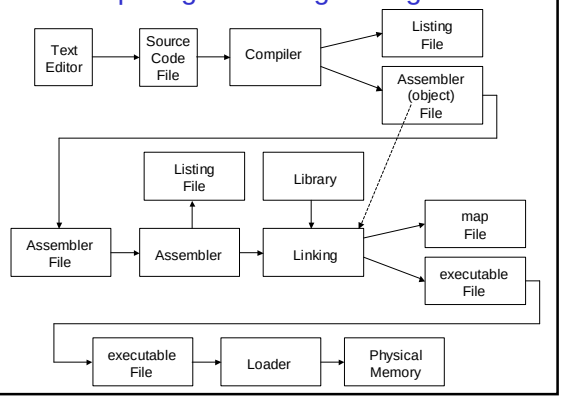
Preparing/Executing a Program



MASM 8086 Instruction - Basic Structure

Label *Operator* *Operand[s]* *;Comment*

Label - optional alphanumeric string
1st character must be **a-z**,**A-Z**,**?**,@,_,**$**
Last character must be **:**
Operator - assembler language instruction
mnemonic an instruction format for humans
assembler translates mnemonic into hexadecimal *opcode*
example `mov` is f8h
Operand[s] - 0 to 3 pieces of data required by instruction
can be several different forms
delineated by commas
immediate, register name, memory data, memory address
Comment - Extremely useful in assembler language
general rule is 1 comment per instruction
These fields are separated by White Space (tab, blank, \n, etc.)

8086 Instruction - Example

Label *Operator* *Operand[s]* *;Comment*

```
INIT: mov   ax, bx      ; Copy contents of bx into ax
```

Label - **INIT:**
Operator - **mov**
Operands - **ax** and **bx**
Comment - alphanumeric string between **;** and **\n**

- Not case sensitive
- Unlike other assemblers, destination operand is first
- **mov** is the *mnemonic* that the
  - assembler translates into an *opcode*
- unassemble translates *opcode* into *mnemonic*

Anatomy of MASM Source File

- Assembler Program Divided Into Segments
  - Not Exactly the same as Memory Segments
  - Program can have Several Segments; Only 1 Active
- Segments Defined in 1 or More Modules
  - contain instructions, data, directives
- Each Module is a Separate File
  - Assembler Translates Modules to Object Files
- Linker Does Several Things
  - Combines Multiple Object Files
  - Resolves Relative Addresses
  - Inserts Loader Code
  - Creates Executable

Assembler Language Segment Types

- Stack
  - For Dynamic Data Storage
  - Source File Defines Size
  - Must Have Exactly 1 - *Context Switching*
- Data
  - For Static Data Storage
  - Source File Defines Content (Optional)
  - Can Have 0 or More
- Code
  - For Machine Instructions
  - Must Have 1 or More

Using MASM Assembler

- to get help:

```
C:\> masm /h
```

- Can just invoke MASM with no arguments:

```
C:\> masm
Source Filename           [.ASM]:    hello
Object Filename      [HELLO.OBJ]:
Source Listing         [NUL.LST]:
Cross Reference        [NUL.CRF]:
```

- `.ASM` - Assembler Source File Prepared by Programmer
- `.OBJ` - Translated Source File by Assembler
- `.LST` - Documents “Translation” Process
  - Errors, Addresses, etc.
- `.CRF` - Symbol Table

Using MASM Assembler/Linker (Cont.)

- Another way to invoke assembler:
  ```
  C:\> masm hello,,hello,hello
  ```
- This Causes MASM to Create:
  ```
  HELLO.OBJ    HELLO.LST    HELLO.CRF
  ```
- Yet Another Way:
  ```
  C:\> masm hello
  ```
- This Causes MASM to Create:
  ```
  HELLO.OBJ
  ```
- Next Step is to Create Executable File using the Linker:
  ```
  C:\> link hello
  ```
- This Causes Linker to Create:
  ```
  HELLO.EXE
  ```

MASM Assembler Language

- Each Module Contains 4 Types of Statements:
  1. Executable Instructions
  2. MASM Assembler Directives
  3. MASM Macroinstruction Definitions
  4. MASM Macroinstruction Calls

Executable Instr. Instructions that the x86 can fetch from memory and execute

MASM Dir. Programmer supplied directives that guide the "translation" process

MASM Macro Defs. and Calls Similar to "Functions" explored in a laboratory assignment

x86 Instruction Type Classifications

- DATA TRANSFER
  - General `mov  ax, DAT1 ;ax gets contents of mem`
  - Strings `cmpsb  ;if DS:SI=ES:DI then ZF=1`
  - Special Purpose `xchg  ax, bx  ;ax gets bx and bx gets ax`
- ARITHMETIC/LOGIC
  - Integer `add  ax, bx  ;ax gets ax+bx`
  - ASCII, BCD `aaa  ;changes ASCII # to int.`
  - Floating Point `fadd  DAT  ;ST get ST+DAT`
  - Logical `and  ax, bx  ;ax gets ax AND bx`
  - Shifting `ror  ax, 2  ;ax contents shifted-2 right`
- CONTROL TRANSFER
  - Branching `jnz  LABEL1  ;if ZF=1 then IP=LABEL1`
  - Interrupt `int  21h  ;invoke INT handler 21h`
  - Subroutine `call  SUB1  ;invoke subroutine, SUB1`
  - Modify Flag `cli  ;IF gets zero`
  - Halt Processor `hlt  ;need RESET to run again`
  - No Operation `nop  ;takes up space/time`

x86 Instruction Set Summary
(Not Included in Following Slides)

- Floating Point - 8087
- Special Protected Mode 80386
- MMX (DSP) Pentium MMX
- SSE - Special SIMD Pentium III
- Others (few specialized added with each gen.)

x86 Instruction Set Summary
(Data Transfer)

```
CBW             ;Convert Byte to Word in AX
CWD             ;Convert Word to Double in AX,DX
IN              ;Input
LAHF            ;Load AH with 8080 Flags
LDS             ;Load pointer to DS
LEA             ;Load EA to register
LES             ;Load pointer to ES
LODS            ;Load memory at SI into AX
MOV             ;Move
MOVS            ;Move memory at SI to DI
OUT             ;Output
POP             ;Pop
POPF            ;Pop Flags
PUSH            ;Push
PUSHF           ;Push Flags
SAHF            ;Store AH into 8080 Flags
SCAS            ;Scan memory at DI compared to AX
SEG             ;Segment register
STOS            ;Store AX into memory at DI
XCHG            ;Exchange
XLAT            ;Translate byte to AL
```

x86 Instruction Set Summary
(Arithmetic/Logical)

```
AAA             ;ASCII Adjust for Add in AX
AAD             ;ASCII Adjust for Divide in AX
AAM             ;ASCII Adjust for Multiply in AX
AAS             ;ASCII Adjust for Subtract in AX
ADC             ;Add with Carry
ADD             ;Add
AND             ;Logical AND
CMC             ;Complement Carry
CMP             ;Compare
CMPS            ;Compare memory at SI and DI
CWD             ;Convert Word to Double in AX,DX
DAA             ;Decimal Adjust for Add in AX
DAS             ;Decimal Adjust for Subtract in AX
DEC             ;Decrement
DIV             ;Divide (unsigned) in AX(,DX)
IDIV            ;Divide (signed) in AX(,DX)
IMUL            ;Multiply (signed) in AX(,DX)
INC             ;Increment
```

x86 Instruction Set Summary
(Arithmetic/Logical Cont.)

```
MUL                 ;Multiply (unsigned) in AX(,DX)
NEG                 ;Negate
NOT                 ;Logical NOT
OR                  ;Logical inclusive OR
RCL                 ;Rotate through Carry Left
RCR                 ;Rotate through Carry Right
ROL                 ;Rotate Left
ROR                 ;Rotate Right
SAR                 ;Shift Arithmetic Right
SBB                 ;Subtract with Borrow
SCAS                ;Scan memory at DI compared to AX
SHL/SAL             ;Shift logical/Arithmetic Left
SHR                 ;Shift logical Right
SUB                 ;Subtract
TEST                ;AND function to flags
XLAT                ;Translate byte to AL
XOR                 ;Logical Exclusive OR
```

x86 Instruction Set Summary
(Control/Branch Cont.)

```
CALL                ;Call
CLC                 ;Clear Carry
CLD                 ;Clear Direction
CLI                 ;Clear Interrupt
ESC                 ;Escape (to external device)
HLT                 ;Halt
INT                 ;Interrupt
INTO                ;Interrupt on Overflow
IRET                ;Interrupt Return
JB/JNAE             ;Jump on Below/Not Above or Equal
JBE/JNA             ;Jump on Below or Equal/Not Above
JCXZ                ;Jump on CX Zero
JE/JZ               ;Jump on Equal/Zero
JL/JNGE             ;Jump on Less/Not Greater or Equal
JLE/JNG             ;Jump on Less or Equal/Not Greater
JMP                 ;Unconditional Jump
JNB/JAE             ;Jump on Not Below/Above or Equal
JNBE/JA             ;Jump on Not Below or Equal/Above
JNE/JNZ             ;Jump on Not Equal/Not Zero
JNL/JGE             ;Jump on Not Less/Greater or Equal
```

x86 Instruction Set Summary
(Control/Branch)

```
JNLE/JG             ;Jump on Not Less or Equal/Greater
JNO                 ;Jump on Not Overflow
JNP/JPO             ;Jump on Not Parity/Parity Odd
JNS                 ;Jump on Not Sign
JO                  ;Jump on Overflow
JP/JPE              ;Jump on Parity/Parity Even
JS                  ;Jump on Sign
LOCK                ;Bus Lock prefix
LOOP                ;Loop CX times
LOOPNZ/LOOPNE       ;Loop while Not Zero/Not Equal
LOOPZ/LOOPE         ;Loop while Zero/Equal
NOP                 ;No Operation (= XCHG AX,AX)
REP/REPNE/REPNZ     ;Repeat/Repeat Not Equal/Not Zero
REPE/REPZ           ;Repeat Equal/Zero
RET                 ;Return from call
SEG                 ;Segment register
STC                 ;Set Carry
STD                 ;Set Direction
STI                 ;Set Interrupt
TEST                ;AND function to flags
WAIT                ;Wait
```

MASM Program Example

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                   ;
;   This is an example program for EE/CS 3724/3124.  It prints the  ;
;   character string "Hello World" to the DOS standard output       ;
;   using the DOS servie interrupt, function 9.                     ;
;                                                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
hellostk  SEGMENT BYTE STACK 'STACK'    ;Define the stack segment
          DB 100h DUP(?)    ;Set maximum stack size to 256 bytes (100h)
hellostk  ENDS

hellodat  SEGMENT BYTE 'DATA' ;Define the data segment
dos_print EQU 9                         ;define a constant via EQU
strng     DB  'Hello World',13,10,'$'  ;Define the character string
hellodat  ENDS

hellocod  SEGMENT BYTE 'CODE' ;Define the Code segment
START:    mov ax, SEG hellodat         ;ax <-- data segment start address
          mov ds, ax                   ;ds <-- initialize data segment register
          mov ah, dos_print            ;ah <-- 9 DOS 21h string function
          mov dx,OFFSET strng          ;dx <-- beginning of string
          int 21h                      ;DOS service interrupt
          mov ax, 4c00h                ;ax <-- 4c DOS 21h program halt function
          int 21h                      ;DOS service interrupt
hellocod  ENDS
          END       START              ; 'END label' defines program entry
```

Another Way to define Segments

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Use 'assume' directive to define segment types                  ;
;                                                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
hellostk  SEGMENT         ;Define a segment
          DB 100h DUP(?)
hellostk  ENDS

hellodat  SEGMENT           ;define a segment
dos_print EQU 9                       ;define a constant
strng     DB  'Hello World',13,10,'$'  ;Define the character string
hellodat  ENDS

hellocod  SEGMENT           ;define a segment
          assume cs:hellocod, ds:hellodat, ss: hellostk
START:    mov ax, hellodat            ;ax <-- data segment start address
          mov ds, ax                  ;ds <-- initialize data segment register
          mov ah, dos_print           ;ah <-- 9 DOS 21h string function
          mov dx,OFFSET strng         ;dx <-- beginning of string
          int 21h                     ;DOS service interrupt
          mov ax, 4c00h               ;ax <-- 4c DOS 21h program halt function
          int 21h                     ;DOS service interrupt
hellocod  ENDS
END       START
```

Yet another way to define Segs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Use .stack,.data,.code directivs to define segment types        ;
;   Also use .medium to define code model                           ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          .model medium
          .stack 100h        ; reserve 256 bytes of stack space

          .data
dos_print EQU 9                        ;define a constant
strng     DB  'Hello World',13,10,'$'  ;Define the character string


          .code

START:    mov ax, SEG strng           ;ax <-- data segment start address
          mov ds, ax                  ;ds <-- initialize data segment register
          mov ah, dos_print           ;ah <-- 9 DOS 21h string function
          mov dx,OFFSET strng         ;dx <-- beginning of string
          int 21h                     ;DOS service interrupt
          mov ax, 4c00h               ;ax <-- 4c DOS 21h program halt function
          int 21h                     ;DOS service interrupt

END       START
```

Masm Assembler Directives

end *label*	end of program, lable is entry point
proc far|near	begin a procedure; far, near keywords specify if procedure in different code segment (far), or same code segment (near)
endp	end of procedure
page	set a page format for the listing file
title	title of the listing file
.code	mark start of code segment
.data	mark start of data segment
.stack	set size of stack segment

Data Allocation Directives

db	define byte
dw	define word (2 bytes)
dd	define double word (4 bytes)
dq	define quadword (8 bytes)
dt	define tenbytes
equ	equate, assign numeric expression to a name
Examples:	
db 100 dup (?)	define 100 bytes, with no initial values for bytes
db "Hello"	define 5 bytes, ASCII equivalent of "Hello".
maxint equ	32767
count equ	10 * 20 ; calculate a value (200)

Memory Models

.model *mname* define a memory model for a program. The memory model will affect the size and number of the code, data segments. The memory model also affects the default procedure calls generated by the assembler (*near* calls for tiny,small, compact; *far* calls for medium, large, huge).

Model name can be:

tiny	code, data combined <= 64K
small	1 code, 1 data; code < = 64k, data <= 64k
medium	data <= 64k, code any size, multiple code segs, 1data
compact	code <= 64k, data any size, mulitple data segs, 1 code
large	code, data any size; multiple code, data segs
huge	same as large, but single array can be > 64k.
flat	no segments, all 32-bit addresses for code, data. Protected mode only

Target Processor Directives

.586 allow all nonpriviledged pentium instructions

.486 allow all nonpriviledged 486 instructions

other directives for processor types are similar:

.386

.286

.186

.8086

In the lab, simply use *.586* to access all of the Pentium instructions.