Assembly

Pemrograman Bahasa Assembly

Buku ini merupakan suatu referensi maupun tutorial untuk mendalami bahasa assembly. Organisasi dan Arsitektur Komputer merupakan prasyarat utema yang harus pernah di ambil sebagai matakuliah dasar untuk dapat lebih cepat mengerti bahasa assembly.

Referensi dan Tutorial

Tutorial Assembler

- NUMBERING SYSTEMS
- BAGIAN 1 BAHASA ASSEMBLY
- BAGIAN 2 MEMORY ACCESS
- BAGIAN 3 VARIABLES
- BAGIAN 4 INTERRUPTS
- BAGIAN 5 LIBRARY OF COMMON FUNCTIONS - EMU8086.INC
- BAGIAN 6 INSTRUKSI ARITHMETIC AND LOGIC
- BAGIAN 7: KONTROL ALUR PROGRAM
- BAGIAN 8: PROCEDURES
- BAGIAN 9: STACK
- BAGIAN 10: MACROS
- BAGIAN 11 MEMBUAT SISTEM OPERASI SENDIRI
- BAGIAN 12 PENGENDALIAN DEVICE EXTERNAL

NUMBERING SYSTEMS

(Sistem Angka)

Ada banyak cara untuk merepresentasikan nilai numeric yang sama. Dulu orang menggunakan stik untuk menghitung, kemudian belajar bagaimana menggambarkan stik di tanah dan di kertas. Sehingga, angka 5 di gambarkan sebagai : **| | | | |** (untuk 5 stik). Kemudian, penggunaan huruf romawi mulai digunakan untuk membedakan symbol dari berbagai angka yang diwakili oleh stik : **| | | |** masih berarti 3 stik, tetapi **V berarti 5** stik, dan **X** digunakan untuk mewakili **10** stik.

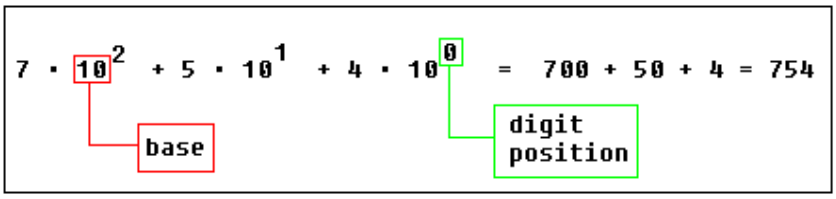
Menggunakan stik untuk menghitung merupakan ide yang brilian pada saat itu. Dan menggunakan symbol untuk mewakilinya dalah lebih baik dari ide sebelumnya.

Decimal System (Sistem Desimal)

Kebanyakan orang sekarang menggunakan representasi nilai decimal untuk mnghitung . Dalam system decimal ada 10 digit :

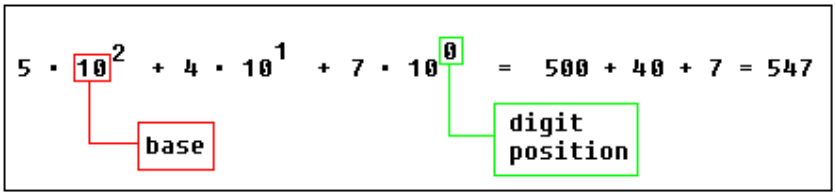
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Digit-digit tersebut dapat merepresentasikan nilai apa saja, contoh : **754**.
Nilai dibentuk dari jumlah setiap digit, lalu dikalikan dengan pangkat **base** sesuai posisi digitnya (ada 3 digit, dihitung dari nol, dalam kasusu ini base-nya adalah **10** karena ada 10 digit dalam system desimal):

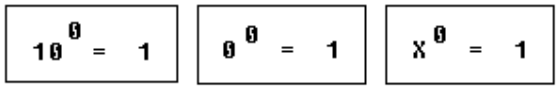


Posisi tiap digit adalah sangat penting, contohnya jika kita tempatkan "7" pada akhir digit :

547
akan terbentuk nilai yang lain dari sebelumnya :



Catatan penting: tiap angka dalam pangkat nol hasilnya 1, bahkan nol pangkat nol hasilnya juga 1:



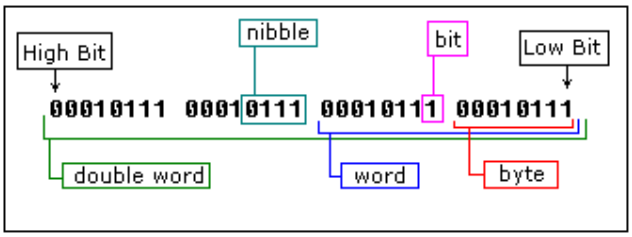
Binary System (Sistem Binary)

Komputer tidaklah secerdas manusia (mungkin belum), sangatlah mudah untuk membuat mesin elektronik dengan dua keadaan (state) : **on** and **off**, or **1** and **0**. Komputer menggunakan system biner, yaitu system dengan 2 digit:

0, 1

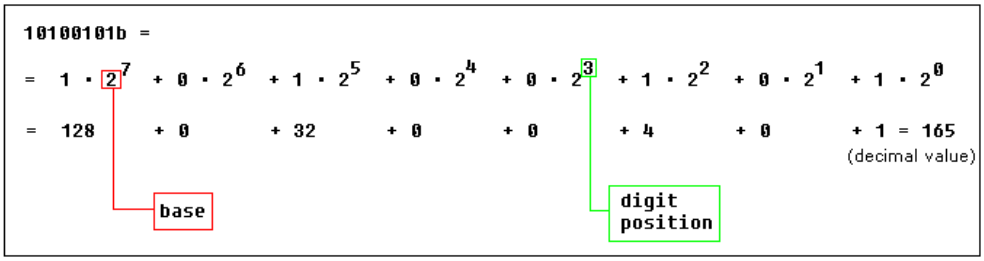
Dan dengan demikian **base-nya** adalah **2**.

Setiap angka digit dalam system biner disebut **BIT**, jika bentuk 4 bit disebut **NIBBLE**, bentuk 8 disebut **BYTE**, bentuk 2 (two) byte disebut **WORD**, bentuk two word disebut **DOUBLE WORD** (jarang dipakai):



Ada aturan untuk menambahkan **"b"** pada akhir angka biner , dengan cara ini kita dapat menentukan bahwa 101b adalah angka biner dengan nilai decimal dari 5.

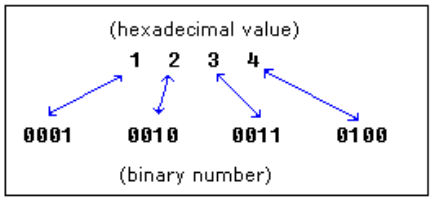
Angka biner **10100101b** sama dengan nilai decimal dari 165:



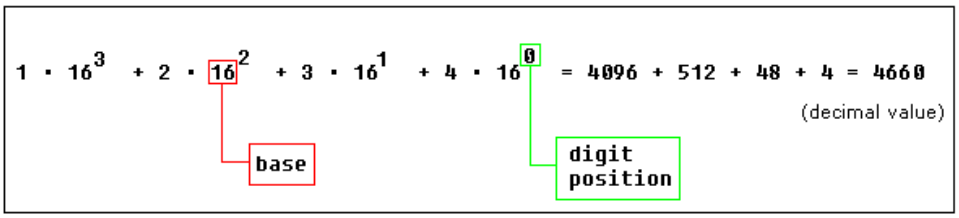
Hexadecimal System (Sistem Hexadesimal)

Sistem Hexadecimal menggunakan 16 digit:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
Dan dengan demikian **base-nya** adalah **16**.
Angka Hexadecimal lebih simple dan mudah untuk dibaca. Juga lebih mudah untuk merubah angka dari system biner ke system hexadecimal dan sebaliknya , setiap nibble (4 bit) dapat di rubah ke digit hexadecimal digit menggunakan table berikut :

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

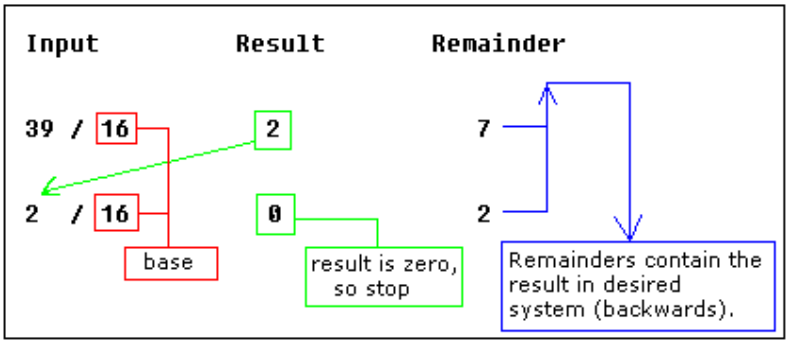


Ada aturan untuk menambahkan **"h"** diakhir angka hexadecimal , dengan cara ini kita dapat menentukan bahwa 5Fh adalah angka hexadecimal dengan nilai decimal dari 95. Kita juga dapat menambahkan **"0"** (nol) pada awal angka hexadecimal yang lalu dimulai dengan huruf (A..F), contohnya **0E120h**.
Angka hexadecimal **1234h** adalah sama dengan nilai decimal dari 4660:



Mengkonversi dari system decimal ke system lainnya.

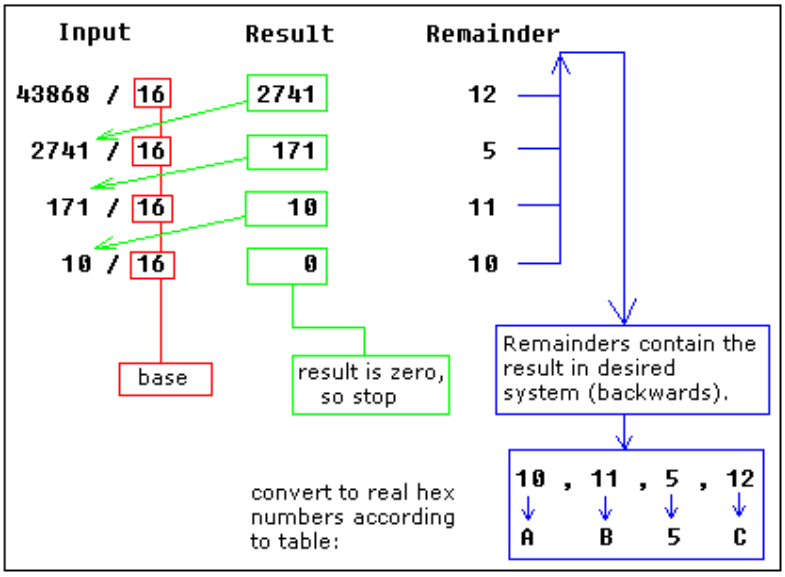
Untuk merubah/mengkonversidari system decimal ke system lainnya , diperlukan untuk membagi nilai decimal dengan **base** dari system nilai yang akan dihasilkan , untuk itu kita harus selalu ingat bahwa hasil baginya (**result)** dan juga sisa hasil baginya (**remainder)**, proses mebagian dilakukan hingga hasilnya (**result**) adalah nol. Sisa hasil bagi (**remainder)** kemudian digunakan untuk mewakili nilai dalam system. Sekarang coba rubah nilai dari **39** (base 10) ke Sistem *Hexadecimal* (base 16):



Seperti terlihat diatas kita mendapatkan angka hexadecimal: **27h**. Semua sisa hasil bagi (remainders) berada dibawah **10** , Jadi kita tidak menggunakan symbol huruf .

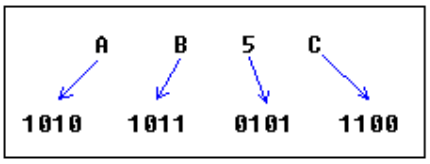
Contoh lainnya:

rubah **43868** ke bentuk hexadecimal:



Hasilnya adalah **0AB5Ch**, kita gunakan **tabel diatas** untuk merubah sisa hasil bagi yang lebih dari **9** ke symbol huruf yang sesuai .

Dengan menggunakan prinsip yang sama kita dapat merupah bentuk biner (gunakan **2** sebagai pembagi), atau merubah ke angka hexadecimal , dan kemudian merubahnya ke angka biner menggunakan **tabel diatas**:

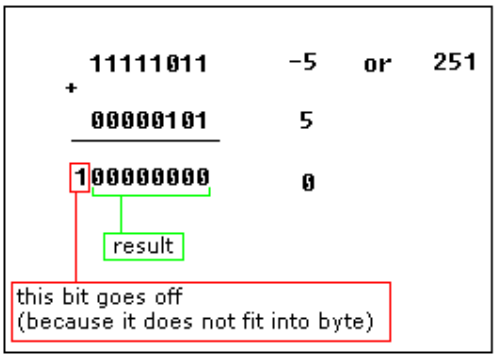


Seperti terlihat kita dapatkan angka biner : **10101011010111100b**

Signed Numbers (Angka Bertanda)

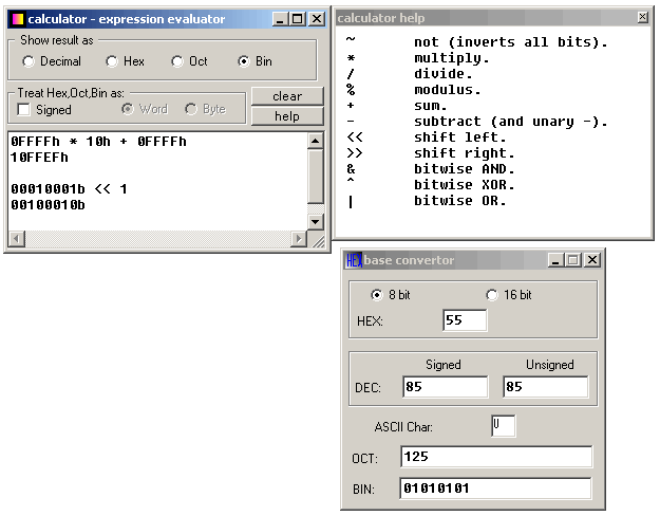
Tidak ada cara untuk meyakinkan bahwa byte hexadecimal **0FFh** adalah positive atau negative, hal tersebut dapat direpresentasikan dengan nilai decimal "**255**"dan "**- 1**".
8 bit dapat digunakan untuk membuat kombinasi sebanyak **256** (termasuk nol), sehingga kita dapat dengan mudah menduga bahwa kombinasi **128** pertama (**0..127**) akan merepresentasikan angka positive dan kombinasi **128** kemudian (**128..256**) akan merepresentasikan angka negative. Dalam rangka untuk mendapatkan "**- 5**", kita harus mengurangi **5** dari kombinasi angka (**256**),sehingga didapatkan: **256 - 5 = 251**.

Menggunakan cara yang susah ini utnuk merepresnetasikan angka negative memiliki suatu arti, dalam matematika jika kita menambahkan "**-5**" dengan "**5**" kita dapatkan nol. Hal ini terjadi jika prosesor menambahkan dua byte **5** and **251**, hasilnya adalah lebih dari **255**, karena terjadi overflow pada processor, sehingga didapatkan angka nol!



Jika kombinasi **128..256** menggunakan selalu **1** sebagai high bit , Maka mungkin dapat digunakan untuk menentukan tanda dari angka tersebut . Dengan cara yang sama dapat digunakan **words** (nilai 16 bit), 16 bit menghasilkan kombinasi sebesar **65536** , 32768 kombinasi pertama (**0..32767**) digunakan untuk merepresentasikan angka positive , dan 32768 kombinasi lainnya (**32767..65535**) mewakili angka negative. Ada cara yang mudah dalam **emu8086** untuk merubah angka , dan membuat perhitungan dari tiap ekspresi numeric, yang kita perlukan disini adalah klik menu **Math**:

Base converter mengijinkan kita untuk mengkonversi angka dari system nilai apa saja ke system nilai lainnya. Ketikan nilai di text-box, dan nilai akan secara otomatis di rubah ke semua system nilai. Kita dapat bekerja baik dalam nilai **8 bit** dan **16 bit**.



Multi base calculator dapat digunakan untuk membuat kalkulasi diantara angka yang mempunyai system nilai yang berbeda dan mengkonversinya dari satu system ke system lainnya. Ketip ekspresi dan tekan enter , hasilnya akan muncul dalam system nilai yang di pilih. Kita dapat bekerja dengan nilai hingga **32 bit**. Jika tanda **Signed** di check mengasumsikan bahwa semua nilai (kecuali decimal dan double word) diperlakukan sebagai **signed**. Double word selalu diperlakukan sebagai suatu nilai signed, jadi **0FFFFFFFFh** adalah dikonversi menjadi **-1**.

Contoh kita ingin menghitung: 0FFFFh * 10h + 0FFFFh (batas lokasi memori maximum yang dapat diakses oleh CPU 8086). Jika kita check **Signed** dan **Word** kita akan dapatkan -17 (karena dievaluasi sebagai (-1) * 16 + (-1) . Untuk membuat kalkulasi dengan nilai unsigned, jangan check tanda **Signed** sehingga kita akan evaluasi menjadi 65535 * 16 + 65535 dan kita seharusnya mendapatkan 1114095. Kita juga dapat mengunakan **base converter** untuk menkonversi digit non-decimal ke nilai decimal signed , dan melakukan kalkulasi dengan nilai decimal.

Operasi yang mendukung adalah:

~ not (inverts semua bit).
* multiply(perkalian).
/ divide (pembagian).
% modulus(modulo).
+ sum(penjumlahan).
- subtract (dan unary -) (pengurangan).
<< shift left (penggeseran bit ke kiri).
>> shift right(penggeseran bit ke kanan).
& bitwise AND (operator relasi and).
^ bitwise XOR(operator relasi xor).
| bitwise OR (operator relasi or).

Angka biner harus mempunyai suffix "**b**", contoh: 00011011b
Angka Hexadecimalharus mempunyai suffix "**h**", dan dimulai dengan nol jika digit pertama adalah symbol huruf (A..F), contoh:
0ABCDh
Angka Octal (base 8) harus mempunyai suffix "**o**", contoh: 77o

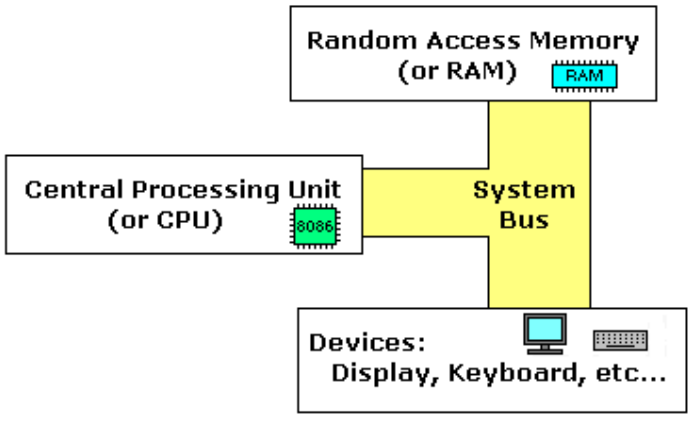
Bagian 1

Bahasa Assembly

Tutorial ini dimaksudkan bagi mereka yang tidak familiar dengan assembler sama sekali , atau yang sedikit mengetahui bahasa tersebut . Tentu saja jika anda mempunyai pengetahuan dalam bahasa pemrograman lainnya (basic, c/c++, pascal...) hal tersebut akan sangan membantu. Tetapi bahkan jika anda sudah familiar dengan assembler, tidak ada salahnya untuk mempelajari syntax dari emu8086. Dalam pelajaran ini anda juga sudah diasumsikan mempelajari system nilai seperti pada awal buku ini , jika sudah anda dapat melewati bagian pertama dari buku ini.

Apakah bahasa Assembly itu?

Bahasa assembly adalah bahasa level rendah dari bahasa pemrograman. Anda perlu mengetahui mengenai struktur computer (Arsitektur dan Sistem Komouter) dalam rangka untuk memahami sesuatunya yang akan dipelajari dalam buku ini. Secara sederhana model computer dapat digambarkan sebagai berikut :



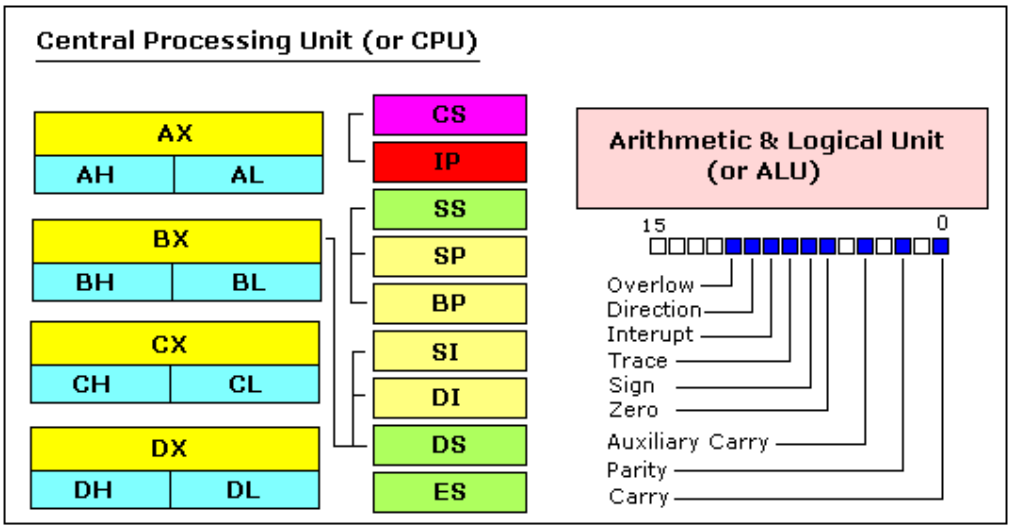
System bus (warna kuning) menghubungkan dengan bermacam-macam komponen komputer.

CPU merupakan jantung computer, hamper seluruh perhitungan dilakukan didalam **CPU**.

RAM merupakan tempat dimana program di loading untuk di

eksekusi.

CPU



GENERAL PURPOSE REGISTER

CPU 8086 mempunyai 8 general purpose register, dimana tiap register mempunyai namanya sendiri-sendiri :

- **AX** - the accumulator register (dibagi menjadi **AH / AL**).
- **BX** - the base address register (dibagi menjadi **BH / BL**).
- **CX** - the count register (dibagi menjadi **CH / CL**).
- **DX** - the data register (dibagi menjadi **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

disamping nama-nama register, dimana seorang programmer dapat menentukan penggunaannya untuk setiap general purpose register. **Tujuan utama dari register adalah untuk menyimpan angka (variable).** Ukuran register diatas adalah 16 bit, seperti **0011000000111001b** (bentuk binary, atau **12345** dalam bentuk decimal. Empat (4) general purpose register (AX, BX, CX, DX) dibuat dari dua 8 bit register yang terpisah , contoh jika AX= **0110000000111001b**, maka AH=**00110000b** dan AL=**00111001b**. Demikian juga, jika kita memodifikasi setiap 8 bit register, maka 16 bit register akan di

update, begitu juga sebaliknya. Begitu juga sama dengan 3 register lainnya, "H" berarti bagian high dan "L" berarti bagian low. Karena register berada dalam CPU, mereka lebih cepat dibandingkan dengan memory. Pengaksesan lokasi memory diperlukan penggunaan suatu system bus, jadi pengaksesan memori memakan waktu lebih banyak disbanding pengaksesan register. Pengaksesan data dalam register biasanya tidan membutuhkan waktu. Oleh karena itu kita harus menyimpan variable dalam register. Kumpulan register sangat kecil dan hamper semua register mempunyai kegunaan yang khusus yang penggunaannya dibatasi sebagaimana suatu variable , tetapi register masih merupakan tempat yang baik untuk menyimpan data sementara untuk suatu perhitungan .

SEGMENT REGISTER

- **CS** – menunjuk pada segmen yang berisi program sekarang (current program).
- **DS** – biasanya menunjuk pada segmen dimana variable didefinisikan.
- **ES** – register segmen extra, tergantung programmer mendefinisikan penggunaanya.
- **SS** – menunjuk pada segmen yang berisi stack.

Walaupun mungkin menyimpan data apapun dalam segmen register, tapi hal ini tidaklah dianjurkan. Segmen register mempunyai tujuan yang sangat khusus, yaitu menunjuk blok memori yang dapat diakses. Segment register bekerja bersama dengan general purpose register untuk mengakses setiap nilai memory. Contohnya, jika kita akan mengakses memori pada physical address **12345h** (hexadecimal), kita perlu mengeset **DS = 1230h** dan **SI = 0045h**. Ini bagus karena dengan cara ini kita dapat mengakses lebih banyak memory daripada dengan register tunggal yang terbatas pada nilai 16 bit. CPU membuat kalkulasi dari physical address dengan mengalikan segment register dengan 10h dan menambahkan general purpose register (1230h * 10h + 45h = 12345h):

```
 12300
+ 0045
-------
 12345
```

Address yang dibentuk dengan 2 register disebut **effective address**. Defaultnya register **BX, SI** dan **DI** bekerja dengan segmen register **DS, BP** dan **SP** bekerja dengan segmen register **SS**. General purpose register lainnya tidak dapat dibentuk dari

suatu effective address!. Demikian juga , walaupun **BX** dapat dibentuk dari effective address, **BH** and **BL** tidak dapat melakukannya.

SPECIAL PURPOSE REGISTERS

- **IP** - the instruction pointer.
- **flags register** – menentukan keadaan sekarang (current state) dari microprocessor.

IP register selalu bekerja bersama dengan segmen register **CS** dan menunjuk pada instruksi yang sedang dieksekusi. **flags register** dimodifikasi secara otomatis oleh CPU setelah operasi matematika, hal ini dapat menentukan tipe yang dihasilkan, dan untuk menentukan kondisi uitnuk mentransfer control ke bagian program. Seacra umum kita tidak dapat megakses register ini secara langsung, cara yang dapat dilakukan untukmengkasesnya adalah dengan mengakses AX dan general register lainnya, ada kemungkinan merubah nilai dari system register dengan menggunakan beberapa trik yang akan dipelajari nanti.

Bagian 2

MEMORY ACCESS

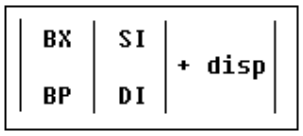
Untuk mengakses memory kita dapat menggunakan 4 register : **BX, SI, DI, BP**. register-register diatas dapat dikombinasikan didalam symbol tanda kurung kotak **[ ]** , kita bias mendapatkan lokasi memori yang berbeda. Kombinasi ini didukung dengan mode pengalamatan (addressing modes) sebagai berikut :

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

d8 –merupakan penggantian nilai yang terdekat untuk 8 bit signed (contoh: 22, 55h, -1, etc...)

d16 – merupakan penggantian nilai yang terdekat untuk 16 bit signed (contoh: 300, 5517h, -259, etc...).

Penggantian dapat menjadi nilai yang terdekat atau offset dari variable atau keduanya. Jika ada beberapa nilai, assembler mengevaluasi semua nilai tersebut dan mengkalkulasi sebagai nilai terdekat tunggal. Pengantian di dalam atau diluar symbol **[]** , maka assembler meng-generates kode mesin yang sama dengan kedua cara tersebut. Penggantian nilai **signed** bertanda), dapat dilakukan baik pada tanda positive atau negative. Umumnya compiler memperhatikan perbedaan antara **d8** and **d16**, dan men-generate kode mesin yang diperlukan. contoh, diasumsikan **DS = 100**, **BX = 30**, **SI = 70**. mode pengalamatannya adalah : **[BX + SI] + 25** yaitu dikalkulasi oleh processor ke physical address: **100 * 16 + 30 + 70 + 25 = 1725**. Defaultnya segmen register **DS** adalah digunakan untuk semua mode kecuali dengan register **BP**, untuk hal tersebut segmen register **SS** digunakan. Jadi ada cara mudah untuk mengingat semua kemungkinan tersebut dengan menggunakan bagan di bawah ini:



Kita dapat membentuk semua kombinasi yang valid dengan mengambil hanya satu item tiap kolom atau melewati kolom tersebut dengan tidak mengambil darinya . seperti yang kita lihat **BX** dan **BP** tidak pernah bersama-samar. **SI** dan **DI** demikian juga, contoh disini merupakan mode pengalamatan yang valid : **[BX+5]** , **[BX+SI]** , **[DI+BX-4]**, nilai dalam segment register (CS, DS, SS, ES) disebut **segment**, dan nilai dalam purpose register (BX, SI, DI, BP) disebut **offset**. Saat DS berisi nilai **1234h** dan SI berisi nilai **7890h** hal terbut dapat di tulis sebagai **1234:7890**. Physical address akan menjadi 1234h * 10h + 7890h = 19BD0h. Jika nol ditambahkan ke angka decimal lalu di kalikan 10, maka **10h = 16**, jasi jika nol ditembahkan pada nilai hexadecimal, hal tersebut dikalikan dengan 16, contoh:

7h = 7

70h = 7*16=112

Hal tersebut dapat dikatakan bahwa dalam pandangan compiler terhadap tipe data, prefix yang digunakan adalah :
byte ptr – untuk byte.
word ptr – untuk word (2 bytes).
contoh:

```
byte ptr [BX]     ; mengakses byte .
   atau
word ptr [BX]     ; mengakses word .
```

assembler mendukung prefix seperti dibawah :

b. - untuk **byte ptr**

w. - untuk **word ptr**

Dalam banyak kasus dapat mengkalkulasi tipe data secara otomatis.

INSTRUKSI MOV

- mengutip **operan ke dua** (source) ke **operand pertama** (destination).
- Operan source dapat suatu nilai terdekat, general-purpose register atau lokasi memory.
- Register destination dapat suatu general-purpose register, atau lokasi memory.
- kedua operand harus mempunyai ukuran yang sama, baik byte atau word.

these types of operands are supported:

MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

for segment registers only these types of **MOV** are supported:

MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
SREG: DS, ES, SS, and only as second operand: CS.

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

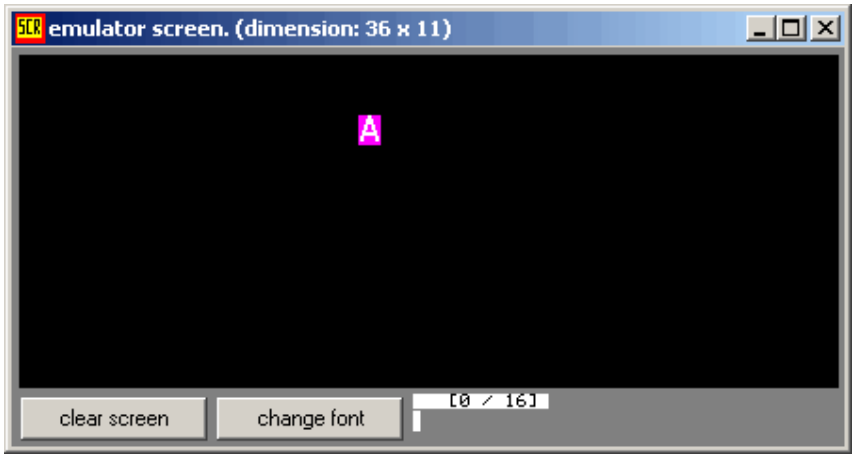
memory: [BX], [BX+SI+7], variable, etc...

Instruksi **MOV** tidak dapat digunakan untuk mengeset nilai register **CS** dan **IP**.

Contoh program mendemonstrasikan penggunaan instruksi **MOV**:

```
ORG 100h          ; directive yg diperlukan utk 1 segment .com program.
MOV AX, 0B800h    ; set AX ke nilai hex dr B800h.
MOV DS, AX        ; copy nilai AX ke DS.
MOV CL, 'A'       ; set CL ke kode ASCII  'A', yaitu 41h.
MOV CH, 1101_1111b ; set CH ke nilai binary.
MOV BX, 15Eh      ; set BX ke 15Eh.
MOV [BX], CX      ; copy isi   CX ke memory  pada alamat  B800:015E
RET               ; kembali ke os.
```

Kita dapat **copy & paste** program di atas ke editor emu8086 , dan tekan tombol [**Compile and Emulate**] (atau **F5**). Window emulator akan terbuka dengan program yang di loaded, lalu click tombol [**Single Step**] dan amati nilai register. Tanda "**;**" adalah digunakan untuk membuat komentar program dan akan diabaikan oleh compiler . Keluaran dari program diatas seharusnya seberti di bawah ini:



Sebenarnya program diatas menulis secara langsung ke video memory, jadi kita dapat melihat bahwa **MOV** merupakan instruksi yang powerful.

Bagian 3

VARIABLES

Variabel adalah suatu lokasi di memori. Bagi programmer akan lebih mudah memiliki suatu nilai yang disimpan pada variable dengan nama "**var1**" dari pada pada alamat 5A73:235B, khususnya kalau kita punya 10 atau lebih variabel. Dalam emu8086, compiler mendukung dua tipe variabel: **BYTE** dan **WORD**.

Syntax deklarasi variabel:

name **DB** *value*

name **DW** *value*

DB – kepanjangan dari Define Byte.
DW – kepanjangan dari Define Word.

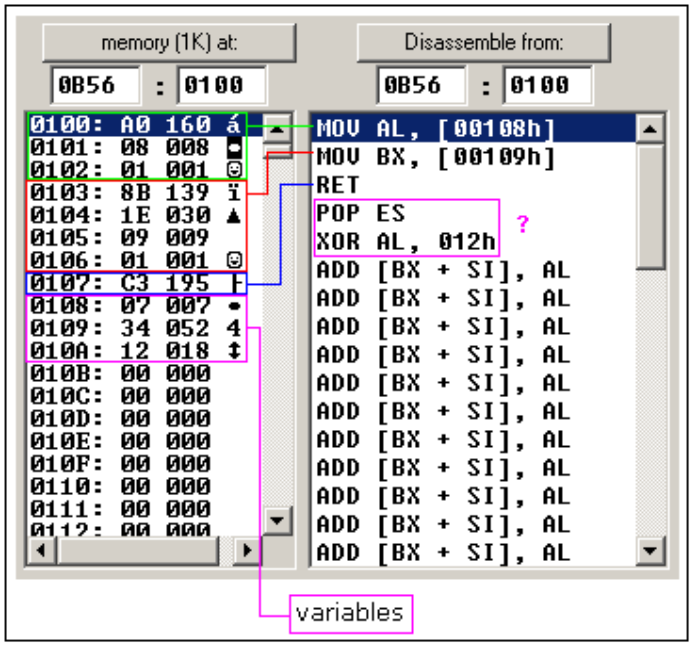
name – kombinasi dari alphanumerik, diawali dengan huruf. Mungkin juga tidak bernama (maka hanya punya alamat saja).

value – berupa nilai numeric yang didukung oleh system nilai(hexadecimal, binary, atau decimal), atau simbol "**?**" untuk variable yang tidak diinisialisasikan.

Seperti pada instruksi **MOV** yaitu digunaka untuk mengutip nilai dari source ke destination. Contoh lain instruksi **MOV** :

```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; program berhenti.

VAR1 DB 7
var2 DW 1234h
```

Tulis kode diatas pada editor emu8086 dan tekan **F5** untuk compile dan memuatkan ke emulator, hasilnya seharusnya :



Seperti yang terlihat seperti contoh ini, kecuali variable diganti dengan lokasi actual di memori. Saat compiler membuat kode mesin, hal tersebut secara otomatis mengganti semua nama variable dengan **offsets**. Defaultnua segmen di loaded di register **DS** (jika file .**COM** nilai yang di loaded pada register **DS** di set sama dengan nilai pada register **CS** - code segment). Dalam baris pertama pada memori adalah sebuah **offset**, baris kedua adalah suatu nilai **hexadecimal** , baris ketiga adalah nilai **decimal** , dan yang terakhir adalah nilai karakter **ASCII** . Kompiler tidan case sensitive, jadi "**VAR1**" and "**var1**" mengacu pada variable yang sama. Offset dari **VAR1** adalah **0108h**, dan alamat lengkapnya adalah **0B56:0108**. Offset dari **var2** adalah **0109h**, alamat lengkapnya adalah **0B56:0109**, Variabel ini adalah bertipe **WORD** sehingga perlu **2 BYTES**. Hal tersebut diasumsikan bahwa low byte (byte rendah) disimpan pada alamat yang rendah (lower address), sehingga **34h** berada sebelum **12h**. Kita dapat melihat bahwa ada suatu instruksi lainnya setelah instruksi **RET**, Ini terjadi karena disassembler tidak tahu dimana data dimulai , dia hanya memproses nilai dalam memori dan memahami sebagai instruksi 8086 yang sah (kita akan pelajari nanti). Kita juga bahkan dapat menulis program yang sama menggunakan hanya direktive **DB**:

```
ORG 100h  ; hanya suatu directive utk
          ; membuat  file .com sederhana .

DB 0A0h
DB 08h
DB 01h

DB 8Bh
DB 1Eh
DB 09h
DB 01h

DB 0C3h

DB 7

DB 34h
DB 12h
```

Tulis program diatas dalam editor emu8086 , dan tekan **F5** untuk compile dan memuat dalam emulator. Hasilnya seharusnya sama dengan kode disassembled,dan berfungsi sama! Seperti yang kita duga, compiler hanya mengkonversi source code ke suatu byte, ini disebut **machine code**, processor hanya mengerti **machine code** dan menjalankannya. **ORG 100h** adalah directive compiler (yang mengatakan pada compiler bagaimana menangani source code). Directive ini sangat penting saat kita bekerja dengan variable . Direktive tersebut mengatakan pada compiler bahwa file executable akan di muat pada **offset** 100h (256 bytes), jadi compiler harus menghitung alamat yang benar untuk semua variable ketika dia mengganti nama variable dengan nama **offset-nya**. Directives tidak pernah dikonversi ke **machine code** . Mengapa file executable di loaded pada **offset 100h**? OS menjaga beberapa data tentang program dalam 256 bytes pertama dari **CS** (code segment), seperti parameter command line dan sebagainya. Walaupun benar bagi hanya file .**COM**, file **EXE** di loaded pada offset **0000**, dan umumnya menggunakan segmen khusus untuk variable . Kita bicarakan file exe tidak disini.

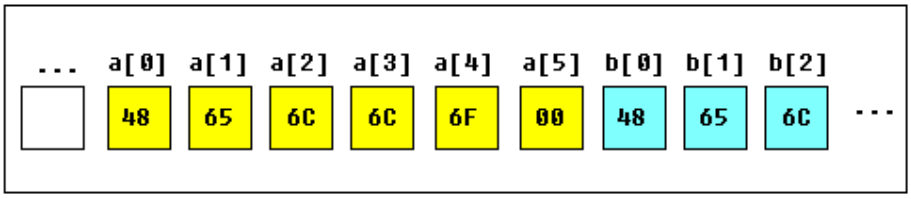
Arrays

Array dapat dilihat sebagai suatu rantai variable . Text string merupakan contoh dari array byte, tiap karakter diwakili sebuah nilai kode ASCII (0..255). contohnya :

a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h

b DB 'Hello', 0

b adalah benar-benar suatu kopi dari larik *a*, Ketika compiler melihat di dalam tanda petik, secara otomatis akan mengkonversinya ke dalam byte . Bagan dibawah ini menunjukan bagian memori dimana larik dideklarasikan :



Kita dapat mengakses nilai tiap elemen dalam array dengan menggunakan kurung kotak, contohnya :

MOV AL, a[3]

Kita juga dapat menggunakan memory index register **BX, SI, DI, BP**, contohnya:

MOV SI, 3

MOV AL, a[SI]

jika kita perlu mendeklarasikan array yang besar dapat kita gunakan operator **DUP**. Syntax **DUP**:

number DUP (value(s))

number – banyaknya duplikasi yang dibuat (nilai konstan).

value – ekspresi yang akan diduplikasi oleh DUP .

contoh:
c DB 5 DUP(9)

atau:
c DB 9, 9, 9, 9, 9

contoh lainnya:

d DB 5 DUP(1, 2)

atau :

d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2

tentu saja , kita dapat mengunakan **DW** sebagai ganti dari **DB** jika diperlukan untuk menjaga nilai yang lebih besar dari 255, atau lebih kecil dari -128. **DW** tidak dapat untuk mendeklarasikan strings.

Mendapatkan Alamat Suatu Variabel

Ada suatu instruksi yaitu **LEA** (Load Effective Address) merupakan alternative dari operator **OFFSET** . Baik **OFFSET** dan **LEA** dapat digunakan untuk mendapatkan offset address dari suatu variabel . **LEA** malah lebih powerful karena dia juga mendapatkan alamat dari *index variabelnya*. Mendapatkan alamat dari suatu variable dapat sangat berguna dalam suatu situasi, contohnya saat kita akan mengirim parameter ke suatu prosedur, contoh :

```
ORG 100h

MOV    AL, VAR1 ; check nilai dari
                ; VAR1 dg
                ;memindahkannya ke  AL.
LEA    BX, VAR1  ; ambil alamat
                ;VAR1 di  BX.

MOV    BYTE PTR [BX], 44h    ; modifikasi
                        ;isi dari VAR1.
MOV    AL, VAR1          ; check nilai dari
                ;VAR1 dg memindahkannya
                ; ke  AL.

RET
VAR1   DB  22h

END
```

Contoh lainnya penggunaan **OFFSET** sebagai ganti **LEA**:

```
ORG 100h

MOV    AL, VAR1          ; check nilai
          ; VAR1 dg memindahkannya ke AL.
```

```
MOV   BX, OFFSET VAR1      ; ambil almt
        ;  VAR1 di  BX.
MOV   BYTE PTR [BX], 44h   ; modifikasi
        ;isi  VAR1.
MOV   AL, VAR1             ; check nilai
        ;VAR1 dg memindahkannya ke AL.
RET
VAR1  DB  22h
END
```

Kedua contoh diatas mempunyai fungsi yang sama. perhatikan:
LEA BX, VAR1

MOV BX, OFFSET VAR1

akan dikompile ke dalam kode mesin yang sama dengan :

MOV BX, num

num adalah nilai 16 bit dari offset variable.

Hanya register ini yang dapat digunakan dalam kurung kotak (seperti pointer memory): **BX, SI, DI, BP**!

Constant (Konstanta)

Konstan hamper sama dengan variable tetapi keberadaanya hanya sampai program di kompilasi (assembled). Setelah definisi sebuah konstan, nilai yang diassign tidak dapat dirubah. Untuk mendeklarasikannya dengan directive **EQU**:

name EQU < any expression >

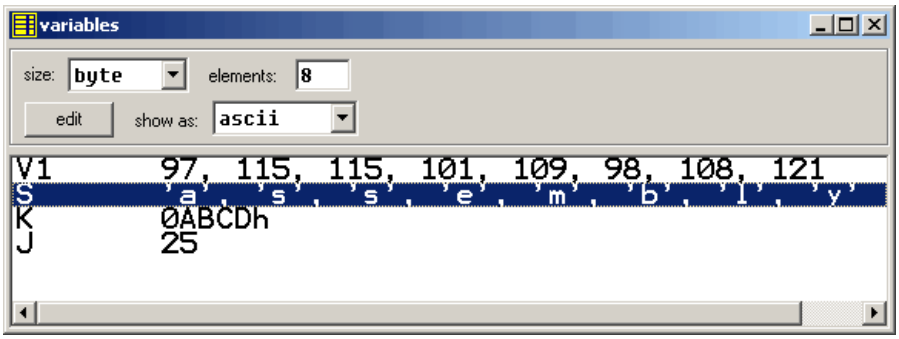
contoh:

```
k EQU 5

MOV AX, k
```

dengan cara lain:

```
MOV AX, 5
```

Kita dapat melihat variable sementara program kita sedang dieksekusi dengan memilih "**Variables**" dari menu "**View**" di emu8086.



untuk melihat array kita harus klik pada variable dan mengeset property **Elements** kepada ukuran array-nya . Dalam bahasa assembly tidak ada batasan tipe datanya, jadi tiap variable dapat direpresentasikan sebagai suatu array . Variabel dapat dilihat dalam system nilai :

- **HEX** - hexadecimal (base 16).
- **BIN** - binary (base 2).
- **OCT** - octal (base 8).
- **SIGNED** - signed decimal (base 10).
- **UNSIGNED** - unsigned decimal (base 10).
- **CHAR** - ASCII char code (256 symbol, beberapa tidak kelihatan).

Kita dapat mengedit nilai variable saat program kita sedang berjalan , double click pada variable tersbut, atau pilih dan klik tombol **Edit** . Adalah hal yang mungkin untuk memasukkan angka pada setiap system, nilai hexadecimalharus mempunyai suffix "**h**", binary "**b**" , octal "**o**" , nilai decimal tidak memerlukan suffix. String dapat di masukkan dengan cara sebagai berikut : **'hello world', 0** (string ini diakhiri dengan nol). Arrays dapat dimasukkan dengan :

1, 2, 3, 4, 5

(array dapat berupa bytes atau word, tergantung dari **BYTE** atau **WORD** yang dipilih saat mengedit variable).

Ekspresi secara otomatis di konversi, contoh, saat ekspresi dimasukkan: **5 + 2** akan dikonversi ke **7** dst...

BAGIAN 4

INTERRUPTS

Interrupt dapat dipandang sebagai sejumlah fungsi. Fungsi-fungsi ini memudahkan pemrograman, bahkan menulis kode untuk mencetak karakter, kita dapat dengan mudah memanggil interrupt dan fungsi tersebut akan melakukannya untuk kita. Ada juga fungsi interrupt yang berkerja dengan disk drive dan hardware lainnya. Pada saat kita memanggil suatu fungsi hal itu dinamakan **software interrupts**. Interrupt juga dipicu oleh hardware lainnya, ini disebut **hardware interrupts**. Saat ini kita akan membahas pada **software interrupts**. Untuk membuat **software interrupt** ada suatu instruksi yaitu **INT**, dengan sintak:

INT value

Dimana **value** berupa angka antara 0 - 255 (atau 0 - 0FFh), umumnya kita menggunakan angka hexadecimal. Anda mungkin berpikir bahwa hanya ada 256 function, tapi sebenarnya kurang tepat . Tiap interrupt masih punya sub-function. Untuk menentuka suatu sub-function register **AH** harus diset sebelum pemanggilan interrupt. Tiap interrupt mempunyai hingga 256 sub-function (jadi kita dapatkan 256 * 256 = 65536 function). Secara umum register **AH** yang digunakan, tetapi suatu saat register lain juga dipakai. Pada umumnya register lain digunakan untuk mengirim parameter dan data ke sub-function. Contoh berikut ini menggunakan **INT 10h** sub-function **0Eh** untuk mencetak pesan "Hello!". Fungsi ini menampilkan karakter di layar, keuntunganThis functions displays a character on the screen.

```
ORG    100h ; directive utk membuat file .com.
; sub-function yang dipakai
; tidak memodifikasi register AH pd saat
; return, jadi hanya di set sekali saja.
MOV    AH, 0Eh    ; pilih sub-function.
; INT 10h / 0Eh sub-function
; menerima kode ASCII sbg
;Karakter yg akan dicetak
;di register AL .
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MOV    AL, 'H'    ; ASCII code: 72
INT    10h        ; cetak!
MOV    AL, 'e'    ; ASCII code: 101
INT    10h        ; cetak!
```

```
MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; cetak!
MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; cetak!
MOV    AL, 'o'    ; ASCII code: 111
INT    10h        ; cetak!
MOV    AL, '!'    ; ASCII code: 33
INT    10h        ; cetak!
RET               ; returns ke OS.
```

[**Compile and Emulate**] lalu, jalankan!

BAGIAN 5

PUSTAKA FUNSI UMUM (EMU8086.INC)

Untuk membuat pemrogram lebih mudah, ada beberapa fungsi umum yang dapat disertakan dalam program kita. Untuk membuat program kita menggunakan fungsi yang didefinisikan di file lainnya kita harus menggunakan directive **INCLUDE** diikuti dengan nama file yang diacu. Kompiler secara otomatis mencari file tersebut di folder yang sama dimana file sumber diletakkan, dan jika tidak ditemukan maka compiler akan mencari di forlder **Inc**. Saat ini anda mungkin belum faham benar isi dari **emu8086.inc** (terletak di folder **Inc**), tapi tidak masalah, karena yang kita butuhkan adalah apa yang dapat dikerjakannya. Untuk memakai fungsi dalam **emu8086.inc** kita harus memiliki baris dalam kode didalam source seperti ini :

```
include 'emu8086.inc'
```

emu8086.inc mendefinisikan banyak makro sebagai berikut:

- **PUTC char** - macro dengan 1 paramete, mencetak sebuah karakter ASCII pada posisi kursor sekarang.
- **GOTOXY col, row** - macro dengan 2 parameters, mengeset posisi kursor.
- **PRINT string** - macro dengan 1 parameter, mencetak sebuah string.
- **PRINTN string** - macro dengan 1 parameter, mencetak string. Sama dengan PRINT tetapi otomatis menambahkan "carriage return" di akhir string.
- **CURSOROFF** – mematikan kursor text.
- **CURSORON** - menghidupkan kursor text.

Untuk menggunakan makro diatas, tinggal ketik nama makro dan parameter yang dibutuhkan , contohnya:

```
include emu8086.inc

ORG    100h
```

```
PRINT 'Hello World!'

GOTOXY 10, 5

PUTC 65         ; 65 - kode ASCII untuk 'A'
PUTC 'B'

RET             ; return ke OS.
END             ; directive utk menghentikan
                ; kompiler.
```

Saat compiler memproses kode kita, dia mencari file **emu8086.inc** untuk makro yang dideklarasikan dan menggantikannya dengan nama makro dalam kode yang sebenarnya. Umumnya makro relative bagian yang kecil dalam kode, keseringan menggunakan makro menyebabkan pengeksekusian kita menjadi semakin besar (penggunaan procedure lebih baik untuk optimasi ukuran).

emu8086.inc juga mendefinisikan procedure sebagai berikut :

- **PRINT_STRING** – procedure utnuk mencetak null terminated **string** pada posisi kursor saat ini , menerima alamat string di dalam register **DS:SI**. Untuk menggunakannya deklarasikan: **DEFINE_PRINT_STRING** sebelum directive **END**.

- **PTHIS** – procedure untuk mencetak null terminated **string** pada posisi kursor saat ini (seperti PRINT_STRING), tetapi menerima alamat string dari stack. ZERO TERMINATED string harus di definisikan setelah instruksi CALL, contoh :
  CALL PTHIS
  db 'Hello World!', 0
  Untuk menggunakannya deklarasikan: **DEFINE_PTHIS** sebelum direktive **END**.

- **GET_STRING** – procedure untuk mendapatkan input null terminated **string** dari user, string yang diterima ditulis dalam buffer di register **DS:DI**, ukuran buffer berada di **DX**. Prosedur ini menghentikan input saat tombol 'Enter' ditekan. Cara menggunakannya: **DEFINE_GET_STRING** sebelum direktive **END**.

- **CLEAR_SCREEN** – procesedur membersihkan layar, (sebenarnya hanya menggulung layar saja satu window),

dan mengeset posisi kursor di atasnya. Cara pemakaiannya: **DEFINE_CLEAR_SCREEN** sebelum direktive **END**.

- **SCAN_NUM** – procedure untuk menerima input berupa **angka** multi-digit **SIGNED** (bertanda) dari keyboard, dan menyimpan hasilnya di register **CX**. Cara memakainya : **DEFINE_SCAN_NUM** sebelum direktive **END**.

- **PRINT_NUM** – prosedur mencetak **angka signed** dalam register **AX**. Pemakaiannya: **DEFINE_PRINT_NUM** dan **DEFINE_PRINT_NUM_UNS** sebelum direktive **END**.

- **PRINT_NUM_UNS** – prosedur mencetak **angka unsigned** dalam register **AX**. Pemakainnya: **DEFINE_PRINT_NUM_UNS** sebelum direktive **END**.

Untuk menggunakan semua prosedur diatas kita harus mendeklarasikan fungsi di bawah file sumber (sebelum direktive **END**), dan kemudian menggunakan instruksi **CALL** diikuti dengan nama prosedurnya, contohnya :

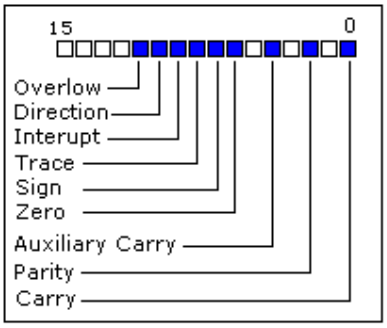
```
include 'emu8086.inc'
ORG    100h

LEA    SI, msg1       ; Tanya angka yg di input
CALL   print_string   ;
CALL   scan_num       ; ambil angka di CX.

MOV    AX, CX         ; copy angka ke AX.

; cetak string berikut:
CALL   pthis
DB  13, 10, 'Anda memasukkan: ', 0

CALL   print_num      ; cetak angka di AX.

RET                   ; return ke OS.

msg1   DB  Masukan  sebuah Agka: ', 0

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ;diperlukan utk
                      ;print_num.
DEFINE_PTHIS

END                   ; directive utk menghentikan
                      ;kompiler.
```

Pertama-tama compiler memproses deklarasi (ada makro biasa yang di ekspan menjadi prosedur). Saat compiler menerima instruksi **CALL** dia mengganti nama prosedur dengan alamat dari kode dimana prosedur dideklarasikan. Saat instruksi **CALL** mengeksekusi control maka prosedur dipindahkan. Hal ini sangat berguna, karena jika kita memanggil prosedur 100 kali dalam kode program kita masih mempunyai ukura program yang kecil .Tampaknya agak membingungkan bukan ? Tidak masalah hanya masalah waktu saja m=yang menuntuk kita untuk memahaminya prinsip dasar ini.

BAGIAN 6

INSTRUKSU LOGIK DAN ARITMETIK

Setiap instruksi Arithmetic dan logika berpengaruh pada register status pada prosesor (atau **Flag**)



Dalam register 16 bit, tiap bit disebut **flag** dan dapat bernilai **1** atau **0**.

- **Carry Flag (CF)** - flag ini diset ke **1** jika ada **unsigned overflow**. Contoh saat kita menambah byte **255 + 1** (hasilnya tidak di jangkauan 0...255). Saat tidak ada **overflow** flag ini diset ke **0**.

- **Zero Flag (ZF)** – diset ke **1** jika hasilnya **zero**. Untuk hasil selain zero, flag diset ke **0**.

- **Sign Flag (SF)** – di set ke **1** jika hasilnya **negative**. Saat hasilnya **positive** diset ke **0**. Sebenarnya flag ini mengambil nilai dari msb (most significant bit).

- **Overflow Flag (OF)** – di set ke **1** saat ada **signed overflow**. contohnya, jika kita menambah byte **100 + 50** (hasilnya diluar -128...127).

- **Parity Flag (PF)** – di set ke **1** jika ada angka **genap** dari satu bit dalam hasilnya , dan di set ke **0** jika ada angka **ganjil**. Jika hasilnya berupa word hanya 8 low bit yang dianalisa.

- **Auxiliary Flag (AF)** – di set ke **1** jika ada **unsigned overflow** untuk low nibble (4 bit).

- **Interrupt enable Flag (IF)** – Jika flag diset **1** CPU mendaptkan interrupt dari device eksternal.

- **Direction Flag (DF)** – Flag ini digunakan oleh beberapa instruksi untuk memproses rantai data, jika flag di set ke **0** – proses diselesaikan ke depan, Jika flag bernilai **1** proses diselesaikan ke belakang.

Ada 3 kelompok instruksi.

Kelompok pertama: ADD, SUB,CMP, AND, TEST, OR, XOR

Tipe operand yang mendukungnya :

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Setelah operasi di antara operan, hasilnya selalu disimpan di operan pertama . Instruksi **CMP** dan **TEST** berakibat hanya pada flag dan tidak menyimpan hasilnya (instruksi ini digunakan utnuk membuat keputusan selama eksekusi program).

Instruksi-instruksi yang berpengaruh hanya pada flag : **CF**, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **ADD** – menambah operand ke dua ke operand pertama.
- **SUB** – Mengurangi operan kedua dari operand yang pertama.
- **CMP** – Mengurangi operand kedua dari operand pertama **hanya flagnya saja**.
- **AND** - Logical AND antara semua bit dari dua operand . Aturannya :

  1 AND 1 = 1

  1 AND 0 = 0

  0 AND 1 = 0

  0 AND 0 = 0

  Kita dapatkan **1** hanya jika kedua bit bernilai **1**.

- **TEST** – Sama dengan **AND tapi hanya untuk flag saja**.
- **OR** - Logical OR antara semua bit dari dua operand. Aturannya:

1 OR 1 = 1

1 OR 0 = 1

0 OR 1 = 1

0 OR 0 = 0

Kita dapatkan **1** hanya jika **salah satu** operand bit bernilai **1**.

- **XOR** - Logical XOR (exclusive OR) antara semua bit dari dua operand. Aturannya:

  1 XOR 1 = 0

  1 XOR 0 = 1

  0 XOR 1 = 1

  0 XOR 0 = 0

  Kita dapatkan **1** hanya jika kedua operand bit-nya berbeda.

Kelompok Kedua: MUL, IMUL, DIV, IDIV

Tipe operand yang mendukungnya :

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

instruksi **MUL** dan **IMUL** mempengaruhi hanya flag:
CF, **OF**
Jika hasilnya diatas ukruan operand flag di set ke 1, jika tidak **0**.

Untuk flag **DIV** dan **IDIV** tidak didefinisikan.

- **MUL** – Perkalian Unsigned:

  Jika operand-nya **byte**:
  AX = AL * operand.

jika operand-nya **word**:
(DX AX) = AX * operand.

- **IMUL** - Perkalian Signed:

  jia operand-nya **byte**:
  AX = AL * operand.

  jika operand-nya **word**:
  (DX AX) = AX * operand.

- **DIV** – Pembagian Unsigned:

  Jika operand-nya **byte**:
  AL = AX / operand
  AH = remainder (modulus). .

  jika operand-nya **word**:
  AX = (DX AX) / operand
  DX = remainder (modulus). .

- **IDIV** – Pembagian Signed:

  jika operand-nya **byte**:
  AL = AX / operand
  AH = remainder (modulus). .

  jika operand-nya **word**:
  AX = (DX AX) / operand
  DX = remainder (modulus). .

Kelompok ketiga: INC, DEC, NOT, NEG

Tipe Operan yang mendukung:

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

instruksi INC, **DEC** mempengaruhi hanya flag:
ZF, **SF**, **OF**, **PF**, **AF**.

NOT tidak mempengaruhi flag apapun!

NEG hanya flag:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **NOT** – cadangan untuk tiap operand bit.
- **NEG** – membuat operang menjadi negatif (two's complement). Sebenarnya dicadangkan untuk tiap bit operand dan ditambahkan 1. contoh 5 menjadi -5, dan -2 menjadi 2.

BAGIAN 7

ALUR KONTROL PROGRAM

Pengendalian alur program merupakan bagian yang penting, Disini program kita dapat menentukan keputusan berdasarkan kondisi.

- **unconditional jump**

  Instruksi dasarnya adalah bahwa mengendalikan transfer dari satu titik ke titik painya didalam program yaitu instruksi **JMP,** sintaknya :

  JMP label

  deklarasikan *label* dalam program, tulis namanya dan tambahkan "**:**" setelah nama label, label berupa kompinasi karakter alphanumerik yang tidak dimulai dengan angka, contoh:

  ```
  label1:
  label2:
  a:
  ```

  Label dapat dideklarasikan pada baris terpisah atau sebelum instruksi lainnya, contoh:

  ```
  x1:
  MOV AX, 1

  x2: MOV AX, 2
  ```

  contoh instruksi **JMP :**

  ```
  org   100h

  mov   ax, 5       ; set ax to 5.
  mov   bx, 2       ; set bx to 2.

  jmp   calc        ; go to 'calc'.

  back: jmp stop    ; go to 'stop'.

  calc:
  add   ax, bx      ; add bx to ax.
  jmp   back        ; go 'back'.
  ```

```
stop:
ret             ; return to operating system.
```

Tentu saja ada cara mudah untuk mengkalkulasi dua angka, tetapi diatas merupakan contoh yang baik juga mengenai penggunaan instruksi **JMP** . Seperti terlihat dari contoh diatas **JMP** dapat mentransfer kendali baik secara forward dan backward. Dia dapat melompat kemanapun dalam code segment (65,535 bytes).

- **Short Conditional Jumps**

Tidak seperti instruksi **JMP,** instruksi ini mengerjakan unconditional jump, dimana ada instruksi yang mengerjakan suatu conditional jumps (jump hanya jika kondisinya terpenuhi). Instruksi ini di bagi menjadi tiga kelompok , Pertama hanya mengetest flag tunggal , Kedua membandingkan angka sebagai signed, dan Ketiga membandingkan angka sebagai unsigned.

Instruksi Jump dalam menge-test flag tunggal

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE

JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

Seperti terlihat ada instruksi yang mengerjakan hal yang sama, perlu diingat bahwa jika kita mengkompile instruksi **JE kita akan mendapatkan** disassembled sebagai: **JZ**, **JC** adalah assembled yang sama dengan **JB** dan sebagainya, Perbedaan nama digunakan untuk membuat program lebih mudah dimengerti, lebih mudah meng-kode-kan dan yang paling penting adalah mudah untuk diingat. Setiap offset yang di-dissembler tidak mempunyai petunjuk mengenai instruksi aslinya seperti apa, oleh karena itulah digunakan nama yang umum. Jika kita meng-emulate kode ini kita akan melihat seluruh instruksi di-assembled kedalam **JNB**, operational code (opcode) untuk instruksi ini adalah **73h** instruksi ini mempunyai panjang tetap dua byte , byte kedua merupakan angka dari byte tersebut ditambahkan ke register **IP** jika kondisinya terpenuhi. Karena instruksi ini hanya mempunyai 1 byte untuk menyimpan offset yang terbatas untuk mengirimkan control ke -128 bytes ke belakang atau 127 bytes ke depan, nilai ini selalu signed.

```
  jnc a
  jnb a
  jae a

  mov ax, 4
a: mov ax, 5
ret
```

Instruksi jump untuk angka Signed

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

Instruksi jump untuk angka unsigned

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ

JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

Umumnya, jika diperlukan untuk membandingkan nilai numeric instruksi **CMP** digunakan (sama dengan instruksi **SUB** (subtract) , tetapi tidak menyimpan hasilnya, hanya berpengaruh pada flagnya saja). Logikanya sederhana, contoh : Bandingkan 5 dand 2, 5 - 2 = 3

hasilnya bukan nol (zero flag diset ke 0).

contoh lainnya:

bandingkan 7 dan 7,

7 - 7 = 0

hasilnya nol! (Zero Flag diset ke 1 dan **JZ** atau **JE** tidak melompat).

contoh instruksi **CMP** dan conditional jump:

```
include "emu8086.inc"
```

```
org   100h

mov   al, 25    ; set al to 25.
mov   bl, 10    ; set bl to 10.

cmp   al, bl    ; compare al - bl.

je    equal     ; jump if al = bl (zf = 1).

putc  'n'       ; if it gets here, then al <> bl,
jmp   stop      ; so print 'n', and jump to stop.

equal:          ; if gets here,
putc  'y'       ; then al = bl, so print 'y'.

stop:

ret             ; gets here no matter what.
```

Cobalah contoh diatas dengan angka yang lain untuk **AL** dan **BL**, buka flag dengan meng-klik tombol flag , gunakan single step dan lihat yang terjadi. Kita juga dapat memakai **F5** untuk me-recompile dan reload program ke emulator.

loops

instruction	operation and jump condition	opposite instruction
LOOP	decrease cx, jump to label if cx not zero.	DEC CX and JCXZ
LOOPE	decrease cx, jump to label if cx not zero and equal (zf = 1).	LOOPNE
LOOPNE	decrease cx, jump to label if cx not zero and not equal (zf = 0).	LOOPE
LOOPNZ	decrease cx, jump to label if cx not zero and zf = 0.	LOOPZ
LOOPZ	decrease cx, jump to label if cx not zero and zf = 1.	LOOPNZ

JCXZ	jump to label if cx is zero.	OR CX, CX and JNZ

Pada dasarnya loop sama dengan jump, mungkin saja mengkodekan loop tanpa menggunakan instruksi lo op dengan hanya menggunakan jump dan compare dan seperti inilah loop bekerja . semua instruksi loop menggunakan register **CX** untuk menghitung langkah, seperti diketahui register CX memiliki 16 bit dan nilai maksimum dapat menjangkau 65535 atau FFFF,bagaimanapun juga dapat saja meletakkan loop dalam loop sehingga nilai jangkauannya dapat sebesar 65535 * 65535 * 65535 hingga batas stack ram penuh. Dimungkinkan menyimpan nilai asli pada register cx dengan instruksi **push cx** dan mengembalikannya ke asalnya jika internal loop berakhir dengan **pop cx**, contoh :

```
org 100h

mov bx, 0  ; total step counter.

mov cx, 5
k1: add bx, 1
   mov al, '1'
   mov ah, 0eh
int 10h
push cx
mov cx, 5
k2: add bx, 1
mov al, '2'
mov ah, 0eh
int 10h
push cx
          mov cx, 5
          k3: add bx, 1
          mov al, '3'
          mov ah, 0eh
          int 10h
          loop k3    ; internal in internal loop.
pop  cx
loop  k2     ; internal loop.
pop cx
loop k1           ; external loop.

ret
```

Semua conditional jump memiliki satu batasan yang besar, tidak seperti instruksi **JMP** mereka hanya jump sebesar **127** bytes kedepan dan **128** bytes kebelakang. Kita dapat mengatasi batasan ini dengantrik :

- Ambil insruksi opposite conditional jump dari table diatas, lalu jump ke *label_x*.
- Guanakan instruksi **JMP** untuk melompat ke lokasi yang diinginkan.
- buat *label_x:* setelah instruksi **JMP.**

Contoh :

```
include "emu8086.inc"

org    100h

mov    al, 5
mov    bl, 5

cmp    al, bl     ; bandingkan al - bl.

; je equal        ; hanya ada 1 byte

jne    not_equal  ; jump jika al <> bl (zf = 0).
jmp    equal
not_equal:


add    bl, al
sub    al, 10
xor    al, bl

jmp skip_data
db 256 dup(0)     ; 256 bytes
skip_data:


putc   'n'        ; jika disini ,maka  al <> bl,
jmp    stop       ; cetak 'n', dan jump ke stop.

equal:            ; jika disini,
putc   'y'        ; maka al = bl, jadi cetak 'y'.

stop:

ret
```

Contoh lainnya, menggunakan metode yang disediakan nilai terdekat dalam label. Jika nilai terdekat diawali $ maka jump dilakukan, kalau

tidak compiler menghitung instruksi dimana jump secara langsung diberikan ke offset, contoh :

```
org    100h

; unconditional jump forward:
; skip over next 3 bytes + itself
; the machine code of short jmp instruction
takes 2 bytes.
jmp $3+2
a db 3    ; 1 byte.
b db 4    ; 1 byte.
c db 4    ; 1 byte.

; conditional jump back 5 bytes:
mov bl,9
dec bl      ; 2 bytes.
cmp bl, 0   ; 3 bytes.
jne $-5     ; jump 5 bytes back

ret
```

BAGIAN 8

PROSEDURE

Procedure merupakan bagian dari kode yang dapat dipanggil dari program dalam rangka tugas tertentu. Prosedur membuat program lebih terstruktur dan mudah dimengerti. Umumnya prosedur mengembalikan titik dimana dia dipanggil, sintaknya :

```
name PROC
        ; kode dituliskan disini
        ; dalam badan prosedur ...

RET
name ENDP
```

name – nama prosedur, nama ini harus sama dia awal dan akhir prosedur.

RET menyatakan kembali ke OS setelah pemanggilan prosedur. **PROC** dan **ENDP** merupakan direktiv compiler ,jadi tidak di-assembled ke dalam kode mesin. Kompiler hanya mengingat alamat dari prosedur.

Instruksi **CALL** digunakan untuk memanggil prosedur.

Contoh:

```
ORG    100h

CALL   m1

MOV    AX, 2

RET              ; kembali ke OS.

m1     PROC
MOV    BX, 5
RET              ; kembali ke pemanggilnya
m1     ENDP

END
```

contoh diatas memanggil prosedur **m1**, yang mengerjakan **MOV BX, 5**, dan kembali ke instruksi selanjutnya setelah **CALL**: **MOV AX, 2**.

Ada beberapa cara untuk mengirim parameter ke prosedur, yang paling mudah adalah mengirimkan parameter menggunakan register, disini contoh lainnya dari prosedur yang menerima dua parameter dalam register **AL** dan **BL** , kalikan kedua parameter dan kirim hasilnya ke register **AX**:

```
ORG    100h

MOV    AL, 1
MOV    BL, 2

CALL   m2
CALL   m2
CALL   m2
CALL   m2

RET                ; kembali ke OS.

m2     PROC
MUL    BL          ; AX = AL * BL.
RET                ; kembali ke pemanggil.
m2     ENDP

END
```

contoh dalam diatas nilai register **AL** di update setiap saat prosedur di panggil, register **BL** tetap tidak berubah, jadi algoritma ini menghitung **2** pangkat **4**, hasil akhirnya di register **AX** adalah **16** (or 10h).

Cotoh lainnya, prosedur mencetak pesan *Hello World!* :

```
ORG    100h

LEA    SI, msg       ; load address msg ke SI.

CALL   print_me

RET                ; ke OS.

;
=========================================================
; prosedur ini mencetak string,stringy harus null
; terminated (ada nol  diakhir string),
; alamat string hrs berada di register SI :
print_me     PROC
```

```
next_char:
   CMP  b.[SI], 0    ; check jika nol ke stop
   JE   stop         ;

   MOV  AL, [SI]     ; selanjutnya ambil karakter ASCII .

   MOV  AH, 0Eh      ; teletype nomer function.
   INT  10h          ; gunakan interrupt utk mencetak karakter di AL.

   ADD  SI, 1        ; advance index dari string array.

   JMP  next_char    ; kembali dan cetak karakter lainnya.

stop:
RET                  ; kembali ke pemangil proc.
print_me     ENDP
;
=========================================================

msg    DB  'Hello World!', 0   ; null terminated string.

END
```

prefix "**b.**" sebelum [SI]berarti bahwa kita perlu membandingkan byte, bukan word. Jika kita perlu membandingkan word tambahkan prefix "**w.**". Jika salah satu operand pembandingnya berupa register, hal tersebut tidak diperlukan karena compiler tahu ukuran tiap register .

BAGIAN 9

STACK

Stack adalah suatu area di memori yang menyimpan data sementara. Stack digunakan dengan instruksi **CALL** untuk menyimpan alamat yang dikembalikan pada prosedur ,instruksi **RET** mengambil nilai ini dari stack dan mengembalikannya ke offset. Hampir sama kejadiannya jika instruksi **INT** memanggil interrupt, dia menyimpan register sflag stack, code segment dan offset. instruksi **IRET** digunakan untuk mengembalikan dari pemanggilan interrupt. Kita juga dapat menggunakan stack untuk menyimpan data lainnya, ada dua instruksi yang bekerja dengan stack :

PUSH - menyimpan nilai 16 bit dalam stack.

POP – mengambil nilai 16 bit dari stack.

Syntax for **PUSH** instruction:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bit variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Syntax for **POP** instruction:

POP REG
POP SREG
POP memory
REG: AX, BX, CX, DX, DI, SI, BP, SP.

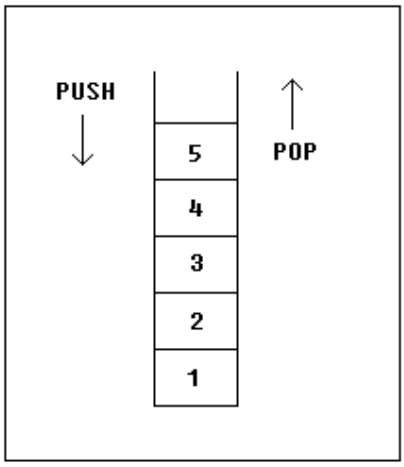
SREG: DS, ES, SS, (except CS).

memory: [BX], [BX+SI+7], 16 bit variable, etc...

Catatan:

- **PUSH** dan **POP** bekerja dengan nilai 16 bit saja !
- **PUSH immediate** bekerja hanya pada CPU 80186 dan sesudahnya!

Stack menggunakan algoritma **LIFO** (Last In First Out) , artinya jika kita push nilai satu per satu ke dalam stack : **1, 2, 3, 4, 5**
nilai pertama yang dapat kita pop adalah **5**, lalu **4**, **3**, **2**, dan terakhir **1**.



Jumlah yang di **PUSH** dan di **POP harus sama**, jika tidak stack akan corrupted dan tidak dapat kembali ke OS . Kita gunakan instruksi **RET** untuk kembali ke OS, jadi jika program mulai ada alamat yang dikembalikan di stack (umumnya 0000h). Instruksi **PUSH** dan **POP** berguna karena kita tidak memiliki cukup register untuk dioperasikan, triknya adalah :

- Simpan nilai original dari register dalam stack (dg **PUSH**).
- Pakai register untuk tujuan apapun.
- Kembalikan nilai original register dari stack (dg **POP**).

contoh:

```
ORG    100h

MOV    AX, 1234h
PUSH   AX          ; simpan nilai AX di stack.

MOV    AX, 5678h   ; modify nilai AX .

POP    AX          ; kembalikan nilai original AX

RET

END
```

Contoh lainnya kegunaan stack dalam pertukaran nilai :

```
ORG    100h

MOV    AX, 1212h   ; simpan 1212h di AX.
MOV    BX, 3434h   ; simpan 3434h di BX


PUSH   AX          ; simpan nilai AX di stack.
PUSH   BX          ; simpan nilai BX di stack.

POP    AX          ; set AX ke  nilai original BX.
POP    BX          ; BX ke  nilai original AX
RET

END
```

Pertukaran terjadi karena stak memakai algoritma **LIFO** (Last In First Out) , jadi saat kita push **1212h** dan kemudian **3434h**, pada saat pop kita mendapatkan yang pertama **3434h** kemudian **1212h**. Area memori stack di set oleh register **SS** (Stack Segment) , dan register **SP** (Stack Pointer) . Umumnya OS mengeset nilai register ini saat program mulai .

"**PUSH *source***" instruksi ini mengerjakan:

- Kurangi **2** dari register **SP** .
- Tulis nilai ***source*** ke alamat **SS:SP**.

"**POP *destination***" instruksi ini mengerjakan :

- Tulis nilai alamat **SS:SP** ke ***destination***.
- tambahkan **2** ke register **SP**.

Alamat sekarang ditunjuk oleh **SS:SP** yang dinamakan **the top of the stack**. Untuk file .**COM** stack segment pada umumnya berada di code segment, dan stack pointer diset pada nilai **0FFFEh**. Pada alamat **SS:0FFFEh** disimpan alamat kembalian untuk instruksi **RET** yang dieksekusi pada akhir program. Klik tombol [**Stack**] pada emulator window. The top of the stack ditandai dengan "**<**".

BAGIAN 10

MAKRO

Macros hampir mirip prosedur, tetapi setidaknya dia hanya ada hingga kode dikompilasi, setelah kompilasi dialkukan semua makro akan dig anti dengan instruksi mesin . Jika kita mendeklarasikan makro dan tidak pernah menggunakannya dalam kode, compiler akan mengabaikannya. **emu8086.inc** merupakan contoh bagaimana makro dapat dipakai, file ini berisi beberapa makro untuk mempermudah pengkodean kita.

```
Macro definition:

name    MACRO  [parameters,...]

            <instructions>

ENDM
```

Tidak seperti prosedur makro harus didefinisikan diatas kode yang akan memakainya, contoh :

```
MyMacro    MACRO  p1, p2, p3

    MOV AX, p1
    MOV BX, p2
    MOV CX, p3

ENDM

ORG 100h

MyMacro 1, 2, 3

MyMacro 4, 5, DX

RET
```

Kode diatas diperluas menjadi:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
```

```
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

Pebedaan **macros** dan **procedures**:

- prosedur dipakai dengan instruksi **CALL**, contoh:

  ```
  CALL MyProc
  ```

- Makro hanya menuliskan namanya saja, contoh:

  ```
  MyMacro
  ```

- Prosedure terletak pada alamat yang spesifik di memori, dan jika kita menggunakan procedure yang sama sebanyak 100 kali, CPU hanya mentransfer control ke bagian memori. Kontrol akan dikembalikan dengan instruksi **RET**. S**tack** yang digunakan untuk menyimpan alamat pengembalian. Instruksi **CALL** memerlukan setidaknya 3 bytes, jadi ukuran output executable file-nya tidak besar ,tidak masalah seberapa banyak pemanggilan prosedur dilakukan.

- Macro akan menambah secara langsung dalam kode program . Jadi jika kita memakai 100 kali makrokompiler akan menambah sebanyak 100 kali pada output exe filnya setiap kali makro tersebut di pakai.

- Stack atau general purpose register harus digunakan untuk mengirim parameter dalam prosedur.

- Untuk mengirimkan parameter ke makro kita dapat mengetikkannya setelah nama makronya, contoh:

  ```
  MyMacro 1, 2, 3
  ```

- Untuk menandai akhir makro gunakan direktiv **ENDM** sudah cukup.

- Untuk memandai akhir procedure harus mengetikan nama prosedur sebelum direktive **ENDP**.

Makro akan diperluas secara langsug dalam kode, sehingga jika ada label didalam definisi makro kita akan mendapatkan pesan error "Duplicate declaration" saat makro dipakai dua kali atau lebih. Untuk menghindari ini gunakan directive **LOCAL** diikuti nama variabelnya , nama label atau procedure, contoh:

```
MyMacro2    MACRO
            LOCAL label1, label2

            CMP  AX, 2
            JE label1
            CMP  AX, 3
            JE label2
            label1:
                        INC  AX
            label2:
                        ADD  AX, 2
ENDM


ORG 100h

MyMacro2

MyMacro2

RET
```

Jika kita akan merencanakan mengunakan makro pada program, mungkin lebih baik menempatka makro pada file yang berbeda. Tempatkan di folder **Inc** dan gunakandirektive **INCLUDE *file-name*** untuk menggunakan makro.

BAGIAN 11

MEMBUAT SISTEM OPERASI SENDIRI

Biasanya, saat komputer dihidupkan akan mencoba me-load 512-byte sector pertama (Cylinder **0**, Head **0**, Sector **1**) dari disk drive, misal **A:** ke lokasi memori **0000h:7C00h** dan memberikan kontrolnya. Jika gagal BIOS mencoba menggunakan MBR dari hard drive primer. Dalam bagian ini akan mencoba mengcover booting dari sebuah floppy drive,dengan prinsip yang sama dapat juga dilakukan di harddisk. Tetapi menggunakan floppy drive memiliki beberapa keuntungan:

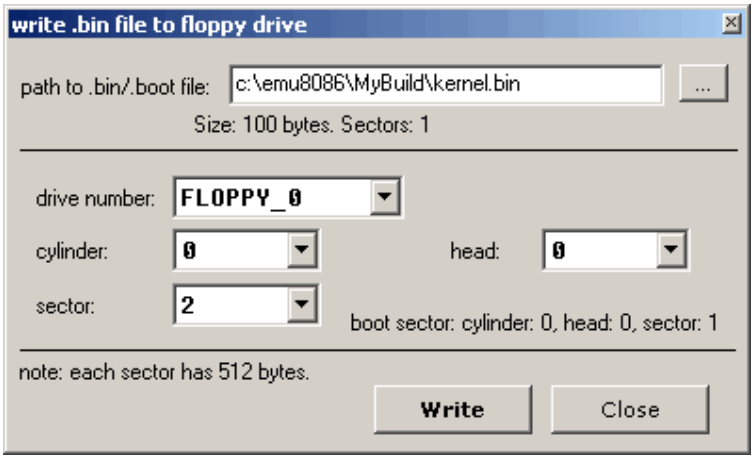
- OS kita tetap utuh (windows, dos, linux, unix, be-os...).
- udah dan aman memodifikasi boot record padaf floppy disk.

contoh floppy disk boot program:

```
; directive utk membuat BOOT file:
#make_boot#

; Boot record di-loaded di 0000:7C00,
; informasikan compiler utk membuat permintaan yang benar:
ORG 7C00h

PUSH    CS   ; yakinkan DS=CS
POP     DS

; load alamat message  ke dalam register SI  :
LEA SI, msg

; teletype id fungsi:
MOV AH, 0Eh

print:   MOV AL, [SI]
        CMP AL, 0
        JZ done
        INT 10h   ; cetak dg teletype.
        INC SI
        JMP print

; tunggu utk  'any key':
done:      MOV AH, 0
          INT 16h
```

```
; simpan nilai  magic di 0040h:0072h:
;   0000h - cold boot.
;   1234h - warm boot.
MOV     AX, 0040h
MOV     DS, AX
MOV     w.[0072h], 0000h ; cold boot.

JMP          0FFFFh:0000h             ; reboot!


new_line EQU 13, 10

msg DB  'Hello ini  Boot Program saya yang pertama!'
    DB  new_line, 'Tekan sembarang tombol utk reboot', 0
```

tulis contoh diatas pada editor emu8086 dan tekan **emulate**. Emulator secara otomatis akan me-load file **.bin** ke **0000h:7C00h** (ini digunakan sebagai suplemen file .binf utk mengetahui dimana akan di load). Kita dapat menjalankan seperti program biasa, atau dapat menggunakan menu **virtual drive** untuk menulis: **write 512 bytes di 7c00h ke boot sector** dari virtual floppy drive (file "**FLOPPY_0**" file di c:\emu8086). Setelah program ditulis ke virtual floppy drive, kita dapat memilih **boot from floppy** dari menu **virtual drive** . File **.bin** untuk boot records terbatas pada 512 bytes (ukuran sektor). Jika OS baru kita berukuran lebih dari nilai tersebut, kita perlu menggunakan suatu boot program untuk me- load data dari sector lainnya (seperti *micro-os_loader.asm*). Contoh OS kecil dapat di temukan di c:\emu8086\examples. Untuk membuat perluasan OS kita (lebih dari 512 bytes), kita dapat tambahkan sector pada floppy disk. Ini direkomendasikan untuk menggunakan file "**.bin**" untuk tujuan ini (untuk membuat file "**.bin**" pilih "**BIN Template**" dari menu "**File**" -> "**New**"). Untuk menulis file "**.bin**" ke virtual floppy, pilih **"Write .bin file to floppy..."** dari menu **"Virtual drive"** emulator, kita harus menulisnya dimanapun di boot sektor (Cylinder: **0**, Head: **0**, Sector: **1**).



kita dapat menggunakan utility ini untuk menulis file **.bin** ke virtual floppy disk ("**FLOPPY_0**" file), dapat juga menu "**write 512 bytes at 7c00h to boot sector**" . bagaimanapun juga, kita perlu ingat bahwa file **.bin** yang didisain untuk menjadi boot record harus selalu di tulis ke cylinder: **0**, head: **0**, sector: **1**

Boot Sector Location:
Cylinder: 0
Head: 0
Sector: 1

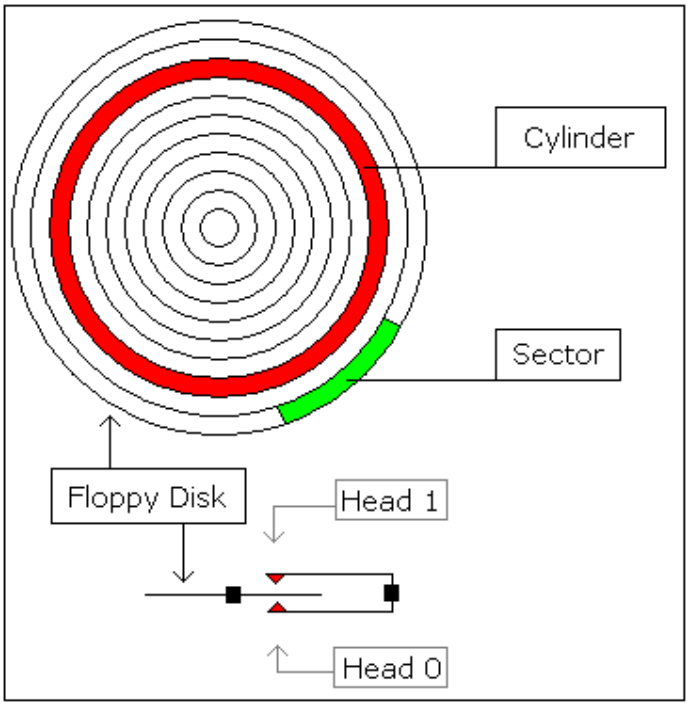
Untuk menulis file .bin kef loopy disk sebenarnya gunakan writebin.asm, lalu kompilasikan ke file .com dan jalankan dari command prompt.

- Untuk menulis tipe boot record : **writebin loader.bin** ;
- untuk menulis tipe modul kernel : **writebin kernel.bin /**k.
  parameter **/k** mengatakan pada program untuk menulis file pada sector ke dua sebagai ganti yang pertama .Hal tersebut bukan menjadi masalah dalam rangka menulis file ke floppy drive, tetapi yang menjadi masalah dimana kita akan menuliskannya .

Catatan: boot record ini tidak kompitabel dengan MS-DOS/Windows, juga Linux atau Unix , OS tidak mengijinkan kita membaca atau menulis file dalam disket hingga kita akan me-reformatnya ,Jadi yakinkan bahwa disket yang kita gunakan tidak berisi data yang penting. Bagaimanapun juga kita dapat menulis dan membaca apa saja dari dank e disk menggunakan

interrupt untuk mengakses disk secara low level , bahkan mungkin memproteksi informasi yang bernilai dari orang lain atau jika seseorang mendapatkan disket mungkin dia akan berpikir bahwa disket tersebut kosong dan akan memformatnya, karena pilihan defaultnya ada di windows operating system

Struktur floppy drive dan diskette yang ideal:



Untuk disket **1440 kb** :

- floppy disk mempunyai 2 side , dan ada 2 head (**0..1**), head drive bergerak keatas permukaan dari disket untuk tiap sisi.
- Tiap sisi memiliki 80 cylinders (di nomori **0..79**).
- tiap cylinder memiliki 18 sectors (**1..18**).
- tiap sector memiliki **512** bytes.
- Total ukuran floppy disk adalah : 2 x 80 x 18 x 512 = **1,474,560** bytes.

Catatan: disket the MS-DOS (windows) yang terformat memiliki free space sedikit berkurang (sekitar 16,896 bytes) karena OS

perlu menempatkan struktur nama file dan direktori (sering disebut FAT atau file system allocation table). Semakin banyak nama file semaki berkurang ruang yang kosong .Yang paling efisien untuk menyimpan file adalah menuliskan secara langsung dengan menggunakan file system , dan untuk beberapa kasus hal ini merupakan yang paling reliable , jika kita dapat menggunakannya. Untuk membaca sector dari floppy drive gunakan **INT 13h / AH = 02h**.

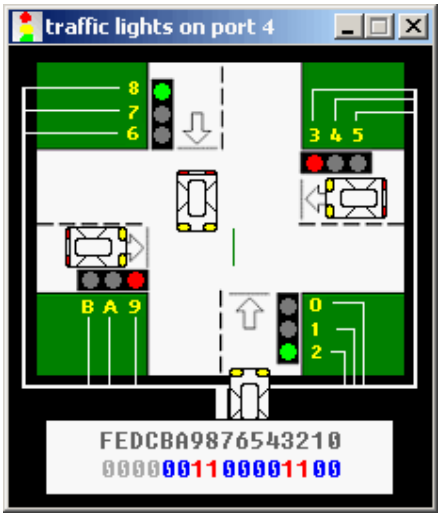
BAGIAN 12

PENGENDALIAN DEVICE EXTERNAL

(ROBOT, STEPPER-MOTOR...)

Ada 7 devices yang di-attach ke emulator: traffic lights, stepper-motor, LED display, thermometer, printer, robot dan simple test . kita dapat melihat device ketika kita menu "**Virtual Devices**" dari emulator. Secara umum , dimungkinkan menggunakan CPU keluarga x86 untuk mengendalikan semua device, perbedaanya berada pada nomor port I/O , ini dapat dimodifikasi menggunakan peralatan elektronik . Biasanya file "**.bin**" dituliskan ke dalam chip Read Only Memory (ROM) , system membaca program dari chip, dan me-load ke RAM dan menjalankannya sebagai program. Prinsip ini digunakan untuk peralatan modern lainnya seperti micro-wave ovens dan lainnya.

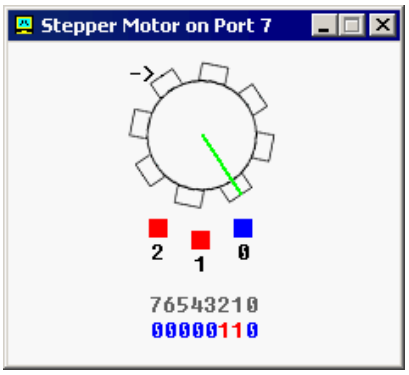
Traffic Lights



Biasanya untuk mengontrol lampu lalu lintas suatu larik nilai digunakan . Nilai di baca dari larik dan dikirimkan ke port, contoh:

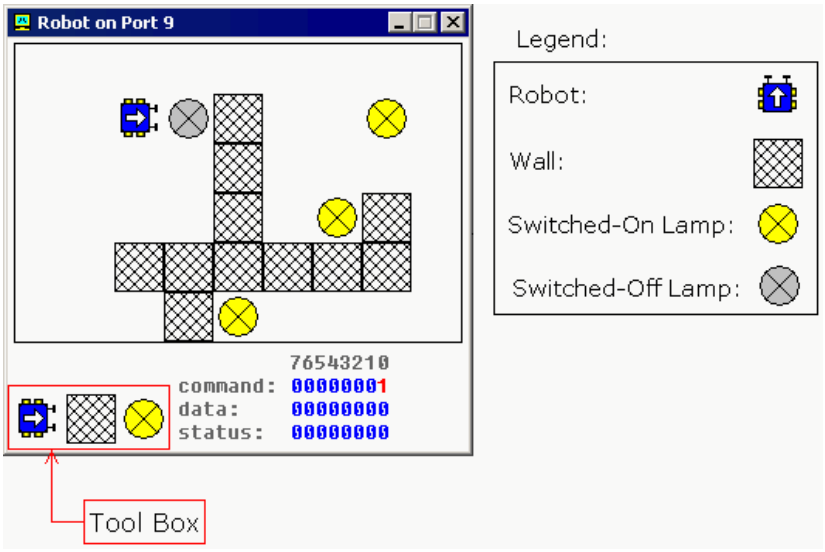
```
; controlling external device with 8086 microprocessor.
; realistic test for c:\emu8086\devices\Traffic_Lights.exe

#start=Traffic_Lights.exe#

name "traffic"


mov ax, all_red
out 4, ax


mov si, offset situation


next:
mov ax, [si]
out 4, ax

; wait 5 seconds (5 million microseconds)
mov     cx, 4Ch    ;    004C4B40h = 5,000,000
mov     dx, 4B40h
mov     ah, 86h
int     15h


add si, 2 ; next situation
cmp si, sit_end
jb  next
mov si, offset situation
jmp next


;                   FEDC_BA98_7654_3210
situation       dw      0000_0011_0000_1100b
s1              dw      0000_0110_1001_1010b
s2              dw      0000_1000_0110_0001b
s3              dw      0000_1000_0110_0001b
s4              dw      0000_0100_1101_0011b
sit_end = $


all_red         equ     0000_0010_0100_1001b
```

Stepper-Motor



Motor dapat separuh dilangkahkan dengan memasang pasangan magnit, Motor juga dapat dilangkahkan penuh dengan memasang pasangan magnit, yang diikuti oleh pasangan magnit yang lain dan pada akhirnya yang diikuti oleh magnit tunggal dan seterusnya. Cara Yang terbaik untuk membuat langkah penuh akan membuat dua separuh langkah-langkah tersebut. Setengah langkah sama dengan **11.25** derajad. satu langkah penuh sama dengan **22.5** derajad. Motor dapat di putar baik secara clock-wise dan counter-clock-wise.

Robot



lihat pada contoh yang ada in c:\emu8086\examples