x86 ASSEMBLY LANGUAGE SYNTAX AND PROGRAM STRUCTURE

CS 348
Implementation of Programming Languages Lab
Computer Science and Engineering Department
Indian Institute of Technology
Guwahati

Contents

- Assembly Statements
- Instructions & Operands
- Data Representation
- Assembler Directives
- Macros

Assembly Statements

- Assembly program is composed of statements with the general form:

[label:]   mnemonic   [operands]   [;comment]

- **Label**
  Marks an address in the program; used for jumps and control flow.
- **Mnemonic**

  Specifies the operation to be performed,

  Maps to the ISA specific machine instruction encoding (opcode + format)
- **Operands**

  Data used by the instruction (registers, memory, or constants).
- **Comment**

  Ignored by the assembler.

Assembly Statements

Three types of statements in assembly language are:

- **Executable Instructions**
  - Generate machine code for the processor to execute on runtime
  - Instructions tell the processor what to do.
- **Assembler Directives**
  - Provide information to the assembler while translating a program
  - Used to define data, select memory model, etc.
  - Non executable: **directives are not part of instruction set**.
- **Macros**
  - Shorthand notations for a group of statements.
  - Sequence of instructions, directives, or other macros.
  - Expand into instructions and/or directives before assembly

Instructions processed by CPU, directives and macros processed by assembler.

Instructions

- Assembly language instructions have the format:

[label:]   mnemonic   [operands]

- Instruction Label (optional)
  - Marks the address of an instruction, must have a colon :
  - Used to transfer program execution to a labelled instruction (jump or branch targets) .
- Mnemonic
  - Identifies the operation specific to ISA (e.g. MOV, ADD, SUB, JMP, CALL)
- Operands
  - Specify the data required by the operation
  - Executable instructions can have zero to three operands
  - Operands can be registers, memory variables, or constants

Instructions

Types of Operands

- Register operands – CPU registers (e.g., eax, ebx)
- Memory operands – variables stored in memory
- Immediate operands – constant values encoded in the instruction

- **The number and type of operands are determined by the instruction and the ISA.**

Instruction Examples

- No operand

```
stc                ; set carry flag
```

- One operand

```
inc eax            ; increment register eax
call clrscr        ; call procedure clrscr
jmp L1             ; jump to instruction with label L1
```

- Two operand

```
add ebx, ecx       ; register ebx = ebx + ecx
sub var1, 25       ; memory variable var1 = var1 - 25
```

- Three operand

```
imul eax, ebx, 5   ; register eax = ebx *5
```

Numeric constants

```
mov     eax,200         ; decimal
mov     eax,0200        ; decimal
mov     eax,0200d       ; explicitly decimal
mov     eax,0d200       ; decimal

mov     eax,0c8h        ; hex
mov     eax,$0c8        ; hex again: the 0 is required
mov     eax,0xc8        ; hex
mov     eax,0hc8        ; hex
```

Numeric constants

```
mov     eax,310q          ; octal
mov     eax,310o          ; octal
mov     eax,0o310         ; octal
mov     eax,0q310         ; octal

mov     eax,11001000b     ; binary
mov     eax,1100_1000b    ; binary
mov     eeax,1100_1000y   ; binary
mov     aex,0b1100_1000   ; binary
mov     eax,0y1100_1000   ; binary
```

Character and String Constants

- A character constant consists of a string up to eight bytes long.
- A character constant with more than one byte will be arranged with little-endian order.

mov eax, 'abcd'

- String Constants are used with data-definition directives such as db
- A string constant is treated as a concatenation of maximum size character constants.

db 'hello' ; string constant

db 'h','e','l','l','o' ;equivaent character constant

Floating Point Constants

- Floating-point constants can be used as operands with data definition directives:

db, dw, dd, dq, dt, do

- These constants are encoded into memory according to the specified data size
- Floating points are expressed as digits, followed by a decimal, then one more digit, and e followed by an exponent.

digits . digits e ± exponent

dq 1.23e4

Floating Point Constants

- NASM provides special operators to control floating-point format
- The special operators are used to produce floating point numbers in other contexts.

```
__float8__ , __float16__ , __float32__ , __float64__
```

```
dd __float32__(3.14)   ; store 3.14 in 32-bit floating-point format
dq __float64__(3.14)   ; store 3.14 in 64-bit floating-point format
```

- NASM cannot perform compile time arithmetic on floating point constants.
- Only constant values can be encoded; arithmetic must occur at runtime

Examples

```
db      -0.2                    ; "Quarter precision"
dw      -0.5                    ; IEEE 754r/SSE5 half precision
dd      1.2                             ;
dd      1.222_222_222                   ; underscores are permitted
dd      0x1p+2                          ; 1.0x2^2 = 4.0
dq      0x1p+32                         ; 1.0x2^32 = 4 294 967 296.0
dq      1.e10                           ; 10 000 000 000.0
dq      1.e+10                          ; synonymous with 1.e10
dq      1.e-10                          ; 0.000 000 000 1
dt      3.141592653589793238462 ; pi
do      1.e+4000                        ; IEEE 754r quad precision
```

Expressions

- Expressions follow syntax **similar to C language**
- Used in constants, directives, and data definitions
- Bitwise OR Operator: The | operator gives a bitwise OR.
- Bitwise XOR Operator: ^ provides the bitwise XOR operation.
- Bitwise AND Operator: & provides the bitwise AND operation.

```
mov eax, 0x0F | 0x30     ; eax = 0x3F
mov eax, 0x0F & 0x03     ; eax = 0x03
mov eax, 0x0F ^ 0x03     ; eax = 0x0C
```

Expressions

- Bit Shift Operators: **Left shift (<<)** and **Right shift (>>)** are bitwise shift operators.

```
mov eax, 1 << 4        ; eax = 16
mov eax, 32 >> 2       ; eax = 8
```

- The operators for **Add, Substract, Multiply, Divide** and **Modulo** are same as C. Signed division operator is // and signed modulo operator is %%.

```
mov eax, 10 + 5        ; eax = 15
mov eax, 20 // 3       ; eax = 6   (signed division)
mov eax, 20 %% 3       ; eax = 2   (signed modulo)
```

Assembler Directives

- Special statements processed by the assembler
- Not translated into machine instructions
- Used to Control code generation, memory layout and in symbol handling

- Broad Categories
  - Mode & CPU control – target architecture and instruction set
  - Symbol visibility – interaction with linker and other modules
  - Section & address control – placement of code and data
  - Diagnostics & numeric behavior – warnings and floating-point handling

Mode & CPU control

BITS

- Specifies the processor operating mode
- Determines instruction encoding and operand sizes

BITS 16 | 32 | 64

CPU

- Restricts assembly to instructions supported by a specific CPU
- Prevents use of unsupported instructions

CPU x86-64

DEFAULT

- Changes assembler default behaviors
- Commonly used to control operand-size assumptions
- Used to ensure predictable instruction encoding

Symbol Visibility Directives

GLOBAL

- Makes a symbol visible to the linker
- Allows other modules to reference it

GLOBAL _start

EXTERN

- Declares a symbol defined in another module
- Used to reference external functions or variables

EXTERN printf

COMMON

- Declares uninitialized global variables
- Storage is allocated by the linker

Program Layout Directives

- **SECTION**
  - Selects the section where subsequent code or data is placed
  - Common sections:
    - .text – code
    - .data – initialized data
    - .bss – uninitialized data

- **ABSOLUTE:**
  - The ABSOLUTE directive can be thought of as an alternative form of SECTION:
  - it causes the subsequent code to be directed at no physical section, but at the hypothetical section starting at the given absolute address.
  - Use Case
    - Low-level system code
    - Bootloaders
    - Memory-mapped hardware access

Warnings & Floating-Point Control

- **`WARNING`**
  - Enables or disables specific classes of assembler warnings
  - Helps control diagnostic output during assembly

- **`FLOAT`**
  - Controls floating-point constant behavior
  - Affects rounding and handling of denormalized numbers

Macros

- Provide symbolic abstraction and code reuse.
- Macros are expanded **before assembly**
- They do **not generate function calls**
- Improve readability and reduce repetitive code
- Expansion happens at **assembly time**, not runtime

Macros

NASM supports two form of macros.

- **Single-Line Macros**
  - Defined using the **%define** directive
  - Perform simple textual substitution

```
%define isTrue 1
```

- Multi line macros are defined similar to function in C.
  - Defined using **%macro** and **%endmacro**
  - Can accept parameters

```
%macro prologue 1
        Push ebp
        Mov ebp,,esp
        Sub esp,%1
%endmacro
```

Comments

- Comments are very important!
  - Explain the program's purpose
  - When it was written, revised, and by whom
  - Explain data used in the program
  - Explain instruction sequences and algorithm used
  - Application-specific explanations
- Single-line comments
  - Begin with a semicolon and terminate at end of line
- Multi-line comments
  - Begin with COMMENT directive and a chosen character.
  - End with the same chosen character.

`Using printf` and `scanf` via `extern`

- `printf` and `scanf` are **C library functions**
- They are **not built into assembly**, so we declare them with `extern`
- We must:
  - Push arguments **right to left**
  - Call the function
  - Clean the stack

nasm -f elf32 filename.asm -o filename.o

gcc -m32 filename.o -o filename

Program Example

```
section .data
    prompt      db "Enter a number: ", 0
    format_in   db "%d", 0
    format_out  db "You entered: %d", 10, 0   ; 10 =
  newline

section .bss
    num resd 1

section .text
    global _start
    extern printf
    extern scanf
_start:
    ; printf("Enter a number: ");
    push prompt
    call printf
    add esp, 4
```

```
; scanf("%d", &num);
push num
push format_in
call scanf
add esp, 8 ;

printf("You entered: %d\n", num);
mov eax, [num]
push eax
push format_out
call printf
add esp, 8

; return 0;
mov eax, 0
Ret
```

Output:
Enter a number: 32
You entered: 32