Disclaimer & Fair Use Statement

- This compiled lecture notes contains copyrighted material, the use of which has not always been specifically authorized by the copyright owner(s). We are making such material available in our efforts to advance understanding in the study of microprocessors and microcontrollers for our undergraduate engineering course. We believe this constitutes fair use of any such copyrighted material as provided by **Sec. 185 of RA 8293 of the Intellectual Property Code of the Philippines** and **Sec. 107 of the US Copyright Law**. In accordance with both laws, the material on this compiled lecture notes are distributed without profit to those who have expressed a prior interest in receiving the included information for education and research purposes.
- For more information go to: **https://www.chanrobles.com/legal7copyright.htm#.X0EWZ9wzaUk and http://www.law.cornell.edu/uscode/17/107.shtml.**

- If you wish to use copyrighted material from this compiled lecture notes for purposes of your own that go beyond fair use, you must obtain permission from the copyright owner(s).

Unit 05 – Assembly Language Programming – Part 1

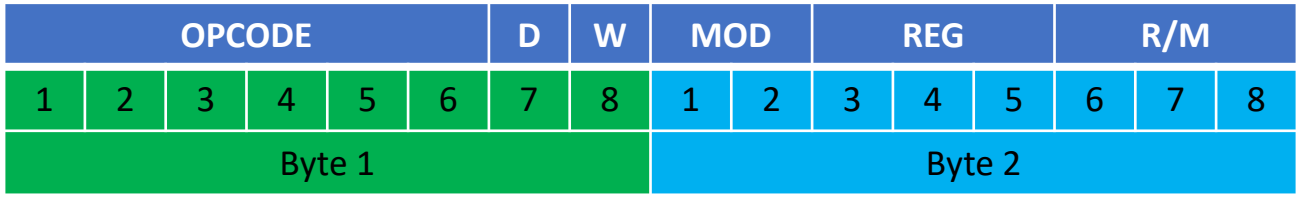
COMPUTER ENGINEERING
UNIVERSITY of SAN CARLOS

Assembly Language Instructions to Machine code

Assembly language

Assembly Code to Machine Code conversion

- An assembly instruction can be coded with 1 to 6 bytes of equivalent machine code



Byte 1: contains three fields:

- **OPCODE** field (6 bits)
  - Specifies the operation like add, subtract, or move
- Register Direction bit (**D** bit)
  - Tells the register operand in REG field is either a source or destination operand
  - If 1, data flow to the REG field from R/M
  - If 0, data flow from the REG field to R/M
- Data/Word size bit (**W** bit)
  - Specifies if operation performs 8-bit or 16-bit data
  - If 0, 8 bits
  - If 1, 16 bits

Byte 2: contains two fields:

- Mode field (**MOD**), 2 bits
- Register field (**REG**), 3 bits
- Register/Memory field (**R/M**) – 3 bits

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

MOD	EXPLANATION
00	Memory mode, no displacement
01	Memory mode, 8-bit displacement follows
10	Memory mode, 16-bit displacement follows
11	Register mode, no displacement

Assembly language

Example:

MOV BL, AL

- **OPCODE** for MOV = 100010
- Need to encode AL as source operand
  - **D** = 0
- **W** bit = 0 (8-bit)
- **MOD** = 11 (register mode)
- **REG** = 000 (code for AL)
- **R/M** = 011

OPCODE						D	W	MOD		REG			R/M		
1	0	0	0	1	0	0	0	1	1	0	0	0	0	1	1
1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Byte 1								Byte 2							
88H								C3H							

Instruction Formats

Assembly language

General Assembler Instruction Format

[Labels:] [Mnemonic] [;Comment]

- **Labels**: provides a symbolic address used in branch instructions
- **Opcode**: specifies the type of instructions
- **Operands**: 8086 family instructions having one, two, or zero operand
- **Comment**: used by programmers for effective reference

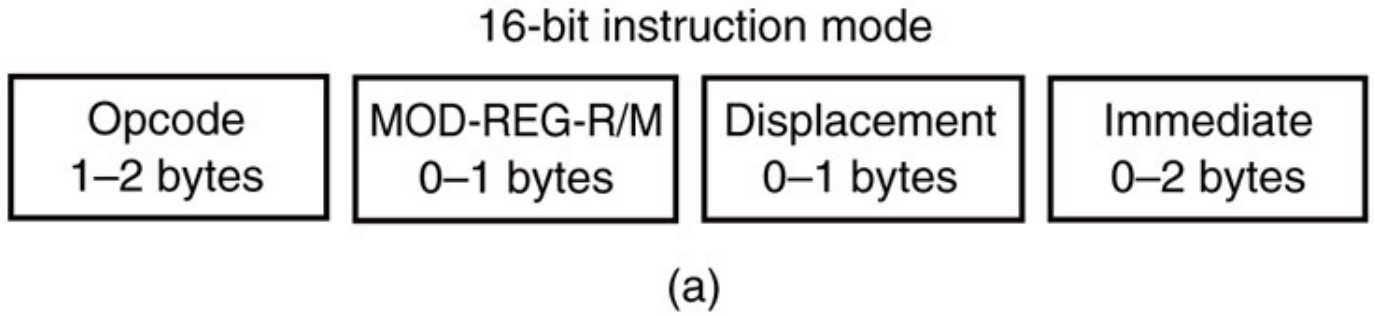
INSTRUCTION FORMAT		
MNEMONIC		
OPCODE	OPERANDS	
MOV	BL,	AL
	DESTINATION OPERAND	SOURCE OPERAND

Example:

START: MOV BL, AL ; copy AL to BL

Instruction formats 8086 to Core2

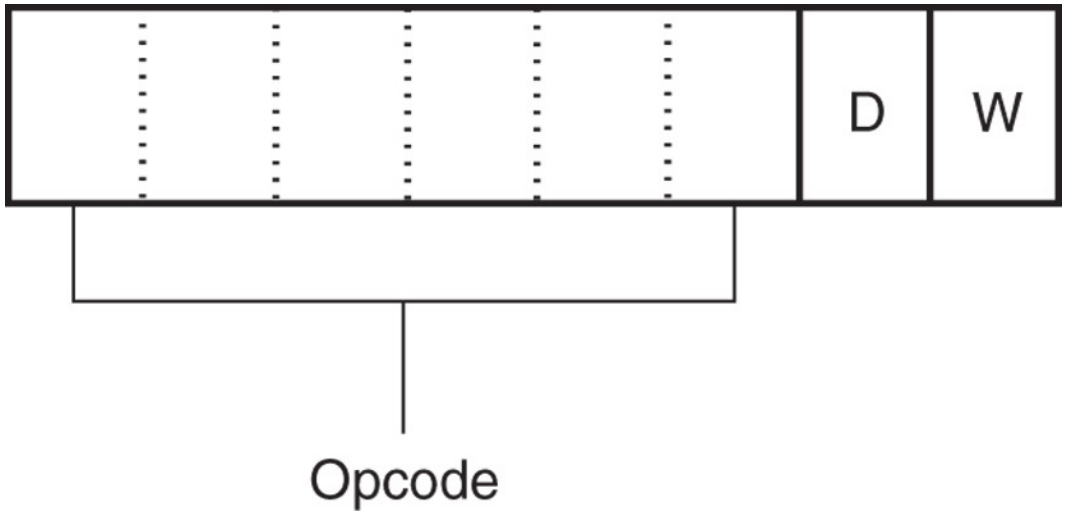
- 80386 and above assume all instructions are 16-bit mode instructions when the machine is operated in the **real mode** (**DOS**).
- in **protected mode** (Windows), the upper byte of the descriptor contains the D-bit that selects either the 16- or 32-bit instruction mode



Instruction Formats

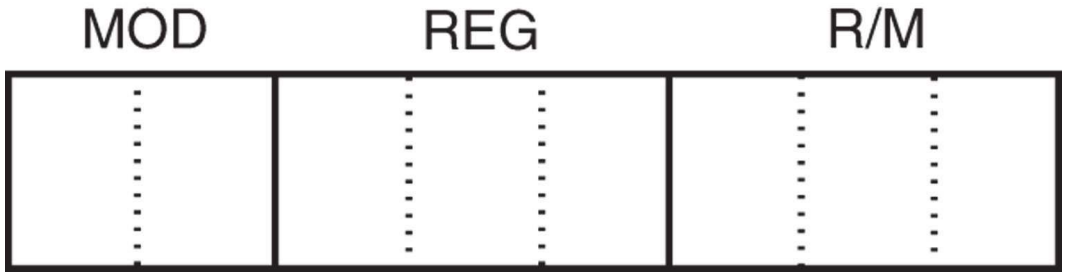
Opcode field

- Selects the operation (addition, subtraction, etc.,) performed by the microprocessor.
  - either 1 or 2 bytes long for most instructions
- General form of the first opcode byte of many instructions.
  - first 6 bits of the first byte are the binary opcode
  - remaining 2 bits indicate the **direction** (**D**) of the data flow, and indicate whether the data are a byte or a **word** (**W**)



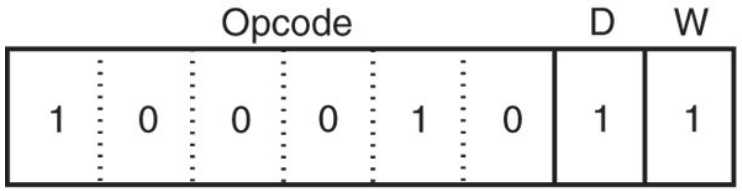
MOD field

- Specifies **addressing mode** (**MOD**) and whether a displacement is present with the selected type.
- If MOD field contains an 11, it selects the register-addressing mode
- Register addressing specifies a register instead of a memory location, using the **R/M** field
- If the MOD field contains a 00, 01, or 10, the R/M field selects one of the data memory-addressing modes.

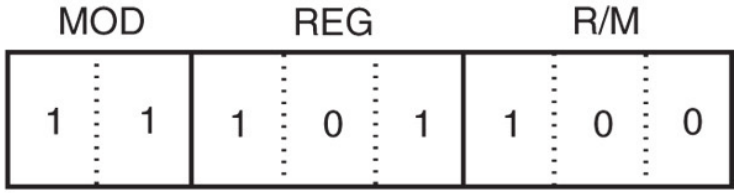


Example: MOV BP, SP

- **D** and **W** bits are a logic 1, so a word moves into the destination register specified in the **REG** field
- **REG** field contains 101, indicating register **BP**, so the **MOV** instruction moves data into register **BP**



Opcode = MOV
D = Transfer to register (REG)
W = Word
MOD = R/M is a register
REG = BP
R/M = SP



– equivalent **machine code** is 1000 1011 1110 1100 or **8BECh**

Assembler/Emulator

Assembler Directives

Instructions to the assembler regarding the program being executed

Control the generation of machine codes and organization of the program;

But no machine codes are generated for assembler directives

Also called '**pseudo instructions**'

Used to :

- specify the start and end of a program
- attach value to variables
- allocate storage locations to input/ output data
- define start and end of segments, procedures, macros etc..

```
; Program for addition of two 8-bit nos.   Comments / Remark

   ASSUME    CS: Code   DS: Data
Data        SEGMENT
   N1       DB  2FH
   N2       DB  0EH
   SUM      DB  1 DUP(?)
Data              ENDS

Code        SEGMENT
            MOV AL,N1
            MOV BL,N2
            ADD  AL,BL
            MOV SUM,AL

Code        ENDS
            END
```

Assembler directives

Program

Assembler directives

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Define Byte
- Define a byte type (8-bit) variable
- Reserves specific amount of memory locations to each variable
- Range :
  - 00_H – FF_H for unsigned value;
  - 00_H – $7F_H$ for positive value and
  - 80_H – FF_H for negative value
- General form : variable **DB** value/ values

Example:

```
LIST DB 7FH, 42H, 35H
```

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Define Word
- Define a word type (16-bit) variable
- Reserves two consecutive memory locations to each variable
- Range : 0000_H – $FFFF_H$ for unsigned value;
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- General form : variable **DW** value/ values

Example:

ALIST **DW** 6512H, 0F251H, 0CDE2H

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

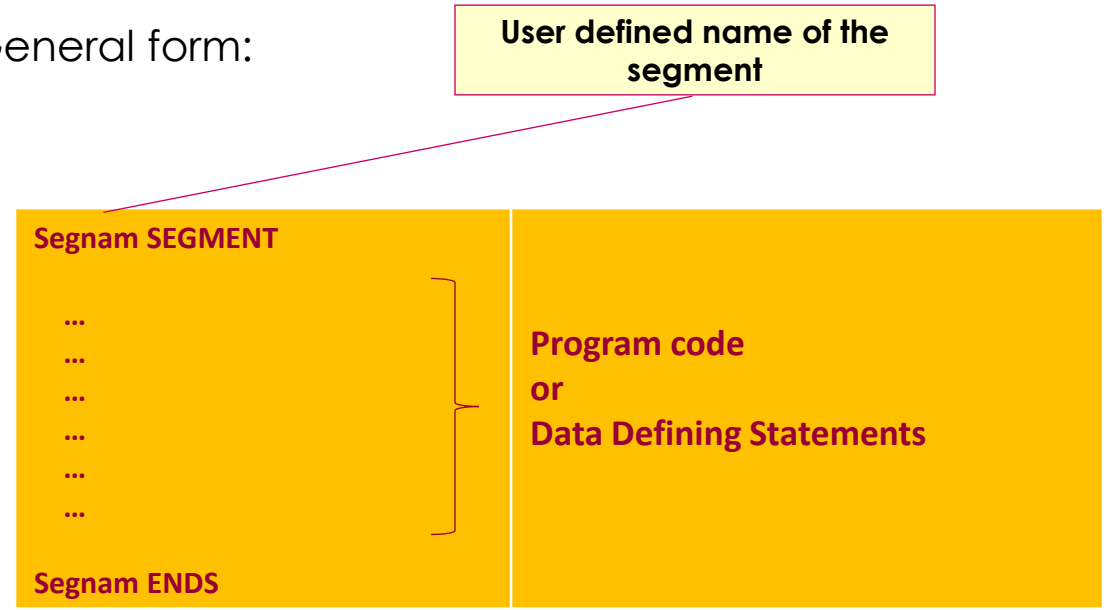
ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT** : Used to indicate the beginning of a code/ data/ stack segment
- **ENDS** : Used to indicate the end of a code/ data/ stack segment
- General form:



Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Informs the assembler the name of the program/ data segment that should be used for a specific segment.
- General form:

ASSUME segreg : segnam, .. , segreg : segnam

Segment Register	User defined name of the segment

Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program/ data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- General form

```
procname PROC[NEAR/ FAR]
        ...
        ...
        ...
        RET
procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR … … … RET ADD64 ENDP	The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return
CONVERT PROC FAR … … … RET CONVERT ENDP	The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- Reserves one memory location for 8-bit signed displacement in jump instructions

Example:

JMP **SHORT** AHEAD	The directive will reserve one memory location for 8-bit displacement named AHEAD

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

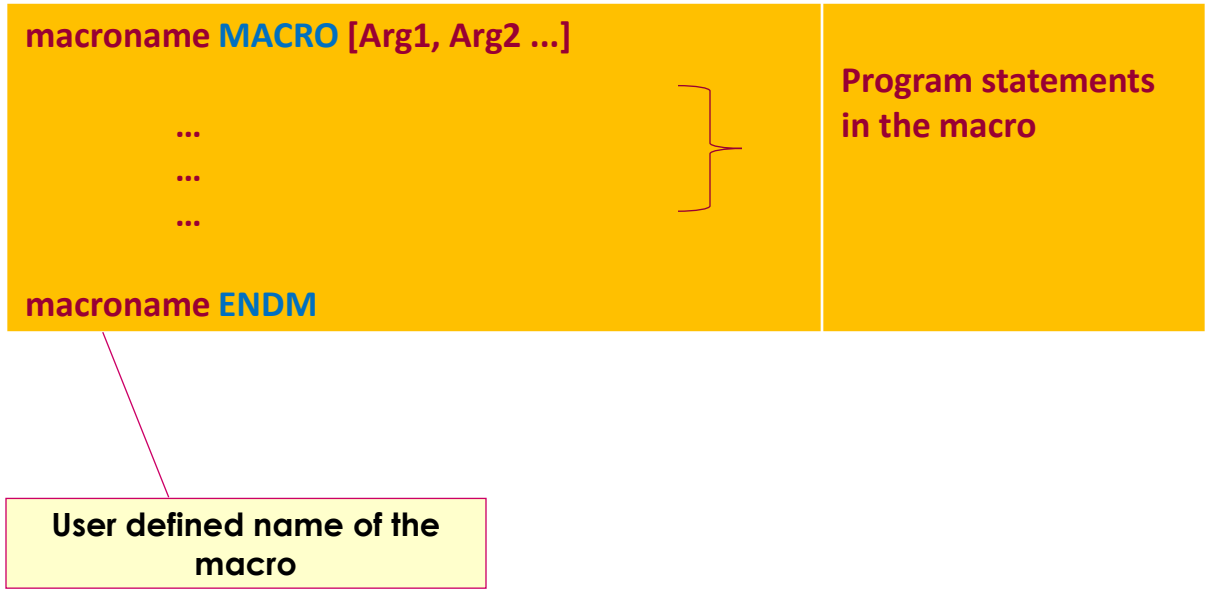
ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro
- General form:



Assemble, Link and Execute Program

How to Build Executable Programs

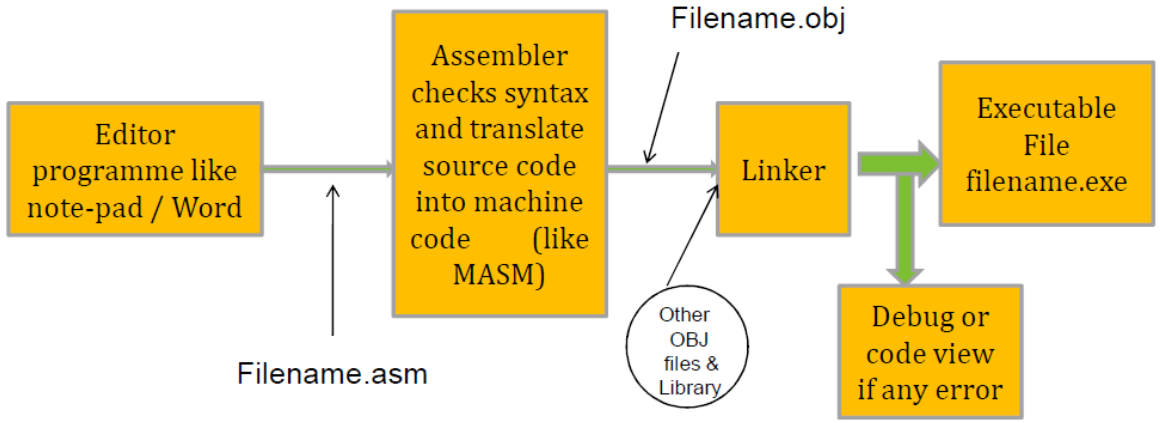
EDITOR

- A text editor is required to create assembly language source files
- The source files contains your source code
- You may use notepad or any other editor that produces ASCII text files

DEBUGGER

- A debugger program allows tracing of program execution
- It also allows examination of registers and memory content
- For 16-bit programs, MASM's debugger named CodeView can be used to debug code

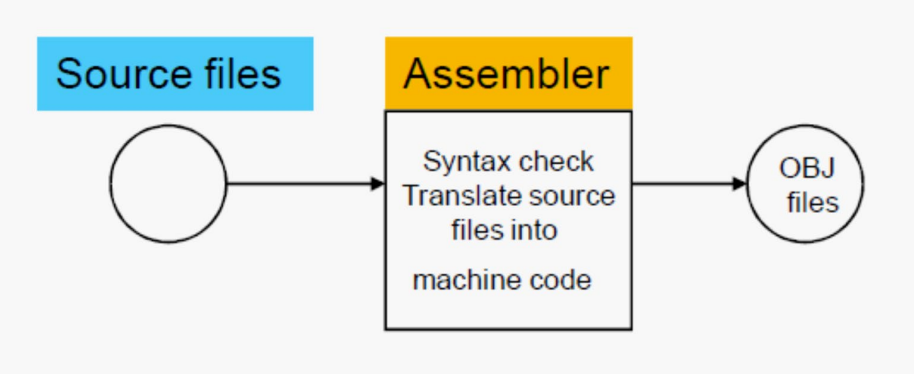
Step & operation	Input	Software	Output
1.Editing / writing programme	Note pad or Word	Note pad / MS-Word	Filename.asm
2.Assemble	Filename.asm	MASM	Filename.obj
3.Link	Filename.obj	LINK	Filename.exe



How to Build Executable Programs

ASSEMBLER

- A program that converts source code programs written in assembly language into object files in machine language
- Popular assemblers include **MASM** (Macro Assembler from Microsoft), **TASM** (Turbo Assembler from Borland), **NASM** (Netwide Assembler fro both Windows and Linux), and **GNU assembler** distributed by the free software foundation.



LINKER

- A linker program combines you program's object file created by the assembler with other object files and link libraries, producing a single executable program
- A linker utility is needed to produce executable files
- Two linkers: **LINK.EXE** and **LINK32.EXE** are provided with the MASM 6.15 distribution to link 16-bit real-address mode and 32-bit protected-address mode program respectively.