

The assembly

Jérôme POUILLER

Pierre-Alexandre ENTRAYGUES

June 28, 2002

Contents

1	How to program in assembly?	3
2	How work a program and how to compile it?	4
2.1	How work programs in generally?	4
2.1.1	The compilation	4
2.1.2	The assembly	4
2.1.3	What is a debugger?	4
2.1.4	What is a disassembler?	5
2.2	What I need to make an executable with my source?	5
3	Representation of binary numbers and variables	6
3.1	Reminding of mathematics	6
3.2	The constants	6
3.3	The variables	7
3.4	The character chains	7
4	Registers and memory	9
4.1	What are type of memory	9
4.2	What is registers?	10
4.3	What are type of registers?	10
4.3.1	Segment registers	10
4.3.2	Pointer registers	10
4.3.3	Index Registers	11
4.3.4	General purpose register	11
4.4	How to store data in memory	11
4.4.1	Immediate mailling	11
4.4.2	Direct mailling	12
4.4.3	Based mailling	12
4.4.4	Based and displacement mailling	12
4.4.5	Based and indexed mailling	12
4.4.6	Based, displacement an indexed mailling	12
4.4.7	Byte number of data	13
5	Program structure and basic instructions	14
5.1	Program structure	14
5.2	Basics instructions	15

6	Conditionnal jumps	16
6.1	Jumps	16
6.2	Flags register	17
6.3	CMP Instruction	18
6.4	How to use the classic loops of evolved languages	19
6.4.1	The IF...THEN...ELSE...	19
6.4.2	The FOR	19
6.4.3	The WHILE	19
6.4.4	The REPEAT	19
6.4.5	The CASE	20
7	Procedures call	21
7.1	Precepts of stack	21
7.2	Optimization	22
7.3	Procedures call	22
7.4	The interruptions	22
8	Input and Output Instructions	24
9	Win32 Assembly	25
10	Application	27
11	Conclusion	29

Chapter 1

How to program in assembly?

Programing in assembly isn't very intuitive : the instructions aren't actions that you can represent to yourself : if you want to write a text, you don't have an instruction to do it , but you have to use 4 instructions.

In Pascal :

```
writeln('Hello world !');
```

In Assembly :

```
text DB 'Hello world !', '$'  
MOV AH,09h  
MOV DX,OFFSET text  
INT 21h
```

That's why you can't program in assembly like you program in others languages like Pascal, Basic or C. In more, you must be careful about a lot of system parameters (stack overflow,...). You must have a method.

The assembly is a langage said of lowly level. This means that you program directly your microprocessor. So, you can optimize your code. Generaly, an assembly code is two to for faster that the same code in Pascal but, this language is different for each class of computer. So, a source code programed for Dos/Windows, using the ix86 instructions of Intel can't be compiled on a Machintosh using a Motorola microprocessor.

Assembly is very useful to program processors which have not evolved language like Z80, 6809.... So it is very used, because it is the only solution to program on-board systems like celular phones or calculators.

Chapter 2

How work a program and how to compile it?

2.1 How work programs in generaly?

2.1.1 The compilation

When you program in evolved languages, you use a compiler when you want to execute your program. This compiler traduces your program on assembly language and makes the translation between evolved language and assembly code. Then your compiler call an assembler.

2.1.2 The assembly

The assembler make traduction between assembly instruction, called mnemonic, and hexadecimal code. For exemple ADD instruction is a mnemonic. The x86 processors have about 200 mnemonics. Indeed, each instruction in assembly can be converted in hexadecimal.

Example : `MOV AX, 0012h` is traduced by `A1 00 12` in hexadecimal. When you run a compiled program, even if it was not programed in assembly, your microprocessor read hexadecimal code. The different instructions use different numbers of clock cycle. A memory access takes more time than a register access. So, try to optimise your code.

2.1.3 What is a debugger?

A debugger is a program showing you what your others programs do. So, you can show with it what is in the memory and what are in registeries... One is include with your Dos or Windows system. You can find it for Windows in our `C:\windows\system path`. If you want one with more capacities, you can download Soft Ice, a powerful debugging tool.

2.1.4 What is a disassembler?

The assembly is very near from the machine language. That's why, it's not very difficult to convert an executable to an assembler source file. A disassembler will traduct hexadecimal code in assembly source. Dasm is one of the best disassemblers.

2.2 What I need to make an executable with my source?

For x86, you have two programs: MASM and TASM. You can download them on the Internet. If you compile your program on Dos/Windows, your program becomes an executable usable only by Dos/Windows.

To compile an assembly program, you must have:

- an .ASM file containing the source code of your program
- a compiler like TASM
- a linker like TLINK

Put your .ASM file in the same folder than TASM. Run TASM with a command line with the name of your .ASM in argument. If they are no error of compilation, TASM will generate a .OBJ file. This is a compiled version of your source code. Run TLINK with a command line with the name of your .OBJ in argument. TLINK will generate the headers and create an executable.

Exemple :

```
D:\tasm>TASM EX Turbo Assembler Version
4.1 Copyright (c) 1988, 1996 Borland International
```

```
Assembling file:  EX.ASM
Error messages:   None
Warning messages: None
Passes:          1
Remaining memory: 372k
```

```
D:\tasm>TLINK EX
Turbo Link Version 7.1.30.1. Copyright (c)
1987, 1996 Borland International
```

NB:To compile an assembly program to .COM file. Run a86.exe with the name of your .ASM in argument.

Chapter 3

Representation of binary numbers and variables

3.1 Reminding of mathematics

The machine code is interpreted by the microprocessor, that's why when you program in assembly, you always reason in binary, or more exactly in hexadecimal, more useful than binary.

Numbers are stored in binary, base two. But Assembly understands others data format. Here are the different data recognized by the Assembly language.

- a bit : One bit is the simplest piece of data that exists. It's either a one or a zero.
- a nibble : The nibble is four bits or half a byte.). This is the basis for the hexadecimal (base 16) number system which is used as it is far easier to understand. Hexadecimal numbers go from 1 to F and are followed by a h to state that they are in hex. i.e. Fh = 15 decimal. Hexadecimal numbers that begin with a letter are prefixed with a 0 (zero).
- a byte : A byte is 8 bits. Its maximum value is FFh (255 decimal). A byte is two nibbles. In hexadecimal it is represented by two digits. The byte is also that size of the 8-bit registers which we will be covering later.
- a word : A word is two bytes. A word has a maximum value of FFFFh (65,536). This is the size of the 16-bit registers.
- a double word : A double word is composed of two words. So, it makes 32 bits.

So : 1 DOUBLE WORD = 2 WORD = 4 BYTES = 8 NIBBLES = 32 BITS

3.2 The constants

Those are values defined by an instruction. Those won't be modified by the program. You can use binary, hexadecimal and text constants too. You will use EQU to define constants.

Example :

```
const1 EQU 16          ;"const1" is equal to 16
const2 EQU 4+23*7     ;"const2" is equal to 322 (4+23*7)
const3 EQU 'Epita'    ;"const3" is equal to Epita
const4 EQU 0101010b  ;"const4" is equal to 42
const5 EQU 0AFh       ;"const5" is equal to 175
                    (note that you have to put a zero
                    on the head of your hexadecimal expression)
```

3.3 The variables

Variables are memory zones which are reserved at the time of the compilation: they will be usable by the program. You can define variable uninitialized.

DB : This instruction initialize one byte memory zones, you can put start values to your variables.

Examples

Declaration of a variable `boo` `DB 0` ;`boo` is initialized to 0

Declaration of three variables `ABC` `DB 1,2,3` ;3 element are declared

Declaration of one variable uninitialized `other` `DB ?` ;declaration of an uninitialized variable

DW : This instruction initialize one word or two bytes memory zones, you can put start values to your variables.

Example

Declaration of a variable `year` `DW 2000`

We are obliged to use a word, because 2000 is higher than the maximum of a byte (255)

DD : This instruction initialize a double word or four bytes memory zones, you can put start values to your variables.

Example

Declaration of a variable `dword` `DD 01AD5F7A6h`

We are obliged to use a double word, because 01AD5F7A6h is higher than the maximum of a word 0FFFFh (note the zero at the beginning)

3.4 The character chains

To declare character chains, you have to use the DB declaration instruction and finish your sentences by a `$` .

Example

```
Declarations of characters chains hello DB 'Hello world !', '$'  
messascii DB 'Here are a message with special characters', '1', '219', '  
$'
```

Please note that: When you initialize your variable with DW or DD, the byte or the word with the leastest weight is the first stored.

DUP : The DUPLICATION of the variables With DUP, you declare x instance the same variables. It is used like that : var D[b,w,d] x DUP values Example
Declaration of a table of 12 elements initialized table DB 3 DUP (1,2,3,4)
The table contains 1 2 3 4 1 2 3 4 1 2 3 4

Chapter 4

Registers and memory

4.1 What are type of memory

You have different types of memory: RAM, where you can store temporary data and ROM, where datas are permanent. In most of the systems, ROM addresses are the least, then come RAM addresses.

FFFF x000	RAM
0000	ROM

In a PC, you have many types of memory:

FFFF:FFFF	XMS
0010:0000	
000F:0000	ROM Bios (Read only)
000D:0000	EMS
000C:0000	Bios Memory Relocation
000A:0000	Video Memory
0000:0000	Conventionnal Memory

At the beginning, programmers only used conventionnal memory. They can now use XMS and EMS in their program. Between 000F and 0010, there is ROM

BIOS, so you can't write it. BIOS relocate him in RAM because RAM is faster. Memory Video is uses to display text or graphic to screen.

4.2 What is registers?

In processors, you have little memory bloc where you can store information. They are called Registers. Registers are very faster than memory, so try to use it a maximum. There are three sizes of registers : 8 bit, 16 bit, 32 bit (since the 386).

4.3 What are type of registers?

There are four types of registers:

- Work or General Purpose registers
- Index registers
- Pointer or offset registers
- Segment registers

Segments and offsets registers are both 16 bits registers. At the beginning, the microprocessor designers decided that nobody will never need to use more than one megabyte('640kb could be enough fo evryone' - Bill Gates - 1986). But to access to one megabyte, 20 bits are needed. Registers have 16 bits. That's why they decided to create an other register : the segment registers. Also two registers are used for adresssing. An adress is represented by : SEGMENT:OFFSET.

4.3.1 Segment registers

- CS : Code segment : contains the address of the segment memory containing the instructions of the program.
- DS : Data segment : contains the address of the segment memory containing the data defined in the program.
- SS : Stack segment : contains the address of the segment memory containing the stack.
- ES, FS, GS : Extra Segments : contains the address of an other segment memory, which can receive other data. FS and GS exist only on 386+.

4.3.2 Pointer registers

- IP : Instruction pointer, associated to the CS register (CS:IP) indicate the next instruction to execute. This register won't be directly modified; it could be indirectly modified by the jump instructions, the under-programs and by the interruptions.
- BP : Base pointer, associated to the segment registry SS (SS:BP) to access to the stack data when you call under-programs.

- SP : Stack Pointer, associated to the segment registry SS (SS:SP) to indicate the segment of the stack.

4.3.3 Index Registers

- SI : Source Index, associated to the DS registry, they are used to make the 32 bits address. It principally used to work on a chain of character. DS:IP
- DI : associated to DS or ES to work on character chains. DS:DI or ES:DI

4.3.4 General purpose register

These are four 32 bit registers. But, only 16 bits are generally used. They can be decomposed in 16 and 8 bits: Example: In EAX 32 bit register, you have :

EAX		
	AX	
	AH	AL

You can make operations on every registers or subregisters

- AX : Accumulator, used at the time of arithmetic operations.
- BX : Base registry
- CX : Counter registry , used in loop
- DX : Data registry : input/output and used by multiply and divide

4.4 How to store data in memory

In assembly, you have different solutions to store data in the memory. The different solutions are important, because translation for a same instruction with different addressing types will be different. In addition, certain addressing types are faster than others. All instructions don't accept all addressing systems.

4.4.1 Immediate mailing

It is the most elementary mailing. It is made between a constant and a register. Example:

```
MOV AL, 12
MOV BX, 1633
```

4.4.2 Direct mailling

It allow to make operations between register and memory compartement. Example:

```
Id1 DW 1633
Id2 DB 12
...
MOV AX, Id1 ; you indicate adresse of memory compartement
MOV AL, Id2 ; MOV AL, Id1 makes an error because AL is a 8 bits
              ; register and Id1 is a data of 16 bits.

MOV Id1, AX
```

4.4.3 Based mailling

Allow to make operation between a register and a memory compartement pointed by DS:register (generaly BX). Example:

```
Id1 DW 1633
...
MOV BX, offset Id1 ; offset instruction return adresse of a label
MOV AX, [BX]       ; equivalent of MOV ax, [DS:BX]. Ax will contain 1633
```

If you use BP for base pointer, SS will used in stack segment.

4.4.4 Based and displacement mailling

Allow to use a constant in addition of base. It is very usefull in table.

```
Id1 DW 2000, 2001, 2002, 2003
...
MOV BX, offset Id1
MOV AX, [bx + 2] ; ax contain the 3rd element of Id1 : 2002
```

4.4.5 Based and indexed mailling

Allow to use an index in addition of base. There too, it is very useful to use table.

```
Id1 DW 2000, 2001, 2002, 2003
...
MOV BX, offset Id1
MOV SI, 2
MOV ax, [BX+SI] ; AX contain the 3rd element of Id1 : 2002
                ; can be writed MOV AX, [bx][si]
```

4.4.6 Based, displacement an indexed mailling

It is a combinaison of precedants types of mailling

```
Id1: DW 2000, 2001, 2002, 2003
...
MOV BX, offset Id1
MOV SI, 2
MOV AX, [BX + SI + 1] ; BX contain 2003
```

This mode is the slower, so, use it carefully.

4.4.7 Byte number of data

In some case, it impossible for processor to know if you work in 8, 16 or 32 bits; so you will use PTR instruction.

Example:

```
Id1: DW 2000h, 2001h, 2002h, 2003h
...
MOV AX, BYTE PTR Id1 ; AX contain 20h
MOV AX, WORD PTR Id1 ; AX contain 2000h
MOV EAX, DWORD PTR Id1 ; AX contain 20002001h
```

Chapter 5

Program structure and basic instructions

5.1 Program structure

An assembler program is made of three parts:

- Header
- Variables declaration
- Code

The header contains the model style of your program. This is preceded by .model. It declares the different segments used by your program.

```
PROG1ST.ASM      ; This is a simple program which displays "Hello World!"
                  ; on the screen.

.model small
.stack           ; You declare there your stack (seen later)
.data           ; You declare there your data

Message db "Hello World\$ " ; only one data : the message to be display

.code

start:
MOV DX,OFFSET Message ; offset of Message is in DX mov ax,SEG Message
MOV AX,SEG Message    ; segment of Message is in AX mov ds,ax
MOV DS,AX              ; DS:DX points to string

MOV AH,9              ; function 9 - display string \
INT 21h               ; call dos service           \ seen
MOV AX,4c00h          ;                               / later
INT 21h               ; return to dos DOS          /

END start             ;end here
```

Data, Stack and Code can be in different segments. If you make a .COM program, all program is stored in the same segment. Whereas in a .EXE the part of program is stored in different segments. You can notice that the assembly header is quite heavy. So, we advise you to code in assembly inside an evolved interface language like Pascal. It will manage the different segments. To make assembly inside Pascal, use `ASM ..<Your code>.. END;`.

5.2 Basics instructions

Assembly has a lot of instructions but there are only twenty that you have to know and that you will use often. The instructions are designed like this : Mnemonic [arg1[, arg2]] [;comments]. You can use in argument register, memory address or constant.

The mnemonic is an instruction, it is generally do of three or four letter. In generaly, if there are two arguments, the first argument will receive the result after execution. Conventionally, we write the mnemonic in capital.

Most important instructions

- MOV (or LD (LoaD) in motorola processor) : It is the most used instruction . It copies (and not moves) the first argument into the second.
- All logic instructions : NOT, AND, OR, XOR
- All bits control instructions : Shifts: SHR, SHL and Rotate: ROL, ROR:
- Simple arithmetic instructions : NEG (make two to two complement), ADD (Addition), ADC (Addition with carry), SUB (Substrac), MUL (Multiplication), IMUL (Signed multiplication), DIV (Division), IDIV (Signed division): IMUL, DIV and IDIV are only present in high level processors
- INC (Increment) and DEC (Decrement)
- NOP : This makes nothing. It can be useful.

In latest processors, you have instructions to manipulate Float number. You can also use evolved instruction to make loop faster. For example, it exists an MMX instruction which can make 5 additions in parallel.

Chapter 6

Conditionnal jumps

6.1 Jumps

You can jump in your program using JMP (or BRA in Motorola CPU). You use labels to name your branch. The label, like in most of programming language, can't be begin with a digit and can only be made with 0-9, a-z, A-Z and `.`. In most of processors, there is three type of jumps: far, near and short jumps. The assembler will choose automatically what type of jump, it must use. In the case of long jump, the assembler will traduct Label with the corresponding adress:

```
0010:0003 ... will be traduct by ...
Branch:
0010:0006 ...
.....
0013:0008 ...
0013:000A JMP Branch          EA 00 10 00 06
0013:000D ...
```

In cases of near and short jumps, the assembler will traduct jump by a number of offset to jump. These jumps are called relatives jumps. These jumps are short because, you can only jump 127 offsest for short jump and 32768 offsets for near jump.

```
Branch:
0006 ... will be traduct by ...
0008 ...
000A JMP Branch          EB FA    (FA = -6)
000C ...
```

We can notice that short jumps use one byte less and less time than far jumps. In all case, a jump can be considerate like an operation (a subtraction or an addition) on the code pointer. JMP and BRA are unconditionnal jumps, but, processors integrate a conditionnal jumps system. The program is plug only if the condition is true. The condition can be determined by the flags register.

6.2 Flags register

There is in processors a special register. It is called flags register. It is modified when an operation is executed and it allow to know information about the previous result. You can use these informations for conditionnal jumps. For Example, if you execute

```
boucle:
mov ax, 0
jz boucle
```

The program will jump. There is 17 bits in the flag register:

Overflow is set if the result is positive while it should be negative. Carry is set if result surpasses maximal number of bit. Overflow can be compared to an 'signed carry'.

```
|11|10|F|E|D|C|B|A|9|8|7|6|5|4|3|2|1|0|
| | | | | | | | | | | | | | | | | '--- CF Carry Flag set if
| | | | | | | | | | | | | | | | | '--- 1
| | | | | | | | | | | | | | | | | '--- PF Parity Flag set if result if pair
| | | | | | | | | | | | | | | | | '--- 0
| | | | | | | | | | | | | | | | | '--- AF Auxiliary Carry Flag set if forth bit is set
| | | | | | | | | | | | | | | | | '--- 0
| | | | | | | | | | | | | | | | | '--- ZF Zero Flag set if result is zero
| | | | | | | | | | | | | | | | | '--- SF Sign Flag set if high order bit is set
| | | | | | | | | | | | | | | | | '--- TF Trap Flag
| | | | | | | | | | | | | | | | | '--- IF Interrupt Flag set if you are in an interrupt
| | | | | | | | | | | | | | | | | '--- DF Direction Flag
| | | | | | | | | | | | | | | | | '--- OF Overflow flag set result is to larger
| | | | | | | | | | | | | | | | | '--- IOPL I/O Privilege Level (286+ only)
| | | | | | | | | | | | | | | | | '----- NT Nested Task Flag (286+ only)
| | | | | | | | | | | | | | | | | '----- 0
| | | | | | | | | | | | | | | | | '----- RF Resume Flag (386+ only)
'----- VM Virtual Mode Flag (386+ only)
```

Auxiliary Carry Flag is used if you work in 4 bit. It is very usefull if you work with BCD (Binary coded Decimal). If trap flag is set, processor calls an interruption after evry instruction. It is very useful to make step by step in a debugger.

So, you can make conditionnal jump:
Mnemonic Meaning Jump Condition

```
JMP Unconditional Jump unconditional
JE Jump if Equal ZF=1
JZ Jump if Zero ZF=1
JS Jump if Signed (signed) SF=1
JC Jump if Carry CF=1
JO Jump if Overflow (signed) OF=1
JP Jump if Parity PF=1
```

JA Jump if Above CF=0 and ZF=0
 JAE Jump if Above or Equal CF=0
 JG Jump if Greater (signed) ZF=0 and SF=OF
 JGE Jump if Greater or Equal (signed) SF=OF

 JB Jump if Below CF=1
 JBE Jump if Below or Equal CF=1 or ZF=1
 JL Jump if Less (signed) SF != OF
 JLE Jump if Less or Equal (signed) ZF=1 or SF != OF

 JNE Jump if Not Equal ZF=0
 JNZ Jump if Not Zero ZF=0
 JNS Jump if Not Signed (signed) SF=0
 JNC Jump if Not Carry CF=0
 JNO Jump if Not Overflow (signed) OF=0

 JNA Jump if Not Above CF=1 or ZF=1
 JNAE Jump if Not Above or Equal CF=1
 JNG Jump if Not Greater (signed) ZF=1 or SF != OF
 JNGE Jump if Not Greater or Equal (signed) SF != OF

 JNB Jump if Not Below CF=0
 JNBE Jump if Not Below or Equal CF=0 and ZF=0
 JNL Jump if Not Less (signed) SF=OF
 JNLE Jump if Not Less or Equal (signed) ZF=0 and SF=OF

You can notice that JE is equivalent to JZ.

6.3 CMP Instruction

CMP instruction allows to compare two operands. In interne, the processor make a soustraction between the two operands. So, if the "nul flag" is activate, the two values are identicals.

Example:

	FLAGS		
	CF	ZF	SF
MOV ax, 2	0	0	0
CMP ax, 2	0	1	0
CMP ax, 0	0	0	0
CMP ax, 3	0	0	1
CMP ax, -14908	1	0	0
CMP ax,			

So you can make equivalent of IF...THEN...ELSE with this instruction.

6.4 How to use the classic loops of evolved languages

6.4.1 The IF...THEN...ELSE...

In Algo	In Assembly
If ax=1 THEN	If1: CMP AX,1
bxj-5	JNZ
ELSE	Then1: MOV BX,5
bxj- 0	JMP EndIf1
cxj- 10	Else1: MOV BX,0
EndIf	MOV CX,10
	EndIf1:

6.4.2 The FOR

In Algo	In Assembly
bxj-0	MOV BX,0
For k=0 to 10	MOV BX,0
bxj-bx+k	MOV CX,0
Fpour	For1: CMP CX,10
	JA EndFor1
	ADD BX,CX
	INC CX
	JMP For1
	EndFor1:

6.4.3 The WHILE

In Algo	In Assembly
bxj-5	MOV BX,5
While bxj>0 do	While1:
bxj-bx-1	CMP BX,0
EndWhile	JLE EndWhile1
	DEC BX
	JMP While1
	EndWhile1:

6.4.4 The REPEAT

In Algo	In Assembly
bxj-10	MOV BX,0
Do	Repeat1:
bxj-bx-1	DEC BX
Until bx j=0	CMP BX,0
	JG Repeat1
	EndRepeat1:

6.4.5 The CASE

In Algo	In Assembly
Case: bx=1:axj-1 bx=2:axj-5 bx=3:axj-10 Else axj-0 Endcase:	Case1 Case1C1: CMP BX,1 JNZ Case1C2 MOV AX,1 JMP EndCase1 Case1C2: CMP BX,2 JNZ Case1C3 MOV AX,5 JMP EndCase1 Case1C3: CMP BX,3 JNZ Other1 MOV AX,10 JMP EndCase1 Other1: MOV AX,0 EndCase1:

Chapter 7

Procedures call

7.1 Precepts of stack

Many processors have a stack. A stack allows to store data. The processor stack is FIFO (First In First Out), ie you can read only the last element. In x86, the instructions are `push ;const, register;` to add an element and `pop ;register;` to read an element. The x86 stack is 16 bits, so you can only store words. `SS:SP` (Stack Segment:Stack Pointer) point on the last element. When you push an element, `SP` is decrease to two (the stack is 16 bit, so you need) and your data is copying to `[SS:SP]`. When you pop a data, you copy `[SS:SP]` to your register and you add 2 to `SP`. Example: You have a stack of 50 blank elements:

100 <code>SP</code> →		<code>SP</code> point on before stack
98		1st blank element
96		2nd blank element

You execute :

```
MOV ax, 42
PUSH ax
MOV ax, 85
PUSH ax
```

Your stack becomes:

100		
98	42	1st element : 42
96 <code>SP</code> →	85	2nd element : 85

You pop an element :

```
POP BX
```

`BX` receive 85, and your stack begins :

100		
98 <code>SP</code> →	42	1st element : 42
96	85	2nd blank element

Sometime, you can use stack to make `MOV` operations not allowed (ex: `MOV ds, es`) You can push and pop all registers with instructions `PUSHA/POPA`. You can push/pop flag register with instructions `PUSHF/POPF`

7.2 Optimization

If you want to pop a lot of data without have to use it, you can modify directly SP, but, a little error can make crash your program. You must be careful when you use stack. You must pop all the elements you have pushed, else, your program will crash. If you have a lot of data to store in stack, you can provoke a "stack overflow". Imagine SP = 0002h, you push a data, so SP = 0000h, you push a data and SP = FFFEh, so you have provoke a stack overflow.

7.3 Procedures call

The CALL instruction allows to jump to procedure and to return at the same place where the procedure will be called. A procedure must be terminated by the RET instruction.

```
CALL <adress>
```

When you call a procedure, CPU push CP in the stack then jump to adress, when he meet RET instruction, it pops CP, and it jumps. So, if the stack is not in the same state at the end of procedure than at the begin, the program don't return to the good offset, and it will crash.

7.4 The interruptions

At the beginning, interruption allows to interrupt the program functioning (for exemple, a key press). The hardware call the interruption, the interruption makes its job, and the program takes back. But, on PC, calling drivers is more used. For exemple, if you want to write a file, you will call INT 21h wich control MS-DOS fonctions. There can be 256 interruptions, but only a few of them are used. When you call an interruption, you must specify a service in AH. The other arguments must be passed in the other registers. For exemple, if you want to display a text, you will call, the service 9 of interrution 21h, and register DX must point on a string terminated by:

```
string:
.db 'Hello Epiteens', '$'
...
...
MOV DX, offset string
MOV ah, 09h
INT 21h
```

In intern, there is in memory an interruption table of 512 byte which begin to 0000:0000. When you call interruption 21h, the processor calls the procedure pointed by the adresse 0000:0042. All interruption are located in first segment of memory.

```
so INT 21h =    MOV AX, [0000:0042]
                SUB SP, 2
                MOV [SS:SP], CS
                SUB SP, 2
```

```
MOV [SS:SP],IP  
JMP [ax]
```

So, if you change offset 0000:0042, you can return interruption!! (cf application)

Chapter 8

Input and Output Instructions

In most processor, you have a I/O adresse bus, it allow to communicate with the hardware. For example, if send data to 61h I/O port, internal buzzer will beep.

```
mov ax, 0FFh  
OUT 61h, ax
```

You can read data with IN *iport_i*, *iregister_i* command. With these two command, you can control all hardware. But, generaly, programmers use drivers (by intermediary of interruption). In a lot a processor, there isn't I/O port. An area of memory adresses is used to hardware. You use the MOV command to control your hardware. To display a graphism, you will use graphic memory.

Example:

```
mov ax, 3  
mov [a000h:0001h], ax
```

will dislay a pixel at left corner

Chapter 9

Win32 Assembly

Asm in Win 32 programs run in protected mode which is available since 80286. But 80286 is now history. So we only have to concern ourselves with 80386 and its descendants. Windows runs each Win32 program in separated virtual space. That means each Win32 program will have its own 4 GB address space. However, this doesn't mean every win32 program has 4GB of physical memory, only that the program can address any address in that range. Windows will do anything necessary to make the memory the program references valid. Of course, the program must adhere to the rules set by Windows, else it will cause the dreaded General Protection Fault. Each program is alone in its address space. This is in contrast to the situation in Win16. All Win16 programs can see each other. Not so under Win32. This feature helps reduce the chance of one program writing over other program's code/data. Memory model is also drastically different from the old days of the 16-bit world. Under Win32, we need not be concerned with memory model or segments anymore! There's only one memory model : Flat memory model. There's no more 64K segments. The memory is a large continuous space of 4 GB. That also means you don't have to play with segment registers. You can use any segment register to address any point in the memory space. In Windows, you can call API. API are like procedure, but they are not internal to your program. Argument of API are passed by pile. So, if you want to open a windows :

```
.386
.model flat,stdcall
    option casemap:none
include masm32\include\windows.inc
include masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib

.data
MsgBoxCaption DB "Expose ASM",0
MsgBoxText DB "Hello C1",0

.code start:
```

```
push MB_OK
push addr MsgBoxCaption
push addr MsgBoxText
push NULL
call MessageBox
push NULL
call ExitProcess
end start
```

You will use '.MODEL Flat, STDCALL' directive to use Flat Memory model
and

Chapter 10

Application

For our application, we wanted to make something which couldn't be made in any other languages. So we have deflected an interrupt. Our program deflects interrupt 1Ch. This interrupt is activated 18 times in one second. We have made a little parallel interface where we have connected 8 LEDs. Our application displays a little animation on the LEDs. Like, it is a resident application, the animation doesn't stop when the program is finished.

```
CODE SEGMENT
ORG 100h
ASSUME CS:CODE

start:
JMP begin          ; jump to real begin
int1Ch LABEL DWORD ;
i1Cofs DW ?        ; We will store address of 1c interrupt
i1Cseg DW ?        ;
nbr    Db 10000000b ; Store parallel port status
;Follow, the code which stay in memory
newi1C PROC FAR    ; New Interrupt
    PUSHF          ; We save flag and all modified registers
    PUSH ax
    push dx

    MOV al, nbr    ; We read parallel port status
    ror al, 1      ; We modify it
    mov dx, 0378h  ; We have constrained to use DX register
                    ; because OUT 0378h, al is not allowed
    out dx, al     ; We send on parallel port
    mov nbr, al    ; We save the new port status

    pop dx         ; We restore registers
    pop ax
    POPF
    PUSHF
    CALL int1Ch    ; We call the old interrupt
```

```

    RETF 2
newi1C ENDP          ; End of procedure

endtsr:

Begin:
MOV AX,351Ch        ; Service 35h of interruption 21h
                   ; return adresse of an interruption

INT 21h
MOV i1Cofs,BX      ; We save offset of interruption 1Ch
MOV i1Cseg,ES      ; We save segment of interruption 1Ch
PUSH CS            ; MOV CS, DS is not allowed, so,
                   ; we use stack
POP DS             ; Service 25h of interruption
                   ; 21h allows to change adress
                   ; of an interruption
MOV DX,offset newi1C ;Segment of new adresse must be in DS,
                   ; and offset must be in DX
MOV AX,251Ch       ;
INT 21h            ; We call it and it's good,
                   ; our program is installed :)
                   ; Now, before quit, we must calculate how
                   ; many byte we must reserve for our application
MOV DX,endtsr-start+100h+15 ;calculate size of program
MOV CL,4
SHR DX,CL
MOV AL,0
MOV AH,31h         ; we return with keeping our code in memory:
                   ; iterruption 21h, service 31h

INT 21h

Code ENDS
    END Start

```

Chapter 11

Conclusion

This application demonstrate the power of assembly language. Assembly is the father of all language programming. But it be a part of prehistory of computer science. Nevertheless, it is still used by crazy of computers or by demomaker who search the most optimized program. It allow to make certain thing impossible in other language. In addition, it allow to understand how to function computer. We can think in spite of his complexity, there will be always people to programm in assembly.

Bibliography

- [MERC189] . MERCIER, *Assembleur, une dcouverte pas pas, Marabout, 1989*, A little book very useful to begin in assembly.
- [SCHAH95] . SCHAKEL, *Programmer en assembleur sur PC, Micro Application, 1995*, You can find the most importants things about assembly inside.
- [INTEL] ntel, *Intel Ex Opcodes and Mnemonics*, The opcode bible. All is refenced, but ask a good comprehension of assembly language
- [MICRO] icrosoft, *API bible : a Microsoft help file*, Impossible to code in assembly with windows without this help.
- [FOG2000] gner Fog 2000, *How to optimize for the Pentium family of microprocessors*, If you want to go more far, download this help. Very good featuring. Great.
- [NoAnomaly] oAnomaly Team *Index des interruptions et fonctions* , Bible of all interruptions in french. Very useful to program in MS-DOS.
- [INTEL] ntel, <http://www.intel.com>, You can find the complete documentation of pentium processor (Intel vol.1-3.pdf)and lot of stuffs. Only for good programmers.
- [MICRO] icrosoft, <http://www.msdn.microsoft.com>, The develloper site of Microsoft.
- [ENGDA96] omy Engdahl, http://www.hut.fi/misc/electronics/circuits/parallel_output.html, 1996, A good information source about parallel port.
- [EPISC00] pis'cours, *Formation Assembleur d'Epis'cours, 2000*