Chapter 1 Highlights

1. HLA software
    a. Installation
    b. Creating and executing a sample project (not to submit this)
    c. Creating **Project 1** and submitting it. See the Instructions file on Modules.
    d. See file called **Projects Guide** in the **Modules** section.
2. HLA-related introductory programming features
    a. Program structure
    b. Programming features such as such as types, variables, …
3. 80x86 Machine
    a. Registers
    b. Memory, concept of "address" of instructions
    c. Machine instructions
4. Variable
5. Data Types
6. Control structures (if, if-else, while, repeat, for, break…).
7. Relational operators (=, < …)
8. Logical operators (&&, ||, …)
    a. These features (4, 5, 6) are conceptually identical to C/C++, with different syntax.
9. Try-exception-endtry (Not covered)
10. HLA IO library: There will be no questions from this part, however, you need to know them for using in projects. Also, they could be used in the body of quiz questions. I have provided a summary of these functions.

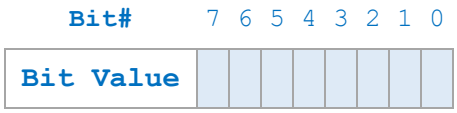
Bit, Byte, and Registers

Bit: basic unit, can be:

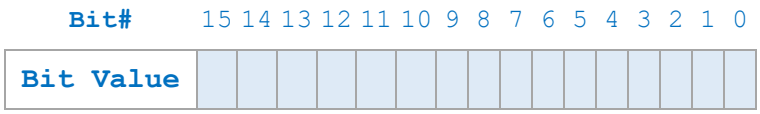
- **0** or **1**,
- **false** or **true**,
- **off** or **on**

Byte = 8 bits

Byte:



Word = 2 bytes

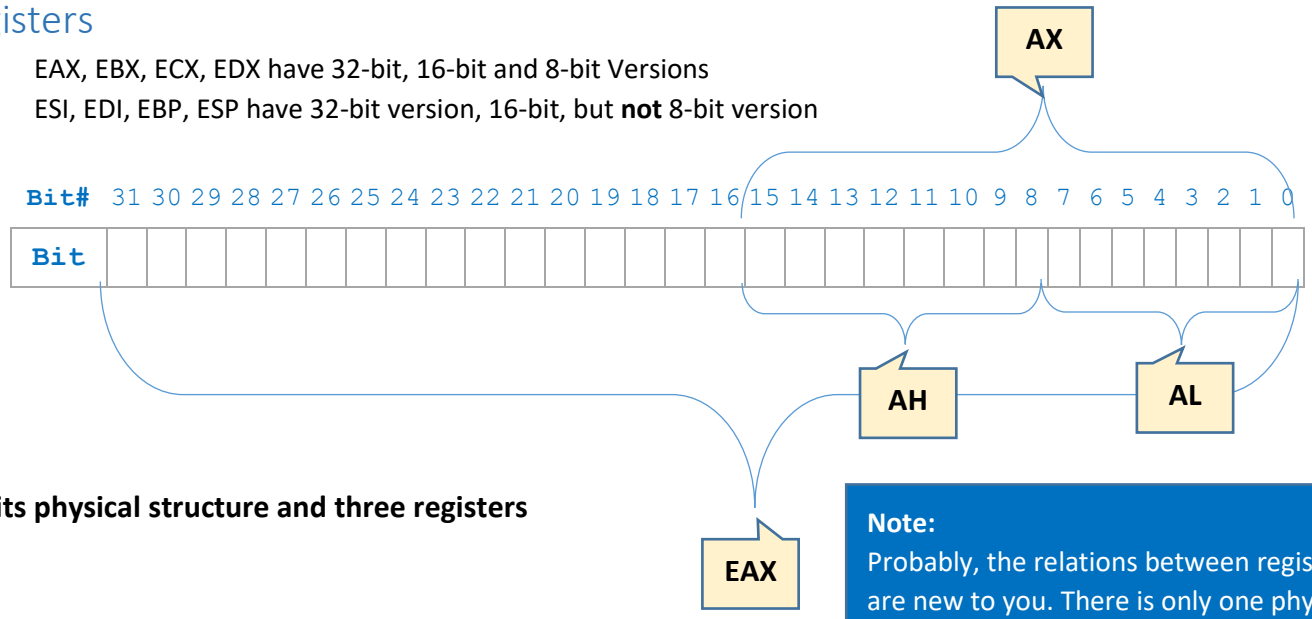


Double Word = 4 bytes

Only bit values (shaded areas) physically exist. Others are for explanation.

Registers

- EAX, EBX, ECX, EDX have 32-bit, 16-bit and 8-bit Versions
- ESI, EDI, EBP, ESP have 32-bit version, 16-bit, but **not** 8-bit version



32 bits physical structure and three registers

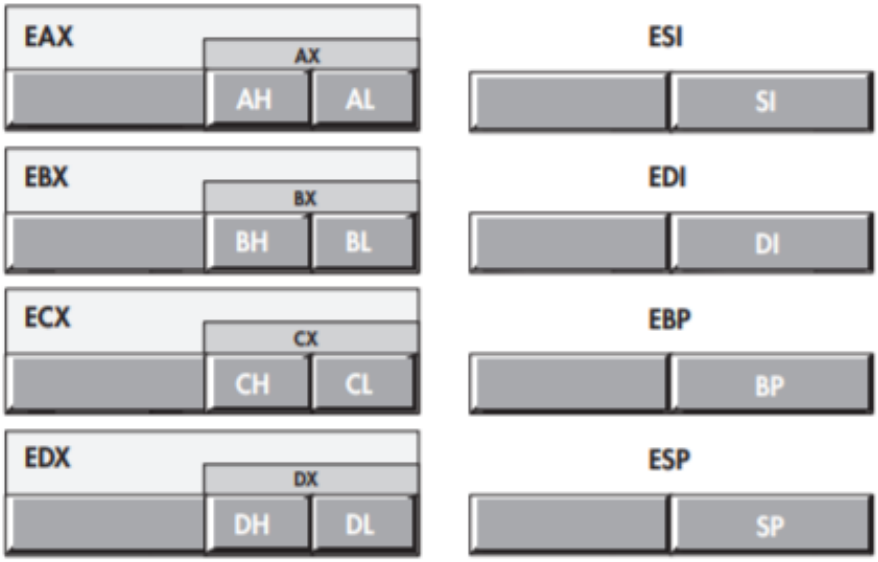
Note:
Probably, the relations between registers are new to you. There is only one physical 32-bit register. But assembly language provides access to all 32 bits, 16 bits, or 8 bits of the register.

Registers List

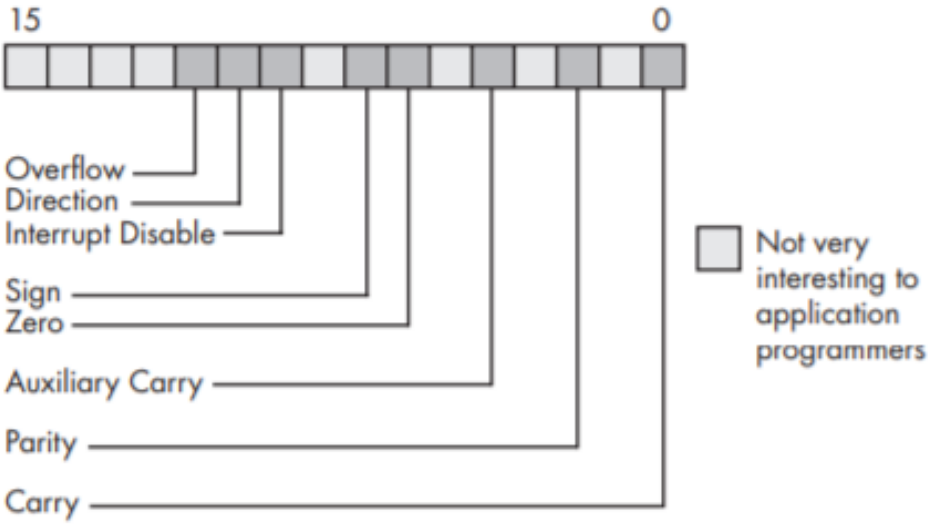
A, B, C, D: Identical structure
Two 8-bits, one 16-bit, one 32-bit Registers
Each A, B, C, and D series:
One 32-bit physical structure, three logical registers. They are overlapped.

SI, DI, BP, SP: 16-bit and 32-bit
Each one has one 32-bit physical structure, and two logical registers

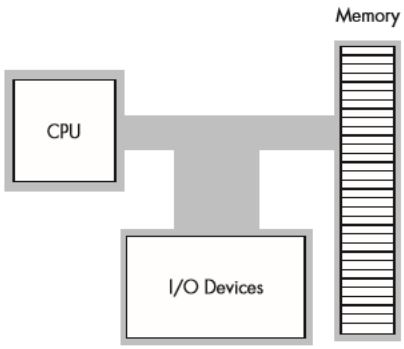
Another Image (from Textbook)
Relations between Registers



Flags Register

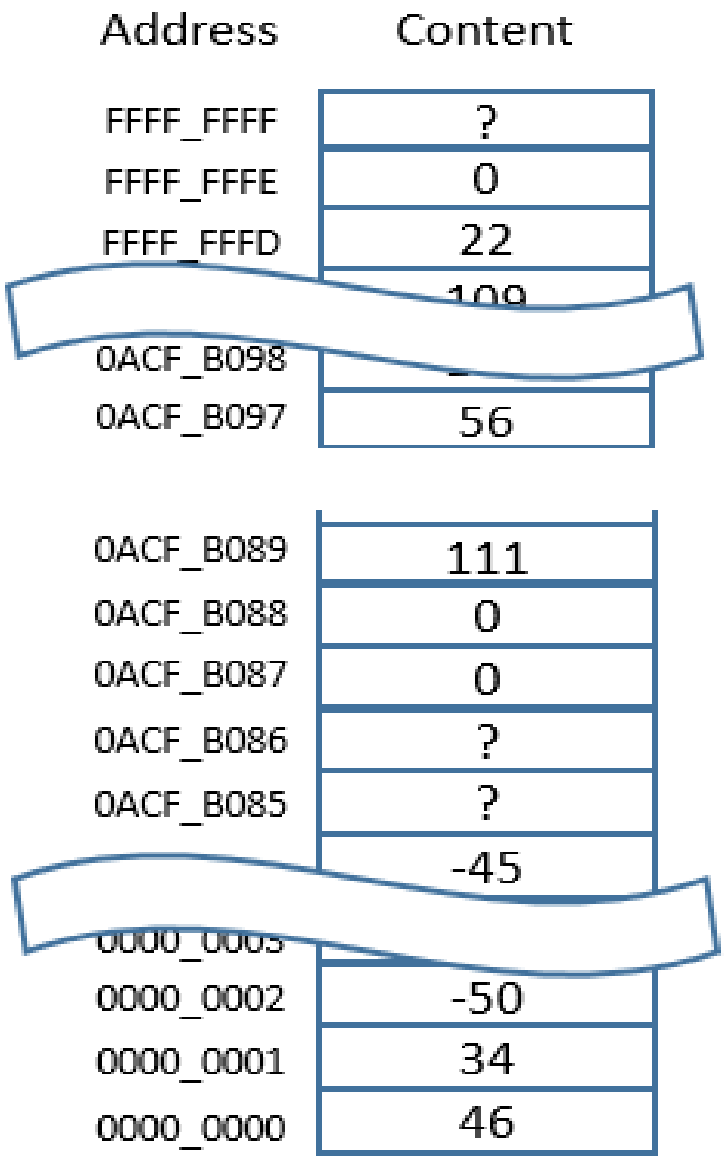


CPU and Memory:



Memory:

- Content
- Address
- HLA memory is byte-wise addressable. Meaning every byte has it own address and not bundled with other bytes, having one address for all.
- Byte
- Word = n bytes = 2 bytes



Basic Instructions

- MOV // Actually it is a copy
- ADD
- SUB

`MOV (From, To)`, To = From

MOV instructions moves data between:

- Register to Register
- Register to memory
- Memory to Register
- Constant to Register
-
- Not Memory to Memory

High-level Language	**Assembly Language**
`x = y`	`MOV (y, Register)` `MOV (Register, x)`
`x = 102` // 102 a constant	`MOV (102, x)`
`102 = x` // Illegal	`MOV (x, 102)` // Illegal

Other restrictions in Assembly:

MOV between two registers and between a register and a memory must be same size:

- 8 bits to 8 bits
- 16 bits to 16 bits
- 32 bits to 32 bits

There is no such restriction when a constant is moved to memory location or register. However, the constant must fit in the register or memory. Chapter 2 talk about this matter.

How you move data between two different size of registers and memory:

Example 1: How to move M8 to R16?
Use one of the general-purpose registers: A, B, C, D series. (A = Al, AH, AX, EAX, B =…)

```
MOV (0, AX);          // Set AX to zero
MOV (M8, AL);         // Now M8 is copied to AX!!
```

Explanation:
Assume M8 in binary: **1011_0110** // Under-score is inserted for easy-reading, some use space
MOV (0, AX) makes AX = 0000_0000_0000_0000
MOV (M8, AL) makes AL = 1011_0110, which makes AX =0000_0000_**1011_0110**

Question: why you need to set AX to zero?
In high level language:

```
x = 0;            // Unnecessary?
x = y;
```

Example 2: How to move M32 to R16

```
MOV (M32, EAX);          // Now M32 is copied to AX??
```

This works if only content of M32 was small enough to fit in 16 bits.
Incorrect if not.
Any solution? No.
If M32 content does fit in 16 bits, the above attempted operation is overflow and incorrect.

How about setting EAX to zero, like previous example. (When it is correct).

No need to MOV (0, EAX) before MOV(M32, EAX). Why not?

ADD instruction:
Same format and same condition as MOV

ADD (y, x)

Adds y to x
Which means x = x + y

x = y + z + t will be as follow:

```
MOV (y, x)
ADD (z, x)
ADD (t, x)
```

```
a = b + 2 * (c + 2*d)        // Assume all are 16-bits or less

1. MOV (d, AX);              // AX = d
2. MOV (AX, BX);             // BX = AX = d
3. ADD (BX, AX);             // AX = AX + BX = d + d = 2*d
4. ADD (c, AX);              // AX = AX + c = 2*d + c
5. MOV (AX, BX);             // BX = AX = 2*d + c = c + 2*d
6. ADD (BX, AX);             // AX = BX + AX = c + 2*d + c + 2*d = 2 *(c + 2*d)
7. ADD (b, AX);              // AX = b + 2* (c + 2*d)
8. MOV (AX, a);              // a = b + 2 * (c + 2*d)
```

Can we do this? Combine 2 & 3: MOV (AX, AX)

```
Exercise: a = 2* b + 2 - (c + 4*d)
Program it with 16-bit integers, Read b, c, d, compute a and print
Do not submit
```

Variables

Declaration:

```
static
    var_name:  type;

    var_name:  type := initial-value;
```

Variable types:

```
int8
int16
int32
boolean
char
```

Variables are case-sensitive

HLA's identifiers are **case neutral**. This means that the **identifiers are case sensitive** insofar as you must always spell an identifier exactly the same way in your **`program`** (even with respect to upper- and lowercase). However, unlike in case-sensitive languages such as C/C++, you may not declare two identifiers in the program whose name differs only by alphabetic case.

You must type variable name the way you declared. And you cannot have another variable with that spelling.

Relational Operators:

```
== (or =)
!= (or <>)
<
<=
>
>=
```

Control Structures

Selection:

```
if (expression) then
    Statements
elseif (expression) then
    Statements
endif;
```

Loops

```
while (expression) do
    Statements
Endwhile;
```

```
repeat
    Statements
until (expression);
```

```
for (init; expression; increment) do
    Statements
endfor ;
```

Logical Operators

```
&&
||
!
```

```
// Exercise, do not submit
// Program to read and add numbers; count number of positives and negatives. Stops when zero is entered.

program EX2;

#include ("stdlib.hhf");

static
  num: int16;
  sum: int16 := 0;
  posCount: int16 := 0;
  negCount: int16 := 0;

begin EX2;
  stdout.put ("Enter a number: ");          // Prompt and read first number
  stdin.get (num);

  while (num != 0) do
    MOV (num, AX);
    ADD (AX, sum);    sum = sum + num                    // Accumulate numbers

    if (num > 0) then                       // Count positives and negatives
       ADD (1, posCount);
    elseif (num < 0) then
       ADD (1, negCount);
    endif;

    stdout.put ("Enter a number: ");        // Prompt and read more numbers
    stdin.get (num);

  endwhile;

  stdout.newln ();                          // Output
  stdout.put ("Sum = ", sum, nl);
  stdout.put ("Number of negatives = ", negCount, nl);
  stdout.put ("Number of Positives = ", posCount, nl);

end EX2;
```

HLA IO Library Functions			
	Name	Examples/Use	Description
1	`stdout.newln`	`stdout.newln();`	Prints a new line

2	**`stdout.putiX`**
	`stdout.puti8`
	`stdout.puti16`
	`stdout.puti32`
		`stdout.puti8(123);`
		`stdout.puti8(AL);`
		`stdout.puti16(23456);`
		`stdout.puti16(DX);`
		`stdout.puti16(i16Var);`
		`stdout.puti32(EAX);`
		`stdout.puti32(i32Var);`
			Prints a single parameter as assigned integer value. The single parameter can be a constant. Register, or a memory variable. It prints on standard output device (monitor).

3	**`stdout.putiXSize`**
	`stdout.put8XSize`
	`stdout.put16XSize`
	`stdout.put32XSize`
		`stdout.putiXSize(val,width,podCh);`
		
		`stdout.puti8Size(AL, 5, ' ');`
		`stdout.puti16Size(BX, 12, ' ');`
		`stdout.puti16Size(i16Var, 6, ' ');`
		`stdout.puti32Size(ECX, 8, '$');`
		`stdout.puti32Size(i32Var, 7, ' ');`
			Prints val with width minimum number of places. If width is negative, the printed value is left-justified, if it is positive, it is right-justified.

	`stdout.put`	`stdout.put(EDX, i16Var, ..)`	Prints list of items.
4	**`stdin.get`**	`stdin.get(EDX, i16Var, ..)`	Reads list of items.
5	**`stdin.getc`**	**`s`**`tdin.getc();`	Reads a character into AL reg.

6	**`Stdin.getiX`**
	`stdin.geti8`
	`stdin.geti16`
	`stdin.geti32`
		
		`Stdin.geti8();`
		`Stdin.geti16();`
		`Stdin.geti32();`
			
			Reads a signed 8-bit value into AL
			Reads a signed 16-bit value into AX
			Reads a signed 32-bit value into EAX

7	`stdin.readln` `stdin.flushInput`	Rarely use. See page 41 if you need to use it.	