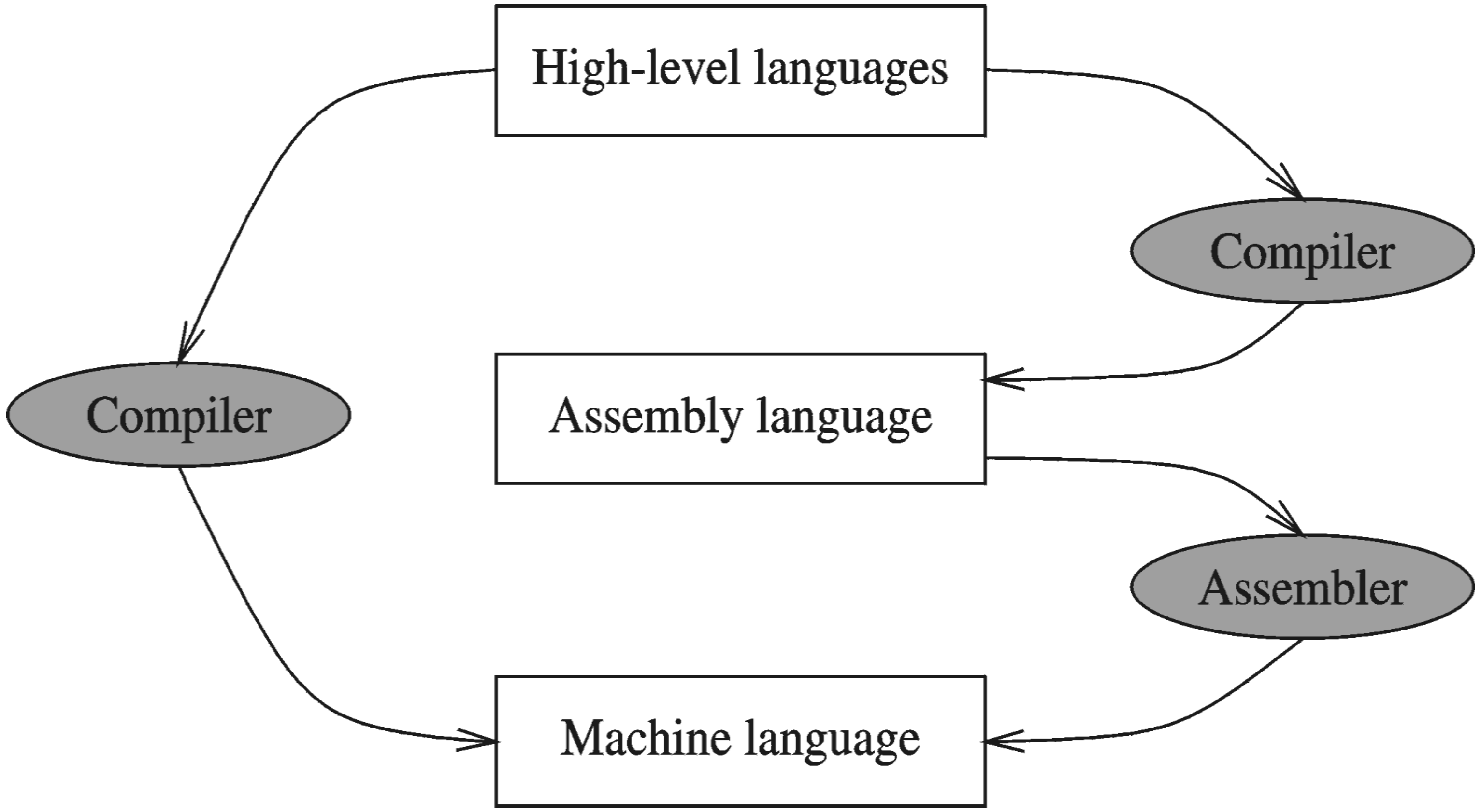
UNIVERSITY OF ASIA PACIFIC

Microprocessor and Assembly Language Lab CSE 312

Dr. Shah Murtaza Rashid Al Masud
Associate Prof.
Dept. of CSE, UAP

Programming Languages

1) Machine Language (Hardware specific low level language)
2) **Assembly Language (Medium to Low Level Language)**
3) High Level Languages



Programming Languages

Machine language is a **language** that has a binary form.

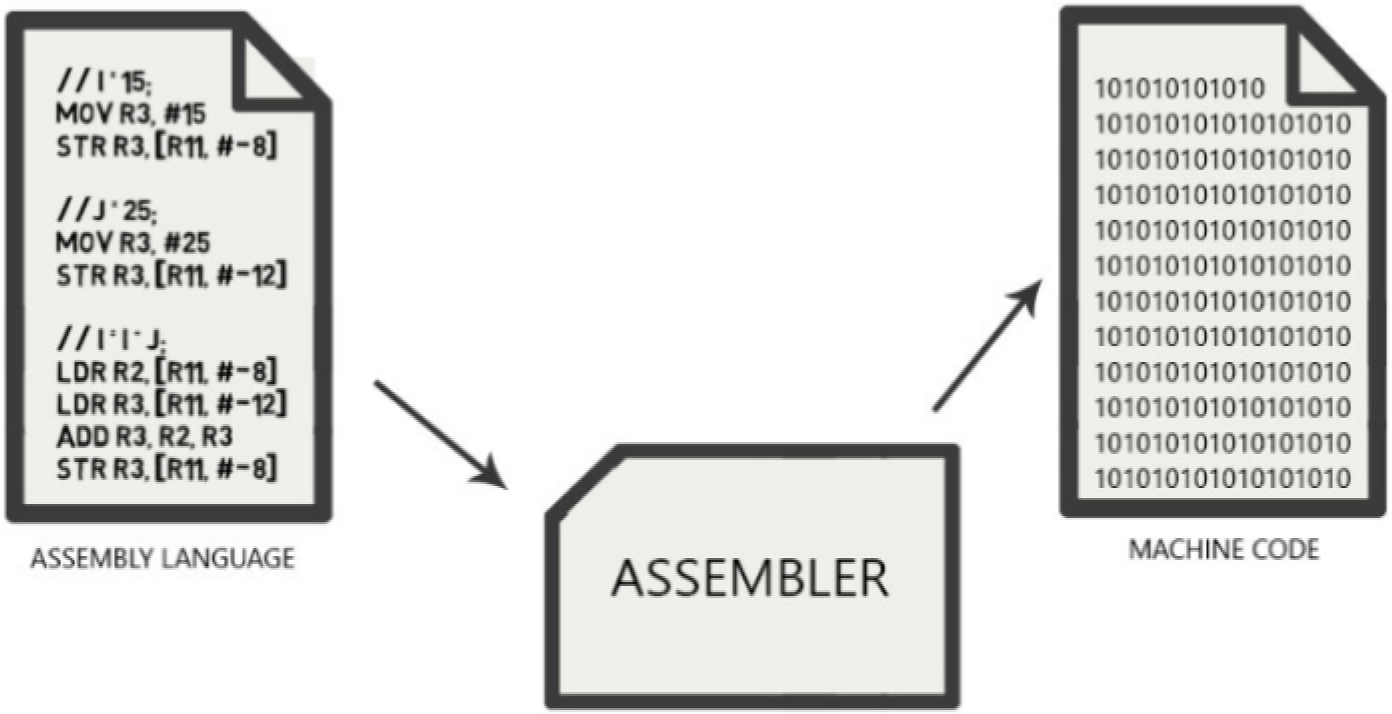
It can be **directly** executed by a computer.

While an **assembly language** is a low-level **programming language** that requires software called an **assembler** (TASM, MASM, EMU8086) to convert it into **machine code**.

Programming Languages

Machine language is the binary language that is easily understood by computers.

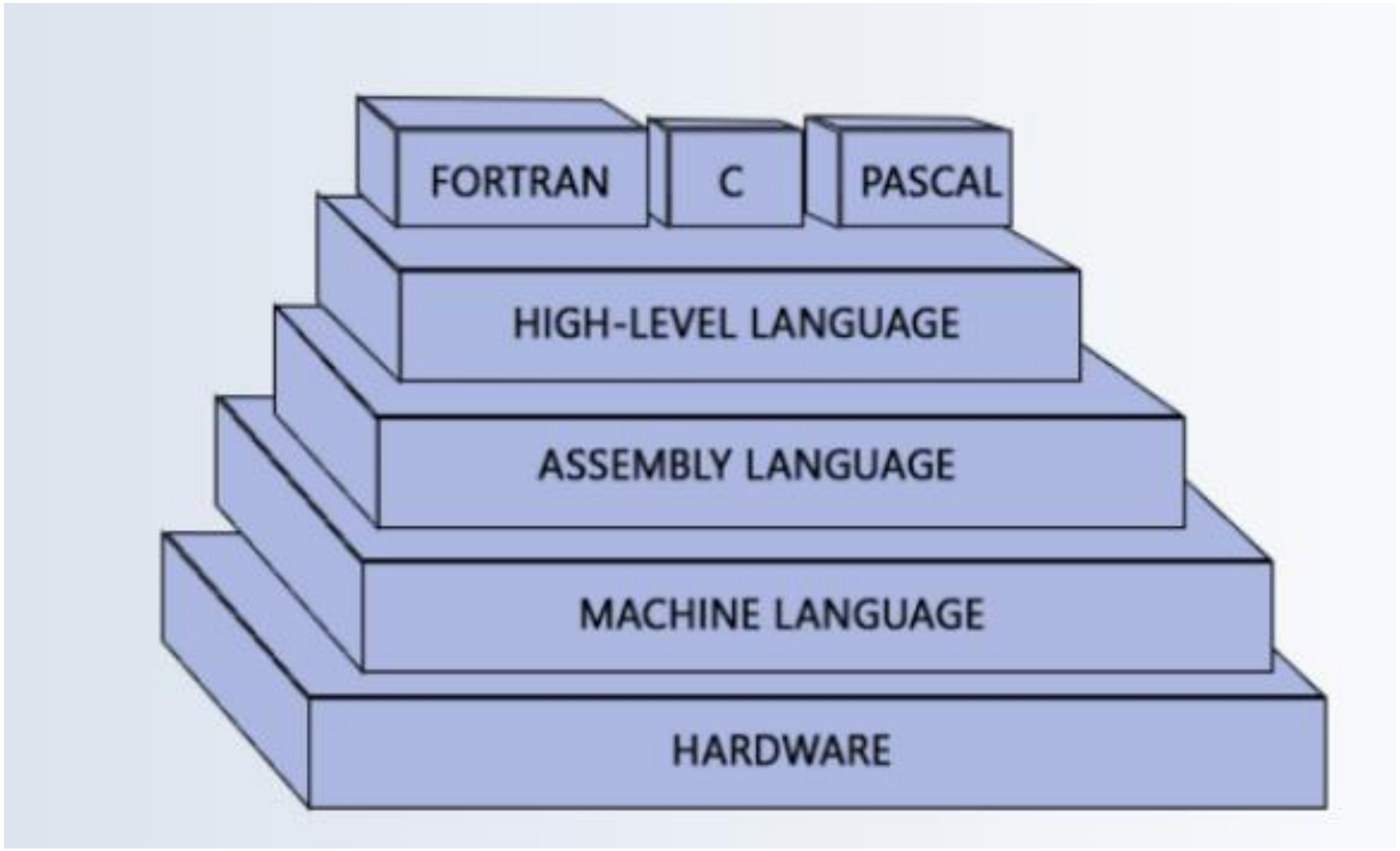
Hence it can be directly executed by CPU with absolutely no need of compilers and interpreters.



```
// I · 15;
MOV R3, #15
STR R3, [R11, #-8]

// J · 25;
MOV R3, #25
STR R3, [R11, #-12]

// I · I · J;
LDR R2, [R11, #-8]
LDR R3, [R11, #-12]
ADD R3, R2, R3
STR R3, [R11, #-8]
```

```
101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
1010101010101010
```

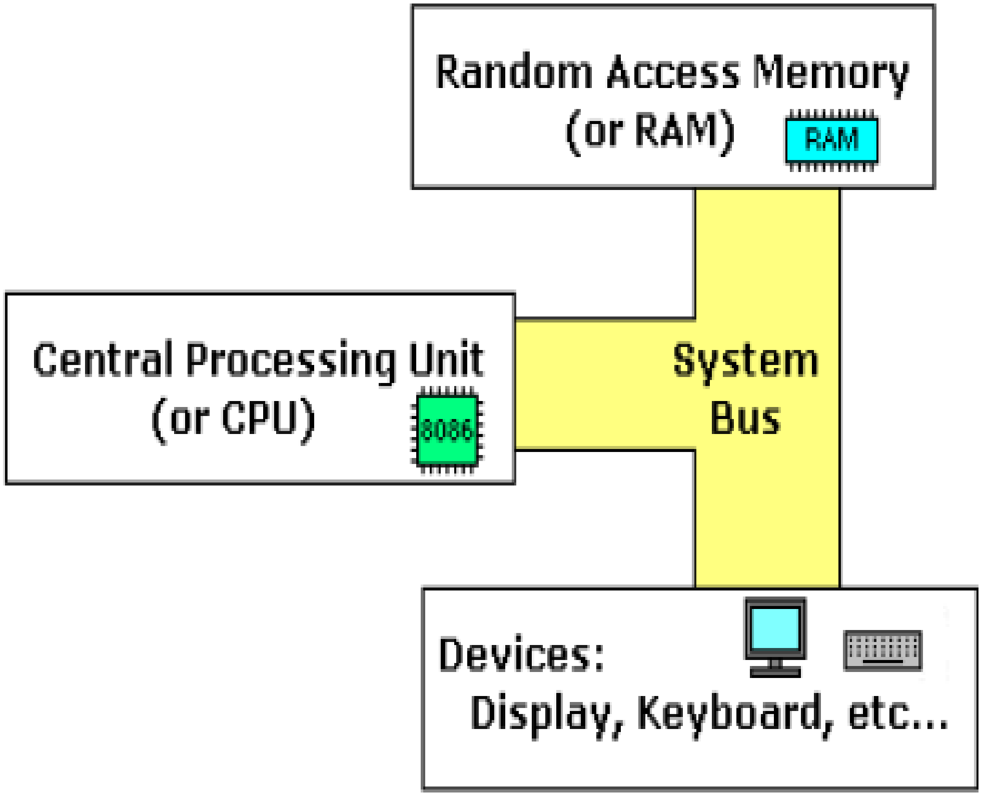
Programming Languages



Comparison between Assembly Language vs Machine Language.

Assembly Language	Machine Language
Assembly language is an intermediate programming language between a high-level programming language and Machine language	Machine language is a low-level language.
Assembly language is English syntaxes, which is understood by the CPU after converting it to low-level language by interpreter and compilers.	Machine language is in the form of 0's and1's (binary format). One showcases the true/on state while zero depicts the false/off state.
Programmers can understand the assembly language, however, CPU cannot.	CPU can directly understand Machine language. No need of compiler or assembler.
Assembly language is a set of instructions which are the same irrespective of platform.	Machine code differs platform to platform.
The codes and instructions of assembly language can be memorized.	Binary codes here can't be memorized.
Modification is not that tough here.	Modification is not possible. It has to be written from scratch for a specific type of CPU.
Here applications are device drivers, low-level embedded systems, and real-time systems	CDs, DVDs and Blu-ray Discs represent an application of binary form.

Simple Computer Model

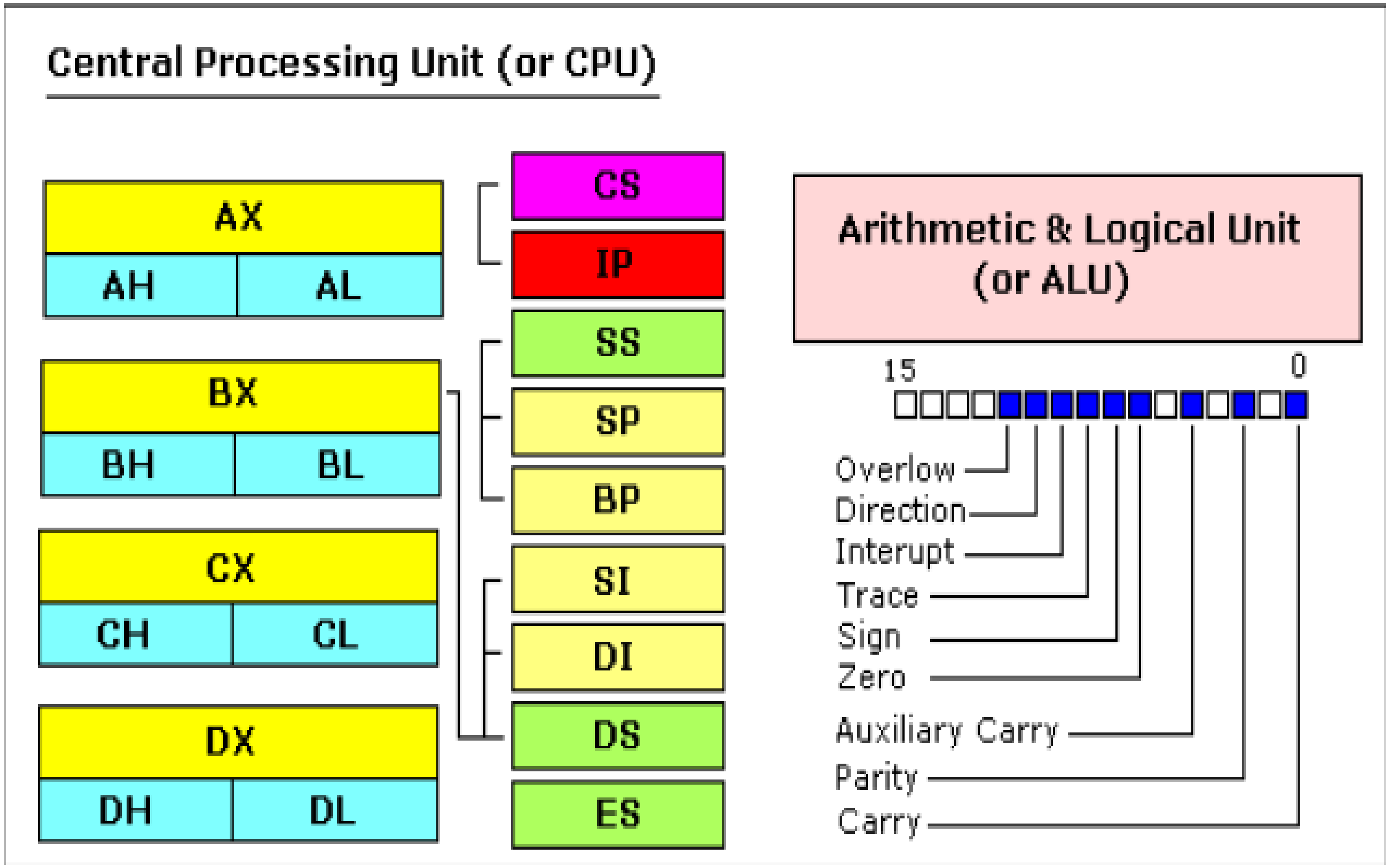


16 bit MP 8086. The **system bus (3 types data bus, address bus, control bus)** (shown in yellow) connects the various components of a computer.

The **CPU** is the heart of the computer, most of computations occur inside the **CPU**.

RAM is a place to where the programs are loaded in order to be executed.

Inside the CPU



CPU Registers

- General purpose registers
- **Segment registers**
- Instruction Pointer
- Flags register
- Special purpose registers
    - Debug registers
    - Machine Control registers

CPU General Purpose Registers

8086 CPU has 8 general purpose registers, each register has its own name:

- **AX** - the accumulator register (divided into **AH / AL**).
- **BX** - the base address register (divided into **BH / BL**).
- **CX** - the count register (divided into **CH / CL**).
- **DX** - the data register (divided into **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

[**N.B.** The size of the above registers is 16 bit]

General Purpose Registers

- AX (accumulator register):
  - Stores operands for arithmetic & data transfer instructions.

- BX Register (base register):
  - Holds the starting base address of a memory within a data segment.

- CX Register (counter register):
  - Primarily used in loop instruction to store loop counter.

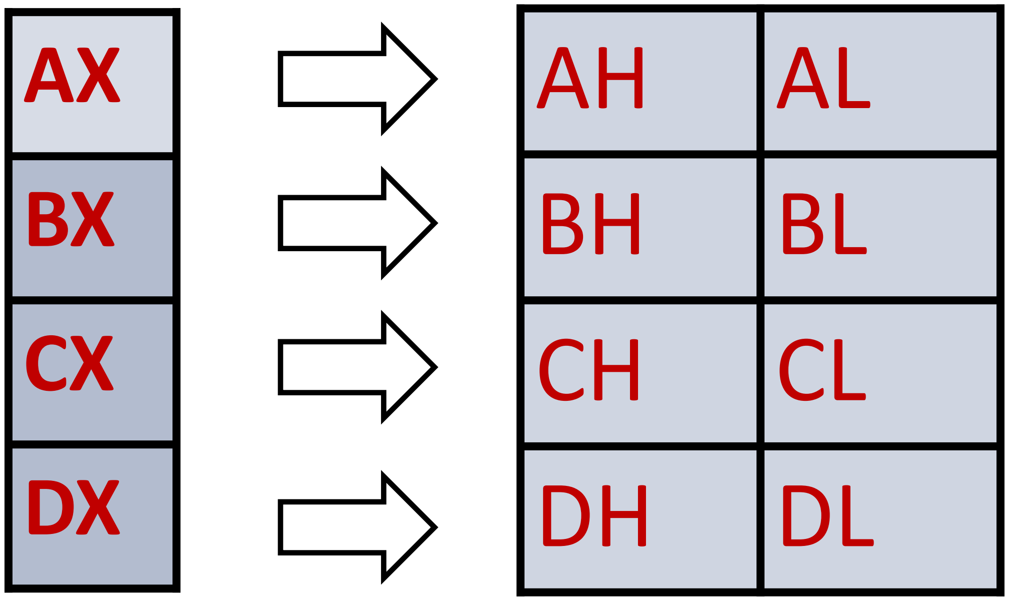
- DX Register (data register)
  - Used to contain I/O port address for I/O instruction.

General Purpose Registers

In computer programming, an **operand** is a term used to **describe** any object that is capable of being manipulated. For **example**, in "1 + 2" the "1" and "2" are the **operands** and the plus symbol is the **operator**.

General Purpose Registers (contd.)

- Each of these 16-bit registers are further subdivided into two 8-bit registers.



General Purpose Registers (contd.)

4 general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers. For example,

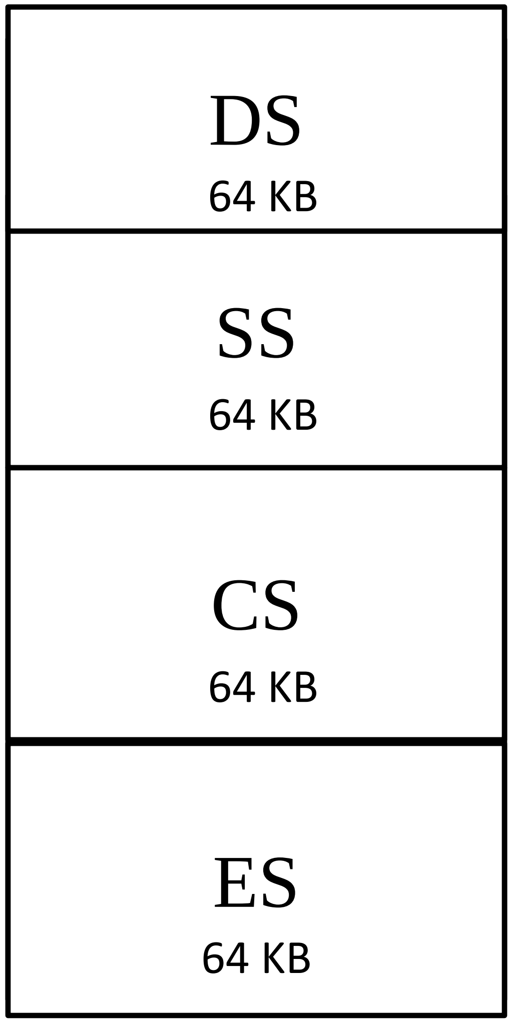
if AX = **00110000 00111001b (bit)**,
then AH = **00110000b** and AL = **00111001b**.

Therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa.
The same is for other 3 registers, "H" is for high and "L" is for low part.

Memory Segmentation (Segment & Special Purpose Registers)

- **Data Segment**
  - contains variable definitions
  - declared by .DATA
- **Stack segment**
  - used to store the stack
  - declared by .STACK size
  - default stack size is 1KB.
- **Code segment**
  - contains program's instructions
  - declared by .CODE

Extra segment



8086 Memory

Memory Segmentation (Segment & Special Purpose Registers)

- **CS** - points at the segment containing the current program.
- **DS** - generally points at segment where variables are defined.
- **ES** - extra segment register, it's up to a coder to define its usage.
- **SS** - points at the segment containing the stack.

- **Special Purpose Registers**
- **IP** - the instruction pointer.
- **Flags Register** - determines the current state of the processor.

1

53+

38

91

Flag Register

- **Flag register determines the current status of the microprocessor.**
- It is modified automatically by CPU after mathematical operations.
- This allows to determine the type of the result.

- 8086 has 16-bit Flag register.
- There are two kinds of flags: Status Flags and Control Flags

Flag Register (contd.)

Status Flags	Control Flags
Carry Flag (CF)	Trap Flag
Auxiliary-carry Flag (AF)	Interrupt Flag
Zero Flag (ZF)	Directional Flag
Sign Flag (SF)	
Parity Flag (PF)	
Overflow Flag (OF)	

Flag Register (contd.)

Flag	Purpose
Carry (CF)	CF = 1 if there is a carry out from the MSB on addition or there is a Borrow into the MSB on subtraction.
Parity (PF)	PF=1 if the low byte of a result has an even number of one bits (even parity).
Auxiliary (AF)	Holds the carry (half–carry) after addition or borrow after subtraction between bit positions 3 and 4 of the result (for example, in BCD addition or subtraction.)
Zero (ZF)	Shows the result of the arithmetic or logic operation. Z=1; result is zero. Z=0; The result is 0
Sign (SF)	Holds the sign of the result after an arithmetic/logic instruction execution. SF=1 if the MSB of a result is 1.

Flag Register (contd.)

Flag	Purpose
Trap (TF)	Enables the trapping through an on-chip debugging feature.

Interrupt (IF)	Controls the operation of the INTR (interrupt request)
	IF = 0 if the INTR pin is disabled
	IF = 1, if the INTR pin isenabled.

Direction (DF)	Selects either the increment or decrement mode for DI and /or SI registers during the string instructions.
Overflow (OF)	Overflow occurs when signed numbers are added or subtracted. An overflow indicates the result has exceeded the capacity of the machine.

Pointer and Index Registers

- **Stack Pointer (SP)**
  - A 16-bit register pointing to program stack
- **Base Pointer (BP)**
  - A 16-bit register used to access data in stack segment
  - Usually used for based indexed or register indirect addressing
- **Source Index (SI) & Destination Index (DI)**
  - 16-bit registers
  - Used to point to memory locations in the data segment

Variables

- **Variable** is a memory location. which contains some known or unknown quantity of information referred to as a **value** or in easy terms, a **variable** is a container for different types of data (like integer, float, …..
- It is much easier to have some value be kept in a variable named "**var1**" then at the address 5A73:235B, especially when you have 10 or more variables.
  - DB - stays for Define byte(s).
  - DW - stays for Define word(s).
  - DD - stays for Define double word(s).
  - DQ - stays for Define quad word(s).
- Syntax for a variable declaration:-

  name **DB** *value*

  name **DW** *value*

Defining Data

Numeric data values

- 100 - decimal
- 100B - binary
- 100H - hexadecimal
- '100' - ASCII
- "100" - ASCII

Data Transfer Instructions

- MOV *target*, *source*
  - reg, reg
  - mem, reg
  - reg, mem
  - mem, immed
  - reg, immed

- Sizes of both operands must be the same.
- reg can be any non-segment register except IP cannot be the target register.

Memory Models

.Model *memory_model*

- tiny: code+data <= 64K (.com program)
- small: code<=64K, data<=64K, one of each
- medium: data<=64K, one data segment
- compact: code<=64K, one code segment
- large: multiple code and data segments
- huge: allows individual arrays to exceed 64K

Assembly Language Program Structure

```
.MODEL….        // selects a standard memory model for the programs.
.STACK 100H   //sets the size of the program stack which may be any size up to 64kb
.DATA              // all variables pertaining to the program are defined in the area.
.CODE             // identifies the part of the program that contains instructions.
MAIN PROC  //creates a name and a address for the beginning of a procedure.
…………….         define main procedures.
…………….
MAIN ENDP   // indicates the end of the procedure.
END MAIN    // terminates assembly  of the program.
```

Example (Print Hello World!)

```
.MODEL SMALL
.STACK 100H
.DATA
message db “Hello World! $”             ; Message to be displayed
.CODE
MAIN  PROC
      mov    ax,@data
      mov    ds,ax

      mov    ah,9h                      ; function to display a string
      mov    dx,offset message          ; offset of Message string
      int    21h                ; Dos Interrupt (initiate the process)

      mov    ax,4C00h                   ; function to terminate
      int    21h                ; Dos Interrupt
MAIN  ENDP
END   MAIN
```

What is 8086 emulator

emu8086 is an emulator of Intel 8086 (AMD compatible) microprocessor with integrated 8086 assembler and tutorials for beginners. emulator runs programs like the real microprocessor in step-by-step mode. it shows registers, memory, stack, variables and flags.

Running The Emulator (emu8086)

8086 Microprocessor Emulator, also known as EMU8086, is an emulator of the program 8086 microprocessor.

It is developed with a built-in 8086 assembler.

This application is able to run programs on both PC desktops and laptops.

This tool is primarily designed to copy or emulate hardware. These include the memory of a program, CPU, RAM, input and output devices, and even the display screen.

Running The Emulator (emu8086)

1.1. Download and install emu8086. It is usually installed in C:\EMU8086 subfolder in the “Windows” directory

2.2. Run emu8086 icon (on the desktop or in the c:\EMU8086 folder of window) It has green color

3.3. If it requests for “Registration key, just ignore it by closing that window You will be left with the emulator (emu8086) IDE

4.4. Copy and paste an Assembly Language program like the “Hello World” program and paste it on editor of emulator

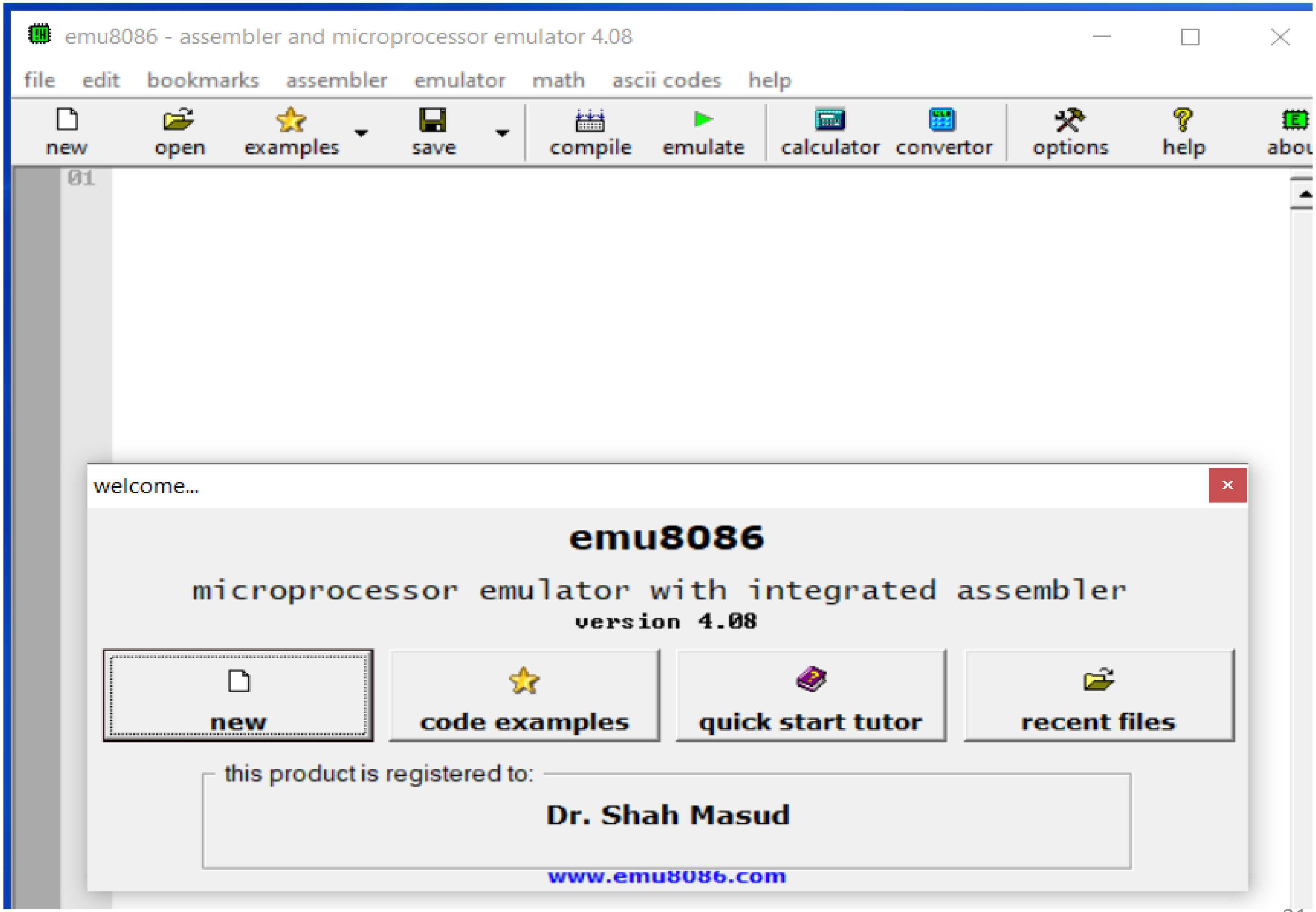
5.5. Compile

6.6. Run (once there is no syntax error)

7.7. Click OK to see/view the output of your program on the Emulator screen.

8.8. After running the program, another menu screen will be displayed, where you have the option to “View” symbol table, variables, listing (containing the object code and source code), emulator screen, etc

Running The Emulator (emu8086)


emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help abou
01
welcome...
new
code examples
quick start tutor
recent files

emu8086

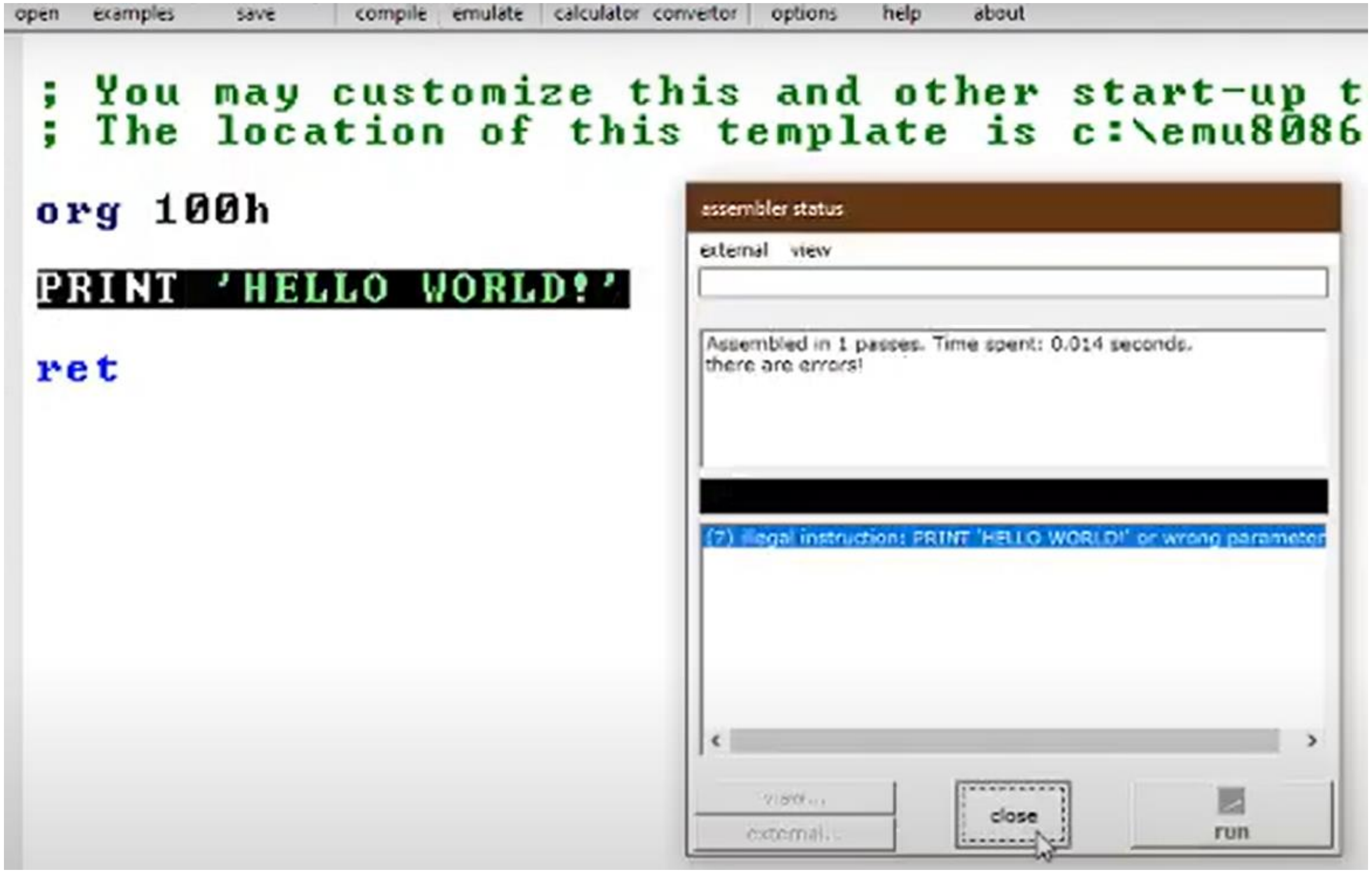
microprocessor emulator with integrated assembler
version 4.08

this product is registered to:

Dr. Shah Masud

www.emu8086.com

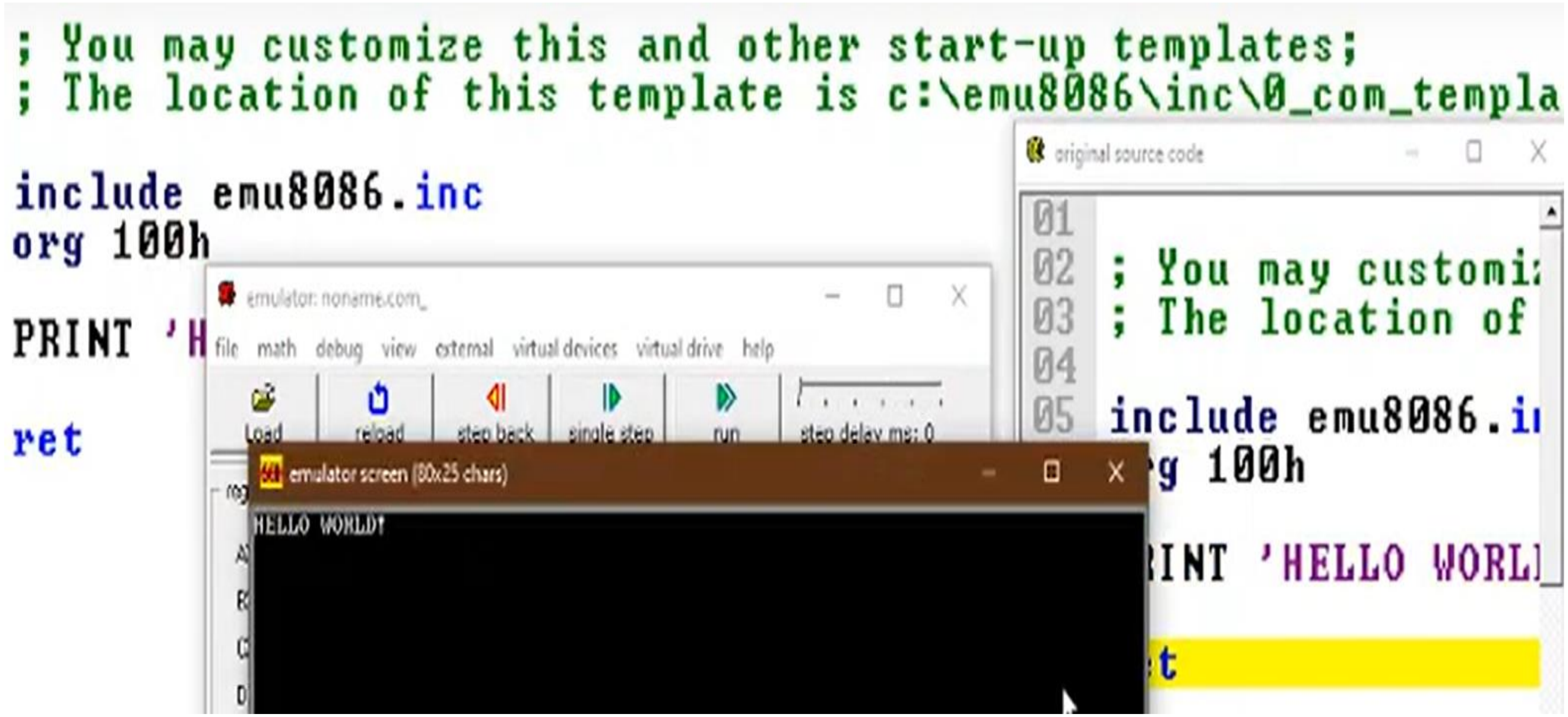
Running The Emulator (emu8086)



```
; You may customize this and other start-up t
; The location of this template is c:\emu8086

org 100h

PRINT 'HELLO WORLD!'

ret
```

```
Assembled in 1 passes. Time spent: 0.014 seconds.
there are errors!

(7) illegal instruction: PRINT 'HELLO WORLD!' or wrong parameter
```

Running The Emulator (emu8086)



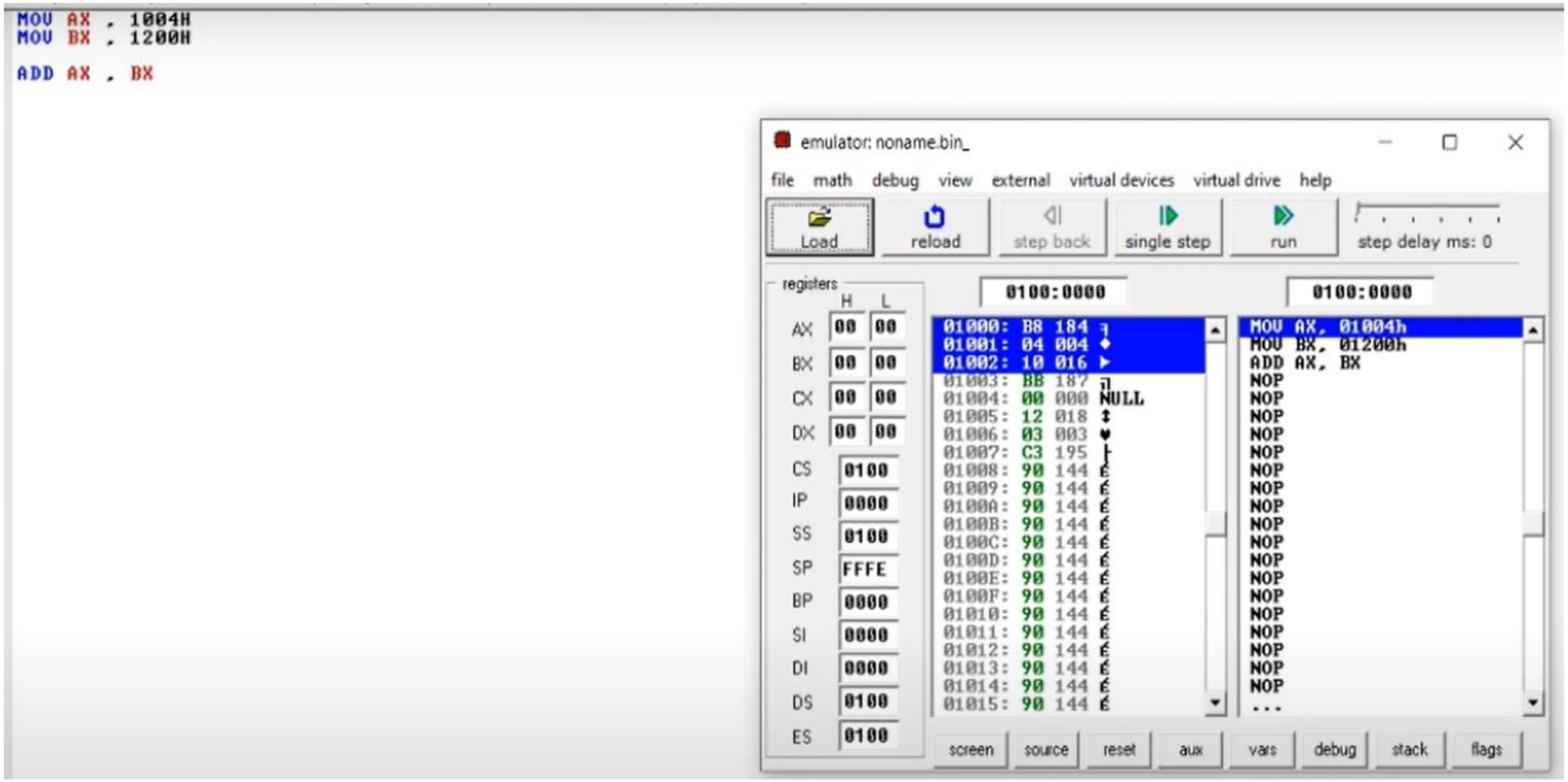
```
; You may customize this and other start-up templates;
; The location of this template is c:\emu8086\inc\0_com_templa

include emu8086.inc
org 100h

PRINT 'H

ret
```

emulator: noname.com_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
emulator screen (80x25 chars)
original source code

```
HELLO WORLD!
```

```
; You may customi
; The location of

include emu8086.i
rg 100h

INT 'HELLO WORL

t
```

Running The Emulator (emu8086)



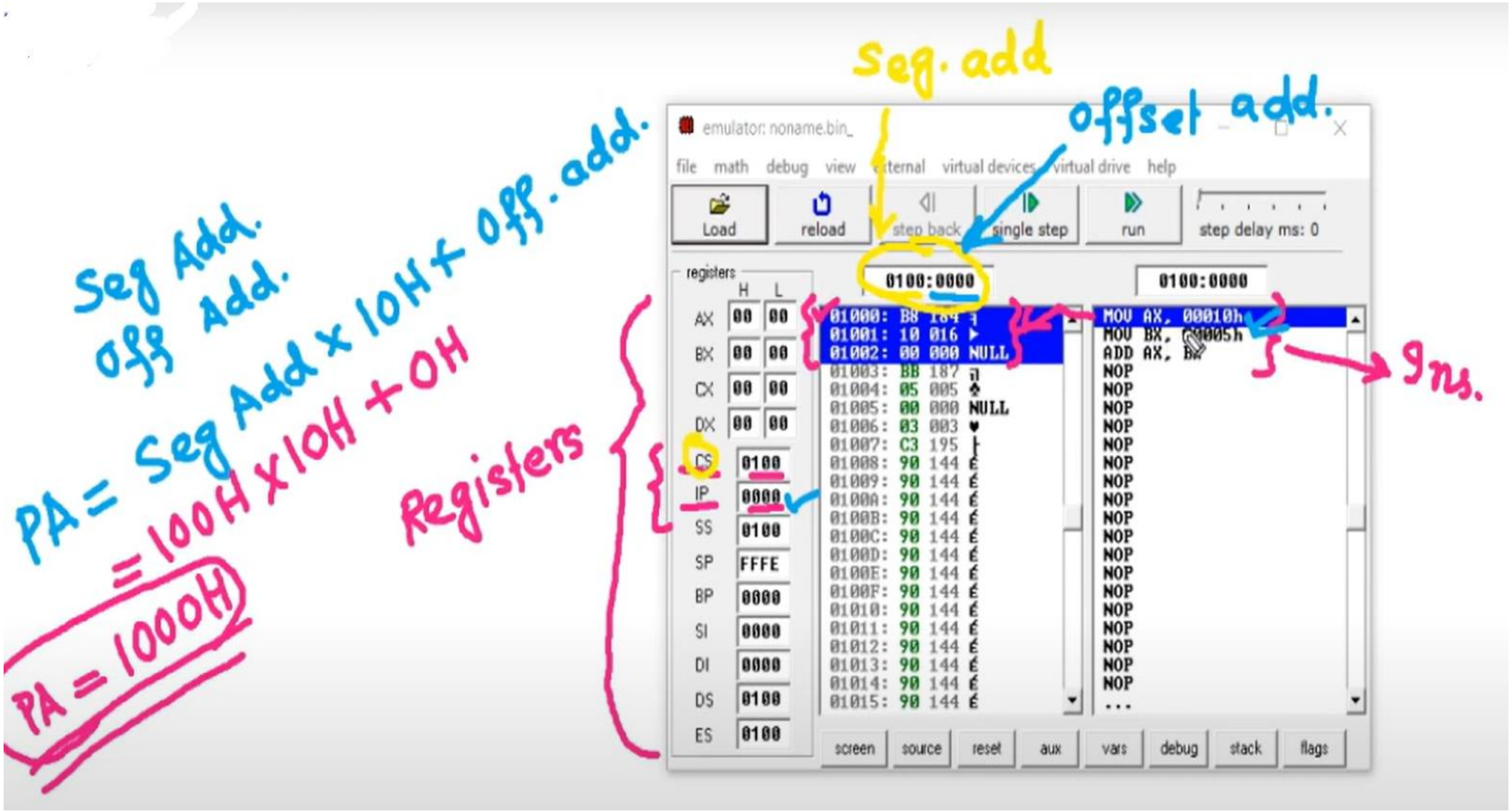
```
MOV AX , 1004H
MOV BX , 1200H

ADD AX , BX
```

```
   H  L
AX 00 00
BX 00 00
CX 00 00
DX 00 00
CS 0100
IP 0000
SS 0100
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0100
ES 0100
```

```
0100:0000
01000: B8 184 ╕
01001: 04 004 ♦
01002: 10 016 ►
01003: BB 187 ╗
01004: 00 000 NULL
01005: 12 018 ↕
01006: 03 003 ♥
01007: C3 195 ├
01008: 90 144 É
01009: 90 144 É
0100A: 90 144 É
0100B: 90 144 É
0100C: 90 144 É
0100D: 90 144 É
0100E: 90 144 É
0100F: 90 144 É
01010: 90 144 É
01011: 90 144 É
01012: 90 144 É
01013: 90 144 É
01014: 90 144 É
01015: 90 144 É
```

```
0100:0000
MOV AX, 01004h
MOV BX, 01200h
ADD AX, BX
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
...
```

Running The Emulator (emu8086)



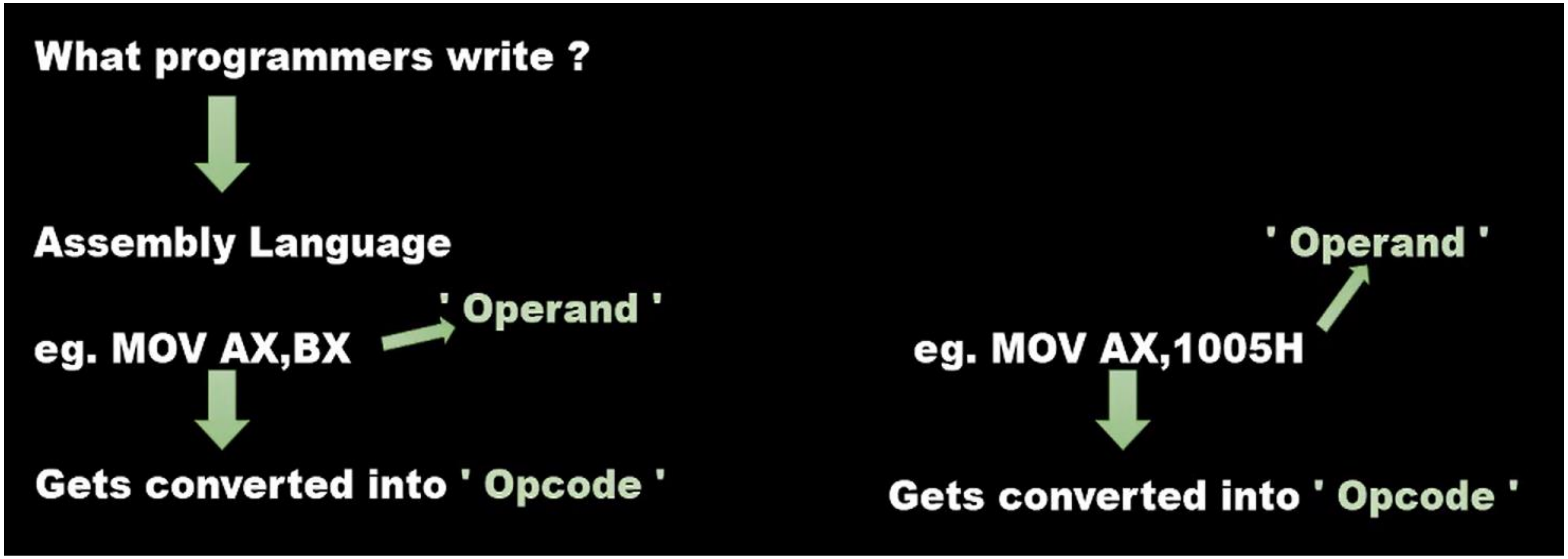
```
registers
     H   L
AX  00  00
BX  00  00
CX  00  00
DX  00  00
CS  0100
IP  0000
SS  0100
SP  FFFE
BP  0000
SI  0000
DI  0000
DS  0100
ES  0100
```

```
0100:0000
01000: B8 184 ╕
01001: 10 016 ►
01002: 00 000 NULL
01003: BB 187 ╗
01004: 05 005 ♣
01005: 00 000 NULL
01006: 03 003 ♥
01007: C3 195 ├
01008: 90 144 É
01009: 90 144 É
0100A: 90 144 É
0100B: 90 144 É
0100C: 90 144 É
0100D: 90 144 É
0100E: 90 144 É
0100F: 90 144 É
01010: 90 144 É
01011: 90 144 É
01012: 90 144 É
01013: 90 144 É
01014: 90 144 É
01015: 90 144 É
```

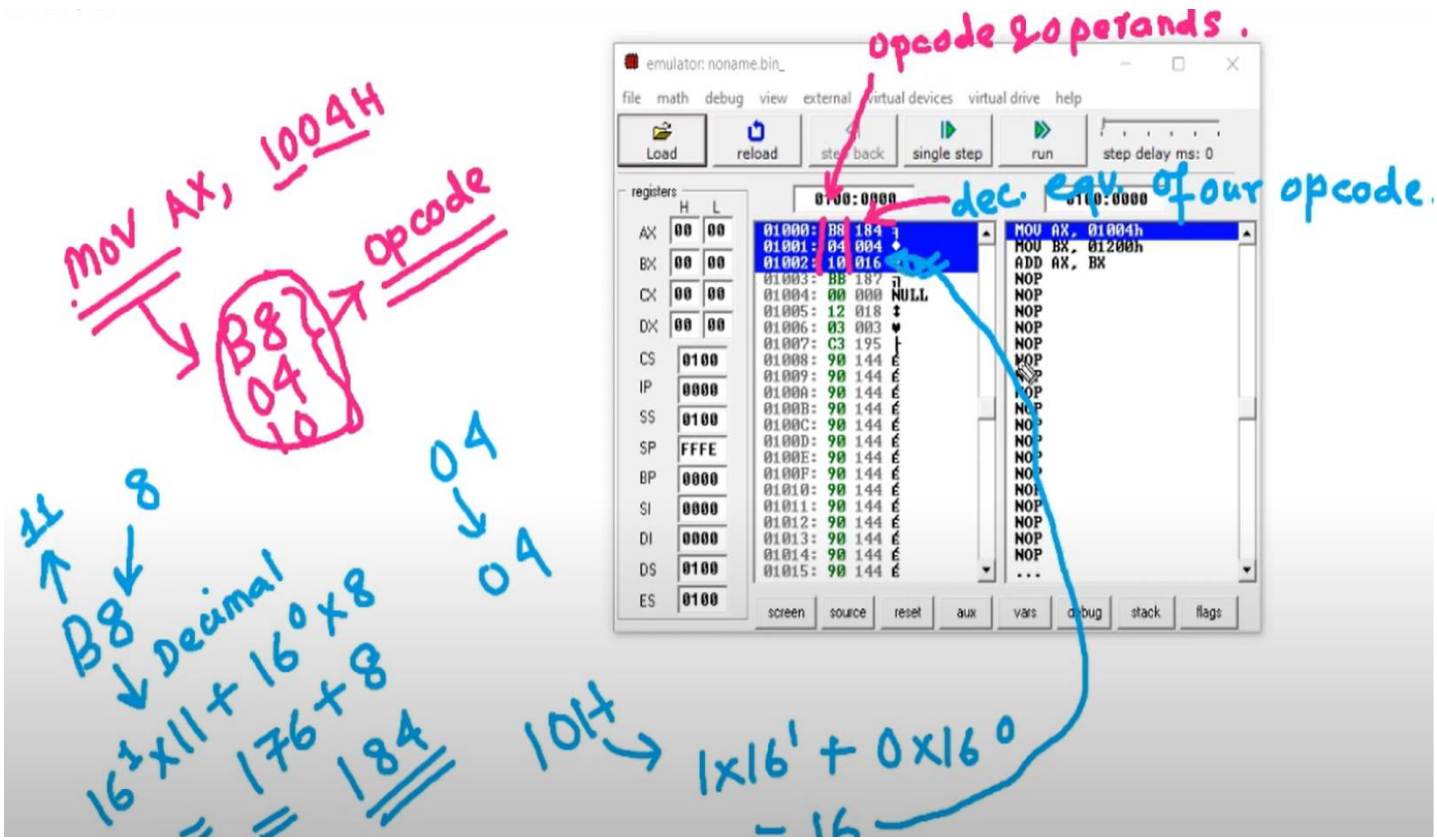
```
0100:0000
MOV AX, 00010h
MOV BX, 00005h
ADD AX, BX
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
...
```

$$PA = \text{Seg Add} \times 10H + \text{off. add.}$$
$$= 100H \times 10H + 0H$$
$$\boxed{PA = 1000H}$$

Running The Emulator (emu8086)



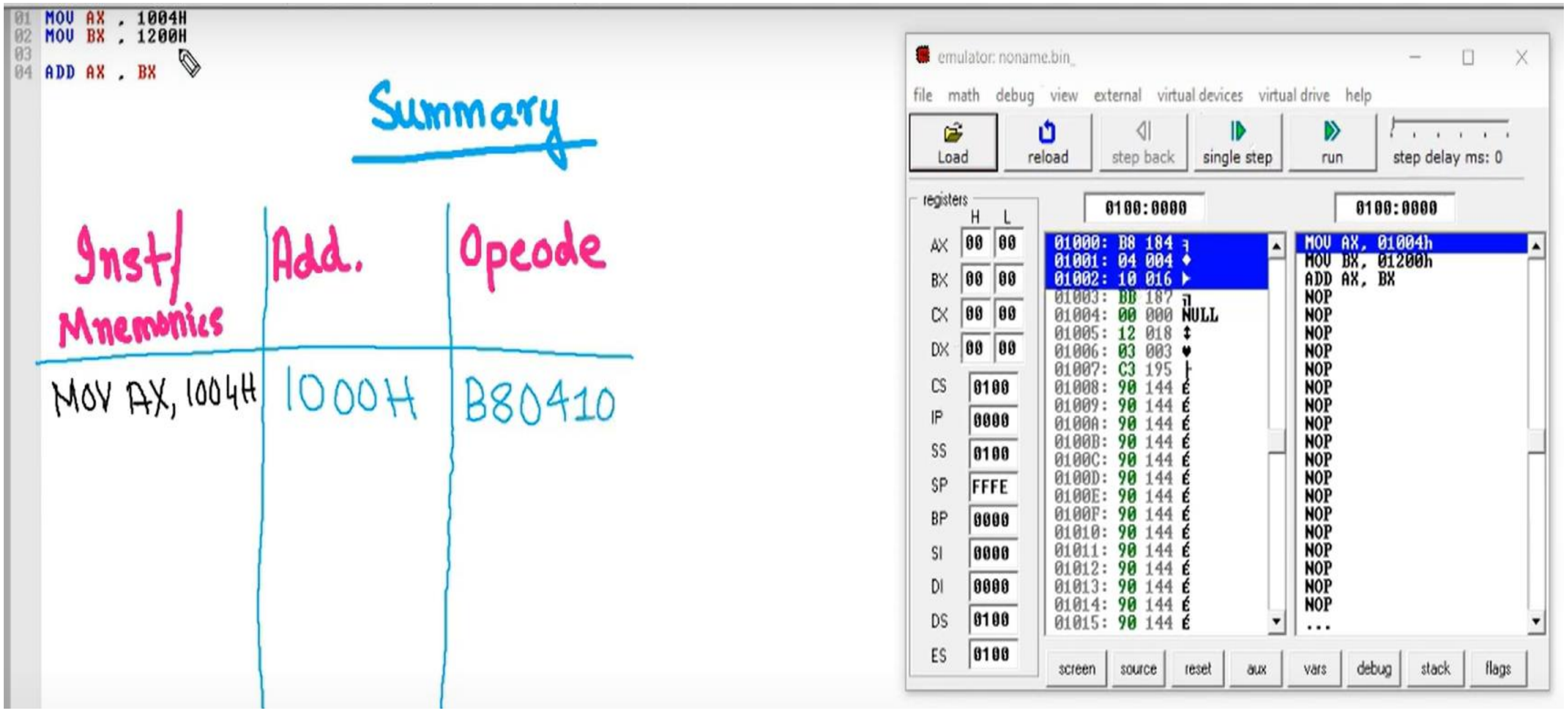
Running The Emulator (emu8086)



```
     H   L
AX   00  00
BX   00  00
CX   00  00
DX   00  00
CS   0100
IP   0000
SS   0100
SP   FFFE
BP   0000
SI   0000
DI   0000
DS   0100
ES   0100
```

```
0100:0000                      0100:0000
01000: B8 184 ╕                MOV AX, 01004h
01001: 04 004 ♦                MOV BX, 01200h
01002: 10 016 ►                ADD AX, BX
01003: BB 187 ╗                NOP
01004: 00 000 NULL             NOP
01005: 12 018 ↕                NOP
01006: 03 003 ♥                NOP
01007: C3 195 ├                NOP
01008: 90 144 É                NOP
01009: 90 144 É                NOP
0100A: 90 144 É                NOP
0100B: 90 144 É                NOP
0100C: 90 144 É                NOP
0100D: 90 144 É                NOP
0100E: 90 144 É                NOP
0100F: 90 144 É                NOP
01010: 90 144 É                NOP
01011: 90 144 É                NOP
01012: 90 144 É                NOP
01013: 90 144 É                NOP
01014: 90 144 É                NOP
01015: 90 144 É                ...
```

Running The Emulator (emu8086)



```
MOV AX , 1004H
MOV BX , 1200H

ADD AX , BX
```

Summary

Inst/ Mnemonics	Add.	Opcode
MOV AX, 1004H	1000H	B80410

registers

	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00

CS	0100
IP	0000
SS	0100
SP	FFFE
BP	0000
SI	0000
DI	0000
DS	0100
ES	0100

```
0100:0000
01000: B8 184 ╕
01001: 04 004 ♦
01002: 10 016 ►
01003: BB 187 ╗
01004: 00 000 NULL
01005: 12 018 ↕
01006: 03 003 ♥
01007: C3 195 ├
01008: 90 144 É
01009: 90 144 É
0100A: 90 144 É
0100B: 90 144 É
0100C: 90 144 É
0100D: 90 144 É
0100E: 90 144 É
0100F: 90 144 É
01010: 90 144 É
01011: 90 144 É
01012: 90 144 É
01013: 90 144 É
01014: 90 144 É
01015: 90 144 É
```

```
0100:0000
MOV AX, 01004h
MOV BX, 01200h
ADD AX, BX
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
...
```

Any Questions?

Thank You