Jump instructions

– do not change flags.

Unconditional jumps

Jump

- near – the target label is in the same segment than the jump
- far – jump to another code segment

Direct jump

jmp label

Near jump

```
        jmp Stop
        xor ax,ax
Stop:   mov ah,4Ch
```

displacement = the difference between the target label and IP (may also be negative)

Machine code:

```
cs:0000 EB 03    jmp Stop
cs:0002 90       nop
cs:0003 33 00    xor ax,ax
cs:0005 B4 4C    Stop: mov ah,4Ch
```

A processor executes the jump adding the displacement to the current value of IP (IP := 0002 + 3 = 0005) => IP will point to the instruction at which the program execution shall continue.

Two-pass assembler

– scans the source assembly language program twice.

The purpose of the **1st pass** is to work out the locations corresponding to symbols (identifiers). To work out these locations, the assembler uses a variable known as the **location counter** (LC). The **symbol table** is created during the first pass; it records the names of variables and labels together with their attributes.

```
            .DATA
LC =    0   Number DW 1234h
        2   Array DW 100 dup(?)
      202   Value DB 5,6,7
      205
```

Symbol table:

Symbol	Segment	Offset	Type
Number	_Data	0	variable: word
Array	_Data	2	variable: word
Value	_Data	202	variable: byte

```
                .CODE
LC =     0  Start:    mov ax,@data
         3            mov ds,ax
         5            mov cx,Number
         9  Next:     dec cx
        10
```

Symbol table:

Symbol	Segment	Offset	Type
Start	_Text	0	label: near
Next	_Text	9	label: near

Problem: forward jumps

A 16-bit displacement $\in \langle$-32768; 32767$\rangle$ is supposed, i.e. the assembler reserves two bytes for the displacement of a forward jump instruction.

In the **2nd pass** the assembler uses the symbol table to generate the machine code. If the displacement is an 8-bit value $\leq$ 127, the second byte is filled with the op-code for instruction **`nop.`**

Operator **`short`** instructs the assembler to use an 8-bit displacement:

```
cs:0000 EB 02   jmp short Stop
cs:0002 33 00   xor ax,ax
cs:0004 B4 4C   Stop: mov ah,4Ch
```

Far jump

The machine code operand of far jump is the complete address of the destination in the order: offset, segment (4 bytes). A processor executes the jump loading IP by the offset and CS by the segment.

If the forward jump is a far jump, we must instruct the assembler to reserve 4 bytes for the operand by defining the far type label:

```
    jmp far ptr StopInAnotherSegment
```

Indirect jump

Near jump

jmp register/memory

A 16-bit operand contains the offset of the instruction, at which the program execution shall continue.

```
.DATA
Address   DW Start
.CODE
Start:    mov ax,offset Stop
          jmp ax
            ...
Stop:     jmp Address
```

Far jump

jmp memory

An operand contains the complete address (offset, segment) of the instruction, at which the program execution shall continue; it is of type:

- dword in 16-bit mode
- fword in 32-bit mode

```
Data SEGMENT
Address DD Continue
Data ENDS

Program1 SEGMENT
ASSUME cs:Program1
Continue: xor ax,ax
          mov ah,4Ch
          int 21h
Program1 ENDS

Program2 SEGMENT
ASSUME cs:Program2, ds:Data
Start:    mov ax,Data
          mov ds,ax
          jmp Address
Program2 ENDS
END Start
```

Conditional jumps

They allow to branch program execution according to the flags ZF, CF, SF, PF a OF.

jcc label

After comparison of unsigned numbers:

Instruction	Meaning – jump if	Condition
jb **jnae** **jc**	below not (above or equal) carry	CF = 1
jae **jnb** **jnc**	above or equal not below not carry	CF = 0
jbe **jna**	below or equal not above	CF = 1 or ZF = 1
ja **jnbe**	above not (below or equal)	CF = 0 and ZF = 0

After comparison of signed numbers:

Instruction	Meaning – jump if	Condition
jl **jnge**	less not (greater or equal)	SF ≠ OF
jge **jnl**	greater or equal not less	SF = OF
jle **jng**	less or equal not above	ZF = 1 or SF ≠ OF
jg **jnle**	greater not (less or equal)	ZF = 0 and SF = OF

Instruction	Meaning – jump if	Condition
je **jz**	equal zero	ZF = 1
jne **jnz**	not equal not zero	ZF = 0
jp **jpe**	parity parity even	PF = 1
jnp **jpo**	not parity parity odd	PF = 0
js	sign	SF = 1

jns	not sign	SF = 0
jo	overflow	OF = 1
jno	not overflow	OF = 0
jcxz	CX is 0	CX = 0
jecxz	ECX is 0	ECX = 0

Conditional jumps must be direct, near and short.

```
cmp al,'x'
je StopFarAhead
inc Count
```

↓

```
cmp al,'x'
jne Continue
jmp StopFarAhead
Continue: inc Count
```

Loop instructions

– do not change flags.

loop label

In 16-bit mode, **loop** decrements register CX and compares it with 0 leaving the flags unchanged. If new CX ≠ 0, jumps to the label. Otherwise the program execution continues with the next instruction.

Label is at the first instruction of the loop. It must be short.

In 32-bit mode, **loop** decrements and tests register ECX.

loope label **loopz** label

- decrement register CX and compare it with 0. If the new contents of register CX ≠ 0 a ZF = 1, jump to the label.

loopne label **loopnz** label

- decrement register CX and compare it with 0. If the new contents of register CX ≠ 0 a ZF = 0, jump to the label.

Example

Read characters typed on the keyboard and store them to variable **IOBuffer** until ENTER is pressed or **MaxNumber** characters are typed.

```
.MODEL small
.STACK 100h
.DATA
MaxNumber EQU 80
IOBuffer DB MaxNumber dup (?)
.CODE
Start: mov ax,@data
       mov ds,ax
       mov bx,offset IOBuffer
       mov cx, MaxNumber
       jcxz Stop
Read:  mov ah,1
       int 21h; store ASCII code of pressed
character to al
       mov [bx],al
       inc bx
       cmp al,0Dh; was ENTER?
       loopnz Read; repeat if not
Stop:  mov ax,4C00h
       int 21h
END Start
```