CS231 ระบบคอมพิวเตอร์และภาษาแอสแซมบลี

บทที่ 9: ระบบปฏิบัติการ MS-DOS และการสร้างโปรแกรมประยุกต์

ผู้ช่วยศาสตราจารย์ ดร. ปวีณ เขื่อนแก้ว

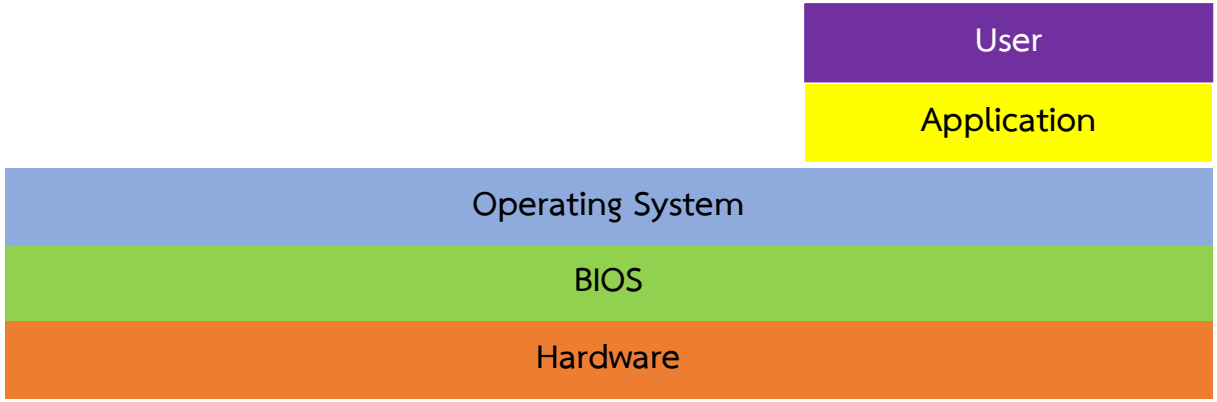
สาขาวิชาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์ มหาวิทยาลัยแม่โจ้

ระบบปฏิบัติการ (operating system) หรือ โอเอส (OS) เป็นระบบซอฟต์แวร์ที่ทำหน้าที่จัดการอุปกรณ์คอมพิวเตอร์และแหล่งซอฟต์แวร์และบริการโปรแกรมคอมพิวเตอร์

ระบบปฏิบัติการมีหน้าที่หลัก ๆ คือ
การจัดสรรทรัพยากรในเครื่องคอมพิวเตอร์ เพื่อให้บริการซอฟต์แวร์ประยุกต์ ในเรื่องการรับส่งและจัดเก็บข้อมูลกับฮาร์ดแวร์ เช่น การส่งข้อมูลภาพไปแสดงผลที่จอภาพ การส่งข้อมูลไปเก็บหรืออ่านจากฮาร์ดดิสก์ การรับส่งข้อมูลในระบบเครือข่าย การส่งสัญญานเสียงไปออกลำโพง หรือจัดสรรพื้นที่ในหน่วยความจำ ตามที่ซอฟต์แวร์ประยุกต์ร้องขอ รวมทั้งทำหน้าที่จัดสรรเวลาการใช้หน่วยประมวลผลกลาง ในกรณีที่อนุญาตให้ซอฟต์แวร์ประยุกต์หลาย ๆ ตัวทำงานพร้อม ๆ กัน

ระบบปฏิบัติการ ช่วยให้ตัว**ซอฟต์แวร์ประยุกต์** ไม่ต้องจัดการเรื่องเหล่านั้นด้วยตนเอง เพียงแค่เรียกใช้บริการจากระบบปฏิบัติการก็พอ ทำให้พัฒนาซอฟต์แวร์ประยุกต์ได้ง่ายขึ้น



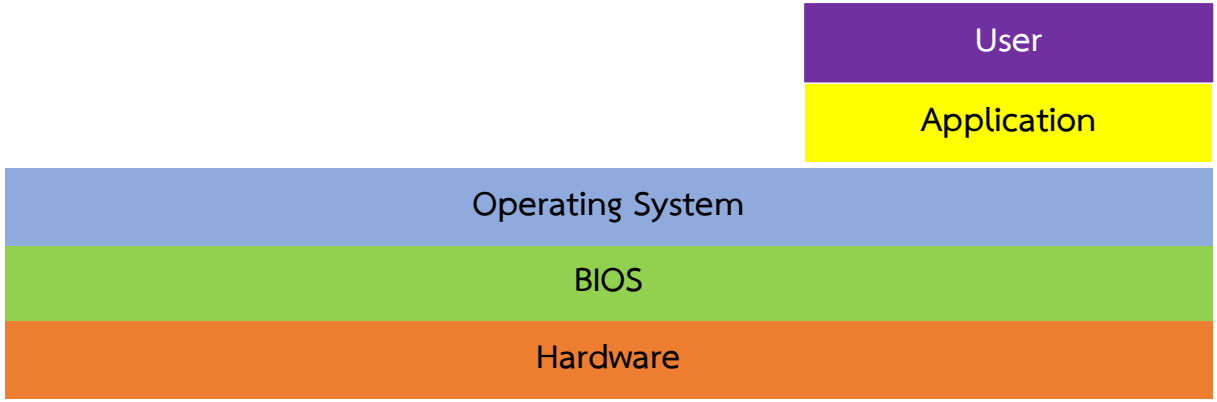
การสร้างโปรแกรมประยุกต์ สามารถเรียกใช้ API (Application Program Interface) จากระบบปฏิบัติการ เพื่อช่วยให้เขียนโปรแกรมได้ ง่าย และสั้นลง

โปรแกรมที่จะรันบนระบบปฏิบัติการได้จะต้อง

- มีรูปแบบตามที่ระบบปฏิบัติการกำหนด (binary file format)
- มีการวางรูปแบบหน่วยความจำตามที่ระบบปฏิบัติการกำหนด (memory model)
- มีจุดเริ่มต้น และจุดสิ้นสุดตามที่กำหนด

การโปรแกรมเพื่อเรียกใช้ API

- เรียกใช้โดยตรง ด้วย Interrupts หรือ Call ไปยังตำแหน่งหน่วยความจำ
- เรียกผ่านชุดพัฒนา (SDK)



MS-DOS : Disk Operating System (ย่อสำหรับระบบปฏิบัติการ Microsoft ดิสก์) เป็นระบบปฏิบัติการสำหรับx86 -based คอมพิวเตอร์ส่วนบุคคลที่ได้รับการพัฒนาโดยส่วนใหญ่ไมโครซอฟท์ โดยรวม MS-DOS การเปลี่ยนชื่อเป็นIBM PC DOSและระบบปฏิบัติการบางระบบที่พยายามเข้ากันได้กับ MS-DOS บางครั้งเรียกว่า "DOS" MS-DOS เป็นระบบปฏิบัติการหลักสำหรับIBM PC ที่เข้ากันได้คอมพิวเตอร์ส่วนบุคคลในช่วงทศวรรษที่ 1980

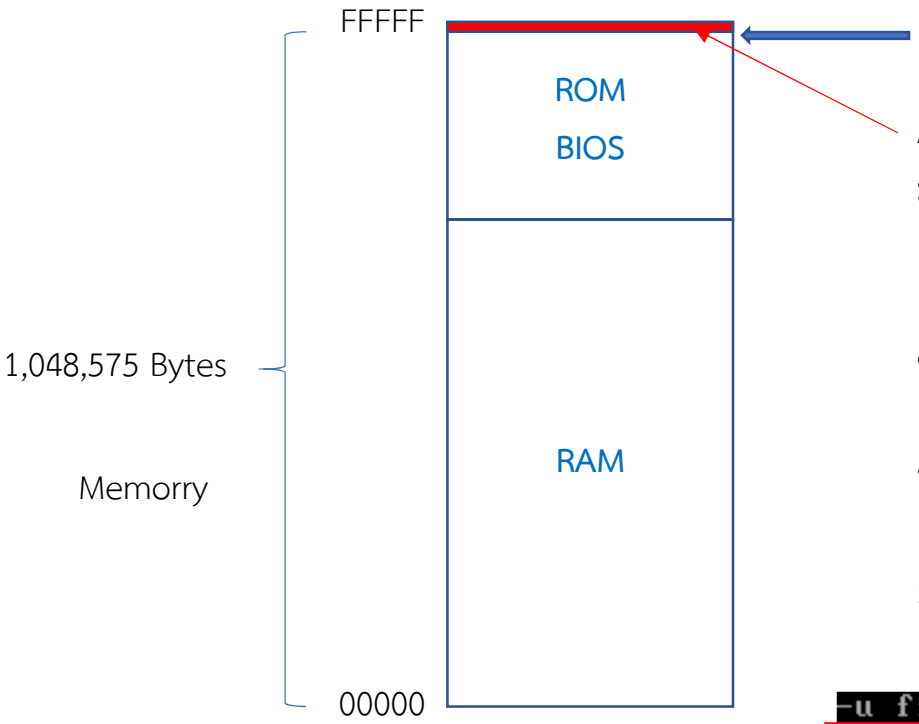
Time-line ที่สำคัญ

- 6-DOS ถูกสร้างขึ้นโดย Tim Paterson ในปี 1980 โดยเลียนแบบมาจากระบบปฏิบัติการ CP/M ที่ทำงานบน CPU 8080/8085 ให้สามารถทำงานบน 8086 ได้
- Microsoft ซื้อสิทธิ 86-DOS ปลายปี 1980
- ต้นปี 1981 Microsoft จ้าง Tim Paterson เข้ามาปรับปรุง 86-DOS ให้ทำงานกัน 8088 (รุ่นราคาถูกของ 8086) ซึ่งถูก IBM ใช้ในการพัฒนา IBM PC XT
- ปลายปี 1981 Microsoft ซื้อ 86-DOS ทั้งหมด และเปลี่ยนชื่อเป็น PC DOS 1.0
- ปี 1982 Microsoft จ้าง **Mark Zbikowski** เข้ามารับช่วงพัฒนา MS-DOS ต่อแทน Tim Paterson
- MS-DOS 7/8 คือ MS-DOS รุ่นสุดท้าย ถูกพัฒนาให้เป็นส่วนหนึ่งของ Windows95 ในปี 1995
- MS-DOS ถูกหยุดพัฒนาในปี 1996 และแทนที่ด้วย WindowsXP

https://hmong.in.th/wiki/MS-DOS

ลำดับการ Boot

1 ไมโครโพรเซสเซอร์ 8086 เมื่อถูกจ่ายไฟเข้าครั้งแรก (Cold boot) จะทำการตั้งค่ารีจิสเตอร์ PC = FFFF0



PC = FFFF0

Address ส่วนนี้ถูก Map ไปยัง ROM
มีพื้นที่ 16 Bytes สำหรับ
Instruction เพื่อใช้ในเปิดเครื่อง

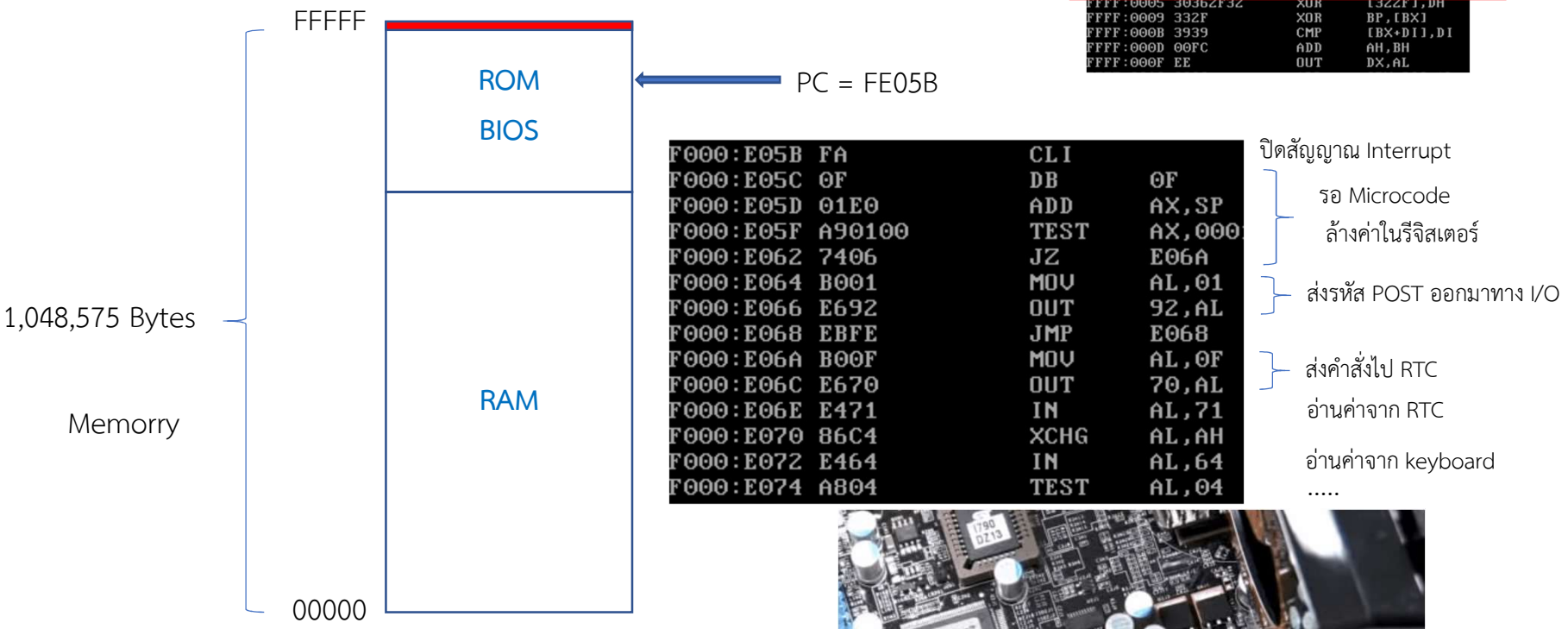
ซึ่ง 16 Bytes ไม่มากพอสำหรับสร้างโปรแกรม
แต่มากพอสำหรับคำสั่ง JMP แบบ Long
พื้นที่ตรงนี้เลยมีคำสั่งเดียว คือ JMP ใน Byte แรก
และ ตำแหน่งเริ่มต้นของโปรแกรม POST (Power on self test) ในอีก 4 Bytes ถัดมา

```
-u ffff:0000
FFFF:0000 EA5BE000F0    JMP     F000:E05B
FFFF:0005 30362F32      XOR     [322F],DH
FFFF:0009 332F          XOR     BP,[BX]
FFFF:000B 3939          CMP     [BX+DI],DI
FFFF:000D 00FC          ADD     AH,BH
FFFF:000F EE            OUT     DX,AL
```

ค่าใน ROM ของ BIOS ของเครื่อง PC

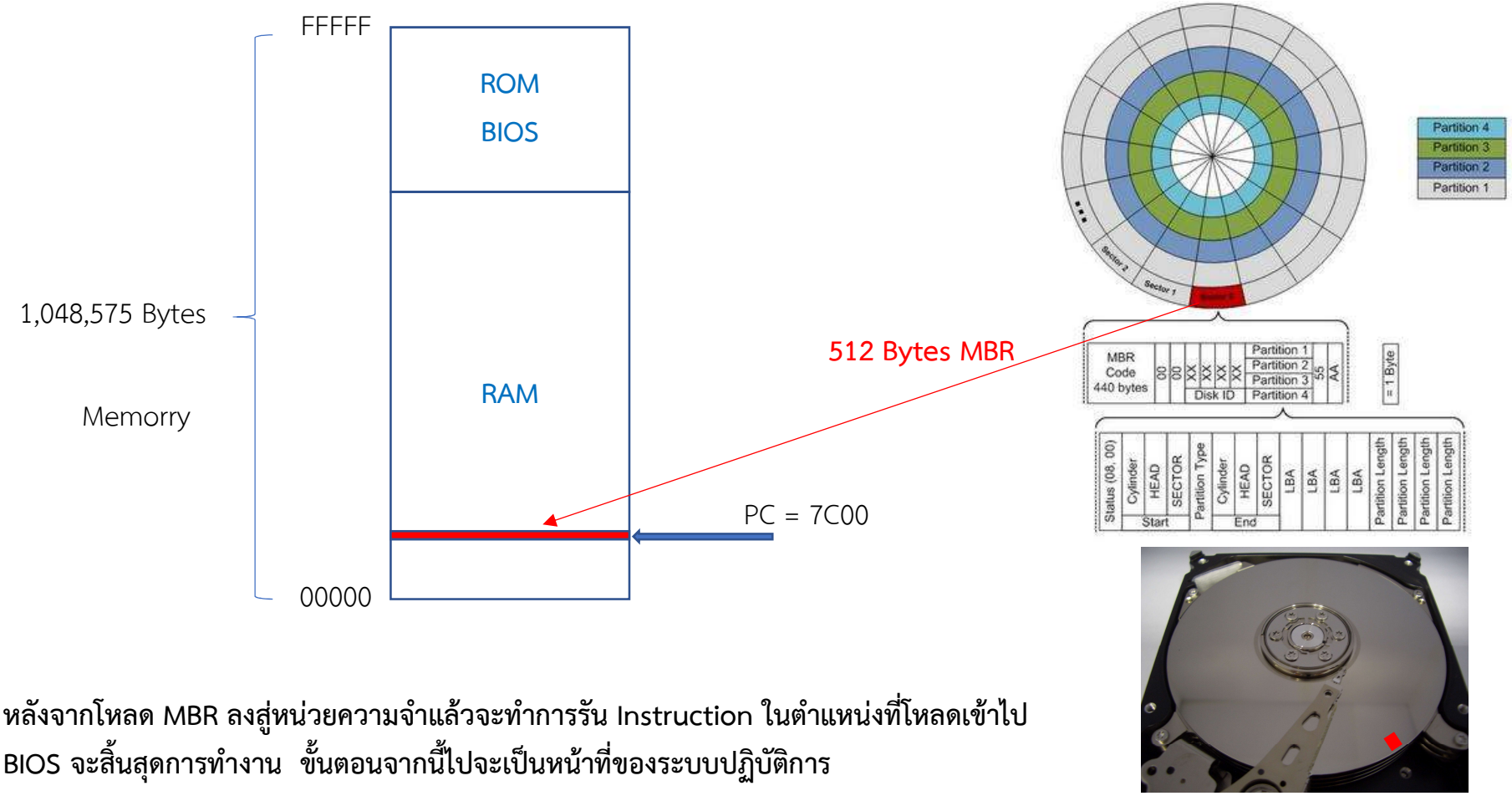
ลำดับการ Boot

2 ไมโครโพรเซสเซอร์ JMP ไปยังตำแหน่งเริ่มต้นของโปรแกรม POST ที่ F000:E05B



ลำดับการ Boot

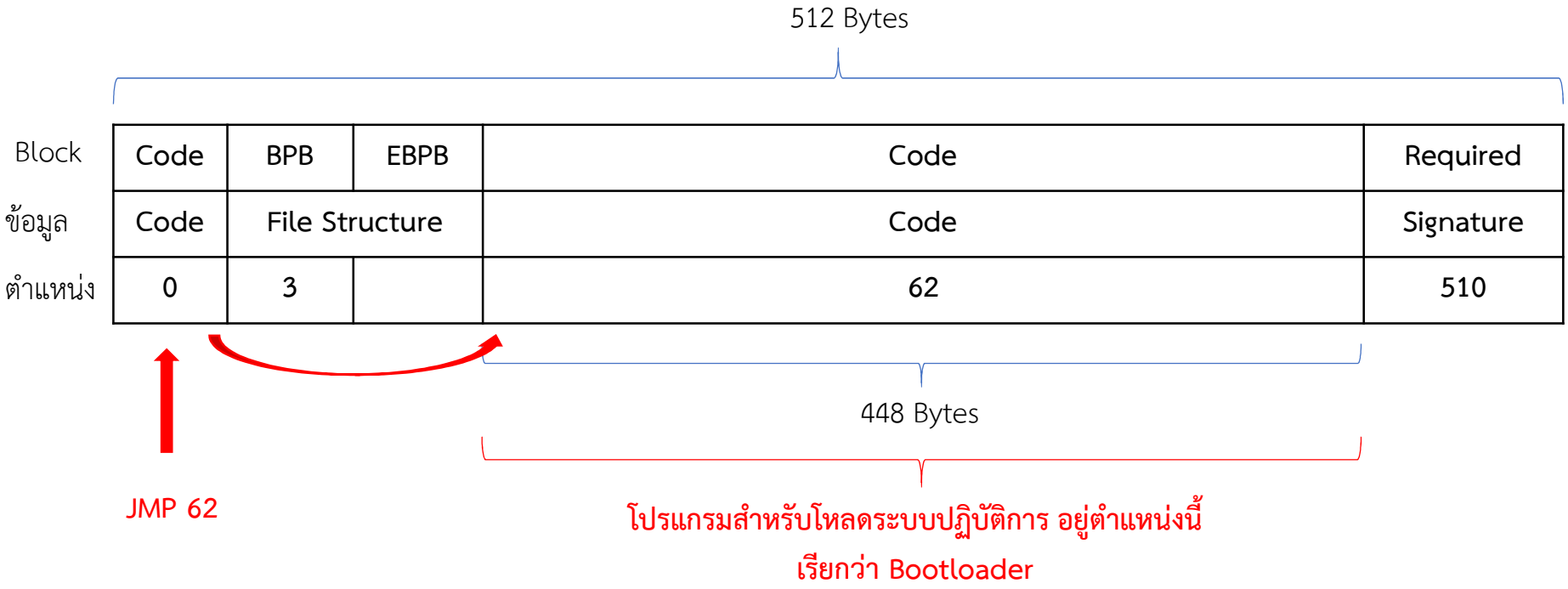
3 เมื่อเสร็จสิ้นการ POST จะทำการตรวจสอบหา Boot device หากพบจะทำการโหลด Sector ลงสู่หน่วยความจำ ตำแหน่ง 7C00
ขั้นตอนนี้เรียกว่า Bootstrap loader



หลังจากโหลด MBR ลงสู่หน่วยความจำแล้วจะทำการรัน Instruction ในตำแหน่งที่โหลดเข้าไป BIOS จะสิ้นสุดการทำงาน ขั้นตอนจากนี้ไปจะเป็นหน้าที่ของระบบปฏิบัติการ

ลำดับการ Boot

4 เมื่อเสร็จสิ้นการ POST จะทำการตรวจสอบหา Boot device หากพบจะทำการโหลด Sector ลงสู่หน่วยความจำ ตำแหน่ง 7C00 ขั้นตอนนี้เรียกว่า Bootstrap loader



ลำดับการ Boot

5 Bootloader ของ MS-DOS มีขึ้นตอนการทำงานดังนี้

1 ตรวจสอบว่าในดิสมีแฟ้ม IO.SYS หรือไม่

2 หากตรวจพบจะทำการโหลดแฟ้มนี้สู่หน่วยความจำ IO.SYS คือ Kernel ของระบบปฏิบัติการ MS-DOS

3 รันโปรแกรม IO.SYS

4 โปรแกรม IO.SYS จะทำหน้าที่โหลด Driver ต่าง ๆเข้าสู่หน่วยความจำ

5 โหลดแฟ้ม MSDOS.SYS เข้าสู่หน่วยความจำ ซึ่งเป็น Kernel ที่รับผิดชอบระบบแฟ้มข้อมูล และการจัดการหน่วยความจำ

6 โหลดแฟ้ม CONFIG.SYS และทำการโหลด driver ที่เหลือซึ่ง user สามารถป้อนเพิ่มได้

7 โหลดแฟ้ม COMMAND.COM เข้าสู่หน่วยความจำ แฟ้มนี้คือโปรแกรม SHELL ที่ใช้ติดต่อกับผู้ใช้

MS-DOS มี API ให้เรียกใช้ได้โดยตรงผ่าน Software Interrupt

Interrupt vector	Description	Version	Notes
20h	Terminate program	1.0+	Implemented in DOS kernel
21h	Main DOS API	1.0+	Implemented in DOS kernel
22h	Program terminate address	1.0+	Return address in calling program
23h	Control-C handler address	1.0+	Default handler is in the command shell (usually COMMAND.COM)
24h	Critical error handler address	1.0+	Default handler is in the command shell (usually COMMAND.COM)
25h	Absolute disk read	1.0+	Implemented in DOS kernel, enhanced in DOS 3.31 to support up to 2 GB partitions
26h	Absolute disk write	1.0+	Implemented in DOS kernel, enhanced in DOS 3.31 to support up to 2 GB partitions
27h	Terminate and stay resident	1.0+	Implemented in COMMAND.COM in DOS 1.0, DOS kernel in DOS 2.0+
28h	Idle callout	2.0+	Called by DOS kernel when waiting for input
29h	Fast console output	2.0+	Implemented by the built-in console device driver or a replacement driver like ANSI.SYS
2Ah	Networking and critical section	3.0+	Called by DOS kernel to interface with networking software
2Bh	Unused		
2Ch	Unused		
2Dh	Unused		
2Eh	Reload transient	2.0+	Implemented in COMMAND.COM
2Fh	Multiplex	3.0+	Implemented in DOS kernel and various programs (PRINT, MSCDEX, DOSKEY, APPEND, etc.) depending on subfunction number

https://en.wikipedia.org/wiki/DOS_API

MS-DOS มี API ให้เรียกใช้ได้โดยตรงผ่าน Software Interrupt

AH	Description	Version
00h	Program terminate	1.0+
01h	Character input	1.0+
02h	Character output	1.0+
03h	Auxiliary input	1.0+
04h	Auxiliary output	1.0+
05h	Printer output	1.0+
06h	Direct console I/O	1.0+
07h	Direct console input without echo	1.0+
08h	Console input without echo	1.0+
09h	Display string	1.0+
0Ah	Buffered keyboard input	1.0+
0Bh	Get input status	1.0+
0Ch	Flush input buffer and input	1.0+
0Dh	Disk reset	1.0+
0Eh	Set default drive	1.0+
0Fh	Open file	1.0+
10h	Close file	1.0+
11h	Find first file	1.0+
12h	Find next file	1.0+
13h	Delete file	1.0+
14h	Sequential read	1.0+
15h	Sequential write	1.0+
16h	Create or truncate file	1.0+
17h	Rename file	1.0+
18h	Reserved	1.0+
19h	Get default drive	1.0+
1Ah	Set disk transfer address	1.0+
1Bh	Get allocation info for default drive	1.0+
1Ch	Get allocation info for specified drive	1.0+
1Dh	Reserved	1.0+
1Eh	Reserved	1.0+
1Fh	Get disk parameter block for default drive	1.0+
20h	Reserved	1.0+
21h	Random read	1.0+
22h	Random write	1.0+
23h	Get file size in records	1.0+
24h	Set random record number	1.0+
25h	Set interrupt vector	1.0+
26h	Create PSP	1.0+
27h	Random block read	1.0+
28h	Random block write	1.0+
29h	Parse filename	1.0+
2Ah	Get date	1.0+
2Bh	Set date	1.0+
2Ch	Get time	1.0+
2Dh	Set time	1.0+
2Eh	Set verify flag	1.0+
2Fh	Get disk transfer address	2.0+
30h	Get DOS version	2.0+
31h	Terminate and stay resident	2.0+
32h	Get disk parameter block for specified drive	2.0+
33h	Get or set Ctrl-Break	2.0+
34h	Get InDOS flag pointer	2.0+
35h	Get interrupt vector	2.0+
36h	Get free disk space	2.0+
37h	Get or set switch character	2.0+
38h	Get or set country info	2.0+
39h	Create subdirectory	2.0+
3Ah	Remove subdirectory	2.0+
3Bh	Change current directory	2.0+
3Ch	Create or truncate file	2.0+
3Dh	Open file	2.0+
3Eh	Close file	2.0+
3Fh	Read file or device	2.0+
40h	Write file or device	2.0+
41h	Delete file	2.0+
42h	Move file pointer	2.0+
43h	Get or set file attributes	2.0+
44h	I/O control for devices	2.0+
45h	Duplicate handle	2.0+
46h	Redirect handle	2.0+
47h	Get current directory	2.0+
48h	Allocate memory	2.0+
49h	Release memory	2.0+
4Ah	Reallocate memory	2.0+
4Bh	Execute program	2.0+
4Ch	Terminate with return code	2.0+
4Dh	Get program return code	2.0+
4Eh	Find first file	2.0+
4Fh	Find next file	2.0+
50h	Set current PSP	2.0+
51h	Get current PSP	2.0+
52h	Get DOS internal pointers (SYSVARS)	2.0+
53h	Create disk parameter block	2.0+
54h	Get verify flag	2.0+
55h	Create program PSP	2.0+
56h	Rename file	2.0+
57h	Get or set file date and time	2.0+
58h	Get or set allocation strategy	2.11+
59h	Get extended error info	3.0+
5Ah	Create unique file	3.0+
5Bh	Create new file	3.0+
5Ch	Lock or unlock file	3.0+
5Dh	File sharing functions	3.0+
5Eh	Network functions	3.0+
5Fh	Network redirection functions	3.0+
60h	Qualify filename	3.0+
61h	Reserved	3.0+
62h	Get current PSP	3.0+
63h	Get DBCS lead byte table pointer	3.0+
64h	Set wait for external event flag	3.2+
65h	Get extended country info	3.3+
66h	Get or set code page	3.3+
67h	Set handle count	3.3+
68h	Commit file	3.3+
69h	Get or set media id	4.0+
6Ah	Commit file	4.0+
6Bh	Reserved	4.0+
6Ch	Extended open/create file	4.0+

https://en.wikipedia.org/wiki/DOS_API

EMU8086 ไม่มี Console และ Shell จึงไม่รองรับ MS-DOS API ทุกตัว
API ที่เรียกใช้ได้จึงมีเพียงเท่านี้

INT 20h

INT 21h
INT 21h/01h
INT 21h/02h
INT 21h/05h
INT 21h/06h
INT 21h/07h
INT 21h/09h
INT 21h/0Ah
INT 21h/0Bh
INT 21h/0Ch
INT 21h/0Eh
INT 21h/19h
INT 21h/25h
INT 21h/2Ah
INT 21h/2Ch

INT 21h/35h
INT 21h/39h
INT 21h/3Ah
INT 21h/3Bh
INT 21h/3Ch
INT 21h/3Dh
INT 21h/3Eh
INT 21h/3Fh
INT 21h/40h
INT 21h/41h
INT 21h/42h
INT 21h/47h
INT 21h/4Ch
INT 21h/56h

INT 33h/0000h
INT 33h/0001h
INT 33h/0002h
INT 33h/0003h

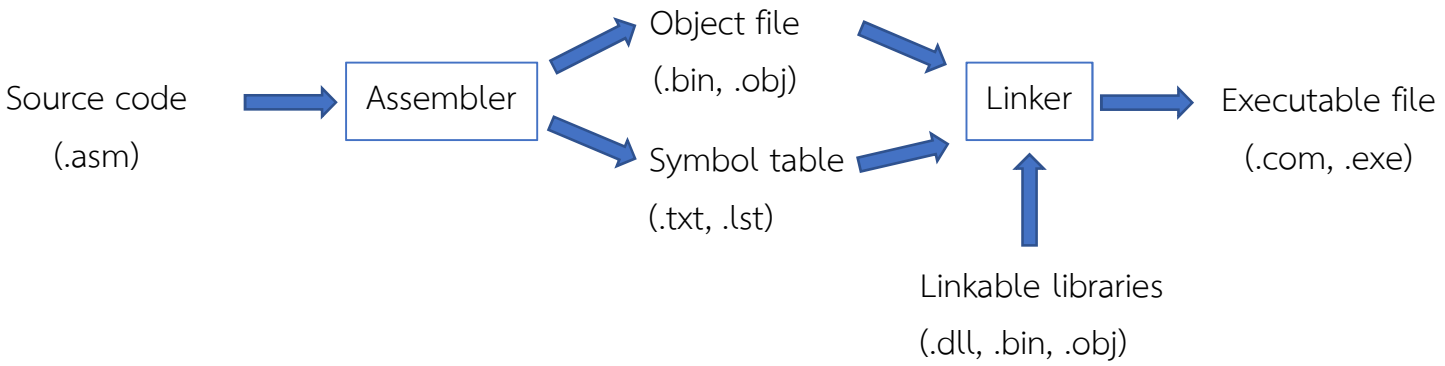
http://www.ablmcc.edu.hk/~scy/CIT/8086_bios_and_dos_interrupts.htm

เมื่อผู้ใช้สั่งให้ MS-DOS ทำการรัน Application ตัวระบบปฏิบัติการจะทำการโหลด แฟ้มโปรแกรม (**Executable file**) เข้าสู่หน่วยความจำ และจัดการย้ายตำแหน่ง Segment ของโปรแกรม Stack และตำแหน่งเริ่มต้นการทำงาน ในรูปแบบที่ MS-DOS สามารถบริหารจัดการได้

แฟ้มที่ได้จากการแปลภาษา Assembly โดย **Assembler** เรียกว่า **Object file** ยังไม่สามารถทำงานได้ทันที เนื่องจาก MS-DOS จะไม่ทราบตำแหน่งของ Segment และข้อมูลอื่น ๆ ที่จำเป็นในการควบคุมการโหลดโปรแกรม เข้าสู่หน่วยความจำ

Object file จะต้องถูกเติมรายละเอียดที่จำเป็น เพื่อให้ MS-DOS สามารถโหลดโปรแกรมเข้าสู่หน่วยความจำได้อย่างถูกต้อง แฟ้มโปรแกรมชนิดนี้เรียกว่า Executable file

โปรแกรมที่ทำหน้าที่แปลง Object file ไปเป็น Executable file คือโปรแกรม **Linker**



MS-DOS รองรับ **Executable file 2 แบบ คือ .com และ .exe**

.com มาจากคำว่า command เหมาะสำหรับโปรแกรมที่มีขนาดเล็ก โปรแกรมทั้งหมดอยู่ใน Segment เดียวกัน โดยเริ่มทำงานที่ offset= 100h ข้อมูล และ stack อยู่รวมกันใน segment เดียว ขณะรัน MS-DOS จะโหลดข้อมูลทั้งแฟ้มเข้าสู่หน่วยความทั้งแฟ้ม ข้อมูลในแฟ้มเหมือนกับการคัดลองโปรแกรมจากหน่วยความจำลงในแฟ้มข้อมูล

ข้อดี

- โปรแกรมได้ง่าย ส่วนประกอบต่าง ๆ ของโปรแกรมอยู่ใน segment เดียวกันหมด
- ทำงานเร็ว เพราะโปรแกรมทั้งแฟ้มถูกโหลดเข้าสู่หน่วยความจำขณะรัน
- ไม่จำเป็นต้องใช้ Linker สามารถสร้างจาก Image ของโปรแกรมในหน่วยความจำได้โดยตรง

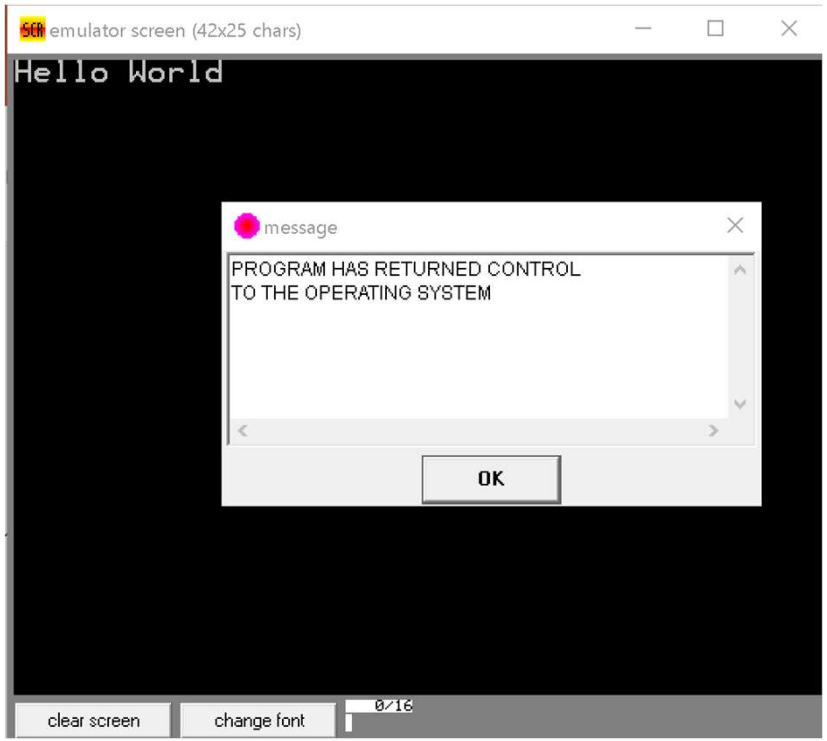
ข้อเสีย

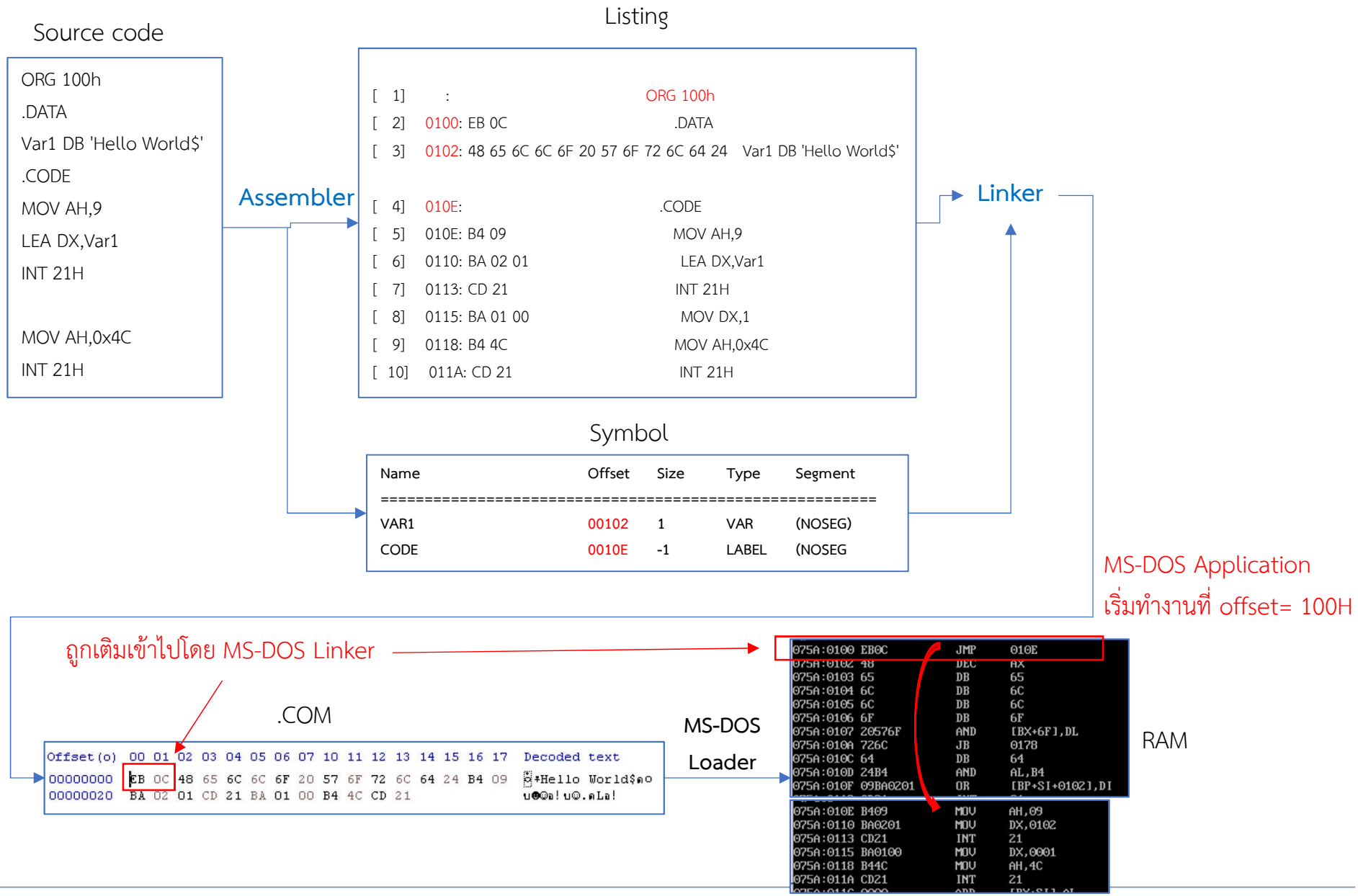
- โปรแกรมขนาดโตเกิน 64kB ไม่ได้ เพราะขนาดจะเกิน 1 segment
- ไม่มี header ไม่รองรับ meta-data

โปรแกรมทั้งหมดที่เราเขียนกันมาในวิชานี้ คือแฟ้ม .com

ตัวอย่างโครงสร้างของโปรแกรมแบบ .com

```
ORG 100h                ; โปรแกรมเริ่มต้นที่ OFFSET = 100H
.DATA                   ; ตำแหน่งนี้คือเป็นข้อมูล
Var1 DB 'Hello World$'
.CODE                   ; ตำแหน่งนี้คือโปรแกรม
MOV AH,9                ; เรียก API สำหรับแสดงสายอักขระ
LEA DX,Var1
INT 21H

MOV AH,0x4C             ; เรียก API จบโปรแกรม
INT 21H
```




Source code
ORG 100h
.DATA
Var1 DB 'Hello World$'
.CODE
MOV AH,9
LEA DX,Var1
INT 21H
MOV AH,0x4C
INT 21H
Assembler
Listing
[1] : ORG 100h
[2] 0100: EB 0C .DATA
[3] 0102: 48 65 6C 6C 6F 20 57 6F 72 6C 64 24 Var1 DB 'Hello World$'
[4] 010E: .CODE
[5] 010E: B4 09 MOV AH,9
[6] 0110: BA 02 01 LEA DX,Var1
[7] 0113: CD 21 INT 21H
[8] 0115: BA 01 00 MOV DX,1
[9] 0118: B4 4C MOV AH,0x4C
[10] 011A: CD 21 INT 21H
Linker
Symbol
Name Offset Size Type Segment
VAR1 00102 1 VAR (NOSEG)
CODE 0010E -1 LABEL (NOSEG
MS-DOS Application
เริ่มทำงานที่ offset= 100H
ถูกเติมเข้าไปโดย MS-DOS Linker
.COM
Offset(o) 00 01 02 03 04 05 06 07 10 11 12 13 14 15 16 17 Decoded text
00000000 EB 0C 48 65 6C 6C 6F 20 57 6F 72 6C 64 24 B4 09
00000020 BA 02 01 CD 21 BA 01 00 B4 4C CD 21
MS-DOS
Loader
075A:0100 EB0C JMP 010E
075A:0102 48 DEC AX
075A:0103 65 DB 65
075A:0104 6C DB 6C
075A:0105 6C DB 6C
075A:0106 6F DB 6F
075A:0107 20576F AND [BX+6F],DL
075A:010A 726C JB 0178
075A:010C 64 DB 64
075A:010D 24B4 AND AL,B4
075A:010F 09BA0201 OR [BP+SI+0102],DI
075A:010E B409 MOV AH,09
075A:0110 BA0201 MOV DX,0102
075A:0113 CD21 INT 21
075A:0115 BA0100 MOV DX,0001
075A:0118 B44C MOV AH,4C
075A:011A CD21 INT 21
RAM

MS-DOS รองรับ **Executable file 2 แบบ คือ .com และ .exe**

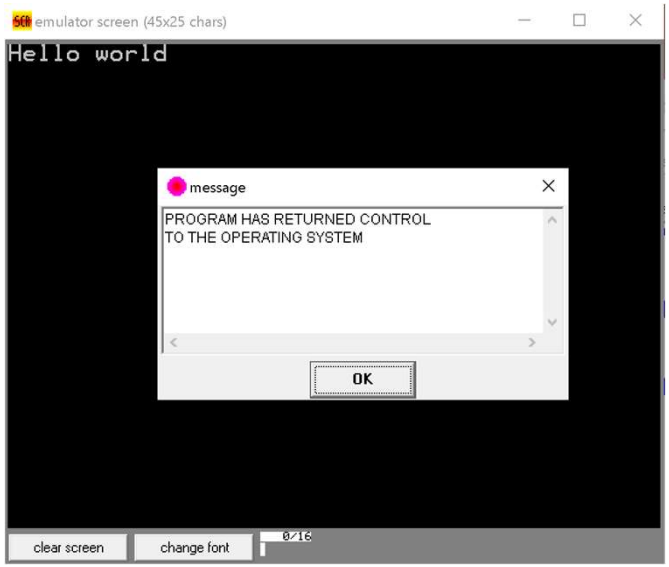
.exe มาจากคำว่า executable เหมาะสำหรับโปรแกรมที่มีขนาดใหญ่ โปรแกรมสามาโหลดเข้าสู่หน่วยความจำได้หลาย Segment หรือโหลดทีละ segment ขณะรันก็ได้ รองรับ Meta-data มีหลาย Sub-format รองรับ static link และ dynamic link
แยก segment ข้อมูล และย้ายตำแหน่งโปรแกรมขณะรันได้
.exe สำหรับ MS-DOS จะเป็นรูปแบบ EXE MZ
.exe สำหรับ Windows จะเป็นรูปแบบ EXE PE ซึ่งรองรับการรวม Resource เช่น เสียง และรูปภาพ icon
ข้อดี
- รองรับโปรแกรมขนาดใหญ่
- รองรับการทำงานที่ซับซ้อน
- ไม่จำเป็นต้องรวม object ทั้งหมดไว้ในแฟ้มเดียว

ข้อเสีย
- ต้องมีโปรแกรมช่วยในการสร้าง (Linker) ไม่สามารถใช้วิธีคัดลอกโปรแกรมลงมาจากหน่วยความจำได้

ตัวอย่างโครงสร้างของโปรแกรมแบบ .exe

โปรแกรมเมอร์ไม่ต้องกลัวว่า stack จะขยายมาชนกับ code เพราะอยู่คนละ segment กับ code และอยู่ segment ล่าง

```
data SEGMENT                นิยาม Segment ชื่อ data
    Var1 DB "Hello world$"
ENDS
stack SEGMENT               นิยาม Segment ชื่อ stack
    DW   128  DUP(0)
ENDS
code1 SEGMENT               นิยาม Segment ชื่อ code1
start:                      ระบุให้เป็นตำแหน่งเริ่มต้น
    MOV AX, data
    MOV DS, AX              ย้ายตำแหน่ง Segment ข้อมูลไปอยู่ที่ data
    MOV ES, AX

    LEA DX, Var1
    MOV AH, 9               เรียก DOS API สำหรับแสดงสายอักขระ
    INT 21H

    MOV AH, 0x4C
    INT 21H                 เรียก DOS API สำหรับจบโปรแกรม
ENDS
end start                   ระบุตำแหน่งให้ Assembler หยุดทำการแปล
```



ตัวอย่างโครงสร้างของโปรแกรมแบบ .exe

Source code

```
data SEGMENT
    Var1 DB "Hello world$"
ENDS
stack SEGMENT
    DW   128  DUP(0)
ENDS
code1 SEGMENT
start:
    MOV AX, data
    MOV DS, AX
    MOV ES, AX

    LEA DX, Var1
    MOV AH, 9
    INT 21H

    MOV AH, 0x4C
    INT 21H
ENDS
end start
```

Assembler

Listing

```
[  1]     :                          data SEGMENT
[  2]  0000: 48 65 6C 6C 6F 20 77 6F 72 6C 64 24   Var1 DB "Hello world$"
[  3]     :                          ENDS
[  4]     :
[  5]     :                          stack SEGMENT
[  6]  0010: 00 00 00 00 00 00 00 00 00 00 00 00 00   DW   128  DUP(0)
[  7]     :                          ENDS
[  8]     :
[  9]     :                          code1 SEGMENT
[ 10]  0110:                           start:
[ 11]  0110: B8 00 00                    MOV AX, data
[ 12]  0113: 8E D8                       MOV DS, AX
[ 13]  0115: 8E C0                       MOV ES, AX
[ 14]     :
[ 15]  0117: BA 00 00                    LEA DX, Var1
[ 16]  011A: B4 09                       MOV AH, 9
[ 17]  011C: CD 21                       INT 21H
[ 18]     :
[ 19]  011E: B4 4C                       MOV AH, 0x4C
[ 20]  0120: CD 21                       INT 21H
[ 21]     :                          ENDS
[ 22]     :
[ 23]     :                          end start
[ 24]     :
[ 25]     :
```

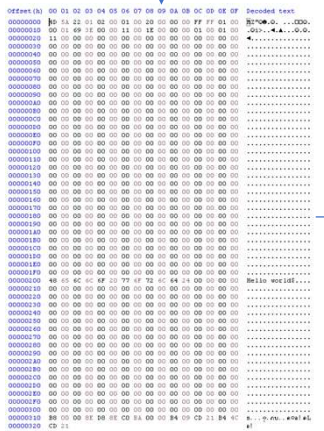
Meta-data

```
0000: 4D    -  exe signature (M)
0001: 5A    -  exe signature (Z)
0002: 22    -  bytes on last page (l.byte)
0003: 01    -  bytes on last page (h.byte)
0004: 02    -  512 byte pages in file (l.byte)
0005: 00    -  512 byte pages in file (h.byte)
0006: 01    -  relocations (l.byte)
0007: 00    -  relocations (h.byte)
0008: 20    -  paragraphs in header (l.byte)
0009: 00    -  paragraphs in header (h.byte)
000A: 00    -  minimum memory (l.byte)
000B: 00    -  minimum memory (h.byte)
000C: FF    -  maximum memory (l.byte)
000D: FF    -  maximum memory (h.byte)
000E: 01    -  SS - stack segment (l.byte)
000F: 00    -  SS - stack segment (h.byte)
0010: 00    -  SP - stack pointer (l.byte)
0011: 01    -  SP - stack pointer (h.byte)
0012: 69    -  check sum (l.byte)
0013: 3E    -  check sum (h.byte)
0014: 00    -  IP - instruction pointer (l.byte)
0015: 00    -  IP - instruction pointer (h.byte)
0016: 11    -  CS - code segment (l.byte)
0017: 00    -  CS - code segment (h.byte)
0018: 1E    -  relocation table adress (l.byte)
0019: 00    -  relocation table adress (h.byte)
001A: 00    -  overlay number (l.byte)
001B: 00    -  overlay number (h.byte)
001C: 01    -  signature (l.byte)
001D: 00    -  signature (h.byte)
001E: 01    -  relocation table - offset inside segment (l.byte)
001F: 00    -  relocation table - offset inside segment (h.byte)
0020: 11    -  relocation table - segment anchor (l.byte)
0021: 00    -  relocation table - segment anchor (h.byte)
0022 to 01FF -  reserved relocation area  (00)
```

Linker

Symbol

Name	Offset	Size	Type	Segment
DATA	00000	-5	SEGMENT	(ITSELF)
VAR1	00000	1	VAR	data
STACK	00001	-5	SEGMENT	(ITSELF)
CODE1	00011	-5	SEGMENT	(ITSELF)
START	00000	-1	LABEL	code1



MS-DOS Loader

```
-u
077B:0000 B86A07      MOV     AX,076A
077B:0003 8ED8        MOV     DS,AX
077B:0005 8EC0        MOV     ES,AX
077B:0007 BA0000      MOV     DX,0000
077B:000A B409        MOV     AH,09
077B:000C CD21        INT     21
077B:000E B44C        MOV     AH,4C
077B:0010 CD21        INT     21
```

```
076A:0000  48 65 6C 6C 6F 20 77 6F-72 6C 64 24 00 00 00 00   Hello world$....
076A:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

RAM

ในการโปรแกรมโดยใช้ API ของระบบปฏิบัติการ นอกจากการเรียก API โดยตรงผ่าน INT แล้ว ยังสามารถโปรแกรมโดยใช้แม่แบบ หรือ Macro ที่จัดทำโดยผู้ผลิต

Macro เป็นเครื่องมือช่วยในการเขียน code อัตโนมัติ ซึ่งโปรแกรม assembler สามารถเข้าใจและนำมาใช้งานได้

```
Macro_name MACRO Parameters
.....
....
Macro_name ENDM
```

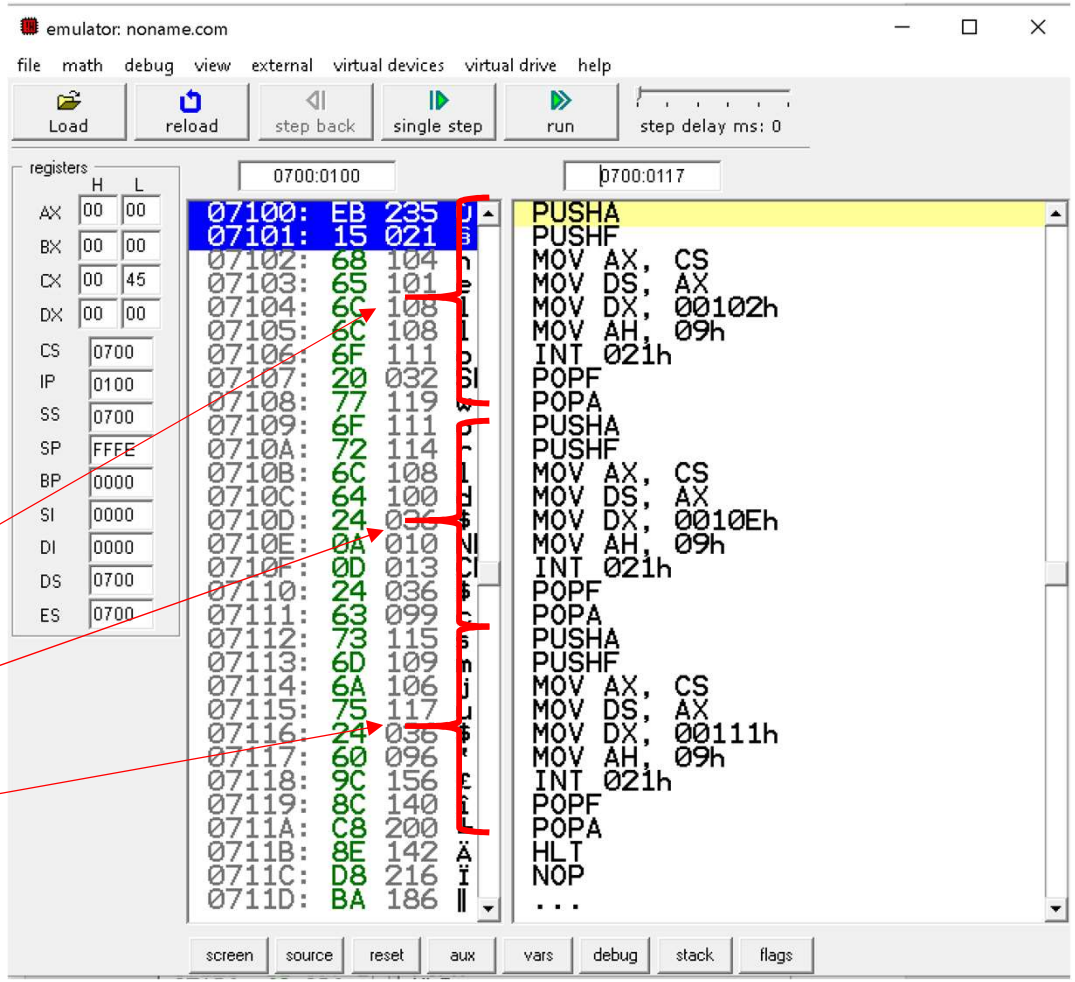
ก่อนการใช้งานต้องทำการประกาศ macro ไว้ก่อนการเรียกใช้เสมอ หรือจะใช้วิธี include จากแฟ้มภายนอกก็ได้

ตัวอย่างการสร้าง Macro สำหรับแสดงสายอักขระ

```
ORG 100h
.data
msg1 db "hello world$"
crlf db 10,13,'$'
msg2 db "csmju$"
.code
printf MACRO string
    pusha
    pushf
    mov ax,cs
    mov ds,ax
    lea dx,string
    mov ah,9
    int 21h
    popf
    popa
printf ENDM
start:
   printf msg1
   printf crlf
   printf msg2
hlt
```

นิยาม macro

เรียกใช้ macro



ตัวอย่าง Macro ช่วย code ที่ให้มากับ EMU8086

Inc/emu8086.inc

https://github.com/AhmadNaserTurnkeySolutions/emu8086/blob/master/inc/emu8086.inc

```
; this macro prints a char in AL and advances
; the current cursor position:
PUTC    MACRO   char
        PUSH    AX
        MOV     AL, char
        MOV     AH, 0Eh
        INT     10h
        POP     AX
ENDM


; this macro prints a string that is given as a parameter, example:
; PRINT 'hello world!'
; new line is NOT added.
PRINT   MACRO   sdat
LOCAL   next_char, s_dcl, printed, skip_dcl

PUSH    AX      ; store registers...
PUSH    SI      ;

JMP     skip_dcl        ; skip declaration.
        s_dcl DB sdat, 0

skip_dcl:
        LEA     SI, s_dcl

next_char:
        MOV     AL, CS:[SI]
        CMP     AL, 0
        JZ      printed
        INC     SI
        MOV     AH, 0Eh ; teletype function.
        INT     10h
        JMP     next_char
printed:

POP     SI      ; re-store registers...
POP     AX      ;
ENDM
```

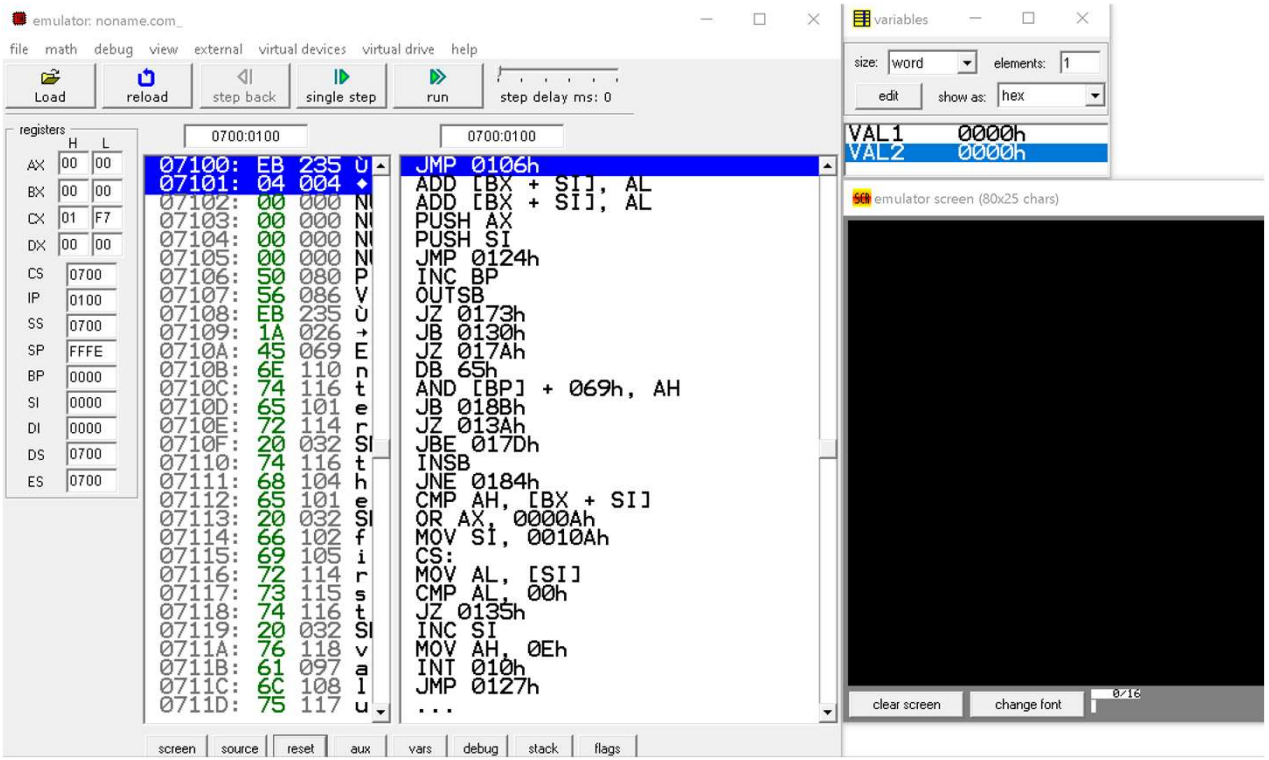
- **PUTC char** - macro with 1 parameter, prints out an ASCII char at current cursor position.
- **GOTOXY col, row** - macro with 2 parameters, sets cursor position.
- **PRINT string** - macro with 1 parameter, prints out a string.
- **PRINTN string** - macro with 1 parameter, prints out a string. The same as PRINT but automatically adds "carriage return" at the end of the string.
- **CURSOROFF** - turns off the text cursor.
- **CURSORON** - turns on the text cursor.

- **PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.

- **PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction. For example:
CALL PTHIS
db 'Hello World!', 0
To use it declare: **DEFINE_PTHIS** before **END** directive.
- **GET_STRING** - procedure to get a null terminated string from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. Procedure stops the input when 'Enter' is pressed. To use it declare: **DEFINE_GET_STRING** before **END** directive.
- **CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare: **DEFINE_CLEAR_SCREEN** before **END** directive.
- **SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare: **DEFINE_SCAN_NUM** before **END** directive.
- **PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END** directive.
- **PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.

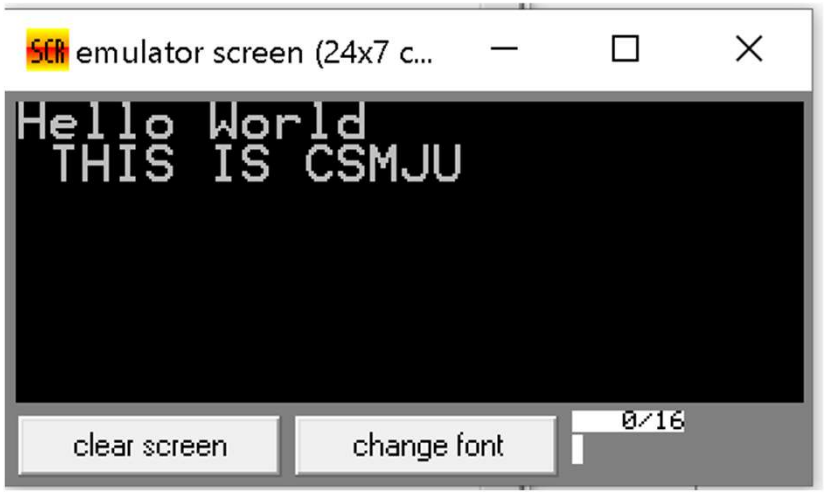
ตัวอย่างการใช้ Macro รับจำนวนเต็ม 2 ค่ามาบวกกันแล้วแสดงผลบนจอภาพ

```
include emu8086.inc
org 100h
.DATA
 val1 dw ?
 val2 dw ?

.CODE
start:
  PRINTN "Enter the first value: "
  call scan_num
  mov val1,cx
  PRINTN ""
  PRINTN "Enter the second value:  "
  call scan_num
  mov val2,cx
  PRINTN ""
  mov ax,val1
  add ax,val2
  call print_num

  HLT

  DEFINE_SCAN_NUM
  DEFINE_PRINT_NUM
  DEFINE_PRINT_NUM_UNS
end
```



ตัวอย่างการใช้ Macro

```
include 'emu8086.inc'
org 100h
.DATA

.CODE
start:
 PRINTN "Hello World"
 PRINTN " THIS IS CSMJU"
 HLT
```



เดิมภาษา C ถูกแปลเป็นภาษาเครื่อง โดยการแปลเป็นภาษา Assembly ก่อนโดยเรียกใช้ macro ที่ถูก include เข้ามาใน source code เช่น stdio , stdlib จากนั้นจึงถูก แปลเป็น object file ด้วย Assembler และทำการ Link ให้เป็น executable โดยโปรแกรม linker ภาษา C จึงใกล้เคียงกับภาษา Assembly เป็นอย่างมาก และสามารถเขียน ภาษา Assembly ปนกับ source code ภาษา C ได้เลย

C++ Copy

```
// asm_overview.cpp
// processor: x86
void __declspec(naked) main()
{
    // Naked functions must provide their own prolog...
    __asm {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE
    }

    // ... and epilog
    __asm {
        pop ebp
        ret
    }
}
```

Assembler ยังรองรับการ Call ที่มีการส่งผ่านพารามิเตอร์ผ่าน stack ในรูปแบบมาตรฐาน โดยมีการจัดการและสร้าง Prolog และ Epilog โดยอัตโนมัติ เรียกว่า INVOKE

การเขียน Application สำหรับระบบปฏิบัติการ Windows จะใช้วิธีการ INVOKE มาแทนการใช้ Macro

เพราะเป็นการ call โปรแกรมที่ถูกแปลเป็นภาษาเครื่องแล้ว ไม่จำเป็นต้องแปลใหม่เหมือนการใช้ Macro

ตัวโปรแกรมที่ call จะอยู่ใน library ที่ถูกรวมเข้าไว้ใน exe ในขั้นตอนของการ link หรือเรียกจากหน่วยความจำ หรือ library ที่ใช้ร่วมกันได้ เช่น DLL เป็นต้น

Assembler ที่ติดมากับ EMU8086 รองรับความสามารถเพียงแค่ Macro เท่านั้น

การเขียนโปรแกรมที่ซับซ้อนทีต้องเรียกใช้การ Invoke ต้องเปลี่ยนไปใช้ Assembler ที่มีความสามารถมากขึ้นตัวอย่างเช่น

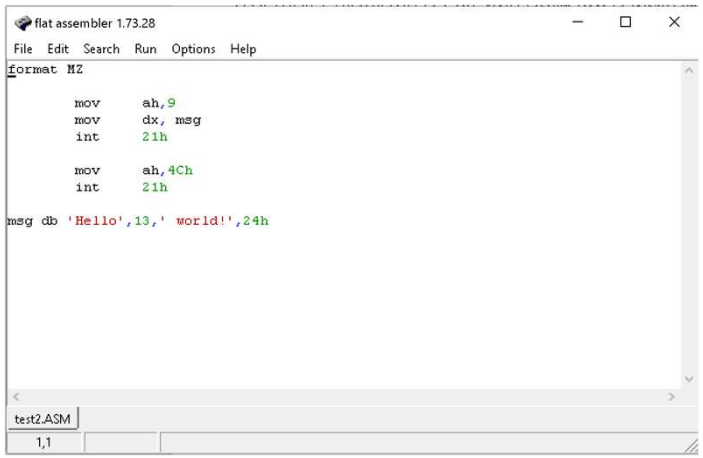
MASM

TASM

Flat asm

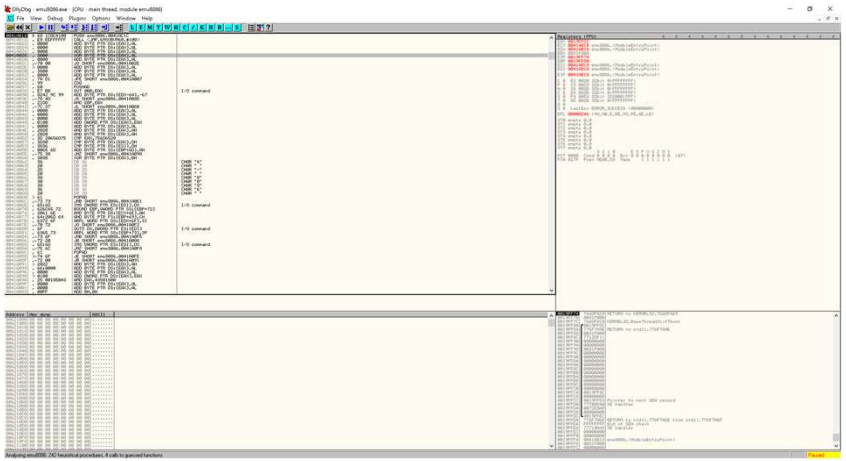
ตัวอย่างโปรแกรมสำหรับพัฒนา Application สำหรับ Microsoft Windows

Assembler

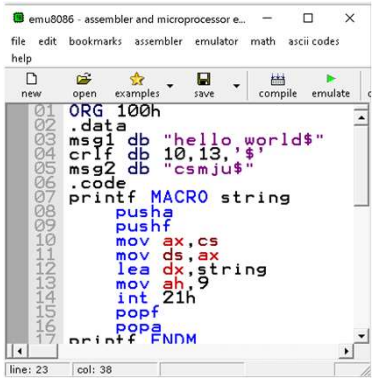


https://flatassembler.net

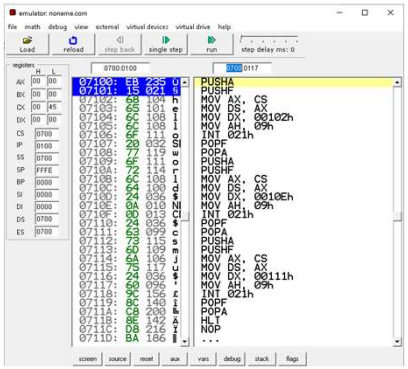
Debugger



http://www.ollydbg.de

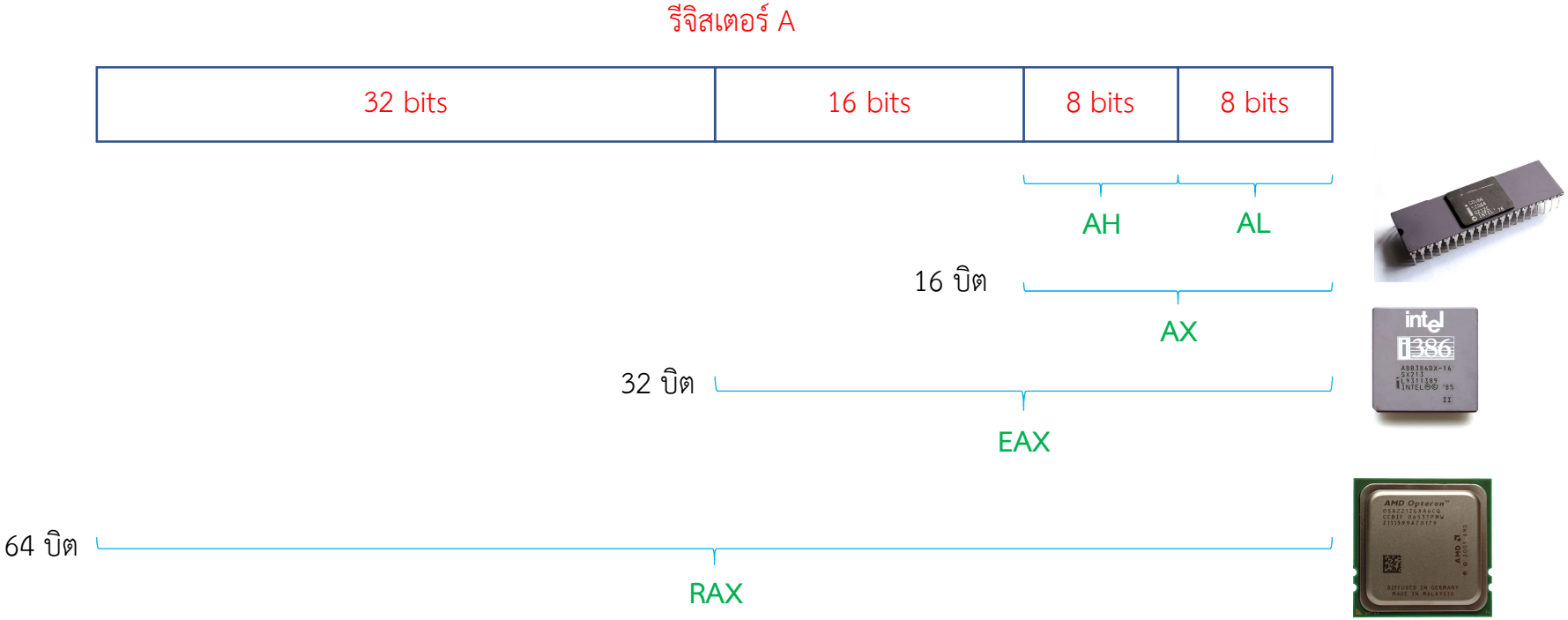


ทำงานเหมือน Emu8086 assembler



ทำงานเหมือน Emu8086 emulator

ไมโครโพรเซสเซอร์ตระกูล X86 ได้ถูกพัฒนาต่อเนื่องมาอย่างยาวนาน แต่ยังคงความเข้ากันได้กับ 8086 โปรแกรมที่เขียนสำหรับ 8086 ยังสามารถทำงานได้กับไมโครโพรเซสเซอร์รุ่นปัจจุบัน



รีจิสเตอร์ทั้งหมดของไมโครโพรเซสเซอร์ x64

	ทั้ง 64 บิต	32 บิตล่าง	16 บิตล่าง	8 บิตล่าง
X64 (CPU)	RAX	EAX	AX	AL
	RBX	EBX	BX	BL
	RCX	ECX	CX	CL
	RDX	EDX	DX	DL
	RSI	ESI	SI	SIL
	RDI	EDI	DI	DIL
	RBP	EBP	BP	BPL
	RSP	ESP	SP	SPL
X87 (FPU)	R8	R8D	R8W	R8B
	R9	R9D	R9W	R9B
	R10	R10D	R10W	R10B
	R11	R11D	R11W	R11B
	R12	R12D	R12W	R12B
	R13	R13D	R13W	R13B
	R14	R14D	R14W	R14B
	R15	R15D	R15W	R15B

คพ 231 ระบบคอมพิวเตอร์และภาษาแอสแซมบลี

โครงสร้างของเครื่องคอมพิวเตอร์และภาษาเครื่อง ภาษาแอสเซมบลีและการแปลภาษาโดยโปรแกรมแอสเซมเบลอร์ เทคนิคการจัดแอดเดรส โครงสร้างของแอสเซมเบลอร์ การเขียนโปรแกรมภาษาแอสเซมบลี การเขียนโปรแกรมแบบแมคโคร โปรแกรมแบบแยกส่วน

Computer organization and machine language; Assembly language and assembler; Addressing technique; Structure of assembler; Programming in assembly language; Macro programming; Module programming

คพ 231 ระบบคอมพิวเตอร์และภาษาแอสแซมบลี

ทำไมต้องเรียน ภาษาแอสแซมบลี ?

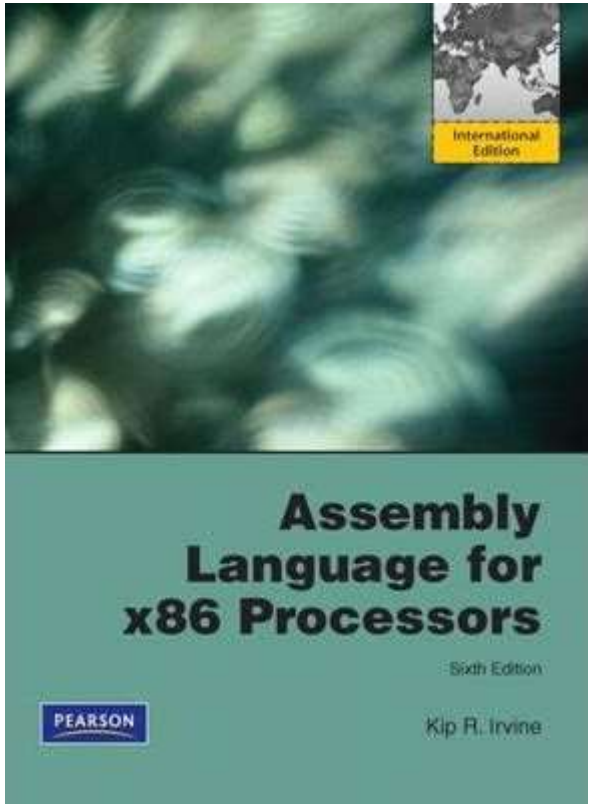
- ศึกษาการทำงานของ Hardware และระบบปฏิบัติการ
- พัฒนาซอฟท์แวร์ที่เน้นประสิทธิภาพด้านความเร็วและใช้ทรัพยากรน้อย
- ค้นหาและแก้ไขจุดบกพร่องของโปรแกรม (Debug)
- ติดต่อ สั่งงาน Hardware โดยตรง
- ต่อยอดทางด้านการรักษาความปลอดภัยในระบบคอมพิวเตอร์

คพ 231 ระบบคอมพิวเตอร์และภาษาแอสแซมบลี

ทำไมต้องเรียน ระบบคอมพิวเตอร์ ?

- เข้าใจส่วนประกอบต่าง ๆ และชิ้นส่วน ของระบบคอมพิวเตอร์
- เข้าใจสถาปัตยกรรมทางด้านคำสั่ง (ISP)
- เข้าใจวิธีการสื่อสารภายในเครื่องคอมพิวเตอร์ (Bus , Addressing)
- เข้าใจข้อจำกัดของหน่วยประมวลผล

หนังสือแนะนำ



Assembly Language for x86 Processors : International Edition

By (author) Kip R. Irvine

For the Intel/Windows/DOS platform

ชิ้นงานที่ 2 วิชา คพ 231 ระบบคอมพิวเตอร์และภาษาแอสแซมบลี

ส่งงานได้จนถึงเที่ยงคืนของวันอาทิตย์ที่ 7 พฤศจิกายน 2564 ทาง Microsoft Team

สอบปลายภาค: **1 พ.ย. 2564 เวลา 15:30 - 18:30 อาคาร 105 ห้อง TBA**

จะ update รายละเอียดทาง MS Team และ Facebook page

แล้วเจอกันใหม่ 1/65