|  |  |  |
| --- | --- | --- |
| **EXP NO: 1** | **STUDY OF 8086 EMULATOR** | **DATE:** |

**AIM:**

To study about the emu8086 emulator and tutorial.

**WHAT IS ASSEMBLY LANGUAGE:**

Assembly language is a low-level programming language. The simple computer model as I see it in figure 1. The **system bus** connects the various components of a computer. The **CPU** is the heart of the computer, most of computations occur inside the **CPU** shown in figure 2. **RAM** is a place to where the programs are loaded in order to be executed.

**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.

4 general purpose registers (AX, BX, CX, DX) are made of two separate 8bit registers, for example if AX= **0011000000111001b**, then AH=**00110000b** and AL=**00111001b**. Therefore, when you modify any of the 8bit registers 16bit register is also updated, and vice-versa. The same is for other 3 registers, "H" is for high and "L" is for low part. Since 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.

**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.

The segment registers have a very special purpose - pointing at accessible blocks of memory.

**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.

**ARITHMETIC AND LOGICAL OPERATIONS:**  
  
Most Arithmetic and Logic Instructions affect the processor status register (or **Flags**) in figure 3.

As you may see there are 16 bits in this register, each bit is called a **flag** and can take a value of **1** or **0**.

* **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**. For example, when you add bytes **255 + 1** (result is not in range 0...255). When there is no overflow, this flag is set to **0**.
* **Zero Flag (ZF)** - set to **1** when result is **zero**. For none zero result this flag is set to **0**.
* **Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. Actually, this flag takes the value of the most significant bit.
* **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).
* **Parity Flag (PF)** - this flag is set to **1** when there is even number of one bit in result, and to **0** when there is odd number of one bit. Even if result is a word only 8 low bits are analysed!
* **Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
* **Interrupt enable Flag (IF)** - when this flag is set to **1** CPU reacts to interrupts from external devices.
* **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward, when this flag is set to **1** the processing is done backward.

**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. |

**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 |

**INSTRUCTIONS:**

There are 3 groups of instructions.

First group: **ADD**, **SUB**, **CMP**, **AND**, **TEST**, **OR**, **XOR**  
  
These types of operands are supported:

|  |
| --- |
| REG, memory memory, REG REG, REG memory, immediate 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, etc...  
**Immediate**: 5, -24, 3Fh, 10001101b, etc...  
  
After operation between operands, result is always stored in first operand.  **CMP** and **TEST** instructions affect flags only and do not store a result (These instructions are used to make decisions during program execution).  
  
These instructions affect these flags only:  
       **CF**, **ZF**, **SF**, **OF**, **PF**, **AF**.

* **ADD** - add second operand to first.
* **SUB** - Subtract second operand to first.
* **CMP** - Subtract second operand from first **for flags only**.
* **AND** - Logical AND between all bits of two operands. These rules apply:

1 AND 1 = 1  
1 AND 0 = 0  
0 AND 1 = 0  
0 AND 0 = 0

As you see we get **1** only when both bits are **1**.

* **TEST** - The same as **AND** but **for flags only**.
* **OR** - Logical OR between all bits of two operands. These rules apply:

1 OR 1 = 1  
1 OR 0 = 1  
0 OR 1 = 1  
0 OR 0 = 0

As you see we get **1** every time when at least one of the bits is **1**.

* **XOR** - Logical XOR (exclusive OR) between all bits of two operands. These rules apply:

1 XOR 1 = 0  
1 XOR 0 = 1  
0 XOR 1 = 1  
0 XOR 0 = 0

As you see we get **1** every time when bits are different from each other.

Second group: **MUL**, **IMUL**, **DIV**, **IDIV**  
  
These types of operands are supported:

|  |
| --- |
| REG memory |

**REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  
**Memory**: [BX], [BX+SI+7], variable, etc...  
  
**MUL** and **IMUL** instructions affect these flags only:  
       **CF**, **OF**  
When result is over operand size these flags are set to **1**, when result fits in operand size these flags are set to **0**.  
  
For **DIV** and **IDIV** flags are undefined.

* **MUL** - Unsigned multiply
* **IMUL** - Signed multiply
* **DIV** - Unsigned divide
* **IDIV** - Signed divide

Third group: **INC**, **DEC**, **NOT**, **NEG**  
  
These types of operands are supported:

|  |
| --- |
| REG memory |

**REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  
**Memory**: [BX], [BX+SI+7], variable, etc...

**INC**, **DEC** instructions affect these flags only:  
       **ZF**, **SF**, **OF**, **PF**, **AF**.  
  
**NOT** instruction does not affect any flags!  
  
**NEG** instruction affects these flags only:  
       **CF**, **ZF**, **SF**, **OF**, **PF**, **AF**.

* **NOT** - Reverse each bit of operand.
* **NEG** - Make operand negative (two's complement). Actually, it reverses each bit of operand and then adds 1 to it. For example, 5 will become -5, and -2 will become 2.

**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. 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.

For example:

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

mov ah, 2

mov dl, 'a'

int 21h

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

**THE STACK:**

Stack is an area of memory for keeping temporary data. Stack is used by **CALL** instruction to keep return address. **RET** instruction gets this value from the stack and returns to that offset.

Quite the same thing happens when **INT** instruction calls an interrupt, it stores in stack flag register, code segment and offset. **IRET** instruction is used to return from interrupt call.

We can also use the stack to keep any other data,  
there are two instructions that work with the stack:

**PUSH** - stores 16bit value in the stack.  
  
**POP** - gets 16bit value from the stack.

|  |
| --- |
| Syntax for **PUSH** instruction:  PUSH REG PUSH SREG PUSH memory PUSH immediate  **REG**: AX, BX, CX, DX, DI, SI, BP, SP. **SREG**: DS, ES, SS, CS. **Memory**: [BX], [BX+SI+7], 16 bit variable, etc... **Immediate**: 5, -24, 3Fh, 10001101b, etc... |

|  |
| --- |
| Syntax for **POP** instruction:  POP REG POP SREG POP memory  **REG**: AX, BX, CX, DX, DI, SI, BP, SP. **SREG**: DS, ES, SS, (except CS). **memory**: [BX], [BX+SI+7], 16 bit variable, etc... |

Notes:

* **PUSH** and **POP** work with 16bit values only!
* Note: **PUSH immediate** works only on 80186 CPU and later.

The stack uses **LIFO** (Last-In First Out) algorithm, this means that if we push these values one by one into the stack:  
 **1, 2, 3, 4, 5** in figure 4**,** the first value that we will get on pop will be **5**, then **4**, **3**, **2**, and only then **1**, It is very important to do equal number of **PUSH**s and **POP**s, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).  
  
**PUSH** and **POP** instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

* Store original value of the register in stack (using **PUSH**).
* Use the register for any purpose.

Restore the original value of the register from stack (using **POP**).

**RESULT:**

Thus, the emu8086 emulator and its tutorial has been studied and sample program has been executed successful