MINISTERUL EDUCAŢIEI ŞI ŞTIINŢEI AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea "Calculatoare, Informatică și Microelectronică"
Filiera Anglofonă

Course Project

Computer Achitecture

Topic:

Floating point multiplication
Algorithm 1

Created by: st.gr. FAF-151
Bîzdîga Stanislav

Verified by: conf.univ.,dr.
Sudacevschi Viorica

Chisinau 2017

TABLE OF CONTENTS

-EOF-

Basic Theory

The architecture of I8086 microprocessor

The I8086 microprocessor architecture consists of two sections:

1) The execution unit (EU)

2) The bus interface unit (BIU)

Execution Unit

The Execution Unit executes all instructions, provides data and addresses to the Bus Interface Unit and manipulates the general registers and the Processor Status Word (Flags register).

The 16-bit ALU performs arithmetic and logic operations, control flags and manipulates the general registers and instruction operands.

The Execution Unit does not connect directly to the system bus. It obtains instructions from a queue maintained by the Bus Interface Unit. When an instruction requires access to memory or a peripheral device, the Execution Unit requests the Bus Interface Unit to read and write data.

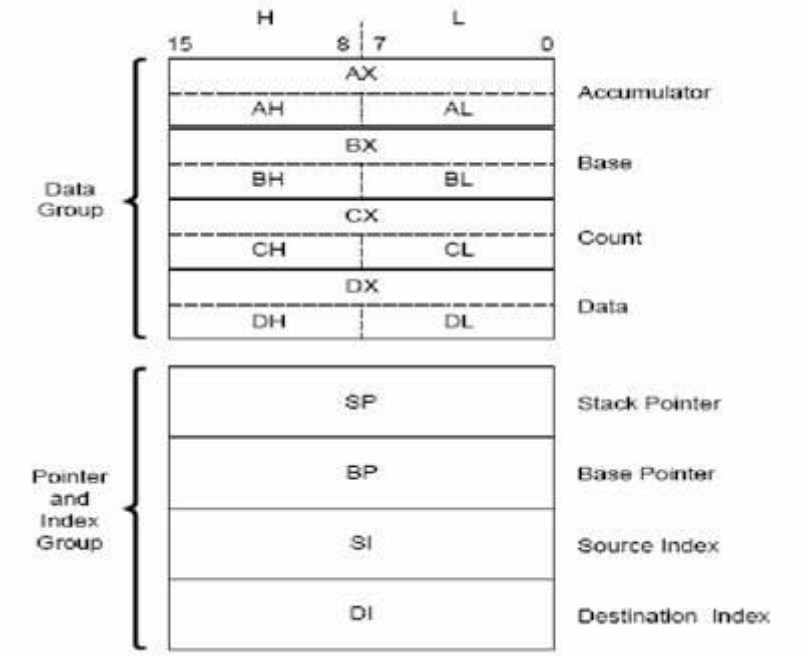
Bus Interface Unit

The Bus Interface Unit facilities communication between the EU and memory or I/O circuits. It is responsible for transmitting address, data, and control signals on the buses. This unit consists of the segment registers, the Instruction Pointer, internal communication registers, a logic circuit to generate a 20 bit address, bus control logic that multiplexers data and address lines, the instruction code queue (6 bytes RAM).

Registers set of I8086

1. *General Purpose Registers*

The CPU has eight 16-bit general registers. The general registers are subdivided into two sets of four registers. These sets are the data registers (also called the H & L group for high and low) and the pointer and index registers (also called the P & I group).



The data registers can be addressed by their upper or lower halves. Each data register can be used interchangeably as a 16-bit register or two 8-bit registers. The pointer and index registers are always accessed as 16-bit values.

SP - Stack Pointer : Always points to top item of the stack.

BP - **Base Pointer**: It is used to access any item in the stack;

SI - Source Index: Contains the address of the current element in the source string;

DI - Destination Index: Contains the address of the current element in the destination string;

Table 1. Implicit Use of General Registers

Register	Operations
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

2. *Segment registers*

The 8086 has a 20-bit address bus for 1 Mbyte external memory but inside the CPU registers have 16 bits that can access 64 Kbytes. The 8086 family memory space is divided into logical segments of up to 64 Kbytes each. The segment registers contain the base addresses (starting locations) of these memory segments.

CS (code segment) - points at the segment containing the current program.

DS (data segment)- generally points at the segment where variables are defined.

ES (extra segment)- extra segment register, it's up to a coder to define its usage.

SS (stack segment)- points at the segment containing the stack.

3. Special purpose registers

IP - the instruction pointer or program counter: Always points to next instruction to be executed. It contains the offset (displacement) of the next instruction from the start address of the code segment.

Flags Register - determines the current state of the processor. It is also called PSW (processor state word). From 16 bits are used only 9. **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.

All flags can be divided into condition (status) flags and control (system) flags.

Condition flags:

1. **0 bit -Carry Flag (CF)** - this flag is set to **1** when there is a carry (borrow) from the 8 or 16 bit in addition or subtraction operation. For example when you add bytes **255 + 1** (result is not in range 0...255). When there is no a carry or borrow this flag is set to **0**. It is also used to store the value of the MSB in shift operations.
2. **2 bit - Parity Flag (PF)** - this flag is set to **1** when there is even number of one bits in result, and to **0** when there is odd number of one bits. Even if result is a word only 8 low bits are analysed!
3. **4 bit - Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
4. **6 bit - Zero Flag (ZF)** - set to **1** when result is **zero**. For none zero result this flag is set to **0**.
5. **7 bit - Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. Actually this flag take the value of the most significant bit.
6. **11 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).

Control flags:

1. **8 bit - Trap Flag (TF)** System flag - Used for on-chip debugging (pas cu pas) when TF=1. In this case the interrupt is generated (int 1) which calls a special routine to show the state of internal registers. There are no instructions to change this flag. The content of PSW is written in one general Rg through the stack to can change it.

2. **9 bit - Interrupt enable Flag (IF)** System flag - when this flag is set to **1** CPU reacts (se permit) to interrupts on INTR input of the mp from external devices. When IF=0 interrupts are not allowed (masked). IF do not react to NMI (non maskable) interrupts and to internal interrupts performed by instruction INT. Instructions CLI (clear interrupt) and STI (set interrupt) are used to control this flag.
3. **10 bit - 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 (increment of SI and DI registers), when this flag is set to **1** the processing is done backward - decrement (instructions CLD and STD)

Main memory model

Instructions and data are stored in main memory. The (main) memory can be modeled as an array of millions of adjacent cells, each capable of storing a binary digit (bit), having value of 1 or 0. These cells are organized in the form of groups of fixed number of cells.

An entity consisting of 8 bits is called a byte, of 16 bits – a word, of 32 bits – a double word. It is, however, customary to express the size of the memory in terms of bytes. For example, if the size of a memory of a personal computer is 256 Mbytes, that is, 256 x 2^{20} =2^{28} bytes.

In order to be able to move a byte in and out of the memory, a distinct address has to be assigned to each byte.

The number of bits, l, needed to distinctly address M bytes in a memory is given by $l = log_2 M$. For example, if the size of the memory is 1 MB, then the number of bits in the address is $log_2(2^{20}) = 20$ bits. Alternatively, if the number of bits in the address is l, then the maximum memory size (in terms of the number of bytes that can be addressed using these l bits is $M = 2^l$.

The addressable memory of I8086 contains 2^{20} bytes (1 Mb). The physical addresses are within the range 00000-FFFFFh.

Locations 0H-7FH (128 bytes) and FFFF0-FFFFF (16 bytes) are reserved for special use (interrupts and system start after reset)

Any 2 neighbour bytes can store a word (16 bits). The smaller address contains the smaller byte. The address of the word is the address of its smaller byte. So, one address can be viewed as a byte address and a word address. This strategy to store data is called Little Endian (the opposite strategy is called Big Endian and it applied by Motorola, Spark and most RISC machines).

Memory segmentation

Segmentation provides a powerful memory management mechanism:

1. It allows programmers to partition their programs into modules that operate independently of one another.
2. Segments provide a way to easily implement object-oriented programs.
3. Segments allow two processes to easily share data.
4. It allows extending the addressability of a processor. In the case of the 8086, segmentation let Intel's designers extend the maximum addressable memory from 64KB to 1MB.

Disadvantage: Difficulties with physical address manipulation in programs.

Memory looks like a linear array of bytes. A single index (address) selects some particular byte from that array. Segmented addressing uses two components to specify a memory location: a segment value and an offset within that segment.

A full segmented address contains a segment component and an offset component **- segment:offset.**

On the 8086 through the 80286, these two values are 16 bit constants. On the 80386 and later, the offset can be a 16 bit constant or a 32 bit constant.

The size of the offset limits the maximum size of a segment. On the 8086 with 16 bit offsets, a segment may be no longer than $2^{16}=2^{6}*2^{10}$=64KB; it could be smaller (and most segments are), but never larger. The 80386 and later processors allow 32 bit offsets with segments as large as $2^{32}=2^{2}*2^{30}$=4GB.

The segment portion is 16 bits on all 80x86 processors. This lets a single program have up to 65,536 different segments in the program.

All memory space is considered as a set of 64 Kbyte size segments. The segments are defined for each application. Segments are considered to be independent and uniquely addressable. For each program can be currently addressed 4 segments using CS, DS, ES and SS. Memory segments can be different, can have common memory spaces or can even coincide. Segment rgs are initialised at the beginning of the application. They contain the base (low) address of the segment which is always a multiple of 16 (4 low bits are considered 0).

Instruction format

Assembly language is the symbolic form of machine language. Assembly programs are written with short abbreviations that represents the actual machine instruction called mnemonics.

The use of mnemonics is more meaningful than that of hex or binary values, which would make programming at this low level easier and more manageable.

Examples: **Mov -** move, **Add –** addition, **Sub –** subtraction, **Mul –** multiplication.

An assembly program consists of a sequence of assembly statements, where statements are written one per line. Each line of an assembly program is split into the following four fields: label, operation code (opcode), operand, and comments.

Label (Optional)	Operation Code (Required)	Operand (Required in some instructions)	Comment (Optional)

Labels are used to provide symbolic names for memory addresses. A label is an identifier that can be used on a program line in order to branch to the labeled line. It can also be used to access data using symbolic names. The operation code (opcode) field contains the symbolic abbreviation of a given operation. The operand field consists of additional information or data that the opcode requires. The operand field may be used to specify constant, label, immediate data, register, or a memory address. The comments field provides a space for documentation to explain what has been done for the purpose of debugging and maintenance. In I8086 instruction consists from one to six bytes.

According to the length of the instructions exists two types of ISA:

1. With fixed length instructions (commonly used in RISC architectures)
2. With variable length instructions (commonly used in CISC architectures)

The advantage of using variable length instructions is that they reduce the amount of memory space required for a program. In I8086 instructions are from one byte to a maximum of 6 bytes in length.
The advantage of fixed length instructions is that they make the job of fetching and decoding instructions easier and more efficient, which means that they can be executed in less time than the corresponding variable length instructions.
Instructions can be classified based on the number of operands as: three-address, two-address, one-address, and zero-address.
Examples:

3 addresses	Add x,y,z	(z)=(x)+(y)
2 addresses	Add ax,bx	(Ax)=(ax)+(bx)
1 addresses	Mul bl	(Ax)=(al)*(bl)
0 addresses	Push bx	Top of the stack ← (bx)

Three-address instruction formats are not common, because they require a relatively long space to hold all addresses.

In two-address instruction one address is an operand and also a result.

In one-address instruction a second address is implicit. Usually it is the accumulator AX. It is used for one operand and the result.

Zero-address instructions are applicable to stack memory and use as address the content of SP (top of the stack).

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in more primitive instructions, which require a less complex CPU. It also results in instruction of shorter length. On the other hand programs contain more total instructions and have a longer execution time. Another problem: with one-address instructions, the programmer has available only one general-purpose register – the accumulator, with multiple address instructions it is common to have multiple general-purpose registers. Because register references are faster than memory references this speeds up execution. Most contemporary machines employ a mixture of two- and three-address instructions.

Addressing Modes

The different ways in which operands can be addressed are called the addressing modes. Addressing modes differ in the way the address information of operands is specified.

EA - actual (effective) address (EA) of the location containing the operand;

The addressing modes available in 8086 are:

1. Immediate Addressing Mode:

According to this addressing mode, the value of the operand is (immediately) available in the instruction itself.

Operand=A,

where A - the content of the address field in the instruction

Typically immediate operand represents constant data (a byte or word). The number is stored in two’s complement form.

Examples:

```
mov al, 48 ; load 30H in AL;
mov cx,2056H
xor si,1 ; invert LSB in SI register;
and al,80H ; highlight MSB of AL
or di, 8000H ; set to 1 MSB of DI
```

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantages: the size of the number is restricted to the size of the address field; a change in the value of an operand requires a change in every instruction that uses the immediate value of such an operand.

2. Register Addressing Mode:

To access the content of the register it is necessary to specify the name of the register. The eight and 16 bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be of the same size.

```
mov  ax, bx ;Copies the value from BX into AX
mov dl, al ;Copies the value from AL into DL
mov ax, ax ;Yes, this is legal and it performs nothing!
add bx,di; bx=bx+di
sub cl,ah ; cl=cl-ah
```

Advantage: the registers are the best place to keep often used variables. Instructions using the registers are shorter and faster than those that access memory.

Disadvantage: limited address space and the limited number of general purpose registers.

3. Indexed Addressing mode: (9 clock c)

In this addressing mode, the address field contains a main memory address and the register, called the index register, contains a positive displacement from that address.

The indexed addressing modes use the following syntax:

```
mov al, disp[si] mov al, [si+disp]
```

```
mov al, disp[di] mov al, [di+disp]
```

The displacement field can be a signed eight bit constant or a signed 16 bit constant.

In such addressing EA=disp+[SI] or [DI]. It is useful in case of iterative operations, when disp is the address of the first element and SI or DI value specified the element. First they are initialised to 0 and after each operation the index register is incremented.

8086 Instruction Set

<table>
<tr>
<td>MOV</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

SREG, memory
memory, SREG
REG, SREG
SREG, REG</td>
<td>Copy operand2 to operand1.

The MOV instruction cannot:
•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.</td>
</tr>
</table>

CALL	procedure name
	label
	4-byte address
		Transfers control to procedure, return address is (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 (this is a far call, so CS is also pushed to stack).
		
		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

XOR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand. These rules apply: 1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0 Example: MOV AL, 00000111b XOR AL, 00000010b ; AL = 00000101b RET

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

AND	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical AND between all bits of two operands. Result is stored in operand1. 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

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

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
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

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
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

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

<table>
<tr>
<td>ADD</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Add.

Algorithm:

operand1 = operand1 + operand2

Example:
MOV AL, 5 ; AL = 5
ADD AL, -3 ; AL = 2
RET</td>
</tr>
<tr>
<td>SHR</td>
<td>memory, immediate
REG, immediate

memory, CL
REG, CL</td>
<td>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</td>
</tr>
<tr>
<td>LOOP</td>
<td>label</td>
<td>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</td>
</tr>
</table>

8086 Interrupts Set

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**.

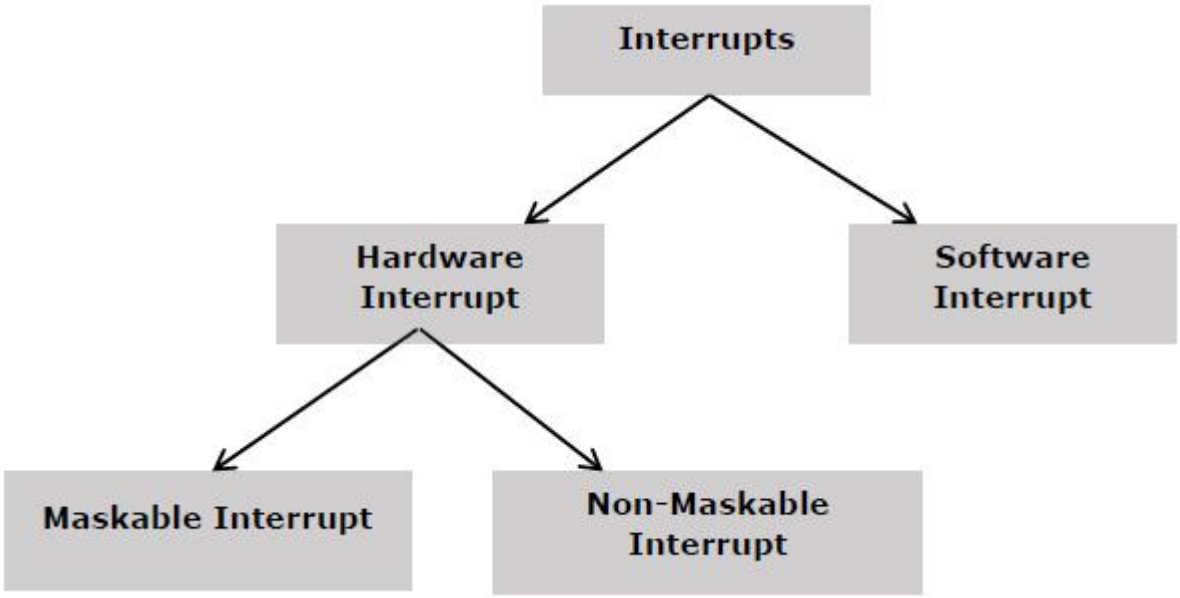
INT 21h / **AH = 1** - read character from standard input, with echo, result is stored in **AL**.
if there is no character in the keyboard buffer, the function waits until any key is pressed.

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

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:



Floating point multiplication

Variant 5:

Mantissa - 16 bits
Expontent - 8bits
[Algorithm 1] - left shifting

Steps:

1) Calculate the sign of m_z

$$Sg\ m_X \oplus Sg\ m_Y = Sg\ m_Z$$

2) Calculate the exponent

$$e_Z = e_X + e_Y$$

3) Determine the absolute value of m_x and m_y

(positive sign mantissa remains the same, negative sign one gets every bit inversed and add +1 bit)

4) Perform the algorithm 1 multiplication

5) Normalize m_z if necessary, i.e.:

Normalize the result so that there is no repeating bit before and after the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrements the exponent by 1.

Example:

m_x = 0.000 0100 0000 0001

m_y = 1.010 0010 0000 0100

e_x = 0.000 0110

e_y = 0.000 0001

1)Sg m_z = 0 xor 0 = 0

2) 0.000 0110+
0.000 0001
0.000 0111

e_z = 0.000 0111

3) $|m_x|$ = 0.000 0100 0000 0001

$|m_y|$ = 0.101 1101 1111 1100

4) Multiplying mantissas:

0000 0100 0000 0001 *
0101 1101 1111 1100
0101 1101 1111 1100 +
01 0111 0111 1111 00
0000 0001 0111 1000 0100 1101 1111 1100

m_z = 0000 0001 0111 1000 0100 1101 1111 1100

5) Normalization:

We can see m_z may be normalized by 5, thus:

Normalized mantissa: m_z = 0.101 1110 0001 0011 0111 1111

Exponent: e_z = 0.000 0111 - 0.000 0101 = *(calculation below)* = 0.000 0001

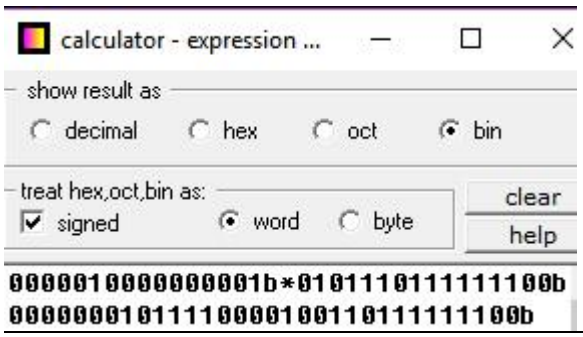
```
0.0000111+
0.1111111
0.0000101+
0.1111111
0.0000100+
0.1111111
0.0000011+
0.1111111
0.0000010+
0.1111111
0.0000001
```

Answer:

m_z = 0.101 1110 0001 0011 0111 1111 0000 0000 (32bit)

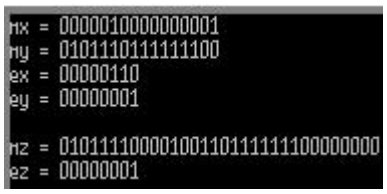
e_z = 0.000 0001 (8 bit)

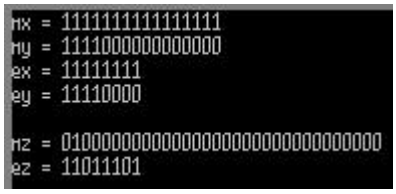
Verification:

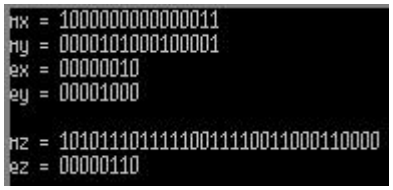


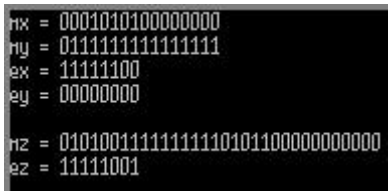
Correct! Since the denormalized answer is 00000001011110000100110111111100

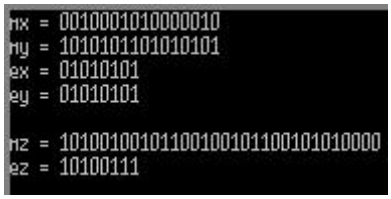
Examples of output in I8086:

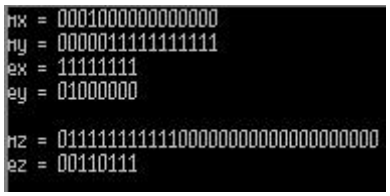


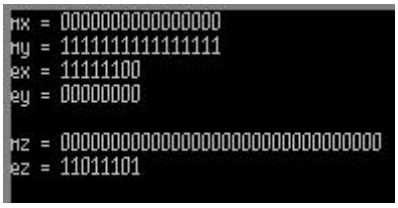


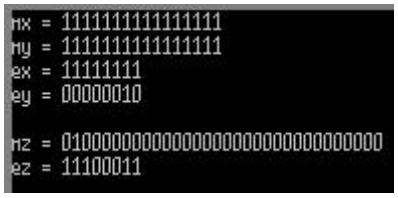


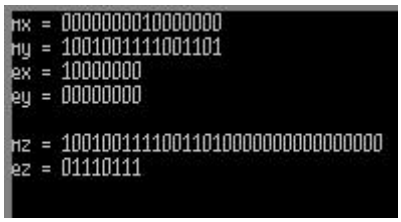


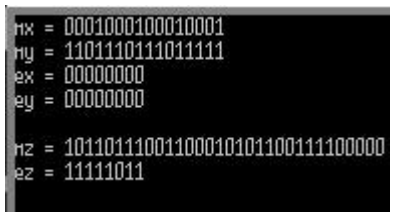












Conclusion:

Assembler helps understand how to make more efficient programs.
Yet, there are lots of multiplication algorithms that may be implemented more efficient than mine, however, this was a great practice that helps understand how to work at machine level.
i8086 Assembler represents the language closest to the machine, making the bridge between a person and hardware. 8086 architecture is very old so having limited tools in 8086 is a benefit, since it makes it very user-friendly , and new users that want to program at machine level benefit from this, since nowadays processors have similar principles of function.

Annex

The code in I8086:

```
.MODEL SMALL
.DATA

sign db 0     ;sign of resulting mantissa
mod_mx dw ?   ;abs( Mx )
mod_my dw ?   ;abs( My )
mx dw ?
my dw ?
mz1 dw ?
mz2 dw ?
mz dw ?,?

ex db ?
ey db ?
ez db ?

show_mx db "mx = $"
show_my db 0Ah, 0Dh, "my = $"
show_ex db 0Ah, 0Dh, "ex = $"
show_ey db 0Ah, 0Dh, "ey = $"

output_mz db 0Ah, 0Ah, 0Dh, "mz = $"
output_ez db 0Ah, 0Dh, "ez = $"
;----------------------------------------
.CODE            ;INITIALIZE DATA SEGMENT
start:

 mov ax,@data
 mov ds,ax

;----------------------------------------
                        ;UTILITY MACROS
;========================================

mod MACRO operand                ;macro for calc. abs(mantissa)
    local negate, exit_mod       ;local labels for avoiding ambiguity

    test operand, 8000h
    jnz negate                  ;if 1, negate the mantissa
    jmp exit_mod

    negate:
       neg operand

    exit_mod:
       nop

mod ENDM

;----------------------------------------
mov dx, offset show_mx    ;DISPLAY MESSAGES
    mov ah, 09          ;AND INPUT CHAR'S
    int 21h
```

```
    input_macro 16         ;input Mx
    mov mx, bx

    mov dx, offset show_my
    mov ah, 09
    int 21h

    input_macro 16         ;input My
    mov my, bx

    mov dx, offset show_ex
    mov ah, 09
    int 21h

    input_macro 8          ;input Ex
    mov ex, bl

    mov dx, offset show_ey
    mov ah, 09
    int 21h

    input_macro 8          ;input Ey
    mov ey, bl

    xor ax, ax
    xor bx, bx
;-----------------------------------------
    CALL calc_sign          ;STEP 1: Calculate sign

;-----------------------------------------
    xor ax, ax              ;STEP 2: Compute exponent (Ez)
    mov ah, ex
    add ah, ey
    mov ez, ah
;-----------------------------------------
                           ;STEP 3: Determine mantissas absolute values
    xor ax,ax
    xor bx,bx
    mov ax,mx
    mov bx,my

    mod ax
    mov mod_mx,ax
    xor ax,ax

    mod bx
    mov mod_my,bx
    xor bx,bx

;-----------------------------------------
                           ;STEP 4: Multiply modulo mantissas
    CALL add_mantissa
;-----------------------------------------
                           ;STEP 5: Normalization (we normalise the modulo
then we put the sign)
    call normalization
```

```
;-----------------------------------------
 xor dx,dx                      ;OUTPUTS

    mov dx, offset output_mz
    mov ah, 09
    int 21h

    call output_mantissa_32


    mov dx, offset output_ez
    mov ah, 09
    int 21h

    xor bx,bx
    mov bl, ez
    call output_exponent


    jmp ENDING

;-----------------------------------------
;IN/OUT-PUT MACROS & OTHER UTILITY PROC
;-----------------------------------------

input_macro MACRO size                  ;macro for user input
    local input
    xor ax, ax                         ;clear the registers
    xor bx, bx
    xor cx, cx
    mov cx, size
    mov ah, 01h                        ;loop N times, as we input N bit
registers
    input:
        int 21h                        ;input the character
        sub al, 30h                    ;al keeps the ascii code of char:
                                       ;30 = 0, 31 = 1
        add bl, al                     ;subtracting 30 to obtain either
                                       ;0 or 1
        shl bx, 1                      ;add that  bit to the result
        loop input                     ;shift the result left
    rcr bx,1
input_macro ENDM
;-----------------------------------------


output_mantissa_32 proc
    mov ah, 02h
    mov bx, mz[2]
    mov cx, mz
    mov si, 32
    clc
outputLoop:
    test bx, 8000h
    jnz out1
    mov dl, 30h
    jmp outFinal
out1:
    mov dl, 31h
outFinal:
    int 21h
```

```
    shl cx, 1
    rcl bx, 1
    dec si
    cmp si, 0
    jnz outputLoop
    ret
output_mantissa_32 endp
;----------------------------------------


output_exponent proc
    mov cx, 8

    output_loop:
        shl bl, 1
        jc output_one
        mov dl, 30h
        jmp printing

        output_one:
            mov dl, 31h

        printing:
        mov ah, 2
        int 21h
        loop output_loop
    ret
output_exponent endp

calc_sign PROC                   ;procedure for checking the sign of M
    xor ax, ax
    mov ax, mx
    xor ax, my
    test ax, 8000h
    jnz sign_neg                ;if 1,the resulting mantissa is negative
    jmp sign_exit

    sign_neg:
        mov sign, 1              ;store the sign in a var

    sign_exit:
    RET
calc_sign ENDP
;----------------------------------------

normalization PROC       ;procedure for normalizing the mantissa
    mov cx, 32
                         ;this will be used for the case when
                         ;we have mantissa 0 to avoid infinite loop
    mov ax, mz1
    mov bx, mz2
    mov dl, ez
    clc
normLoop:
    test ax, 8000h
    jnz foundOne
    shl bx, 1
    rcl ax, 1
    sub dl, 1
    loop normLoop
```

```
foundOne:
    shr ax, 1                 ;we shift 1 bit to the right to put the sign
    rcr bx, 1
    mov cl, sign
    cmp cl, 1
    jne normFinish
    not ax
    not bx
    clc
    add bx, 1
    adc ax, 0

normFinish:
    mov mz, bx
    mov mz[2], ax
    add dl, 1
    mov ez, dl
    ret
normalization ENDP
;=========================================


add_mantissa PROC
    xor ax, ax
    xor bx, bx
    xor cx, cx

    mov ax, mod_mx
    mov bx, mod_my
    xor dx, dx
    mov si, 16

addLoop:
    clc
    test bx, 1
    jz dontAdd
    add cx, ax
    adc dx, 0

dontAdd:
    shr cx, 1
    rcr dx, 1
    shr bx, 1
    dec si
    cmp si, 0
    jne addLoop

    mov mz1, cx
    mov mz2, dx
    ret
add_mantissa ENDP
;-----------------------------------------
              ;PRESS ANY KEY TO CONTINUE
ENDING:
mov  ah,7
int  21h

MOV AX, 4C00H
INT 21H
END start
```