**UNIT III**

**Addressing modes, instruction set and assembly language programming of 8051**

**ADDRESSING MODES OF 8051 :**

The way in which the data operands are accessed by different instructions is known as the addressing mode. There are various methods of denoting the data operands in the instruction. The 8051 microcontroller provides five distinct addressing modes. They are

1. Immediate addressing mode

2. Register addressing mode

3. Direct Addressing mode

4. Register Indirect addressing mode

5. Indexed addressing mode

**Immediate addressing mode :** The addressing mode in which the data operand is a constant and it is a part of the instruction itself is known as Immediate addressing mode. Normally the data must be preceded by a # sign. This addressing mode can be used to transfer the data into any of the registers including DPTR.

MOV A, # 27 H ; The data (constant) 27 is moved to the accumulator register

ADD R1, # 45 H ; Add the constant 45 to the contents of the accumulator

MOV DPTR, # 8245H ; Move the data 8245 into the data pointer register.

MOV P1, # 21 H ; Move the data 21H to port1

**Direct addressing mode**: The addressing mode in which the data operand is in the RAM location (00 -7FH) and the address of the data operand is given in the instruction is known as Direct addressing mode. The direct addressing mode uses the lower 128 bytes of Internal RAM and the SFRs.

MOV R1, 42H ; Move the contents of RAM location 42 into R1 register

MOV 49H, A ; Move the contents of the accumulator into the RAM location 49

ADD A, 56H ; Add the contents of the RAM location 56 to the accumulator

**Register addressing mode** : The addressing mode in which the data operand to be manipulated lies in one of the registers is known as register addressing mode.

MOV A, R0 ; Move the contents of the register R0 to the accumulator

ADD A, R6 ; Add the contents of R6 register to the accumulator

MOV P1, R2 ; Move the contents of the R2 register into port 1

MOV R5, R2 ; This is invalid. The data transfer between the registers is not

allowed.

**Register Indirect addressing mode :** The addressing mode in which a register is used as a pointer to the data memory block is known as Register indirect addressing mode.

MOV A, @ R0 ; Move the contents of RAM location whose address is in R0 into

A (accumulator)

MOV @ R1, B ; Move the contents of B into RAM location whose address is

held by R1

When R0 and R1 are used as pointers, they must be preceded by @ sign

**One of the advantages of register indirect addressing mode is that it makes accessing the data more dynamic than static as in the case of direct addressing mode.**

**Indexed addressing mode :** This addressing mode is usedin accessing the data elements of lookup table entries located in program ROM space of 8051.

MOVC A, @ A+DPTR

The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Here C denotes code. In this instruction the contents of A are added to the 16-bit DPTR register to form the 16-bit address of the data operand.

**Instruction Set of 8051:**

The 8051 microcontroller provides the following groups of instructions.

1. Arithmetic instructions
2. Logical and Compare instructions
3. Single bit instructions
4. Loop, Jump and Call instructions
5. Data transfer instructions

**Arithmetic instructions :**

This group of operators perform arithmetic operations such as addition, subtraction, multiplication and division etc. Arithmetic operations affect the flags, such as Carry Flag (CY), Overflow Flag (OV) etc, in the PSW register.

**ADD (Addition of two 8 bit numbers) :**

ADD A, Source ; A = A + Source

ADD A, #data ; A= A + immediate data

ADD A, Rn ; A = A + [ Rn ]

ADD A, direct ; A = A + [Direct memory]

ADD A, @Rn ; A = A + [Memory pointed to by Rn]

The instruction ADD is used to add two operands. The destination operand is always in register A while the source operand can be a register, immediate data or in memory.

Ex: ADD A, R1

ADD A,#23

**ADDC (Addition of two 16 bit numbers with carry) :**

ADDC A, Source ; A = A + Source + 1

ADDC A, #data ; A= A + immediate data + CY

ADDC A, Rn ; A = A + [ Rn ] + CY

ADDC A, direct ; A = A + [Direct memory] + CY

ADDC A, @Ri ; A = A + [Memory pointed to by Ri] + CY

The instruction ADDC (add with carry) is used to add two 16-bit numbers. It is used when a carry is propagated from lower byte to higher byte.

Ex: ADDC A, R6

ADDC A, #45

Note : @Ri implies contents of memory location pointed to by R0 or R1 and Rn refers to registers R0-R7 of the currently selected register bank.

**DA (Decimal adjust for addition) :**

DA A ; Adjust for BCD addition

This is a decimal adjust instruction. It adjusts the 8-bit value in ACC resulting from operations

like ADD or ADDC and produces two 4-bit digits (in packed Binary Coded Decimal (BCD) format). Effectively, this instruction performs the decimal conversion by adding 00H, 06H, 60H or 66H to the accumulator, depending on the initial value of ACC and PSW.

Ex: DA A

**SUBB (Subtract with borrow) :**

SUBB A, source ; A = A – source – CY

SUBB A, #data ; A= A - immediate data - CY

SUBB A, Rn ; A = A - [ Rn ] - CY

SUBB A, direct ; A = A - [Direct memory] - CY

SUBB A, @Ri ; A = A - [Memory pointed to by Ri] - CY

The SUBB subtracts the specified data byte and the carry flag together from the accumulator, leaving the result in the accumulator.

Ex: SUBB A, R4

SUBB A, #65

**MUL (Multiplication):**

The 8051 microcontroller supports byte by byte multiplication only.

MUL AB ; A x B, place 16-bit result in A and B

In byte by byte multiplication, one of the operands must be in register A, and the second operand must be in register B. After multiplication, the result is in the A and B registers, the lower byte is in A and the upper byte is in B.

Ex: MUL AB ; A = lower byte, B = higher byte

**DIV (Division) :**

In the division of unsigned numbers, the 8051 supports byte over byte only.

DIV AB ; divide A by B

In dividing a byte by a byte, the numerator must be in register A and the denominator must be in B. After the DIV instruction is performed, the quotient is in A and the remainder is in B.

Ex: DIV AB ; A = quotient and B = remainder

**INC (Increment) :**

INC A ; Increment A by 1 (A = A + 1)

INC Rn ; [Rn] = [Rn] + 1

INC @Ri ; [@Ri] = [@Ri] + 1

INC direct ; direct = direct + 1

This is used to increment the contents of the register by 1.

Ex: INC R1

INC DPTR

**DEC (Decrement) :**

DEC A ; Decrement A by 1 (A = A - 1)

DEC Rn ; [Rn] = [Rn] - 1

DEC @Ri ; [@Ri] = [@Ri] - 1

DEC direct ; direct = direct – 1

This is used to decrement the contents of the register by 1.

Ex: DEC R3

DEC DPTR

**Logical instructions :**

Logical instructions perform standard Boolean operations such as AND, OR, XOR, NOT

(compliment). Other logical operations are clear accumulator, rotate accumulator left and

right, and swap nibbles in accumulator.

**AND (Logical AND):**

ANL destination, source ; dest = dest AND source

ANL A, Rn ; A & [Rn]

ANL A, direct ; A & [direct memory]

ANL A, @Ri ; A & [memory pointed to by Ri]

ANL A, #data ; A & immediate data

ANL direct, A ; [direct] = [direct] & A

ANL direct, #data ; [direct] = [direct] & immediate data

This instruction will perform a logical AND on the two operands and place the result in the destination. The destination is normally the accumulator. The source operand can be a register, in memory, or immediate. The ANL instruction is often used to mask (set to 0) certain bits of an operand.

Ex: ANL A, R1

ANL A, #45

**OR (Logical OR):**

ORL destination, source ; dest = dest OR source

ORL A, Rn ; A | [Rn]

ORL A, direct ; A | [direct memory]

ORL A, @Ri ; A | [memory pointed to by Ri]

ORL A, #data ; A | immediate data

ORL direct, A ; [direct] = [direct] | A

ORL direct, #data ; [direct] = [direct] | immediate data

This instruction will perform the OR operation on the two operands, and place the result in the destination. The ORL instruction can be used to set certain bits of an operand to 1. The destination is normally the accumulator. The source operand can be a register, in memory, or immediate.

Ex : ORL A, R5

ORL A, #76

**XOR (Logical XOR) :**

XRL destination, source ; dest = dest XOR source

XRL A, Rn ; A ^| [Rn]

XRL A, direct ; A ^ [direct memory]

XRL A, @Ri ; A ^ [memory pointed to by Ri]

XRL A, #data ; A ^ immediate data

XRL direct, A ; [direct] = [direct] ^ A

XRL direct, #data ; [direct] = [direct] ^ immediate data

This instruction will perform the XOR operation on the two operands, and place the result in the destination. The destination is normally the accumulator. The source operand can be a register, in memory, or immediate.

Ex: XRL A, R3

XRL A, #32

**CPL (Complement) :**

CPL A ; Complement A

This instruction complements the contents of register A. The complement action changes the 0s to 1s and 1s to 0s. This is also called 1’s complement.

Ex: CPL A

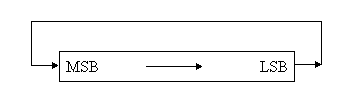
**Rotate and Swap instructions :**

In many applications there is a need to perform a bitwise rotation of an operand. In the 8051 the rotation instructions RL, RR, RLC, and RRC are designed specifically for that purpose. They allow a program to rotate the accumulator right or left. In the 8051, to rotate a byte the operand must be in register A. There are two types of rotations. One is simple rotation of the bits of A, and the other is a rotation through the carry.

**Rotating the bits of A right and left :**

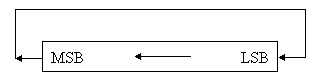
RR A ; rotate right A

In rotate right, the 8 bits of the accumulator are rotated right one bit, and bit D0 exits from the least significant bit and enters into D7 (most significant bit).



RL A ; rotate left A

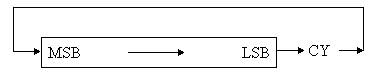
In rotate left, the 8 bits of the accumulator are rotated left one bit, and bit D7 exits from the (most significant bit) and enters into D0 (least significant bit).



**Rotating through the carry :**

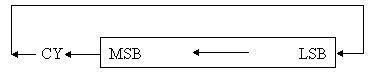
RRC A ; rotate right through carry

In RRC A, as bits are rotated from left to right, they exit the LSB to the carry flag, and the carry flag enters the MSB. In other words, in RRC A, the LSB is moved to CY and CY is moved to the MSB. In reality, the carry flag acts as if it is part of register A, making it a 9-bit register.



RLC A ; rotate left through carry

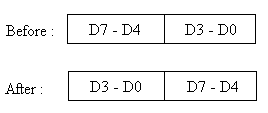
In RLC A, as bits are shifted from right to left they exit the MSB and enter the carry flag, and the carry flag enters the LSB. In other words, in RCL the MSB is moved to CY and CY is moved to the LSB.



**SWAP :**

SWAP A

Another useful instruction is the SWAP instruction. It works only on the accumulator (A). It swaps the lower nibble and the higher nibble. In other words, the lower 4 bits are put into the higher 4 bits and higher 4 bits are put into the lower 4 bits.



Ex: SWAP A

**Single Bit instructions (Boolean Variable instructions) :**

One unique and powerful feature of the 8051 is single bit operation. Single bit instructions allow the programmer to set, clear, move and complement individual bits of a port, memory or register.

**CLR (Clear) :**

CLR bit ; Clear the bit

CLR C ; Clear the carry

This instruction clears (set to 0) the specified bit indicated in the instruction. CLR instruction can operate on the carry flag or any directly addressable bit.

Ex : CLR P2.1

CLR C

**SETB (Set bit ) :**

SETB bit ; Set the bit

SETB C ; Set the carry

This operation sets the specified bit to 1. SETB instruction can operate on the carry flag or any directly-addressable bit.

Ex : SETB P1.3

SETB C

**CPL (Complement ) :**

CPL bit ; Complement the bit

CPL C ; Complement the carry

This operation complements the bit indicated by the operand. CPL instruction can operate on the carry flag or any directly addressable bit.

Ex: CPL P2.1

CPL P2.2

**JB (Jump on bit) :**

JB bit, target ; Jump to target if bit = 1

This instruction jumps to the address indicated if the destination bit is 1, otherwise the program continues to the next instruction. The bit tested is not modified.

Ex: JB ACC.7, target1

JB P1.2, target2

**JNB (Jump on no bit) :**

JNB bit, target ; Jump to target if bit = 0

This instruction jumps to the address indicated if the destination bit is 0, otherwise the program continues to the next instruction. The bit tested is not modified.

Ex: JNB ACC.6, target1

JNB P1.3, target2

**JBC (Jump on bit and clear the bit) :**

JBC bit, target ; Jump to target if bit = 1, then clear the bit

If the source bit is 1, this instruction clears it and branches to the address indicated; else it proceeds with the next instruction.

Ex: JBC P1.3, target1

JBC P1.2, target2

**JC (Jump on carry) :**

JC target ; Jump to target if CY = 1

This instruction branches to the address, indicated by the label, if the carry flag is set, otherwise the program continues to the next instruction.

Ex: JC target1

**JNC (Jump on no carry) :**

JNC target ; Jump to target if CY = 0

This instruction branches to the address, indicated by the label, if the carry flag is **not** set, otherwise the program continues to the next instruction.

Ex: JNC target1

**ANL (AND Logic) :**

ANL C,bit ; AND CY with bit and save it on CY

ANL C,/bit ; AND CY with inverted bit and save it on CY

This instruction ANDs the bit addressed with the carry bit and stores the result in the carry bit itself. If the source bit is a logical 0, then the instruction clears the carry flag; else the carry flag is left in its original value. If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, but the source bit itself is not affected.

Ex : ANL C,P2.7 ; AND carry flag with bit 7 of P2

ANL C,/OV ; AND with inverse of OV flag

**ORL ( OR Logic) :**

ORL C,bit ; OR CY with bit and save it on CY

ORL C,/bit ; OR CY with inverted bit and save it on CY

This instruction ORs the bit addressed with the carry bit and stores the result in the carry bit itself. It sets the carry flag if the source bit is a logical 1; else the carry is left in its original value. If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, but the source bit itself is not affected.

Ex: ORL C,P2.7 ; OR carry flag with bit 7 of P2

ORL C,/OV ; OR with inverse of OV flag

**MOV ( Copy ) :**

MOV b, C ; Copy carry status to bit location (CY = b)

MOV C, b ; copy bit location status to carry (b = CY)

The instruction loads the value of source operand bit into the destination operand bit. One of the operands **must** be the carry flag; the other may be any directly-addressable bit.

Ex : MOV P2.3, C

MOV C, P3.3

**Program branching instructions (LOOP, JUMP and CALL) :**

**Looping instructions :**

Repeating a sequence of instructions a certain number of times is called a loop. In 8051 the loop action is performed by the instruction “DJNZ reg, label”. In this instruction, the register is decremented, if it is not zero, it jumps to the target address referred to by the label. Prior to the start of the loop the register is loaded with the counter for the number of repetitions. Notice that in this instruction both the register decrement and the decision to jump are combined into a single instruction.

Ex: DJNZ R2, label

**Jump instructions:**

In 8051, the jump instructions are divided into two categories as conditional and unconditional.

**Conditional jump instructions :**

The conditional jump is a jump in which the control is transferred conditionally to the target location. Some of the conditional jump instructions are tabulated below.

|  |  |
| --- | --- |
| Instruction | Action |
| JZ | Jump if A = 0 |
| JNZ | Jump if A ≠ 0 |
| DJNZ | Decrement and jump if A ≠ 0 |
| CJNE A,byte | Jump if A ≠ byte |
| CJNE reg, #data | Jump if byte ≠ #data |
| JC | Jump if CY = 1 |
| JNC | Jump if CY = 0 |
| JB | Jump if bit = 1 |
| JNB | Jump if bit = 0 |
| JBC | Jump if bit = 1 and clear bit |

**JZ (Jump on zero) :**

In this instruction the content of register A is checked. If it is zero, it jumps to the target address.

Ex: JZ target

**JNZ (Jump on no zero) :**

In this instruction the content of register A is checked. If it is not zero, it jumps to the target address.

Ex: JNZ target

**CJNE (Compare and jump if not equal ) :**

The 8051 has an instruction for the comparision.

CJNE destination, source, relative address

In the 8051, the actions of comparing and jumping are combined into a single instruction called CJNE ( compare and jump if not equal ). The CJNE instruction compares two operands, and jumps if they are not equal. In CJNE, the destination operand can be in the accumulator or in one of the Rn registers. The source operand can be in a register, in memory, or immediate.

Ex : CJNE A, #0FF, target

CJNE R1, #00, target

All conditional jumps are short jumps, meaning that the address of the target must be within -128 to +127 bytes of the contents of the PC.

**Unconditional jump instructions :**

The unconditional jump is a jump in which control is transferred unconditionally to the target location. In the 8051, there are two unconditional jumps : LJMP (long jump) and SJMP (short jump).

**LJMP (Long jump) :**

LJMP is an unconditional long jump. It is a 3-byte instruction in which the first byte is the opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address allows a jump to any memory location from 0000 to FFFFH.

LJMP target ; Jump to target unconditionally

**SJMP (Short jump) :**

SJMP is an unconditional short jump. It is a 2-byte instruction in which the first byte is the op- code, and the second byte represents the relative address of the target location. The relative address ranges from 00 to FFH.

SJMP target ; Jump to target unconditionally

**CALL instructions :**

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the 8051 there are two instructions for call : LCALL (long call) and ACALL (absolute call).

**LCALL (Long call) :**

It is a 3-byte instruction. In which, the first byte is the op-code and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call subroutines located anywhere within the 64K byte address space of the 8051.

LCALL target ; Jump to subroutine located at target address

**ACALL (Absolute call) :**

It is a 2-byte instruction. In which, the first byte is the op-code and the second byte is used for the address of the target subroutine. ACALL can be used to call subroutines located within 2K bytes address.

ACALL target ; Jump to subroutine located at target address

**RET (Return) :**

When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instruction from the new location. After finishing execution of the subroutine, the instruction RET (return) transfers control back to the caller. Every subroutine needs RET as the last instruction.

RET ; control returns to main program

**Data transfer instructions :**

Data transfer instructions are used to transfer data between an internal RAM location and

SFR location, without going through the accumulator. Data can also be transferred between

the internal and external RAM by using indirect addressing. The upper 128 bytes of data RAM are accessed only by indirect addressing and the SFRs are accessed only by direct addressing.

**MOV (Move):**

MOV destination, source ; destination = source

This instruction moves the source byte into the destination location. The source byte is not affected.

Ex: MOV Ri, direct ; Ri = [direct]

MOV Ri, #data ; Ri = 8-bit immediate data

MOV DPTR, #data 16 ; DPTR = 16-bit immediate data

**MOVC :**

MOVC A,@A+<base register> ; A = Code byte from [@A+<base register>]

This instruction moves a code byte from program memory into ACC. The effective address of the byte fetched is formed by adding the original 8-bit accumulator contents and the contents of the base register, which is either the data pointer (DPTR) or program counter (PC). 16-bit addition is performed and no flags are affected. The instruction is useful in reading the look-up tables in the program memory. If the PC is used, it is incremented to the address of the following instruction before being added to the ACC.

Ex: MOVC A,@A + PC ; A = Code byte from [@A+PC]

MOVC A,@A + DPTR ; A = Code byte from [@A+DPTR]

**MOVX :**

MOVX destination, source ; destination = source

This instruction transfers data between ACC and a byte of external data memory. There are two forms of this instruction, the only difference between them is whether to use an 8-bit or 16-bit indirect addressing mode to access the external data RAM.

Ex : MOVX @DPTR, A ; External[@DPTR] = A

MOVX @Ri, A ; External[@Ri] = A

MOVX A, @DPTR ; A = Data byte from external ram [@DPTR]

MOVX A, @Ri ; A = Data byte from external ram [@Ri]

**PUSH :**

PUSH direct ; PUSH into stack

This instruction increments the stack pointer (SP) by 1. The contents of *Direct*, which is an internal memory location or a SFR, are copied into the internal RAM location addressed by the stack pointer.

Ex: PUSH 22h

PUSH 56h

**POP :**

POP direct ; POP form stack

This instruction reads the contents of the internal RAM location addressed by the stack pointer (SP) and decrements the stack pointer by 1. The data read is then transferred to the *Direct* address which is an internal memory or a SFR.

Ex: POP DPH

POP DPL

**XCH :**

XCH A, byte ; A = byte, byte = A

This instruction swaps the contents of ACC with the contents of the indicated data byte.

Ex : XCH A, Ri ; A = Ri, Ri = A

XCH A, direct ; A = direct, direct = A

XCH A, @Ri ; A = @Ri, @Ri = A

**XCHD :**

XCHD A, @Ri ; Exchange lower order digits

This instruction exchanges the low order nibble of ACC (bits 0-3), with that of the internal RAM location pointed to by Ri register. The high order nibbles (bits 7-4) of both the registers remain

the same.

Ex: XCHD A,@R1

**Time delay generation and calculation :**

For the CPU to execute an instruction takes a certain number of clock cycles. In the 8051 family, these clock cycles are referred to as machine cycles. To calculate the time delay, we use this machine cycles. In the 8051 family, the length of the machine cycle depends on the frequency of the crystal oscillator connected to the 8051 system. The crystal oscillator along with on-chip circuitry, provide the clock source for the 8051 CPU.

In the 8051, one machine cycle lasts 12 oscillator periods. Therefore, to calculate the machine cycle, we take 1/12 of the crystal frequency, then take its inverse.

If the crystal frequency is 11.0592MHz, then

Frequency of the machine cycle = 11.0592MHz/12 = 921.6 KHz

Time period of the machine cycle = 1/921.6KHz = 1.085 s

**Delay calculation :**

A delay subroutine consists of two parts 1) setting a counter and 2) a loop. Most of the time delay is performed by the body of the loop.

MOV A, #55 ; Move the data 55h to A

AGAIN: MOV P1, A ; Move the contents of A to Port1

LCALL DELAY ; Call delay subroutine

CPL A ; Complement the contents of A

SJMP AGAIN ; Go to label AGAIN

DELAY: MOV R3, #0FF ; Move the data FFh to register R3

HERE: DJNZ R3, HERE ; Decrement R3, if R3≠0, then go to label HERE

RET ; Return to main program

We have the following machine cycles for each instruction of the DELAY subroutine.

Machine cycles No. of times the

Instruction is executed

DELAY: MOV R3, #0FF 1 1

HERE: DJNZ R3, HERE 2 255

RET 1 1

Therefore, we have a time delay of [(1x1) + (2x255) + (1x1)] x 1.085s = 555.52s

Another way to get a large delay is to use a loop inside a loop, which is also called a nested loop.

Machine cycles No. of times the

Instruction is executed

DELAY: MOV R5, #0FF 1 1

LOOP: MOV R3, #0FF 1 255

HERE: DJNZ R3, HERE 2 255

DJNZ R5, LOOP 2 255

RET 1 1

Therefore, we have a time delay of [(1x1) + (1x255) + (2x255) + (2x255) + (1x1) ] x 1.085s = 1277 x 1.085s = 1385.545s

**Assembly language programs :**

**Addition :**

MOV A, #DATA1 ; DATA1 is moved to A

MOV R1, #DATA2 ; DATA2 is moved to R1

ADD A, R1 ; Add DATA1 and DATA2 and place the result in A

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A in 8500h

LJMP 03 ; User break

**INPUT :**

DATA1 = 12h

DATA2 = 15h

**RESULT :**

8500h – 27h

**Subtraction :**

CLR C

MOV A, #DATA1 ; DATA1 is moved to A

MOV R1, #DATA2 ; DATA2 is moved to R1

SUBB A, R1 ; Subtract DATA1 - DATA2 and place the result in A

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A in 8500h

LJMP 03 ; User break

**INPUT :**

DATA1 = 25h

DATA2 = 15h

**RESULT :**

8500h – 10h

**Multiplication :**

MOV A, #DATA1 ; DATA1 is moved to register A

MOV B, #DATA2 ; DATA2 is moved to register B

MUL AB ; Multiply the contents of A and B

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A (lower byte) in 8500h

INC DPTR ; Increment the DPTR by 1 (i.e., 8501h)

MOV A,B ; Move the contents of B to A

MOVX @DPTR, A ; Store the content of B (higher byte) in 8501h

LJMP 03 ; User break

**INPUT :**

DATA1 = 02h

DATA2 = 15h

**RESULT :**

8500h – 30h (Lower byte)

8501h – 00h (Higher byte)

**Division :**

MOV A, #DATA1 ; DATA1 is moved to register A

MOV B, #DATA2 ; DATA2 is moved to register B

DIV AB ; Divide the contents of A by B

MOV DPTR, #8500 ; Load the DPTR with 8500h

MOVX @DPTR, A ; Store the content of A (Quotient) in 8500h

INC DPTR ; Increment the DPTR by 1 (i.e., 8501h)

MOV A,B ; Move the contents of B to A

MOVX @DPTR, A ; Store the content of B (Remainder) in 8501h

LJMP 03 ; User break

**INPUT :**

DATA1 = 95h

DATA2 = 10h

**RESULT :**

8500h – 09h (Quotient)

8501h – 05h (Remainder)

**Largest Number :**

MOV DPTR, #8500 ; Load the data pointer with 8500h

MOV R1, #05 ; Move 05h (count) to register R1

MOV B, #00 ; Move 00h to register B

LOOP3: MOVX A, @DPTR ; Move the first data byte to A (i.e.,8500h)

CJNE A, B, LOOP1 ; If A≠B, then go to label LOOP1

LOOP1: JC LOOP2 ; If A<B (CY = 1), then go to label LOOP2

MOV B, A ; Move the contents of register A to B

INC DPTR ; DPTR is incremented by 1

DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

SJMP LOOP4 ; Jump to label LOOP4

LOOP2: INC DPTR ; DPTR is incremented by 1

DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

LOOP4: MOV A, B ; Move the contents of register B to A

MOV DPTR, #8600 ; Load the DPTR with 8600h

MOVX @DPTR, A ; Move the contents of register A to 8600h

LJMP 03 ; User break

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8600h – 56h

**Smallest Number :**

MOV DPTR, #8500 ; Load the data pointer with 8500h

MOV R1, #05 ; Move 05h (count) to register R1

MOV B, #0FF ; Move 00h to register B

LOOP3: MOVX A, @DPTR ; Move the first data byte to A (i.e.,8500h)

CJNE A, B, LOOP1 ; If A≠B, then go to label LOOP1

LOOP1: JNC LOOP2 ; If A>B (CY = 0), then go to label LOOP2

MOV B, A ; Move the contents of register A to B

INC DPTR ; DPTR is incremented by 1

DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

SJMP LOOP4 ; Jump to label LOOP4

LOOP2: INC DPTR ; DPTR is incremented by 1

DJNZ R1, LOOP3 ; Decrement R1, if R1≠0, then go to label LOOP3

LOOP4: MOV A, B ; Move the contents of register B to A

MOV DPTR, #8600 ; Load the DPTR with 8600h

MOVX @DPTR, A ; Move the contents of register A to 8600h

LJMP 03 ; User break

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8600h – 12h

**Ascending order :**

MOV R0, #05 ; Move the count 05h to register R0

LOOP3: MOV DPTR, #8500 ; Load the DPTR with 8500h

MOV A, R0 ; Move the contents of R0 to register A

MOV R1, A ; Move the contents of A to register R1

LOOP2: MOVX A, @DPTR ; Move the contents of DPTR to register A

MOV B, A ; Move the contents of A to register B

INC DPTR ; Increment the DPTR by 1

MOVX A, @DPTR ; Move the contents of DPTR to register A

CLR C ; Clear the Carry (CY = 0)

MOV R2, A ; Move the contents of A to register R2

SUBB A, B ; Subtract A – B, if A>B then CY=0 else CY=1

JNC LOOP1 ; If CY=0, then go to label LOOP1

MOV A, B ; Move the contents of A to register B

MOVX @DPTR, A ; Move the contents of A to DPTR

DEC DPTR ; Decrement the DPTR by 1

MOV A, R2 ; Move the contents of R2 to register A

MOVX @DPTR, A ; Move the contents of A to DPTR

LOOP1: DJNZ R1, LOOP2 ; Decrement R1 by1, if R1≠0, then go to LOOP2

DJNZ R0, LOOP3 ; Decrement R0 by 1, if R0≠0, then go to LOOP3

LJMP 03 ; Stop the execution

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8500h – 12h

8501h – 23h

8502h – 34h

8503h – 43h

8504h – 56h

**Descending order :**

MOV R0, #05 ; Move the count 05h to register R0

LOOP3: MOV DPTR, #8500 ; Load the DPTR with 8500h

MOV A, R0 ; Move the contents of R0 to register A

MOV R1, A ; Move the contents of A to register R1

LOOP2: MOVX A, @DPTR ; Move the contents of DPTR to register A

MOV B, A ; Move the contents of A to register B

INC DPTR ; Increment the DPTR by 1

MOVX A, @DPTR ; Move the contents of DPTR to register A

CLR C ; Clear the Carry (CY = 0)

MOV R2, A ; Move the contents of A to register R2

SUBB A, B ; Subtract A – B, if A>B then CY=0 else CY=1

JC LOOP1 ; If CY=1, then go to label LOOP1

MOV A, B ; Move the contents of A to register B

MOVX @DPTR, A ; Move the contents of A to DPTR

DEC DPTR ; Decrement the DPTR by 1

MOV A, R2 ; Move the contents of R2 to register A

MOVX @DPTR, A ; Move the contents of A to DPTR

LOOP1: DJNZ R1, LOOP2 ; Decrement R1 by1, if R1≠0, then go to LOOP2

DJNZ R0, LOOP3 ; Decrement R0 by 1, if R0≠0, then go to LOOP3

LJMP 03 ; Stop the execution

**INPUT :**

8500h – 23h

8501h – 43h

8502h – 56h

8503h – 12h

8504h – 34h

**RESULT :**

8500h – 56h

8501h – 43h

8502h – 34h

8503h – 23h

8504h – 12h

**Generation of a square wave :**

HERE: SETB P1.0 ; Make P1.0 = 1

LACLL DELAY ; Call delay subroutine

CLR P1.0 ; Make P1.0 = 0

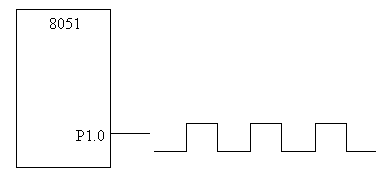
LCALL DELAY ; Call delay subroutine

SJMP HERE ; Go to label HERE

DELAY: MOV R3, #0FF ; Load R3 with FFh

LOOP: DJNZ R3, LOOP ; Decrement R3 by1 and if R3≠0 go to label LOOP

RET ; Return to main program



**Generation of a rectangular wave :**

HERE: SETB P1.0 ; Make P1.0 = 1

LACLL DELAY1 ; Call delay subroutine

CLR P1.0 ; Make P1.0 = 0

LCALL DELAY2 ; Call delay subroutine

SJMP HERE ; Go to label HERE

DELAY1: MOV R5, #0FF ; Load R5 with FFh

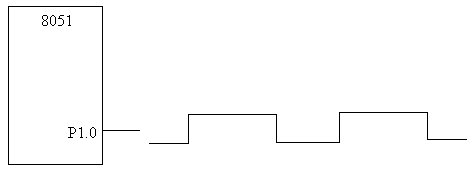
LOOP: MOV R6, #0FF ; Load R6 with FFh

HERE: DJNZ R6, HERE ; Decrement R6 by1 and if R6≠0 go to label HERE DJNZ R5, LOOP ; Decrement R5 by1 and if R5≠0 go to label LOOP RET ; Return to main program

DELAY2: MOV R3, #0FF ; Load R3 with FFh

LOOP: DJNZ R3, LOOP ; Decrement R3 by1 and if R3≠0 go to label LOOP

RET ; Return to main program

****

**@@@**