Chapter 5
8086 Assembly Language Programming

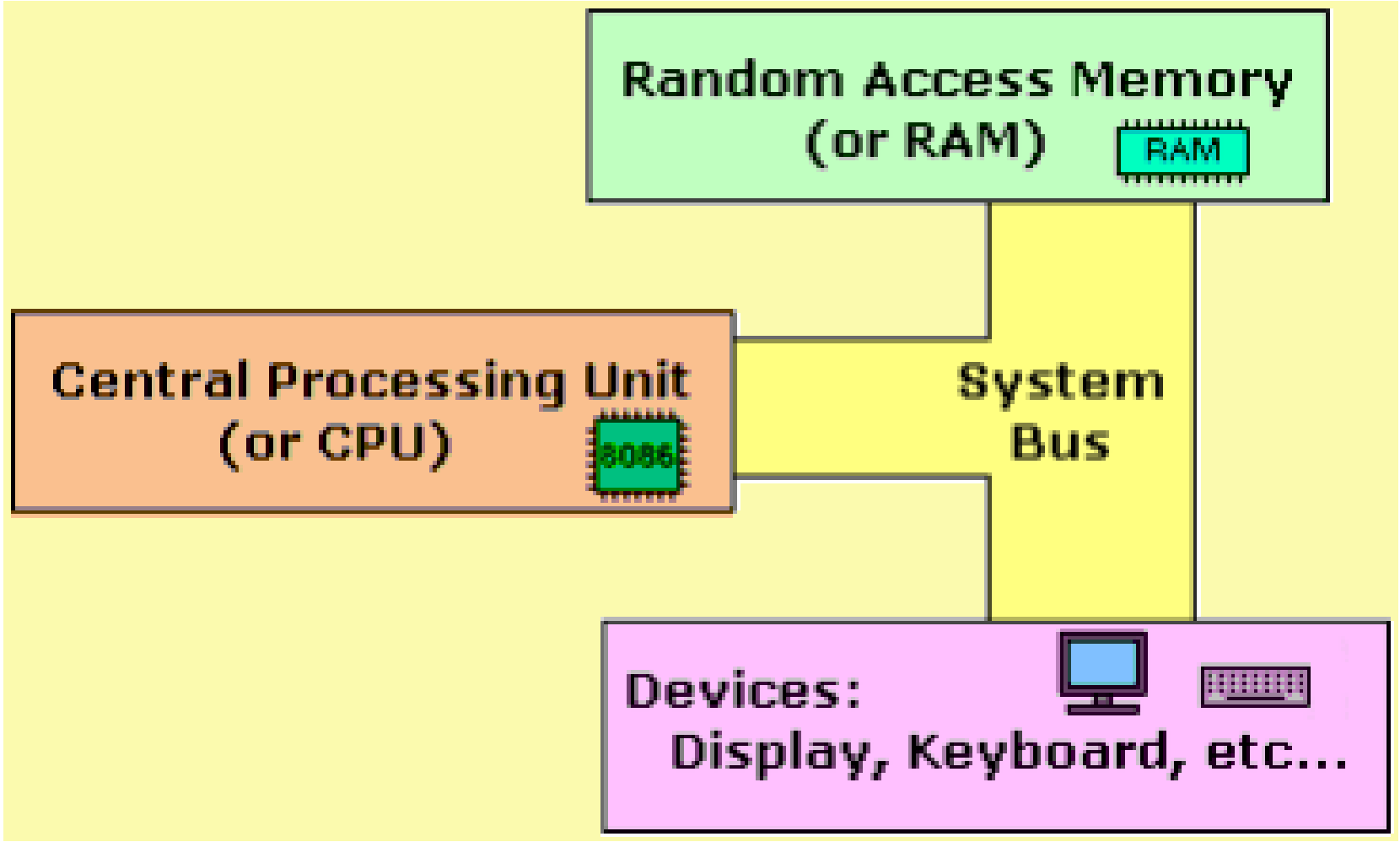
Introduction

- Assembly level programming is very important to low-level embedded system design is used to access the processor instructions to manipulate hardware.
- It is a most primitive machine level language is used to make efficient code that consumes less number of clock cycles and takes less memory as compared to the high-level programming language.
- It is a complete hardware oriented programing language to write a program the programmer must be aware of embedded hardware.

Assembly Level Programming 8086

- The assembly programming language is a low-level language which is developed by using mnemonics.
- The microcontroller or microprocessor can understand only the binary language like 0's or 1's therefore the assembler convert the assembly language to binary language and store it the memory to perform the tasks.
- Before writing the program the embedded designers must have sufficient knowledge on particular hardware of the controller or processor, so first we required to know hardware of 8086 processor.

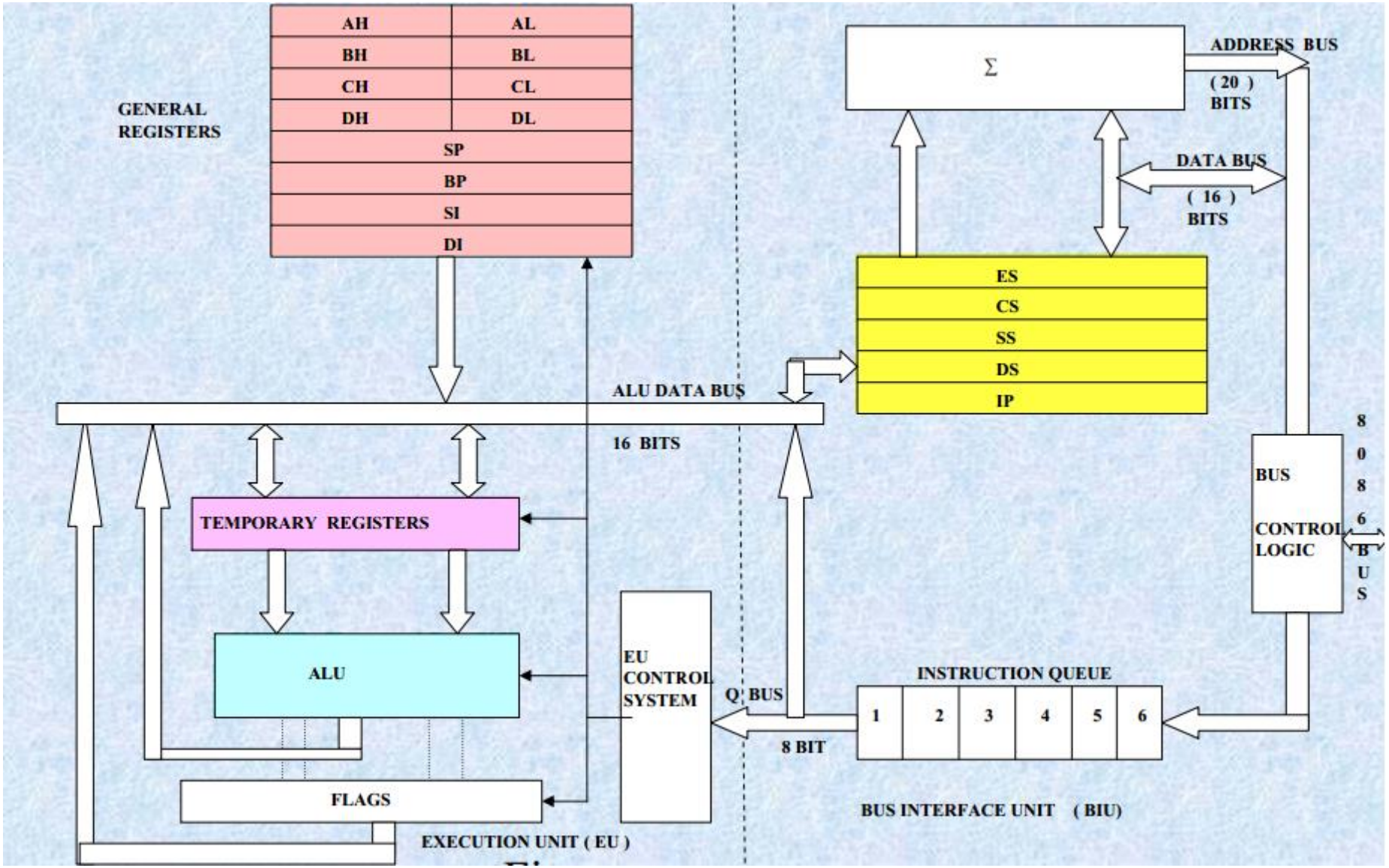
Hardware of the processor



8086 Processor Architecture

- The 8086 is a processor that is represented for all peripheral devices such as serial bus , and RAM and ROM, I/O devices and so on which are all externally connected to CPU by using a system bus.
- The 8086 microprocessor has CISC based architecture, and it has peripherals like 32 I/O, Serial communication, memories and counters/timers.
- The microprocessor requires a program to perform the operations that require a memory for read and save the functions.

8086 Processor Architecture



- The assembly level programming 8086 is based on the memory registers.
- A Register is the main part of the microprocessors and controllers which are located in the memory that provides a faster way of collecting and storing the data.
- If we want to manipulate data to a processor or controller by performing multiplication, addition, etc., we cannot do that directly in the memory where need registers to process and to store the data.
- The 8086 microprocessor contains various kinds of registers that can be classified according to their instructions such as;

- **General purpose registers:** The 8086 CPU has consisted 8-general purpose registers and each register has its own name as shown in the figure such as AX, BX, CX, DX, SI,DI, BP, SP . These all are 16-bit registers where four registers are divided into two parts such as AX, BX, CX, and DX which is mainly used to keep the numbers.
- **Special purpose registers:** The 8086 CPU has consisted 2-special function registers such as IP and flag registers. The IP register point to the current executing instruction and always works to gather with the CS segment register. The main function of flag registers is to modify the CPU operations after mechanical functions are completed and we cannot access directly.
- **Segment registers:** The 8086 CPU has consisted 4- segment registers such as CS, DS, ES, SS which is mainly used for possible to store any data in the segment registers and we can access a block of memory using segment registers.

Simple Assembly Language Programs 8086

The assembly language programming 8086 has some rules such as

- The assembly level programming 8086 code must be written in upper case letters
- The labels must be followed by a colon, for example: label:
- All labels and symbols must begin with a letter
- All comments are typed in lower case
- The last line of the program must be ended with the END directive

- **Op-code:** A single instruction is called as an op-code that can be executed by the CPU. Here the ‘MOV’ instruction is called as an op-code.

- **Operands:** A single piece data are called operands that can be operated by the op-code. Example, subtraction operation is performed by the operands that are subtracted by the operand.

  Syntax: SUB b, c

- **Write a Program For Read a Character From The Keyboard**

```
MOV ah, 1h      //keyboard input subprogram
INT 21h         // character input
                // character is stored in al
MOV c, al       //copy character from alto c
```

- **Write a Program For Reading and Displaying a Character**

```
MOV ah, 1h      // keyboard input subprogram
INT 21h         //read character into al
MOV dl, al      //copy character to dl
MOV ah, 2h      //character output subprogram
INT 21h         // display character in dl
```

- **Write a Program Using General Purpose Registers**

```
ORG 100h
MOV AL, VAR1 // check value of VAR1 by moving it to the AL.
LEA BX, VAR1        //get address of VAR1 in BX.
MOV BYTE PTR [BX], 44h // modify the contents of VAR1.
MOV AL, VAR1        //check value of VAR1 by moving it to the AL.
RET
VAR1 DB 22h
END
```

Write a Program For Displaying Character

```
ORG 100h          ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h    ; set AX to hexadecimal value of B800h.
MOV DS, AX        ; copy value of AX to DS.
MOV CL, 'A'       ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh      ; set BX to 15Eh.
MOV [BX], CX      ; copy contents of CX to memory at B800:015E
RET               ; returns to operating system.

    ;
```

- **Addition**

```
ORG 0000h
MOV DX, #07H    // move the value 7 to the register AX//
MOV AX, #09H    // move the value 9 to accumulator AX//
Add AX, 00H     // add CX value with R0 value and stores the result in AX//
END
```

- **Multiplication**

```
ORG0000h
MOV DX, #04H    // move the value 4 to the register DX//
MOV AX, #08H    // move the value 8 to accumulator AX//
MUL AX, 06H     // Multiplied result is stored in the Accumulator AX //
END
```

- **Subtraction**

```
ORG 0000h
MOV DX, #02H     // move the value 2 to register DX//
MOV AX, #08H     // move the value 8 to accumulator AX//
SUBB AX, 09H     // Result value is stored in the Accumulator A X//
END
```

- **Division**

```
ORG 0000h
MOV DX, #08H      // move the value 3 to register DX//
MOV AX, #19H      // move the value 5 to accumulator AX//
DIV AX, 08H       // final value is stored in the Accumulator AX //
END
```