SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

UNIT – V - MICROCONTROLLER– SECA1404

UNIT 5 MICROCONTROLLER

Introduction - Architecture of 8051 - Memory organization - Addressing modes - Instruction set – Assembly Language Programming - Jump, Loop and Call Instructions - Arithmetic and Logic Instructions - Bit Operations -Programs – Introduction to Arduino.

INTEL 8051 MICROCONTROLLER

- WHAT IS A MICROCONTROLLER?
  - **All of the components needed for a controller were built right onto one chip.**
  - **A one chip computer, or microcontroller was born.**
  - **A microcontroller is a highly integrated chip which includes, on one chip, all or most of the parts needed for a controller.**
  - **The microcontroller could be called a "one-chip solution".**

- **8051 Family**

The 8051 is just one of the MCS-51 family of microcontrollers developed by Intel. The design of each of the MCS-51 microcontrollers are more or less the same. The differences between each member of the family is the amount of on-chip memory and the number of timers, as detailed in the table below.

Table 1 8051 Family

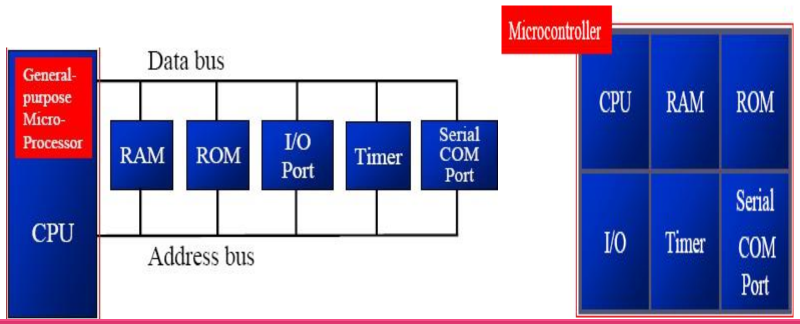
Microcontroller	**On-chip Code Memory**	**On-chip Data Memory**	**Timers**
8051	4K ROM	128 bytes	2
8031	0	128 bytes	2
8751	4K EPROM	128 bytes	2
8052	8K ROM	256 bytes	3
8032	0	256 bytes	3
8752	8K EPROM	256 bytes	3

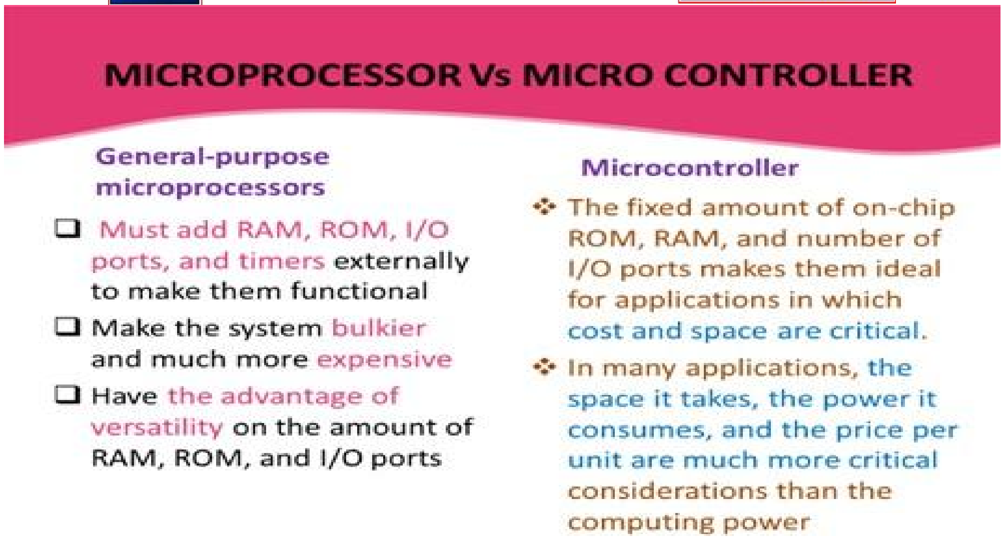
Each chip also contains:

- four 8-bit input/output (I/0) ports
- serial interface

- 64K external code memory space
- 64K external data memory space
- Boolean processor
- 210 bit-addressable locations
- 4us multiply/divide

1.2 <u>MICROPROCESSOR vs MICRO CONTROLLER</u>




MICROPROCESSOR Vs MICRO CONTROLLER

General-purpose microprocessors

- ❑ Must add RAM, ROM, I/O ports, and timers externally to make them functional
- ❑ Make the system bulkier and much more expensive
- ❑ Have the advantage of versatility on the amount of RAM, ROM, and I/O ports

Microcontroller

- ❖ The fixed amount of on-chip ROM, RAM, and number of I/O ports makes them ideal for applications in which cost and space are critical.
- ❖ In many applications, the space it takes, the power it consumes, and the price per unit are much more critical considerations than the computing power

Disadvantages of microprocessor

- The overall system cost is high
- A large sized PCB is required for assembling all the components
- Overall product design requires more time
- Physical size of the product is big
- A discrete components are used, the system is not reliable

Advantages of Microcontroller based System

- As the peripherals are integrated into a single chip, the overall system cost is very less
- The product is of small size compared to microprocessor based system

- As the peripherals are integrated with a microprocessor the system is more reliable
- Though microcontroller may have on chip ROM,RAM and I/O ports, additionROM, RAM I/O ports may be interfaced externally if required
- On chip ROM provide a software security

1.3 8051 Block Diagram

8051 Internal Architecture

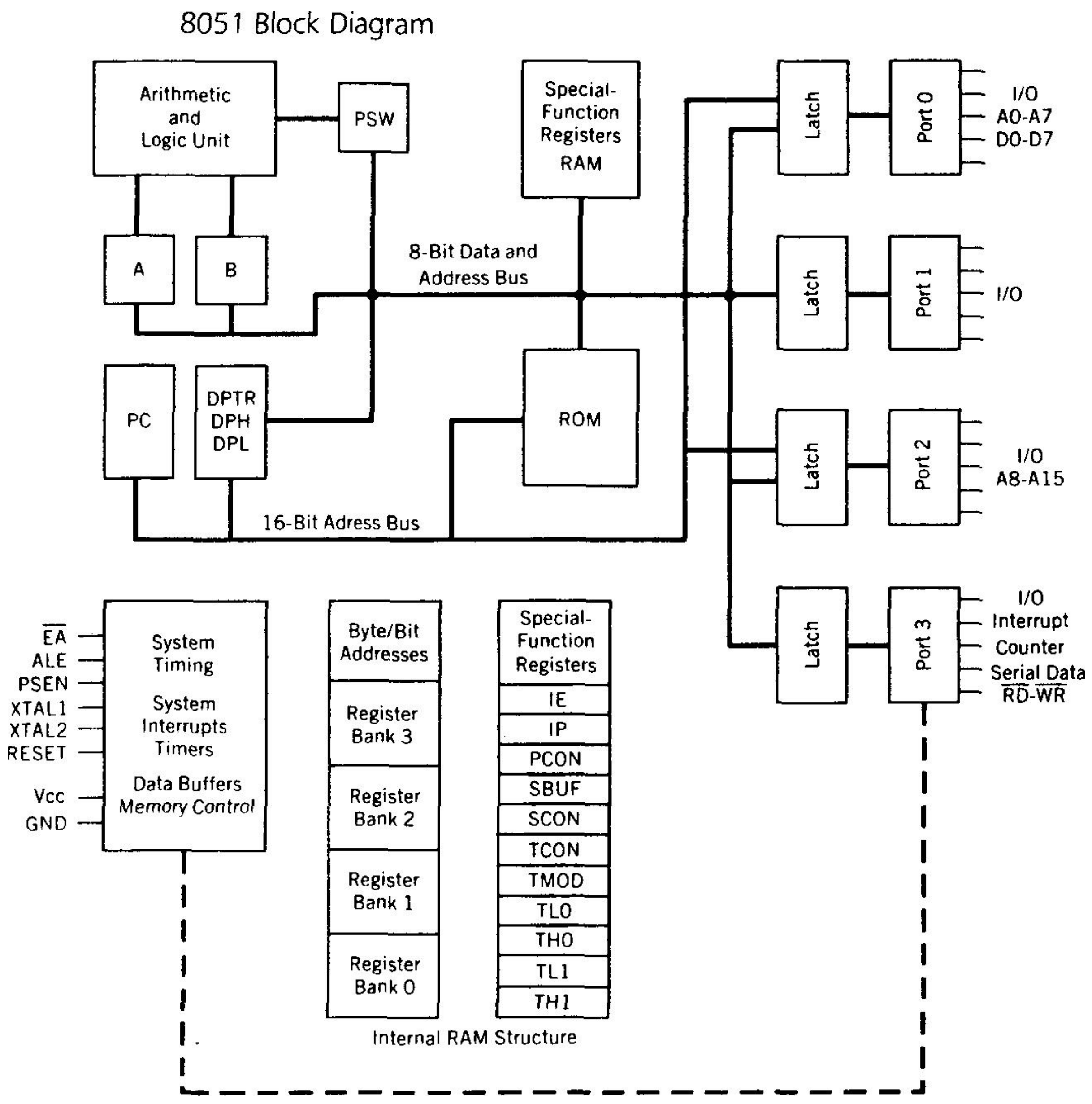


Figure 1. Block Diagram

8051 CPU Registers

- **ACCUMULATOR (ACC)**
  - Operandregister
  - Implicit or specified in the instruction
  - Has an address in on chip SFR bank
- **B REGISTER**
  - to store one of the operands for multiplication and division
  - otherwise, scratch pad
  - considered as a SFR
- **PROGRAM STATUS WORD (PSW)**
  - Set of flags contain status information
  - One of the SFR
- **STACK POINTER (SP)**
  - 8 bit wide register
  - Incremented before data is stored on to the stack using PUSH or CALL instructions
  - Stack defined anywhere on the 128 byte RAM
  - RESET ☾ initiated to 0007H
  - Not a top to down structure
  - Allotted an address in SFR
- **DATA POINTER (DPTR)**
  - 16 bit register
  - contains DPH and DPL
  - Pointer to external RAM address
  - DPH and DPL allotted separate addresses in SFR bank
- **PORT 0 TO 3 LATCHES & DRIVERS**
  - Each i/o port allotted a latch and a driver
  - Latches allotted address in SFR
  - User can communicate via these ports
  - P0, P1, P2,P3
- **SERIAL DATA BUFFER**
  - internally had TWO independent registers
  - TRANSMIT buffer ☾ parallel in serial out (PISO)
  - RECEIVE buffer ☾ serial in parallel out (SIPO)
  - identified by SBUF and allotted an address in SFR

    - byte written to SBUF ☾ initiates serial TX
    - byte read from SBUF ☾ reads serially received data

- TIMER REGISTERS
    - for Timer0 (16 bit register – TL0 & TH0)
    - for Timer1 (16 bit register – TL1 & TH1)
    - four addresses allotted in SFR
- CONTROL REGISTERS
    - IP
    - IE
    - TMOD
    - TCON
    - SCON
    - PCON
    - contain control and status information for interrupts, timers/counters and serial port
    - Allotted separate address in SFR
- TIMING AND CONTROL UNIT
    - derives necessary timing and control signals

For internal circuit and external system bus

- OSCILLATOR
    - generates basic timing clock signal using crystal oscillator
- INSTRUCTION REGISTER
    - decodes the opcode and gives information to timing and control unit
- EPROM & PROGRAM ADDRESS REGISTER
    - provide on chip EPROM and mechanism to address it
    - All versions don’t have EPROM
- RAM & RAM ADDRESS REGISTER
    - provide internal 128 bytes RAM and a mechanism to address internally
- ALU
    - Performs 8 bit arithmetic and logical operations over the operands held by TEMP1 and TEMP 2
    - User cannot access temporary registers

- **SFR REGISTER BANK**
  - set of special function registers
  - address range : 80 H to FF H
  - — Interrupt, serial port and timer units control and perform specific functions under the control of timing and control unit

1.4 Addressing Modes

The five addressing modes are:

- Immediate
- Register
- Direct
- Indirect
- Indexed

Immediate Addressing

If the operand is a constant then it can be stored in memory immediately after the opcode. Remember, values in code memory (ROM) do not change once the system has been programmed and is in use in the everyday world. Therefore, immediate addressing is only of use when the data to be read is a constant.

For example, if your program needed to perform some calculations based on the number of weeks in the year, you could use immediate addressing to load the number 52 (34H) into a register and then perform arithmetic operations upon this data.

```
MOV R0, #34
```

The above instruction is an example of immediate addressing. It moves the data 34H into R0. The assembler must be able to tell the difference between an address anda piece of data. The has symbol (#) is used for this purpose (whenever the assembler sees # before a number it knows this is immediate addressing). This is a two-byte instruction.

Register Addressing

Often we need to move data from a register into the accumulator so that we can perform arithmetic operations upon it. For example, we may wish to move the contents of R5 into the accumulator.

```
MOV A, R5
```

This is an example of register addressing. It moves data from R5 (in the currently selected register bank) into the accumulator.

ADD A, R6

The above is another example of register addressing. It adds the contents of R6 to the accumulator, storing the result in the accumulator. Note that in both examples the destination comes first. This is true of all instructions.

Direct Addressing

Direct addressing is used for accessing data in the on-chip RAM. Since there are 256 bytes of RAM (128 bytes general storage for the programmer and another 128 bytes for the SFRs). That means the addresses go from 00H to FFH, any of which can be stored in an 8-bit location.

MOV A, 67

The above instruction moves the data in location 67H into the accumulator. Note the difference between this and immediate addressing. Immediate addressing uses the data, which is immediately after the instruction. With direct addressing, the operand is an address. The data to be operated upon is stored in that address. The assembler realises this is an address and not data because there is no hash symbol before it.

ADD A, 06

The above instruction adds the contents of location 06H to the accumulator and stores the result in the accumulator. If the selected register bank is bank 0 then this instruction is the same as ADD A, R6.

Indirect Addressing

Register addressing and direct addressing both restrict the programmer to manipulation of data in fixed addresses. The address the instruction reads from (MOV A, 30H) or writes to (MOV 30H, A) cannot be altered while the program is running. There are times when it is necessary to read and write to a number of contiguous memory locations. For example, if you had an array of 8-bit numbers stored in memory, starting at address 30H, you may wish to examine the contents of each number in the array(perhaps to find the smallest number). To do so, you would need to read location 30H, then 31H, then 32H and so on. This can be achieved using indirect addressing. R0 and R1 may be used as pointer registers. We can use either one to store the current memory location and then use the indirect addressing instruction shown below.

MOV A, @Ri

whereRi is either R0 or R1. Now, we can read the contents of location 30H through indirect addressing:

```
MOV        R0,
#30H MOV  A,
@R0
```

The first instruction is an example of immediate addressing whereby the data 30H is placed in R0. The second instruction is indirect addressing. It moves the contents of location 30H into the accumulator.

If we now wish to get the data in location 31H we use the following:

```
  INC     R0
MOV       A,
@R0
```

Once we see how to write a loop in assembly language, we will be able to read theentire contents of the array.

Index Addressing Mode & On-chip ROM Access

- Limitation of register indirect addressing:
- 8-bit addresses (internal RAM)
- DPTR: 16 bits
- MOVC A, @A+DPTR ; "C" means program (code) space ROM

1.5 Special Function Registers (SFRs)

Locations 80H to FFH contain the special function registers. As you can see from the diagram above, not all locations are used by the 8051 (eleven locations are blank). These extra locations are used by other family members (8052, etc.) for the extra features these microcontrollers possess. Also note that not all SFRs are bit- addressable. Those that are have a unique address for each bit. We will deal with each of the SFRs as we progress through the course, but for the moment you should take noteof the accumulator (ACC) at address E0H and the four port registers at addresses 80Hfor P0, 90h for P1, A0 for P2 and B0 for P3. We will later see how easy this makes ready from and

CY	AC	F0	RS1	RS0	OV		P

writing to any of the four ports. **Program Status Word (PSW)**

The PSW is at location D0H and is bit addressable. The table below describes thefunction of each bit

Table 2. Program Status word

Bit	**Symbol**	**Address**	**Description**
PSW.7	CY	D7H	Carry flag
PSW.6	AC	D6H	Auxiliary carry flag
PSW.5	F0	D5H	Flag 0
PSW.4	RS1	D4H	Register bank select 1
PSW.3	RS0	D3H	Register bank select 0
PSW.2	OV	D2H	Overflow flag
PSW.1	--	D1H	Reserved
PSW.0	P	D0H	Even parity flag

Carry Flag The carry flag has two functions.

- Firstly, it is used as the carry-out in 8-bit addition/subtraction. For example, if the accumulator contains FDH and we add 3 to the contents of the accumulator (ADD A, #3), the accumulator will then contain zero and the carry flag will be set. It is also set if a subtraction causes a borrow into bit 7. In other words, if a number is subtracted from another number smaller than it, the carry flag will be set. For example, if A contains 3DH and R3 contains 4BH, the instruction SUBB A, R3 willresult in the carry bit being set (4BH is greater than 3DH).
- The carry flag is also used during Boolean operations. For example, we could AND the contents of bit 3DH with the carry flag, the result being placed in the carry flag - ANL C, 3DH

Register Bank Select Bits Bits 3 and 4 of the PSW are used for selecting the register bank. Since there are four register banks, two bits are required for selecting a bank, as detailed below.

Table3. Register Bank Bits

PSW.4	PSW.3	Register Bank	Address of Register Bank
0	0	0	00H to 07H
0	1	1	08H to 0FH
1	0	2	10H to 17H
1	1	3	18H to 1FH

For example, if we wished to activate register bank 3 we would use the following instructions -

```
SETB    RS1
SETB RS0
```

If we then moved the contents of R4 to the accumulator (MOV A, R4) we would be moving the data from location
1CH to A.

Flag 0

Flag 0 is a general-purpose flag available to the programmer.

Parity Bit

The parity bit is automatically set or cleared every machine cycle to ensure even parity with the accumulator. The number of 1-bits in the accumulator plus the parity bit is always even. In other words, if the number of 1s in the accumulator is odd then the parity bit is set to make the overall number of bits even. If the number of 1s in the accumulator is even then the parity bit is cleared to make the overall number of bits even. For example, if the accumulator holds the number 05H, this is 0000 0101 in binary => the accumulator has an even number of 1s, therefore the parity bit is cleared. If the accumulator holds the number F2H, this is 1111 0010 => the accumulator has an odd number of 1s, therefore the parity bit is set to make the overall number of 1s even. As we shall see later in the course, the parity bit is most often used for detecting errors in transmitted data.

B Register

The B register is used together with the accumulator for multiply and divide operations.

- The MUL AB instruction multiplies the values in A and B and stores the low-byteof the result in A and the high-byte in B.
    - For example, if the accumulator contains F5H and the B register contains 02H, the result of MUL AB will be A = EAH and B = 01H.
- The DIV AB instruction divides A by B leaving the integer result in A and theremainder by B.

- For example, if the accumulator contains 07H and the B register contains 02H, the result of DIV AB will be A = 03H and B = 01H.

The B register is also bit-addressable.

Stack Pointer

The stack pointer (SP) is an 8-bit register at location 81H. A stack is used for temporarily storing data. It operates on the basis of last in first out (LIFO). Putting data onto the stack is called "pushing onto the stack" while taking data off the stack is called "popping the stack." The stack pointer contains the address of the item currently on top of the stack. On power-up or reset the SP is set to 07H. When pushing data onto the stack, the SP is first increased by one and the data is then placed in the location pointed to by the SP. When popping the stack, the data is taken off the stack and the SP is then decreased by one. Since reset initialises the SP to 07H, the first item pushed onto the stack is stored at 08H (remember, the SP is incremented first, then the item is placed on the stack). However, if the programmer wishes to use the register banks 1 to 3, which start at address 08H, he/she must move the stack to another part of memory. The general purpose RAM starting at address 30H is a good spot to place the stack. To do so we need to change the contents of the SP.

```
MOV SP, #2FH.
```

Now, the first item to be pushed onto the stack will be stored at 30H.

Subroutines and the Stack

We have already looked at calling a subroutine in the previous section. Now we will look at the actual call instructions and the effect they have on the stack.

ACALL and LCALL

ACALL stands for absolute call while LCALL stands for long call. These two instructions allow the programmer to call a subroutine. There is a slight difference between the two instructions, the same as the difference between AJMP and LJMP. ACALL allows you to jump to a subroutine within the same 2K page while LCALL allows you to jump to a subroutine anywhere in the 64K code space. The advantage of ACALLover LCALL is that it is a 2-byte instruction while LCALL is a 3-byte instruction.

Help from the Assembler

In the same way that you, the programmer, may use the assembler instruction JMP anytime you need an unconditional jump and the assembler will replace this with the appropriate 8051 jump instruction (SJMP, AJMP or LJMP), you may use the assembler CALL instruction and it will be replaced by the appropriate 8051 subroutine call instruction (ACALL or LCALL - note there is no SCALL instruction).

LCALL Operation

Since the ACALL and LCALL instructions perform almost the same functions we will look at the operation of the LCALL instruction only.

LCALL add16 - long call to subroutine

Encoding - 0001 0010 aaaaaaaaaaaaaaaa (3-byte instruction)

Operation -

```
(PC) <- (PC) + 3
        (SP) <- (SP) + 1
        ((SP)) <- (PC7-PC0)
        (SP) <- (SP) + 1
((SP)) <- (PC15 - PC8)
(PC) <- add15 - add0
```

The operation is as follows. The PC is increased by 3 (because this is a 3-byte instruction). The stack pointer is incremented so that it points to the next empty space on the stack. The third line reads: the contents of the contents of the SP get the low byte of the PC. On system reset the SP is initialised with the value 07H. Therefore, the first item pushed onto the stack will be stored in location 08H. Therefore:

((SP)) is equivalent to (08) - meaning location 08H in memory gets the low bye of the PC - ((SP)) <- (PC7 - PC0)

After the low byte of the PC has been stored on the stack the SP is incremented to pointto the next empty space on the stack (ie; the SP now contains 09H).

SP)) is now equivalent to (09) - meaning location 09H in memory gets the high bye of the PC - ((SP)) <- (PC15 - PC8)

Now that the PC has been stored on the stack the PC is loaded with the 16-bit address (add15 - add0). Subroutines are generally sections of code that will be used many times by the system. A subroutine might be used for taking information from a keyboard or writing data to a serial link. A particular subroutine will be stored at some point in code memory, but it can be called from any location in the program. Therefore, the system needs some way of knowing where to jump back to once execution of the subroutine is complete. The first diagram below shows the contents of the PC and the SP as the instruction LCALL sub (at location 103BH in code memory) is about to be executed. Notice the SP is at its reset value of 07H and the PC contains the address of the next instruction to be executed

RET

A subroutine must end with the *RET* instruction, which simply means *return from subroutine*. Since a subroutine can be called from anywhere in code memory, the *RET* instruction does not specify where to return to. The return address may be different each time the subroutine is called. If you take the flashing LED program from the last section, we called *twoLoopDelay* in the main program after we had turned on the LED. But we also called *twoLoopDelay* from within *threeLoopDelay*. In these two calls the return address is different; in the first call we are returning to the main program once *twoLoopDelay* has completed, but on the second call we are returning to *threeLoopDelay*. The system knows where to return to because, as we have seen above, the return address (ie; the address of the next instruction after the *LCALL*) is stored on the stack. Therefore, the operation of the *RET* instruction is:

RET - return from
subroutine Encoding - 0010
0010 Operation -
(PC15 - PC8) <- ((SP))
(SP) <- (SP) - 1
(PC7 - PC0) <- ((SP))
(SP) <- (SP) - 1

If you look at the diagram above, note the SP contains *09H* - it's pointing at the high-byte of the return address. Therefore, the contents of location *09H* are placed in the high-byte of the PC (PC15 - PC8).

The SP is then decremented (it now contains *08H*) so that it points at the low-byte of the return address. So, the contents of location *08H* are placed in the low-byte of the PC (PC7 - PC0).

In our example above, the PC will now contain *103EH*, and execution takes up immediately after the *LCALL sub* instruction. Also note that the stack is now empty - SP contains 07H.

II INSTRUCTION SET (8051)

2.1 Introduction

The process of writing program for the microcontroller mainly consists of giving instructions (commands) in the specific order in which they should be executed in order to carry out a specific task. As electronics cannot “understand” what for example an instruction “if the push button is pressed- turn the light on” means, then a certain number of simpler and precisely defined orders that decoder can recognise must be used. All commands are known as INSTRUCTION SET.

All microcontrollers compatibile with the 8051 have in total of 255 instructions, i.e. 255 different words available for program writing. At first sight, it is imposing number of odd signs that must be known by heart. However, It is not so complicated as it looks like. Many instructions are considered to be “different”, even though they perform the same operation, so there are only 111 truly different commands.

For example: ADD A,R0, ADD A,R1, ... ADD A,R7 are instructions that perform the same operation (additon of the accumulator and register). Since there are 8 such registers, each instruction is counted separately. Taking into account that all instructions perform only 53 operations (addition, subtraction, copy etc.) and most of them are rarely used in practice, there are actually 20-30 abbreviations to be learned, which is acceptable.

2.2 Types of instructions

Depending on operation they perform, all instructions are divided in several groups:

• Arithmetic Instructions

• Branch Instructions

• Data Transfer Instructions

• Logic Instructions

• Bit-oriented Instructions

The first part of each instruction, called MNEMONIC refers to the operation an instruction performs (copy, addition, logic operation etc.). Mnemonics are abbreviations of the name of operation being executed. For example:

• INC R1 - Means: Increment register R1 (increment register R1);

• JNZ LOOP - Means: Jump if Not Zero LOOP (if the number in the accumulator is not 0, jump to the address marked as LOOP);

The other part of instruction, called OPERAND is separated from mnemonic by at least one whitespace and defines data being processed by instructions. Some of the instructions have no operand, while some of them have one, two or three. If there is more than one operand in an instruction, they are separated by a comma. For example:

• RET - return from a subroutine;

• JZ TEMP - if the number in the accumulator is not 0, jump to the address marked as TEMP;

• ADD A, R3 - add R3 and accumulator;

• CJNE A, #20,LOOP - compare accumulator with 20. If they are not equal, jump to theaddress marked as LOOP;

2.2.1 Arithmetic instructions

Arithmetic instructions perform several basic operations such as addition, subtraction, division, multiplication etc. After execution, the result is stored in the first operand.

For example:

ADD A,R1 - The result of addition (A+R1) will be stored in the accumulator.

ADD A,Rn Adds the register to the accumulator

ADD A,direct Adds the direct byte to the accumulator

ADD A,@Ri Adds the indirect RAM to the accumulator

ADD A,#data Adds the immediate data to the accumulator

ADDC A,Rn Adds the register to the accumulator with a carry flag

ADDC A,direct Adds the direct byte to the accumulator with a carry flag

ADDC A,@Ri Adds the indirect RAM to the accumulator with a carry flag

ADDC A,#data Adds the immediate data to the accumulator with a carry flagSUBB A,Rn Subtracts the register from the accumulator with a borrow

SUBB A,direct Subtracts the direct byte from the accumulator with a borrow

SUBB A,@Ri Subtracts the indirect RAM from the accumulator with a borrow

SUBB A,#data Subtracts the immediate data from the accumulator with a borrow

INC A Increments the accumulator by 1

INC Rn Increments the register by 1

INC Rx Increments the direct byte by 1

INC @Ri Increments the indirect RAM by

1DEC A Decrements the accumulator by 1

DEC Rn Decrements the register by 1

DEC Rx Decrements the direct byte by 1

DEC @Ri Decrements the indirect RAM by

1INC DPTR Increments the Data Pointer by

1 MUL AB Multiplies A and B 1

DIV AB Divides A by B 1

DA A Decimal adjustment of the accumulator according to BCD code

2.2.2 Branch Instructions

There are two kinds of branch instructions:

- Unconditional jump instructions: upon their execution a jump to a new locationfrom where the program continues execution is executed.
- Conditional jump instructions: a jump to a new program location is executed only if a specified condition is met.

Otherwise, the program normally proceeds with the next instruction.

ACALL addr11 Absolute subroutine

call LCALL addr16 Long subroutine

call RET Returns from subroutine

RETI Returns from interrupt subroutineAJMP addr11 Absolute

SJMP rel Short jump (from –128 to +127 locations relative to the following instruction)

JC rel Jump if carry flag is set. Short jump.

JNC rel Jump if carry flag is not set. Short jump.

JB bit,rel Jump if direct bit is set. Short jump.

JBC bit,rel Jump if direct bit is set and clears bit. Short jump.

JMP @A+DPTR Jump indirect relative to the DPTR

JZ rel Jump if the accumulator is zero. Short jump.

JNZ rel Jump if the accumulator is not zero. Short jump.

CJNE A,direct,rel Compares direct byte to the accumulator and jumps if notequal. Short jump.

CJNE A,#data,rel Compares immediate data to the accumulator and jumps if notequal. Short jump.

CJNE Rn,#data,rel Compares immediate data to the register and jumps if notequal. Short jump.

CJNE @Ri,#data,rel Compares immediate data to indirect register and jumps ifnot equal. Short jump.

DJNZ Rn,rel Decrements register and jumps if not 0. Short jump.

DJNZ Rx,rel Decrements direct byte and jump if not 0. Short jump.

NOP No operation

2.2.3 Data Transfer Instructions

Data transfer instructions move the content of one register to another. The register the content of which is moved remains unchanged. If they have the suffix "X" (MOVX), the data is exchanged with external memory.

MOV A,Rn Moves the register to the accumulator MOV A,direct Moves the direct byte to the accumulator

MOV A,@Ri Moves the indirect RAM to the accumulator

MOV A,#data Moves the immediate data to the accumulator

MOV Rn,A Moves the accumulator to the register

MOV Rn,direct Moves the direct byte to the register

MOV Rn,#data Moves the immediate data to the register

MOV direct,A Moves the accumulator to the direct byte

MOV direct,Rn Moves the register to the direct byte

MOV direct,direct Moves the direct byte to the direct
byte

MOV direct,@Ri Moves the indirect RAM to the direct byte

MOV direct,#data Moves the immediate data to the direct byte

MOV @Ri,A Moves the accumulator to the indirect RAM

MOV @Ri,direct Moves the direct byte to the indirect RAM

MOV @Ri,#data Moves the immediate data to the indirect
RAMMOV DPTR,#data Moves a 16-bit data to the data pointer

MOVC A,@A+DPTR Moves the code byte relative to the DPTR to the accumulator(address=A+DPTR)

MOVC A,@A+PC Moves the code byte relative to the PC to the accumulator (address=A+PC)

MOVX A,@Ri Moves the external RAM (8-bit address) to the accumulator

MOVX A,@DPTR Moves the external RAM (16-bit address) to the accumulatorMOVX @Ri,A Moves the accumulator to the external RAM (8-bit address) MOVX @DPTR,A Moves the accumulator to the external RAM (16-bit address)PUSH direct Pushes the direct byte onto the stack

POP direct Pops the direct byte from the stack

XCH A,Rn Exchanges the register with the accumulator

XCH A,direct Exchanges the direct byte with the accumulator

XCH A,@Ri Exchanges the indirect RAM with the
accumulator

XCHD A,@Ri Exchanges the low-order nibble indirect RAM with the accumulator

2.2.4 Logic Instructions

Logic instructions perform logic operations upon corresponding bits of two registers. After execution, the result is stored
in the first operand.

ANL A,Rn AND register to accumulator

ANL A,direct AND direct byte to accumulator ANL A,@Ri AND indirect RAM to accumulator

ANL A,#data AND immediate data to accumulatorANL direct,A AND accumulator to direct byte

ANL direct,#data AND immediae data to direct register ORL A,Rn OR register to accumulator

ORL A,direct OR direct byte to accumulator ORL A,@Ri OR indirect RAM to accumulatorORL direct,A OR accumulator to direct byte

ORL direct,#data OR immediate data to direct byte

XRL A,Rn Exclusive OR register to accumulator

XRL A,direct Exclusive OR direct byte to accumulator

XRL A,@Ri Exclusive OR indirect RAM to accumulator

XRL A,#data Exclusive OR immediate data to accumulator XRL direct,A Exclusive OR accumulator to direct byte

XORL direct,#data Exclusive OR immediate data to direct byte

CLR A Clears the accumulator

CPL A Complements the accumulator (1=0, 0=1)

SWAP A Swaps nibbles within the accumulator

RL A Rotates bits in the accumulator left

RLC A Rotates bits in the accumulator left through carry

RR A Rotates bits in the accumulator right

RRC A Rotates bits in the accumulator right through carry

2.2.5 Bit-oriented Instructions

Similar to logic instructions, bit-oriented instructions perform logic operations. The difference is that these are performed
upon single bits.

CLR C Clears the carry flag

CLR bit Clears the direct bit SETB C Sets the carry flag SETB bit Sets the direct bit

CPL C Complements the carry flag

CPL bit Complements the direct bit

ANL C,bit AND direct bit to the carry flag

ANL C,/bit AND complements of direct bit to the carry flagORL C,bit OR direct bit to the carry flag
ORL C,/bit OR complements of direct bit to the carry flagMOV C,bit Moves the direct bit to the carry flag
MOV bit,C Moves the carry flag to the direct bit

Description of all 8051 instructions

Here is a list of the operands and their meanings:

A - accumulator;

Rn - is one of working registers (R0-R7) in the currently active RAM memory bank;

Direct - is any 8-bit address register of RAM. It can be any general-purpose register or a SFR (I/O port, control register etc.);

@Ri - is indirect internal or external RAM location addressed by register R0 or R1;

#data - is an 8-bit constant included in instruction (0-255);

#data16 - is a 16-bit constant included as bytes 2 and 3 in instruction (0-65535);

addr16 - is a 16-bit address. May be anywhere within 64KB of program memory;

addr11 - is an 11-bit address. May be within the same 2KB page of program memory as the first byte of the following instruction;

rel - is the address of a close memory location (from -128 to +127 relative to the first byte of the following instruction). On the basis of it, assembler computes the value to add or subtract from the number currently stored in the program counter;

bit - is any bit-addressable I/O pin, control or status bit; and

C - is carry flag of the status register (register PSW).

8051 Addressing Modes

8051 has four addressing modes.

1. Immediate Addressing :

Data is immediately available in the instruction.

For example -

ADD A, #77; Adds 77 (decimal) to A and stores in A

ADD A, #4DH; Adds 4D (hexadecimal) to A and stores in A

MOV DPTR, #1000H; Moves 1000 (hexadecimal) to data

pointer

2. Bank Addressing or Register Addressing :
This way of addressing accesses the bytes in the current register bank. Data is available in the register specified in the instruction. The register bank is decided by 2 bits of Processor Status Word (PSW).
For example-

ADD A, R0; Adds content of R0 to A and stores in A

3.. Direct Addressing :

The address of the data is available in the instruction. For example -

MOV A, 088H; Moves content of SFR TCON (address 088H)to A

4. Register Indirect Addressing :

The address of data is available in the R0 or R1 registers as specified in the instruction. For example -

MOV A, @R0 moves content of address pointed by R0 to A

External Data Addressing :

Pointer used for external data addressing can be either R0/R1 (256 byte access) or DPTR (64kbyte access).

For example -

MOVX A, @R0; Moves content of 8-bit address pointed by R0 to A

MOVX A, @DPTR; Moves content of 16-bit address pointed by DPTR to A

External Code Addressing :

Sometimes we may want to store non-volatile data into the ROM e.g. look-up tables.

Such data may require reading the code memory. This may be done as follows -

MOVC A, @A+DPTR; Moves content of address pointed by A+DPTR to A

MOVC A, @A+PC; Moves content of address pointed by A+PC to A

I/O Port Configuration

Each port of 8051 has bidirectional capability. Port 0 is called 'true bidirectional port' as it floats (tristated) when configured as input. Port-1, 2, 3 are called 'quasi bidirectional port'. Port-0 Pin Structure

Port -0 has 8 pins (P0.0-P0.7).

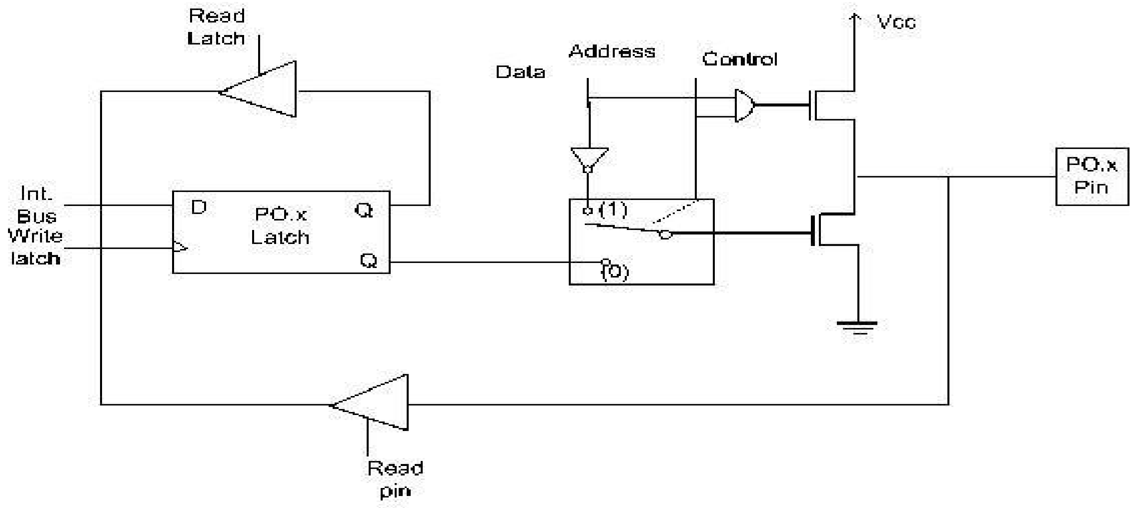


Figure 2 Port-0 Structure

Port-0 can be configured as a normal bidirectional I/O port or it can be used for address/data interfacing for accessing external memory. When control is '1', the port is used for address/data interfacing. When the control is '0', the port can be used as a normal bidirectional I/O port.

Let us assume that control is '0'. When the port is used as an input port, '1' is written to the latch. In this situation both the output MOSFETs are 'off'. Hence the output pin floats. This high impedance pin can be pulled up or low by an external source. When the port is used as an output port, a '1' written to the latch again turns 'off' both the output MOSFETs and causes the output pin to float. An external pull-up is required to output a '1'. But when '0' is written to the latch, the pin is pulled down by the lower MOSFET. Hence the output becomes zero.

When the control is '1', address/data bus controls the output driver MOSFETs. If the address/data bus (internal) is '0', the upper MOSFET is 'off' and the lower MOSFET is 'on'. The output becomes '0'. If the address/data bus is '1', the upper transistor is 'on' and the lower transistor is 'off'. Hence the output is '1'. Hence for normal address/data interfacing (for external memory access) no pull-up resistors are required.

Port-0 latch is written to with 1's when used for external memory
access. Port-1 Pin Structure

Port-1 has 8 pins (P1.1-P1.7) .

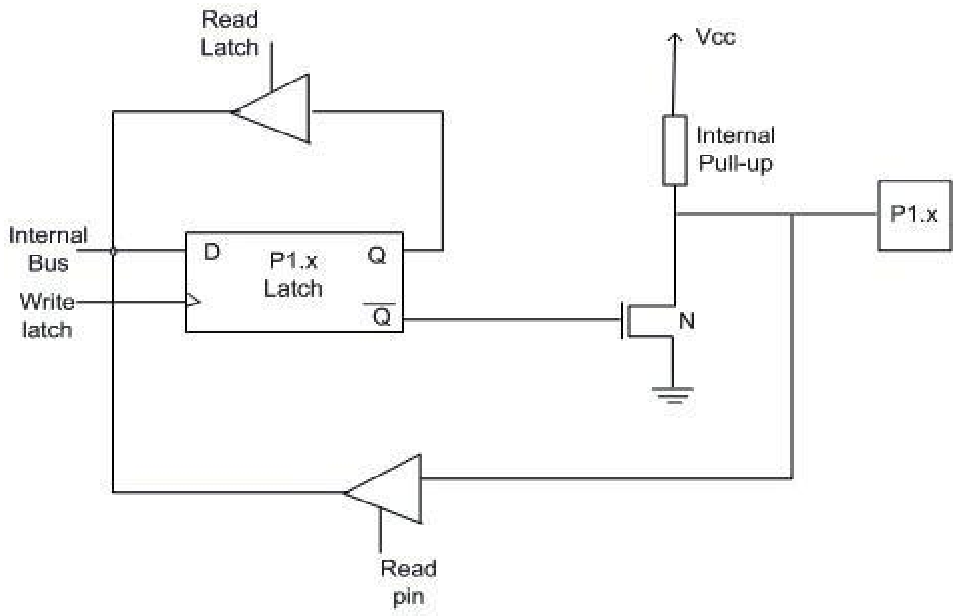
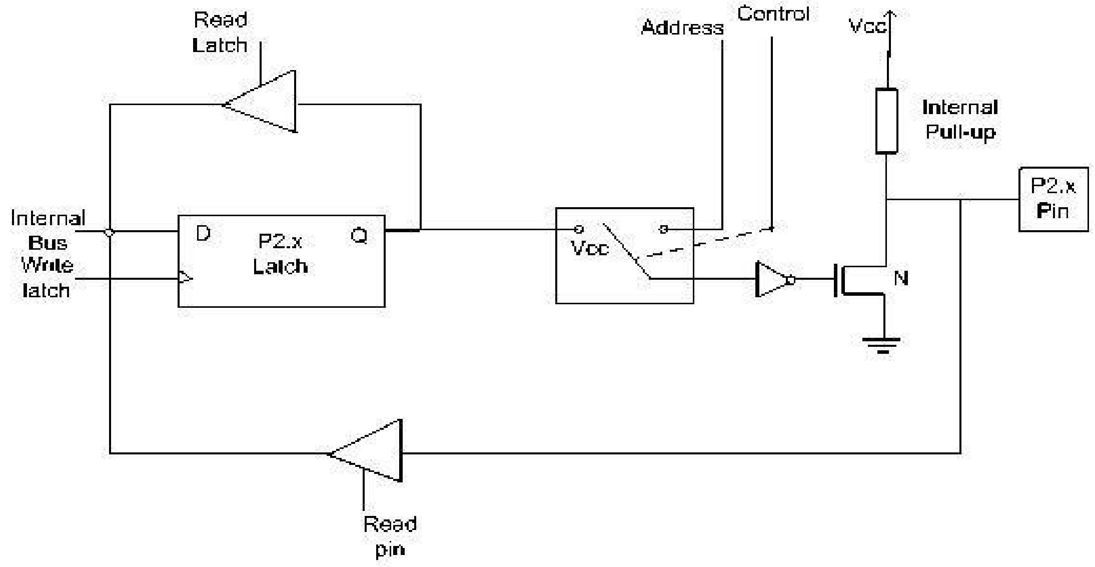


Figure 3 Port 1 Structure

Port-1 does not have any alternate function i.e. it is dedicated solely for I/O interfacing. When used as output port, the pin is pulled up or down through internal pull-up. To use port-1 as input port, '1' has to be written to the latch. In this input mode when '1' is written to the pin by the external device then it read fine. But when '0' is written to the pin by the external device then the external source must sink current due to internal pull-up. If the external device is not able to sink the current the pin voltage may rise, leading to a possible wrong reading.

PORT 2 Pin Structure



Port-2 has 8-pins (P2.0-P2.7) .

Fig 4 Port 2 Structure

Port-2 is used for higher external address byte or a normal input/output port. The I/O operation is similar to Port-1. Port-2 latch remains stable when Port-2 pin are used for external memory access. Here again due to internal pull-up there is limited current driving capability.

PORT 3 Pin Structure

Port-3 has 8 pin (P3.0-P3.7) . Port-3 pins have alternate functions.

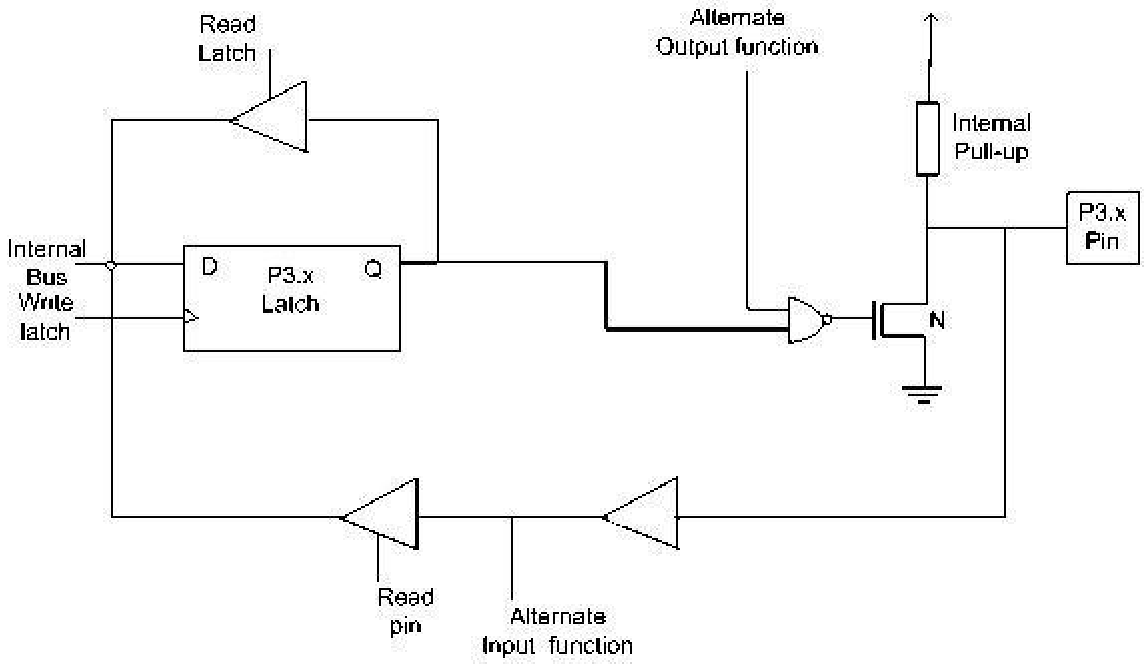


Figure 5 Port 3 Structure

Each pin of Port-3 can be individually programmed for I/O operation or for alternatefunction. The alternate function can be activated only if the corresponding latch hasbeen

written to '1'. To use the port as input port, '1' should be written to the latch. This port also has internal pull-up and limited current driving capability.
Alternate functions of Port-3 pins are -

Table 3 Port function

P3.0	RxD
P3.1	TxD
P3.2	$\overline{INT0}$
P3.3	$\overline{INT1}$
P3.4	T0
P3.5	T1
P3.6	$\overline{WR}$
P3.7	$\overline{RD}$

Note:

1) Port 1, 2, 3 each can drive 4 LS TTL inputs.
2) Port-0 can drive 8 LS TTL inputs in address /data mode. For digital output port, it needs external pull-up resistors.
3) Ports-1,2and 3 pins can also be driven by open-collector or open-drain outputs.
4) Each Port 3 bit can be configured either as a normal I/O or as a special function bit.

8051 MICROCONTROLLER PROGRAMS

1.8 BIT ADDITION USING INTERNAL MEMORY

Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	E5,40		MOV	A,40	Move the content of 40 to accumulator
8002	A8,41		MOV	R0,41	Move the content of 41 to 'R0' register
8004	28		ADD	A,R0	Add the content of 'R0' and 'A'
8005	F5,42		MOV	42,A	Move the content of accumulator to 42
8007	74,00		MOV	A,#00	Initialize the accumulator
8009	34,00		ADDC	A,#00	Add the content of A and 00 with carry
800B	F5,43		MOV	43,A	Move the content of accumulator to 43
800D	12,00,BB		LCALL	00BB	Halt the program

2. 8 BIT ADDITION USING EXTERNAL MEMORY

Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	90,91,00		MOV	DPTR,#9100	Initialize the data pointer
8003	E0		MOVX	A,@DPTR	Move the content of DPTR to Acc.

8004	F8		MOV	R0,A	Move the content of A to R0
8005	A3		INC	DPTR	Increment the data pointer
8006	E0		MOVX	A,@DPTR	Move the content of DPTR to Acc.
8007	28		ADD	A,R0	Add the content of ‘R0’ and ‘A’
8008	A3		INC	DPTR	Increment the data pointer
8009	F0		MOVX	@DPTR,A	Move the content of A to DPTR
800A	74,00		MOV	A,#00	Initialize the accumulator
800C	34,00		ADDC	A,#00	Add the content of A and 00 with carry
800E	A3		INC	DPTR	Increment the data pointer
800F	F0		MOVX	@DPTR,A	Move the content of A to DPTR
8010	12,00,BB		LCALL	00BB	Halt the program

3. 8 BIT SUBTRACTION USING INTERNAL MEMORY

Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	C3		CLR	C	Clear the Carry flag
8001	E5,40		MOV	A,40	Move the content of 40 to accumulator
8003	A8,41		MOV	R0,41	Move the content of 41 to ‘R0’ register
8005	98		SUBB	A,R0	Subtract the content of ‘R0’ from ‘A’
8006	F5,42		MOV	42,A	Move the content of accumulator to 42
8008	12,00,BB		LCALL	00BB	Halt the program

4. 8 BIT SUBTRACTION USING EXTERNAL MEMORY

Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	C3		CLR	C	Clear the Carry flag
8001	90,91,00		MOV	DPTR,#9100	Initialize the data pointer
8004	E0		MOVX	A,@DPTR	Move the content of DPTR to Acc.
8005	F8		MOV	R0,A	Move the content of A to R0
8006	A3		INC	DPTR	Increment the data pointer
8007	E0		MOVX	A,@DPTR	Move the content of DPTR to Acc.
8008	98		SUBB	A,R0	Subtract the content of ‘R0’ from ‘A’
8009	A3		INC	DPTR	Increment the data pointer
800A	F0		MOVX	@DPTR,A	Move the content of A to DPTR
801B	12,00,BB		LCALL	00BB	Halt the program

5. 8 BIT MULTIPLICATION USING INTERNAL MEMORY

Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	E5,40		MOV	A,40	Move the content of 40 to accumulator
8002	85,41,F0		MOV	0F0,41	Move the content of 41 to ‘B’ register
8005	A4		MUL	AB	Multiply the content of ‘A’ and ‘B’
8006	F5,42		MOV	42,A	Move the content of accumulator to 42
8008	E5,F0		MOV	A,0F0	Move the content of ‘B’ to accumulator
800A	F5,43		MOV	43,A	Move the content of accumulator to 43
800C	12,00,BB		LCALL	00BB	Halt the program

6. 8 BIT MULTIPLICATION USING EXTERNAL MEMORY

Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	90,91,00		MOV	DPTR,#9100	Initialize the data pointer
8003	E0		MOVX	A,@DPTR	Move the content of DPTR to Acc.
8004	F5,F0		MOV	0F0,A	Move the content of ‘A’ to ‘B’ register
8006	A3		INC	DPTR	Increment the data pointer
8007	E0		MOVX	A,@DPTR	Move the content of DPTR to Acc.
8008	A4		MUL	AB	Multiply the content of ‘A’ and ‘B’
8009	A3		INC	DPTR	Increment the data pointer
800A	F0		MOVX	@DPTR,A	Move the content of ‘A’ to DPTR
800B	E5,F0		MOV	A,0F0	Move the content of ‘B’ to accumulator
800D	A3		INC	DPTR	Increment the data pointer
800E	F0		MOVX	@DPTR,A	Move the content of A to DPTR
800F	12,00,BB		LCALL	00BB	Halt the program

7. 8 BIT DIVISION USING INTERNAL MEMORY

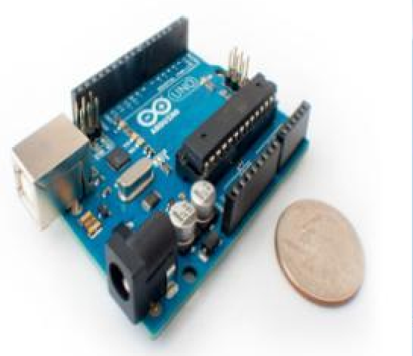
Memory Address	Hex code	Label	Mnemonics		Comments
			Opcode	Operand	
8000	E5,40		MOV	A,40	Move the content of 40 to accumulator
8002	85,41,F0		MOV	0F0,41	Move the content of 41 to ‘B’ register
8005	84		DIV	AB	Divide the content of ‘A’ and ‘B’
8006	F5,42		MOV	42,A	Move the content of accumulator to 42
8008	E5,F0		MOV	A,0F0	Move the content of ‘B’ to accumulator
800A	F5,43		MOV	43,A	Move the content of accumulator to 43
800C	12,00,BB		LCALL	00BB	Halt the program

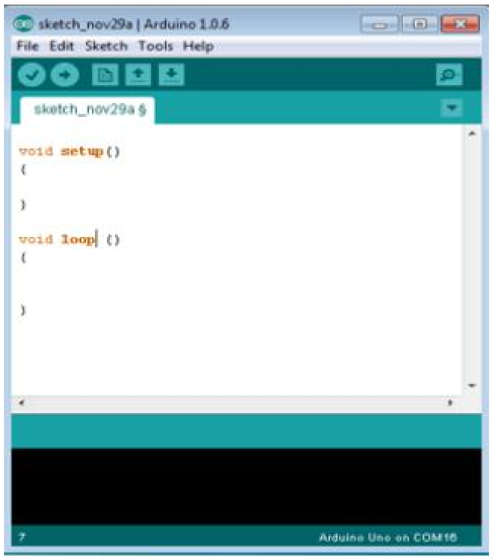
Introduction to Arduino

Arduino is a prototype platform (open-source) based on an easy-to-use hardware and software. It consists of a circuit board, which can be programed (referred to as a microcontroller) and a ready-made software called Arduino IDE (Integrated Development Environment), which is used to write and upload the computer code to the physical board.

The key features are:

- Arduino boards are able to read analog or digital input signals from different sensors and turn it into an output such as activating a motor, turning LED on/off, connect to the cloud and many other actions.
- You can control your board functions by sending a set of instructions to the microcontroller on the board via Arduino IDE (referred to as uploading software).
- Unlike most previous programmable circuit boards, Arduino does not need an extra piece of hardware (called a programmer) in order to load a new code onto the board. You can simply use a USB cable.
- Additionally, the Arduino IDE uses a simplified version of C++, making it easier to learn to program.
- Finally, Arduino provides a standard form factor that breaks the functions of the microcontroller into a more accessible package





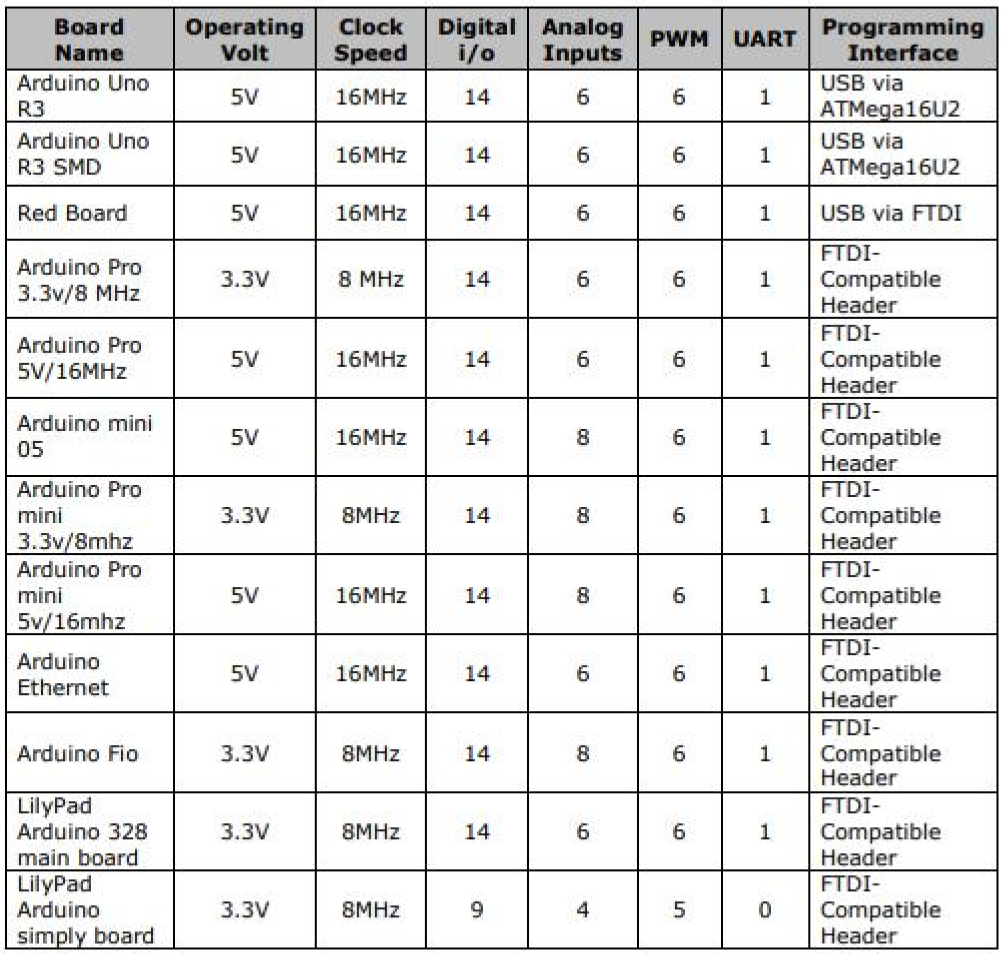
```
void setup()
{

}

void loop ()
{


}
```

Board Types

Various kinds of Arduino boards are available depending on different microcontrollers used. However, all Arduino boards have one thing in common: they are programed through

the Arduino IDE. The differences are based on the number of inputs and outputs (the number of sensors, LEDs, and buttons you can use on a single board), speed, operating voltage, form factor etc. Some boards are designed to be embedded and have no programming interface (hardware), which you would need to buy separately. Some can run directly from a 3.7V battery, others need at least 5V.
Here is a list of different Arduino boards available.

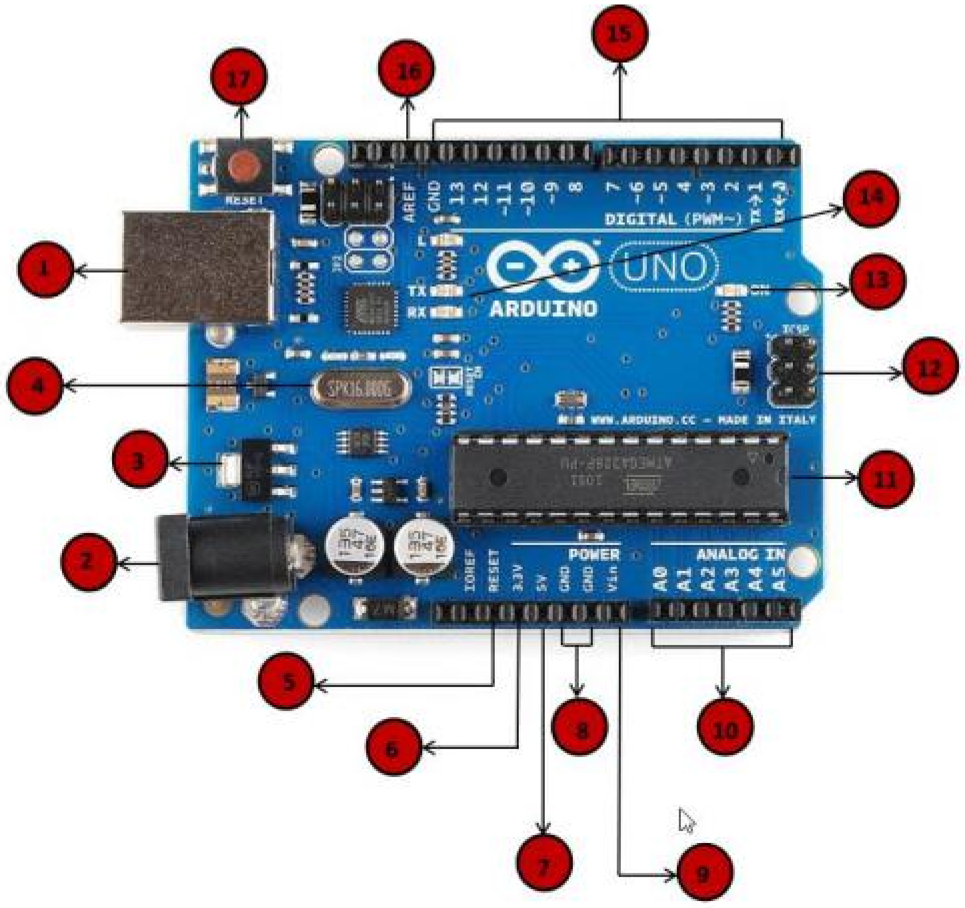
Arduino boards based on ATMEGA328 microcontroller



Board Name	Operating Volt	Clock Speed	Digital i/o	Analog Inputs	PWM	UART	Programming Interface
Arduino Uno R3	5V	16MHz	14	6	6	1	USB via ATMega16U2
Arduino Uno R3 SMD	5V	16MHz	14	6	6	1	USB via ATMega16U2
Red Board	5V	16MHz	14	6	6	1	USB via FTDI
Arduino Pro 3.3v/8 MHz	3.3V	8 MHz	14	6	6	1	FTDI-Compatible Header
Arduino Pro 5V/16MHz	5V	16MHz	14	6	6	1	FTDI-Compatible Header
Arduino mini 05	5V	16MHz	14	8	6	1	FTDI-Compatible Header
Arduino Pro mini 3.3v/8mhz	3.3V	8MHz	14	8	6	1	FTDI-Compatible Header
Arduino Pro mini 5v/16mhz	5V	16MHz	14	8	6	1	FTDI-Compatible Header
Arduino Ethernet	5V	16MHz	14	6	6	1	FTDI-Compatible Header
Arduino Fio	3.3V	8MHz	14	8	6	1	FTDI-Compatible Header
LilyPad Arduino 328 main board	3.3V	8MHz	14	6	6	1	FTDI-Compatible Header
LilyPad Arduino simply board	3.3V	8MHz	9	4	5	0	FTDI-Compatible Header

ARDUINO – BOARD DESCRIPTION

In this chapter, we will learn about the different components on the Arduino board. We will study the Arduino UNO board because it is the most popular board in the Arduino board family. In addition, it is the best board to get started with electronics and coding. Some boards look a bit different from the one given below, but most Arduinos have majority of these components in common.



- Power USB

  Arduino board can be powered by using the USB cable from your computer. All you need to do is connect the USB cable to the USB connection (1).
- Power (Barrel Jack)

  Arduino boards can be powered directly from the AC mains power supply by connecting it to the Barrel Jack (2).
- Voltage Regulator

  The function of the voltage regulator is to control the voltage given to the Arduino board and stabilize the DC voltages used by the processor and other elements.
- Crystal Oscillator

The crystal oscillator helps Arduino in dealing with time issues. How does Arduino calculate time? The answer is, by using the crystal oscillator. The number printed on top of the Arduino crystal is 16.000H9H. It tells us that the frequency is 16,000,000 Hertz or 16 MHz.

- Arduino Reset

  You can reset your Arduino board, i.e., start your program from the beginning. You can reset the UNO board in two ways. First, by using the reset button (17) on the board. Second, you can connect an external reset button to the Arduino pin labelled RESET (5).

- Pins (3.3, 5, GND, Vin)
- 3.3V (6): Supply 3.3 output volt
- 5V (7): Supply 5 output volt
- Most of the components used with Arduino board works fine with 3.3 volt and 5 volt.
- GND (8)(Ground): There are several GND pins on the Arduino, any of which can be used to ground your circuit.
- Vin (9): This pin also can be used to power the Arduino board from an external power source, like AC mains power supply

- Analog pins

  The Arduino UNO board has five analog input pins A0 through A5. These pins can read the signal from an analog sensor like the humidity sensor or temperature sensor and convert it into a digital value that can be read by the microprocessor.
- Main microcontroller

  Each Arduino board has its own microcontroller (11). You can assume it as the brain of your board. The main IC (integrated circuit) on the Arduino is slightly different from board to board. The microcontrollers are usually of the ATMEL Company. You must know what IC your board has before loading up a new program from the Arduino IDE. This information is available on the top of the IC. For more details about the IC construction and functions, you can refer to the data sheet.
- ICSP pin

  Mostly, ICSP (12) is an AVR, a tiny programming header for the Arduino consisting of MOSI, MISO, SCK, RESET, VCC, and GND. It is often referred to as an SPI (Serial Peripheral Interface), which could be considered as an "expansion" of the output. Actually, you are slaving the output device to the master of the SPI bus.
- Power LED indicator

  This LED should light up when you plug your Arduino into a power source to indicate that your board is powered up correctly. If this light does not turn on, then there is something wrong with the connection.

- TX and RX LEDs

  On your board, you will find two labels: TX (transmit) and RX (receive). They appear in two places on the Arduino UNO board. First, at the digital pins 0 and 1, to indicate the pins responsible for serial communication. Second, the TX and RX led (13). The TX led flashes with different speed while sending the serial data. The speed of flashing depends on the baud rate used by the board. RX flashes during the receiving process.
- Digital I / O

  The Arduino UNO board has 14 digital I/O pins (15) (of which 6 provide PWM (Pulse Width Modulation) output. These pins can be configured to work as input digital pins to read logic 14 values (0 or 1) or as digital output pins to drive different modules like LEDs, relays, etc. The pins labeled "~" can be used to generate PWM.
- AREF

  AREF stands for Analog Reference. It is sometimes, used to set an external reference voltage (between 0 and 5 Volts) as the upper limit for the analog input pins.

TEXT / REFERENCE BOOKS

1. Ramesh Goankar, "Microprocessor architecture programming and applications with 8085 / 8088", 5th Edition, Penram
International Publishing.
2. A.K.Ray and Bhurchandi, "Advanced Microprocessor", 1st Edition, TMH Publication.
3. Kenneth J.Ayala, "The 8051 microcontroller Architecture, Programming and applications" 2nd Edition ,Penram
international.
4. Doughlas V.Hall, "Microprocessors and Digital system", 2nd Editon, Mc Graw Hill,1983.
5. Md.Rafiquzzaman, "Microprocessors and Microcomputer based system design", 2nd Editon,Universal Book Stall, 1992.
6. Hardware Reference Manual for 80X86 family", Intel Corporation, 1990.
7. Muhammad Ali Mazidi and Janice Gillispie Mazidi, "The 8051 Microcontroller and Embedded Systems", 2nd Edition,
Pearson.
8. "Arduino Made Simple" by Ashwin Pajankar

Question Bank

1. Explain the architecture of 8051 microcontroller in detail with the help of neat diagram.
2. Gives notes on (a) Instruction format (b) Signed and Unsigned conditional branch instruction.
3. Define addressing modes. With suitable examples explain 8051 addressing modes in detail.
4. Write a detailed note on assembler dependent instruction and programming