Emu8086

BEKKOUCHE Mohammed

m.bekkouche@esi-sba.dz

Emu8086 in summary

- **emu8086** is the emulator for the 8086 microprocessor (Intel and AMD compatible).
- It is especially useful for beginners learning assembly language.
- It compiles source code and runs it on the emulator **step by step**.
- The emulator executes programs in **single-step mode**, allowing observation of the execution.
- Registers, flags (FLAGS), and memory can be monitored while the program runs.
- The **ALU** helps visualize internal CPU operations.
- Assembly code runs in a **virtual computer**.

Emu8086 in summary

8086 compatibility

Machine code for the 8086 is presented in the course as compatible with later Intel generations. This gives 8086 code a strong portability advantage.

- The syntax used by emu8086 is simpler than that of many assemblers.
- It remains suitable for programs targeting 8086 machine code.
- Its simplicity makes it a useful environment for learning assembly programming.

Where to start?

1. Start Emu8086 from the Start menu or by running `Emu8086.exe`.
2. Select **Examples** from the **File** menu.
3. Click **Emulate** or press `F5`.
4. Click **Single Step** or press `F8`.
5. Observe how the code is executed and inspect the register values.
6. Open other examples; the examples are heavily commented.

What is the assembler

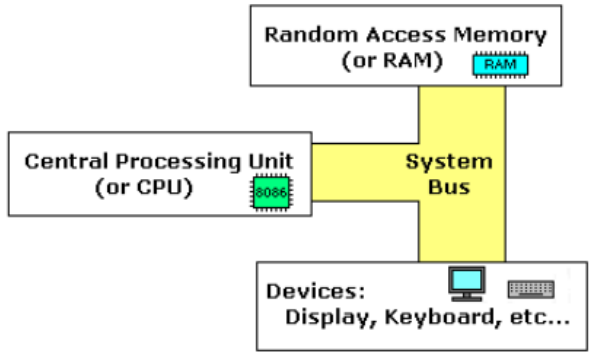
- Assembly is a **low-level language**; its operation is very close to machine language.
- It is closely linked to the microprocessor architecture: registers, memory addresses, interrupts, system calls, and related mechanisms.
- A major advantage is precise control over memory and the instructions executed by the processor.
- Assembly programs can be fast and compact.

Example from the course

A simple “Hello, World!” program is described as much smaller in assembly than in C.

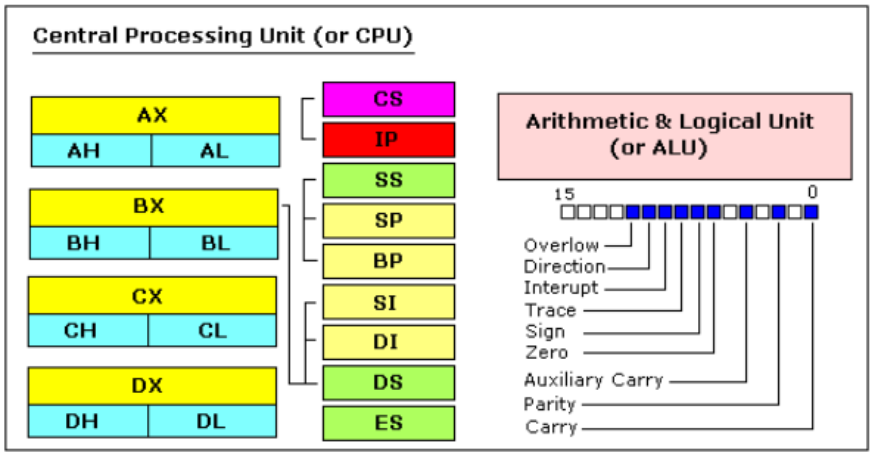
Basic structure of a computer

- **Processor**: performs most computations.
- **RAM**: holds programs while they are executed.
- **Bus system**: connects the main components.



The bus system connects the various components of the computer.

Inside the CPU



General purpose registers

- The 8086 has **8 general purpose registers**.
- `AX` - accumulator (`AH`/`AL`)
- `BX` - base address register (`BH`/`BL`)
- `CX` - count register (`CH`/`CL`)
- `DX` - data register (`DH`/`DL`)
- `SI` - source index
- `DI` - destination index
- `BP` - base pointer
- `SP` - stack pointer

```
AX = AH | AL
```

```
BX = BH | BL
```

```
CX = CH | CL
```

```
DX = DH | DL
```

General purpose registers

- The programmer determines the use of each general-purpose register.
- The main purpose of a register is to keep a number.
- The register size is **16 bits**.
- Example: 0011000100111001b = 12601 in decimal.
- `AX` can be split into two 8-bit registers:

AX	AH	AL
00110001 00111001b	00110001b	00111001b

Changing `AH` or `AL` changes the value represented by `AX`, and the same applies to `BX`, `CX`, and `DX`.

General purpose registers

- Registers are located inside the processor and are therefore much faster than memory.
- Accessing memory requires the system bus and generally takes longer.
- Register data can be accessed very quickly.
- It is therefore useful to keep temporary values in registers when possible.
- The number of registers is small, and several registers have special purposes.

Prefer registers for frequently used temporary data.

Segment registers

`CS`	points to the segment containing the current program
`DS`	usually points to the segment where variables are defined
`ES`	additional segment register
`SS`	points to the segment containing the stack

Important point

Segment registers have a special role: they identify accessible memory blocks. They should not be treated as ordinary data registers.

Segment registers

- Segment registers work together with general-purpose registers to access memory.
- Example: to reach effective address `ABCDEh`, use `DS=ABC0h` and `SI=00DEh`.
- The processor computes:

$$\boxed{\text{ABC0h} \times 10h + \text{00DEh} = \text{ABCDEh}}$$

Why segmentation?

The segment value extends the address range beyond what a single 16-bit register can represent.

Segment registers

- The address formed with a segment and an offset register is an **effective address**.
- By default, `BX`, `SI`, and `DI` use `DS`.
- `BP` and `SP` use `SS`.
- Other general-purpose registers cannot form an effective address.
- Although `BX` can participate, `BH` and `BL` cannot.

Segment + offset register $\rightarrow$ effective address

Registers for special purpose

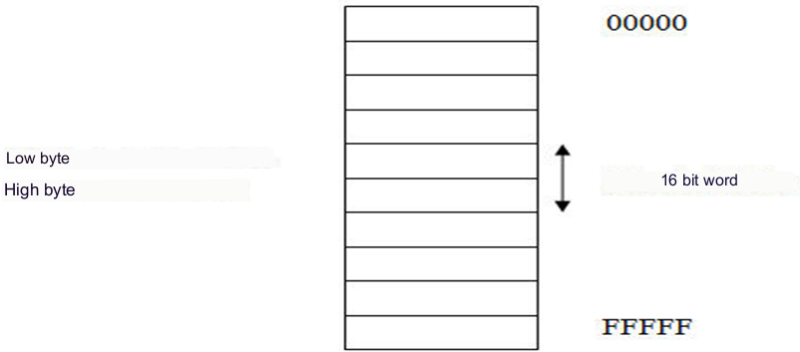
- `IP` always works with `CS` and points to the instruction being executed.
- The `FLAGS` register is modified automatically after many operations.
- Flags provide information about the result and support conditional control transfers.

IP + CS

FLAGS

Memory

- Logically, 8086 memory is organized as a collection of addressable locations.
- Segmentation allows the programmer to work with logical memory blocks while still reaching physical/effective addresses.



Access memory

- The registers used to form memory addresses are **BX, SI, DI, BP**.
- Combining these registers inside square brackets gives several addressing forms.

```
[BX]   [SI]   [DI]   [BP]
[BX+SI]   [BX+DI]   [BP+SI]   [BP+DI]
[BX+SI+disp]   [BP+DI+disp]   ...
```

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

Access memory

- `d8`: 8-bit signed immediate displacement, e.g. `22`, `55h`, `-1`.
- `d16`: 16-bit signed immediate displacement, e.g. `300`, `5517h`, `-259`.
- A displacement may come from an immediate value, a variable offset, or a combination.
- The assembler evaluates the expression and generates one final displacement.
- The displacement may appear inside or outside the square brackets.
- It is a signed value and may be positive or negative.

Access memory

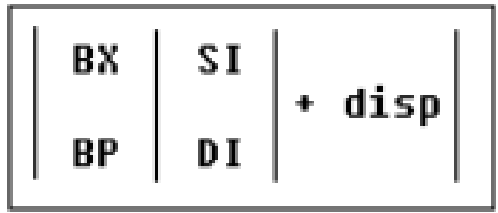
Example

Suppose `DS=100`, `BX=30`, and `SI=70`. For `[BX+SI]+25`:

$$\boxed{100 \times 16 + 30 + 70 + 25 = 1725}$$

- By default, `DS` is used for these modes.
- When `BP` is used, the default segment is `SS`.

Access memory



- Valid addressing modes can be remembered by combining one element from each available column.
- `BX` and `BP` never appear together.
- `SI` and `DI` never appear together.

Access memory

- The value in `CS`, `DS`, `SS`, or `ES` is called the **segment**.
- The value in `BX`, `SI`, `DI`, or `BP` is used as the **displacement/offset**.

`ABCDh:7890h`

Example

If `DS=0ABCDh` and `SI=7890h`, then

$$\text{ABCDh} \times 10h + 7890h = 0B3560h.$$

Access memory

- To specify the data type of a memory operand, use:
- `byte ptr` for one byte.
- `word ptr` for one word (two bytes).
- emu8086 also supports shorter prefixes `b.` and `w.`.

```
byte ptr [BX]   access one byte
word ptr [BX]   access one word
```

The MOV statement

- Syntax: `MOV Destination, Source`
- Copies the source operand into the destination operand.
- The source may be an immediate value, a general-purpose register, or a memory location.
- The destination may be a general-purpose register or a memory location.
- Both operands must have the same size: byte or word.

Destination $\leftarrow$ Source

The MOV statement

```
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
```

Examples of operands

REG: `AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.`
memory: `[BX], [BX+SI+7], variable.`
immediate: `5, -24, 3Fh, 10001101b.`

The MOV statement

For segment registers

Accepted forms include:

```
MOV SREG, memory   MOV memory, SREG
   MOV REG, SREG   MOV SREG, REG
```

`SREG`: `DS`, `ES`, `SS`, and `CS` only as a source.

Restriction

`MOV` cannot be used to modify `CS` or `IP`.

The MOV statement

```
ORG 100h
MOV AX, 0B800h
MOV DS, AX
MOV CL, 'A'
MOV CH, 11011111b
MOV BX, 15Eh
MOV [BX], CX
RET
```

- `ORG 100h` starts a simple one-segment `.COM` program.
- `DS` receives the segment value.
- `CX` is stored through the memory operand `[BX]`.

Execution in the emulator

Compile and emulate with `F5`, then use **Single Step** to observe register changes.

Variables

- A variable is a **memory location** identified by a name.
- A symbolic name is easier to use than a raw address.
- The course uses two main variable types: `BYTE` and `WORD`.

variable name $\longrightarrow$ memory location

Variables

Declaration syntax

```
name DB value
name DW value
```

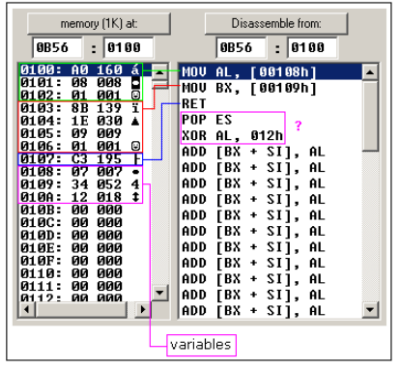
- `DB`: Define Byte.
- `DW`: Define Word.
- The name starts with a letter and may contain letters and digits.
- The value may be decimal, hexadecimal, binary, or `?` for an uninitialized variable.

Variables

```
ORG 100h
MOV AL, var1
MOV BX, var2
RET
VAR1 DB 7
var2 DW 1234h
```

- `VAR1` is a byte variable.
- `var2` is a word variable.

Variables



Observation

The emulator/disassembler shows the variables as memory addresses after assembly.

Variables

- The first memory-list column is the physical address.
- The second column shows the hexadecimal value.
- The third column shows the decimal value.
- The last column shows the ASCII character value.
- Variable names are case-insensitive in the assembler.
- A word occupies two bytes; the low byte is stored at the lower address.

`34h` at lower address $\rightarrow$ `12h` at next address

Variables

Why can instructions appear after `RET`?

The disassembler does not know exactly where the data begins. It simply interprets bytes in memory as valid 8086 instructions.

The processor executes bytes; the source program gives them meaning.

Variables

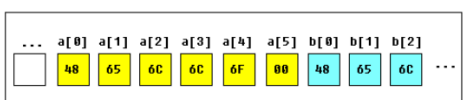
```
ORG 100h
DB 0A0h
DB 08h
DB 01h
DB 8Bh
DB 1Eh
DB 09h
DB 01h
DB 0C3h
DB 7
DB 34h
DB 12h
```

- The same machine code can be described directly using `DB` values.
- The assembler converts the source representation into bytes.
- Those bytes form the **machine code** executed by the processor.

Arrays

- Arrays can be viewed as strings of variables.
- A character string is a byte array whose elements are ASCII codes.

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```



Arrays

- Array elements are accessed with square brackets.

```
MOV AL, a[3]

MOV SI, 3
MOV AL, a[SI]
```

Index registers

`BX, SI, DI, BP` can be used when forming memory addresses.

Arrays

DUP syntax

```
number DUP (value(s))
```

```
c DB 5 DUP (9)
```

which is equivalent to:

```
c DB 9, 9, 9, 9, 9
```

```
d DB 5 DUP (1, 2)
```

which is equivalent to ten bytes containing the repeated sequence `1,2`.

Get the address of a variable

- LEA and OFFSET can be used to obtain the offset address of a variable.

```
ORG 100h
MOV AL, VAR1
LEA BX, VAR1
MOV BYTE PTR [BX], 44h
MOV AL, VAR1
RET
VAR1 DB 22h
```

Get the address of a variable

`LEA BX, VAR1` $\iff$ `MOV BX, OFFSET VAR1`

- Both forms are compiled to the same kind of machine operation shown in the course: `MOV BX, num`.
- `num` is the 16-bit offset of the variable.
- Only `BX`, `SI`, `DI`, and `BP` can be used inside square brackets as memory pointers.

Constants

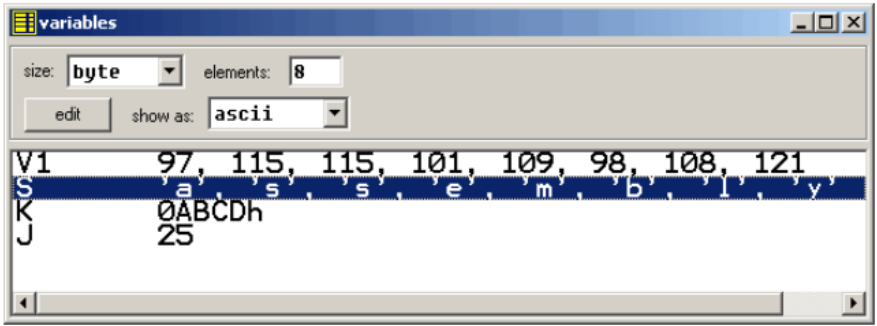
- Constants are similar to variables, but they exist only during assembly.
- After definition, their value cannot be changed.
- The `EQU` directive defines a symbolic constant.

```
k EQU 5
MOV AX, k
```

$$\texttt{MOV AX, k} \equiv \texttt{MOV AX, 5}$$

Variables

- Variables can be displayed during execution using **View** → **Variables**.



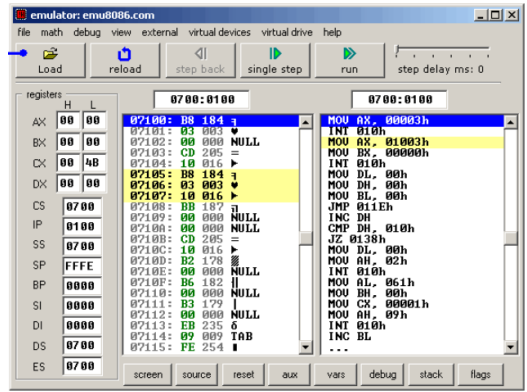
Variables

- To display an array, select a variable and set the **Elements** property to the array size.
- Assembly has no strict data types, so a variable can be displayed as an array.
- The value can be viewed as:

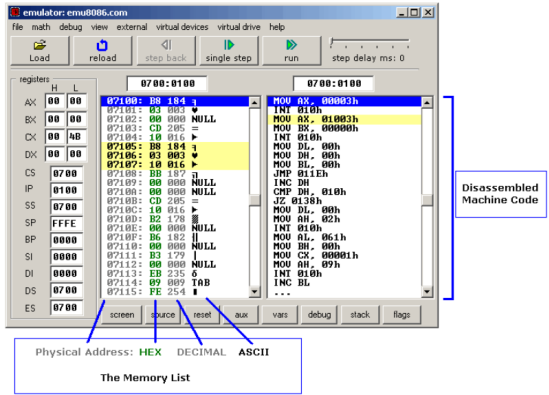
HEX BIN OCT SIGNED UNSIGNED CHAR

Using the emulator

- Click **Emulate** to load the source into the emulator.
- The emulator can also load executables created by other assemblers using **Show emulator** from the Emulator menu.



Using the emulator



Basic sequence

File → Examples → load an example → compile → load into the emulator.

Using the emulator

- **Single Step**: executes one instruction at a time and stops after each instruction.
- **Run**: executes instructions with the selected step delay.
- Double-click a register to open the **Extended Viewer**, where the value is shown in several forms and can be changed.
- Double-click a memory item to inspect a word value; the low byte is at the selected address and the high byte is at the next address.
- **Flags**: view and change flags during execution.