The 8086 instruction set

BEKKOUCHE Mohammed

m.bekkouche@esi-sba.dz

The branch instructions

- These are the program jump instructions
- They allow you to make jumps in the execution of a program (sequence break)
- These instructions do not affect Flags

O	D	I	T	S	Z	A	P	C

- In this category we find all branch, loop and interrupt instructions

Unconditional branch instructions

Syntax: `JMP label`

JMP instruction performs an unconditional jump to the specified label. Contrary to a conditional jump, the jump is always performed; the FLAGS register does not intervene in this operation.

Example:

```
MOV AX, 00      ; Initialization from AX to 0
MOV BX, 00      ; Initialization from BX to 0
MOV CX, 01      ; Initialization from CX to 1
L20:
ADD AX, 01      ; Increment AX
ADD BX, AX      ; Add AX to BX
SHL CX, 1       ; shift left of CX, this doubles the value of CX
JMP L20         ; repeat instructions
```

Unconditional branch instructions

Syntax: `CALL label`

The notion of procedure in assembler corresponds to that of subroutine in other languages. The procedure is named `calcul`. After instruction B, the processor goes to instruction C of the procedure, then continues until it encounters RET and returns to instruction D.

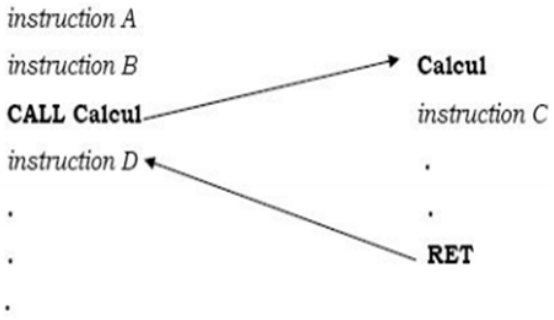


FIG. – Calling a procedure

Unconditional branch instructions

A procedure is a sequence of instructions performing a specific action, which are grouped together for convenience and to avoid having to write them several times in the program.

The procedures are located by the address of their first instruction, to which we associate a label in assembler.

The execution of a procedure is triggered by a calling program. A procedure can itself call another procedure, and so on.

lineA procedure is called by the CALL statement.

lineThe end of a procedure is marked by the RET instruction.

Unconditional branch instructions

Example:

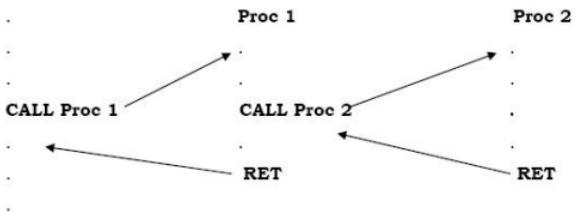
```
ORG 100h
CALL p1
ADD AX, 1
RET             ; return to OS.
p1 PROC         ; declaration procedure.
MOV AX, 1234h
RET             ; return to caller.
p1 ENDP
```

Unconditional branch instructions

RET does not take an argument and allows control to be returned to the calling subroutine: the processor goes to the instruction placed immediately after the CALL.

But how does the processor find the value of the address of this instruction?

The problem is complicated by the fact that we can have any number of nested calls, as shown in the figure.



Unconditional branch instructions

The return address, used by RET, is actually saved on the stack by the CALL instruction. When the processor executes the RET instruction, it pops the address on the stack (like POP), and stores it in IP.

CALL instruction therefore performs the following operations:

1. Stack the value of IP. At this time, IP points to the instruction following the CALL.
2. Place in IP the address of the first instruction of the procedure (given as argument).

And the RET statement: Pop a value and store it in IP.

Unconditional branch instructions

Noticed:

If the procedure belongs to the same segment as the main program, it is said to be of the NEAR type, otherwise it is of the FAR type.

The difference between them is that in the first case the processor must stack a single value in the stack: the IP register. In the second case it is necessary to stack the IP register as well as the CS segment register and, of course, unstack them during the return of the procedure.

Unconditional branch instructions

Parameter passing:

In general, a procedure performs processing on data (parameters) which are provided by the calling program, and produces a result which is transmitted to this program. Several strategies can be employed:

- **Passing by register:** the values of the parameters are contained in registers of the processor. This is a simple method, but only suitable if the number of parameters is small.
- **Passing through the stack:** parameter values are stacked. The procedure reads the stack.

Unconditional branch instructions

Example with pass by register:

We are going to write a procedure (SUM) which calculates the sum of 2 natural numbers of 16 bits. The integers are passed through the AX and BX registers, and the result will be placed in AX.

```
ORG 100h
MOV AX, 6
MOV BX, Truc
CALL SUM
MOV Truc, AX
RET
Truc DW 10
SUM PROC
ADD AX, BX          ; AX <- AX + BX
RET
SUM ENDP
```

The call to the SUM procedure adds 6 to the variable Truc.

Unconditional branch instructions

Example with passing through the stack:

This technique implements a new register, BP (Base Pointer), which allows values to be read from the stack without popping them or modifying SP. The BP register allows a special indirect addressing mode, of the form: `MOV AX, [BP+6]`.

`MOV BP, SP` BP points on top

`MOV AX, [BP]` Read without popping.

`MOV AX, [BP+2]` 2 because 2 bytes per stack word

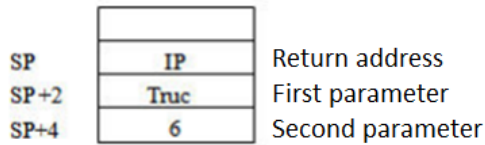
Unconditional branch instructions

Example with passing through the stack:

The call to the procedure (SUM2) with passing through the stack is:

```
PUSH 6
PUSH Truc
CALL SUM2
```

SUM2 procedure will read the stack to obtain the value of the parameters. The top of the stack contains the return address (old value of IP stacked by CALL). Each element of the stack occupies two bytes.



Unconditional branch instructions

Example with passing through the stack:

The SUM2 procedure is written therefore:

```
SUM2 PROC          ; AX <- arg1 + arg2
MOV BP, SP         ; address stack top
MOV AX, [BP+2]     ; load argument 1
ADD AX, [BP+4]     ; add argument 2
RET
SUM2 ENDP
```

Conditional branch instructions

All conditional jump instructions take the same type of operand.

JA	étiquette	saut si supérieur (C =0 et Z =0)
JAE	étiquette	saut si supérieur ou égal (C =0)
JB	étiquette	saut si inférieur (C =1)
JBE	étiquette	saut si inférieur ou égal (C =1 ou Z =1)
JC	étiquette	saut si retenue (C =1)
JCXZ	étiquette	saut si CX vaut 0
JE	étiquette	saut si égal (Z =1)
JG	étiquette	saut si supérieur (Z =0 et S =0)
JGE	étiquette	saut si supérieur ou égal (S =0)
JL	étiquette	saut si inférieur (S ≠0)
JLE	étiquette	saut si inférieur ou égal (Z =1 ou S ≠0)
JNC	étiquette	saut si pas de retenue (C =0)
JNE	étiquette	saut si non égal (Z =0)
JNO	étiquette	saut si pas de débordement (O =0)
JNP	étiquette	saut si pas de parité (P =0)
JNS	étiquette	saut si pas de signe (S =0)
JO	étiquette	saut si débordement (O =1)
JP	étiquette	saut si parité (P =1)
JS	étiquette	saut si signe (S =1)

Conditional branch instructions

Example:

```
include 'emu8086.inc'
ORG 100h
MOV AL, 5
CMP AL, -5 ; ZF = 0 and SF = 0
JG label1
PRINT 'AL is not greater -5.'
JMP exit
label1:
PRINT 'AL is greater -5.'
exit:
RET
```

Conditional branch instructions

Tests (if) can be written in assembly in different ways depending on the condition and the jump statement used.

An *if* (op1 [operator] op2) then Action statement can be translated as follows:

```
CMP op1, op2
Jxx et1        ; Jxx <=> NOT [operator]
Action
et1:
```

Conditional branch instructions

An *if* (op1 [operator] op2) then Action1 else Action2 statement can be translated as follows:

```
CMP op1, op2
Jxx et1        ; Jxx <=> [operator]
Action2
JMP et2
et1:
Action1
et2:
```

Conditional branch instructions

```
org 100h
mov ax, ...
cmp val, 0
je et1
mov bl, val
div bl
jmp et2
et1:
mov al, -1
et2:
ret
val db ...
```

We want to perform the division of one number by another. Division is only possible if the divisor is nonzero. We need to test if the divisor operand is null before performing the division. If the divisor is zero, the result is -1; otherwise, the result is the quotient.

Loop statements

Syntax: `LOOPx label`

Instruction	Update	Branch
LOOP	$CX \leftarrow CX - 1$	$CX \neq 0$
LOOPZ, LOOPE	$CX \leftarrow CX - 1$	$(CX \neq 0)$ and $(Z = 1)$
LOOPNZ, LOOPNE	$CX \leftarrow CX - 1$	$(CX \neq 0)$ and $(Z = 0)$

Loop statements

LOOP works with the CX register which acts as a loop counter. LOOP decrements the counter without changing any of the flags. If the counter is different from 0, a jump to the operand label of the instruction LOOP is realized.

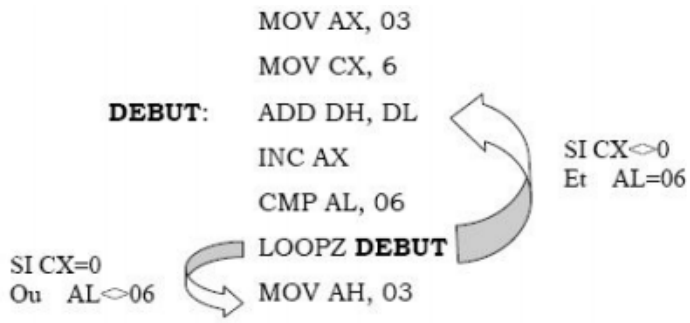
```
CX = CX - 1
If CX <> 0 then
    jump
Otherwise
    no jump, continue
```

Loop statements

There is a jump to the label specified in operand as long as the counter is non-zero and the indicator Z is 1.

```
CX = CX - 1
If CX <> 0 and ZF = 1 then
    jump
Otherwise
    no jump, continue
```

Loop statements

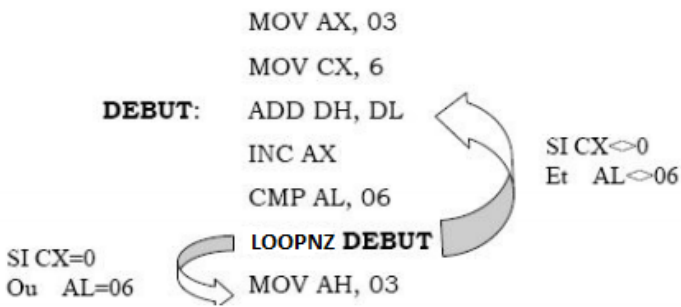


Loop statements

There is a jump to the label specified in operand as long as the counter is non-zero and the Z indicator is equal to 0.

```
CX = CX - 1
If CX <> 0 and ZF = 0 then
    jump
Otherwise
    no jump, continue
```

Loop statements



Loop statements

LOOP statements make it easier to write "for" type loops thanks to the auto-decrementing of the CX register and the branching condition based on it.

A *for* $i := 10$ down to 1 do Action loop can be translated as follows:

```
MOV CX, 10
et1:
Action
LOOP et1
```

Loop statements

Other types of loops can be translated using branching statements depending on the condition.

A *while* (op1 [operator] op2) do Action loop can be translated as follows:

```
et1:
CMP op1, op2
Jxx et2        ; Jxx <=> NOT [operator]
Action
JMP et1
et2:
```

Loop statements

A *repeat Action until* (op1 [operator] op2) statement can be translated as follows:

```
et1:
Action
CMP op1, op2
Jxx et1        ; Jxx <=> NOT [operator]
```

Loop statements

```
ORG 100h
MOV AX, 1
MOV BX, 3
MOV CX, 0
Etq:
MUL BX
INC CX
CMP CX, BX
JNE Etq
RET
```

$$AX := 1,\ BX := 3,\ CX := 0$$

$$\text{Repeat} \quad AX := AX \times BX; \quad CX := CX + 1$$

$$\text{until} \quad (CX = BX)$$

Loop statements

This program calculates $A := A \times 3$ three times:

$$(1 \times 3) = 3 \qquad (3 \times 3) = 9 \qquad (9 \times 3) = 27$$

Loop statements

```
ORG 100h
MOV AX, 1
MOV BX, 3
MOV CX, 0
Etq:
CMP CX, BX
JE Fin
MUL BX
INC CX
JMP Etq
Fin:
RET
```

$$AX := 1,\ BX := 3,\ CX := 0$$
$$\text{While } (CX \neq BX) \text{ do}$$
$$AX := AX \times BX; \quad CX := CX + 1$$

Interrupt Instructions

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

- Software interrupts
- Hardware interrupts

Interrupt Instructions

An interrupt, as its name suggests, interrupts the normal execution of program instructions to execute other instructions before returning to normal operation.

An interruption can occur in two ways:

- It can be called upon by the hardware to perform a process (read a key on the keyboard for example). In this case the hardware triggers an IRQ (Interrupt ReQuest). This is a hardware interrupt.
- It can be requested by the running software with the INT instruction. This is a software interrupt.

Interrupt Instructions

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

```
INT value
```

where the value can be a number between 0 and 255 (or 0 to 0FFh), generally hexadecimal.

Each interrupt can have subfunctions. To specify a subfunction, the AH register must be set before calling the interrupt. Each interrupt can have up to 256 sub-functions. Usually AH is used, while other registers are used to pass parameters and data.

Interrupt Instructions

To call the "00h" sub-function of the "21h" interrupt, we execute:

```
MOV AH, 00h      ; sub-function
INT 21h          ; call to interrupt
```

One of the main interrupts used is "21h", and its functions are numerous.

Interrupt Instructions

- The 09h function: is used to display a sequence of characters from the address contained in DX. The string must be terminated with the character $.

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

Interrupt Instructions

- The 0Ah function: allows you to enter a character string in DS:DX.
- The first byte is the size of the buffer; the second byte is the number of characters actually read.
- This function does not add '$' to the end of the string.
- To print with INT 21h / AH = 9, insert the dollar character at the end and start printing from the address DS:DX+2.