Assembly Language

BEKKOUCHE Mohammed

m.bekkouche@esi-sba.dz

MODULE PROGRAM

I- PRESENTATION OF THE MACHINE

- Functional description of the machine
- Internal code and internal format of an instruction
- Internal structure of a program
- Description of the symbolic language (general syntax of the language)

II- PRESENTATION OF THE ASSEMBLY LANGUAGE

- General structure of a source program (symbolic)
- Guidelines
- Transfer Instructions
- Arithmetic instructions
- Instructions for comparison, loops (repetitions) and branches
- Bit manipulation instructions (logics and shifts)
- Stack Statements
- Procedure instructions and interruptions
- String and prefix processing instructions

III- MACROS INSTRUCTIONS

IV- EXTENDED INSTRUCTIONS (multi media instructions ,. . .)

Calculation of the mark of the Operating Systems 2 Module

60% Exam Score

40% TP score

Introduction

Introduction to Microprocessors

- Thanks to progress in integration, the increase in performance has focused on:
  - the operating speed.
  - the width of the processed words (8, 16, 32, 64 bits).
  - the number and complexity of the operations that can be carried out.
- The objective of this module is twofold:
  - present the basic notions necessary to understand systems using microprocessors,
  - and carry out practical work to program in machine language (assembler).

A processor, in two words

- A processor is the heart of any computer: it executes the instructions that make up the programs we ask it to execute. Instructions are stored in memory (outside the processor).
- These instructions (which together make up assembly language, or assembler) are very simple but nevertheless allow, by combining them, to perform any programmable operation.
- To execute a program, the processor reads instructions into memory, one by one. It knows at all times the address (the place in the memory) to which is the next instruction to be executed because it stores this address in its ordinal counter.

A processor, in two words

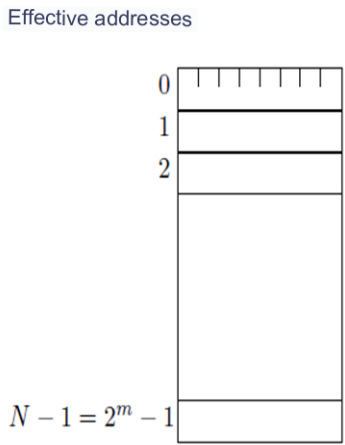
- Instructions act on data that is located either in memory or in processor registers. A register is a memory element internal to the processor and containing a value.
- The number of registers is very limited, 14 in this case for the 8086.
- To access data in memory, you must specify its address.
- To access data in a register, you must specify its name (each register has a name which is a string of characters).

A processor, in two words

- The 8086 is a 16-bit processor, that is, it processes 16-bit encoded data.
- A few definitions:
  - a bit is a binary value which, by convention, can take the value 0 or 1;
  - a byte is data coded on 8 bits;
  - a word is data coded on 16 bits;
  - one Kbyte is a set of 1024 bytes.

Memory

- Memory is a sequence of bytes, all uniquely numbered by an integer between 0 and $N-1$.
- We then say that the capacity of memory is N bytes.
- We always have N which is a power of 2, $N = 2^m$.
- The effective address (EA) of a byte in memory is the number associated with it.



Memory

- For the 8086, and therefore for us 8086 programmers, a byte is designated by a couple:

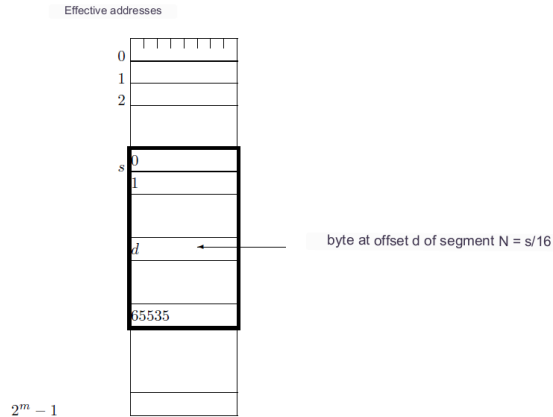
$$\text{(segment number, displacement in the segment)}$$

- which constitute a segmented address (SA).
- A segment is a set of 64 Kbytes consecutive. The displacement (or offset) specifies a particular byte in a segment.
- Segment and displacement are coded on 16 bits and can therefore take a value between 0 and 65535.

$$\boxed{\text{effective address} = 16 \times \text{segment} + \text{displacement}}$$

Memory

- Segmented addressing. In bold lines, the segment starting at the effective address s has been represented, and its number is therefore N $= \frac{s}{16}$. s must be a multiple of 16 to be the 1st byte of a segment .



Memory

Exercise

Give the effective addresses from the segmented addresses (all in hexadecimal).

Where does each segment start and end?

Segmented address	Actual address
06D1:000D	
059E:0000	
0700:00FB	
FFFF:0005	

Memory

Solution

Give the effective addresses from the segmented addresses (all in hexadecimal).

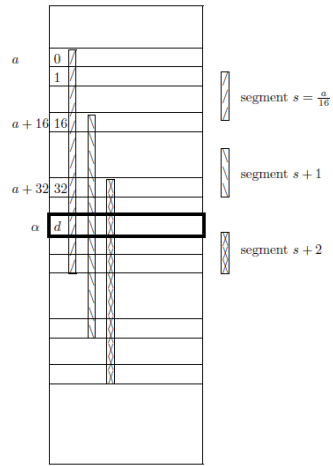
Segmented address	Effective address	Segment start		Segment end	
		Seg Addr	Eff Addr	Seg Addr	Eff Addr
06D1:000D	06D1D	06D1:0000	06D10	06D1:FFFF	16D0F
059E:0000	059E0	059E:0000	059E0	059E:FFFF	159DF
0700:00FB	070FB	0700:0000	07000	0700:FFFF	16FFF
FFFF:0005	FFFF5	FFFF:0000	FFFF0	FFFF:FFFF × FFFF:000F ✓	10FFEF × FFFFF ✓

Memory

- A consequence of the relationship stated above between effective address and segmented address is that a given effective address byte can be accessed in several ways.
- More specifically , to each EA corresponds $2^12 =$ 4096 different SAes.

Memory

- The segments: we have represented a few segments to show their overlap and the fact that a memory byte has several segmented addresses.
- For example, the effective address byte indicated in the figure can be addressed in multiple ways , depending on whether it is considered to be in segment s , s+1 , s+2 , ...
- In each of these cases, the segmented address of byte indicated in the figure is (s, d) , (s + 1, d - 16) , ...
- Note that each memory byte belongs to $2^12=$ 4096 different segments.



Memory

Exercise

Give two segmented addresses for each real (effective) address (everything is in hexadecimal).

Effective address	Segmented address 1	Segmented Address 2
FFFFF		
08A9F		

Memory

Exercise / Solution

Give two segmented addresses for each real (effective) address (everything is in hexadecimal).

Effective address	Segmented address 1	Segmented Address 2
FFFFF	FFFF:000F	FFFE:001F
08A9F	0700:1A9F	0800:0A9F

Memory

During its execution, a program uses several memory segments (see fig. of the next slide):

- a segment contains the program instructions (the code segment);
- a segment contains the program data (the data segment);
- a segment contains the program stack (the stack segment). This very important segment.

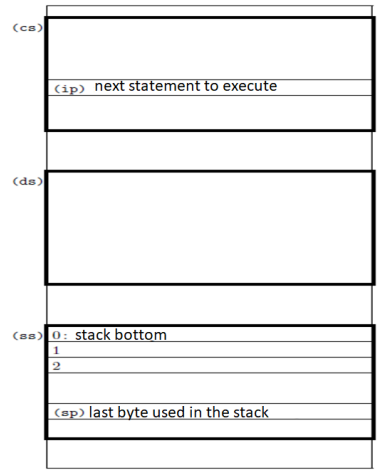
Remark

These three segments can be anywhere in memory and even partially or totally overlap.

Memory

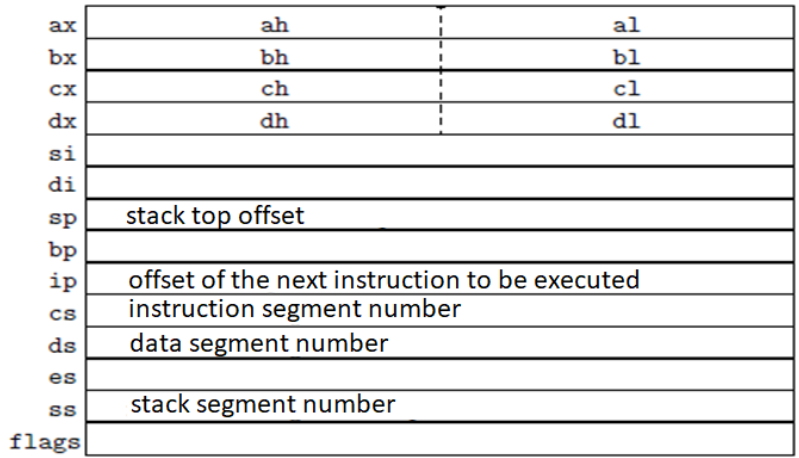
Structure of memory. We have represented in bold lines the segments of code, data and stack that are created when a program is triggered to run (from top to bottom – in reality, of course, they can be located anywhere in memory).

- Each segment is referenced by a segment register.
- The different segments can partially or completely overlap.



The registers of the 8086

The 8086 has 14 16-bit registers



The registers of the 8086

- **AX** general-purpose register containing data. The least significant 8 bits are called AL and the most significant 8 bits are called AH.
- **BX** general-purpose register containing data. Like AX, BX decomposes into BL and BH.
- **CX** general-purpose register containing data. Like AX, CX decomposes into CL and CH.
- **DX** general-purpose register containing data. Like AX, DX decomposes into DL and DH.
- **SI** general-purpose register generally containing the displacement in a segment of data.
- **DI** general-purpose register generally containing the displacement in a segment of data.
- **BP** register used to address data in the stack.
- **SP** stack pointer register.
- **IP** instruction pointer register (ordinal counter). This register indicates the next instruction to be executed.

The 8086

FLAGS – processor state flag register

Some bits of this register have names. These are all binary indicators:

- O the overflow bit is set by most arithmetic instructions to indicate whether there has been an overflow when calculating (a number too large or too small).
- D direction bit. It will be described later.
- S the sign bit is set by most arithmetic instructions to indicate the sign of the result (positive or negative).
- Z the zero bit is set by most arithmetic instructions to indicate that the result of the calculation is 0.

The 8086

C the carry bit is set by most arithmetic instructions to indicate whether the calculation resulted in a carry that needs to be carried over to subsequent calculations.

A the auxiliary carry bit is set to 1 by most arithmetic instructions to indicate whether the last operation generated a carry from bit number 3 to bit number 4; it is set to 0 otherwise.

P the parity bit is set by most arithmetic instructions. It indicates whether the 8 least significant bits of the result contain an even number of 1.

The registers of the 8086

Segment registers

The 8086 also includes 16-bit registers to hold segment numbers:

- **CS** code segment – segment containing the running program.
- **DS** data segment – segment containing the data.
- **ES** auxiliary segment register for addressing data.
- **SS** stack-segment – segment containing the stack.

The values of the CS, DS, and SS registers are automatically initialized by the operating system at the program's launch. Consequently, these segments are implicit, meaning that if you want to access a piece of data in memory, you only need to specify its offset without worrying about the segment.

The coding of numbers

- The 8086 instructions manipulate integer numeric values encoded on 1 or 2 bytes.
- We describe here the coding of integers. We distinguish four types of coding:
  - unsigned for necessarily positive integers;
  - signed for positive or negative integers;
  - compacted decimal, or binary-coded decimal (BCD);
  - non-compacted decimal.
- It's up to the programmer to decide what coding he uses and write his program accordingly.
- The computer does not know which coding is used. It just executes instructions.

Unsigned representation of integers

- With l bits, it is possible to encode 2^l different values.
- If $l = 8$, we can code 256 different values.
- This encoding allows representing any integer between 0 and $2^l - 1$ with l bits.
- Since only positive numbers are represented, this coding is qualified as unsigned.

Valeur	Codage
0	00000000
1	00000001
2	00000010
3	00000011
...	...
84	01010100
...	...
254	11111110
255	11111111

Signed representation of integers

If we want to be able to represent positive and negative numbers, the natural idea is, rather than coding numbers between 0 and 255, to represent numbers between -128 and $+127$, which always represents 256 different values to encode with 8 bits.

Valeur	Codage
-128	10000000
-127	10000001
...	...
-45	11010011
...	...
-1	11111111
0	00000000
1	00000001
2	00000010
...	...
84	01010100
...	...
126	01111110
127	01111111

Signed representation of integers

- This coding is called two's complement coding.
- In the signed coding, we see that the most significant bit is 1 for all negative numbers, 0 for all positive numbers. This bit therefore indicates the sign of the coded number. Also, this bit is called the sign bit.
- Care must be taken that if the sign bit is 0, the value represented is the same in signed or unsigned coding. On the other hand, if the sign bit is equal to 1, the coded value is greater than 127 in unsigned coding, less than 0 in signed coding.

Coding decimal

In decimal coding, a decimal digit (thus between 0 and 9) is coded on 4 bits. 4 bits coding in principle 16 different values, only 10 of these 16 combinations are actually used in decimal coding, the others then having no meaning.

Valeur	Codage
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Coding decimal

There are two types of representation, compacted or not. A digit requiring 4 bits for its coding, two digits can be represented per byte. This encoding is called compacted decimal encoding. It is also possible to put only one digit per byte, the 4 most significant bits of the byte being unused. This encoding is called non-compacted decimal. For example, 19 is coded:

compacted decimal	00011001	
non-compacted decimal	00000001	00001001

Two bytes are required in the non-compacted representation.

Two's complement of a number

Here we show how to get the two's complement of a number. This method gives the signed binary representation of a negative number. The algorithm is simple. Let's calculate the representation of the number $-n$ (where n is a positive number).

1. write n in binary form
2. complement all the bits ($0 \rightarrow 1$, $1 \rightarrow 0$)
3. add the value 1 to this binary number

Two's complement of a number

We present this algorithm on an example. Let's calculate the two's complement of -24:

8-bit binary representation of 24	is	00101000
by complementing the bits		11010111
by adding 1		11011000

The two's complement of 00101000 is therefore 11011000. The signed binary representation of -24 is therefore 11011000.

Extension of the sign

Extending the sign of a data consists, during the transfer of the value of a byte in a word, in copying the sign bit of the byte over all the bits of the most significant byte of the word.

For example, extending the sign of 10011110 into a word gives

11111111 10011110