TRANSFER OF CONTROL WITHIN A PROGRAM I — ASSIGNMENT 7

Object To see how the normal sequential execution of machine instructions can be broken and to gain an appreciation of the usefulness of this type of operation in a simple application.

Approximate time required Two hours

'I want you to do something different'

'Buy it if it costs less than £50'

'Do that ten times and then . . . '

7.1 Introduction

The previous assignments have been concerned with program instructions from the data transfer and data manipulation groups. The flexibility and versatility of the stored program concept on which computers are based, results primarily from the computer's ability to transfer control, or branch, to an instruction that is not in sequential order. This is achieved with instructions in the branching and transfer of control group. All the instructions in this group act on the program counter and, as will be seen, it is possible to execute a block of instructions many times over with the number of times determined by program data.

7.2 Jump instructions

A jump instruction is used to break normal sequential execution and branch to a different part of the program. This is accomplished by loading the address of the desired next instruction into the program counter thus forcing the processor to fetch the contents of this new location for its next instruction opcode. This new address is usually specified in the instruction.

e.g. JMP 28B3

which in machine code looks like:

address	A	C3	JMP operation
	A + 1	B3	l.s. byte of address
	A + 2	28	m.s. byte of address

Execution of this instruction causes an unconditional jump to memory location 28B3 for the next instruction.

When writing a program in symbolic assembly language however, it is usual to use a label to indicate the destination address of a jump instruction which is only translated into its absolute hexadecimal address form during coding. Thus the above instruction would take the form:

JMP LB1

where LB1 is a label associated with the instruction stored at memory location 28B3.

There is another type of jump instruction which results in the next instruction being fetched from the specified branch address only if a specified condition is satisfied. Otherwise the next sequential instruction is fetched. These are conditional branches.

e.g JNZ LB1

read as jump-not-zero to LB1 results in the next instruction being fetched for the address associated with LB1 only if the zero flag of the processor is not set (Z = 0).

All conditional jump instructions result in the status of a processor flag being examined to see if the branch is to be executed.

The conditions that may be specified are summarised in the table below.

Op-Code	Condition	Flag Status
JNZ	not zero	Z = 0
JZ	zero	Z = 1
JNC	no carry	C = 0
JC	carry	C = 1
JPO	parity odd	P = 0
JPE	parity even	P = 1
JP	plus	S = 0
JM	minus	S = 1

Conditional Jump Instructions

Exercise 7.1

Use single step execution to observe the action of the unconditional jump instructions below.

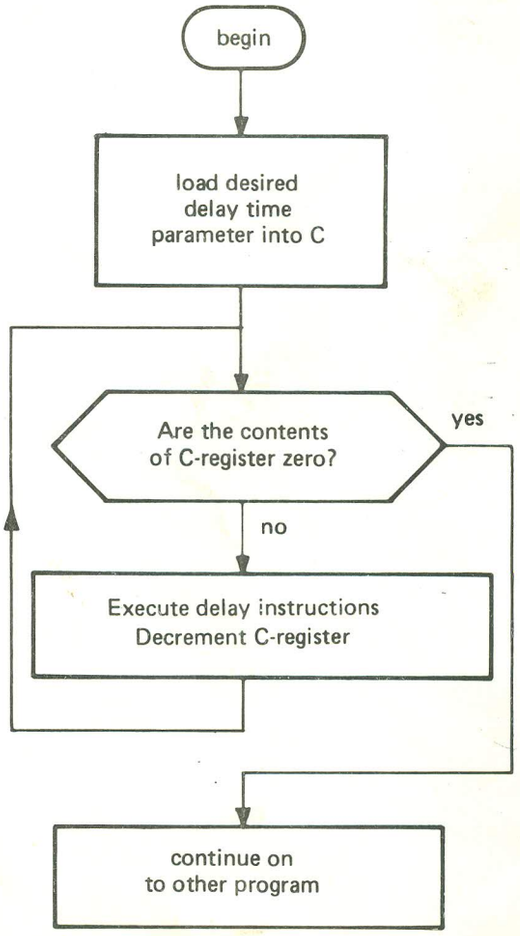
Address	Instruction
2800	JMP 2810
	.
	.
	.
2810	JMP 2800

Example 7.1

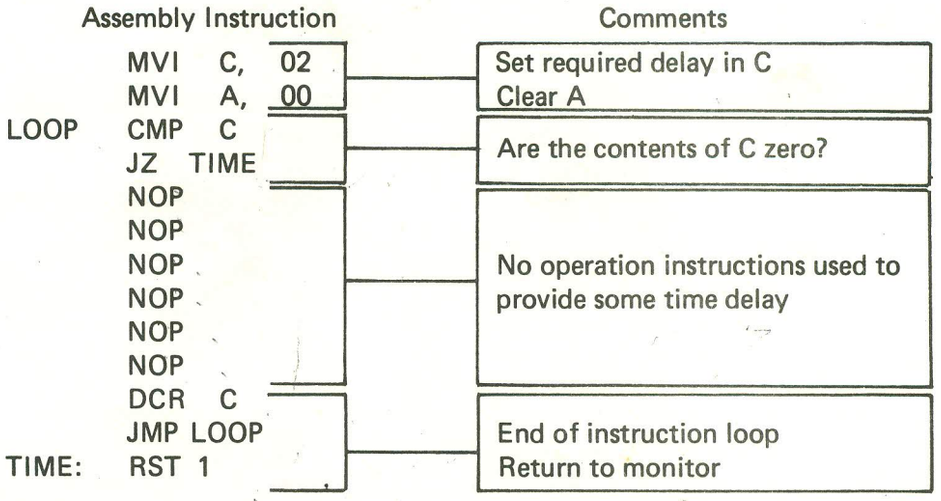
A common requirement when a microcomputer is interfaced to other equipment is to compute a time delay in the program. Since each machine instruction takes a specific amount of time to be executed (typically a few microseconds), it is possible to compute a time delay by executing a group of instructions a preset number of times. The delay obtained is then approximately proportional to the number of times this group of instructions (or loop) is executed.

Before any program involving branching can be written, it is necessary to plan the logical sequence of events to be carried out to achieve the desired goal. A number of techniques may be used, but the most useful for assembly language programs is the construction of a flowchart. This simply is a diagram which indicates the sequence of, and actions required in, a program and the points where branching is required.

The flowchart below is for a simple program to compute a delay. The delay time is determined by the number of program loops that are executed which in turn is determined by the contents of register C.



This flowchart can be translated into the following assembly language program.



In this program, six no-operation instructions (NOP) have been used to provide a basic delay without affecting any processor registers or memory. These can be substituted with different instructions as will be seen in the next assignment.

The restart instruction, RST 1, is used here simply to return control to the monitor program after execution of the time delay. It will be explained more fully in Assignment 10.

Exercise 7.2

Code example program 7.1 and use single step execution to follow the conditional and unconditional branching instructions. Before execution of the jump-on-zero (JZ) instruction, inspect the most-significant-but-one bit of the flags register (the zero flag) and hence predict the next instruction to be executed after the conditional branch.

SUMMARY

Conditional and unconditional branch instructions have been examined. An appreciation of their use and how a flowchart can be used to plan a program has been gained.

TRANSFER OF CONTROL WITHIN A PROGRAM II — ASSIGNMENT 8

Object To examine the machine instructions associated with subroutines and to see how subroutines can be used in a simple application example.

Approximate time required Two hours

'You needn't do that every time. Charlie does it for all the programs – it saves trouble, paper and errors if he does it.'

8.1 Introduction

Frequently in a program it may be necessary to perform a sub-task many times over. It is very desirable not to have to repeat the section of machine code each time the sub-task is required. To avoid this necessity, special instructions are made available, which enable a single copy of instructions for the sub-task to be used as many times as required. This single set of instructions is then called a *subroutine.* Subroutines not only save program memory because repetition of instructions is avoided; they also provide the opportunity to construct a programme in convenient sections which can be written and tested independently, and then collected together to perform the overall task.

8.2 Subroutine instructions

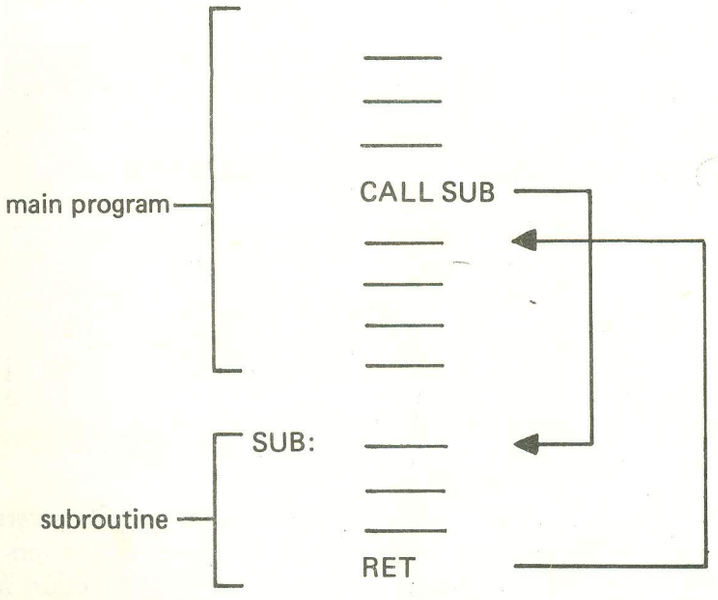
In order to make a subroutine available for repeated use it must be possible to transfer control from the main program at any desired point to the start of the subroutine, and when the subroutine is finished to return control to the next sequential instruction in the main program. The two basic instructions provided to enter a subroutine and return from it are:

```
CALL   nnnn
```

and

```
RET
```

where nnnn is the two-byte starting address of the subroutine. As illustrated in the following program memory diagram, the instruction CALL SUB at any point in the main program causes a jump to the subroutine starting at location SUB. The subroutine must end with the instruction RET, which returns control to the main program at the instruction following the CALL.



When the microprocessor executes a subroutine CALL instruction, in addition to setting the programme counter to the address (labelled SUB in the example) of the start of the subroutine, it must also remember the current address reached by the main program so that it can return control to the right point when the RETurn instruction is executed. This is accomplished by saving the current contents of the program counter register (PC) on the stack.

8.3 The stack

A stack is simply a last-in first-out queue. It is implemented as a set of successive read/write memory locations, together with a register called a 'stack pointer' which holds the address of the entry at the top of the stack. In principle several stacks may be set up, but in this book 'stack'

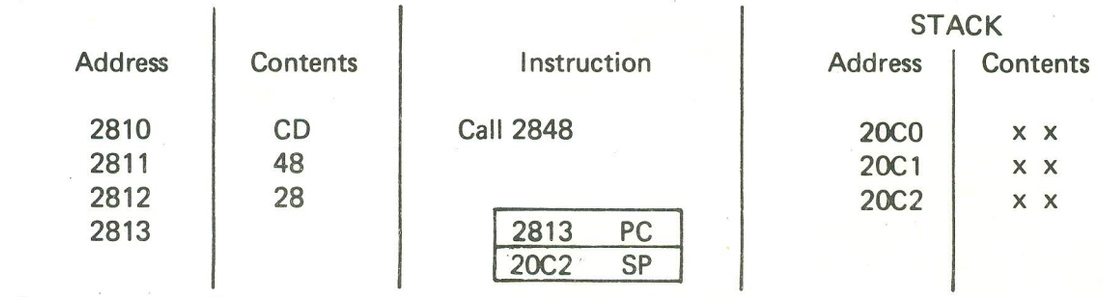
refers only to the stack for which special provision is made in the microprocessor. This uses as the stack pointer the register SP. SP may be initially set at any arbitrary value, although usually on the MAT385 it is set to 20C2 by the instruction 'Initialise stack pointer'.

A byte is entered into the stack by first decrementing the stack pointer, so that it contains the next address beyond the head of the stack, and then placing the byte in the location at that new address. This operation is sometimes referred to as 'pushing' the byte onto the stack.

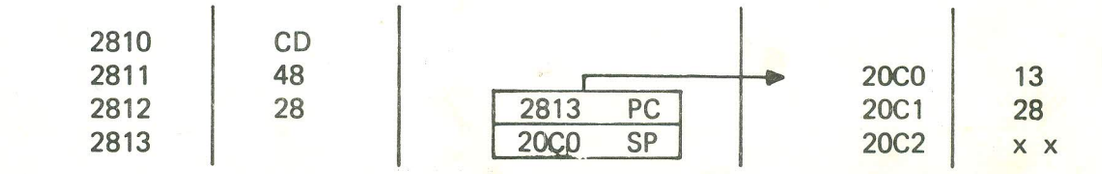
When a byte is no longer required to remain on the stack the converse process, 'popping' the byte from the stack, must first transfer the byte to wherever it is next required, and then increment the stack pointer so that it still points to the head of the reduced stack.

The CALL and RET instructions are just two which push and pop information. The following program diagram shows in more detail how the stack is used by them. Note that two bytes are pushed or popped in each case.

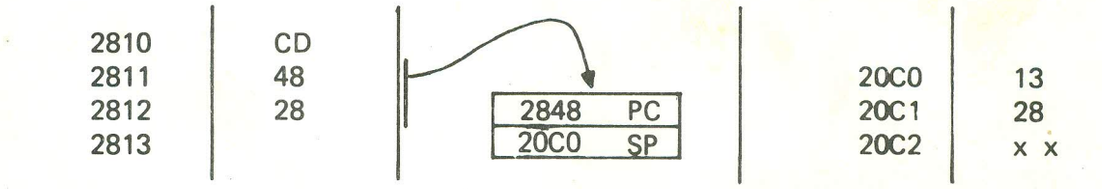
8.4 Operation of Subroutine instructions



i) Instruction CALL just started to be executed – PC updated.



ii) PC contents saved on stack and SP decremented to point to last entry



iii) Control transferred to subroutine start address 2848.

Whenever a RETurn instruction is executed, the process illustrated is reversed. The program counter is loaded (least significant byte first) with the contents at the top of the stack, the stack pointer is incremented twice and control returns to the instruction following the subroutine call. Subroutine call and return instructions are available that are conditional in just the same way and for the same range of conditions as conditional jump instructions.

8.5 Other uses of the stack

The stack may also be used as a temporary deposit for processor register contents.

For example: PUSH PSW (Push processor status word)

This pushes the contents of the accumulator and flags register into the top of the stack. There

The range of exclusive OR instructions provided is similar to the OR instructions and is used extensively in detecting and correcting errors that may occur when transmitting binary information.

For example,

XRA M

A	E

This results in the bit-by-bit exclusive OR operation between the contents of the A-register and the contents of the memory location whose address is contained in the H and L registers. The result is placed in the A-register:

$$(A) \leftarrow (A)\ XOR\ ((H)(L))$$

e.g.

$$(A) = 10011011$$
$$((H)(L)) = 11001101$$
$$(A)\ XOR\ ((H)(L)) = 01010110$$

P is set (even)

6.4.4 Rotate

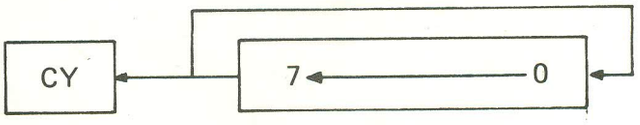
The three previous instruction types – logical AND, OR, and XOR – performed the bit-by-bit logical operation between two 8-bit patterns. In addition, the logical group contains instructions to shift (rotate) a binary value left or right one place. This is useful, for example, when performing binary multiplication and division: a left shift is a x2 operation and a right shift is a ÷ 2 operation.

For example,

RLC

0	7

The contents of the A-register are rotated left one place. The least significant bit and the carry flag are both set to the value shifted out of the most significant bit position:



e.g

(A) = 1 0 0 1 0 1 1 0

RLC (A) = 0 0 1 0 1 1 0 1 (CY)←1

6.4.5 Compare

The compare instructions are very useful since they compare two values – the contents of the A-register and either immediate data or the contents of a processor register or memory location – without modifying either value. The result of the comparison affects the flags and as will be seen in the next assignment, these may be tested to determine the next operation to be performed.

e.g.

CMP B

B	8

The contents of the B-register are compared with the contents of the A-register. The Z flag is set to 1 if the contents are equal, the CY flag is set to 1 if the contents of A are less than the contents of B.

is a similar instruction for each of the pairs of registers, BC, DE and HL. A register pair may be loaded with the contents of the top of the stack by a POP instruction.

e.g. POP BC

transfers the contents of the address given by SP to register C and the contents of the address given by SP+1 to register B.

Example 8.1

Example 7.1 was a sub-program to compute a time delay. This can conveniently be made into a subroutine by replacing the RST 1 instruction at the end by a RET instruction. In order to make this subroutine useful for other parts of an imaginary application program, it is desirable to set the delay time in the main program and not in the subroutine. In this way the same subroutine can implement a variety of delay times.

The contents of processor registers are a convenient means of passing parameters to a subroutine.

```
main program
        LXI   SP, 20C2      Initialise stack pointer
        MVI   C,  02        Load TIMDLY parameter
        CALL TIMDLY
        RST 1               Return to monitor
                .
                .
                .
sub-routine
TIMDLY: MVI   A,  00        Time delay subroutine
 LOOP : CMP   C
        JZ   TIME
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        DCR   C
        JMP   LOOP
 TIME : RET
```

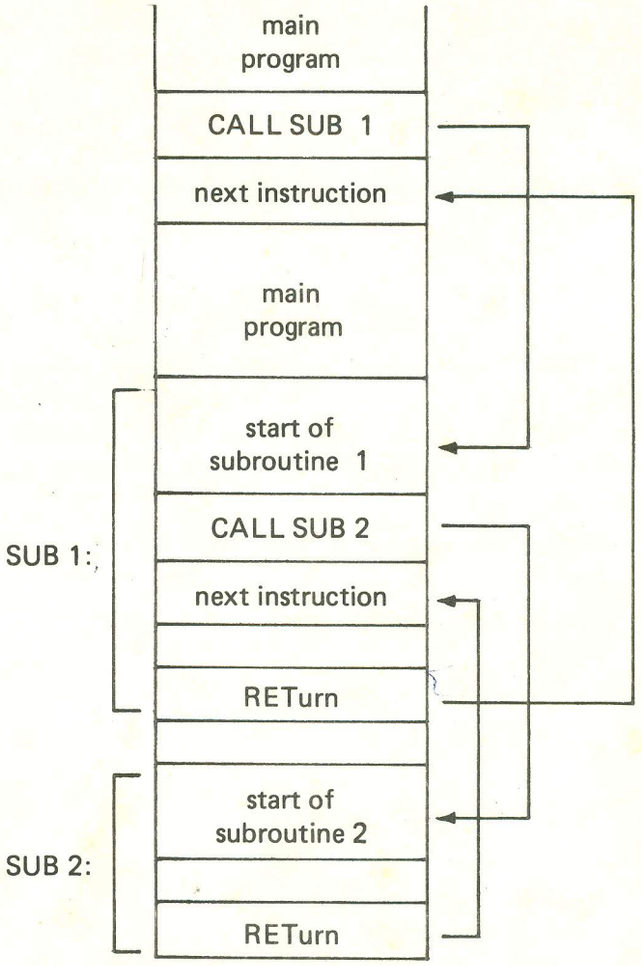
Exercise 8.1

Code the program of example 8.1 and use single step execution to confirm its operation. The program can also be executed at full speed; however, because the delay time is very short, the program returns control to the monitor apparently immediately.

8.6 Nested subroutines

The time delay which the above program implements can be greatly extended by making the basic delay instructions require more time to be executed. One way of doing this is to replace the NOP instructions with a subroutine call. There is a subroutine already stored in the monitor program memory which implements a time delay. The delay parameter for this subroutine is passed via the DE register pair – the larger the contents of DE (treated as a 16-bit value) the longer the delay time. The maximum delay of this subroutine is 0.5 seconds. By using this as the basic delay unit in subroutine TIMDLY the program can implement a delay of many seconds (up to over two minutes).

It is quite in order for a subroutine to call another subroutine in this way and indeed for that to call another. The stack mechanism ensures correct return address linkage. Subroutines may be nested in this way to a depth determined only by the available read/write memory for the stack.



Nested subroutines

If a subroutine called by a program overwrites the contents of some processor registers, the calling program can use the stack to save their contents if needed later.

Example 8.2

The program of the previous example can be extended to incorporate the monitor delay subroutine as the basic time delay unit by replacing the six NOP instructions with:

```
LXI  D,  FFFF

CALL DELAY          (CALL 05F1)
```

Subroutine TIMDLY therefore becomes:

```
TIMDLY : MVI   A   00
  LOOP : CMP   C
         JZ    TIME
         LXI   D,  FFFF     | maximum delay from
         CALL DELAY         | monitor subroutine DELAY
         DCR   C
         JMP   LOOP
  TIME : RET
```

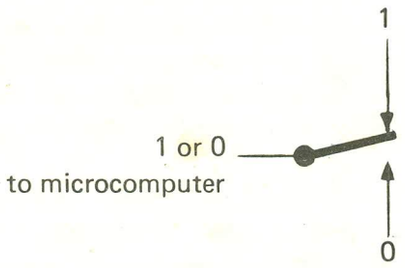
9.1 Introduction

Assignment 3 introduced the basic concept of a microcomputer acting as a digital component that can examine digital input signals and perform functions as a consequence of these inputs to yield output signals. External devices outside the microcomputer produce or accept signals which are not necessarily digital in nature and hence not compatible in a voltage level sense with the inputs and outputs of the microcomputer.

Special interface circuitry is therefore required to transform these external signals into a form suitable for the microcomputer. The circuitry involved in this function is often specifically designed for a particular application. However, this assignment is concerned with the basic mechanisms used in a microprocessor system to produce logical inputs and outputs. These may then require additional circuitry to *interface* the microcomputer to the specific external input and output signals. This latter aspect will be dealt with in the context of specific applications.

9.2 Simple inputs and outputs

A simple logical input to a microcomputer can be produced by a single pole switch. The figure below shows how a logic 0 or 1 voltage level can be presented to the microcomputer depending on the switch position.



Similarly, a simple logical output from a microcomputer can be displayed using a light emitting diode (LED).

The MAT385, therefore, contains a set of 8 switches which can be used as logical inputs and a set of 8 LEDs which can be used to display logical outputs. A logical 1 corresponds to an LED being on and a logical 0 to an LED being off.

An LED indicator or a switch cannot be connected directly to the microcomputer bus since the bus is used by the microprocessor to communicate with all the devices in the system. The information on the bus is therefore continuously changing as instructions are fetched from memory and executed. Information intended for an output indicator must therefore be *latched* by a suitable circuit. The processor can then send data to the output latching device which captures the data at the appropriate time determined by bus control signals and then provides a continuous output until new data is sent to it. Similarly, an input from a switch must be isolated from the data bus until the microprocessor is ready to read its logical value.

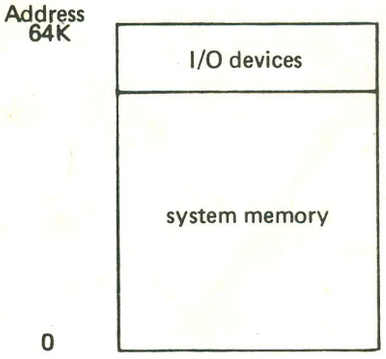
Considerable flexibility is available if a system incorporates a programmable input-output device to provide the necessary latching and isolation. These devices are usually organised into ports or groups of inputs and outputs, often of 8 bits. Each port can usually be programmed to be an input port or an output port or sometimes a mixture of inputs and outputs.

9.3 Memory mapped and programmed I/O

A typical microcomputer system may contain a number of input-output devices all connected to the same bus. It is necessary for each device to be separately addressed by the microprocessor.

There are basically two approaches to organising the addressing associated with the transfer of input-output data between a microcomputer bus and an input-output device. The first, memory mapped input-output, partitions the available memory address space of the microcomputer into two areas. One is a range of addresses associated with actual system memory (ROM and RAM), the other area is reserved for input-output devices. A memory map illustrating this ap-

proach is shown below.

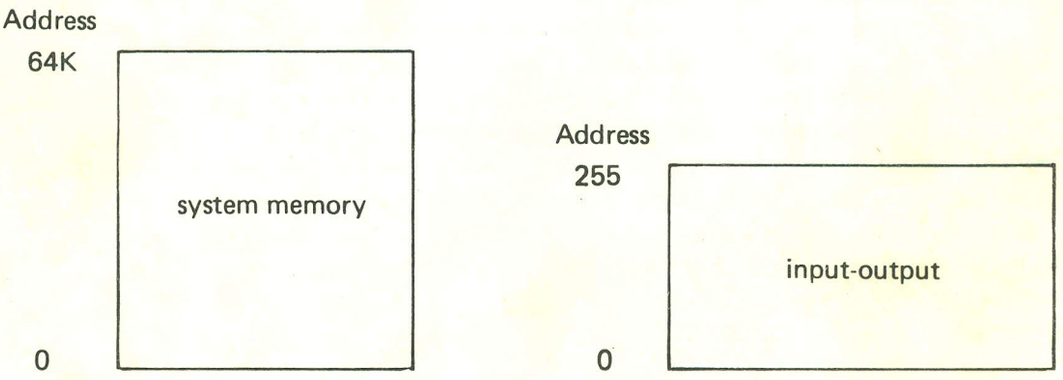


Memory mapped I/O

Each input or output operation with this method is similar to a normal memory access and indeed the same instructions are used for both memory and input-output data transfers. The appropriate address is output on the address bus and recognised either by a memory device or an input-output device (or port) and the appropriate data is transferred on the data bus. This approach therefore has the advantage of having all the addressing modes used for memory access available for input-output data transfers. The major disadvantage is that the range of addresses available for memory is restricted.

The alternative approach is programmed input output (or mapped input-output). With this method, input-output data transfers are accomplished by means of special instructions executed by the processor – IN and OUT for the Intel 8085. The microprocessor generates an input-output request signal to inform input-output devices (and memory) that the address on the address bus is for an input-output device. This provision means that no system memory space has to be reserved for input-output devices.

A typical scheme is illustrated in the following figure.



Programmed input-output on the Intel 8085

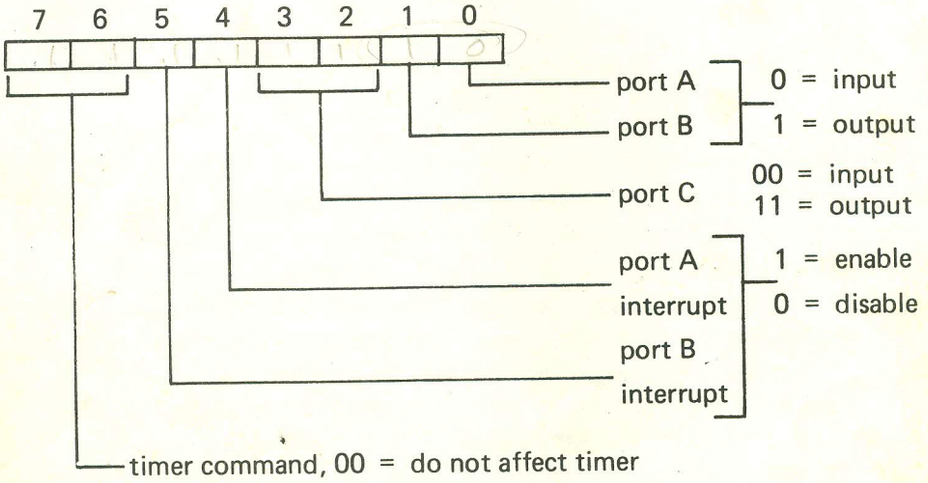
Note that although the Intel 8085 has, in common with most types of microprocessors, programmed input-output instructions, it is also possible to design a memory mapped input-output scheme around the microcomputer bus.

9.4 Port initialisation on the MAT385

Input-output on the MAT385 is mainly organised into a number of 8-bit ports and programmed input-output is used. This assignment is concerned with the input and output of data to and from the two ports associated with the row of eight switches and the eight LEDs.

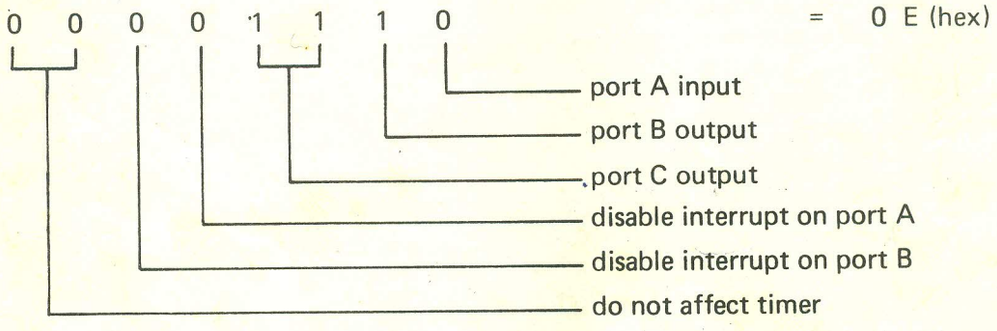
These two ports are part of a flexible, programmable input-output device. They are in fact incorporated in a RAM memory integrated circuit (an Intel 8155) which also incorporates a timer device. Conceptually, however, these input-output ports and the timer are quite separate from the RAM memory and can therefore be considered independently.

Since this programmable input-output device may be programmed so that its ports operate in either an input or output mode, it must first be programmed into the configuration intended. This is achieved by writing to it command information. For the 8155 this command information is a single byte in which each bit is assigned some command significance:



This command byte indicates that the timer and the input-output ports are programmed together and that there is a third port (C). This port is used to control the input-output interface and for this assignment is set as an output port. The interrupt capability of ports A and B is not used. The command byte is transferred to the Intel 8155 using a specific address, 20 (hex), and the programmed output instruction, OUT.

A typical command byte is:



The instruction

OUT address

transfers the contents of the A-register to the addressed input-output device. Hence the instruction sequence:

```
MVI  A  0E

OUT  20
```

transfers command data 0E (hex) to the command register in the Intel 8155. This configures port A to be 8 inputs, port B to be 8 outputs, port C to be an output port and the timer is not affected.

Data is transferred between the processor and these ports by using an IN or OUT instruction with a specific address for each port. Thus the instruction:

IN address

transfers data from the addressed port to the A-register.

The addresses used in this assignment are summarised in the following table:

Address (hex)	Use
20	command register
21	port A data
22	port B data
23	port C control

On the MAT385 the eight switch inputs are connected to port 21 and the eight LED output indicators are connected to port 22, port 23 must be set to 00 to enable the switch inputs.

The 'single step' command provided by the monitor program uses the timer within the same device to determine when a single instruction has been executed. The monitor program, therefore, modifies the command byte within the device as each single step operation is performed. It is necessary, when performing input-output operations using the single step facility, to write the intended command byte in memory location 20FF as well as in the command register. The monitor program then automatically transfers the information in this memory location to the command register as each single step operation is executed. This is illustrated in the following example.

Example 9.1 Transferring data to the LEDs

In the following example, port 22 is configured to be an output port and port 21 to be an input port. An arbitrary data byte is transferred to the LED indicators on port 22.

```
MVI   A,  0E      command byte
STA   20FF        save command byte to permit
                  single step operation
OUT   20          transfer command to 8155
MVI   A   00      load control port data
OUT   23          set control port
MVI   A,  55      load LED data
OUT   22          transfer data to LEDs
```

Exercise 9.1

Code the above program and load it into memory starting at address 2800. Use the 'single step' command to confirm its operation. Change the LED data and repeat the exercise. Replace the OUT 22 instruction with an IN 21 instruction and single step through the program. After execution of the IN instruction, the accumulator should contain the switch input data; confirm this using the 'examine registers' command.

Example 9.2

The example and exercise above can be extended to provide a continuous transfer for data set on the switches to the LED indicators.

```
                MVI   A,  0E
                STA   20FF
                OUT   20
                MVI   A   00
                OUT   23
IN LABEL:       IN    21
                OUT   22
                JMP IN   LABEL
```

Exercise 9.2

The program above can be executed continuously after its operation has been confirmed by single step operation. Changing a switch input will result in the corresponding indicator changing.

This program can be adapted to produce, for example, indicator outputs that are the complement of the switch inputs, by complementing the input data. A time delay between reading the switches and writing to the indicators may be introduced using the delay subroutine introduced in the previous assignment.

SUMMARY

The nature of input-output in a microcomputer system has been introduced. The exercises have involved the configuration of a programmable input-output device and have illustrated simple input and output data transfers.