8086 processor development

Hello World

```
org 100h   ;starts at address 100 in memory

jmp main   ;jump to label main

message: db 'Hello World', 0

print:
   mov ah, 0eh         ;function to BIOS interrupt 10h
._loop:
   lodsb               ;loading the first part of message to al register
   cmp al, 0           ;comparing al register to 0(checking the null Pointer Of str)
   je .done
   int 10h             ;BIOS interrupt to print the first al register value on screen
   jmp ._loop          ;continue till the al is 0(occurs when si points to nullPoint)

.done:
   ret                 ;goes back to the call i.e in main label

main:
   mov si, message     ;loads the address of message to si register
   call print          ;calling print

ret
```

Segmentation

DS(data segment) register holds the value of segment.

```
org 100h

mov [0xff], 0x30   ;loads the hex 30 to RAM address FF(255)
                   ;actual physical RAM address is Data segment address(DS)*16 + 0x30
```

We can change the default data segment address as follows:

```
org 100h
; add your code here
mov ax, 0x300
mov ds, ax
mov [0xff], 0x30
ret
```

Sp - stack pointer grows downwards, decreased sp value

Call instructruction pushes the return address of subroutine to stack, so as to return back after execution of subroutine instructions using pop.

Writing data to RAM

```
org 100h

mov bx, 0xfff0  ;load register b with hex value FFF0
mv [bx], 0x30   ;load the hex value 30 to the address stored in b register which is FFF0
                ;this only works with bx register in 8086
```

For storing only 1 byte:

```
org 100h
mov bx, 0xfff0
mov byte [bx], 0xff
ret
```

For storing a word(2 bytes):

```
org 100h
mov bx, 0xfff0
mov word [bx], 0xff
ret
```

Reading data from RAM

```
org 100h

mov ax, [0x00] ;loads the value from RAM address 0x00 to register A
```

```
; You may customize thi
; The location of this

org 100h

mov bx, 0xff0
mov ax, [bx]

ret
```

Interrupt

Interrupts are those stored instructions in RAM corresponding to predetermined instructions. Example INT 0x00 - takes the first 4 bytes of RAM (segment:offset) having address of interrupt 0. These interrupts stored in little endian format to create a vector table.

Vector table

Contains 256 addresses stored in RAM starting from 0x00 having the address (segment and offset)

Custom interrupt:

```
org 100h

push ds    ;backup of data segment is stored in stack
mov ax, 0
mov ds, ax ;setting ds to 0x00 of RAM(vector table)
mov [0x00], handle_int0 ;the mem address of subroutine handle_int0 gets stored in 0x00 as offset
pop ds
int 0x00 ;calls the subroutine handle_int0
ret

handle_int0:
   mov ah, 0eh
   mov al, 'A'
   int 0x10
   iret
```

Talking with hardware

Out - writing to a hardware

```
org 100h

mov al, 'A'
```

```
out 130, al ;130 is the open port of printer
ret
```

IN - Reading from hardware

```
org 100h

in al, 110 ;reading from hardware port 110 to al
ret
```

Condition instructions

- CMP - finds the difference between the operands at sets the ZERO flag(ZF) if equal
- JE/JZ - (jump if equal/jump if zero), jumps if the ZF is 1

```
org 100h

mov al, 10
cmp al, 10
je _equal
jmp _exit

_equal:
    mov ah, 0eh
    mov al, 'A'
    int 0x10

_exit:

ret
```

- JNE - jump if not equal to zero or ZF = 0

- JA - if CF = 0 and ZF = 0. That is if the operand 1 is greater than operand 2, jump
- JB - if ZF = 1. That is if operand 2 is greater than operand 1, jump

```
org 100h

mov al, 11
cmp al, 15
jb _equal
jmp _exit

_equal:
    mov ah, 0eh
    mov al, 'A'
    int 0x10

_exit:

ret
```

- LODSB - load byte at DS:[SI] into AL. Then updates SI to +1 or -1 based on the DF flag

Algorithm:

- AL = DS:[SI]
- if DF = 0 then
    - SI = SI + 1

  else
    - SI = SI - 1

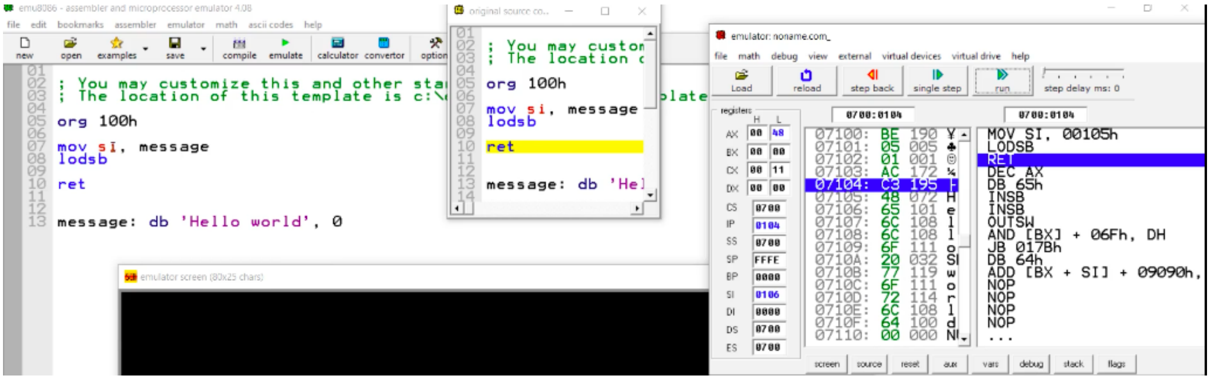
Example:

```
ORG 100h

LEA SI, al
MOV CX, 5
MOV AH, 0Eh

m: LODSB
INT 10h
LOOP m

RET
```



- STOSB - Stores byte in AL into ES:[DI] . Update DI

Algorithm:

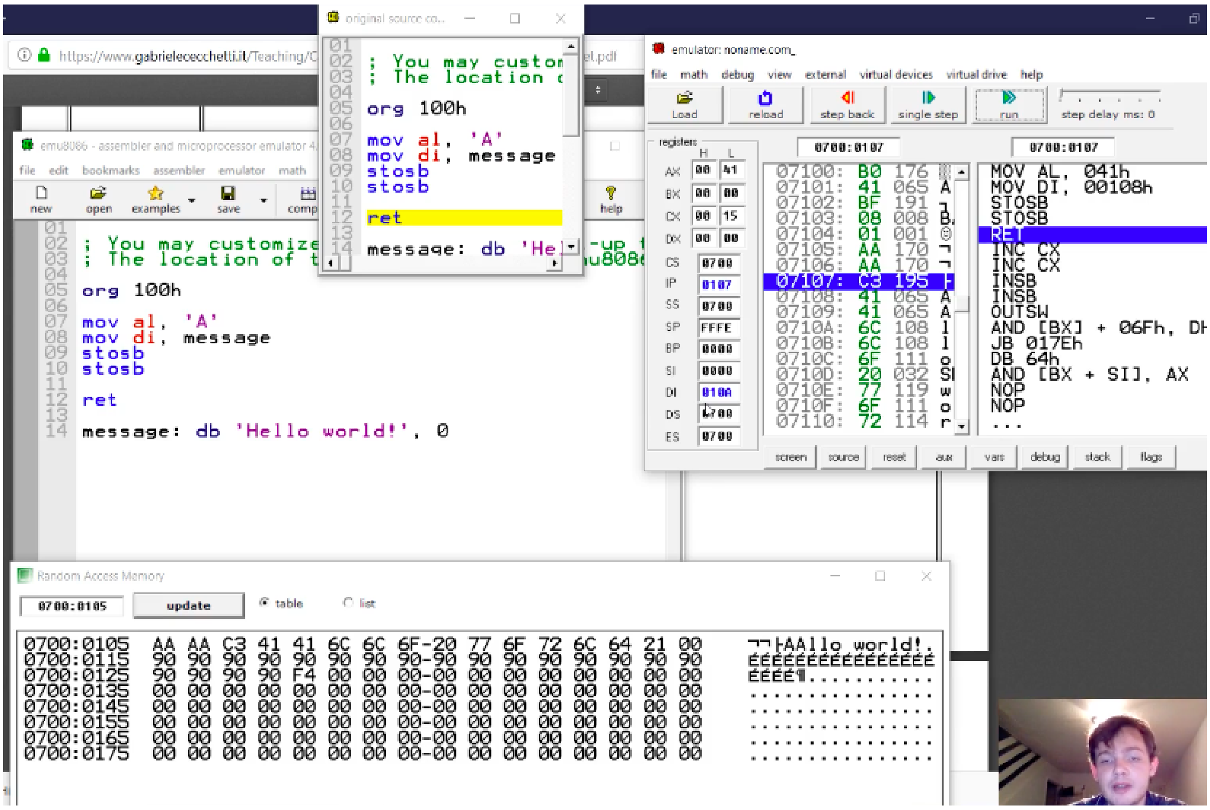
- ES:[DI] = AL
- if DF = 0 then
    - DI = DI + 1

  else
    - DI = DI - 1

Example:

```
ORG 100h

LEA DI, al
```



Modern x86 processor development

1. Linking C program with assembly code:

main.c:

```
#include <stdio.h>
extern int my_asm();

int main(int argc, char** argv) {
    int r = my_asm();
    printf("Value: %i\n", r);
}
```

file.asm

```
    global _my_asm

    section .text
_my_asm:
    mov eax, 10
    ret
```

nasm -fwin32 file.asm ⇒ file.obj
gcc main.c file.obj -omain ⇒ main.exe

Output: Value 10

- BP:

Base Pointer (BP) – The 16-bit BP register mainly **helps in referencing the parameter variables passed to a subroutine**. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

2. Linking Assembly with C using function parameters

main.c

```c
#include <stdio.h>
extern int my_asm(int a, int b);
int main(int argc, char** argv)
{
    int r = my_asm(5, 10);
    printf("Value: %i\n", r);
    return 0;
}
```

File.asm

```asm
    global  _my_asm

    section .text
_my_asm:
    push ebp
    mov ebp, esp
    mov eax, [ebp+12]
    pop ebp
    ret
```

- Here Initial base pointer address is stored in the stack
- Then the current stack pointer formed after the push of base pointer is loaded as the current base pointer
- Then the base pointer + 12 value is stored to eax which is the value 10 and bp + 8 is 5
- This is occurred by the push ebp(4) + call my_asm(4) + 4 address = value 5
- push ebp(4) + call my_asm(4) + 4 + 4 address = value 10
- EAX register returns value 10
- Note: calling the function makes it store the current address to stack taking 4 bytes

Scope of a variable

1.

{
    lint a = 3;
}

Here the value is stored in stack as 3 and is only accessible within the curly braces. When leaving the braces the stack pointer goes back to the previous state making it unable to access the value 3

2.

{
    int a = 5;
    int b = 10;
    return a+b;
}

```
main.c > main(int, char **)
#include <stdio.h>
extern int my_asm(int a, int b);
int main(int argc, char** argv)
{
    int r = my_asm(5, 10);
    printf("Value: %i\n", r);

    {
        int a = 30;
    }

    return 0;
}
```

2.1

file.asm

```
    global _my_asm
    section .text
_my_asm:
    push ebp

    ; esp = 1000
    ; ebp = 1000
    mov ebp, esp
    ; esp = 996
    sub esp, 8
    ; int a = 30;
    mov dword[ebp-4], 30
    ; int b = 80;
    mov dword[ebp-8], 80

    mov eax, dword[ebp-4]
    add eax, dword[ebp-8]

    add esp, 8

    pop ebp
    ret
```

2.2

file.asm

```
    global _my_asm
    section .text
_my_asm:
    push ebp
    mov ebp, esp
    ; {
    sub esp, 8

    ; int a = input[0]
    mov eax, dword[ebp+8]
    mov dword[ebp-4], eax

    ; int b = input[1]
    mov eax, dword[ebp+12]
    mov dword[ebp-8], eax

    ; return a + b
    mov eax, dword[ebp-4]
    add eax, dword[ebp-8]

    ; }
    add esp, 8

    pop ebp
```

Stored in the RAM as follows

<<<<<<<<<<Stack Pointer>>>>>>>>>>>>>>>>>
Reserved 1
Reserved 2
Reserved 3
Reserved 4
Reserved 5
Reserved 6
Reserved 7
Reserved 8

<<<<<<<<<<Base Pointer>>>>>>>>>>>>>>>>>
Base pointer original address
Base pointer original address
Base pointer original address
Base pointer original address
Call function address
Call function address
Call function address
Call function address
5
……..
……..
……….
10
………
………
……….

Returning Structure in assembly

1.

```c
#include <stdio.h>
struct test
{
    char buf[30];
};

extern struct test my_asm();

int main(int argc, char** argv)
{
    struct test a = my_asm();
    printf("%c", a.buf[0]);
    return 0;
}
```

```
    global  _my_asm

    section .text
_my_asm:
    push ebp
    mov ebp, esp

    mov eax, [esp+8]
    mov byte [eax], 'A'

    pop ebp
    ret
```

2.

```c
#include <stdio.h>
struct test
{
    int a;
    char b;
};

extern struct test my_asm();

int main(int argc, char** argv)
{
    struct test a = my_asm();
    printf("%i %c", a.a, a.b);
    return 0;
}
```

```
    global  _my_asm

    section .text
_my_asm:
    push ebp
    mov ebp, esp

    mov eax, 438373
    mov edx, 'A'

    pop ebp
    ret
```

Pointers

```
#include <stdio.h>


extern int my_asm(int* p);

int main(int argc, char** argv)
{
    int a = 50;
    int* ptr = &a;

    printf("%i\n", my_asm(ptr));

    return 0;
}
```

```
; --------------------------------

    global  _my_asm

    section .text
_my_asm:
    push ebp
    mov ebp, esp

    mov eax, [esp+8]
    mov eax, [eax]
    pop ebp
    ret
```

Passing structure to Assembly

```
#include <stdio.h>

struct test
{
    char buf[30];
};

extern int my_asm(struct test t);

int main(int argc, char** argv)
{
    struct test t;
    t.buf[0] = 'A';
    t.buf[1] = 'B';
    t.buf[2] = 'C';
    printf("%i\n", my_asm(t));

    return 0;
}
```

```
    global  _my_asm

    section .text
_my_asm:
    push ebp
    mov ebp, esp

    mov eax, [esp+9] ; ABC@
    pop ebp
    ret
```

Receive input from keyboard

```
#include <stdio.h>

extern char my_asm();

int main(int argc, char** argv)
{
    char c = my_asm();
    printf("%c\n", c);
    return 0;
}
```

```
    global  _my_asm
    extern _getchar
    section .text
_my_asm:
    push ebp
    mov ebp, esp
    call _getchar
    ; eax = char they entered
    pop ebp
    ret
```