

Nama : Alfian Yulianto

Nim : L20180121

Modul 1

1. Apa yang dimaksud dengan kode 'ASCII', buatlah tabel kode ASCII yang setandar tidakperlu extended, tuliskan kode ASCII dalam format angka desimal, binary dan hexadesimal serta karakter dan simbol yang dikodekn.

| Binary | Oct | Dec | Hex | Glyph | Binary | Oct | Dec | Hex | Glyph | Binary | Oct | Dec | Hex | Glyph |
|----------|-----|-----|-----|-------|----------|-----|-----|-----|-------|----------|-----|-----|-----|-------|
| 010 0000 | 040 | 32 | 20 | sp | 100 0000 | 100 | 64 | 40 | @ | 110 0000 | 140 | 96 | 60 | ` |
| 010 0001 | 041 | 33 | 21 | ! | 100 0001 | 101 | 65 | 41 | A | 110 0001 | 141 | 97 | 61 | a |
| 010 0010 | 042 | 34 | 22 | " | 100 0010 | 102 | 66 | 42 | B | 110 0010 | 142 | 98 | 62 | b |
| 010 0011 | 043 | 35 | 23 | # | 100 0011 | 103 | 67 | 43 | C | 110 0011 | 143 | 99 | 63 | c |
| 010 0100 | 044 | 36 | 24 | \$ | 100 0100 | 104 | 68 | 44 | D | 110 0100 | 144 | 100 | 64 | d |
| 010 0101 | 045 | 37 | 25 | % | 100 0101 | 105 | 69 | 45 | E | 110 0101 | 145 | 101 | 65 | e |
| 010 0110 | 046 | 38 | 26 | & | 100 0110 | 106 | 70 | 46 | F | 110 0110 | 146 | 102 | 66 | f |
| 010 0111 | 047 | 39 | 27 | ' | 100 0111 | 107 | 71 | 47 | G | 110 0111 | 147 | 103 | 67 | g |
| 010 1000 | 050 | 40 | 28 | (| 100 1000 | 110 | 72 | 48 | H | 110 1000 | 150 | 104 | 68 | h |
| 010 1001 | 051 | 41 | 29 |) | 100 1001 | 111 | 73 | 49 | I | 110 1001 | 151 | 105 | 69 | i |
| 010 1010 | 052 | 42 | 2A | * | 100 1010 | 112 | 74 | 4A | J | 110 1010 | 152 | 106 | 6A | j |
| 010 1011 | 053 | 43 | 2B | + | 100 1011 | 113 | 75 | 4B | K | 110 1011 | 153 | 107 | 6B | k |
| 010 1100 | 054 | 44 | 2C | , | 100 1100 | 114 | 76 | 4C | L | 110 1100 | 154 | 108 | 6C | l |
| 010 1101 | 055 | 45 | 2D | - | 100 1101 | 115 | 77 | 4D | M | 110 1101 | 155 | 109 | 6D | m |
| 010 1110 | 056 | 46 | 2E | . | 100 1110 | 116 | 78 | 4E | N | 110 1110 | 156 | 110 | 6E | n |
| 010 1111 | 057 | 47 | 2F | / | 100 1111 | 117 | 79 | 4F | O | 110 1111 | 157 | 111 | 6F | o |
| 011 0000 | 060 | 48 | 30 | 0 | 101 0000 | 120 | 80 | 50 | P | 111 0000 | 160 | 112 | 70 | p |
| 011 0001 | 061 | 49 | 31 | 1 | 101 0001 | 121 | 81 | 51 | Q | 111 0001 | 161 | 113 | 71 | q |
| 011 0010 | 062 | 50 | 32 | 2 | 101 0010 | 122 | 82 | 52 | R | 111 0010 | 162 | 114 | 72 | r |
| 011 0011 | 063 | 51 | 33 | 3 | 101 0011 | 123 | 83 | 53 | S | 111 0011 | 163 | 115 | 73 | s |
| 011 0100 | 064 | 52 | 34 | 4 | 101 0100 | 124 | 84 | 54 | T | 111 0100 | 164 | 116 | 74 | t |
| 011 0101 | 065 | 53 | 35 | 5 | 101 0101 | 125 | 85 | 55 | U | 111 0101 | 165 | 117 | 75 | u |
| 011 0110 | 066 | 54 | 36 | 6 | 101 0110 | 126 | 86 | 56 | V | 111 0110 | 166 | 118 | 76 | v |
| 011 0111 | 067 | 55 | 37 | 7 | 101 0111 | 127 | 87 | 57 | W | 111 0111 | 167 | 119 | 77 | w |
| 011 1000 | 070 | 56 | 38 | 8 | 101 1000 | 130 | 88 | 58 | X | 111 1000 | 170 | 120 | 78 | x |
| 011 1001 | 071 | 57 | 39 | 9 | 101 1001 | 131 | 89 | 59 | Y | 111 1001 | 171 | 121 | 79 | y |
| 011 1010 | 072 | 58 | 3A | : | 101 1010 | 132 | 90 | 5A | Z | 111 1010 | 172 | 122 | 7A | z |
| 011 1011 | 073 | 59 | 3B | ; | 101 1011 | 133 | 91 | 5B | [| 111 1011 | 173 | 123 | 7B | { |
| 011 1100 | 074 | 60 | 3C | < | 101 1100 | 134 | 92 | 5C | \ | 111 1100 | 174 | 124 | 7C | |
| 011 1101 | 075 | 61 | 3D | = | 101 1101 | 135 | 93 | 5D |] | 111 1101 | 175 | 125 | 7D | } |
| 011 1110 | 076 | 62 | 3E | > | 101 1110 | 136 | 94 | 5E | ^ | 111 1110 | 176 | 126 | 7E | ~ |
| 011 1111 | 077 | 63 | 3F | ? | 101 1111 | 137 | 95 | 5F | _ | | | | | |

2. Carilah daftar perintah bahasa assembly untuk mesin intel keluaran x86 lengkap (dari buku referensi atau internet). Daftar perintah ini dapat digunakan sebagai pedoman untuk memahami program 'boot.asm dan 'kernel.asm'.

Assembly AT&T dan NASM

Ada dua sintaks bahasa assembly, yaitu dalam format AT&T dan NASM. Sintaks AT&T banyak dipakai dalam lingkungan GNU seperti GNU Assembler, dan menjadi format default GNU Debugger (GDB). Sedangkan format NASM dipakai oleh netwide assembler dan banyak dipakai di lingkungan windows.

Perlu dicatat bahwa perbedaan NASM dan AT&T ini hanya masalah sintaks saja, keduanya menghasilkan bahasa mesin yang sama persis

Beberapa perbedaan antara format AT&T dan NASM adalah:

- Baris komentar diawali dengan “;” semicolon untuk NASM. AT&T mengawali komentar dengan # (hash)
- Dalam format AT&T, setiap register diawali dengan %. NASM tidak menggunakan %.
- Dalam format AT&T, setiap nilai literal (konstanta) diawali dengan \$. NASM tidak menggunakan \$.
- Pada perintah yang menggunakan operand sumber dan tujuan, format AT&T menuliskan tujuan sebagai operand kedua (contoh: CMD <source>,<dest>). Sedangkan NASM menuliskan tujuan sebagai operand pertama (contoh: CMD <dest>,<source>).

Register

Register adalah variabel internal yang sudah built-in di dalam prosesor yang bisa dipakai oleh programmer untuk bermacam-macam keperluan. Karena register posisinya di prosesor, bukan di memory, maka menggunakan register sebagai variabel jauh lebih cepat dibanding menggunakan variabel yang disimpan di suatu alamat di memori.

Berikut adalah jenis-jenis register yang ada pada prosesor Intel.

| Kategori | Nama | Penjelasan |
|-----------------|--------------------|--|
| General Purpose | EAX, EBX, ECX, EDX | Lebar data 32 bit, boleh dipakai untuk keperluan apa saja. E adalah Extended (karena awalnya register general purpose hanya 16 bit). |

| | | |
|------------------|-------------------------|---|
| | AX,BX,CX,DX | 16 bit bawah dari register 32 bit di atas. AX adalah bagian 16 bit bawah dari EAX. |
| | AH,AL,BH,BL,CH,CL,DH,DL | Bagian 8 bit dari register 16 bit di atas . AH adalah 8 bit atas dari AX. AL adalah 8 bit bawah dari AX. |
| Segment Register | CS, SS, DS, ES, FS, GS | Digunakan untuk menunjuk 16 bit awal alamat memori. CS = Code, SS = Stack, DS = Data, ES,FS,GS = Extra segment register |
| Offset Register | | Digunakan untuk menunjuk 16 bit akhir alamat memori. Alamat memori ditunjukkan dengan gabungan segment dan offset. |
| | EBP | Dipakai sebagai offset frame dalam stack. Biasanya menunjuk pada bottom of stack frame di suatu fungsi. ESP menunjukkan puncak stack, EBP menunjuk dasar stack. |
| | ESI | Biasanya dipakai untuk offset string sumber dalam operasi yang melibatkan blok memori. |
| | EDI | Biasanya dipakai untuk offset string tujuan dalam operasi yang melibatkan blok memori. |
| | ESP | Stack pointer, menunjukkan puncak dari stack. |
| Special | EFLAGS | Tidak bisa dipakai programmer, hanya dipakai prosesor untuk hasil operasi logical dan state. |
| | EIP | Tidak bisa dipakai programmer, hanya dipakai prosesor untuk menunjukkan alamat memori yang berisi instruksi berikutnya yang akan dieksekusi. |

Perhatikan gambar di bawah ini untuk melihat register-register yang ada dalam prosesor keluarga IA32 (Intel Architecture 32 bit).

courtesy of iu-bremen.de

Dalam gambar di atas terlihat bahwa register-register Extended (berawalan E) adalah register 32 bit. Agar kompatibel program-program sebelumnya ketika register hanya ada 16 bit, maka register yang lain adalah bagian bit bawah dari versi extendednya. Contohnya adalah register ESI dan SI. Register SI adalah 16 bit paling bawah dari ESI. Pada register EAX, AX adalah 16 bit paling bawah dari EAX. Register AX pun dipecah lagi menjadi 8 bit atas AH dan 8 bit bawah AL. Programmer bebas menggunakan yang mana saja sesuai kebutuhannya.

The Classic “Hello World”

Cukup sudah berteori, kini kita mulai berbasah-basah. Mari kita buat program pertama dalam assembly yang menampilkan teks “Hello World”. Dalam artikel ini saya menggunakan format syntax Intel, bukan AT&T. Silakan ketik source berikut dan simpan dalam nama hello.asm



```
1 ; section text khusus buat code
2 section .text
3
4 global _start
5
```

```

6 _start:
7 ; syscall = &gt; write(1,msg,len)
8 mov edx,len ; panjang string dimasukkan dalam register EDX
9 mov ecx,msg ; alamat memori yang menyimpan string dimasukkan dalam register ECX
10 mov ebx,1 ; file descriptor (1=stdout=defaultnya console) disimpan dalam register EBX
11 mov eax,4 ; Nomor syscall 4 adalah fungsi sys_write()
12 int 0x80 ; panggil system call dengan interrupt 80 hexa.
13
14 ; syscall = &gt; exit(0)
15 xor ebx,ebx ; membuat EBX menjadi 0 sebagai return code ketika exit
16 mov eax,1 ; nomor syscall 1 adalah fungsi exit()
17 int 0x80 ; panggil system call dengan interrupt 80 hexa.
18
19 section .data ; section data khusus buat data/variable
20 msg db "Hello, World!",0xa ; String diikuti dengan 0xA yaitu new line \n.
21 len equ $ - msg ; Panjang string didapat dengan mengurangi address di baris ini dengan alamat string.

```

Setelah itu kita akan mengcompile file ASM itu menjadi object code berformat ELF dengan NASM (netwide assembler). Setelah itu akan terbentuk file hello.o yang harus dilink dengan linker LD agar menjadi format executable.



```

1 $ nasm -f elf hello.asm
2 $ ld -s -o hello hello.o
3 $ ./hello
4 Hello, World!

```

Selamat, anda telah berhasil membuat program Hello World dalam bahasa Assembly. Program di atas sangat sederhana, kita memanggil system call write() untuk menampilkan string (msg), kemudian kita memanggil system call exit() untuk keluar dari program dan program selesai. String msg kita taruh dalam section .data karena section tersebut khusus untuk menyimpan data/variabel. Sedangkan instruksi assembly disimpan dalam section .text karena section text khusus untuk menyimpan code.

Hello World Opcode

Untuk melihat keterkaitan antara assembly dan bahasa mesin kita bisa melihat opcode dari program assembly yang kita buat dengan program objdump pada gambar berikut ini.

```
$ objdump -s -j .data ./hello
```

```
./hello:      file format elf32-i386
```

```
Contents of section .data:
```

```
80490a4 48656c6c 6f2c2057 6f726c64 210a      Hello, World!.
```

```
$ objdump -d -m i386:intel ./hello
```

```
./hello:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048080 <.text>:
```

| | | | | |
|----------|----------------|---|-----|----------------|
| 8048080: | ba 0e 00 00 00 | ↔ | mov | edx, 0xe |
| 8048085: | b9 a4 90 04 08 | ↔ | mov | ecx, 0x80490a4 |
| 804808a: | bb 01 00 00 00 | ↔ | mov | ebx, 0x1 |
| 804808f: | b8 04 00 00 00 | ↔ | mov | eax, 0x4 |
| 8048094: | cd 80 | ↔ | int | 0x80 |
| 8048096: | bb 00 00 00 00 | ↔ | mov | ebx, 0x0 |
| 804809b: | b8 01 00 00 00 | ↔ | mov | eax, 0x1 |
| 80480a0: | cd 80 | ↔ | int | 0x80 |

opcode =
machine language

assembly

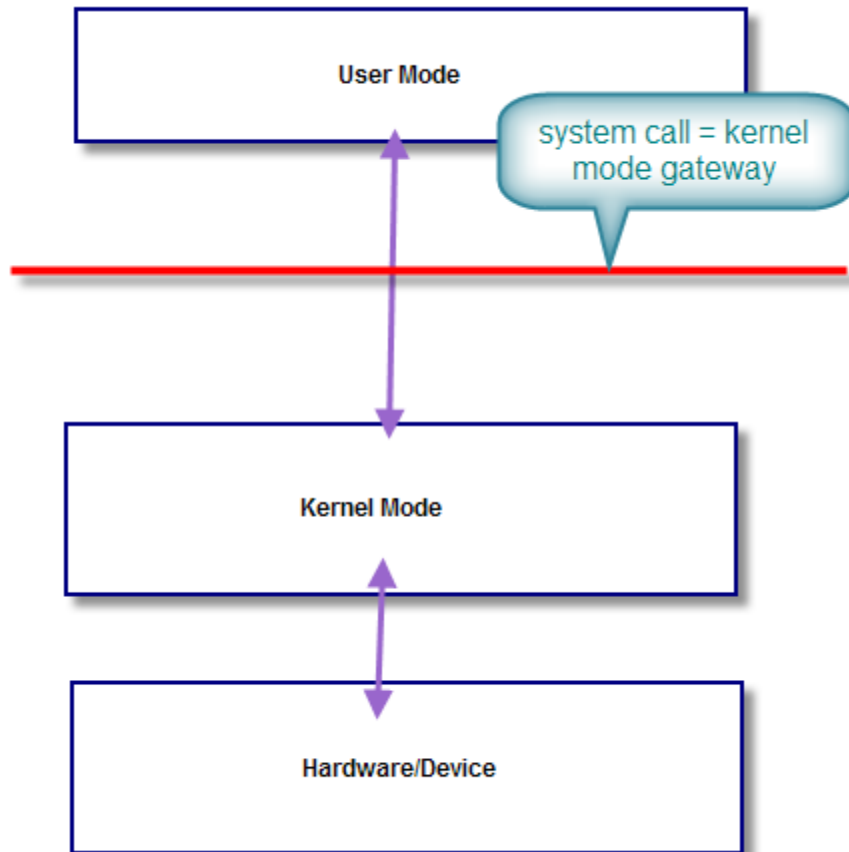
address of
"Hello, World!"

Opcode di sebelah kiri adalah versi bahasa mesin dari bahasa assembly di sebelah kanannya. Hal ini menunjukkan eratnya kaitan antara assembly dan bahasa mesin. Contohnya adalah instruksi assembly INT 0x80 diterjemahkan ke bahasa mesin: 0xCD 0x80 (dalam hexa) atau 11001101 (binary dari 0xCD) 10000000 (binary dari 0x80).

Perhatikan bahwa pada source code assembly, "MOV EDX, len" setelah dicompile diterjemahkan menjadi "MOV EDX, 0xE". Hal ini karena len adalah konstanta berisi panjang string "Hello, World!" yaitu sepanjang 14 karakter. Instruksi assembly pada source code "MOV ECX, msg" setelah dicompile diterjemahkan menjadi "MOV ECX, 0x80490a4". Mengapa msg diterjemahkan menjadi 0x80490a4? Hal ini karena msg adalah address dari string "Hello, World!" sehingga setelah dicompile diterjemahkan menjadi alamat 0x80490a4. Terlihat juga pada gambar di atas pada lokasi 0x80490a4 terdapat string "Hello, World!".

System Call

Dalam program hello world di atas kita memanfaatkan system call untuk menampilkan teks di layar monitor. System call adalah gerbang menuju kernel mode bagi program yang membutuhkan servis yang hanya bisa dikerjakan oleh kernel.



System call di Linux dipanggil dengan menggunakan interrupt 80 hexa. Nomor system call dimasukkan dalam register EAX. Daftar lengkap nomor syscall di Linux bisa dibaca di file header: `/usr/include/asm/unistd.h`. Berikut adalah cuplikan isi dari file `unistd.h`



```
1 #ifndef _ASM_I386_UNISTD_H_
2 #define _ASM_I386_UNISTD_H_
3
4 /*
5  * This file contains the system call numbers.
6  */
7
```

```

8 #define __NR_restart_syscall 0
9 #define __NR_exit 1
10 #define __NR_fork 2
11 #define __NR_read 3
12 #define __NR_write 4
13 #define __NR_open 5
14 #define __NR_close 6
15 #define __NR_waitpid 7
16 #define __NR_creat 8

```

Dalam contoh hello world kita memanfaatkan system call nomor 4 (write) dan nomor 1 (exit). Untuk mengetahui cara pemakaian dan argumen untuk system call tersebut, kita bisa gunakan man di Linux.



```

1 $ man 2 write
2 SYNOPSIS
3 ssize_t write(int fd, const void *buf, size_t count);
4 $ man 2 exit
5 void _exit(int status);

```

Dari manual system call write meminta 3 argument: yaitu file descriptor bertipe integer, alamat memori tempat string berada, dan terakhir adalah panjang string bertipe integer. Ketika argumen tersebut disimpan dalam register mulai dari EBX, ECX dan EDX. Argumen pertama di EBX, argumen kedua di ECX dan ketiga di EDX. Register EAX dipakai untuk menyimpan nomor system call. Dari manual system call exit meminta 1 argument: yaitu kode status bertipe integer yang disimpan dalam register EBX.

Dalam contoh hello world di atas kita menggunakan 3 intruksi assembly yaitu MOV, XOR dan INT. Mari kita bahas intruksi tersebut.

Instruksi MOV

Kita menggunakan MOV untuk menyalin data dari sumber ke tujuan. Sumber dan tujuan bisa alamat memori, atau register. Perhatikan contoh berikut:

| NASM/Intel | AT&T | Deskripsi |
|---------------|-------------------|---|
| MOV EAX, 0x51 | MOVL \$0x51, %EAX | Mengisi register EAX dengan nilai 51 hexa |
| MOV ESP, EBP | MOVL EBP, ESP | Menyalin isi register EBP ke register ESP |

Perbedaan antara sintaks NASM dan AT&T adalah arah pengkopian. Dalam sintaks NASM, tujuan ada pada operand pertama, sedangkan dalam sintaks AT&T tujuan adalah operand ke-2.

Instruksi XOR

Instruksi XOR digunakan untuk melakukan operasi logika Exclusive OR. XOR akan menghasilkan 0 bila kedua operandnya sama, dan menghasilkan 1 bila tidak sama. XOR ini biasanya dipakai untuk membuat register menjadi 0 dengan melakukan XOR untuk operand yang sama seperti pada contoh hello world tersebut.

| NASM/Intel | AT&T | Deskripsi |
|-------------|---------------|--|
| XOR EBX,EAX | XOR %EAX,%EBX | XOR isi EBX dengan EAX, hasilnya disimpan di EBX |

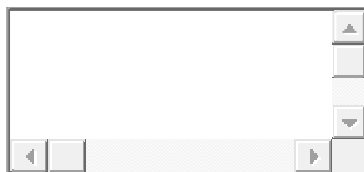
Instruksi INT

Instruksi INT digunakan untuk mengirim sinyal interrupt ke prosesor. Dalam contoh di atas kita memakai interrupt nomor 80 hexa untuk meminta layanan dari kernel.

| NASM/Intel | AT&T | Deskripsi |
|------------|------------|-----------------------------------|
| INT 0x80 | INT \$0x80 | Memanggil interrupt nomor 80 hexa |

Contoh Lain: Hello World X Times

Kali ini kita akan memodifikasi program hello world di atas agar bisa menampilkan pesan yang sama berkali-kali tergantung dari argumen yang dimasukkan user.



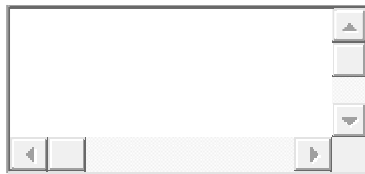
```
1 section .text
2
3 global _start
4
5 _start:
6 pop eax ; pop number of argc (diabaikan)
7 pop eax ; pop argv[0] (diabaikan karena berisi nama program)
8 pop eax ; pop argv[1] (ini dipakai untuk stringtoint)
9 call stringtoint ; ECX berisi argumen bertipe integer sebagai counter
10
11 _print:
12 push ecx          ; selamatkan counter di stack karena ECX dipakai juga di _print_hello
13 call _print_hello ; print hello world
14 pop ecx           ; ambil lagi counter dari stack karena akan dipakai untuk looping
```

```

15 loop _print      ; kurangi ECX dengan 1, bila belum 0 kembali ke _print
16
17 ; ini system call exit(0)
18 mov ebx,0
19 mov eax,1
20 int 0x80
21
22 _print_hello:    ; syscall write(1,msg,len)
23 mov edx,len
24 mov ecx,msg
25 mov ebx,1
26 mov eax,4
27 int 0x80
28 ret
29
30 stringint:       ; mengubah string di lokasi yang ditunjuk EAX menjadi integer di ECX
31 ; EAX address of string
32 xor ecx,ecx      ; clear ECX
33 xor ebx,ebx      ; clear EBX
34 mov bl,[eax]     ; BL berisi kode ASCII string di lokasi yang ditunjuk EAX
35 sub bl,0x30      ; Kode ascii angka adalah 30h-39h, dikurangkan dengan 30h
36 add ecx,ebx      ; ECX ditambah EBX, ECX berisi nilai integer
37 ret
38
39 section .data
40 msg db "Hello, World!",0xa
41 len equ $ - msg

```

Simpan source code di atas dengan nama `helloxtimes.asm`, lalu compile dan link seperti di bawah ini.



```

1 $ nasm -f elf helloxtimes.asm
2 $ ld -s -o helloxtimes helloxtimes.o
3 $ ./helloxtimes 1
4 Hello, World!
5 $ ./helloxtimes 2
6 Hello, World!
7 Hello, World!
8 $ ./helloxtimes 3
9 Hello, World!
10 Hello, World!
11 Hello, World!
12 $ ./helloxtimes 4
13 Hello, World!
14 Hello, World!
15 Hello, World!
16 Hello, World!

```

Kita belajar beberapa instruksi baru dalam contoh ke-2 ini, yaitu looping, penggunaan argumen dan prosedur, sedangkan system call yang dipakai tetap sama, yaitu `write()` dan `exit()`.

Program kali ini menerima argumen berupa integer 1-9 yang dipakai sebagai counter berapa kali pesan akan muncul di monitor. Argumen ini diambil dari stack dengan instruksi POP. Pada puncak stack ada argc, yaitu jumlah argumen ketika program dijalankan. Di bawahnya berisi address dari argv[0] yaitu nama program. Kemudian di bawahnya lagi baru berisi address dari argv[1] yaitu parameter/argumen pertama. Perhatikan pada baris ke-6 s/d baris ke-8 ada instruksi POP EAX sebanyak tiga kali. Ini dilakukan karena yang diperlukan ada pada posisi ke-3 sehingga kita harus membuang 2 elemen di puncak sebelum bisa mengambil address argv[1]. Address argumen ke-1 diambil dari POP lalu disimpan dalam register EAX. Karena bentuknya masih string, maka harus diubah dulu menjadi integer dengan memanggil prosedur stringtoint pada baris ke-9.



Instruksi POP untuk Mengambil Argumen

Ketika program dijalankan dengan satu argumen seperti “./helloxtimes 7”. Maka jumlah argumen (argc) akan berisi 2, yaitu nama program itu sendiri, dan satu argumennya. ARGV akan disimpan pada puncak stack, dan elemen di bawahnya berisi alamat memori dari nama program, dan dibawahnya lagi berisi alamat memori dari argumen pertama.

Perhatikan gambar di atas yang menunjukkan proses pengambilan alamat memori berisi string argumen pertama dari stack. Dalam contoh tersebut argumen adalah string “7”, yaitu karakter berkode ASCII 37 hexa diikuti dengan ASCII 0 (karakter NULL). Alamat memori berisi string argumen pertama itu diambil dari stack dan disimpan di register EAX.

Pada prosedur stringtoint, register EAX berisi address string yang akan diubah menjadi integer. Kita hanya mengambil karakter pertama saja, pada baris ke-35 dengan instruksi MOV, isi memori pada address yang ditunjuk oleh EAX dicopy ke register BL.

“MOV EBX, [EAX]” berbeda dengan “MOV EBX,EAX”. MOV EBX,[EAX] berarti menyalin isi memori pada alamat yang disimpan di EAX ke dalam

register EBX. Sedangkan MOV EBX,EAX berarti menyalin isi register EAX ke register EBX

Saya menggunakan register BL karena kode ASCII lebarnya hanya 8 bit. Bila benar berisi angka, maka register BL akan berisi nilai 30h-39h (kode ascii untuk “0”-“9”). Setelah itu register BL dikurangi dengan 30h untuk mendapatkan nilai dari 0-9. Setelah itu hasilnya ditambahkan ke register ECX sehingga kembali dari prosedur ini dengan nilai integer hasil konversi di register ECX.

Setelah mendapatkan nilai argumen bertipe integer di register ECX, selanjutnya ECX ini perlu diselamatkan dulu dalam stack (baris 12) sebab ECX akan dipakai dalam prosedur `_print_hello` (baris 13). ECX dipakai sebagai alamat string msg ketika memanggil system call `write()`. Setelah kembali dari prosedur `_print_hello`, nilai ECX perlu dikembalikan seperti semula dengan `POP ECX` (baris 14) sebab akan dipakai sebagai counter dalam `LOOP` (baris 15). Ketika menjalankan instruksi `LOOP`, register ECX akan dikurangi 1, kemudian bila `ECX > 0` maka program akan lompat ke `_print`. Bila ECX bernilai 0, maka loop berhenti dan menjalankan system call `exit(0)`.

Setelah memahami cara kerja program contoh ke-2 itu. Sekarang mari kita bahas instruksi baru yang ada di sana: `CALL`, `RET`, `PUSH`, `POP`, `LOOP`.

Instruksi PUSH dan POP

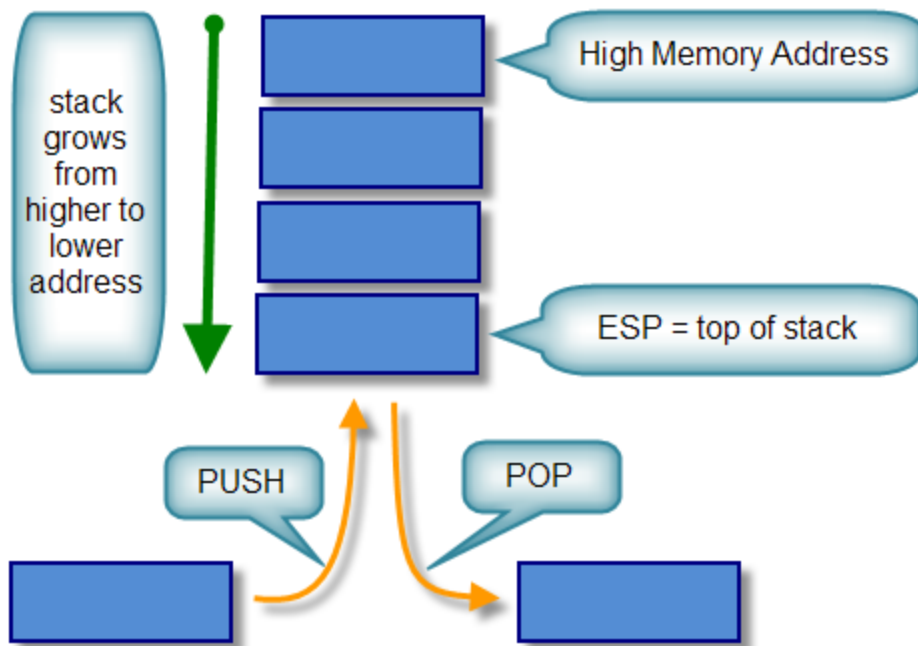
Instruksi `PUSH` digunakan untuk menyimpan nilai ke dalam stack. Kebalikannya adalah instruksi `POP` untuk mengambil nilai dari stack. Stack dalam Linux membesar ke alamat memori yang lebih rendah. Puncak stack ada di alamat rendah, sedangkan dasar stack ada di alamat yang lebih tinggi.

| NASM/Intel | AT&T | Deskripsi |
|-------------------------|--------------------------|------------------------------------|
| <code>PUSH value</code> | <code>PUSHL value</code> | Menyimpan nilai ke dalam stack |
| <code>POP dest</code> | <code>POPL dest</code> | Mengambil nilai dari stack ke dest |

Struktur Data Stack

Stack adalah struktur data yang mirip seperti tumpukan piring. Data yang diambil dari stack adalah data yang dimasukkan terakhir, atau

istilahnya adalah LIFO (last in first out). Jadi kalau kita ingin mengambil data di tengah-tengah tumpukan, caranya adalah dengan mengambil dulu data dari puncak sampai habis, sehingga data yang kita inginkan berada di puncak stack.



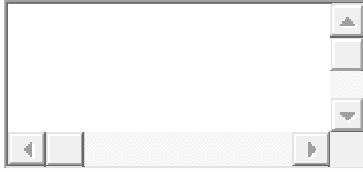
PUSH dan POP dalam Stack

Reguster ESP menunjukkan alamat memori dari puncak stack. Setiap ada instruksi PUSH, maka register ESP berkurang (ingat stack bertumbuh ke alamat yang makin mengecil) karena puncak stack berubah. Begitu pula bila sebaliknya bila ada instruksi POP, maka register ESP akan bertambah.



1 PUSH EAX

PUSH EAX di atas sama dengan dua instruksi di bawah ini:



```
1 SUB ESP, 4  
2 MOV DWORD PTR SS:[esp], EAX
```

PUSH EAX (4 byte) bisa dilakukan dengan mengurangi ESP dengan 4, kemudian menyalin isi EAX ke memori di lokasi `SS:[ESP]`, yaitu di segment stack pada offset yang ditunjuk oleh ESP. `DWORD PTR` menunjukkan bahwa lebar data yang akan disalin ke memori dalam instruksi `MOV` itu selebar 4 byte.



```
1 POP EAX
```

POP EAX di atas sama dengan dua instruksi di bawah ini:



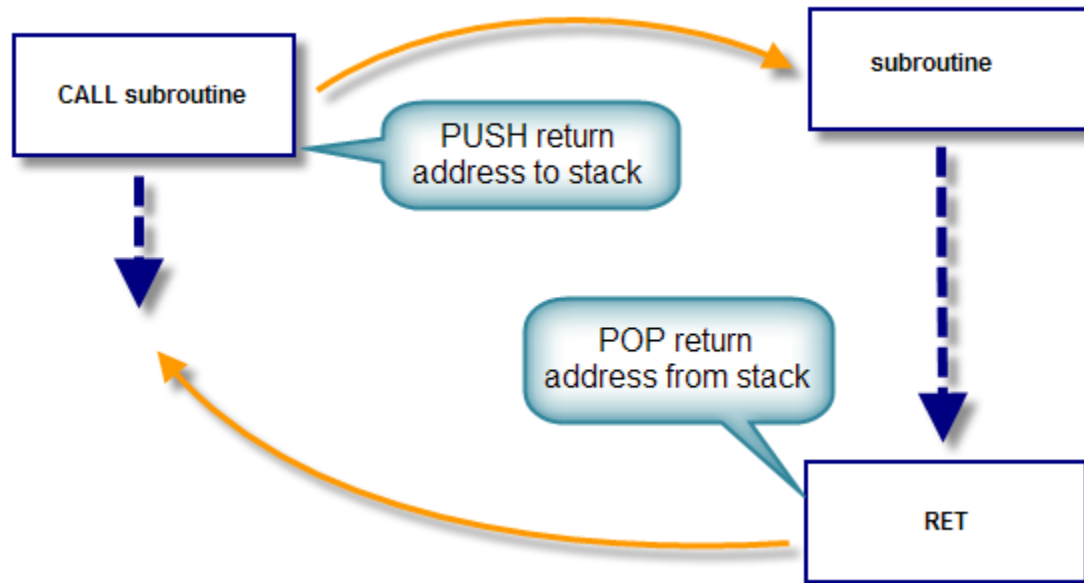
```
1 MOV EAX, DWORD PTR SS:[esp]  
2 ADD ESP, 4
```

POP EAX (4 byte) bisa dilakukan dengan menyalin isi memori di lokasi `SS:[ESP]`, yaitu di segment stack pada offset yang ditunjuk oleh ESP ke register EAX, lalu menambahkan ESP dengan 4. `DWORD PTR` menunjukkan bahwa lebar data yang akan disalin ke memori dalam instruksi `MOV` itu selebar 4 byte.

Instruksi **CALL** dan **RET**

Instruksi **CALL** digunakan untuk memanggil sebuah prosedur. Sedangkan **RET** dipakai untuk kembali dari prosedur kembali ke lokasi setelah instruksi pemanggilan. Ketika instruksi **CALL** dijalankan, prosesor menyimpan alamat instruksi sesudah instruksi **CALL** ke dalam stack (return address), kemudian prosesor lompat ke alamat subroutine yang dituju. Ketika instruksi **RET** dijalankan, maka prosesor mengambil (POP) return address (alamat yang di-push ketika **CALL**), kemudian loncat ke alamat tersebut.

| NASM/Intel | AT&T | Deskripsi |
|------------------|------------------|--------------------------------|
| CALL subroutine1 | CALL subroutine1 | Memanggil prosedur subroutine1 |
| RET | RET | Kembali dari prosedur |



Instruksi LOOP

LOOP digunakan untuk melakukan looping sejumlah nilai yang ada pada register ECX. Ketika ada instruksi LOOP, prosesor akan mengurangi nilai ECX dengan 1, kemudian membandingkan hasilnya. Bila nilai ECX sekarang masih > 0 , maka prosesor akan loncat ke alamat yang ditunjuk dalam LOOP. Bila nilai ECX sekarang menjadi 0, prosesor tidak akan loncat, tapi melanjutkan mengerjakan instruksi selanjutnya setelah LOOP.

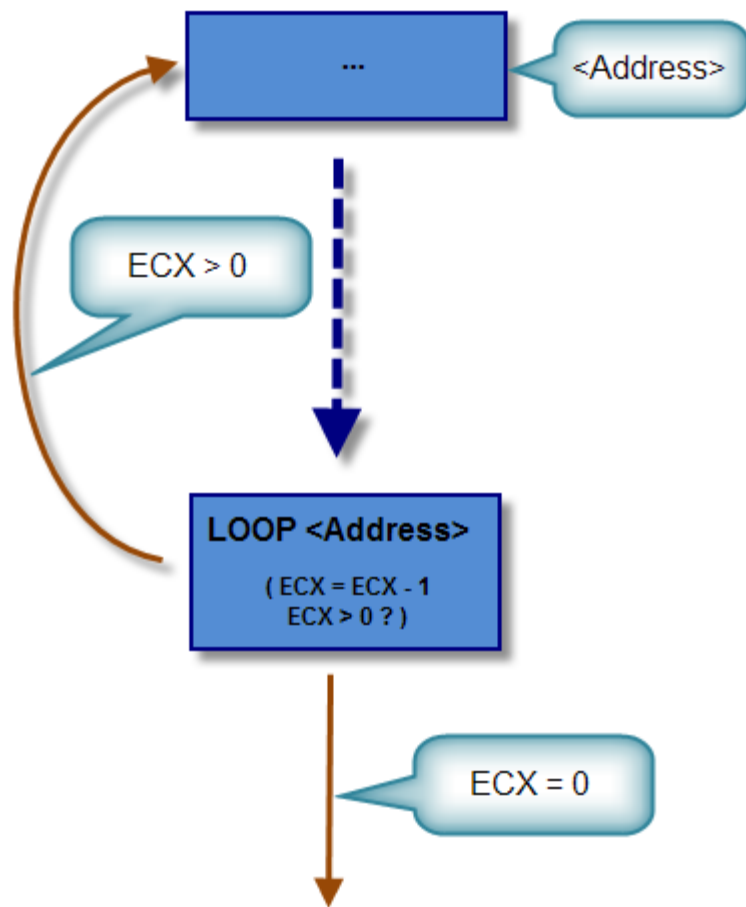
| NASM/Intel | AT&T | Deskripsi |
|--------------|--------------|--|
| LOOP address | LOOP address | Looping ke alamat yang ditunjukkan oleh address bila $ECX > 0$. |

Instruksi tunggal “LOOP address” ekuivalen dengan 2 instruksi assembly berikut:



1 DEC ECX ; DECREMENT: $ECX = ECX - 1$, register ECX dikurangi 1

2 JNZ address ; JUMP IF NOT ZERO: Bila ECX masih belum 0, JUMP ke address



Pada gambar di atas ada dua kondisi yang mungkin yaitu $ECX > 0$ atau $ECX = 0$. Mungkin ada yang bertanya, lho bagaimana dengan kondisi $ECX < 0$? Ingat komputer hanya mengenal 2 simbol, yaitu 0 dan 1, jadi pada dasarnya tidak ada "-1" atau "-0" dalam representasi binary. Bilangan negatif direpresntasikan dengan pengkodean two-complement, silakan baca di [signed number representation](#) karena itu diluar topik yang kita bahas sekarang.

Bila ECX bernilai 0 sebelum mengerjakan instruksi LOOP, maka yang terjadi adalah program akan looping sebanyak 0xFFFFFFFF atau 4.294.967.295 kali. Hal ini terjadi karena $0 - 1 = -1$ yang dalam binary adalah 0xFFFFFFFF dengan sistem two-complement.