CSE504 Compiler Design

Assembly Language (NASM)

YoungMin Kwon

Overview

- We will learn how to write assembly programs.
- We will cover some assembly instructions used in the class, not the whole instructions set.

Hello World (hello.asm)

```
global _start           ; expose where _start is to the linker

section .text           ; where the code is

_start:                 ; the start label
    ; write(1, msg, 13)
    mov     rax, 1
    mov     rdi, 1
    mov     rsi, msg
    mov     rdx, len
    syscall

    ; exit(0)
    mov     rax, 60
    xor     rdi, rdi
    syscall

section .data                   ; where the initialized data is

    msg:    db    "Hello, World", 10
    len     equ   $ - msg       ; $ means here, $ - msg is msg length
```

To Compile hello.asm

- `nasm -felf64 hello.asm`
- `ld -o hello hello.o`

Sections

- `.text` section
  - Code will be placed here
- `.data` section
  - Initialized data will be placed here
  - `db, dw, dd, dq` for byte, word (2 byte), double word (4 byte), quadruple word (8 byte)
- `.bss` section
  - Uninitialized data will be placed here
  - `resb, resw, resd, resq`

mov instruction

- `mov dst, src`
  - dst can be registers or memory locations
  - src can be registers, memory locations, constants
- E.g.
  - `mov rax, 1          ; store 1 at rax register`
  - `mov rax, rbx        ; copy rbx to rax register`
  - `mov rax, [rbx]      ; copy the contents of memory pointed by rbx to rax`
  - `mov [rbx], rax      ; copy rax to the memory pointed by rbx`
  - `mov dword [rbx], 1 ; store 1 as a dword to the memory pointed by rbx`
  - `mov rax, msg        ; copy the address of msg to rax`
  - `mov [rax], [rbx] ;` **not supported**

Registers

- 8-bit registers
  - AL/AH, CL/CH, DL/DH, BL/BH, SPL, BPL, SIL, DIL, R8B-R15B
- 16-bit registers
  - AX, CX, DX, BX, SP, BP, SI, DI, R8W-R15W
- 32-bit registers
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D-R15D
- 64-bit registers
  - RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8-R15

Declaring Initialized Data

- In .data section
- `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 0xBEEF              ; 0xEF 0xBE`
- `dw 'a'                 ; 0x61 0x00 (it's just a number)`
- `dw 'ab'                ; 0x61 0x62 (character constant)`
- `dw 'abc'               ; 0x61 0x62 0x63 0x00 (string)`
- `dd 0xDEADBEEF          ; 0xEF 0xBE 0xAD 0xED`
- `dd 1.234567e20         ; floating-point constant`
- `dq 0xBEEFCAFEFEEDBEEF ; eight byte constant`
- `dq 1.234567e20         ; double-precision float`
- `buf: times 64 db 0; 64 bytes of 0`

Declaring Uninitialized Data

- In .bss section
- `buff: resb 64 ; reserve 64 bytes`
- `wvar: resw 1 ; reserve a word`
- `dwarray: resd 10 ; reserve 10 dwords`
- `qwarray: resq 10 ; reserve 10 qwords`

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
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
```

Constants

- `mov eax,'abcd'` ; 0x64636261 is copied to eax not 0x61626364. When eax is stored in memory, it will be read as `'a' 'b' 'c' 'd'`.
- `db 'hello'` ; string constant
  `db 'h','e','l','l','o'` ; equivalent character constants.
- Single quotes (‘…’) and double quotes (“…”) are equivalent.
- Backquotes (`…`) support C-style escape sequences.
  - \n, \r, \f, \’, \”, \`, …
- `equ`: defines a constant
  - `len equ 13`
  - `len equ $-msg` ; $ means the current position. So, $-msg is the length of msg

syscall

- A call to an OS service to get input, to produce output, or to exit the program...
- E.g.

```
mov     rax, 1   ; write
mov     rdi, 1   ; FD (stdout)
mov     rsi, msg ; address of the string
mov     rdx, len ; length of the string
syscall

mov     rax, 60  ; exit
xor     rdi, rdi ; exit code 0
syscall
```

- List of Linux System calls
  - http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Quiz

- Write echo.asm that reads a string and prints the string until q is entered.
- Read system call:

```
mov     rax, 0   ; read
mov     rdi, 2   ; FD (stdin)
mov     rsi, msg ; address of the buffer
mov     rdx, len ; length of the buffer
syscall
```

  – rax has the number of bytes read.

Some Arithmetic Instructions

- `add dst, src`
  - dst = dst + src
  - dst can be registers, memory locations
  - src can be registers, memory locations, immediates
- `sub dst, src`
  - dst = dst – src
- `mul src`
  - rdx:rax = rax * src
- `div src`
  - rax = rdx:rax / src (quotient)
  - rdx = rdx:rax % src (remainder)
- `inc dst`
  - dst = dst + 1
- `dec dst`
  - dst = dst - 1

Some Bitwise Logical Instructions

- `and dst, src`
  - dst = dst & src
- `or  dst, src`
  - dst = dst | src
- `neg dst`
  - dst = two's complement of dst (invert the bits and add 1)
- `not dst`
  - dst = one's complement of dst (invert the bits)
- `xor dst, src`
  - dst = dst xor src
- `shl, shr, rol, ror,` …

Stack Manipulation Instructions

- `push src`
  - push src to the stack, decrease rsp by 8
- `pop dst`
  - dst = top of the stack, increase rsp by 8.
- `pushf/popf`
  - push/pop the flag register
- `call label`
  - push the current IP and jump to the label
- `ret`
  - pop IP (jump to the top of the stack)

Flag Register and cmp/test

- Flags
  - zero flag: set when the result of an operation is 0
    - `xor ax, ax`
  - sign flag: set when the sign bit is set
    - ```
      mov ax, 0x7ff
      inc ax
      ```
  - carry flag: set when an overflow occurred
    - ```
      mov ax, 0xffff
      add ax, 1
      ```
  - Parity flag: set when the result of an operation has even number of ones.
    - ```
      mov ax, 2
      inc ax   (0000_0011b)
      ```
- `test src dst`
  - set the flag as if `and src dst` is run without updating src
- `cmp src dst`
  - set the flag as if `sub src dst` is run without updating src
      ```

Flag Register and Jump Instructions

- `jmp label`
  - jump to the label (set IP to the address of the label)
- `jz label, js label, jc label, jp label`
  - jump to label if zero, sign, carry, and parity flags are set respectively
- `jnz label, jns label, jnc label, jnp label`
  - jump if zero, sign, carry, and parity flags are NOT set respectively
- `je label, jne label, jg label, jge, label, jl label, jle label`
  - after `cmp src, dst` (`sub src, dst`), jump to label, if src is equal to, not equal to, greater than, greater than or equal to, less than, less than or equal to dst respectively.

```
;;
;; Example: Print Number
;;
global  _start, _exit, _print_num

section .text

_start:
    push    qword 123   ;; push the parameter
    call    _print_num  ;; call the print method
    add     sp, 8       ;; pop the parameter
    call    _exit       ;; we are done. exit.

_exit:
    mov     rax, 60     ;; exit(0)
    xor     rdi, rdi
    syscall

_print_num:
    push    rbp,        ;; save bp
    mov     rbp, rsp    ;; copy sp to bp
    push    rax         ;; save registers used
    push    rdx         ;; in this function
    push    rsi
    push    rdi

    mov     rax, [rbp + 16]  ;; the parameter
    mov     rsi, endbuf      ;; end of buffer
    mov     byte [rsi], `\n` ;; new line
    mov     rdi, 10

loop:
    xor     rdx, rdx   ;; rdx:rax / 10
    div     rdi        ;; rax:quotient, rdx:remainder

    add     dl, '0'    ;; update the string
    dec     rsi        ;; update the string pointer
    mov     [rsi], dl

    cmp     rax, 0     ;; check if we are done
    jg      loop

    mov     rdx, endbuf + 1 ;; length of the string
    sub     rdx, rsi   ;; string begins at rsi
    mov     rax, 1
    mov     rdi, 1
    syscall            ;; write(1,...)

    pop     rdi        ;; restore the registers
    pop     rsi
    pop     rdx
    pop     rax
    pop     rbp
    ret                ;; return to the caller

section .bss
    buf:    resb    100     ;; allocating a buffer
    endbuf  equ     $ - 1   ;; end of buffer
```

Quiz

- Write a function (_gcd) that takes two 64-bit numbers from the stack (like _print_num) and return their GCD.
- Using _gcd, the print the GCD and LCM of the two numbers.