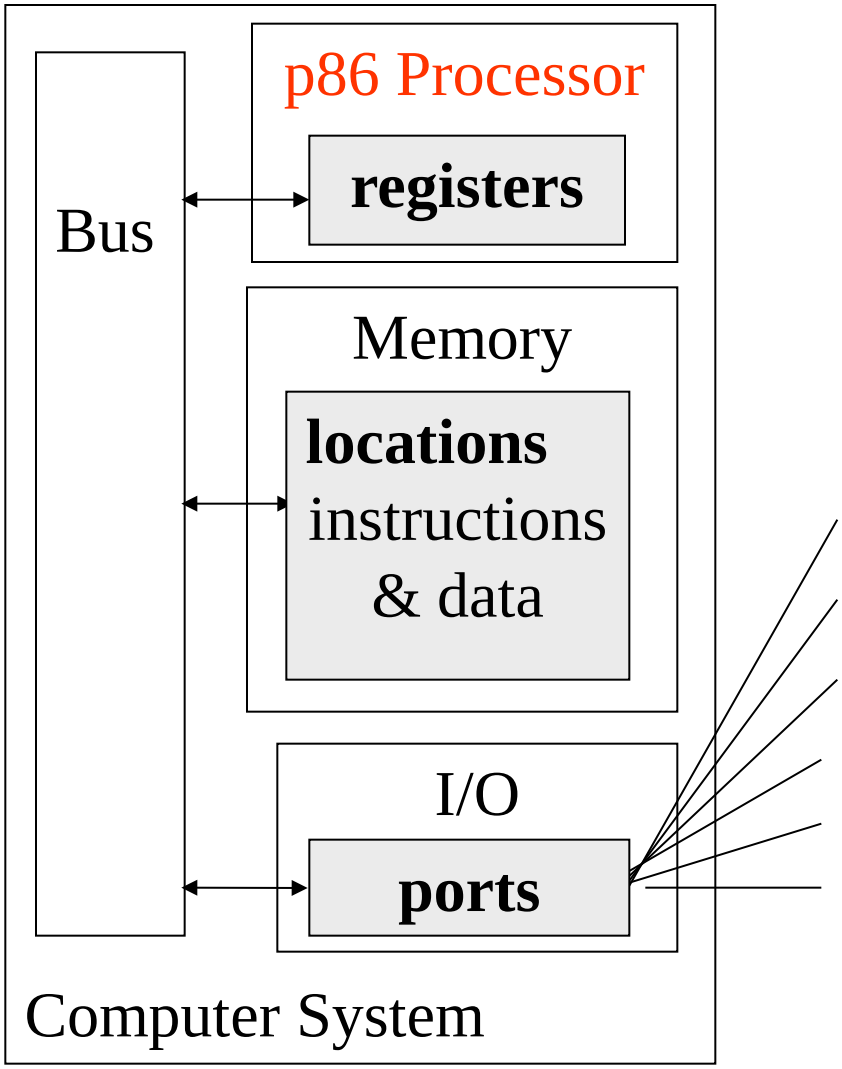
Virgo/Libra

Portions of Ch. 12, Appendix 12A, Ch. 13, and Ch. 14.2 of Stallings 9th ed have been incorporated into these slides.

Virgo/Libra Computer System Model



Connected Devices

keyboard

display

Virgo Computer System Model

- Registers: **16-bits** wide
- Memory Cells: **8-bit** contents within each location
  16-bit address → 2^{16} (or 64K) locations
- I/O Ports: read/write **8-bit** values per port #
  16-bit address → 64K ports

- We will model the behaviour of the components (and of programs!) in terms of these state variables

Stored Program Concept

1. instructions are encoded into binary values
2. encodings of instructions are loaded into memory
3. processor retrieves and executes instructions from memory (one at a time)

- instructions encodings have **variable length**
  - 1 to 6 bytes
  - not every binary value 1 to 6 bytes long corresponds to an instruction

p-86 Register Set

Four 16-Bit General Purpose Registers

- can access 16-bits, high (H) byte, low (L) byte

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
	8 bits	8 bits

P-86 Register Set (cont'd)

16-Bit Addressing Registers (no 8-bit access!)

IP	Instruction Pointer (i.e. PC)
SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index

P-86 Register Set (contd)

FLAGS Register (status flags – one bit/flag)

- 16-bit reg, but only some bits have meaning
- treat as individual bits, not 16-bit value
- ignore unused bits

ZF	Zero Flag
CF	Carry Flag
SF	Sign Flag
OF	Overflow Flag
IF	Interrupt Flag

data manipulation & conditional control flow

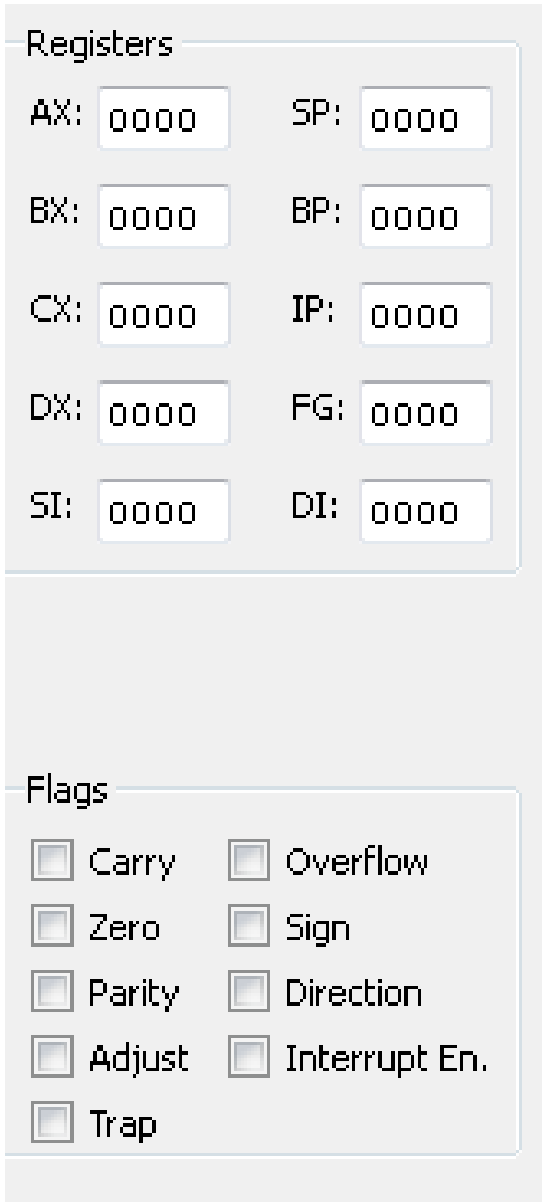
P-86 Register Set (contd)

Other Registers NOT in Programmer's Model

- support the execution of instructions
- **cannot** be accessed directly by programmers
- often larger than 16-bits
- temporary reg's (scratchpad values)

IR Instruction Register

Virgo Programmer's Model Screenshot



Data Transfer Example

MOV (Move) Instruction

syntax: MOV dest , src

semantics: dest := src

- copy src value to dest register
- register and memory operands only
  - *Will see later how to read/write from/to* I/O...

Register Addressing Mode

- allows a register to be an operand
  - as source: copy register value
  - as destination: write value to register

e.g. MOV AX, DX ; value in DX is
; copied to AX

→ AX := DX

- register addressing mode for both dest and src
- dest and src must be same size
  - MOV AH, CL ; This is OK
  - MOV AL, CX ; This is **not OK** (why?)

Syntax Note on Constants

- Decimal → default
- Hexadecimal:
  - **always** starts with digit 0 – 9
  - after start may include 0 – 9, and A – F
    - case insensitive for A – F
  - ends with "**H**" (or "**h**") or starts with a **0x**
- e.g. `1h, 23H, 0A2cH, 2a3Dh, 0x2a3D`

What does `AH` refer to ?????

Immediate Addressing Mode

- allows **constant** to be specified as **source**
  - source value assembled into the instruction
  - value obtained from IR as instruction executed

e.g. MOV AL, 5 ; AL is 8 bit dest

  - instruction encoding includes 8-bit value 05h
- what about: MOV AX, 5
  - 16-bit dest: encoding includes 16-bit value 0005h

- what about MOV 4, BH ; lets be ridiculous
  - dest as immediate value ?

Direct Addressing Mode

- specify the **address of a memory operand**
  - specify address as a constant value
  - address gets encoded as part of instruction
  - must be known when program is assembled !
- use square brackets “[” and “]” to clarify immediate vs. direct!
- [x] means x is the address of the operand

```
MOV     AX, 3FC0h    ;  AX := 3FC0h
MOV     AX, [ 3FC0h ] ; AX:= contents from 2 memory
                        cells starting at 3FC0h
```

endian ???

Byte Order – Endian Scheme

- What order do we store and fetch values that occupy more than one location?
- e.g. 12345678h can be stored in 4x 8-bit locations
  - what order should be used?

Byte Order (example: 12345678)

Address	Value (1)	Value (2)
184	12	78
185	34	56
186	56	34
187	78	12

- low
- high
- lsbyte
- each "part" is also stored according to the endian scheme

- i.e. read top down or bottom up?
  - Big endian: "big end first"
  - Little endian: "little end first"

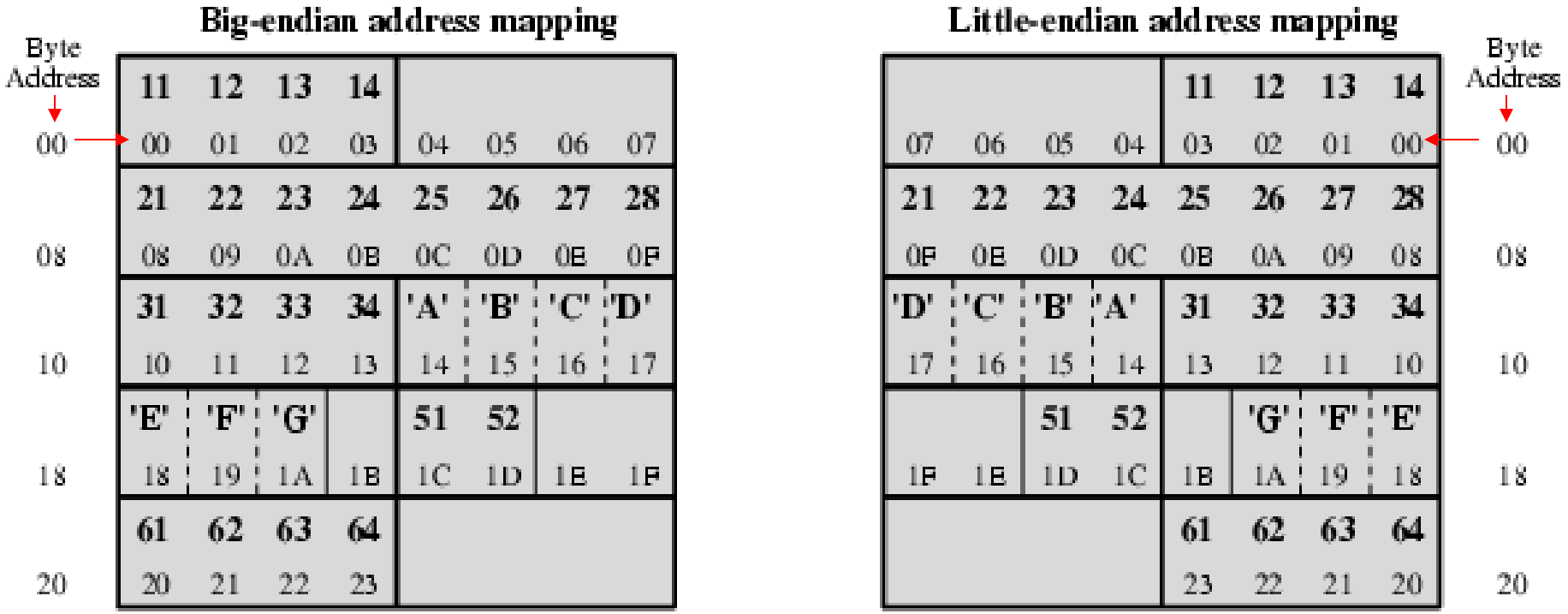
Byte Order Names

- The problem is called Endian
- Value (1) has the least significant byte in the highest address
  - This is called big-endian
- Value (2) has the least significant byte in the lowest address
  - This is called little-endian

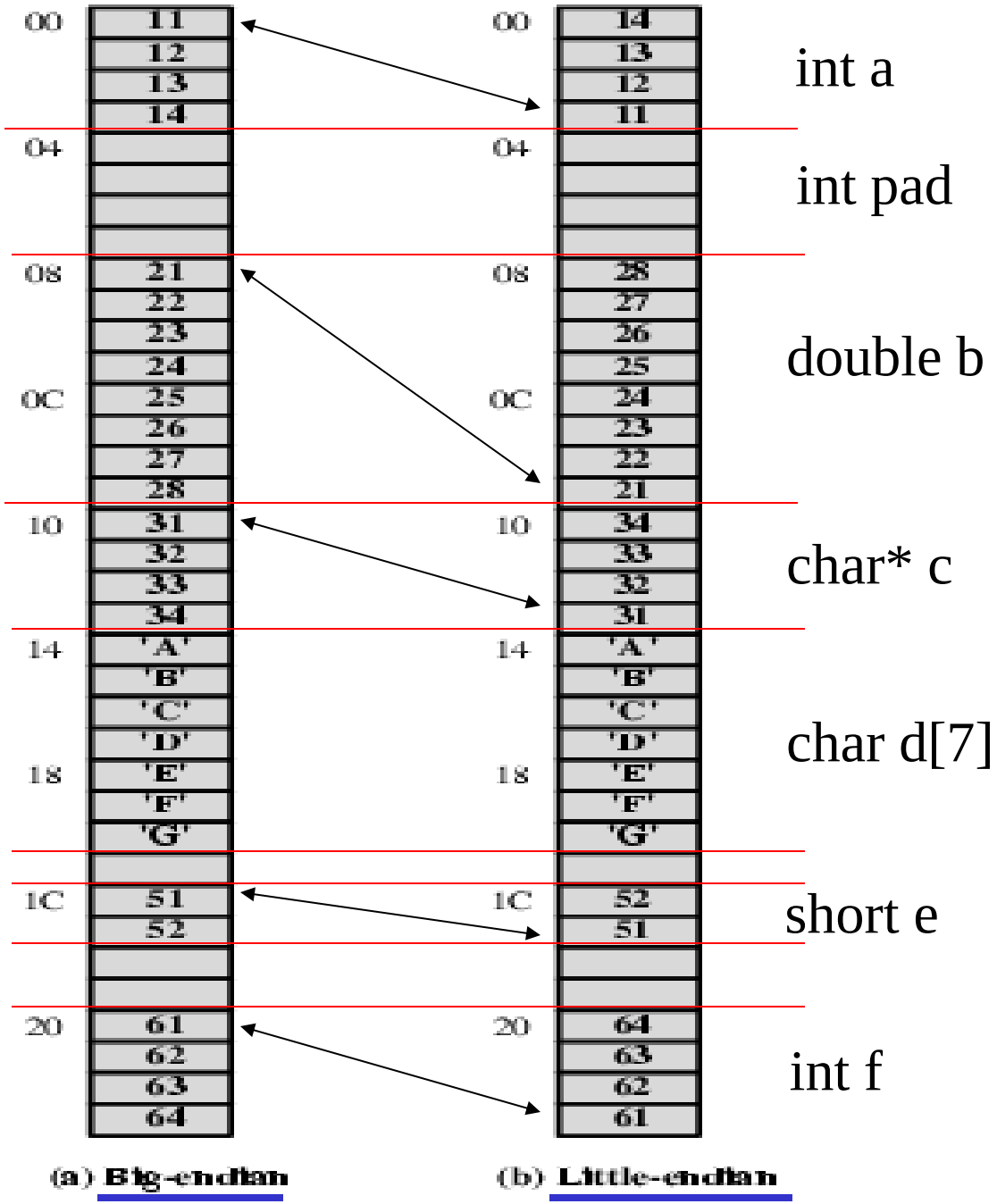
Example of C Data Structure

```
struct{
   int      a;     //0x1112_1314                    word
   int      pad;   //
   double   b;     //0x2122_2324_2526_2728          doubleword
   char*    c;     //0x3132_3334                    word
   char     d[7];  //'A','B','C','D','E','F','G'   byte array
   short    e;     //0x5152                         halfword
   int      f;     //0x6162_6364                    word
} s;
```

4 bytes



Alternative Endian Views of Memory Map



int a

int pad

double b

char* c

char d[7]

short e

int f

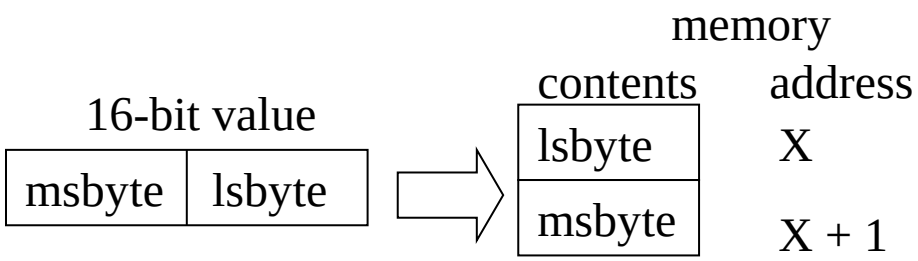
Standard Endian Scheme?

- Pentium (80x86), VAX: little-endian
- IBM 370, Motorola 680x0 (Mac), most RISC: big-endian
- Internet is big-endian
  - Makes writing Internet programs on PC more awkward!
  - WinSock provides `htoi` and `itoh` (Host to Internet & Internet to Host) functions to convert
- ARM processors can be configured to load/store multi-byte values in either big- or little-endian (using 'E-bit' control register)

Little Endian Memory Storage

little endian (Intel)

- the least significant byte at the low address



Simple Indirect Addressing Mode

more complex forms later!

- **simple form**: use current contents of a register as the address of an operand
- **only** these registers can be used:
  - for memory operands: **BX, BP, SI, DI**
  - for I/O operands: **DX**

e.g. MOV CX, [BX]

- contents of BX are used as the **address** of the **memory** containing value (16-bit, little endian) to load into CX
- only makes any sense if earlier instruction(s) put a useful address into BX!

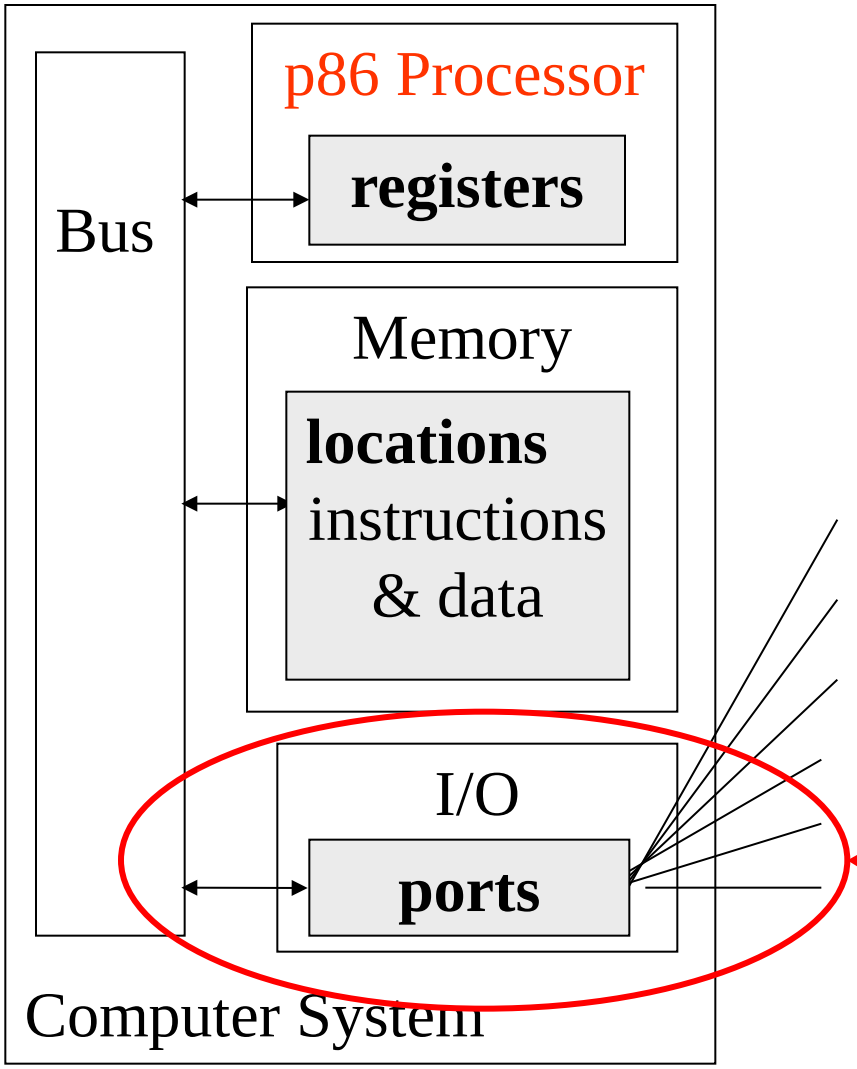
Indirect Addressing Mode (contd)

- potential ambiguity?

MOV CX, [BX] v.s. MOV CX, BX

- register, immediate and direct are **static** modes
  - operand bound to instruction at assemble-time
- indirect is a **dynamic** mode
  - operand bound to instruction at run-time
  - depends on values at time instruction executed
  - more powerful! ☺ more complicated! ☹

Recall…**Virgo Computer System Model**



Connected Devices

keyboard

display

We know how to access memory, but how do we access I/O ports?

Manipulating I/O Ports

- MOV allows only register and memory operands
- so . . . what accesses I/O ports?
  - IN read a value from a port
  - OUT write a value to a port
- IN / OUT: always use AL (or AX) and DX

I/O Port Example

- For now: **OUT DX, AL**
  - the 8-bit value in AL is written to the I/O port addressed by the contents of DX
    - → indirect mode to specify I/O port!
- Display character at the "current" cursor position:
  - write 7-bit ASCII encoded char to port **04E9h**
  - must set up DX to point to I/O port
  - must set up AL to contain char
  - write: display char and "advance" cursor

I/O Example (contd)

```
MOV DX,04E9h   ; set display port address
MOV AL,30h     ; char = '0' (ASCII!!)
OUT DX,AL      ; put char on display
```

(whew!)

- Enough for a simple program?

  MOV and OUT

Program Fragment

```
MOV     DX, 04E9h
MOV     AL, 'H'
OUT     DX, AL

MOV     AL, 'i'
OUT     DX, AL
HLT
```

output ‘H’

output ‘i’

Assembler Program

label definition *(like a bookmark in your code)*

```
; simple program that displays 'Hi'
start:
 mov    DX, 04E9h     ; get display port address
 mov    AL, 'H'        ; display 'H'
 out    DX, AL
;
 mov    AL, 'i'        ; display 'i'
 out    DX, AL
 hlt                   ;  STOP!
 .END    start
```

comments start with " ; "

where program starts

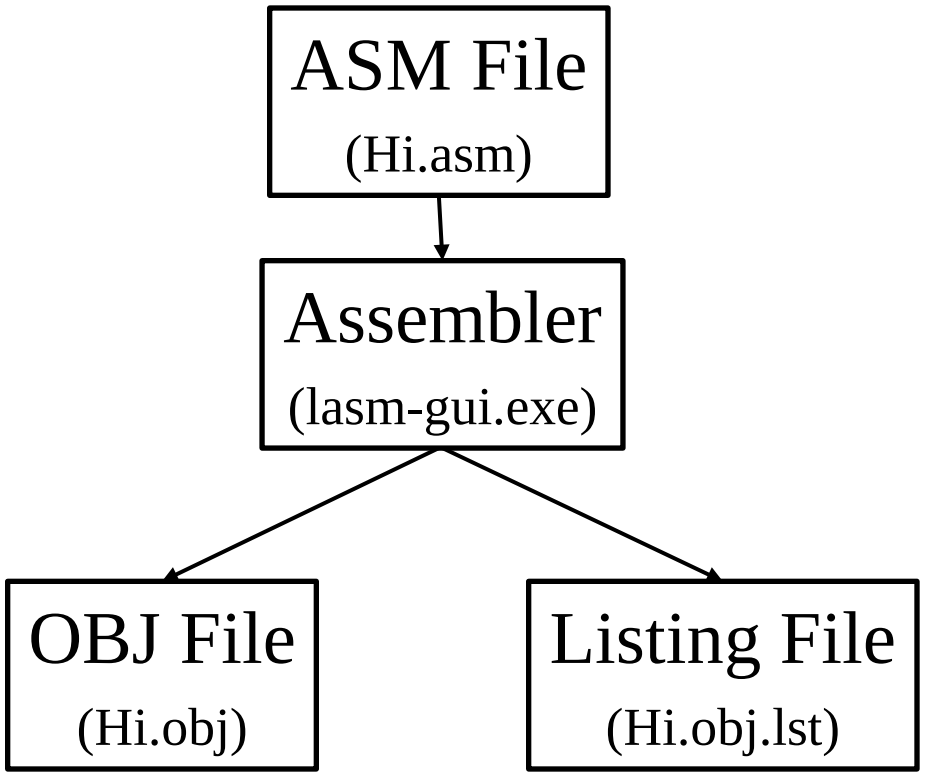
Directive

Assembler Program

- above is “source” code – human-oriented
- must be converted to binary values for loading into memory
- ASSEMBLER is a program that encodes / translates this sort of repr. of a program into the internal repr. required to run it.
- CROSS ASSEMBLERS translate into internal repr. for different machines

Assembly Process

(briefly, will see again in detail soon)



Operand Compatibility

- operands must have compatible sizes
- if register mode is used, then no ambiguity

  → operand size = register size
- But no register operands → potential ambiguity!
- Consider:

```
MOV     AX, 1
MOV     [ BX ], 1
MOV     [1234h], 0
```

16-bit operand
no ambiguity!

8-bit or 16-bit moves?
default?

Operand Compatibility (contd)

- need syntax to remove ambiguity
- qualify off-processor access using:

    WORD PTR word pointer – 16-bit operand

    BYTE PTR byte pointer – 8-bit operand

e.g. no ambiguity with:

```
MOV BYTE PTR  [ BX ],  1      ;8 bit dest
MOV WORD PTR  [1234h], 0  ;16 bit dest
```