LABORATORY WORK NO. 6
DATA TRANSFER INSTRUCTIONS

1. Object of laboratory

Study of data transfer instructions for the I8086 microprocessor, including the input-output instructions.

2. Theoretical considerations

Data transfer is one of the most common tasks when programming in an assembly language. Data can be transferred between registers or between registers and the memory. Immediate data can be loaded to registers or to memory. The transfer can be done on bye, word or double word size. The two operands must have the same size. Data transfer instructions don't affect the flags (excepting the ones that have this purpose). They are classified as follows:

- „classical" data transfer instructions
- address transfer instructions
- flag transfer instructions
- input/output instructions.

2.1. „Classical" transfer instructions

Include the following instructions:

```
MOV   <d>, <s>
XCHG  <d>, <s>
XLAT
PUSH  <s>
POP   <d>
```

Data is copied from source to destination with the MOV instruction. The syntax of this is instruction is:

MOV {register | memory}, {register | memory | immediate data}

This instruction copies the source operand to the destination. Right after a MOV instruction is executed, the source operand and the destination have the same value. The old content of the destination is overwritten.

Example:

```
DATA        SEGMENT
MEM         LABEL BYTE          ;byte and
MEMW        DW ?                ;word
VCT   DB          100 DUP (?)   ;vector
DATA ENDS
CODE SEGMENT
      ASSUME CS:CODE, DS:DATA
      … …
      MOV AX, 7           ;immediate data to register
      MOV MEM, 7          ;immediate byte to directly
                          ;addressed memory
      MOV MEMW, 7         ;immediate word to directly
                          ;addressed memory
      MOV VCT[BX], 7      ;immediate byte to indirectly
                          ;addressed memory
      MOV MEMW, DS        ;segment register to memory
      MOV MEMW, AX        ;general register to directly addressed memory
      MOV VCT[BX], AL     ;general register to indirectly
                          ;addressed memory
      MOV AX, MEMW        ;directly addressed memory to general register
      MOV AL, VCT[BX]     ;indirectly addressed memory to
                          ;general register
      MOV DS, MEMW        ;directly addressed memory to
                          ;segment register
      MOV AX, BX          ;general register to general register
      MOV DS, AX          ;general register to segment register
      MOV CX, ES          ;segment register to general register
      … …
CODE ENDS
```

The following MOV instructions are not permitted: immediate data to segment register, memory location to memory location, segment register to segment register and MOV to the CS segment register.

MOV instructions that require two instructions are presented below.

Example:

```
                    ;immediate data to segment register
        MOV AX, 1000H
        MOV DS, AX
                    ;memory location to memory location
        MOV AX, MEM1
        MOV MEM2, AX
                    ;segment register to segment register
        MOV AX, DS
        MOV ES, AX
```

Newer processors allow these instruction, however early versions of assemblers will not recognize them as valid instructions.

Data, respectively source and destination operands interchange is done with the XCHG instruction. Its syntax is presented below:

XCHG {register | memory}, {register | memory}

Example:

```
        XCHG AX, BX         ;interchanges ax with bx
        XCHG MEM16, AX      ;interchanges the memory
                            ;word mem16 with the ax register
        XCGH DL, MEM8       ;interchanges the memory byte mem8
                            ;with register dl
        XCGH AH, CL         ;interchanges ah with cl
```

The XLAT instruction coverts the content of register AL, using a translation table. A pointer to the start of the table should be in register BX. The content of register AL is interpreted as an index in the table. The result of the conversion is given by the value of the byte that is placed at this address in the table. The syntax is as follows:

XLAT [segment register : offset]

Using a reference to an address in the XLAT instruction is necessary when the table is not located in the data segment, which is the only implicit segment for this instruction. It allows the assembler to determine the segment register that has to be used for the execution of the instruction.

Here is an example that translates a Hexadecimal digit in a printable ASCII code:

Example

```
                              ;hexadecimal to ASCII conversion
                              ;input : al = hexadecimal digit
                              ;output : al = the corresponding ASCII code
CONV    PROC NEAR
        MOV BX,  OFFSET TABEL
        XLAT CS:TABEL
        RET
CONV            ENDP
TABEL           DB ‘0123456789ABCDEF’           ;ASCII code table
```

The PUSH and POP instructions are used for data transfer to and from a stack.

The stack is a memory location used for temporary data storage. The top of the stack address is managed automatically, by hardware, through a register that points to the top of the stack, namely SP register. This is why these instructions, PUSH and POP, only allow access to the top of the stack. The data that is placed on the stack can be accessed in reverse order of the placement (LIFO system- Last In First Out). Initially the stack contains no data. As data is being placed, during the execution of the program, the stack grows in size, towards smaller addresses. As data is being extracted from the stack, its size is decreasing, by successively freeing the locations that have the smallest address.

The instructions for subroutine call, namely CALL, INT and return from subroutines, RET and IRET, automatically use the stack for saving and restoring the return addresses.

The PUSH instruction is used to put a 2 byte operand on the stack. The POP instruction is used to extract the last value from the stack. The syntaxes for these instructions are:

```
PUSH {register | memory}
POP  {register | memory}
```

When pushing an operand on the stack, the first thing that is done is decrementing the stack pointer SP by 2 and copy the operand to this memory location. When extracting from the stack, first the value on the top of the stack is copied and the SP is incremented by 2.

The PUSH and POP instructions are usually used in pairs. Normally, the number of pushes has to be equal to the number of pops to/from the stack to bring the stack to its initial state. The words are popped in the reverse order of the pushes.

Example

```
INT     PROC FAR
        PUSH DS
        PUSH AX
        PUSH CX
        PUSH SI
        PUSH BP
        … …
        POP BP
        POP SI
        POP CX
        POP AX
        POP DS
        IRET
INT     ENDP
```

If there is no need to restore the values pushed on the stack , e.g. parameter transfer to a procedure, the stack can be freed by adding a number to the SP registers (unloading the stack).

Example:

```
        PUSH AX
        PUSH BX
        PUSH CX
        … …
        ADD SP, 6
```

The values that are not on the top of the stack can still be accessed by indirect addressing, using the BP register as base register:

Example:
```
        PUSH AX
        PUSH CX
        PUSH DX
        MOV BP, SP
        … …
        MOV AX, [BP+4]
        MOV CX, [BP+2]
        MOV DX, [BP+0]
        … …
        ADD SP, 6
```

Here is an example of a loop that is included in another loop, using the CX register as a counter in for both loops.

Example:
```
MOV CX, 10                      ;init counter for outer loop
ET1:                            ;start of outer loop
        ;… …
        PUSH CX                 ;saving counter outer  loop
        MOV CX, 20              ;init counter inner loop
    ET2:                        ;start of inner loop
        ;… …
    LOOP ET2
        POP CX                  ;restore counter outer loop
                                ;outer loop
        ;… …
LOOP ET1
```

2.2. Instructions for address transfer

They are used for loading effective addresses (16 bits) or physical ones (32 bits) into registers or register pairs. There are 3 such instructions:

```
LEA <d>, <s>
LDS <d>, <s>
LES <d>, <s>
```

The LEA instruction loads the effective address of the source operand, that has to be a memory location, to the general register that is specified as the destination. Its syntax is as follows:
LEA {register}, {memory}

The LDS and LES instructions load the physical address that is contained by the source operand, which has to be a double memory word, to the segment register that is specified by the instruction mnemonic, DS and ES, and to the general register that is specified as destination. The instruction mnemonic is:

```
LDS {register}, {memory}
LES {register}, {memory}
LFS {register}, {memory}
LGS {register}, {memory}
```

The LEA instruction can be used for loading the effective address of an operand that is placed in the memory, by direct or indirect addressing.

Example:

```
        LEA DX, ALFA
        LEA DX, ALFA[SI]
```

The effect of the first instruction can be also obtained by using the next instruction:

```
        MOV DX, OFFSET ALFA
        MOV DX, OFFSET ALFA[SI] is an incorrect instruction
```

This option is quicker, but can only be obtained in the case of operands specified by direct addressing.

Example:

```
DATA SEGMENT
STRING              DB    "THIS IS A STRING"
FPSTRING            DD    STRING        ; FAR POINTER TO STRING
POINTERS            DD    100 DUP (?)
DATA ENDS
CODE SEGMENT
        … …
LES DI, FPSTRING          ;the address contained in the source location is
                          ;loaded to
                          ; the pair es:di
LDS SI, POINTERS[BX]      ;the address contained in the source location is
                          ;loaded to
                          ;the pair ds:si
        … …
CODE ENDS
```

2.3. Flag Transfer instructions

In the I8086 microprocessor’s set of instructions there are instructions for loading and storing the flags. The syntax is:

LAHF
SAHF
PUSHF
POPF

The least significant byte of the flag register can be loaded to the AH register using the LAHF register, and also the content of the AH register can be stored to the low byte with the SAHF instruction. The structure of the low byte is:

bit	7	6	5	4	3	2	1	0
	SF	ZF	x	AF	x	PF	x	CF

The whole flag register can be pushed and restored only to the stack register, the instructions to are PUSHF and POPF. The flag register’s structure is:

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	x	x	x	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

2.4. Input/output instructions

I/O ports, are constituent elements of interfaces. They connect central units with peripheral devices.

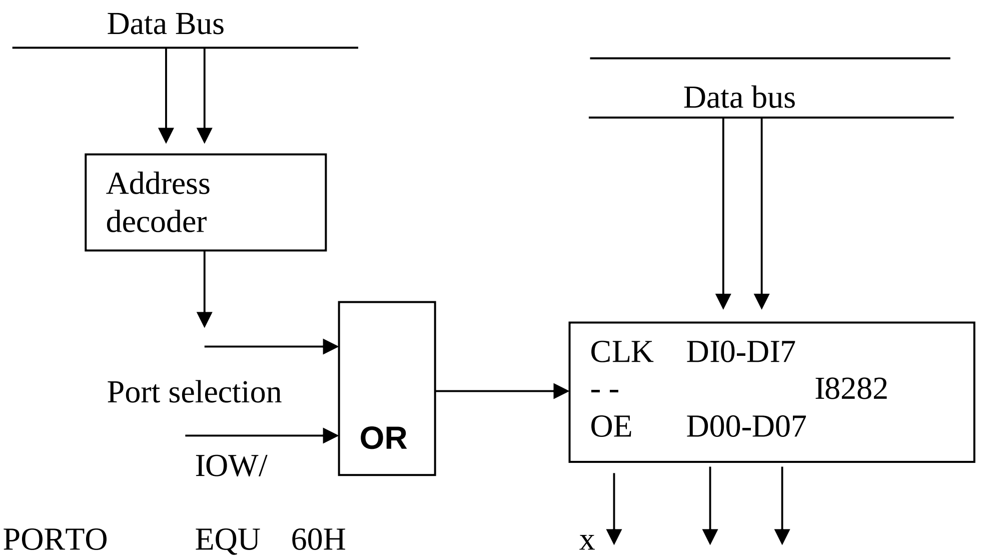
Each peripheral device has its own address through which it can be selected by the central unit. From the central unit’s point of view, the peripheral registers can be either input registers or output ones. For transfers of data to these registers, we use the OUT instruction, and for getting, reading data we use the IN instruction. Their syntaxes are:

```
IN      {AX | AL}, {peripheral immediate address | DX}
OUT     {peripheral immediate address | DX }, {AX | AL}
```

The peripheral register's address can be specified by an immediate 8 bit data or by previously storing the I/O address in the DX register. Using DX allows the usage of a larger address than 255.

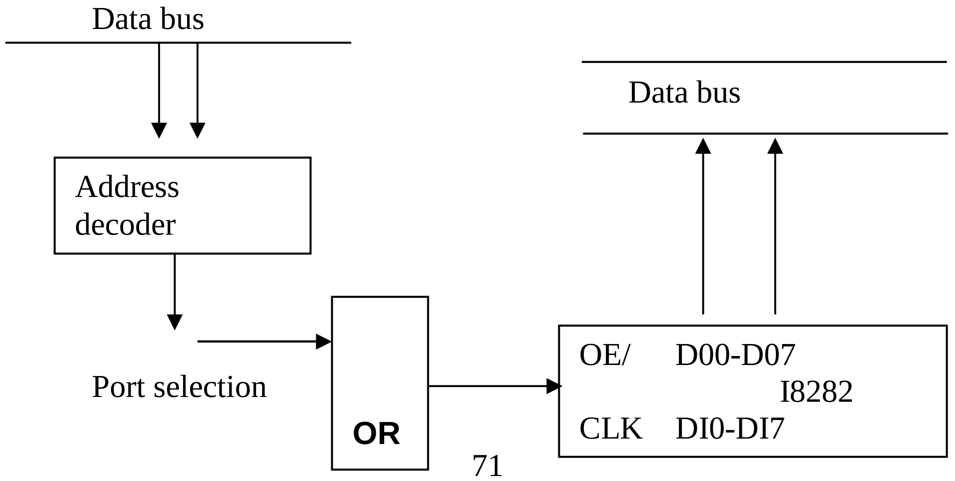
Data transfer is made between the central unit's accumulator and the peripheral registers. This transfer can be of 8, 16 ore 32 bits, depending on the register one uses, either AL, AX or EAX.

Example 1:



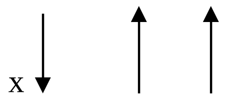
```
PORTO       EQU    60H
        …
        MOV    AL, 50
        OUT    PORTO, AL
```

Example 2:



$\longrightarrow$

IOR/



```
PORTI EQU   80H
      …
      IN    AL, PORTI
```

The IN and OUT instructions are the only instructions that allow interaction between the processor and other devices. Some computer architectures have their memory organized in such a way that the areas from the memory space are dedicated to some peripheral equipments and not to data space e.g. Video RAM memory. Access to these memory areas will actually mean access to a peripheral equipment. Such input/output systems are called „memory-mapped” (inputs/outputs organized as memory areas).

Let’s consider that a peripheral equipment requires a state port and a data port, both on 8 bits. In a regular input/output system, there are two input ports, for instance 0F8H and 0F9H, dedicated to that equipment. In a memory-mapped system there are two addresses, usually adjacent, for instance C800:0000 and C800:0001, corresponding to the state and data ports. The state-read and data-read sequences, in the two input/output types are:

```
IN    AL, 0F8H        ;read state
IN    AL, 0F9H        ;read data
MOV   ES, 0C800H
MOV   AL, ES:[0]      ;read state
MOV   AL, ES:[1]      ;read data
```

Example: in a PC-AT system, the first serial port uses other ports, starting with 3F8H, but at the same time, the access to the port can be done through the memory, at the address 40:0000. For COM2: ports starting with 2F8H or through the memory, at 40:0002.

3. Lab tasks

1. Study of the shown examples.
2. The students will write a program, which copies a string of values from consecutive memory locations to another location, placed in a different data segment.

3. The students will write a program that duplicates the last two elements of a stack without using push or pop instructions. They will only access the stack using the BP and SP registers.
4. The PC speaker is programmed as follows:
   a) the frequency of the sound is programmed in the next sequence:

```
MOV   AL, 36H               ;the 8253's circuit mode word
OUT   43H, AL
MOV   AX, FRECVENTA         ;the frequency is loaded to ax
OUT   42H, AL        ;the least significant byte is sent
MOV   AL, AH
OUT   42H, AL        ; the most significant byte is sent
```

b) the sound is being validated:

```
IN    AL, 61H
OR    AL, 3   ;logical or between al and immediate data
                     ;the validation bits are positioned
OUT   61H, AL
```

c) the sound is invalidated:

```
IN    AL, 61H
AND   AL, 0FCH       ;logical and between al and
                     ;immediate data
                     ;the validation bits are erased
OUT   61H, AL
```

N.B. Previous example may not work on arbitrary PC configuration. In time I/O port addresses may change.

5. Write a program that fills a 5 byte memory area, located at consecutive addresses with a value that is loaded by direct addressing to al. They will write more programs, using different addressing modes. Which program is the most efficient?
6. Write a program that transfers two memory words that are placed at successive addresses to another address, using the stack instructions.
7. The students have to write the shortest program that duplicates the last 10 words that were put on the stack, to the stack.

Solved problems:
Modify the content of two words from the memory, using their far addresses (32 bit address). Hint: use the LDS and LES instructions.

Solution:

```
_DATA SEGMENT PUBLIC 'DATA'
      X          DW 10
      Y          DW 15
      ADR_X      DD X
      ADR_Y      DD Y
_DATA ENDS
_CODE SEGMENT PARA PUBLIC 'CODE'
 ASSUME CS:_CODE
 START      PROC FAR
      PUSH DS
      XOR AX,AX
      PUSH AX
      MOV   AX, _DATA          ;initializing the segment register
      MOV   DS, AX
      LDS   SI, ADR_X     ;load address of x to DS:SI -> far address
                               ; 32 bits
      LES   DI, ADR_Y     ;load address of y to ES:DI -> far address
                               ;32 bits
      MOV   WORD PTR [SI], 20  ;the x variable is modified, by indexed
                               ;addressing
      MOV   WORD PTR ES:[DI], 30      ;the y variable is modified, by
                                      ;indexed addressing
      RET                 ;exiting to DOS
START ENDP
_CODE ENDS
      END START
```

The program reads all the keys from the keyboard, until 0 is pressed. It will display the ASCII codes of these keys. Use the XLAT instruction.

```
_DATA SEGMENT
      TAB_CONV DB '0123456789ABCDEF'      ;conversion table
      MESAJ        DB    '-HAS THE ASCII CODE'
      TASTA        DB    2 DUP (?) , 0DH, 0AH, '$'
_DATA ENDS
```

```
_COD SEGMENT PARA PUBLIC 'CODE'

 ASSUME CS:_COD, DS:_DATA
 START  PROC FAR
         PUSH DS
         XOR AX,AX
         PUSH AX
         MOV   AX, _DATA            ;initializing the data segment register
         MOV   DS, AX

 AGAIN:
         MOV   AH, 1          ;echo reading of a key
         INT   21H
         CMP   AL, '0'
         JZ    FINISHED
         MOV   AH,AL                ;saving key code
         LEA   BX, TAB_CONV         ;the conversion table's offset to BX
         AND   AL, 0FH              ;only the first 4 bits (nibble) are taken
         XLAT  TAB_CONV             ;convert the second nibble
         MOV   TASTA+1, AL          ;it is the code's second digit
         MOV   AL, AH               ;the initial code of the key
         MOV   CL, 4                ;we shift to the right with 4 positions
         SHR   AL, CL               ;shift
         XLAT  TAB_CONV             ;converting the first nibble
         MOV   TASTA, AL            ;the ASCII code of the first nibble
                                    ;
         LEA   DX, MESAJ
         MOV   AH, 9H               ;display the key's code
         INT   21H
         JMP AGAIN

FINISHED :
         RET
START ENDP
 _COD ENDS
         END START
```