

8086

MICROPROCESSOR

Microprocessor History

1st Generation

- Early 1970's
- 4 - bit - nibble data
- Low Performance
- Limited System Capabilities
- Low Cost
- Like PPS 4
- Special Purpose Applications
- Uses: Calculators, Toys

2nd Generation

- 1973-74
- 8-bit - 1 byte wide
- Higher Performance
- Larger System Capabilities
- Greater ease of Programming
- Like 8085
- Uses: Electronic Instruments, Cash Registers, Printers

3rd Generation

Mid 1970's
16 - bit
Higher Performance
Special and general
purpose microcomputer
applications
Like **8086**
Uses: Electronic
instruments, Word
Processing systems

8086 Microprocessor

Common Signals		
Name	Function	Type
AD15-AD0	Address/Data Bus	Bidirectional, 3-State
A19/S6-A16/S3	Address/Status	Output, 3-State
BHE/S7	Bus High Enable/Status	Output, 3-State
MN/MX	Minimum/Maximum Mode Control	Input
\overline{RD}	Read Control	Output, 3-State
\overline{TEST}	Wait On Test Control	Input
READY	Wait State Control	Input
RESET	System Reset	Input
NMI	Non-Maskable Interrupt Request	Input
INTR	Interrupt Request	Input
CLK	System Clock	Input
VCC	+5V	Input
GND	Ground	
Minimum Mode Signals (MN/MX = VCC)		
Name	Function	Type
HOLD	Hold Request	Input
HLDA	Hold Acknowledge	Output
\overline{WR}	Write Control	Output, 3-State
M/ \overline{IO}	Memory/IO Control	Output, 3-State
DT/ \overline{R}	Data Transmit/Receive	Output, 3-State
\overline{DEN}	Data Enable	Output, 3-State
ALE	Address Latch Enable	Output
\overline{INTA}	Interrupt Acknowledge	Output
Maximum Mode Signals (MN/MX = GND)		
Name	Function	Type
$\overline{RQ}/\overline{GT1}, 0$	Request/Grant Bus Access Control	Bidirectional
\overline{LOCK}	Bus Priority Lock Control	Output, 3-State
S2-S0	Bus Cycle Status	Output, 3-State
QS1, QS0	Instruction Queue Status	Output

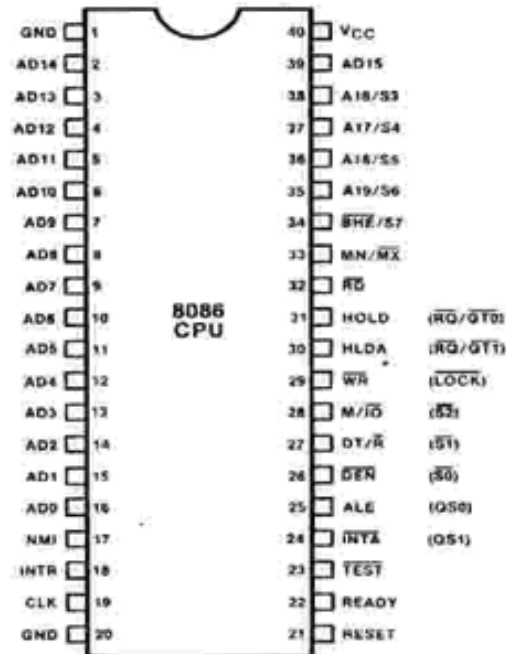


Figure 4-1. 8086 Pin Definitions

The world's most popular architecture

- An extension of 8080 8 - bit Microprocessor from Intel Corporation and cousin of Z80.
- Mainly evolved due to requirements for larger memory systems.
- With Vast Instructions set
- **The World's most popular architecture 8086** - 1st 16-bit microprocessor Introduced by Intel Corporation in 1978.

8086 Features

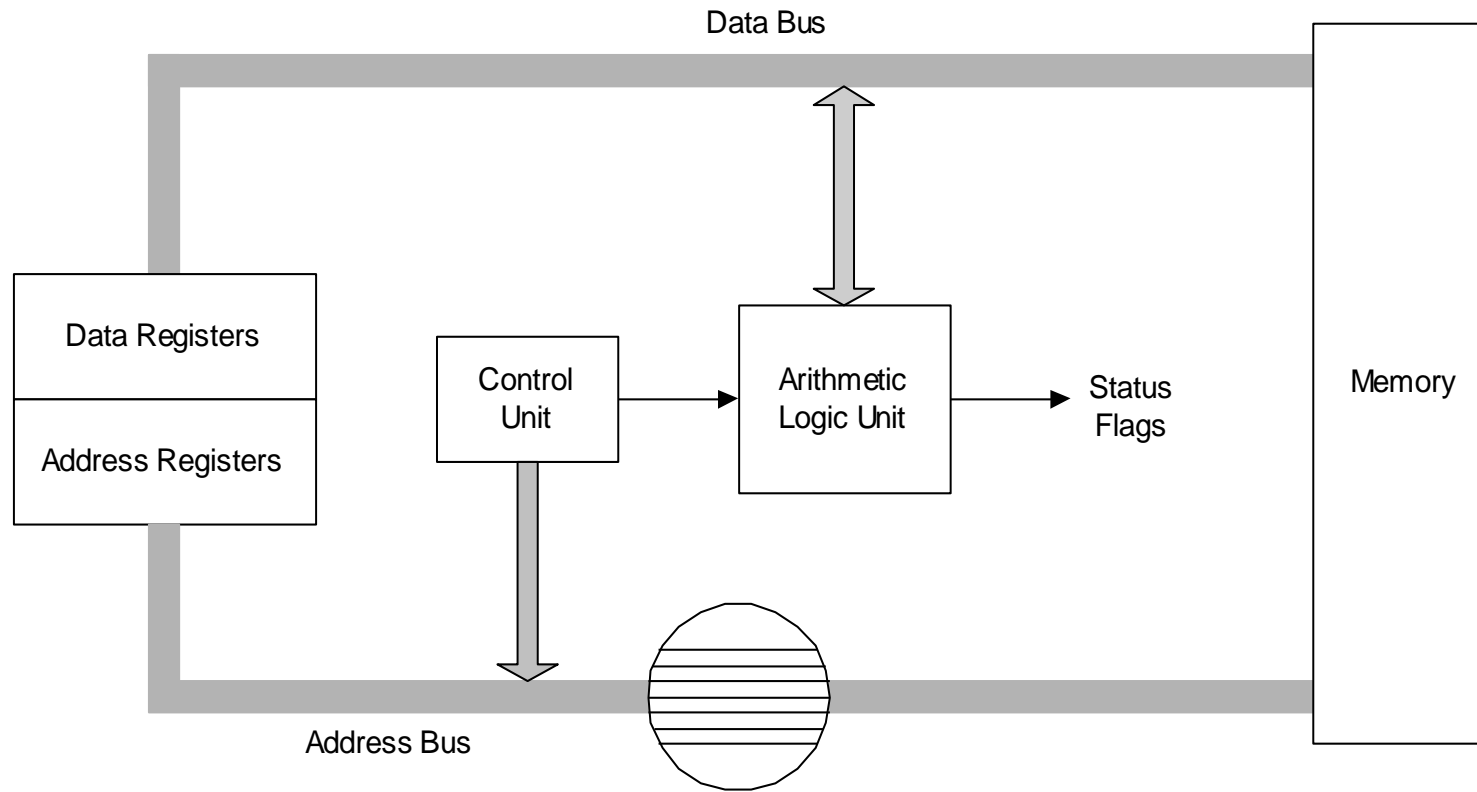
- 16-bit Arithmetic Logic Unit
- 16-bit data bus (8088 has 8-bit data bus)
- 20-bit address bus - $2^{20} = 1,048,576 = 1 \text{ meg}$

The address refers to a byte in memory. In the 8088, these bytes come in on the 8-bit data bus. In the 8086, bytes at even addresses come in on the low half of the data bus (bits 0-7) and bytes at odd addresses come in on the upper half of the data bus (bits 8-15).

The 8086 can read a 16-bit word at an even address in one operation and at an odd address in two operations. The 8088 needs two operations in either case.

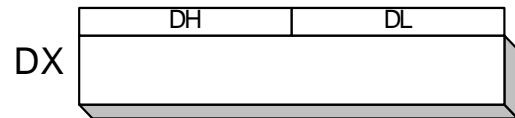
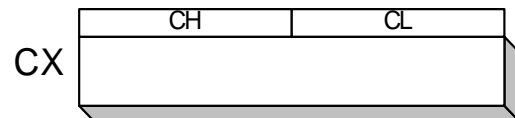
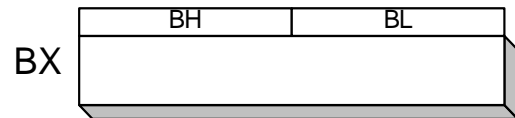
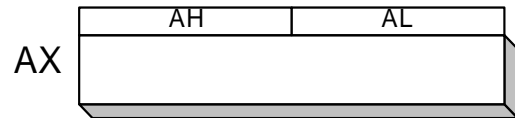
The least significant byte of a word on an 8086 family microprocessor is at the lower address.

Simplified CPU Design



Intel 16-bit Registers

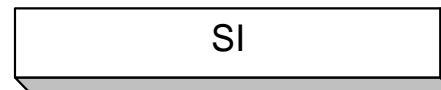
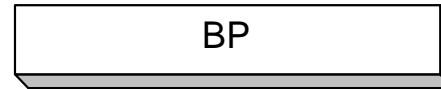
General Purpose



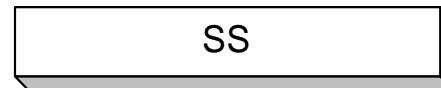
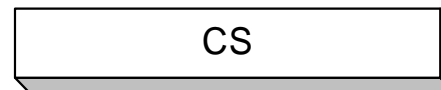
Status and Control



Index



Segment



8086 Architecture

- The 8086 has two parts, the Bus Interface Unit (BIU) and the Execution Unit (EU).
- The BIU fetches instructions, reads and writes data, and computes the 20-bit address.
- The EU decodes and executes the instructions using the 16-bit ALU.
- The BIU contains the following registers:
 - IP - the Instruction Pointer
 - CS - the Code Segment Register
 - DS - the Data Segment Register
 - SS - the Stack Segment Register
 - ES - the Extra Segment Register

The BIU fetches instructions using the CS and IP, written CS:IP, to construct the 20-bit address. Data is fetched using a segment register (usually the DS) and an effective address (EA) computed by the EU depending on the addressing mode.

The EU contains the following 16-bit registers:

AX - the Accumulator

BX - the Base Register

CX - the Count Register

DX - the Data Register

SP - the Stack Pointer \ defaults to stack segment

BP - the Base Pointer /

SI - the Source Index Register

DI - the Destination Register

These are referred to as general-purpose registers, although, as seen by their names, they often have a special-purpose use for some instructions.

The AX, BX, CX, and DX registers can be considered as two 8-bit registers, a High byte and a Low byte. This allows byte operations and compatibility with the previous generation of 8-bit processors, the 8080 and 8085. 8085 source code could be translated in 8086 code and assembled. The 8-bit registers are:

AX --> AH,AL

BX --> BH,BL

CX --> CH,CL

DX --> DH,DL

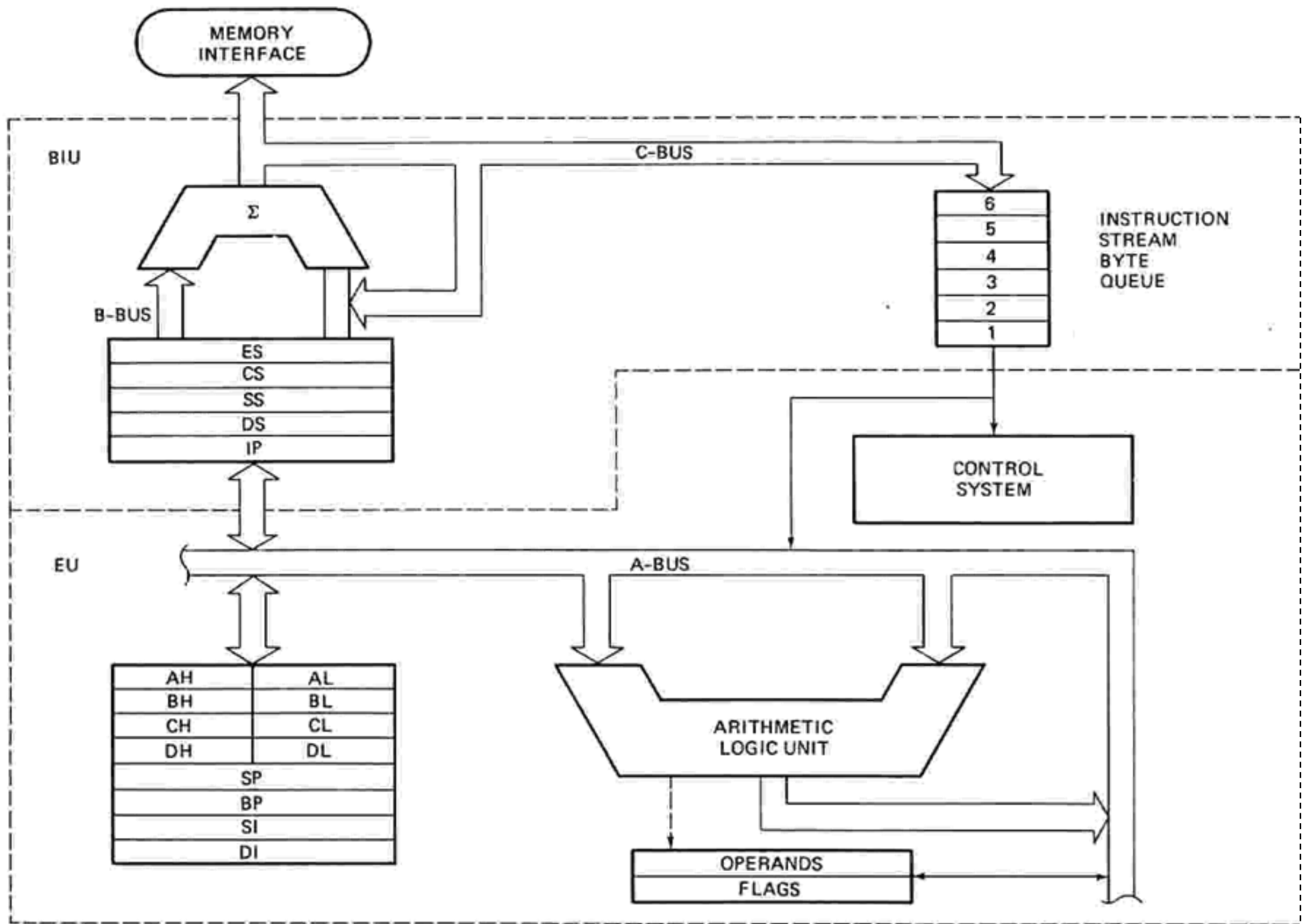


FIGURE 2-7 8086 internal block diagram. (Intel Corp.)

Registers

<i>Class</i>	<i>Register</i>	<i>Purpose</i>
Data:	AX,BX	“general” purpose
	CX	string and loop ops only
	DX	mult/div and I/O only
Address:	SP	stack pointer
	BP	base pointer (can also use BX)
	SI,DI	index registers
Segment:	CS	code segment
	SS	stack segment
	DS	data segment
	ES	extra segment
Control:	IP	instruction pointer (lower 16 bit of PC)
	FLAGS	C, Z, N, B, P, V and 3 control bits

8086 Programmer's Model

BIU registers
(20 bit adder)

ES
CS
SS
DS
IP

Extra Segment
Code Segment
Stack Segment
Data Segment
Instruction Pointer

EU registers

AX
BX
CX
DX

AH	AL
BH	BL
CH	CL
DH	DL
SP	
BP	
SI	
DI	
FLAGS	

Accumulator
Base Register
Count Register
Data Register
Stack Pointer
Base Pointer
Source Index Register
Destination Index Register

8086/88 internal registers 16 bits (2 bytes each)

AX	AH	AL	Accumulator
BX	BH	BL	Base
CX	CH	CL	Count
DX	DH	DL	Data

} Data group

AX, BX, CX and DX are two bytes wide and each byte can be accessed separately

SP	Stack pointer
BP	Base pointer
SI	Source index
DI	Destination index
IP	Instruction pointer

} Pointer and index group

These registers are used as memory pointers.

Flags _H	Flags _L	Status and control flags
--------------------	--------------------	--------------------------

Flags will be discussed later

ES	Extra
CS	Code
DS	Data
SS	Stack

} Segment group

Segment registers are used as base address for a segment in the 1 M byte of memory

The 8086/8088 Microprocessors: Registers

- Registers
 - Registers are in the CPU and are referred to by specific names
 - Data registers
 - Hold data for an operation to be performed
 - There are 4 data registers (AX, BX, CX, DX)
 - Address registers
 - Hold the address of an instruction or data element
 - Segment registers (CS, DS, ES, SS)
 - Pointer registers (SP, BP, IP)
 - Index registers (SI, DI)
 - Status register
 - Keeps the current status of the processor
 - On an IBM PC the status register is called the FLAGS register
 - In total there are fourteen 16-bit registers in an 8086/8088

Data Registers: AX, BX, CX, DX

- Instructions execute faster if the data is in a register
- AX, BX, CX, DX are the data registers
- Low and High bytes of the data registers can be accessed separately
 - AH, BH, CH, DH are the high bytes
 - AL, BL, CL, and DL are the low bytes
- Data Registers are general purpose registers but they also perform special functions
- AX
 - Accumulator Register
 - Preferred register to use in arithmetic, logic and data transfer instructions because it generates the shortest Machine Language Code
 - Must be used in multiplication and division operations
 - Must also be used in I/O operations

- **BX**

- Base Register
- Also serves as an address register

- **CX**

- Count register
- Used as a loop counter
- Used in shift and rotate operations

- **DX**

- Data register
- Used in multiplication and division
- Also used in I/O operations

Pointer and Index Registers

- Contain the offset addresses of memory locations
- Can also be used in arithmetic and other operations
- **SP: Stack pointer**
 - Used with SS to access the stack segment
- **BP: Base Pointer**
 - Primarily used to access data on the stack
 - Can be used to access data in other segments
- **SI: Source Index register**
 - is required for some string operations
 - When string operations are performed, the SI register points to memory locations in the data segment which is addressed by the DS register. Thus, SI is associated with the DS in string operations.

- **DI: Destination Index register**
 - is also required for some string operations.
 - When string operations are performed, the DI register points to memory locations in the data segment which is addressed by the ES register. Thus, DI is associated with the ES in string operations.
- The SI and the DI registers may also be used to access data stored in arrays

Segment Registers - CS, DS, SS and ES

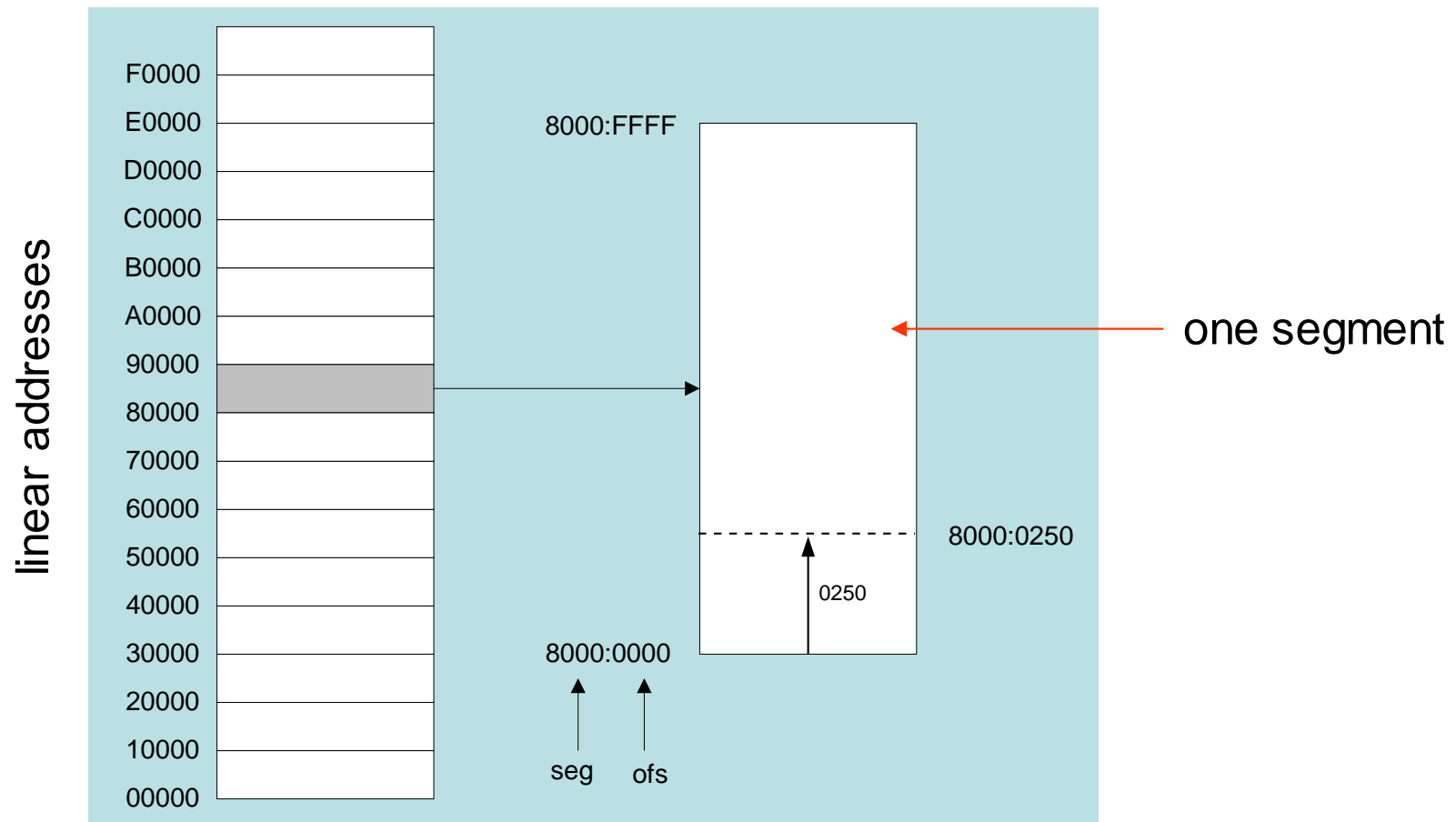
- Are Address registers
- Store the memory addresses of instructions and data
- Memory Organization
 - Each byte in memory has a 20 bit address starting with 0 to $2^{20}-1$ or 1 meg of addressable memory
 - Addresses are expressed as 5 hex digits from 00000 - FFFFF
 - Problem: But 20 bit addresses are **TOO BIG** to fit in 16 bit registers!
 - Solution: Memory Segment
 - Block of 64K (65,536) consecutive memory bytes
 - A segment number is a 16 bit number
 - Segment numbers range from 0000 to FFFF
 - Within a segment, a particular memory location is specified with an offset
 - An offset also ranges from 0000 to FFFF

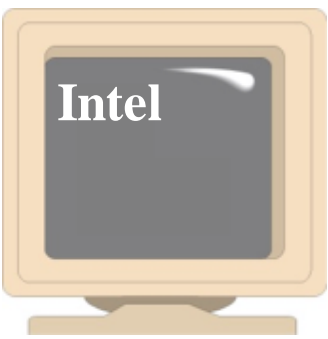
Segmented Memory Architecture (*real mode*)

- A **segment** addresses 64K of memory
- A **segment register** contains the starting location of a segment
 - the absolute location of a segment can be obtained by appending a hexadecimal zero
- An **offset** is the distance from the beginning of a segment to a particular instruction or variable

Segmented Memory

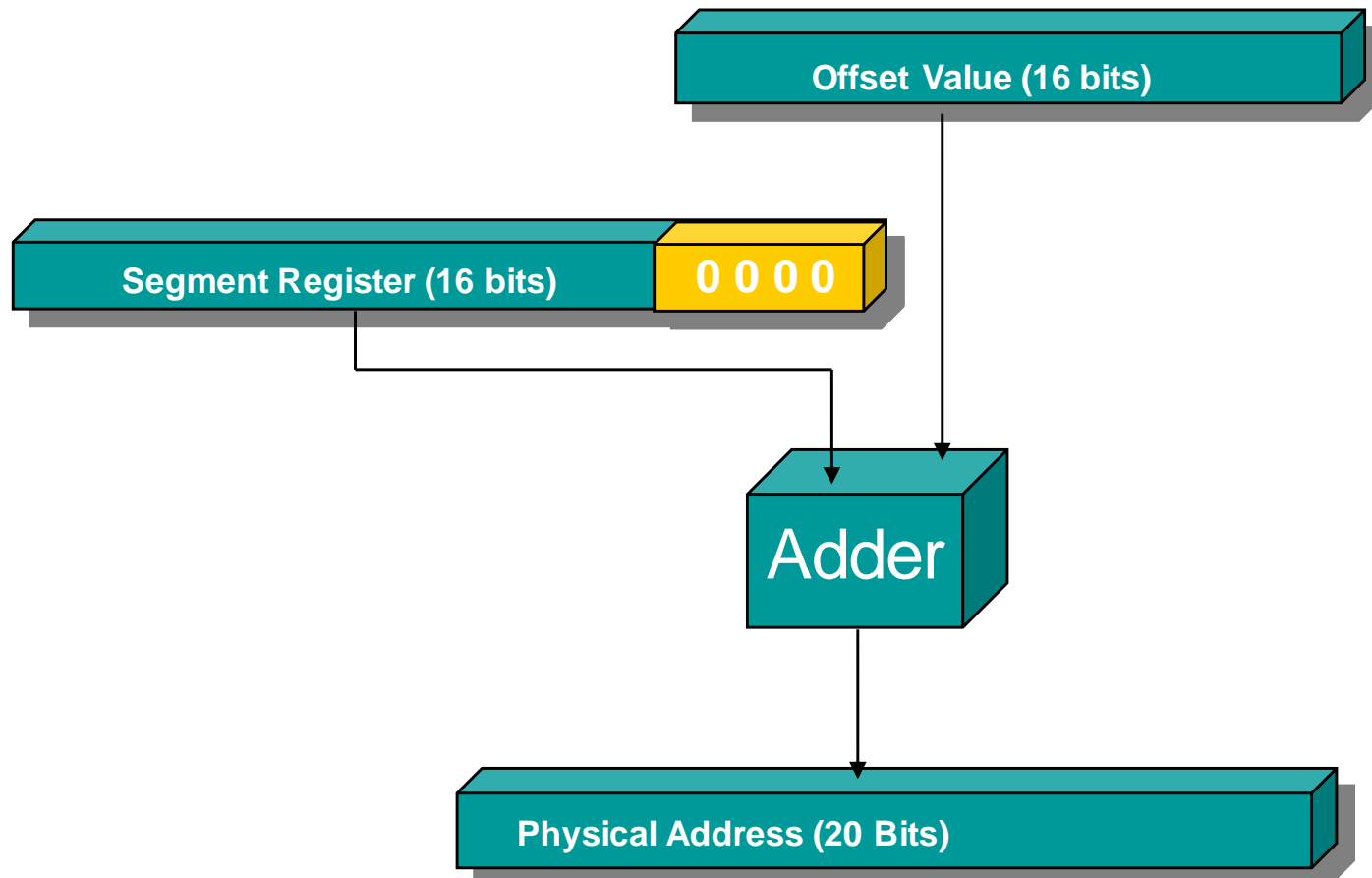
Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset

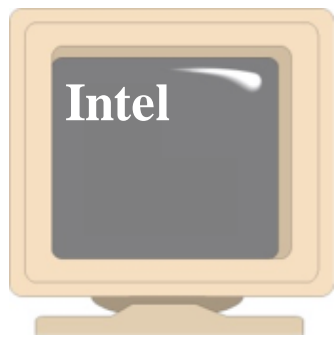




Memory Address Generation

- The BIU has a dedicated adder for determining physical memory addresses

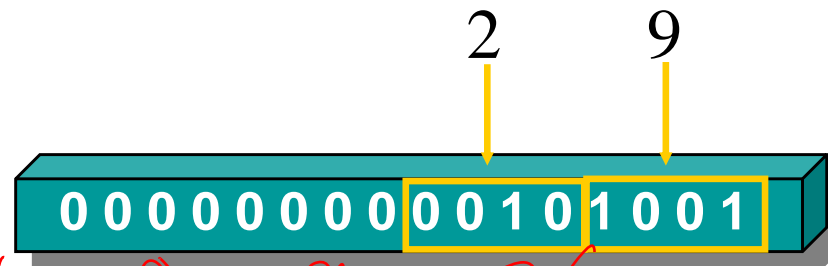




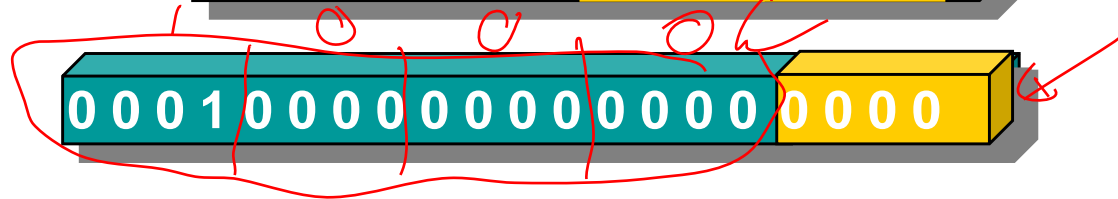
Example Address Calculation

- If the data segment starts at location 1000h and a data reference contains the address 29h where is the actual data?

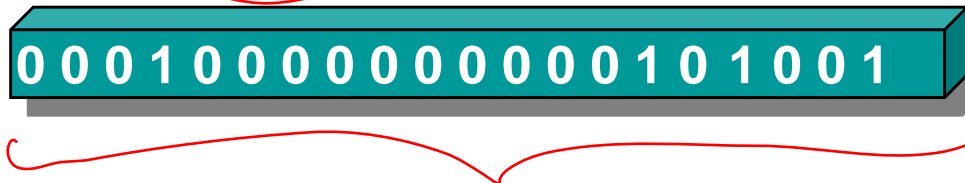
Offset:



Segment:



Address:



Segment:Offset Address

- Logical Address is specified as **segment:offset**
- Physical address is obtained by shifting the segment address 4 bits to the left and adding the offset address
- Thus the physical address of the logical address **A4FB:4872** is

$$\begin{array}{r} \text{A4FB0} \\ + \text{4872} \\ \hline \text{A9822} \end{array}$$

Segment *offset*

Physical Address

Example

- If DS=7FA2H and the offset is 438EH

a) Calculate the physical address

$$7FA20 + 438E = 83DAE$$

b) calculate the lower range

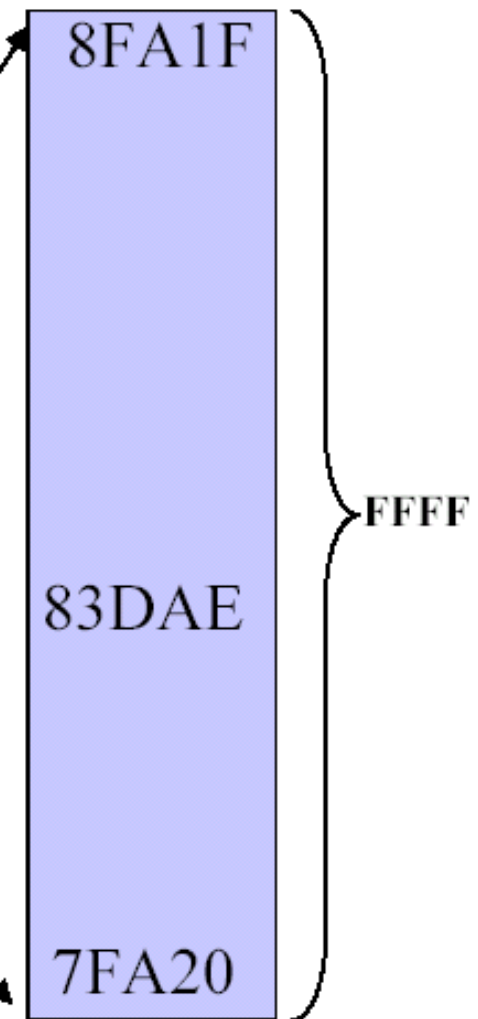
$$7FA20 + 0000 = 7FA20$$

c) Calculate the upper range of the data segment

$$7FA20 + FFFF = 8FA1F$$

d) Show the logical Address

7FA2:438E



Your turn . . .

What linear address corresponds to the segment/offset address 028F:0030?

$$028F0 + 0030 = 02920$$

Always use hexadecimal notation for addresses.

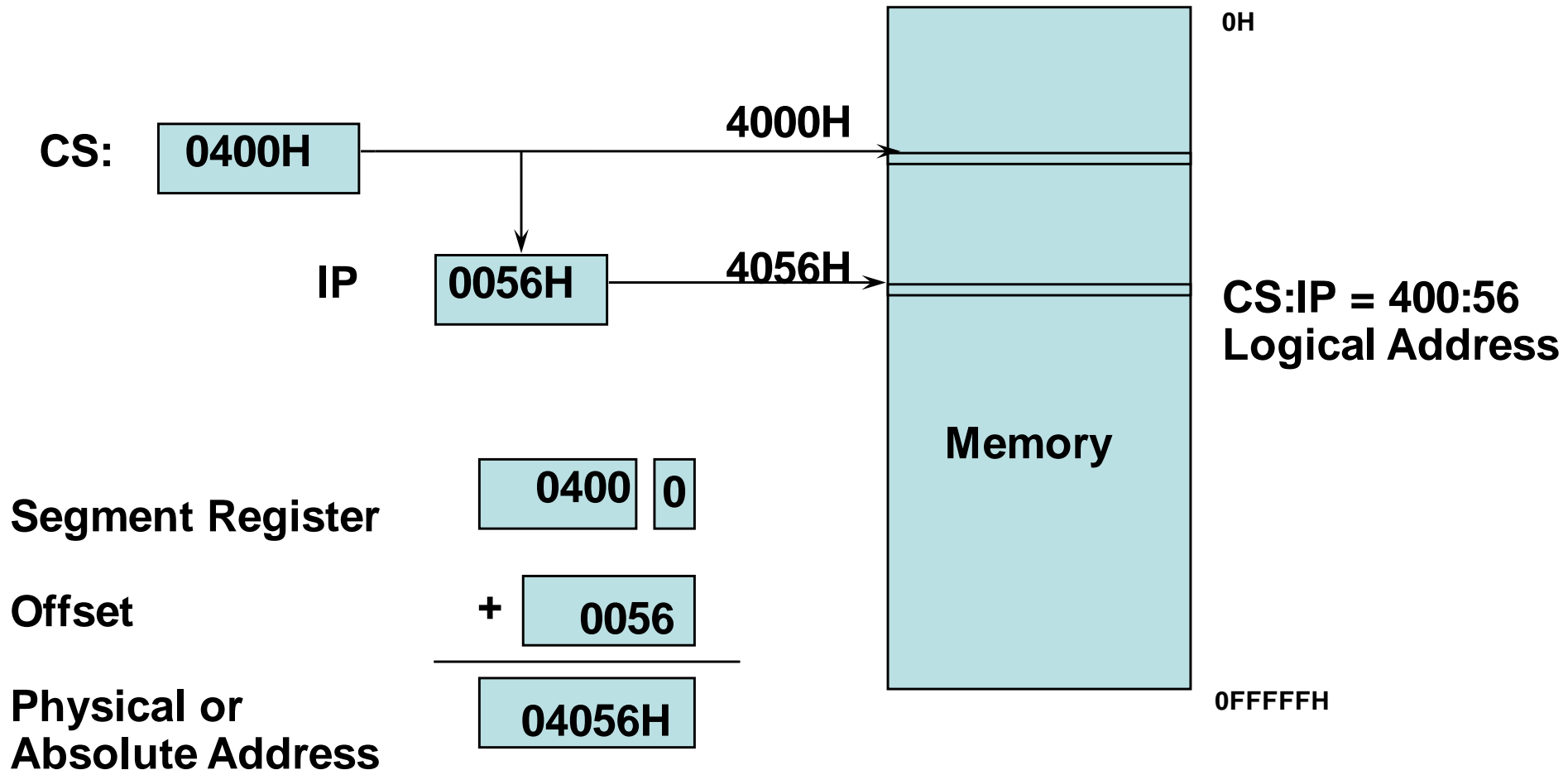
Your turn . . .

What segment addresses correspond to the linear address 28F30h?

Many different segment-offset addresses can produce the linear address 28F30h. For example:

28F0:0030, 28F3:0000, 28B0:0430, . . .

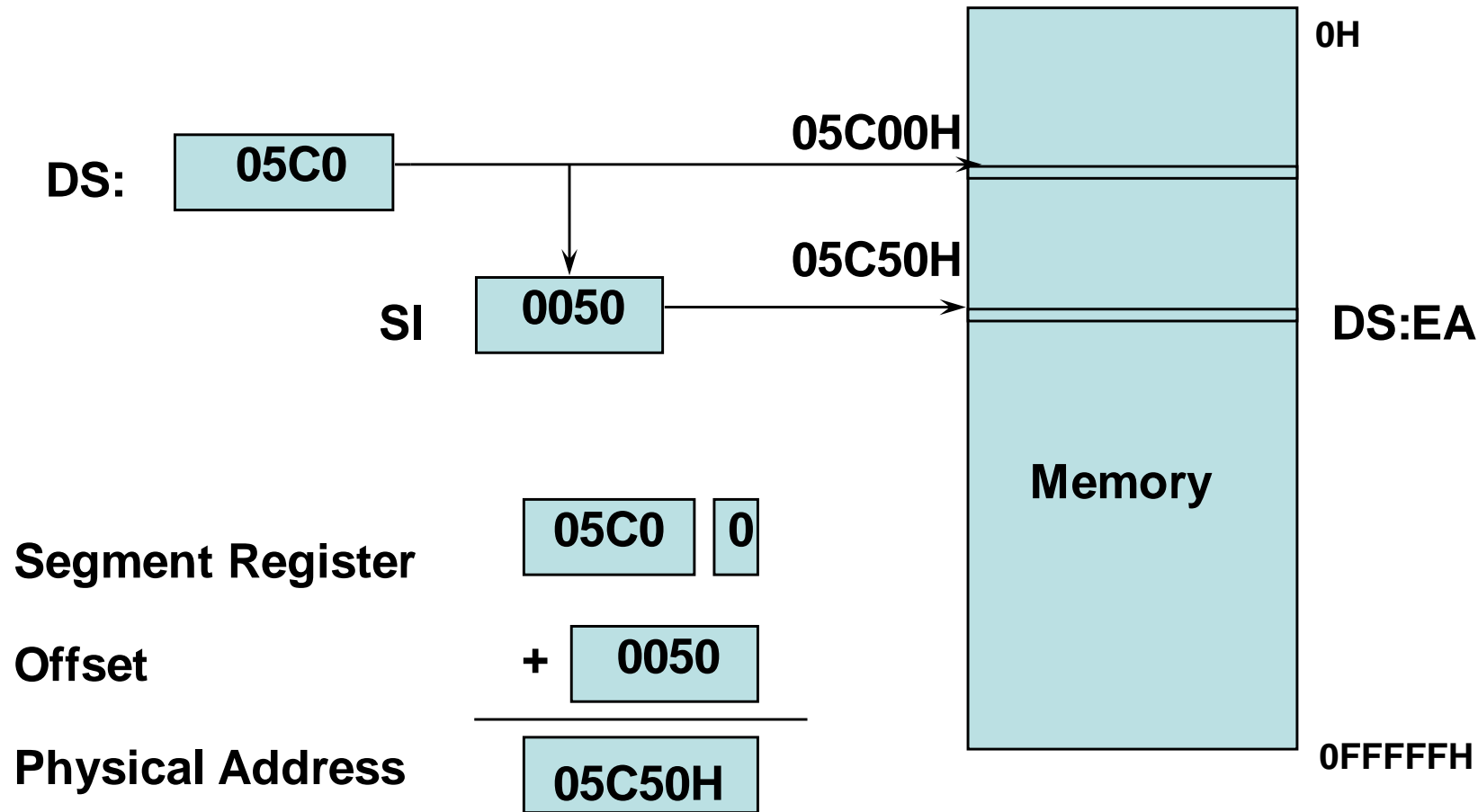
The Code Segment



The offset is the distance in bytes from the start of the segment.
The offset is given by the IP for the Code Segment.
Instructions are always fetched with using the CS register.

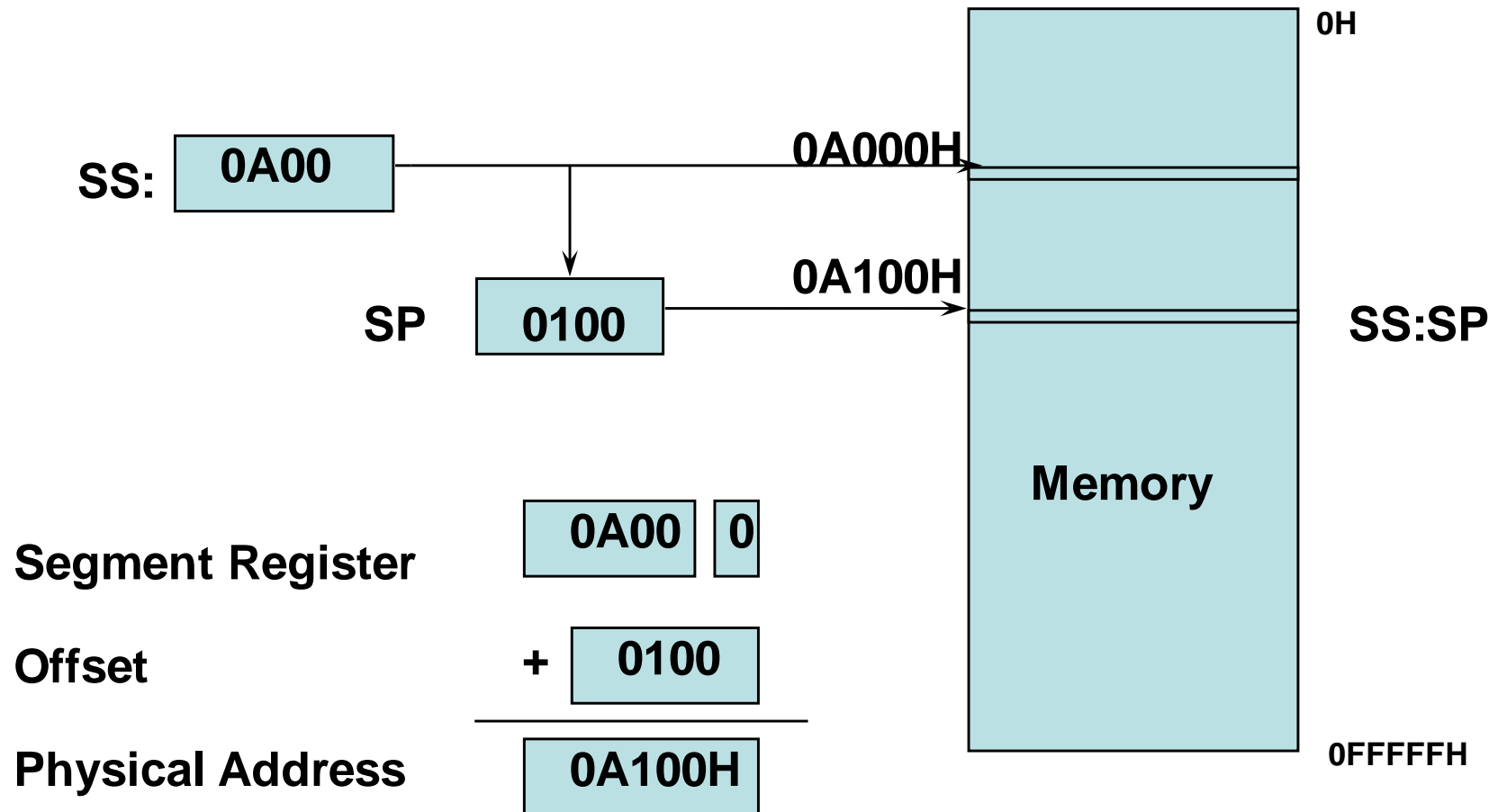
The physical address is also called the absolute address.

The Data Segment



Data is usually fetched with respect to the DS register.
The effective address (EA) is the offset.
The EA depends on the addressing mode.

The Stack Segment



The offset is given by the SP register.

The stack is always referenced with respect to the stack segment register.

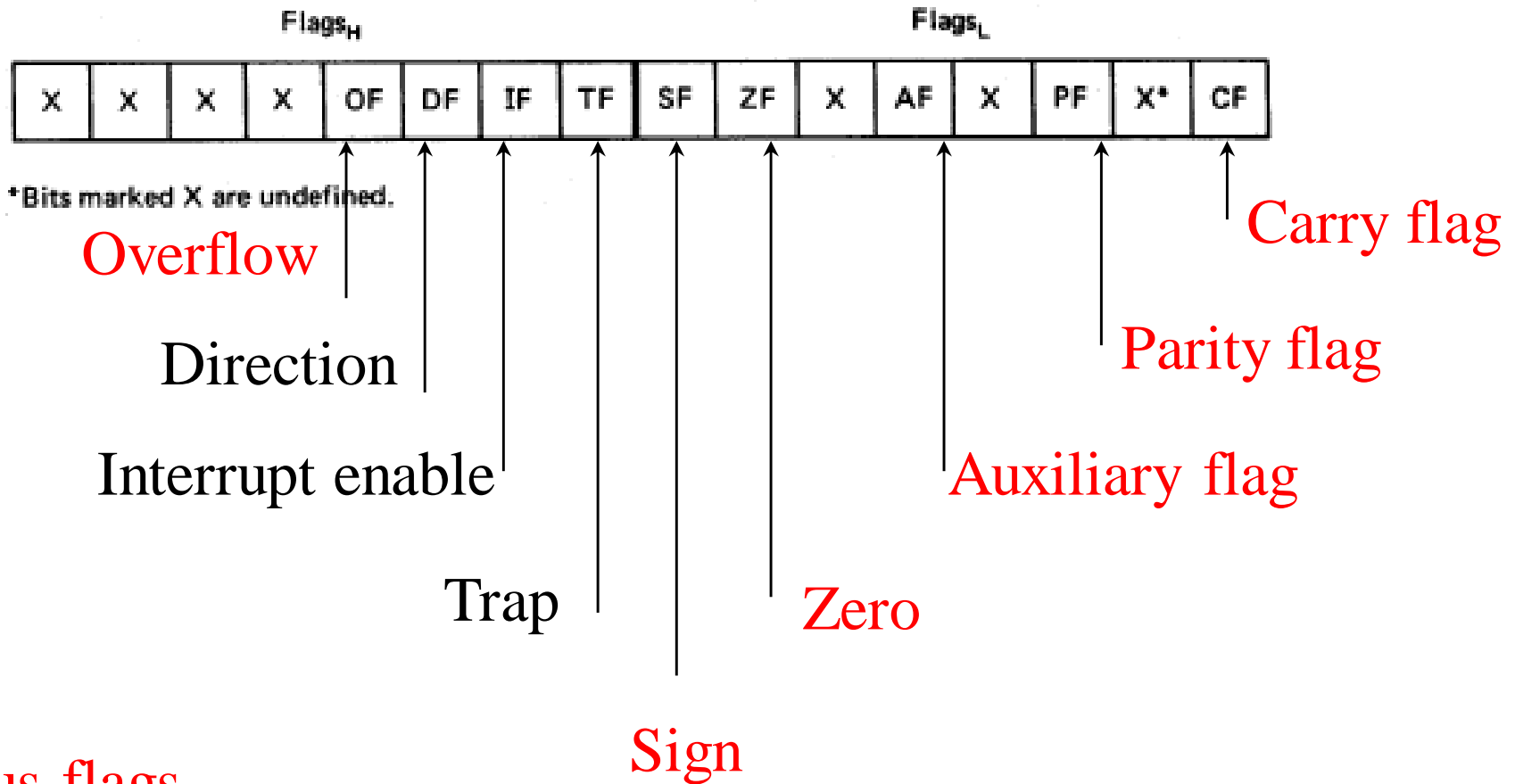
The stack grows toward decreasing memory locations.

The SP points to the last or top item on the stack.

PUSH - pre-decrement the SP

POP - post-increment the SP

Flags



6 are status flags
3 are control flag

Flag Register

- Conditional flags:
 - They are set according to some results of arithmetic operation. You do not need to alter the value yourself.
- Control flags:
 - Used to control some operations of the MPU. These flags are to be set by you in order to achieve some specific purposes.

Flag					O	D	I	T	S	Z		A		P		C
Bit no.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- CF (carry) Contains carry from leftmost bit following arithmetic, also contains last bit from a shift or rotate operation.

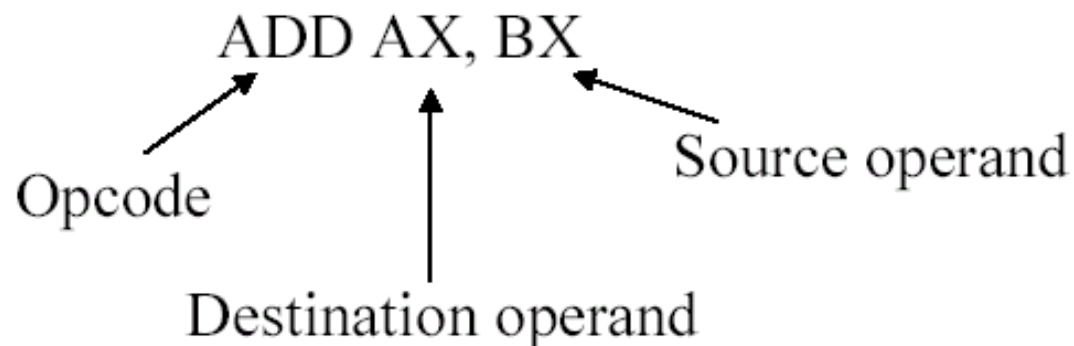
Flag Register

- OF (overflow) Indicates overflow of the leftmost bit during arithmetic.
- DF (direction) Indicates left or right for moving or comparing string data.
- IF (interrupt) Indicates whether external interrupts are being processed or ignored.
- TF (trap) Permits operation of the processor in single step mode.

- SF (sign) Contains the resulting sign of an arithmetic operation (1=negative)
- ZF (zero) Indicates when the result of arithmetic or a comparison is zero. (1=yes)
- AF (auxiliary carry) Contains carry out of bit 3 into bit 4 for specialized arithmetic.
- PF (parity) Indicates the number of 1 bits that result from an operation.


Software


- The sequence of commands used to tell a microcomputer what to do is called a program,
- Each command in a program is called an instruction
- 8088 understands and performs operations for 117 basic instructions
- The native language of the IBM PC is the machine language of the 8088
- A program written in machine language is referred to as machine code
- In 8088 assembly language, each of the operations is described by alphanumeric symbols instead of just 0s or 1s.



Instructions

LABEL: INSTRUCTION ; COMMENT


Address identifier


Does not generate any machine code

Ex. START: MOV AX,BX ; copy BX into AX

• Addressing modes

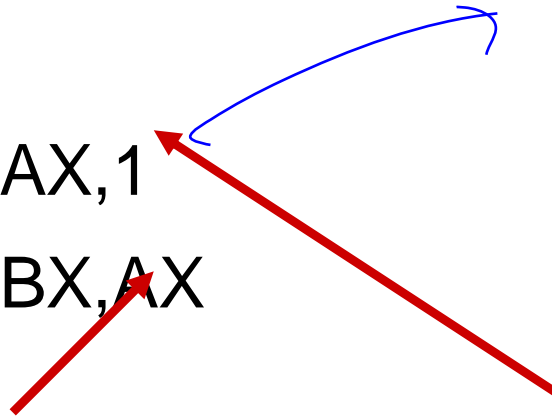
- **Register** and **immediate** modes we have already seen

MOV AX,1
MOV BX,AX

register

immediate

AX = 1
AX ← 1



3F03 - 80x86 assembler

- Typical addressing modes
 - Absolute address mode

MOV AX,[0200]

value stored in memory location DS:0200

mov AX, 200

mov AX, [BX]

3F03 - 80x86 assembler

- Typical addressing modes
 - Register indirect

MOV AX,[BX]



value stored at address contained in DS:BX

3F03 - 80x86 assembler

- Typical addressing modes
 - Displacement

MOV DI,4

MOV AX,[0200+DI]


value stored at DS:0204

MOV AX, [0204]

3F03 - 80x86 assembler

- Typical addressing modes
 - Indexed

```
MOV BX,0200
```

```
MOV DI,4
```

```
MOV AX,[BX+DI]
```



value stored at DS:0204

3F03 - 80x86 assembler

- Typical addressing modes

- Memory indirect

```
MOV DI,0204
```

```
MOV BX,[DI]
```

```
MOV AX,[BX]
```

If DS:0204 contains 0256,
then AX will contain
whatever is stored at
DS:0256

3F03 - 80x86 assembler

- Typical addressing modes

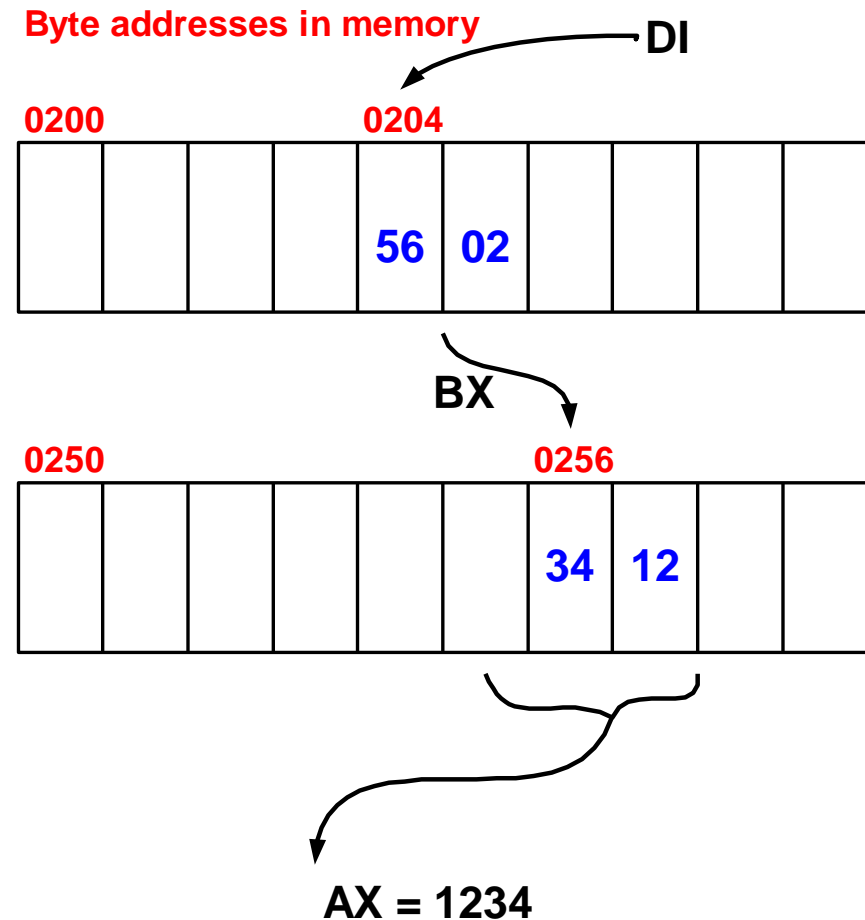
- Memory indirect

```
MOV DI,0204
```

```
MOV BX,[DI]
```

```
MOV AX,[BX]
```

If DS:0204 contains 0256,
then AX will contain
whatever is stored at
DS:0256



8086 in Maximum Mode

The IBM PC is a maximum mode 8088 system. When an 8086/8088 is used in the maximum mode (MN/MX pin grounded) it requires the use of an 8288 Bus Controller. The system can support multiple processors on the system bus by the use of an 8289 Bus Arbiter.

The following signals now come from the 8288: ALE, DT/R', DEN, and INTA'.

The M/IO', RD', and WR' signals are replaced by:

MRDC' - memory read command

MWTC' - memory write command

IORC' - I/O read command

IOWC' - I/O write command

AMWC' - Advanced memory write command

AIOWC' - Advanced I/O write command

The advanced commands become active earlier in the cycle to give devices an earlier indication of a write operation.

8086 Maximum Mode

When in the maximum mode, the 8086/8088 has 3 status lines that are connected to the 8288 and provide it with the information it needs to generate the system bus signals. The information provided by the status bits is as follows.

S2'	S1'	S0'	<u>operation</u>	<u>signal</u>
0	0	0	Interrupt Acknowledge	INTA'
0	0	1	Read I/O port	IORC'
0	1	0	Write I/O port	IOWC', AIOWC'
0	1	1	Halt	none
1	0	0	Instruction Fetch	MRDC'
1	0	1	Read Memory	MRDC'
1	1	0	Write Memory	MWTC', AMWC'
1	1	1	Passive	none

Direct Memory Access - DMA

DMA allows data to go between memory and a peripheral, such as a disk drive, without going through the cpu.

The DMA controller takes over the address bus, data bus, and control bus. The 8237A DMA Controller is a commonly used device and is in the IBM PC.

Figure 11-4 is a simplified block diagram showing the use of a DMA controller. For example, to read a disk file the following operations occur.

- 1. Send a series of commands to the disk controller to find and read a block of data.**
- 2. When the controller has the first byte of the block, it sends a DMA request DREQ to the DMA controller.**
- 3. If that input of the DMA controller is unmasked, the DMA controller sends a hold-request HQR to the cpu.**
- 4. The cpu responds with a hold-acknowledge HLDA and floats its buses.**
- 5. The DMA controller then takes control of the buses.**
- 6. The DMA controller sends out the memory address and DMA acknowledge DACK0 to the disk controller.**
- 7. The DMA controller asserts the MEMW' and IOR' lines.**

Memory

Terminology

Volatile - data is lost when power is turned off.

Nonvolatile - retains data when powered off.

Random Access - all data locations accessed in the same amount of time.

Sequential Access - data accessed in varying amounts of time, e.g., tape.

ROM - Read Only Memory.

RAM - Random Access Memory

By convention, RAM in a PC is really Read/Write Memory and ROM (EPROM) in a PC, although random access memory, is not referred to as RAM.

Examples

VOLATILE

Static RAM

Dynamic RAM

NONVOLATILE

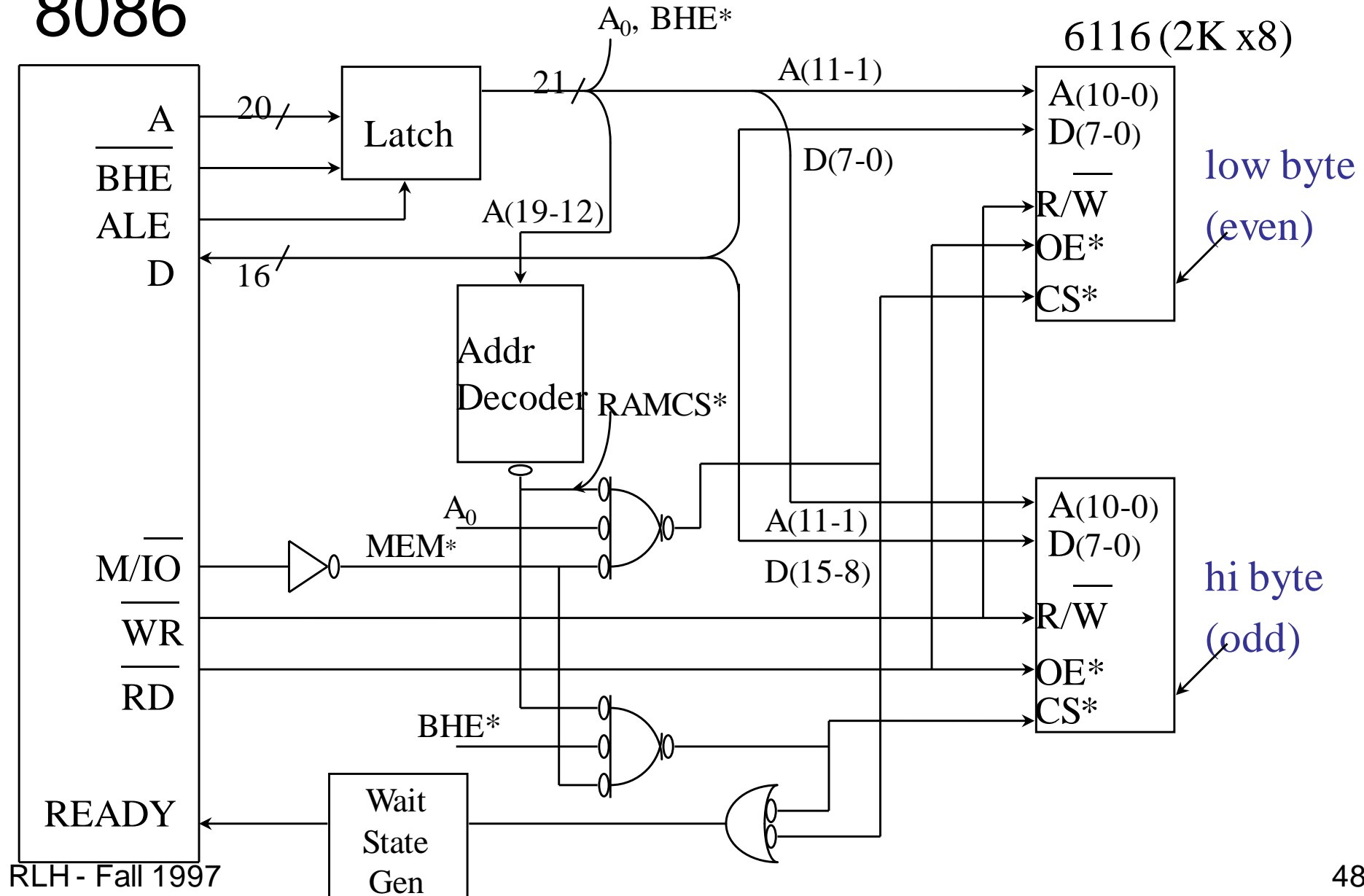
ROM, PROM, EPROM, EEPROM, FLASH

Disk, tape

Magnetic core, magnetic bubble

Interface 8086 to 6116 static RAM

8086



Introduction

to

8086

Assembly Language

Programming

What Is Assembly Language

- Machine-Specific Programming Language
 - one-one correspondence between statements and native machine language
 - matches machine instruction set and architecture
- IBM-PC Assembly Language
 - refers to 8086, 8088, 80186, 80286, 80386, 80486, and Pentium Processors

What Is An Assembler?

- Systems Level Program
 - translates assembly language source code to machine language
 - object file - contains machine instructions, initial data, and information used when loading the program
 - listing file - contains a record of the translation process, line numbers, addresses, generated code and data, and a symbol table

Why Learn Assembly Language?

- Learn how a processor works
- Understand basic computer architecture
- Explore the internal representation of data and instructions
- Gain insight into hardware concepts
- Allows creation of small and efficient programs
- Allows programmers to bypass high-level language restrictions
- Might be necessary to accomplish certain operations

Data Representation

- Binary 0-1
 - represents the state of electronic components used in computer systems
- Bit - Binary digit
- Byte - 8 Bits
 - smallest addressable memory location (on the IBM-PC)
- Word - 16 Bits
 - Each architecture may define its own “wordsize”
- Doubleword - 32 Bits
- Quadword - 64 Bits
- Nybble - 4 Bits

Numbering Systems

- Binary - Base 2
 - 0, 1
- Octal - Base 8
 - 0, 1, 2, ... 7
- Decimal - Base 10
 - 0, 1, 2, ..., 9
- Hexadecimal (Hex)
 - 0, 1, ..., 9, A, B, ..., F
- Raw Binary format
 - All information is coded for internal storage
 - Externally, we may choose to express the information in any numeration system, or in a decoded form using other symbols

Decoding a Byte

- Raw
 - 01010000b
- Hex
 - 50h
- Octal
 - 120₈
- Decimal
 - 80d
- Machine Instruction
 - Push AX
- ASCII Character code
 - 'P'
- Integer
 - 80 (eighty)
- BCD
 - 50 (fifty)
- Custom code ???

Machine Language

- A language of numbers, called the Processor's Instruction Set
 - The set of basic operations a processor can perform
- Each instruction is coded as a number
- Instructions may be one or more bytes
- Every number corresponds to an instruction

Assembly Language vs Machine Language Programming

- Machine Language Programming
 - Writing a list of numbers representing the bytes of machine instructions to be executed and data constants to be used by the program
- Assembly Language Programming
 - Using symbolic instructions to represent the raw data that will form the machine language program and initial data constants

Assembly Language Instructions

- Mnemonics represent Machine Instructions
 - Each mnemonic used represents a single machine instruction
 - The assembler performs the translation
- Some mnemonics require operands
 - Operands provide additional information
 - register, constant, address, or variable
- Assembler Directives

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 - assembly 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

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*

Assembler Language Segment Types

- Stack
 - For dynamic data storage
 - Source file defines size
 - Must have exactly 1
- Data
 - For static data Storage
 - Source file defines size
 - 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 - Listing file, documents “Translation” process
 - » Errors, Addresses, Symbols, etc
- .CRF – Cross reference file

x86 Instruction Set Summary

(Data Transfer)

CBW	;Convert Byte to Word AL → AX
CWD	;Convert Word to Double in AX →DX,AX
IN	;Input
LAHF	;Load AH from 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 Flags
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
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)
MUL	;Multiply (unsigned) in AX(,DX)
IMUL	;Multiply (signed) in AX(,DX)
INC	;Increment

x86 Instruction Set Summary

(Arithmetic/Logical Cont.)

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

Assembler Directives

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

Assembler 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)

Program Example

```
;;;;;;;;;;;;;
;
;   This is an example program.   It prints the
;   character string "Hello World" to the DOS standard output
;   using the DOS service 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 directives to define segment types
;
;
; reserve 256 bytes of stack space
.stack 100h

.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

```


Program Statements

`name operation operand(s) comment`

- Operation is a predefined or reserved word
 - mnemonic - symbolic operation code
 - directive - pseudo-operation code
- Space or tab separates initial fields
- Comments begin with semicolon
- Most assemblers are not case sensitive

Program Data and Storage

- Pseudo-ops to define data or reserve storage
 - DB - byte(s)
 - DW - word(s)
 - DD - doubleword(s)
 - DQ - quadword(s)
 - DT - tenbyte(s)
- These directives require one or more operands
 - define memory contents
 - specify amount of storage to reserve for run-time data

Defining Data

- Numeric data values
 - 100 - decimal
 - 100B - binary
 - 100H - hexadecimal
 - '100' - ASCII
 - "100" - ASCII
- Use the appropriate DEFINE directive (byte, word, etc.)
- A list of values may be used - the following creates 4 consecutive words
`DW 40CH,10B,-13,0`
- A ? represents an uninitialized storage location
`DB 255,?, -128, 'x'`

Naming Storage Locations

- Names can be associated with storage locations

ANum DB -4

DW 17

ONE

UNO DW 1

X DD ?

- These names are called variables

- A**N**um refers to a byte storage location, initialized to FCh
- The next word has no associated name
- O**NE and **U**NO refer to the same word
- X** is an uninitialized doubleword

Arrays

- Any consecutive storage locations of the same size can be called an array

```
X DW 40CH,10B,-13,0
```

```
Y DB 'This is an array'
```

```
Z DD -109236, FFFFFFFFH, -1, 100B
```

- Components of X are at X, X+2, X+4, X+8
- Components of Y are at Y, Y+1, ..., Y+15
- Components of Z are at Z, Z+4, Z+8, Z+12

DUP

- Allows a sequence of storage locations to be defined or reserved
- Only used as an operand of a define directive

```
DB 40 DUP (?)
```

```
DW 10h DUP (0)
```

```
DB 3 dup ("ABC")
```

Word Storage

- Word, doubleword, and quadword data are stored in reverse byte order (in memory)

Directive	Bytes in Storage
DW 256	00 01
DD 1234567H	67 45 23 01
DQ 10	0A 00 00 00 00 00 00 00
X DW 35DAh	DA 35

Low byte of X is at X, high byte of X is at X+1

Named Constants

- Symbolic names associated with storage locations represent addresses
- Named constants are symbols created to represent specific values determined by an expression
- Named constants can be numeric or string
- Some named constants can be redefined
- No storage is allocated for these values

Equal Sign Directive

- `name = expression`
 - expression must be numeric
 - these symbols may be redefined at any time

```
maxint = 7FFFh
```

```
count = 1
```

```
DW count
```

```
count = count * 2
```

```
DW count
```

EQU Directive

- name EQU expression
 - expression can be string or numeric
 - Use < and > to specify a string EQU
 - these symbols cannot be redefined later in the program

sample EQU 7Fh

aString EQU <1.234>

message EQU <This is a message>

Data Transfer Instructions

- *MOV target, source*
 - reg, reg
 - mem, reg
 - reg, mem
 - mem, imm
 - reg, imm
- Sizes of both operands must be the same
- reg can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

Sample MOV Instructions

```
b db 4Fh
w dw 2048
```

```
mov bl,dh
```

```
mov ax,w
```

```
mov ch,b
```

```
mov al,255
```

```
mov w,-100
```

```
mov b,0
```

- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)

- You can assign a size attribute using LABEL

```
LoByte LABEL BYTE
```

```
aWord DW 97F2h
```

Addresses with Displacements

```
b db 4Fh, 20h, 3Ch
```

```
w dw 2048, -100, 0
```

```
mov bx, w+2
```

```
mov b+1, ah
```

```
mov ah, b+5
```

```
mov dx, w-3
```

- Type checking is still in effect

- The assembler computes an address based on the expression

- *NOTE: These are address computations done at assembly time*

MOV ax, b-1

will not subtract 1 from the value stored at b

eXCHanGe

- *XCHG target, source*
 - reg, reg
 - reg, mem
 - mem, reg
- MOV and XCHG cannot perform memory to memory moves
- This provides an efficient means to swap the operands
 - No temporary storage is needed
 - Sorting often requires this type of operation
 - This works only with the general registers

Arithmetic Instructions

ADD *dest, source*

SUB *dest, source*

INC *dest*

DEC *dest*

NEG *dest*

- *Operands must be of the same size*

- *source* can be a general register, memory location, or constant
- *dest* can be a register or memory location
 - except operands cannot both be memory

Program Segment Structure

- Data Segments
 - Storage for variables
 - Variable addresses are computed as offsets from start of this segment
- Code Segment
 - contains executable instructions
- Stack Segment
 - used to set aside storage for the stack
 - Stack addresses are computed as offsets into this segment
- Segment directives
 - `.data`
 - `.code`
 - `.stack size`

Memory Models

- `.Model memory_model`
 - tiny: code+data \leq 64K (.com program)
 - small: code \leq 64K, data \leq 64K, one of each
 - medium: data \leq 64K, one data segment
 - compact: code \leq 64K, one code segment
 - large: multiple code and data segments
 - huge: allows individual arrays to exceed 64K
 - flat: no segments, 32-bit addresses, protected mode only (80386 and higher)

Program Skeleton

```
.model small
.stack 100H
.data
    ;declarations
.code
main proc
    ;code
main endp
    ;other procs
end main
```

- Select a memory model
- Define the stack size
- Declare variables
- Write code
 - organize into procedures
- Mark the end of the source file
 - optionally, define the entry point