

**Microprocessors and Microcontrollers**  
**Prof. Santanu Chattopadhyay**  
**Department of E & EC Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 28**  
**8051 Microcontroller (Contd.)**

So, other addressing that we have is the absolute at absolute jump type of instruction. So, the absolute jump instruction so, the jump address is specified explicitly the problem with relative addressing is that this jump address has to be within 256 bytes from the current location. So, that may not be always possible.

So, what we can do is that we can take a small relative jump and from there we may have to take a long jump. So, that is by means of absolute addressing.

(Refer Slide Time: 00:49)

**8051 Instruction Format**

□ relative addressing

Op code	Relative address
---------	------------------

here: sjmp here ;machine code=80FE (FE=-2)  
 Range = (-128 ~ 127)

□ Absolute addressing (limited in 2k current mem block)

(A10-A8)	Op code	(A7-A0)
0700	1	org 0700h
0700 E306	2	ajmp next ;next=706h
0702 00	3	nop
0703 00	4	nop
0704 00	5	nop
0705 00	6	nop
0706 00	7	next: end
	8	end

07FEh

So, you cannot always deny the use of absolute addressing. So, in case of absolute jump instruction in 8051 so, the we have got different version 1 inversion is the AJMP instruction. So, this AJMP so, is the is an absolute jump, but here this it is limited to 2 kilobyte from the in the that is the current memory block so, limited in 2 kilobyte. So, you cannot go beyond to kilobyte.

So, this why because this address that is mentioned is this a 0 to a 7 and this a 8 to a 10. So, total 11 bits are devoted for keeping the offset and by 11 bits. So, you can mention the numbers in the range of total range of numbers that you can have is 2 kilobyte.

So, as a result so, it cannot be more than 2 kilobyte. So, we can have this this AJMP next instruction. So, this next value is 0 7 this next value is here. So, that is this instruction ends as 0 7 0 5. So, next instruction this one is at address 0 7 0 6 this is the 0 7 0 6 so, this 0 7 0 6. So, this is it will be. So, u one is the u one is the code that we have for AJMP. So, actually these bits are actually mixed. So, you cannot really differentiate very easily so, you have to distribute it and then only it will come. So, this is not 0 7. So, this will be based on if you if you just look into the code then it will be like this.

So, here this 0 7 0 6 is put directly it is distributed in these bits and these bits.

(Refer Slide Time: 02:45)

**8051 Instruction Format**

Long distance address

Op code	A15-A8	A7-A0
---------	--------	-------

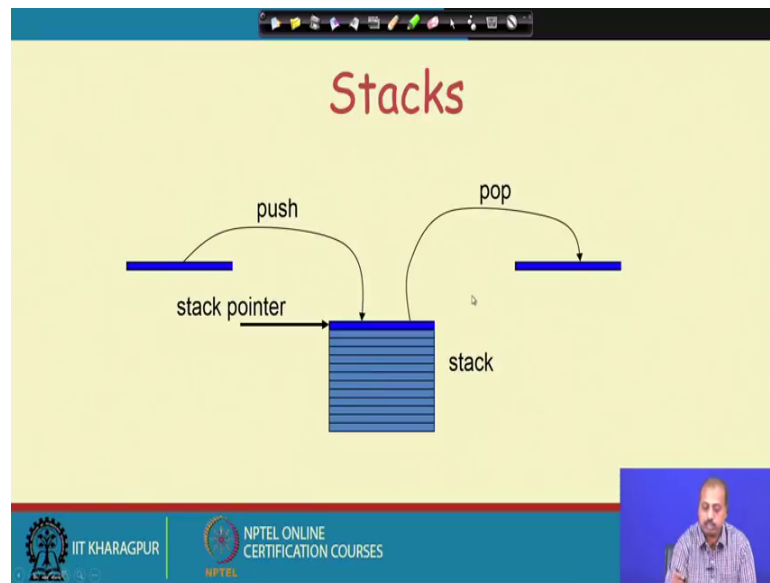
Range = (0000h ~ FFFFh)

0700	1	orl 0700h
0700 020707	2	ljmp next ;next=0707h
0703 00	3	nop
0704 00	4	nop
0705 00	5	nop
0706 00	6	nop
	7	next:
	8	end

The slide also features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, and a small video inset of the presenter.

So, this is actually this this this is the other version of this jump instruction, which is ljmp. So, this is ljmp instruction.

(Refer Slide Time: 02:57)



So, this is this is ljmp there is a long jump. So, here we do not have the restriction. So, in the sjmp instruction we had this in the sjmp instruction we had this thing this offset is only 11 bits a 0 to a 10, but in this ljmp instruction. So, offset is total 16 bits. So, you can branch across the total 64 k address space that you have. So, the range is very high all 0 0 0 0 to FFFFh.

So, here when we are saying like ljmp next so, ljmp next so, it is. So, we it the value will be calculated. So, next address is 0 7 0 7 and the off code is 0 2 and then the, whatever be the address. So, that will be coming here. So, that way this long address will be calculated.

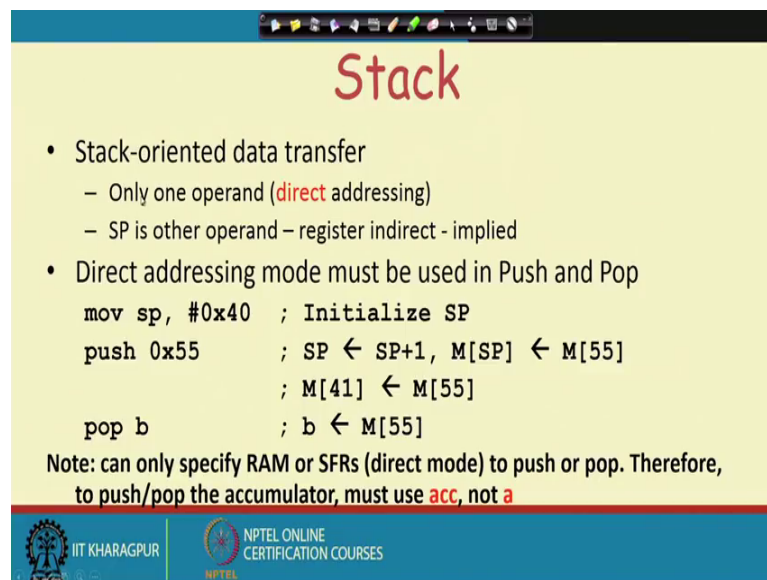
So, we have got these 3 variants of jump instruction we have got the relative jumps specified with the relative jump instruction specified as sjmp, we have got this absolute jump instruction specified by ajmp and this absolute jump instruction is is a 2 byte instruction and a relative jump instruction is also a 2 byte instruction, but this ljmp instruction is a 3 byte instruction.

So, that way this ljmp is slightly costly in terms of the area requirement and when this microcontroller is being used you know that this memory space is scarce. So, we would like to we like to save as much memory as possible in the code part. So, if it is not absolutely necessary we should not use ljmp. So, we should restrict us to ajmp instruction because that will require one byte less otherwise there is no difference.

The next important concept that we have is the stack. So, the stack as we have seen that is so, it is a part of memory and the stack pointer it points to the top of the stack.

So, we can have 2 operations push and pop, but the push operation will push the next whatever the value that we want to put into the stack into the location pointed to by stack pointer. And the pop will take out the value from the top of the stack and get into the location that we want.

(Refer Slide Time: 05:20)



The slide is titled "Stack" in a large, red, serif font. It contains two main bullet points. The first bullet point is "Stack-oriented data transfer", which has two sub-points: "Only one operand (direct addressing)" and "SP is other operand – register indirect - implied". The second bullet point is "Direct addressing mode must be used in Push and Pop". Below the bullet points is a block of assembly code: `mov sp, #0x40 ; Initialize SP`, `push 0x55 ; SP ← SP+1, M[SP] ← M[55]`, `; M[41] ← M[55]`, and `pop b ; b ← M[55]`. Below the code is a note: "Note: can only specify RAM or SFRs (direct mode) to push or pop. Therefore, to push/pop the accumulator, must use **acc**, not **a**". At the bottom of the slide, there are logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

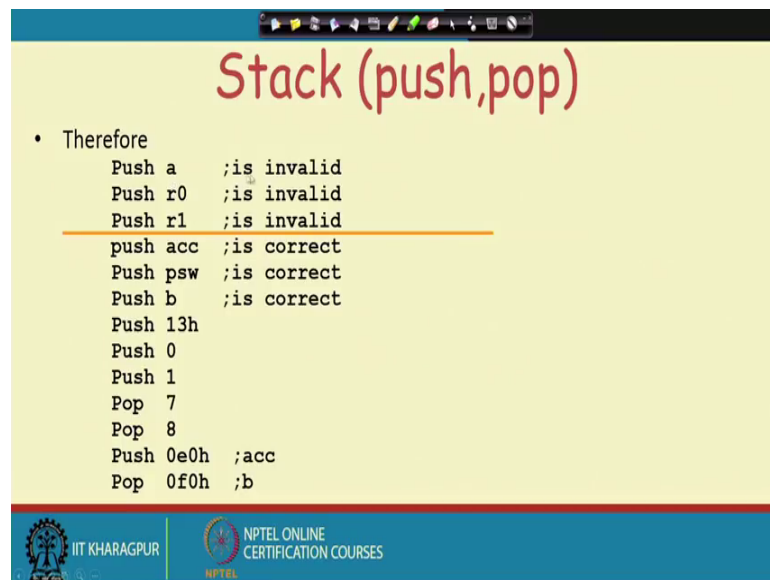
So, for stack oriented data transfer. So, only one operand can be specified. So, we have to just tell the value that we are trying to push or pop. And the stack pointer is the other operand so, because the memory access will be in terms of the stack pointer only. So, in on the only direct addressing it can be used. So, we can so, before doing any per stack operation the most important thing is to initialize the stack pointer.

So, if we do not initialize the stack pointer properly then this stack pointer may be pointing to some garbage location and then this push pop will arbitrarily modify some memory location. So, what is done? So, for so, before doing any push swap operation the program should first set this stack pointer and that may be done at the beginning of the program and later on when it is. So, as it is doing other operate the push pop operation. So, since the program has set it properly it is expected that it will not corrupt any of the important program and data parts.

So, here move SP comma hash 0 x 4 0 hex. So, the value of 40 hex will be put into the stack pointer. So, stack pointer is initialized to that then push 0 x 55. So, what happens is that the memory look first this stack pointer is updated stack pointer is updated by one incremented by 1, then the memory location stack pointer gets the value 55 x. So, what happens actually is the since stack pointer is at 40. So, memory location 41 will get the content of memory location 55. So, this is again you see that one way of transferring data between 2 memory locations.

So, from memory location 55 we are transferring the data to the memory location 41 the other instruction that we have is the pop instruction. So, you can again mention some address or you can mention these special registers. So, pop b. So, b gets the value of memory location 55. So, you see that can only the we can only specify RAM or SFR registers to push or pop. So, we can to push pop accumulates. So, we should use acc and not a. So, a cannot be used because a is treated as a register, but acc is treated as a special function resistance that is why we cannot use a, but we have to use acc in the instruction, but anyway apart from that. So, you can directly specify the memory addresses for those registers.

(Refer Slide Time: 08:00)



The slide is titled "Stack (push, pop)" in red text. It contains a list of assembly instructions with comments indicating their validity. The instructions are:

- Therefore
- Push a ;is invalid
- Push r0 ;is invalid
- Push r1 ;is invalid
- push acc ;is correct
- Push psw ;is correct
- Push b ;is correct
- Push 13h
- Push 0
- Push 1
- Pop 7
- Pop 8
- Push 0e0h ;acc
- Pop 0f0h ;b

The slide also features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES at the bottom.

So, push a is invalid push r 0 is invalid, push r 1 is invalid. So, these 3 are invalid because these are not. So, I cannot mention registers like this, we can write like push accumulator push acc, push psw, push b, push 13 x, push 0. So, all these can be done or

you can even write in terms of the special the register numbers like push 0 e 0 hex. So, that is same as this push acc then pop 0 f 0 hex. So, that is same as you know this pop b. So, that way so, these push pop instructions. So, these assembly assemblers so, they will not accept these things they, but they will accept this one.

So, that is just for can say some sort of convenience.

(Refer Slide Time: 08:52)

The slide is titled "Exchange Instructions" in red. It lists four instructions under the heading "two way data transfer":

- `XCH a, 30h` ; a ↔ M[30]
- `XCH a, R0` ; a ↔ R0
- `XCH a, @R0` ; a ↔ M[R0]
- `XCHD a, R0` ; exchange "digit"

Hand-drawn blue circles highlight the second operand of each instruction. Below the code, a diagram shows two 8-bit registers: `a[7..4] a[3..0]` and `R0[7..4] R0[3..0]`. Arrows indicate that only the lower 4 bits (`[3..0]`) are exchanged between the two registers. The slide footer includes the IIT KHARAGPUR logo and the text "NPTEL ONLINE CERTIFICATION COURSES". A small video inset of the presenter is visible in the bottom right corner.

So, there is another type of data transfer instruction which which is known as the exchange. So, exchange. So, this exchanges the values of 2 bytes so, XCH a comma 30 hex. So, the content of memory location a will be exchanged with memory location sorry content of register a will be exchanged with memory location 30. And again the same thing that all this instruction this a part is fixed. So, XCH a so, this part is fixed. So, only the second thing can be specified.

So, you can say like XCH a comma at the rate r 0. So, call so, a will be exchanged with memory location r 0.8 2 by r 0 and there is a special version XCHD. So, it is the exchange digit. So, only 4 bits are exchanged. So, this lower order 4 bits will be exchanged, like between a and r 0 this lower order 3 4 bits will get exchanged ok.

So, this is useful in many cases like particularly for operating system design. So, this type of exchange instructions are useful, particularly when we are going to implement

say semaphore type of synchronization primitives then this type of exchange of data will be useful.

(Refer Slide Time: 10:18)

## Bit-Oriented Data Transfer

- transfers between individual bits.
- Carry flag (C) (bit 7 in the PSW) is used as a single-bit accumulator
- RAM bits in addresses 20-2F are bit addressable

```

mov C, P0.0

mov C, 67h

mov C, 2ch.7
    
```

RAM	
Byte address	Bit address
27	3F 3E 3D 3C 3B 3A 39 38
26	37 36 35 34 33 32 31 30
25	2F 2E 2D 2C 2B 2A 29 28
24	27 26 25 24 23 22 21 20
23	1F 1E 1D 1C 1B 1A 19 18
22	17 16 15 14 13 12 11 10
21	0F 0E 0D 0C 0B 0A 09 08
20	07 06 05 04 03 02 01 00
1F	
18	Bank 3
17	
16	Bank 2
15	
14	Bank 1
13	
12	
11	
10	
0F	
0E	
0D	
0C	
0B	
0A	
09	
08	
07	Default register bank for RD-R?
06	
05	
04	
03	
02	
01	
00	

Byte address	Bit address
7F	7F 7E 7D 7C 7B 7A 79 78
7E	77 76 75 74 73 72 71 70
7D	6F 6E 6D 6C 6B 6A 69 68
7C	66 65 64 63 62 61 60
7B	5F 5E 5D 5C 5B 5A 59 58
7A	57 56 55 54 53 52 51 50
79	4F 4E 4D 4C 4B 4A 49 48
78	47 46 45 44 43 42 41 40

General purpose RAM

Next, we will look into bit oriented data transfer. So, this is a very powerful feature that this 8051 microcontroller has and for that matter most of the microcontrollers, we will see that it has got this bit addressability and this makes it powerful, because many times we do we have to set the different bits in a different fashion.

So, making the mask pattern accordingly and setting the bits that way. So, we it requires a large amount of code and maybe we just put said the number then do left shift or right shift to come to the proper pattern and all that. So, those can be avoided and we can just access individual bits directly to put the values there. So, this bit oriented data transfer. So, it transfers between individual bits. So, this carry flag a carry that is c that is used as single bit it can be used as a single bit accumulator and this ram bit 20 to 2F so, they are all bit addressable.

Like we can have move C comma P0.0. So, this port 0s bit number 0 will come to the carry flag then this 67 hex like say this one. So, this can be moved to the carry flag. So, we can have it like this then this 2C hex dot 7. So, 2 see hex is this 1 dot 7. So, this is a same this is the same as that bit 67 67 hex. So, that is same as that one. So, it will be sorry yeah. So, bit number 67 x and these 2 C x dot 2 C x dot 7 they are same. So, they will come to the carry flag.

So, they say we can use these bit oriented instructions for data transfer.

(Refer Slide Time: 12:00)

### SFRs that are Bit Addressable

SFRs with addresses ending in 0 or 8 are bit-addressable.  
(80, 88, 90, 98, etc)

Notice that all 4 parallel I/O ports are bit addressable.

Byte address	Bit address	Byte address	Bit address
98	9F 9E 9D 9C 9B 9A 99 98	SCON	FF
90	97 96 95 94 93 92 91 90	P1	F0 F6 F5 F4 F3 F2 F1 F0
8D	not bit addressable	TH1	E0
8C	not bit addressable	TH0	D0
8B	not bit addressable	TL1	E7 E6 E5 E4 E3 E2 E1 E0
8A	not bit addressable	TL0	B8
89	not bit addressable	TMOD	- - - BC BB BA B9 B8
88	8F 8E 8D 8C 8B 8A 89 88	TCON	B0
87	not bit addressable	PCON	B7 B6 B5 B4 B3 B2 B1 B0
83	not bit addressable	DPH	A8
82	not bit addressable	DPL	A0
81	not bit addressable	SP	A7 A6 A5 A4 A3 A2 A1 A0
80	87 86 85 84 83 82 81 80	P0	99
			not bit addressable

Now, the special function registers that are Bit Addressable so, you see that if you look into the pattern like say this P 0 then this T con. So, they are bit addressable others are not. So, you see that that addresses which end with 0 or 8 are bit addressable like 8 0 8 8 9 0 9 8. So, they are bit addressable and all 4 parallel IO ports. So, they are all bit addressable. So, that is there. So, P 0 P 1 P 2 P 3 they are bit addressable and this registers who end with 8 or 0 that is divisible by 8. So, they are all bit addressable.

(Refer Slide Time: 12:45)

## Data Processing Instructions

Arithmetic Instructions  
Logic Instructions



Next we look into data processing instructions there are several data processing instructions.

(Refer Slide Time: 12:50)

## Arithmetic Instructions

- Add
- Subtract
- Increment
- Decrement
- Multiply
- Divide
- Decimal adjust

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

In 8 0 5 1 like add subtract increment decrement multiply divide decimal adjust. So, like that.

(Refer Slide Time: 13:03)

## Arithmetic Instructions

Mnemonic	Description
ADD A, byte	add A to byte, put result in A
ADDC A, byte	add with carry
SUBB A, byte	subtract with borrow
INC A	increment A
INC byte	increment byte in memory
INC DPTR	increment data pointer
DEC A	decrement accumulator
DEC byte	decrement byte
MUL AB	multiply accumulator by b register
DIV AB	divide accumulator by b register
DA A	decimal adjust the accumulator

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

So, we can have like adequate instruction, like Add A comma byte Add A 2 byte and put the result in A register, Add C comma byte. So, Add C A comma byte. So, it will add with carry. So, it is a plus byte plus carry will be put into the A register, then subtract

with borrow SUBB. So, that will be there. So, this way we can have a number of such instructions.

(Refer Slide Time: 13:30)

**ADD Instructions**

add a, byte ;  $a \leftarrow a + \text{byte}$

addc a, byte ;  $a \leftarrow a + \text{byte} + C$

These instructions affect 3 bits in PSW:

C = 1 if result of add is greater than FF

AC = 1 if there is a carry out of bit 3

OV = 1 if there is a carry out of bit 7, but not from bit 6, or visa versa.

Bit	7	6	5	4	3	2	1	0
Flag	CY	AC	FO	RS1	RS0	OV	F1	P
Name	Carry Flag	Auxiliary Carry Flag	User Flag 0	Register Bank Select 1	Register Bank Select 0	Overflow flag	User Flag 1	Parity Bit



So, you will see some of them like add a, comma byte a gets a plus byte and a dc add CA comma byte a gets a plus by plus C. These instructions will affect 3 bits of PSW like it will affect this carry bit CY auxiliary carry bit and the overflow bit.



So, if C equal to one if the result of add is greater than FF. So, the carry is generated then the carry flag will be set this they auxiliary carry will be 1, if there is a carry from bit 3 to bit 4. So, if there is a carry then the after the after the nibble if there is a carry, then this bit will be set to one and this overflow will be set if there is an overflow carry coming out of bit 7, but not from bit 6. So, there is a carry out of bit 7, but not from bit 6 or vice versa then if the overflow flag will be set.

(Refer Slide Time: 14:22)

## Instructions that Affect PSW bits

Instruction	Flag			Instruction	Flag		
	C	OV	AC		C	OV	AC
ADD	X	X	X	CLR C	0		
ADDC	X	X	X	CPL C	X		
SUBB	X	X	X	ANL C,bit	X		
MUL	0	X		ANL C,/bit	X		
DIV	0	X		ORL C,bit	X		
DA	X			ORL C,/bit	X		
RRC	X			MOV C,bit	X		
RLC	X			CJNE	X		
SETB C	1						



So, these are the instructions that will affect the PSW register like the add affects C carry overflow and auxiliary carry add C again affects all of them.

Now, some of the interesting things that we have got is multiply instruction that will set the carry to 0 and the overflow is affected, but not the other one. Similarly division so, this will set the carry flag to 0 and it will not affect the overflow, but not the accumulator then SETB carry. So, as the name suggests the carry should be set to one say bit carry. So, this carry is set to one similarly clear carry will make the carry bit 0 complement carry. So, whatever be the current value of the carry bit. So, that will be complemented then we have got a logical C bit.

So, this is so, this will be ending the bit with the carry. So, that may be if this bit is 0. So, it is clear carry. So, if this bit is one. So, this will be some the previous will continue. So, you can position you can mention some bit number with which the carry will be ended then we have got this move C. So, we will see these instructions.

(Refer Slide Time: 15:32)



### ADD Examples

```
mov a, #3Fh
add a, #D3h
```

0011 1111
<u>1101 0011</u>
0001 0010

- What is the value of the C, AC, OV flags after the second instruction is executed?

C = 1  
AC = 1  
OV = 0




Now like this addition example like move a comma 3 FX and then at a comma has D 3. So, what will be the value of this? So, if we do this addition. So, it will be like this. Now, so, the a the affect the things that are affected is carry equal to 1 is equal to 1 and OV equal to 0.

(Refer Slide Time: 15:58)

### Signed Addition and Overflow

<b>2's complement:</b>	
0000 0000	00 0
...	
0111 1111	7F 127
1000 0000	80 -128
...	
1111 1111	FF -1

0111 1111	(positive 127)
<u>0111 0011</u>	(positive 115)
1111 0010	(overflow cannot represent 242 in 8 bits 2's complement)
1000 1111	(negative 113)
<u>1101 0011</u>	(negative 45)
0110 0010	(overflow)
0011 1111	(positive)
<u>1101 0011</u>	(negative)
0001 0010	(never overflows)



Then if you is so, 2's complement notation. So, we have got these since this number it is 7 F and this is 8 0.

So, 80 is minus 128 and this number is 127, if you do the addition then the result it should be minus 1. So, in the binary notation if we do the additions 11 or so, these the auxiliary single equivalent is FF, which is minus 1.

So, similarly if this is 127 there so, this is number 127 this is 115. So, if you add it then there will be an overflow, because this value 242 cannot be represented in 8 bit two's complement format. So, there is an overflow and similarly we have got say negative of 113. So, this is coded like this negative of 45 coded like this again there will be an overflow, because we cannot represent the number there, but if we do it say this is positive and this is negative. So, after doing the addition there is no overflow.

(Refer Slide Time: 16:58)

**Addition Example**

```

; Computes Z = X + Y
; Adds values at locations 78h and 79h and puts them in 7Ah
-----
X      equ    78h
Y      equ    79h
Z      equ    7Ah
-----
      org 00h
      ljmp Main
-----
Main:  org 100h
      mov a, X
      add a, Y
      mov Z, a
      end
  
```

Handwritten diagram:

```

0000: ljmp #0100
-----
0100:
  
```

The diagram shows a vertical dashed line representing memory addresses. A blue box encloses the instruction at 0000: ljmp #0100. An arrow points from this instruction to the start of the Main routine at 0100:.

So, that way we can have different addition results affecting the carry auxiliary carry overflow flags. Now suppose we are writing a program that will be doing this Z equal to X plus Y. So, it will add the contents of locations 78 hex and 79 hex and put them in the register into the memory location 7A.

So, what we do? So, we define 3 constants X Y and Z and as I said that this equ are equal. So, this is an assembler directive that tells the assembler that in my program wherever I am writing X you should replace it with 78. So, that is a constant number and org 00 hex. So, this is this way this means that also this is the assembly will start at location 00 hex.

So, this the program when it is loaded. So, the first instruction LJMP main will be at location 0 0 hex. So, if you look into the memory map. So, it will be like this at location 0 0 0 0. So, it will put the instruction LJMP main now for main. So, this is before that we have got this org 100. So, this at location from 0 1 0 0 my actual program starts. So, there I have got this move X and all these codes they are starting from this point onwards. So, when we start the program the start the system. So, it should go to location 0 1 0 0.

So, the assembler when they are generating the object code so, it will do it like this that at location 0 0 0 0 it will put the instruction LJMP 1000 and at location 0 1 0 0 onwards. So, it will be putting the code for this this piece of program. So, when the program will actually be executed the PC, when we reset the processor the PC will be 0 0 00. So, it will come to this instruction and it will make this executed LJMP 0 1 0 0 as a result it will start executing from this point onwards.

So, what are we doing here? So, first we are moving the content of X move a comma 78 hex. So, content of memory location 78 will come to a then add a comma Y. So, content of memory location 79 will be added to a and then move Z comma a. So, content content of the accumulator will be saved into the memory location 7 a hex. So, this way this program will be executed and do the addition.

(Refer Slide Time: 19:48)

**The 16-bit ADD example**  
 ; Computes Z = X + Y (X,Y,Z are 16 bit)  
 ;-----  
 X equ 78h  
 Y equ 7Ah  
 Z equ 7Ch  
 ;-----  
 org 00h  
 ljmp Main  
 ;-----  
 org 100h  
 Main:  
 mov a, X  
 add a, Y  
 mov Z, a  
 mov a, X+1  
 adc a, Y+1  
 mov Z+1, a  
 end

Handwritten notes on the slide:  
 78, 79 X  
 7A, 7B Y  
 7C, 7D Z  
 78 X  
 79 X+Y  
 a X+Y

So, next we look into a slightly more complex version, where this XYZ these numbers are 16 bit numbers. So, instead of being 8 bit number they are 16 bit numbers, but 8 0 5 1

will do 8 bit addition only. So, we have to be careful that from 1 stage to the next stage. So, that carry has to be propagated.

So, what we do? So, now, individual 7 8 so, individual locations as 16 bit the number X is 16 bit. So, it is. So, we assume that at sorry we assume that at location 7 8 hex, we have got the number X. So, the location 7 8 a 7 8 and 7 9 they will hold the value of X and then 7 A and 7 B, they will hold the value of Y and these 7 C and 7 D they will correspond to the variables they will correspond to the variable Z. So, this X Y and Z we are defining their addresses as 7 8 7 7 8 7 a and 7 CX.

Now, the program that we write is like this that first, we get the content of X into the a register. So, first byte 7 8 the bytes the memory location 7 8 content comes to the a register. So, if this is the location 7 8 and this is the location 7 9, now since intel follows the convention that the higher order byte will be at higher order address.

So, here I have got the X the lower part of it and at the next location I will have the higher part of it. So, first the lower bytes are to be added. So, the 7 8 content of memory location 7 8 comes to a and then add a comma Y. So, this will add the content of memory location 7 A with accumulator.

And the whatever be the result. So, that we are storing at location 7 C hex and then content of memory location X plus 1 that is X is 7 8 so, 7 8 plus 1 7 9. So, content of memory location 7 9 will come to a and then we will be adding the content of memory location Y plus 1 that is 7 b, with that a and here instead of using add we are using add C, because previously some carry would have been generated and that carry has to be added now because. So, we have got 16 bit numbers. So, 2 16 bit numbers so, we just added this lower lower order byte now from here there may be one carry generated. So, that carry has to be added in the next phase.

So, that is why this adc is used. So, this adc is instruction. So, this will be adding a and Y plus 1 and the carry will also be added. And the result will be stored at memory location Z plus 1 that is the location 7 7 D that is higher order byte. So, it will go to the higher order address ok.

(Refer Slide Time: 23:00)

The slide is titled "Subtract" and contains the following content:

SUBB A, byte	subtract with borrow
--------------	----------------------

Example:  
SUBB A, #0x4F ; A ← A - 4F - C

Notice that  
There is **no** subtraction WITHOUT borrow.  
Therefore, if a subtraction without borrow is desired,  
it is necessary to **clear the C** flag.

Example:  
CLR C  
SUBB A, #0x4F ; A ← A - 4F

The slide footer includes the IIT KHARAGPUR logo and the text "NPTEL ONLINE CERTIFICATION COURSES". A small video inset of a speaker is visible in the bottom right corner.

So, next you see the subtract instruction SUBB subtract with borrow. So, subtract with borrow. So, this will be doing like say a comma SUBB a comma hash 0 X 4 F. So, what it will do. So, accumulator will get a minus 4 F minus C, but we do not have any sub instruction. So, there is no instruction where this b is not there. So, this you do not have any instruction where this b is absent though we for add we have got add and add C, but here we do not have anything only.

So, whenever you are doing a subtraction. So, we have to be careful that we clear the carry first. So, whenever before doing any subtract operation, we have to clear the carry first and then we have to say like subtract with borrow a comma whatever subtraction we are interested in since so, that way we have to do it?





(Refer Slide Time: 24:00)

## Increment and Decrement

INC A	increment A
INC byte	increment byte in memory
INC DPTR	increment data pointer
DEC A	decrement accumulator
DEC byte	decrement byte

DPL

- The increment and decrement instructions do **NOT** affect the C flag.
- Notice we can **only** INCREMENT the data pointer, not decrement.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES

Then we have got the increment and decrement instructions like in ins INC A it increments a then INC byte it increments byte INC DPTR it increments the data pointer similarly you have got decrement a decrement by type of instruction. So, the increment and decrement instructions they do not affect the carry flag. So, this is one difference compared to say 8 0 8 5 where they were affecting the carry flag, but they will affect the 0 flag, but they will not affect the carry flag. So, the only another interesting thing that we have is that we can only increment that data pointer we cannot decrement the data pointer.

So, if you if you have an array and in your external storage and if you have an array in your external storage and you say that. So, DPTR is pointing to this then for accessing successive locations. So, you can increase the DPTR well, but if you want to go upward, that is not possible so, you cannot decrement the DPTR value. So, for that purpose you have to use that DPL register. So, you have to use the DPL register. So, you can increment decrement the DPL, but when you are you are using a pair whenever you are using a pair. So, you cannot do this thing. So, it you have to write into DPTR is always increment only.

(Refer Slide Time: 25:38)

**Example: Increment 16-bit Word**

- Assume 16-bit word in R3:R2

```
mov a, r2
add a, #1 ; use add rather than increment to affect C
mov r2, a
mov a, r3
addc a, #0 ; add C to most significant byte
mov r3, a
```

The diagram shows two registers, R2 and R3, each divided into two bytes. R2 has a checkmark in the lower byte, and R3 has a checkmark in the upper byte. An arrow points from the lower byte of R2 to the upper byte of R3, indicating the propagation of the carry flag.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, next we look into an example of implementing a 16 bit word. So, move. So, assume that the 16 bit word is stored in the pair r 3 r 2. So, r 3 has got higher order byte r 2 has got the lower order byte. So, first we get the content of r 2 into a register with add a comma hash 1. So, rather than increment because if we if I you I can I could have used increment here, but if I use increment instruction then the carry flag will not get affected, but since it is a 16 bit addition we want that the from that that the carry should be generated. So, when this addition is done? So, when you are adding these to this carry should be generated so, that this carry can be propagated to the next stage of addition.

So, that is why instead of doing this the increment we use this add one then the result we are storing in the r 2 register and the r 2 register value is incremented and then we have to. So, it may generate some carry. So, then r 3 value is moved on to this a register then add C a comma has 0. So, this will be a we already have r 3 and if some carry has been generated. So, that carry will get added and since r 2 is already added.

So, we do not have this r 2 to be added again. So, this is 0. So, that that way we get this ultimately this a value has to be moved to r 3. So, so this r 2 r 3 pair is incremented and r 2 is incremented by 1, if a carry has been generated that carry gets added with the content of r 3 that is how this program is executing?

So, these ways if you are have if you are doing 16 bit operation. So, many a times instead of using the increment decrement instruction, we have to use this add instruction or subtract instruction to make the program carry.