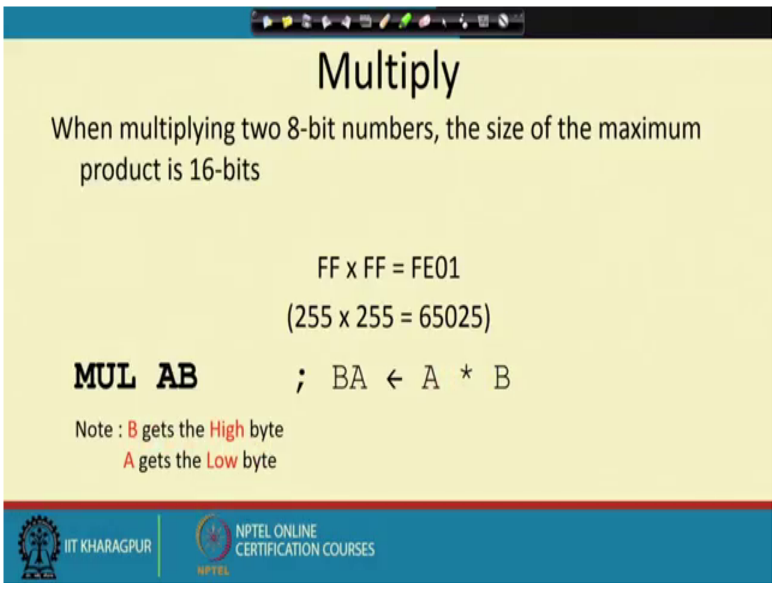
Microprocessors and Microcontrollers
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture - 29
8051 Microcontroller (Contd.)

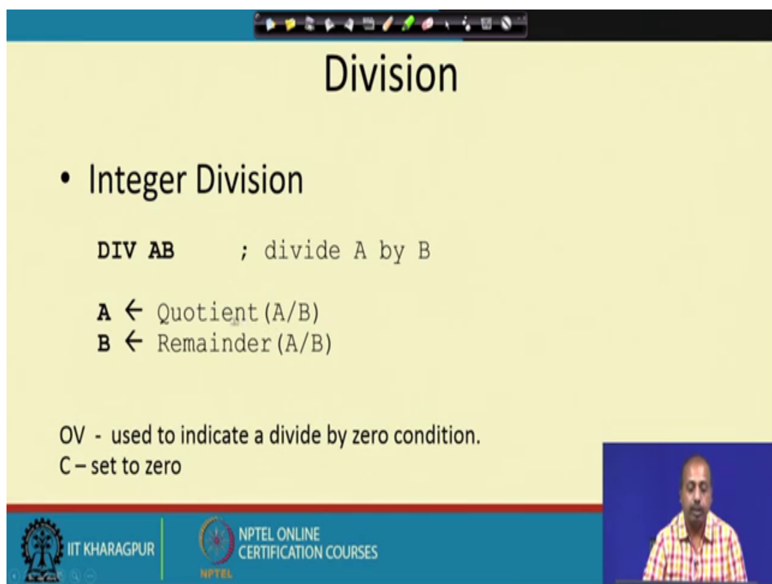
(Refer Slide Time: 00:28)



Next, we consider the multiplication instruction, now to multiplying two 8-bit numbers, so size of the maximum product is a 16 bit two 8-bit numbers multiplied it will give a 16 bit number. So, for example, this FF multiplied by FF set it will produce FE01 which is just 65025, 255 multiplied by 255. The instruction here that we have is MUL AB, and this whole instruction is the opcode you see. So, though A B they are specified separately, but they are we do not have we do not have any other operand that can be specified; and apparently it seems that there is a mistake between A and B, there should be a comma, but it is nothing like that.

So, the whole instruction is MUL blank A B and for multiplication this A and B registers should have the two numbers to be multiplied. And as a result this B register will have the higher order byte, and this A register will have get the lower order byte. For this particular multiplication example that we have so this 0 1, it will go to A register and FE will come to that B register, so that there is a no other way that by which you can do a multiplication in 8051.

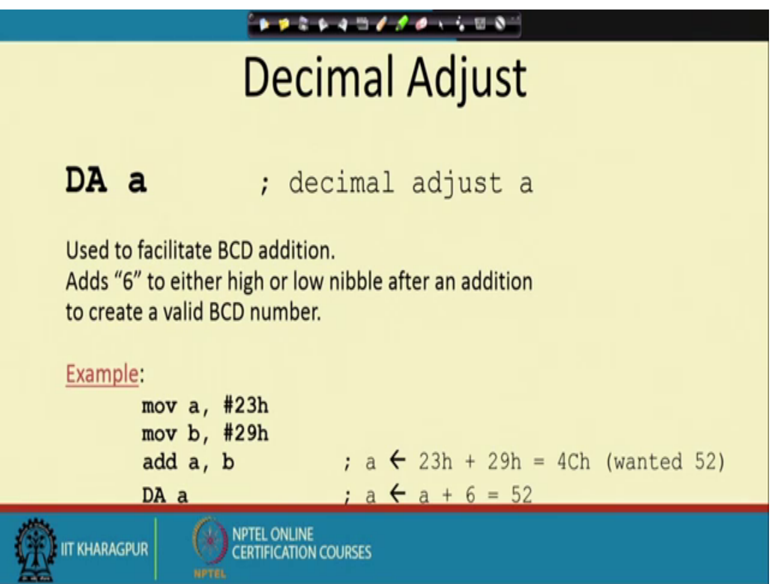
(Refer Slide Time: 01:29)



On the other hand, this division operation should DIV AB. So, here also we do not have any other way to specify other operands, so it will divide A by B, and A gets the quotient and B gets the remainder so that is fixed. And overflow OV bit is used to indicate it divide by 0 error condition. If there is an overflow, the division operation then if this OV bit is set so that will mean that there was a divide by 0 error, and this carry will always be set to 0. So, irrespective of the situation so carry will be set to 0, so that is the division operation.

So, we have got multiplication, division, if you remember 8085, we do not have any multiplication division operation in 8085; but in a 8051, we have got multiplication and division though only 8 bit in nature. So, if you want a higher size multiplication then you have to follow some multiplication algorithm by which you can multiply 8-bit numbers at a time. So, each 8-bit number can be considered as a digit, accordingly you can try to devise one multiplication and division algorithm.

(Refer Slide Time: 02:35)

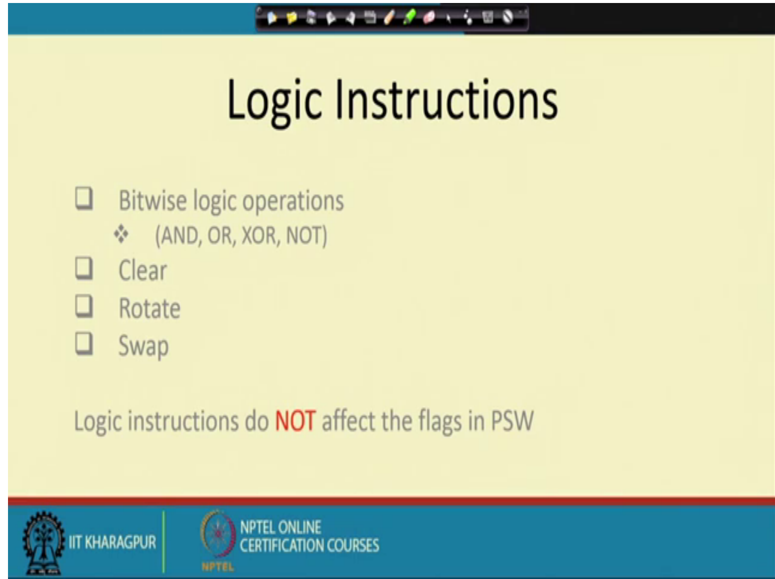


Another instruction that is that we are familiar with 8085 also DA a - decimal adjust accumulator. So, here also we have that version. But in case of 8085, you remember that DAA this whole thing was a single word; in this 8051 we have got DA as a word and a as another word, but again you cannot have any other operand excepting a. So, these operands are fixed. So, this operand being fixed, so you cannot modify it. So, what is required is that we should do the multiplication, we should do this addition and then maybe we can do this adjustment.

For example, in this case we are moving the number 23 h to A register, 29 h to B register, and assuming that we are treating them as a decimal number. So, if it is a addition a b, so in case of hexadecimal the value is 4 C h. So, what in this particular example what we wanted is that we will do a decimal addition that is 9 plus 3 - 12. So, 2 1 carry then 2 plus 2 - 4 plus 1 - 5 52. So, the result that we expect is 52.

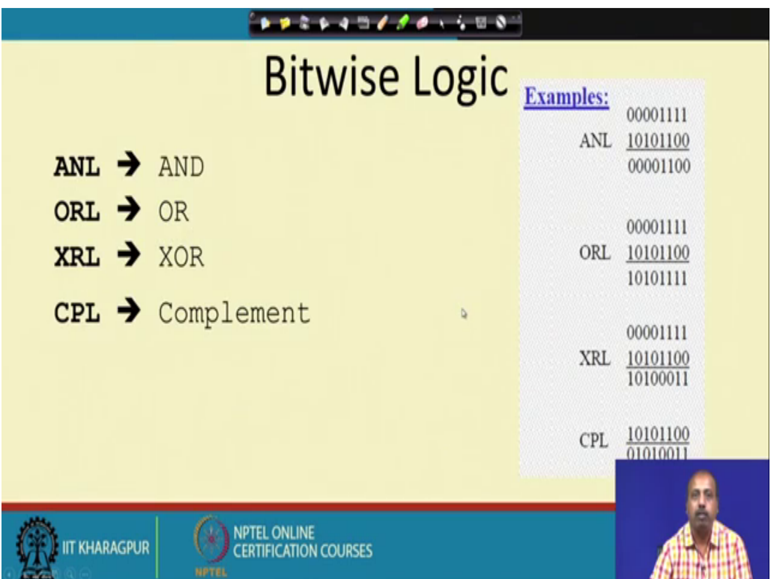
But if you do a normal addition operation then you will get add A B where it will give us 4 C hex. And as we know that for this conversion from this from this hexadecimal number to the corresponding BCD number, so we have to add six to the lower order nibble after the addition. So, 6 is added, so with 4 this a equal to a plus 6, so the number from 4 c you it will become 52, so that way we can do this decimal adjust instruction for converting into BCD number.

(Refer Slide Time: 04:23)



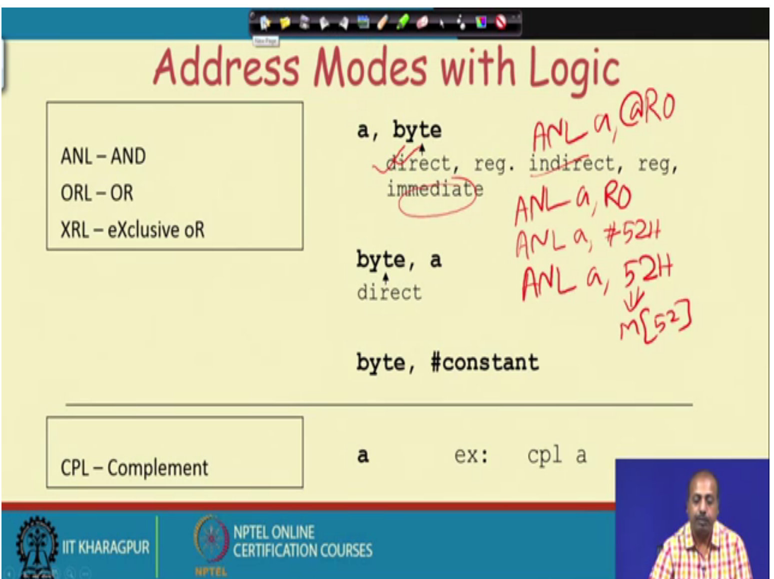
Then we have got logic instructions AND, OR, XOR and NOT. So, these are the bitwise logic operations. Then we have got clear, so we can clear a bit. So, we can rotate a register, a register can be rotated via carry without carry etcetera. Then we have got swapping, swapping of nibbles. So, this logic instructions they do not affect the flags in the PSW register. So, this is one of the very important thing to notice that these logic instructions will not affect the PSW flags. So, we have to be careful while doing this check.

(Refer Slide Time: 04:58)



So, ANL is AND logical. So, this will be doing bitwise anding. Then we have got ORL for OR logical, XRL for xoring operation, and CPL for compliment. So, these are the examples by which this logical operations are done.

(Refer Slide Time: 05:17)



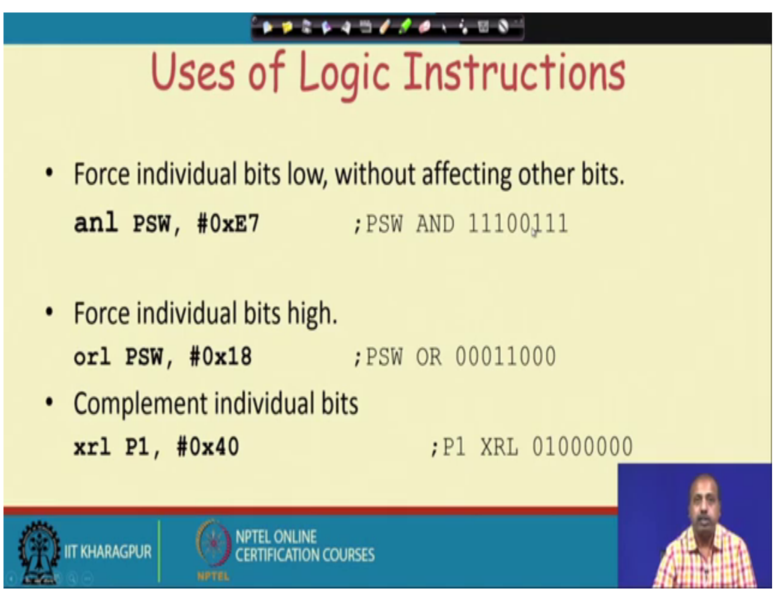
Then address mode. So, we can have this ANL is and instruction. So, you can specify a comma byte where the so one of the operand has to be the a register, the accumulator and with the byte can be direct. So, you can have a direct 8-bit number, you can have registered indirect via say R 0, R 1 or you can have. So, you can have an immediate number sorry you can have an immediate number, so hash like you can say like say AND logical, you can say ANL a comma hash say 52 hex So, a will be anded with 52 hex, so that is the immediate mode that is this immediate mode.

So, you can have this direct mode like you can say like AND logical a comma 52 hex. So, this means a will be anded with memory location 52. So, if this will this means the memory location ram location 52 hex. So, with that the anding will be done, so that is the direct mode. Then we have got registered indirect here. So, we can write in terms of at the rate R 0 at the rate R 1 like that.

So, you can write like ANL a comma at the rate R 0, so R 0 and R 1, so you can say those two registers then we have that is registered indirect or you can have direct register directly. So, you can say ANL a comma R 0. So, R 0 will be anded with AL with a register. So, this way we can have different and logical instruction. Or we can have byte

is other way. So, this is the destination. So, direct byte and a; can have byte constant that also can be specified and the byte hash constant. Then we can have this compliment instruction like CPL a. So, the compliment it will be complemented, the a register will be complemented and we will get the value there.

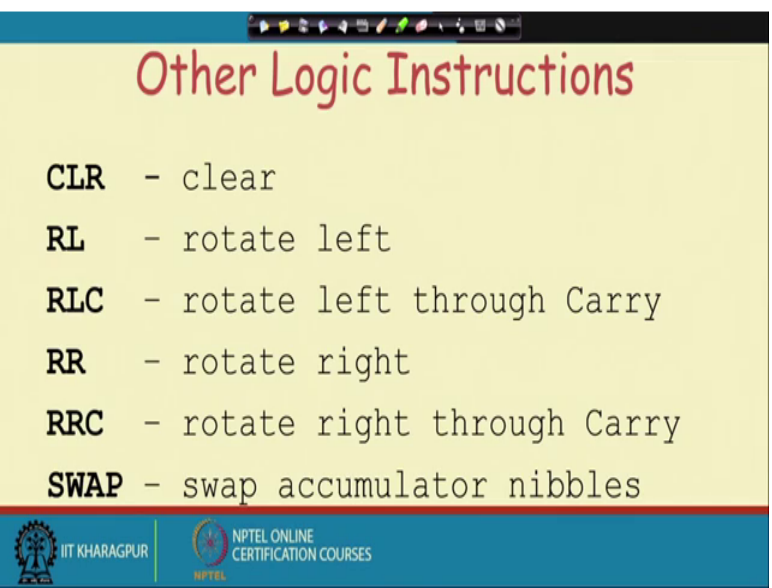
(Refer Slide Time: 07:18)



So, we can force individual bits to low without affecting other bits like you can say like a. So, this is a single bit sort of thing. So, p a anl logical PSW hash a 0xE7. So, the e seven this is the bit pattern 11100111. So, when you do anding with PSW, so PSW these two bits will be set to 0, rest of the bits PSW registered will continue to have its previous value or you can force individual bits high like you can say or, so you can have you can say orl PSW 0x18. So, these two bits will be set to one whatever be the previous values of the bits in PSW register. So, these are useful when you are changing the register banks that then you have four register bank register bank select bits are there.

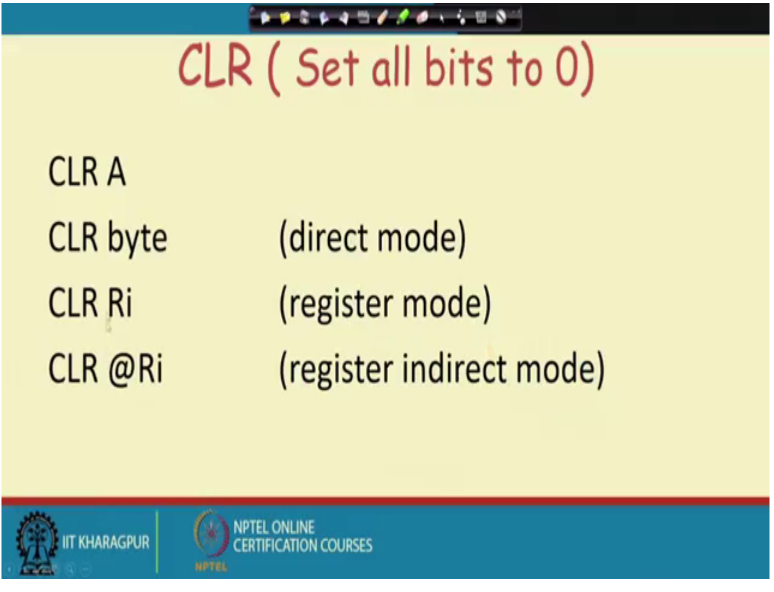
So, without affecting other PSW bits, so if you want to change this bank fillet, then you can use this OR instruction or say AND instruction. So, depending upon the bit pattern you want to set similarly you can have compliment bits by the xrl instruction. So, this will complement only this bit number six of the PSW of the p1 of the port p 1.

(Refer Slide Time: 08:32)



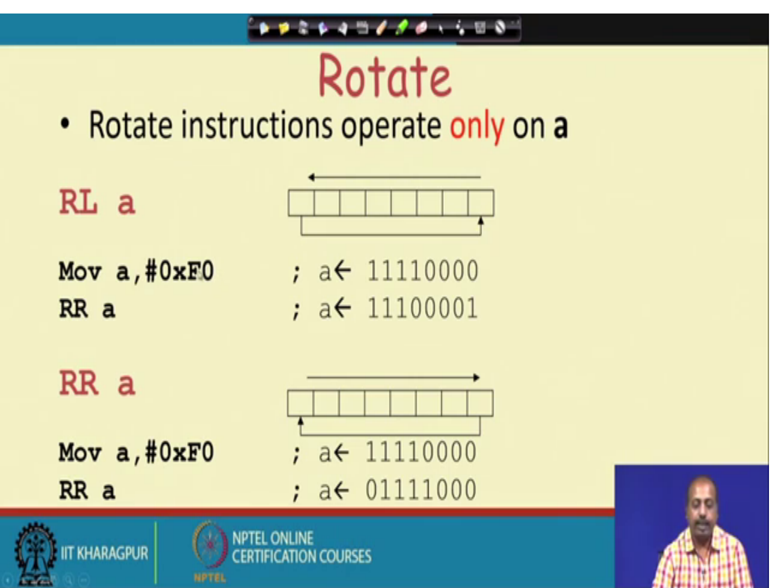
Other logical instruction we have got clear, we have got rotate left, and then there is a rotate left through carry. So, this carry will also get affected the RL instruction will not affect the carry, but RLC instruction will affect the carry. RR will rotate right; RRC will rotate right through carry and swap it will be accumulated nibbles will be swapped. So, these accepting swap other instructions we have seen in 8085 also those rotations are same as we have got in 8085, but swap it will swap the two nibbles of the accumulator.

(Refer Slide Time: 09:10)



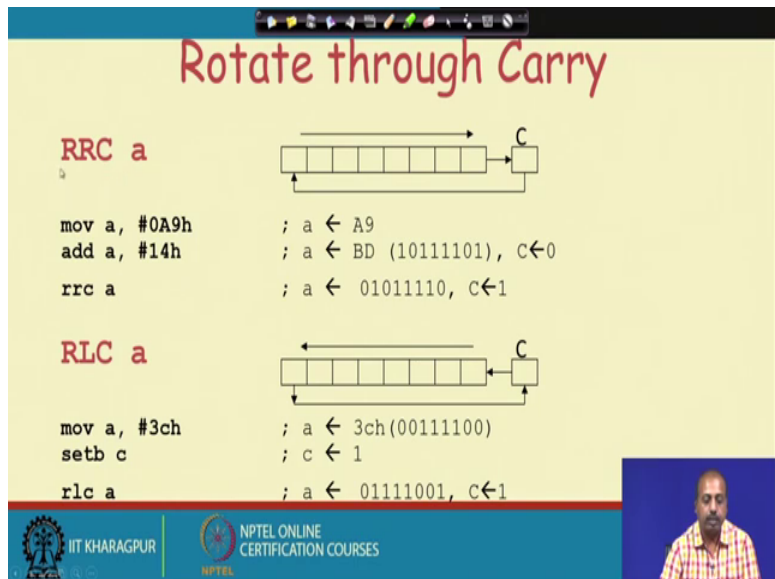
Then clear. So, clear A will set all bits to 0, it will reset all bits or accumulator to 0. So, you can also say clear byte, where byte is specified in a direct mode. So, you can specify the byte. Then clear Ri, so this i can be 0 or one. So, you can have it in the registered modes sorry it can be 1 to 7, all those registers R 0 to R 7 all those registers can come. So, we can have this all these register specified. Then clear Ri, so this is in the registered indirect mode. So, you can clear some memory location. So, this way you can clear the bits of some memory locations.

(Refer Slide Time: 09:47)



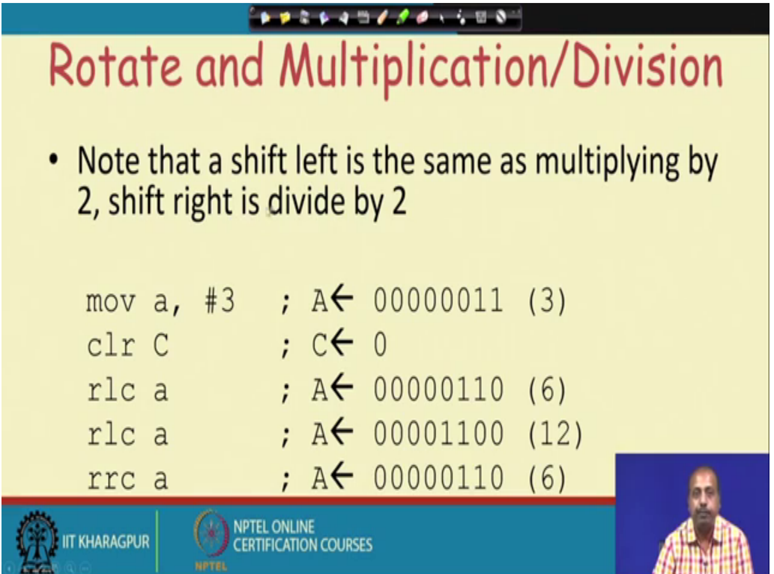
This is the RL a instruction. So, as I was telling that it will rotate left. So, this MSB that we have will come to the least significant position, and these bits will get shifted by one position then we have got rotate right. So, rotate right will this should this is wrong this should be RL sorry this is not RR this is RL. So, then RR, so this is the rotate right so, here this is rotated right. So, this bits will be going here and this bit will come to this LSB will come to the MSB position so that is rotate right.

(Refer Slide Time: 10:22)



Then RRC, so it is rotate right through carry. So, this carry comes to the MSB position and this LSB position goes to carry and rest of the bits are shifted. So, and then similarly we can RLC where this carry will come to the LSB, MSB will go to carry and rest of the bits will get shifted. So, these are examples by which we can do this thing. So, here also you get this set b instruction that is for setting one particular bit. So, set b c will set the carry bit to 1.

(Refer Slide Time: 10:57)

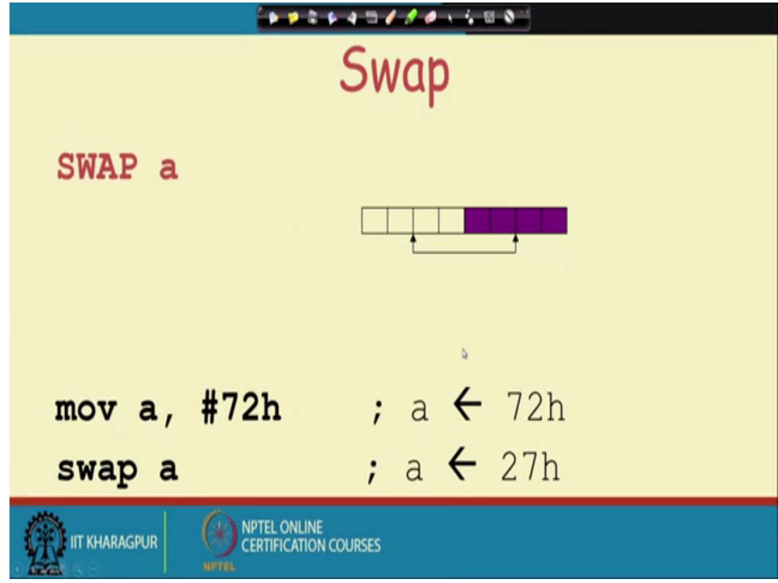


Shift and left is same as multiplying by 2 and shit right is divide by 2. So, any number if you multiply it, it by 2, so it gets left shifted by one position and if you are multiplying a dividing by 2, so that is right shifting by one position if. So, this is an example first in the first instruction move a comma has 3. so we get the immediate number 3 in the accumulator, then we clear the carry. And then we once we execute one RLC instruction rotate left through carry. So, what happens is that this carry comes to the least significant position and this one get shifted to the next position.

As a result, the number that we get is 6. You see it is a multiplication by 2. If you do the step once more then what will happen, so another 0 gets shifted. So, this 1 1 get shifted by one more position. So, you get the number to 12. So, it is 6 multiplied by 2 - 12. on the other hand. So, these are actually multiplication by 2. On the other hand, if you do RRC right rotate right through carry then the bits gets shifted towards the right side and you get the number 12 divided by 2, so get 6.

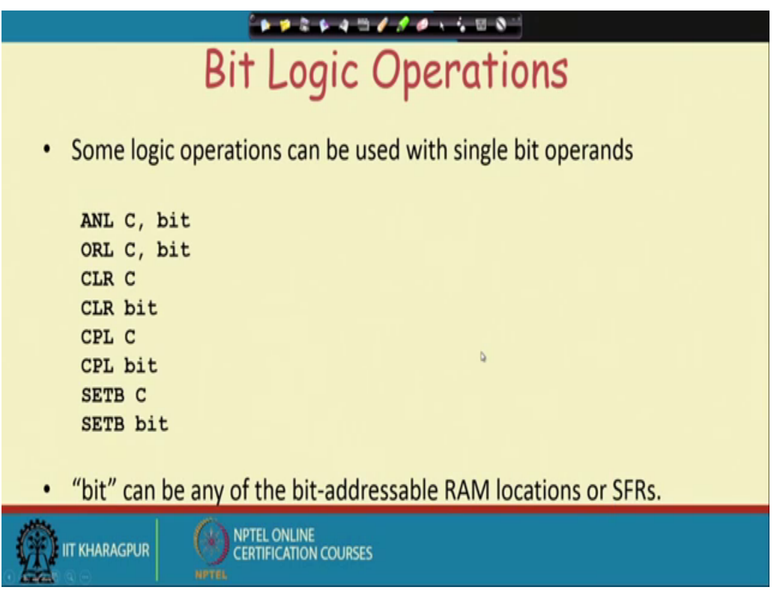
So, this is useful. So, in many of the applications, so the multiplication and division, they are by powers of 2. And as a result, instead of going for costly multiplication division operation, so we can go for this rotate type of instruction with much less number of clock cycles needed. This is particularly true for signal processing applications where we have got many such multiplication and division by 2 as the basic of operand.

(Refer Slide Time: 12:43)



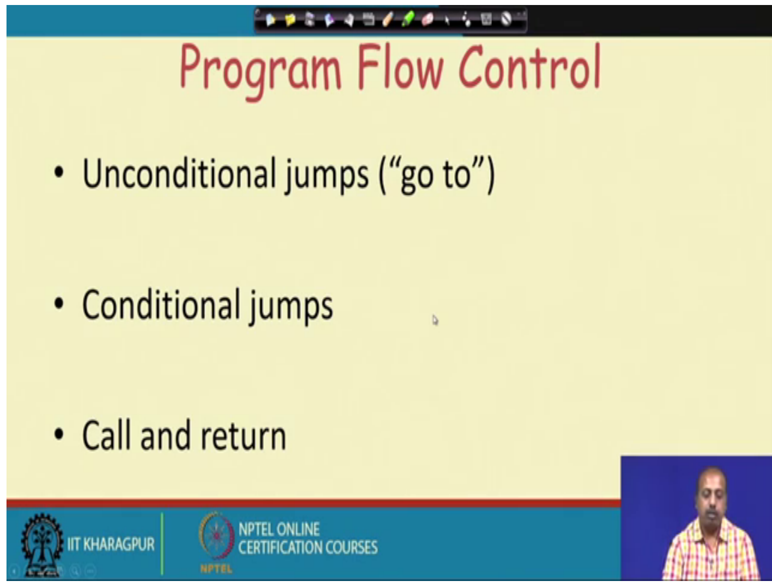
So, this is the swap instruction. So, swap will swap the nibbles, this is higher ordered nibble and the lower ordered nibble. Like if you move a comma hash 72 hex. So, this four bits contain seven these four bits contain two. So, if you execute a swap instruction. So, they will get exchanged. So, the content of a will become 27. So, they are shifted they are swapped like this.

(Refer Slide Time: 13:09)



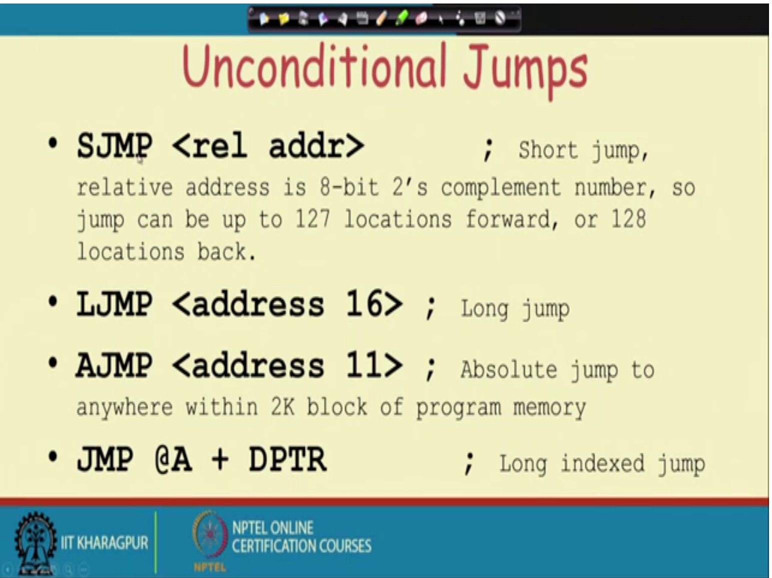
There are some bit logic operations bit level logic operations also like you can specify individual bits instead of byte. So, you can specify individual bit also like ANL C, bit. So, it is and logical carry with some bit. Similarly, all logical carry with some bit, clear carry, clear bit, compliment carry, compliment bit, so these are bit level operations for logic bit level logic operations you can say. So, bit can be any of the bit addressable ram locations or of a some location of the special function register or SFR, so that way we can have this bit level operations done. And same as that byte level operation, here we can do bit level things.

(Refer Slide Time: 13:54)



Next, we look into conditional jump instruction, unconditional jump instruction, then the conditional jump instruction, and call and return instructions.

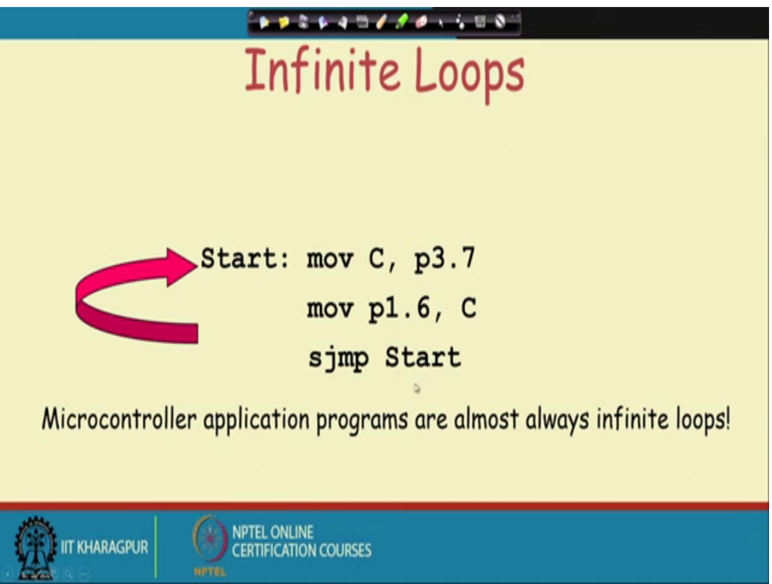
(Refer Slide Time: 14:06)



So, unconditional jump, so we have seen that there is a SJMP instruction for short jump it is relative addressing with 8-bit 2's compliment number it can go 2 minus 127 plus 1 minus 128 to plus 127 location. Then we have got LJMP address the 16-bit address that is the long jump; and there is AJMP, which is a 11 bit address, so offset is 11 bit. So, this range is 2 k, in 2 k block of program memory you can jump. The jump at the rate a plus

DPTR, so this is long indexed jump. So, DPTR with that A value will be added and that value the processor will jump to that particular address.

(Refer Slide Time: 14:51)

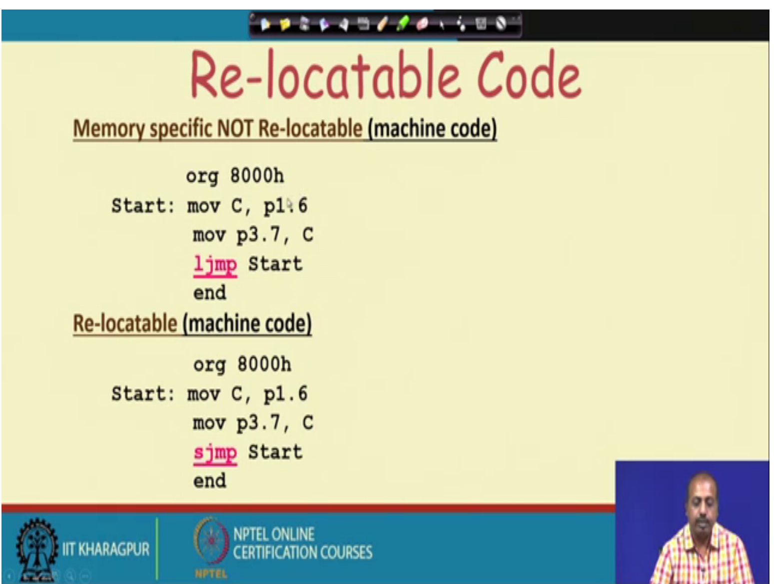


So, you can create some small infinite loops in this fashion. So, actually what happens is that many of the microcontroller applications. So, they are embedded system application. So, embedded systems like say a temperature monitoring system, so it controls the temperature of a room, so that program it will continually sense the temperature values and accordingly control the heater or the cooler.

So, that way that program never ends, so that continually looks into the temperature values and does the control, so that program does not end. So, you can if you look into that this type of programs then you will see that at the end the program branches back to some infinite loop and that way it continues.

So, it is the typical example can be like this. So, move c comma p 3.7. So, this port 3.7 bit comes to the carry register as a the carry flag, and then the carry flag is moved to p 1.6. So, as a result, this port threes bit number 7 is moved to port ones bit number 6 and then SJMP starts. So, it actually continually copies the content of this bit onto this p 1.6. So, this way this we can have infinite loop. So, SJMP type of instruction. So, they can be helpful if this program is small enough then we can have this SJMP type of instruction.

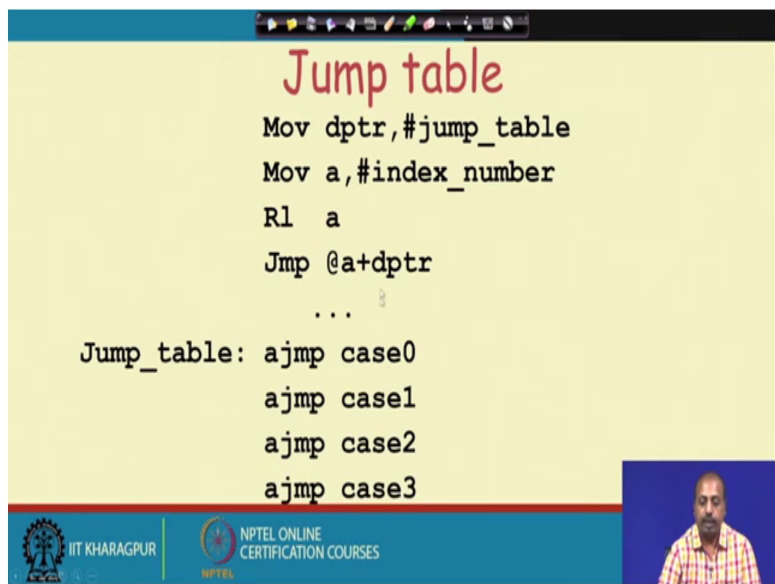
(Refer Slide Time: 16:20)



Then this memory re-locatable code so, this is the memory specific not re-locatable. So, the some ports are like here. So, this code is memory specific like this is ljmp start. So, it is 8000. So, next time the program is loaded from some other address, so this ljmp start will not work correctly as we have discussed previously.

But see this when it is sjmp instruction on the other hand, so this is re-locatable. In the sense that we have seen that while coding this instruction, so we note down the distance from this point to the point where you want to jump, so that value is kept as the part of the instruction. So, that value will be added to the program counter, and the system will jump to that address. So, this ljmp instruction, so they produce non re-locatable code; whereas, this sjmp instruction they produce re-locatable code.

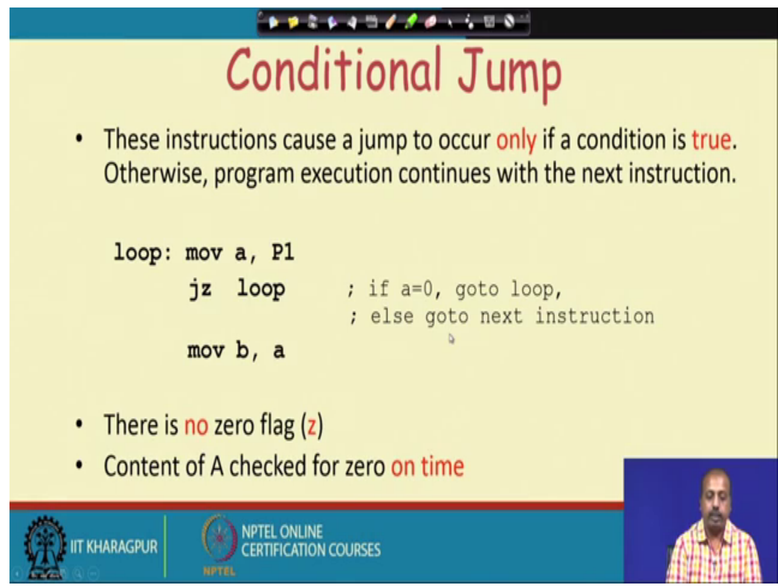
(Refer Slide Time: 17:19)



So, we can have also jump table. So, this dptr jump tables so say in the a registered we have got this index number then we rotate left. So, rotate left a, so that will be multiplying the index number by 2. And then we in the jump table we put all these cases case 0, case one, case two, case three, so there are put here. Now, each of these instructions they will take 2 bytes. So, if this jump table address is 9000, so this at the rate a, so a value for the first one this index number is say 5.

So, 5 means that will be blotted left a sorry multiplied by 2, so it is 10. So, as a result, it will jump to dptr plus 10. So, dptr plus 10 will correspond to actually the case 5, because each of these ajmp instruction will take 2 bytes. So, that case 2 will be at byte number 10 starting from the and offset jump table. So, we can have this type of jump tables implemented you by means of this jump instruction. This jump at the rate dptr instruction is useful when you are implementing this type of a code lookup tables.

(Refer Slide Time: 18:32)



Then we have got conditional jumps. So, these instructions will cause a jump to occur only if a condition is true; otherwise the program will continue like say here move a comma p 1 and jz loop. So, this will continually be jump on 0. So, if a 0 flag is set, then it will be coming to this loop instruction this point back and then otherwise it will move the a register content to b registered content.

So, there is no 0 flag, actually and if you look into this PSW register, so we do not have any explicit 0 flag. So, how does it check? So, it checks the content of a register whether it is 0 or not at the time of executing this instruction. So, that way it is in some sense it is good because it does not depend on the some operation.

So, whenever the accumulator content is 0, so if you execute a z instructions, so it will find that the accumulator is 0. So, in this case it is documented also that way that if a equal to 0 go to loop. So, we have never said that if the 0 flag is set go to 0 or go to the loop go to the loop point, but otherwise so other flags carry etcetera so we have got some specific bit in the PSW register, but for 0 flag we do not have any such PSW bit.

(Refer Slide Time: 19:58)

Conditional jumps

Mnemonic	Description
JZ <rel addr>	Jump if a = 0
JNZ <rel addr>	Jump if a != 0
JC <rel addr>	Jump if C = 1
JNC <rel addr>	Jump if C != 1
JB <bit>, <rel addr>	Jump if bit = 1
JNB <bit>,<rel addr>	Jump if bit != 1
JBC <bir>, <rel addr>	Jump if bit =1, &clear bit
CJNE A, direct, <rel addr>	Compare A and memory, jump if not equal

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

So, these are the various instructions of jump. So, jz relative address so jump, so they are all relative. So, one interesting thing that you note with the conditional jumps is that the all these condition jumps they are relative jump. So, they are offset cannot be more than 256 minus 128 plus 127 it always has to be in that range. So, this JZ, JNZ, JC, JNC so they are all different conditional jump instructions; and they are all relative jump instructions. So, what about if you need to jump beyond say 256 bytes. So, in that case you should have a small conditional jump followed by one unconditional jump.

So, what I mean is that you can put a JZ instruction like say JZ L 1 and that L 1 you put another I am sorry in this L 1, you put say another jump instruction. So, if you have to make really a long jump then you put one ljmp say L 2 where L 2 is far away in the program. So, it will be somewhere here. So, what we do so by a small conditional jump we come to this point L 1; and from that L 1, we put a long jump and come to L 2. So, this may be one way out since we do not have this unconditional jumps because this conditional jumps to be long jumps they are all relative jump and very they are short jumps.

(Refer Slide Time: 21:50)

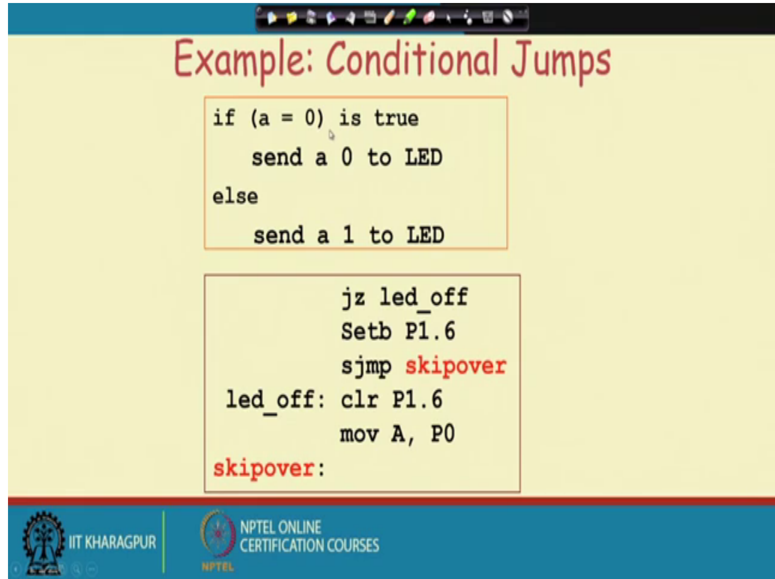
Conditional jumps

Mnemonic	Description
JZ <rel addr>	Jump if a = 0
JNZ <rel addr>	Jump if a != 0
JC <rel addr>	Jump if C = 1
JNC <rel addr>	Jump if C != 1
JB <bit>, <rel addr>	Jump if bit = 1
JNB <bit>,<rel addr>	Jump if bit != 1
JBC <bir>, <rel addr>	Jump if bit =1, &clear bit
CJNE A, direct, <rel addr>	Compare A and memory, jump if not equal

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Then there is a CJNE instruction this is a very complex instruction in the sense that it compares the content of A register with the memory location. So, you can say like CJNE. You can say like CJNE A comma say 52 hex comma l one something like this where this content of a registered will be compared with the memory location 52 and if they are not equal so a and 52 location if they are not equal then the system will jump to the relative address, otherwise it will not. So, this is a very complex instruction that way, but that is many times that is useful in a single instruction we can compare and also jump it is basically jump on 0 and this condition checking. So, both of them are taken together into a single instruction.

(Refer Slide Time: 22:40)



Then we have this is the conditional and say this is one condition jump example like in the high level suppose we are acting like this if a equal to 0 is true then send a 0 to LED, that is LED will be turned off. And if we want to say otherwise we want to send them 1 to LED, so LED will be turned on.

So, what we do if so that a equal to 0, so they are here I mean the accumulator register. So, JZ led off. So, it jump on 0 to led off, led off will turn off the led. So, it will clear this port p 1.6 assuming that the led is connected to port ones bit number six and then it will be sjmp to skip over. So, it otherwise it will come to this it will come to this point, it will clear this it, it will set b sorry jump on 0 to led off. So, if it is not 0, then it will turn on the led, so that is that second part of als part is coded here. So, set b p 1.6, so the led is turned on then it is sjmp skip over. So, it will come to this point.

Otherwise, it will be coming to this point it will be clearing this p one point six and then this instruction is redundant maybe it is just reading the port 0 from a. It is reading port 0 into the a register something like that, but that has this particular statement does not have any correspondence with this piece of code it is just an example.

(Refer Slide Time: 24:10)

More Conditional Jumps

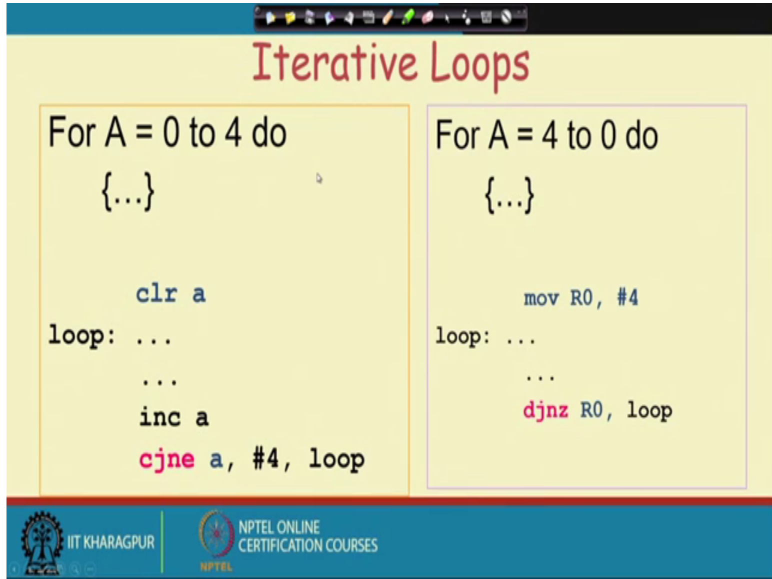
Mnemonic	Description
CJNE A, #data <rel addr>	Compare A and data, jump if not equal
CJNE Rn, #data <rel addr>	Compare Rn and data, jump if not equal
CJNE @Rn, #data <rel addr>	Compare Rn and memory, jump if not equal
DJNZ Rn, <rel addr>	Decrement Rn and then jump if not zero
DJNZ direct, <rel addr>	Decrement memory and then jump if not zero

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

More conditional jump instructions we can have CJNE a data relative address, so that we have seen. Then CJNE Rn, so we can compare this register with some data and then jump to relative address then we have got CJNE memory address like R 0 and R 1 they can be specified at the rate Rn data relative address. Then DJNZ decrement Rn and jump on not 0. So, this is a again another complex instruction DJNZ decrement and jump on not 0. So, normally when you are executive some loop instructions, so where you have seen that we try we decrement the loop counter and if the loop counter is not 0, then we jump back to the start of the loop again.

So, normally it may we have to use two instruction one for decrementation and another for compare and then so jump so, in there instead of that. So, this DJNZ is a single instruction by which you can decrement the register value and then if the register value does not become 0, so you can jump back to the relative address. And then we have got this DJNZ direct address and then this relative address, so the direct and this relative address. So, here also the same thing, it will decrement the memory location content and then if the content is nonzero, then it will be jumping to the relative address.

(Refer Slide Time: 25:36)

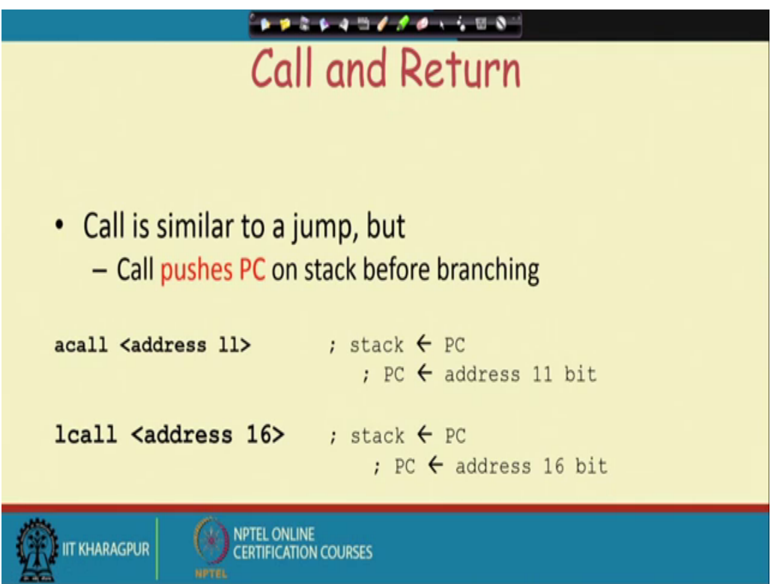


So, this is an iterative look like say for a equal to 0 to 4, so we are executing something here. So, this loop when it is converted into a program in machine like 8051 program, so it will look something like this. First, this A has to be set to 0. So, this clear a, then this loop body. So, loop body is executed. Then I value is implemented by 1, and we compare whether this a has become equal to 4 or not. So, if a is not equal to 4, then we go back to this loop. So, this way it mimics the behavior of this loop here.

Of course, you can say that this particular check CJNE etcetera, etcetera. So, this should be done earlier because otherwise normally this after this group is initialized to 0, it get condition check, but in this case it is not required because assuming that a is not modified immediately. So, here a value is 0.

So, there is no point doing this check at the beginning. So, this is fine. So, this will work fine for this particular port. On the other hand, if you have got a loop which goes in the descending way. So, for a equal to four down to 0, descending loop, so here all you have to do it like this and say we take the register R 0 initialize it to 4, and then we say djnz R 0 loop. So, decrement jump on not 0 R 0 to loop. So, you can do it like this. So, with we have got iterative loops like this.

(Refer Slide Time: 27:06)



Then we can have call and return instruction. So, call is similar to jump, but it will also push the pc value onto the stack before branching. So, we have got two versions like just like this jump we had got ajmp and ljmp. So, we have got a call and l call. So, a call the offset is address is 11 bit wide, so this pc address is 11 bit. And l call address is 16 bit wide, so the pc address is 16 bit.

So, in the stack the in both the cases execution is same. So, in case of a call this stack will be loading will be loaded with the next value of from the program counter for returning, and then the program counter will get that at 11 bit address. And in case of l call, this again the same thing this program counter will be saved next program counter value for return will be saved into the stack and program computer will be loaded with the 16 bit address.