

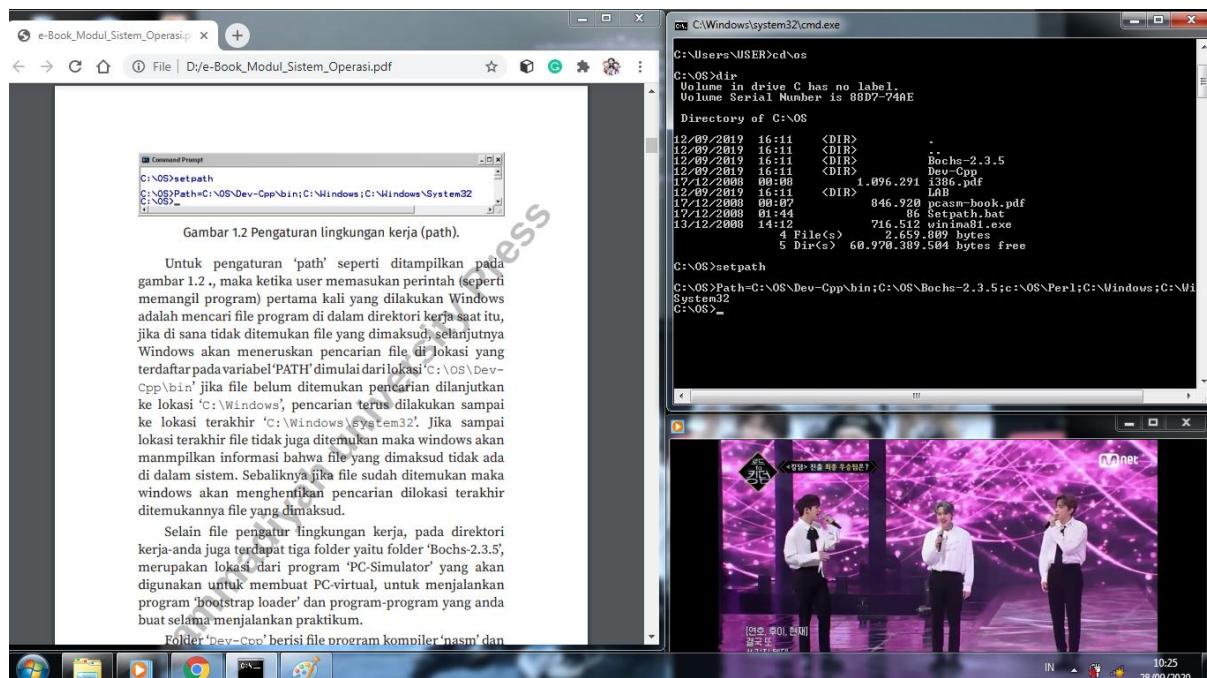
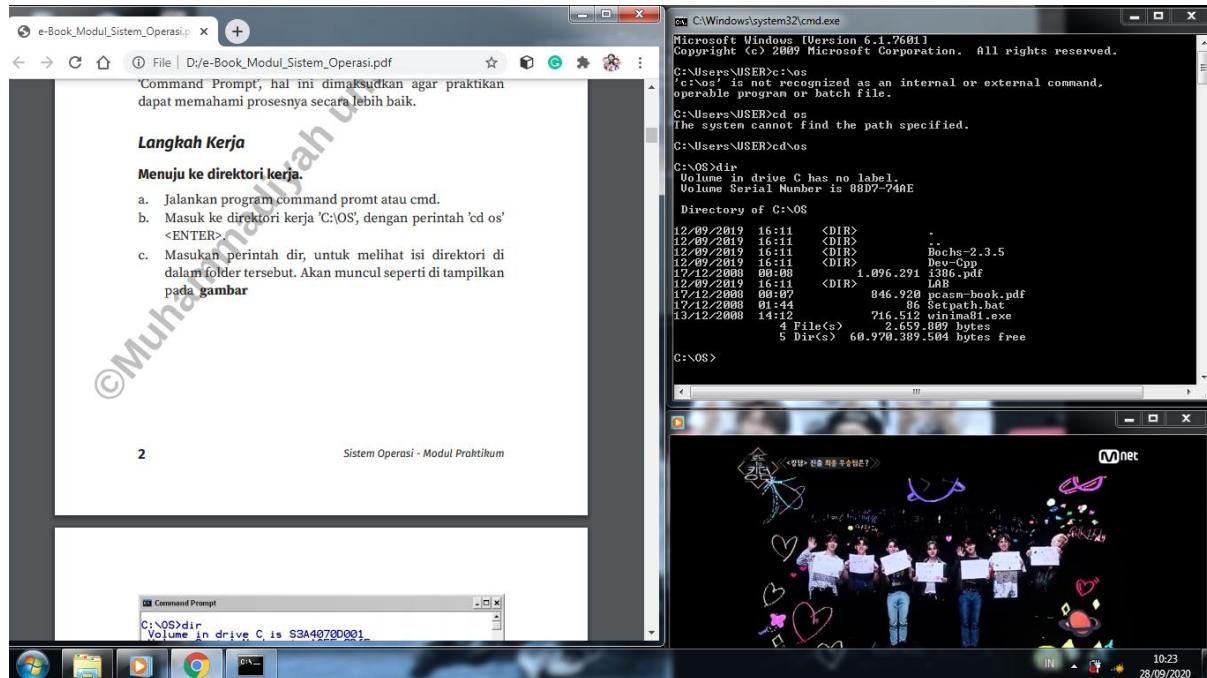
Nama : Fiya Is Karima

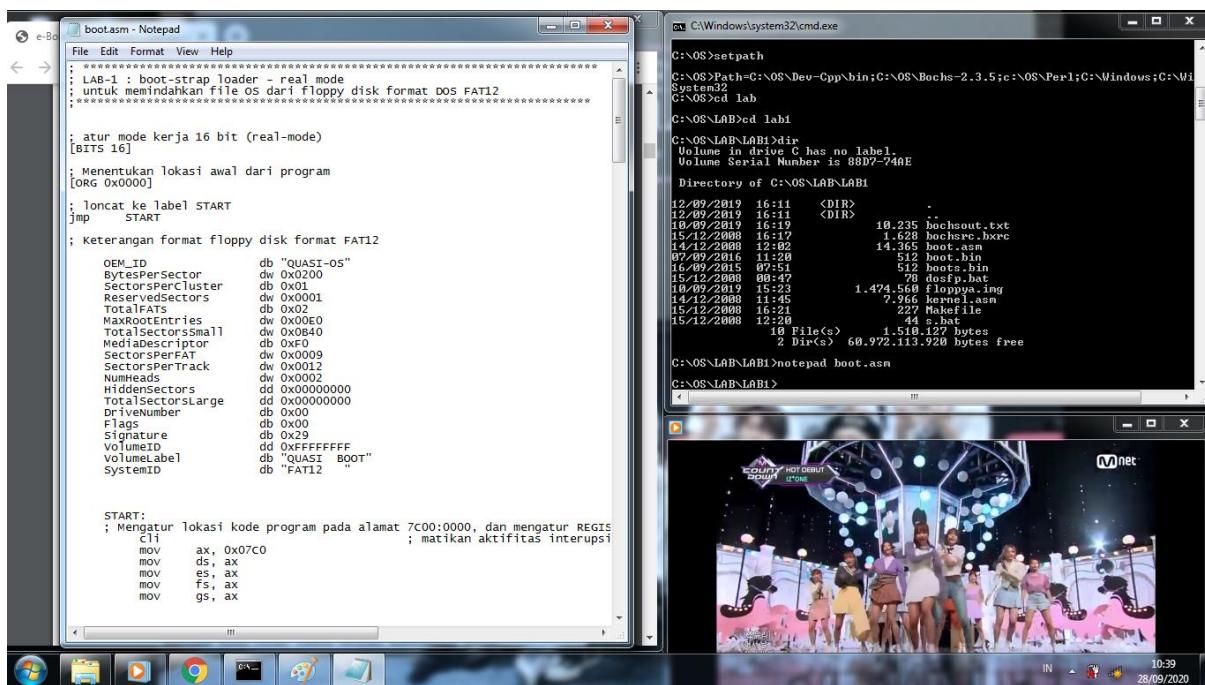
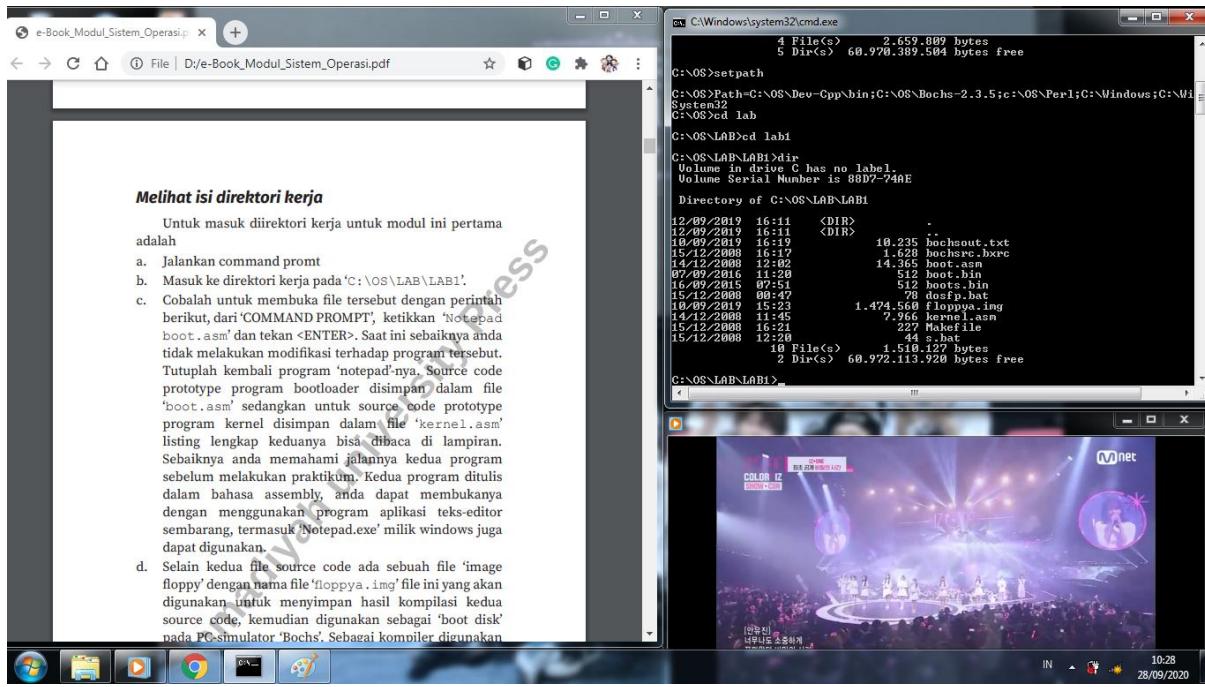
NIM : L200199271

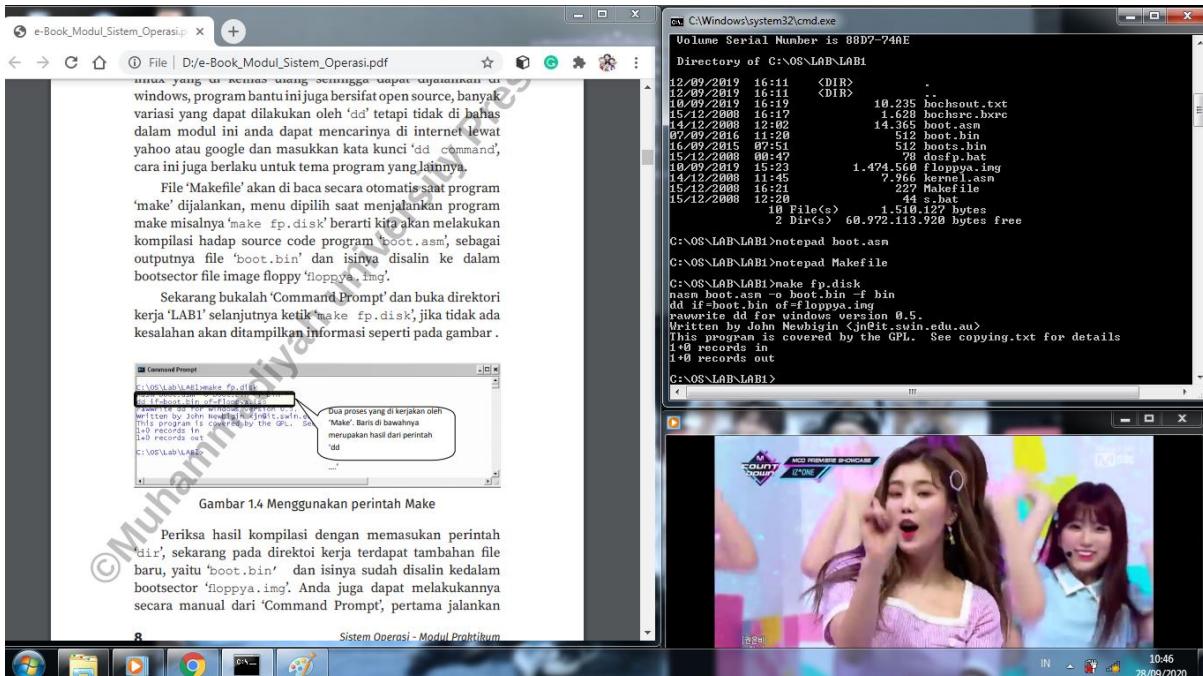
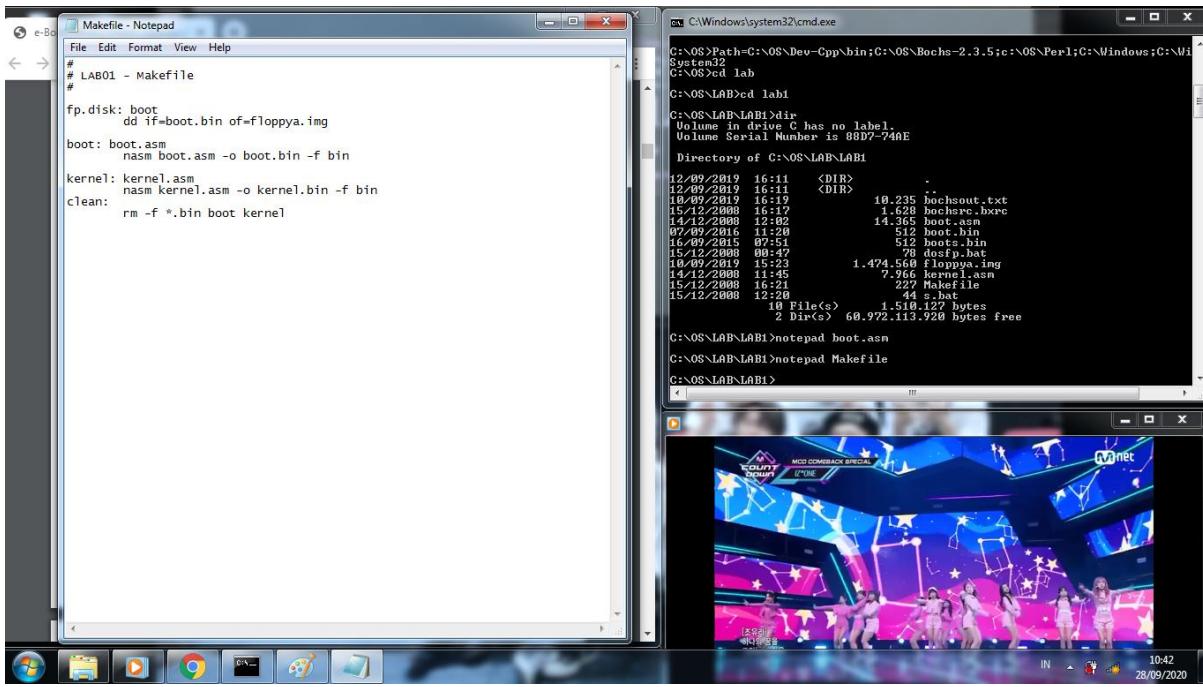
Kelas : G

## TUGAS 1

Berikut percobaan menggunakan CMD

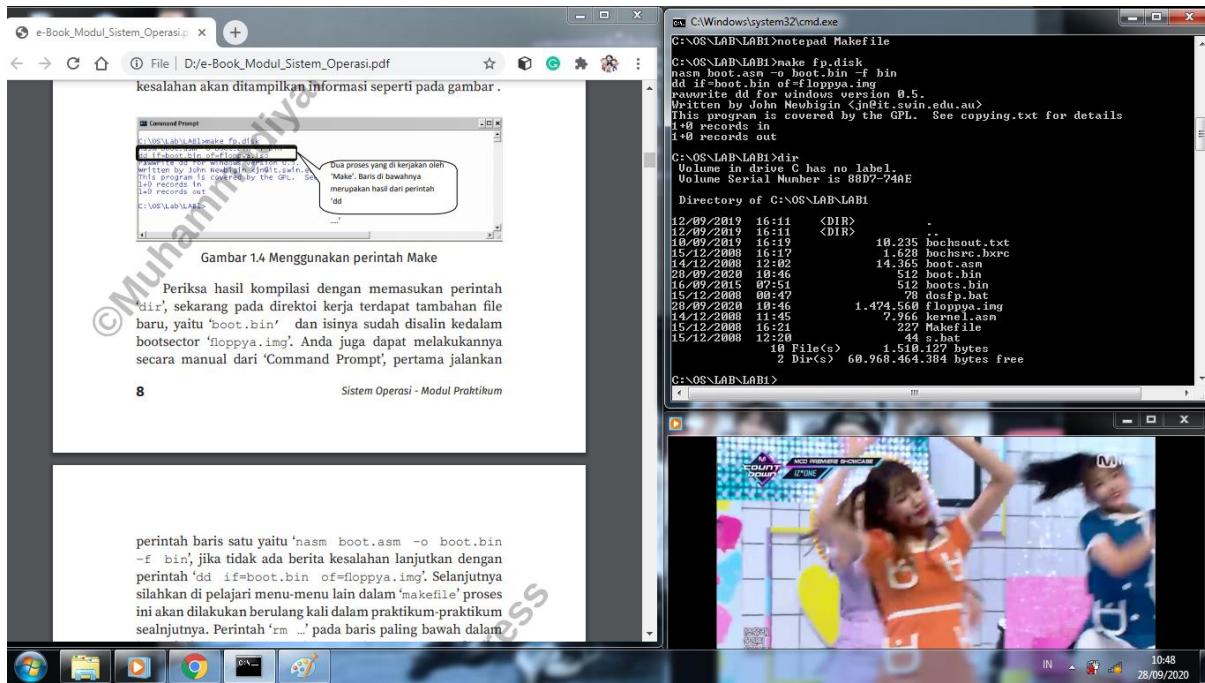






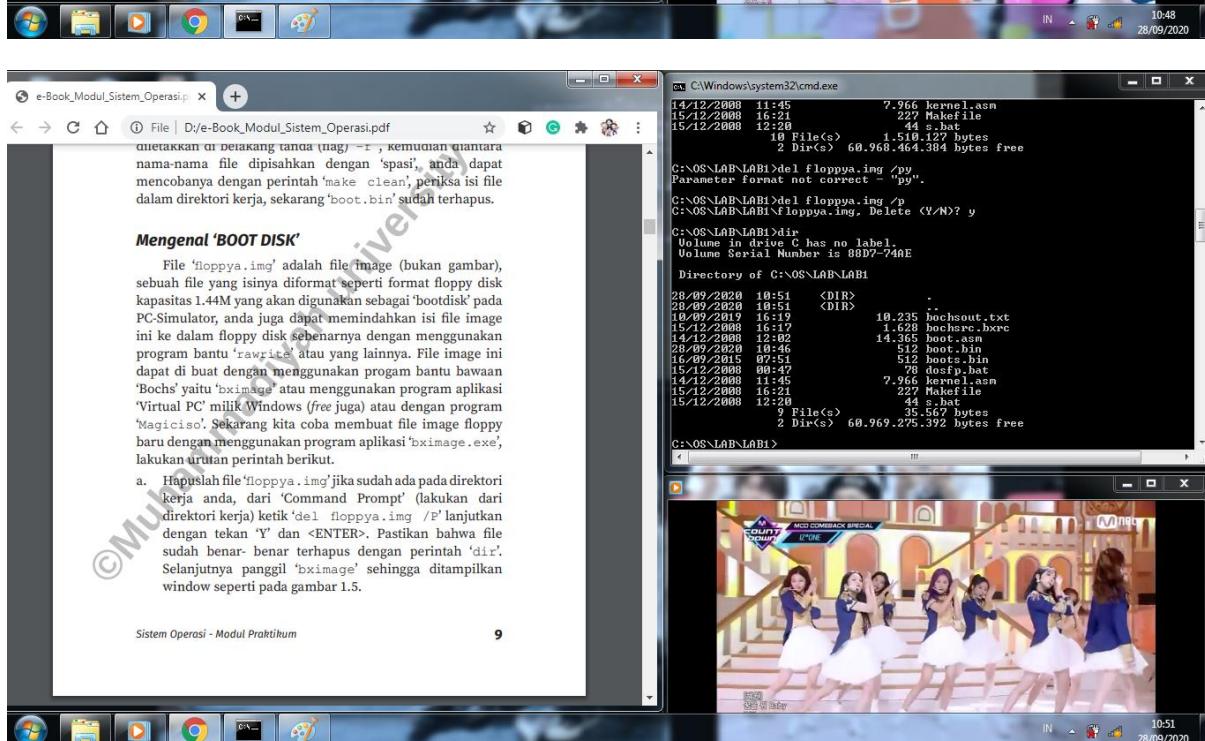
Gambar 1.4 Menggunakan perintah Make

Periksa hasil komplilasi dengan memasukan perintah 'dir', sekarang pada direktori kerja terdapat tambahan file baru, yaitu 'boot.bin' dan isinya sudah disalin kedalam bootsector 'floppya.img'. Andi juga dapat melakukannya secara manual dari 'Command Prompt', pertama jalankan



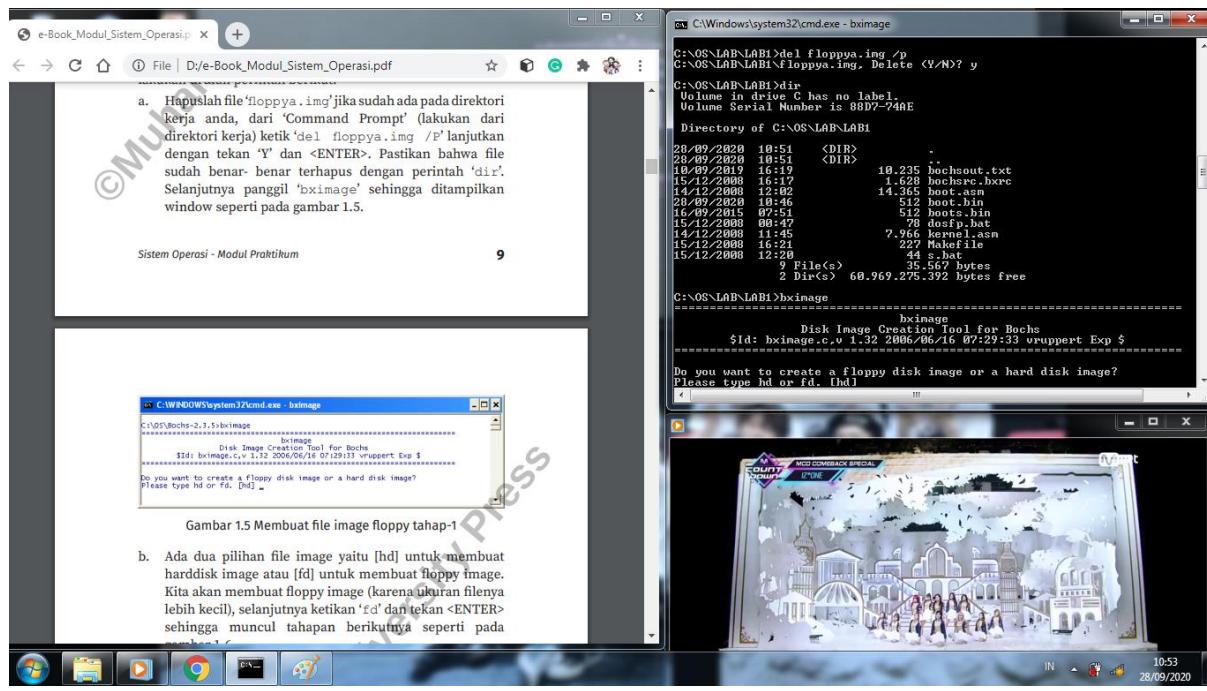
8 Sistem Operasi - Modul Praktikum

perintah baris satu yaitu 'nasm boot.asm -o boot.bin -f bin', jika tidak ada berita kesalahan lanjutkan dengan perintah 'dd if=boot.bin of=floppya.img'. Selanjutnya silahkan di pelajari menu-menu lain dalam 'makefile' proses ini akan dilakukan berulang kali dalam praktikum-praktikum selanjutnya. Perintah 'rm ...' pada baris paling bawah dalam



Sistem Operasi - Modul Praktikum

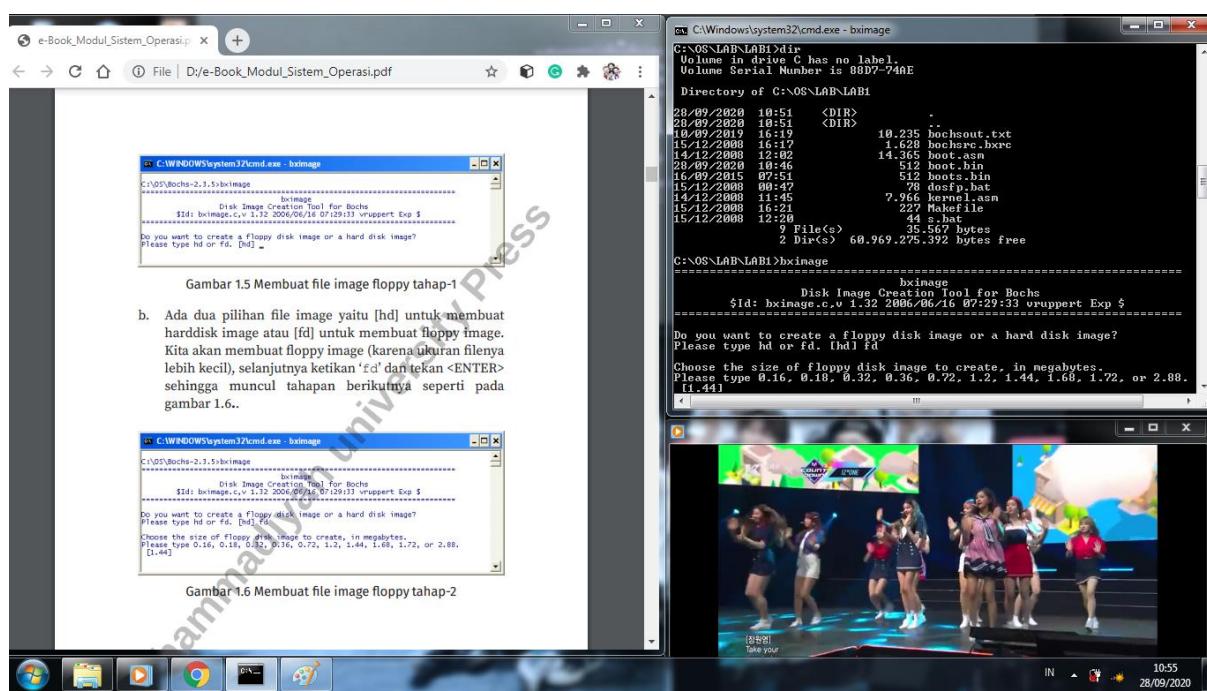
9



Gambar 1.5 Membuat file image floppy tahap-1

- Hapuslah file 'floppya.img' jika sudah ada pada direktori kerja anda, dari 'Command Prompt' (lakukan dari direktori kerja) ketik 'del floppya.img /P' lanjutkan dengan tekan 'Y' dan <ENTER>. Pastikan bahwa file sudah benar-benar terhapus dengan perintah 'dir'. Selanjutnya panggil 'bximage' sehingga ditampilkan window seperti pada gambar 1.5.

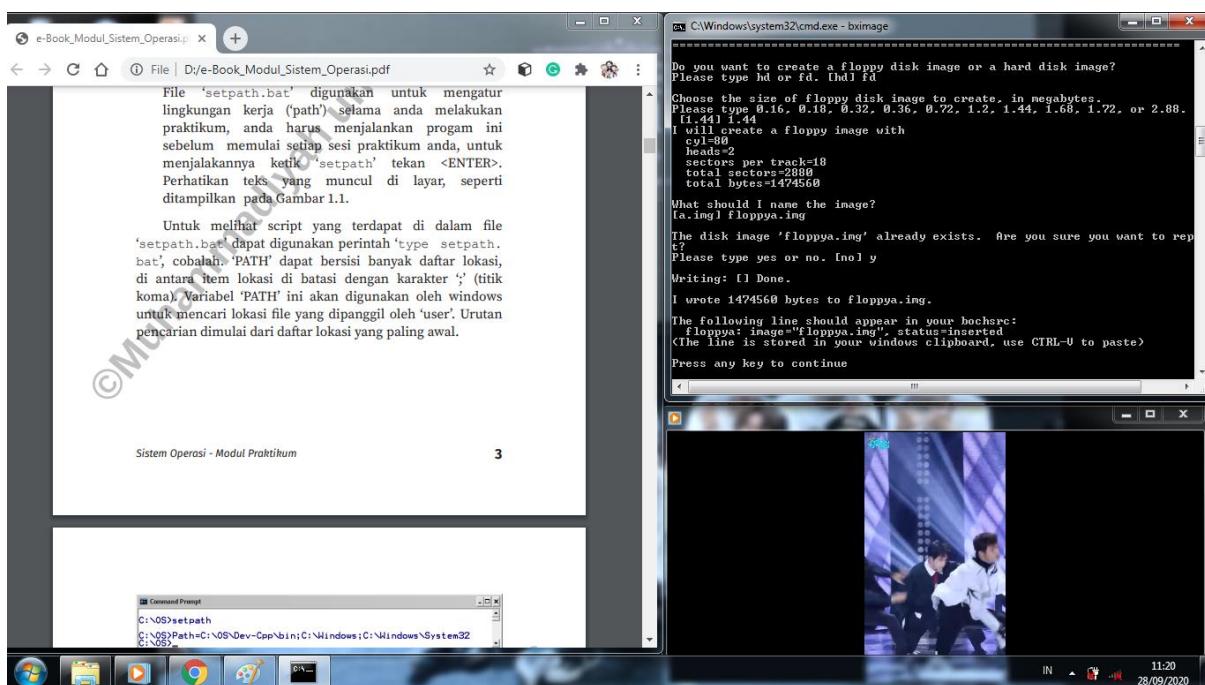
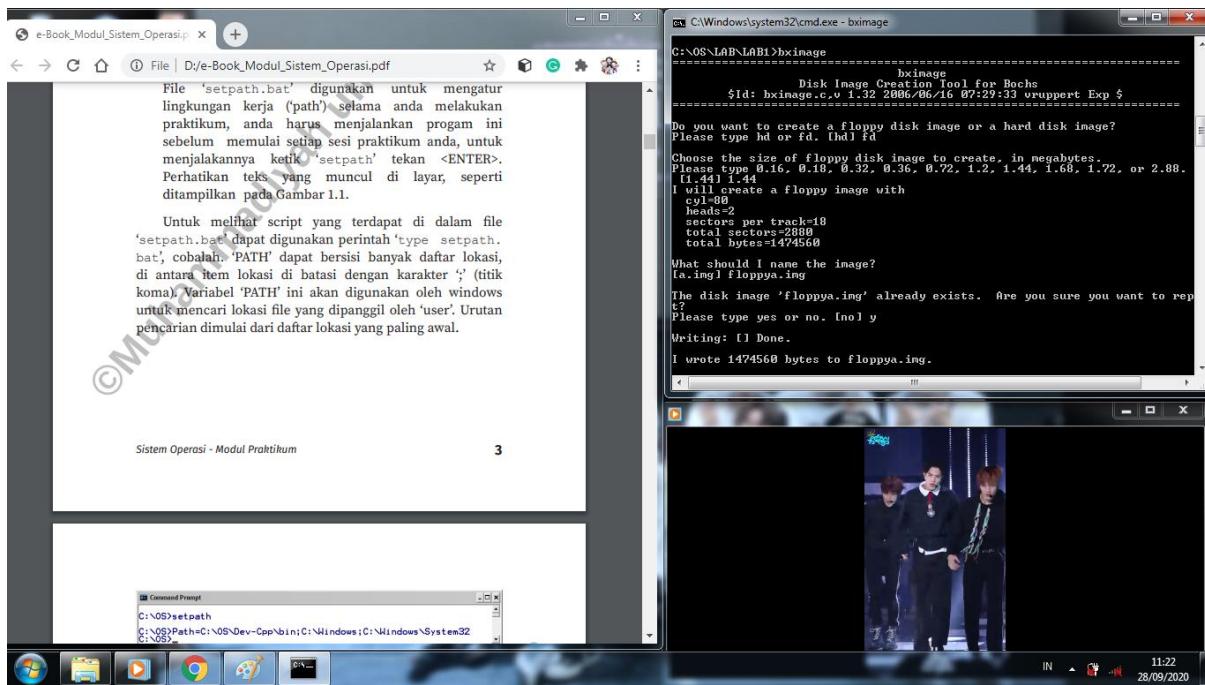
9

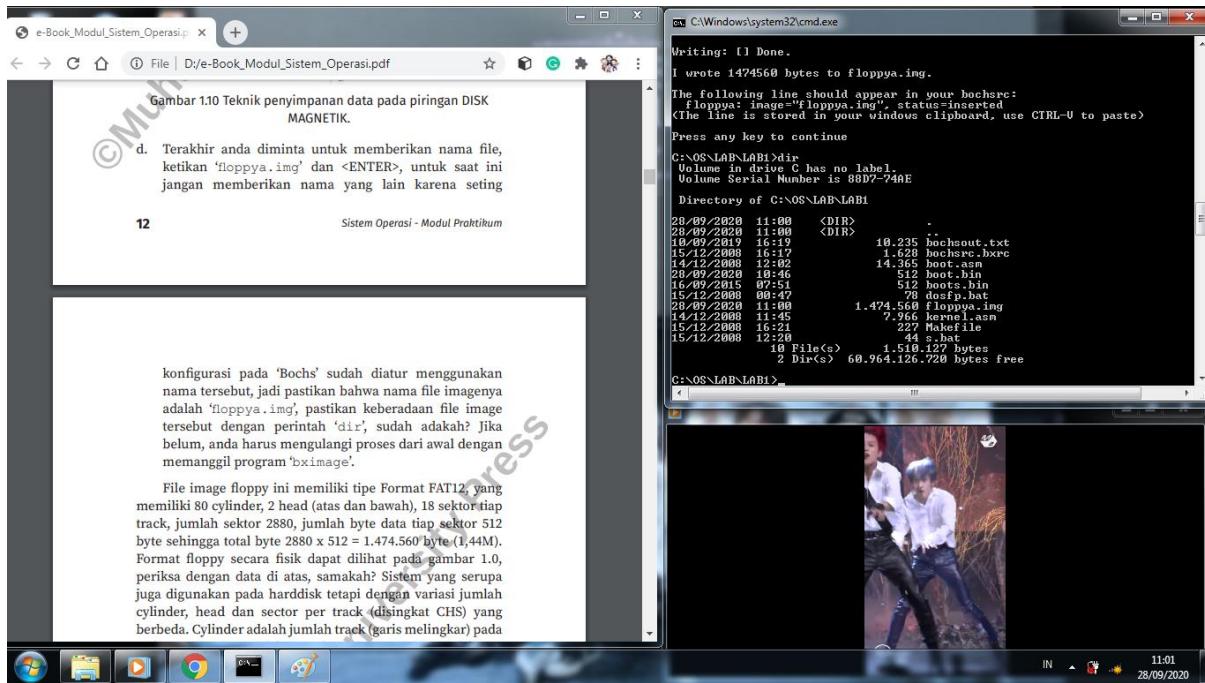


Gambar 1.5 Membuat file image floppy tahap-1

- Ada dua pilihan file image yaitu [hd] untuk membuat harddisk image atau [fd] untuk membuat floppy image. Kita akan membuat floppy image (karena ukuran filenya lebih kecil), selanjutnya ketikan 'fcl' dan tekan <ENTER> sehingga muncul tahapan berikutnya seperti pada gambar 1.6..

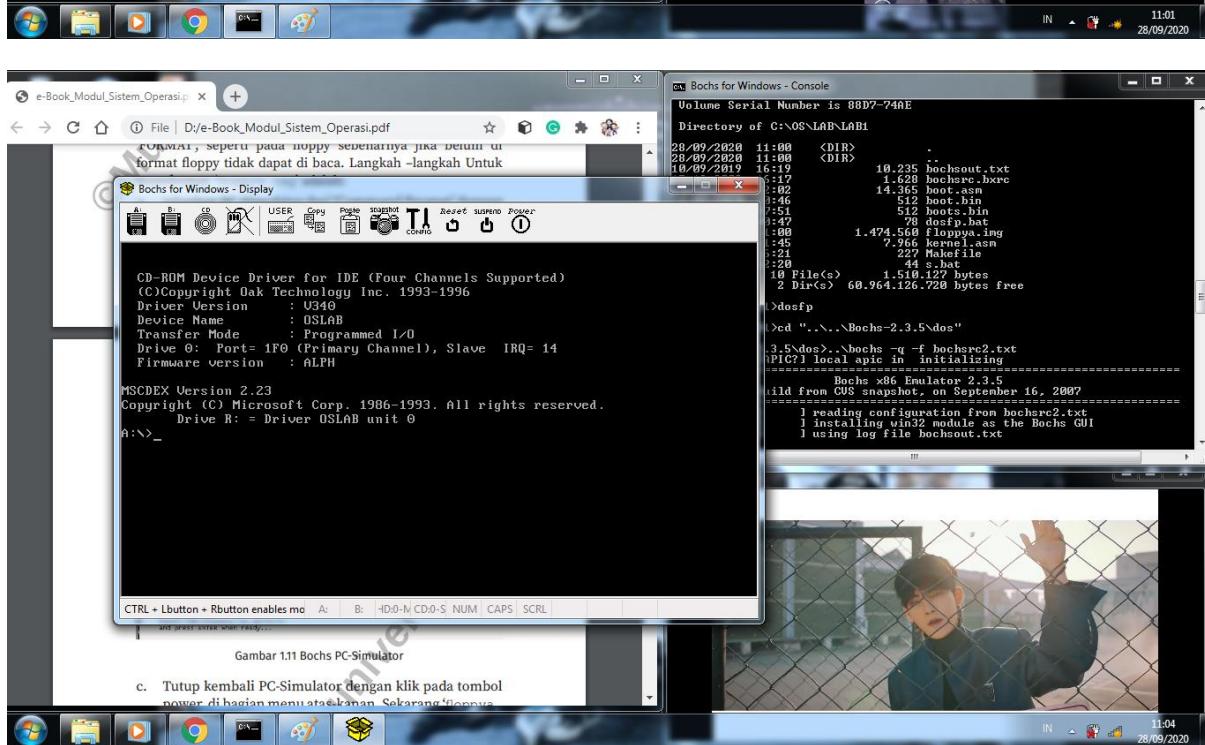
Gambar 1.6 Membuat file image floppy tahap-2





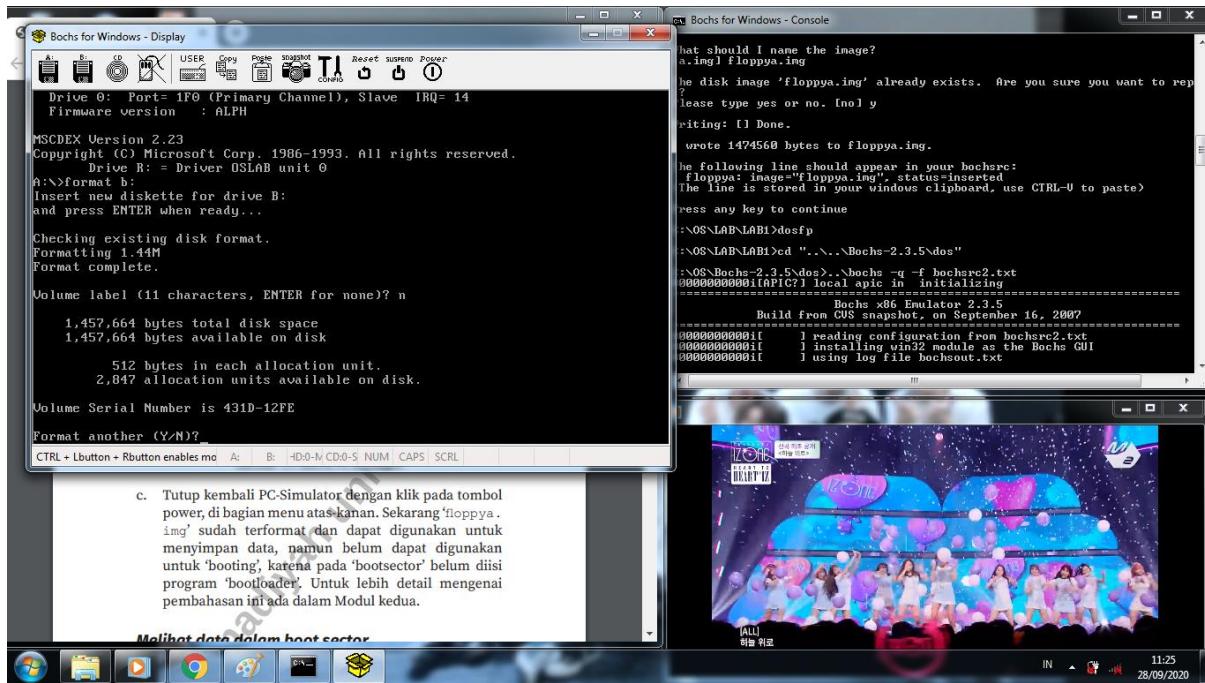
konfigurasi pada 'Bochs' sudah diatur menggunakan nama tersebut, jadi pastikan bahwa nama file imaganya adalah 'floppya.img', pastikan keberadaan file image tersebut dengan perintah 'dir', sudah adakah? Jika belum, anda harus mengulangi proses dari awal dengan memanggil program 'bximage'.

File image floppy ini memiliki tipe Format FAT12, yang memiliki 80 cylinder, 2 head (atas dan bawah), 18 sektor tiap track, jumlah sektor 2880, jumlah byte data tiap sektor 512 byte sehingga total byte  $2880 \times 512 = 1,474,560$  byte (1,44MB). Format floppy secara fisik dapat dilihat pada gambar 1.0, periksa dengan data di atas, samakah? Sistem yang serupa juga digunakan pada harddisk tetapi dengan variasi jumlah cylinder, head dan sector per track (disingkat CHS) yang berbeda. Cylinder adalah jumlah track (garis melingkar) pada



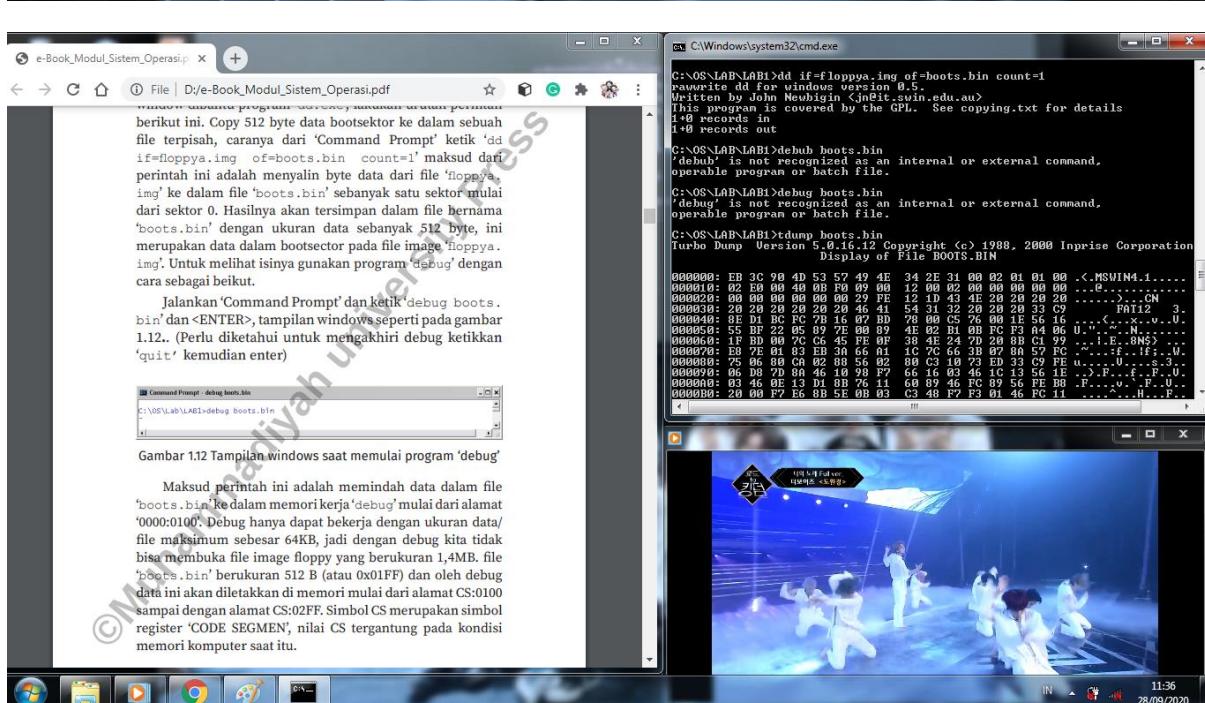
c. Tutup kembali PC-Simulator dengan klik pada tombol power di bagian menu atas kanan. Sekarang 'floppya





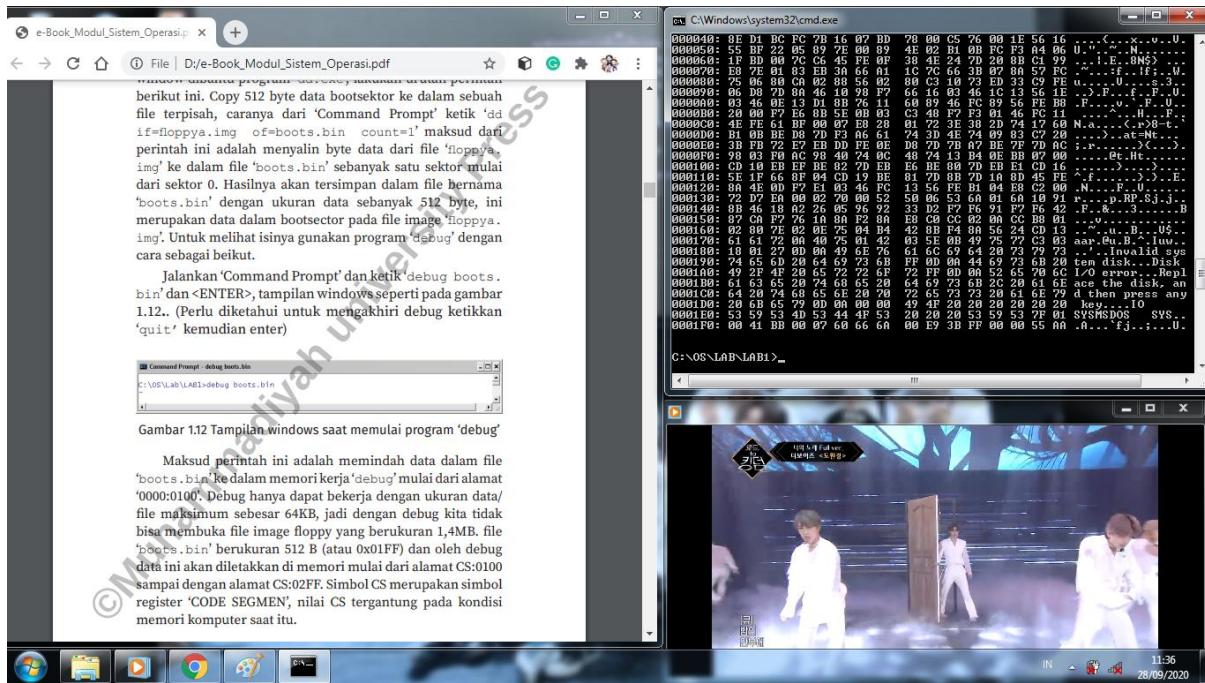
c. Tutup kembali PC-Simulator dengan klik pada tombol power, di bagian menu atas kanan. Sekarang 'floppya.img' sudah terformat dan dapat digunakan untuk menyimpan data, namun belum dapat digunakan untuk 'booting', karena pada 'bootsector' belum diisi program 'bootloader'. Untuk lebih detail mengenai pembahasan ini ada dalam Modul kedua.

Melihat data dalam boot sector



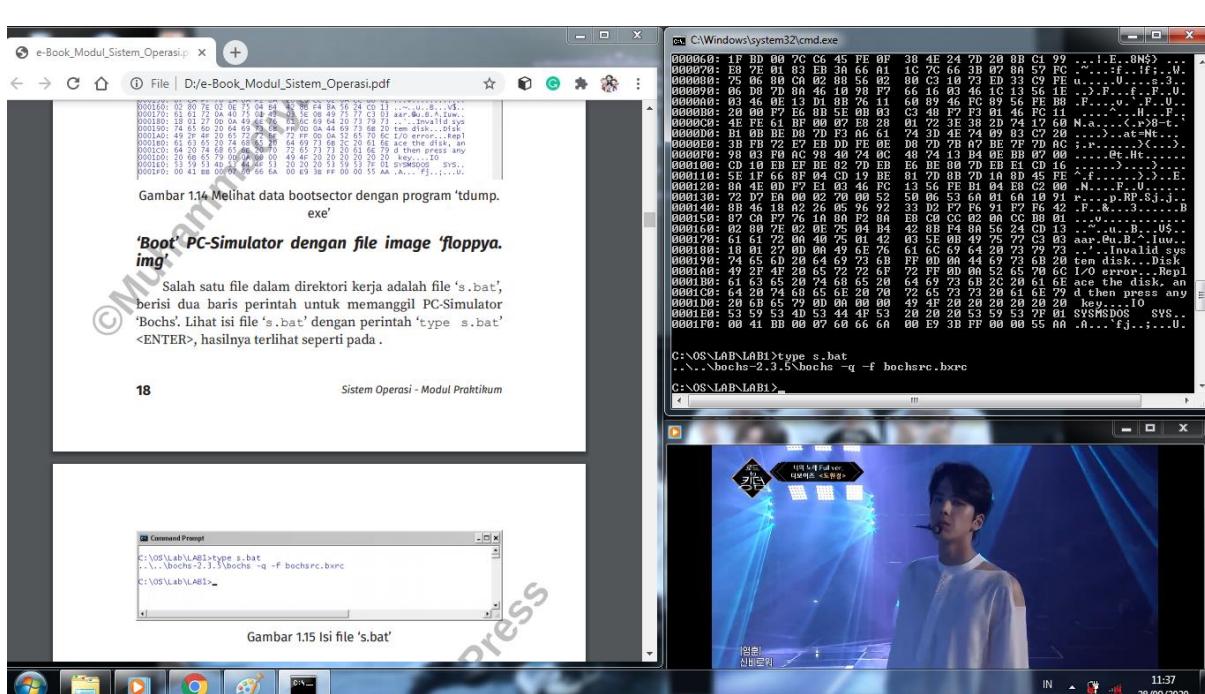
Gambar 1.12 Tampilan windows saat memulai program 'debug'

Maksud perintah ini adalah memindah data dalam file 'boots.bin' ke dalam memori kerja 'debug' mulai dari alamat '0000:0100'. Debug hanya dapat bekerja dengan ukuran data/file maksimum sebesar 64KB, jadi dengan debug kita tidak bisa membuka file image floppy yang berukuran 1,4MB. file 'boots.bin' berukuran 512 B (atau 0x01FF) dan oleh debug data ini akan diletakkan di memori mulai dari alamat CS:0100 sampai dengan alamat CS:02FF. Simbol CS merupakan simbol register 'CODE SEGMENT', nilai CS tergantung pada kondisi memori komputer saat itu.



Gambar 1.12 Tampilan windows saat memulai program 'debug'

Maksud perintah ini adalah memindah data dalam file 'boots.bin' ke dalam memori kerja 'debug' mulai dari alamat '0000:0100'. Debug hanya dapat bekerja dengan ukuran data/ file maksimum sebesar 64KB, jadi dengan debug kita tidak bisa membuka file image floppy yang berukuran 1,4MB. file 'boots.bin' berukuran 512 B (atau 0x01FF) dan oleh debug data ini akan diletakkan di memori mulai dari alamat CS:0100 sampai dengan alamat CS:02FF. Simbol CS merupakan simbol register 'CODE SEGMENT', nilai CS tergantung pada kondisi memori komputer saat itu.

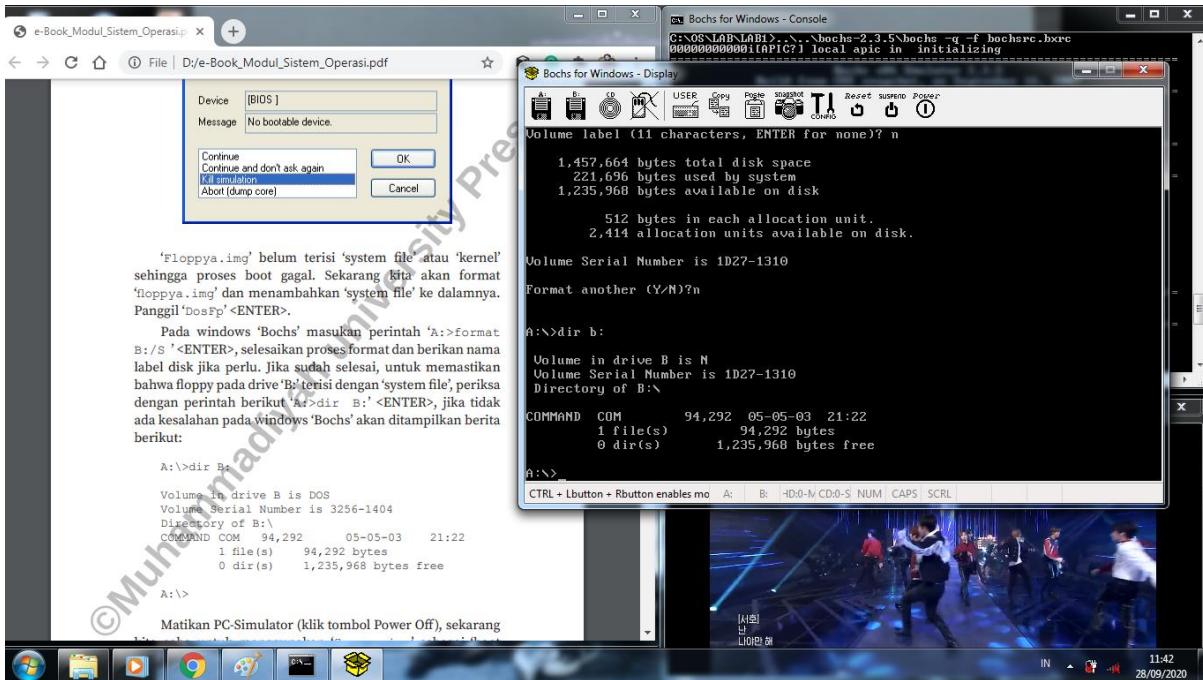
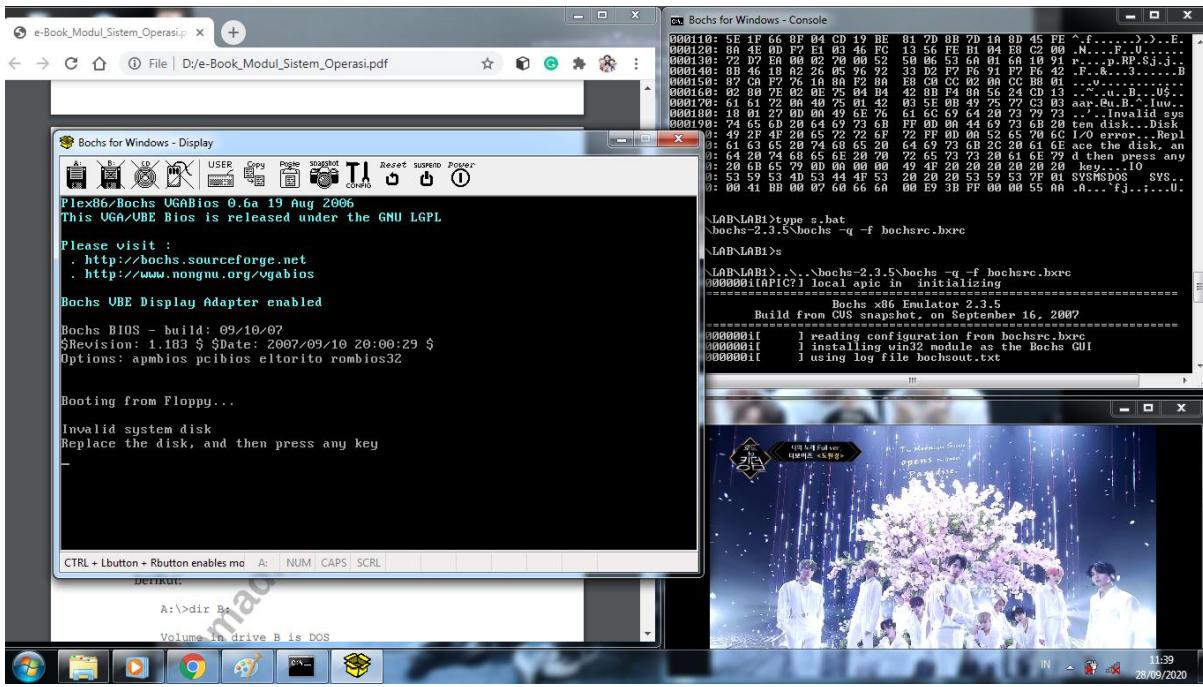


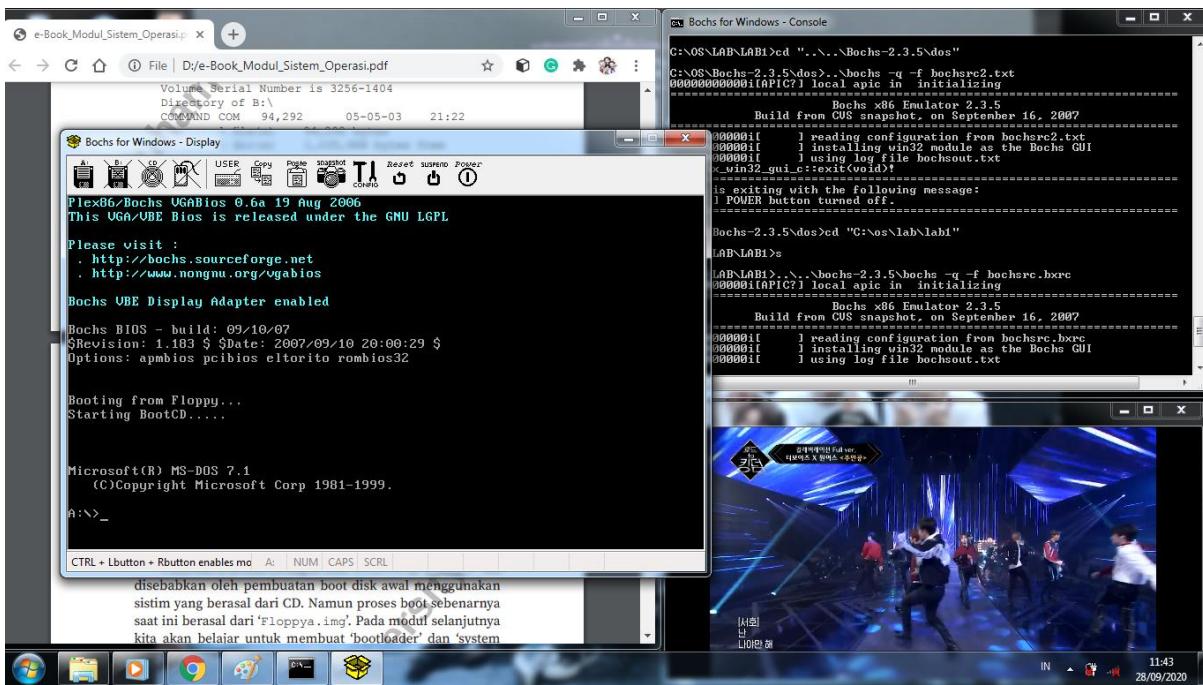
Gambar 1.14 Melihat data bootsector dengan program 'tdump. exe'

#### 'Boot' PC-Simulator dengan file image 'floppy. img'

Salah satu file dalam direktori kerja adalah file '.bat', berisi dua baris perintah untuk memanggil PC-Simulator 'Bochs'. Lihat isi file '.bat' dengan perintah 'type s.bat' <ENTER>, hasilnya terlihat seperti pada .

Gambar 1.15 Isi file 's.bat'





1. Kode **ASCII** (American Standard Codes for International Interchange) adalah kumpulan kode-kode yang dipergunakan untuk mempermudah interaksi antara user dan komputer.

**Kode Standar Amerika untuk Pertukaran Informasi** atau **ASCII** (*American Standard Code for Information Interchange*) merupakan suatu standar internasional dalam kode huruf dan simbol seperti Hex dan Unicode tetapi ASCII lebih bersifat universal, contohnya 124 adalah untuk karakter "|". Ia selalu digunakan oleh komputer dan alat komunikasi lain untuk menunjukkan teks. Kode ASCII sebenarnya memiliki komposisi bilangan biner sebanyak 8 bit. Dimulai dari 00000000 hingga 11111111. Total kombinasi yang dihasilkan sebanyak 256, dimulai dari kode 0 hingga 255 dalam sistem bilangan Desimal.

Karakter	Nilai Unicode (heksadesimal)	Nilai ANSI ASCII (desimal)	Keterangan
NUL	0000	0	Null (tidak tampak)
SOH	0001	1	Start of heading (tidak tampak)
STX	0002	2	Start of text (tidak tampak)
ETX	0003	3	End of text (tidak tampak)
EOT	0004	4	End of transmission (tidak tampak)
ENQ	0005	5	Enquiry (tidak tampak)
ACK	0006	6	Acknowledge (tidak tampak)
BEL	0007	7	Bell (tidak tampak)
BS	0008	8	Menghapus satu karakter di belakang kursor (Backspace)
HT	0009	9	Horizontal tabulation
LF	000A	10	Pergantian baris (Line feed)
VT	000B	11	Tabulasi vertikal

FF	000C	12	Pergantian baris (Form feed)
CR	000D	13	Pergantian baris (carriage return)
SO	000E	14	Shift out (tidak tampak)
SI	000F	15	Shift in (tidak tampak)
DLE	0010	16	Data link escape (tidak tampak)
DC1	0011	17	Device control 1 (tidak tampak)
DC2	0012	18	Device control 2 (tidak tampak)
DC3	0013	19	Device control 3 (tidak tampak)
DC4	0014	20	Device control 4 (tidak tampak)
NAK	0015	21	Negative acknowledge (tidak tampak)
SYN	0016	22	Synchronous idle (tidak tampak)
ETB	0017	23	End of transmission block (tidak tampak)
CAN	0018	24	Cancel (tidak tampak)
EM	0019	25	End of medium (tidak tampak)
SUB	001A	26	Substitute (tidak tampak)
ESC	001B	27	Escape (tidak tampak)
FS	001C	28	File separator
GS	001D	29	Group separator
RS	001E	30	Record separator
US	001F	31	Unit separator
SP	0020	32	Spasi
!	0021	33	Tanda seru (exclamation)
"	0022	34	Tanda kutip dua
#	0023	35	Tanda pagar (kres)
\$	0024	36	Tanda mata uang dolar
%	0025	37	Tanda persen
&	0026	38	Karakter ampersand (&)
'	0027	39	Karakter Apostrof
(	0028	40	Tanda kurung buka
)	0029	41	Tanda kurung tutup
*	002A	42	Karakter asterisk (bintang)
+	002B	43	Tanda tambah (plus)
,	002C	44	Karakter koma
-	002D	45	Karakter hyphen (strip)
.	002E	46	Tanda titik
/	002F	47	Garis miring (slash)
0	0030	48	Angka nol
1	0031	49	Angka satu
2	0032	50	Angka dua
3	0033	51	Angka tiga
4	0034	52	Angka empat
5	0035	53	Angka lima
6	0036	54	Angka enam
7	0037	55	Angka tujuh
8	0038	56	Angka delapan
9	0039	57	Angka sembilan
:	003A	58	Tanda titik dua
;	003B	59	Tanda titik koma
<	003C	60	Tanda lebih kecil
=	003D	61	Tanda sama dengan

>	003E	62	Tanda lebih besar
?	003F	63	Tanda tanya
@	0040	64	A keong (@)
A	0041	65	Huruf latin A kapital
B	0042	66	Huruf latin B kapital
C	0043	67	Huruf latin C kapital
D	0044	68	Huruf latin D kapital
E	0045	69	Huruf latin E kapital
F	0046	70	Huruf latin F kapital
G	0047	71	Huruf latin G kapital
H	0048	72	Huruf latin H kapital
I	0049	73	Huruf latin I kapital
J	004A	74	Huruf latin J kapital
K	004B	75	Huruf latin K kapital
L	004C	76	Huruf latin L kapital
M	004D	77	Huruf latin M kapital
N	004E	78	Huruf latin N kapital
O	004F	79	Huruf latin O kapital
P	0050	80	Huruf latin P kapital
Q	0051	81	Huruf latin Q kapital
R	0052	82	Huruf latin R kapital
S	0053	83	Huruf latin S kapital
T	0054	84	Huruf latin T kapital
U	0055	85	Huruf latin U kapital
V	0056	86	Huruf latin V kapital
W	0057	87	Huruf latin W kapital
X	0058	88	Huruf latin X kapital
Y	0059	89	Huruf latin Y kapital
Z	005A	90	Huruf latin Z kapital
[	005B	91	Kurung siku kiri
\	005C	92	Garis miring terbalik (backslash)
]	005D	93	Kurung sikur kanan
^	005E	94	Tanda pangkat
-	005F	95	Garis bawah (underscore)
`	0060	96	Tanda petik satu
a	0061	97	Huruf latin a kecil
b	0062	98	Huruf latin b kecil
c	0063	99	Huruf latin c kecil
d	0064	100	Huruf latin d kecil
e	0065	101	Huruf latin e kecil
f	0066	102	Huruf latin f kecil f
g	0067	103	Huruf latin g kecil
h	0068	104	Huruf latin h kecil
i	0069	105	Huruf latin i kecil
j	006A	106	Huruf latin j kecil
k	006B	107	Huruf latin k kecil
l	006C	108	Huruf latin l kecil
m	006D	109	Huruf latin m kecil
n	006E	110	Huruf latin n kecil
o	006F	111	Huruf latin o kecil

p	0070	112	Huruf latin p kecil
q	0071	113	Huruf latin q kecil
r	0072	114	Huruf latin r kecil
s	0073	115	Huruf latin s kecil
t	0074	116	Huruf latin t kecil
u	0075	117	Huruf latin u kecil
v	0076	118	Huruf latin v kecil
w	0077	119	Huruf latin w kecil
x	0078	120	Huruf latin x kecil
y	0079	121	Huruf latin y kecil
z	007A	122	Huruf latin z kecil
{	007B	123	Kurung kurawal buka
:	007C	124	Garis vertikal (pipa)
}	007D	125	Kurung kurawal tutup
~	007E	126	Karakter gelombang (tilde)
DEL	007F	127	Delete
	0080	128	Dicadangkan
	0081	129	Dicadangkan
	0082	130	Dicadangkan
	0083	131	Dicadangkan
IND	0084	132	Index
NEL	0085	133	Next line
SSA	0086	134	Start of selected area
ESA	0087	135	End of selected area
	0088	136	Character tabulation set
	0089	137	Character tabulation with justification
	008A	138	Line tabulation set
PLD	008B	139	Partial line down
PLU	008C	140	Partial line up
	008D	141	Reverse line feed
SS2	008E	142	Single shift two
SS3	008F	143	Single shift three
DCS	0090	144	Device control string
PU1	0091	145	Private use one
PU2	0092	146	Private use two
STS	0093	147	Set transmit state
CCH	0094	148	Cancel character
MW	0095	149	Message waiting
	0096	150	Start of guarded area
	0097	151	End of guarded area
	0098	152	Start of string
	0099	153	Dicadangkan
	009A	154	Single character introducer
CSI	009B	155	Control sequence introducer
ST	009C	156	String terminator
OSC	009D	157	Operating system command
PM	009E	158	Privacy message
APC	009F	158	Application program command
	00A0	160	Spasi yang bukan pemisah kata
i	00A1	161	Tanda seru terbalik

¢	00A2	162	Tanda sen (Cent)
£	00A3	163	Tanda Poundsterling
¤	00A4	164	Tanda mata uang ( <i>Currency</i> )
¥	00A5	165	Tanda Yen
⋮	00A6	166	Garis tegak putus-putus ( <i>broken bar</i> )
§	00A7	167	Section sign
„	00A8	168	Diaeresis
©	00A9	169	Tanda hak cipta (Copyright)
ª	00AA	170	Feminine ordinal indicator
«	00AB	171	Left-pointing double angle quotation mark
‐	00AC	172	Not sign
‐	00AD	173	Tanda strip ( <i>hyphen</i> )
®	00AE	174	Tanda merk terdaftar
‐	00AF	175	Macron
°	00B0	176	Tanda derajat
±	00B1	177	Tanda kurang lebih (plus-minus)
²	00B2	178	Tanda kuadrat (pangkat dua)
³	00B3	179	Tanda kubik (pangkat tiga)
ˊ	00B4	180	Acute accent
µ	00B5	181	Micro sign
¶	00B6	182	Pilcrow sign
.	00B7	183	Middle dot

## 2. Bahasa Assembly untuk x86 terbagi menjadi 3 bagian utama yaitu :

### 1. Komentar

Komentar diawali dengan tanda titik koma (;).

; ini adalah komentar

### 2. Label

Label diakhiri dengan tanda titik dua (:).

Contoh: main: ,loop: ,proses: ,keluar:

### 3. Assembler directives

Directives adalah perintah yang ditujukan kepada assembler ketika sedang menerjemahkan program kita ke bahasa mesin.

Directive dimulai dengan tanda titik. **.model** : memberitahu assembler berapa memori yang akan dipakai oleh program kita.

Ada model tiny, model small, model compact, model medium, model large, dan model huge.

**.data** : memberitahu assembler bahwa bagian di bawah ini adalah data program.

**.code** : memberitahu assembler bahwa bagian di bawah ini adalah instruksi program.

**.stack** : memberitahu assembler bahwa program kita memiliki stack.

Program EXE harus punya stack. Kira-kira yang penting itu dulu.

Semua directive yang dikenal assembler adalah: .186 .286 .286c .286p .287 .386 .386c .386p

.387 .486 .486p .8086 .8087

.alpha .break .code .const .continue .cref .data .data? .dosseg .else .elseif .endif .endw .err .err1

.err2 .errb

```
.errdef .errdif .errdifi .erre .erridn .erridni .errnb .errndef .errnz .exit .fardata .fardata? .if .lall .lfcond  
.list .listall .listif .listmacro  
.listmacroall .model .no87 .nocref .nolist .nolistif .nolistmacro .radix .repeat .sall .seq .sfcond  
.stack  
.startup .tfcond .type .until .untilcxz .while .xall .xref .xlist.
```

### Definisi data

**DB** : define bytes. Membentuk data byte demi byte. Data bisa data numerik maupun teks.

catatan: untuk membentuk data string, pada akhir string harus diakhiri tanda dolar (\$).

sintaks: {label} DB {data} contoh: teks1 db "Hello world \$" **DW** : define words.

Membentuk data word demi word (1 word = 2 byte).

sintaks: {label} DW {data} contoh: kucing dw ?, ?, ? ; mendefinisikan tiga slot 16-bit yang isinya don't care

(disimbolkan dengan tanda tanya)

**DD** : define double words. Membentuk data doubleword demi doubleword (4 byte).

sintaks: {label} DD {data} **EQU** : equals. Membentuk konstanta. sintaks: {label} EQU {data}

contoh: sepuluh EQU 10

Ada assembly yang melibatkan bilangan pecahan (floating point), bilangan bulat (integer), DF (define far words),

DQ (define quad words), dan DT (define ten bytes).

### Perpindahan data

**MOV** : move. Memindahkan suatu nilai dari register ke memori, memori ke register, atau register ke register.

sintaks: MOV {tujuan}, {sumber}

contoh:

*mov AX, 4C00h ; mengisi register AX dengan 4C00(hex).*

*mov BX, AX ; menyalin isi AX ke BX. mov CL, [BX] ; mengisi register CL dengan data di memori yang alamatnya ditunjuk BX.*

*mov CL, [BX] + 2 ; mengisi CL dengan data di memori yang alamatnya ditunjuk BX lalu geser maju 2 byte.*

*mov [BX], AX ; menyimpan nilai AX pada tempat di memori yang ditunjuk BX. mov [BX] - 1, 00101110b*

*; menyimpan 00101110(bin) pada alamat yang ditunjuk BX lalu geser mundur 1 byte.*

**LEA** : load effective address. Mengisi suatu register dengan alamat offset sebuah data.

sintaks: LEA {register}, {sumber} contoh: lea DX, teks1 **XCHG** : exchange. Menukar dua buah register langsung.

sintaks: XCHG {register 1}, {register 2} Kedua register harus punya ukuran yang sama.

Bila sama-sama 8 bit (misalnya AH dengan BL) atau sama-sama 16 bit (misalnya CX dan DX), maka pertukaran bisa dilakukan. Sebenarnya masih banyak perintah perpindahan data, misalnya IN, OUT, LODS, LODSB, LODSW, MOVS, MOVSB, MOVSW, LDS, LES, LAHF, SAHF, dan XLAT.

## Operasi logika

**AND** : melakukan bitwise and. sintaks: AND {register}, {angka} AND {register 1}, {register 2} hasil disimpan di register 1.

contoh: mov AL, 00001011b mov AH, 11001000b and AL, AH ;sekarang AL berisi 00001000(bin),

sedangkan AH tidak berubah.

**OR** : melakukan bitwise or. sintaks: OR {register}, {angka} OR {register 1}, {register 2} hasil disimpan di register 1.

**NOT** : melakukan bitwise not (*one's complement*) sintaks: NOT {register} hasil disimpan di register itu sendiri.

**XOR** : melakukan bitwise eksklusif or. sintaks: XOR {register}, {angka} XOR {register 1}, {register 2} hasil disimpan di register 1. Tips: sebuah register yang di-XOR-kan dengan dirinya sendiri akan menjadi berisi nol.

**SHL** : shift left. Menggeser bit ke kiri. Bit paling kanan diisi nol. sintaks: SHL {register}, {banyaknya}

**SHR** : shift right. Menggeser bit ke kanan. Bit paling kiri diisi nol. sintaks: SHR {register}, {banyaknya}

**ROL** : rotate left. Memutar bit ke kiri. Bit paling kiri jadi paling kanan kali ini. sintaks: ROL {register},

{banyaknya} Bila banyaknya rotasi tidak disebutkan, maka nilai yang ada di CL akan digunakan sebagai banyaknya rotasi.

**ROR** : rotate right. Memutar bit ke kanan. Bit paling kanan jadi paling kiri. sintaks: ROR {register}, {banyaknya} Bila banyaknya rotasi tidak disebutkan, maka nilai yang ada di CL akan digunakan sebagai banyaknya rotasi.

Ada lagi : RCL dan RCR.

## Operasi matematika

**ADD** : add. Menjumlahkan dua buah register.

sintaks: ADD {tujuan}, {sumber} operasi yang terjadi: tujuan = tujuan + sumber.

carry (bila ada) disimpan di CF.

**ADC** : add with carry. Menjumlahkan dua register dan carry flag (CF).

sintaks: ADC {tujuan}, {sumber} operasi yang terjadi: tujuan = tujuan + sumber + CF.

carry (bila ada lagi) disimpan lagi di CF.

**INC** : increment. Menjumlah isi sebuah register dengan 1.

Bedanya dengan ADD, perintah INC hanya memakan 1 byte memori sedangkan ADD pakai 3 byte.

sintaks: INC {register}

**SUB** : subtract. Mengurangkan dua buah register.

sintaks: SUB {tujuan}, {sumber} operasi yang terjadi: tujuan = tujuan - sumber.

borrow (bila terjadi) menyebabkan CF bernilai 1.

**SBB** : subtract with borrow. Mengurangkan dua register dan carry flag (CF).

sintaks: SBB {tujuan}, {sumber} operasi yang terjadi: tujuan = tujuan – sumber – CF.

borrow (bila terjadi lagi) menyebabkan CF dan SF (sign flag) bernilai 1.

**DEC** : decrement. Mengurang isi sebuah register dengan 1.

Jika SUB memakai 3 byte memori, DEC hanya memakai 1 byte. sintaks: DEC {register}

**MUL** : multiply. Mengalikan register dengan AX atau AH.

sintaks: MUL {sumber} Bila register sumber adalah 8 bit,

maka isi register itu dikali dengan isi AL, kemudian disimpan di AX.

Bila register sumber adalah 16 bit, maka isi register itu dikali dengan isi AX,

kemudian hasilnya disimpan di DX:AX. Maksudnya, DX berisi high order byte-nya, AX berisi low order byte-nya.

**IMUL** : signed multiply. Sama dengan MUL,

hanya saja IMUL menganggap bit-bit yang ada di register sumber sudah dalam bentuk *two's complement*.

sintaks: IMUL {sumber}

**DIV** : divide. Membagi AX atau DX:AX dengan sebuah register.

sintaks: DIV {sumber} Bila register sumber adalah 8 bit (misalnya: BL), maka operasi yang terjadi: -AX dibagi BL,

-hasil bagi disimpan di AL, -sisa bagi disimpan di AH.

Bila register sumber adalah 16 bit (misalnya: CX), maka operasi yang terjadi: -DX:AX dibagi CX,  
-hasil bagi disimpan di AX, -sisa bagi disimpan di DX.

**IDIV** : signed divide. Sama dengan DIV, hanya saja IDIV menganggap bit-bit yang ada di register sumber sudah dalam bentuk *two's complement*.

sintaks: IDIV {sumber}

**NEG** : negate. Membuat isi register menjadi negatif (*two's complement*).

Bila mau *one's complement*, gunakan perintah NOT. sintaks: NEG {register} hasil disimpan di register itu sendiri.

## Pengulangan

**LOOP** : loop. Mengulang sebuah proses. Pertama register CX dikurangi satu.

Bila CX sama dengan nol, maka looping berhenti. Bila tidak nol, maka lompat ke label tujuan.

sintaks: LOOP {label tujuan} Tips: isi CX dengan nol untuk mendapat jumlah pengulangan terbanyak.

Karena nol dikurang satu sama dengan -1, atau dalam notasi *two's complement* menjadi FFFF(hex) yang sama dengan 65535(dec).

**LOOPE** : loop while equal. Melakukan pengulangan selama CX ≠ 0 dan ZF = 1. CX tetap dikurangi 1 sebelum diperiksa.

sintaks: LOOP {label tujuan}

**LOOPZ** : loop while zero. Identik dengan LOOPE.

**LOOPNE** : loop while not equal.

Melakukan pengulangan selama CX ≠ 0 dan ZF = 0. CX tetap dikurangi 1 sebelum diperiksa.

sintaks: LOOPNE {label tujuan}

**LOOPNZ** : loop while not zero. Identik dengan LOOPNE.

**REP** : repeat. Mengulang perintah sebanyak CX kali. sintaks: REP {perintah assembly} contoh:

`mov CX, 05 rep inc BX`; register BX ditambah 1 sebanyak 5x.

**REPE** : repeat while equal. Mengulang perintah sebanyak CX kali, tetapi pengulangan segera dihentikan bila didapati ZF = 1.

sintaks: REPE {perintah assembly}

**REPZ** : repeat while zero. Identik dengan REPE.

**REPNE** : repeat while not equal. Mengulang perintah sebanyak CX kali, tetapi pengulangan segera dihentikan bila didapati ZF = 0.

sintaks: REPNE {perintah assembly}

**REPNZ** : repeat while not zero. Identik dengan REPNE.

## Perbandingan

**CMP** : compare. Membandingkan dua buah operand. Hasilnya mempengaruhi sejumlah flag register.

sintaks: CMP {operand 1}, {operand 2}. Operand ini bisa register dengan register , register dengan isi memori, atau register dengan angka.

CMP tidak bisa membandingkan isi memori dengan isi memori. Hasilnya adalah:

Kasus	Bila operand 1 < operand 2	Bila operand 1 = operand 2	Bila operand 1 > operand 2
Signed binary	OF = 1, SF = 1, ZF = 0	OF = 0, SF = 0, ZF = 1	OF = 0, SF = 0, ZF = 0
Unsigned binary	CF = 1, ZF = 0	CF = 0, ZF = 1	CF = 0, ZF = 0

## Lompat-lompat

**JMP**: jump. Lompat tanpa syarat. Lompat begitu saja. sintaks: JMP {label tujuan}

**Lompat bersyarat** sintaksnya sama dengan JMP, yaitu perintah jump diikuti label tujuan.

PERINTAH	ARTI	SYARAT	KASUS	KETERANGAN ("OP" = OPERAND)	MENGIKUTI CMP?
<b>JA</b>	jump if above	CF =	unsigned	lompat bila op 1 > op	ya
<b>JNBE</b>	jump if not below or equal	0 $\wedge$ ZF = 0		2	
<b>JB</b>	jump if below	CF =	unsigned	lompat bila op 1 < op	ya
<b>JNAE</b>	jump if not above or equal	1 $\wedge$ ZF = 0		2	
<b>JAE</b>	jump if above or equal	CF = 0 $\vee$ ZF = 1	unsigned	lompat bila op 1 $\geq$ op	ya
<b>JNB</b>	jump if not below			2	
<b>JBE</b>	jump if below or equal	CF = 1 $\vee$ ZF = 1	unsigned	lompat bila op 1 $\leq$ op	ya
<b>JNA</b>	jump if not above			2	
<b>JG</b>	jump if greater	OF = 0 $\wedge$ ZF = 0	signed	lompat bila op 1 > op	ya

<b>JNLE</b>	jump if not less or equal				
<b>JGE</b>	jump if greater or equal	OF = 0 v ZF = 1	signed	lompat bila op 1 ≥ op 2	ya
<b>JNL</b>	jump if not less than				
<b>JL</b>	jump if less than	OF = 1 ∧ ZF = 0	signed	lompat bila op 1 < op 2	ya
<b>JNGE</b>	jump if not greater or equal				
<b>JLE</b>	jump if less or equal	OF = 1 v ZF = 1	signed	lompat bila op 1 ≤ op 2	ya
<b>JNG</b>	jump if not greater				
<b>JE</b>	jump if equal	ZF = 1	keduanya	lompat bila op 1 = op 2	ya
<b>JZ</b>	jump if zero	ZF = 1	keduanya	lompat bila op 1 = op 2	ya
<b>JNE</b>	jump if not equal	ZF = 0	keduanya	lompat bila op 1 ≠ op 2	ya
<b>JNZ</b>	jump if not zero	ZF = 0	keduanya	lompat bila op 1 ≠ op 2	ya
<b>JC</b>	jump if carry	CF = 1	N/A	lompat bila carry flag = 1	tidak
<b>JNC</b>	jump if not carry	CF = 0	N/A	lompat bila carry flag = 0	tidak
<b>JP</b>	jump on parity	PF = 1	N/A	lompat bila parity flag = 1	tidak selalu
<b>JPE</b>	jump on parity even			lompat bila bilangan genap	
<b>JNP</b>	jump on not parity	PF = 0	N/A	lompat bila parity flag = 0	tidak selalu
<b>JPO</b>	jump on parity odd			lompat bila bilangan ganjil	
<b>JO</b>	jump if overflow	OF = 1	N/A	lompat bila overflow flag = 1	tidak
<b>JNO</b>	jump if not overflow	OF = 0	N/A	lompat bila overflow flag = 0	tidak
<b>JS</b>	jump if sign	SF = 1	N/A	lompat bila bilangan negatif	tidak
<b>JCXZ</b>	jump if CX is zero	CX = 0000	N/A	lompat bila CX berisi nol	tidak

## Operasi stack

**PUSH** : push. Menambahkan sesuatu ke stack.

Sesuatu ini harus register berukuran 16 bit (pada 386+ harus 32 bit), tidak boleh angka, tidak boleh alamat memori.

Maka Anda tidak bisa mem-push register 8-bit seperti AH, AL, BH, BL, dan kawan-kawannya.  
sintaks: push {register 16-bit sumber}

contoh: push DX push AX Setelah operasi push, register SP (stack pointer) otomatis dikurangi 2 (karena datanya 2 byte).

Makanya, "top" dari stack seakan-akan "tumbuh turun".

**POP** : pop. Mengambil sesuatu dari stack.

Sesuatu ini akan disimpan di register tujuan dan harus 16-bit. Maka Anda tidak bisa mem-pop menuju AH, AL, dkk.

sintaks: POP {register 16-bit tujuan}

contoh: POP BX Setelah operasi pop, register SP otomatis ditambah 2 (karena 2 byte), sehingga "top" dari stack "naik" lagi.

Tip: karena register segmen tidak bisa diisi langsung nilainya, Anda bisa menggunakan stack sebagai perantaranya.

Contoh kodennya: mov AX, seg teks1 push AX pop DS

**PUSHF** : push flags. Mem-push **semua** isi register flag ke dalam stack.

Biasa dipakai untuk membackup data di register flag sebelum operasi matematika. Sintaks: PUSHF ;(saja).

**POPF** : pop flags. Lawan dari pushf. Sintaks: POPF ;(saja).

**POPA** : pop all general-purpose registers.

Adalah ringkasan dari sejumlah perintah dengan urutan:

*pop DI pop SI pop BP pop SP pop BX pop DX pop CX pop AX*

Urutan sudah ditetapkan seperti itu.

sintaks: POPA ;(saja). Jauh lebih cepat mengetikkan POPA daripada mengetik POP-POP-POP yang banyak itu.

**PUSHA** : push all general-purpose registers. Lawan dari POPA,

dimana PUSHA adalah singkatan dari sejumlah perintah dengan urutan yang sudah ditetapkan:  
*push AX push CX push DX push BX push SP push BP push SI push DI*

## Operasi pada register flag

**CLC** : clear carry flag. Menjadikan CF = 0. Sintaks: CLC ;(saja).

**STC** : set carry flag. Menjadikan CF = 1. Sintaks: STC ;(saja).

**CFC** : complement carry flag. Melakukan operasi NOT pada CF. Yang tadinya 0 menjadi 1, dan sebaliknya.

**CLD** : clear direction flag. Menjadikan DF = 0. Sintaks: CLD ;(saja).

**STD** : set direction flag. Menjadikan DF = 1.

**CLI** : clear interrupt flag. Menjadikan IF = 0, sehingga interrupt ke CPU akan di-disable.

Biasanya perintah CLI diberikan sebelum menjalankan sebuah proses penting yang riskan gagal bila diganggu.

**STI** : set interrupt flag. Menjadikan IF = 1.

Perintah lainnya

**ORG** : origin. Mengatur awal dari program (bagian static data).

Analoginya seperti mengatur dimana letak titik (0, 0) pada koordinat Cartesius.

sintaks: ORG {alamat awal}

Pada program COM (program yang berekstensi .com), harus ditulis "ORG 100h" untuk mengatur alamat mulai dari program pada 0100(hex),

karena dari alamat 0000(hex) sampai 00FF(hex) sudah dipesan oleh sistem operasi (DOS).

**INT** : interrupt. Menginterupsi prosesor.

Prosesor akan:

1. Membackup data registernya saat itu,
2. Menghentikan apa yang sedang dikerjakannya,
3. Melompat ke bagian interrupt-handler (entah dimana kita tidak tahu, sudah ditentukan BIOS dan DOS),
4. Melakukan interupsi,
5. Mengembalikan data registernya,
6. Meneruskan pekerjaan yang tadi ditunda.

sintaks: INT {nomor interupsi}

**IRET** : interrupt-handler return.

Kita bisa membuat interrupt-handler sendiri dengan berbagai cara.

Perintah IRET adalah perintah yang menandakan bahwa interrupt-handler kita selesai, dan prosesor boleh melanjutkan pekerjaan yang tadi tertunda.

**CALL** : call procedure. Memanggil sebuah prosedur.

sintaks: CALL {label nama prosedur}

**RET** : return. Tanda selesai prosedur.

Setiap prosedur harus memiliki RET di ujungnya.

sintaks: RET ;(saja)

**HLT** : halt. Membuat prosesor menjadi tidak aktif.

Prosesor harus mendapat interupsi dari luar atau di-reset supaya aktif kembali.

**Jadi, jangan gunakan perintah HLT untuk mengakhiri program!!**

Sintaks: HLT ;(saja). **NOP** : no operation.

Perintah ini memakan 1 byte di memori tetapi tidak menyuruh prosesor melakukan apa-apa selama 3 clock prosesor.

Berikut contoh potongan program untuk melakukan *delay* selama 0,1 detik pada prosesor Intel 80386 yang berkecepatan 16 MHz.

*mov ECX, 533333334d ;ini adalah bilangan desimal idle: nop loop idle*

**Buat 5 contoh program dalam bahasa assembly**

**1. Program assembly pertama**

```
.model small
.code
org 100h
main :
mov AH, 02
mov DL, 65
int 21h
int 20h
end main
```

Simpan sebagai ‘HurufA.asm’. Kemudian assemble file tadi dengan assembler, misalnya TASM (turbo assembler).

Program kita mengeluarkan output huruf A di layar.

## 2. Program assembly ke 2

Program ini akan menampilkan sebuah string pada console jika dijalankan.

Berikut ini adalah source codenya:

```
.386
.MODEL TINY
CSEG SEGMENT PARA PUBLIC USE16 'CODE'
ASSUME CS:CSEG
org 100h
start:
;;Routine utama

pushfd ;save flag
pushad ;save isi semua GPR
;; Menampilkan karakter dilakukan melalui interrupt 10h, service 0Eh
;; dari bios video card, karakter yang akan ditampilkan diberikan
;; melalui register si
lea si,msg
mov ah,0Eh ;gunakan service 0Eh
mov bl,07h ;warna foreground
xor bh,bh ;gunakan page 0, akan muncul bug tanpa ini
MORE_DIS:mov al,cs:0+[si] ;karakter yang akan ditulis ada pada al
cmp al,'$';'$' menandakan akhir string
je NO_MORE_DIS
inc si
int 10h ;panggil int 10h (serice dari video bios)
jmp MORE_DIS
NO_MORE_DIS:
popad ;restore semua GPR
popfd ;restore semua flags
retn ;return / akhiri program
;;Definisi Variabel
msg:
DB 10,13
DB 10,13
DB 0,0,0,0,0,0,'Your wish is my command',10,13,10,13
DB 0,0,0,0,0,0,0,'Tweaking your chipset...',10,13,'$'
```

```
CSEG ENDS  
end start
```

Source code di atas adalah source code dalam masm (Microsoft Assembler).  
Program di atas akan menghasilkan output:  
Your wish is my command  
Tweaking your chipset...

### 3. Program assembly ke 3

```
section .data
```

```
        string db "Vendor ID adalah 'XXXXXXXXXXXX'", 0xA  
section .text  
global _start  
_start:  
        xor eax,eax  
        cpuid  
        mov edi,string  
        mov [edi+25],ebx  
        mov [edi+29],edx  
        mov [edi+33],ecx  
        mov eax,4  
        mov ebx,1  
        mov ecx,string  
        mov edx,41  
        int 0x80  
        mov eax,1  
        xor ebx,ebx  
        int 0x80
```

**Program ini menghasilkan output berupa Menampilkan ID Vendor prosesor  
Dengan Instruksi Assembly X86**

### 4. Program assembly ke 4

```
.MODEL TINY  
.CODE  
CODE SEGMENT BYTE PUBLIC 'CODE'  
ASSUME CS:CODE, DS:CODE  
ORG 0100H  
MOV AH, 9  
INT 21H  
RET  
DB 'HELLO WORLD$'  
CODE ENDS
```

**Program ini akan menampilkan output tulisan "HELLO WORLD"**

### 5. Program assembly ke 5

```

.486
.MODEL FLAT
.CODE
PUBLIC _myFunc
_myFunc PROC
    ; Subroutine Prologue
    push ebp      ; Save the old base pointer value.
    mov ebp, esp ; Set the new base pointer value.
    sub esp, 4   ; Make room for one 4-byte local variable.
    push edi      ; Save the values of registers that the function
    push esi      ; will modify. This function uses EDI and ESI.
    ; (no need to save EBX, EBP, or ESP)

    ; Subroutine Body
    mov eax, [ebp+8]    ; Move value of parameter 1 into EAX
    mov esi, [ebp+12]   ; Move value of parameter 2 into ESI
    mov edi, [ebp+16]   ; Move value of parameter 3 into EDI

    mov [ebp-4], edi    ; Move EDI into the local variable
    add [ebp-4], esi    ; Add ESI into the local variable
    add eax, [ebp-4]    ; Add the contents of the local variable
                        ; into EAX (final result)

    ; Subroutine Epilogue
    pop esi      ; Recover register values
    pop edi
    mov esp, ebp ; Deallocate local variables
    pop ebp ; Restore the caller's base pointer value
    ret
_myFunc ENDP
END

```

## Arsitektur x86

Ada ALU (arithmetic logic unit), ada register, ada I/O ke system bus, dan ada interkoneksi internal CPU itu sendiri.

Prosesor x86 memiliki 5 kelompok register, yaitu:

- 1. General Purpose Registers** Adalah register yang bisa dipakai untuk berbagai keperluan. Pada prosesor 8086 dan 286, register ini besarnya 16 bit. Register yang ada yaitu:

**AX** : accumulator register. Biasanya dipakai untuk menyimpan hasil hitungan matematika dan untuk menentukan service call.

**BX** : base register. Biasanya dipakai untuk menunjuk indeks alamat memori.

**CX** : counter register. Biasanya dipakai untuk pengulangan (loop).

**DX** : data register. Biasanya dipakai untuk menyimpan data keluar/masuk prosesor, serta dipakai di operasi perkalian dan pembagian.

Register generik ini masing-masing bisa “dipecah” menjadi dua, satu untuk MSB (high-order byte) dan satunya lagi untuk LSB (low-order byte).

Register AX bisa dipecah menjadi AH (AX high) dan AL (AX low), demikian juga ada BH, BL, CH, CL, DH, DL.

Sebagai contoh, bila AX sedang berisi 0110 0111 1101 0011(bin), maka AH berisi 0110 0111(bin) dan AL berisi 1101 0011(bin).

Untuk prosesor 386 ke atas (yang sudah 32 bit), terdapat register EAX (extended AX), EBX, ECX, dan EDX.

Masing-masing kapasitasnya 32 bit.

**2. Segment Registers** adalah register yang tugasnya mencatat blok memori (baca: segmen) yang sedang digunakan.

Bila isinya diubah sembarangan, program bisa kacau.

Register ini kaitannya dengan memori sementara :

**CS** : code segment. Menunjuk segmen memori tempat kode program yang sekarang sedang jalan.

**DS** : data segment. Menunjuk segmen memori tempat data-data program disimpan (seperti pre-defined string,

buffer string, dan konstanta-konstanta).

**SS** : stack segment. Menunjuk segmen memori tempat “top” dari stack saat ini.

“Segmen”nya, bukan “top”nya. Pembahasan tentang stack masih nanti.

**ES** : extra segment. Adalah register “bonus” yang belum tentu dipakai, tergantung programnya. Misalnya untuk menunjuk alamat video memory pada program game yang fokus pada grafis.

Pada prosesor 386 ke atas, ada tambahan extra segment lagi yang bernama **FS** dan **GS**.

Huruf “F” dan “G” hanyalah urutan abjad sesudah “E” (ES), tidak ada arti khususnya.

Semua register segmen, baik di prosesor 16-bit maupun 32-bit, kapasitasnya adalah 16-bit.

**3. Pointer Registers** Adalah register yang berfungsi sebagai pointer.

Isi dari register ini adalah alamat memori, makanya dia dikatakan “menunjuk” (*to point*) ke suatu alamat memori tertentu.

Kapasitas register pointer adalah 16 bit. Karena hanya 16 bit, maka tentu saja tidak semua alamat memori bisa ditunjuknya.

Besar memori yang mampu ditangani hanya  $2^{16}$  byte = 64 KB (saja), sangat kecil.

Padahal prosesor 16 bit bisa menangani memori sampai 1 MB.

Maka dari itu, register pointer ini bekerja berpasangan dengan register segmen menghasilkan alamat memori lengkap,

dan mampu menangani sampai 1 MB.

**SP** : stack pointer. Menunjuk ke “top” dari stack-nya langsung.

Operasi push dan pop akan mengubah isi dari SP. Penjelasan tentang stack masih nanti.

**BP** : base pointer. Menunjuk ke sebuah alamat di bagian data program.

Bekerjasama dengan DS. Biasa dipakai untuk menunjuk elemen array.

**IP** : instruction pointer. Menunjuk ke alamat memori tempat instruksi berikutnya, di bagian kode program.

Anggap saja IP adalah program counter (PC). Register ini diubah otomatis sejalan dengan jalannya program.

Pada prosesor 386 ke atas, terdapat register ESP, EBP, dan EIP yang kapasitasnya 32 bit sehingga mampu menangani memori sampai  $2^{32}$  byte = 4 GB.

**4. Index Registers** Register ini digunakan oleh operasi string dan block transfer di memori.

**SI** : source index.

**DI** : destination index.

Kapasitasnya 16 bit. Pada prosesor 386 ke atas, terdapat ESI dan EDI yang kapasitasnya 32 bit.

**5. Flag Registers** Namanya juga "bendera", register ini berfungsi menandakan suatu keadaan "ya" (mengibarkan bendera)

atau "tidak" (tidak mengibarkan bendera). Tentu saja tidak ada bendera di dalam CPU Anda, yang ada hanyalah bit 1 atau 0. Register bendera ini masing-masing besarnya 1 bit saja.

**OF** : overflow register. Bila terjadi overflow pada operasi matematika, maka OF bernilai 1. Bila tidak, isi 0.

**SF** : sign flag. Jika suatu operasi menghasilkan angka negatif, SF berisi 1.

**ZF** : zero flag. Jika suatu operasi menghasilkan angka nol, ZF berisi 1.

**CF** : carry flag. Berisi bit carry pada operasi penjumlahan atau bit borrow pada operasi pengurangan.

**DF** : direction flag. Menunjukkan arah pembacaan byte (maju atau mundur) pada operasi string.

**PF** : parity flag. Bila angka yang dihitung genap atau ganjil (tergantung sistemnya mau genap apa ganjil), PF berisi 1.

**AF** : auxiliary flag. Berisi 1 setelah penjumlahan dua bilangan BCD. Fungsinya seperti carry flag (CF) setelah bit ke-4 (bukan ke-8)

**TF** : trap flag. Menunjukkan mode debugging on atau off. Ini urusan internal CPU.

**IF** : interrupt flag. Bila IF bernilai 0, maka interupsi ke prosesor akan diabaikan.

Register yang hanya terdapat di 80286 ke atas:

**NT** : nested task.

**IOPL** : I/O protection level. (2 bit)

**PE** : protection enable.

**MP** : monitor co-processor.

**EM** : emulate co-processor.

**TS** : task switched.

**ET** : extention type.

**RF** : resume flag.

**VF** : virtual 8086 mode.

Jadi itulah register yang ada di prosesor Pentium/AMD (x86).

Lebih sedikit dibanding MIPS karena di MIPS ada 32 register dimana 16 diantaranya adalah general purpose,

masih ditambah lagi 12 register general-purpose untuk floating point. Semuanya 32 bit.

Di x86, register general purpose cuma 4, 16 bit saja.

Makanya ketika mendesain program, manfaatkan sumberdaya prosesor yang terbatas ini sebaiknya.

#### Manajemen memori

1. Bagian “reserved” sudah diatur oleh sistem operasi dan tidak bisa kita utak-atik lagi.

Untuk program COM (program DOS yang berekstensi .com), bagian reserved ini sudah pasti ukurannya dan letaknya,

yaitu dari alamat memori 0000(hex) sampai 00FF(hex).

2. Bagian static data berisi data terdefinisi yang tidak berubah, seperti konstanta, string terdefinisi, dan buffer.

3. Bagian program adalah instruksi-instruksi program yang nanti kita tulis dalam bahasa assembly.

Tentu yang dijalankan oleh prosesor bukanlah bahasa assembly-nya, tapi sudah dalam bentuk bahasa mesin.

4. Bagian dynamic data, tempat dimana variabel temporer berada.

Perintah “malloc” di bahasa C memesan tempat di bagian ini.

Dynamic data bisa membesar dan mengecil, tergantung banyaknya data yang diciptakan dan dihapus.

5. Terakhir adalah stack, memori yang pasti disediakan di setiap program.

Dasar dari stack terletak di alamat paling tinggi, dan tumbuh turun ke bawah.

Nah, “top” dari stack bisa saja sajatabrakan dengan dynamic data; jika demikian akan muncul error “stack overflow.”

Memori di komputer kita yang hitungannya sudah gigabyte, dibagi-bagi menjadi beberapa blok (dikenal sebagai segmen).

Satu segmen berukuran 64 KB.

Apa bedanya program COM dan EXE? Program COM maksimal memakai memori hanya **satu** segmen,

yaitu 64 KB (makanya kecil sekali). Zaman DOS dulu (dimana memori kecil dan sangat mahal), banyak sekali program COM.

Sebaliknya, program EXE bisa memakai **lebih dari satu** segmen.

Bisa saja static data-nya banyak sekali, sampai disimpan bersemen-semen.

Stacknya juga barangkali banyak sekali, memakai puluhan segmen.

Coba jalankan Microsoft Word (winword.exe), lalu buka task manager.

Lihat memori yang dipakainya (di bagian private use working set).

Waktu pertama kali dijalankan, di komputer saya tertampil 14280 KB.

Bagi dengan 64, maka Microsoft Word ini memakai 224 segmen di RAM.

#### Bermain dengan alamat memori

Dikenal dua jenis alamat memori. Yang pertama adalah alamat absolut, yaitu alamat yang secara fisik diakses oleh prosesor.

Yang kedua adalah alamat relatif, yaitu alamat yang didasarkan pada pembagian segmen dan pergeseran (offset).

Misalkan kita kembali ke zaman dulu, dimana prosesor masih 16-bit dan memori komputer masih 1 MB.

Satu megabyte berarti 1024 KB, sama dengan 1.048.576 byte. Setiap byte ini punya alamat absolut masing-masing, yaitu dari 00000(hex) sampai FFFF(hex).

Komputer membagi memori 1 MB tadi seperti ini:

Segmen 0000(hex) dimulai dari alamat absolut 00000(hex) sampai 0FFFF(hex) (alamat 0 – 65535).

Segmen 0001(hex) dimulai dari alamat absolut 00010(hex) sampai 1000F(hex) (16 – 65551).

Dan seterusnya sampai segmen FFFF(hex).

Cara mengubah alamat relatif ke alamat absolut:

1. Kalikan angka segmen dengan 16(dec) (= 10(hex))

2. Kemudian jumlahkan dengan offset-nya.

Notasi untuk alamat relatif adalah [segmen]:[offset] dalam heksadesimal. Misal 0000:0001 dan FF87:550A.

Pembagian seperti tadi mengakibatkan adanya tumpang tindih (overlap) antarsegmen.

Ilustrasinya:

0																								
1																								
2																								
3																								
4																								
5																								

Dalam tabel ini, baris adalah segmen, kolom adalah offset.

Tanda x adalah byte yang saat itu ada di memori.

Lihat byte yang saya tandai dengan merah (tanda x warna merah) itu.

Kita boleh bilang bahwa dia terletak di segmen 0002 digeser 6 byte, bukan? Maka alamat relativnya adalah **0002:0006**.

Tapi kita juga boleh bilang kalau dia terletak di segmen 0001 digeser 22 byte (= 16(hex)) bukan?

Kalau begitu alamatnya menjadi **0001:0016**. Baik 0002:0006 maupun 0001:0016 menunjuk ke alamat absolut yang sama, yaitu **00026(hex)**.

Demikian juga byte yang saya tandai dengan warna hijau. Dia bisa diakses dengan:

*Alamat absolut = 0003A(hex)*

*Alamat relatif #1 = 0003:000A*

*Alamat relatif #2 = 0002:001A*

*Alamat relatif #3 = 0001:002A*

Pada alamat relatif #1, “0003” disimpan pada register segmen (entah itu SS, DS, CS, atau ES).

Sedangkan “000A” disimpan pada register pointer (entah itu SP, IP, atau BP).

Ilustrasi di atas sebenarnya kurang tepat, karena saya menggambarkan satu segmen sebesar 16 byte saja.

Kenyataannya, satu segmen adalah 64 KB.