Laboratory Work 4

4 Intel x86 instruction set (continued)

4.1 Objectives

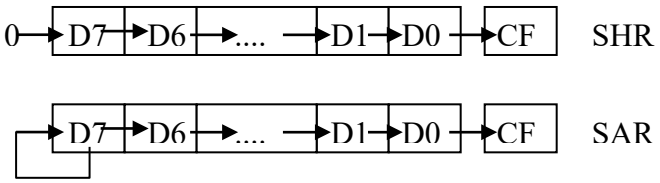
This laboratory work continues the previous one, several assembly language instructions are described, including: shift, rotate, unrestricted and conditional jumps, in/out, array and special instructions.

4.2 Instructions

4.2.1 Shift and rotate instructions

SHL, (SAL), SHR and SAR instructions

These instructions perform shift left or right of the specified operand. The logical shift (SHL & SHR) copies each bit in the neighboring locations (left or right) and the free positions are filled up with 0. The arithmetic shifts (SAL & SAR) consider a signed number inside the operand, hence these shifts are basically a multiplication and division with the powers of 2 (e.g. shift left with 2 binary positions is equivalent to a multiplication by 4). The arithmetic shift right keeps the operand sign, because of the usage of signed numbers. Note that each type of shift records the bit which was shifted-out in the carry flag CF, as illustrated below:



Instruction syntax:

SHL <parameter_1>, <parameter_2>
SAL <parameter_1>, <parameter_2>
SHR <parameter_1>, <parameter_2>
SAL <parameter_1>, <parameter_2>

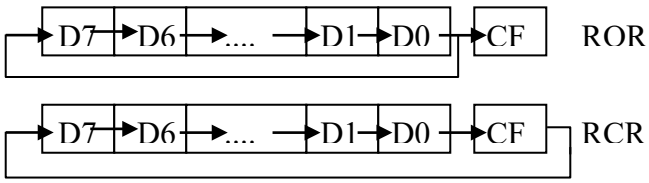
The first paramenter is according to the previous definitions, while the second specifies the number of binary positions of the shift operation. This parameter can be 1 or if the value is higher, it must be specified in register CL.

Examples:

```
shl     ax,1
shr     var, cl
```

ROR, ROL, RCR, RCL instructions

These instructions perform right or left rotations of the operand, based on the number of binary positions. The difference between the previous shift operations is the fact that the freed binary position is added to the beginning or the end of the binary number (depending on the left or right rotation). RCL and RCR perform the rotation with the involvement of the carry flag CF, while ROR and ROL don't include this flag.



Instruction syntax:

ROR <parameter_1>, <parameter_2>
ROL <parameter_1>, <parameter_2>
RCR <parameter_1>, <parameter_2>
RCL <parameter_1>, <parameter_2>

4.2.2 Jump instructions

These instructions are used for controlling the execution sequence of instructions. There are 2 types of jumps:

- Branch and loops
- Call and return

4.2.2.1 Call and return instructions

Using routines (functions and procedures) allow modular and structured assembly language programs. This is useful when several operations are repeated, hence they can be written as routines and called a specific number of times.

CALL şi RET instructions

The CALL instruction performs a jump to the specified address in the instruction. Befor the jump, the CPU saves on the program stack the return address. The RET instruction placed at the end of the routine performs the return in the appellant program. Hence, it extracts from the program stack the return address and makes the jump to it.

Instruction syntax:

CALL <address>
RET [<constant>]

where:

<address> - label with the name of the routine
<constant> - indicates the number of positions it must download from the stack before returning from the routine; this parameter usually is missing.

Examples:

```
call    rut_sum
call    1000:100
ret
ret     2
```

4.2.2.2 Branch and loop instructions

These instructions modify the execution sequence of instructions. There are unconditional jump instructions, which execute in any condition, and conditional jumps which execute when a specific condition is fulfilled (such as a flag or combination of flags).

Instruction syntax:

JMP <address>
J<cc> <address>

where:

<address> - label
<cc> - combination of letters that imply the condition

The table below exhibits conditional jump variants that imply the testing of a single flag. The following 2 tables show the conditional jump instructions used after a compare instruction.

Instruction	Condition	Equivalent instructions	Explanations
JC	CF=1	JB,JNAE	Jump if carry was used
JNC	CF=0	JNB,JAE	Jump if carry was not used
JZ	ZF=1	JE	Jump if result is zero
JNZ	ZF=0		Jump if result is not zero
JS	SF=1		Jump is result is negative
JNS	SF=0		Jump is result is negative positive
JO	OF=1		Jump if overflow of quantity
JNO	OF=0		Jump if no overflow of quantity
JP	PF=1		Jump if result is even
JNP	PF=0		Jump if result is not even

Conditional jump instructions used after comparing 2 unsigned numbers:

Instruction	Condition	Tested flags	Equivalent instructions	Explanations
JA	>	CF=0,ZF=0	JNBE	Jump if greater
JAE	>=	CF=0	JNB,JNC	Jump if greater or equal
JB	<	CF=1	JNAE,JC	Jump if lower
JBE	<=	CF=1 sau ZF=1	JNA	Jump if lower or equal
JE	=	ZF=1	JZ	Jump if equal
JNE	!=	ZF=0	JNZ	Jump if not eqal

Conditional jump instructions used after comparing 2 signed numbers (represented in 1s complement):

Instruction	Condition	Tested flags	Equivalent instructions	Explanations
JG	>	SF=OF sau ZF=0	JNLE	Jump if greater
JGE	>=	SF=OF	JNL	Jump if greater or equal
JL	<	SF!=OF	JNGE	Jump if lower
JLE	<=	SF!=OF sau ZF=1	JNG	Jump if lower or equal
JE	=	ZF=1	JZ	Jump if equal
JNE	!=	ZF=0	JNZ	Jump if not equal

Please note that the comparison of the 2 operands is performed between the first and second parameter.

Examples:

```
        cmp     ax, 100h
        je      et1             ; jump if ax=100h
        cmp     var1,al
        jb      mai_mic         ; jump if var1<al
        add     dx,cx
        jo      eroare          ; jump if overflow of quantity
```

LOOP, LOOPZ, LOOPNZ instructions

These instructions allow the implementation of control structures similar to "for", "while", "do-until" from higher level programing languages. These instructions execute the following sequence of instructions: decrement the CX register (this is used as a counter), test for the terminating condition and jump to label (declared at the beginning of the loop sequence) if the test condition is not fulfilled.

Instructions syntax:

```
LOOP        <address>
LOOPZ       <address>
LOOPNZ      <address>
```

where: <address> is a label declared by the programmer in the source code.

Instruction LOOP has CX=0 as the terminating condition (counter reaches 0), while LOOPZ tests for ZF=0 and LOOPNZ tests for ZF=1.

Example:

```
        mov     ecx, 100
et1:    .....
        loop    et1     ; the loop executes 100 times
```

4.2.3 In/out instructions

These instructions are used for transferring data with the registers (ports) of the in/out interface. Note that these are the only instructions in Intel CPUs that utilize ports.

IN & OUT instructions

```
IN      <accumulator>, <port_address>
OUT     < port_address >, <accumulator>
```

where:

<accumulator> - register AX for 16 bit transfer or AL for 8 bit transfer
<port_address> - address expressed on 8 bit or register DX

Examples:

```
        in      al, 20h
        mov     dx, port_address
        out     dx, ax
```

4.2.4 Array instructions

These instructions have the purpose to speed up the access to the elements of an array or vector. The implicitly use the index registers ESI and EDI for addressing the elements of a

source and destination array. These registers are automatically increased or decreased to get to the next or previous elements in the array. The DF flag indicates the direction (1 for incrementing and 0 for decrementing). Register CX is used to count the number of operations performed. After each execution, register CX is decreased.

MOVSB, MOVSW instructions

These instructions transfer one element from the source array to the destination array. MOVSB works on byte (8 bit), while MOVSW on word (16 bit). The second instruction increments or decrements the index registers by 2, because each element of the array occupies 2 bytes. Note that these instructions don't have parameters.

Example:

```
        mov     esi, offset source_array        ; "offset" is a keyword that determines the offset address of the variable
        mov     edi, offset destination_array
        mov     ecx, array_length
et:     MOVSB                   ; DS:[SI]=>ES:[DI], SI++, DI++, CX--
        jnz     et
```

LODSB, LODSW, STOSB and STOSW instructions

The first 2 instructions perform successive load of the elements of an array in the accumulator register. The next 2 instruction perform the reverse operation (store the accumulator register in an array). These instructions also use the index registers (ESI for load and EDI for store), they are decremented or incremented automatically, while register CX is decremented. The letter "B" or "W" at the end indicate byte or word (8 or 16 bit transfer).

CMPSB, CMPSW, SCASB and SCASW instructions

The first 2 instructions perform the comparison between the elements of 2 array, while the last 2 instructions compare the content of the accumulator register with each element of the array (similar to a scanning operation).

REP, REPZ, REPE, REPNZ, REPNZ instructions

These instructions allow the multiple execution of an instruction on an array. By using any of these instructions in front of an array instruction, it forces the CPU to repeat the operations until a terminating condition is fulfilled. The first instruction REP has CX=0 as the terminating condition. REPZ and REPE allow the operation to repeat until the result is 0 or the operands are equal. For REPNZ and REPNE, the operation is repeated as long as the result is not 0 or the operands are different.

Examples:

```
        mov     esi, offset source_array
        mov     edi, offset destination_array
        mov     ecx, lungime_sir
        rep     movsb           ; transfers the source array to the destination array
```

4.2.5 Special instructions

This category contains instructions that effect the CPU operating mode.

CLC, STC, CMC instructions
These instructions modify the value of the CF flag as follows:

CLC clears flag, CF=0
STC sets flag, CF=1
CMC reverses the value, CF=NOT CF

CLI şi STI instructions
These instructions clear and set the interrupt flag IF. In the set state IF=1, the CPU detects all maskable interrupts, and in the inverse state it blocks all maskable interrupts.

CLD şi STD instructions
These instructions modify the state of the direction flag DF. This flag controls the navigation direction of array operations: DF=0 for incrementation or DF=1 for decrementation.

NOP, HLT şi HIT instructions
The NOP instruction does nothing and is mostly used for timing, for delaying operations or for implementing wait loops. The HLT instruction (halt) forces the CPU to stop. The HIT instruction determines a temporary stop of the CPU, until an interrupt signal or reset signal occurs.

CPUID instruction
Allows the identification of the CPU type.

4.3 Exercises

1. Please follow the online document at:
   http://users.utcluj.ro/~madalin/SM/labs/W4.pdf