Pandit Deendayal Petroleum University
School of Technology
Department of Computer Science and Engineering
Odd Semester 2022-2023
<u>Course student handout file (LAB MANUAL)</u>
INDEX

Name of the course: Microprocessor Programming & Interfacing LAB	Course Code: 20CP202P
Program: B. Tech. Branch: CSE – Computer Engineering	Semester: 3th Academic Year: 2022-23
Name of Course Coordinator: Dr. Samir Patel	
Subject Teachers (Division wise/Batch wise): 1. Dr. Samir Patel (G10 and G12), Mr. Dharmesh N Khandhar (G1-G9, G11)	

1	Departmental Vision & Mission
2	Program educational objectives (PEOs) of Department
3	Program Outcomes (POs)
4	Program Specific Outcomes (PSOs)
5	Academic Calendar
6	Class Time Table and Faculty Time Table with office hours
7	Course Outcomes (COs), Course Syllabus, Pre requisites for the course
8	Lesson Plan
9	Program Articulation Matrix and Course Articulation Matrix
10	Evaluation Scheme and Rubrics
11	Tutorials, Assignments, Case Studies, Quiz, Presentations etc.
12	Copy of Mid and End semester Examination Question Papers (Old and Current), solution of current examination with stage-wise marking scheme
13	Course covered beyond syllabus
14	Actual Engagement of Class
15	Attendance Record (Up to Mid Semester Examination and Up to End semester Examination)
16	Details for Remedial Classes (list and identification of slow learners, actions taken)
17	Justification for Course Outcomemapping with Exams and Assessments
18	Result of students (marks of mid, end and internal assessment components)
19	Direct Attainment of COs and POs and interpretation (Result analysis)
20	Indirect Attainment of POs through Course Exit Survey (Just before end sem. exam)
21	Final Attainment of COs and POs and interpretation (Result analysis), Actions to be taken if COs and POs are not achieved
22	Sample answer scripts of mid sem., end sem. exam and assignments ofGood, Better and Best performing students (at least five copies of each assessment tool)
23	Class notes (Lecture PPT & Lab manual etc.) in Soft/ Hard copy

Date: 25/07/2022

Signature of Subject Teachers **Signature of Department Coordinator (IQAC)** **Signature of Head of the Department**

Departmental Vision & Mission

Vision

"To contribute to the society by imparting transformative education and producing globally competent professionals having multidisciplinary skills and core values to do futuristic research & innovations."

Mission

- To accord high quality education in the continually evolving domain of Computer Engineering by offering state-of-the-art undergraduate, postgraduate, doctoral programs.
- To address the problems of societal importance by contributing through the talent we nurture and research we do:
- To collaborate with industry and academia around the world to strengthen the education and multidisciplinary research ecosystem.
- To develop human talent to its fullest extent so that intellectually competent and imaginatively exceptional leaders can emerge in a range of computer professions.

Program educational objectives (PEOs) of Department

The Program Educational Objectives of B. Tech. (Computer Engineering) program are:

1. To prepare graduates who will be successful professionals in industry, government, academia, research, entrepreneurial pursuit and consulting firms
2. To prepare graduates who will make technical contribution to the design, development and production of computing systems
3. To prepare graduates who will get engage in lifelong learning with leadership qualities, professional ethics and soft skills to fulfill their goals
4. To prepare graduates who will adapt state of the art development in the field of computer engineering

Program Outcomes (POs)

Undergraduate engineering program are designed to prepare graduates to attain the following program outcomes:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. Design / development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

The graduates of CSE department will be able to:

1. Develop computer engineering solutions for specific needs in different domains applying the knowledge in the areas of programming, algorithms, hardware-interface, system software, computer graphics, web design, networking and advanced computing.
2. Analyze and test computer software designed for diverse needs.
3. Pursue higher education.

Academic Calendar Odd Sem 22-23

		11	12	13	14	15	16	17	FACULTY DEVELOPMENT PROGRAMME WEEK
		18	19	20	21	22	23	24	FACULTY DEVELOPMENT PROGRAMME WEEK
1	JULY 2022	25	26	27	28	29	30	31	COMMENCEMENT OF ODD SEMESTER: July 25
2	AUGUST	1	2	3	4	5	6	7	
3		8	9	10	11	12	13	14	RAKSHA BANDHAN
4		15	16	17	18	19	20	21	INDEPENDENCE DAY, JANMASHTAMI
5		22	23	24	25	26	27	28	
6	SEP	29	30	31	1	2	3	4	SAMVATSSARI
7		5	6	7	8	9	10	11	
8		12	13	14	15	16	17	18	**MID-SEM EXAMINATIONS**
9		19	20	21	22	23	24	25	
10	OCT	26	27	28	29	30	1	2	COURSE FEEDBACK WEEK
11		3	4	5	6	7	8	9	DUSSHERA
12		10	11	12	13	14	15	16	
13		17	18	19	20	21	22	23	
14		24	25	26	27	28	29	30	DIWALI WEEK
15	NOV	31	1	2	3	4	5	6	
16		7	8	9	10	11	12	13	GURU NANAK JAYANTI
17		14	15	16	17	18	19	20	COMPLETION OF ODD SEMESTER: Nov. 18
		21	22	23	24	25	26	27	**FOET Practical Exams : Nov.21 Onwards** **FOLS Sem. End Examination : Nov. 21 Onwards**
	DEC	28	29	30	1	2	3	4	**FOET Sem. End Examination : Nov.28 Onwards**
		5	6	7	8	9	10	11	
		12	13	14	15	16	17	18	Rural Internship for FOLS Students: Dec. 17, 2022 to Jan. 10, 2023
		19	20	21	22	23	24	25	
		26	27	28	29	30	31	1	**WINTER BREAK**

- **TOTAL WEEKS: 17**
- **Applicable to FOLS & FOET (Except B.Tech. Sem.2)**

20CP202P					Microprocessor Programming & Interfacing LAB					
Teaching Scheme					Examination Scheme					
L	T	P	C	Hrs/Week	Theory			Practical		Total Marks
					MS	ES	IA	LW	LE/Viva	
0	0	2	1	2	-	-	-	50	50	100

Course Outcomes (COs), Course Syllabus, Pre requisites for the course

COURSE OBJECTIVES

- To impart the basic concepts of microprocessor
- To be familiar with writing assembly language programs
- To understand and implement concepts about interfacing
- To apply the concepts for interfacing different peripherals like keyboard, display, etc.
- Compare different advanced processors.

LIST OF EXPERIMENTS:

Following list gives some programming examples. Faculty can prepare their own list in same manner keeping above guidelines and syllabus in mind.

1. Write an 8086 assembly language program for exchanging two 8-bit numbers, add two 8-bit numbers. Store result in a variable, display number on screen.
2. Write an 8086 assembly language program to read a number from keyboard and do addition, subtraction, multiplication and division and display the answer on screen, find the minimum from block of N 8-bit numbers, to check the string is palindrome or not.
3. Write an 8086 assembly language program to sort an array of 8-bit numbers. find the number of 1's binary representation of given 8-bit number, to count the length of string.
4. Write an 8086 assembly language program to covert a number from one base to another base, to compute even parity and insert it as MSB in 8-bit number.
5. Write an 8086 assembly language program in C using ASM directive, to check the number is prime or not, programs related to interfacing with devices.

COURSE OUTCOMES

On completion of the course, student will be able to

CO1. Describe the various features of microprocessor.

CO2. Explain various elements of 8086 microprocessor architecture.

CO3. Select required instructions by considering the addressing modes.

CO4. Analyse different concepts of programmable interfacing with microprocessor.

CO5. Compare different features of advance microprocessors.

CO6. Use assembly language to program 8086 for Interfacing.

TEXT/REFERENCE BOOKS

1. K.R.Venugopal, Microprocessor x86 programming, BPB
2. Ramesh S. Gaonkar Pub: Microprocessor Architecture, Programming, and Applications with the 8085, Penram International.
3. N. Senthil Kumar, M. Saravanan, S. Jeevanathan, S. K. Shah, Microprocessors and Interfacing, Oxford
4. Daniel Tabak, Advanced Microprocessors, McGrawHill
5. Douglas Hall, Microprocessor & Interfacing, TMH

END SEMESTER EXAMINATION PATTERN

Max. Marks: 100
Exam Duration: 2 Hrs

Part A: Evaluation Based on the class performance and Laboratory book
50 Marks

Part B: Viva Examination based conducted experiments 50 Marks

END SEMESTER EXAMINATION QUESTION PAPER PATTERN

For evaluation of the course,
50% weightage will be given to the Internal Assessment and LPW
50% weightage will be given to the End Semester Exams Practical/Viva.

Course Articulation Matrix

CO1. Describe the various features of microprocessor.
CO2. Explain various elements of 8086 microprocessor architecture.
Co3. Select required instructions by considering the addressing modes.
CO4. Analyse different concepts of programmable interfacing with microprocessor.
CO5. Compare different features of advance microprocessors.
CO6. Use assembly language to program 8086 for Interfacing.

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PS01	PSO2	PSO3
CO 1	**3**	2	1	1	-	-	-	1	1	-	-	1	-	-	-
CO 2	3	3	1	1	-	-	-	1	1	-	-	1	-	-	-
CO 3	1	1	**3**	2	-	-	-	1	2	-	-	1	-	-	-
CO 4	1	3	1	2	-	-	-	2	2	-	-	1	-	-	-
CO 5	1	-	**3**	-	-	**3**	-	-	1	-	-	1	-	-	-
CO 6	1	1	3	3	2	-	-	1	1	-	-	1	3	-	-

Program Articulation Matrix

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PS01	PSO2	PSO3
3	2	3	3	3	2	2	2	3	2	2	3	3	3	3

Correlation levels 1, 2 or 3 as defined below:
1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

Evaluation Scheme and Rubrics

Course code: 20CP202P Course name: Microprocessor Programming & Interfacing LAB

Course Outcomes (CO's): On completion of the course, students will be able to

CO1. Describe the various features of microprocessor.
CO2. Explain various elements of 8086 microprocessor architecture.
Co3. Select required instructions by considering the addressing modes.
CO4. Analyse different concepts of programmable interfacing with microprocessor.
CO5. Compare different features of advance microprocessors.
CO6. Use assembly language to program 8086 for Interfacing.

Co Assessment Tools (Direct Assessment):

Various assessment tools used to evaluate CO's (Rubrics) and the frequency with which the assessment processes are carried out are listed below.

Name of the School : SOT				
Lab Practical Work/ESE				
Name of the Department : Computer Science and Engineering				
Semester	**Date**	**Day**	**Time**	**Name of the Course & Code**
3				LPW- 20CP202P- Microprocessor Programming & Interfacing LAB (50 Marks)
3				Practical/viva ESE- 20CP202P- Microprocessor Programming & Interfacing LAB (50 Marks)

Note:

1. Marks (earned by the student) should be declared and shown to them by the given date, and then uploaded on TCSiON.

2. All marks of Mid Term and the internal assessment to be shared with the student one week before the commencement of the End Term exam

Introduction to 8086 Microprocessor

8086 Microprocessor is an enhanced version of 8085 Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086

The most prominent features of a 8086 microprocessor are as follows −

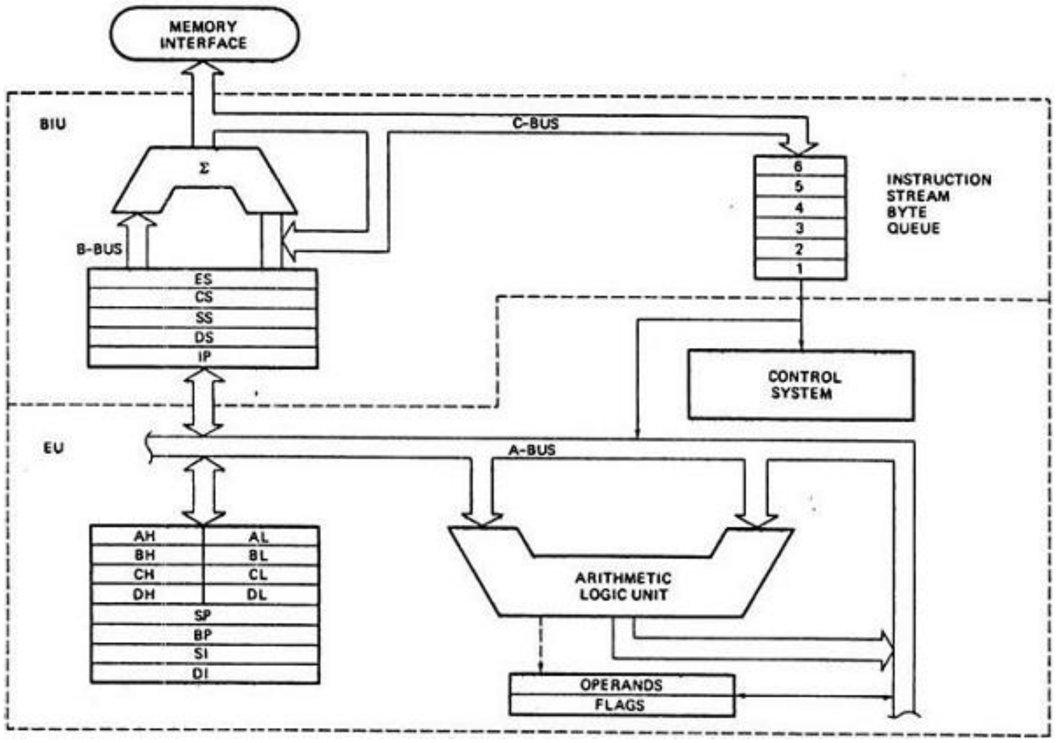
- It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
- It is available in 3 versions based on the frequency of operation −
    - 8086 → 5MHz
    - 8086-2 → 8MHz
    - (c)8086-1 → 10 MHz
- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.
- Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.
- Execute stage executes these instructions.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

Comparison between 8085 & 8086 Microprocessor

- **Size** − 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.
- **Address Bus** − 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- **Memory** − 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- **Instruction** − 8085 doesn’t have an instruction queue, whereas 8086 has an instruction queue.
- **Pipelining** − 8085 doesn’t support a pipelined architecture while 8086 supports a pipelined architecture.
- **I/O** − 8085 can address 2^8 = 256 I/O's, whereas 8086 can access 2^16 = 65,536 I/O's.
- **Cost** − The cost of 8085 is low whereas that of 8086 is high.

Architecture of 8086

The following diagram depicts the architecture of a 8086 Microprocessor −



8086 Microprocessor is divided into two functional units, i.e., **EU** (Execution Unit) and **BIU** (Bus Interface Unit).

EU (Execution Unit)

Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.

Let us now discuss the functional parts of 8086 microprocessors.

ALU

It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.

Flag Register

It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups − Conditional Flags and Control Flags.

Conditional Flags

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags –

- **Carry flag** − This flag indicates an overflow condition for arithmetic operations.
- **Auxiliary flag** − When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.
- **Parity flag** − This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.
- **Zero flag** − This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.
- **Sign flag** − This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.
- **Overflow flag** − This flag represents the result when the system capacity is exceeded.

Control Flags

Control flags controls the operations of the execution unit. Following is the list of control flags −

- **Trap flag** − It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.
- **Interrupt flag** − It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.
- **Direction flag** − It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

General purpose register

There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually to store 8-bit data and can be used in pairs to store 16bit data. The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. It is referred to the AX, BX, CX, and DX respectively.

- **AX register** − It is also known as accumulator register. It is used to store operands for arithmetic operations.
- **BX register** − It is used as a base register. It is used to store the starting base address of the memory area within the data segment.
- **CX register** − It is referred to as counter. It is used in loop instruction to store the loop counter.
- **DX register** − This register is used to hold I/O port address for I/O instruction.

Stack pointer register

It is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack.

BIU (Bus Interface Unit)

BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory. EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

It has the following functional parts −

- **Instruction queue** − BIU contains the instruction queue. BIU gets upto 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.
- Fetching the next instruction while the current instruction executes is called **pipelining**.
- **Segment register** − BIU has 4 segment buses, i.e. CS, DS, SS& ES. It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to executed by the EU.
  - **CS** − It stands for Code Segment. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.
  - **DS** − It stands for Data Segment. It consists of data used by the program andis accessed in the data segment by an offset address or the content of other register that holds the offset address.
  - **SS** − It stands for Stack Segment. It handles memory to store data and addresses during execution.
  - **ES** − It stands for Extra Segment. ES is additional data segment, which is used by the string to hold the extra destination data.
- **Instruction pointer** − It is a 16-bit register used to hold the address of the next instruction to be executed.

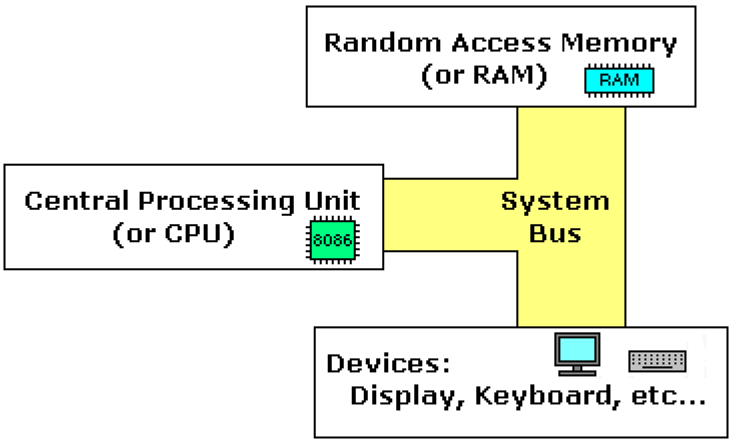
8086 assembler tutorial for beginners

This tutorial is intended for those who are not familiar with assembler at all, or have a very distant idea about it. Of course if you have knowledge of some high level programming language (java, basic, c/c++, pascal...) that may help you a lot.
But even if you are familiar with assembler, it is still a good idea to look through this document in order to study emu8086 syntax.

It is assumed that you have some knowledge about number representation (hex/bin), if not it is highly recommended to study **numbering systems tutorial** before you proceed.

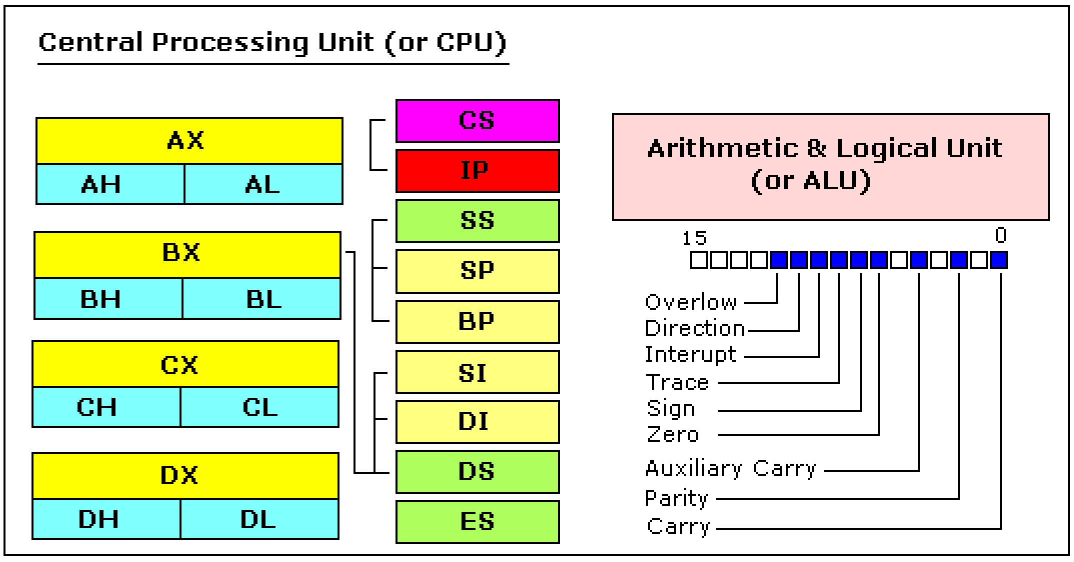
What is assembly language?

Assembly language is a low level programming language. You need to get some knowledge about computer structure in order to understand anything. The simple computer model as I see it:



The **system 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



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.

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bit, it's something like: **0011000000111001b** (in binary form), or **12345** in decimal (human) form.

4 general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= **0011000000111001b**, 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.

Because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special Purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

segment 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.

Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.

Segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, since this way we can access much more memory than with a single register that is limited to 16 bit values. CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h * 10h + 45h = 12345h):

```
 12300
+ 0045
 -----
 12345
```

The address formed with 2 registers is called an **effective address**.
By default **BX, SI** and **DI** registers work with **DS** segment register;
BP and **SP** work with **SS** segment register.
Other general purpose registers cannot form an effective address!
also, although **BX** can form an effective address, **BH** and **BL** cannot.

Special purpose registers

- **IP** - the instruction pointer.
- **flags register** - determines the current state of the microprocessor.

IP register always works together with **CS** segment register and it points to currently executing instruction.
Flags register is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.
Generally you cannot access these registers directly, the way you can access AX and other general registers, but it is possible to change values of system registers using some tricks that you will learn a little bit later.

Instruction SET

The 8086 microprocessor supports 8 types of instructions −

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Let us now discuss these instruction sets in detail.

Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group −

Instruction to transfer a word

- **MOV** − Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** − Used to put a word at the top of the stack.
- **POP** − Used to get a word from the top of the stack to the provided location.
- **PUSHA** − Used to put all the registers into the stack.
- **POPA** − Used to get words from the stack to all registers.
- **XCHG** − Used to exchange the data from two locations.
- **XLAT** − Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- **IN** − Used to read a byte or word from the provided port to the accumulator.
- **OUT** − Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- **LEA** − Used to load the address of operand into the provided register.
- **LDS** − Used to load DS register and other provided register from the memory
- **LES** − Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- **LAHF** − Used to load AH with the low byte of the flag register.
- **SAHF** − Used to store AH register to low byte of the flag register.
- **PUSHF** − Used to copy the flag register at the top of the stack.
- **POPF** − Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Following is the list of instructions under this group −

Instructions to perform addition

- **ADD** − Used to add the provided byte to byte/word to word.

- **ADC** − Used to add with carry.
- **INC** − Used to increment the provided byte/word by 1.
- **AAA** − Used to adjust ASCII after addition.
- **DAA** − Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- **SUB** − Used to subtract the byte from byte/word from word.
- **SBB** − Used to perform subtraction with borrow.
- **DEC** − Used to decrement the provided byte/word by 1.
- **NPG** − Used to negate each bit of the provided byte/word and add 1/2's complement.
- **CMP** − Used to compare 2 provided byte/word.
- **AAS** − Used to adjust ASCII codes after subtraction.
- **DAS** − Used to adjust decimal after subtraction.

Instruction to perform multiplication

- **MUL** − Used to multiply unsigned byte by byte/word by word.
- **IMUL** − Used to multiply signed byte by byte/word by word.
- **AAM** − Used to adjust ASCII codes after multiplication.

Instructions to perform division

- **DIV** − Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** − Used to divide the signed word by byte or signed double word by word.
- **AAD** − Used to adjust ASCII codes after division.
- **CBW** − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** − Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group −

Instructions to perform logical operation

- **NOT** − Used to invert each bit of a byte or word.
- **AND** − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** − Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- **SHL/SAL** − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Instructions to perform rotate operations

- **ROL** − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group −

- **REP** − Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** − Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** − Used to compare two string bytes/words.
- **INS/INSB/INSW** − Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** − Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** − Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −

Instructions to transfer the instruction during an execution without any condition −

- **CALL** − Used to call a procedure and save their return address to the stack.
- **RET** − Used to return from the procedure to the main program.
- **JMP** − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions −

- **JA/JNBE** − Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** − Used to jump if above/not below instruction satisfies.
- **JBE/JNA** − Used to jump if below/equal/ not above instruction satisfies.
- **JC** − Used to jump if carry flag CF = 1
- **JE/JZ** − Used to jump if equal/zero flag ZF = 1
- **JG/JNLE** − Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** − Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** − Used to jump if less than/not greater than/equal instruction satisfies.

- **JLE/JNG** − Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** − Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** − Used to jump if not equal/zero flag ZF = 0
- **JNO** − Used to jump if no overflow flag OF = 0
- **JNP/JPO** − Used to jump if not parity/parity odd PF = 0
- **JNS** − Used to jump if not sign SF = 0
- **JO** − Used to jump if overflow flag OF = 1
- **JP/JPE** − Used to jump if parity/parity even PF = 1
- **JS** − Used to jump if sign flag SF = 1

Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

Following are the instructions under this group −

- **STC** − Used to set carry flag CF to 1
- **CLC** − Used to clear/reset carry flag CF to 0
- **CMC** − Used to put complement at the state of carry flag CF.
- **STD** − Used to set the direction flag DF to 1
- **CLD** − Used to clear/reset the direction flag DF to 0
- **STI** − Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** − Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

- **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** − Used to jump to the provided address if CX = 0

Interrupt Instructions

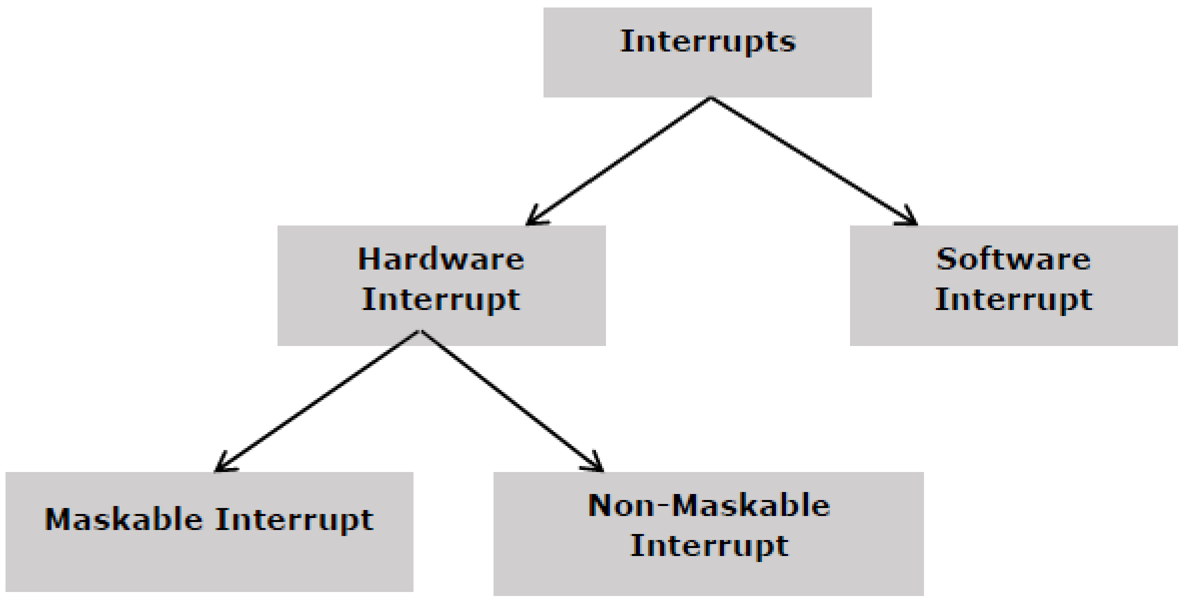
These instructions are used to call the interrupt during program execution.

- **INT** − Used to interrupt the program during execution and calling service specified.
- **INTO** − Used to interrupt the program during execution if OF = 1
- **IRET** − Used to return from interrupt service to the main program

Interrupts

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

The following image shows the types of interrupts we have in a 8086 microprocessor −



Hardware Interrupts

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR)and it is of type 2 interrupt.

When this interrupt is activated, these actions take place −

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0.

INTR

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends ‘0’ on INTA pin twice. The first ‘0’ means INTA informs the external device to get ready and during the second ‘0’ the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor −

- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.
- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes −

INT- Interrupt instruction with type number

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location ‘type number’ × 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e. −

- **TYPE 0** interrupt represents division by zero situation.
- **TYPE 1** interrupt represents single-step execution during the debugging of a program.
- **TYPE 2** interrupt represents non-maskable NMI interrupt.
- **TYPE 3** interrupt represents break-point interrupt.
- **TYPE 4** interrupt represents overflow interrupt.

The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

INT 3-Break Point Interrupt Instruction

It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 3×4 = 0000CH
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

INTO - Interrupt on overflow instruction

It is a 1-byte instruction and their mnemonic **INTO**. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.

Its execution includes the following steps −

- Flag register values are pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of word location 4×4 = 00010H
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0

Addressing Modes

The different ways in which a source operand is denoted in an instruction is known as **addressing modes**. There are 8 different addressing modes in 8086 programming –

Immediate addressing mode

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

Example

MOV CX, 4929 H, ADD AX, 2387 H, MOV AL, FFH

Register addressing mode

It means that the register is the source of an operand for an instruction.

Example

MOV CX, AX ; copies the contents of the 16-bit AX register into
; the 16-bit CX register),
ADD BX, AX

Direct addressing mode

The addressing mode in which the effective address of the memory location is written directly in the instruction.

Example

MOV AX, [1592H], MOV AL, [0300H]

Register indirect addressing mode

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

Example

MOV AX, [BX] ; Suppose the register BX contains 4895H, then the contents
; 4895H are moved to AX
ADD CX, {BX}

Based addressing mode

In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

Example

MOV DX, [BX+04], ADD CL, [BX+08]

Indexed addressing mode

In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements.

Example

MOV BX, [SI+16], ADD AL, [DI+16]

Based-index addressing mode

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

Example

ADD CX, [AX+SI], MOV AX, [AX+DI]

Based indexed with displacement mode

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement.

Example

MOV AX, [BX+DI+08], ADD CX, [BX+SI+16]

Complete 8086 instruction set

Quick reference:

	CMPSB				MOV		
AAA	CMPSW	JAE	JNBE	JPO	MOVSB	RCR	SCASB
AAD	CWD	JB	JNC	JS	MOVSW	REP	SCASW
AAM	DAA	JBE	JNE	JZ	MUL	REPE	SHL
AAS	DAS	JC	JNG	LAHF	NEG	REPNE	SHR
ADC	DEC	JCXZ	JNGE	LDS	NOP	REPNZ	STC
ADD	DIV	JE	JNL	LEA	NOT	REPZ	STD
AND	HLT	JG	JNLE	LES	OR	RET	STI
CALL	IDIV	JGE	JNO	LODSB	OUT	RETF	STOSB
CBW	IMUL	JL	JNP	LODSW	POP	ROL	STOSW
CLC	IN	JLE	JNS	LOOP	POPA	ROR	SUB
CLD	INC	JMP	JNZ	LOOPE	POPF	SAHF	TEST
CLI	INT	JNA	JO	LOOPNE	PUSH	SAL	XCHG
CMC	INTO	JNAE	JP	LOOPNZ	PUSHA	SAR	XLATB
CMP	IRET	JNB	JPE	LOOPZ	PUSHF	SBB	XOR
	JA				RCL		

Operand types:

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

SREG: DS, ES, SS, and only as second operand: CS.

memory: [BX], [BX+SI+7], variable, etc...(see **Memory Access**).

immediate: 5, -24, 3Fh, 10001101b, etc...

Notes:

- When two operands are required for an instruction they are separated by comma. For example:

  ```
  REG, memory
  ```

- When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:

  ```
  AL, DL
  DX, AX
  ```

m1 DB ?
AL, m1
m2 DW ?
AX, m2

- Some instructions allow several operand combinations. For example:

  memory, immediate
  REG, immediate

  memory, REG
  REG, SREG

- Some examples contain macros, so it is advisable to use **Shift + F8** hot key to *Step Over* (to make macro code execute at maximum speed set **step delay** to zero), otherwise emulator will step through each instruction of a macro. Here is an example that uses PRINTN macro:

```
include 'emu8086.inc'
ORG 100h
MOV AL, 1
MOV BL, 2
PRINTN 'Hello World!'   ; macro.
MOV CL, 3
PRINTN 'Welcome!'       ; macro.
RET
```

These marks are used to show the state of the flags:

1 - instruction sets this flag to **1**.
0 - instruction sets this flag to **0**.
r - flag value depends on result of the instruction.
? - flag value is undefined (maybe **1** or **0**).

Some instructions generate exactly the same machine code, so disassembler may have a problem decoding to your original code. This is especially important for Conditional Jump instructions (see "Program Flow Control" in Tutorials for more information).

Instructions in alphabetical order:

Instruction	Operands	Description
AAA	No operands	ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values. It works according to the following Algorithm: if low nibble of AL > 9 or AF = 1 then: • AL = AL + 6 • AH = AH + 1 • AF = 1 • CF = 1 else • AF = 0 • CF = 0 in both cases: clear the high nibble of AL. Example: MOV AX, 15 ; AH = 00, AL = 0Fh AAA ; AH = 01, AL = 05 RET C Z S O P A r ? ? ? ? r
AAD	No operands	ASCII Adjust before Division. Prepares two BCD values for division. Algorithm: • AL = (AH * 10) + AL • AH = 0 Example: MOV AX, 0105h ; AH = 01, AL = 05

		AAD ; AH = 00, AL = 0Fh (15) RET C Z S O P A ? r r ? r ?
AAM	No operands	ASCII Adjust after Multiplication. Corrects the result of multiplication of two BCD values. Algorithm: • AH = AL / 10 • AL = remainder Example: MOV AL, 15 ; AL = 0Fh AAM ; AH = 01, AL = 05 RET C Z S O P A ? r r ? r ?
AAS	No operands	ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values. Algorithm: if low nibble of AL > 9 or AF = 1 then: • AL = AL - 6 • AH = AH - 1 • AF = 1 • CF = 1 else • AF = 0 • CF = 0 in both cases: clear the high nibble of AL.

		Example: MOV AX, 02FFh ; AH = 02, AL = 0FFh AAS ; AH = 01, AL = 09 RET | C | Z | S | O | P | A | | r | ? | ? | ? | ? | r |
ADC	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Add with Carry. Algorithm: operand1 = operand1 + operand2 + CF Example: STC ; set CF = 1 MOV AL, 5 ; AL = 5 ADC AL, 1 ; AL = 7 RET | C | Z | S | O | P | A | | r | r | r | r | r | r |
ADD	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Add. Algorithm: operand1 = operand1 + operand2 Example: MOV AL, 5 ; AL = 5 ADD AL, -3 ; AL = 2 RET | C | Z | S | O | P | A | | r | r | r | r | r | r |
AND	REG, memory	Logical AND between all bits of two operands. Result is stored in operand1.

	memory, REG REG, REG memory, immediate REG, immediate	These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Example: MOV AL, 'a' ; AL = 01100001b AND AL, 11011111b ; AL = 01000001b ('A') RET C Z S O P 0 r r 0 r
CALL	procedure name label 4-byte address	Transfers control to procedure. Return address (IP) is pushed to stack. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset. If it's a far call, then code segment is pushed to stack as well. Example: ORG 100h ; for COM file. CALL p1 ADD AX, 1 RET ; return to OS. p1 PROC ; procedure declaration. MOV AX, 1234h RET ; return to caller. p1 ENDP C Z S O P A unchanged
CBW	No operands	Convert byte into word.

		Algorithm: if high bit of AL = 1 then: • AH = 255 (0FFh) else • AH = 0 Example: MOV AX, 0 ; AH = 0, AL = 0 MOV AL, -5 ; AX = 000FBh (251) CBW ; AX = 0FFFBh (-5) RET C Z S O P A unchanged
CLC	No operands	Clear Carry flag. Algorithm: CF = 0 C 0
CLD	No operands	Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 0 D 0

CLI	No operands	Clear Interrupt enable flag. This disables hardware interrupts. Algorithm: IF = 0 I 0
CMC	No operands	Complement Carry flag. Inverts value of CF. Algorithm: if CF = 1 then CF = 0 if CF = 0 then CF = 1 C r
CMP	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Compare. Algorithm: operand1 - operand2 result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result. Example: MOV AL, 5 MOV BL, 5 CMP AL, BL ; AL = 5, ZF = 1 (so equal!) RET C Z S O P A r r r r r r

CMPSB	No operands	Compare bytes: ES:[DI] from DS:[SI]. Algorithm: • DS:[SI] - ES:[DI] • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ SI = SI + 1 ○ DI = DI + 1 else ○ SI = SI - 1 ○ DI = DI - 1 Example: open **cmpsb.asm** from c:\emu8086\examples C Z S O P A r r r r r r
CMPSW	No operands	Compare words: ES:[DI] from DS:[SI]. Algorithm: • DS:[SI] - ES:[DI] • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ SI = SI + 2 ○ DI = DI + 2 else ○ SI = SI - 2 ○ DI = DI - 2 example: open **cmpsw.asm** from c:\emu8086\examples C Z S O P A r r r r r r

CWD	No operands	Convert Word to Double word. Algorithm: if high bit of AX = 1 then: • DX = 65535 (0FFFFh) else • DX = 0 Example: MOV DX, 0 ; DX = 0 MOV AX, 0 ; AX = 0 MOV AX, -5 ; DX AX = 00000h:0FFFBh CWD ; DX AX = 0FFFFh:0FFFBh RET C Z S O P A unchanged

DAA	No operands	Decimal adjust After Addition.
		Corrects the result of addition of two packed BCD values.
		
		Algorithm:
		
		if low nibble of AL > 9 or AF = 1 then:
		
		• AL = AL + 6
		• AF = 1
		
		if AL > 9Fh or CF = 1 then:
		
		• AL = AL + 60h
		• CF = 1
		
		Example:
		MOV AL, 0Fh ; AL = 0Fh (15)
		DAA ; AL = 15h
		RET
		
		C Z S O P A
		r r r r r r

DAS	No operands	Decimal adjust After Subtraction.
		Corrects the result of subtraction of two packed BCD values.
		
		Algorithm:
		
		if low nibble of AL > 9 or AF = 1 then:
		
		• AL = AL - 6
		• AF = 1
		
		if AL > 9Fh or CF = 1 then:
		
		• AL = AL - 60h
		• CF = 1
		
		Example:
		MOV AL, 0FFh ; AL = 0FFh (-1)
		DAS ; AL = 99h, CF = 1
		RET
		
		C | Z | S | O | P | A
		r | r | r | r | r | r

DEC	REG memory	Decrement. Algorithm: operand = operand - 1 Example: MOV AL, 255 ; AL = 0FFh (255 or -1) DEC AL ; AL = 0FEh (254 or -2) RET Z | S | O | P | A r | r | r | r | r CF - unchanged!

DIV	REG memory	Unsigned divide. Algorithm: when operand is a **byte**: AL = AX / operand AH = remainder (modulus) when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus) Example: MOV AX, 203 ; AX = 00CBh MOV BL, 4 DIV BL ; AL = 50 (32h), AH = 3 RET C Z S O P A ? ? ? ? ? ?
HLT	No operands	Halt the System. Example: MOV AX, 5 HLT C Z S O P A unchanged
IDIV	REG memory	Signed divide. Algorithm: when operand is a **byte**: AL = AX / operand AH = remainder (modulus) when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus) Example: MOV AX, -203 ; AX = 0FF35h MOV BL, 4 IDIV BL ; AL = -50 (0CEh), AH = -3 (0FDh) RET

		C Z S O P A ? ? ? ? ? ?
IMUL	REG memory	Signed multiply. Algorithm: when operand is a **byte**: AX = AL * operand. when operand is a **word**: (DX AX) = AX * operand. Example: MOV AL, -2 MOV BL, -4 IMUL BL ; AX = 8 RET C Z S O P A r ? ? r ? ? CF=OF=0 when result fits into operand of IMUL.
IN	AL, im.byte AL, DX AX, im.byte AX, DX	Input from port into **AL** or **AX**. Second operand is a port number. If required to access port number over 255 - **DX** register should be used. Example: IN AX, 4 ; get status of traffic lights. IN AL, 7 ; get status of stepper-motor. C Z S O P A unchanged
INC	REG memory	Increment. Algorithm: operand = operand + 1 Example: MOV AL, 4 INC AL ; AL = 5 RET

		Z S O P A r r r r r CF - unchanged!
INT	immediate byte	Interrupt numbered by immediate byte (0..255). Algorithm: Push to stack: ◦ flags register ◦ CS ◦ IP • IF = 0 • Transfer control to interrupt procedure Example: MOV AH, 0Eh ; teletype. MOV AL, 'A' INT 10h ; BIOS interrupt. RET C Z S O P A I unchanged 0
INTO	No operands	Interrupt 4 if Overflow flag is 1. Algorithm: if OF = 1 then INT 4 Example: ; -5 - 127 = -132 (not in -128..127) ; the result of SUB is wrong (124), ; so OF = 1 is set: MOV AL, -5 SUB AL, 127 ; AL = 7Ch (124) INTO ; process error. RET

IRET	No operands	Interrupt Return. Algorithm: Pop from stack: ○ IP ○ CS ○ flags register | C | Z | S | O | P | A | popped

JA	label	Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if (CF = 0) and (ZF = 0) then jump
		Example:
		  include 'emu8086.inc'
		  ORG 100h
		  MOV AL, 250
		  CMP AL, 5
		  JA label1
		  PRINT 'AL is not above 5'
		  JMP exit
		label1:
		  PRINT 'AL is above 5'
		exit:
		  RET
		| C | Z | S | O | P | A |
		unchanged

JAE	label	Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 0 then jump
		Example:
		  include 'emu8086.inc'
		  ORG 100h

		MOV AL, 5 CMP AL, 5 JAE label1 PRINT 'AL is not above or equal to 5' JMP exit label1: PRINT 'AL is above or equal to 5' exit: RET C | Z | S | O | P | A unchanged
JB	label	Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 1 CMP AL, 5 JB label1 PRINT 'AL is not below 5' JMP exit label1: PRINT 'AL is below 5' exit: RET C | Z | S | O | P | A unchanged
JBE	label	Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 or ZF = 1 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 5

		CMP AL, 5 JBE label1 PRINT 'AL is not below or equal to 5' JMP exit label1: PRINT 'AL is below or equal to 5' exit: RET C Z S O P A unchanged
JC	label	Short Jump if Carry flag is set to 1. Algorithm: if CF = 1 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 255 ADD AL, 1 JC label1 PRINT 'no carry.' JMP exit label1: PRINT 'has carry.' exit: RET C Z S O P A unchanged
JCXZ	label	Short Jump if CX register is 0. Algorithm: if CX = 0 then jump Example: include 'emu8086.inc' ORG 100h MOV CX, 0 JCXZ label1 PRINT 'CX is not zero.' JMP exit

		label1: PRINT 'CX is zero.' exit: RET C Z S O P A unchanged

JE	label	Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned.
		
		Algorithm:
		
		          if ZF = 1 then jump
		Example:
		   include 'emu8086.inc'
		   ORG 100h
		   MOV AL, 5
		   CMP AL, 5
		   JE label1
		   PRINT 'AL is not equal to 5.'
		   JMP exit
		label1:
		   PRINT 'AL is equal to 5.'
		exit:
		   RET
		C Z S O P A
		unchanged

JG	label	Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		          if (ZF = 0) and (SF = OF) then jump
		Example:
		   include 'emu8086.inc'
		   ORG 100h
		   MOV AL, 5
		   CMP AL, -5
		   JG label1
		   PRINT 'AL is not greater -5.'
		   JMP exit
		label1:

		PRINT 'AL is greater -5.' exit: RET C | Z | S | O | P | A unchanged

JGE	label	Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		        if SF = OF then jump
		Example:
		  include 'emu8086.inc'
		  ORG 100h
		  MOV AL, 2
		  CMP AL, -5
		  JGE label1
		  PRINT 'AL < -5'
		  JMP exit
		label1:
		  PRINT 'AL >= -5'
		exit:
		  RET
		
		C | Z | S | O | P | A
		unchanged

JL	label	Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		        if SF <> OF then jump
		Example:
		  include 'emu8086.inc'
		  ORG 100h
		  MOV AL, -2
		  CMP AL, 5
		  JL label1
		  PRINT 'AL >= 5.'
		  JMP exit
		label1:
		  PRINT 'AL < 5.'

		exit: RET C Z S O P A unchanged

JLE	label	Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		        if SF <> OF or ZF = 1 then jump
		Example:
		  include 'emu8086.inc'
		  ORG 100h
		  MOV AL, -2
		  CMP AL, 5
		  JLE label1
		  PRINT 'AL > 5.'
		  JMP exit
		label1:
		  PRINT 'AL <= 5.'
		exit:
		  RET
		C Z S O P A
		unchanged

JMP	label
	4-byte
	address
		Unconditional Jump. Transfers control to another part of the program. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.
		
		Algorithm:
		
		        always jump
		Example:
		  include 'emu8086.inc'
		  ORG 100h
		  MOV AL, 5
		  JMP label1 ; jump over 2 lines!
		  PRINT 'Not Jumped!'
		  MOV AL, 0
		label1:

		PRINT 'Got Here!' RET C | Z | S | O | P | A unchanged

JNA	label	Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		        if CF = 1 or ZF = 1 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 2
		  CMP AL, 5
		  JNA label1
		  PRINT 'AL is above 5.'
		  JMP exit
		label1:
		  PRINT 'AL is not above 5.'
		exit:
		  RET
		C | Z | S | O | P | A
		unchanged

JNAE	label	Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		        if CF = 1 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 2
		  CMP AL, 5
		  JNAE label1
		  PRINT 'AL >= 5.'
		  JMP exit
		label1:

		```
		   PRINT 'AL < 5.'
		exit:
		   RET
		```
		 C | Z | S | O | P | A — unchanged

JNB	label	Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 0 then jump
		Example:
		```
		   include 'emu8086.inc'

		   ORG 100h
		   MOV AL, 7
		   CMP AL, 5
		   JNB label1
		   PRINT 'AL < 5.'
		   JMP exit
		label1:
		   PRINT 'AL >= 5.'
		exit:
		   RET
		```
		 C | Z | S | O | P | A — unchanged

JNBE	label	Short Jump if first operand is Not Below and Not Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if (CF = 0) and (ZF = 0) then jump
		Example:
		```
		   include 'emu8086.inc'

		   ORG 100h
		   MOV AL, 7
		   CMP AL, 5
		   JNBE label1
		   PRINT 'AL <= 5.'
		   JMP exit
		```

		label1: PRINT 'AL > 5.' exit: RET C Z S O P A unchanged

JNC	label	Short Jump if Carry flag is set to 0.
		
		Algorithm:
		
		        if CF = 0 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 2
		  ADD AL, 3
		  JNC label1
		  PRINT 'has carry.'
		  JMP exit
		label1:
		  PRINT 'no carry.'
		exit:
		  RET
		C Z S O P A
		unchanged

JNE	label	Short Jump if first operand is Not Equal to second operand (as set by CMP instruction). Signed/Unsigned.
		
		Algorithm:
		
		        if ZF = 0 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 2
		  CMP AL, 3
		  JNE label1
		  PRINT 'AL = 3.'
		  JMP exit

		label1: PRINT 'Al <> 3.' exit: RET | C | Z | S | O | P | A | unchanged

JNG	label	Short Jump if first operand is Not Greater then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if (ZF = 1) and (SF <> OF) then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 2
		  CMP AL, 3
		  JNG label1
		  PRINT 'AL > 3.'
		  JMP exit
		label1:
		  PRINT 'Al <= 3.'
		exit:
		  RET
		
		| C | Z | S | O | P | A |
		unchanged

JNGE	label	Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if SF <> OF then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 2
		  CMP AL, 3
		  JNGE label1
		  PRINT 'AL >= 3.'

		JMP exit label1: PRINT 'Al < 3.' exit: RET C Z S O P A unchanged
JNL	label	Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed. Algorithm: if SF = OF then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, -3 JNL label1 PRINT 'AL < -3.' JMP exit label1: PRINT 'Al >= -3.' exit: RET C Z S O P A unchanged
JNLE	label	Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed. Algorithm: if (SF = OF) and (ZF = 0) then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, -3 JNLE label1

		PRINT 'AL <= -3.' JMP exit label1: PRINT 'Al > -3.' exit: RET C | Z | S | O | P | A unchanged
JNO	label	Short Jump if Not Overflow. Algorithm: if OF = 0 then jump Example: ; -5 - 2 = -7 (inside -128..127) ; the result of SUB is correct, ; so OF = 0: include 'emu8086.inc' ORG 100h MOV AL, -5 SUB AL, 2 ; AL = 0F9h (-7) JNO label1 PRINT 'overflow!' JMP exit label1: PRINT 'no overflow.' exit: RET C | Z | S | O | P | A unchanged
JNP	label	Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 0 then jump Example: include 'emu8086.inc'

		ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNP label1 PRINT 'parity even.' JMP exit label1: PRINT 'parity odd.' exit: RET C Z S O P A unchanged
JNS	label	Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if SF = 0 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNS label1 PRINT 'signed.' JMP exit label1: PRINT 'not signed.' exit: RET C Z S O P A unchanged
JNZ	label	Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if ZF = 0 then jump Example:

		include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 00000111b ; AL = 7
		  OR AL, 0 ; just set flags.
		  JNZ label1
		  PRINT 'zero.'
		  JMP exit
		label1:
		  PRINT 'not zero.'
		exit:
		  RET
		C Z S O P A
		unchanged

JO	label	Short Jump if Overflow. Algorithm: if OF = 1 then jump Example: ; -5 - 127 = -132 (not in -128..127) ; the result of SUB is wrong (124), ; so OF = 1 is set: include 'emu8086.inc' org 100h MOV AL, -5 SUB AL, 127 ; AL = 7Ch (124) JO label1 PRINT 'no overflow.' JMP exit label1: PRINT 'overflow!' exit: RET C Z S O P A unchanged
JP	label	Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

		Algorithm:
		
		if PF = 1 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 00000101b ; AL = 5
		  OR AL, 0 ; just set flags.
		  JP label1
		  PRINT 'parity odd.'
		  JMP exit
		label1:
		  PRINT 'parity even.'
		exit:
		  RET
		C Z S O P A
		unchanged

JPE	label	Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		if PF = 1 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 00000101b ; AL = 5
		  OR AL, 0 ; just set flags.
		  JPE label1
		  PRINT 'parity odd.'
		  JMP exit
		label1:
		  PRINT 'parity even.'
		exit:
		  RET
		C Z S O P A
		unchanged

JPO	label	Short Jump if Parity Odd. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		        if PF = 0 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 00000111b ; AL = 7
		  OR AL, 0 ; just set flags.
		  JPO label1
		  PRINT 'parity even.'
		  JMP exit
		label1:
		  PRINT 'parity odd.'
		exit:
		  RET
		
		C | Z | S | O | P | A
		unchanged

JS	label	Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		        if SF = 1 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 10000000b ; AL = -128
		  OR AL, 0 ; just set flags.
		  JS label1
		  PRINT 'not signed.'
		  JMP exit
		label1:
		  PRINT 'signed.'
		exit:
		  RET
		
		C | Z | S | O | P | A
		unchanged

JZ	label	Short Jump if Zero (equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		        if ZF = 1 then jump
		Example:
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AL, 5
		  CMP AL, 5
		  JZ label1
		  PRINT 'AL is not equal to 5.'
		  JMP exit
		label1:
		  PRINT 'AL is equal to 5.'
		exit:
		  RET
		C Z S O P A
		unchanged

LAHF	No operands	Load AH from 8 low bits of Flags register. Algorithm: AH = flags register AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. C Z S O P A unchanged
LDS	REG, memory	Load memory double word into word register and DS. Algorithm: • REG = first word

		• DS = second word Example: ORG 100h LDS AX, m RET m DW 1234h DW 5678h END AX is set to 1234h, DS is set to 5678h. C Z S O P A unchanged
LEA	REG, memory	Load Effective Address. Algorithm: • REG = address of memory (offset) Example: MOV BX, 35h MOV DI, 12h LEA SI, [BX+DI] ; SI = 35h + 12h = 47h Note: The integrated 8086 assembler automatically replaces **LEA** with a more efficient **MOV** where possible. For example: org 100h LEA AX, m ; AX = offset of m RET m dw 1234h END

		C | Z | S | O | P | A unchanged
LES	REG, memory	Load memory double word into word register and ES. Algorithm: • REG = first word • ES = second word Example: ORG 100h LES AX, m RET m DW 1234h DW 5678h END AX is set to 1234h, ES is set to 5678h. C | Z | S | O | P | A unchanged
LODSB	No operands	Load byte at DS:[SI] into AL. Update SI. Algorithm: • AL = DS:[SI] • if DF = 0 then ○ SI = SI + 1 else ○ SI = SI - 1

		Example: ORG 100h LEA SI, a1 MOV CX, 5 MOV AH, 0Eh m: LODSB INT 10h LOOP m RET a1 DB 'H', 'e', 'l', 'l', 'o' C | Z | S | O | P | A unchanged
LODSW	No operands	Load word at DS:[SI] into AX. Update SI. Algorithm: • AX = DS:[SI] • if DF = 0 then ○ SI = SI + 2 else ○ SI = SI - 2 Example: ORG 100h LEA SI, a1 MOV CX, 5 REP LODSW ; finally there will be 555h in AX. RET a1 dw 111h, 222h, 333h, 444h, 555h C | Z | S | O | P | A unchanged

		⬆
LOOP	label	Decrease CX, jump to label if CX not zero. Algorithm: • CX = CX - 1 • if CX <> 0 then ○ jump else ○ no jump, continue Example: include 'emu8086.inc' ORG 100h MOV CX, 5 label1: PRINTN 'loop!' LOOP label1 RET C Z S O P A unchanged ⬆
LOOPE	label	Decrease CX, jump to label if CX not zero and Equal (ZF = 1). Algorithm: • CX = CX - 1 • if (CX <> 0) and (ZF = 1) then ○ jump else ○ no jump, continue Example: ; Loop until result fits into AL alone, ; or 5 times. The result will be over 255 ; on third loop (100+100+100), ; so loop will exit.

		include 'emu8086.inc' ORG 100h MOV AX, 0 MOV CX, 5 label1: PUTC '*' ADD AX, 100 CMP AH, 0 LOOPE label1 RET C Z S O P A unchanged
LOOPNE	label	Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0). Algorithm: • CX = CX - 1 • if (CX <> 0) and (ZF = 0) then ○ jump else ○ no jump, continue Example: ; Loop until '7' is found, ; or 5 times. include 'emu8086.inc' ORG 100h MOV SI, 0 MOV CX, 5 label1: PUTC '*' MOV AL, v1[SI] INC SI ; next byte (SI=SI+1). CMP AL, 7 LOOPNE label1 RET v1 db 9, 8, 7, 6, 5

		C Z S O P A unchanged

LOOPNZ	label	Decrease CX, jump to label if CX not zero and ZF = 0.
		
		Algorithm:
		
		• CX = CX - 1
		• if (CX <> 0) and (ZF = 0) then
		    ○ jump
		
		  else
		
		    ○ no jump, continue
		
		Example:
		; Loop until '7' is found,
		; or 5 times.
		
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV SI, 0
		  MOV CX, 5
		label1:
		  PUTC '*'
		  MOV AL, v1[SI]
		  INC SI ; next byte (SI=SI+1).
		  CMP AL, 7
		  LOOPNZ label1
		  RET
		  v1 db 9, 8, 7, 6, 5
		C Z S O P A
		unchanged

LOOPZ	label	Decrease CX, jump to label if CX not zero and ZF = 1. Algorithm: • CX = CX - 1 • if (CX <> 0) and (ZF = 1) then

		◦ jump
		
		else
		
		◦ no jump, continue
		
		Example:
		; Loop until result fits into AL alone,
		; or 5 times. The result will be over 255
		; on third loop (100+100+100),
		; so loop will exit.
		
		  include 'emu8086.inc'
		
		  ORG 100h
		  MOV AX, 0
		  MOV CX, 5
		label1:
		  PUTC '*'
		  ADD AX, 100
		  CMP AH, 0
		  LOOPZ label1
		  RET
		
		C Z S O P A
		unchanged

MOV	REG, memory memory, REG REG, REG memory, immediate REG, immediate SREG, memory memory, SREG REG, SREG SREG, REG	Copy operand2 to operand1. The MOV instruction <u>cannot</u>: • set the value of the CS and IP registers. • copy value of one segment register to another segment register (should copy to general register first). • copy immediate value to segment register (should copy to general register first). Algorithm: operand1 = operand2 Example: ORG 100h MOV AX, 0B800h ; set AX = B800h (VGA memory).

		MOV DS, AX ; copy value of AX to DS. MOV CL, 'A' ; CL = 41h (ASCII code). MOV CH, 01011111b ; CL = color attribute. MOV BX, 15Eh ; BX = position on screen. MOV [BX], CX ; w.[0B800h:015Eh] = CX. RET ; returns to operating system. C Z S O P A unchanged
MOVSB	No operands	Copy byte at DS:[SI] to ES:[DI]. Update SI and DI. Algorithm: • ES:[DI] = DS:[SI] • if DF = 0 then ○ SI = SI + 1 ○ DI = DI + 1 else ○ SI = SI - 1 ○ DI = DI - 1 Example: ORG 100h CLD LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSB RET a1 DB 1,2,3,4,5 a2 DB 5 DUP(0) C Z S O P A unchanged

MOVSW	No operands	Copy **word** at DS:[SI] to ES:[DI]. Update SI and DI. Algorithm: • ES:[DI] = DS:[SI] • if DF = 0 then ○ SI = SI + 2 ○ DI = DI + 2 else ○ SI = SI - 2 ○ DI = DI - 2 Example: ORG 100h CLD LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSW RET a1 DW 1,2,3,4,5 a2 DW 5 DUP(0) C Z S O P A unchanged
MUL	REG memory	Unsigned multiply. Algorithm: when operand is a **byte**: AX = AL * operand. when operand is a **word**: (DX AX) = AX * operand. Example: MOV AL, 200 ; AL = 0C8h MOV BL, 4 MUL BL ; AX = 0320h (800) RET

		C Z S O P A r ? ? r ? ? CF=OF=0 when high section of the result is zero.
NEG	REG memory	Negate. Makes operand negative (two's complement). Algorithm: • Invert all bits of the operand • Add 1 to inverted operand Example: MOV AL, 5 ; AL = 05h NEG AL ; AL = 0FBh (-5) NEG AL ; AL = 05h (5) RET C Z S O P A r r r r r r
NOP	No operands	No Operation. Algorithm: • Do nothing Example: ; do nothing, 3 times: NOP NOP NOP RET C Z S O P A unchanged
NOT	REG memory	Invert each bit of the operand. Algorithm: • if bit is 1 turn it to 0.

		• if bit is 0 turn it to 1. Example: MOV AL, 00011011b NOT AL ; AL = 11100100b RET C Z S O P A unchanged
OR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical OR between all bits of two operands. Result is stored in first operand. These rules apply: 1 OR 1 = 1 1 OR 0 = 1 0 OR 1 = 1 0 OR 0 = 0 Example: MOV AL, 'A' ; AL = 01000001b OR AL, 00100000b ; AL = 01100001b ('a') RET C Z S O P A 0 r r 0 r ?
OUT	im.byte, AL im.byte, AX DX, AL DX, AX	Output from **AL** or **AX** to port. First operand is a port number. If required to access port number over 255 - **DX** register should be used. Example: MOV AX, 0FFFh ; Turn on all OUT 4, AX ; traffic lights. MOV AL, 100b ; Turn on the third OUT 7, AL ; magnet of the stepper-motor. C Z S O P A unchanged

POP	REG SREG memory	Get 16 bit value from the stack. Algorithm: • operand = SS:[SP] (top of the stack) • SP = SP + 2 Example: MOV AX, 1234h PUSH AX POP DX ; DX = 1234h RET C | Z | S | O | P | A unchanged
POPA	No operands	Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack. SP value is ignored, it is Popped but not set to SP register). Note: this instruction works only on **80186** CPU and later! Algorithm: • POP DI • POP SI • POP BP • POP xx (SP value ignored) • POP BX • POP DX • POP CX • POP AX C | Z | S | O | P | A unchanged
POPF	No operands	Get flags register from the stack. Algorithm: • flags = SS:[SP] (top of the stack)

		• SP = SP + 2 C Z S O P A popped
PUSH	REG SREG memory immediate	Store 16 bit value in the stack. Note: **PUSH immediate** works only on 80186 CPU and later! Algorithm: • SP = SP - 2 • SS:[SP] (top of the stack) = operand Example: MOV AX, 1234h PUSH AX POP DX ; DX = 1234h RET C Z S O P A unchanged
PUSHA	No operands	Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used. Note: this instruction works only on **80186** CPU and later! Algorithm: • PUSH AX • PUSH CX • PUSH DX • PUSH BX • PUSH SP • PUSH BP • PUSH SI • PUSH DI C Z S O P A

		unchanged
PUSHF	No operands	Store flags register in the stack. Algorithm: • SP = SP - 2 • SS:[SP] (top of the stack) = flags C | Z | S | O | P | A unchanged
RCL	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 left through Carry Flag. The number of rotates is set by operand2. When **immediate** is greater then 1, assembler generates several **RCL xx, 1** instructions because 8086 has machine code only for this instruction (the same principle works for all other shift/rotate instructions). Algorithm: shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position. Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCL AL, 1 ; AL = 00111001b, CF=0. RET C | O r | r OF=0 if first operand keeps original sign.
RCR	memory, immediate REG, immediate memory,	Rotate operand1 right through Carry Flag. The number of rotates is set by operand2. Algorithm:

	CL
	REG, CL
		shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position.
		
		Example:
		`STC ; set carry (CF=1).`
		`MOV AL, 1Ch ; AL = 00011100b`
		`RCR AL, 1 ; AL = 10001110b, CF=0.`
		`RET`
		C O
		r r
		OF=0 if first operand keeps original sign.

REP	chain instruction	Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • go back to check_cx else • exit from REP cycle Z r
REPE	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times. Algorithm: check_cx: if CX <> 0 then

		• do following chain instruction • CX = CX - 1 • if ZF = 1 then: ○ go back to check_cx else ○ exit from REPE cycle else • exit from REPE cycle example: open **cmpsb.asm** from c:\emu8086\examples Z r
REPNE	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 0 then: ○ go back to check_cx else ○ exit from REPNE cycle else • exit from REPNE cycle Z r

REPNZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 0 then: ○ go back to check_cx else ○ exit from REPNZ cycle else • exit from REPNZ cycle Z r
REPZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Zero), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 1 then: ○ go back to check_cx

		else
		  ◦ exit from REPZ cycle
		else
		• exit from REPZ cycle
		
		Z
		r

RET	No operands or even immediate	Return from near procedure. Algorithm: • Pop from stack: ◦ IP • if immediate operand is present: SP = SP + operand Example: ORG 100h ; for COM file. CALL p1 ADD AX, 1 RET ; return to OS. p1 PROC ; procedure declaration. MOV AX, 1234h RET ; return to caller. p1 ENDP C Z S O P A unchanged
RETF	No operands or even immediate	Return from Far procedure. Algorithm: • Pop from stack: ◦ IP

		○ CS • if immediate operand is present: SP = SP + operand C Z S O P A unchanged
ROL	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 left. The number of rotates is set by operand2. Algorithm: shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position. Example: MOV AL, 1Ch ; AL = 00011100b ROL AL, 1 ; AL = 00111000b, CF=0. RET C O r r OF=0 if first operand keeps original sign.
ROR	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 right. The number of rotates is set by operand2. Algorithm: shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position. Example: MOV AL, 1Ch ; AL = 00011100b ROR AL, 1 ; AL = 00001110b, CF=0. RET C O r r OF=0 if first operand keeps original sign.
SAHF	No operands	Store AH register into low 8 bits of Flags register. Algorithm:

		flags register = AH AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. C Z S O P A r r r r r r
SAL	memory, immediate REG, immediate memory, CL REG, CL	Shift Arithmetic operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 0E0h ; AL = 11100000b SAL AL, 1 ; AL = 11000000b, CF=1. RET C O r r OF=0 if first operand keeps original sign.
SAR	memory, immediate REG, immediate memory, CL REG, CL	Shift Arithmetic operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • The sign bit that is inserted to the left-most position has the same value as before shift. Example: MOV AL, 0E0h ; AL = 11100000b SAR AL, 1 ; AL = 11110000b, CF=0. MOV BL, 4Ch ; BL = 01001100b SAR BL, 1 ; BL = 00100110b, CF=0. RET C O

		r r OF=0 if first operand keeps original sign.
SBB	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Subtract with Borrow. Algorithm: operand1 = operand1 - operand2 - CF Example: STC MOV AL, 5 SBB AL, 3 ; AL = 5 - 3 - 1 = 1 RET C Z S O P A r r r r r r
SCASB	No operands	Compare bytes: AL from ES:[DI]. Algorithm: • AL - ES:[DI] • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ DI = DI + 1 else ○ DI = DI - 1 C Z S O P A r r r r r r
SCASW	No operands	Compare words: AX from ES:[DI]. Algorithm: • AX - ES:[DI]

		• set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ DI = DI + 2 else ○ DI = DI - 2 | C | Z | S | O | P | A | | r | r | r | r | r | r |
SHL	memory, immediate REG, immediate memory, CL REG, CL	Shift operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 11100000b SHL AL, 1 ; AL = 11000000b, CF=1. RET | C | O | | r | r | OF=0 if first operand keeps original sign.
SHR	memory, immediate REG, immediate memory, CL REG, CL	Shift operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • Zero bit is inserted to the left-most position. Example: MOV AL, 00000111b SHR AL, 1 ; AL = 00000011b, CF=1. RET | C | O |

		r r OF=0 if first operand keeps original sign.
STC	No operands	Set Carry flag. Algorithm: CF = 1 C 1
STD	No operands	Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 1 D 1
STI	No operands	Set Interrupt enable flag. This enables hardware interrupts. Algorithm: IF = 1 I 1
STOSB	No operands	Store byte in AL into ES:[DI]. Update DI. Algorithm: • ES:[DI] = AL • if DF = 0 then

		◦ DI = DI + 1 else ◦ DI = DI - 1 Example: ORG 100h LEA DI, a1 MOV AL, 12h MOV CX, 5 REP STOSB RET a1 DB 5 dup(0) C Z S O P A unchanged
STOSW	No operands	Store word in AX into ES:[DI]. Update DI. Algorithm: • ES:[DI] = AX • if DF = 0 then ◦ DI = DI + 2 else ◦ DI = DI - 2 Example: ORG 100h LEA DI, a1 MOV AX, 1234h MOV CX, 5 REP STOSW RET

		a1 DW 5 dup(0) C Z S O P A unchanged
SUB	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Subtract. Algorithm: operand1 = operand1 - operand2 Example: MOV AL, 5 SUB AL, 1 ; AL = 4 RET C Z S O P A r r r r r r
TEST	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical AND between all bits of two operands for flags only. These flags are effected: **ZF, SF, PF.** Result is not stored anywhere. These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Example: MOV AL, 00000101b TEST AL, 1 ; ZF = 0. TEST AL, 10b ; ZF = 1. RET C Z S O P 0 r r 0 r

XCHG	REG, memory memory, REG REG, REG	Exchange values of two operands. Algorithm: operand1 < - > operand2 Example: MOV AL, 5 MOV AH, 2 XCHG AL, AH ; AL = 2, AH = 5 XCHG AL, AH ; AL = 5, AH = 2 RET C Z S O P A unchanged
XLATB	No operands	Translate byte from table. Copy value of memory byte at DS:[BX + unsigned AL] to AL register. Algorithm: AL = DS:[BX + unsigned AL] Example: ORG 100h LEA BX, dat MOV AL, 2 XLATB ; AL = 33h RET dat DB 11h, 22h, 33h, 44h, 55h C Z S O P A unchanged
XOR	REG, memory memory, REG REG, REG memory,	Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand. These rules apply: 1 XOR 1 = 0

	immediate REG, immediate	1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0 Example: MOV AL, 00000111b XOR AL, 00000010b ; AL = 00000101b RET C Z S O P A 0 r r 0 r ?

C	Z	S	O	P	A
0	r	r	0	r	?

Quick reference:

INT 10h/00h
INT 10h/01h
INT 10h/02h
INT 10h/03h
INT 10h/05h
INT 10h/06h
INT 10h/07h
INT 10h/08h
INT 10h/09h
INT 10h/0Ah
INT 10h/0Ch
INT 10h/0Dh
INT 10h/0Eh
INT 10h/13h
INT 10h/1003h
INT 11h
INT 12h
INT 13h/00h
INT 13h/02h
INT 13h/03h
INT 15h/86h
INT 16h/00h
INT 16h/01h
INT 19h
INT 1Ah/00h
INT 20h
INT 21h
INT 21h/01h
INT 21h/02h
INT 21h/05h
INT 21h/06h
INT 21h/07h
INT 21h/09h
INT 21h/0Ah
INT 21h/0Bh
INT 21h/0Ch
INT 21h/0Eh
INT 21h/19h
INT 21h/25h
INT 21h/2Ah
INT 21h/2Ch
INT 21h/35h
INT 21h/39h
INT 21h/3Ah
INT 21h/3Bh
INT 21h/3Ch
INT 21h/3Dh
INT 21h/3Eh
INT 21h/3Fh
INT 21h/40h
INT 21h/41h
INT 21h/42h
INT 21h/47h
INT 21h/4Ch
INT 21h/56h
INT 33h/0000h
INT 33h/0001h
INT 33h/0002h
INT 33h/0003h

---

the short list of supported interrupts with descriptions:

INT 10h / AH = 0 - set video mode.
input:
AL = desired video mode.

these video modes are supported:

00h - text mode. 40x25. 16 colors. 8 pages.

03h - text mode. 80x25. 16 colors. 8 pages.

13h - graphical mode. 40x25. 256 colors. 320x200 pixels. 1 page.
example:

```
mov al, 13h
mov ah, 0
int 10h
```

INT 10h / AH = 01h - set text-mode cursor shape.
input:
CH = cursor start line (bits 0-4) and options (bits 5-7).
CL = bottom cursor line (bits 0-4).

when bit 5 of CH is set to **0**, the cursor is visible. when bit 5 is **1**, the cursor is not visible.

```
; hide blinking text cursor:
        mov ch, 32
        mov ah, 1
        int 10h

; show standard blinking text cursor:
        mov ch, 6
        mov cl, 7
        mov ah, 1
        int 10h

; show box-shaped blinking text cursor:
        mov ch, 0
        mov cl, 7
        mov ah, 1
        int 10h

;       note: some bioses required CL to be >=7,
;       otherwise wrong cursor shapes are displayed.
```

INT 10h / **AH = 2** - set cursor position.
input:
DH = row.
DL = column.
BH = page number (0..7).

example:

```
        mov dh, 10
        mov dl, 20
        mov bh, 0
        mov ah, 2
        int 10h
```

INT 10h / **AH = 03h** - get cursor position and size.
input:
BH = page number.
return:
DH = row.
DL = column.
CH = cursor start line.
CL = cursor bottom line.

INT 10h / **AH = 05h** - select active video page.
input:
AL = new page number (0..7).
the activated page is displayed.

INT 10h / **AH = 06h** - scroll up window.
INT 10h / **AH = 07h** - scroll down window.
input:
AL = number of lines by which to scroll (00h = clear entire window).
BH = attribute used to write blank lines at bottom of window.
CH, CL = row, column of window's upper left corner.
DH, DL = row, column of window's lower right corner.

INT 10h / **AH = 08h** - read character and attribute at cursor position.

input:
BH = page number.
return:
AH = attribute.
AL = character.

INT 10h / **AH = 09h** - write character and attribute at cursor position.

input:
AL = character to display.
BH = page number.
BL = attribute.
CX = number of times to write character.

INT 10h / **AH = 0Ah** - write character only at cursor position.

input:
AL = character to display.
BH = page number.
CX = number of times to write character.

INT 10h / **AH = 0Ch** - change color for a single pixel.

input:
AL = pixel color
CX = column.
DX = row.

example:

```
mov al, 13h
mov ah, 0
int 10h     ; set graphics video mode.
mov al, 1100b
mov cx, 10
mov dx, 20
mov ah, 0ch
int 10h     ; set pixel.
```

INT 10h / **AH = 0Dh** - get color of a single pixel.

input:
CX = column.
DX = row.
output:
AL = pixel color

INT 10h / **AH = 0Eh** - teletype output.

input:
AL = character to write.

this functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary. the printing is always done to current active page.

example:

```
mov al, 'a'
mov ah, 0eh
int 10h

; note: on specific systems this
; function may not be supported in graphics mode.
```

INT 10h / **AH = 13h** - write string.

input:
AL = write mode:
  bit 0: update cursor after writing;
  bit 1: string contains attributes.
BH = page number.
BL = attribute if string contains only characters (bit 1 of AL is zero).
CX = number of characters in string (attributes are not counted).
DL,DH = column, row at which to start writing.
ES:BP points to string to be printed.

example:

```
        mov al, 1
        mov bh, 0
        mov bl, 0011_1011b
        mov cx, msg1end - offset msg1 ; calculate message size.
        mov dl, 10
        mov dh, 7
        push cs
        pop es
        mov bp, offset msg1
        mov ah, 13h
        int 10h
        jmp msg1end
        msg1 db " hello, world! "
        msg1end:
```

INT 10h / **AX = 1003h** - toggle intensity/blinking.

input:
BL = write mode:
  0: enable intensive colors.
  1: enable blinking (not supported by the emulator and windows command prompt).
BH = 0 (to avoid problems on some adapters).

example:

```
mov ax, 1003h
mov bx, 0
int 10h
```

---

bit color table:
character attribute is 8 bit value, low 4 bits set fore color, high 4 bits set background color.
note: the emulator and windows command line prompt do not support

background blinking, however to make colors look the same in dos and in full screen mode it is required to turn off the background blinking.

HEX	BIN	COLOR
0	0000	black
1	0001	blue
2	0010	green
3	0011	cyan
4	0100	red
5	0101	magenta
6	0110	brown
7	0111	light gray
8	1000	dark gray
9	1001	light blue
A	1010	light green
B	1011	light cyan
C	1100	light red
D	1101	light magenta
E	1110	yellow
F	1111	white

note:

```
; use this code for compatibility with dos/cmd prompt full screen mode:
mov     ax, 1003h
mov     bx, 0   ; disable blinking.
int     10h
```

INT 11h - get BIOS equipment list.
return:
AX = BIOS equipment list word, actually this call returns the contents of the word at 0040h:0010h.

Currently this function can be used to determine the number of installed number of floppy disk drives.

Bit fields for BIOS-detected installed hardware:

bit(s)	Description
15-14	Number of parallel devices.
13	Reserved.
12	Game port installed.
11-9	Number of serial devices.
8	Reserved.

7-6	Number of floppy disk drives (minus 1):
	00 single floppy disk;
	01 two floppy disks;

10 three floppy disks;
11 four floppy disks.

5-4 Initial video mode:
00 EGA,VGA,PGA, or other with on-board video BIOS;
01 40x25 CGA color.
10 80x25 CGA color (emulator default).
11 80x25 mono text.

3 Reserved.
2 PS/2 mouse is installed.
1 Math coprocessor installed.
0 Set when booted from floppy.

INT 12h - get memory size.
return:
AX = kilobytes of contiguous memory starting at absolute address 00000h, this call returns the contents of the word at 0040h:0013h.

Floppy drives are emulated using *FLOPPY_0(..3)* **files.**

INT 13h / **AH = 00h** - reset disk system.

INT 13h / **AH = 02h** - read disk sectors into memory.
INT 13h / **AH = 03h** - write disk sectors.
input:
AL = number of sectors to read/write (must be nonzero)
CH = cylinder number (0..79).
CL = sector number (1..18).
DH = head number (0..1).
DL = drive number (0..3 , for the emulator it depends on quantity of FLOPPY_ files).
ES:BX points to data buffer.

return:
CF set on error.
CF clear if successful.
AH = status (0 - if successful).
AL = number of sectors transferred.

Note: each sector has **512** bytes.

INT 15h / **AH = 86h** - BIOS wait function.
input:
CX:DX = interval in microseconds

return:
CF clear if successful (wait interval elapsed),
CF set on error or when wait function is already in progress.

Note:
the resolution of the wait period is 977 microseconds on many systems (1 million microseconds - 1 second).
Windows XP does not support this interrupt (always sets CF=1).

INT 16h / **AH = 00h** - get keystroke from keyboard (no echo).
return:
AH = BIOS scan code.
AL = ASCII character.
(if a keystroke is present, it is removed from the keyboard buffer).

INT 16h / **AH = 01h** - check for keystroke in the keyboard buffer.
return:
ZF = 1 if keystroke is not available.
ZF = 0 if keystroke available.
AH = BIOS scan code.
AL = ASCII character.
(if a keystroke is present, it is not removed from the keyboard buffer).

INT 19h - system reboot.
Usually, the BIOS will try to read sector 1, head 0, track 0 from drive **A:** to 0000h:7C00h. The emulator just stops the execution, to boot from floppy drive select from the menu: **'virtual drive' -> 'boot from floppy'**

INT 1Ah / **AH = 00h** - get system time.

return:
CX:DX = number of clock ticks since midnight.
AL = midnight counter, advanced each time midnight passes.

notes:
there are approximately **18.20648** clock ticks per second,
and **1800B0h** per 24 hours.
AL is not set by the emulator.

INT 20h - exit to operating system.

The short list of emulated MS-DOS interrupts -- INT 21h

DOS file system is emulated in **C:\emu8086\vdrive\x** (x is a drive letter)

If no drive letter is specified and current directory is not set, then **C:\emu8086\MyBuild\** path is used by default. **FLOPPY_0,1,2,3** files are emulated independently from DOS file system.

For the emulator physical drive **A:** is this file **c:\emu8086\FLOPPY_0** (for BIOS interrupts: **INT 13h** and boot).

For DOS interrupts (**INT 21h**) drive **A:** is emulated in this subdirectory: **C:\emu8086\vdrive\a\**

Note: DOS file system limits the file and directory names to 8 characters, extension is limited to 3 characters;
example of a valid file name: **myfile.txt** (file name = 6 chars, extension - 3 chars). extension is written after the dot, no other dots are allowed.

INT 21h / **AH=1** - read character from standard input, with echo, result is stored in **AL**.
if there is no character in the keyboard buffer, the function waits until any key is pressed.

example:

```
mov ah, 1
int 21h
```

INT 21h / AH=2 - write character to standard output.
entry: **DL** = character to write, after execution **AL = DL**.

example:

```
mov ah, 2
mov dl, 'a'
int 21h
```

INT 21h / AH=5 - output character to printer.
entry: **DL** = character to print, after execution **AL = DL**.

example:

```
mov ah, 5
mov dl, 'a'
int 21h
```

INT 21h / AH=6 - direct console input or output.

parameters for output: **DL** = 0..254 (ascii code)
parameters for input: **DL** = 255

for output returns: AL = DL
for input returns: **ZF** set if no character available and **AL = 00h**, **ZF** clear if character available.
AL = character read; buffer is cleared.

example:

```
mov ah, 6
mov dl, 'a'
int 21h      ; output character.

mov ah, 6
mov dl, 255
int 21h       ; get character from keyboard buffer (if any) or set ZF=1.
```

INT 21h / **AH=7** - character input without echo to AL.
if there is no character in the keyboard buffer, the function waits until any key is pressed.

example:

```
mov ah, 7
int 21h
```

INT 21h / **AH=9** - output of a string at **DS:DX**. String must be terminated by '**$**'.

example:

```
org 100h
mov dx, offset msg
mov ah, 9
int 21h
ret
msg db "hello world $"
```

INT 21h / **AH=0Ah** - input of a string to **DS:DX**, fist byte is buffer size, second byte is number of chars actually read. this function does **not** add '$' in the end of string. to print using **INT 21h** / **AH=9** you must set dollar character at the end of it and start printing from address **DS:DX + 2**.

example:

```
org 100h
mov dx, offset buffer
mov ah, 0ah
int 21h
```

```
jmp print
buffer db 10,?, 10 dup(' ')
print:
xor bx, bx
mov bl, buffer[1]
mov buffer[bx+2], '$'
mov dx, offset buffer + 2
mov ah, 9
int 21h
ret
```

the function does not allow to enter more characters than the specified buffer size.
see also **int21.asm** in c:\emu8086\examples

INT 21h / **AH=0Bh** - get input status;
returns: **AL = 00h** if no character available, **AL = 0FFh** if character is available.

INT 21h / **AH=0Ch** - flush keyboard buffer and read standard input.
entry: **AL** = number of input function to execute after flushing buffer (can be 01h,06h,07h,08h, or 0Ah - for other values the buffer is flushed but no input is attempted); other registers as appropriate for the selected input function.

INT 21h / **AH= 0Eh** - select default drive.

Entry: **DL** = new default drive (0=A:, 1=B:, etc)

Return: **AL** = number of potentially valid drive letters

Notes: the return value is the highest drive present.

INT 21h / **AH= 19h** - get current default drive.

Return: AL = drive (0=A:, 1=B:, etc)

INT 21h / **AH=25h** - set interrupt vector;
input: **AL** = interrupt number. **DS:DX** -> new interrupt handler.

INT 21h / **AH=2Ah** - get system date;
return: **CX** = year (1980-2099). **DH** = month. **DL** = day. **AL** = day of week (00h=Sunday)

INT 21h / **AH=2Ch** - get system time;
return: **CH** = hour. **CL** = minute. **DH** = second. **DL** = 1/100 seconds.

INT 21h / **AH=35h** - get interrupt vector;
entry: **AL** = interrupt number;
return: **ES:BX** -> current interrupt handler.

INT 21h / **AH= 39h** - make directory.
entry: **DS:DX** -> ASCIZ pathname; zero terminated string, for example:

```
org 100h
mov dx, offset filepath
mov ah, 39h
int 21h

ret
```

```
filepath DB "C:\mydir", 0     ; path to be created.
end
```
the above code creates **c:\emu8086\vdrive\C\mydir** directory if run by the emulator.

Return: **CF** clear if successful **AX** destroyed. **CF** set on error **AX** = error code.
Note: all directories in the given path must exist except the last one.

INT 21h / **AH= 3Ah** - remove directory.

Entry: **DS:DX** -> ASCIZ pathname of directory to be removed.

Return:

CF is clear if successful, **AX** destroyed **CF** is set on error **AX** = error code.

Notes: directory must be empty (there should be no files inside of it).

INT 21h / **AH= 3Bh** - set current directory.

Entry: **DS:DX** -> ASCIZ pathname to become current directory (max 64 bytes).

Return:

Carry Flag is clear if successful, **AX** destroyed.
Carry Flag is set on error **AX** = error code.
Notes: even if new directory name includes a drive letter, the default drive is not changed,
only the current directory on that drive.

INT 21h / **AH= 3Ch** - create or truncate file.

entry:

CX = file attributes:

```
mov cx, 0      ;  normal - no attributes.
mov cx, 1      ;  read-only.
mov cx, 2      ;  hidden.
mov cx, 4      ;  system
mov cx, 7      ;  hidden, system and read-only!
mov cx, 16      ;  archive
```

DS:DX -> ASCIZ filename.

returns:

CF clear if successful, **AX** = file handle.
CF set on error **AX** = error code.

note: if specified file exists it is deleted without a warning.

example:

```
org 100h
mov ah, 3ch
mov cx, 0
mov dx, offset filename
mov ah, 3ch
int 21h
jc err
mov handle, ax
jmp k
filename db "myfile.txt", 0
handle dw ?
err:
; ....
k:
ret
```

INT 21h / **AH= 3Dh** - open existing file.

Entry:

AL = access and sharing modes:

```
mov al, 0   ; read
mov al, 1   ; write
mov al, 2   ; read/write
```

DS:DX -> ASCIZ filename.

Return:

CF clear if successful, **AX** = file handle.
CF set on error **AX** = error code.

note 1: file pointer is set to start of file.
note 2: file must exist.

example:

```
org 100h
mov al, 2
mov dx, offset filename
mov ah, 3dh
int 21h
jc err
mov handle, ax
jmp k
filename db "myfile.txt", 0
handle dw ?
err:
; ....
k:
ret
```

INT 21h / **AH= 3Eh** - close file.

Entry: **BX** = file handle

Return:

CF clear if successful, **AX** destroyed.
CF set on error, **AX** = error code (06h).

INT 21h / **AH= 3Fh** - read from file.

Entry:

BX = file handle.
CX = number of bytes to read.
DS:DX -> buffer for data.

Return:

CF is clear if successful - **AX** = number of bytes actually read; 0 if at EOF (end of file) before call.
CF is set on error **AX** = error code.

Note: data is read beginning at current file position, and the file position is updated after a successful read the returned **AX** may be smaller than the request in **CX** if a partial read occurred.

INT 21h / **AH= 40h** - write to file.

entry:

BX = file handle.
CX = number of bytes to write.
DS:DX -> data to write.

return:

CF clear if successful; **AX** = number of bytes actually written.
CF set on error; **AX** = error code.

note: if **CX** is zero, no data is written, and the file is truncated or extended to the current position data is written beginning at the current file position, and the file position is updated after a successful write the usual cause for **AX** < **CX** on return is a full disk.

INT 21h / **AH= 41h** - delete file (unlink).

Entry:

DS:DX -> ASCIZ filename (no wildcards, but see notes).

return:

CF clear if successful, **AX** destroyed. **AL** is the drive of deleted file (undocumented).
CF set on error **AX** = error code.

Note: DOS does not erase the file's data; it merely becomes inaccessible because the FAT chain for the file is cleared deleting a file which is currently open may lead to filesystem corruption.

INT 21h / **AH= 42h** - SEEK - set current file position.

Entry:

AL = origin of move: **0** - start of file. **1** - current file position. **2** - end of file.
BX = file handle.
CX:DX = offset from origin of new file position.

Return:

CF clear if successful, **DX:AX** = new file position in bytes from start of file.
CF set on error, AX = error code.

Notes:

for origins **1** and **2**, the pointer may be positioned before the start of the file; no error is returned in that case, but subsequent attempts to read or write the file will produce errors. If the new position is beyond the current end of file, the file will be extended by the next write (see AH=40h).

example:

```
org 100h
mov ah, 3ch
mov cx, 0
mov dx, offset filename
mov ah, 3ch
int 21h  ; create file...
mov handle, ax

mov bx, handle
mov dx, offset data
mov cx, data_size
mov ah, 40h
int 21h ; write to file...

mov al, 0
mov bx, handle
mov cx, 0
mov dx, 7
mov ah, 42h
int 21h ; seek...

mov bx, handle
mov dx, offset buffer
mov cx, 4
mov ah, 3fh
int 21h ; read from file...

mov bx, handle
mov ah, 3eh
int 21h ; close file...
ret

filename db "myfile.txt", 0
handle dw ?
data db " hello files! "
data_size=$-offset data
buffer db 4 dup(' ')
```

INT 21h / **AH= 47h** - get current directory.

Entry:

DL = drive number (00h = default, 01h = A:, etc)
DS:SI -> 64-byte buffer for ASCIZ pathname.

Return:

Carry is clear if successful
Carry is set on error, **AX** = error code (0Fh)

Notes:

the returned path does not include a drive and the initial backslash.

INT 21h / **AH=4Ch** - return control to the operating system (stop program).

INT 21h / **AH= 56h** - rename file / move file.

Entry:

DS:DX -> ASCIZ filename of existing file.
ES:DI -> ASCIZ new filename.

Return:

CF clear if successful.
CF set on error, **AX** = error code.

Note: allows move between directories on same logical drive only; open files should not be renamed!

mouse driver interrupts -- INT 33h

INT 33h / **AX=0000** - mouse ininialization. any previous mouse pointer is hidden.

returns:
if successful: **AX**=0FFFFh and **BX**=number of mouse buttons.
if failed: **AX**=0

example:

```
mov ax, 0
int 33h
```

see also: mouse.asm in examples.

INT 33h / **AX=0001** - show mouse pointer.

example:

```
mov ax, 1
int 33h
```

INT 33h / **AX=0002** - hide visible mouse pointer.

example:

```
mov ax, 2
int 33h
```

INT 33h / **AX=0003** - get mouse position and status of its buttons.

returns:
if left button is down: **BX**=1
if right button is down: **BX**=2
if both buttons are down: **BX**=3
CX = x
DX = y

example:

```
mov ax, 3
int 33h

; note: in graphical 320x200 mode the value of CX is doubled.
; see mouse2.asm in examples.
```

8086 assembler tutorial for beginners

Interrupts

Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character you can simply call the interrupt and it will do everything for you. There are also interrupt functions that work with disk drive and other hardware. We call such functions **software interrupts**.

Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.

To make a **software interrupt** there is an **INT** instruction, it has very simple syntax:

INT value

Where **value** can be a number between 0 to 255 (or 0 to 0FFh), generally we will use hexadecimal numbers.
You may think that there are only 256 functions, but that is not correct. Each interrupt may have sub-functions.
To specify a sub-function **AH** register should be set before calling interrupt. Each interrupt may have up to 256 sub-functions (so we get 256 * 256 = 65536 functions). In general **AH** register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.

The following example uses **INT 10h** sub-function **0Eh** to type a "Hello!" message. This functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary.

```
ORG    100h          ; instruct compiler to
make simple single segment .com file.

; The sub-function that we are using
; does not modify the AH register on
; return, so we may set it only once.

MOV    AH, 0Eh    ; select sub-function.
```

```
; INT 10h / 0Eh sub-function
; receives an ASCII code of the
; character that will be printed
; in AL register.

MOV    AL, 'H'    ; ASCII code: 72
INT    10h        ; print it!

MOV    AL, 'e'    ; ASCII code: 101
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'o'    ; ASCII code: 111
INT    10h        ; print it!

MOV    AL, '!'    ; ASCII code: 33
INT    10h        ; print it!

RET               ; returns to operating
system.
```

Copy & paste the above program to the source code editor, and press [**Compile and Emulate**] button. Run it!

Memory Access

To access memory we can use these four registers: **BX, SI, DI, BP**.

Combining these registers inside **[ ]** symbols, we can get different memory locations.

These combinations are supported (addressing modes):

[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]

d8 - stays for 8 bit signed immediate displacement (for example: 22, 55h, -1)

d16 - stays for 16 bit signed immediate displacement (for example: 300, 5517h, -259).

Displacement can be a immediate value or offset of a variable, or even both. If there are several values, assembler evaluates all values and calculates a single immediate value.

Displacement can be inside or outside of the **[ ]** symbols, assembler generates the same machine code for both ways.

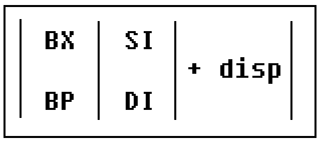
Displacement is a **signed** value, so it can be both positive or negative.

Generally the compiler takes care about difference between **d8** and **d16**, and generates the required machine code.

For example, let's assume that **DS = 100**, **BX = 30**, **SI = 70**.
The following addressing mode: **[BX + SI] + 25**
Is calculated by processor to this physical address: **100 * 16 + 30 + 70 + 25 = 1725**.

By default **DS** segment register is used for all modes except those with **BP** register, for these **SS** segment register is used.

There is an easy way to remember all those possible combinations using this chart:



All valid combinations can be formed by taking only one item from each column or skipping the column by not taking anything from it. **BX** and **BP** never go together. Neither **SI** and **DI** do.

Examples of valid addressing modes:

[BX+5]

[BX+SI]

[DI+BX-4]

The value in segment register (CS, DS, SS, ES) is called a **segment**, and the value in general purpose register (BX, SI, DI, BP) is called an **offset**.

When DS contains value **1234h** and SI contains the value **7890h** it can be also recorded as **1234:7890**. The physical address will be 1234h * 10h + 7890h = 19BD0h.

If zero is added to a decimal number it is multiplied by 10, however **10h = 16**, so If zero is added to a hexadecimal value, it is multiplied by 16, for example:

7h = 7
70h = 112

In order to say the compiler about data type, these prefixes should be used:

byte ptr - for byte.
word ptr - for word (two bytes).

for example:

```
byte ptr [BX]     ; byte access.
    or
word ptr [BX]     ; word access.
```

Emu Assembler supports shorter prefixes as well:

b. - for **byte ptr**
w. - for **word ptr**

in certain cases the assembler can calculate the data type automatically.

MOV instruction

- copies the **second operand** (source) to the **first operand** (destination).
- the source operand can be an immediate value, general-purpose register or memory location.
- the destination register can be a general-purpose register, or memory location.
- both operands must be the same size, which can be a byte or a word.

these types of operands are supported:

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

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

for segment registers only these types of **MOV** are supported:

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

SREG: DS, ES, SS, and only as second operand: CS.

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable

The **MOV** instruction cannot be used to set the value of the **CS** and **IP** registers.

A short program that demonstrates the use of **MOV** instruction:

```
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.
```

Once the above program is typed into the code editor, and [**Compile and Emulate**] button is pressed (**F5** key)

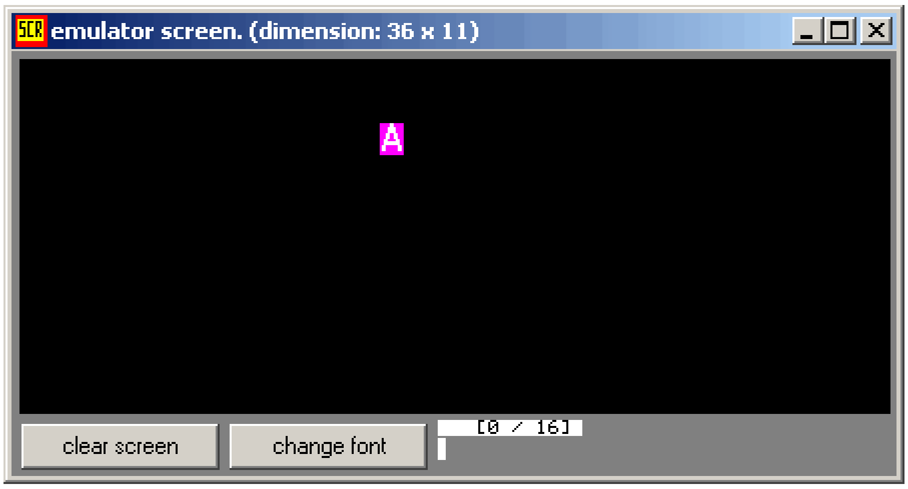
The emulator window should open with this program loaded, clicking [**Single Step**] button should change the register values.

How to do **copy & paste**:

1. Select the above text using mouse, click before the text and drag it down until everything is selected.
2. Press **Ctrl + C** combination to copy.
3. Click inside the source code editor and press **Ctrl + V** combination to paste.

"**;**" is used for comments, anything after "**;**" symbol is ignored.

The result of a working program:



Actually the above program writes directly to video memory, now we can see that **MOV** is a very powerful instruction.

Variables

Variable is a memory location. For a programmer 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.

Our compiler supports two types of variables: **BYTE** and **WORD**.

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

DB - stays for Define Byte.
DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

As you probably know from *part 2* of this tutorial, **MOV** instruction is used to copy values from source to destination.
Let's see another example with **MOV** instruction:

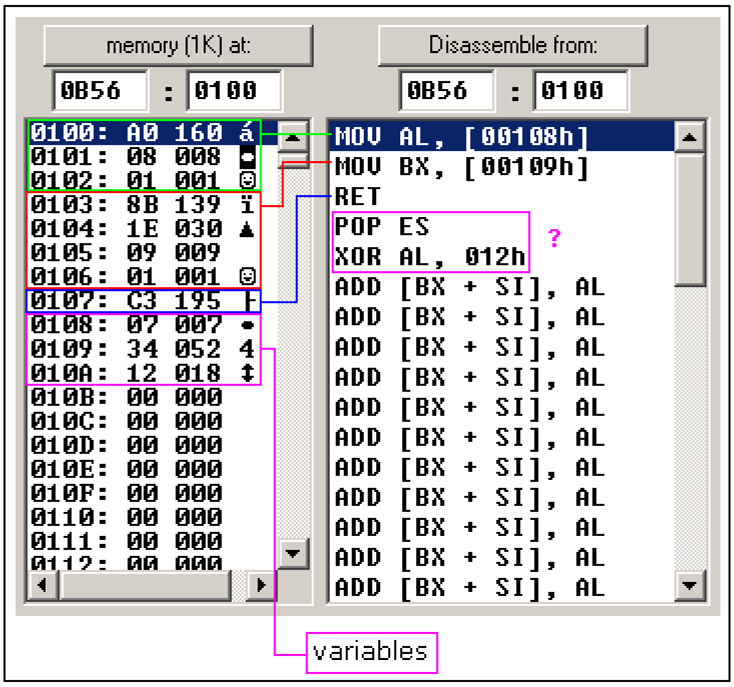
```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; stops the program.

VAR1 DB 7
var2 DW 1234h
```

Copy the above code to the source editor, and press **F5** key to compile it and load in the emulator. You should get something like:



```
0B56 : 0100                 0B56 : 0100

0100: A0 160 á              MOV AL, [00108h]
0101: 08 008 ◘              MOV BX, [00109h]
0102: 01 001 ☺              RET
0103: 8B 139 ï              POP ES
0104: 1E 030 ▲              XOR AL, 012h
0105: 09 009                ADD [BX + SI], AL
0106: 01 001 ☺              ADD [BX + SI], AL
0107: C3 195 ├              ADD [BX + SI], AL
0108: 07 007 •              ADD [BX + SI], AL
0109: 34 052 4              ADD [BX + SI], AL
010A: 12 018 ↕              ADD [BX + SI], AL
010B: 00 000                ADD [BX + SI], AL
010C: 00 000                ADD [BX + SI], AL
010D: 00 000                ADD [BX + SI], AL
010E: 00 000                ADD [BX + SI], AL
010F: 00 000                ADD [BX + SI], AL
0110: 00 000
0111: 00 000
0112: 00 000
```

As you see this looks a lot like our example, except that variables are replaced with actual memory locations. When compiler makes machine code, it automatically replaces all variable names with their **offsets**. By default segment is loaded in **DS** register (when **COM** files is loaded the value of **DS** register is set to the same value as **CS** register - code segment).

In memory list first row is an **offset**, second row is a **hexadecimal value**, third row is **decimal value**, and last row is an **ASCII** character value.

Compiler is not case sensitive, so "**VAR1**" and "**var1**" refer to the same variable.

The offset of **VAR1** is **0108h**, and full address is **0B56:0108**.

The offset of **var2** is **0109h**, and full address is **0B56:0109**, this variable is a **WORD** so it occupies **2 BYTES**. It is assumed that low byte is stored at lower address, so **34h** is located before **12h**.

You can see that there are some other instructions after the **RET** instruction, this happens because disassembler has no idea about where the data starts, it just processes the values in memory and it understands them as valid 8086 instructions (we will learn them later).

You can even write the same program using **DB** directive only:

```
ORG 100h

DB 0A0h
DB 08h
DB 01h

DB 8Bh
DB 1Eh
DB 09h
DB 01h

DB 0C3h

DB 7

DB 34h
DB 12h
```

Copy the above code to the source editor, and press **F5** key to compile and load it in the emulator. You should get the same disassembled code, and the same functionality!

As you may guess, the compiler just converts the program source to the set of bytes, this set is called **machine code**, processor understands the **machine code** and executes it.

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.
Why executable file is loaded at **offset** of **100h**? Operating system keeps some data about the program in the first 256 bytes of the **CS** (code segment), such as command line parameters and etc.
Though this is true for **COM** files only, **EXE** files are loaded at offset of **0000**, and generally use special segment for variables. Maybe we'll talk more about **EXE** files later.

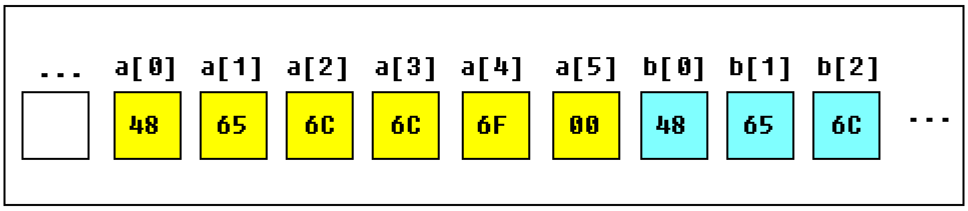
Arrays

Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

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

b is an exact copy of the *a* array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:



You can access the value of any element in array using square brackets, for example:

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

You can also use any of the memory index registers **BX, SI, DI, BP**, for example:

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

If you need to declare a large array you can use **DUP** operator. The syntax for **DUP**:

number DUP (value(s))
number - number of duplicate to make (any constant value).
value - expression that DUP will duplicate.

for example:

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

is an alternative way of declaring:

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

one more example:

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

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings.

Getting the Address of a Variable

There is **LEA** (Load Effective Address) instruction and alternative **OFFSET** operator. Both **OFFSET** and **LEA** can be used to get the offset address of the variable.
LEA is more powerful because it also allows you to get the address of an indexed variables. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

Reminder:

In order to tell the compiler about data type,
these prefixes should be used:

BYTE PTR - for byte.
WORD PTR - for word (two bytes).

For example:

```
BYTE PTR [BX]     ; byte access.
   or
WORD PTR [BX]     ; word access.
```

assembler supports shorter prefixes as well:

b. - for **BYTE PTR**
w. - for **WORD PTR**

in certain cases the assembler can calculate the data type automatically.

Here is first example:

```
ORG 100h

MOV    AL, VAR1              ; check value of VAR1
by moving it to 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 AL.

RET

VAR1   DB  22h

END
```

Here is another example, that uses **OFFSET** instead of **LEA**:

```
ORG 100h

MOV    AL, VAR1              ; check value of VAR1
by moving it to AL.

MOV    BX, OFFSET 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 AL.

RET

VAR1   DB  22h

END
```

Both examples have the same functionality.

These lines:

```
LEA BX, VAR1
MOV BX, OFFSET VAR1
```

are even compiled into the same machine code: `MOV BX, num`
num is a 16 bit value of the variable offset.

Please note that only these registers can be used inside square brackets (as memory pointers): **BX, SI, DI, BP**!
(see previous part of the tutorial).

Constants

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants **EQU** directive is used:

```
name EQU < any expression >
```

For example:

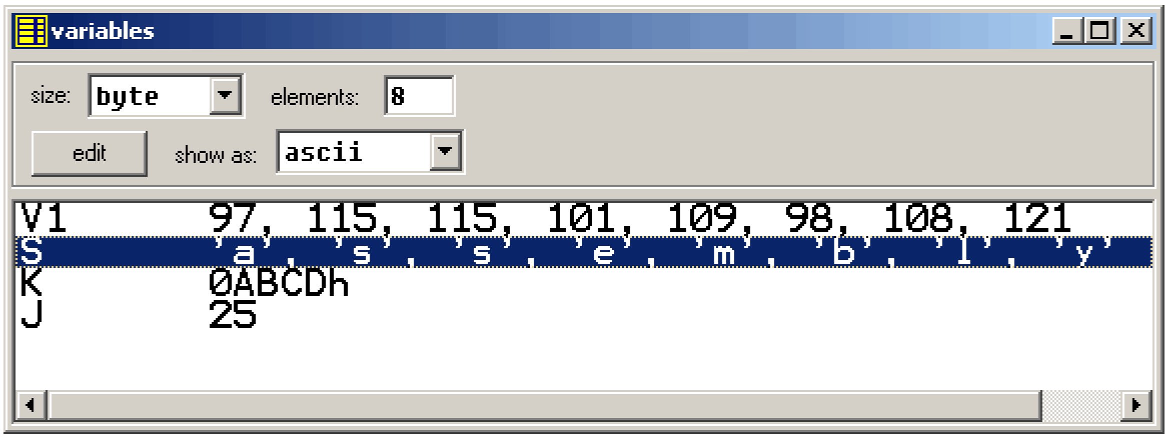
```
k EQU 5

MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

You can view variables while your program executes by selecting "**Variables**" from the "**View**" menu of emulator.



```
V1        97, 115, 115, 101, 109, 98, 108, 121
S         'a', 's', 's', 'e', 'm', 'b', 'l', 'y'
K         0ABCDh
J         25
```

To view arrays you should click on a variable and
set **Elements** property to array size. In assembly language there are not strict data types, so any variable can be presented as an array.

Variable can be viewed in any numbering system:

- **HEX** - hexadecimal (base 16).
- **BIN** - binary (base 2).
- **OCT** - octal (base 8).
- **SIGNED** - signed decimal (base 10).
- **UNSIGNED** - unsigned decimal (base 10).
- **CHAR** - ASCII char code (there are 256 symbols, some symbols are invisible).

You can edit a variable's value when your program is running, simply double click it, or select it and click **Edit** button.

It is possible to enter numbers in any system, hexadecimal numbers should have "**h**" suffix, binary "**b**" suffix, octal "**o**" suffix, decimal numbers require no suffix. String can be entered this way:
'hello world', 0
(this string is zero terminated).

Arrays may be entered this way:
1, 2, 3, 4, 5
(the array can be array of bytes or words, it depends whether **BYTE** or **WORD** is selected for edited variable).

Expressions are automatically converted, for example:
when this expression is entered:
5 + 2
it will be converted to **7** etc...

List of Experiments to be performed by students

Sr. No.	Name of Experiment
1	Study of 8086 Architecture and basis of programming
2	Study Instruction set and addressing modes of 8086 Microprocessor
3	Study of Assembler Directives for ALP 8086 Microprocessor
4	ALP to perform addition, subtraction, multiplication and division of 8 bit, 16 bit and 32 bit data
5	ALP to convert largest and smallest number from array
6	ALP to count the no of ODD and EVEN numbers from the given array of data.
7	ALP to arrange the data in ascending/descending order
8	ALP to check the number is prime or not
9	ALP to find the factorial of a given number, reading the number from keyboard
10	Read the string from the keyboard using macro and display it on the screen
11	ALP to convert a number entered through Keyboard in hexadecimal and decimal using procedures (near/far)
12	ALP to validate username and password
13	ALP to perform IO operation using 8086 Emulator