventions (more Wikipedia).
- The most important points are:
- From left to right, pass as many parameters as will fit in registers. The order in which registers are allocated, are:
  - For integers and pointers, **rdi, rsi, rdx, rcx, r8, r9**.
  - For floating-point (float, double), xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7.

For Lab 10. Passing parameters

- Note that 3rd parameter in **rdx** is also used in **imul** and **idiv**
- To save/restore it around execution of imul or idiv use **push/pop** the rdx into/from a stack (dynamically allocated memory):

```
push rdx      ; save temporally on stack
mov rdx, 0
mov rax, 10
imul 5        ; rax = 10*5
pop rdx       ; restore from stack
```

References

1. NASM documentation
   https://www.nasm.us/doc/
2. An official list of the instruction codes can be found in appendix B:
   https://www.nasm.us/doc/nasmdocb.html