Introduction to Microcontrollers

Dinesh Sharma

Department Of Electrical Engineering
Indian Institute Of Technology, Bombay

Sept. 22, 2010

Need For a programmable device

- Design of complex integrated circuits has a high fixed cost component.
- If sold in large quantities, the cost of an integrated circuit can be quite low.
- It makes sense to design a programmable device which would perform different functions in different products.
- Now this device can be used for a variety of applications and will thus be required in large quantities.
- The function performed by the device will be selected by a digital input. This is called an **instruction**.
- The device is given a series of instructions to perform a function. This specific sequence of instructions is a **program**.

The Processor

- The programmable digital device which can run a program is called a processor.
- Processors can be large and complex. However, a (relatively) simple processor which is implemented on a single chip using VLSI technology is called a **microprocessor**.
- The term is applied these days even to complex processors – a pentium has upwards of 100 million transistors!
- The basic logic in a processor just performs the following loop endlessly:
  - Fetch next instruction
  - Decode it
  - Execute the instruction

Instruction Fetching

- Each instruction is encoded as a unique combination of digital bits.
- We need a device to store all the instructions to be executed.
- This storage device is called a **memory**.
- The memory device stores different instructions at different locations.
- Each location is also encoded as a combination of digitatl bits. This is known as the **address**.
- The way an instruction is fetched is that the processor outputs an address and the memory returns the contents stored at this address. These contents are interpreted as the instruction.

Fetch Next Instruction

How do we fetch the **Next** instruction?

- The processor contains storage elements too. These are called **registers**
- One particular register contains the address of the instruction to be executed.
- This is called the **Instruction Pointer**.
- Instruction pointer is automatically incremented every time it is used.
- Therefore, on next use, we shall fetch the next instruction, not the same one.
- For this reason, it also known as the **Program Counter**.

Registers in the processor

- Just like the instruction pointer, the processor may contain other registers which are used for specific purposes.
- In addition, it may have general purpose registers which are used for storing data.
- Registers which are specifically used to store addresses are called pointers. For example, the Instruction pointer and the stack pointer.
- Registers whose contents can be manipulated through instructions are placed in a **register file** and are often identified by indices allotted to them.

Instruction Decoding

- The processor has different execution units – such as an adder, a subtractor, a circuit to AND two quantities etc.
- The decoder contains special digital circuitry which will look for some predefined bit combinations.
- Depending on which bit combination is found, a particular execution unit is activated.
- For example, if the top two bits are 0 and 1 respectively, an 8085 processor will interpret it as a MOV instructions, which copies data from one register to another.
- The remaining bits of the instruction are used to determine which is the source register and which is the destination.

The Execute operation

- The decode operation activates a selected circuit on the processor.
- The data to be fed to this execution is knwon as *operand(s)*.
- The operands may be the contents of data registers on the processor chip or contents of some memory location.
- If the operand is in the (external) memory, it needs to be fetched first, just like the instruction.
- After the operation has been performed, the result stored in a register or in memory.
- Again, if the destination of the result (as specified by the instruction) is in memory, a memory **write** cycle has to be carried out.

Changing the program flow

A programme need not be confined to executing sequential instruction.

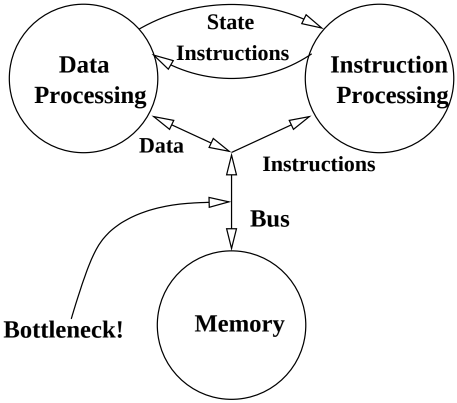
- An instruction can cause a new value to be loaded in the Instruction pointer. As a result, when the next instruction is fetched, it will be from this new address and not from the auto-incremented value.
- Modification of the instruction pointer can be made conditional.
- This way one can implement various “if” and “goto” kind of constructs.
- These can be combined by the program to construct loops like “while” and “for”.

What if we want an external event, rather than the program, to change the program flow?

Interrupts

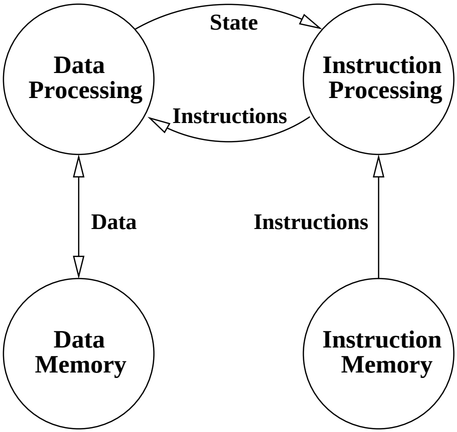
- If we want an external event to modify the sequence of operations being carried out under program control, We have to enable this in hadware.
- This is called “interrupting” the program.
- When a signal arrives at an interrupt pin, and this facility is currently enabled, the processor completes the currently executing instruction, then suspends the regular program flow.
- It stores the address from where the next instruction *would have been fetched*.
- It now starts executing a different program, called the interrupt service routine.
- When the interrupt service routine gets over, execution of the interrupted program re-starts from the saved value of the instruction pointer.

Von Neumann Architecture



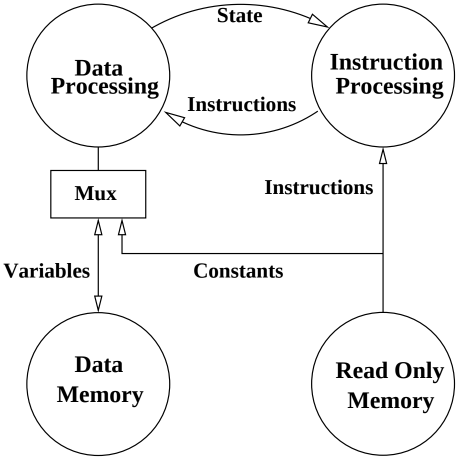
- A common bus is used for data as well as instructions.
- The system can become ‘bus bound’.

Harvard Architecture



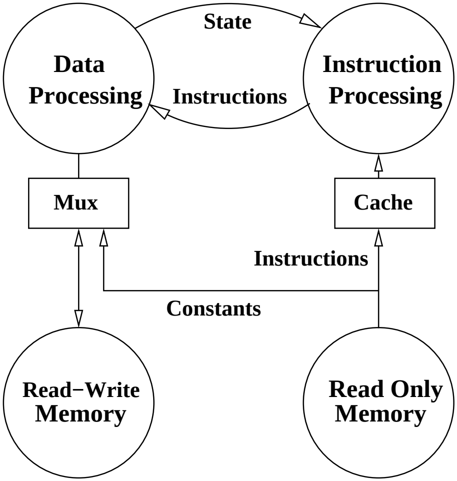
- Separate data and instruction paths
- Good performance
- Needs 2 buses $\rightarrow$ expensive!
- Traffic on the buses is not balanced.
- Instruction bus may remain idle.

Modified Harvard Architecture



- Constants can be stored with Instructions in ROM.
- Better Bus balancing is possible.
- Typically, 1 instruction read, 1 constant read, 1 data read and 1 result write per instruction.
- 2 mem ops per bus.

Modified Harvard with Cache



- Cache allows optimum utilization of bus bandwidths.
- Each operation need not be balanced individually.

8051 Architecture

- Most microprocessor systems use
  - a processor
  - ROM
  - RAM
  - Port I-O chips (like 8255A)
  - Counter/Timers (like 8253/8254)
  - Serial I/O
- Microcontrollers like 8051 typically put all these functions in the same chip.
- The 8051 makes use of the fact that Harvard architecture can be used internally without incurring the cost of an additional external bus, Since ROM and RAM are inside the chip.

The 8051 family

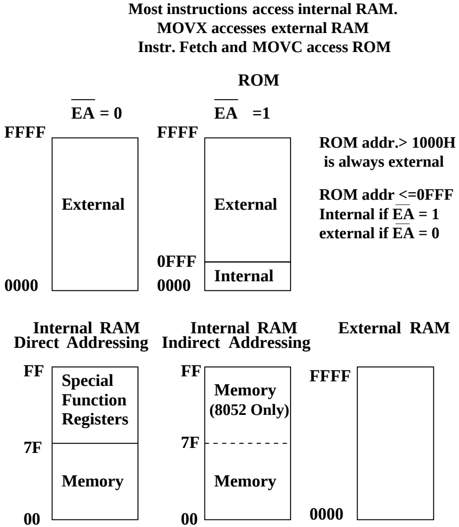
- Even though an 8051 uses internal ROM and RAM, an external bus is desirable in order to add additional ROM, RAM or peripheral chips.
- Most microcontrollers are used in systems which serve a fixed application: such as washing machines or cameras. Resources not used by this particular application would then be wasted.
- Most microcontrollers are therefore produced as a family of chips with different combinations of internal resources.
- One picks the chip from the family which best matches ones application.
- The standard 8051 comes with 4K ROM, 128 bytes RAM, 4 8bit ports and 2 timer counters.
- Other members of the family are the ROMless 8031, 8052 with 256 bytes of RAM and 3 counter/timers etc.

8051 Compatible Chips

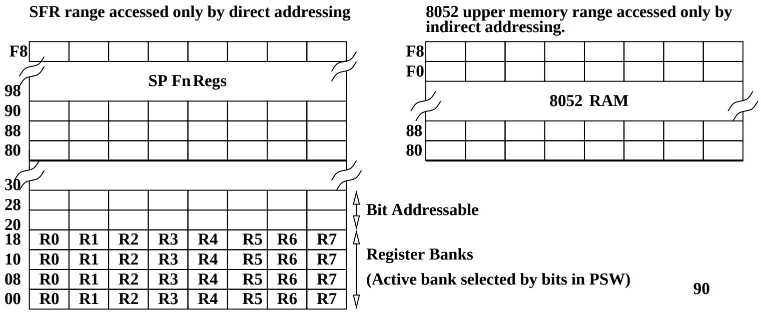
- Since the 8051 family is widely used, other manufacturers such as Atmel, Infinion, Philips and Western Digital have introduced variants which provide additional utilities.
- The Atmel 89C51 series replaces ROM by flash memory to permit re-programming. This is extremely convenient during development and for field upgradation.
- Microcontrollers are quite inexpensive. Retail prices of 89C52 range between Rs. 60/ to Rs. 100/- in single IC quantity.
- All that is required is to add a crystal, a few capacitors and a power supply to add microprocessor control to any system.

8051 Memory

- Because of its Harvard architecture, 8051 has independent address spaces for ROM and RAM.
- The instruction encodes which address space is being used.
- For example, MOV uses internal RAM, MOVC uses ROM and MOVX uses external RAM.
- External and Internal ROM are selected according to the address and the level at the $\overline{\text{EA}}$ pin.


Most instructions access internal RAM.
MOVX accesses external RAM
Instr. Fetch and MOVC access ROM
ROM
EA = 0
EA =1
FFFF
External
0000
FFFF
External
0FFF
Internal
0000
ROM addr.> 1000H is always external
ROM addr <=0FFF
Internal if EA = 1
external if EA = 0
Internal RAM Direct Addressing
Internal RAM Indirect Addressing
External RAM
FF
Special Function Registers
7F
Memory
00
FF
Memory (8052 Only)
7F
Memory
00
FFFF
0000

internal RAM organization


SFR range accessed only by direct addressing
F8
SP Fn Regs
98
90
88
80
30
28
20
18
10
08
00
R0
R1
R2
R3
R4
R5
R6
R7
Bit Addressable
Register Banks
(Active bank selected by bits in PSW)
8052 upper memory range accessed only by indirect addressing.
F8
F0
8052 RAM
88
80

Special Function Register Area

F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8									CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2								A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH				PCON	87
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

Registers

If RAM is internal, there is very little difference between an internal RAM location and a “register”. For notational convenience, certain internal RAM addresses are referred to as registers. Many instructions use these registers as operands implicitly or explicitly. Following registers are located in the “Special Function Register” area of the the internal RAM (addresses > 7F).

- The accumulator or A resides at address E0H. Register B used implicitly by multiply and divide instructions resides at address F0H.
- The Program Status Word or PSW resides at address D0H. It contains flags etc.
- Pointer registers DPH and DPL reside at locations 83H and 82H respectively. Taken together, these form a 16 bit pointer register called DPTR. These operate like the HL

Stack and Stack pointer

- Since the internal memory is limited to 128 bytes in an 8051 and to 256 bytes in 8052, internal memory pointers are 8 bits in size.
- The stack in 8051 family must reside in the internal memory. The stack pointer is therefore an 8 bit register. This is a special function register located at address 81H.
- The stack grows **upwards** using pre-increment and post decrement for the stack pointer.
- **Single bytes** are pushed on the stack or popped from it.
- Any **directly addressed byte** may be pushed or popped.
- The stack pointer is initialized to 07 at power on.

The Program Status Word

- PSW is located at the address D0H in SFR memory area.
- It contains flags etc. and is bit addressable.

CY	AC	F0	RS1	RS0	OV	F1	P

- Carry, Auxiliary Carry and Parity flags work like 8085 flags.
- F0 and F1 can be used as flag variables.
- RS1 and RS0 determine which Register Bank is active.
- OV flag is set on overflow.
  The overflow condition depends on the operation which has been carried out.

The Overflow flag

- After addition/subtraction, the overflow flag is set if the result has the wrong sign due to the operation. For example, 60H and 70H are positive, but when added, will give D0H, which would be interpreted as a negative number because the most significant bit is set. This will set the OV flag.
- After multiplication, OV is set if the result is > 255. The upper byte of product is placed in B.
- After division, OV is set if divide by 0 is attempted.

Register Banks

- The lowest 32 bytes of internal RAM can also be referred to as 4 banks of 8 registers each.
- The registers are called R0 to R7. At a given time, only one bank of registers is active.
- Two bits in a special function register called Program Status Word (PSW) decide which bank is active.
- PSW is located at address D0H in the SFR area.
- R0 refers to location 0 if bank 0 is active, to location 8 if bank 1 is active and so on.
- For example, if bank 2 is active, R6 refers to location $2 \times 8 + 6 =$16H.
- This allows foreground and background processing to have their own registers which reside in different banks.

Bit Addressable Memory

- 8051 provides special instructions where the operands are bits. Addresses used in these instructions are assumed to be bit addresses.
- Bit addresses form yet another address space.
- 16 bytes of internal RAM from address 20H to 2FH and certain Special Function Registers are bit addressable.
- Bit address 00 is the least significant bit of internal RAM byte address 20, bit address 07 is the most significant bit of the same bye, bit address 08 is the least significant bit of internal byte address 21 and so on.
- The 16 bytes from 20 to 2F take up bit addresses from 00 to 7F. Higher bit addresses are used for bits in special function registers.

Addressing modes

The 8051 provides the following addressing modes:

Immediate	Constants must be specified with a # symbol. A number without the # symbol is taken to be an address.
Direct	The address value is given directly as a number. Most assemblers define shorthands for special addresses. Register names ACC, B, SP, DPTR, DPH and DPL are in fact shorthands defined by the assembler. These are simply direct addresses.
Indirect	Data objects may be referred to by pointers, which are just variables containing addresses. Registers R0 and R1 can be used as pointers for internal RAM (which uses 8 bit addresses). 16 bit registers PC and DPTR can be used as pointers for ROM or external RAM.

Addressing modes

Indexed Some instructions use the sum of two registers as addresses. This is convenient for tables etc. where the address will be the sum of the start address of the table and a multiple of the index.

bit addressing Instructions operating on bits expect a bit address. The bit address can be specified as a direct address or in byte-addr.bit-No. format.

For example ACC.3 means bit 3 of the accumulator. Of course, the byte should be bit addressable.

Notation

Constants	Constants must carry a # symbol before them. ADD A, #5 means add the constant 5 to A, while ADD A, 5 means add the contents of location 5 in internal RAM to A.
Pointers	Operands specified by a pointer carry an @ signal before them. ADD A, @R0 means add to A the contents of the memory location whose address is in R0 and put the result in A.
Registers	Registers which can be used as 8bit pointers (R0 and R1) will be referred to as Ri. Any register from the set R0 to R7 will be represented as Rn.

Interpreting address by context

Addresses specified as numbers will be interpreted as byte addresses or bit addresses based on context.

MOV A, 5 : copy the contents of location 5 to A.
MOV C, 5 : copy the value of bit 5 of location 20H to carry.

The difference between the two is that the destination is a byte in the first case and a bit in the second case. So the address must be interpreted accordingly.

Bit addresses can also be specified in byte-addr.bit-No. format.

MOV C, ACC.5 means copy bit 5 of the accumulator (direct byte address E0H) to the carry flag, (which is actually bit 7 of PSW.

Data movement Instructions

- MOV dst, src
  Here dst can be A, Rn, a directly addressed byte or an indirect reference @Ri. The source can be A, Rn, direct, @Ri or #immediate. Both source and destination cannot use an R register.
- MOV DPTR, #data16
  The 16 bit immediate data value is loaded in DPTR. This is like LXI in 8085.
- MOV C, bit-addr and MOV bit-addr, C
  These instructions operate on single bits, copying the bit from the given bit address to carry or from carry to the given bit address.

Moving Data to and from the stack

The stack resides in internal RAM. It grows upwards as the stack pointer is pre-incremented to push and post decremented to pop. Byte sized objects are pushed and popped in 8051.

- push direct
  The stack pointer is pre-incremented and the value at the given direct addr. is copied to the addr. in stack ptr.
- pop direct
  The byte in internal RAM at the addr. given by SP is read. The stack pointer is then decremented and the byte read from the stack is written to the given addr.

Pushing and popping the stack pointer

As a special case, consider when SP itself is popped from stack. We want to transfer the value stored in the stack to SP. But will it get modified due to post decrementing of the stack pointer?

Assume SP contains 1A and the internal memory at addr. 1A contains 56. POP SP will read the address pointed to by SP (location 1A), find 56 there, decrement SP (to 19) and store the read value 56 at the destination addr. given in the instruction (which in this special case is SP itself).

So the decremented value 19 in SP will get overwritten by 56 and SP will have the value read from the stack, as desired.

Exchanging Data

- XCH A, src
  Here src can be a direct address, Rn or @Ri. The src byte then contains the value of A and A acquires the original value of the source byte.
- XCHD A, @Ri
  This exchanges just the lower nibble of the two operands. The upper nibble retains its original value.
- Exchanging the upper nibble
  The upper nibble may be exchanged by combining the above instructions. If we do
  XCH A, @Ri
  XCHD A, @Ri
  Both instructions exchange the lower nibble while the first exchanges the upper nibble also. Thus the upper nibble is exchanged once, while the lower nibble is restored to its

Swapping Nibbles

There is a special instruction to swap the upper and lower nibbles in A. The instruction is:
SWAP A

Also, the accumulator can be cleared by the instruction
CLR A

and complemented by the instruction
CPL A

Data movement from ROM

- MOVC A, @A+DPTR
  Move a constant from the ROM to A. The address to be accessed in ROM is the sum of the 8 bit value in A and the 16 bit value in DPTR. This is convenient for table look up. the start address of the table is put in DPTR and the offset derived from the index is in A.
- MOVC A, @A+PC
  This is like MOVC A @A+DPTR, except that the 16 bit pointer value is taken from (updated) PC instead of DPTR. This is used when constants are at some fixed offset from the current instruction. The offset is placed in A. Using this instruction allows one to access the constant irrespective of the absolute address at which the program is loaded.

Moving Data to and from External RAM

Data movement to and from the external RAM always involves A. The address in the External RAM is given as @DPTR or @Ri.

- MOVX A, @DPTR, MOVX @DPTR, A
  The 16 bit address to be used for the external RAM is taken to be the contents of DPTR.
- MOVX A, @Ri, MOVX @Ri, A
  The upper byte of the external address is latched using port output commands. The lower byte of the external RAM address is provided by Ri.

Addition and Subtraction

ADD, ADDC and SUBB are available as instructions for addition, addition with carry and subtraction with borrow. **Subtraction without borrow is not available.**

Addition and subtraction is always performed with A as the destination. The other operand can be a directly addressed byte, Rn, @Ri or #immediate.

Carry, Auxiliary Carry, Sign and Parity bits are set as in the case of 8085. The Overflow bit is set as the XOR of carry out of bits 6 and 7. This indicates a sign reversal due to addition/subtraction of legitimate positive or negative numbers.

Incrementing, decrementing and decimal adjust

The instructions INC and DEC add 1 to or subtract 1 from their operands. The operand can be A, a directly addressed byte, Rn or @Ri. **No flags are affected.**

A 16 bit increment is carried out by the instruction INC DPTR. Notice that there is no decrement for DPTR

The instruction DA carries out a decimal adjust after an addition. This involves adding 6 to either or both nibbles of A if there is a carry from them during addition or if their value is >9. The flags C and AC reside in the two most significant bits of PSW.

Multiplication and Division

8051 provides instructions for unsigned multiplication and division of 8 bit operands.

The multiply instruction (MUL AB) multiplies the contents of registers A and B. The MSB of the product is placed in B and the less significant byte is placed in A. The Overflow flag is set if the result is > 255.

The divide instruction (DIV AB) divides A by B and places the quotient in A and the remainder in B.

The overflow flag will be set if a divide by 0 is attempted. Else, the overflow flag is cleared.

Logical Instructions

We can perform bitwise AND, OR and XOR operations using ANL, ORL and XRL instructions.

These instruction take a destination and a source operand. The destination operand can be A or a directly addressed byte.

If A is the destination, the source operand can be a directly addressed byte, Rn, @Ri or # immediate.

If a direct address is the destination, the source operand can be A or # immediate. For example:

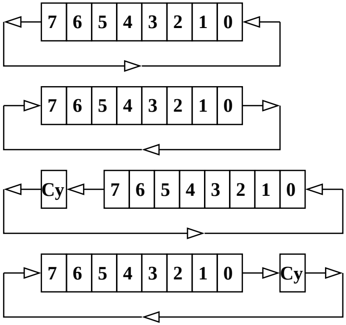
```
ANL   A, R5        ; AND A with R5 and put the result in A.
ORL   87H, #80H    ; OR the contents o addr 87H with 80H,
                   ; store result in 87H.
```

Logical instructions **do not** set flags. If the destination is a port, The latch (and not the pin) values are used for evaluation.

Rotate Instructions

Rotate instructions operate on A.
We can rotate towards left or right, with or without carry.

RL A	; Rotate A left
RR A	; Rotate A right
RLC A	; Rotate A left with Cy
RRC A	; Rotate A rt with Cy



Unlike 8085, flags are not affected by RL or RR and only the carry flag is affected by RLC and RRC.

Swap, clear and complement

All these instructions operate on A.

SWAP A swaps the upper and lower nibbles of A. It is equivalent to 4 shifts without carry in left or right direction.

CLR A makes clears all bits of A.

CPL A complements all bits of A.

Notice that CLR and CPL can also operate on bit sized operands such as carry or a bit address.

Bit oriented instructions

We copy a bit from a bit address to carry, or from carry to a bit address. We can also clear, set or complement the carry or the content of a bit address.

We can also perform AND or OR operation on **bits**

MOV	C,bit	;C = bit	MOV	bit,C	;bit = C
CLR	bit	;make bit=0	CLR	C	;make C=0
SETB	bit	;make bit=1	SETB	C	;make C=1
CPL	bit	;bit = $\overline{\text{bit}}$	CPL	C	;C=$\overline{\text{C}}$
ANL	C,bit	;C = C AND bit	ANL	C,/bit	;C=V AND $\overline{\text{bit}}$
ORL	C,bit	;C = C OR bit	ORL	C,/bit	;C=V OR $\overline{\text{bit}}$

Jump, Call and Return

Jump instructions modify the program counter such that the next instruction to be executed is from some specified program address.

Call instructions do the same, but they save the updated value of PC on the stack, (lower byte first) before overwriting it with the destination address.

RET pops the return address from the stack (higer order byte first) and copies it to PC.

8051 has various jump and call instructions which differ from each other in how the destination address is specified.

LJMP and LCALL instructions

In these jump and call instructions, the destination address is given as a 16 bit direct address.

The assembler syntax is
LJMP Label
LCALL Label

The 16 bit absolute address corresponding to the Label is placed in the instruction.

LJMP is encoded as 02H Addr(High) Addr(Low), while LCALL is encoded as 12H Addr(High)Addr(Low).

AJMP and ACALL instructions

LCALL and LJMP encode the full 16 bit destination address. Therfore, these are 3 byte instructions

If the destination address is within the same 2KB address space as the updated PC, shorter jump/call instruction can be used.

If the destination address is in the same 2K address space as the updated PC, it has the same 5 most significant bits as the PC.
Then we need not specify these and can come up with 2 byte instructions for jump/call.
These are the 2 byte AJMP and ACALL instructions.

AJMP and ACALL instructions

The assembler syntax for AJMP and ACALL is

AJMP Label
ACALL Label

Only 11 least significant bits of the destination label are included in the instruction. The instruction is encoded as
AJMP : A10 A9 A8 0 0 0 0 1 A7 A6 A5 A4 A3 A2 A1 A0
ACALL: A10 A9 A8 1 0 0 0 1 A7 A6 A5 A4 A3 A2 A1 A0

Only A10 to A0 are copied to PC, so the 5 most significant bits of the PC remain unchanged.

Therefore control can only be transferred between statements which lie in the same 2K page of program memory.

SJMP instruction

In this variant, the destination is provided as an 8 bit signed offset from the updated PC. The assembler syntax is

SJMP Label
The assembler actually calculates the offset of the label from the the instruction following SJMP, and encodes it in the instruction as an 8 bit signed number.

There is no corresponding CALL instruction.
This instruction can be used only if the target label is within +127 to -128 locations from the next instrucion.

It is encoded as: 80H, offset.

JMP @A+DPTR

This statement calculates the 16 bit sum of DPTR with A and jumps to this address. This is useful for managing jump tables.

Suppose we have an array of AJMP statements, of the type AJMP Label-i, each occupying 2 bytes.

If we want to execute the i'th AJMP statement, we can load the start address in DPTR and 2*i (offset) in A.

Now jmp @A+DPTR will jump to the i'th statement, which will in turn, take us to Label-i.

Conditional jumps

Conditional jumps are like SJMP. These specify the target address as an 8bit signed offset from the updated PC.

JB	bit-addr, Label	jump to Label if the addressed bit is set.
JNB	bit-addr, Label	jump to Label if the addressed bit is cleared.
JBC	bit-addr, Label	jump if bit set, clear the bit.
JC	Label	jump to Label if carry set.
JNC	Label	jump to Label if carry cleared.
JZ	Label	jump to Label if A = 0.
JNZ	Label	jump to Label if A is non zero.

Notice that there is no Z flag in 8051.
JZ and JNZ examine the accumulator contents to decide whether to jump.

Looping constructs

There are two conditional jumps which are particularly useful for looping.

DJNZ counter, Label;
decrements the counter and jumps to the given label if it is not zero. The counter could be Rn or a directly addressed byte. This is obviously useful for counted loops.

CJNE arg1, arg2, Label;
arg1 and arg2 are compared and the jump is taken if these are not equal. This is useful for loops which last till the loop variable attains a given value.

If arg2 is an immediate value, arg1 can be A, Rn or @Ri.
In addition to this, A can be compared to a directly addressed byte. The carry flag is set if arg2 $>$ arg1.

Miscellaneous Instructions

In addition to the instructions discussed upto now, the 8051 has the following instructions:

NOP	Do nothing
RETI	Return from Interrupt

RETI works just like RET, accept that it also restores interrupt enable values to their original settings.

(Interrupts may be automatically disabled based on their priority levels while an interrupt handler runs. These need to be re-enabled when the handler terminates.)

Read Modify Write Operations

Read Modify Write operations include logic operations ANL, ORL and XRL because the destination (either A or a directly addressed byte) is also one of the source operands.
Increment and decrement operations INC and DEC, DJNZ (because it uses decrement) and complement bit instruction CPL also belong to this class. The conditional jump JBC is also a read-modify-write operation.

Less obviously, instructions which just write to a single bit are also read-modify-write operations.
Thus, CLR Px.y, SETB Px.y and MOV Px.y, C
read Px, use masking and write back Px to write a single bit without affecting others.

When Read Modify Write operations are carried out on a port, the **latch** and **not the pin** values are read.