KOM206 - ORGANISASI KOMPUTER
PENGENALAN BAHASA *ASSEMBLY* DAN TIPE REGISTER

TUJUAN PRAKTIKUM

1 Mahasiswa mampu menjelaskan fungsi CPU dan register pada lingkungan x86
2 Mahasiswa dapat menjelaskan jenis dan fungsi register
3 Mahasiswa dapat membuat program sederhana menggunakan bahasa *assembly*

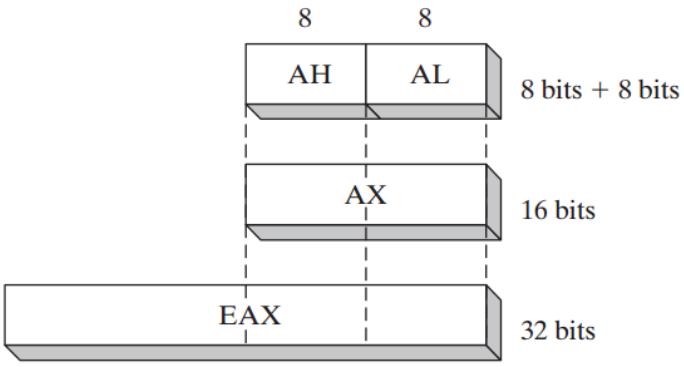
TEORI PENUNJANG

Memori komputer pada dasarnya merupakan *array byte* yang menggunakan perangkat lunak untuk instruksi dan data. Meskipun memori relatif cepat, namun ada kebutuhan untuk pemindahan sejumlah kecil data yang lebih cepat untuk memungkinkan CPU untuk menjalankan instruksi lebih cepat. Salah satu jenis memori yang lebih cepat adalah memori *cache*, yang mungkin 10 kali lebih cepat memori utama. Tipe kedua dari memori yang lebih cepat adalah register CPU. *Cache* berukuran beberapa megabyte, sementara CPU hanya memiliki beberapa register.

CPU x86-64 memiliki 16 *general purpose register* (GPR) dan 16 *floating point register* (FPR). Tiap GPR lebarnya 64-bit (sesuai dengan arsitektur CPU) dan tersimpan di dalam CPU. *Software* dapat mengakses register dalam ukuran 64, 32, 16, atau 8-bit.

Selain GPR, CPU juga memiliki *special purpose register* (SPR). Berikut contoh GPR dan SPR pada CPU 16-bit.

- `ax` - *accumulator*
- `bx` - *base register*
- `cx` - *count register*
- `dx` - *data register*
- `si` - *source index*
- `di` - *destination index*
- `bp` - *base pointer* (*for function frames*)
- `sp` - *stack pointer*



Gambar 1 *General purpose registers*

GPR secara umum digunakan untuk aritmetika dan pemindahan data. Untuk mengakses *low byte* dari `ax` dapat menggunakan `al`, sementara untuk mengakses *high byte* dari `ax` dapat menggunakan `ah`. Hal ini berlaku juga untuk register `bx`, `cx`, dan `dx`.

Ketika CPU i386 dibuat, register dilebarkan menjadi 32-bit dan diganti nama menjadi `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, dan `esp`. *Software* dapat mengakses 16-bit terendah dengan menggunakan nama awalnya. Sementara, SPR hanya dapat diakses dengan nama 32-bit dan 16-bit nya saja.

32-bit	16-bit	8-bit (*high*)	8-bit (*low*)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Beberapa register memiliki kegunaan khusus:

- `EAX` secara otomatis digunakan untuk instruksi perkalian dan pembagian.
- CPU secara otomatis menggunakan `ECX` sebagai loop counter
- Data alamat ESP pada *stack* biasanya digunakan sebagai *extended stack pointer register*.
- `ESI` dan `EDI` digunakan oleh instruksi transfer memori dengan kecepatan tinggi.
- `EBP` digunakan oleh *high-level languages* untuk mereferensi parameter fungsi dan lokal variabel dalam *stack*.

MATERI PRAKTIKUM

Format Bahasa Assembly

Dalam penggunaan bahasa *assembly* di x86, instruksi dituliskan dalam format seperti berikut:

```
[LABEL:]     MNEMONIC [OPERAND1], [OPERAND2], [OPERAND3]
```

Label merupakan *identifier* untuk sebuah baris ataupun blok kode, biasanya label diikuti dengan tanda titik dua (:). *Mnemonic* merupakan nama dari instruksi, misalnya `ADD`, `SUB`, `MOV`, dan lain-lain. *Operand* dalam instruksi dapat berjumlah dari nol hingga tiga *operand*, tergantung pada operasi yang digunakan. Setiap *operand* dipisahkan dengan tanda koma (,).

Mode Pengalamatan

Mode pengalamatan memberikan berbagai cara untuk mengakses *operand*. Terdapat empat mode pengalamatan:

1 *Register addressing*

Dalam mode pengalamatan ini, instruksi memilih satu atau lebih register yang merepresentasikan *operand*. Mode pengalamatan ini memiliki kecepatan eksekusi yang tinggi karena seluruh operasi dilaksanakan di dalam CPU.

```
INC AH        ; increment register AH
MOV AX, BX    ; memindahkan isi dari register BX ke register AX
```

Register sumber dan tujuan dapat dari berbagai GPR, baik 8-bit, 16-bit, 32-bit, atau 64-bit. Segmen register CS, DS, SS, ES, FS, atau GS juga dapat digunakan dalam mode pengalamatan ini.

2 *Immediate addressing*

Mode pengalamatan ini menggunakan nilai konstan dalam operasinya. Dalam instruksi dengan dua *operand*, *operand* pertama merupakan register atau lokasi memori sementara *operand* kedua merupakan nilai konstan.

```
MOV AX, 3H    ; memindahkan nilai 3 heksadesimal ke register AX
ADD AX, 5H    ; menambahkan nilai 5 heksadesimal ke register AX
```

3 *Direct Memory Adressing*

Dalam *direct memory addressing*, salah satu *operand* merujuk ke sebuah lokasi memori dan *operand* lainnya mereferensikan ke register.

```
MOV AL, [1A33D4H] ; memindahkan nilai dalam memori 1A33D4H ke AL
```

4 *Indirect Memory Addressing*

Indirect memory addressing biasanya digunakan ketika register terdiri atas alamat data, bukan nilai data yang akan diakses. Metode pengalamatan ini menggunakan GPR dalam format kurung siku, misalnya `[BX]`, `[BP]`, `[EAX]`, `[EBX]`, dan lain-lain.

```
MOV AX, [EBX] ; memindahkan nilai pada alamat EBX ke register AX
```

System Calls

System calls adalah sebuah API untuk antarmuka antara *user-space* dan *kernel-space.* Berikut beberapa jenis *system call*.

EAX	Name	EBX	ECX	EDX	ESX	EDI
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Contoh penggunaan *syscall* yang ekivalen dengan memanggil fungsi `exit(1)`:

```
mov    eax, 1         ; system call number (sys_exit)
int    0x80           ; call kernel
```

Program 1 Hello, world!

```
segment  .text                  ;code segment
       global _start            ;must be declared for linker
_start:                         ;tell linker entry point
        mov edx,len             ;message length
        mov ecx,msg             ;message to write
        mov ebx,1               ;file descriptor (stdout)
        mov eax,4               ;system call number (sys_write)
        int 0x80                ;call kernel
        mov eax,1               ;system call number (sys_exit)
        int 0x80                ;call kernel

segment .data                   ;data segment
msg     db 'Hello, world!',0xa  ;our dear string
```

Program 2 Menampilkan 9 bintang

```
section .text
       global  _start         ;must be declared for linker (gcc)
_start:                       ;tell linker entry point
        mov     edx,len       ;message length
        mov     ecx,msg       ;message to write
        mov     ebx,1         ;file descriptor (stdout)
        mov     eax,4         ;system call number (sys_write)
        int     0x80          ;call kernel
        mov     edx,9         ;message length
        mov     ecx,s2        ;message to write
        mov     ebx,1         ;file descriptor (stdout)
        mov     eax,4         ;system call number (sys_write)
        int     0x80          ;call kernel
        mov     eax,1         ;system call number (sys_exit)
        int     0x80          ;call kernel
section .data
msg     db 'Displaying 9 stars',0xa   ;a message
len     equ $ - msg                   ;length of message
s2      times 9 db '*'
```

Latihan

1 Buat program sederhana untuk:
   - menyimpan nilai 012h, 034h, dan 067h ke masing-masing offset 0150h, 0151h, dan 0152h
   - menjumlahkan ketiga nilai tersebut
   - menyimpan hasil penjumlahannya ke offset 0153h dan register AX.

2 Buat program sederhana untuk:
   - Menyimpan nilai 014h, 0FFh
   - Menjumlahkan nilai tersebut dan menyimpannya ke register
   - Hasil penjumlahan tersebut di kurangi dengan 03Ah
   - Hasil pengurangan disimpan ke register AX

Debugging Assembly with gdb

We've only covered a handful of useful `gdb` commands in this tutorial to get you started using `gdb`. Many of these commands can be shortened (e.g., "`si`" is equivalent to "`stepi`"). The `gdb` debugger itself has a nice help feature. Simply type "`help`" to get a list of commands, and `help [command]` will give you more information on that command. Again, more complete documentation on gdb is available through the "`info gdb`" command typed in the UNIX shell.

Command	Example	Description
`run`		start program
`quit`		quit out of gdb
`cont`		continue execution after a break
`break [addr]`	`break *_start+5`	sets a breakpoint
`delete [n]`	`delete 4`	removes nth breakpoint
`delete`		removes all breakpoints
`info break`		lists all breakpoints
`stepi`		execute next instruction
`stepi [n]`	`stepi 4`	execute next n instructions
`nexti`		execute next instruction, stepping over function calls
`nexti [n]`	`nexti 4`	execute next n instructions, stepping over function calls
`where`		show where execution halted
`disas [addr]`	`disas _start`	disassemble instructions at given address
`info registers`		dump contents of all registers
`print/d [expr]`	`print/d $ecx`	print expression in decimal
`print/x [expr]`	`print/x $ecx`	print expression in hex
`print/t [expr]`	`print/t $ecx`	print expression in binary
`x/NFU [addr]`	`x/12xw &msg`	Examine contents of memory in given format
`display [expr]`	`display $eax`	automatically print the expression each time the program is halted
`info display`		show list of automatically displays
`undisplay [n]`	`undisplay 1`	remove an automatic display

Here is sample assembly program with several memory items defined:

```
segment .data
a       dd      4
b       dd      4.4
c       times   10 dd 0
d       dw      1, 2
e       db      0xfb
f       db      "hello world", 0

segment .bss
g       resd    1
h       resd    10
i       resd    100
```

Printing with gdb

Letter	Format
d	Decimal (default)
x	Hexadecimal
t	Binary
u	Unsigned
f	Floating point
i	Instruction
c	Character
s	String
a	Address

Let's see a a few commands in action in gdb:

```
(gdb) p a
$32 = 4
(gdb) p/a &a
$33 = 0x601018 <a>
(gdb) p b
$34 = 1082969293
(gdb) p/f b
$35 = 4.4000001
(gdb) p/a &b
$36 = Ox60101c <b>
```

```
(gdb) p/x &b
$37 = 0x60101c
(gdb) p/a &c
$39 = 0x601020 <c>
(gdb) p/a &d
$40 = 0x601048 <d>
(gdb) p/a &e
$41 = 0x60104c <e>
(gdb) p/a &f
$42 = 0x60104d <f>
(gdb) p/a &g
$43 = 0x601070 <g>
(gdb) p/a &h
$45 = 0x601074 <h>
(gdb) p/a &i
$46 = 0x60109c <i>
```

We see that `gdb` handles `a` perfectly. It gets the type right and the length. It needs the /`f` option to print `b` correctly. Notice that `a` is located at address 0x601018 which is 24 bytes after the start of a page in memory. `gdb` will prohibit accessing memory before `a`, though there is no hardware restriction to the previous 24 bytes. We see that the data segment variables are placed in memory one after another until `f` which starts at 0x60104d and extends to 0x601058. There is a gap until the bss segment which starts with `g` at address 0x601070. The `bss` data items are placed back to back in memory with no gaps.

Examining Memory

Letter	Size	Bytes
b	Byte	1
h	Halfword	2
w	Word	4
g	Giant	8

Here are some examples of examining memory:

```
(gdb) x/w &a
0x601018 <a>: 0x4
(gdb) x/fw &b
0x60101c <b> : 4.4000001
(gdb) x/fg &b
```

```
Ox60101c <b> : 5.3505792317228316e-315
(gdb) x/10dw &c
Ox601020 <c>: 0 0 0 0
Ox601030 <c+16>: 0 0 0 0
Ox601040 <c+32>: 0 0
(gdb) x/2xh &d
Ox601048 <d>: Ox0001 Ox0002
(gdb) x/12cb &f
Ox60104d <f>: 104 'h'101 'e'108 '1'108 '1'111 'o'32' '119' ... Ox601055 <f+8> :
114 'r'108 '1'100 'd'O '¥000'
(gdb) x/s &f
Ox60104d <f>: "hello world"
```

Things match what you expect if you use the correct format and size. I first printed `b` with the correct size and then with the giant size (8 bytes). `gdb` interpreted 8 bytes of memory starting at the address of `b` as a double getting the wrong exponent and fraction. The use of the count field is quite useful for dumping memory.

DAFTAR PUSTAKA

[1] Seyfarth R. 2012. Introduction to 64 Bit Intel Assembly Language Programming for Linux. CreateSpace

[2] Irvine K. 2011. Assembly Language for X86 Processors. Pearson Education, Limited [Online]. Available: http://books.google.co.id/books?id=0k20RAAACAAJ

[3] Cavanagh J. 2013, X86 Assembly Language and C Fundamentals. CRC Press, 2013.

[4] http://www.tutorialspoint.com/assembly_programming